Notes on Contemporary Machine Learning for Physicists

Jared Kaplan

Department of Physics and Astronomy, Johns Hopkins University

Abstract

These are lecture notes on Neural-Network based Machine Learning, focusing almost entirely on very recent developments that began around 2012. There are a ton of materials on this subject, but most are targeted at an engineering audience, whereas these notes are intended for those focused on 'theory', but from an extremely pragmatic perspective.

Contents

1 A Brief Conceptual Tour of Contemporary ML

At the beginning of a course I always like to provide a philosophical/conceptual tour of the subject matter. Usually I apologize and say "this won't make much sense, but hopefully it'll give you some organizing principles". In this case, the field may be simple enough that this will make some sense right away!

1.1 Curve Fitting with at Least a Million Parameters

If at any point Machine Learning seems confusing, complicated, jargon-filled, etc, then just remember... it's really just curve fitting, or 'regression', with a very, very large number of parameters.

In ML, we typically represent data as a vector x in a high dimensional (often $\gg 100$) vector space. Canonical examples include a vector of all the pixel intensities in an image, or a vector that can represent all of the words in a language-model's vocabulary. That means that we need a way to parameterize a complex, non-linear function that acts on this space. Neural networks are the most obvious modular construction that can represent a large class of such functions.

We build a NN by taking our data x^i, performing an affine transformation, and applying a simple non-linear function to the resulting components. The most frequent, and now-canonical example is

$$f^i(x) = \max\left(0, W^i_j x^j + b^i\right) \tag{1.1.1}$$

The $\max(0, y)$ function is referred to as a 'Rectified Linear Unit' or ReLU in the literature. It's perhaps the simplest possible non-linear function, as its piecewise linear with two pieces, one of which is zero. The matrix W and vector b are typically referred to as 'weights' and 'biases'. Together they are the 'neural network parameters'. When a NN trains or 'learns', it is simply improving its choice of values for W and b in order to better accomplish a specified task.

Our example f above represents a single layer of a NN. A general NN is simply a composition of such functions, explicitly

$$\max\left(0, W^i_{n,j}\max\left(0, W^j_{n-1,k}\max\left(0, \cdots\right) + b^j_{n-1}\right) + b^i_n\right) \tag{1.1.2}$$

In most cases each layer of the NN depends on separate parameters W_n, b_n for the nth layer. The object we just wrote down is referred to as a fully connected NN. For historical reasons it's often also called a 'multilayer perceptron' model.

We can form much more complicated NN *architectures* by splitting up our vector space x into sub-components, and having different (smaller) matrices act on the subcomponents in a variety of complicated ways. We can also apply more general operators or linear-algebra constructions to the results. Perhaps the most famous examples of these ideas are the Convolutional Neural Network (CNN) and Recurrent NN (RNN), of which the LSTM or 'Long Short-Term Memory' network is a more involved example. We'll discuss them in detail later on.

If our initial vector space has high dimension (eg $\sim 10^3$), as is typical, then W will necessarily contain *a lot* of parameters, at least of order 10^6. It seems that the field of NN-based ML has only developed recently because of the computational requirements to deal with so many parameters, and such large data samples. In particular, excitement in the field was kicked off in 2012 by the 'AlexNet' paper of Krizhevsky, Sutskever, and Hinton.

Why are we here? It makes sense for physicists to think about this subject because NNs are much more like condensed matter systems than algorithms. They have enough parameters that they're not so far from a kind of 'thermodynamic limit'.

1.2 What's the Goal of Learning?

In the last section we gave a very brief explanation of what a NN is – a very general non-linear function on a high-dimensional vector space. But what should we do with it?

All we really want is to do 'curve fitting' or 'regression', ie we want to learn values for the NN parameters $\theta = \{W_k, b_k\}$ so that the complete function F represented by the NN does something interesting or useful. That means that we need to (1) specify a goal and (2) figure out how to learn values for θ so that our NN does a good job of achieving that goal.

In the ML context, these goals are almost always represented by a **loss function**. This is a function that depends on both θ and the data, which is minimized for values of θ where the NN is performing well. The canonical example is if we try to draw a straight line through a scatter plot of data, then our loss function might be the least squares loss

$$L = \sum_i |y_i - F_\theta(x_i)|^2 \tag{1.2.1}$$

and we learn the θ – in this case just the slope and intercept – so that the line passes through the data. Most loss functions in machine learning can be understood and justified in terms of probability and information theory, most often using the **maximum likelihood** principle.

In general in ML, we would like to find functions that classify images, write paragraphs, play games like Atari or Go, or generate a host of never-before-seen images. Soon we will discuss how to specify loss functions that lead to functions that accomplish these goals. Generally speaking they are classified as

- Supervised Learning: For each data point we have a specific goal, as in the regression example above. This also means that we can explicitly take derivatives of the loss function.
- Reinforcement Learning: How to learn a policy that determines what action an agent should take in a given environment. In RL we can evaluate whether the policy is good or bad on average, over some time interval, but we do not know if each moment-to-moment choice the AI makes is optimal. The canonical example is a game-playing AI.
- Unsupervised Learning: We want to somehow extract patterns from the data, naively without a clear specification of what a 'pattern' is. In practice we usually make progress here by turning UL into something closer to SL. Note that if we do UL very, very well, we might also implicitly solve a lot of SL and RL problems.

The difference is largely about how close the connection is between the goal and the data itself; when the connection's more immediate, learning is easier, quicker, and more efficient. We'll discuss all of this in detail soon.

Figuring out how to specify, or learn to specify, better and more abstract goals is an important area of research. It's especially important when considering **safety**, as simple goals often have unintended consequences.

1.3 Optimization – How to Learn

The third absolutely crucial ingredient in contemporary ML is optimization – given a goal and a set of NN parameters, how do we determine values of the parameters so that the NN performs well? This is called optimization because we are 'optimizing' the parameters to minimize the loss, or maximize performance.

In practice, essentially all forms of optimization used in ML are versions of stochastic gradient descent, where

$$\theta_{n+1} = \theta_n - \epsilon \nabla_\theta L(\theta; X) \tag{1.3.1}$$

where ϵ is typically called the learning rate. This is a discrete process of updating the parameters to try to minimize the loss. The word 'stochastic' is used here because we do not compute the loss over the full dataset, but only over a smaller, randomly selected **batch** of data from the full distribution.

There are many, many fancier versions of gradient descent in use, with largely uninformative names like Momentum, RMSProp, Adafactor, Adam, etc. There are also algorithms that attempt

to use an adaptive learning rate. That said, in practice the field has converged to primarily use Momentum and Adam. We'll explain what these are and why they might be a good idea later on. But the 'why' of these algorithms isn't all that well understood. Here are some quick comments:

- The main issue we seem to confront is ill-conditioning – the fact that the very, very high dimensional 'loss landscape' has some very steep and some very shallow directions. This can be formalized by comparing the largest and smallest eigenvalues of the Hessian of the loss.
- Another obvious issue we can confront is that the loss function will not be convex, and so it may have local minima and saddle points. It's actually not clear (to me) to what extent this is a significant problem in ML, but it's widely discussed as a potential problem. Note that a random matrix will tend to have as many positive as negative eigenvalues, so naively we should expect the local loss landscape to have many saddle points.
- As physicists it's tempting to think about the gradient descent update in the continuum limit, where $\epsilon \to 0$ and we're smoothly 'rolling' down the loss potential. But *the continuum limit is a terrible approximation to efficient optimization.* In practice, we want our discrete steps to be as large as possible, so that gradients at successive steps are uncorrelated – otherwise, we're 'wasting' information.
- There's a ton of theory on optimization, but it's unclear how well it applies, as most treatments focus on the convex case. Furthermore, formal theory papers often recommend schedules for the learning rate that lead to prove-able convergence, but that aren't very effective in practice.
- It's worth noting that NN parameters have a lot of degeneracies – for example, permutations of the rows of matrices in each layer. So even global optima will be very non-unique.

Optimization is all about getting a good result while using as little computation and time as possible. Note that because NNs and datasets are extremely large (and constantly pushing the frontier), memory use is also a major constraint on optimization.

There are also interesting 2nd order optimization methods, which use information about the Hessian (or a potentially less familiar object, the Fisher information matrix) to optimize faster. These methods aren't very widely used, as the cost of computing the Hessian hasn't been convincingly demonstrated to be worth the gain.

Improving and better understanding optimization is an important area of research, though as with many areas of ML, it's challenging to demonstrate convincingly broad improvements.

1.4 Algorithmic Differentiation and Backprop

To optimize our NN efficiently we need to be able to take derivatives of very general classes of functions. This seems challenging, because these functions may be represented by complicated computer programs involving nested loops, matrix multiplication, and all sorts of non-linear functions. How should we go about it?

Here are two naive approaches that we won't want to use. The first is symbolic differentiation – if we have a symbolic representation of our function, we can compute the derivative analytically,

and then write a program based on the resulting formula. This won't work because it's inefficient and insufficiently general. To see why it's inefficient, consider $f(x_i) = \prod_i x_i$. The gradient of this function is very simple conceptually, but if you write it out symbolically (without some insight), it actually involves a ton of symbols. For an even worse example, think about what you'd do with the determinant of a matrix!

The second naive approach is finite differences like

$$f'(x) = \frac{f(x+\epsilon) - f(x)}{\epsilon} \tag{1.4.1}$$

In general, this is a very problematic due to finite precision and the presence of order ϵ^2 terms. For example, consider the function $f(x) = 10^{10} + x^2$, and say we want to compute its derivative at $x = 10/9 = 1.111\cdots$. Whatever we choose for ϵ, we will end up with horrendous rounding or truncation errors. For instance if $\epsilon = 10^{-3}$ and we have 16 digits of precision, we'll compute

$$\frac{(10^{10} + \frac{10^2}{9^2} + 2 \times \frac{10}{9} 10^{-3} + \frac{10^2}{9^2} 10^{-6}) - 10^{10} - \frac{10^2}{9^2}}{10^{-3}} \to 2.2234 \tag{1.4.2}$$

So we only end up with 5 digits of precision, and even at that level we get the wrong answer!

Algorithmic Differentiation

Fortunately, there's a third way, which is referred to as *Algorithmic Differentiation*[1] or sometimes *Automatic Differentiation*... either way it's AD. It's what's used by NN libraries like Pytorch and Tensorflow, though it's also used in a wide variety of other contexts.

The idea of algorithmic dfifferentiation is quite simple. Given that a computer is literally computing some function $f : X \to Y$, it must be possible to decompose the function f into a number of very elementary steps. Each of these building blocks can be differentiated explicitly (ie analytically), with the result evaluated numerically. This fact plus the chain rule means that given any program for evaluating f, we can break it down into constituent pieces and use the chain rule to obtain the full derivative, without any approximations that will blow up the numerical error.

Typically in general and also in codes like tensorflow, the decomposition of a function into more or less elementary pieces is referred to as the computational graph.

Backprop

By the way, you've probably heard of the **backpropagation** algorithm. It's really just a re-statement of the chain rule. That is if our function is

$$f(g_i(\theta_j)) \tag{1.4.3}$$

then we can compute the derivative as

$$\nabla_{\theta_j} f = \sum_i \frac{\partial f}{\partial g_i} \times \frac{\partial g_i}{\partial \theta_j} \tag{1.4.4}$$

[1]For an extensive discussion see the book by Griewank and Walter on 'Evaluating Derivatives'.

This can be computed by first evaluating $\partial_j g_i$ or first evaluating $\partial_i f$. If we compute $\partial_j g_i$ first we are propagating forward. But we are 'propagating backwards' if we first compute the derivative $\frac{\partial f}{\partial g_i}$, which is at the 'end' of the computation, and then we propagate 'backward' towards the beginning of the computation, by computing $\frac{\partial g_i}{\partial \theta_j}$ and multiplying. With more nested functions we'd keep 'propagating further backward'. That's it.

The reason to perform backprop is that when we have a small number of outputs and a high-dimensional input, then it's more efficient than computing in the forward direction. Let's look at the computation in a bit more detail, and see exactly how much less efficient it is to compute the derivatives vs the function itself.

If our NN has non-linearities $\phi(x)$, then a two-layer NN function will take the form

$$y^i = \phi(W_2^{ij}\phi(W_1^{jk}x_k + b_1^j) + b_2^i) \tag{1.4.5}$$

where repeated indices are summed. The layers have dimensions D_2, D_1, D_0 for the outputs, hidden layer, and the data itself, respectively. Assuming ϕ isn't especially complicated, the largest source of computation when we simply evaluate y_i will be the matrix multiplications by W_2 and by W_1. These will require D_2D_1 and D_1D_0 component multiplications, respectively. The biases just requires D_2 and D_1 additions, which are negligible in comparison.

How about the evaluation of the derivatives with respect to the parameters via backprop? This will also be dominated by the matrices. We can compute (here the i index isn't summed over)

$$\frac{\partial y^i}{\partial W_2^{ij}} = \phi'\left(W_2^{ib}\phi(W_1^{bc}x_c + b_1^b) + b_2^i\right) \times \phi(W_1^{jk}x_k + b_1^j) \tag{1.4.6}$$

This provides the derivatives with respect to W_2. But to continue the process at the next layer, we need to compute the derivative wrt to the hidden-layer activations ϕ_j, which is (i and j aren't summed here)

$$\frac{\partial y^i}{\partial \phi_j} = \phi'\left(W_2^{ik}\phi(W_1^{kc}x_c + b_1^b) + b_2^i\right) \times W_2^{ij} \tag{1.4.7}$$

With this information we can proceed to compute gradients at the next layer. Notice that we have had to perform *two matrix multiplications* per layer to evaluate the derivatives via backprop, whereas we only had one matrix multiplication per layer for the forward evaluation of the NN. This is a useful rule of thumb when estimating the number of computations and speed of evaluation of backprop.

I haven't written out the derivatives with respect to the bias b_2, as it's a negligible fraction of the computation. But it's very important – we most definitely need to learn b as well as W when we train our NN!

1.5 Architectures

Although fully connected NNs are still very common as sub-components, in practice, for most tasks, other 'architectures' are used. At this moment there are really only three basic ideas that have been shown to be powerfully useful:

- CNN: Convolutional Neural Networks are mostly used for images, and are easily motivated by translation invariance. They slide fixed 'filters' over the image to compute convolutions.
- RNN: Recurrent NNs and their stabler cousin, the LSTM, are very widely used for sequence data.
- Attention: A third idea that's become very important in the last couple of years is attention, and a specific pure-attention model called the Transformer. Attention-based models process data by highlighting or up-weighting specific features based on learned correlations.

The first two can be easily motivated by symmetry, while the third's a bit more abstract, but might be motivated dynamically, by considering the kinds of correlations present in real-world data.

Another aspect of NNs that might be placed under the aegis of 'architecture' is whitening – that is, we would like information to 'flow through' a NN in a stable way, without signals blowing up or vanishing. Several of the most highly cited papers from the last five years simply suggest a strategy for whitening, and often that strategy in effect modifies the activation functions or effective architecture in some simple way.

1.6 Overfitting and Generalization

A shocking feature of NNs is that if P is the number of parameters and D is the dataset set size, typically we're in the regime $P \gg D$. Metaphorically, this means that when we train NNs, in effect we're fitting a 10th order polynomial to 3 data points!

The obvious problem with fitting a 10th order polynomial to 3 datapoints is that it will **overfit**, rather than learn any meaningful pattern in the data. Somehow NNs mostly don't overfit too much, and in practice overfitting doesn't necessarily get worse as P increases. The absence of overfitting is closely related to the idea of **generalization** – a ML model generalizes well if it does well not just on the training data, but on other 'test' data that it has not encountered before. A related question is whether ML models learn general, robust features first, or if they learn from the bottom-up, starting with 'details'. My sense is that generalization and overfitting are not very well understood in the field, and elementary findings often seem to surprise seasoned researchers. That said, will give a simple account later in the course which seems to explain the rough features, in particular why $P \gg D$ isn't a catastrophic problem.

To compensate for overfitting, models are often **regularized**, which means that they are constrained or perturbed in some way to decrease overfitting and improve generalization. The most common intuitions are that

- If models are constrained to be simple – which might mean, for example, that their parameters are penalized for exceeding some small range – then one might hope to learn more general patterns. A common strategy is adding a norm of the parameters to the loss.
- If models are robust to perturbations, either in their parameters or the data, then they're less likely to overfit. This idea leads to one of the most popular regularization schemes, called Dropout, where some fixed random portion of NN weights or activations are simply set to zero.

It's worth noting that in some cases, overfitting isn't necessarily a problem, or just reduces to optimization. This is true in the infinite data limit, and RL is often, in some sense, in this limit.

1.7 Datasets

In many respects, *datasets have been more important to machine learning than algorithms or architectures*. Large datasets motivate and enable new accomplishments, and have become the main way that new ideas are benchmarked. This phenomenon has a sociological dark side, as many papers are written in order to compete for the best result on a given dataset, so much of the work in the field involves tuning for a limited and rather artificial prestige.

Here are some extremely common datasets:

- MNIST & SVHN: At present these are essentially 'toy' or 'warmup' datasets. The former contains 5×10^4 images of handwritten digits, and all are $28 \times 28 = 784$ pixels. Street View House Numbers has a smaller and larger version; the large version has about 5×10^5 small color images of individual digits from home addreses.

- CIFAR: These images are small 32×32 color images of a variety of objects, so that this dataset is a bit more challenging than SVHN. There's a 10 and 100 class version. This dataset is still quite small, however, so progress on CIFAR image classification is mostly about avoiding overfitting.

- ImageNet: This is the predominant large image dataset, with about 10^6 color images of total size 256×256, and with 1000 different potential labels. AlexNet's progress on ImageNet kicked off the present wave of NN research.

- Penn Treebank: This is a small language model dataset.

- Billion Word Benchmark: A larger language modeling dataset. It's scrambled at the sentence level, so that it doesn't have any long-term dependencies (eg no self-consistent paragraphs).

- Mujoco: This is a famous proprietary continuous control environment with semi-realistic physics. It's one of the few most common reinforcement learning environments, and has a continuous action space. Even simpler examples are 'classic control' problems.

- Atari: Atari is one of the most common RL environments, including ~ 50 different Atari games. Montezuma's Revenge is famously difficult due to its sparse rewards and the necessity of exploration; most other Atari games have been solved to a super-human level.

Of course there are many other datasets large and small, but these are the most common. MNIST, SVHN, CIFAR, Penn Treebank, and Atari are all small enough that you can train ML models to (in some sense) solve them on a laptop or in Google colab. At the moment, training a vision model on ImageNet will typically cost hundreds of dollars in cloud GPU time... and there are some proprietary datasets much larger than ImageNet. It's increasingly difficult to conduct cutting edge ML research without major computing resources.

1.8 Setting Up an MNIST Classifier

A simple fully connected NN with a few layers is sufficient to do pretty well on the MNIST dataset of hand-drawn numerals. And pretty much any optimizer will do for the learning process (default learning rates, and any $\sim$ 10s or 100s batch size should work). The only ingredients we need are a method of outputting a classification and an explicit specification of the loss.

For classification, we will have our network output probabilities $p_i(x)$ for the image x to be the numeral $i = 0, 1, \cdots, 9$. For this we use the softmax function

$$p_i(x) = \sigma_i(a_j) = \frac{e^{a_i - a_{\max}}}{\sum_j e^{a_j - a_{\max}}} \tag{1.8.1}$$

where the a_j are called 'logits', and are 10 real numbers outputted by the network. So this will be our final layer. When making classification decisions, we just choose the digit with the maximum p_i, which is also simply the digit with maximum a_i.

For the loss, it's standard to use the maximum likelihood, aka cross-entropy, aka KL-divergence between the correct distribution and the NN output distribution

$$L_\theta(p_i, y_j, x) = -\sum_{i=0}^{9} y_i \log p_i(x) = -\log p_{\text{actual}}(x) \tag{1.8.2}$$

where $y_i = (0, \cdots, 0, 1, 0, \cdots, 0)$ is a vector with a 1 in the position of the digit and 0s elsewhere. This is often called a 'one-hot' vector.

1.9 Open Problems

At the **empirical level**, one might argue that there's a single, predominantly important open problem, namely that of **sample efficiency**. Although it varies from case to case, most experts would agree that ML algorithms tend to need far more data than humans and animals to learn a complex task. The most extreme example is perhaps Dota, where OpenAI's Dota agents play the game for the equivalent of hundreds of years in order to achieve human pro level performance. And a more practically relevant example is robotics, where the main problem is that physical robots need too much real-world experience to learn complex manipulation tasks like folding laundry. Most of the more specific open problems below are reflected in poor sample efficiency.

- Fundamentals: Our basic theoretical understanding of all forms of learning could be significantly improved. Even in the case of SL for computer vision, the well-known 'adversarial example' problem suggests we could be missing something big. In the case of RL and UL, our understanding is even more preliminary, and these fields are still viewed as incomplete.

- Generalization: We do not have a good understanding of when and why models generalize, nor do we know how to make models that extrapolate beyond the distribution they were trained on. This will be important for robustness (eg adversarial examples) as well.

- Transfer Learning: When humans learn one task, they can usually transfer that knowledge to a host of related tasks, so that they do not need as much experience to learn. This is closely tied to generalization and sample efficiency.

- Meta-Learning and One-Shot Learning: Can ML models 'learn how to learn'? Relatedly, when they encounter a single new example (for instance an agent encounters a new source of reward), do they immediately learn to seek out similar instance in the future? We are just beginning to build models that can accomplish these feats.

- Model-Based RL: We plan for the future, and imagine future states in order to make decisions now. Most RL agents do not do this, rather they just collect a ton of experience and reinforce successful behaviors while discouraging poorly performing behaviors. But incorporating planning and a 'world model' into RL is a major open problem that many researchers are currently focusing on.

To be clear, the relative importance and meaning of these problems is debated. Most researchers view them as fundamental, but it's not impossible that they could be partially resolved by simply training larger models on larger and more varied data distributions.

Another large and multifacted open problem is **safety**. We can break it down into components:

- How can we teach agents about complex human values and goals, which are very difficult to specify in terms of a simple reward function? How can we make sure they pursue those goals, accounting for specific safety constraints, even when encountering new situations and environments?

- How can we design robust agents, which have a clear 'understanding' of what they know and why, so they can adapt correctly to new information, 'off the training distribution'... or at least behave cautiously? Can we remove (sufficiently simple?) adversarial examples?

- How can we ensure that as AI gets deployed in more and more domains, it's used altruistically, rather than to eg maximize the balance of a bank account or the dominance of a nation state?

1.10 Probability and Random Variables (Mostly Notation)

Before moving on, let's establish some notation. Much of this is pretty standard, but it's easy to work as a physicist without really internalizing it. And it's useful to be explicit about the distinction between a sample of data and abstract random variables representing that sample.

We'll often discuss probability distributions. Sometimes I'll write them as $p(X)$, which suggests the full distribution; other times I'll write $p(x)$, which instead suggests 'the probability of a given $x \in X$'. But I can't promise I'll be wholly consistent. If I have a joint probability over X and Y, I can write $p(X, Y)$. Note then that I can get a distribution over just X by marginalizing, so that

$$p(x) = \sum_y p(x, y) \tag{1.10.1}$$

Conditional probabilities are written $p(x|y)$ = 'the probability of x, given y'. Note that

$$p(x,y) = p(x|y)p(y) = p(y|x)p(x) \tag{1.10.2}$$

and this is/gives Bayes rule (we get the usual Bayes rule by dividing by one of the probability distributions). Marginalization makes sense in words when performed over these decompositions, ie

$$p(x) = \sum_y p(x|y)p(y) = \sum_y p(y|x)p(x) \tag{1.10.3}$$

The first equality means that the probability of x is given by the probability of x, given y, times the probability of each y, once we sum over y. The second says that since $p(y|x)$ is a probability distribution for y by itself, when we sum it over y with fixed x, we get 1.

We will often talk about 'random variables', which are algebraic variables representative of a distribution $p(X)$. Typically a (literal) number or vector is written $x \sim p$ which reads as 'x drawn from the distribution p', whereas the abstract random variable itself is written as X. The latter is something we can do algebra with, without taking an expectation value over the distribution. So we can talk about X^3 or e^X, and compute things like $X^2(1+Y^2)$ for different random variables X and Y. Very often if we talk about X and Y we will assume they are independent, but they may not always be, ie they could be drawn from some $p(X,Y) \neq p(X)p(Y)$. Note though that expectations are always linear, so

$$\langle X+Y\rangle = \int dxdy(x+y)p(x,y) = \langle X\rangle + \langle Y\rangle \tag{1.10.4}$$

even if X and Y are not independent. But of course $\langle XY\rangle \neq \langle X\rangle\langle Y\rangle$ if they are dependent!

Note that it's common to study many random variables X_i where $i = 1, \cdots, N$ drawn from the same distribution $p(X)$. In this way we can use the X_i as abstract representatives of a concrete dataset $\{x_i\}$ composed of literal numbers. Then we can take an expectation over all of the X_i to see how various functions of a literal, finite sized dataset will be expected to behave.

2 Information Theory Background

There are actually two somewhat different types of information theory – that which deals with communication (as developed by Shannon), and that which deals with the intrinsic information in an object or sequence (as discussed by Kolmogorov). Both are very interesting to consider in the context of ML, but in this section we will focus on the first type, which has many more practical applications. It is also closely connected with probability theory and statistics.

2.1 Entropy in Information Theory

In statistical physics we think of entropy as simply

$$S = \log \Omega \tag{2.1.1}$$

where Ω is simply the multiplicity of possible states, presumed to be otherwise equally likely. But for a variety of reasons, it's more useful to think of entropy as a **function of the probability distribution** rather than as depending on a number of states or configurations.

If we have Ω states available, and we believe the probability of being in any of those states is **uniform**, then this probability will be

$$p_i = \frac{1}{\Omega} \tag{2.1.2}$$

for any of the states $i = 1, \cdots, \Omega$. So we see that

$$S = -\log p \tag{2.1.3}$$

for a uniform distribution with all $p_i = p = 1/\Omega$. But notice that this can also be viewed as

$$S \equiv -\sum_i p_i \log p_i \tag{2.1.4}$$

since all p_i are equal, and their sum is 1 (since probabilities are normalized). We have not yet demonstrated it for distributions that are not uniform, but this will turn out to be the most useful general notion of entropy. It can also be applied for continuous probability distributions, where

$$S \equiv -\int dx\, p(x) \log p(x) \tag{2.1.5}$$

Note that $0 \log 0 = 0$ (this is what we find from a limit).

This definition of entropy arises in information theory as follows. Say we have a long sequence

$$\cdots 011100101000011 \cdots \tag{2.1.6}$$

If this is truly a fully random sequence, then the number of possibilities is 2^N if it has length N, and the entropy is just the log of this quantity. This also means that the entropy per bit is

$$S_{bit} = \frac{1}{N} \log 2^N = \log 2 \tag{2.1.7}$$

But what if we know that in fact 0s occur with probability p, and 1s with probability $1 - p$? For a long sequence, we'll have roughly pN of the 0s. And by the central limit theorem, we'll be much less likely to have sequences with significantly more or fewer 0s. So this means that the number of such messages will be

$$\begin{aligned} \frac{N!}{(pN)!((1-p)N)!} &\approx \frac{N^N}{(pN)^{pN}(N-pN)^{N-pN}} \\ &= \left(\frac{1}{p^p(1-p)^{1-p}}\right)^N = e^{NS_{bit}} \end{aligned} \tag{2.1.8}$$

where we see that

$$S_{bit} = -p \log p - (1-p) \log(1-p) \tag{2.1.9}$$

So the entropy S that we've studied in statistical mechanics, ie the log of the number of possible sequences, naturally turns into the definition in terms of probability distributions discussed above.

This has to do with information theory because NS quantifies *the amount of information we actually gain by observing the sequence*. Notice that as Warren Weaver wrote in an initial popularization of Shannon's ideas: "The word information in communication theory is not related to what you do say, but to what you could say. That is, information is a measure of one's freedom of choice when one selects a message."

