Deep Learning: A Mathematical Overview

Prof. Anne Gelb
Dartmouth College
Mathematics

Sriram Bapatla
Dartmouth College
Mathematics

Yusuf Olokoba
Dartmouth College
Computer Science

Jack Zhang
Dartmouth College
Mathematics

December 3, 2019

Abstract

In this paper, we perform an overview of deep learning as it is today from a mathematical perspective. We start with the fundamentals of learning, then go on to analyze how different tasks are learnt, what data is used to learn such tasks, and how to better understand said data. This paper is by no means comprehensive. It is intended to simply provide be a high level overview of some of the mathematical foundations of deep learning.

Contents

1 Neural Networks: An Overview

At the highest level, neural networks are function estimators. Given a set of inputs $\{x_1, \ldots, x_n\}$ and a corresponding set of outputs $\{y_1, \ldots, y_n\}$, a neural network *learns* the model $f(x)$ which best approximates the given data $f(x_i) = y_i$.

The simplest neural network to consider is a linear regressor, which learns the w^*, b^* that best fits some given linear data $f(x_i) = w^* x_i + b^* = y_i$. Unfortunately, your linear regressor will not spark a revolution because most observable data we consider are non-linear. But thankfully with only a few additions, we can in fact build highly sophisticated models which learn highly complex relationships in data. To do so, we introduce the *Perceptron*:

$$f(x) = \sigma(wx + b)$$

Where σ is the *activation function*, w is the weight, and b is some bias. The activation function $\sigma(\cdot)$ is used to enable the perceptron to learn non-linear features, depending on the choice of the activation. The rest of the formulation is simply a linear regressor.

You might be thinking that a single perceptron would still not be able to learn a large variety of relationships, and you would be correct. Although there is active debate on the topic, it has been emperically shown that increasing the number of perceptrons in a model increases its *capacity* [1], that is its ability to learn more complex relationships. Hence, we introduce the Multi-Layer Perceptron (MLP):

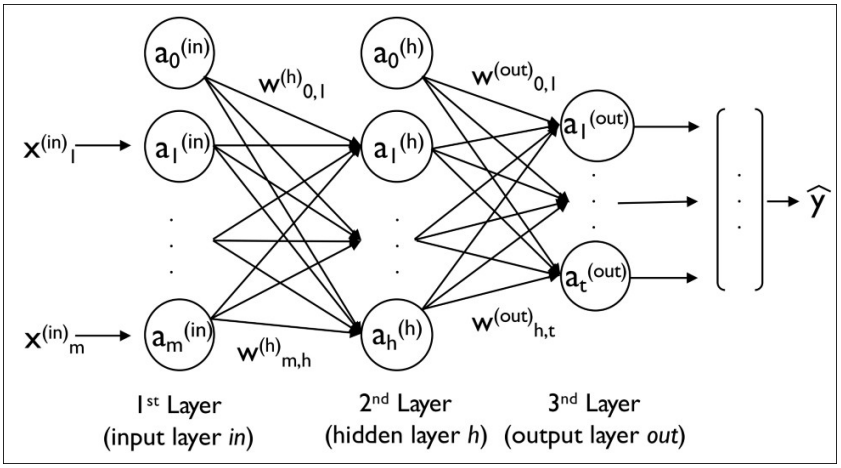

Figure 1.1: Multi-Layer Perceptron. Source

The MLP forms the basis of deep neural network topologies. We will discuss two notable topologies: autoencoders and convolutional neural networks.

1.1 Autoencoders and Representation Learning

Autoencoder models are trained in *unsupervised learning tasks*, where the objective is to reproduce some input $\vec{x}$ as closely as possible, using less information than that of the provided input.

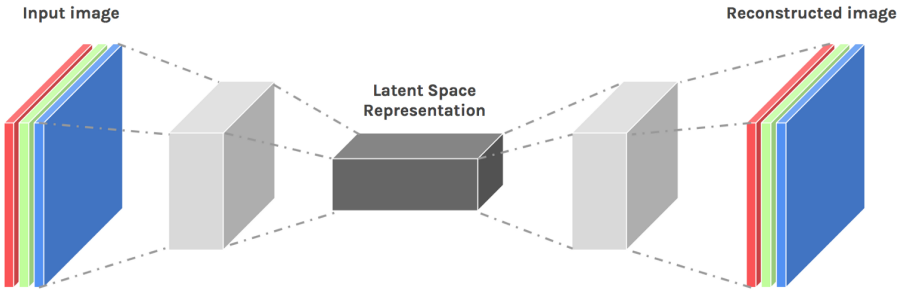

Figure 1.2: Autoencoder architecture. Source

An autoencoder is comprised of two sub-networks, an *encoder* which learns a significantly lower-dimensional feature space for its input, and a decoder which learns a transformation from the learned feature space back to the input domain, while preserving as much useful information as possible.

One way to consider autoencoders is as a non-linear principal component analysis over the input space. They are extremely useful for compression tasks, but also for representation learning.

In representation learning, the task is to learn not only the most defining feature of a given distribution, but also the [principal] factors of variation within the distribution. This is extremely useful for generative tasks, where one might want to generate completely new samples by interpolating between certain features within the distribution. Disentanglement and causal inference are significant [next steps] in representation learning.

1.2 Convolutional Neural Networks

Convolutional Neural Networks ('CNN' hereafter) are architected for feature extraction from images. Computers store images as stacks two-dimensional matrices where elements correspond to pixel intensity at their spatial locations within the image. The task of extracting features from an image requires special [precautions] due to the fact that certain features, say a cat face, might occur at several distinct regions within a given image. Hence the network would require feature extraction units that are spatially-

invariant. To do so, we utilize the discrete convolution:

$$I(x,y) \circledast K(\cdot,\cdot) = \sum_{i=1}^{k}\sum_{j=1}^{k} I(x+i, y+k) \cdot K(k-i, k-j)$$

Where $I(x,y)$ is an image indexed at positions (x,y) and $K(\cdot,\cdot)$ is a learned convolution kernel. During training, the CNN learns a series of kernels $\{K_1(\cdot,\cdot), \ldots, K_n(\cdot,\cdot)\}$ which respond to useful features for the task being learnt.

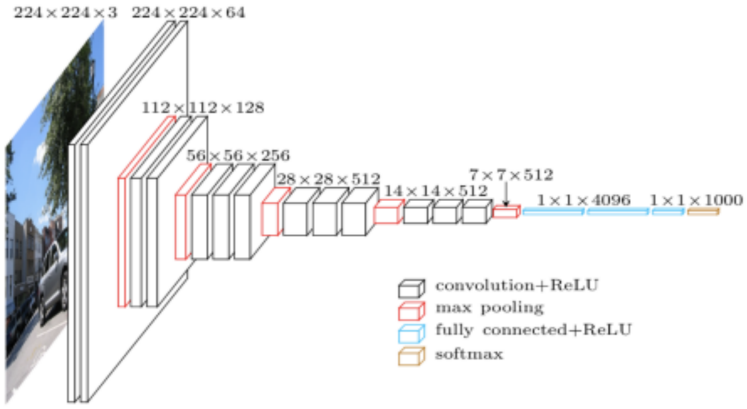

Figure 1.3: CNN architecture. Source

This design is inspired by the mamallian visual cortex [2], which recognizes features using an ensemble of feature extractors. Earlier kernels learn to recognize the simplest features like lines at different orientations whereas later kernels learn more complex representations, like cat faces, as the combinations of the simpler learned kernels.

1.3 Other Notable Architectures

Some other notable architectures include Recurrent Neural Networks (RNN) and Long-Short Term Memory (LSTM) models. These architectures are targeted for learning time-series data, where previously-encountered data might influence the learnt features of new data. They are usually used for language modeling problems.

2 Learning in Neural Networks

Given a neural network and some training data, how does the network learn the data? The objective is for the network to learn a model that best fits the data. To reach this objective, we need two things: a notion of how well the current model performs (covered in more detail in chapter 3), and a way to tune the current model based on its evaluated performance to yield some new model. The latter problem falls into the broad category of numerical optimization methods.

2.1 1st Order Optimization

Consider an arbitrary neural network $f(\vec{p})$ with of n parameters which is trained to produce outputs $\hat{y}$ that minimize some cost function $C(\vec{p})$:

$$\hat{y} = f(\vec{p})$$
$$C(\vec{p}) = |\hat{y} - y|$$

Our model learns the parameters $\vec{p} = \{w_1, b_1, \ldots\}$ which defines a model that best fits the given data $\{y_1, y_2, \ldots\}$. We wish to iteratively update $\vec{p}$ to minimize $C(\vec{p})$. Taking a first order Taylor expansion of C at $\vec{p}$, we can estimate $C(\vec{p}+\Delta\vec{p})$ for a small perturbation $\Delta\vec{p}$:

$$C(\vec{p} + \Delta\vec{p}) \approx C(\vec{p}) + \Delta\vec{p}^T \nabla C(\vec{p})$$
$$\nabla C(\vec{p}) = \left[\frac{\delta C(\vec{p})}{\delta \vec{p}_1}, ..., \frac{\delta C(\vec{p})}{\delta \vec{p}_s}\right]^T$$

Hence:

$$C(\vec{p} + \Delta\vec{p}) < C(\vec{p}) \text{ if } \Delta\vec{p}^T \nabla C(\vec{p}) < 0$$

It is apparent that $\Delta\vec{p} \cdot \nabla C(\vec{p})$ is maximized when $\Delta p = \nabla C(\vec{p})$. Hence we arrive at the gradient descent update scheme:

$$\vec{p}^{(t+1)} = \vec{p}^{(t)} - \eta \nabla C(\vec{p}^{(t)})$$

We call η the *learning rate*, and keep it at a small value since we are merely approximating ∇C.

2.1.1 Backpropagation

Given the gradient descent update scheme, we need to compute $\nabla C(\vec{p})$. Due to the topology of the neural network, we can analytically compute it. Consider a toy artificial neural network, $N = \{a^{(1)}, \ldots, a^{(L)}\}, \forall a^{(i)} \in N, a^{(i)} \in \mathbb{R}$. At each layer, we have:

$$a^{(i)} = \sigma^{(i)}(z^{(i)})$$
$$z^{(i)} = w^{(i)}a^{(i-1)}$$

Where $\sigma^{(i)}$ is an activation function, usually one of $tanh$, $sigmoid$, $ReLU$, $softmax$, and so on. We ignore the bias term for illustration purposes. Now suppose we train our network on a single example y, we have:

$$\begin{aligned}
C &= |a^{(L)} - y| \\
\frac{\delta C}{\delta w^{(L)}} &= \frac{\delta C}{\delta a^{(L)}} \cdot \frac{\delta a^{(L)}}{\delta z^{(L)}} \cdot \frac{\delta z^{(L)}}{\delta w^{(L)}} \\
\frac{\delta C}{\delta w^{(L-1)}} &= \frac{\delta C}{\delta a^{(L)}} \cdot \frac{\delta a^{(L)}}{\delta z^{(L)}} \cdot \frac{\delta z^{(L)}}{\delta a^{(L-1)}} \cdot \frac{\delta a^{(L-1)}}{\delta z^{(L-1)}} \cdot \frac{\delta z^{(L-1)}}{\delta w^{(L-1)}} \\
&\ldots \\
\frac{\delta C}{\delta w^{(1)}} &= \frac{\delta C}{\delta a^{(L)}} \cdot \frac{\delta a^{(L)}}{\delta z^{(L)}} \cdot \frac{\delta z^{(L)}}{\delta a^{(L-1)}} \cdot \ldots \cdot \frac{\delta z^{(2)}}{\delta a^{(1)}} \cdot \frac{\delta a^{(1)}}{\delta z^{(1)}} \cdot \frac{\delta z^{(1)}}{\delta w^{(1)}}
\end{aligned}$$

As we continue to unwind this, we work our way backward from the output layer $a^{(L)}$ to the input layer $a^{(1)}$. This defines the backpropagation technique to compute $\nabla C(\vec{p})$.

