Artificial Neural Network Architecture and Optimization

Leon T Johnson

Spring 2018

1 Introduction

In this paper, I'll discuss the architecture of Artificial Neural Networks (ANNs) and Deep Neural Networks (DNNs). Then, I'll show how they may be used in identifying music audio samples. For the sake of notation definition, I'll use bold lowercase letters, e.g., $\boldsymbol{x}$, to represent vectors (letters not in bold, x, will represent scalars), and capital letters, e.g., X, to represent matrices. I'll use bold capital letters, e.g., $\boldsymbol{X}$, to represent tensors. Note, $\boldsymbol{\zeta}_{i,j}$ would be a vector. Of course, $\boldsymbol{v}^T$ (a row vector) will represent $\boldsymbol{v}$ transpose, and $\cdot$ represents the normal dot-product.

2 Understanding Neural Networks

2.1 The Neuron and the Network

2.1.1 The Neuron

We start with the basic building block of Neural Networks, the neuron.

Our goal here is to have the neuron accept an observation (our input) with three parts $\boldsymbol{x} = (x_1, x_2, x_3)^T$, or more, and make a determination – our output. The determination, or output, will be a number or vector. So, the neuron takes these three inputs, sends them through some function $\alpha(\boldsymbol{x})$, and then outputs a determination, say $y = \alpha(\boldsymbol{x}) \in \mathbb{R}$.

To illustrate, say that our observation was x_1, the weather, x_2, how we feel, and x_3, the time of day. In this example, suppose also that y is the probability that we go outside. Our

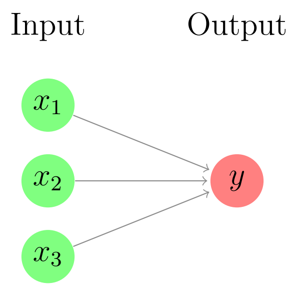

Figure 1: A neuron (in red), which takes three input components, and outputs y.

neuron function, α, accepts $\boldsymbol{x}$, considers the weather, how we feel, and the time of day, and makes a determination as to the probability (or good-idea-ness) that we should go outside. This is a basic neuron.

2.1.2 The Network

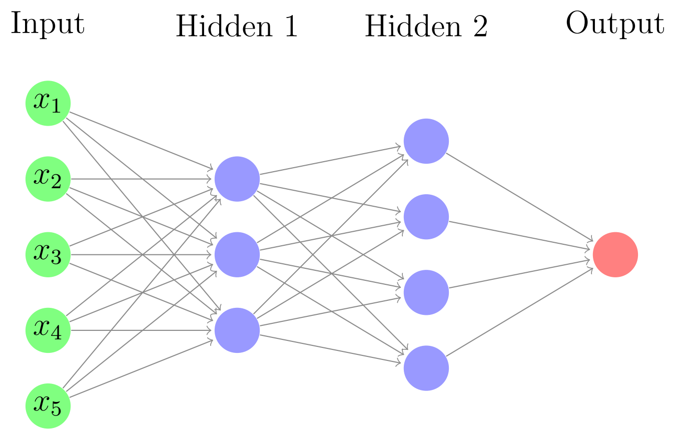

Figure 2: A Fully Connected Feed-Forward Network, i.e., no loops.

Referring to the network above as an example, we have the input $\boldsymbol{x} = (x_1, x_2, ..., x_p)^T$ (the above shows $p = 5$ visually), represented as the first *layer* of neurons, called the *input layer*. The sole purpose of each neuron in the input layer is to output the corresponding component, x_i, of the input, $\boldsymbol{x}$, into the network. So, for a given input neuron i, the value in x_i is then "copied", and sent as an input to each neuron in the second layer called the *hidden layer*. This naming convention, simply put, describes a layer of neurons between the input layer and output neuron. A network with at least two hidden layers is called a *deep* *neural network*, and and these are the backbone of the glamorous (but often misunderstood) AI technique called deep learning.

The hidden layer is difficult to interpret, but it suffices to say that each neuron in a hidden layer makes its own unique determination based on its parameters and its input from the previous layer. One could say that each neuron in a hidden layer is making a "deeper," more abstract decision based on the outputs of the preceding neurons. For example, given a bunch of pixels of a picture, one hidden layer neuron could be determining whether there is an eye in the picture, while another hidden layer neuron could be determining whether there is a nose.

Before moving further, it's worth mentioning that the goal with a deep neural network is to be able to take an input $\boldsymbol{x} = (x_1, x_2, ..., x_p)^T$, and correctly (or at least approximately) classify it as one of m classes, or assign it a value ν. E.g., given numerical representations of p weather patterns $x_1, x_2, ..., x_p$, what action of m actions are we likely to take, or alternatively, what is an estimate value ν of some other weather parameter given these weather patterns.

2.2 The Activation Function

Above, we discussed a neuron which took an observation $\boldsymbol{x}$, sent it through a function $\alpha(\boldsymbol{x})$, and output a determination, $y = \alpha(\boldsymbol{x})$. We call this determination function the *activation function*. The name makes sense if we think of the neuron firing or not (i.e., to activate, or not activate) based on the output of α. Originally, the activation function for a neuron α would take integer inputs $x_1, x_2, ..., x_p$, and produce a boolean output, hence the idea of firing or not. However, in the real world, our observation $\boldsymbol{x}$ or determination y may have many forms. So, the activation function should take multiple forms to accommodate.[1]

2.2.1 Weights or Anchor Vectors

The makeup of α is largely dependent on *weights* (or anchor vectors) $\boldsymbol{w}$, and maybe a *bias* b'. For each neuron in each non-input layer, there will always be as many weights as there inputs to that neuron. So, each weight represents the "importance" of the corresponding

[1]In fact, as we'll see later, the network learns best when the output of α is closer to continuous, so we can detect small changes.

input value to the output, or rather the amount to which they are correlated with the output. A neuron's bias represents a sort of threshold value.

For example, when we introduced the activation function as historically taking integer inputs and outputting a boolean value, α would look something like this:

$$\alpha(\boldsymbol{x}) = \begin{cases} 1, & \boldsymbol{w} \cdot \boldsymbol{x} > b' \\ 0, & \boldsymbol{w} \cdot \boldsymbol{x} \leq b' \end{cases}$$

Where b' is some predefined threshold. In this way, our activation function is a step function $z = \boldsymbol{w} \cdot \boldsymbol{x}$, and the neuron is called a *perceptron*. Given a set of weights, if we input an $\boldsymbol{x}$ that correlates strongly with the neuron's designated output, then $\boldsymbol{x}$'s components will be magnified by the weights, and z will be large compared to our threshold value b'. In this case, the neuron "perceives" a firing situation – "output 1." (Vice versa for $z \leq b'$.)

We could rewrite this activation function as follows

$$\alpha(\boldsymbol{x}) = \begin{cases} 1, & \boldsymbol{w} \cdot \boldsymbol{x} + b > 0 \\ 0, & \boldsymbol{w} \cdot \boldsymbol{x} + b \leq 0 \end{cases}$$

where $b = -b'$ is redefined. This form is less unwieldy, and it comes with advantages as we use alternative activation functions. In general, a neuron's activation function takes the form $\alpha(z)$ where $z = \boldsymbol{w} \cdot \boldsymbol{x} + b$, and α is some real valued, non-linear function.[2] It follows, then, that there are infinitely many activation functions.

It is uncommon that we prefer the outputs for our neurons to be boolean. If we want to make small adjustments to our weights and biases to optimize our network, we'd like to see how the output change a little bit based on small changes in α (from each neuron). To this end, we will concern ourselves with four popular activation functions which satisfy these requirements: ReLU, Leaky ReLU, Softplus, and the Sigmoid. For illustrations sake, Figure 3 contains a plot for each of these functions.

[2]We want α to be non-linear, so we can take advantage of the precision of deep neural networks. Say, refer to Figure 2, and let f, g, h be linear functions. If the outputs of the neurons in the first hidden layer were $\{y_i : y_i = f_i\left(\boldsymbol{w_{1,i}} \cdot \boldsymbol{x} + b_{1,i}\right),\ i = 1,2,3)\}$ then the output of the next layer of neurons could be $\{u_i : u_i = g_i\left(\boldsymbol{w_{2,i}} \cdot \boldsymbol{y} + b_{2,i}\right),\ i = 1,2,3,4)\}$, and similarly, our output could be $z = h\left(\boldsymbol{w_3} \cdot \boldsymbol{u} + b_3\right)$. It's easy to see that z is just a linear function of linear functions, which is linear. If we simply had one neuron that employed this function, there's no benefit gained from multiple layers (let alone multiple neurons). In fact, this one neuron would essentially be doing the work of a standard linear regression model.

2.2.2 The Rectified Linear Unit (ReLU)

In general, the ReLU function (rectifying unwanted linearity with piecewise-ness) is

$$\alpha(\boldsymbol{x}) = \max(0, z(\boldsymbol{x})), \quad z(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} + b$$

For weights $\boldsymbol{w}$ and bias b. This activation function is very popular because of its unchanging gradient of 1 for positive $z(\boldsymbol{x})$ values (we can get a precise understanding of how the output changes with small changes in $\boldsymbol{w}$ and b). We will see later that it is also good because it averts *vanishing gradients* for positive values of $z(\boldsymbol{x})$. On the zero-valued end, we'll see in Section 2.3 that this activation function causes unnecessary neurons to "die" when $z(\boldsymbol{x})$ is often negative (i.e., $\boldsymbol{x}$ is usually minimally correlated). This is a good thing when you consider the efficiency and low processing cost of training neurons in small numbers.

However, the zero-valued end also yields a zero-valued gradient for negative inputs, $z(\boldsymbol{x})$, when it is possible that the neuron is more correlated in general but not for a particular $\boldsymbol{x}_0$. So, say we start training with $\alpha(\boldsymbol{x}_0; \boldsymbol{w}_0, b_0) < 0$. We'll see later that in the case of ReLU, there will be no change in $\boldsymbol{w}_0, b_0$, as the gradient for this iteration will be zero. We'll see further how this causes the neuron to die, and our network to potentially lose precision. That is, for some other data point, it may be that $\alpha(\boldsymbol{x}_1; \boldsymbol{w}_0, b_0) > 0$. We would then have positive activation and non-zero gradient, and we could have updated the weights and biases for a better overall approximation of the final output. Nonetheless, this is a small price to pay with larger networks, and the ReLU is still a more popular activation function.

2.2.3 The Leaky ReLU

The so called Leaky ReLU function is

$$\alpha(\boldsymbol{x}) = \begin{cases} z(\boldsymbol{x}), & z(\boldsymbol{x}) > 0 \\ \gamma z(\boldsymbol{x}), & z(\boldsymbol{x}) \leq 0 \end{cases}$$

where $z(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} + b$, and γ is some small real-valued minimizing factor such as $\gamma = 0.01$. Fairly quickly, we can see how this resolves some of the issues with the typical ReLU function. We can still discriminate between observations which are not strongly correlated with the output, and we avoid losing possibly valuable neurons.

However, the the issue with this method, which it shares with its predecessor, is that the function is not differentiable at $z(\boldsymbol{x}) = 0$. So, if for any reason the weights and bias give us a perfectly zero-valued output, it will become clear later that our net could not update this neuron further. The chances of this are very slim, but it is ostensibly still a risk.[3]

2.2.4 The Softplus

Before moving on, the Softplus gets its name from a variation on the ReLU function. If we let $z = z(\boldsymbol{x})$, then the ReLU function is often written as z^+. A "softened" version of this function removes the sharp point at zero as follows:

$$\alpha(\boldsymbol{x}) = \ln\left(1 + \exp[z(\boldsymbol{x})]\right)$$

Here, we have an activation function whose gradient is never zero, so we won't lose neurons, and it is everywhere differentiable. For small negative $z(\boldsymbol{x})$ values, we can still get a good amount of use out of the neuron, and our updates will be meaningful. Additionally, it maintains a fairly constant gradient for positive $z(\boldsymbol{x})$ values, so we can be sure that changes to these neurons are precise when correlation is high.

However, even though the function is not zero, we still have a similar issue with negative $z(\boldsymbol{x})$ values. It becomes very difficult to significantly improve weights and biases when they yield a negative $z(\boldsymbol{x})$ value. We will understand better how this is so later, but this phenomenon is called *vanishing gradients*. With vanishing gradients, changes in weights and and biases are only very small, so the overall effect to the output is often close to zero. In this way, the neuron reaches a point where it cannot significantly be updated or optimized.