2.2 Choice, Uncertainty, and Entropy

Shannon showed in his original paper that entropy has a certain nice property. It characterizes how much 'choice' or 'uncertainty' or 'possibility' there is in a certain random process. This should seem sensible given that we just showed that entropy counts the amount of information that a sequence can carry.

Let's imagine that there are a set of possible events that can occur with probabilities $p_1, \cdots, p_n$. We want a quantity that measures how much 'possibility' there is in these events. For example, if $p_1 = 1$ and the others are 0, then we'd expect there's no 'possibility'. While if $n \gg 1$ and the p_i are uniform, then 'almost anything can happen'. We want a quantitate measure of this idea.

We will show that entropy is the unique quantity $H(p_i)$ with the following natural properties:

1. H is continuous in the p_i.

2. If all p_i are equal, so that all $p_i = 1/n$, then H should increase with n. That is, having more equally likely options increases the amount of 'possibility' or 'choice'.

3. If the probabilities can be broken down into a series of events, then H must be a weighted sum of the individual values of H. For example, the probabilities for events A, B, C

$$\{A, B, C\} = \left\{\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\right\} \tag{2.2.1}$$

can be rewritten as the process

$$\{A, B \text{ or } C\} = \left\{\frac{1}{2}, \frac{1}{2}\right\} \quad \text{then} \quad \{B, C\} = \left\{\frac{2}{3}, \frac{1}{3}\right\} \tag{2.2.2}$$

We are requiring that such a situation follows the rule

$$H\left(\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\right) = H\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{1}{2} H\left(\frac{2}{3}, \frac{1}{3}\right) \tag{2.2.3}$$

We will formalize this soon using the conditional entropy.

We will show that up to a positive constant factor, the entropy S is the only quantity with these properties.

Before we try to give a formal argument, let's see why the logarithm shows up. Consider

$$\begin{aligned} H\left(\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4}\right) &= H\left(\frac{1}{2},\frac{1}{2}\right)+2\frac{1}{2}H\left(\frac{1}{2},\frac{1}{2}\right) \\ &= 2H\left(\frac{1}{2},\frac{1}{2}\right) \end{aligned} \tag{2.2.4}$$

Similarly

$$H\left(\frac{1}{8},\cdots,\frac{1}{8}\right)=3H\left(\frac{1}{2},\frac{1}{2}\right) \tag{2.2.5}$$

and so this is the origin of the logarithm. To really prove it, we will take two fractions, and note that $\frac{1}{s^n}\approx\frac{1}{t^m}$ for sufficiently large n and m. The only other point we then need is that we can approximate any other numbers using long, large trees.

Here is a formal argument. Let

$$A(n)\equiv H\left(\frac{1}{n},\cdots,\frac{1}{n}\right) \tag{2.2.6}$$

for n equally likely possibilities. By using an exponential tree we can decompose a choice of s^m equally likely possibilities into a series of m choices among s possibilities. So

$$A(s^m)=mA(s) \tag{2.2.7}$$

We have the same relation for some t and n, ie

$$A(t^n)=nA(t) \tag{2.2.8}$$

By taking arbitrarily large n we can find an n,m with $s^m\leq t^n<s^{m+1}$, and so by taking the logarithm and re-arranging we can write

$$\frac{m}{n}\leq\frac{\log t}{\log s}\leq\frac{m}{n}+\frac{1}{n} \tag{2.2.9}$$

which means that we can make

$$\left|\frac{m}{n}-\frac{\log t}{\log s}\right|<\epsilon \tag{2.2.10}$$

Using monotonicity we have that

$$mA(s)\leq nA(t)<(m+1)A(s) \tag{2.2.11}$$

So we also find that

$$\left|\frac{m}{n} - \frac{A(t)}{A(s)}\right| < \epsilon \tag{2.2.12}$$

and so we conclude via these relations that

$$A(t) = K \log t \tag{2.2.13}$$

for some positive constant K, since we can make ϵ arbitrarily small.

But now by continuity we are essentially done, because we can approximate any set of probabilities p_i arbitrarily well by using a very fine tree of equal probabilities.

Notice that the logarithm was picked out by our assumption 3 about the way that H applies to the tree decomposition of a sequence of events.

2.3 KL Divergence or Relative Entropy, and the Fisher Metric

The relative entropy, called the Kullback–Leibler (KL) Divergence in the ML community, provides one measure of the similarity between two probability distributions. However, the KL is not a true distance metric, since it's not symmetric and does not satisfy the triangle inequality.

The KL is the answer to the question 'if I have data described by a probability distribution p, and a theory for that data q, what's the rate at which data collection will inform me that my theory differs from the true distribution'?

To see this, let us consider the concrete example of a binomial distribution. If we observe N examples from a true distribution with probabilities p and $1-p$, then the number of possible sequences is roughly N choose pN. But the model with probabilities q and $1-q$ will assign a probability to this outcome given by

$$\begin{aligned} \frac{N!}{(pN)!((1-p)N)!} \times q^{Np}(1-q)^{N(1-p)} &\approx \frac{N^N q^{Np}(1-q)^{N(1-p)}}{(pN)^{pN}(N-pN)^{N-pN}} \\ &= \left(\left(\frac{q}{p}\right)^p \left(\frac{1-q}{1-p}\right)^{1-p}\right)^N = e^{-ND_{KL}(p||q)} \end{aligned} \tag{2.3.1}$$

The 'rate' idea comes from the dependence on N data points. More generally we will find that

$$P = 2^{-N\sum_i p_i \log p_i} \times \prod_i q_i^{Np_i} = 2^{-N\sum_i p_i \log(p_i/q_i)} \tag{2.3.2}$$

which can be demonstrated using a multinomial expansion. So the KL divergence is

$$D_{KL}(p||q) = \sum_i p_i \log\left(\frac{p_i}{q_i}\right) \tag{2.3.3}$$

As long as p and q are valid probability distributions we have

$$D_{KL}(p||q) \geq 0 \tag{2.3.4}$$

with equality only when $p = q$. Non-negativity follows from our original expression for the total probability P, because choosing $q \neq p$ decreases the total probability, increasing D_{KL}. *Minimizing D_{KL} is the same thing as maximizing the likelihood of the data given the model q.*

Let's think a bit more about the two reasons why the KL isn't a metric – failure of the triangle inequality, and asymmetry:

- Triangle Inequality: If we take binary models with $p = 1, p = 1/2, p = 0$ then the KL divergence between the first and last is infinite, but the KL divergence between each and the $p = 1/2$ model is finite.

- Asymmetry: Consider some distribution p which is a linear combination of two Gaussians, so it looks like 'two bumps', ie a bi-modal distribution. Let's think briefly about the choice of Gaussian q minimizing $D_{KL}(p||q)$ vs $D_{KL}(q||p)$.

  In the former case, we pay a huge penalty if there are any points x where $p(x)$ is non-negligible, but where q is tiny. This suggests that the q minimizing $D_{KL}(p||q)$ will be a broad distribution encompassing both components of p.

  In the latter case where we are instead minimizing $D_{KL}(q||p)$, we pay a huge penalty if p is tiny when q is significant. That means that in between the two bumps, where p is tiny, we need q to be tiny as well. This suggests that q will tend to be identified with one of the bumps of p, and ignore the other one.

  This toy thought experiment shows that the asymmetry of the KL has a dramatic effect on the kind of results we obtain when we minimize it.

Fisher Information Metric

Now let's think a bit more about the dependence of the KL on continuous parameters. If our probability distribution $q_\theta(x)$ depends on θ, then its natural to investigate how

$$D_{KL}(q_{\theta_0}||q_\theta) \tag{2.3.5}$$

behaves for small $\theta - \theta_0$. Interestingly, the first term in the expansion vanishes as

$$\int dx\, q_\theta(x) \nabla_\theta \log q_\theta(x) = \nabla_\theta \int dx\, q_\theta(x) = 0 \tag{2.3.6}$$

The second derivative (Hessian) is not zero, but it is symmetric! It is the Fisher matrix

$$\begin{aligned} F_{\mu\nu} &\equiv \nabla_\mu \nabla_\nu D_{KL}(q_0||q_\theta) \\ &= -\int dx\, q_0(x) \nabla_\mu \nabla_\nu \log q_\theta(x) \\ &= -\int dx\, q_0(x) \left(\frac{\nabla_\mu \nabla_\nu q_\theta(x)}{q_\theta(x)} - \frac{\nabla_\mu q_\theta(x) \nabla_\nu q_\theta(x)}{q_\theta(x)^2} \right) \\ &= \int dx\, q_0(x) \left(\nabla_\mu \log q_\theta(x) \nabla_\nu \log q_\theta(x) \right) \end{aligned} \tag{2.3.7}$$

Note that since $D_{KL}(q_0||q_\theta)$ is positive, *we know that F is positive definite.* It is both the covariance of $\nabla_\mu \log q_\theta(x)$ and the Hessian of the $D_{KL}(q_0||q_\theta)$. If we call

$$S = \log q_\theta(x) \tag{2.3.8}$$

the score, then the Fisher is the covariance of the gradient of the score. This is another way of immediately seeing that it is positive definite.

The Fisher matrix is interesting because it provides a local, coordinate independent metric on the space of probability distributions q_θ. From a statistics/experimental point of view, it provides us with an idea of the local 'resolving power' of the data X, as it tells us how much we have to change the parameters to get a significantly different prediction for the data.

Note though that the Fisher really is only local – we cannot obtain $D_{KL}(p||q)$ by integrating the Fisher, because once q differs from p more than infinitesimally, further changes in q relative to p are not captured by the Fisher matrix. This is a funny fact. If we have

$$D_{KL}(p_\theta||p_{\theta+\delta\theta_1}) + D_{KL}(p_{\theta+\delta\theta_1}||p_{\theta+\delta\theta_2}) + D_{KL}(p_{\theta+\delta\theta_2}||p_{\theta+\delta\theta_3}) + \cdots + D_{KL}(p_{\theta+\delta\theta_{n-1}}||p_{\theta+\delta\theta_n})$$

Then it would seem that we obtain a 'distance'

$$\delta_1 F_0 \delta_1 + \delta_2 F_1 \delta_2 + \cdots \delta_n F_{n-1} \delta_n \neq \int d\theta \hat{n}(\theta) F(\theta) \hat{n}(\theta) \tag{2.3.9}$$

Where equality fails because the LHS is quadratic in the infinitesimals. We can instead define something more like

$$\int d\theta \left[\lim_{\epsilon \to 0} \frac{1}{\epsilon} \sqrt{D_{KL}(p_\theta||p_{\theta+\epsilon\hat{n}(\theta)})} \right] = \int d\theta \sqrt{\hat{n}^i(\theta) F_{ij}(\theta) \hat{n}^j(\theta)} \tag{2.3.10}$$

which would produce a kind of dimensionless distance along $\hat{n}(\theta)$. Furthermore, we could specify that $\hat{n}$ was parallel transported along itself, so that we'd obtain a geodesic distance. But the magnitude of this distance is neither the KL nor the square root of the KL between start and finish.

Another thing one can define is a kind of coordinate independent volume element

$$dV = d^D\Theta \sqrt{\det F} \tag{2.3.11}$$

This is a natural way of measuring volumes on parameter space; it is coordinate independent. These properties make it a naturallly 'ignorant' prior. Specifically we can take the prior probability that parameters will lie in $d^D\Theta$ to be

$$\frac{d^D\Theta \sqrt{\det F}}{\int d^D\Theta \sqrt{\det F}} \tag{2.3.12}$$

assuming the denominator is finite. This is called the *Jeffrey's Prior.*

Concrete Example

Let's look at the KL between two Gaussians. WLOG we can assume the true distribution has mean 0 and $\sigma = 1$. Then the KL will be

$$\int dx \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \left(-\frac{x^2}{2} + \frac{(x-\mu)^2}{2\sigma^2} + \log \sigma \right) = \frac{\mu^2}{2\sigma^2} + \frac{1}{2\sigma^2} - \frac{1}{2} + \log \sigma \tag{2.3.13}$$

Though it's not entirely obvious, the expression on the right is positive, and only vanishes for $\mu = 0$ and $\sigma = 1$. The Fisher wrt σ, μ is simply a diagonal matrix with eigenvalues 2 and 1.

2.4 Maximum Entropy: Boltzmann and Gaussian Distributions

Many of the simple probability distributions that we encounter in statistics (and in physics) can be derived from a maximum entropy principle – they are the distribution with maximum entropy given some simple constraint. This is a fairly standard way of deriving the Boltzmann distribution, but it also provides a very elegant way to think about the central limit theorem. Many other distributions (not covered here) can be derived in a similar way.

Botlzmann Distribution

Boltzmann factors can be derived in a simple and principled way – they are the probabilities that **maximize the entropy given the constraint that the *average* energy is held fixed**. We can formalize this with some Lagrange multipliers, as follows.

We want to fix the expectation value of the energy and the total probability while maximizing the entropy. We can write this maximization problem using a function (Lagrangian)

$$L = -\sum_i p_i \log p_i + \beta \left(\langle E \rangle - \sum_i p_i E_i \right) + \nu \left(\sum_i p_i - 1 \right) \tag{2.4.1}$$

where we are maximizing/extremizing L with respect to the p_i and β, ν; the latter are Lagrange multipliers.

Varying gives the two constraints along with

$$-\log p_i - 1 - \beta E_i + \nu = 0 \tag{2.4.2}$$

which implies that

$$p_i = e^{\nu - 1 - \beta E_i} \tag{2.4.3}$$

Now ν just sets the total sum of the p_i while β is determined by the average energy itself. So we have re-derived Boltzmann factors in a different way. This derivation also makes it clear what abstract assumptions are important in arriving at $e^{-\beta E}$.

Boltzmann Factors and the KL

Let's say I have a probability distribution $p(X)$ and I re-weight it so that $q(x) = p(x)e^{\mu - \beta E(x)}$. If we fix the zero-point (overall constant shift) of the 'energies' $E(x)$ so that the distribution remains normalized, and work in units where $\beta = 1$, then

$$1 = \int dx\, q(x) = \int dx\, p(x) e^{-E(x)} \tag{2.4.4}$$

and the KL is

$$\begin{aligned} D_{KL}(p||q) &= \int dx\, p(x) \log \frac{p(x)}{p(x)e^{-E(x)}} \\ &= \int dx\, p(x) E(x) \\ &= \langle E \rangle_p \end{aligned} \tag{2.4.5}$$

So the KL divergence is the expectation value of $E(x)$ in the old distribution (given that $E(x)$ includes an overall shift so that probabilities remain normalized).

Gaussian Distributions and the Central Limit Theorem

In fact, we can understand many other common distributions as the consequence of a simple constraint + maximum entropy.

For example, **the Gaussian distribution has maximum entropy given the constraint of a fixed mean and variance**. If we write down the 'action' for the function $p(x)$ as

$$S = \int dx \left[-p(x) \log p(x) + \lambda_1 (p(x)x - \mu) + \lambda_2 (p(x)x^2 - \sigma^2) + \nu \left(p(x) - 1 \right) \right] \tag{2.4.6}$$

then we find ab 'equation of motion' from varying $p(x)$ as

$$\log p(x) - 1 + \lambda_1 x + \lambda_2 x^2 + \nu = 0 \tag{2.4.7}$$

which implies that

$$p(x) = \mathcal{N} e^{\lambda_2 x^2 + \lambda_1 x} \tag{2.4.8}$$

so that it must be a Gaussian, where λ_1, λ_2 are simply fixed by the fact that the mean is μ and the variance σ^2. The normalization is fixed in the obvious way.

Note that this provides a very elegant way to think[2] about the **central limit theorem**, which we will prove in a much more formal way when we discuss probability. The idea is that if we add together many different random variables drawn from various distributions, we should expect the resulting distribution to have (in essence) as much entropy as possible. And in this section we just showed that the distribution with maximum entropy, given a fixed mean and variance, is the Gaussian. This is pretty close to a proof that a sum of a very large number of random variables much approach a Gaussian distribution, ie the central limit theorem!

[2] I thank Brice Menard for emphasizing this perspective.

2.5 Conditional Entropy and Mutual Information

Here are two other interesting notions of entropy or information.

Conditional Entropy

The condition entropy is the expectation over y of a conditional probability distribution $p(x|y)$. In math this means

$$\begin{aligned} S(X|Y) &\equiv -\sum_{x,y} p(x,y) \log p(x|y) \\ &= -\sum_{y} p(y) \sum_{x} p(x|y) \log p(x|y) \end{aligned} \tag{2.5.1}$$

Note that this is highly asymmetric in $x \leftrightarrow y$. In the simplification we have used $p(x|y) = p(x,y)/p(y)$, ie Bayes rule. The conditional entropy is non-negative.[3]

Note that with these definitions we have the factorization rule

$$S(X,Y) = S(Y) + S(X|Y) \tag{2.5.2}$$

This is just a restatement of our most non-trivial 'axiom' for the entropy itself.

Mutual Information

The mutual information is

$$\begin{aligned} I(X,Y) &= S(X) + S(Y) - S(X,Y) \\ &= \sum_{x,y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \\ &= S(X) - S(X|Y) \\ &= S(Y) - S(Y|X) \end{aligned} \tag{2.5.3}$$

This is a measure of how much one random variable knows about another. It's always non-negative, is symmetric in $x \leftrightarrow y$, and only vanishes if the variables are independent. Note that we can interpret

$$I(X,Y) = D_{KL}\left[p(x,y)||p(x)p(y)\right] \tag{2.5.4}$$

which provides a demonstration that the mutual information is non-negative.

[3] In the quantum mechanical generalization, we replace probabilities with density matrices, and we obtain classical information theory in the limit of diagonal density matrices. The general case is more subtle, and quantum conditional entropy isn't necessarily positive.

Monotonicity of KL Divergence

The KL divergence tells us about the rate at which we learn that our model q is not the same as the ground truth distribution p. This suggests that if we have joint distributions $q(x, y)$ and $p(x, y)$, then this rate is larger as compared to if we only observe X from $q(x), p(x)$. This implies

$$D_{KL}(p(x, y)||q(x, y)) \geq D_{KL}(p(x)||q(x)) \tag{2.5.5}$$

called the monotonicity of relative entropy. We can prove it mathematically by noting that the difference between the two sides is

$$\sum_i p(x_i) \sum_j \frac{p(x_i, y_j)}{p(x_i)} \log \left(\frac{p(x_i, y_j)/p(x_i)}{q(x_i, y_j)/q(x_i)} \right) = \sum_i p(x_i) D_{KL}(p(Y|x_i)||q(Y|x_i)) \geq 0 \tag{2.5.6}$$

where its positive because its a positive combination of KL divergences of conditional probability distributions. Using the relation between the KL and the mutual information, we can use this prove the strong sub-additivity[4] inequality $S_{XY} + S_{YZ} \geq S_Y + S_{XYZ}$.

2.6 t-SNE

This is a famous technique for projecting high-dimensional data in X into a lower-dimensional space Y while preserving some of the structure, particularly its local structure and clustering. SNE stands for 'Stochastic Neighbor Embedding'. The method works using basic ideas from information theory. You should play with visualizations of it here (https://distill.pub/2016/misread-tsne/).

In t-SNE, the similarity of datapoints i and j is given by the conditional probability $p_{j|i}$ that datapoint i would pick datapoint j as its neighbor if neighbors were picked in proportion to their probability density under a Gaussian centered at x_i. That is

$$p_{j|i} = \frac{e^{-\frac{(x_j - x_i)^2}{\sigma_i^2}}}{\sum_{k \neq i} e^{-\frac{(x_k - x_i)^2}{\sigma_i^2}}} \tag{2.6.1}$$

where σ_i^2 is a variance.

We can try to preserve these probabilities in the lower-dimensional space Y, so that

$$q_{j|i} = \frac{e^{-(y_j - y_i)^2}}{\sum_{k \neq i} e^{-(y_k - y_i)^2}} \tag{2.6.2}$$

If the map from $X \to Y$ correctly models this notion of similarity, then $q_{j|i} = p_{j|i}$. Since these are probability distributions, it's natural to try to make them similar by minimizing the KL Divergence

$$L = \sum_i D_{KL}(P_i||Q_i) = \sum_{i,j} p_{j|i} \log \frac{p_{j|i}}{q_{j|i}} \tag{2.6.3}$$

[4]These proofs are very simple for classical probability distributions, relying only on conditionalization. In the quantum case the analogous statements remain true, but are much less obvious.

This is biased towards maintaining the local structure in the data, because distant points have small $p_{j|i}$ and thus do not contribute much to the KL.

But how do we determine σ_i? We would like smaller σ_i in dense regions, and larger σ_i in sparse regions. t-SNE chooses each σ_i in order to fix the entropy of the distribution $p_{j|i}$ (or equivalently its perplexity, which is the exponential of the entropy):

$$\text{Perp}(P_i) = 2^{S(P_i)} \tag{2.6.4}$$

where base 2 entropies are used by convention. It's typical to choose this parameter between 5 and 50 in applications. Perplexity should be interpreted as a smooth measure of the number of neighbors.

We can choose the σ_i right at the start by computing the perplexities of the $p_{j|i}$ distributions in the original, high-dimensional space X. Then we can use optimization to minimize the KL-divergence loss – we just start t-SNE with a random initialization of the map $f : X \to Y$ and minimize the loss through gradient descent.

2.7 Aside: Limits on Computation

In practice, computation has a number of inefficiencies. Despite decades of progress, my impression is that most of the waste is still due to electrical resistance and heat dissipation. We'll perform a relevant estimate in a moment. But what are the ultimate limits on computation?

We have seen several times that irreversible processes are those that create entropy. Conversely, for a process to be reversible, it must not create any entropy.

However, computers tend to create entropy, and thus waste, for what may be a surprising reason – they *erase* information. For example, say we add two numbers, eg

$$58 + 23 = 81 \tag{2.7.1}$$

We started out with information representing both 58 and 23. Typically this would be stored as an integer, and for example a 16 bit integer has information, or entropy, $16 \log 2$. But at the end of the computation, we don't remember what we started with, rather we just know the answer. Thus we have created an entropy

$$S = 2 \times (16 \log 2) - (16 \log 2) = 16 \log 2 \tag{2.7.2}$$

through the process of erasure!

Since our computer will certainly be working at finite temperature, eg room temperature, we will be forced by the laws of thermodynamics to create heat

$$Q = k_B T (16 \log 2) \approx 5 \times 10^{-20} \text{ Joules} \tag{2.7.3}$$

Clearly this isn't very significant for one addition, but it's interesting as its the fundamental limit.

Furthermore, computers today are very powerful. For example, it has been estimated that while training AlphaGo Zero, roughly

$$10^{23} \text{ float point operations} \tag{2.7.4}$$

were performed. Depending on whether they were using 8, 16, or 32 bit floating point numbers (let's assume the last), this meant that erasure accounted for

$$Q = k_B T(32 \log 2) \times 10^{23} \sim 8000 \text{ Joules} \tag{2.7.5}$$

of heat. That's actually a macroscopic quantity, and the laws of thermodynamics say that it's impossible to do better with irreversible computation!

But note that this isn't most of the heat. For example, a currently state of the art GPU like the Nvidia Tesla V100 draw about 250 Watts of power and perform at max about 10^{14} flop/s. This means their theoretical minimum power draw is

$$Q = k_B T(16 \log 2) \times 10^{14} flop/s \sim 10^{-5} \text{ Watts} \tag{2.7.6}$$

Thus state of the art GPUs are still tens of millions of times less efficient than the theoretical minimum. We're much, much further from the theoretical limits of computation than we are from the theoretical limits of heat engine efficiency.

In principle we can do even better through *reversible computation*. After all, there's no reason to make erasures. For example, when adding we could perform an operation mapping

$$(x, y) \to (x, x + y) \tag{2.7.7}$$

for example

$$(58, 23) \to (58, 81) \tag{2.7.8}$$

so that no information is erased. In this case, we could in principle perform any computation we like without producing any waste heat at all. But we need to keep all of the input information around to avoid creating entropy and using up energy.

3 Probability and Maximum Likelihood

Our goal is to understand how ideas from information theory and probability combine to help us choose natural loss functions for a wide variety of ML tasks. As we will see, the **maximum likelihood** principle plays a predominant role. Maximum likelihood is very closely connected to the KL divergence, which we discussed in the last section. We'll also show how these ideas can be partially justified from a Bayesian perspective.

3.1 Setup and Some Philosophy

A large portion – probably the significant majority – of machine learning tasks[5] are set up in such a way that we have a model $p_\theta(X)$ for the *probability distribution* over the *data* X that depends

[5]Another option is that our models provide a way to make a decision, but don't have a probabilistic interpretation, usually because imposing a probabilistic interpretation would require normalizing an intractable distribution. This is like knowing which state has the lowest energy without knowing the full partition function... and in fact such models are usually called 'Energy Models'.

on *parameters* θ. We have a finite set of data points $\{x_i\}$, and we want to *learn* some sort of 'best estimate' for the parameters θ that 'explain' the distribution of the data X. In this framing, we usually imagine that there is some 'ground truth' underlying distribution $P(X)$ from which our data sample is drawn, but we can only access it via sampling.

Furthermore, we'll almost always approach this problem by identifying some sort of objective or *loss* that depends on both X and p_θ, which we may write as $L(X; p_\theta)$ or more concretely $L(x_i; \theta)$ as a function of literal data points. Our goal will be to *optimize* for this objective, ie to minimize the loss L. If we're searching for specific θ_{opt}, then these choices result in the choice of an *estimator* for θ.

We could also try to learn a distribution over the θ themselves – this is like learning a probability distribution over the 'worlds' θ that we may be living in. The distinction between a point estimate and a distribution over θ is associated with the complicated and somewhat fuzzy frequentist vs bayesian discussion.

Given this framing, there are a number of questions about how to proceed; some rather practical, others rather philosophical:

1. What should our goal be – to determine a fixed parameter value θ_{opt} that's best, or should we be learning (more generally) a probability distribution $q(\theta)$ that best matches the data?

2. How should we *choose the loss function*? What does this choice depend on?

3. How do we optimize θ or the distribution over Θ to determine the best possible values?

4. What aspects of this process are stochastic vs deterministic? Is it the data X that's random, or the value of the parameters θ, or neither/both?

5. The parameters θ are usually quite arbitrary. As theorists, we may in fact prefer to think about $p(X)$ as some sort of infinite dimensional space of possible probability distributions, where the θ just parameterize a subset of that space. This makes us wonder:

    - Is there a more natural basis for $p(X)$, and thus for θ? Natural with respect to what?
    - What sort of bias are we introducing by choosing the subset of all $p(X)$ parameterized by a neural network $p_\theta(X)$?

We'll be able to address some of these questions.

The probabilistic approach has some limitations, most importantly that we may be able to accomplish many tasks without ever defining a normalizable probability distribution. For instance, if I want to classify an image, I don't need to output a probability distribution over labels – I just need to choose one! And similarly, if I want to generate an image, I don't need to know its probability – I just want a pretty picture! But despite these observations, the philosophy we have outlined remains the dominant paradigm. And it *can't hurt* to build a probabilistic model, when it's possible, even if we don't always need one.

3.2 Central Limit Theorem

It's fun to derive the centrally important Central Limit Theorem. First let's give some formal setup to make it easy to do in general. The formalism will be useful for other purposes as well. Aside for fun and importance, this is a good exercise to get comfortable with random variables.