Scaling this up to several input samples $\{y_1, ..., y_n\}$ simply involves avergaing the computed gradients over each y_i. Similarly, scaling this to fully connected networks involves taking the sum of the computed partials over each input neuron. Consider a neuron i in a given layer l, $a_i^{(l)}$. We have:

$$\frac{\delta C}{\delta a_k^{(l)}} = \sum_{j=1}^{n_L} \frac{\delta C}{\delta a_j^{(l+1)}} \cdot \frac{\delta a_j^{(l+1)}}{\delta z_j^{(l+1)}} \cdot \frac{\delta z_j^{(l+1)}}{\delta a_k^{(l)}}$$

Special care must be taken when choosing an activation function $\sigma^{(i)}$, because:

$$\frac{\delta a^{(l)}}{\delta z^{(l)}} = \sigma'(z^{(l)})$$

Now consider the *tanh* and *sigmoid* activation functions. They have a desirable property in that they *squish* their activations to a predictable range, and their gradients around the origin are also of reasonable magnitude. The problem is that at extrema, these activations produce gradients with magnitude ≈ 0. It becomes apparent from the above equations that once this happens, the model becomes unable to learn as no gradient

magnitude gets backpropagated to earlier layers.

Nowadays, we tend to use Parametric Rectified Linear Unit (PReLU) activation which has a fixed negative gradient:

$$\sigma(x) = \begin{Bmatrix} x & \text{when } x \geq 0 \\ \alpha x & \text{otherwise} \end{Bmatrix}$$
$$\sigma'(x) = \begin{Bmatrix} 1 & \text{when } x \geq 0 \\ -\alpha & \text{otherwise} \end{Bmatrix}$$

And in order to retain the nice property of reasonable gradient magnitudes that *sigmoid* and *tanh* have, we introduce layers that normalize the activations over a batch [3] or instance [4] to have a given or learned mean and standard deviation.

Another problem similar to vanishing gradients is exploding gradients. This is usually as a result of training data with skewed covariance. We usually preprocess training data to have a zero mean and unit standard deviation. There are also training techniques that impose a gradient penalty when computing training losses [8].

2.2 1st Order Update Schemes

We have seen the formulation of gradient descent as it relates to ANN's. Now we will consider the practical implications of the vanilla implementation, and how these implications inform different first order update schemes that are used in the field today.

In vanilla gradient descent (referred to as full-batch gradient descent in the machine learning community), we compute the gradient of the loss landscape $\nabla C(\vec{p})$ by averaging contributions over all n samples of our training data $D, |D| = n$. Considering that we now train models with hundreds of thousands and millions of data samples, performing a single step of the full-batch gradient descent becomes computationally intractable given current hardware limitations. A healthy compromise is to perform each gradient descent step over a randomly selected mini-batch $B \subset D, |B| << |D|$. This forms the basis of Stochastic Gradient Descent.

2.2.1 Stochastic Gradient Descent

Given a dataset D such that $|D| = n$, we choose a random minibatch B with $|B| = m$. We then perform gradient descent with this minibatch:

$$C(\vec{p}) = \frac{1}{m} \sum_{i=1}^{m} |\hat{y}_i - y_i|$$
$$\vec{p}^{(t+1)} = \vec{p}^{(t)} - \frac{\eta}{m} \sum_{i=1}^{m} \nabla C_i(\vec{p}^{(t)})$$

In practice, SGD has a number of interesting properties. Because of its stochasticity its gradient updates tend to be noisy, even though they generally reach the same minimums. This noise is often a feature rather than a bug: when the optimizer reaches a sharp local minimum, the noise allows it to escape this minimum. Unfortunately, SGD is not very robust against saddle points. A saddle point is a region with near-zero gradients, but one which isn't a minimum. In regions like this, the optimizer will slow down to a halt since $\nabla C(\vec{p}) \approx \vec{0}$.

2.2.2 Momentum

We make a modification to the SGD algorithm to add a *momentum* term. Conceptually, one could imagine the optimizer as a ball on the optimization landscape. Its motion at a given update becomes defined not only by the gradient of the hypersurface beneath it, but by its velocity up to that point. Hence we have the update scheme for SGD+Momentum:

$$\vec{v}^{(t+1)} = \beta \vec{v}^{(t)} + \eta \nabla C(\vec{p}^{(t)})$$
$$\vec{p}^{(t+1)} = \vec{p}^{(t)} - \vec{v}^{(t+1)}$$

Where β is a hyperparameter, usually chosen to be 0.9 or 0.99. The momentum term allows the optimizer to escape local minima and saddle points, as it has 'inertia'.

2.2.3 Adaptive Sub-Gradient

Now consider another challenge for the optimizer. On some loss landscapes, the gradients in one axis could be significantly higher in magnitude than in other axes. On such an ill-conditioned hypersurface, SGD will oscillate significantly on the more-skewed axis and move very slowly on the less-skewed axis.

A natural way to prevent this is to keep a per-parameter learning rate, η_i. The AdaGrad

optimizer computes its per-parameter learning rate by dividing the nominal learning rate η by the rolling sum of all past gradients for a given parameter:

$$\begin{aligned}\vec{g}^{(t+1)} &= \sqrt{\sum_{s=1}^{t} \left(\nabla C(\vec{p})^{(s)}\right)^2} \\ \vec{\eta}^{(t+1)} &= (\eta * \vec{1}) \oslash \vec{g}^{(t+1)} \\ \vec{p}^{(t+1)} &= \vec{p}^{(t)} - \vec{\eta}^{(t+1)} \circ \nabla C(\vec{p}^{(t)})\end{aligned}$$

Where $\circ$ is the Hadamard product and $\oslash$ is the Hadamard division.

2.2.4 Root Mean Square Propagation

The obvious problem with the AdaGrad scheme is that as the number of optimizer steps t increases, the effective learning rate becomes smaller and smaller. As a result, the RMSProp update rule is proposed. Instead of normalizing the nominal learning rate by the sum of all past gradients, RMSProp utilizes a rolling average:

$$\begin{aligned}\vec{g}^{(t+1)} &= \sqrt{(\beta \vec{g}^{(t)})^2 + (1-\beta)(\nabla C(\vec{p}))^2} \\ \vec{\eta}^{(t+1)} &= (\eta * \vec{1}) \oslash \vec{g}^{(t+1)} \\ \vec{p}^{(t+1)} &= \vec{p}^{(t)} - \vec{\eta}^{(t+1)} \circ \nabla C(\vec{p}^{(t)})\end{aligned}$$

Where β is a hyperparameter, typically set to 0.9.

2.2.5 Adaptive Moment Estimation

The Adaptive Moment Estimation (Adam) optimizer, which is the most commonly used optimizer, combines SGD+Momentum and RMSProp:

$$\begin{aligned}\vec{v}^{(t+1)} &= \beta_1 \vec{v}^{(t)} + (1-\beta_1)\nabla C(\vec{p}^{(t)}) \\ \vec{g}^{(t+1)} &= \sqrt{(\beta_2 \vec{g}^{(t)})^2 + (1-\beta_2)(\nabla C(\vec{p}))^2} \\ \vec{\eta}^{(t+1)} &= \eta * (\vec{v}^{(t+1)} \oslash \vec{g}^{(t+1)}) \\ \vec{p}^{(t+1)} &= \vec{p}^{(t)} - \vec{\eta}^{(t+1)} \circ \nabla C(\vec{p}^{(t)})\end{aligned}$$

Where β_1 and β_2 are parameters usually chosen to be 0.9 and 0.999 respectively.

2.3 2nd Order Optimization

The gradient descent method finds the minimum of a given function by taking steps that minimize the first order Taylor approximation of the given function. What if we could construct a better approximation of the given function? To do so, we could use the second order Taylor approximation:

$$f(\vec{x}+\Delta\vec{x}) \approx f(\vec{x}) + \Delta\vec{x}^T\nabla f(\vec{x}) + \frac{1}{2}\Delta\vec{x}^T\nabla^2 f(\vec{x})\Delta\vec{x}$$

Hence we expect to minimize this approximation with respect to $\Delta\vec{x}$ when:

$$\frac{\delta f(\vec{x}+\Delta\vec{x})}{\delta\Delta\vec{x}} = \nabla f(\vec{x}) + \nabla^2 f(\vec{x})\Delta\vec{x} = 0$$

This gives us an update scheme for our optimization:

$$\begin{aligned}\Delta\vec{x} = \vec{x}_{(k+1)} - \vec{x}_{(k)} &= -H_{(k)}^{-1}\nabla f(\vec{x}_{(k)}) \\ \therefore \vec{x}_{(k+1)} &= \vec{x}_{(k)} - H_{(k)}^{-1}\nabla f(\vec{x}_{(k)}) \\ \text{where } H_{(k)} &= \nabla^2 f(\vec{x}_{(k)})\end{aligned}$$

This defines **Newton's Method** used for root finding. It is worth noting that we can only expect this method to converge when the Hessian is positive definite. Suppose we have chosen some $\vec{x}$ such that $\nabla f(\vec{x}) \approx 0$. At one such critical point, our approimation becomes:

$$\begin{aligned}f(\vec{x}+\Delta\vec{x}) &= f(\vec{x}) + \frac{1}{2}\Delta\vec{x}^T\nabla^2 f(\vec{x})\Delta\vec{x} \\ \therefore f(\vec{x}+\Delta\vec{x}) - f(\vec{x}) &= \frac{1}{2}\Delta\vec{x}^T\nabla^2 f(\vec{x})\Delta\vec{x}\end{aligned}$$

It becomes apparent that for convergence, it must be the case that $\forall\Delta\vec{x}, f(\vec{x}+\Delta\vec{x}) - f(\vec{x}) > 0$. Hence the need for H to be positive definite. In practice however, we often cannot guarantee H being positive definite, so instead we construct a surrogate Hessian G for which we can guarantee positive definiteness. We do so by ensuring that $\forall\lambda_i \in \lambda(G), \lambda_i > 0$. Let $\vec{v}_i \in \mathbb{R}^n, H\vec{v}_i = \lambda_i\vec{v}_i$ be an eigenvector of H. We can define

$G = (H + \mu I)$ so that:

$$\begin{aligned} G\vec{v}_i &= (H + \mu I)\vec{v}_i \\ &= H\vec{v}_i + \mu I\vec{v}_i \\ &= \lambda_i\vec{v}_i + \mu\vec{v}_i \\ &= (\lambda_i + \mu)\vec{v}_i \end{aligned}$$

Hence with $\mu >> 0$, we can almost guarantee that $G \succ 0$. This defines the **Levenberg-Marquadt Modification** used to *dampen* the Hessian. Finally, it is worth noting that the Hessian has the nice propery of being able to step directly to the minimum of a quadratic function. Consider the function:

$$\begin{aligned} f(\vec{x}) &= \frac{1}{2}\vec{x}^T Q\vec{x} - \vec{x}^T\vec{b} \\ f'(\vec{x}) &= Q\vec{x} - \vec{b} \\ f''(\vec{x}) &= Q \end{aligned}$$

Assuming that $Q = Q^T$ and $\exists\ Q^{-1}$, then Newton update gives:

$$\begin{aligned} \vec{x}_{(1)} &= \vec{x}_{(0)} - Q^{-1}\left[Q\vec{x}_{(0)} - \vec{b}\right] \\ &= \vec{x}_{(0)} - \vec{x}_{(0)} + Q^{-1}\vec{b} \\ &= Q^{-1}\vec{b} \end{aligned}$$

2.3.1 Symmetric Rank 1 Update

When optimizing in a small class of problems, like in dynamic simulations with negligible stochasticity, we might be able to analytically compute the Hessian. But in most practical problems, it is either impractical or impossible to compute the Hessian, talkless of its inverse. Thankfully, this should not prevent us from taking advantage of 2nd order information about $f(\vec{x})$. Indeed, we can approximate the Hessian from previous update steps. This forms the basis of **Quasi-Newton Methods**. First, we define some terms:

$$\begin{aligned} \vec{y}_{(k+1)} &= \nabla f(\vec{x}_{(k+1)}) - \nabla f(\vec{x}_{(k)}) \\ \Delta\vec{x}_{(k+1)} &= \vec{x}_{(k+1)} - \vec{x}_{(k)} \end{aligned}$$

We wish to iteratively compute H (or its inverse). We will do so in two steps: first we create an initial estimate; then we will iteratively update the estimate as we take steps.

