1 Introduction

1.1 The Framework of Neural Networks

A single layer of width k in a neural network is given by $g : \mathbb{R}^n \times \mathbb{R}^{nk} \times \mathbb{R}^k \to \mathbb{R}^k$, where

$$g(a, W, \Theta) = \sigma(Wa + \Theta),$$

where W is a k by n matrix of weights, Wa is a matrix vector product, and Θ is a vector of offsets, and σ is a non-linearity that acts in a componentwise fashion. Typical choices of σ include

$$\begin{aligned} \sigma(x) &= \frac{1}{1+e^{-x}}, && \text{Sigmoid,} \\ \sigma(x) &= \tanh(x), && \text{Hyperbolic Tan,} \\ \sigma(x) &= \max(0, x), && \text{Rectifier Linear Unit.} \end{aligned}$$

To make a neural network, we can compose a finite sequence of such functions $\{g_i\}$ if we suppose that $n_{i+1} = k_i$. Note that the scalar function σ_i may change between layers. Letting g_i be the ith layer, with weights W_i and offsets Θ_i, a neural network with H layers is given by

$$N(a, W, \Theta) = g_H(g_{H-1}(\ldots g_2(g_1(a, W_1, \Theta_1), W_2, \Theta_2)\ldots), W_H, \Theta_H).$$

Neural networks are extremely effective at various machine learning tasks, including image classification. In this setting, the output of the neural network is typically a vector of length m of non-negative entries summing to one, representing a probability distribution on m image classes. This probability distribution is the conditional probability of each class given the input image a. Given some training data of properly labelled images $\{a_j, l_j\}$, we show those to the neural network, and then look for weights and offsets which makes the network match closely the training data. Before talking about how this is done, let us first establish when this is possible. Is it possible to find W and Θ such that $a \mapsto N(a, W, \Theta)$ approximates an image classification function? We will instead answer a more general question; is it possible for a neural network $a \mapsto N(a, W, \Theta)$ to approximate an arbitrary continuous function on a compact set?

1.2 The Expressive Power of Neural Networks

It is obvious that our stated goal is not possible for certain choices of function σ_i. In particular, if $\sigma_i = Id$ for all i, then N becomes a linear function of a, and the answer to our question is no. Fortunately, there are some good theorems on this subject:

Theorem 1 ([3]). Let $K \subset \mathbb{R}^n$ be compact. Consider the set $\Sigma = \text{span}\{\sigma(Wa + \Theta) \mid W \in \mathbb{R}^{m\times n}, \Theta \in \mathbb{R}^m\}$ where $a \in K$. Then Σ is dense in $C(K; \mathbb{R}^m)$ if and only if σ_1 is not almost everywhere a polynomial.

Remark: The theorem pertains to a two layer network with $\sigma_2 = Id$, and σ_1 unspecified. It says that, provided we take a large enough first layer, a 2 layer neural network can approximate any continuous function if and only if σ_1 is not almost everywhere a polynomial.

Remark: Here is a sketch of the proof for the case $m = n = 1$ and σ smooth. The set Σ then becomes

$$\Sigma = \text{span}\{\sigma(wa + \theta) \mid w, \theta \in \mathbb{R}\}.$$

I will demonstrate that $\bar{\Sigma}$ (closure taken in supremum norm) contains all polynomials in a. Indeed, we observe that

$$\frac{\sigma((w+h)a + \theta) - \sigma(wa + \theta)}{h} \in \Sigma,$$

for all h. Since K is compact, we observe that

$$\frac{d}{dw}\sigma(wa + \theta) \in \bar{\Sigma},$$

and, iterating this argument,

$$\frac{d^k}{dw^k}\sigma(wa + \theta) \in \bar{\Sigma}.$$

Selecting $w = 0$, we obtain $a^k\sigma^{(k)}(\theta) \in \bar{\Sigma}$ for all k, and since σ is not almost everywhere a polynomial, for all k there exists $\theta \in \mathbb{R}$ such that $\sigma^{(k)}(\theta) \neq 0$. This demonstrates the utility of the offset parameter, and shows that $\bar{\Sigma}$ contains all polynomials in the variable a, and hence $\bar{\Sigma} = C(K, \mathbb{R})$. There is also a version of this density result for $L^p(K)$ functions, which may be more useful for image classification.