Cumulants and Generating Functions

Given any probability distribution $p(X)$ for a variable x, we also say that $x \sim p$ for the words 'x is a random variable drawn from the distribution p'. The moments of x are

$$\langle X^n \rangle = \int dx\, p(x) x^n \tag{3.2.1}$$

and these are generated by the Fourier Transform of p, also called the characteristic function

$$\tilde{p}(k) = \int dx\, p(x) e^{-ikx} \tag{3.2.2}$$

The moments are generated in the sense that the nth derivative of $\tilde{p}(k)$ wrt k produces moments when evaluated at zero. The *cumulant generating function* is just the log of $\tilde{p}$, that is

$$G = \log \tilde{p}(k) \tag{3.2.3}$$

It has the nice property that it only generates the *connected part* of the moments, which are called the cumulants, so that

$$G = \sum_{n=1}^{\infty} \frac{(-ik)^n}{n!} \langle X^n \rangle_c \tag{3.2.4}$$

So for example the variance of the original distribution is simply $\langle X^2 \rangle_c$. This pattern of phenomena may be familiar from the Free Energy in Statistical Mechanics and from expansions in QFT.

To prove the fact that G generates the connected part, it's easiest to work backward. If we had an G generating the connected part, then it's easy to see that e^G would generate all possible correlations, connected or disconnected, so that $e^G = \tilde{p}$. We see this diagrammatically at kth order by drawing k points and grouping them together in all possible ways, and then counting the number of such groupings. In equations this is

$$\sum_{n=1}^{\infty} \frac{(-ik)^n}{n!} \langle X^n \rangle = \exp\left[\sum_{n=1}^{\infty} \frac{(-ik)^n}{n!} \langle X^n \rangle_c \right] = \prod_n \sum_{p_n} \left[\sum_{n=1}^{\infty} \frac{(-ik)^{n p_n}}{p_n!} \left(\frac{\langle X^n \rangle_c}{n!} \right)^{p_n} \right] \tag{3.2.5}$$

Matching powers of k gives

$$\langle x^m \rangle = \sum_{\{p_n\}} \prod_n \frac{1}{p_n! (n!)^{p_n}} \langle x^n \rangle_c^{p_n} \tag{3.2.6}$$

and leads to the graphical interpretation.

Quick Examples

Note that a delta function is a distribution with non-trivial mean, but vanishing $n > 1$ cumulants. A Gaussian distribution is the unique distribution with general mean and variance, but vanishing $n > 2$ cumulants. Our formalism makes this obvious because it dictates that F must be quadratic. Similarly, distributions with vanishing $n > k$ moments are just the Fourier transform of the exponential of a polynomial!

If we have two random variables $X \sim p$ and $Y \sim q$ (read 'x drawn from p'), where p and q are independent distributions, then obviously

$$\langle X + Y \rangle = \langle X \rangle + \langle Y y \rangle \tag{3.2.7}$$

Cumulants are defined in such a way that they continue this property to all orders, so that

$$\langle (X + Y)^n \rangle_c = \langle X^n \rangle_c + \langle Y^n \rangle_c \tag{3.2.8}$$

for the cumulants (but obviously not for the expectation values!). To see this, it's sufficient to simply note that $G_{pq} = G_p + G_q$. In particular, we learn the well-known and elementary fact that the standard deviation of a sum of random variables is the RMS of the individual standard deviations.

Central Limit Theorem

The proceeding discussion was useful because it makes it very easy to prove the central limit theorem. Let's imagine we have a random variable

$$X = \sum_{i=1}^{N} X_i \tag{3.2.9}$$

where X_i are some independent random variables. Then the distribution for X is

$$p(X) = \int d^N \vec{x} p(x_i) \delta(X - \sum_i x_i) \tag{3.2.10}$$

and so the characteristic function is simply

$$\tilde{p}_X(K) = \left\langle e^{-ik \sum_j x_j} \right\rangle = \tilde{p}_i(k_i = K) \tag{3.2.11}$$

and we can compute the moments of X from it.

This immediately tells us that

$$\langle X \rangle_c = \sum_{i=1}^{N} \langle X_i \rangle_c, \quad \langle X^2 \rangle_c = \sum_{i,j=1}^{N} \langle X_i X_j \rangle_c, \cdots \tag{3.2.12}$$

If the variables x_i are independent then all cross cumulants vanish, and so

$$\langle X^n \rangle_c = \sum_{i=1}^{N} \langle X_i^n \rangle_c \tag{3.2.13}$$

If all variables come from the same distribution then this is just $N\langle X^n\rangle_c$.

In the limit that N is large, this proves the central limit theorem. This follows because if we rescale to the variable

$$y = \frac{X - N\langle X_1\rangle_c}{\sqrt{N}} \tag{3.2.14}$$

we find a random variable with zero mean, and all its cumulants scale as $N^{1-n/2}$. So at large N only the variance is non-vanishing, and thus we have a Gaussian distribution.

Our methods make it clear when the central limit theorem would apply to N non-independent random variables – what we need is that the sum over the correlated cumulants is sub-leading as compared to the independent terms $\sum_{i=1}^{N}\langle X_i^n\rangle_c$, which grow linearly with N.

3.3 Estimators, Bias, and Variance

Let's imagine that we believe that some data X was generated by randomly drawing samples from a probability distribution $P(X|\theta)$, where we view θ as parameters for our model. An estimator is a function or algorithm that produces a map $f : X \to \theta$ that 'estimates' the parameters θ (or a sub-set of them, or a function of them) using a data sample X.

This approach is very useful practically. Philosophically, it tends to be associated with Frequentist (in contrast to Bayesian) statistics. With the latter we would instead determine a probability distribution over θ reflecting what we know about θ from our prior and the data.

We are discussing estimators for the obvious reason that in ML, the learning process is simply an algorithm to take the data and turn it into a prediction for the NN parameters θ. A difficulty that we'll almost always gloss over is that real world data simply was not drawn from a distribution $P(X|\theta)$ parameterized by NN parameters. Presumably the world is much, much more complicated, and our NN models will merely approximate it in a largely uncontrolled way.

Intuitively, we would like our estimators to be *unbiased*, so that as we accumulate more and more data, they actually converge towards the correct underlying value of the parameters. Furthermore, we would like them to have a *small variance*, so that they have as little 'noise' as possible. Concretely, we want to minimize variance so that we can predict θ as precisely as possible with as little data as possible.

A major topic discussed in ML is the *tradeoff between bias and variance*. This may seem sort of trivial for now, though it'll become more involved when we study Reinforcement Learning. Let's clarify via some simple examples.

3.3.1 Defining an Estimator as a Random Variable

Let's say we have a sample of N datapoints $\{x_i\}$, regarded as literal sampled data values. An estimator will be a function f that predicts the parameters $\theta_{\text{estimated}} = f(x_i)$.

To analyze our estimator, it's convenient to think of the x_i as separate, abstract random variables X_i drawn from the data distribution $P(X)$ from the 'world'. In this sense we can talk about the estimator as a random variable

$$\Theta_{\text{est}} = f(X_i) \tag{3.3.1}$$

since a function of random variables can itself be regarded as a new random variable. Note that f can depend both directly and indirectly on N, ie we have a different estimator for each value of N.

3.3.2 Bias of an Estimator

The bias is just $\langle \Theta_{\rm est} \rangle - \theta_{\rm true}$, or in slightly more detail

$$\langle \Theta_{\rm est} \rangle - \theta_{\rm true} = \langle f(X_i) \rangle - \theta_{\rm true} \tag{3.3.2}$$

To be clear, let's consider the most elementary example, a Gaussian. If

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.3.3}$$

then we'd like to estimate μ, σ from a finite set of data samples. The 'obvious' way to do this is to compute the mean via

$$\mu_{\rm est} = \frac{1}{N}\sum_i^N X_i \tag{3.3.4}$$

It's expectation is

$$\begin{aligned} \langle \mu_{\rm est} \rangle &= \frac{1}{N}\sum_i^N \langle X_i \rangle = \frac{N}{N}\int dx \frac{x}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \\ &= \mu \end{aligned} \tag{3.3.5}$$

so we see that this is an *unbiased* estimator of the mean μ. Note that in the second step we used the fact that expectation values are linear.

If we're not careful, we might estimate σ in a biased way. You might think that we can estimate

$$\sigma^2_{\rm naive} = \frac{1}{N}\sum_i^N \left(X_i - \frac{1}{N}\sum_j^N X_j \right)^2 \tag{3.3.6}$$

It's expectation is

$$\begin{aligned} \langle \sigma^2_{\rm naive} \rangle &= \frac{1}{N}\left\langle \sum_{i=1}^N \left(X_i^2 + \frac{1}{N^2}\left(\sum_j X_j\right)^2 - \frac{2}{N} X_i \sum_j X_j \right) \right\rangle \\ &= \frac{1}{N}\left\langle \sum_{i=1}^N \left(1 + \frac{N}{N^2} - \frac{2}{N}\right) X_i^2 + \sum_{j\neq i} X_i X_j \left(\frac{2}{N^2} - \frac{2}{N}\right) \right\rangle \\ &= \frac{N-1}{N}\langle X^2 \rangle - \frac{N-1}{N}\langle X \rangle^2 \\ &= \frac{N-1}{N}\langle (X-\mu)^2 \rangle \end{aligned} \tag{3.3.7}$$

but as you can see, this gives a *biased estimate* of the variance. For any finite value of N, we will estimate that the variance is too small by a factor of $1 - \frac{1}{N}$. The correct (unbiased) estimator is instead

$$\sigma^2_{\text{unbiased}} = \frac{1}{N-1}\sum_i^N \left(X_i - \frac{1}{N}\sum_j^N X_j\right)^2 \tag{3.3.8}$$

This has an intuitive interpretation in terms of 'degrees of freedom'. The mean is the 'center of mass' of the X_i, and so it doesn't contribute to the estimate. That's why our σ_{naive} was wrong; it didn't account for the fact that the variance is being computed with respect to this center of mass. This is a famous and very basic result in statistics called the 'Bessel correction'.

3.3.3 Variance of an Estimator

The variance of an estimator is just the variance of θ_{est}. Note that the variance of the estimator is not the same thing as the variance of the underlying true distribution. But it's easy to get confused.

The variance of our estimator for μ would just be

$$\begin{aligned}
\langle \text{Var}(\mu_{\text{est}})\rangle &\equiv \langle \mu^2_{\text{est}}\rangle - \langle \mu_{\text{est}}\rangle^2 \\
&= \left\langle \left(\frac{1}{N}\sum_i^N X_i\right)^2 \right\rangle - \left(\left\langle \frac{1}{N}\sum_i^N X_i \right\rangle\right)^2 \\
&= \frac{1}{N}\sigma^2
\end{aligned} \tag{3.3.9}$$

However, we can more clearly separate out the 'variance' of the distribution and that of the estimator by considering the variance of the estimator σ^2_{est}

$$\text{Var}(\sigma^2_{\text{unbiased}}) = \left\langle \left(\frac{1}{N-1}\sum_i^N \left(X_i - \frac{1}{N}\sum_j^N X_j\right)^2\right)^2 \right\rangle - \sigma^4 \tag{3.3.10}$$

where we note that the second term is the square of the expectation of the variance estimator, which (since it's unbiased) is $(\sigma^2)^2$. We can write

$$\left\langle \left(\frac{1}{N-1}\sum_i^N \left(X_i - \frac{1}{N}\sum_j^N X_j\right)^2\right)^2 \right\rangle = \frac{1}{(N-1)^2} v^{ia}v^{ib}v^{jc}v^{jd}\langle X_a X_b X_c X_d\rangle \tag{3.3.11}$$

where $v^{ij} = \delta^{ij} - \frac{1}{N}$. Note that

$$\langle x_a x_b x_c x_d\rangle = \left(\delta_{ab}\delta_{cd} + \delta_{ac}\delta_{bd} + \delta_{ad}\delta_{bc}\right)\sigma^4 + \cdots \tag{3.3.12}$$

where the ellipsis involves the mean, which cancel when contracted with v^{ij}. So we find

$$\begin{aligned}\frac{\sigma^4}{(N-1)^2}\left((v^{ia}v^{ia})^2+2(v^{ia}v^{ib}v^{ja}v^{jb})\right) &= \frac{\sigma^4}{(N-1)^2}\left((N-1)^2+2(N-1)\right)\\ &= \left(1+\frac{2}{N-1}\right)\sigma^4 \end{aligned} \tag{3.3.13}$$

which is amusingly reminiscent of the large N expansion. To evaluate the terms, we note that v^{ij} is a matrix with eigenvalues consisting of $N-1$ ones and 1 zero, so the first term is $(\mathrm{Tr}[v^2])^2=(N-1)^2$ while the second term is $\mathrm{Tr}[v^4]=N-1$. So we find

$$\mathrm{Var}(\sigma^2_{\mathrm{unbiased}})=\frac{2}{N-1}\sigma^4 \tag{3.3.14}$$

for a Gaussian. This makes it clear the the variance of an estimator is different from the variance of the original distribution.

3.3.4 Unclean Denoising

As an application of these ideas, consider the problem of 'denoising' – we have a large set of images, but they have been corrupted in some way by 'noise'. The goal is to process them into clean images, without the noise.

The naive approach would be to learn a map from noisy images to clean images. But this requires having clean versions of all of the images. What if we don't have any clean images?

Actually, if we simply try to learn a map from one noisy image to a different noisy image of the same object, in the process we'll learn a map from noisy images to clean images! Do you see why? (Depending on the type of noise, we may need to select somewhat different sorts of loss functions.) The point is that while the noisy outputs may have high variance, as long as they're essentially unbiased, the best map we can learn will be one that removes the noise. This simple observation was the basis of an NVIDIA 'Noise2Noise' paper in early 2018.

3.4 Why Optimize a Particular Loss Function?

It's very often the case that our neural network $p_\theta(x)$ computes a probability distribution on the data space X. For example, the NN may compute a probability that the next word in a sentence is 'the', or that a particular configuration of pixels is really in the dataset. Alternatively, the network may compute the probability that a datapoint $x\in X$ has some property y, so that $p_\theta(x)$ is the probability distribution over Y. For instance, the NN may assign a probability that an image contains the numeral '7'.

In either case, there's an intuitively natural objective – we would like to **maximize the likelihood** that the model assigns to the empirical data. This means that we will use the estimator

$$\theta_{\mathrm{opt}}=\mathrm{argmax}_\theta\left[p_\theta(X)\right] \tag{3.4.1}$$

Formally this is a product of $p_\theta(x_i)$ over all the data points x_i, which we can greatly simplify by taking a logarithm, so that we need only perform

$$\theta_{\text{opt}} = \text{argmax}_\theta \left[\frac{1}{N} \sum_{i=1}^{N} \log p_\theta(x_i) \right] \tag{3.4.2}$$

where we divided by N to compute the mean for simplicity, as the overall scale was arbitrary. So we can estimate θ by finding the value that maximizes the likelihood of the data sample; this is maximum likelihood estimation (MLE). This translates into a proposal for the loss in models with a probability interpretation, ie we should minimize

$$L(X;\theta) = -\frac{1}{N} \sum_{i=1}^{N} \log p_\theta(x_i) \tag{3.4.3}$$

Before moving on with the discussion, let's emphasize that this loss is essentially the same as using the **KL Divergence as the loss**. It's also often referred to as a 'cross-entropy loss'. To see why, note that this could be re-written as

$$\begin{aligned} L(X;\theta) &= -\sum_{i=1}^{N} P(X) \log \left(p_\theta(x_i) \right) \\ &= S(P) + \sum_{i=1}^{N} P(x_i) \log \left(\frac{P(x_i)}{p_\theta(x_i)} \right) \\ &= S(P) + D_{KL}(P(X)||p_\theta(X)) \end{aligned} \tag{3.4.4}$$

where only the KL term has any dependence on θ. This is called a 'cross-entropy' because of the expression in the first line, which looks like an entropy made from P and p_θ.

We should also emphasize that this framework applies when the data X is divided up into (x, y) where eg y represents a label for the data x. In this case it's very common to use the loss

$$L(X;\theta) = -\frac{1}{N} \sum_{i=1}^{N} \log p_\theta(y_i|x_i) \tag{3.4.5}$$

so that we maximize the likelihood that the model predicts the correct label y_i for each x_i. This is the most common situation in supervised learning, including image classification and autoregressive sequence prediction (one of the primary tools in language modeling).

The maximum likelihood principle should seem intuitively reasonable, but is it really necessary? And is it the best we can do?

3.4.1 Bayesian Perspective

We can motivate the maximum likelihood principle in a compelling way using Bayesian statistics. The Bayesian philosophy of probability and, perhaps more importantly, of decision theory, goes as

follows. We begin with some sort of prior expectation $p_{\text{prior}}(\theta)$ of the distribution over the parameters θ. Then, as we gather data, we *update* our beliefs by Bayes rule, forming the posterior

$$p_{\text{belief}}(\theta|X) = \frac{p_{\text{model}}(X|\theta)}{p_{\text{world}}(X)} p_{\text{prior}}(\theta) \tag{3.4.6}$$

From our point of view as observers, we have no direct information about $p_{\text{world}}(X)$ except through sampling.

If we accept this Bayesian procedure on a philosophical level, then it determines a distribution over θ that we should 'believe in' after seeing the data X. If we were to fully embrace the Bayesian perspective, we would simply compute p_{belief} itself.

However in practice we rarely even try to obtain p_{belief} as a distribution because it's too complicated, ie too computationally expensive. Instead we can try to obtain a point estimate, which in the Bayesian context is called a 'Maximum A Posteriori Estimation', sometimes abreviated 'MAP'. The most likely single value (or point estimate) of the parameter θ will be the one that maximizes $p_{\text{belief}}(\theta|X)$, which means that it must maximize

$$\theta_{\text{opt}} = \text{argmax}_\theta \left[\frac{1}{N} \log p_{\text{prior}}(\theta) + \frac{1}{N} \sum_{i=1}^{N} \log p_{\text{model}}(x_i|\theta) \right] \tag{3.4.7}$$

where we included some $1/N$ factors for a nice large N limit. Thus from this Bayesian perspective, we can *derive* the maximum likelihood principle if we assume that $p_{\text{prior}}(\theta)$ is actually a uniform distribution over θ.

If the prior is not uniform, then we instead find a different principle, which pushes us towards values of θ that are favored by the prior. As we will see, this provides a motivation for *regularizing* our models. For instance if our prior is a Gaussian centered at $\theta = 0$, then it effectively punishes us for 'learning' values of θ that are large numbers. That said, note that a true prior becomes less and less important as we see more and more data, whereas typical regularization schemes add a fixed constant term to the loss.

3.4.2 Minimizing the Variance and the Cramer-Rao Bound

Whatever estimator we use to determine θ_{opt}, we would like it to have as small a variance[6] as possible. In fact it's pretty easy to show that among estimators, the θ_{opt} estimated by maximum likelihood has the minimum possible variance. This means that the mean squared error between the estimated and true θ will be as small possible. The underlying reason is that the data is itself sampled (we assume) from $p(X|\theta_{\text{true}})$, and MLE is making direct use of this parameterization $p(X|\theta)$. That's why MLE is more natural and 'better' (lower variance) than other estimators.

To prove this, let us first establish a more general result, and then explain why the minimal variance of MLE follows. We can obtain a bound on the variance of an estimator using the general

[6]Maximum likelihood is also unbiased. A version of this is the statement that it is 'consistent', ie that in the infinite data limit we are guaranteed to converge to the correct value of θ, if it is unique.

fact (basically the Cauchy-Schwarz inequality)

$$\frac{\mathrm{Cov}(A,B)^2}{\mathrm{Var}(A)\mathrm{Var}(B)} \leq 1 \tag{3.4.8}$$

for any random variables. Here $\mathrm{Cov}(A,B)$ is the cumulant $\langle AB\rangle_c$ (so A, B are mean-subtracted).

Let's imagine that we have a set of data X drawn from a distribution $p(X|\theta)$ dependent on some parameters θ. We have some function $\phi(\theta)$ that we would like to estimate via $\hat{\phi}(X)$, where $\hat{\phi}$ is our estimator. We define the score as

$$S = \partial_\theta \log p(X|\theta) \tag{3.4.9}$$

and note that the expectation value $\langle S\rangle = 0$ because

$$\langle S\rangle = \partial_\theta \int p(X|\theta) dX \tag{3.4.10}$$

Now we assume that our estimator is unbiased, so that

$$\langle\hat{\phi}\rangle = \int \hat{\phi}(X) p(X|\theta) dX = \phi(\theta) \tag{3.4.11}$$

and note that as a consequence $\partial_\theta\phi(\theta)$ is related to the score via

$$\partial_\theta\phi(\theta) = \int \hat{\phi}(X) S(X) p(X|\theta) dX = \mathrm{Cov}(S,\hat{\phi}) \tag{3.4.12}$$

With a single parameter θ, we can then write the inequality

$$\frac{\mathrm{Cov}(S,\hat{\phi})^2}{\mathrm{Var}(\hat{\phi})\mathrm{Var}(S)} = \frac{(\partial_\theta\phi)^2}{\mathrm{Var}(\hat{\phi})F} \leq 1 \tag{3.4.13}$$

where $F = \mathrm{Var}(S)$ is the Fisher. When θ has many dimensions, we find in general that

$$\mathrm{Var}(\hat{\phi}) \geq (\nabla^i_\theta\hat{\phi}^\alpha) F^{-1}_{ij} (\nabla^j_\theta\hat{\phi}_\alpha) \tag{3.4.14}$$

which follows from the one-dimensional case if we work in a basis that diagonalizes F.

We have proved the Cramer-Rao inequality. It gives us a lower bound on the variance of any estimator. This is interesting because we would like our estimators to have as small variance as possible. The proof of the bound also tells us that in order to have minimal variance, an estimator must be 'aligned' with the score S, ie it must be that $\hat{\phi}$ varies in proportion with S.

Conclusions from Cramer-Rao

But the MLE is trying to maximize $\log p(X|\theta)$, and the score is just the derivative of this quantity. So the MLE is perfectly aligned with the score. The conclusion is that the MLE of equation (3.4.2)

has minimum possible variance. This provides another reason to use maximum likelihood for our loss function.

However, the logic isn't quite as straightforward as it may seem. What we have shown is that given a sample of N datapoints, if we actually find the θ that maximizes the likelihood, then MLE will minimize the variance of the θ will be minimum among estimators. But *this doesn't necessarily tell us much about the variance of the process of optimizing θ by maximum likelihood, eg the process of gradient descent from some initial θ_0 to a final θ.* For instance, it doesn't tell us how many samples we need to estimate the gradient for gradient descent, or even that the maximum likelihood loss has minimal gradient variance among possible loss functions. It only tell us that MLE has minimal variance for the final result of θ.

Another caveat to this analysis is that in the real world, $p(X|\theta)$ is just a model, and the real world does not actually correspond to any choice of the parameters θ. In this situation it's unclear to what extent our argument holds.

3.4.3 Regression Losses from Maximum Likelihood

You've almost certainly performed regression with a mean squared error loss:

$$L(\theta) = \frac{1}{B}\sum_{i=1}^{B}(f_\theta(x_i) - y_i)^2 \tag{3.4.15}$$

This loss is just the result of applying maximum likelihood in a situation where you know ahead of time that the errors will be Gaussian.

That is, if we know that the data is distributed as $y \in \mathcal{N}(\mu = F(x), \sigma)$ for a constant σ, then the likelihood is

$$p(y|x) = \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{1}{2\sigma^2}(y-F(x))^2} \tag{3.4.16}$$

and so the log likelihood will be

$$L(\theta) = \frac{1}{B}\sum_{i=1}^{B}\left[\frac{(f_\theta(x_i) - y_i)^2}{2\sigma^2} - \log\sigma\right] \tag{3.4.17}$$

Since the variance σ^2 is constant over the data, we can just ignore the last term.

If we know instead that the data has a different error, then we should use another loss. In most cases, the errors will have larger spread than a Gaussian, which means that our loss function should not penalize large deviations as heavily. For example, if the errors are distributed as $e^{-|x|}$ then we'd want an L^1 type loss

$$L(\theta) = \frac{1}{B}\sum_{i=1}^{B}|f_\theta(x_i) - y_i| \tag{3.4.18}$$

And if the errors are distributed according to a power law, such as the rather extreme case of $\frac{1}{x^2+\sigma^2}$ (extreme because it will have very large tails, so that its variance is infinite) then we would expect

$$L(\theta) = \frac{1}{B}\sum_{i=1}^{B} \log\left[1 + \frac{(f_\theta(x_i) - y_i)^2}{\sigma^2}\right] \tag{3.4.19}$$

to be the best choice of loss function.

In practical applications in ML, the differences among these choices are not obviously important, but they're worth keeping in mind. Occassionally a 'Huber loss' is used, which interpolates between Gaussian and exponential errors, ie between quadratic and linear behavior (linear dominating at large error, corresponding to an exponential, rather than Gaussian tail).

3.4.4 Calibrated Probabilities?

Will the probabilities from our models be well-calibrated? That is, when the model says that it assigns a probability of 99% that a given image contains a cat, will it be right 99/100 times?

Very naively yes, assuming that we hold 'everything else' constant, so that the model isn't changing in other ways while it learns probability assignments. For example, if a model actually gets the label cat correct a fraction f of the time, then after collecting a lot of data it will be trying to minimize a loss function associated with cats

$$L = -f \log p_{\text{cat}} - (1-f)\log(1 - p_{\text{cat}}) \tag{3.4.20}$$

This function is minimized for $p_{\text{cat}} = f$, which means that the model should eventually learn to assign probabilities that reflect the frequencies with which it is successful on the training data.

However, this analysis holds a lot constant that isn't really constant at all, so it's not necessarily a good guide to the performance of real models.

3.4.5 Comments on Regularization

Regularization is the process of mitigating overfitting.

We mentioned that from a Bayesian perspective, it's natural to include a regularization term in the loss corresponding to the prior... but that in fact, this regularization term should become less and less important as we accumulate more data. That said, we can sort of justify a fixed regularization term by noting that when training, we will go over the full dataset many many times (a pass over the full dataset is often called an 'epoch' of training), and we should not view the nth pass over the data as providing more information. Thus when training for many epochs, as is typical, even from a Bayesian perspective it's reasonable to include regularization terms.

Typical regularization terms are

$$L_{\text{reg}} = \lambda(W)^\alpha \tag{3.4.21}$$

where $\alpha = 1, 2$ are most common. These can be interpreted as Gaussian or exponential priors on the parameters W. It's less typical to regularize the bias terms b, though it can be done. Note that

$\alpha = 2$ and $\alpha = 1$ have rather different effects; the former only tends to suppress particularly large parameters, while the latter continues to exert 'pressure' on the parameters all the way to $W = 0$, and can thus help to induce a sparser spectrum of NN parameters.

There is another, very different approach to regularization – we can insert stochasticity into the training process to enhance robustness. SGD already does this, in a sense, via sub-sampling of the data distribution. We might also add noise to the data itself. But the most popular, and perhaps (nearly?) best form of regularization is a very simple method called **Dropout**.

With dropout, at each iteration of training, we simply zero out a fraction p of the NN activations. To avoid biasing the network, we then multiply the value of the remaining activations by $\frac{1}{1-p}$. Another option is to replace the second step with a factor of p in all activations post-training.

I don't think it's well-understood to what extent Dropout works better than other similar methods one might imagine, and if so, why. Though my impression from the lore is that adding noise to the parameters of a NN is significantly better than simply adding noise to the inputs.

3.5 Aside: Inserting Random Sampling into NNs

In ML, its *absolutely crucial to be able to differentiate the loss or goal with respect to the NN parameters*. But naively, if the output of a network is a discrete decision, this would seem to be impossible. We have already implicitly solved this problem in the case of discrete classification, by having the network output continuous probabilities, rather than discrete decisions, and by formulating a loss that's also a continuous function of those probabilities. But there are some other situations where it's useful to use 'tricks' to preserve differentiability.

The simplest version is the 'Reparameterization Trick' – it's rather trivial. Say we want our NN function to depend in part on some random numbers inserted into the middle of a computation. Naively we can't differentiate through this process, since the computation gets 'interrupted' by the insertion of the random values. But it remains differentiable if we instead insert the randomly sampled numbers at the beginning of the computation, and then have the neural network itself transform their distribution. So for example, the random numbers may be sampled from a Gaussian, and the NN maps that to a new Gaussian with mean and variance $(\mu(X), \sigma^2(X))$ where X is the (non-random) data, and μ and σ^2 are learned functions.