A good property to impose on H is that its gradients agree with f at $\vec{x}_{(k+1)}$ and $\vec{x}_{(k)}$:

$$\begin{aligned}
\nabla f(\vec{x}_{(k)}) &= H\vec{x}_{(k)} \\
\nabla f(\vec{x}_{(k+1)}) &= H\vec{x}_{(k+1)} \\
\therefore \nabla f(\vec{x}_{(k+1)}) - \nabla f(\vec{x}_{(k)}) &= H\vec{x}_{(k+1)} - H\vec{x}_{(k)} \\
\vec{y}_{(k+1)} &= H\Delta\vec{x}_{(k+1)}
\end{aligned}$$

This is the **Secant Condition**. Hence if we take n steps, each in a direction that is linearly-independent from the others, we have:

$$\begin{bmatrix}\vec{y}_1 & \dots & \vec{y}_n\end{bmatrix} = H \begin{bmatrix}\Delta\vec{x}_1 & \dots & \Delta\vec{x}_n\end{bmatrix}$$

Then H is uniquely determined as $[\Delta\vec{x}_1 \dots \Delta\vec{x}_n]$ must be invertible:

$$\begin{aligned}
H &= \begin{bmatrix}\vec{y}_1 & \dots & \vec{y}_n\end{bmatrix} \begin{bmatrix}\Delta\vec{x}_1 & \dots & \Delta\vec{x}_n\end{bmatrix}^{-1} \\
\begin{bmatrix}\Delta\vec{x}_1 & \dots & \Delta\vec{x}_n\end{bmatrix} &= H^{-1} \begin{bmatrix}\vec{y}_1 & \dots & \vec{y}_n\end{bmatrix}
\end{aligned}$$

This defines the initial approximation of the Hessian inverse:

$$\Delta\vec{x}_{(k)} = H^{-1}_{(k)}\vec{y}_{(k)} \tag{2.1}$$

For the iterative update, we wish to find a $H^{-1}_{(k+1)}$ that satisfies the above equation. To do so, we consider performing a symmetric rank one update on $H^{-1}_{(k)}$:

$$H^{-1}_{(k+1)} = H^{-1}_{(k)} + \alpha_{(k)}\vec{z}_{(k)}\vec{z}^T_{(k)} \tag{2.2}$$

Imposing the secant condition on the symmetric rank 1 update, we have:

$$\begin{aligned}
\Delta\vec{x}_{(k)} &= H^{-1}_{(k+1)}\vec{y}_{(k)} \\
&= \left[H^{-1}_{(k)} + \alpha_{(k)}\vec{z}_{(k)}\vec{z}^T_{(k)}\right]\vec{y}_{(k)} \\
\Delta\vec{x}_{(k)} &= H^{-1}_{(k)}\vec{y}_{(k)} + (\alpha_{(k)}\vec{z}_{(k)}\vec{z}^T_{(k)})\vec{y}_{(k)} \\
\text{but } \vec{a}\vec{a}^T\vec{b} &\equiv (\vec{a}\cdot\vec{b})\vec{a} \equiv \vec{a}^T\vec{b}\vec{a}
\end{aligned}$$

So:

$$\Delta\vec{x}_{(k)} - H^{-1}_{(k)}\vec{y}_{(k)} = \alpha_{(k)}\vec{z}^T_{(k)}\vec{y}_{(k)}\vec{z}_{(k)} \tag{2.3}$$

Taking the outer product of each side:

$$\begin{aligned}\left(\Delta\vec{x}_{(k)} - H_{(k)}^{-1}\vec{y}_{(k)}\right)\left(\Delta\vec{x}_{(k)} - H_{(k)}^{-1}\vec{y}_{(k)}\right)^T &= \alpha_{(k)}^2\vec{z}_{(k)}^T\vec{y}_{(k)}\vec{z}_{(k)}\left(\vec{z}_{(k)}^T\vec{y}_{(k)}\vec{z}_{(k)}\right)^T \\ &= \alpha_{(k)}^2\left(\vec{z}_{(k)}^T\vec{y}_{(k)}\right)\vec{z}_{(k)}\vec{z}_{(k)}^T\left(\vec{z}_{(k)}^T\vec{y}_{(k)}\right) \\ &= \left(\alpha_{(k)}(\vec{z}_{(k)}^T\vec{y}_{(k)})^2\right)\left(\alpha_{(k)}\vec{z}_{(k)}\vec{z}_{(k)}^T\right)\end{aligned}$$

Hence we have our symmetric rank one update:

$$\alpha_{(k)}\vec{z}_{(k)}\vec{z}_{(k)}^T = \frac{\left(\Delta\vec{x}_{(k)} - H_{(k)}^{-1}\vec{y}_{(k)}\right)\left(\Delta\vec{x}_{(k)} - H_{(k)}^{-1}\vec{y}_{(k)}\right)^T}{\alpha_{(k)}(\vec{z}_{(k)}^T\vec{y}_{(k)})^2}$$

$$H_{(k+1)}^{-1} = H_{(k)}^{-1} + \frac{\left(\Delta\vec{x}_{(k)} - H_{(k)}^{-1}\vec{y}_{(k)}\right)\left(\Delta\vec{x}_{(k)} - H_{(k)}^{-1}\vec{y}_{(k)}\right)^T}{\alpha_{(k)}(\vec{z}_{(k)}^T\vec{y}_{(k)})^2}$$

But we still have an unknown term, $\vec{z}_{(k)}$. Taking the dot product of $\vec{y}_{(k)}$ and both sides of equation (2.3), we have:

$$\begin{aligned}\left(\Delta\vec{x}_{(k)} - H_{(k)}^{-1}\vec{y}_{(k)}\right)\cdot\vec{y}_{(k)} &= \left(\alpha_{(k)}\vec{z}_{(k)}^T\vec{y}_{(k)}\vec{z}_{(k)}\right)\cdot\vec{y}_{(k)} \\ \vec{y}_{(k)}^T\left(\Delta\vec{x}_{(k)} - H_{(k)}^{-1}\vec{y}_{(k)}\right) &= \alpha_{(k)}\left(\vec{z}_{(k)}^T\vec{y}_{(k)}\right)^2\end{aligned}$$

Hence we arrive at the symmetric rank 1 Hessian inverse update:

$$H_{(k+1)}^{-1} = H_{(k)}^{-1} + \frac{\left(\Delta\vec{x}_{(k)} - H_{(k)}^{-1}\vec{y}_{(k)}\right)\left(\Delta\vec{x}_{(k)} - H_{(k)}^{-1}\vec{y}_{(k)}\right)^T}{\vec{y}_{(k)}^T\left(\Delta\vec{x}_{(k)} - H_{(k)}^{-1}\vec{y}_{(k)}\right)}$$

2.3.2 Davidson-Fletcher-Powell Update

The SR1 update is susceptible to two problems: first, we cannot guarantee that $H_{(k)} \succ 0$; and second, if $\vec{y}_{(k)}^T\left(\Delta\vec{x}_{(k)} - H_{(k)}^{-1}\vec{y}_{(k)}\right) \approx 0$, the method becomes numerically unstable. The DFP method addresses these concerns by performing a symmetric rank 2 update of the form:

$$H_{(k+1)}^{-1} = H_{(k)}^{-1} + \alpha_{(k)}\Delta\vec{x}_{(k)}\Delta\vec{x}_{(k)}^T + \beta_{(k)}\left(H_{(k)}^{-1}\vec{y}_{(k)}\right)\left(H_{(k)}^{-1}\vec{y}_{(k)}\right)^T$$

Where:

$$\alpha_{(k)} = \frac{1}{\Delta\vec{x}_{(k)}^T \vec{y}_{(k)}}$$
$$\beta_{(k)} = \frac{-1}{\vec{y}_{(k)}^T H_{(k)}^{-1} \vec{y}_{(k)}}$$

With DFP it can be shown that if $H_{(0)}^{-1} \succ 0$, then $\forall k > 0,\ H_{(k)}^{-1} \succ 0$.

2.3.3 Broyden-Fletcher-Goldfarb-Shanno Update

The BFGS update computes the Hessain instead of its inverse. Much of the formulation from the SR1 update holds, as we can simply switch $\Delta\vec{x}_{(k)}$ and $\vec{y}_{(k)}$ in the SR1 update scheme:

$$H_{(k+1)} = H_{(k)} + \frac{\left(\vec{y}_{(k)} - H_{(k)}\Delta\vec{x}_{(k)}\right)\left(\vec{y}_{(k)} - H_{(k)}\Delta\vec{x}_{(k)}\right)^T}{\Delta\vec{x}_{(k)}^T\left(\vec{y}_{(k)} - H_{(k)}\Delta\vec{x}_{(k)}\right)}$$

Hence we can apply the SR2 update rule from DFP:

$$H_{(k+1)} = H_{(k)} + \frac{\vec{y}_{(k)}\vec{y}_{(k)}^T}{\vec{y}_{(k)}^T\Delta\vec{x}_{(k)}} - \frac{\left(H_{(k)}\Delta\vec{x}_{(k)}\right)\left(H_{(k)}\Delta\vec{x}_{(k)}\right)^T}{\Delta\vec{x}_{(k)}^T H_{(k)}\Delta\vec{x}_{(k)}}$$

With this, we have an update scheme for $H_{(k+1)}$. But we need to compute $H_{(k+1)}^{-1}$ for the minimization. Suppose we have an invertible matrix $A \in \mathbb{R}^{n\times n}$ and some vectors $\vec{u}, \vec{v} \in \mathbb{R}^n$. Also suppose that $1 + \vec{v}^T A^{-1}\vec{u} \neq 0$, then:

$$(A + \vec{u}\vec{v}^T)^{-1} = A^{-1} - \frac{A^{-1}\vec{u}\vec{v}^T A^{-1}}{1 + \vec{v}^T A^{-1}\vec{u}}$$

This defines the **Sherman-Morrison Matrix Inversion Formula** [5]. Applying this to the BFGS update, we compute $H_{(k+1)}^{-1}$ as:

$$H_{(k+1)}^{-1} = H_{(k)}^{-1} + \left(1 + \frac{\vec{y}_{(k)}^T H_{(k)}^{-1}\vec{y}_{(k)}}{\Delta\vec{x}_{(k)}^T\vec{y}_{(k)}}\right)\frac{\Delta\vec{x}_{(k)}\Delta\vec{x}_{(k)}^T}{\Delta\vec{x}_{(k)}^T\vec{y}_{(k)}} - \frac{\Delta\vec{x}_{(k)}\vec{y}_{(k)}^T H_{(k)}^{-1} + \left(\Delta\vec{x}_{(k)}\vec{y}_{(k)}^T H_{(k)}^{-1}\right)^T}{\vec{y}_{(k)}^T\Delta\vec{x}_{(k)}}$$

2.3.4 2nd Order Optimization and Deep Learning

Second-order optimization is not an active research area in deep learning, and is simply not used in the vast majority of architectures. Central to the lack of interest in this area are the complexity, performance cost, and memory usage of these methods. The

memory cost is perhaps the most glaring issue, as storing a Hessian has a $O(n^2)$ space complexity.

We do see second order optimization used, but typically in gradient-associative learning models. A most prominent example of this is the use of L-BFGS in style transfer [7].

3 Measuring Learning: Losses

In this chapter, we discuss the other half of the learning process: a notion of how well the current model performs. First we will discuss regression losses as they are the easiest to grasp. We will then take a look at losses on probability distributions.