Remark: This theorem holds for neural networks with 2 layers. Most neural networks used today for machine learning tasks have many more than 2 layers, but the extension to this case is clear; let $\sigma(W, wa + \theta)$ be an expression for the output of a neuron in the penultimate layer, where a is scalar. Here w, θ are the weights for the first layer, and W represents all other weights contributing to the output of that neuron. Then the same result holds for the span of these functions, provided $\theta \mapsto \sigma(W, \theta)$ is not a polynomial.

1.3 Training Neural Networks

Now that we know that a simple network can approximate a wide range of functions, we may ask how to obtain the weights and offsets accomplishing this for an image classification task. The ideal weights and offsets should allow us to discriminate between the image classes we wish to feed to the neural network. One idea to select these by hand, however this is difficult, as it is not clear how to select the weights to make it clear when, for example, a dog appears in an image. In applications neural networks are used to automatically learn good weights and offsets from training data through gradient descent. Let us now write W to represent both the weights and offsets. Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be the image classification function we are trying to approximate; if image a is in class i, then $f(a) = e_i$. The error between the predictions of our network and the true function on a labelled data set $\{a_j, f(a_j)\}_{j=1}^N$ can be calculated as

$$l(W) = \frac{1}{N}\sum_{j=1}^{N} ||f(a_j) - N(a_j, W)||^2.$$

We wish to decrease the value of l, and so it is natural to train this network via gradient descent. We take a trajectory $W(t)$ satisfying

$$\frac{dW(t)}{dt} = -\nabla_W l(W(t)),$$

or, in a discrete time approximation

$$W(t_{k+1}) = W(t_k) - \eta \nabla_W l(W(t_k)),$$

where η is a hyperparameter known as the learning rate. Provided η is small enough, this update is guaranteed to decrease the value of l until a critical point is reached. The hope is that at this critical point the network closely approximates the desired function.

1.4 What can go wrong

One would hope that gradient descent allows a network to distinguish image classes in the same way that a human would. Experiments show that this is is not the case, however. For example, consider training a neural network to recognize the digits 0 to 9 from the MNIST data set. This data set consists of 28x28 greyscale images of handwritten digits; it is a classical machine learning task to train a neural network to automatically classify these images. The authors of [4] trained a 784-100-10 network on this task with sigmoid activation functions (the numbers indicate the neurons in each layer, with $28^2 = 784$ input neurons, 100 intermediate neurons, and 10 neurons in the final layer, one for each of the classes). If the standard MNIST data is used with $\eta = 1$, the error decreases steadily. However, if the data is manipulated via a black-white inversion (i.e. applying the transformation $x \mapsto 1 - x$ to the pixel values), the error stays roughly constant, with small perturbations. This is not ideal, because it is just as easy to distinguish the characters in the regular data set as it is in the inverted data set. To analyse this, let us consider a scalar neural network with scalar input a. The gradients for the standard data and transformed data are

$$a\sigma'(aw + \theta) \qquad (1-a)\sigma'((1-a)w + \theta),$$

and so it makes sense that we get different error trajectories, since the gradients are different. The problem here is that our approach depends on the numerical representation of the data, and not necessarily the information it carries. The natural solution to this problem is to view the two sets of data as different parametrizations of the same underlying object. This naturally brings in the concept of a manifold.

1.5 Neural Networks on Manifolds

We now consider our neural network as a map $N : A_i \times M \to A_0$, where A_i, M and A_o are smooth manifolds representing the input activation manifold, weight manifold, and output activation manifold, respectively. The neural network we were working with earlier in this document is now a representation of N in a particular set of coordinates. If we put a metric on the output activation manifold, we can define a real valued error function $l : M \to \mathbb{R}$. To define a gradient, we will assume that M is a Riemannian manifold, with metric g. The instantaneous rate of change of l along a path γ is given by

$$\begin{aligned}\frac{d}{dt} l(\gamma(t)) &= dl(\gamma(t)) \cdot \gamma'(t), \\ &= \langle \nabla_g l(\gamma(t)), \gamma'(t) \rangle_{\gamma(t)},\end{aligned}$$

where $\nabla_g l$ is the Riemannian gradient. By Cauchy Shwarz, the tangent vector $\gamma'(t)$ where this inner product is minimized is $\gamma(t)' = -\nabla_g l$. Thus, to follow the direction of steepest descent of l on the manifold, we should consider the trajectory