That is, we can draw a random variable from a learned Gaussian by sampling $z \sim \mathcal{N}(0, 1)$ and then computing

$$\mu(X) + \sigma(X)z \tag{3.5.1}$$

to produce a random variable drawn from $\mathcal{N}(\mu(X), \sigma(X))$. Below we will develop a version of this trick for discrete distributions.

Gumbel Distribution – Sampling Without a Softmax

Let's begin with a more-or-less natural question. Say we have a vector of n real numbers x_i. We can map this to a discrete distribution via the softmax

$$p_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3.5.2}$$

so that the p_i are positive and sum to 1, and so form a probability distribution over n discrete possibilities. Now if we like we can sample from this distribution of p_i. But we might ask... is there a way to sample from this discrete distribution without ever computing the p_i?

Specifically, we would like to sample by simply taking

$$\text{argmax}_i\{x_i + z_i\} \tag{3.5.3}$$

for a noise vector z_i drawn from a to-be-determined distribution $g(z)$, chosen so that the resulting discrete choices are distributed according to the p_i above.

Thus we would like to choose a distribution $g(z)$ so that

$$p_k = \frac{e^{x_k}}{\sum_j e^{x_j}} = \int dz_k g(z_k) \prod_{j\neq k}^{n} \left[\int dz_j g(z_j)\Theta(x_k + z_k - x_j - z_j)\right] \tag{3.5.4}$$

where $\Theta(x)$ is a step function which is 1 for $x > 0$ and 0 otherwise. Rather than evaluate the integrals, a natural way to proceed is to differentiate this expression with respect to all the x_j for $j \neq k$. This turns the $n-1$ step functions into delta functions, and acts simply on the LHS as well, giving

$$(n-1)!\frac{e^{\sum_{j=1}^n x_j}}{\left(\sum_j e^{x_j}\right)^n} = (n-1)!\prod_{j=1}^{n} p_j = \int dz_k g(z_k) \prod_{j\neq k}^{n} g(x_k + z_k - x_j) \tag{3.5.5}$$

It turns out that the *Gumbel distribution*[7]

$$g(z) = e^{-z-e^{-z}} = \partial_z\left(e^{-e^{-z}}\right) \tag{3.5.6}$$

satisfies this relation! It's straightforward to verify this via direct computation. Choosing $k = 1$ WLOG, the RHS of our relation above is

$$\int dz g(z)g(x_{12}+z)g(x_{13}+z)\cdots g(x_{1n}+z) = \int dz e^{-z-e^{-z}} e^{-z-x_{12}-e^{-z-x_{12}}} \cdots e^{-z-x_{1n}-e^{-z-x_{1n}}}$$

Changing variables to $e^z = y$, this is

$$\int_0^\infty dy\, y^{n-1} e^{-y} e^{x_{12}-ye^{-x_{12}}} \cdots e^{x_{1n}-ye^{-x_{1n}}} \quad = \quad e^{-x_{12}-\cdots-x_{1n}} \int_0^\infty dy\, y^{n-1} e^{-y(1+e^{-x_{12}}+\cdots e^{-x_{1n}})}$$

and the last integral is easy to perform, yielding the desired result.

So we can either compute the p_i from the x_i and then sample from them, or alternatively, we can add Gumbel-distributed noise to the x_i, and then simply take the argmax of $x_i + z_i$.

[7]The cumulative distribution $G(z) = e^{-e^{-z}}$ has the property that $[G(z)]^n = G(z - \log n)$. It's one of only three families of distributions that can 'stably' characterize the probability distribution associated with the maximum of a set of n random variables. For info see eg http://www.math.nus.edu.sg/ matsr/ProbI/Lecture12.pdf

Differentiable Almost-Discrete Distributions

The last section was fun, but the real reason to introduce the Gumbel distribution is to fix a problem with sampling. If we compute p_i from the x_i, and then sample from the p_i distribution, there's no way to differentiate the result with respect to x_i. Note that here we are imagining that the x_i themselves are actually the output of a NN acting on some other underlying data, so what we're really interested in doing is taking derivatives with respect to some NN parameters that x_i secretly depend on. But to do that, we have to be able to differentiate with respect to x_i to even get started (via the chain rule).

We can use the Gumbel distribution to create a *fully differentiable* discrete stochastic distribution. That is, if we sample $z_i \sim g(z_i)$ from the Gumbel and compute

$$f_i(x; z; \beta) = \frac{e^{\beta(x_i+z_i)}}{\sum_j e^{\beta(x_j+z_j)}} \tag{3.5.7}$$

then the result is a fully differentiable function of the x_i which *samples* from the p_i distribution determined by the x_i. It may seem very strange that we have re-introduced the softmax, but we've also included a parameter β because in the limit that $\beta \to \infty$, we simply recover softmax $\to$ argmax (in its one-hot version).

So you should think of f_i as a regulated version of argmax, with β a free parameter. In this form, f_i implements a 're-parameterization trick' for discrete distributions.

3.6 Aside: Measures of Distance Between Probability Distributions

The KL Divergence plays an extremely prominent role in ML because of its connection to maximum likelihood. Even more generally, the KL is useful because if we can sample data x from an underlying distribution P, then we can easily estimate the KL between P and some model q via

$$\begin{aligned} D_{KL}(P||q) &= \int dx P(x) \log\left(\frac{P(x)}{q(x)}\right) \\ &\approx \frac{1}{N}\sum_{i=1}^{N} \log\left(\frac{P(x_i)}{q(x_i)}\right) \\ &= -S(P) - \frac{1}{N}\sum_{i=1}^{N} \log\left(q(x_i)\right) \end{aligned} \tag{3.6.1}$$

where the constant entropy $S(P)$ does not depend on our model q.

My sense is that in ML we only favor the KL over other measures of distance between probability distributions because it is so easy and convenient to compute via sampling. Thus its interesting to consider other measures of distance between probability distributions p and q:

- We can modify the KL rather trivially by symmetrizing it, but the result still doesn't satisfy the triangle inequality. But we can go further and define the Jensen-Shannon divergence

$$\text{JSD}(p||q) = \frac{1}{2}D_{KL}\left(p||\frac{1}{2}(p+q)\right) + \frac{1}{2}D_{KL}\left(q||\frac{1}{2}(p+q)\right) \tag{3.6.2}$$

Apparently one can prove that the square root of the JSD satisfies the triangle inequality, which means that it's actually a metric on the space of probability distributions.

- The KL is part of a family of distance measures with the property that the local metric is the Fisher information. These are called 'F-divergences'.
- Another famous distance measure is the 'Earth Mover Distance' or the Wasserstein metric. But to define it, we must *specify a geometry* on the underlying space X that $p(X)$ is a distribution over. (This wasn't a requirement for the KL or the JSD.) The reason for the name is that we can visualize the probability distributions as 'lumps of dirt', and the EMD effectively computes the minimum amount of work we do to 'move the dirt' to turn one lump into the other. This explains the EMD's dependence on the geometry of X – moving the dirt further means doing more work, leading to a larger EMD.

Undoubtably there are problems with using the KL as a distance metric, and there have been interesting suggestions in the literature that using other metrics can lead to be better ML results.

4 Optimization

In this section we will discuss optimization – the specific process we use to 'optimize' the NN parameters in order to 'learn' to get to a minimum of our chosen loss function.

4.1 Intro to 'Optimizers'

'Optimizers' are simple algorithms for finding a minimum of the loss function.

We will often write update rules associated with the nth update in terms of gradients

$$g_i^{(n)} = \left.\frac{\partial L(X;\theta)}{\partial \theta^i}\right|_{\theta=\theta^{(n)}} = \nabla_i L(X;\theta) \tag{4.1.1}$$

The gradient is usually the primary source of information about how to improve the model and learn. We always average $g_i^{(n)}$ over a *batch* of B data points x_i sampled from the training set X, but below we won't explicitly indicate this unless its relevant. Some comments:

- As physicists, its natural to think about the continuum limit of these optimizers, where the steps become infinitesimal. But *the continuum limit is a very undesirable limit for optimization.* We want to be as computationally efficient as possible, which means that we'd like to take very large (discrete) steps. In particular, ideally *the 'information' in sequential updates should be as independent as possible*, as otherwise we're being wasteful. Concretely, this would mean that gradients & updates on subsequent steps shouldn't be very correlated. This is as far as possible from the continuum limit.
- It's worth keeping track of *dimensional analysis* – if the loss L and the NN parameters θ_i had units, what units would the other parameters in the optimizer need to have? We'll see that this provides a fair bit of insight into the behavior of various optimizers.

- The 'Stochastic' in Stochastic Gradient Descent refers to the fact that we use batches smaller than the full dataset to do optimization. We'll talk more about this later on, but it's 'stochastic' because there's randomness associated with these sub-samples of the data. Note that when we choose the batch size B, we're deciding *how much noise is in the gradient signal*. Naively, larger batches may be better because they're less noisy, but too-large batches are just a waste. And small batches could be good if noise is somehow beneficial, as some have argued.

Let's begin by briefly noting two optimizers that are rarely used, but fairly natural:

Evolution

Perhaps the most naive algorithm is to simply explore in random directions in parameter space, and keep some combination of the updates that do best. This is a good baseline to have in mind; we'd expect it to be a lot worse than the methods below, since they utilize the gradient of the loss as an indicator of how to best update the parameters.

Newton's Method

Using the Hessian of the loss H, this is the update rule for parameters

$$\theta^{(n+1)} = \theta^{(n)} - \epsilon \left[H^{(n)}\right]^{-1} g^{(n)} \tag{4.1.2}$$

where the learning rate $\epsilon \lesssim 1$ is a free parameter. In the quadratic limit with $\epsilon = 1$, we simply jump to the minimum immediately.

Note that Newton's method is *dimensionfully correct*, so ϵ is just a dimensionless number. However, in NNs H will not necessarily be a positive definite matrix, so its meaning here is unclear.

When $\epsilon < 1$ it's interesting to see what happens in the toy case of a quadratic 1-d potential. If $L(\theta) = \frac{1}{2}\lambda\theta^2$, then we have

$$\theta^{(n+1)} = \theta^{(n)} - \epsilon\frac{\lambda\theta^{(n)}}{\lambda} = (1-\epsilon)\,\theta^{(n)} \tag{4.1.3}$$

so that we converge towards the minimum exponentially fast, with a rate set by $(1-\epsilon)$. Note that in the absence of noise or higher order terms, we never overshoot the minimum if $\epsilon < 1$.

The obvious downside of this method is that if H has small eigenvalues, then some updates may be enormous. This could be dangerous if H and g_n are noisy, or if higher order terms are important.

At the moment, few researchers use second order optimization methods (especially this one; we'll see a slightly more popular choice later on) due to their high computational cost (and coding complexity) vs apparent benefits.

4.2 Stochastic Gradient Descent (SGD)

The simplest update rule would seem to be

$$\theta^{(n+1)} = \theta^{(n)} - \epsilon g^{(n)} \tag{4.2.1}$$

but there's more to this than meets the eye. Note that the learning rate ϵ has units of

$$[\epsilon] = \frac{[\theta]^2}{[loss]} \tag{4.2.2}$$

which is a bit odd. It implies that we will need different ϵ if we parameterize our network differently, or if we rescale or re-parameterize the loss function.

I mentioned Newton's method before SGD because it already suggests a solution to the 'problem' of the dimensionality of ϵ. Namely that *we really need some sort of metric on the space of gradients.* Or at the very least, we should be cognizant of the fact that we have already implicitly chosen such a metric. To see this, note that we can re-interpret the gradient via

$$\frac{-\nabla_\theta L(\theta)}{|\nabla_\theta L(\theta)|} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \arg\min_{|\delta| \leq \epsilon} L(\theta + \delta) \tag{4.2.3}$$

Here the Euclidean metric on parameter space appears twice, on both the left and right sides... but there's nothing particularly special or natural about that metric.

4.3 Momentum and Why It Helps

The momentum update keeps a running velocity with update rule

$$v^{(n+1)} = m v^{(n)} + g^{(n)} \tag{4.3.1}$$

where $m \lesssim 1$ is the 'momentum'; a typical value is $m = 0.9$. Then the parameter update is

$$\theta^{(n+1)} = \theta^{(n)} - \epsilon v^{(n)} \tag{4.3.2}$$

So we're updating the parameters with a smeared combination of the current gradient and, roughly speaking, the last $\frac{1}{1-m}$ gradients. Dimensional analysis here is the same as with SGD, and m is a pure (dimensionless) number.

Why[8] is this useful? Let's contrast it with an approach that might seem equally good – we could just increase the batch size by a factor of $\frac{1}{1-m}$, and also increase the learning rate[9] by the same factor. What's better about momentum as compared to this approach? Naively it might seem that momentum should be strictly worse, since it's updating the parameters using old, and potentially 'stale' information about the loss landscape.

At least in very simplified cases, such as convex optimization (NN learning landscapes are not convex), momentum actually helps a lot in the important and challenging case where the optimization problem is *ill-conditioned, meaning that there's a large range of Hessian eigenvalues.* The intuition here is simple – by averaging over a sequence of updates we can smooth out our evolution through a 'canyon' in the loss function.

[8] For a beautiful and extensive discussion, see: https://distill.pub/2017/momentum/

[9] To be clear, I'm assuming gradients are averaged within a batch, not summed.

In a quadratic potential $\frac{1}{2}\lambda\theta^2$, the gradient is

$$g = \lambda\theta \tag{4.3.3}$$

and so pure SGD updates are

$$\theta^{(n+1)} = \theta^{(n)} - \epsilon\lambda\theta^{(n)} = (1-\epsilon\lambda)\theta^{(n)} \tag{4.3.4}$$

This has three phases, involving exponential divergence, and same-sign or alternating exponential convergence. The range of ϵ where we converge is

$$0 < \epsilon < \frac{2}{\lambda} \tag{4.3.5}$$

This means that if we have a high-dimensional problem with a range of different λ – what we're really interested in – then our ϵ will be limited by the largest λ. That's why SGD doesn't work so well for ill-conditioned problems.

If we instead use momentum, we have a pair of update rules

$$\begin{aligned} \theta^{(n+1)} &= \theta^{(n)} - \epsilon v^{(n+1)} = (1-\lambda\epsilon)\theta^{(n)} - \epsilon m v^{(n)} \\ v^{(n+1)} &= m v^{(n)} + \lambda\theta^{(n)} \end{aligned} \tag{4.3.6}$$

We can solve this by viewing the update as a 2×2 matrix multiplication by

$$\begin{pmatrix} 1-\lambda\epsilon & -m\epsilon \\ \lambda & m \end{pmatrix} \tag{4.3.7}$$

acting on a (θ, v) vector. This matrix has eigenvalues

$$e_\pm = \frac{1}{2}\left(1 + m - \lambda\epsilon \pm \sqrt{(m-\lambda\epsilon+1)^2 - 4m}\right) \tag{4.3.8}$$

and we need to have $-1 < e_\pm < 1$ for stable convergence. This leads to the bounds

$$0 < \epsilon < \frac{2}{\lambda}(1+m) \tag{4.3.9}$$

where the case $m = 0$ is conventional SGD. So momentum naively allows us to increase the step size by a factor of 2 without losing convergence. But this improvement is not meaningful, since the units and scale of ϵ were arbitrary. For example, we could have re-defined our parameters via $\epsilon \to m\epsilon/(1+m)$ and we wouldn't reap any benefit. However, the convergence rate(s) are more meaningful.

There was a single convergence rate of $(1-\epsilon\lambda)$ in SGD. Here we have two convergence rates $e_\pm$, and we'll be limited by the slower rate. Thus we can obtain critical (best) convergence by taking $e_+ = e_-$, in which case

$$m = \left(1 - \sqrt{\epsilon\lambda}\right)^2 \tag{4.3.10}$$

and we have an exponential convergence rate of

$$e_\pm = 1 - \sqrt{\epsilon\lambda} \tag{4.3.11}$$

Note that this is actually meaningfully better than SGD because *the dependence on λ has been softened*, so that the convergence rate depends more weakly on the Hessian eigenvalues λ with momentum as compared to SGD. That's the advantage that momentum can buy us.

But to really demonstrate this effect, we must imagine that we have at least two dimensions with $\lambda_1 \ll \lambda_2$, and that we're simultaneously optimizing in both. In this case we want to minimize $e_\pm$ for both directions. In fact this occurs when $e_+ = e_-$ for both directions, which simply requires that both discriminants vanish, so that

$$(m - \lambda_1\epsilon + 1)^2 - 4m = 0, \quad (m - \lambda_2\epsilon + 1)^2 - 4m = 0 \tag{4.3.12}$$

We can straightforwardly solve these equations for ϵ and m, giving

$$\epsilon = \left(\frac{2}{\sqrt{\lambda_1} + \sqrt{\lambda_2}}\right)^2, \quad m = \left(\frac{\sqrt{\lambda_1} - \sqrt{\lambda_2}}{\sqrt{\lambda_1} + \sqrt{\lambda_2}}\right)^2 \tag{4.3.13}$$

Substituting these values and identifying the largest eigenvalue, we find a convergence rate

$$\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \tag{4.3.14}$$

where $\kappa = \lambda_1/\lambda_2 > 1$ is the condition number. In contrast, with gradient descent the convergence rate is $\frac{\kappa-1}{\kappa+1}$, so with momentum, for purposes of convergence we have effectively taken the square root of the condition number!

Given this analysis, it's natural to wonder if by using a 3-stage update rule, where we maintain both a 'momentum' and an 'acceleration', we could do even better, and eg take the cube root of the condition number. This seems like an amusing toy question. Note that even if we can manage this improvement, it's much less of a gain as compared to the transition from SGD to momentum. From this point of view we can interpret Newton's method as having taken the 'infinite root' of the condition number, so that it's always 1 in the quadratic approximation of the loss.

4.4 RMSProp and Adam

The idea of RMSProp and Adam is to improve conditioning by suppressing updates in directions that have already been updated a lot, and enhancing updates in directions that have not changed as much. Intuitively, this is a good idea because highly-updated components tend to correspond to oscillations where we are jumping back and forth across a steep 'canyon' in the loss. RMSProp implements this idea, and Adam essentially just adds momentum to it. Currently Adam is probably the most commonly used optimizer in ML.

In both algorithms we accumulate a total gradient variance during training. That is we define a vector of variances in parameter space

$$V_i^{(n+1)} = \beta V_i^{(n)} + (1 - \beta)\left(g_i^{(n)}\right)^2 \tag{4.4.1}$$

for some $\beta \lesssim 1$, which acts as a 'momentum parameter' for the variances. Note that we are accumulating the squares of the *individual components*, not of the vector norm. This is biased towards 0 early on, so we can define

$$\hat{V}_i^{(n+1)} = \frac{V_i^{(n+1)}}{1-\beta^n} \tag{4.4.2}$$

to correct for this problem. (We didn't bother correcting this bias in the case of momentum.) Note that this works because

$$\begin{aligned} \hat{V}_i^{(n+1)} &= \frac{\left(g_i^{(n)}\right)^2}{\sum_{m=0}^{n-1}\beta^m} + \beta V_i^{(n)} \\ &\sim \max\left(\frac{1}{n}, 1-\beta\right)\left(g_i^{(n)}\right)^2 + \beta V_i^{(n)} \end{aligned} \tag{4.4.3}$$

Then we define the update

$$\theta_i^{(n+1)} = \theta_i^{(n)} - \epsilon \frac{g_i^{(n)}}{\sqrt{\hat{V}_i^{(n)}}} \tag{4.4.4}$$

where g_n^i is the gradient. In fact there's an extra parameter in the denominator to make sure it doesn't get too small; usually this is set to 10^{-8} or so. Officially, the bias correction wasn't included in RMSProp, which simply came from a slide at a talk, but it is unambiguously included in Adam.

So dimensionally, in Adam and RMSProp the units are such that

$$[\epsilon] = [\theta] \tag{4.4.5}$$

as the loss function and parameter scale cancel out of the ratio in RMSProp. This is quite different from SGD and Momentum. You might wonder why the denominator has a square root in it. As far as I can tell, the power-law in the denominator was set for good empirical performance, and I'm not aware of a good theoretical justification.

What is this doing? Clearly it is discouraging updates to parameter directions where there have been, historically, a lot of updates, and encouraging updates in small-parameter-update directions. This suggests that it's helping to fix the conditioning of the loss landscape. But is it really?

Let's imagine a toy quadratic model with a Hessian H. If H is diagonal in the basis of the parameters, so that literally H_{ij} is a diagonal matrix, then the components will all evolve independently. If $H_{ii} = \lambda_i$, then notice that since RMSProp is dimensionless, it will (eventually!) remove all dependence on the λ_i. So in this simple toy scenario, if β isn't too close to 1, RMSProp has the approximate effect of turning

$$L(\Theta_i) = \sum_i \lambda_i \Theta_i^2 \text{ with RMSProp} \quad \rightsquigarrow \quad L(\Theta_i) = \sum_i |\Theta_i| \text{ with SGD} \tag{4.4.6}$$

as optimization problems, since the updates will be of constant magnitude, set by ϵ. In practice the parameter β is usually set very close to one, eg $\beta = 0.999$ is the default, so this behavior would take a long time to set in. So in fact I would not necessarily expect this picture to provide a good guide to the true behavior of RMSProp or Adam with default parameters.

Note that these algorithms help with conditioning in a different way from Newton's method. SGD with $L = |\theta|$ converges *linearly* to within the learning rate ϵ of the minimum in $|\theta|/\epsilon$ steps, after which it oscillates. Newton's method instead converges *exponentially*, and doesn't oscillate.

It would seem that the benefits of RMSProp depend on alignment of the Hessian and the parameters. But note that if H is block diagonal, then RMS prop will still help with stabilizing the learning rate between different blocks. This can easily occur if different layers in a NN have different scales for their parameters.

What about Adam? It's just RMSProp with momentum for the gradients in the numerator. It's not clear if adding momentum to RMSProp provides the same benefits as adding momentum to SGD, but it might be interesting to study. In any case, Adam and Momentum are the two predominant optimizers in ML research at the moment, with Adam serving as the more flexible and stable choice.

4.5 Batch Size Selection and the Gradient Noise Scale

When optimizing, we need to choose a batch size B. How should we do that? The key idea is that B determines how noisy the gradients are, and so we want to choose it in such a way that the gradients do not have more noise than signal. The reason to choose a larger B (if possible) is to have *data parallelism*, so that we can parallelize the optimization process as much as possible, and take as few optimization steps as possible while doing a fixed amount of total computation. That way, optimization happens faster at no additional cost.

We would like to minimize $L(\theta)$ using an SGD-like optimizer, so the relevant quantity is the gradient $G(\theta) = \nabla L(\theta)$. However, optimizing $L(\theta)$ directly would be wasteful if not impossible, since it would require processing the entire data distribution every optimization step. Instead, we obtain an estimate of the gradient by averaging over a collection of samples from ρ, called a batch:

$$G_{\text{est}}(\theta) = \frac{1}{B}\sum_{i=1}^{B} \nabla_\theta L_{x_i}(\theta); \qquad x_i \sim \rho \tag{4.5.1}$$

This approximation forms the basis for stochastic optimization methods such as mini-batch stochastic gradient descent (SGD) and Adam. The gradient is now a random variable whose expected value (averaged over random batches) is given by the true gradient. Its variance scales inversely with the batch size B[10]:

$$\begin{aligned} \mathbb{E}_{x_{1\cdots B}\sim\rho}\left[G_{\text{est}}(\theta)\right] &= G(\theta) \\ \text{cov}_{x_{1\cdots B}\sim\rho}\left(G_{\text{est}}(\theta)\right) &= \frac{1}{B}\Sigma(\theta), \end{aligned} \tag{4.5.2}$$

[10]This is strictly true only when training examples are sampled independently from the same data distribution. For example, when batches are sampled without replacement from a dataset of size D, the variance instead scales like $\left(\frac{1}{B} - \frac{1}{D}\right)$. For simplicity, we restrict ourself to the case where $B \ll D$ or where batches are sampled *with* replacement, but our conclusions can be altered straightforwardly to account for correlated samples.

where the per-example covariance matrix is defined by

$$\begin{aligned}\Sigma(\theta) &\equiv \mathrm{cov}_{x\sim\rho}\left(\nabla_\theta L_x(\theta)\right) \\ &= \mathbb{E}_{x\sim\rho}\left[\left(\nabla_\theta L_x(\theta)\right)\left(\nabla_\theta L_x(\theta)\right)^T\right] - G(\theta)\,G(\theta)^T.\end{aligned} \tag{4.5.3}$$

The key point here is that the minibatch gradient gives a noisy estimate of the true gradient, and that larger batches give higher quality estimates. We are interested in how useful the gradient is for optimization purposes as a function of B, and how that might guide us in choosing a good B. We can do this by connecting the noise in the gradient to the maximum improvement in true loss that we can expect from a single gradient update. To start, let G denote the true gradient and H the true Hessian at parameter values θ. If we perturb the parameters θ by some vector V to $\theta - \epsilon V$, where ϵ is the step size, we can expand true loss at this new point to quadratic order in ϵ:

$$L(\theta - \epsilon V) \approx L(\theta) - \epsilon G^T V + \frac{1}{2}\epsilon^2 V^T H V. \tag{4.5.4}$$

If we had access to the noiseless true gradient G and used it to perturb the parameters, then Equation 4.5.4 with $V = G$ would be minimized by setting $\epsilon = \epsilon_{\mathrm{max}} \equiv \frac{|G|^2}{G^T H G}$. However, in reality we have access only to the noisy estimated gradient G_{est} from a batch of size B, thus the best we can do is minimize the expectation $\mathbb{E}[L(\theta - \epsilon G_{\mathrm{est}})]$ with respect to ϵ. This expected value can be evaluated using Equation 4.5.2:

$$\mathbb{E}[L(\theta - \epsilon G_{\mathrm{est}})] = L(\theta) - \epsilon|G|^2 + \frac{1}{2}\epsilon^2\left(G^T H G + \frac{\mathrm{tr}(H\Sigma)}{B}\right). \tag{4.5.5}$$

Minimizing this equation with respect to ϵ leads to:

$$\epsilon_{\mathrm{opt}}(B) = \mathrm{argmin}_\epsilon \mathbb{E}\left[L(\theta - \epsilon G_{\mathrm{est}})\right] = \frac{\epsilon_{\mathrm{max}}}{1 + \mathcal{B}_{\mathrm{noise}}/B} \tag{4.5.6}$$

as the optimal step size, which produces an optimal improvement in the loss from the noisy gradient:

$$\Delta L_{\mathrm{opt}}(B) = \frac{\Delta L_{\mathrm{max}}}{1 + \mathcal{B}_{\mathrm{noise}}/B}; \qquad \Delta L_{\mathrm{max}} = \frac{1}{2}\frac{|G|^4}{G^T H G}. \tag{4.5.7}$$

Above, we have defined the *noise scale* as:

$$\mathcal{B}_{\mathrm{noise}} = \frac{\mathrm{tr}(H\Sigma)}{G^T H G}, \tag{4.5.8}$$

Note that our definition of the noise scale is independent of the size of the full training set. If we use a step size larger than twice ϵ_{opt}, the loss may *increase*, leading to divergence.

We should emphasize that this analysis is best viewed as a *constraint on the stability* of optimization, rather than as a way to choose the optimal step size. In fact, choosing the step size via a line search leads to very poor optimization performance.

Implications and Simplifications

Equation 4.5.7 implies that when the batch size is much smaller than the noise scale, $B \ll \mathcal{B}_{\text{noise}}$, the second term in the denominator dominates the first, so increasing the batch size B linearly increases the progress in loss. This is the small batch regime, where increases in batch size linearly speed up training. By contrast, when $B \gg \mathcal{B}_{\text{noise}}$, then the first term dominates, so that increasing B has almost no effect on the progress in loss. This is the large batch regime where increases in batch size do not speed up training and simply waste computation; the switch between the two occurs at $B \approx \mathcal{B}_{\text{noise}}$.