3.1 Regression Losses

These losses have geometrical intuition. The notable losses in this category are ℓ_1 and ℓ_2 losses. Given some vector $\vec{y} \in \mathbb{R}^n$, we define the ℓ_p norm as:

$$\ell_p(\vec{y}) = \left(\sum_{i=1}^{n} |\vec{y}_i|^p \right)^{\frac{1}{p}}$$

ℓ_1 loss is also known as Mean Absolute Error (MAE) loss whereas ℓ_2 loss is also known as Mean Square Error (MSE). Other losses such as Huber loss and $\log - \cosh$ loss behave similarly to these losses:

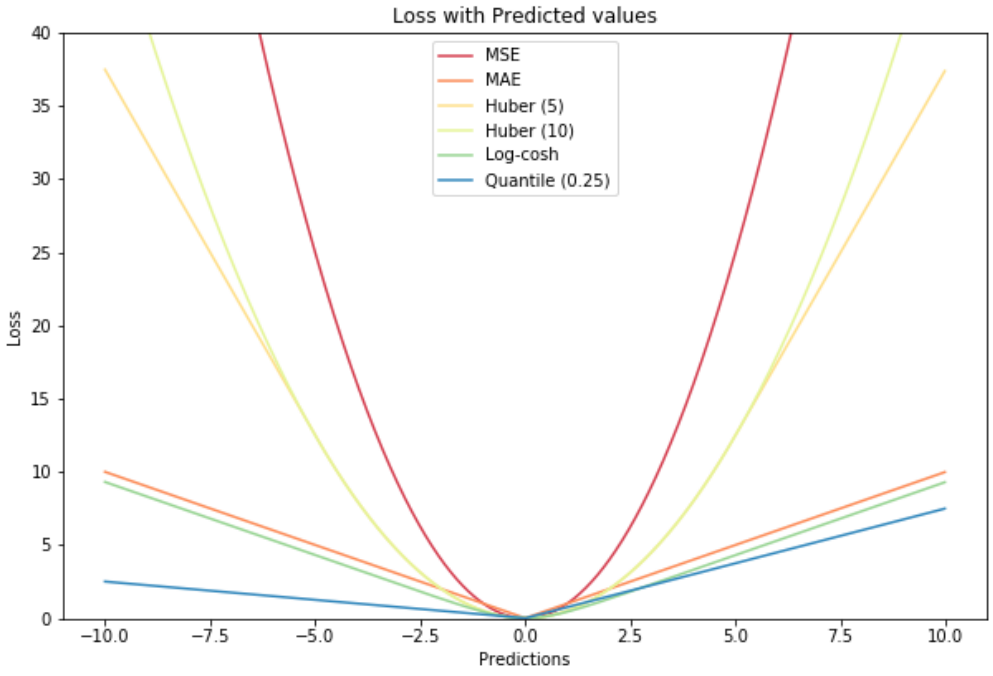

Figure 3.1: Common regression losses. Source

3.2 Distributions and Cross Entropy

Distribution losses are typically used in classification problems, where the task is to build a model $q(x) \mapsto \mathbb{R}^m$ which classifies some input x to be in one of m choices. The nominal classification of sample x during training is encoded as a one-hot vector $\hat{y}$, such that $||\hat{y}||_0 = 1$.

Cross-entropy loss takes its foundations from information theory. First, we define information as the log-likelihood of some event x, based on a given probability distribution:

$$I(x) = -\log p(x)$$

From this definition, we define entropy as the expected value of information given our probability distribution:

$$\begin{aligned}\mathbb{E}_{x\sim\Omega(p)}[I] &= \mathbb{E}_{x\sim\Omega(p)}\left[-\log p(x)\right] \\ &= -\sum_{x\in\Omega(p)} p(x)\log p(x)\end{aligned}$$

With this, we can define cross-entropy as the expected value of information in an estimated distribution $q(x)$ over our true distribution $p(x)$:

$$\begin{aligned}CE(p \mid\mid q) &= \mathbb{E}_{x\sim\Omega(p)}\left[-\log q(x)\right] \\ &= -\sum_{x\in\Omega(p)} p(x)\log q(x)\end{aligned}$$

3.3 Kullback-Leibler Divergence

Similar to cross entropy, KL-divergence also takes its roots in information theory. Suppose we have a true probability distribution $p(x)$ and we generate an estimated distribution $q(x)$, a good measure for the accuracy of our estimated distribution would be to compute the *likelihood ratio* of a given sample within the distributions:

$$LR(p \mid\mid q)_x = \frac{p(x)}{q(x)}$$

Going further, we can compute the likelihood ratio over all samples by taking the product of the likelihood ratios for each samples in our dataset:

$$LR(p \mid\mid q) = \prod_{x\in D}\frac{p(x)}{q(x)} = \sum_{x\in D}\log\frac{p(x)}{q(x)}$$

With this, we define the KL-Divergence as the expected value of the likelihood ratio over our true distribution $p(x)$:

$$D_{KL}(p \mid\mid q) = \mathbb{E}_{x\sim\Omega(p)}\left[LR(p \mid\mid q)_x\right] = \sum_{x\in D} p(x)\cdot\log\frac{p(x)}{q(x)}$$

A different way to formulate D_{KL} is as the expected value of information loss between the true distribution and estimated distribution:

$$D_{KL}(p \mid\mid q) = \mathbb{E}_{x\sim\Omega(p)}[\log p(x) - \log q(x)] = \sum_{x\in D} p(x)\cdot\log\frac{p(x)}{q(x)}$$

It is worth noting that KL-divergence is **not** a metric because it is not symmetric:

$$D_{KL}(p \mid\mid q) \neq D_{KL}(q \mid\mid p)$$

Additionally, KL-divergence does not respect the Cauchy-Schwarz inequality. Finally, eagle-eyed readers might notice that if $\exists x \in D,\ p(x) > 0,\ q(x) = 0$ then $D_{KL}(p \mid\mid q) = \infty$. As a result, we typically perform *smoothing* to ensure that the sample space of p and q are equal:

$$\begin{aligned}
\Omega^* &= \Omega(p) \cup \Omega(q) \\
p^*(x) &= \begin{Bmatrix} p(x) - \frac{|\Omega^*\setminus\Omega(p)|}{|\Omega(p)|} & \text{when } x \in \Omega(p) \\ \epsilon & \text{otherwise} \end{Bmatrix} \\
q^*(x) &= \begin{Bmatrix} q(x) - \frac{|\Omega^*\setminus\Omega(q)|}{|\Omega(q)|} & \text{when } x \in \Omega(q) \\ \epsilon & \text{otherwise} \end{Bmatrix}
\end{aligned}$$

3.4 Wasserstein Metric

The Wasserstein metric originated from the study of optimal transport. The metric was used to quantify the amount of work done in the transfer from one mass distribution to another. However, the Wasserstein metric is also useful in defining the distance between the spaces of random variables. We will analyse the Wasserstein metric from a measure-theoretic perspective, then allude to its applications in deep learning.

Let (S, d) represent a complete, separable metric space and some 0 in the space (the point chose as 0 may be arbitrary and does not affect of construction here). Suppose $1 \leq p < \infty$, then define $\mathfrak{M}_p(S)$ be the collection of all probability measures P on the Borel sets of S such that $\int_S d^p(X, 0)dP(X)$ is finite.

Let $\mathfrak{M}_\infty(S)$ be the set of all probability measures on S with bounded support. If

$P_1, P_2 \in \mathfrak{M}_p$, then the ℓ_p Wasserstein distance between P_1 and P_2 is defined by:

$$W_p(P_1, P_2) = \left(inf \int d^p(X, Y) d\mu(X, Y) \right)^{\frac{1}{p}}$$

Where μ could be any probability distribution (metric) on the $S \times S$ (this represents the smallest σ-algebra that contains $S \times S$). Furthermore, the marginals of this product measure must have marginals P_1 and P_2.

Further, we define:

$$W_\infty(P_1, P_2) = inf \| d(X, Y) \|_\infty^{(\mu)} \tag{3.1}$$

Now, we assert that the Wasserstein functions W_p are bona fide metrics on the set $\mathfrak{M}_p$ for $1 \leq p \leq \infty$ [10].

3.5 Earth Mover's Distance

In deep learning, we use the Wasserstein-1 metric (technically, an approximation thereof) in multi-class classification problems where there is a notion of proximity between classes [9]. A prime example of this is building a classifier to produce a linear rating (say, 1-10). If the classifier misclassifies some example, a distribution loss function like cross-entropy or KL-divergence would only characterize the wrong classification, but not how far the predicated label is from the true label. With EMD, we don't have this limitation.

If a trained model produces outputs into non-negative, multidimensional space, the we can see the model as a measure. The ground metric here is defined as some inherent measure on the feature space. We use the ground metric to establish the Wasserstein metric on the output space. Consider a mapping from $\mathcal{X} \in \mathbb{R}^D$ into the space $\mathcal{Y} = R_+^K$ of measures on some finite set $\mathcal{K}$ with $|\mathcal{K} = K|$. Then we assume that $\mathcal{K}$ has a ground metric $d_\mathcal{K}$, which measures the similarity of output dimensions. Further, we learn over some hypothesis space $\mathcal{H}$ of predictors $h_\theta : \mathcal{X} \to \mathcal{Y}$ that is parametrized by $\theta \in \Theta$.

We sample independent identically distributed training sample $= \{(x_1, y_1), \ldots (x_n, y_n)\}$ from some joint distribution $\mathcal{P}_{X \times Y}$. Given a measure of performance, Σ, we want to find the predictor h_θ that minimizes the expected risk, which is $\mathbb{E}[\Sigma(h_\theta(x), y)]$.

Since it's hard to optimize Σ, we find:

$$min_{h_\theta \in \mathcal{H}} \Big\{ \mathbb{E}_S \big[\ell(h_\theta(x), y \big] = \frac{1}{n} \sum_{i=1}^{n} \ell(h_\theta(x_i), y_i) \Big\}$$

Consider a cost function $c : \mathcal{K} \times \mathcal{K} \to \mathbb{R}$. Then the optimal transport distance to

transport the mass in probability measure μ_1 to match that of μ_2 is:

$$W_c(\mu_1, \mu_2) = inf_{\gamma \in \Pi(\mu_1, \mu_2)} \int_{\mathcal{K} \times \mathcal{K}} c(\kappa_1 . \kappa_2) \gamma(d\kappa_1, d\kappa_2)$$

We define $\Pi(\mu_1, \mu_2)$ to be the set of joint probability measures on $\mathcal{K} \times \mathcal{K}$ with marginals μ_1, μ_2. Now if the cost function is just the metric $d^p_{\mathcal{K}}$, then this equation is just the Wassertein distance. When $p = 1$, we call the metric the Earth Mover's Distance.

In most implementations, EMD loss is not computed exactly because of its computational cost. Instead, we exploit the fact that the EMD is proportional to the difference in the *shapes* of the distributions. Hence, if we can estimate the difference between the shapes of $p(x)$ and $q(x)$, we would have a loss that scales with the true EMD loss. To quickly estimate the shape of the distributions, we use the cumulative distribution function of each distribution:

$$EMD(p \mid\mid q) \approx \sqrt{\sum_{x \in D} \frac{|CDF(p(x)) - CDF(q(x))|^2}{|D|}}$$

4 Learning to Learn: Bayesian Optimization

In training a neural network, there are a number of *hyperparameters* that impact the performance of the model. In fact, we have already encountered one: the learning rate for a first order optimizer. There are numerous other hyperparameters, including the topology of the model for procedural models; the size and amount of data being provided for training; and so on. If we consider the *accuracy* of a model as a function of its hyperparameters $\vec{x}$, the task of finding the best model becomes an optimization problem:

$$\vec{x}^{\,*} = \arg\min_{\vec{x} \in X} f(x)$$

This forms the foundation of the **meta-learning** field in machine learning. The hyperparameter space is not guaranteed to be differentiatable, hence we are unable to use gradient-based numerical optimization methods. We need a bounded black-box global optimizer.

The simplest optimizer to consider is a uniform grid search, where we discretize the hyperparameter space into a grid and evaluate $f(\vec{x})$ at these grid points. The obvious problem with this approach is how expensive it becomes, as it suffers gravely from the curse of dimensionality. Instead, we turn to Bayesian optimization.