In any case, all three of the above methods are fairly popular and very useful for estimating values associated with a given input. Given the advantages and disadvantages of them all, a scientist can determine which is best for the given situation.

[3]Throughout this paper, I've been using terms like "learn," "train," "update," etc. Recall that the purpose of a neural network is to classify or assign some value to a new observation $\boldsymbol{x}$. To be able to do this accurately, we need to tune the model (our network). And, to tune it, we use data (a bunch of observations, $\boldsymbol{x}_i$) with known classifications or values, and we make changes to our weights and biases so that the network is at least correctly classifying the data we know. This will be discussed more in depth later, but it's worth mentioning to keep us on track.

2.2.5 The Sigmoid

Where the above three methods are beneficial if we want to estimate a numerical value assigned to an observation, the Sigmoid function is nice when we want to correctly classify an observation as one or more of m classes (e.g., a 'yes' or 'no' question is seen as $m = 2$). The Sigmoid function is

$$\sigma(\boldsymbol{z}) = \frac{1}{1 + \exp[-\boldsymbol{z}]}$$

Quite immediatley, we can see that for large values of $z(\boldsymbol{x})$ when our neuron is highly correlated with the output, $\alpha(\boldsymbol{x})$ becomes closer to 1, which makes intuitive sense. Analogously, toward 0, for the opposite.

The vanishing gradients on either end of this function can be a hazard, but in a way, they confine the output where we want it. That is, between about -2 and 2, even small changes in $z(\boldsymbol{x})$ can drastically change α. In this way, correlations, if they exist, will tend toward 1, and weak correlations will tend toward 0, which is what we want. Basically, the activation function can be seen as decisive, or clearly distinct.[4]

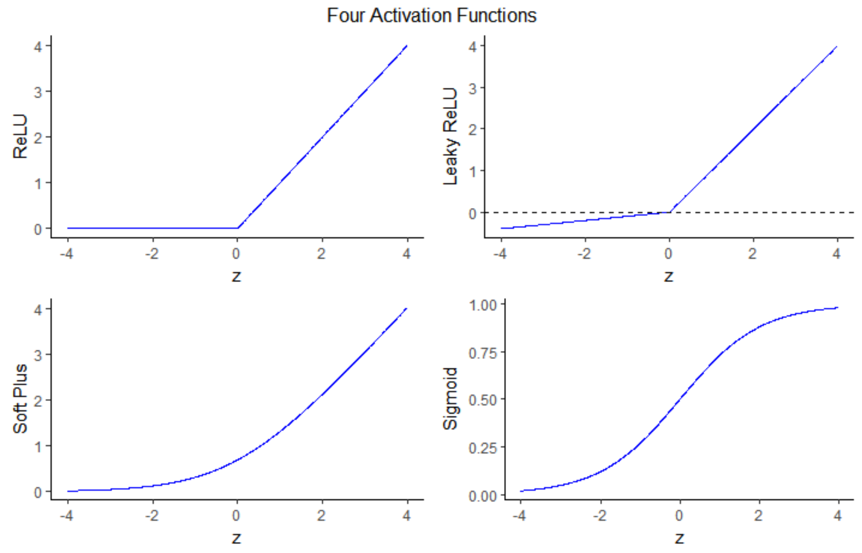

Figure 3: Four Activation Functions, as discussed.

[4]Another popular activation function is $t_1(z) = \tanh(z)$, or alternatively $t_2(z) = \frac{1}{2}(\tanh(z) + 1)$ which behaves virtually the same as the sigmoid, but in a more drastic way. Recall also that $\sigma(z) = \frac{1}{2}(\tanh(z/2)+1)$. So, we could write a general form of the "sigmoid" function as $t_c(z) = \frac{1}{2}(\tanh(cz) + 1)$ for some constant c.

2.3 Optimization

Again, we want our neural network to afford us the ability to classify or approximate some unknown value $\boldsymbol{y}$ given a corresponding observation $\boldsymbol{x}$. To be able to do this, we need data with known outcomes. With this data, we can "train" our model (the network) by changing weights and biases, such that the correlation determinations yield accurate overall classification (or value assignment) at least on this "training set" of data. For instance, suppose we are to expect inputs (or observations) with p components, $\boldsymbol{x} = (x_1, x_2, ..., x_p)^T$. We want to be able to take this input and either correctly classify it as one of $m > 0$ classes, or we want to approximate some numerical value assigned to the observation.

Now, suppose also that we're given a set of n observations from the same population whose assignment/classification has already been established. These observations could be $\boldsymbol{x}_i = (x_{i1}, x_{i2}, ..., x_{ip})^T$ and their assignment/classification could be $\boldsymbol{y}_i = (y_{i1}, y_{i2}, ..., y_{ir})^T$ for $i = 1, 2, ..., n$. That is, the final output for any $\boldsymbol{x}_i$ could be a vector of dimension r. The idea of optimization is to use this *training set* of observations to optimize the activation functions in our neural network so that any input $\boldsymbol{x}$ into the network will yield a result very close to what would be the real result.

2.3.1 *Notation

Before describing the cost function, and subsequently discussing the mechanics of neural net optimization, it makes sense to define the notation which will be used. We're given the training set $\mathscr{T} = \{\boldsymbol{x}_i : i = 1, 2, ..., n\}$, as described above, where each observation has a corresponding expected outcome $\boldsymbol{y}_i = (y_{i1}, y_{i2}, ..., y_{ir})^T$ (in the below network, we have $r = 2$).

In a network, π, the input vector $\boldsymbol{x}_i = (x_{i1}, x_{i2}, ..., x_{ip})^T$ is represented as a the *input layer* (in Figure 4, $p = 5$). Each node x_{ij} in the input layer "outputs" the jth component of $\boldsymbol{x}_i$. Any network will have L layers. The activation function (output) from the jth node in the lth layer is represented by scalar α_{lj}. We can think of the activation "function" for the input layer as $\alpha_{0j} = x_{ij}$. The link between the kth node in the $(l-1)$th layer to the jth node in the lth layer is weighted $w_{jk}^{(l)}$ (e.g., the blue link in Figure 4 is weighted $w_{32}^{(2)}$), and the jth node in the lth layer has a bias $b_j^{(l)}$ or simply b_{lj}. There are $\ell(l)$ nodes in layer l.

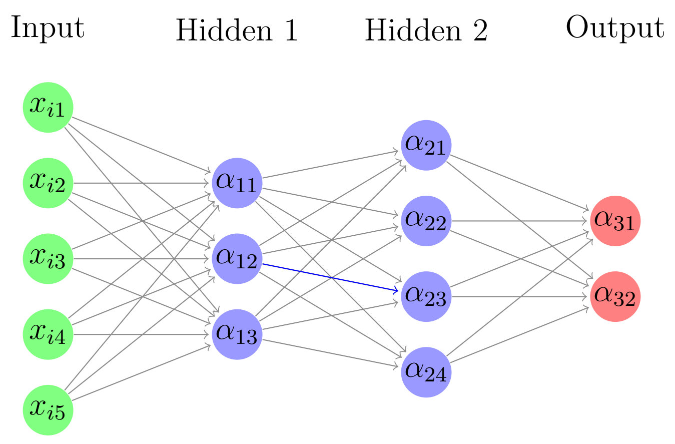

Figure 4: Our example network. Note the blue link.

More often than not, in this paper we'll refer to these values in their vector/matrix form. The vector containing all the output activations for layer l is $\boldsymbol{\alpha}_l$ where $[\boldsymbol{\alpha}_l]_j = \alpha_{lj}$. We represent all the weights for layer l in the matrix W_l, where each element in the matrix $[W_l]_{jk} = w_{jk}^{(l)}$ (e.g., the dimension for W_1 in the example above would be 3×5). Row j of W_l, or $\boldsymbol{w}_{lj}^T$, contains the weights assigned to the $\ell(l-1)$ incoming links from the previous layer into the jth node of layer l. All the biases in the lth layer are represented in $\boldsymbol{b}_l$.

Define the "weighted input" $\boldsymbol{z}_l$ so that the jth element is $z_{lj} = \boldsymbol{w}_{lj} \cdot \boldsymbol{\alpha}_{l-1} + b_j^{(l)}$. This is the input for α_{lj}, i.e., $\alpha_{lj} = \alpha_{lj}(z_{lj})$. So, going along with the previous discussion on activation functions, the input for one neuron is the output from the previous layer. We write $\boldsymbol{\alpha}_l$ as the vector containing the activation outputs for each neuron in the lth layer, so the jth element of $\boldsymbol{\alpha}_l$ is $\alpha_{lj}(z_{lj})$. So, we can redefine the weighted input $\boldsymbol{z}_l$ for the lth layer as $\boldsymbol{z}_l = W_l \boldsymbol{\alpha}_{l-1} + \boldsymbol{b}_l$. Again, this is the input for the vector function $\boldsymbol{\alpha}_l = \boldsymbol{\alpha}_l(\boldsymbol{z}_l)$, where $[\boldsymbol{\alpha}_l]_j = \alpha_{lj}(z_{lj})$.[5] We'll define the vector $\boldsymbol{v} = \left(w_{11}^{(1)}, w_{12}^{(1)}, ..., w_{\ell(L)\ell(L-1)}^{(L)}, b_1^{(1)}, b_2^{(1)}, ..., b_{\ell(L)}^{(L)}\right)^T$ to represent all the weights and biases in neural network (so for the network in Figure 3, $\dim(\boldsymbol{v}) = (5 \cdot 3 + 3 \cdot 4 + 4 \cdot 2) + 9 = 44$). And finally, the output of a network π with weights and biases $\boldsymbol{v}$ and $\boldsymbol{x}$ as an input is $\boldsymbol{\pi}(\boldsymbol{x}; \boldsymbol{v}) = \boldsymbol{\alpha}_L$. We may say that $\boldsymbol{v}_\pi$ corresponds to π.

[5]To be clear, $\alpha_{lj}(z)$ is actually the same activation function $\alpha(z)$ for each node in our network. But, I've chosen the above notation since writing α_{lj} is easier than writing $\alpha(z_{lj})$, and the notation also comes in handy later in the paper.

2.3.2 Cost

Optimization comes by minimizing the error in our neural network. This error or *cost* could be defined in many ways, but it always acts as a representation of the average "difference" between the output(s) from our network's final layer, and the expected outcome(s). For one observation, the cost must meet three guidelines: 1. It must be non-negative, 2: It must approach 0 as the net improves, 3: It must increase if the net becomes less accurate. Note also that in general, the cost function is an average, because we'll see later that optimization is actually reached by minimizing the *average* cost of individual observations in our training set.[6] Suppose our network has L layers. One simple (albeit vanilla) cost function C is the average square of the standard Euclidian norm.[7]

$$C(\boldsymbol{v}) = \frac{1}{n}\sum_{i=1}^{n} \|\boldsymbol{y}_i - \boldsymbol{\pi}(\boldsymbol{x}_i; \boldsymbol{v})\|^2 \tag{1}$$

where $\boldsymbol{\pi}(\boldsymbol{x}_i; \boldsymbol{v}) = \boldsymbol{\alpha}_L$ for our neural network when $\boldsymbol{x}_i$ is the input (in fact, $\boldsymbol{\alpha}_L$ depends on $\boldsymbol{x}_i$ as well as $\boldsymbol{v}$). Sometimes, $C' = C/2$ is used to clean up the derivatives in optimization calculations (i.e., so that interpretation is more clear, as we'll see in Section 2.5), and sometimes $C'' = nC$ is used when n is unknown, or if there is a constant influx of training data. To be sure, there are many cost functions we could employ. When using sigmoid neurons in the output layer, for instance, the *cross-entropy* cost function C^* (see Section 2.5) is often a much better choice than any of these. In this way, we will always choose the cost function based on the activation function of the output layer. Whatever the definition, C is the foundation for optimization or training of our network.

[6] For the remainder of this paper, I'll use $\boldsymbol{x}$ to denote the current observation (e.g., for which we're calculating a cost), and $\boldsymbol{y}$ as its corresponding outcome. In this way, the cost for an individual $\boldsymbol{x}$ would then be C_x, and the cost for one $\boldsymbol{x}_i$ of many would be C_i. In general, we're looking at any and all $\boldsymbol{x} \in \mathcal{T}$.