The situation gets even simpler if we make the (unrealistic) assumption that the optimization is perfectly well-conditioned – that the Hessian is a multiple of the identity matrix. If that is the case, then Equation 4.5.8 reduces to:

$$\mathcal{B}_{\text{simple}} = \frac{\text{tr}(\Sigma)}{|G|^2}, \tag{4.5.9}$$

which says that the noise scale is equal to the sum of the variances of the individual gradient components, divided by the global norm of the gradient – essentially a measure of how large the gradient is compared to its variance. It is also a measure of the scale at which the estimated and true gradient become close in L^2 space (having non-trivial dot product) – the expected normalized L^2 distance is given by:

$$\frac{\mathbb{E}\left[|G_{\text{est}} - G|^2\right]}{|G|^2} = \frac{1}{B}\frac{\text{tr}(\Sigma)}{|G|^2} = \frac{\mathcal{B}_{\text{simple}}}{B}, \tag{4.5.10}$$

In practice, we find that $\mathcal{B}_{\text{simple}}$ and $\mathcal{B}_{\text{noise}}$ typically differ only by a small constant multiplicative factor, particularly when we employ common training schemes that improve conditioning.

Predictions for Data/Time Efficiency Tradeoffs

Thus far our analysis has only involved a single point in the loss landscape. But equation 4.5.7 nevertheless predicts the dependence of training speed on batch size remarkably well, even for full training runs that range over many points in the loss landscape. By averaging Equation 4.5.7 over multiple optimization steps, we find a simple relationship between training speed and data efficiency:

$$\frac{S}{S_{\text{min}}} - 1 = \left(\frac{E}{E_{\text{min}}} - 1\right)^{-1}. \tag{4.5.11}$$

Here, S and S_{min} represent the actual and minimum possible number of steps taken to reach a specified level of performance, respectively, and E and E_{min} represent the actual and minimum possible number of training examples processed to reach that same level of performance. Since we are training at fixed batch size, we have $E_{\text{tot}} = BS_{\text{tot}}$. We define the *critical batch size* by an empirical fit to the above equation, as

$$\mathcal{B}_{\text{crit}} = \frac{E_{\text{min}}}{S_{\text{min}}}. \tag{4.5.12}$$

Our model predicts $\mathcal{B}_{\text{crit}} \approx \mathcal{B}_{\text{noise}}$, where $\mathcal{B}_{\text{noise}}$ is appropriately averaged over training. Note that the noise scale can vary significantly over the course of a training run, so the critical batch size also depends on the level of performance to which we train the model.

The resulting tradeoff curve in serial time vs total compute has a hyperbolic shape. The goal of optimization is to reach a given level of performance with minimal S and E – but there are tradeoffs involved, as very small S may require very large E, and vice versa. When we choose $B = \mathcal{B}_{\text{crit}}$, the two sides of Equation 4.5.11 are both 1, so that training takes twice as many passes through the training data as an optimally data-efficient (small-batch) run would take, and twice as many optimization steps as an optimally time-efficient (large-batch) run would take.

4.6 Natural Gradients

We have encountered two significant esthetic issues with optimization:

- The learning rate ϵ is dimensionless in Newton's method, but not in the case of 1st order gradient descent algorithms. This means that we have to know something about the scale of the loss and of the parameters to choose ϵ appropriately.
- Gradient descent implicitly involves a choice of metric on the space of parameters, but the choice of an L^2 metric seems to be arbitrary, and would not be invariant under a re-parameterization of the NN parameters.

Let's illustrate the second point explicitly. Then we will see how these problems can be 'naturally' resolved. (The best reference I've found on this subject is https://arxiv.org/abs/1412.1193, and I recommend it as a first resource if you'd like to learn more.)

Explicit Illustration of Coordinate Dependence

Consider what would happen if we re-parameterize our network in terms of some new parameters Φ via general functions

$$\theta_i = f_i(\Phi^\mu) \tag{4.6.1}$$

A simple SGD update on Φ would be

$$\Phi_\mu^{(n+1)} = \Phi_\mu^{(n)} - \epsilon \nabla_i L \left(f^i(\Phi^\mu)\right) \left(\nabla_\mu f^i\right) \tag{4.6.2}$$

This means that if we perform gradient descent on Φ, the induced update of Θ will become

$$\begin{aligned} \theta_i^{(n+1)} &= \theta_i^{(n)} - \epsilon \frac{\partial f_i}{\partial \Phi^\mu} \frac{\partial L}{\partial \theta^j} \frac{\partial f^j}{\partial \Phi^\mu} \\ &= \theta_i^{(n)} - \epsilon \left(\frac{\partial f_i}{\partial \Phi^\mu} \frac{\partial f^j}{\partial \Phi^\mu} \right) \frac{\partial L}{\partial \theta^j} \\ &= \theta_i^{(n)} - \epsilon \left(\nabla_\mu f_i \nabla^\mu f^j \right) \nabla_j L \end{aligned} \tag{4.6.3}$$

This is a different update rule! We found a new update because in effect, the inverse metric on the parameters has changed from

$$\epsilon \delta^{ij} \to \epsilon \delta^{\mu\nu} \left(\frac{\partial f^i}{\partial \Phi^\mu} \frac{\partial f^j}{\partial \Phi^\nu} \right) \tag{4.6.4}$$

This is what we would expect as a consequence of the diffeomorphism f. Because the SGD update rule does not have any natural geometric properties, it is not invariant to changes of coordinates on 'model space'. But our choice of coordinates was essentially arbitrary, so this should be disturbing!

The choice of SGD assumes that the metric on NN parameters is simply L^2, ie $\delta\theta^2$. But given any other inverse metric M we could perform an update

$$\theta_i^{(n+1)} = \theta_i^{(n)} - \epsilon M_i^j g_j^{(n)} \tag{4.6.5}$$

where SGD corresponds to $M^{ij} = \delta^{ij}$, and Newton's method using $M = H^{-1}$ (which isn't a metric, and perhaps doesn't exist). Let's formalize these ideas.

Formalization of Metric Dependence

Our NN model itself (or any model) is some function $p_\theta : X \to Y$. The loss is a function

$$L(X, Y) = L[X, p_\theta(X)] \tag{4.6.6}$$

In fact it's a functional when written in this way, since it maps the space of functions p_θ to numbers. But it's just a function if we parameterize our NN with a finite set of parameters θ_i. As usual, we estimate it by sampling from X.

Additionally, we may choose a distance metric

$$D[p_\theta, p_\Phi] \geq 0 \tag{4.6.7}$$

that measures how different two models are. We expect that $D[p_\theta, p_\theta] = 0$ and in a perfect world, D would also satisfy the triangle inequality

$$D[p, q] + D[q, t] \geq D[p, t] \tag{4.6.8}$$

and be symmetric. If so it gives us a metric on the space of models. When we tried to define SGD, we were implicitly picking the metric

$$D_{L^2}[p_\theta, p_\Phi] = \sqrt{\sum_i (\theta^i - \Phi^i)^2} \tag{4.6.9}$$

for NN models with the same set of parameters. This is definitely a metric, but there's nothing particularly natural about it, other than its simplicity.

Now we can use our metric to formalize the extraction of an update rule as

$$\delta\theta = \underset{D[p_\theta, p_{\theta+\delta\theta}] \leq \epsilon}{\arg\min} L(\theta + \delta\theta) \tag{4.6.10}$$

where we assume ϵ to be infinitesimal, so that this really means

$$\delta\theta = \underset{\delta\theta^T M \delta\theta \,=\, \epsilon}{\arg\min} g^T \delta\theta \tag{4.6.11}$$

for a local metric M. It's very easy to see by studying a Lagrangian

$$L = \lambda(\delta\theta^T M \delta\theta - \epsilon) + g^T \delta\theta \tag{4.6.12}$$

where λ is a Lagrange multiplier, that this leads to an update

$$\delta\theta \propto -M^{-1} g \tag{4.6.13}$$

The Newton's method update would arise with $M = H$, though recall that H isn't a metric.

Natural Gradients from a Natural Metric

Our analysis shows that gradient descent necessarily involves a choice of metric, and that conventional SGD presumes a Euclidean metric on the NN parameters (in the whatever parameterization is chosen). But if we had a natural metric on model space, then we could perform gradient descent in a more natural basis.

We have already discussed various metrics on model space, with our largest focus on the KL divergence

$$D_{KL}(p||q) = \int dx\, p(x) \log\left(\frac{p(x)}{q(x)}\right) \tag{4.6.14}$$

As we've discussed, this is *not a metric*, but it does measure a kind of distance or separation on model space. Furthermore, infinitesimally it is a metric, that is

$$\begin{aligned} D_{KL}(p_{\theta+\delta\theta}||p_\theta) &= \int dx\, p_{\theta+\delta\theta}(x) \log\left(\frac{p_{\theta+\delta\theta}(x)}{p_\theta(x)}\right) \\ &\approx \frac{1}{2}\delta\theta^i \delta\theta^j F_{ij}(\theta) \end{aligned} \tag{4.6.15}$$

where F_{ij} is the positive-definite and symmetric Fisher information matrix, and we recall

$$F_{ij}(\theta) = \int dx\, p_\theta(x) \left(\nabla_i \log p_\theta(x) \nabla_j \log p_\theta(x)\right) \tag{4.6.16}$$

concretely in terms of our model $p_\theta(x)$. This can be evaluated approximately by sampling x from the data distributions. Note that F is manifestly positive semi-definite.

The natural gradient method uses F as a metric, giving an update

$$\delta\theta = -\epsilon F^{-1} g \tag{4.6.17}$$

with a dimensionless learning rate ϵ.

One reason for interest in the natural gradient is that F makes no reference to the parameter space or its representation. Thus it will be coordinate re-parameterization invariant. A second reason to like the natural gradient update is that, in fact F has a close relationship to a certain approximation of the Hessian, the Gauss-Newton matrix. This means that the natural gradient update can often be viewed as an approximation to Newton's method.

If you're interested in exploring natural gradients for NN optimization, you'll want to read about 'KFAC', an approximate implementation that, with some work, can be used in tensorflow.

(Watch Out for) the Empirical Fisher

You might be tempted to compute a matrix that looks like the Fisher via sampling from the empirical data distribution, ie we can write a pseudo-Fisher or 'empirical Fisher' matrix as

$$\tilde{F} = \int dx P(x) \left(\nabla_i \log p_\theta(x) \nabla_j \log p_\theta(x)\right) \tag{4.6.18}$$

where $P(x)$ is the ground truth distribution for the data. Note that this is not the same thing as the Fisher, because the data is being sampled from the ground truth distribution P rather than the model p_θ. This will be positive definite, but it's not the correct matrix to use for natural gradients.

4.7 Aside: More on 2nd Order Methods

In the wider world of (convex) numerical optimization, second order methods are quite popular. These can mostly be viewed as more efficient versions of Newton's method. Let's take a look at how they work. In all cases we'll think about expanding the loss as

$$L = L_0 + \theta_i g^i + \frac{1}{2}\theta_i \theta_j H^{ij} + \cdots \tag{4.7.1}$$

(where we WLOG set the initial point to $\theta = 0$) and work with the Hessian and gradients. To this order, if we assume that H is positive definite, then we are searching for a solution to

$$H\theta = g \tag{4.7.2}$$

When H is not positive definite, as is typical, our goal is less clear.

4.7.1 Gauss-Newton Matrix

We ran into the problem that H isn't positive definite. One way to deal with this for certain types of loss functions is to use the Gauss-Newton matrix in place of the Hessian. Let's assume that our loss function is

$$L = \left\langle (y - f_\theta(x))^2 \right\rangle \tag{4.7.3}$$

where the expectation is over (x, y) in the dataset. In this case the Hessian with respect to θ will contain two terms

$$H_{\alpha\beta} = \langle \nabla_\alpha f_\theta \nabla_\beta f_\theta \rangle + \langle (y_i - f_\theta(x_i)) \nabla_\alpha \nabla_\beta f_\theta \rangle \tag{4.7.4}$$

The first term is the Gauss-Newton matrix, which is guaranteed to be positive semi-definite. So if we like, we can use it in place of the full Hessian. Hopefully this will also provide a more conservative approach, since in theory it's less likely to have nearly vanishing eigenvalues.

4.7.2 Conjugate Gradients

If H is positive definite, we can view it as a local metric on the parameters. In this sense we can imagine having a list of vectors v_i that are mutually orthogonal with respect to H. Then the problem we want to solve can be decomposed so that

$$\theta = \alpha_i v_i \tag{4.7.5}$$

and we need only solve

$$\alpha_i H v_i = g \tag{4.7.6}$$

which we can do by dotting into v_i so that

$$\alpha_i = \frac{v_i^\dagger g}{v_i^\dagger H v_i} \tag{4.7.7}$$

which determines the α_i. This is all rather trivial; we're just working in a basis set by the v_i.

We can turn this into an algorithm by choosing to move in conjugate directions. So to start out we simply take a gradient step to the new point

$$\theta_1 = 0 - \delta_0 = -g \tag{4.7.8}$$

since $\theta_0 = 0$. On the next step, naive gradient descent suggests moving by

$$-g - H(-g) = Hg - g \tag{4.7.9}$$

However, instead we will insist that this step is orthogonal to the original step in the H metric. For this its useful to define the kth residual

$$r_k = -g - A\theta_k \tag{4.7.10}$$

and then the kth step is

$$\delta_k = r_k - \sum_{i<k} \frac{\delta_i^\dagger H r_k}{\delta_i^\dagger H \delta_i} \delta_i \tag{4.7.11}$$

which is simply the Graham-Schmidt orthogonalization of the sequence $(r_0, r_1, \cdots, r_k)$ in the metric H. The update rule is

$$\theta_{k+1} = \theta_k + \alpha_k \delta_k \tag{4.7.12}$$

where

$$\alpha_k = \frac{\delta_k^\dagger r_k}{\delta_k^\dagger H \delta_k} \tag{4.7.13}$$

In the limit where g is a sum of a few eigenvectors of H, we are just taking a few steps.

4.7.3 (L-)BFGS (Broyden–Fletcher–Goldfarb–Shanno)

This is an algorithm to approximate Newton's method without directly computing the Hessian.

The basic idea is to approximate the Hessian using information gained from successive parameter values and gradients. Notice that

$$\nabla_i L_{k+1} \approx \nabla_i L_k + (\theta_{k+1} - \theta_k)^j H_{ij} \tag{4.7.14}$$

where this is only approximate because we have expanded in small $|\theta_{k+1} - \theta_k|$. Given the parameter values and gradients, we can re-interpret this equation as an approximate formula for the Hessian itself via

$$(\theta_{k+1} - \theta_k)^j H_{ij}^{(k+1)} = \nabla_i L_{k+1} - \nabla_i L_k \tag{4.7.15}$$

Of course this does not even come close to determining the full H. Rather, we can try to build up such a formula via successive updates

$$H^{(k+1)} = H^{(k)} + \delta H^{(k)} \tag{4.7.16}$$

We only learn a little about H on each update.

To simplify notation, we will write

$$\begin{aligned} d_k &= \theta_{k+1} - \theta_k \\ y_k &= \nabla L(\theta_{k+1}) - \nabla L(\theta_k) \end{aligned} \tag{4.7.17}$$

To parameterize the information we obtain with each update, we will choose a simple basis for δH using

$$\delta H = \alpha u u^T + \beta v v^T \tag{4.7.18}$$

with $u = y_k$ and $v = B_k d_k$. This at least accords with dimensional analysis.

Now we can just determine α, β. Our defining equation implies that

$$d_k \cdot (H_k + \alpha y_k y_k^T + \beta (H_k d_k)(H_k d_k)^T) = y_k \tag{4.7.19}$$

which can be solved via

$$\begin{aligned} \alpha &= \frac{1}{y_k^T d_k} \\ \beta &= -\frac{1}{d_k^T H_k d_k} \end{aligned} \tag{4.7.20}$$

so that

$$H_{k+1} = H_k + \frac{y_k y_k^T}{y_k^T d_k} - \frac{(H_k d_k)(d_k H_k)^T}{d_k^T H_k d_k} \tag{4.7.21}$$

Now we can use H_{k+1} to compute an update via Newton's method. For that purpose, it's useful to note that B_{k+1} can be inverted in terms of B_k^{-1} via

$$(H_{k+1})^{-1} = \left(1 - \frac{d_k y_k^T}{d_k \cdot y_k}\right) H_k^{-1} \left(1 - \frac{y_k d_k^T}{d_k \cdot y_k}\right) + \frac{d_k d_k^T}{d_k \cdot y_k} \tag{4.7.22}$$

This can be computed efficiently without storing extra matrices by directly multiplying it out and expressing it in terms of scalars like $y_k^T B_k^{-1} y_k$.

5 Architectures

It's finally time to talk about the Neural Networks themselves!

In this section we will discuss aspects of NN design that refine our elementary picture in terms of affine transformation plus simple non-linearities. Here are some considerations that we should expect to be important in NN design:

- Can the function that we're interested in learning actually be represented in the form of this type of NN? This is necessary, but likely not sufficient. In most cases it's also not a problem that we directly measure or address.

- Will (the relevant) information be able to propagate through the network with high probability, so that training can provide a good signal for learning? At the most basic level, we need to make sure that signals do not vanish or explode, so that the loss landscape is relatively smooth. This relates to both literal architecture of the NN and the magnitude and distribution of data and parameters. So to properly understand it we will need to think about how to initialize the parameters. We'll also need to account for the behavior of the non-linearities, ie the 'activation functions', and think about how to 'whiten' the information at all stages of the computation.

- Does the most relevant information propagate in a small number of steps? As a concrete example, a recurrent network that's learning to model language has to 'remember' all of the words in a sentence to understand context, and this means that to read W words intelligently it needs to execute W sequential steps. Networks that can correlate words early and late in a paragraph without tens or hundreds of steps might be expected to perform better, and do.

- Does the network have the right symmetries for the data, or are we wasting capacity by forcing the network to learn this information? Architecture imposes *inductive bias* on the problem – that is, we are biasing the distribution of all possible NN parameters by presuming that there are fixed relations between them. We can view this as a kind of prior on the NN weights.

- We can view the NN as a function that's attempting to linearize certain non-linear features, and the architecture determines what sort of features this can naturally apply to.

- Perhaps related to all of the above... can the network efficiently learn to forget irrelevant information? Of course there may be tradeoffs here with information propagation.

- Finally, it's possible that in the future architectures will be optimized for better scaling, for example by exhibiting *model parallelism*. This could mean that the best architectures at very large scales are only optimal because they can be run on a host of GPUs.

In this section we will consider some of these questions, and what they suggest about NN architecture. My discussion of RNNs, LSTMs, and CNNs is rather brief compared to most other sources, since these topics are quite simple and have been covered extensively elsewhere.

5.1 Info Propagation – Activations, Whitening, Initialization, Resnets

Let's first discuss aspects of NN design related to encouraging and channeling the propagation of information. This includes activation functions, various 'norms' that whiten the activations, and parameter initialization. Typically these three topics are discussed separately, but I've combined them as I view them as component tools for pursuing a unified goal – to represent a rich class of functions while maintaining a smooth loss landscape.

5.1.1 Explicit Activation Functions

Perhaps the most basic feature of NNs are their activation functions – what non-linearities should we choose, and when and why?

Relatively little seems to be known about this subject. My impression is that this is because aside from a few very basic features, activation functions really don't matter very much. This would follow from the theory that NNs are just very general function approximators, and so as long the modular pieces from which they're constructed aren't pathological, their precise form isn't so important. But it is crucial to avoid certain pathologies that prevent information propagation.

My sense is that you can succeed in almost all tasks using only these activation functions:

- ReLU $= \max(0, x)$ – this is the default activation function, to be used if there isn't a reason to use one of the others. It's important that it *does not saturate* on at least one side, so that the network continues to receive gradient signals. This function may also be especially useful for the (very common) tasks where we want to distill a lot of information down to a low-dimensional decision – eg policies in RL, and classification in SL – since it's projecting the data onto one side of a hyperplane.

- Trivial/Identity – if you just want a linear transformation. For example, this is relevant for embedding spaces that map words to vectors.

- Tanh – if you want outputs in a fixed range. But note that this saturates on both sides, so it's not a great choice for intermediate layers.

- Sigmoid – if you want outputs that have a probabilistic interpretation. To be clear, this is the function that maps a vector x_i to

$$\sigma_i(x) = \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i - x_{\max}}}{\sum_j e^{x_j - x_{\max}}} \tag{5.1.1}$$

  where the second expression has better numerical stability when the components of x are large.

Given the issue with saturation, you might expect that a 'leaky ReLU' or some function like $qx + (1-q)\text{ReLU}(x)$ for some q would be a good choice. It's sometimes used when it's important for information to continue to propagate through a large number of layers. And a very closely related idea is that of 'ResNets', or 'Residual Networks', which we'll discuss below. But surprisingly, 'leaky ReLU' functions are not especially common, and don't seem to help much with training.

Some experts say that in certain situations, modified or smoothed versions of the ReLU can perform a bit better, but I don't know of a simple and empirically-supported reason why they're better. Two examples of these functions are the 'swish' and the 'GeLU'.

It's often noted that a network with L layers can encode $\sim N^L$ functions, where N is itself some large number. With ReLU's this is obvious, since we can think of each ReLU activation as a hyperplane filter, and composing L of these can slice a high-dimensional space into an exponentially growing number of cells. It's certainly important that NN are highly expressive. However, I don't think that this result (the exponential growth specifically) has all that much to do with the success of NNs, because most of these functions are probably extremely difficult or impossible to learn via optimization. Instead I'd expect NN performance has a lot more to do with the very special kind of data provided by the world, which has a great deal of correlation, structure, and simplicity. But maybe I'm wrong!

5.1.2 Whitening Functions: Weight Norm, Layer Norm, Batch Norm

One way to encourage information to flow through a network[11] is to make sure that at each stage of the computation, the distribution of the NN activations follows a 'reasonable' distribution, with order one mean and standard deviation. If we like, we can simply amend our activation functions to do this! That's the idea behind Layer Norm and Batch norm. When we use these methods we are 'whitening' our data, our model, or combinations thereof.

The simplest possibility is that we can whiten our model, without worrying about the data itself. So we can replace the weights of any given layer be

$$W_{ij} = g\frac{v_{ij}}{\sqrt{\frac{1}{N}\sum_{ab} v_{ab}}} \tag{5.1.2}$$

where g sets the overall scale of the weights, and is optimized separately. This is called weight normalization. Apparently it helps with optimization a bit, though it's less common than Layer Norm and Batch Norm. I only mention it as an appetizer.

A more interesting and dynamic approach is to whiten the actual activations. We can only do this if we have a large sample of data, and so there are two natural directions – we can use all of the activations in a given layer, or we can use all of the activations in a given batch (or in principle both). With layer norm, after a given layer such as

$$X_n^i = \mathrm{ReLU}(W^{ij}X_{n-1,j} + b^i) \tag{5.1.3}$$

we can compute the mean and variance

$$\mu_n = \frac{1}{L}\sum_i X_n^i, \quad \sigma_n^2 = \frac{1}{L}\sum_i (X_n^i - \mu_n)^2 \tag{5.1.4}$$

[11] The role of batch norm in smoothing the loss landscape has been demonstrated both theoretically and empirically (https://arxiv.org/pdf/1805.11604.pdf)

and then replace X_n^i with

$$\hat{X}_n^i = \frac{X_n^i - \mu_n}{\sigma_n} \tag{5.1.5}$$

That's the layer norm. Note though that here the i index runs over the activations within a layer. Each data point in the batch is treated separately. We can simply think of this as *a modified definition for the layer itself.*

Usually, once we perform a layer-norm, we will then introduce an additional scale and bias term for each activation. But now these are parameters in the NN, so that the mean and variance of the activations (within a layer) will be under direct control.

At its simplest, batch norm uses the same, very basic equations, except that we *average over the batch for each activation*, rather than averaging over all the activations for each datapoint. But then batch norm may seem rather confusing... what do we do if we only have a single data point!?

To solve this problem in batch norm, typically during training we store accumulated means and variances (smoothed during training with a momentum-type parameter) for each layer. Then we store these values, and use them when testing. This makes it possible to apply batch norm to a single data point, once the model has been trained.

Note that Layer Norm can simply be viewed as a complicated alteration in the function that the NN represents. This isn't true of Batch Norm until we freeze the model, ie it isn't true until after the model is trained. This is an advantage of Layer Norm.

5.1.3 Initialization

Neural Networks are at least as complicated as a sequence of many matrix multiplications. But such products are dangerous – they may vanish or blow up, depending on the spectra of eigenvalues. So we should try to initialize NN weights so that the distribution of the activations and gradients will be the same from layer to layer. This makes initialization a natural part of 'whitening'.

With ReLU activation functions, from one layer to the next we have

$$X_n^i = \mathrm{ReLU}(W_j^i X_{n-1}^j + b^i) \tag{5.1.6}$$

The ReLU will kill roughly half of the components. Aside from that consideration, we need to think about is how the distribution of W and b affects that of X_n, accounting for the distribution of X_{n-1}. Let's assume that there are L_m variables X_m, so the weights are $L_n \times L_{n-1}$ matrices.

Note that it's a general fact that for independent random variables

$$\begin{aligned} \mathrm{Var}(AB) &= \langle (AB)^2 \rangle - \langle AB \rangle^2 = \langle A^2 \rangle \langle B^2 \rangle - \langle A \rangle^2 \langle B \rangle^2 \\ &= \mathrm{Var}(A)\langle B \rangle^2 + \langle A \rangle^2 \mathrm{Var}(B) + \mathrm{Var}(A)\mathrm{Var}(B) \end{aligned} \tag{5.1.7}$$

since $\mathrm{Var}(A) = \langle A^2 \rangle - \langle A \rangle^2$, which we can use to understand the distribution of WX. Also note that if a random variable is symmetric around 0, so that $p(x) = p(-x)$, then

$$\langle \mathrm{ReLU}(x)^2 \rangle = \frac{1}{2}\langle x^2 \rangle = \frac{1}{2}\mathrm{Var}(x) \tag{5.1.8}$$

The variance is unclear, since the effect of the ReLU on the mean is undetermined.

The distribution of X_n^i will have variance

$$\begin{aligned}
\mathrm{Var}(X_n^i) &= \mathrm{Var}(\mathrm{ReLU}(W_j^i X_{n-1}^j + b^i)) \\
&\approx \frac{1}{2}\mathrm{Var}(W_j^i X_{n-1}^j + b^i) \\
&= \frac{1}{2}\mathrm{Var}(b^i) + \frac{1}{2}\sum_{j=1}^{L_{n-1}} \left(\mathrm{Var}(W_j^i)\langle X_{n-1}^j\rangle^2 + \mathrm{Var}(X_{n-1}^j)\langle W_j^i\rangle^2 + \mathrm{Var}(W_j^i)\mathrm{Var}(X_{n-1}^j)\right) \\
&\approx \frac{1}{2}\mathrm{Var}(b^i) + \frac{1}{2}\sum_{j=1}^{L_{n-1}} \mathrm{Var}(W_j^i)\mathrm{Var}(X_{n-1}^j)
\end{aligned} \tag{5.1.9}$$

So this gives us a guess for the relation between the variances of subsequent layers. It's common to simply initialize $b^i = 0$ and rely on the random initialization of the weights to distinguish between the neurons.