First, we discuss properties of functions over which we optimize with Bayesian optimization:

1. We expect that f is 'continuous'. This constraint is a consequence of how we might model f for the optimization. This is a weak constraint; if a given basis of the hyperparameter space X is discrete, we can usually make it continuous by encoding it into a one-hot vector $\hat{y}$ such that $||\hat{y}||_0 = 1$. And we can then rediscretize it using a logistic function (like $softmax$) followed by a quantizer.

2. Following from above, we expect that $X \subset \mathbb{R}^d$, with $d \leq 20$ for best results.

3. X is a hyperrectangle or simplex in $\mathbb{R}^d$. This is the case as the optimization is typically bounded within some range on each basis of the space.

4. f is derivative-free, so we cannot use first- or second-order optimization methods on it.

5. And most importantly, f is expensive to calculate. We are unable to evaluate it enough times in reasonable time to understand its structure.

There are two main components of the Bayesian optimization: an inference model; and a sampling scheme.

4.1 Building a Surrogate: Gaussian Processes

In Bayesian optimization, we construct a *surrogate model* for f, one which is much easier to evaluate and maximize, and use this surrogate to perform the optimization. Gaussian Processes ('GP' hereafter) are typically used to model the objective function, although other methods like decision trees might be used. A Gaussian Process is a stochastic process such that any finite sub-collection of random variables has a multivariate Gaussian distribution. A stochastic process is an indexed collection of random variables $\{f(y) : y \in Y\}$ where Y is the index set. Essentially, a Gaussian Process describes a *distribution over functions* in the domain X.

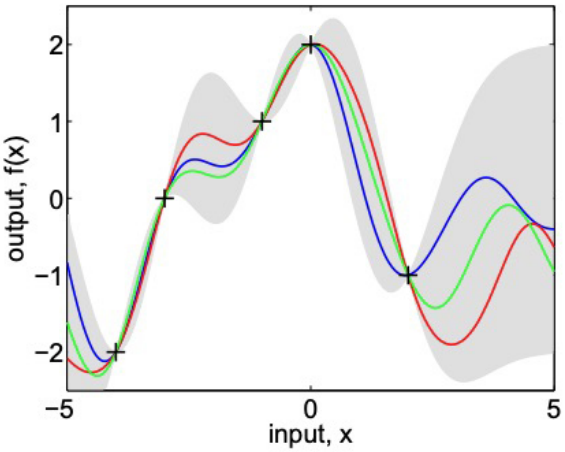

Figure 4.1: Distribution of functions

For a 1D parameter space, we designate the distribution $\{f(x)|\ x \in X\}$ where the value $f(x) \sim \mathcal{N}(\mu(x), \sigma(x, x'))$, where $\mu(x)$ is the mean of f over X and $\sigma(x, x')$ is the covariance kernel of the Guassian Process that determines the linkages between different points in the domain. We apply Bayesian Inference to the Gaussian Process:

$$\text{posterior} \propto (\text{prior} \times \text{likelihood})$$

Which is given in closed form when using a Gaussian Process. We construct a mean vector by evaluating the mean function at each x_i and a covariance matrix by pairwise evaluation of the kernel. Doing this at k points, we have a prior on:

$$f(x_{1:k}) \sim \mathcal{N}(\mu_0(x_{1:k}), \Sigma_0(x_{1:k}, x_{1:k}))$$

With this, we have an updated posterior upon evaluation of a new x, $f(x)$:

$$f(x)|\ f(x_{1:n}) \sim \mathcal{N}(\mu_n(x), \sigma_n^2(x))$$

Where:

$$\mu_n(x) = \Sigma_0(x, x_{1:n})\Sigma_0(x_{1:n}, x_{1:n})^{-1}\left(f(x_{1:n}) - \mu_0(x_{1:n})\right) + \mu_0$$
$$\sigma_n^2(x) = \Sigma_0(x, x) - \Sigma_0(x, x_{1:n})\Sigma_0(x_{1:n}, x_{1:n})^{-1}\Sigma_0(x_{1:n}, x)$$

Where Σ_0 is the covariance matrix. Hence with this, we have a closed-form solution for Bayesian updates. However, there is a multitude of considerations to make when choosing a *covariance kernel*, which determines Σ_0.

4.2 Common Covariance Kernels

We want to choose a covariance kernel $K(x_i, x_j)$ which has the following properties:

1. **Stationary**. We say that $K(x_i, x_j)$ is stationary if $K(x_i, x_j) = g(x_i - x_j)$ where g is an arbitrary function. This property guarantees translational invariance of the kernel.

2. **Isotropic**. We say that $K(x_i, x_j)$ is stationary if $K(x_i, x_j) = g(|x_i - x_j|)$ where g is an arbitrary function. This property guarantees rigid motion invariance of the kernel.

3. **Semi-Positive Definiteness**. The Gram Matrix $G_{i,j} = K(x_i, x_j)$ must be such that $\forall \vec{v} \in \mathbb{R}^n,\ \vec{v}^T G \vec{v} \geq 0$ with $G \in \mathbb{R}^{n \times n}$.

4.2.1 Squared Exponential Kernel

$$K_{\text{SE}}(x_i, x_j) = \exp\left(-\frac{(x_i - x_j)^2}{2l^2}\right)$$

Where l defines the length scale. As $l >> 0$, the correlation of distant values increases and the graph becomes smoother. Some argue that the squared-exponential covariance kernel is too idealistic, and that a rougher shape would be more representative of real-world phenomena. Hence the motivation for the Matern kernel.

4.2.2 Matern Kernel

$$K_{\text{Matern}}(x_i, x_j) = \frac{2^{1-v}}{\Gamma(v)}\left(\sqrt{2v}\frac{|x_i - x_j|}{\rho}\right)^v K_v\left(\sqrt{2v}\frac{|x_i - x_j|}{\rho}\right)$$

Where v and ρ are positive parameters, Γ is the gamma function, and K_v is a Weber function (a Bessel function of the second kind). v is usually any $p + \frac{1}{2}$, $p \in \mathbb{Z}^+$. As $v \to \infty$, the function becomes smoother. Setting $v = \frac{1}{2}$, $K_{\text{Matern}}(x_i, x_j) = \exp(-\frac{|x_i - x_j|}{\rho})$. This opens up the γ-class of covariance functions:

$$K_\gamma(x_i, x_j) = \exp\left(-\left(\frac{|x_i, x_j|}{\rho}\right)^\gamma\right) \text{ for } 0 < \gamma \leq 2$$

4.3 Kernels and Hyper-hyper Parameter Optimization

The task of choosing a suitable kernel for covairance is itself somewhat of an optimization problem. In this section, we discuss some considerations.

First, we need a good metric on which to measure the performance of the kernel. An important metric to use is **generalization error**. This is a measure of how well the model will be able to generalize to unseen data. The kernel that performs best with respect to this metric is usually the most optimal choice for a Gaussian Process.

Once a covariance kernel is chosen, there are usually hyperparameters which affect the performance of the model. For the squared-exponent kernel, the sole hyperparameter is the length scale l. For the Mattern kernel, there are v, ρ. We usually use a **maximum a-posteriori** probability estimate ('MAP' estimate):

$$\eta^* = \arg\max_\eta P(f(x_{1:n})|\ \eta)$$

Where η is the kernel's hyperparameters. The MAP estimate gives us the maximum for the data we have.

4.4 Exploration and Exploitation: The Acquisition Function

We have explored how to build and update the surrogate model using Bayesian inference. The other half of the Bayesian Optimization process is finding a scheme to sample the objective function, with which we can use the samples to update our surrogate and optimize. To do so, we use an **Acquisition Function** which provides proposals for new samples to take. The choice of acquisition function is made with the central consideration of balancing of **exploration and exploitation**.

We want an acquisition function that explores the objective landscape, as doing so increases chances of finding a global minimum. On the other hand, we want the acquisition function to also exploit already-explored regions to find proximal maxima.

4.4.1 Expected Improvement

Expected Improvement is a common choice for an acquisition function. First, we assume that we have taken n samples of the objective function $f(x)$. Let $f(x^*)$ be the maximum over n samples. We then consider the utility of our prospective sample as:

$$u(x) = (f(x) - f(x^*))^+$$

The expected value of $u(x)$ provided data is given in closed form based on the Gaussian Process model:

$$EI(x) = \mathbb{E}\left[u(x)|\ x_{1:n}, f(x_{1:n})\right]$$
$$x^*_{EI} = \arg\max_{x \in X} EI(x)$$

4.4.2 Knowledge Gradient

Given n observations of $f(x)$, Expected Improvement assumes that the maximum exists within those points. In this, EI assumes that the objective is noise-free and uses a risk-averse policy. Knowledge Gradient however does not make these assumptions:

$$KG_n(x) = \mathbb{E}_n\left[\mu^*_{n+1} - \mu^*_n|\ x_{n+1} = x\right]$$
$$x^*_{KG} = \arg\max_{x \in X} KG(x)$$

The Knowledge Gradient acquisition function focuses on improving the GP model to accurately model the underlying objective $f(x)$. In this sense, Knowledge Gradient prioritizes exploration of the objective landscape as opposed to exploitation of already-visited regions.

4.5 Inference and Markov Chain Monte Carlo

Going beyond Bayesian Optimization, a fundamental problem we face is the task of estimating the geometry of a distribution. In Bayesian Optimization, we achieve this by utilizing value heuristics within acquisition functions. In Bayesian Inference, this task is central in computing a posterior given an observation.

For some distribution D over a finite set X, in which we know $p(x)$ for $x \in X$ but not the geometry of the distribution, we can use the Markov Chain Monte Carlo method to approximate this distribution. First, we introduce the Monte Carlo sampling method.

4.5.1 Monte Carlo Sampling

We define some multi-dimensional space X with some density $p(x)$ defined on it. First consider the samples $\{x^1, \ldots, x^N\}$ drawn from independent, identical distributions (i.i.d). The Monte Carlo method to approximate this density is to define the point-mass function:

$$p_N(x) = \frac{1}{N} \sum_{i=1}^{N} \delta_{x^{(i)}}(x) \tag{4.1}$$

Where $\delta_{x^{(i)}}(x)$ is the Dirac-delta function. We have:

$$I_N(f) = \frac{1}{N} \sum_{i=1}^{N} f(x^i) \xrightarrow{a.s.} I(f) = \int_X f(x)p(x)dx \tag{4.2}$$

Where $a.s.$ means "almost surely" in a measure-theoretic sense. Equation (4.2) is correct under the assumption that the sampling is unbiased and extenseive with respect to the Law of Large Numbers. One might also analyze the error with the Central Limit Theorem for the convergence to the integral. Notice that if we consider an integral of f over $D \subset \mathbb{R}^n$, then we can use this approach to solve for the integral of f[1]. Observe:

$$\int_D f(x) \frac{1}{V(D)} dx = \lim_{N \to \infty} \frac{V(D)}{N} \sum_{i=1}^{N} f(x^{(i)})$$

Where $V(D)$ is the volume of the domain $D \in \mathbb{R}^n$ and we assume $p(X)$ is the uniform distribution over D. This defines the vanilla Monte Carlo sampling method. Because the method becomes inefficient and intractable as $\dim X >> 0$, we introduce a sampling heuristic to make the method more tractable.

[1] This method is very different from techniques such as the Simpson's rule because it is non-deterministic

4.5.2 Rejection Sampling

Rejection sampling falls under generation techniques that sample random variables in more than one step. These methods are called **resampling methods**. The first step is usually to provide a sample from an appropriate distribution; and the second step is utilizing a typically stochastic correction mechanism to redirect the sample such that the sample becomes approximately representative of the distribution of interest.

Rejection sampling uses an auxiliary density q for the generation of random quantities from distributions that are not amenable to analytic treatment. We use q to make generations from one such distribution π. We need that $\pi(x) \leq Aq(x)$ for some $A < \infty$ and $x \in X$, where X is some probability space. We call q the blanketing density or an envelope and A is the envelope constant.