[7] This is the cost for the whole training set, which means the cost for one individual observation $\boldsymbol{x}$ is $C_x = \|\boldsymbol{y} - \boldsymbol{\alpha}_L\|^2$.

2.3.3 Gradient Descent

With network π and training data $\mathscr{T}$ where there is a known output $\boldsymbol{y}_i$ for each $\boldsymbol{x}_i \in \mathscr{T}$, our aim is fit the function $\boldsymbol{y} = \boldsymbol{\pi}(\boldsymbol{x}; \boldsymbol{v})$ to the data $\mathscr{T}$ by minimizing the cost C by updating parameters $\boldsymbol{v}$. Much like Least Squares Regression, where we fit our linear model to some data by minimizing the sum of squares by optimizing parameters, notice that training a neural network is essentially a non-linear regression problem, since $\boldsymbol{\pi}$ is a non-linear function of $\boldsymbol{v}$. Now, with linear regression, there are not too many parameters, and it is simple to find an explicit equation for each optimized parameter estimate. Neural networks on the other hand can have exponentially more parameters, and each are dependent on one another. Thus, we need an algorithm to find our estimates.

Gradient Descent is a common algorithm for regression problems, and we use it (paired with backpropogation, described in the next section) to make updates to our weights and biases such that each update decreases C (our aggregation of residuals in the regression problem). Since $C \geq 0$, the cost will approach 0. In Eq. 1, the only variable entity is $\boldsymbol{v}$, which consists of all the weights and biases in the network. So, we think of the cost as a function of these weights and biases, and we have the cost $C(\boldsymbol{v})$. For small enough changes $\Delta\boldsymbol{v}$, we can linearly approximate the corresponding change in the cost ΔC with

$$\Delta C \approx \nabla C \cdot \Delta \boldsymbol{v} \tag{2}$$

We want $\Delta C \leq 0$ for each update, so C approaches 0.[8] To ensure this, we make our change $\Delta\boldsymbol{v} = -\eta\nabla C$, for some small *learning rate*, η. Thus, $\Delta C \approx -\eta\|\nabla C\|^2 \leq 0$.

To conceptualize this, think about each component of $\Delta\boldsymbol{v}$ and ∇C. For a small enough change, $\Delta_i C \approx \Delta v_i \, \partial C/\partial v_i$. So, to ensure that $\Delta C < 0$, we just need $\Delta v_i = -\eta \, \partial C/\partial v_i$, for some small η. In other words, we want to make an update to $\boldsymbol{v}$ in the opposing direction to the gradient ∇C, so that the change in C is negative, bringing us closer to the minimum. Further, recall that C is evaluated as the average of the individual observations' costs, and it can be proven that ∇C is then computed by evaluating the gradients ∇C_i for each input $\boldsymbol{x}_i \in \mathscr{T}$, and averaging them.[9]

[8]Think of the C in Eq. 1 as the cost after an update. So, after each update, the cost C will change. Eq. 2 represents the change in the cost after an iteration of updating $\boldsymbol{v}$.

[9]$C = 1/n \sum_{i=1}^{n} \|\boldsymbol{y_i} - \boldsymbol{\alpha}_L\|^2 = 1/n \sum_{i=1}^{n} C_i \;\rightarrow\; \nabla C = \nabla 1/n \sum_{i=1}^{n} C_i = 1/n \sum_{i=1}^{n} \nabla C_i$.

We'll see later just how $\boldsymbol{v}_i$ is dependent on $\boldsymbol{x}_i$. With this in mind, notice that evaluating Eq. 1 (or even Eq. 2 for that matter) could become very computationally costly when n is very large (consider having 10^5 training observations). *Stochastic Gradient Descent* is a method used to cut down on computer usage by applying gradient descent to random subsets $\mathscr{T}_u \subset \mathscr{T}$.

Basically, with stochastic gradient descent, we randomly subset $\mathscr{T}$ into q mutually exclusive and collectively exhaustive subsets $\{\mathscr{T}_u : u = 1, 2, ..., q\}$. We take our first subset $\mathscr{T}_1$, make updates to $\boldsymbol{v}$, then move on to the next subset $\mathscr{T}_2$, make further updates, and continue in this way until $\mathscr{T}$ is exhausted. This completes the first *epoch* of training. We can choose q depending on computing power and time available, and we can complete any number of training epochs we like. Stochastic gradient descent is a widely used technique, and it turns out to perform very well (since not everyone has a quantum computer). So we'll assume its employment for the remainder of this paper.

2.3.4 Backpropogation

As stated earlier, the goal here is to make changes to the vector of weights and biases $\Delta\boldsymbol{v}$, that will bring our cost function C closer and closer to zero. Since we set $\Delta\boldsymbol{v} = -\eta\nabla C$, we need to know the gradient $\nabla C(\boldsymbol{v})$, which means we need the partials $\dfrac{\partial C}{\partial w_*}$ and $\dfrac{\partial C}{\partial b_*}$ for all the weights and biases in our neural network.

It used to be, circa the 1970s, that all the weights and biases in a network were changed by the same amount (i.e., based only on η and the overall gradient), and this could take a very long time. But a 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams, [9], showed that an algorithm called *backpropogation* could make the changes more efficient and optimization more expedient by considering the strength of the correlation between a node's output and the final outcome.

The idea here is that we start with an observation $\boldsymbol{x}$, evaluate the cost C_x (or error E_x, as used in the Rumelhart, et. al. paper, supposing we use sigmoid neurons), and "backpropogate" that cost through the network, determining the effect each node had on the cost, and make changes to the weights and biases accordingly. That is, if a node has a small effect on the cost, we make a smaller change to its weights and biases than if the node has a larger effect. Let δ_{lj} be this *effect* on the cost C_x by the jth node in the lth layer. To determine effect of this node, it makes sense only to analyze the weighted input to the activation function, which is z_{lj}. This gives us a direct route to changes in the weights and biases of that node. Thus, we have

$$\delta_{lj} = \frac{\partial C_x}{\partial z_{lj}}$$

Let $\boldsymbol{\delta}_l$ be the vector containing the effects of each node in the lth layer on the cost C_x, where $[\boldsymbol{\delta}_l]_j = \delta_{lj}$. To proceed with the backpropogation algorithm, we need four fundamental equations:

1. The effect of the last layer $\boldsymbol{\delta}_L$ in terms of the cost function and the final outputs.
2. The effect of layer l in terms of the effect of layer $l+1$.
3. The bias differentials of C_x in terms of the effects.
4. The weights differentials of C_x in terms of the effects.

(Effect of Last Layer) We start with the definition for the effect of the final layer with $\mathscr{d}(L) = r$ nodes

$$\boldsymbol{\delta}_L = \nabla_z C_x = \left(\frac{\partial C_x}{\partial z_{L1}}, \frac{\partial C_x}{\partial z_{L2}}, \dots, \frac{\partial C_x}{\partial z_{Lr}}\right)^T \tag{3}$$

For an individual component or output node, we use the chain rule and the fact that all nodes in this layer are actually employing the same activation function α. Also note that $\frac{\partial \alpha_{Lk}}{\partial z_{Lj}} = 0$ for $k \neq j$. So,

$$\frac{\partial C_x}{\partial z_{Lj}} = \sum_{k=1}^{r} \frac{\partial C_x}{\partial \alpha_{Lk}} \cdot \frac{\partial \alpha_{Lk}}{\partial z_{Lj}} = \frac{\partial C_x}{\partial \alpha_{Lj}} \cdot \frac{\partial \alpha_{Lj}}{\partial z_{Lj}} = \frac{\partial C_x}{\partial \alpha_{Lj}} \alpha'(z_{Lj})$$

Now, define $\circ$ as the Hadamard product, or the element-wise multiplication between two matrices (or vectors, in this case). Then, if $\nabla_\alpha C_x$ is defined by replacing z with α in $\nabla_z C_x$ above, we have

$$\boldsymbol{\delta}_L = \nabla_\alpha C_x \circ \boldsymbol{\alpha}'(\boldsymbol{z}_L) \tag{4}$$

(Effect of Layer l) We start with the definition: layer $l < L$ has r' nodes, and layer $l+1$ has r''. Recall

$$\boldsymbol{\delta}_l = \nabla_z C_x = \left(\frac{\partial C_x}{\partial z_{l1}}, \frac{\partial C_x}{\partial z_{l2}}, \dots, \frac{\partial C_x}{\partial z_{lr'}}\right)^T$$

Similarly, for an individual node, we have by the chain rule[10]

$$\frac{\partial C_x}{\partial z_{lj}} = \sum_{k=1}^{r''} \frac{\partial C_x}{\partial z_{(l+1)k}} \cdot \frac{\partial z_{(l+1)k}}{\partial z_{lj}} = \sum_{k=1}^{r''} \frac{\partial z_{(l+1)k}}{\partial z_{lj}}\, \delta_{(l+1)k} \tag{5}$$

Recalling the equation for $z_{(l+1)k}$, and recognizing $\frac{\partial \alpha_{lk}}{\partial z_{lj}} = 0$ when $j \neq k \in \{1, 2, ..., r'\}$,

$$z_{(l+1)k} = \boldsymbol{w}_{(l+1)k} \cdot \boldsymbol{\alpha}_l(\boldsymbol{z}_l) + b_k^{(l+1)} \quad \rightarrow \quad \frac{\partial z_{(l+1)k}}{\partial z_{lj}} = w_{kj}^{(l+1)} \frac{\partial \alpha_{lj}}{\partial z_{lj}} = w_{kj}^{(l+1)}\, \alpha'(z_{lj})$$

Substituting this back into Eq. 5, we have

$$\frac{\partial C_x}{\partial z_{lj}} = \sum_{k=1}^{r''} w_{kj}^{(l+1)}\, \alpha'(z_{lj})\, \delta_{(l+1)k} = \alpha'(z_{lj}) \sum_{k=1}^{r''} w_{kj}^{(l+1)}\, \delta_{(l+1)k}$$

But, this is only one element of $\nabla_z C_x$ for layer l. So,

$$\boldsymbol{\delta}_l = \boldsymbol{\alpha}'(\boldsymbol{z}_l) \circ \left(W_{l+1}^T \boldsymbol{\delta}_{l+1}\right) \tag{6}$$

(Bias Differentials) To calculate the bias differentials for layer l, recall that

$$\boldsymbol{z}_l = W_l \boldsymbol{\alpha}_{l-1} + \boldsymbol{b}_l = W_l \boldsymbol{\alpha}(\boldsymbol{z}_{l-1}) + \boldsymbol{b}_l \tag{7}$$

So, for the jth node in a layer l, we have $\frac{\partial z_{lj}}{\partial b_{lj}} = 1$ since b_{lj} is not a component of W_l or $\boldsymbol{z}_{l-1}$. Also, note that for all j, $\frac{\partial z_{lk}}{\partial b_{lj}} = 0$ for $j \neq k$. So, if layer l has r nodes,

$$\frac{\partial C_x}{\partial b_{lj}} = \sum_{k=1}^{r} \frac{\partial C_x}{\partial z_{lk}} \cdot \frac{\partial z_{lk}}{\partial b_{lj}} = \frac{\partial C_x}{\partial z_{lj}} = \delta_{lj}$$

Define ∇_b as the gradient in terms of $\boldsymbol{b}_l$, and our bias differentials are contained in the lth effect vector

$$\nabla_b C_x = \boldsymbol{\delta}_l \tag{8}$$

In other words, the effect on the cost of changing the biases of layer l *is* the effect of layer l, and vice versa.