Now the question is, what do we want or expect from the activations? If we want the activations X_m^i to have individual variance 1 for all layers m, then we should choose

$$\mathrm{Var}(W_j^i) = \frac{2}{L_{n-1}} \tag{5.1.10}$$

where we have ignored the variance of the biases, assuming that it will be chosen to be much smaller. Another possibility is that we would like the variance across an entire layer to be 1, ie $\sum_i \mathrm{Var}(X_n^i) = 1$, so that $\mathrm{Var}(X_n^i) = \frac{1}{L_n}$. In that case we should choose

$$\mathrm{Var}(W_j^i) = \frac{2}{L_n} \tag{5.1.11}$$

to propagate this normalization. The initialization procedure used in tensorflow actually chooses

$$\mathrm{Var}(W_j^i) = \frac{2}{L_n + L_{n-1}} \tag{5.1.12}$$

as this was suggested in a paper by Glorot and Bengio. This should be fine for fairly shallow networks, but it could be problematic for very deep networks, as it introduces a product over extraneous factors of 2. I don't know why tensorflow doesn't simply choose $\frac{2}{L_n}$ or $\frac{2}{L_{n-1}}$; this seems like a poor decision (but maybe I'm naive?). Note that the choice of L_n or L_{n-1} is essentially irrelevant, since signal propagation through the full network is controlled by their product over n.

In our analysis we have only studied forward propagation (evaluating the NN function), rather than the magnitude of gradients. The analysis of gradients is very similar, and leads to the same final result, except with L_{n-1} replaced by L_n. See 1502.01852 for some details.

Also, as a final comment – instead of initializing variables at different scales, one could alternatively initialize them all to have an order-one fixed scale, but then multiply the various activation functions by the relevant scale, instead. Then everything in the network would be 'order one'. It's not obvious if this is better or worse than standard practice, but it's probably worth considering.

5.1.4 Residual Connections

If you're training a very deep model, you should worry that information won't propagate through it. A simple way to solve this problem is to make your layers (or your blocks of layers) compute

$$X_{n+1} = X_n + F_n(X_n) \tag{5.1.13}$$

where F is some non-trivial NN function, so that your network is implementing the identity plus some function at each layer. That way, even if early layers aren't doing well, later layers still have access to (a lot of?) the original information in the data. The paper that first proposed the 'Resnet' has more citations than the papers that established the standard model of particle physics.

It is often said that there are two versions of the resnet, where what I wrote above is V2. The first version implements

$$X_{n+1} = \phi(X_n + F_n(X_n)) \tag{5.1.14}$$

where ϕ might be an activation function or some norm (like batch norm). This V1 resnet doesn't make a ton of sense if ϕ is a ReLU, as it would defeat the residual structure (though such networks do seem to exist and be in use).

Virtually all state of the art models for both image processing and language modeling are deep, and therefore use residual connections.

Note that with our nicer 'V2' resnet structure, we have, recursively

$$\begin{aligned} X_n &= X_{n-1} + F_{n-1}(X_{n-1}) \\ &= X_1 + \sum_{k=1}^{n-1} F_k(X_k) \end{aligned} \tag{5.1.15}$$

so that the last layer is the input plus various functions applied to the activations at every prior layer. This means that for deep models, the variances can blow up simply because we have so many layers, and so many different paths through the network. This can be 'fixed up' by dividing the initializations by a power of $\frac{1}{\sqrt{L}}$ where L is the number of residuall blocks, and the power is determined by the number of sub-layers within a block.

5.2 Recurrent Structures

Our minds maintain memories, and make decisions from moment-to-moment based on a combination of memory and present inputs. These decisions determine both the actions we take and the memories we subsequently store. Recurrent NNs are basically the simplest possible structure with these properties.

Aside from that motivation, we can also easily motivate RNNs by thinking about how to process sequence data, such as language. We read from left to right (in English), processing text letter-by-letter or word-by-word. If we read aloud, or translate from English to another language, at each step we might 'output' a result and store a 'memory'. This makes sense because many sequences, such as text, depend in some more or less complicated way on the proceeding **tokens** (letters or words) in

the sequence. We can also use recurrent models in stranger ways, for example by assigning an order to the pixels in an image, and 'reading' it pixel-by-pixel.

So without further ado, let's define the two most common recurrent NN structures.

5.2.1 RNNs

Let's imagine the input sequence comes in the form of ordered tokens x_k, which are themselves vectors in a d_x-dimensional space. Our recurrent model will have a 'memory' or hidden state h_k that's a vector of dimension d_h. The simplest possible model simply produces

$$h_{k+1} = \mathrm{ReLU}(W^{ij}(h_k, x_k)_j + b^i) \tag{5.2.1}$$

where (h_k, x_k) is simply the concatenation of the two vectors, so that the weight W has dimension $d_h \times (d_x + d_h)$, and the bias b has dimension d_h. That's all there is to a RNN!

After producing the h_k, we can do whatever we like with them. For example, we could feed them into another RNN, which is 'stacked on top of the first one'. Or we could use the h_k to perform some language task, such as classifying the sentiment of the sentence the LM just read.

Note that RNNs have an obvious problem, which is also shared with other very deep models – they can easily lose or over-amplify information within their recurrent structure.

5.2.2 Embedding Matrices

If our individual tokens are words, then... there's a lot of words! This is typically handled by using an *embedding matrix*, which is a linear map of dimension $d_{\mathrm{emb}} \times d_{\mathrm{vocab}}$, with typical values like $d_{\mathrm{vocab}} \sim 10^4$-$10^5$ and $d_{\mathrm{emb}} \sim 500$. It's only after processing text with a learned embedding matrix that we would input it into a recurrent model.

5.2.3 LSTMs

'Long Short-Term Memory' networks modify the very basic RNN structure to better preserve and utilize memory. When written out in equations, the modifications look quite involved, but they're quite simple when presented intuitively. There are a host of other options with different design choices, but the LSTM has been the most popular choice. I haven't bothered to write up the details because they're already nicely displayed here (http://colah.github.io/posts/2015-08-Understanding-LSTMs/).

Note that in terms of parameter counts, LSTMs contain 4 matrices of size $d_h \times (d_x + d_h)$, so they have

$$4d_h(d_x + d_h) \tag{5.2.2}$$

parameters, ignoring biases. This is also roughly the number of computations that they perform during a forward pass (evaluating themselves as a function) on a single token.

Aside: Normalization Prescriptions

Note that we can't use batch norm on the recurrent links in an RNN or LSTM. You might think it would be OK – we could just use a different norm for each sequential step – but we wouldn't know what to use at test time. But we can use layer norm, since layer norm is really just a change in the definition of the activations of a layer, as a function.

5.3 Convolutional Structures

Convolutional NNs account for the approximate translation invariance of the world, along with (much more approximate) small local diffeomorphism invariance.

CNNs

Convolutional NNs slide a set of 'filters' over an image. They also either use 'pooling' layers or a > 1 'stride' for the filters in order to spatially compress an image. For a given convolutional layer, there are three parameters to set

- Filter size, typically $3 \times 3, 5 \times 5$, etc.
- Stride – do the filters slide over the image in increments of a single pixel (stride = 1), or do they skip pixels, thereby compressing the result spatially
- Number of channels – how many filters do we have? Note that a color image begins with 3 channels. Typically CNN filters act on all channels in a given layer.

So for example, a CNN layer with 5×5 filters acting on an existing layer with 8 channels and mapping to a new set of activations with 16 channels will include NN weight matrices that are $5 \times 5 \times 8 = 200$ on one side and 16 on the other side, ie the will be 200×16 matrices, in effect. It's typical that successive CNN layers reduce image size spatially while adding more and more channels.

Convolutional NNs are primarily motivated by translation symmetry. One can also argue that they are natural from the point of view of small, general distortions interpreted as local diffeomorphisms. For example, the latter might motivate maximum-based pooling layers, as max-pooling will decrease sensitivity to distortion. These ideas can be formalized in equations (see work by Stephane Mallat and others), but as far as I know the results do not help us to choose better architectures, or to explain in detail why one CNN architecture is better than another.

Style Transfer

NNs can be used to achieve a fairly interesting and impressive effect – a kind of 'transfer of artistic style' (1508.06576). The idea is to look at the correlations between neurons in different layers when a trained CNN processes an image with the desired style. Then, we take a new image (usually a photograph) and apply gradient descent to the pixels of the image, with a loss that combines preservation of the original image with mimicry of the inter-layer correlations found in the stylized image. The results can be very cool.

ResNets

Nearly all nearly state-of-the-art convolutional image-processing models have a ResNet structure as described in our information propagation section. These were the first resnets, though the same structure is also now used for other networks, such as Transformers.

5.4 Attention and the Transformer

Attention is essentially a NN used as a highlighter.

A successful initial application of this idea was for sequence alignment. For example, say you want to understand how to translate from English to French, and you have equivalent sentences in each. For a given French word, you'd like to know which English words correspond with it. This means that you'd like to be able to point to a French word, and be able to immediately see a highlighted version of the English sentence that tells you where to look to find the equivalent meaning.

But perhaps the most important current application is *self-attention*. For instance, given an English paragraph, if I point to a given word, it's useful to know which other words are particularly relevant to the chosen word's meaning (and vice versa). One might want to process text by providing a new version of a sentence with these connections specifically emphasized. Very roughly speaking, that's what a layer in a *Transformer* does.

It's important to note that when we refer to attention we mean 'soft attention', where some initial vector V_α gets re-weighted by another vector or matrix that we can back-propagate through. One can also consider 'hard attention', where we make a hard sub-selection, but that's not the focus here. One would probably need to learn hard attention using RL.

5.4.1 Basic Idea of Attention

Highlighting portions of data means up-weighting and down-weighting relative importance. This naturally translates into the mathematical statement that given a vector of value V^i, we map

$$V^i \to \sigma^i V^i \tag{5.4.1}$$

for some weights $\sigma^i \in [0, 1]$. We'd like these σ to depend on something relevant.

In the literature, the dependencies are often referred to as keys and queries, and represented by K and Q. The idea is that we submit a query-vector, and it matches up with a key-vector. So if we separate out the 'key' and 'query' from the value, then

$$V_\alpha \to \sigma_\alpha(Q, K)V_\alpha \tag{5.4.2}$$

where K is a key associated with the query Q that helps us to decide what value V to highlight.

It's easy to come up with examples where the terminology makes sense. For instance, maybe V is an image, and what we want is to highlight sections of it. The query might correspond with a desire to highlight, say, a particular kind of animal that may be present in the image. Then the key will be some piece of information that tells us where the various kinds of animals are within the image. In this case (and many others), the key K will likely be the result of some computation applied to the image itself.

5.4.2 Transformers

Transformers act on their data through multi-headed self-attention. The 'self' here means that the K and Q both come from some computation applied to the original data V.

Another way of thinking about what K and Q do is via selective embedding spaces. The transformer takes words in their associated (large) embedding space and projects them into smaller embedding spaces which are used for the selection/highlighting/enhancement that may be relevant for a specific task (eg finding the objects of verbs). Perhaps different subspaces handle grammar, causal relations, vocabulary choice, etc... each of these may correspond to a different head in a given Transformer layer. (One can see this happening in some specific examples, but when I talk about these categories I'm just making them up.)

Let's discuss the version of attention that appears in the Transformer model. In that case we have some d_{seq} length list of inputs, eg a sequence of words. Furthermore, each word or token is itself a $d_{\text{embedding}}$ dimensional vector in an embedding space. So the input will be V, a $d_{\text{seq}} \times d_{\text{embedding}}$ tensor. The original paper chose $d_{\text{embedding}} = n_{\text{seq}} = d_{\text{model}}$, and used $d_{\text{model}} = 512$, though significantly larger models are in use now. It's not at all obvious that $d_{\text{embedding}} = n_{\text{seq}}$ is optimal.

It would be very computationally expensive to use matrices that act directly on a $d_{\text{emb}} \times n_{\text{seq}} > 10^5$ dimensional vector space of the full V! Sequence models avoid this problem by only looking at one token at a time, and storing the rest in memory (hopefully). What about Transformers?

Self-Attention

The idea is that a given self-attention head will have a key and a query for each position in the sequence. The data comes in as a matrix V^{se}, where s denotes position in the sequence, and e denotes the embedding index. So we will have keys

$$K^{ks} = W_K^{ke} V_e^s, \quad Q^{sk} = W_Q^{ke} V_e^s \tag{5.4.3}$$

We also have queries

$$Q^{sk} = W_Q^{ke} V_e^s \tag{5.4.4}$$

where W_Q and W_K are matrices. Here the dimension of the keys and queries is d_k, where in the original paper they choose to have 8 attention heads, and so to choose $d_k = 512/8 = 64$ (but this isn't obligatory). You can think of d_K as the dimension of a sort of new, per-head embedding space. Notice that W_K and W_Q do not act in the sequence-space at all.

Taken together, we then have the self-attention map

$$\text{Self} - \text{Attention}(V)^s = \text{softmax}_{s'} \left(\frac{Q^{sk} K_k^{s'}}{\sqrt{d_K}} \right) V_{s'}^e \tag{5.4.5}$$

where Q and K were defined above, and I put a subscript on the softmax to indicate what axis we're softmaxing over. In the literature the numerator is typically written as QK^T. Notice that in fact

$$S^{ss'} = \text{softmax}_{s'} \left(\frac{Q^{sk} K_k^{s'}}{\sqrt{d_K}} \right) \tag{5.4.6}$$

is a $n_{\text{seq}} \times n_{\text{seq}}$ matrix of values. The matrix is itself made from a quadratic form acting on V. So this is how the words effect each other! Once we determine $S^{ss'}$, we are just multiplying the $V^{s'e}$ matrix by it, and acting only on the s-indices, and not on the embedding indices... so far.

But there's one more ingredient to keep the dimensionality from getting out of hand. We also project the embedding down again after applying self attention, so that

$$V^{se} \to W_V^{ke} V_e^s \tag{5.4.7}$$

which is a lower dimensional vector. So the full transformation for one head is

$$\text{Head}^{h,sk}(V) = \text{softmax}_{s'}\left(\frac{Q^{h,sk'}K_{k'}^{h,s'}}{\sqrt{d_K}}\right)V_{s'e}W_V^{h,ek} \tag{5.4.8}$$

where K and Q were defined above, and I have added an 'h' label to them to emphasize that there is a different Q, K, and thus a different underlying W_Q, W_K, for each of the heads.

To get the full layer, we concatenate the heads back together to give ourselves a new output. Finally, we multiply by a matrix $W_O^{e,(hk)}$

$$Z^{se} = W_O^{e,(hk_i)}\text{Concat}(\text{head}_1^{sk_1}, \text{head}_2^{sk_2}, \cdots, \text{head}_{n_{\text{heads}}}^{sk_{n_{\text{heads}}}}) \tag{5.4.9}$$

with the same dimension as our input V^{se}, where (hk_i) denotes both indices (as a tensor product).

While the original paper chose for Z and V to have the same dimensions, and this is useful for layer-stacking, it's not conceptually necessary. There's not much of a natural correspondence between Z and V, since the 'e' index of Z is just made from the concatenation of $n_{\text{heads}} = d_{\text{model}}/d_k$ different attention heads. Furthermore, we could even have different heads with different sizes. As usual in ML, we're doing violence to our vector spaces.

Let's count parameters. We have $3d_k d_{\text{emb}}$ parameters in W_K, W_Q, W_V for each head, and then d_{emb}^2 parameters in W_O, so this leads to

$$P_{\text{MHA}} = 3(d_k n_{\text{heads}})d_{\text{emb}} + d_{\text{emb}}^2 = 4d_{\text{emb}}^2 \tag{5.4.10}$$

parameters in the self-attention mechanism.

Fully Connected Layer

We also act with a feed-forward network that acts *word-by-word* independently. And in fact the original paper includes two linear transformations with a ReLU in between. So this is just a fully connected network acting on the 'e' index of the Z^{se} output via

$$Z^{se} \to W_{2,x}^e \text{ReLU}(W_1^{xe'}Z_{e'}^s + b_1^x) + b_2^e \tag{5.4.11}$$

In the original paper, they chose the dimensionality of the 'in between' layer to be $d_{\text{ff}} = 4d_{\text{model}} = 2048$, so this is the range of the x index. This means that there are

$$P_{\text{FC}} = 2d_{\text{ff}}d_{\text{emb}} + d_{\text{ff}} + d_{\text{emb}} \tag{5.4.12}$$

parameters in this fully connected layer, where we have also included biases, though they will be negligible in practice. Note that

$$\frac{P_{\text{FC}}}{P_{\text{MHA}}} \approx \frac{d_{\text{ff}}}{2d_{\text{emb}}} \tag{5.4.13}$$

but it's typical to take $d_{\text{ff}} = 4d_{\text{emb}}$, so that there are twice as many FC parameters as MHA parameters.

Building a Transformer

To finish off a Transformer, we need to give it a Resnet-like structure, add many copies of Layer Norm, and include a positional encoding in the initial sequence data.

The position encoding is necessary because the layers of the transformer do not have any absolute or relative reference points. To see this, note that *none of the matrices we defined above act on the s indices*! So for this purpose, we simply add to the initial tensor V^{se} a constant tensor P^{se}. One option is to learn this encoding by letting P^{se} be initialized randomly and be learned through training. Another option is to choose some fixed P, such as various sines and cosines (as in the original paper). The original paper claims these work equally well.

As for the resnet-esque and layer norm structures... this means that given an initial Z, we compute the multi-headed attention operation on V to obtain a Z, and then we simply apply layer norm to their sum to give $X = \text{LN}(V + Z)$. Next we apply the fully-connected layer to this, add it back to itself, and perform another layer norm. So this looks like

$$X_n = \text{LN}\left(V_n + \text{MHA}(V_n)\right)) \tag{5.4.14}$$

and then the output is

$$V_{n+1} = \text{LN}\left(X_n + \text{FC}(X_n)\right)) \tag{5.4.15}$$

where LN is layer norm, FC is the fully connected layer described above, and MHA is the multi-headed attention layer described above. A transformer with many layers just performs n_{layers} sequential operations of this form.

Finally, we may take the outputs and perform some final operation to obtain logits and probabilities, which can be used to sample or autoregressively predict.

So including the embedding matrix itself (connecting a vocabulary of tokens to an embedding vector space), we have a parameter count

$$P \approx 2n_{\text{layers}}d_{\text{emb}}(d_{\text{ff}} + 2d_{\text{emb}}) + d_{\text{emb}}n_{\text{vocab}} \tag{5.4.16}$$

where typically n_{vocab} is 10 to 100 times larger than the other dimensions. If we scale all of these dimensions together as we increase d_{seq} (not sure if they should!), then the Transformer parameter count is cubic in d_{seq}. Transformers processing sequences with ~ 500 tokens (words) are common in the literature, and involve models with much less than a billion parameters. So this suggests that

with a trillion parameters it'll be possible to use contexts of order 10 pages of text, or significantly more (!), the length of a short story, assuming we scale up all parameters uniformly.

Note that as described, we've actually only covered a pure-Encoder or pure-Decoder Transformer. The original model had an Encoder-Decoder structure, and was used for translation. In this version, the Encoder acts as described above, but the Decoder actually has two attention layers, one that's pure self-attention, and another that obtains keys and queries from the Encoder (and inputs), but values from the outputs. In that way, when writing a translation, the model could both self-attend (ie to the new sequence it's writing) and also simultaneously attend to the inputs (ie the sequence it's translating from).

Quantity of Computation

How much computation is necessary to run a Transformer?

For the fully connected layers, we multiply by a $d_{\text{emb}} \times d_{\text{ff}}$ matrix followed by a $d_{\text{ff}} \times d_{\text{emb}}$, for every vector in the context. That means that we must perform roughly $2n_{\text{seq}}d_{\text{emb}}d_{\text{ff}}$ operations to evaluate one layer of a Transformer. Note that this dwarfs the operations associated with the biases and ReLUs. It's also noteworthy that this is merely linear in n_{seq}.

For the self-attention mechanism, we need to compute the $S^{ss'}$ matrix for each head. For the Q part this requires $n_{\text{seq}} \times d_{\text{emb}} \times d_k$ operations, plus the same number for the K part. Multiplying them together takes $n_{\text{seq}}^2 d_k$ operations. Thus computing $S^{ss'}$ requires $n_{\text{seq}}^2 d_k^2 + 2n_{\text{seq}}d_{\text{emb}}d_k$ operations. Acting with W_V uses $d_k d_{\text{emb}} n_{\text{seq}}$ operations, while W_O uses $n_{\text{seq}}d_{\text{emb}}^2$ operations. Acting $S^{ss'}$ on $W_V V$ then requires $n_{\text{seq}}^2 d_{\text{k}}$ operations. So we see that a single layer of the Transformer requires

$$\begin{aligned} \text{Compute} &= FC + Att \\ &\approx 2n_{\text{seq}}d_{\text{emb}}d_{\text{ff}} + n_{\text{head}}\left[n_{\text{seq}}^2 d_k + 2n_{\text{seq}}d_{\text{emb}}d_k + n_{\text{seq}}d_{\text{emb}}d_{\text{k}}\right] + n_{\text{seq}}d_{\text{emb}}^2 \\ &= 2n_{\text{seq}}d_{\text{emb}}d_{\text{ff}} + n_{\text{head}}\left[n_{\text{seq}}^2 d_k + 3n_{\text{seq}}d_{\text{emb}}d_k\right] + n_{\text{seq}}d_{\text{emb}}^2 \\ &= 2n_{\text{seq}}d_{\text{emb}}(d_{\text{ff}} + 2d_{\text{emb}}) + n_{\text{seq}}^2 d_{\text{emb}} \end{aligned} \tag{5.4.17}$$

computations for one layer on the forward pass, where we have only kept the dominant terms, and in the last line we set $n_{\text{head}}d_k = d_{\text{emb}}$. From the perspective of parameter counting, very large n_{seq} has very low cost, but it does require significantly greater total computation. Furthermore, it's noteworthy that we can scale up n_{seq} until its of order $12d_{\text{emb}}$ without the quadratic term in sequence length dominating.

6 A Toy Model for Generalization and Overfitting

Can ML algorithms *generalize* from the specific examples in a dataset in order to really understand the underlying data distribution? Practically speaking, this questions is most often addressed by comparing performance on the training set to the test set, so that the question of generalization amounts to the issue of overfitting. Note that subject of improving generalization and avoiding overfitting is called *regularization*.

But there's more to generalization than just overfitting. Unfortunately, my sense is that we don't know a great deal about it. To get us started, here's an intermingled list of questions, lore, and comments on it:

- What sort of abstractions or features do NNs learn, in what order, and why? In simple cases it's reasonable to expect that NNs will learn robust features first, as these will correspond to larger and more consistent training gradients. But I'm not sure to what extent this... generalizes. It's an interesting research question.
- An oft-repeated item of lore is that NNs will learn to *interpolate, but not to extrapolate*. This may be true, but it's not very well-defined. In particular, since our data (and models) live in extremely high-dimensional spaces, the data will be very sparsely distributed, eliding the distinction. And once we map to some lower-dimensional feature space, it's less clear why extrapolation can't succeed.
- A better defined version of "won't extrapolate" is the question of whether it's possible for ML algorithms to generalize beyond the data distribution that they've been trained on.
- Another item of lore is that wide minimum of the loss generalize better than sharp minima. Conceptually, this makes sense because general patterns shouldn't be as sensitive to small changes in the NN parameters. There's some evidence for this, but typically it's only studied in parameter space, which means that the results aren't re-parameterization invariant.
- As we've noted before, NNs are typically in a regime where the number of parameters $\gg$ number of points in the dataset. Naively, this would lead us to expect dramatic overfitting, but that's not what we observe. In fact, typically *larger models generalize better than smaller models, as long as the larger models are stopped early*. In many cases the worst generalization performance occurs when the number of parameters is equal to the number of data points, with better generalization away from this regime in either direction.
- However, it is possible to get NNs to memorize the dataset – classifiers can be trained with random labels and get perfect accuracy on the training set (and zero generalization, of course). But this typically takes much, much more training than is typical.

To improve generalization, there are three common techniques for regularization

- Constrain the capacity of the model, eg by limiting the range of parameters by adding an auxiliary loss.
- Add noise to the data or parameters to avoid overfitting. The most common technique is dropout.
- Simply stop training early, once the validation loss stops decreasing, even if the training loss continues to go down.

Generalization, overfitting, regularization, and early stopping are typically illustrated by drawing the train and test loss during training, which has a characteristic form. We'll study a toy example in the next section, which explains many surprising features of NNs and overfitting.

A Solvable Linear Student/Teacher Example

Let's consider a toy example[12] involving a linear regression model.

For this purpose, we will work in a 'student teacher' framework where the student model tries to learn to copy the teacher model. So the student has a vector of P weights $w(t)$, where t is the training-time, while the teacher has weights $\bar{w}$. To learn a general lesson, we will average over all possible $\bar{w}$ drawn from some distribution. Note that this has a major conceptual disadvantage – real data isn't generated by a random model, but by models with a great deal of non-trivial structure, so our analysis of generalization will not be sensitive to such structure.

We will assume that there are D data points x, so that the set of all datapoints is a matrix X of DP numbers. The outputs of the teacher network, which are what we want to learn, are

$$y = \bar{w}x + \epsilon \tag{6.0.1}$$

where we will add some noise ϵ to these results. We will assume that both $\bar{w}$ and ϵ are drawn from Gaussians with variance σ_w and σ_ϵ. We will also assume that X are Gaussian distributed with zero mean and unit variance. The goal is to learn $\bar{w}$ from the noisy data (x, y). We study the generalization dynamics *averaged over all* $\bar{w}$.

The objective is just the squared error

$$\begin{aligned} L &= \frac{1}{2}\sum_{i=1}^{D}(y_i - \hat{y}_i)^2 = \frac{1}{2}\sum_{i=1}^{D}(w\cdot x - \bar{w}x - \epsilon)^2 \\ &= \frac{1}{2}\left(w_\alpha w_\beta + \bar{w}_\alpha\bar{w}_\beta - 2w_\alpha w_\beta\right)X^{\alpha i}X_i^\beta + \epsilon_i X^{\alpha i}(w_\alpha - \bar{w}_\alpha) + \frac{\epsilon^2}{2} \end{aligned} \tag{6.0.2}$$

where we are using the full dataset. Here α, β indices range over the P parameters, while i, j indices range over the D points in the dataset. The gradient of the loss is

$$\partial_\alpha L = \left[\left(w_\beta - \bar{w}_\beta\right)X_i^\beta + \epsilon_i\right]X_\alpha^i \tag{6.0.3}$$

For convenience we can study the continuum limit of learning where

$$\dot{w}_\alpha = -\left[\left(w_\beta - \bar{w}_\beta\right)X_i^\beta + \epsilon_i\right]X_\alpha^i = y_i X_\alpha^i - w_\beta X_i^\beta X_\alpha^i \tag{6.0.4}$$

We can solve this differential equation, and then use it to compute the generalization error

$$E_{\text{gen}} = \langle (y - \hat{y})^2\rangle_{x,\epsilon} \tag{6.0.5}$$

for new data points x, ϵ.

First, in the limit of $t \to \infty$, we will obtain

$$w_\beta X_i^\beta X_\alpha^i = y_i X_\alpha^i \tag{6.0.6}$$

[12]This example is stolen from 1710.03667, though many earlier references come to similar conclusions.

so that the w_β are determined by multiplying on the right with the inverse (or pseudo-inverse) of the matrix $X_i^\beta X_\alpha^i$.

To determine the time-dependence of w, we can diagonalize

$$X_i^\beta X_\alpha^i = \Sigma_\alpha^\beta = V^{\beta\pi}\Lambda_{\pi\pi}(V^T)_\alpha^\pi \tag{6.0.7}$$

where Λ is diagonal and V is orthogonal. Then we can write

$$y_i X_\alpha^i = \tilde{s}_\pi (V^T)_\alpha^\pi \tag{6.0.8}$$

where $\tilde{s}$ is a vector. If we now write $w = zV^T$ then we find the simplification

$$\begin{aligned} \dot{z}V^T &= y_i X_\alpha^i - w_\beta X_i^\beta X_\alpha^i \\ &= \tilde{s}V^T - z\Lambda V^T \end{aligned} \tag{6.0.9}$$

so that

$$\dot{z} = \tilde{s} - z\Lambda \tag{6.0.10}$$

meaning that we have fully diagonalized the dynamics in terms of the eigenvalues λ_α of Λ. Note that if $\bar{z} = \bar{w}V$, then we expect

$$yX^T = \bar{z}\Lambda V^T + \tilde{\epsilon}\Lambda^{1/2}V^T \tag{6.0.11}$$

where $\tilde{\epsilon}$ is drawn from a Gaussian with variance σ_ϵ^2, so that

$$\dot{z}_\alpha = (\bar{z}_\alpha - z_\alpha)\lambda_\alpha + \tilde{\epsilon}_\alpha\sqrt{\lambda_\alpha} \tag{6.0.12}$$

where we recall that $\alpha = 1, \cdots, P$ has as many values as parameters in the model.