$$\gamma'(t) = -\nabla_g l(\gamma(t)). \tag{1}$$

In coordinates w, with coordinate chart φ, it is not difficult to show that

$$\nabla_g l(\varphi^{-1}(w)) = G^{-1}(w) \nabla l(\varphi^{-1}(w)),$$

where the second ∇ here is the regular gradient of l in these coordinates, and G is the metric in these coordinates. Hence, the discrete time trajectory becomes

$$W(t_{k+1}) = W(t_k) - \eta G^{-1}(W(t_k)) \nabla l(\varphi^{-1}(W(t_k))).$$

Comparing this to above, we observe that the classical gradient descent implicitly assumes that the coordinates we have selected are orthonormal. Our new approach is intrinsic at the level of equation (1), however, since the descent is defined using intrinsic properties. This guarantees that if N and $\tilde{N}$ are different parametrizations of the same neural network, i.e.

$$N(a, W) = \tilde{N}(a, \tilde{\varphi}(\varphi^{-1}(W))),$$

and $W(t)$ and $\tilde{W}(t)$ are the resulting trajectories under Riemannian gradient descent starting at W_0 and $\tilde{W}_0$ respectively, then

$$l(\varphi^{-1}(W(t))) = l(\tilde{\varphi}^{-1}(\tilde{W}(t))).$$

This holds because $W(t)$ and $\tilde{W}(t)$ are just different coordinate representations of the same ODE on the abstract weight manifold.

This formulation tells us how to maintain the same error trajectory under reparametrizations of the weight space. Our motivating problem (inverting MNIST) was a re-parametrization of the activation space, so how do we deal with this? Observe the following

$$\begin{aligned}(1-a)w + \theta &= -aw + (w + \theta), \\ &= a\tilde{w} + \tilde{\theta}.\end{aligned}$$

This makes it clear that inverting the data has the same affect as applying the following reparametrization to the weight/offset space

$$(w, \theta) \mapsto (-w, w + \theta) = (\tilde{w}, \tilde{\theta}).$$

As such, the inversion of MNIST has the same effect as re-parametrizing the weight space. In general, any affine transformation on the activation space will have the same effect as the dual transformation on the weight space. It is now clear how to eliminate the problem that arises when MNIST is inverted; we need only decide on a Riemannian metric in our standard set of coordinates, and then ensure that we update this metric to reflect the re-parametrization $(w, \theta) \mapsto (-w, w + \theta)$ after the inversion occurs. How do we select this metric? We could declare that for the standard MNIST data, the regular coordinates of $\mathbb{R}^k$ (k is number of weights) are orthonormal, as we do in traditional gradient descent. However, this is totally arbitrary; we would like a metric which does not have an obvious orthonormal coordinate system. I will now motivate such a metric. It is one of several described in [5].

2 The Outer Product Metric

Our error function puts greater emphasis on data points where the predicted class from the network is very different from the label. Observe

$$\nabla l(w) = \frac{1}{N} \sum_{j=1}^{N} \nabla l(a_j, w) = -\frac{2}{N} \sum_{j=1}^{N} (f(a_j) - N(a_j, W)) \nabla_W N(a_j, W),$$

and thus the terms where $f(a_j) - N(a_j, W)$ is large tend to dominate. This may put too much emphasis on outliers, causing the network to over fit. Thus, we would prefer a minimization scheme which spreads out the improvement amongst the data set more equitably. Formally, let m be the instantaneous rate of change of the error function. Let $W(t)$ be a weight trajectory with initial velocity v, which has m as its instantaneous rate of change in error, i.e.

$$m = \nabla_W l(W) \cdot v = \frac{1}{N} \sum_{j=1}^{N} \nabla l(a_j, w) \cdot v.$$

Note that this implies that m is the average instantaneous rate of change on over the data points. Of all v satisfying this equation, we wish to find one minimizing the variance between the rate of change on each data point and the average improvement. In other words, minimize

$$\frac{1}{N} \sum_{j=1}^{N} (\nabla_W l(a_i, W) \cdot v - m)^2 = \frac{1}{N} \sum_{j=1}^{N} (\nabla_W l(a_i, W) \cdot v)^2 - m^2.$$