(Weight Differentials) Refer again to Eq. 7. Suppose that there are r nodes in layer l and now r' nodes in layer $l-1$. Note, index k' is different from a particular k.

$$z_{lj} = \boldsymbol{w}_{lj} \cdot \boldsymbol{\alpha}_{l-1} + b_{lj} = \sum_{k'=1}^{r'} w_{jk'}^{(l)} \alpha_{(l-1)k'} + b_{lj} \quad \rightarrow \quad \frac{\partial z_{lj}}{\partial w_{jk}^{(l)}} = \alpha_{(l-1)k}$$

[10]The index k refers to the node in the $l+1$th layer, while the index j refers to the node in the lth layer

Again, we use the chain rule, and that for $j \neq j'$, $\dfrac{\partial z_{lj'}}{\partial w_{jk}^{(l)}} = 0$ (since $w_{jk}^{(l)}$ is not a component of $z_{lj'} \neq z_{lj}$). Using Eq. 8 we have

$$\frac{\partial C_x}{\partial w_{jk}^{(l)}} = \sum_{j'=1}^{r} \frac{\partial C_x}{\partial z_{lj'}} \cdot \frac{\partial z_{lj'}}{\partial w_{jk}^{(l)}} = \frac{\partial C_x}{\partial z_{lj}} \cdot \frac{\partial z_{lj}}{\partial w_{jk}^{(l)}} = \delta_{lj}\alpha_{(l-1)k} \tag{9}$$

Now, for layer l, define $n \times r$ matrix $\boldsymbol{\nabla}_w C_x$ so that $[\boldsymbol{\nabla}_w C_x]_{jk} = \dfrac{\partial C_x}{\partial w_{jk}^{(l)}}$, and using Eq. 9,

$$\boldsymbol{\nabla}_w C_x = \boldsymbol{\delta}_l \boldsymbol{\alpha}_{l-1}^T \tag{10}$$

So, if you think about it, the effect on the cost of changing the weights in a layer is dependent on the effect of that layer and the outcomes of the previous layer. We can also say that the network learns the weights of layer l faster when the effect of a layer is high or the corresponding output of the previous layer is high.

2.3.5 Training the Network

Now, looking at Eqs. 4, 6, 8, and 10, we can proceed with Stochastic Gradient Descent. We call making updates to the weights and biases *training* the net. So, without further ado:

1. Set an initial state for all weights and biases $\boldsymbol{v}_0$, pick learning rate η, and number of epochs.[11]

2. Randomly subset our training data into q subsets $\{\mathcal{T}_u : u = 1, 2, ..., q\}$ as described in Section 2.3.3.

3. (Next u) For each $\boldsymbol{x}_i \in \mathcal{T}_u$, calculate ∇C_{ui} where

$$\nabla C_{ui} = \left(\frac{\partial C_{ui}}{\partial w_1}, \frac{\partial C_{ui}}{\partial w_2}, ..., \frac{\partial C_{ui}}{\partial w_k}, \frac{\partial C_{ui}}{\partial b_1}, \frac{\partial C_{ui}}{\partial b_2}, ..., \frac{\partial C_{ui}}{\partial b_{k'}} \right)^T$$

i.e., there are k weights and k' biases in the network.[12]

3.1. (Next i) Send $\boldsymbol{x}_i \in \mathcal{T}_u$ through (evaluating $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, ..., \boldsymbol{\alpha}_L$ along the way), and evaluate C_{ui} (the cost for x_i)

[11] In the next section, we discuss smart ways to choose the first initial state, learning rate, number of epochs, and other so-called hyperparameters.

[12] Here, every single weight in the network, $w_{**}^{(*)}$, is represented as w_{i} for some $1 \leq \mathrm{i} \leq k$, and similarly every bias b_{**} is represented as $b_{\mathrm{i}'}$ for some $k < \mathrm{i}' \leq k'$, for the sake of simplicity in notation.

3.2. Use Eqs. 4, 6, 8, and 10 to calculate ∇C_{ui}.

3.3. Repeat 3.1 and 3.2 for $x_{i+1}, x_{i+2}, ..., x_m$ so we have ∇C_{ui} for all $i = 1, 2, ..., m$ observations in $\mathscr{T}_u$.

4. Calculate the subset's differentials $[\nabla C_u]_j = \frac{1}{m}\sum_{i=1}^{m}[\nabla C_{ui}]_j$, and update the weights and biases $\Delta w_{\mathrm{i}} = -\eta[\nabla C_u]_{\mathrm{i}}$, and $\Delta b_{\mathrm{i}'} = -\eta[\nabla C_u]_{k+\mathrm{i}'}$. Note, i is different from i.

5. Repeat steps 3 and 4 (updating weights and biases after each iteration), until you get through the final subset $\mathscr{T}_q$.

6. Repeat steps 2-5 for as many epochs as desirable. The net should improve with epochs.

2.4 Deep Neural Networks (DNNs)

So far, we've discussed how neural networks function with fully-connected layers (reference Figure 2). In most cases, a fully connected neural network can yield very good outputs. However, with more neurons and more layers, the calculations can become quite cumbersome, not to mention increasingly difficult to interpret. A *Deep Neural Net* is one with two or more hidden layers. As we said before, these hidden layers can make more abstract determinations on the input data the deeper they go. To keep the DNN from becoming too overbearing on our processor, a few methods have been developed which not only allow us to better interpret the determinations being made in deeper layers, but also to cut down on computing power.

2.4.1 Convolutions

Refer again to the fully-connected network in Figure 2. Here, each neuron in any given hidden layer is receiving the same input from the previous layer, but they all assign their own set of weights and biases. Suppose that our input layer consists of tens of thousands of components, and each hidden layer is comparable in magnitude. A fully connected neural network could become very complicated to train, so we will strategically minimize the amount of unique weights and biases the network needs to worry about.

For the sake of illustration, we'll suppose that our input is a picture with dimension $a \times b$ pixels. The value contained in each pixel will represent an input neuron (e.g., the pixel could be an RGB vector $\boldsymbol{p} = (p_r, p_g, p_b)^T$ for the amount of red, green, and blue in the pixel – in which case we'd have $\boldsymbol{x} \in \mathbb{R}^{3ab}$ – or it could be a gray-scale value $g \in [0, 1]$, $\boldsymbol{x} \in \mathbb{R}^{ab}$).

Our first goal is to task each neuron in our first hidden layer with "looking" for a certain fundamental (least abstract) feature of the input. So, for a picture, maybe this feature is an edge, or a corner, or a blue colored arc (or, for an array of millisecond snippets of a song, this could be a particular frequency pattern, say). This feature will consist of more than one pixel, so the *receptive field* of each hidden neuron must be a rectangle (maybe) of size $s_1 \times s_1$ pixels.[13] In essence, our first layer will be scanning the input neurons, with each neuron responsible for a part of the input matrix. We want to have some overlap, though, because we wouldn't want the feature to exist on the border of two receptive fields. We call this overlap the *stride*. For our example, we'll set the stride $s = 1$. The size of s is largely dependent on the size of our input; too small, and we may be using unnecessary processing power, but too large and we may be missing a feature. To illustrate, we have the first neuron in the first hidden layer looking at the first part of the input matrix:

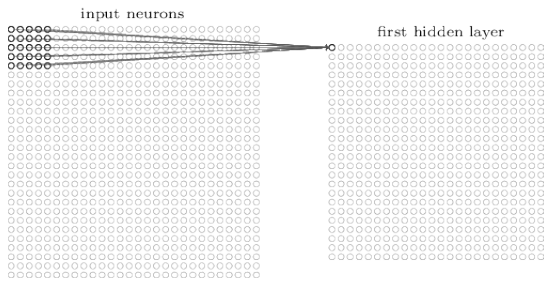

Figure 5: Mapping a filter to the first hidden neuron. (Credit: Nielsen, Michael)

For the next hidden neuron, s_1 remains, but we move over by our stride $s = 1$:

[13]It is a good rule of thumb to keep s_i small enough so that it represents some simple component of the input, and not so large that it can be broken down into multiple parts. It's also worth noting that in some literature, *kernel* or *filter* is used instead of receptive field.

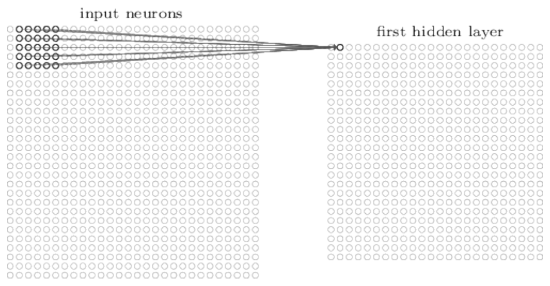

Figure 6: Mapping a filter to the second hidden neuron. (Credit: Nielsen, Michael)

We continue in this way until the input pixels have been exhausted. So, the number of neurons in the first hidden layer will be $\big((a-s_1)/s+1\big)\times\big((b-s_1)/s+1\big)$ (we need to ensure that our dimensions and stride agree). We call this process of scanning in fields of a previous layer *convolution*.

Now, referring to Figures 5 and 6, recall that we are only *visualizing* the input as a matrix. The receptive field of any given neuron is then represented in vector form $\boldsymbol{r}_j$. To visualize this, suppose we had a 3×3 picture, and the receptive field is size 3×1 (individual columns), with a stride of 1. We could represent the neural network as follows (we "flatten" the matrix going left to right, row by row).

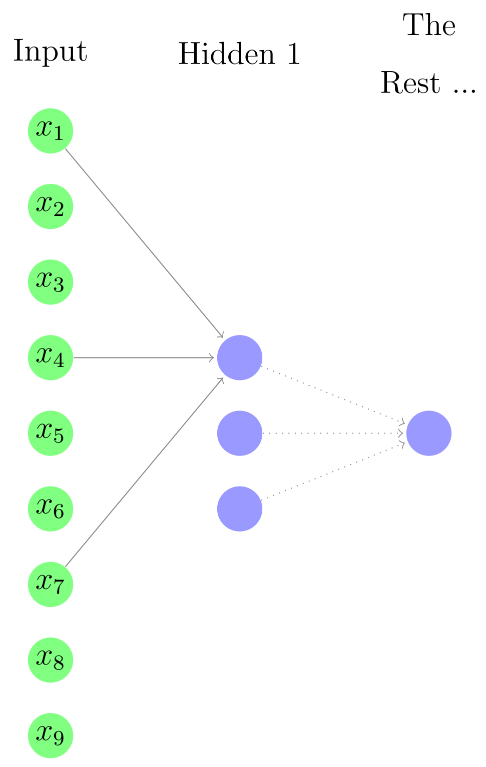

Figure 7: Another interpretation of the convolutional mapping.

Now, the first neuron in the first hidden layer is receiving input from the first receptive field $\boldsymbol{r}_1$ (the first column of the 3×3 picture). We follow suit for the rest of the neurons, and we can see how this relates to the larger general case. In this way, we define the following general formula for the output of the jth neuron in our hidden layer l.

$$\alpha_{lj} = \alpha_l \left(\boldsymbol{w}_l \cdot \boldsymbol{r}_j + b_l \right) \tag{11}$$

where $\boldsymbol{w}_l$ is *shared* among all the neurons in layer 1, as is the bias b_l, and α_l is the activation function we've chosen for layer l.[14]

Overall, our network is trying to classify our input into m classes, and for each class (in our case, these classes will be image types) there will probably be more than one fundamental feature to consider. Indeed, these fundamental features need to eventually add up to something recognizable. So, we may want to scan for an edge, a curve, etc.; anything that

[14]This is called a Convolutional Mapping, since we can redefine Eq. 11 as $\boldsymbol{\alpha}_l = \boldsymbol{\alpha}_l(\boldsymbol{w}_l * \boldsymbol{x} + \boldsymbol{b}_l)$, where $*$ is the convolution operation defined by using receptive fields, as above. We see in Figure 7 that with less links from layer feeding into neuron, we cut down on parameters we need to backpropogate for.

may be needed at the lowest level to create some recognizeable feature. In this way, the first hidden layer will typically be made up of a set of convolutional *feature maps*. Let's suppose that the first hidden layer is looking for vertical edges, horizontal edges, and diagonal edges. Then, we may illustrate the first neuron in each of the three feature maps of the first hidden layer as follows:

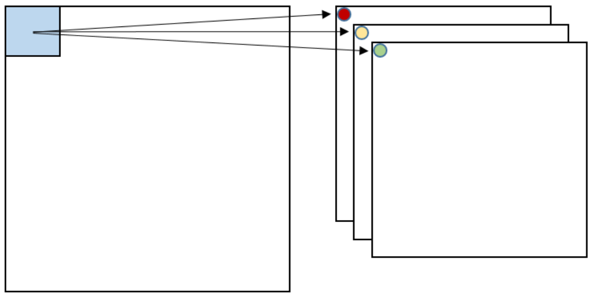

Figure 8: The receptive field (in blue) for the first neuron of each feature mapping in the first hidden layer.