In our linear model we can study generalization mode-by-mode. *In a deep model this would be a higher order differential equation, with coupling between the modes.* But otherwise the behavior may be qualitatively similar. The error in each mode of the linear model will have time dependence

$$\bar{z}_\alpha - z_\alpha = (\bar{z}_\alpha - z_\alpha(0))e^{-\lambda_\alpha t} - \frac{\tilde{\epsilon}_\alpha}{\sqrt{\lambda_\alpha}}\left(1 - e^{-\lambda_\alpha t}\right) \tag{6.0.13}$$

This means that the time-dependent generalization error will be

$$\begin{aligned} E_{\text{gen}}(t) &= \frac{1}{P}\sum_\alpha \langle(\bar{z}_\alpha - z_\alpha)^2\rangle + \sigma_\epsilon^2 \\ &= \frac{1}{P}\sum_\alpha \left[\left(\sigma_{\bar{w}}^2 + \sigma_{w_0}^2\right)e^{-2\lambda_\alpha t} + \frac{\sigma_\epsilon^2}{\lambda_\alpha}\left(1 - e^{-\lambda_\alpha t}\right)^2\right] + \sigma_\epsilon^2 \end{aligned} \tag{6.0.14}$$

where we got the second line by assuming no correlation between $\epsilon, \bar{z}$, or z_0, the initial value of the z, which is determined by the initialization scheme for the parameters.

This formula for the generalization error is a key result. We see that it's composed of two terms. The first strictly decreases with time at rates set by the eigenvalues of Λ. In the absence of any error, and in a model with expressivity to fit the full data distribution, the model improves over time. The second term grows at a rate and to a size set by the Λ eigenvalues. It's responsible for overfitting, as the model overfits to the specific features (errors!) in the finite dataset. *Thus we see that the spectrum of eigenvalues λ_α is key to the generalization behavior of our model, and this in turn will depend on P vs D.* Note that vanishing eigenvalues $\lambda_\alpha = 0$ behave rather differently from small, finite eigenvalues. The former form a frozen subspace that never learns, and therefore never overfits. In contrast, the small positive eigenvalues can lead to bad overfitting problems at late times.

In the limit where $P, D \to \infty$ with $r = D/P$ is fixed, and the matrix X has random Gaussian entries, the matrix XX^T has eigenvalues that approach the Marchenko-Pasteur distribution

$$\rho(\lambda) = \frac{1}{2\pi} \frac{\sqrt{(\lambda_+ - \lambda)(\lambda - \lambda_-)}}{\lambda} + (1 - r)\delta(\lambda) \tag{6.0.15}$$

for $\lambda \in [\lambda_-, \lambda_+]$ or $\lambda = 0$, and where $\lambda_\pm = (1 \pm \sqrt{r})^2$. The second term only contributes when $r < 1$. This distribution then governs the dependence of overfitting on the ratio $r = D/P$. The very interesting fact is that as $P \to \infty$, so that $r \to 0$, all eigenvalues are either 0 or 1! Thus models with very, very large number of parameters do not overfit. We also see that if $r \to \infty$, so that $P \ll D$, there also isn't much overfitting. Very serious overfitting only occurs when $\lambda \approx 0$ is possible, and that's only the case when $r \approx 1$.

Another nice feature of the model is that we can analytically compare different regularization schemes. It turns out that L^2 regularization happens to be optimal here (because the noise was assumed Gaussian), but that optimal early stopping performs almost as well. Unfortunately I don't think there's a useful way to mock up the effects of dropout on this model and compare them.

7 Unsupervised Learning

On the most abstract level, the goal of UL is to 'understand the world', or at least the data, without specifying any more specific task. If this could be accomplished, then other forms of learning might be trivialized. So in principle, Unsupervised Learning should be the final frontier[13] of ML. However, in practice researchers have made progress on UL by making it much more concrete, and in many cases turning UL into something much closer to SL.

To formalize UL, the dominant approach is to ask **how can we model the data distribution** $P(X)$? Note that a model for $P(X)$ will allow us to sample from it, meaning that when we're doing **UL, we are typically learning a generative model**, ie models that generate new data samples. This may be extremely useful (in the near future) for RL, since it allows agents to plan and dream.

But it's worth pausing to emphasize how ambiguous $P(X)$ may be. For instance, take the example of pictures of cats, which are some sort of sub-space within the space of pixel color intensities. Clearly this is a very small subspace – but what's the role and interpretation of probability? Are

[13]This is quite plausible... at present, we already have examples where we can solve a variety of SL problems by learning one UL representation, and then quickly 'fine-tuning' it to solve any other given SL task.

pictures of cats defined by human consensus, or by the space of images that can be captured by pointing a camera at a physical cat? And what would the boundaries of this space actually look like... are they defined by human perceptual ambiguity? Do we care if a certain lighting pattern is very unlikely for a camera, but could be trivially achieved in a cat image drawn as a cartoon?

We can also talk about similar questions in language modeling. Here the task is much easier to specify – we can ask for the probability distribution of the next word given a previous sequence of words. But this could depend a lot on author, subject, etc.. and some statistical properties of language are likely very robust (ie a given word-sequence either is or isn't a sensible sentence) and some are quite variable; it's unclear which to focus on, or how to control that focus. (This could probably be stated much more nicely by an NLP expert.)

These rather philosophical considerations are important for determining our goals. Do we really want an explicit probability distribution, or do we just want to be able to randomly sample from it? The way we answer that last question provides the most important dichotomy within the space of UL models, with VAEs, Autoregressive Models, and Flow-Based models on one side (explicit $P(X)$), and GANs on the other side (only sampling).

As a final note, there are many classical UL-ish algorithms that extract features from the data, or find patterns in it, but don't provide generative models. Examples include k-means, singular value decompositions, and t-SNE.

7.1 Latent Space – A Beautiful Dream

Underlying a great mean approaches to UL is the dream of a 'latent space' where everything suddenly becomes simple. Intuitively, 'understanding' something means knowing how to describe it efficiently, capturing it's most important features and ignoring those aspects that are either irrelevant or easily inferred. From this point of view, understanding has a lot to do with (mildly lossy) compression.

Thus a classic NN structure related to UL is the **autoencoder**, which we typically view as a map $A : X \to X$ via

$$A(x) = p_\theta(q_\phi(x)) \tag{7.1.1}$$

where $q_\phi : X \to Z$ and $p_\theta : Z \to X$. In this case Z is the latent space. The key point here is that we choose $\dim(Z) \ll \dim(X)$, so that the autoencoder is compressing the data. The hope is that Z now 'knows' about the important features of X. The autoencoder objective is simply good reconstruction, ie we want $A \approx$ the identity map on X.

Latent spaces appear in many, many other contexts. For example, the *embedding matrix* that maps words $\to$ vectors is another example. There are a host of ways to learn good word embedding vectors; probably the best way is to simply train a powerful language model. But even very simple techniques like PCA on the matrix of word correlations (words are correlated if they appear close together, see eg the 'Latent Semantic Analysis' algorithm), can reveal a surprising amount of structure. This is a huge topic with a long, rich history, as well as many recent advances. Images, language, and many other types of data can now be embedded into huge vector spaces, where 'addition' and linear interpolation have a clear semantic meaning.

Many of the models we discuss below realize some latent space representation for the data, often without directly optimizing for it. Many interesting state of the art models involve tweaks that improve on latent space representations.

7.2 Maximizing the Likelihood of the Data

A large class of generative models simply try to find a model $p_\theta(X)$ that's as similar as possible to the ground truth $P(X)$. The training objective is typically maximum likelihood of the data

$$L = -\frac{1}{N}\sum_{i=1}^{N} \log p_\theta(x_i) \tag{7.2.1}$$

which means that it's exactly equivalent to minimizing $D_{KL}(P||p_\theta)$ by sampling. There are three fairly common strategies

- Autoregressive Models: These models learn to generate data sequentially, by predicting $p_\theta(w_{n+1}|w_n \cdots w_1; z)$ for the 'words' w_i in a sequence and perhaps some latent variables z. This is extremely natural for language. It's less natural for images, but it can be done by choosing an ordering for the pixels... and this actually works quite well.

- 'Flow' Models: These models learn to construct a bijective (an explicitly invertible) map $f : Z \to X$ from a latent space Z to the data space X, where the probability distribution $q(z)$ on the latent space is assumed to be a known, fixed, simple distribution such as a Gaussian. Thus $p_\theta(x)$ is just determined by $q(f^{-1}(x))$. The main challenge here is learning a sufficiently expressive invertible map.

- Variational Autoencoders (VAE): These models learn a typically uncontrolled, non-invertible map $f : Z \to X$. They maximize the log-likelihood of the data by replacing it with a 'surrogate objective', ie a *lower bound* on the log-likelihood. Depending on how one explains these models, they may seem totally trivial, or much more complicated than the two ideas above... but they're really in the same category.

Autoregressive models are basically self-explanatory. The only challenge with flow-based models is building a flexible, invertible map. So we'll spend a bit more time on VAEs, since they're the least transparent of the three ideas above.

7.3 Variational Autoencoders (VAE)

Our real goal is to learn the probability distribution $p_\theta(X)$ for the data, and to be able to sample from it efficiently.

7.3.1 VAEs, Very Pragmatically

Let's first start by talking about autoencoders. An autoencoder expresses the identity map[14] on the data distribution

$$\mathbf{1}(y;x) = \int dz\, p_\theta(y;z) q_\phi(z;x) \tag{7.3.1}$$

where p and q are the decoder and encoder (represented by neural networks), and the latent space Z is usually lower-dimensional than X, so that the encoder compresses $X \to Z$.

Note that autoencoders are usually deterministic, in which case the above clearly makes sense. We might instead think about p and q as probability distributions, where for example we sample $y \sim p_\theta(\cdot|z)$, and $z \sim q_\phi(\cdot|x)$.

We would like to learn θ, ϕ so that we can use $p_\theta(x;z)$ to sample $x \in X$ by sampling z. If we simply train the model to approximate the identity map on the data, we could then try sampling random z and the computing $p_\theta(z)$, or if $p_\theta(Y|z)$ is a probability distribution, then we can sample y from it. Why not just stop there and declare victory?

The problem is that by itself, our pure autoencoder setup tells us nothing about the distribution of the latent variables z. For all we know, the autoencoder will learn a really weird pattern of z values associated to datapoints x, and we won't have any way to interpolate between them, or to sample from z in a sensible way.

We can solve this problem by combining an autoencoder loss (which leads p and q to learn to reconstruct the identity map) with an auxiliary loss that forces Z to have a simple and tractable distribution, such as a multi-variate Gaussian. That is, we can make the loss be

$$\text{Loss} = \text{Reconstruction Error} + \beta D_{KL}(q_\phi(Z|X)||q_{\text{simple}}(Z)) \tag{7.3.2}$$

where $q_{\text{simple}}(Z)$ is some very simple, standardized distribution like a Gaussian. Our coefficient β just lets us trade off how much we care about good reconstruction vs having a smooth distribution over z. When β is large, we know that $q_\phi(Z|x)$ will be quite smooth and simple in Z, meaning that if we sample $y \sim p_\theta(z)$ where $z \sim q_{\text{simple}}$, then we'll always get a good interpolation among data points. If β is small, then reconstruction will be good, so our samples for z near maxima of $q_\phi(z|x)$ will look good, but we'll be less certain for other z.

The thing we just defined was called a β-VAE in the literature. Practically speaking, that's it. But it turns out that we can provide a nice theoretical justification of this construction and pick out a natural value $\beta = 1$. The idea is to use maximum likelihood, but to maximize a lower bound on the likelihood instead of the probability itself. Let's see how.

7.3.2 VAEs, More Theoretically

Before deriving the VAE loss, let's try to better define the object's we'll be manipulating.

We should first emphasize that latent space Z has been made up by us. What we're really interested in is $p_\theta(X)$, while the latent space is just a tool to make it easier to study it. Furthermore,

[14] In the deterministic limit q_ϕ is a delta-function, and we find $p_\theta(y; q_\phi(x))$.

it's worth reminding ourselves that $p_\theta(X|Z)$ and $q_\phi(Z|X)$ are genuine probability distributions over X and Z, and not merely deterministic functions from $Z \leftrightarrow X$. That is, for each z the $p_\theta(X|z)$ is a probability distribution. Given that we have a latent space, there's now a joint distribution $p_\theta(X, Z)$ where

$$p_\theta(X) = \int dZ\, p_\theta(X, Z) \tag{7.3.3}$$

and furthermore, by Bayes rule

$$p_\theta(x, z) = p_\theta(x|z)p_\theta(z) = p_\theta(z|x)p_\theta(x) \tag{7.3.4}$$

Since the latent space was made up by us, we are free to choose $p_\theta(z)$ to be whatever we like!

In particular, we can eg choose $p_\theta(Z)$ to be totally independent of θ, and just use some simple Gaussian distribution over Z with fixed mean and variance. Thus we'll now write $p_\theta(Z) = p_{\text{simple}}(Z)$ as a fixed distribution. This means Bayes rule has become

$$p_\theta(x, z) = p_\theta(x|z)p_{\text{simple}}(z) = p_\theta(z|x)p_\theta(x) \tag{7.3.5}$$

which we will use below.

Now we will derive a bound on our NN model's $p_\theta(X)$ log-likelihood. The idea is to use a 'variational' bound. The word variational means the same thing here as in physics, namely...

Variational: To physicists, the 'variational approximation' is based on the trivial-seeming observation that if $f(x)$ is a function for which we know $f(x) \geq 0$, and we are interested in determining $\min[f(x)]$, then by definition $\min[f(x)] < f(x_0)$ for any x_0. So by trying out some different values for x_0 we can obtain an (uncontrolled) approximation for $\min[f(x)]$. This can seem non-trivial when 'x' is drawn from a high or infinite dimensional space (eg if x is a 'set of points' in the space of functions), but it's still just as simple.

Variational Bound and Maximum Likelihood: In the VAE context, we are interested in the relationship between a latent-space distribution over Z and the data distribution over X. We will now introduce an extra learned conditional probability distribution $q_\phi(z|x)$. We can write

$$\begin{aligned} \log p_\theta(x) &= \log p_\theta(z, x) - \log p_\theta(z|x) \\ &= \log p_\theta(z, x) - \log q_\phi(z|x) + (\log q_\phi(z|x) - \log p_\theta(z|x)) \end{aligned} \tag{7.3.6}$$

Since this is true for any z where the logs are well-defined, it's also true if we evaluate its expectation on $z \sim q_\phi(z|x)$, giving

$$\log p_\theta(x) = \mathbb{E}_{z\sim q_\phi(z|x)} \left[\log p_\theta(z, x) - \log q_\phi(z|x)\right] + D_{KL}[q_\phi(z|x)||p_\theta(z|x)] \tag{7.3.7}$$

Since the KL divergence is positive, we can define

$$\mathcal{L}(\phi; x) \equiv \mathbb{E}_{z\sim q_\phi(z|x)} \left[\log p_\theta(z, x) - \log q_\phi(z|x)\right] \tag{7.3.8}$$

and set a lower bound

$$\log p_\theta(x) \geq \mathcal{L}(\phi; x) \tag{7.3.9}$$

We can make this look more useful by noting that via Bayes rule

$$\mathcal{L}(\phi; x) = \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p_\theta(x|z)\right] - D_{KL}\left[q_\phi(z|x)||p_{\text{simple}}(z)\right] \tag{7.3.10}$$

where we have recalled that $p_\theta(Z) = p_{\text{simple}}(Z)$.

Notice that *the first term looks a lot like a log probability of x sampled through the autoencoder.* We take x, compute $q_\phi(Z|x)$, then sample a specific z, then compute the log of the conditional probability $p_\theta(x|z)$. If $q_\phi(z|x)$ was just a delta function, this would just be a reconstruction error (literally a log probability of reconstructing the right x).

Optimizing the VAE means making the lower bound as tight as possible, which means maximizing $\mathcal{L}$. Since D_{KL} is positive, we want to minimize it, ie make $q_\phi(z|x)$ as similar as possible to $p(z)$. This variational method is identical to the variational approximation used by physicists for quantum mechanical wave-functions and ground states. Instead of maximizing the probability, we are *maximizing a surrogate loss, a lower bound on the log probability.*

Reparameterization: The original VAE paper states that this bound is difficult to optimize on its own due to high-variance. I think what they really mean is that it's not possible to backprop through z if it's an intermediate layer and is stochastic, but there's no problem backpropping if we use the reparameterization trick and introduce the stochasticity at the level of the graph input. Recall we defined the reparameterization trick in section 3.5.

Anyway, we can approximate conditional distributions $z \sim q_\phi(z|x)$ as $z = g_\phi(\epsilon, x)$ where $\epsilon \sim p(\epsilon)$ is an auxiliary random variable drawn from a simple fixed distribution $p(\epsilon)$, and $g_\phi : (\epsilon, x) \to z$ is a potentially complicated deterministic function that can eg be parameterized by a neural network.

In this way writing $q_\phi(z|x) = g_\phi(\epsilon, x)$ and using our original description of $\mathcal{L}$, we now have

$$\begin{aligned}\mathcal{L}(\phi, \theta; x) &= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p_\theta(x|z)\right] - D_{KL}\left[q_\phi(z|x)||p(z)\right] \\ &= \mathbb{E}_{\epsilon\sim p(\epsilon)}\left[\log p_\theta(x|g_\phi(\epsilon, x))\right] - D_{KL}\left[q_\phi(g_\phi(\epsilon, x)|x)||p(g_\phi(\epsilon, x))\right]\end{aligned} \tag{7.3.11}$$

Now the first term is just an auto-encoder loss (though it's 'noised' by ϵ, which from the autoencoding perspective seems like it'll make results worse), while the second term can be viewed as a regularizer.

For example, we can choose $g_\phi(\epsilon, x)$ so that a NN maps $x \to (\mu_\phi(x), \sigma_\phi(x))$ for a Gaussian parameterized by ϵ, where ϵ is drawn from a Gaussian with unit variance. In that case, if we assume that p is also e.g. Gaussian, then it's easy to just directly evaluate the KL divergence term analytically. All the KL term does is penalize the autoencoder for the difference between $q_\phi(z|x)$ and a unit variance Gaussian for z. It just regularizes the magnitudes of (μ_ϕ, σ_ϕ).

7.4 Fake It 'Til You Make It (GANs)

Generative Adversarial Networks take a different approach – most importantly, they are not maximizing the log-likelihood of the data, or even estimating it explicitly.

Instead, these models use two networks, a generator and a discriminator. The generator maps from a latent space Z to the data space X, as usual. However, the discriminator's goal is to differentiate real data from generated data. Thus as training progresses, it learns ways in which the generator 'looks fake', and thereby 'teaches' the generator what 'mistakes' its making.

We can formalize this by defining networks $G_\theta(z)$ as the generator and $D_\phi(x)$ is the discriminator, where $D_\phi(x)$ produces a probability that x is real. In general, there are many possible choices of loss function. Once the loss functions are specified, we can iteratively update the discriminator and generator parameters during training.

For the generator we want a loss that's minimized when $G_\theta(z)$ performs well. One example is

$$L_G = -\mathbb{E}_z \log D_\phi \left(G_\theta(z)\right) \tag{7.4.1}$$

as this is minimized when G fools D all of the time. Another possibility mentioned in the original GAN paper is $+\log(1 - D_\phi(G_\theta(z)))$, but this has the problem that when $D(G(z)) \approx 0$, so that the discriminator is very confident the generator is a fake, there isn't much gradient signal. This is a problem of saturation.

For the discriminator we want a loss that's minimized when it can judge real from fake samples. If we view the 'data' as the real/fake labels and the 'model' as the discriminator's probability estimates, then a natural choice is the cross-entropy loss

$$L_D = -\mathbb{E}_x \log D_\phi(x) - \mathbb{E}_z \log \left(1 - D_\phi \left(G_\theta(z)\right)\right) \tag{7.4.2}$$

which (up to a constant) is the KL between the true real/fake distribution and the discriminator's predictions.

GAN Issues and Suggestions

The lore is that GANs are very hard to train, and that gain training is often unstable. There are several potential issues

- Training to win the GAN 'game' is qualitatively different from other loss functions. We are iteratively updating G_θ and D_ϕ, and they are in direct competition. Thus in theory *it's possible for the generator and discriminator to literally chase each other in circles.*
- A more common practical problem is that either the discriminator or generator can gain the upper hand and establish permanent dominance. Most likely the discriminator gets better than the generator, and the latter never learns to fool it at all.
- There's also a risk of *mode collapse*. Perhaps the generator simply learns to generate a few images which are near-identical to images in the training set, and never learns to do anything else. In this case the generator has found a trivial, useless solution to the game.

GANs are nevertheless used because they tend to generate impressively sharp, convincing images. The fake images with the greatest wow-factor have mostly been generated with GANs.

Techniques for improving and stabilizing GANs are hotly debated in the literature. Well-established advice from the first successful CNN GAN paper, 'DCGAN', suggests using all-convolutional architectures and batchnorm whenever possible. I'm not knowledgeable enough to dispense any further wisdom.

7.5 Improving the Latent Space

Can we make the latent space representations accord with our expectations for what information about the data should be important? For example, we might like to have a discrete variable that specifies which MNIST digit our generative model should make, and then many more ambiguous continuous variables representing other properties of the digit. Let's address this in the GAN context.

For this purpose, we will decompose the latent space into a 'code' vector c and a pure noise vector z, where c contains the information we would like to specify. Using a GAN generator, we would like to specify that $G(z, c)$. Then we will design a loss function that rewards the generator for associating c with the desired data properties.

Roughly speaking, what we would like is for the mutual information between the code c and the generated data $x = G(z, c)$ to be high. Or we want the posterior probability distribution $P_G(c|x)$ to have low entropy. But we don't know $P_G(c|x)$. Instead, we can use the variational principle to find a bound for it. We add a NN $Q_\phi(c|x)$ and we write

$$\begin{aligned} I(C; G_\theta(z, C)) &= S(C) - S(C|G_\theta(z, c)) \\ &= S(C) + \mathbb{E}_{x\sim G_\theta(z,c)} \left[\mathbb{E}_{c'\sim P_G(\cdot|x)} \log P(c'|x)\right] \\ &= S(C) + \mathbb{E}_{x\sim G_\theta(z,c)} \left[\mathbb{E}_{c'\sim P_G(\cdot|x)} \log Q_\phi(c'|x) + D_{KL}(P(C|x)||Q_\phi(C|x))\right] \\ &\geq S(C) + \mathbb{E}_{x\sim G_\theta(z,c)}\mathbb{E}_{c'\sim P_G(\cdot|x)} \log Q_\phi(c'|x) \end{aligned} \tag{7.5.1}$$

Here we can basically ignore the $S(c)$ as it's just a constant. This still isn't good enough because (apparently) we need to sample $c' \sim P_G(c|x)$. But actually

$$\mathbb{E}_{x\sim G_\theta(z,c)}\mathbb{E}_{c'\sim P_G(c|x)} \log Q_\phi(c'|x) = \mathbb{E}_{c\sim P(c)}\mathbb{E}_{x\sim G_\theta(z,c)} \log Q_\phi(c|x) \tag{7.5.2}$$

so we don't need the posterior. We can easily show this by noting

$$\begin{aligned} \int p(x) \int p(y|x) f(x, y) dy dx &= \int p(x, y) f(x, y) dx dy \\ &= \int p(y) p(x|y) f(x, y) dx dy \int p(x'|y) dx' \\ &= \int p(y) p(x|y) dx dy \int f(x', y) p(x'|y) dx' \end{aligned} \tag{7.5.3}$$

If we identify $c = y$ and $c' = y'$ in this algebra, we get the identify above.

This means that we can add

$$-\lambda \mathbb{E}_{c\sim P(c)}\mathbb{E}_{x\sim G_\theta(z,c)} \log Q_\phi(c|x) \tag{7.5.4}$$

to the GAN loss function with some coefficient λ, as minimizing this will maximize the lower bound on the mutual information between the code c and the data x that $G_\theta(z, c)$ generates. See 1606.03657 for details of the implementation.

8 Reinforcement Learning

Although I referred to UL as the 'final frontier' of learning, on a more practical level, RL actually differs more from SL than the techniques used in practice for UL. So in many respects RL is the most subtle and least intuitive area of ML.

Most pedagogical treatments of RL begin with the classical or 'tabular' setting, where all the configurations and actions that the agent encounters can be 'tabulated', and one can take a direct, deductive approach. I will forego this classical preamble and simply discuss NN based RL directly. That said, and as always – feel free to consult other sources[15] on this topic! My choice of presentation is non-standard.

To even discuss RL we need to introduce some foundational ideas. We are imagining that our ML algorithm is an **agent** in a world that can be specified in terms of **states** and **actions**. Both states and actions could be either continuous, or discrete, or a combination of the two. In virtually all cases we imagine that time has been discretized into time steps, where on each time step the agent has some information about the state, and can choose to take an action. Typically our agent moves about and acts until the end[16] of an **episode**, which might eg be a single Atari or Go game. The agent receives a **reward** after each timestep (sometimes it's always just 0 until the end of an episode) and a total accumulated **return**, and its goal is to choose actions that maximize the summed rewards from an episode.

Here are some famous degenerate limits of RL:

- Multi-Armed Bandit or Slot Machine: Trivial state space, discrete action space, (typically) stochastic rewards.
- Supervised Learning: Potentially enormous state space, but episodes consist of only a single step. There may be a large but finite action space, so that it's possible to explore all possible actions for each state. We'll discuss this comparison (of RL to SL) more later.
- Gridworlds: Finite state space, discrete actions, deterministic transitions.

There are several important distinctions that come up when we discuss RL. The first is a distinction between types of algorithms, where the extreme examples are **Policy Gradients** and **Q-Learning**, and there are many algorithms that interpolate between these two. In the case of policy gradients, we learn a NN policy $\pi_\theta : S \to A$ that produces a choice of action for any given state. The problem is then to find a way to update θ so that the policy improves and the agent gathers more and more reward as it trains. In contrast, in Q-learning we do not learn a policy, but an expected value $Q(s, a)$ associated with taking an action a in a state s. If we really knew the optimal value $Q(s, a)$ for all s, a then we could just choose a to maximize $Q(s, a)$ in a state s, and thereby derive an optimal policy. But if $Q(s, a)$ isn't optimal, we may make the wrong choice[17] of

[15] I particularly recommend spinning up: https://spinningup.openai.com/

[16] If we wish for the agent to exist forever, we might artificially break its infinite life into finite episodes for convenience.

[17] Really a Q should be associated with some policy, if Q is the expected reward rather than an optimal reward.

action. We interpolate between PG and Q-L because it's often good to have both a policy and some idea of the value to be gained by taking various actions.

The other, related distinction is between **on-policy** and **off-policy** learning. With the former, we only learn from the experience collected with our current policy. Vanilla PG is on-policy. With off-policy learning, we can use experience gained while acting according to one policy in order to learn about the consequences of another policy. Q-learning can be off-policy. Clearly off-policy learning could be more sample efficient, since we can use data collected long ago, when our policy was less developed. But off-policy data must be used carefully as it may be **stale**, and only relevant to some prior, less-developed policy.

