Hyperparameters for Dense Neural Networks

Christopher James Hettinger
Brigham Young University

Hyperparameters for Dense Neural Networks

Christopher James Hettinger

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Mathematics

Brigham Young University

ABSTRACT

Hyperparameters for Dense Neural Networks

Christopher James Hettinger
Department of Mathematics, BYU
Doctor of Philosophy

Neural networks can perform an incredible array of complex tasks, but successfully training a network is difficult because it requires us to minimize a function about which we know very little. In practice, developing a good model requires both intuition and a lot of guess-and-check. In this dissertation, we study a type of fully-connected neural network that improves on standard rectifier networks while retaining their useful properties. We then examine this type of network and its loss function from a probabilistic perspective. This analysis leads to a new rule for parameter initialization and a new method for predicting effective learning rates for gradient descent. Experiments confirm that the theory behind these developments translates well into practice.

Keywords: neural networks, backpropagation, gradient descent, crelu

Acknowledgments

Thanks to Jeff Humpherys and Tyler Jarvis for years of mentorship; to Emily Evans, Chris Grant, and Jared Whitehead for correcting my errors and clarifying my explanations; to Lonette Stoddard for guidance on the administrative aspects of graduate school; and to Tanner Christensen for collaborating on the research that led to this work.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

The study of neural networks has exploded in recent years, with new models and training techniques being published at an incredible and accelerating rate. Unsurprisingly, the theory of neural networks has lagged behind the practice. Model decisions — even those as fundamental as the number of layers or the learning rate — are made based on heuristics and empirical searches because we lack the necessary understanding to make the correct choices a priori. These searches are notorious for requiring both tremendous computing resources to train the models and large numbers of man-hours to find the right hyperparameters, with the latter leading academics to joke that the best models are found by 'graduate student descent.' The goal of this work is to examine fully-connected neural networks, the basic model on which all other neural networks are based, and search for mathematical insights that can reduce the amount of guess-and-check needed to successfully train one.

To this end, we begin by focusing on networks with a concatenated relu (crelu) activation scheme. This allows us to keep all of the benefits of the relu activation, which has been the gold standard in deep learning for years, while gaining some additional desirable properties. In particular, the use of crelu makes it possible to initialize networks such that they are both better-behaved during gradient descent and more amenable to mathematical analysis than their relu counterparts.

We then simplify our models through reparametrization. Whereas conventional neural networks use both weight matrices and bias vectors on each layer, this formulation uses only weight matrices. This change further facilitates mathematical analysis without compromising the networks' expressivity — the ability to approximate arbitrary functions. We also take advantage of the fact that the crelu activation can be realized as a matrix multiplication, allowing us to describe the network entirely in linear algebraic terms.

After establishing that crelu networks without bias vectors are at least as capable as conventional relu networks, we derive explicit formulas for the updates given to the weight matrices during gradient descent. This allows us to mitigate the problematic tendency of gradient descent to make large changes to small parameters (causing instability) and small changes to large parameters (requiring long training times). We do this by improving upon the standard initialization method and using normal random variables with optimal variance for each layer.

We then turn our attention to the problem of selecting a good learning rate. To do this, we consider the loss along a straight line through the parameter space. This allows us to realize the loss as a function of the learning rate λ. We then compute the expected values (given the previously-established random initialization scheme) of the first and second derivatives of this function, thus approximating it with a parabola.

From the exact coefficients of this parabola, we derive a useful quantity called the 'scaling factor' which is easy to compute and describes the relationship between network architecture and their optimal learning rates. We then test networks of different widths and depths on two data sets and show that they respond to different learning rates in the way predicted by the scaling factor.

1.1 Contributions

The results presented here are not only of theoretical interest; they have immediate practical value. The type of network discussed here is easily implemented with standard software tools, as are the methods for making such a network train more effectively. The main contributions of this dissertation are as follows:

- The crelu network without bias is a natural replacement for the currently-standard type of dense neural network, which has bias vectors and uses relu activations.