Again, each level on the right side of Figure 7 represents a feature mapping (e.g., maybe the one with the yellow circle is scanning for horizontal edges). The left side, of course, is the input, and the blue box is the receptive field for each neuron in the first hidden layer. In the above case, we'll have $3 \times \big((a - s_1)/s + 1\big) \times \big((b - s_1)/s + 1\big)$ neurons in the first hidden layer (one for each neuron in each of the three feature mappings).

Realistically, though, simple edges will not be enough to discern between multiple images. So, we need a network with multiple convolutions (represented by multiple hidden layers). For example, if we want to know if there's a cat in the image, we would break down our one path of convolutions with questions in (reverse) sequential order: Is there a cat? Is there a head? Is there an ear? Is there a curvelinear contour? Is there a curve segment? We can see how this could be represented in a decision tree for convolutions and feature maps, but in general, we're thinking of the network working backwards. So, working from the first hidden layer in Figure 8, we may have another convolutional (hidden) layer following it, which takes the outputs of vertical, horizontal, and diagonal edges, and scans for six (say) other more abstract features like squares, triangles, etc.

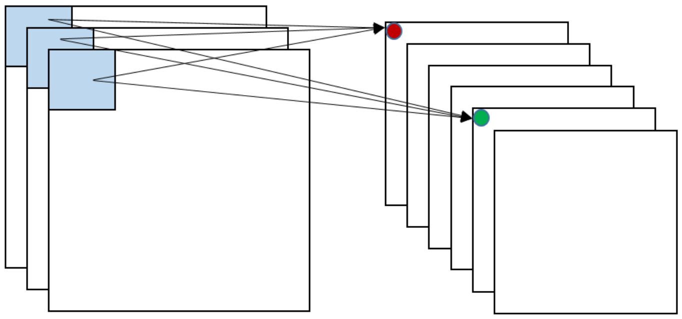

Figure 9: Left = Hidden Layer 1, Right = Hidden Layer 2. The receptive field of the first neurons in two feature maps of the second hidden layer.

Let's say that the feature maps containing red and green circles are scanning for squares and triangles, respectively. Notice now that each neuron in each feature map in the second layer is taking input from a receptive field that consists of an aggregation of pixels from the original input. That is, with each convolution, we are further abstracting from the original input. Again, this illustration can be represented in vector form. We have another similar set of neuron activation functions, where for each feature map (above, we have six), all neurons share the same weights and biases, but they are individually taking input from different sections (receptive fields) of the previous layer's outputs.[15]

Much like the fully connected network described before this section, our net is learning weights and biases which will maximize the classification accuracy of our model. So, when I say that the first layer is looking for edges, and the next is looking for shapes, etc., I am relaying what most networks will tend to do in this situation. The feature maps will *probably* look like that's what they're doing; breaking the problem down into its constituent parts (e.g., see Figure 1 in [3]).

2.4.2 Pooling Layers

Now, after some number of convolutions, we have a layer which represents the aggregation of some number of feature maps. To abstract even further from this layer, we can pool the

[15]Again, to be clear, the feature maps on the right side of Figures 8 and 9 are only *representations* of one layer. That is, all the feature maps consist of neurons in the same layer. All that is different is that each neuron is only linked to select neurons from the previous layer, determined by the receptive field.

feature maps into condensed versions of themselves. So, for instance, take one of the six feature maps from the second layer in Figure 8. To pool this map is to choose a region size (say, 2×2) and a stride, and scan the feature map much like before. But this time, instead of scanning the region looking for one particular feature, we are consolidating each region into one value. To visualize this, suppose we have the feature map and pooling layer below

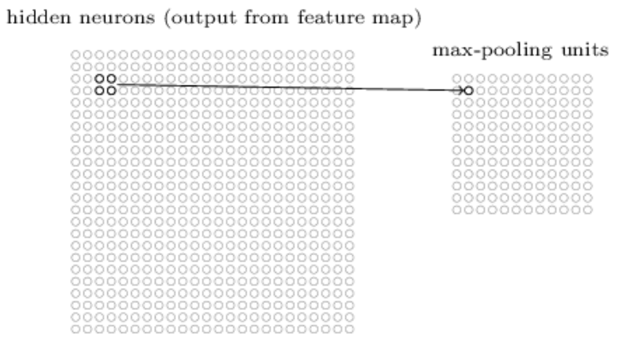

Figure 10: Pooling one feature map. (Credit: Nielsen, Michael)

There are a few ways we can pool a region into one value. Let α_{lj} be the jth neuron in layer l (think of the indicated neuron in Figure 10), and the pooled version of region $\boldsymbol{r}^*_{(l-1)j} = (\alpha_{(l-1)1}, \alpha_{(l-1)2}, \alpha_{(l-1)3}, \alpha_{(l-1)4})^T$ from layer $l-1$, (elements of $\boldsymbol{r}^*_{(l-1)j}$ are ordered and defined by the region's shape). One method is *max-pooling* where $\alpha_{lj} = \max\{\boldsymbol{r}^*_{(l-1)j}\}$. L2 pooling, and L1 pooling are also options: $\alpha_{lj} = \|\boldsymbol{r}^*_{(l-1)j}\|$, or $\alpha_{lj} = |\boldsymbol{r}^*_{(l-1)j}|$, respectively (here, $\|\cdot\|$ and $|\cdot|$ represent the L2 and L1 norms as usual).[16]

This process of pooling is repeated for all the feature maps in the layer, and we end up with a similarly represented layer with smaller "map" sizes. So, for illustrations sake

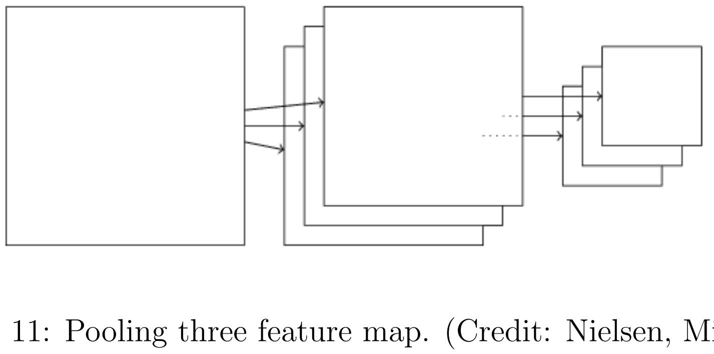

Figure 11: Pooling three feature map. (Credit: Nielsen, Michael)

[16]In fact, we could conceive of other norms which will sufficiently consolidate the information in a feature map.

Here, the last layer on the right consists of the pooled versions of the feature maps which precede them.

It's not difficult to see how the backpropogation algorithm is only slightly affected by pooling and convolution. I.e., each neuron in layer l is only affected by the neurons in layer $l-1$ which are in its receptive field. So, in calculating the effect of layer $l-1$ on each neuron in layer l, we have only use the differentials of those neurons in layer $l-1$ which are in the corresponding receptive field. We can see how the algorithm will be less intensive than using fully connected layers as before, since there are less links, and therefore less differentials.

2.4.3 Fully Connected Layers

Finally, it turns out to be beneficial to sum up our feature maps into one fully connected layer (and sometimes even better to follow up this fully connected layer with another fully connected layer before the output). That is, each neuron in each feature map (pooled or not) will be linked to a every neuron in a fully connected layer. So, much like before, the weights and biases of this layer are different for each neuron, and the number of weights and biases for each neuron in the fully connected layer is equal to the number of all neurons in the feature map layer. Continuing the example above, consider two neurons from two separate maps

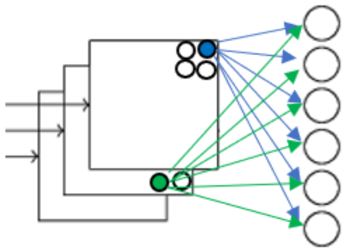

Figure 12: Going from a layer of feature maps to a fully connected layer.

The connections above will follow suit for all neurons in this feature map layer.

2.4.4 An Example

Take this example deep neural network [2].[17]

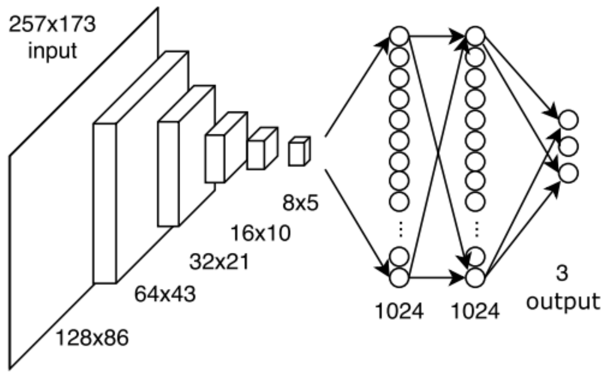

Figure 13: Example DNN with 64 (not annotated) feature maps in the first convolution.

From the left, we have our input layer (in matrix form). Then, the first convolutional layer scans the input for 64 different feature maps with receptive field size and stride $s_1 = 3$, $s = 2$, respectively.[18] Then, we have four further convolutional and pooling layers with the same field size and stride. Then, all $64 \times 8 \times 5$ neurons in the last convolutional layer are linked to 1024 neurons in a fully connected layer, twice, then to three output nodes.[19]

2.5 Improving the Network

In the above sections, we discussed a general neural network, how it works, and how it learns. However, depending on the nature of our input and desired output vectors, computing power, and other factors, the way we build our network can greatly affect how it performs. Here, we'll talk a bit about the issues with the "typical" neural network as they pertain to different situations, and how we can improve them.

[17]There are probably hundreds of convolutional neural networks with the purpose of identifying pictures, for instance. Mask R-CNN, [5], and Faster R-CNN, [8], are two of the most recent developments in this area.

[18]The 64 feature maps are represented above as depth (left to right), rather than levels, as we've used above. In the paper from where this image came, the authors specify this.

[19]The number 1024 is chosen by the authors of the paper, based on the problem at hand.

2.5.1 Saturating (Dying) Neurons

Before we get ahead of ourselves, let's first assume that our desired outcome $\boldsymbol{y}$ for a given input $\boldsymbol{x}$ looks like $[\boldsymbol{y}]_i \in \{0, 1\}$. In other words, our output is purely binary.[20] Suppose we chose the quadratic cost function C in Eq. 1, for a network with Sigmoid activation functions for the output layer of neurons.

To illustrate neuron saturation and its affect on updating weights and biases, let's escape from this network for now, and look at just a single neuron with a single input and a single output, with $n = 1$. We'll treat this neuron as a separate neural network on its own, and give it the cost function $C' = 1/2C$, just to make the derivatives a bit cleaner, with a Sigmoid activation function. In this simple example, the idea is the same as a full network, but distilled down to the elemental level of our activation input/output vectors (hence one input and one output). So, you could imagine that for a random neuron activation α_{lj} in the full network, we're only looking at how it is affected by *one* of the nodes in the previous layer.

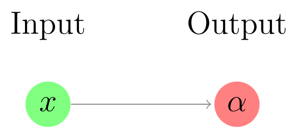

Figure 14: A single neuron in our network.

To recap, we have the cost function $C' = \frac{1}{2}(y-\alpha)^2$, where y is the desired outcome of our single input x, the activation $\alpha = \sigma(z)$ is the result of our neural network, and $z = wx + b$ (again, our expected outcome is binary, and our neuron is Sigmoid).[21]

The Problem

As we discussed before, the updates to this neuron's weight and bias change by $-\eta\frac{\partial C'}{\partial w}$, and $-\eta\frac{\partial C'}{\partial b}$ (but $-\eta$ is constant, so we ignore it here). Both of these end up depending on the derivative of the activation function σ. So, to see how the cost function affects our network

[20]Neural networks are usually used to classify an observation into one or more classes, and a binary vector like $\boldsymbol{y}$ does just that. E.g., if $\boldsymbol{x}$ belongs only to class j, then the jth element in $\boldsymbol{y}$ is a 1, and the others are 0; this is an example of a "one hot" vector. Alternatively, we could also have multiple 1s if need be.

[21]It makes sense we'd choose the Sigmoid activation function here. Because our expected outcome is binary, the Sigmoid lends itself well given its (0, 1) range.