For example, enveloping $\mathcal{N}(0,1)$ with a multiple of $t_1(0, 2.5)$, which is the Cauchy density. It is common that the kernel of π is known but the constant ensuring it integrates to 1 cannot be obtained. It turns out that the rejection method does not need such scaling factor, which is a major advantage in Bayesian statistics. Recall that the kernel of a probability distribution π is the form of the probability distribution function in which any factors that are not functions of any of the variables in the domain are omitted.

With some $u \in \mathcal{U}_{[0,1]}$, we independently draw x from q and accept x as a value generated from π if $u \leq \frac{\pi(x)}{Aq(x)}$. Otherwise, we reject x and reinitialize until an x is accepted. We will now show that the rejection method effectively generates values from π by showing the conditional density of $[x|Auq(x) \leq \pi(x)]$ is π.

$$\begin{aligned} f(x|Auq(x) \leq \pi(x)) &= \frac{Pr(Auq(x) < \pi(x)|x)f(x)}{\int Pr(Auq(x) \leq \pi(x)|x)f(x)}dx \\ &= \frac{[\frac{\pi(x)}{aq(x)}]q(x)}{\int [\frac{\pi(x)}{aq(x)}]q(x)}dx \\ &= \frac{\pi(x)}{\int \pi(x)}dx \\ &= \pi(x) \text{ if } \pi \text{ is normalized} \end{aligned}$$

Hence, we have $f(x|Auq(x) \leq \pi(x)) = 1$, so the rejection method effectively generates values from π. The efficiency of the method is directly related to the similarity between

π and q since:

$$\begin{aligned} Pr(Auq(x) < \pi(x)) &= \int Pr(Auq(x) < \pi(x)|x)q(x)dx \\ &= \int (\frac{\pi(x)}{Aq(x)})q(x)d \\ &= \frac{1}{A}\int \pi(x)dx \end{aligned}$$

Thus, A must be chosen as close as possible to $\int \pi(x)dx$, which is equal to one if π is normalized. We hope to increase the acceptance rate while still keeping q an envelope of π.

Note that often, two densities might be similar in the region concentrating the bulk of probability but their tail behaviors very different, which may lead to a large A (this is undesirable). It is also worth noting that sometimes, a total envelope can only be achieved with a low acceptance rate. We can use a small A if we resign to just having a high probability of enveloping π rather than having Aq certainly envelop π.

Finally, if the computation of π is costly then some other auxiliary function $s(x) \leq \pi(x)$ may be used as a pretest so that π is squeezed by s and q. We then verify that $u < \frac{s(x)}{Aq(x)}$ while knowing $\frac{s(x)}{Aq(x)} \leq \frac{\pi(x)}{q(x)}$. If the pretest is efficient (s is close to π), then we know q is close to π.

The iteration scheme for rejection sampling is given as:

Algorithm 1: Rejection Sampling

Result: sample set $\{x_{(i)}\}$
while $i < N$ **do**
  Sample x^* from $q(x)$ and u from $\mathcal{U}_{(0,1)}$
  if $u < \frac{p(x^*)}{Aq(x^*)}$ **then**
    $x_{(i)} = x^*$
    accept
  else
    reject
  end
end

Rejection sampling can be impractical because we may not be able to find an efficient $q(x)$. For instance, if the constant factor M is very large, then the acceptance rate of the algorithm will be very low. As a consequence, the rate of convergence would be too

slow for practical purposes:

$$Pr(x \text{ accepted}) = Pr\Big(u < \frac{p(x)}{Mq(x)}\Big) = \frac{1}{M}$$

As M increases, the probability of acceptance decreases in inverse proportionally.

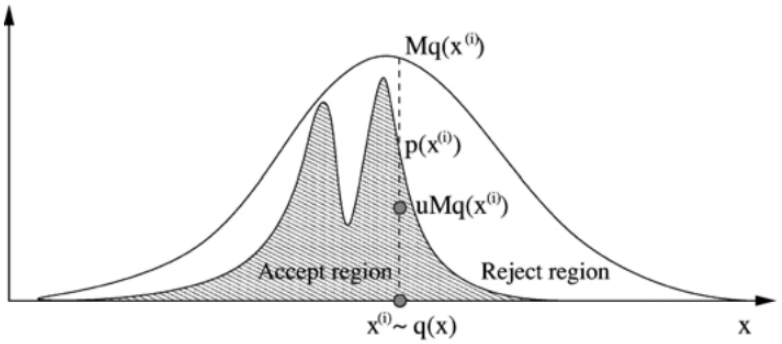

Figure 4.2: Rejection sampling schematic [11]

4.5.3 Markov Chains

We wish to create a Markov chain that approximates some unknown distribution from through only sampling the distribution. This chain should *spend more time* in regions of higher density. Then the convergence rate to the steady-state of this Markov chain should be proportional to the convergence rate to the distribution of interested.

First, we define some conditions on a Markov chain T:

1. **Memoryless**. We want $p(x_{(i)}|x_{(i-1)}, x_{(i-2)}, \ldots, x_{(1)}) = T(x_{(i)}|x_{(i-1)})$.

2. **Homogeneous**. We want $T = T(x_{(i)}|x_{(i-1)})$ to be invariant for all i such that $\sum_{x_{(i)}} T(x_{(i)}|x_{(i-1)}) = 1$

3. **Ergodic**. For some n, T^n has only positive entries (represented with a directed graph G, G must be strongly connected).

4. **Aperiodic**. Regardless of initial state, the probability of reaching any state is non-zero as $n \to \infty$, where n is the number of "steps" (so we cannot get trapped in cycles).

Note that since $T \in \mathbb{R}^{m \times m}$ is a real square matrix, the Perron-Frobenius theorem applies. As each column of T sums to 1, the greatest eigenvalue has absolute value 1 and the remaining eigenvalues are all smaller in magnitude. Further, the second largest

eigenvalue in magnitude determines the convergence rate of T to the steady state. We want this second smallest eigenvalue to be as small in magnitude as possible. When we are in a continuous space, the transition matrix T called an integral kernel, K, and $p(x^i)$ is an eigen-function (instead of a vector):

$$\int p(x_{(i)})K(x_{(i+1)}|\ x_{(i)})dx_{(i)} = p(x_{(i+1)})$$

4.5.4 Markov Chain Monte Carlo

In Markov Chain Monte Carlo, we wish to construct a suitable Markov Chain transition matrix and also ensure its efficient convergence to the density of interest. Consider a distribution p_x, $x \in S$ with $\sum_x p_x = 1$ where the state space S can be a subset of the line or even a d-dimensional subset of $\mathbb{R}^d$. The problem posed and solved by the Metropolis et al. (1953) was how to construct a Markov chain with stationary distribution π such that $\pi(x) = p_x$, $x \in S$.

Let Q be any irreducible transition matrix on S satisfying the symmetry condition $Q(x, y) = Q(y, x)$ for all $x, y \in S$. Define a Markov Chain $(\theta^{(n)})_{n\geq 0}$ as having transition from x to y proposed according to $Q(x, y)$. The proposed value for $\theta^{(n+1)}$ is accepted with the probability $min\{1, \frac{p_y}{p_x}\}$ and rejected otherwise, leaving the chain in state x. The transition probabilities $P(x, y)$ of the above chain $(\theta^{(n)})_{n\geq 0}$ are given as:

$$\begin{aligned} Pr(x, y) = Pr(\theta^{(n+1)}) &= P(\theta^{(n+1)} = y, TA|\theta^{(n)} = x) \\ &= Pr(\theta^{(n+1)} = y, \theta^{(n)} = x)Pr(TA) \\ &= Q(x, y) \min\left\{1, \frac{p_y}{p_x}\right\} \end{aligned}$$

Where TA is transition acceptance. If $y = x$, then:

$$\begin{aligned} &Pr(x, x) = Pr(\theta^{(n+1)} = x, TA|\theta^{(n)} = x) + Pr(\theta^{(n=1)} \neq x, \bar{TA}|\theta^{(n)} = x) \\ &= Pr(\theta^{(n+1)} = x|\theta^{(n)} = x)p(TA) + \sum_{y\neq x} Pr(\theta^{(n+1)} = y, \bar{TA}|\theta^{(n)} = x) \\ &= Q(x, x) + \sum_{y\neq x} Q(x, y)\left[1 - min\left\{1, \frac{p_y}{p_x}\right\}\right] \end{aligned}$$

We have obtained the stationary distribution of this chain. First we show that p_x satisfies reversibility. For $x \neq y$, we have $\pi(x)Pr(x, y) = \pi(y)Pr(y, x)$ for all $x, y \in S$. For $x \neq y$, suppose WLOG, $p_y > p_x$, we have $p_xPr(x, y) = p_xQ(x, y) = Q(x, y)min\{1, \frac{p_y}{p_x}\}p_y = p_yPr(y, x)$.

Thus, the chain is reversible and there is a stationary distribution. If Q is aperiodic, so is p and the stationary distribution is also the limiting distribution. Note that not all stationary distributions also constitute as limiting distributions. Thus, $P\pi = \pi \nRightarrow \lim_{n\to\infty} P^n(x, y) = \pi(y)$, where P is the transition probability.

4.5.5 Metropolis-Hastings

Consider a distribution π from which a sample must be drawn via Markov chains. This is needed when the non-iterative generation of π is infeasible. In this case, a transition kernel $p(\theta, \phi)$ must be constructed so that π is the equilibrium distribution of the chain. Consider the reversible chains where the kernel p satisfies:

$$\forall_{(\theta,\phi)}\ \pi(\theta)p(\theta, \phi) = \pi(\phi)p(\phi, \theta)$$

This is called **detailed balance**. Detailed balance is a sufficient (but not necessary) condition to ensure convergence. The kernel $p(\theta, \phi)$ consists of 2 elements: an arbitrary transition kernel $q(\theta, \phi)$ and a probability $\alpha(\theta, \phi)$ where:

$$p(\theta, \phi) = q(\theta, \phi)\alpha(\theta, \phi),\ \theta \neq \phi$$

The transition kernel defines a density $p(\theta, \cdot)$, for every $\phi \neq \theta$. Further, $P(\theta, \theta) = 1 - \int q(\theta, \phi)\alpha(\theta, \phi)d\phi$. Then we have:

$$p(\theta, A) = \int_A q(\theta, \phi)\alpha(\theta, \phi)d\phi + I(\theta \in A)\left[1 - \int q(\theta, \phi)\alpha(\theta, \phi)d\phi\right]$$

For a subset A of the parameter space. The transition kernel defines a mixed distribution for the new state ϕ of the chain. Hastings (1970) proposed an acceptance probability:

$$\alpha(\theta, \phi) = \min\left\{1, \frac{\pi(\phi)q(\phi, \theta)}{\pi(\theta)q(\theta, \phi)}\right\}$$

This α combined with some transition kernel should give a reversible chain. Roberts and Smith (1994) showed that if q is irreducible and aperiodic, and $\alpha(\theta, \phi) > 0$ for all (θ, ϕ), then the algorithm gives a irreducible, aperiodic chain with transition kernel p

and limiting distribution π.

Algorithm 2: Metropolis Hastings

Result: sample set $\{x_{(i)}\}$
initialize $x_{(0)} \in X$
for $i = 0 : N - 1$ **do**
  Sample x^* from $q(x^* | \, x_{(i)})$ and u from $\mathcal{U}_{(0,1)}$
  if $u < \alpha(x_{(i)} | \, x^*)$ **then**
    $x_{(i+1)} = x^*$
    accept
  else
    $x_{(i+1)} = x_{(i)}$
    reject
  end
end

The figure below shows the results of performing MCMC with the Metropolis-Hastings Algorithm on the distribution $p(x) \propto 0.3\exp(-0.2x^2)+0.7\exp(-0.2(x-10)^2)$. It is quite apparent that the histogram from from sampling convergences to the true distribution over iterations with the proposal distribution $\mathcal{N}(x^i, 100)$ for 5000 iterations [11].