Another distinction is **model-free** vs **model-based** RL. Most NN-based RL thus far has been model-free, meaning that the agents do not have or learn an explicit model of the 'world' in which they live. Many researchers are very hopeful about the prospects of model-based RL, where the agent does learn a model of the world and can therefore use it to make explicit plans.

Distinguishing between **continuous** vs **discrete action spaces** can be important when it comes to the way algorithms are implemented. With discrete actions we can directly enumerate them all, whereas with a continuous action space we can only ever sample. Robotics typically involves continuous action spaces, while many games only have discrete choices.

A conceptual mystery throughout machine learning is the problem of **attribution** – how do models know what features in the data are relevant to the decisions they must make? In SL this is the question of eg how does a model learn the features of images that come together to distinguish airplanes and lawnmowers. In RL the problem's much greater still, because the model somehow must discover which of its actions were relevant for success or failure, and which irrelevant. Ameliorating this attribution problem has been a major focus in RL research.

Finally, another perennial RL concern is **explore vs exploit** – should we simply act according our policy, or take other random (or otherwise chosen) actions in order to further explore the state space and perhaps find ever better strategies? The simplest way to explore more is to follow an **ϵ-greedy** policy, which just means that ϵ of the time we take a random action, while $1 - \epsilon$ of the time we take the action dictated by our policy. Often ϵ decreases according to some schedule during training. Other more sophisticated methods try to use curiosity, memory, or state-counting to motivate an agent to explore new states.

8.1 Notation

The reward an agent receives after timestep comes from a reward function, which in general may be

$$r_t = R(s_t, a_t, s_{t+1}) \tag{8.1.1}$$

though typically it may only depend on s_t or a_t. I'll write τ for a sequence of time steps $\{t\}$, and t, T for specific times. We are interested in the total return, which in general could take the form

$$R_\tau(T, \gamma) = \sum_{t=0}^{T} \gamma^t r_t \tag{8.1.2}$$

where γ is called the discount factor, and T is the horizon. We may take $T \to \infty$ or $\gamma \to 1$ to get an infinite time horizon or an undiscounted reward. Aside from convergence, another reason to include a finite horizon or a discount factor is (as we'll discuss later) to decrease the variance of the reward.

The probability distribution over visited states takes the form

$$P[\tau;\pi] = \rho_0(s_0)\prod_{t=0}^{T-1} P(s_{t+1}|s_t,a_t)\pi(a_t|s_t) \tag{8.1.3}$$

where π is the policy and $P(s_{t+1}|s_t,a_t)$ is the transition function from one state to the next. Very often $P(s_{t+1}|s_t,a_t)$ is very complicated, stochastic, and unknown. We included a ρ_0 distribution over initial states. The policy may be directly specified, or specified implicitly from a Q function. It can be stochastic (ie it gives a distribution over a) or deterministic; the latter is just a special case.

The expected return is

$$\begin{aligned} J[\tau;\pi] &= \int \mathcal{D}s_t\mathcal{D}a_t\, P[\tau;\pi]R(\tau) \\ &= \int \left(\prod_{t=0}^{T-1} ds_t da_t\right)\rho_0(s_0)\prod_{t=0}^{T-1} P(s_{t+1}|s_t,a_t)\pi(a_t|s_t)\left(\sum_{t=0}^{T}\gamma^t R(s_t,a_t,s_{t+1})\right) \\ &= \langle R(\tau)\rangle_{\tau\sim\pi} \end{aligned} \tag{8.1.4}$$

where for clarity I wrote it all out explicitly. We are searching for a policy π that maximizes J. This function is closely related to a more general on-policy **value function**

$$V^\pi(s) \equiv \langle R(\tau)\rangle_{\tau\sim\pi,s_0=s} \tag{8.1.5}$$

starting in state s and acting according to a policy π (so it depends crucially on π). We often write $V^*(s)$ as the optimal value function starting in state s, ie the expected value if one were to use the optimal policy. We often refine this to the Q function

$$Q^\pi(s,a) \equiv \langle R(\tau)\rangle_{\tau\sim\pi,s_0=s,a_0=a} \tag{8.1.6}$$

which gives the value of s,a as the initial state and action. There is also a Q^*, the optimal state-action value function. Typically for us all value and policy functions will be parameterized by NNs.

8.2 Policy Gradients

PGs are the way that we learn a policy function $\pi_\theta : S \to A$. But why the term 'gradient'? Because we won't actually learn from a legitimate policy loss; rather we will identify a simple surrogate objective whose *gradient* is equal to the gradient of performance, even though it's not actually the quantity that we want to minimize globally.

To derive our surrogate loss, we simply note that

$$\begin{aligned}\nabla_\theta J[\pi_\theta] &= \int \mathcal{D}s_t \mathcal{D}a_t\, \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t) \nabla_\theta \pi_\theta(a_t|s_t) R(\tau) \\ &= \int \mathcal{D}s_t \mathcal{D}a_t\, P[\tau;\pi] R(\tau) \left(\nabla_\theta \log \pi_\theta(a_t|s_t)\right) \\ &= \langle R(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t) \rangle_{\tau\sim\pi} \end{aligned} \tag{8.2.1}$$

so our surrogate loss

$$L = -R(\tau) \log \pi_\theta(a_t|s_t) \tag{8.2.2}$$

has a gradient equal to that of (minus) the expected return. Notice that the second derivative of the surrogate loss L is not equal to the second derivative of the expected return J; this is why our algorithm computes 'policy gradients', and gradients only!

Since this is an expectation over trajectories sampled from π_θ (and the typically unknown transition function), we can estimate it by collecting a sample of taking the mean. That is our gradient is

$$\nabla_\theta J[\pi_\theta] \approx \frac{1}{B} \sum_{i=1}^{B} \sum_{t=0}^{T} R(\tau_i) \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{8.2.3}$$

for a batch of B trajectories.

Note that here in RL-land, and unlike in supervised learning, *the data distribution depends on the policy itself, meaning that the data we see depends on the NN parameters θ*. Also, as previously emphasized, we have only computed the *gradient of the returns from the policy, and so our surrogate loss does not provide a measure of performance*. It is not necessarily good for the surrogate loss to go down!

Rewards-to-Go and Baselines

There are two elementary but important improvements we can make to the Vanilla PG above. These **reduce the variance** of the policy gradient without altering its mean; this means that they should improve the sample efficiency of RL. The first is to simply subtract a **baseline** $b(s_t)$ from the return $R(\tau)$. The second is to trade total return at a given time for the returns-to-go, ie the returns from the future. Intuitively, it seems obvious that we should not need to include rewards from early states at time $t' < t$, as these cannot be effected by actions at time t.

To show that rewards $R(s_{t'}, a_{t'}, s_{t'+1})$ for $t' < t$ can be dropped, it suffices to show that these terms are actually zero in expectation. That is we want to study

$$f(t, t') \equiv \nabla_\theta \log \pi_\theta(a_t|s_t) R(s_{t'}, a_{t'}, s_{t'+1}) \tag{8.2.4}$$

and show that $\langle f(t,t')\rangle_{\tau\sim\pi_\theta} = 0$ for $t' < t$. Note that this won't be true if $t' > t$. To show this, what we really care about is

$$\begin{aligned}\langle f(t,t')\rangle_{\tau\sim\pi_\theta} &= \langle f(t,t')\rangle_{s_t,a_t,s_{t'},a_{t'},s_{t'+1}} \\ &= \int ds_t da_t ds_{t'} da_{t'} ds_{t'+1} f(t,t') P(s_t,a_t|\pi_\theta,s_{t'},a_{t'},s_{t'+1}) P(s_{t'},a_{t'},s_{t'+1}|\pi_\theta) \\ &= \int dS_{t'} P(s_{t'},a_{t'},s_{t'+1}|\pi_\theta) R(S_{t'}) \int dS_t \nabla_\theta \log\pi_\theta(a_t|s_t) P(S_t|\pi_\theta,S_{t'})\end{aligned} \tag{8.2.5}$$

where in the last line I wrote $S_t = \{s_t, a_t\}$, $S_{t'} = \{s_{t'}, a_{t'}, s_{t'+1}\}$ for notational concision. The inner part of the last expression then gives

$$\int ds_t da_t \nabla_\theta \log\pi_\theta(a_t|s_t) P(S_t|\pi_\theta, S_{t'}) = 0 \tag{8.2.6}$$

because its the gradient of a normalized probability. This last step wouldn't be possible if $t' > t$ because then we could not factor the probability using a conditional $P(S_t|\pi_\theta, S_{t'})$.

So now we need only use the reward-to-go from any given time step until the end of the episode. We also want to show that we can take $R_{t>t_0} \to R_{t>t_0} - b(s_{t_0})$ without changing the expectation of the policy gradient. This is even simpler, as we need only note that

$$\langle \nabla_\theta \log\pi_\theta(a_t|s_t) b(s_t)\rangle_{a_t\sim\pi_\theta} = 0 \tag{8.2.7}$$

since we are dealing with a normalized probability distribution over a_t.

In total, these developments imply that instead of the total return we can use the baseline-subtracted rewards to go

$$\sum_{t'=t}^{T} R(s_{t'},a_{t'},s_{t'+1}) - b(s_t) \tag{8.2.8}$$

A typical $b(s_t)$ to use is a learned value function $V^\pi(s_t)$ that uses a NN to predict the returns; it's typically trained using a least squares loss for the difference between V^π and the returns.

It's also worth noticing that

$$Q^\pi(s_t,a_t) \equiv \left\langle \sum_{t'=t}^{T} R(s_{t'},a_{t'},s_{t'+1}) \right\rangle_{\tau\sim\pi_\theta} \tag{8.2.9}$$

by definition. Sometimes the **advantage function**

$$A^\pi(s_t,a_t) = Q^\pi(s_t,a_t) - V^\pi(s_t) \tag{8.2.10}$$

is used in place of the return, but really this is equivalent in expectation to using the returns-to-go minus a baseline.

Generalized Advantage Estimation

Typically we don't really want (conceptually) to discount rewards. A better way to think about the discount factor γ is as a trade between bias and variance. Since the full rewards-to-go from the future can have very high variance, we can get a stabler policy gradient signal if we discount effects from the far future. Some large scale applications of RL even anneal $\gamma \to 1$ as training progresses.

With this in mind, one can study generalized advantage functions

$$\begin{aligned} A_t^{(1)} &= -V(s_t) + r_t + \gamma V(s_{t+1}) \\ A_t^{(2)} &= -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \end{aligned} \tag{8.2.11}$$

etc. The choice $A_t^{(1)}$ has low variance but high bias, while $A_t^{(k)}$ can have low bias but high variance. The bias decreases with increasing k because it is $V(s_{t+k})$ which biases our estimate of the advantage, and this term is becoming more and more suppressed. $A_t^{(\infty)}$ is literally just the empirical return minus a baseline. A 'Generalized Advantage Function' is

$$A_t^{GAE(\gamma,\lambda)} = (1-\lambda)\sum_{k=1}^{\infty} \lambda^{k-1} A_t^{(k)} \tag{8.2.12}$$

which allows us to balance bias and variance with a continuous parameter λ.

8.3 Q-Learning

The other line of thinking in RL is based on Q-learning, ie learning and improving the $Q(s,a)$ function. To learn Q, we use recursive self-consistency equations called the **Bellman equations**, which take the form

$$Q^\pi(s,a) = R(s,a) + \gamma \left\langle Q^\pi(s',a') \right\rangle_{a'\sim\pi, s'\sim P(\cdot|s,a)} \tag{8.3.1}$$

where states s' are sampled from the transition function, and actions a' from the policy. Here γ is the discount factor. There is also a Bellman equation for the optimal Q^*

$$Q^*(s,a) = R(s,a) + \gamma \max_{a'} \left\langle Q^*(s',a') \right\rangle_{s'\sim P(\cdot|s,a)} \tag{8.3.2}$$

Note that optimal policy may be degenerate. The RHS of these equations are very frequently referred to as a 'Bellman backup', or perhaps a one-step (in time) Bellman backup.

To learn $Q^*(s,a)$ as a NN, one can just compute the mean squared error of the Bellman equation, and then perform gradient descent. Note that for this purpose we stop the gradients on the Q^* on the RHS. Thus we can just go back and forth collecting experience based on Q^* and then performing gradient descent to improve our estimation of Q^*.

It's worth noting that *the Q-learning objective isn't what we really care about*, ie even the gradient of Q-learning is not pointing us towards a better policy, but only towards a more accurate knowledge of Q for the policy dictated by Q. Lore/experience suggests that Q-learning is more brittle and harder to use than PGs, though it can (sometimes) be more sample-efficient.

Bias and Double Q

Our Q function update rule introduces a subtle bias – if there are errors in Q that tend to make $Q(s,a)$ too large for a given a, then these will also cause us to select for that a in the maximum. We will then tend to over-estimate $Q(s,a)$, because we only see upward fluctuations when we update. A common trick to improve Q-learning is to use two Q functions, a standard one and a 'target' Q function. Then we update the target by copying parameters from the standard Q, but we only do this infrequently. This helps to avoid this subtle bias.

'DDPG' and Continuous Action Spaces

The Q-learning procedure we have outlined may be difficult to apply to continuous action spaces, because we need to compute a maximum over actions. We can easily remedy this by learning a policy for $a = \pi(s)$, where we simply train π to find the argmax of $Q(s,a)$. This procedure, where we learn both Q and π, is called 'DDPG'. The name 'Deep Deterministic Policy Gradients' is sort of unfortunate, meaningless, and misleading; really this is Q-learning even though it has PG in the name.

Distributional RL

There are many, many other modifications of Q-learning designed to make it learn more efficiently or robustly. One simple modification that seems to have significant empirical support is to learn the *probability distribution of rewards* rather than the expected reward given this distribution. But as yet I don't think there's a satisfying theoretical explanation for why distributional RL works so well.

Intuition from Tabular Setting and Dynamic Programming

Imagine that there are a finite number of states and actions, and that transitions are deterministic. In that case $Q(s,a)$ can simply be a lookup table of values... and so this is called the tabular setting.

Here we can solve the Q-learning problem exactly in finite time. The Bellman equation

$$Q^*(s,a) = \mathcal{T}Q(s,a) \equiv R(s,a) + \gamma \max_{a'} \left\langle Q^*(s',a') \right\rangle_{s' \sim P(\cdot|s,a)} \tag{8.3.3}$$

still applies, but now it's a rule for *refreshing entries in the lookup table*. And with a finite number of states and actions (and deterministic transitions), this process is guaranteed to converge in a finite time. Conceptually, this should be intuitive with a bit of thought – each application of $\mathcal{T}$ propagates us through the finite list of states, taking all possible actions, so in a finite number of steps we cover them all, and discover the proper values. The algorithm that simply replaces $Q \to \mathcal{T}Q$ iteratively is called **Value Iteration** in the RL literature.

To formally demonstrate the convergence of value iteration, we want to show that the Bellman backup operator $\mathcal{T}$ is a *contraction mapping* (in the infinity norm). So we are interested in showing that

$$\max_{s,a} |\mathcal{T}X(s,a) - \mathcal{T}Y(s,a)| \leq \gamma \max_{s,a} |X(s,a) - Y(s,a)| \tag{8.3.4}$$

since this means that acting with $\mathcal{T}^n$ will converge as $n \to \infty$. To see this

$$\begin{aligned} \max_{s,a} \langle |\max_{a'} X(s',a') - \max_{a'} Y(s',a')| \rangle_{s' \sim P(\cdot|s,a)} &\leq \max_{s} |\max_{a'} X(s,a') - \max_{a'} Y(s,a')| \\ &\leq \max_{s,a'} |X(s,a) - Y(s,a)| \end{aligned} \tag{8.3.5}$$

In the last step note if WLOG $\max f > \max g$ then we have

$$\max_x f(x) - \max_x g(x) = f(x^*) - \max_x g(x) \leq f(x^*) - g(x^*) = \max_x |f(x) - g(x)| \tag{8.3.6}$$

This demonstrates that our learning process will converge (note that this argument also works with non-deterministic transitions).

There is another algorithm called **Policy Iteration** where one defines a policy π, computes its associated value function V^π, then improves the policy using that value function, then iterates until the policies stop changing. This is typically faster than value iteration (because we only use the present policy to identify the relevant associated value functions, not the optimal value function).

Value and Policy Iteration are essentially identical to 'Dynamic Programming', a more generally applied algorithmic method. Perhaps the most famous dynamic programming problem is: given a set of items with a weight w and value v, determine the collection of items with largest total value given the constraint that the total weight must be less than some fixed, given bound. This is called the *Knapsack Problem*. It is identical to a tabular RL problem where actions correspond to adding something to the backpack, and states correspond to states of the backpack. Some actions are not be available in some states because there's only one of each item.

8.4 Brief Comparison of Supervised Learning to RL

If we like, we can turn any SL task into an RL task. This provides a bit of intuition.

Imagine we have a task where there's a model $p_\theta(y|x)$ to be learned. We could imagine that y is continuous or discrete, but since so many tasks involve discrete labels, we'll assume that $y = 1, 2, \cdots, n_L$, and that x is some high dimensional vector, eg an image. The loss function for SL will then be a cross-entropy (KL without ground truth entropy)

$$L_{\text{SL}} = -\sum_{y=1}^{n_L} R(y|x) \log p_\theta(y|x) \tag{8.4.1}$$

where the most relevant case is the function $R(y|x) = 1$ for a single y and 0 otherwise (though other possibilities would apply if we eg did label smoothing). When we perform SL, we simply differentiate with respect to θ. Let's compare this to RL.

Policy Gradient

When we reformulate this in the PG setup, we can use an identical 'policy' network $p_\theta(y|x)$, where we view y as an action, and we define the policy

$$\pi_\theta(x) = \text{argmax}_y \; p_\theta(y|x) \tag{8.4.2}$$

Furthermore, the reward for that action is literally $R(y|x)$. So the difference between SL and RL is that in the RL case, we only see $R(\pi_\theta(x)|x)$ on each iteration, whereas with SL we always see which label was the correct one. The policy gradient surrogate loss is

$$L_{\mathrm{RL}} = -R(\pi_\theta(x)|x) \log p_\theta(\pi_\theta(x)|x) \tag{8.4.3}$$

This means that *with a random policy*

$$\langle \nabla_\theta L_{\mathrm{SL}} \rangle = \langle \nabla_\theta L_{\mathrm{RL}} \rangle_{\pi_{\mathrm{random}}} \tag{8.4.4}$$

However, the RL policy gradient is mostly going to be zero on a random policy; it will only give a signal on average for a fraction $1/n_L$ images. Thus we should expect that early in training, RL will simply be equivalent to SL but with a data-efficiency penalty of $1/n_L$.

But later in training, the policy will have better performance, and so it will receive a larger fraction of signal. But this will mean that the SL and RL gradients will not be equivalent. More specifically

$$\nabla_\theta L_{\mathrm{SL}} = -\sum_{y=1}^{n_L} R(y|x) \nabla_\theta \log p_\theta(y|x) \tag{8.4.5}$$

while

$$\begin{aligned} \nabla_\theta L_{\mathrm{RL}} &= -\sum_{y=1}^{n_L} p_\theta(y|x) R(y|x) \nabla_\theta \log p_\theta(y|x) \\ &= -\sum_{y=1}^{n_L} R(y|x) \nabla_\theta p_\theta(y|x) \end{aligned} \tag{8.4.6}$$

So the SL and RL gradients are quite different – SL maximizes likelihood (products of probabilities), while RL maximizes the accuracy, ie the sum of the probabilities.

Note that with some ϵ-greediness we have a guarantee that some part of the signal will always imitate SL. And in the limit $\epsilon \to 1$, so that the choice of classification labels is random, we find that RL turns back into SL with a factor of n_L decrease in sample efficiency. So *randomized RL looks like n_L-times less efficient SL. The policy gradient procedure, which maximizes accuracy, will likely be more sample efficient than this, but may the price due to 'poor exploration'.* That is, we could simply never guess the label 'cat' for our images, and never learn that cats exist!

Q-Learning Version

We could also just use a Q-learning type loss

$$L_{\mathrm{Q}} = \frac{1}{2} \left(R(y|x) - p_\theta(y|x) \right)^2 \tag{8.4.7}$$

where we re-interpret $p_\theta(y|x)$ as the Q-function, since R is always 0 or 1. Here each sample (x_i, y_i) that we collect provides some useful data. And because the 'game tree' is trivial, this data *never*

becomes stale (assuming we store the image as the state, the chosen y as the action, and the R value, though this will soon lead to storing the full dataset!). But when our policy isn't very good, we typically just learn what a given image isn't, rather than what it is, so we'd expect that early on learning is slowed down by a factor of n_L. It's somewhat interesting that later on, when we discourage bad results, we'll always be focusing on discouraging the next-most-likely wrong classification.

8.5 Don't Walk Off a Cliff: TRPO and PPO

A potential problem with Policy Gradients is that they may update the policy too aggressively, and in unpredictable ways. Unlike in SL where we are trying to minimize a loss that we have in hand, with PG we really only have the gradient, so it's hard to know how large a step we should be taking. (Note that both methods discussed in this section can be freely combined with methods for variance reduction, ie the inclusion of learned value functions and the use of an advantage in place of the return. Real-world applications will almost always use such methods, but they are conceptually orthogonal to the value-add of these algorithms.)

TRPO

In our discussion of optimization, we already covered methods that should greatly mitigate this problem – natural gradients. To explain why... note that it would be eminently reasonable to impose a constraint that the updated policy shouldn't differ very much from the original policy. But 'differ much' should have an invariant meaning! One way to make it so is to constrain

$$D_{KL}(n+1||n) \equiv \langle D_{KL}\left(\pi_{\theta^{(n+1)}}(\cdot|s)||\pi_{\theta^{(n)}}(\cdot|s)\right)\rangle_{s\sim\pi_{\theta^{(n)}}} \tag{8.5.1}$$

where the expectation just means we're averaging over states actually visited by the prior policy. But if we compute the update as the policy gradient $\nabla_\theta J[\pi]$ subject to a constraint on the KL, that directly means that infinitesimally we should just update

$$\theta^{(n+1)} = \theta^{(n)} - \epsilon F^{-1}\nabla_\theta J \tag{8.5.2}$$

where F is the Fisher information matrix, ie the 2nd derivative of the $D_{KL}(n+1||n)$. The TRPO algorithm says that this is (almost) how we should update parameters. The only addition is that TRPO adds a back-tracking line search, to literally ensure that the D_{KL} constraint is actually bounded by some fixed hyper-parameter.

PPO

A practical problem with TRPO is that it requires computing F^{-1}, ie it's a second order method. PPO, which actually comes in various versions (and includes several other practically helpful but theoretically not-super-well-justified tricks), solves the problem of taking too-large steps without requiring the evaluation of F^{-1}. All of these methods are less theoretically elegant than TRPO, but they end up performing as well or better empirically, it seems.

There's an obvious way to try to accomplish what we did with TRPO – we can just add $D_{KL}(n+1||n)$ to the PG loss with some coefficient, in order to penalize our model for changing the policy too quickly. The problem with this is that it's then hard to determine the right coefficient for D_{KL}. So this version of PPO must also come with a dynamical mechanism to scale the coefficient during training and keep it in a healthy range.

Another idea is just to clip the gradients (actually, set them to zero here) when they become too large. Thus PPO uses a modified loss function

$$L\left(s,a,\theta^{(n)},\theta\right) = -\min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta^{(n)}}(a|s)}A^{\pi_{\theta^{(n)}}}(s,a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta^{(n)}}(a|s)}, 1-\epsilon, 1+\epsilon\right)A^{\pi_{\theta^{(n)}}}(s,a)\right) \quad (8.5.3)$$

where $A^{\pi_{\theta^{(n)}}}$ is the advantage. This looks complicated, but it's really just dealing with a few different cases, where the advantage and/or the change in policy is positive or negative. First, note that the ratios are equivalent (for the gradient computed at $\theta^{(n)}$) to just using $\log \pi_\theta$. If the ratio is very close to 1, then we just use the first term and get the usual PG. If the ratio is large, then we can end up using the second term, which clips the loss, and therefore sets the gradient to zero. The signs work out because when the advantage is positive we want the policy to increase the probability of taking the associated action, whereas when the advantage is negative we want to update the policy in the opposite direction.

PPO implementations include a lot of other details. For instance, they usually use a small replay buffer and take several steps using mini-batches drawn from this buffer before refreshing it (note that the clipping above has no effect on the first step, since it's the loss, not the gradient that is clipped!). They also typically perform a kind of 'batch norm' on the advantages, dividing them by the standard deviation within a batch of rollouts (presumably to keep gradient magnitudes of roughly fixed normalization to avoid big steps). If you look in detail at various versions and implementations of PPO, they may handle these and other subtleties differently.

8.6 AlphaZero and Self-Play

AlphaGo and AlphaZero were one of the most impressive examples of ML in the last few years; these systems play chess and Go far better than the best humans, and also better than the best prior AI (Stockfish, for Chess). These systems combine what we discussed above with two new ideas: **self-play** and **Monte Carlo Tree Search**.

The power of self-play as a natural training curriculum may be the most important lasting lesson from AlphaZero. And it's very easy to describe – we simply have multiple versions of our ML game-playing agent play against itself over and over again to generate experience. There are a variety of more or less sophisticated ways to do this, from simply playing a single version of the agent over and over with different random seeds, to including a whole 'ecosystem' of agents at various levels of abilities. But AlphaZero simply maintains a single network that plays itself, rather than an ecosystem.

AlphaZero uses a neural network that computes $f_\theta(s) = (\vec{p}, V)$ where $\vec{p}$ are the probabilities of selecting each move – so this is a policy network – and V is a value function, estimating the win probability. The network itself is just a ResNet CNN with ReLU non-linearities and batch norm.

MCTS

MCTS uses the neural network to explore a large number of possible future moves, and then deduce refined (ie better than the network by itself) evaluations of move probabilities. As it plays it stores data (s_t, π_t, r_t) including the state, the MCTS-predicted probabilities π_t, and the win/loss (ie reward) of the game r_t. The NN training loss then involves a least-squares error term for the value function and a cross-entropy term for the policy, so that

$$L = (r - v)^2 - \sum_a \pi_s(a) \log p(a|s) \tag{8.6.1}$$

where the policy $p(s, a)$ gives the probability of taking an action a in the state s. Thus the policy network is trained to approximate the MCTS rollouts, which are more intelligent, as they involve planning. Note that this is only possible because the game dynamics are known and deterministic, ie in this case we can really do **model-based RL**.

But how do we compute $\pi_s(a)$, the MCTS result? The search involves two functions, an action value $Q(s, a)$ and a 'confidence' term $U(s, a)$. Both are determined through the search process. We define

$$U(s, a) \propto c_{\text{puct}} p(a|s) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \tag{8.6.2}$$

where $N(s, a)$ is a visit-count tracker that we update during the MCTS. The c_{puct} is a hyper-parameter. Thus $U(s, a)$ decreases as we visit more and more nodes that follow from the node (s, a), so that we become increasingly confident of our determination of the value of (s, a). The $Q(s, a)$ function is defined to be

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{s'|s,a\to s'} V_\theta(s') \tag{8.6.3}$$

where V_θ is the value function of the NN. Thus $Q(s, a)$ is determined by looking ahead and seeing what game-state values $V_\theta(s')$ are achieved by following the most likely (well-played) paths from a given (s, a), and averaging over them.

Our decision about which states to explore is determined by the sum of these functions, so at any given time during MCTS we choose

$$a = \text{argmax}_a \left[Q(s, a) + U(s, a) \right] \tag{8.6.4}$$

as the next state to explore.

The final $\pi_s(a)$ is then simply assigned as

$$\pi_s(a) = \frac{N(s, a)^{1/\tau}}{\sum_b N(s, b)^{1/\tau}} \tag{8.6.5}$$

where τ is a temperature hyper-parameter. So we try to make moves that take us down game trees that we visited a lot, and we visit game trees based on our expectation of their value.