(a neuron), we need to analyze the equations for the weight and bias differentials, as well as the derivative for our activation function σ.

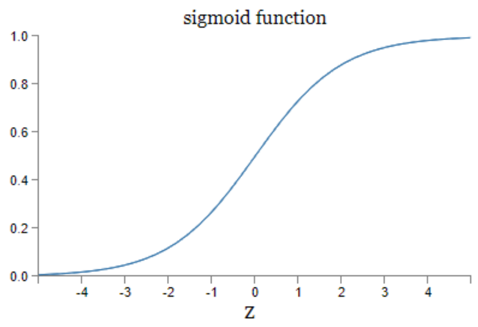

Figure 15: The Sigmoid Function

$$\begin{aligned}\frac{\partial C'}{\partial w} &= \frac{\partial C'}{\partial \sigma(z)}\frac{\partial \sigma(z)}{\partial z}\frac{\partial z}{\partial w} \\ &= -(y-\sigma(z))\sigma'(z)x \\ &= -(y-\alpha)\sigma'(z)x\end{aligned} \qquad \begin{aligned}\frac{\partial C'}{\partial b} &= \frac{\partial C'}{\partial \sigma(z)}\frac{\partial \sigma(z)}{\partial z}\frac{\partial z}{\partial b} \\ &= -(y-\sigma(z))\sigma'(z) \\ &= -(y-\alpha)\sigma'(z)\end{aligned} \qquad \begin{aligned}\sigma'(z) &= \frac{1}{1+e^{-z}}dz = \frac{e^z}{1+e^z}dz \\ &= \frac{e^z}{1+e^z} - \frac{e^{2z}}{(1+e^z)^2} \\ &= \sigma(z)(1-\sigma(z))\end{aligned}$$

Suppose first that our network is very wrong, so $|y-\alpha| \approx 1$ (remember, we're dealing with boolean values). Since x remains constant throughout the training process, the weight differential is then dependent on $\sigma'(z)$. But, with an $\alpha = \sigma(z)$ that is very wrong, it must be either very close to 1 or very close to 0. Either way, we see that $\sigma'(z)$ is then very close to 0. Therefore, when the output is very wrong (when we want the most change to occur, considering the time/epochs available), the weight will change very little; the same is true for bias. This slowly changing, "defective" neuron is called *saturated* or *dying.*[22]

[22]Here, the most changes in w and b occur when our output is right in the middle, around 0.5. Too many dying neurons cause our net to be inefficient/inaccurate. **Note:** a properly learned neuron could also be saturated, in a good way.

Solution 1: Cross-Entropy

Suppose we choose a different cost function, called the *Cross-Entropy Cost*. This function would increase its entropy as the input crosses the boundary between being fairly right/wrong to being very right/wrong. It's important to note here that the cross-entropy cost function should only be used in boolean situations, and this will quickly become apparent. So, let's go back to our neuron, and instead use the cross-entropy cost function[23]

$$C^* = -y\ln(\alpha) - (1-y)\ln(1-\alpha) \tag{12}$$

Before going further, let's prove to ourselves that this is in fact a cost function for our boolean situation, namely[24]: 1. $C^* \geq 0$, 2. $\alpha \to y \Rightarrow C^* \to 0$, and 3. $\alpha \to (1-y) \Rightarrow C^* \to \infty$ or $C^* \to K$ for some $K > 0$. First, if the desired outcome is $y = 1$, then the second term is zero which leaves us with $C^* = -\ln(\alpha)$. For $\alpha \in (0,1]$, this is non-negative. If $\alpha \to y = 1$, this function approaches 0, and alternatively if $\alpha \to (1-y) = 0$, this function approaches ∞. Analogously, if we choose $y = 0$, the first term is zero, which leaves us with $C^* = -\ln(1-\alpha)$ which is non-negative for $\alpha \in (0,1]$, and it is clear that conditions 2. and 3. also hold.

Now, we'll proceed in the same way, by calculating the differentials for our weights and biases. Recall that our activation function $\alpha = \sigma(z)$, and that $\sigma'(z) = \sigma(z)(1-\sigma(z))$.

$$\begin{aligned}
\frac{\partial C^*}{\partial w} &= \frac{\partial C^*}{\partial \sigma(z)}\frac{\partial \sigma(z)}{\partial z}\frac{\partial z}{\partial w} \\
&= \left(-\frac{y}{\sigma(z)} + \frac{1-y}{1-\sigma(z)}\right)\sigma'(z)x \\
&= \frac{\sigma(z)(1-y) - y(1-\sigma(z))}{\sigma(z)(1-\sigma(z))}\sigma'(z)x \\
&= \frac{\sigma(z) - y}{\sigma(z)(1-\sigma(z))}\sigma'(z)x \\
&= x(\sigma(z) - y) = x(\alpha - y)
\end{aligned}$$

$$\begin{aligned}
\frac{\partial C^*}{\partial b} &= \frac{\partial C^*}{\partial \sigma(z)}\frac{\partial \sigma(z)}{\partial z}\frac{\partial z}{\partial b} \\
&= \left(-\frac{y}{\sigma(z)} + \frac{1-y}{1-\sigma(z)}\right)\sigma'(z) \\
&= \frac{\sigma(z)(1-y) - y(1-\sigma(z))}{\sigma(z)(1-\sigma(z))}\sigma'(z) \\
&= \frac{\sigma(z) - y}{\sigma(z)(1-\sigma(z))}\sigma'(z) \\
&= \sigma(z) - y = \alpha - y
\end{aligned}$$

[23]Let's be clear. For this example and the next, I'll be introducing the cost function for *one* input x. In the general case, we'll be using stochastic gradient descent, and the cost will be over all $|\mathcal{T}_u| = m$ observations in the current mini batch $\mathcal{T}_u$, an average $C = 1/m\sum_{i=1}^{m} C_i$, as discussed in the sections above, where C_i is "C" as written, for observation $\boldsymbol{x}_i$. The effects discussed will always be the same for the general case, but they will practically be seen over vectors and matrices, rather than just one component.

[24]Before moving forward, recall that due to the nature of $\alpha(z) = \sigma(z)$, we have $\alpha \in (0,1) \subset (0,1]$.

Now, suppose again that we are very wrong (i.e., $(y-\alpha)\approx 1$). We can see now that the differentials for both the weight and bias are maximized. Analogously, when we are more right (i.e., $(y-\alpha)\approx 0$), the differentials are clearly minimized. This means that the more 'wrong' our neurons are, the faster they will learn, and the more 'right' they are, the less they will change. Which is exactly a solution to the dying neuron problem.

To be fair, this network could still benefit from another improvement. That is, supposing we had a vector output $\boldsymbol{\pi}_L(\boldsymbol{x})$, and we want to classify $\boldsymbol{x}$ as *exactly one* of multiple classes. In this case, we may want to guarantee that there is one component $[\boldsymbol{\pi}]_j$, say, that is higher than the rest so we can conclude that it is most likely the class we're looking for.

Solution 2: Softmax and Log-Likelihood

As stated before, sometimes we are given an observation $\boldsymbol{x}$, and we want to classify it as exactly one of m classes (e.g., maybe $\boldsymbol{x}$ represents a song's waveform, and we want to determine if it is exactly one of m genres, maybe $j^* \in \{1,2,...,m\}$ is the right one). In this way, we should want to have an activation function *in the last layer* such that the model's output $\boldsymbol{\pi}(\boldsymbol{x})$ is sort of a discrete probability density function for which each component $\boldsymbol{\pi}_j$ is the probability $\boldsymbol{x}$ belongs to class j. We also want a paired cost function which is high when the value for the expected class $\boldsymbol{\pi}_{j^*}$ is low, and vice versa. Here, I'll introduce both a new activation function *Softmax*, and cost function *Log-Likelihood*[25], which do just that. First, we define the softmax activation function

$$\mathcal{S}_{Lj} = \mathcal{S}(z_{Lj}) = \frac{e^{z_{Lj}}}{\sum_{k=1}^{m} e^{z_{Lk}}}$$

where L is the last layer of our network. To recap, our desired output for a given component $\alpha_{Lj} \in (0,1)$. So, the idea behind an activation function is for it to approach 1 when there is a strong relationship between the weighted input and the neuron's output (i.e., z_{Lj} is high), and approach 0 when there is not (in a way, think Sigmoid).

Looking at the activation function $\alpha_{Lj} = \mathcal{S}_{Lj}$, our output will always be non-negative (because $e^x \geq 0$ for $x \in \mathbb{R}$), and it will be in between 0 and 1 by construction. $\mathcal{S}$ naturally scales the output so that the sum of all outputs in the layer is 1. When z_{Lj} is high compared to

[25]To be clear, the "Log-Likelihood" cost is not to be confused with the Log-Likelihood Ratio Hypothesis Test of H_0 and H_a given a data set X in Bayesian Statistics. The difference is only subtle, but it exists.

z_{Lk} for all $k \neq j$, each $\mathcal{s}_{Lk}$ decreases accordingly, due to the added amount in the denominator so the sum remains 1, and $\mathcal{s}_{Lj} \geq \mathcal{s}_{Lk}$.[26] Finally, since the denominator is the sum of all possible numerators, $\boldsymbol{\mathcal{s}}_L$ is virtually a probability density function with a mode such that our esimate of j^* is $j' = \underset{j}{\operatorname{argmax}} \mathcal{s}_{Lj}$. The outstanding benefit of this fact is that the value $\mathcal{s}_{Lj}$ can then be interpreted as our estimated probability that the input $\boldsymbol{x}$ belongs to class j.

The softmax activation function provides outstanding benefits when it comes to interpretting the output. But, it needs a cost function to go along with it, such that the weights and biases are trained efficiently, avoiding neuron saturation. As we saw in the last solution, the cross-entropy cost function seemed made just for the sigmoid activation function (in fact, it was built especially by integrating just that). So, we can imagine that the best fruits from a softmax output layer will only come with a cost function especially constructed.

As it turns out, the Log-Likelihood cost works in just the same way for the softmax activation function as the cross entropy cost works for the sigmoid activation function. As we did before, let's construct a small network for the sake of illustration. This network will consist of one input x, and an output from the network $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, ..., \alpha_m)^T$ where $\alpha_j = \mathcal{s}(z_j)$, and $z_j = w_j x + b_j$. In this case, we define the expected outcome for x to be $\boldsymbol{y} = (y_1, y_2, ..., y_m)^T$ where $\boldsymbol{y}_{j^*} = 1$ for some j^*, and $\boldsymbol{y}_j = 0$ for all $j \neq j^*$. In other words, $\boldsymbol{y}$ is a "one-hot" vector with a 1 at position j^*, and 0 everywhere else. So, $\boldsymbol{y}$ indicates that x belongs to class j^*. With this, we introduce the Log-Likelihood Cost function C_ℓ, and an illustration:

$$C_\ell = -\ln(\alpha_{j^*}), \quad \text{where } \alpha_{j^*} = \mathcal{s}_{j^*} = \mathcal{s}(z_{j^*}) \tag{13}$$

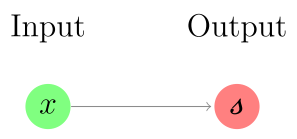

Figure 16: A softmax network. (The red output node actually represents the m neurons of our output layer.)

[26]Think of this as the monotonicity of the activation function $\mathcal{s}$. Note, if $\mathcal{s}(z_j) = e^{z_j} / \sum_k e^{z_k}$, then $\partial \mathcal{s} / \partial z_{t \neq j} = -e^{z_j} e^{z_t} / \left(\sum_k e^{z_k}\right)^2 = -\mathcal{s}(z_j)\mathcal{s}(z_t)$. So, as we increase the value of the input z_j, and thereby $\mathcal{s}(z_j)$, all other outputs $\mathcal{s}(z_t)$ will decrease proportionally.

First, we'll check that C_ℓ meets the three properties discussed earlier. First, due to the nature of $\mathscr{s}$, $\alpha_{j^*} \in (0,1)$, so $C_\ell \geq 0$. If $\boldsymbol{y}$ is our expected outcome, and $\alpha_{j^*} = \mathscr{s}_{j^*}$ is low (i.e., near 0), then $C_\ell \to \infty$. Alternatively, when $\mathscr{s}_{j^*}$ is high (i.e., near 1), then $C_\ell \to 0$. So, C_ℓ acts the way a cost function should.