In this setting, m is the average improvement over the entire data set. This is a quadratic minimization problem with linear constraints, and hence is easy to solve. The associated Lagrangian is

$$L(v, \lambda) = v^T Q(W) v - m^2 + \lambda(\nabla_W l(W) \cdot v - m),$$

where $Q(W) = \frac{1}{N}\sum_{j=1}^{N} \nabla_W l(a_j, W)\nabla_W l(a_j, W)^T$. Hence, a solution to this minimization problem solves

$$Q(W)v = -\lambda \nabla_W l(W),$$

for some λ. If $Q(W)$ is invertible, we have

$$v = -\lambda Q(W)^{-1}\nabla_W l(W),$$

and therefore the update is

$$W(t_{k+1}) = W(t_k) - \eta Q(W(t_k))^{-1}\nabla_W l(W(t_k)).$$

Comparing this to equation (1), we see that this would be the update for gradient descent on a Riemannian weight manifold with metric

$$G(W) = \frac{1}{N}\sum_{j=1}^{N} \nabla_W l(a_j, W)\nabla_W l(a_j, W)^T. \tag{2}$$

This is the outer product metric in W coordinates, called so because it is a sum of outer products. Clearly, this matrix is positive semi-definite. We may re-write it as

$$G(W) = \frac{1}{N}UU^T, \qquad U = \left[\nabla_W l(a_1, W), \dots, \nabla_W l(a_N, W).\right]$$

Note that U is $|W| \times N$, and hence U^T is $N \times |W|$. This makes it clear that G is in fact a degenerate Riemannian metric whenever there are more weights than data points, which is common in applications (though not always, especially if dataset augmentation is used). This is not a serious problem even though the inverse of G appears in the gradient descent formula; a degenerate metric simply gives a non-unique solution to the quadratic minimization problem we defined above. We have proven the following proposition, from [5]:

Theorem 2. *Among all directions v yielding the same instantaneous rate of change of the error function, a direction provided by gradient descent using the outer product metric spreads out the improvement most equitably (in the l^2 sense) among the data points.*

2.1 Training Using Riemannian Gradient Descent Depends only on Linear Span of the Activations

In our motivating example, we showed how to standardize training of a neural network under affine changes to the input activations. This section generalizes this idea by showing that the same can be done for simultaneous affine transformations of the activations in every layer. More specifically, when we use Riemannian gradient descent, the error trajectory depends only on the span of the activations at each layer of the network. This allows us to explore a variety of network architectures by simply varying the initial point of the descent.

There is one notational change to include here; assume that each layer has a neuron with constant value 1. This allows us to write the inner product plus offset operation as a simple inner product, and affine transformations applied to the original activations become linear transformations on the new activations.

Theorem 3. *Take l as the error function for a neural network N. Take $\tilde{l}$ as the error function for a neural network $\tilde{N}$ which has identical architecture to N, with the exception that an invertible linear transformation is applied to the activations of each layer before they are fed to the next layer. Suppose that these networks are trained by Riemannian gradient descent. Then for the error trajectory $\tilde{l}(\tilde{W}(t))$ starting at $\tilde{W}_0$, there is a corresponding initialization for the standard network W_0 producing a weight trajectory $W(t)$ satisfying*

$$l(W(t)) = \tilde{l}(\tilde{W}(t)).$$

Proof. Since the weights are always combined with the activations via a dot product, and the linear transformations used to define $\tilde{N}$ are invertible, there is a linear transformation B on the weight space which reverses the linear transformations involved in $\tilde{l}$. In other words,

$$l(W) = \tilde{l}(BW).$$

As such, $\tilde{l}$ is simply a reparametrization of l, and so define $\tilde{W} = BW$. For the continuous time version of gradient descent, the error trajectories are therefore the same for both networks. The rest of this proof is dedicated to showing that the discrete time approximations are actually identical as well. The update for l is given by

$$W \mapsto W - \eta G^{-1}(W)\nabla_W l(W).$$

The update for $\tilde{l}$ in the $\tilde{W}$ variables is given by

$$\tilde{W} \mapsto \tilde{W} - \eta \tilde{G}^{-1}(\tilde{W})\nabla_{\tilde{W}} \tilde{l}(\tilde{W}).$$

Observe that

$$\begin{aligned}\nabla_{\tilde{W}} \tilde{l}(\tilde{W}) &= \nabla_{\tilde{W}} l(B^{-1}\tilde{W}),\\ &= (B^{-1})^T \nabla_W l(W).\end{aligned}$$