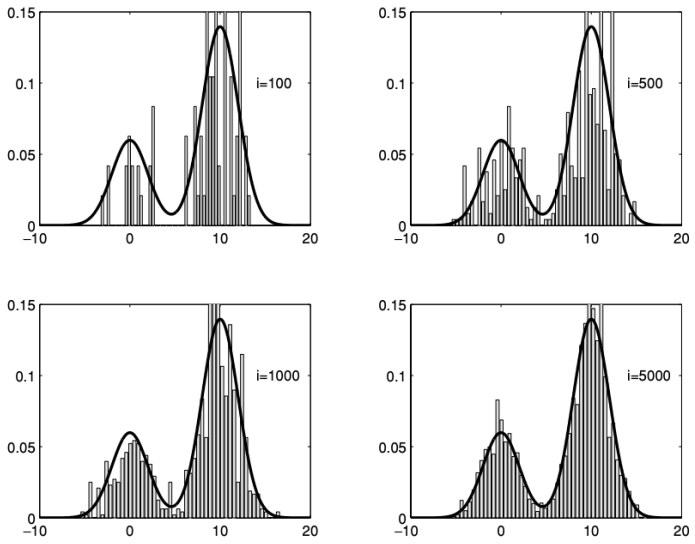

Figure 4.3: MH results [11]

We used the normal distribution in the above example but it is crucial to select a good proposal distribution. For instance, if the standard deviation of the proposal distribution is too low, then the chain will likely get stuck as some single mode of the distribution. However, if the standard deviation is too high, then we get high correlation as the

rejection probability is too high. The objective is to have a high acceptance probability while still visiting all the nodes. We will discuss two common choices for this:

1. **Symmetric Chains**. A chain is symmetric if its transition kernel p is symmetric so $p(\theta, \phi) = p(\phi, \theta)$. For Metropolis-Hastings, this means $\alpha = \{1, \frac{\pi(\phi)}{\pi(\theta}\}$, and does not depend on q. Notice q only depends on $|\phi - \theta|$.

2. **Random Walk Chains**. A random walk is a Markov chain with evolution given by $\theta^{(n)} = \theta^{(n_1)} + \omega_j$, where ω_j is some random variable. In general ω_j are i.i.d with density f_ω. Then we have $q(\theta, \phi) = f_\omega(\phi - \theta)$. Often, f_ω is chosen to be the normal distribution. Then a larger variance of f_ω may increase coverage of the parameter space but decrease acceptance rate. One method is to set the variance of f_ω as cV where c is some tuning parameter that changes based on the current acceptance rate.

A useful monitoring device of the method is given by the average percentage of iterations for which moves are accepted. If $q(\cdot, \cdot)$ and $\pi(\cdot, \cdot)$ are continuous, generally smaller step sizes lead to greater acceptance rates but slower convergence rates.

Markov Chain Monte Carlo is particularly advantageous because we are able to derive certain properties of the distribution by approximating it up to a constant factor. This is important because normalization is often a non-trivial task. Also, the algorithm can be parallelized so that multiple independent chains are evaluated at the same time.

5 Understanding Data: Spectral Analysis

Much of the data we deal with in application is complex, whether spatially or temporally. But notwithstanding this complexity, we can usually recharacterize data into a much more fundamental form, one which allows us to learn more about the data. Take music for example: we perceive the world around us as the superposition of individual sounds. And even though we lose information about the individual sounds constituent in this superposition, our auditory system is able to filter the superposition, isolating the individual sounds.

In this chapter, we will explore spectral analysis starting with 1D signals (sounds) before looking at different applications thereof in deep learning. But first, let us explore the nature of signals and how we digitize them.

5.1 Characterizing Signals

We begin with continuous signals $x(t)$, which have a response $\forall t \in \mathbb{R}$. In order to store such a signal on a computer, we must discretize it:

$$P = \left\{ x\left(\frac{t}{\alpha}\right) \;\middle|\; t \in \mathbb{Z}^+ : t < n \right\}$$

Where α is the sampling frequency, n is the sample count, and P is the discretized signal. This defines the **fixed-interval sampling** scheme. The critical parameter is α, the sampling frequency. We can choose an α such that we can completely reconstruct the continuous signal once it has been discretized. For this, we turn to the Nyquist-Shannon sampling theorem which states that a bandlimited continuous-time signal can be sampled and perfectly reconstructed from its samples if the signal is sampled over twice as fast as it's highest frequency component:

$$\alpha > 2f_{\max}$$

The human auditory system has a dynamic range of 20Hz-20KHz. The Nyquist-Shannon theorem is the reason why most audio media has a sample rate of 44.1KHz and above. With this, we can delve into how we might deconstruct a complex signal into its constituents. First, we discuss the Fourier Transform.

5.2 Fourier Transform

The Fourier transform is a complex transform which deconstructs a waveform into its constituent frequencies:

$$\mathcal{F}_x(f) = \int_{-\infty}^{\infty} x(t) \cdot \exp\left(-2\pi i f t\right) dt$$

Where f is a given frequency. This gives us another signal, where the response at each frequency corresponds to the amount of that frequency within the complex wave:

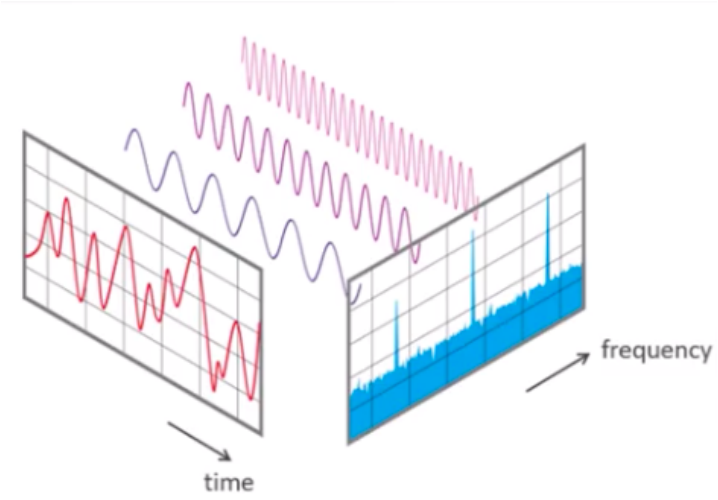

Figure 5.1: Fourier Transform. Source

The Fourier transform has a number of interesting properties, but we will focus on its relation between convolutions and multiplications:

$$f(t) \circledast g(t) \xrightarrow{F.T} \mathcal{F}_f(t) \circ \mathcal{F}_g(t)$$
$$f(t) \circ g(t) \xrightarrow{F.T} \mathcal{F}_f(t) \circledast \mathcal{F}_g(t)$$

The DFT has the implicit assumption that the wave under consideration is *stationary*. A stationary wave is one where each constituent frequency is always present, with no frequency being created or destroyed through the duration of the signal. Most waveforms we will consider in practice do not have this property. As a result, we reach a glaring problem with the DFT: we lose all temporal information about the frequencies within the wave. We know how much of each frequency is in the wave, but we have no clue where they occur. Hence we have the motivation for the Short Time Fourier Transform.

5.3 Short Time Fourier Transform

The Short Time Fourier Transform addresses the temporal uncertainty in the DFT by further discretizing the signal to small enough durations such that we can assume each short duration to be stationary:

$$STFT_x(f, \tau) = \int_{-\infty}^{\infty} (x(t) \cdot \omega^*(t - \tau)) \cdot \exp(-2\pi i f t)\, dt$$

Where $\omega(\cdot)$ is the window function and τ is the window translation. The window function is designed to only have a response over a small duration. We say that the window is *compactly supported*. It is then 'slid' across the signal to compute the DFT for each duration. Common choices for the window function include Dirichlet (rectangular), Gaussian, Blackman, Hann, and Hanning windows.

Although a significant improvement over the DFT, the STFT still suffers from the same problem as the DFT. As the window's support becomes more compact, we lose frequency resolution especially in lower frequencies. But as the window's support becomes broader, we lose temporal information. This is an instantiation of the Heisenberg Uncertainty Principle. This forms the motivation for the Wavelet Transform.

5.4 Wavelet Transform

The Wavelet transform generalizes the STFT by allowing for a multi-scale spectral analysis:

$$\Psi_x(\tau, s) = \frac{1}{\sqrt{|s|}} \int_{-\infty}^{\infty} x(t) \cdot \psi^* \left(\frac{t - \tau}{s} \right) dt$$

Where τ is translation, s is scale, and ψ is **mother wavelet**. Wavelets are compactly-supported periodic functions. Unlike in the DFT, one is free to choose the mother wavelet, perhaps applying domain-specific knowledge about the problem at hand. The choice of a mother wavelet must meet two requirements:

1. **Admissibility Condition**:

$$\int_{-\infty}^{\infty} \psi(t) dt = 0$$

   This condition implies that the Fourier transform of the wavelet is zero at zero frequency [13].

2. **Regularity Condition**:

$$\forall 0 \leq m \leq M, \int_{-\infty}^{\infty} t^m \psi(t) dt = 0$$

This condition implies that the wavelet be locally concentrated in both time and frequency. Hence the first M moments of the wavelet are zero.

3. **Unit Energy**:

$$\int_{-\infty}^{\infty} |\psi(t)|^2 dt = 1$$

This is not necessarily a requirement. The mother wavelet is usually normalized to have unit energy.

Common wavelets include the Haar wavelet, the Daubechies wavelet, and the Mexican Hat wavelet:

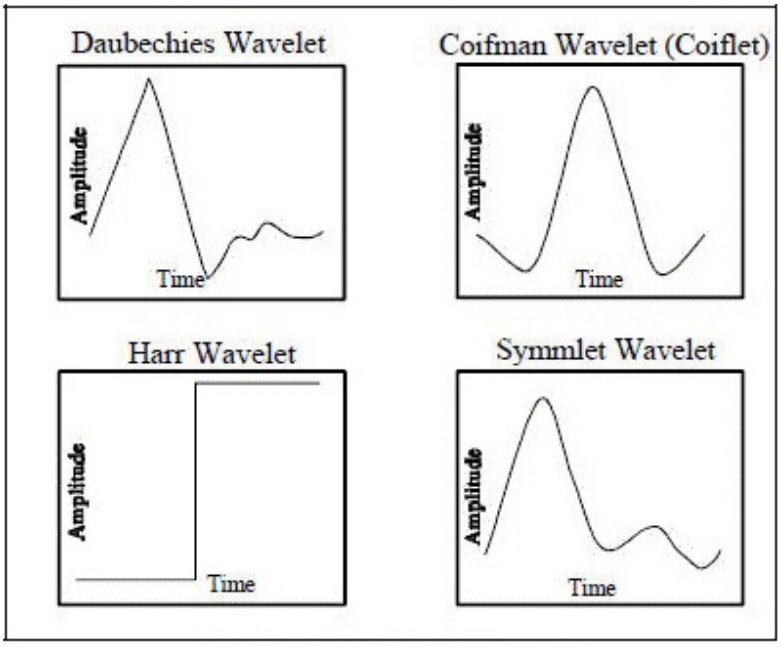

Figure 8. Sample Wavelet Functions

Figure 5.2: Common Wavelets. Source.

5.5 Applications in Deep Learning

We will discuss two important applications of spectral analysis, specifically the DFT, in deep learning: audio analysis and image convolutions.

5.5.1 Audio and Spectrograms

Just like we provide numerical and image data to neural networks, we can also provide audio data to neural networks for regression, classification, and representation tasks. However, we must take special precautions due to how audio is digitized.