Now, notice that our network outputs a vector $\boldsymbol{\mathscr{s}}$ where $\mathscr{s}_j = \mathscr{s}(z_j)$. So, the question is how do our output layer weights and biases update based on our cost? We again investigate the differentials.[27]

$$\begin{aligned} \mathscr{s}(z_j) &= \frac{e^{z_j}}{\sum_k e^{z_k}} \\ \rightarrow \frac{\partial \mathscr{s}}{\partial z_j} &= \frac{e^{z_j}}{\sum_k e^{z_k}} - \frac{e^{2z_j}}{\left(\sum_k e^{z_k}\right)^2} \\ &= \mathscr{s}(z_j)(1 - \mathscr{s}(z_j)) \end{aligned}$$

$$\begin{aligned} \frac{\partial C_\ell}{\partial w_{j^*}} &= \frac{\partial C_\ell}{\partial \mathscr{s}(z_{j^*})} \frac{\partial \mathscr{s}(z_{j^*})}{\partial z_{j^*}} \frac{\partial z_{j^*}}{\partial w_{j^*}} \\ &= -\frac{1}{\mathscr{s}(z_{j^*})} \mathscr{s}(z_{j^*})(1 - \mathscr{s}(z_{j^*}))x \\ &= -(1 - \mathscr{s}(z_{j^*}))x \\ \text{and} \quad \frac{\partial C_\ell}{\partial b_{j^*}} &= -(1 - \mathscr{s}(z_{j^*})) \end{aligned}$$

As usual, we want the weights and biases to learn faster the more wrong they are, and vice versa. Above we can see immediately that when our net is "more wrong," $\mathscr{s}_{j^*}$ is closer to 0, and we see how the changes then are maximized, which is what we want. Alternatively, when our net is "more right," $\mathscr{s}_{j^*}$ is closer to 1, and we see that the changes are minimized. So, C_ℓ along with $\mathscr{s}$ form a pairing that will help avert unwanted neuron saturation.

Solution 3: Momentum Based Gradient Descent

Stochastic Gradient Descent is one way to improve on the computational speed of Gradient Descent.[28] We can improve on the algorithm's vulnerability to saturation if we consider the "momentum" of the weights-bias vector $\boldsymbol{v} = (w_1, w_2, ..., w_k, b_1, b_2, ..., b_{k'})^T$.

[27]Notice here that given $\boldsymbol{x}$ with expected class j^*, we are only making updates to the weights and biases of the output node j^* (remember, these depend on the weights and biases from layers prior, so those are updated as well). In practice, we go through many observations $\boldsymbol{x}_i : i = 1, 2, ..., n$, and make updates along the way. It is important that $\mathscr{T}$ has observations from all classes, so we can expect to update all the weights and biases of the last layer at least once per epoch.

[28]In fact, another improvement to Stochastic Gradient Descent is to lump all the data in our mini batches $\mathscr{T}_*$ in matrix form $X_* = (\boldsymbol{x}_1; \boldsymbol{x}_2; ...; \boldsymbol{x}_p)$, where each column of X_* is an observation $\boldsymbol{x}_i \in \mathscr{T}_*$. Then, instead of looping through each observation in $\mathscr{T}_*$ as a vector, we go through the same motions with one matrix (using matrix multiplication, etc.). This method may update the net faster, so long as the computer can handle the large matrix operations.

Traditionally, after each epoch of training, standard Gradient Descent makes updates to the "position" of the weights-bias vector $\boldsymbol{v}$ by always moving it in the direction of the theoretical value $\boldsymbol{v}^*$, where $C(\boldsymbol{v}^*) = 0$. These updates (as we've discussed them so far) have been proportional to some constant value η. But, as we've shown, sometimes a weight or bias is so wrong that the changes made in the time available may not bring $\boldsymbol{v}$ significantly closer to that theoretical value.

What if we could train the network to notice that instead of making several small jumps for very wrong v_i values, it could instead make a few increasingly larger jumps. To illustrate, we define ν_i as the *velocity* of the weight or bias v_i, and we'll define the constant μ as the *momentum coefficient*. Consider the new order of updates where $\rightharpoonup$ denotes "updates to", and $*^{(\mathrm{i})}$ denotes the ith iteration of $*$, or the current udpate in training:

$$\begin{aligned} \nu_i^{(\mathrm{i})} &\rightharpoonup \nu_i^{(\mathrm{i+1})} = \mu\nu_i^{(\mathrm{i})} - \eta\frac{\partial C}{\partial v_i} \\ v_i^{(\mathrm{i})} &\rightharpoonup v_i^{(\mathrm{i+1})} = v_i^{(\mathrm{i})} + \nu_i^{(\mathrm{i+1})} \end{aligned} \tag{14}$$

To understand this better, we first set our initial velocity $\nu_i^{(0)} = 0$. Now, if there were no momentum in our system, $\mu = 0$, then the update remains the same as usual; each update is proportionally constant. But, if we pick $\mu = 1$, then first we'll have $\nu_i^{(1)} = -\eta\partial C/\partial v_i$, and the first update will be as usual $\Delta v_i^{(1)} = v_i^{(1)} - v_i^{(0)} = \nu_i^{(1)} = -\eta\partial C/\partial v_i$. But subsequently, we'll have $\Delta v_i^{(2)} = \nu_i^{(2)} = -2\eta\partial C/\partial v_i$. Compounding, we can see how the changes can increase with iteration. A quick calculation shows

$$\Delta v_i^{(\mathrm{i+1})} = \nu_i^{(\mathrm{i+1})} = \mu^{\mathrm{i+1}}\nu_i^{(0)} - \eta\,\frac{\partial C}{\partial v_i}\sum_{k=0}^{\mathrm{i}}\mu^k, \quad \mu \geq 0 \tag{15}$$

Now, we can see just how the changes in weights and biases will increase after each epoch, based on our momentum coefficient μ. In this way, the weights and biases have a sort of momentum or acceleration as they move toward $\boldsymbol{v}^*$. It is fairly easy to see from Eq. 15 how increasing $\mu > 0$ increases this momentum.

One issue with this method is when $\boldsymbol{v}$ is close to $\boldsymbol{v}^*$ after a few iterations of increasing ν_i, there is a very present danger of overshooting $\boldsymbol{v}^*$ by a lot, which sometimes brings the weight/bias back to where it started. We can counteract this problem by gauging how we choose the momentum coefficient. Smaller μ vales mean less momentum (maybe, more friction if you will), and larger μ values mean more momentum. Further, we can improve on this method by changing μ_u based on the previously calculated cost C_{u-1} from minibatch $\mathcal{T}_{u-1}$; i.e., if C_{u-1} is small, make μ_u small, and alternatively if C_{u-1} is large. If we do this, Eqs. 14 and 15 will change such that the velocity at iteration i+1 is dependent only on η, the differential, and $\mu_{\text{i+1}}$. It will not be dependent on previous iterations of the velocity.

2.5.2 Overfitting and Regularization

The purpose of a neural network, as discussed earlier, is to be able to take a new input $\boldsymbol{x}$, and output an accurate determination $\boldsymbol{y}$. This determination could be classifying $\boldsymbol{x}$ into some class(es), or to assign a numerical value. The first step in creating this network is training it with a data set $\mathcal{T}$, which contains many "example" observations $\boldsymbol{x}_i$, for which their outcome $\boldsymbol{y}_i$ is known. With this information, we use the techniques discussed above to adjust the weights and biases in the network, minimizing the cost function, ostensibly bringing us closer to a model that will accurately classify/assign *new* observations like the ones in $\mathcal{T}$.

In practice, we'll have a data set $\mathbb{X} = \{\mathcal{T}, \mathbb{T}, \mathcal{V}\}$, where each subset of data are mutually exclusive, and the outcomes $\boldsymbol{y}_i$ are known for all $\boldsymbol{x}_i \in \bigcup \mathbb{X}$. $\mathcal{T}$ is the training data set as described above, $\mathbb{T}$ is the *test* set, and $\mathcal{V}$ is the *validation* set. The test set is used to try out the model; to get an idea of how the net would perform on observations not seen during training. The validation set can be used to determine the hyperparameters (learning rate, number of layers/neurons, weight/bias initialization, etc.) of the model before training it. Sometimes, $\mathcal{V} = \emptyset$, and $\mathcal{T}$ is instead used to determine hyperparameters (say, if $|\mathbb{X}|$ is small), but it is a good practice to ensure that everything in the training set is wholly new to the model, and that the test set is used expressly for testing.[29]

[29] Hyperparameters are usually determined via trial and error. We are trying to determine the best overall network design before we start training it. This forces us to run many versions of the network on $\mathcal{V}$, and pick the version (i.e., set of hyperparameters) that performs the best. As we'll expound on in the coming paragraph, this renders our model biased, in a way, toward the validation data. So, if it learns/trains on the same data set, we only increase that bias. Similarly, if the net is tested on the same data set, we lose the whole point of the test, which is to evaluate the model on observations its never seen.

Problem

As we've seen in the way we update weights and biases that after a point, neurons stop learning significantly. By design, once a neuron is outputting an activation that aligns well with coming inputs and their expected outputs, w and b should change very minimally. The problem, then, is that they still change. After each epoch of training on $\mathscr{T}$, the neural network becomes closer and closer to reaching 100% accuracy in classifying or assigning the observations in $\mathscr{T}$. However, this so called precision is only an indication of how well the network is learning the subtle intricacies of the data in $\mathscr{T}$. After all, the purpose of our network isn't really to be able to properly classify observations in $\mathscr{T}$, or even really in $\mathbb{X}$ for that matter.

Our problem arises when our network meets an observation that is vastly different (or, just different enough) from the ones in $\mathscr{T}$. True, we can use $\mathbb{T}$ to estimate how well our model will perform "in the field," and make adjustments accordingly, but this is only an estimate. In the end, we need a way to minimize the sensitivity of our network's training algorithm to atypical observations (or noise) in $\mathscr{T}$. We want our network to be robust enough to properly handle all new observations, but not so sensitive during training that it assumes abnormalities in the real world will always behave the same as they do during training.

Solution 1: L2 Regularization

Regularization is a technique which changes the way that weights and biases are updated during training. The idea is to keep the magnitudes of the weights small,[30] so that any large variations in $\boldsymbol{x}$ do not affect the network too much, and therefore abnormalities in the testing (or real world) data are not magnified due to how weights were trained from $\mathscr{T}$. One way to do this is using *L2 Regularization*, which adds a scaling term to the original cost function C_0. Let $\boldsymbol{w}$ contain all weights in the net, $n = |\mathscr{T}|$, call λ the *weight decay* factor, and set $\|\cdot\|$ as the standard Euclidean (L2) norm, and the new regularized cost is

$$C_{L2} = C + \frac{\lambda}{2n}\|\boldsymbol{w}\|^2 \tag{16}$$

Note: Before moving on, it should be noted that the 2 in the denominator of the regularization term is really to clear up interpretation later (much like C' in Section 2.5). In fact,

[30]Biases are left free to better capture the uniqueness of inputs. As we've seen above, Δb is not affected by x in the way that Δw is, so it is not the focus here.

we could have done without it, but for illustration's sake, we keep it in for our version of L2 Regularization.

Now, at first glance, we can see how large weight magnitudes are penalized in our new cost function, but it is not yet clear how this new term affects how our weights are being updated. We proceed again by differentiating our cost function in terms of a particular weight w_j (again, biases are not affected)

$$\frac{\partial C_{L2}}{\partial w_j} = \frac{\partial C}{\partial w_j} + \frac{\lambda}{n} w_j$$

Recall that in gradient descent (for now, with zero momentum), our weight is updated as follows (we'll use the same notation as in the momentum discussion):

$$\begin{aligned} \Delta w_j^{(\mathrm{i+1})} &= -\eta \frac{\partial C_{L2}}{\partial w_j} \\ &= -\eta \frac{\partial C}{\partial w_j} - \frac{\eta\lambda}{n} w_j^{(\mathrm{i})} \\ \rightarrow \quad w_j^{(\mathrm{i+1})} &= w_j^{(\mathrm{i})} + \Delta w_j^{(\mathrm{i+1})} \\ &= \left(1 - \frac{\eta\lambda}{n}\right) w_j^{(\mathrm{i})} - \eta \frac{\partial C}{\partial w_j} \end{aligned}$$

Consider the first and last lines of the operations above. We can interpret the new style of update as first re-scaling $w_j^{(\mathrm{i})}$ by a factor of $\gamma = 1 - \eta\lambda/n < 1$, then making the normal update. So, after each update, all weights decrease in magnitude by a set factor, and then updates are made as usual to these new weights. Notice the dependency of γ on n; any changes to λ need to be done with n in mind, i.e., larger n should mean a larger weight decay factor to keep γ comparable to the new data set. Or, keeping λ constant, as we increase n, we decrease the amount to which we re-scale $\boldsymbol{w}$.