Also, we have

$$\tilde{G}(BW) = (B^T)^{-1} G(W) B^{-1},$$

and therefore

$$\tilde{G}^{-1}(BW) = B G^{-1}(W) B^T,$$

and hence

$$\begin{aligned}\tilde{W} - \eta \tilde{G}^{-1}(\tilde{W})\nabla_{\tilde{W}} \tilde{l}(\tilde{W}) &= BW - \eta B G^{-1}(W) B^T (B^{-1})^T \nabla_W l(W),\\ &= B(W - \eta G^{-1}(W)\nabla_W l(W)),\end{aligned}$$

which we recognize as the update for l modified by B. As such, we can compare the error after the update as

$$\begin{aligned}l(W - \eta G^{-1}(W)\nabla_W l(W)) &= \tilde{l}(B(W - \eta G^{-1}(W)\nabla_W l(W))),\\ &= \tilde{l}(\tilde{W} - \eta \tilde{G}^{-1}(\tilde{W})\nabla_{\tilde{W}} \tilde{l}(\tilde{W})).\end{aligned}$$

Hence, the error for each network after the first update is identical. It is easy to see that this continues to hold for subsequent updates. Therefore, if we initialize the modified network at $\tilde{W}_0$, we will get the same learning trajectory if we initialize the standard network at $B^{-1}\tilde{W}_0$. □

Remark: This theorem says that if one uses an intrinsic metric like the outer product metric, we can explore many network architectures by fixing the network architecture and varying the initial point. The trouble is that very few practitioners use intrinsic Riemannian metrics; as stated above, the classic approach tacitly assumes the base coordinates are orthogonal, so the metric is trivial. All is not lost, however. One neat way to interpret this result is to assume that $\tilde{G}$ is the identity matrix. This is what a researcher does when they come up with a new network architecture, and then optimize it using gradient descent, Then, we are effectively doing vanilla gradient descent on the tilde variables, and this shows that we can find the same error trajectory by modifying the initial point, and taking the update

$$W \mapsto W - \eta B^{-1}(B^{-1})^T \nabla_W l(W). \tag{3}$$

So, if the tilde variables are such that vanilla gradient descent works extremely well, then we can accomplish the same thing by updating with equation (3). A variety of methods in use can be explained using this approach.

a) **Data Whitening**: Training data is sometimes whitened before it is fed to a neural network. This means that the mean of each pixel in the training data is subtracted from each data point, and the covariance matrix is set to the identity, all of which forms an affine map on the input activations. More specifically, assuming that we have subtracted the mean value of each pixel from the data set, the covariance matrix of the input vector X has ijth component

$$\mathrm{Cov}(X)_{ij} = E[X_i X_j],$$

and assuming that $\mathrm{Cov}(X)$ is non-singular, we can transform the input data by multiplying by a matrix A satisfying

$$A^T A = \mathrm{Cov}(X)^{-1},$$

to obtain transformed data $\tilde{X} = AX$ which has covariance matrix equal to the identity. Apparently networks tend to train faster if you first whiten the data [1], and this analysis here shows that the same can be done by using the original data and a non-trivial Riemannian metric. This metric should actually be easy to calculate. Indeed, assuming for simplicity that X already has mean 0, then whitening the data has the same effect as multiplying the input weights by A^T. Hence, $\tilde{W} = A^T W$. As such, the matrix B given above is equal to A^T in its first $n_1 \times n_1$ block, and is the identity everywhere else. Hence,

$$B^{-1}(B^{-1})^T = A^{-T}A^{-1} = \mathrm{Cov}(X).$$

b) **Batch Normalization**: It can also be shown that, under some assumptions, batch normalization ([2]) is a reparametrization of a standard network and can thus be obtained by using a standard network architecture and a non-trivial Riemannian metric.

Since these highly effective techniques can all be obtained by using a non-trivial Riemannian metric on the weight space, it is intriguing to formulate a search for an optimal Riemannian metric in which gradient descent would behave extremely well.

References

[1] *Efficient Backprop, in Neural Networks: Tricks of the Trade*, Springer, 1998.

[2] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, arXiv, (2015).

[3] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, *Multilayer feedforward networks with a nonpolynomial activation function can approximate any function*, Neural Networks, (1993).

[4] G. Marceau-Caron and Y. Ollivier, *Practical riemannian neural networks*, preprint, (2016).

[5] Y. Ollivier, *Riemannian metrics for neural networks I: feedforward networks*, Information and Inference, (2015).