The simplest thing to do would be to provide a raw waveform to a model, providing it either as an image or as a 1D array:

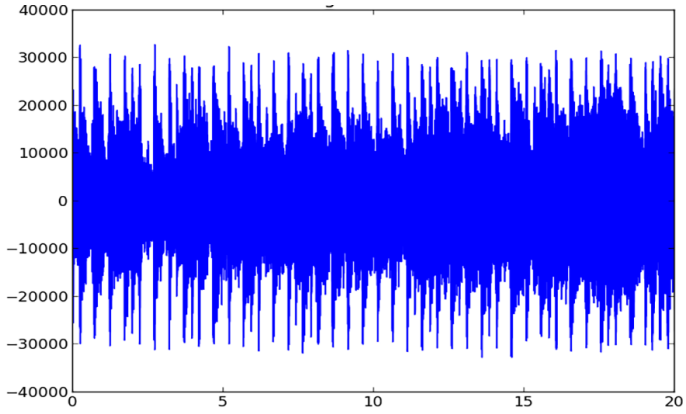

Figure 5.3: Generic Waveform.

But as we have motivated in this chapter, we can use the STFT to construct a spectrogram over the signal, and feed this to a CNN:

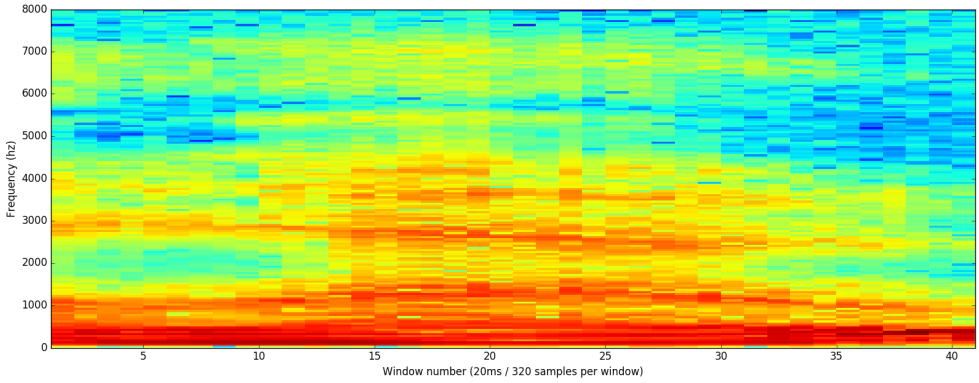

Figure 5.4: Audio Spectrogram. Source.

With this, we can leverage the vast amount of techniques developed for convolutional networks to analyse audio, all done with little specification to the peculiarities of audio.

5.5.2 Images and Convolutions

Finally, the DFT provides a considerable increase in the performance of convolutional networks. Recall that in a CNN, a series of convolution kernels are learnt to extract features from the images within a given dataset. The forward pass during training consists of hundreds-to-thousands of convolutions. Given an m-by-m convolution on an n-times-n image, the complexity of performing the convolution in pixel space is $O(m^2n^2)$.

Now recall from Fourier Theory that $f(t) \circledast g(t) \xrightarrow{F.T} \mathcal{F}(t)_f \circ \mathcal{F}(t)_g$. Hence:

$$I(x, y) \circledast K(\cdot, \cdot) = \mathcal{F}^{-1} \left(\mathcal{F}_I(i, j) \circ \mathcal{F}_K(i, j) \right)$$

Where $\mathcal{F}$ is the Fast Fourier Transform (FFT) and $\mathcal{F}^{-1}$ is the Inverse Fast Fourier Transform (IFFT). The 2D (inverse) FFT has a complexity of $O(n^2 \log n)$. The convolution kernel is padded so that $m = n$. Hence $O(n^2 \log n) < O(m^2n^2)$, showing a drastic improvement in performance as $n >> 0$. The Nvidia cuFFT library which is used in many deep learning frameworks is hand-crafted to use the FFT to perform convolutions on their graphics processors.

6 Understanding Datasets: Bayes Error

In developing and deploying neural networks, the vast majority of attention is given to the model architectures, with very little attention given to the data from which the model will learn. We consider how to analyze the *learnability* of datasets with respect to the learning objective. We will consider this problem with respect to classification problems. Our objective is to characterize the lower bound of the error rate of a given dataset.

When dealing with classifiers and classification problems, there is usually some overlying architecture mapping some data to an element of a discrete set of classes. The architecture can take the form of a CNN, an autoencoder, or even a simple linear regression. We can generalize this relationship with the equation:

$$F(X) = c_i, c_i \in C$$

Where C is the collection of all discrete classes, and X is the data. Hence F is a mapping $F : X \to C$. However in practice the algorithms used to classify data are not deterministic, but rather probabilistic. With this perspective, given a prior distribution of data as well as the likelihood (the distribution of X conditional on the fact it belongs to a class i), the result is the posterior distribution. This result uses Bayes Theorem to relate conditional probabilities. Consider an N-class classification problem with:

$$\begin{aligned} \text{Prior } P(c_i)_{c_i \in L} \text{ s.t. } \sum_{i=1}^{L} P(c_i) = 0 \\ \text{Likelihood } P(X \mid c_i) \\ \text{Posterior } P(c_i \mid X) \end{aligned}$$

In a 2-class problem with classes w_1 and w_2, we could have the posterior distribution look something like this:

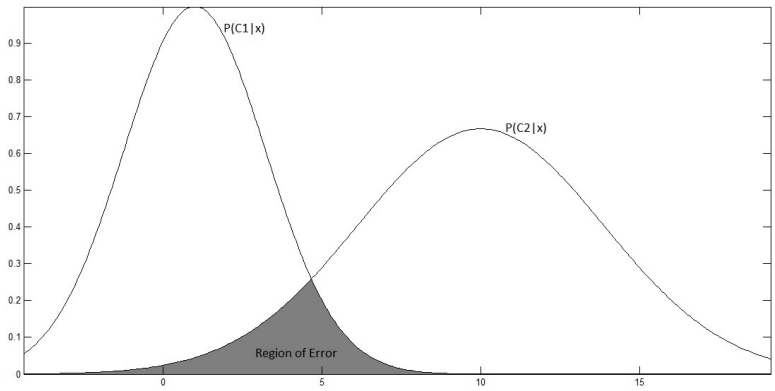

Figure 6.1: Binary classification distribution.

The error term (highlighted in grey) is the *Bayes Error Rate*, which is the minimal achievable error rate inherent in the distribution of the data. Considering an m-class classification problem, we have i.i.d. pairs of training data in the form $\{x_i, y_i\}_{i=1}^n$. Each x_i is a realization of the random variable $X \in \mathbb{R}^d$ (each feature vector in this case is d-dimensional) and each y_i is a realization of the random variable $Y \in \{1, 2, ..., m\}$. In this experimental setup, with prior probabilities of the m classes $p_k = P(Y = k)$ having the prior probabilities sum to 1 across the support of Y, and with conditional feature densities $f_k(x) = P(x \mid Y = k), k = 1, ..., m$ (these can be thought of as the likelihood function), the BER is:

$$\epsilon^m = 1 - \int \max\{p_1 f_1(x), p_2 f_2(x), ..., p_m f_m(x)\} dx$$

In analyzing the equation above, we can see that each term of the max function is the posterior probability $P(Y = k \mid x)$, which mirrors the probabilities in the image above. If we were to simplify the equation above to a 2-class problem, with priors p_1 and p_2, with conditional densities f_1 and f_2, we have posterior probabilities $p_1 f_1$ and $p_2 f_2$. We can break down the posterior support into 2 spaces that span it: one where $p_1 f_1 \leq p_2 f_2$ and the other where $p_2 f_2 \leq p_1 f_1$, so the integral for the 2-class case becomes:

$$\epsilon^2 = 1 - \left(\int_{p_1 f_1 \leq p_2 f_2} p_2 f_2(x) dx + \int_{p_2 f_2 \leq p_1 f_1} p_1 f_1(x) dx \right)$$

The implications of this rate are significant:

1. **It holds irrespective of the architecture used for the classification problem**. In reality the true underlying distribution of the data and the classes is unknown. CNN's, autoencoders, and other architectures try to estimate these spaces, but do not provide concrete function definitions of the spaces themselves. For example, in the two-class diagram above, a neural network could approximate the posterior distributions for the data, but the true distributions remain unknown. So one facet of classifying data is trying best to model the posterior probability distributions.

2. **It can be used to quantify the separability of data**. Much of AFRL's image recognition efforts are moving towards the problem of bridging the measured-synthetic data gap. One of the underlying issues in this problem is the separability of measured and synthetic data. Confusion matrices and tSNE plots are often used to represent algorithm performance, but they have some implications associated with their conclusions:

    - **They depend on the architecture used for the classification problem**. The confusion matrix performance as well as tSNE scatter vary with the strength of the neural network.

- **They do not yield strong quantitative results**. The confusion matrix yields some quantitative performance results, but they are a function of the test data. If test data that is highly similar or dissimilar to the training data, the performance metric will vary.

The BER estimate addresses these issues. As previously discussed, it characterizes an innate quality in the training data, independent of the architecture. Furthermore, it gives a numerical value for the separability of data instead of a simply visual explanation (tSNE) or varying performance metrics based on test data (confusion matrices). While the metric has many strengths, **it cannot be directly calculated** for many distributions of data, especially complex distributions from which image data is sampled. Thus, it is a quantity that must be estimated.

6.1 Estimating the Bayes Error Rate

There have been many ways to estimate the Bayes error rate, and many of them rely on divergence measures between distributions. The Bhattacharyya distance [14], Chernoff Bound [15], and other divergence measures provide some bounds for the BER, but they happen to be parametric estimates that require the assumption of a probability distribution before the estimation.

In Sekeh et. al. [16], a non-parametric estimate of the Bayes error is introduced:

$$\mathbb{R}^{(i,j)}_{n_i,n_j} = \mathbb{R}^{(i,j)}_{n_i,n_j}(X)$$

Where $\mathbb{R}^{(i,j)}_{n_i,n_j}$ is the number of cross-labels connecting class i to class j in a global minimum-spanning tree connecting all samples with all classes. With this, the authors show that this statistic is an estimator of δ^m_{ij}, which bounds the BER:

$$\frac{\mathbb{R}^{(i,j)}_{n_i,n_j}(X)}{2n} \to \delta^m_{ij}$$

With this, the an upper- and lower-bound for the error rate is derived:

$$\frac{m}{m-1}\left[1 - \left(1 - \frac{2m}{m-1}\sum_{i=1}^{m-1}\sum_{j=i+1}^{m}\delta^m_{ij}\right)^{\frac{1}{2}}\right] \leq \epsilon^m \leq \sum_{i=1}^{m-1}\sum_{j=i+1}^{m}\delta^m_{ij}$$

References

[1] Blumer, A. et. al. (1989). *Learnability and the Vapnik-Chervonenkis Dimension* .

[2] Kuzovkin, I., Vicente, R., Petton, M. et al. (2018). *Activations of deep convolutional neural networks are aligned with gamma band activity of human visual cortex*.

[3] Ioffe & Szegedy. (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*.

[4] Ulyanov, Vedaldi, & Lempitsky (2016). *Instance Normalization: The Missing Ingredient for Fast Stylization*.

[5] Póczos, B. & Tibshirani, R. (2013). *Convex Optimization*. Carnegie Mellon University. School of Computer Science. Department of Machine Learning.

[6] Tibshirani, R. (2017). *Quasi-Newton Methods*. Carnegie Mellon University. Department of Statistics.

[7] Gatys et. al. (2016). *Image Style Transfer Using Convolutional Neural Networks*.

[8] Gulrajani et. al. (2017). *Improved Training of Wasserstein GANs*.

[9] Frogner et al. (2015). *Learning with a Wasserstein Loss*.

[10] Givens, Shortt. (1984). *A Class of Wasserstein Metrics for Probability Distributions*.

[11] Andrieu, C. et. al. (2003). *An Introduction to MCMC for Machine Learning*.

[12] Hastings W.K. (1970). *Monte Carlo Sampling Methods Using Markov Chains and Their Applications*.

[13] Valens, C. (1999). *A Really Friendly Guide to Wavelets*.

[14] Schweppes, F. (1967). *On the Bhattacharyya Distance and the Divergence between Gaussian Processes*.

[15] Impagliazzo, R. & Kabanets, V. (2010). *Constructive Proofs of Concentration Bounds*.

[16] Sekeh et. al. (2019). *Learning to Bound the Multi-class Bayes Error*.