- The proportional initialization is the unique choice that gives weight matrices updates of the appropriate relative size. Its special case, the symmetric proportional initialization, also takes advantage of crelu's unique properties.

- The explicit formulas for the first and second derivatives of the loss with respect to the learning rate allow us to derive the scaling factor, which in turn tells us how to change the learning rate to compensate for changes to the network architecture. This greatly reduces the need to test learning rates empirically.

- The included experiments validate the use of the quadratic approximation in determining the learning rate. They also show that the scaling factor can be used to predict whether training with a given fixed learning rate will converge or diverge, significantly increasing its utility.

In addition to being useful on their own, many of the results included here have the potential to be stepping stones toward deeper understanding of fully connected networks, as well as toward analogous results for other network types.

Chapter 2. Background

The following is a general-purpose introduction to both fully connected and convolutional neural networks.

2.1 Neurons

A basic neuron takes a vector $x \in \mathbb{R}^d$ of inputs and returns a single value $\hat{y} \in \mathbb{R}$ as output. To do this, the neuron needs three components:

- A weight vector $w \in \mathbb{R}^d$
- A bias $b \in \mathbb{R}$
- An activation function $a : \mathbb{R} \to \mathbb{R}$

The output of the neuron is computed in the following way:

$$\hat{y} = a\left(w \cdot x + b\right).$$

Here, $w \cdot x$ represents the dot product of w and x. It is sometimes convenient to break down the operation of the neuron into two steps by defining an intermediate variable $z \in \mathbb{R}$; That is,

$$\begin{aligned} z &= w \cdot x + b, \\ \hat{y} &= a(z). \end{aligned}$$

Suppose that we have a data set of inputs $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$ and corresponding targets $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$ and we want a neuron's outputs to approximate the targets. That is, for each $x^{(i)}$ we want the $\hat{y}^{(i)}$ produced by the neuron to be as close to $y^{(i)}$ as possible. In order to make this happen, we first need to specify what we mean by 'close.' This is done by choosing a loss function $\ell(\hat{y}, y)$. The loss function tells us how poorly the neuron is doing at the task of producing accurate outputs. A high loss indicates bad performance, a low loss indicates good performance, and zero loss indicates perfect performance. We compute the overall loss L on a data set by simply averaging the loss for each data point.

$$L = \frac{1}{n} \sum_{i=1}^{n} \ell\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Choosing (or designing) an appropriate loss function is critical and often nontrivial. What we want a model to do and what the loss function incentivizes it to do are not always the same thing.

Logistic Regression. Several popular models can be expressed in terms of a single neuron. For example, suppose we have a binary classification task with $y \in \{0, 1\}$. The logistic function

$$a(z) = \frac{1}{1 + e^{-z}}$$

is a natural choice of activation function because it produces values between zero and one.

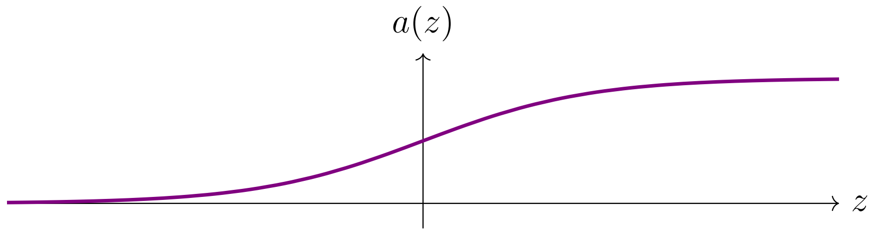

Figure 2.1: The logistic, or ‘sigmoid,’ activation function.