Solution 2: L1 Regularization

The only difference between this regularization method and the last is the norm begin used to measure the weights' scale. In this case, we have a new regularized cost of

$$C_{L1} = C + \frac{\lambda}{n} |\boldsymbol{w}|$$

where $|\boldsymbol{w}|$ represents the L1 norm $\sum |w_j|$.

Now, again looking at the differentials for gradient descent,

$$\begin{aligned} \Delta w_j^{(\mathrm{i+1})} &= -\eta \frac{\partial C_{L1}}{\partial w_j} \\ &= -\frac{\eta\lambda}{n}\mathrm{sign}^*(w_j) - \eta\frac{\partial C}{\partial w_j} \\ \rightarrow \quad w_j^{(\mathrm{i+1})} &= w_j^{(\mathrm{i})} + \Delta w_j^{(\mathrm{i+1})} \\ &= w_j^{(\mathrm{i})} - \frac{\eta\lambda}{n}\mathrm{sign}^*(w_j) - \eta\frac{\partial C}{\partial w_j} \end{aligned}$$

where $\mathrm{sign}^*(w_j)$ represents the sign of the weight so that $\mathrm{sign}^*(0) = 0$. Now, instead of first decreasing the magnitude of w_j by a factor, we decrease its magnitude (send it closer to 0) by a constant. Intuitively, this means that the L1 Regularization method will make more drastic changes to smaller weights, and less so for large weights; whereas the L2 method makes more drastic changes to larger weights, and less so for smaller ones.

Solution 3: Dropout

First, let's revisit the issue of overfitting, restated in another way. The network is training based on some training set $\mathcal{T}$, and the neurons in the network are going to adjust based on any phenomenon they sense in the data. What if we made a few other networks, each slightly different from the next, and trained them on the same data. We could then run all of them for some observation(s), and accept the determination that was output the most. That is in a way, we let these separate neural networks vote on the answer, and we accept the one with the most votes.

Dropout is a way to simulate this many-network model. Suppose we construct a network with h neurons combined among the hidden layers, say. Then, depending on the amount of training data available, we randomly select k of these h neurons, turn them off (so, we now have $h - k$ hidden neurons), and train this subnetwork on a subset $\mathcal{T}_1 \subset \mathcal{T}$. Then, we reset the network, and randomly select (with replacement) k more neurons, and train the model on $\mathcal{T}_2 \subset \mathcal{T}$. All of our selections $\mathcal{T}_i$ are mutually exclusive and collectively exhaustive to $\mathcal{T}$, and we do this for each training epoch. So we'd expect that for any abnormalities in the data, some neurons are not exposed, and we can think of those neurons as a balance for those which were exposed. In the end, our network will tend to be less sensitive to strange inputs, less likely to assume the output will be the same as it would have been during training.

Solution 4: Artificially Expanding Data

This solution is very problem dependent, but it can be extremely effective. The basic idea stems from the fact that neural networks are not nearly as smart as we are. Our brains are very skilled in filling in the gaps, but a neural network is not nearly as adept. So, we need to take extra care in training the network, ensuring it's exposed to as many variations of nature as possible. Data can be expensive, but sometimes there are gaps that could be intuitively filled in (i.e., we provide the intuition computers cannot). For example, as humans, we can hear a song in slow motion, and still be able to tell whether it is Brubeck's *Take Five* or not. A network needs to be exposed to the notion of "slow motion" before it can accomplish such a feat.

So, the idea with artificially expanding data (when the data is not already available; of course, more real world data is always better) is that we take some of the data available, mess with it a bit in natural ways, and add the results to bolster our training data. For songs, this could mean creating versions of observations that emulate slow motion, or have added distortion to the background. For pictures, we could translate, rotate, or reflect available images. For weather patterns, we could running models with different initial states, and including the new observations. These strategies can be very easy or time consuming, but they are typically extremely straight-forward and can provide fruitful results. [6]

Solution 5: Early Stopping

Simply put, over fitting is basically a consequence of the network making fine-tuning adjustments, so that its classification accuracy over $\mathscr{T}$ is nearing perfect. After a certain time, though, our diminishing returns are so small that it's not worth letting the model run any longer. In fact, letting it run longer will be what causes overfitting. So, a very simple method for averting overfitting is to stop the model after there has been no significant improvement in say t epochs (often, $t = 10$ is a good start). Adjust t based on the model's behavior.

2.5.3 Initializing Hyperparameters

Let's pause for a second. Don't be fooled by all this talk about how to create a neural net from the bottom up. *In practice, you don't have to do all this work!* There are a bunch of neural nets out there, e.g., [10], [12], ready for people to just plug-and-play ...

But that's no fun! So, we press on.

Now, when I use the word hyperparameter, I mean any value that affects the architecture of our network before any training occurs. This could be the number of neurons in each hidden layer, the number of hidden layers, learning rate, regularization factor, momentum coefficient, etc. It is worth mentioning also that we have partitioned our data $\mathbb{X} = \{\mathscr{T}, \mathbb{T}, \mathscr{V}\}$ into a training set, a test set, and a validation set. When adjusting our hyperparameters, we test them out (or validate them) by looking at the classification accuracy using the validation set. This ensures that we are not biasing our network by the data it will be trained on, $\mathscr{T}$.

To be completely frank, optimizing the initial conditions of hyperparameters (and their behavior) is a game of guess and check. Granted, it can be done in a systematic way, and we can reference hyperparameters chosen by other researchers in similar situations, but from what I've seen there are not many theory based methods that are ubiquitous in the field. Generally, our goal when setting hyperparameters is to get some sort of outcome that is better than chance.

We start with a very simple version of the model that we plan to create, using only a small subset of our data. This minimizes the amount of time it takes to run through an epoch, so we can monitor the effects of our hyperparameterization. Then, we work to optimize the hyperparamters. Once we've decided they are sufficiently optimized, we train the network on $\mathscr{T}$, then use the test set $\mathbb{T}$ to evaluate the model's accuracy. We can always go back and adjust the hyperparameters again at this point, and then reset the weights and biases before re-training.

Yoshua Bengio provides a superb guide for initializing hyperparamters in Section 3.1 of [1], which is far better than I could ever attempt, so I refer the reader to that document. However, I will use discussion provided by Michael Nielson [6] to expound on Bengio's paper as it pertains to the starting architecture, learning rate and learning rate schedule, and activation function choice.

A Starting Architecture

Before we can move on with optimizing hyperparameters, we need to have a sort of skeleton neural network. This skeleton will depend on the type of network we plan to employ. But in essence, our skeleton will be the simplest version of the final product. For instance, if we plan on having a network with only fully connected hidden layers, the skeleton should have an input layer, one hidden layer with the minimal amount of neurons recommended (refer to Bengio), and then an output layer. If we plan on using convolutions, our network skeleton should consist of one convolutional layer, with a strategically chosen receptive field size. And, if we plan to use pooling, we should have one pooling layer that is not too aggressive in its region size. As we go through adjusting hyper parameters, we can adjust the number of hidden layers, region/field size, etc., incrementally trying new values, searching for the best outcomes. Finally, as stated above, we will only use a small subset of our data $\mathcal{V}$, so that we can monitor our changes to hyperparameters quickly.

Learning Rate

According to some authors [6, 1], $\eta_0 = 0.1$ is generally a good initial value. From this starting point, it's a good idea to chart the accuracy of the model against the epoch. If the learning rate is too large (as would make sense), our updates to the weights and biases $\boldsymbol{v}$ will often overshoot the vector $\boldsymbol{v}^*$ discussed earlier, and we will see oscillations. If the learning rate is too small, we will definitely see a smooth decline in classification accuracy with epoch, but it will be too slow.

Test the network using η_0, and chart the classification accuracy. If the graph oscillates, lower η_0 by a factor of 10, that is $\eta_0 = .1 \rightharpoonup \eta_1 = .01$. Alternatively, if the graph shows a smooth decline, increase η_0 by a factor of 10. Continue this process until we find a threshold value η_i that causes our graph to oscillate, but if decreased by a factor of 10 will show a smooth decrease. Now, fine-tune this value by incrementally decreasing it by values of a lower order of magnitude until the chart no longer oscillates.

This learning rate after fine-tuning is very close to the optimized value, but it is not final until after we optimize the other hyperparameters. That is, as we update hyperparameters, we may need to go back to η_i and improve it further.

Learning Rate Schedule

The learning rate need not stay constant throughout the whole training process of our neural net. One way to ensure that we maintain a smooth decrease, with each epoch, in classification accuracy on $\mathcal{V}$ is to decrease our learning rate based on how improved the model is after each epoch. So, suppose we set $t = 10$. We decrease our learning rate by a factor $\beta < 1$

when the classification accuracy does not increase in t epochs. We do this until the learning rate reaches some threshold value η_∞.

Activation Function Choice

The choice of activation function for the output layer will depend on the type of output we are looking for, and the best cost function which supports it (see Section 2.5). For the inner layers, the type of function (along with its parameters) are pretty much chosen by running through examples and picking the one with the best result. The ReLU function tends to support convolutional layers fairly well, and the Sigmoid or tanh functions apparently work well on fully connected layers, [6]. Again, none of this will be an exact science.

2.5.4 Other Techniques

Even though the improvements mentioned here are ostensibly the most popular for the majority of DNN applications, they are only the tip of the iceberg when it comes to improving neural networks. Techniques like the *The Hessian Technique*, [11], and other optimization algorithm alternatives to GD can be used to greatly influence the accuracy of a network. I didn't include them in this paper because they tend to pertain only to very specific situations.

3 Applying CNNs to Audio

[See the attached PowerPoint Slides, and recorded discussion.]

References

[1] Bengio, Y. *Practical recommendations for gradient-based training of deep architectures.* (2012). https://arxiv.org/pdf/1206.5533.pdf

[2] Choi, K., Fazekas, G., & Sandler, M. *Explaining Deep Convolutional Neural Networks on Music Classication.* (2016). Queen Mary University of London, London, The United Kingdom. https://arxiv.org/pdf/1607.02444.pdf

[3] Choi, K., Fazekas, G., Sandler, M., & Kim, J. *Auralization of Deep Convolutional Neural Networks: Listening to Learned Features.* Queen Mary University of London, London, The United Kingdom and Naver Labs. http://ismir2015.uma.es/LBD/LBD24.pdf

[4] Foote, K. *A Brief History of Deep Learning.* (2017). http://www.dataversity.net /brief-history-deep-learning/

[5] He, K., Gkioxari, G., Dollár, P., & Girshick, R. *Mask R-CNN.* Facebook AI Research (FAIR). (2018). https://arxiv.org/pdf/1703.06870.pdf

[6] Nielsen, M. *Neural Networks and Deep Learning.* (2017). http://neuralnetworks anddeeplearning.com.

[7] Reingold, E. *Artificial Neural Networks Technology.* (2009). http://www.psych. utoronto.ca/users/reingold/courses/ai/cache/neural4.html

[8] Ren, S., He, K., Girshick, R., & Sun, J. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* https://arxiv.org/pdf/1506.01497.pdf

[9] Rumelhart, D., Hinton, G., & Williams, R.. *Learning representation by backpropogating errors.* Nature, 323(9):533-536, 1986. https://www.iro.umontreal.ca/∼vincentp/ ift3395/lectures/backprop_old.pdf

[10] *SciKit-Learn Supervised Learning Neural Networks.* http://scikit-learn.org /stable/modules/neural_networks_supervised.html

[11] Srihari, S. *Machine Learning: The Hessian Matrix.* `http://www.cedar.buffalo.edu/~srihari/CSE574/Chap5/Chap5.4-Hessian.pdf`

[12] *TF-Learn API.* `http://tflearn.org/`