We then need a loss function that rewards the values produced by the logistic function for being close to the target values and punishes them for being far away. We'll use the binary cross-entropy function

$$\ell(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

because it incentivizes the behavior we want (by producing higher loss values for predicted labels further from the target) and because, as we will soon see, it pairs very well with the logistic activation function. In general, it is important to choose activation functions and loss functions that work well together. Notice that ℓ is designed such that one of its two terms will vanish for either choice of y:

$$\ell(\hat{y}, 1) = -\log(\hat{y}),$$
$$\ell(\hat{y}, 0) = -\log(1 - \hat{y}).$$

By graphing the loss function separately for each of the two targets, we see that it has the desired property of being zero when $\hat{y} = y$ and increasing as $\hat{y}$ moves away from y. In fact, ℓ goes to infinity as $\hat{y}$ approaches the opposite target.

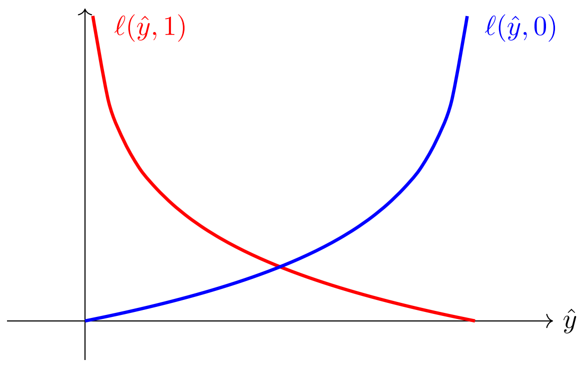

Figure 2.2: The binary cross-entropy loss function.

With the activation function and loss function chosen, all that remains is to choose values for w and b. Specifically, we want the values of w and b that will make the overall loss L for our model as low as possible.

This is the part where we actually get to do some machine learning. Rather than set w and b by hand, we ask the neuron to learn its own parameters through gradient descent. To do this, we need to compute the derivatives of the loss function with respect to each parameter. First, we look at how the loss changes when we change the output $\hat{y}$.

$$\begin{aligned}\frac{\partial \ell}{\partial \hat{y}} &= -y\frac{\partial}{\partial \hat{y}}\log(\hat{y}) - (1-y)\frac{\partial}{\partial \hat{y}}\log(1-\hat{y}) \\ &= -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}\end{aligned}$$

In order to change $\hat{y}$, we need to change z, so we compute that derivative next.

$$\frac{\partial \hat{y}}{\partial z} = \frac{\partial}{\partial z}a(z) = \frac{\partial}{\partial z}\frac{1}{1+e^{-z}} = \frac{e^{-z}}{(1+e^{-z})^2} = a(z)(1-a(z))$$

Finally, we need to figure out how changing w and b — the parameters we actually control — will affect z. We'll do b first because it's easy.

$$\frac{\partial z}{\partial b} = \frac{\partial}{\partial b}(w \cdot x + b) = 1$$

Really easy. Calculating $\partial\ell/\partial w$ is almost as easy, but requires a little extra thought because w is a vector. That means that $\partial\ell/\partial w$ is also a vector, each entry of which is the derivative of ℓ with respect to the corresponding entry of w. With that in mind, we get

$$\frac{\partial z}{\partial w} = \frac{\partial}{\partial w}(w \cdot x + b) = x$$

Note that x is indeed a vector of the same dimension as w, as expected.

Putting all of these derivatives together, we can see how changes in our parameters will change our loss function:

$$\begin{aligned}\frac{\partial \ell}{\partial w} &= \frac{\partial \ell}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w} = \left(\frac{\hat{y}-y}{\hat{y}(1-\hat{y})}\right)a(z)(1-a(z))x, \\ \frac{\partial \ell}{\partial b} &= \frac{\partial \ell}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial b} = \left(\frac{\hat{y}-y}{\hat{y}(1-\hat{y})}\right)a(z)(1-a(z)).\end{aligned}$$

If the derivative of the loss function with respect to a parameter is positive, then increasing that parameter will increase the loss. We want to decrease the loss, so we will move our parameters in the direction opposite the derivative.

That's exactly what gradient descent is. We choose a small positive learning rate $\lambda > 0$ and we update our parameters with a simple rule:

$$w \leftarrow w - \lambda \frac{\partial L}{\partial w}$$
$$b \leftarrow b - \lambda \frac{\partial L}{\partial b}$$

Recall that L is just an average of the losses from individual points, so

$$\frac{\partial L}{\partial w} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial \ell\left(\hat{y}^{(i)}, y^{(i)}\right)}{\partial w}$$
$$\frac{\partial L}{\partial b} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial \ell(\hat{y}^{(i)}, y^{(i)})}{\partial b}$$

Using this update rule repeatedly will eventually yield nearly optimal values for w and b and the neuron will then classify the dataset as accurately as it can. This particular model — a single neuron that uses the logsitic activation function and the binary cross-entropy loss function — is known as logistic regression.

Support Vector Machines. With different choices of activation and loss function, we can realize other popular models as well. For example, we build a support vector machine. This means doing binary classification a little differently by using targets $y \in \{-1, 1\}$ with the identity activation

$$a(z) = z$$

and the hinge loss function

$$\ell(\hat{y}, y) = \max(0, 1 - y\hat{y})$$

In this case, the loss function looks quite different.

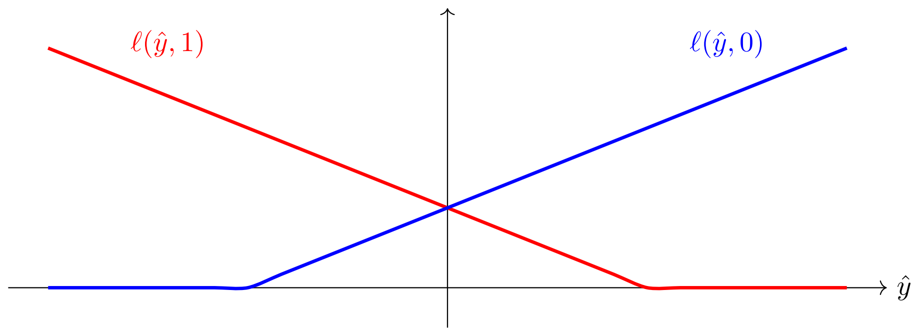

Figure 2.3: The binary hinge loss function.

Rather than incentivizing the neuron to output exactly y, it asks $\hat{y}$ to be at least 1 if $y = 1$ and at most -1 if $y = -1$. While this approach looks quite different from logistic regression, both are reasonable and the two tend to produce similar results.

As before, we can find the optimal values of w and b by working out $\partial\ell/\partial w$ and $\partial\ell/\partial b$ and then using gradient descent to minimize L. The derivatives will be quite different in this case, of course, due to the different loss and activation functions.

Linear Regression. Neurons aren't limited to classification. They can also solve regression problems where y is allowed to vary continuously. If we again use the identity activation function

$$a(z) = z,$$

and introduce the squared error loss function

$$\ell(\hat{y}, y) = (\hat{y} - y)^2,$$

then we have a linear regression model. In this case, the loss function tries to drive $\hat{y}$ toward y by penalizing errors in both directions equally.

In practice, it would be silly to find w and b through gradient descent because there is a formula for computing them directly. Gradient descent would still work, just not as quickly.

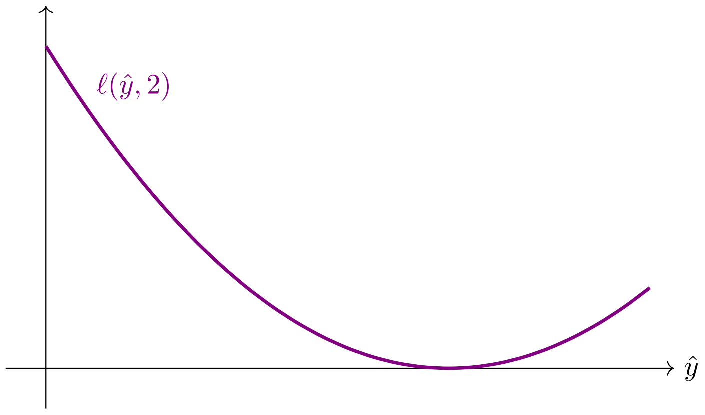

Figure 2.4: An example squared error loss function.

Nonlinear Modeling. While single neurons can be made to perform a variety of tasks, they have a glaring limitation: they can only learn linear patterns in the data. Figure 2.5 shows this in the case of logistic regression. The neuron does very well when it's possible to separate the two classes with a straight line (in higher dimensions, the analog would be a hyperplane) but can't handle even simple curves.

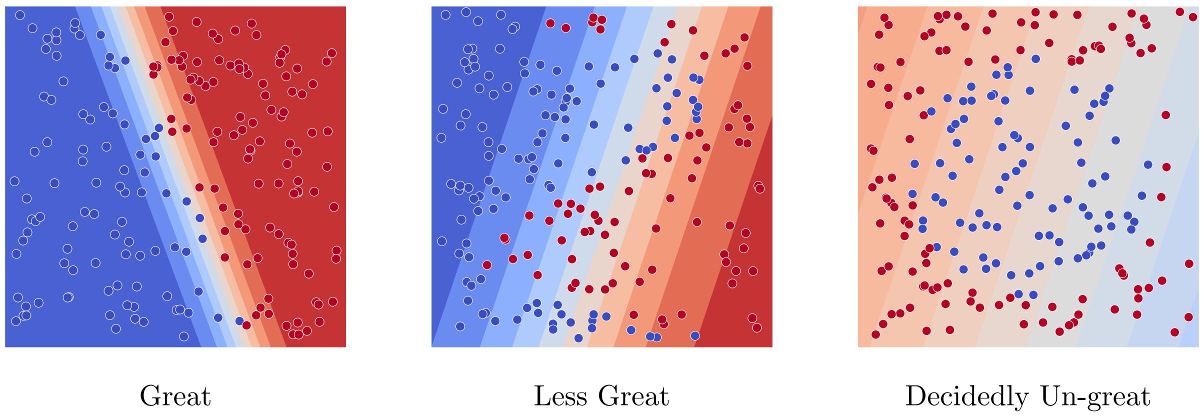

Figure 2.5: Here we see that logistic regression does very well when the data is linearly separable, but poorly otherwise because it can't learn nonlinear patterns in the data. The dots represent the data points and their colors represent the corresponding targets — blue for zero and red for one. The colored background shows the output of the neuron at each location in the data space, which ranges from zero to one.

There are a variety of machine learning methods for dealing with nonlinear patterns. For the simple data sets in Figure 2.5, we could engineer new features by hand that allowed linear models to succeed. With higher-dimensional data and more complex patterns, this quickly becomes a terrible approach. It also directly contradicts our goal, which is to have the model learn as much as possible on its own.

In pursuit of the flexibility to learn a variety of nonlinear functions, we're going to combine multiple neurons together into a single model — a neural network. The goal of doing so is to reduce the need to engineer features by hand and let gradient descent do the hard work.

2.2 Neural Networks

In the above example, we were asking a single neuron to take a two-dimensional input and return a one-dimensional output. To describe this model visually, We can depict input dimensions as squares and neurons as circles and end up with a simple diagram like the following.

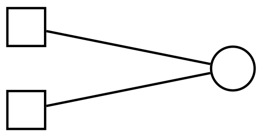

Figure 2.6: A single neuron with two inputs.

In a neural network, some neurons can use other neurons' outputs as their inputs. For example, we might expand on the model above by having four neurons that each take the two-dimensional input and produce ouputs. Then the final neuron can take the four earlier neurons' outputs as its input and attempt to approximate the target with its output.

A groups of neurons which receive the same inputs is called a layer. The network in Figure 2.7 has two layers: one consisting of the four intermedite neurons, and one consisting of the single output neuron. The layers preceding the output layer are called hidden layers.

In this new model, each neuron has its own weight vector, bias, and activation function. Genrally, all of the neurons in a given layer will use the same activation function. The loss function only applies directly to the output of the final neuron, but the parameters of every neuron affect the final output and so must be learned.

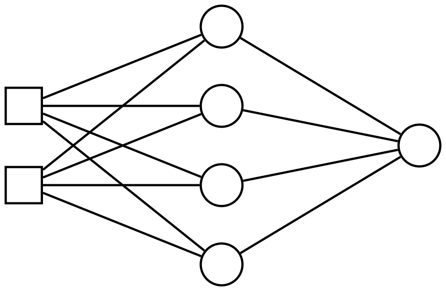

Figure 2.7: A neural network with a single hidden layer.

In Figure 2.8, we see that this expanded model is much more flexible. It can still handle linear patterns, but it can also deal with other shapes. However, this model comes with more parameters and the relationships between them are more complex than in a single neuron. We need a way of efficiently computing the derivatives with respect to each in order to train our network.

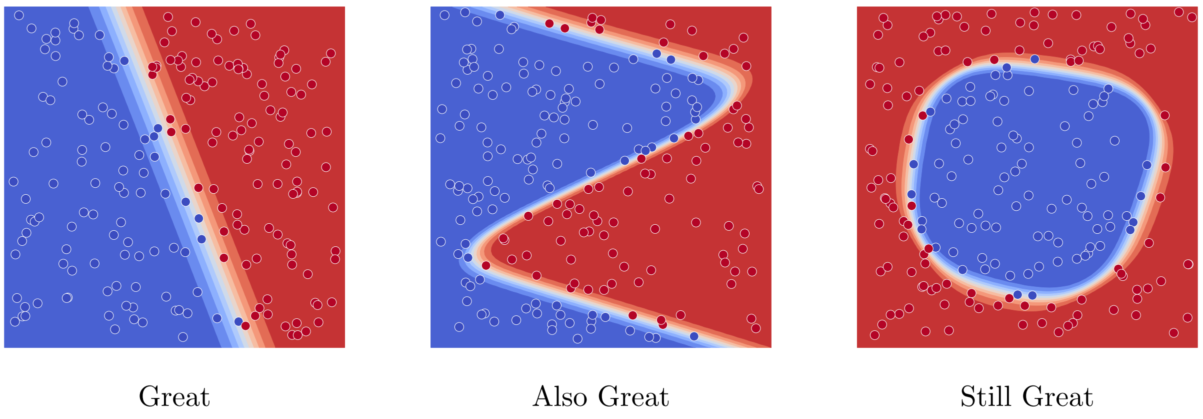

Figure 2.8: A network with just four hidden neurons easily fits these simple nonlinear patterns without any manual feature engineering. In this case, all neurons used logistic activation functions.

Backpropagation. Let's return to the diagram of our neural network with a single hidden layer. Each neuron in the hidden layer has a weight vector, and the first step in computing the outputs of these neurons is to take the dot product of the input with each weight vector. That's exactly what happens when we multiply a matrix by a vector. This allows us to talk about neural networks in a much more efficient way. Rather than referring to the weight vectors of individual neurons, we can talk about a layer having a weight matrix. The ith row of the weight matrix is the weight vector of the ith neuron in the layer. Similarly, we can think of the layer as having a single bias vector, the ith entry of which is the bias value of the ith neuron.

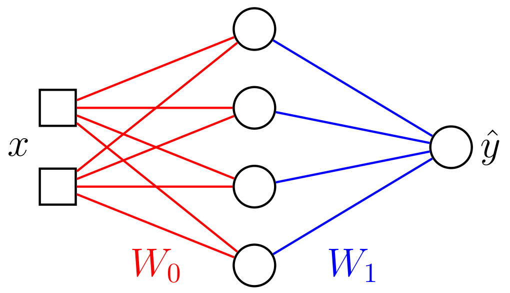

Figure 2.9: The weights of a layer comprise a matrix.

The Forward Pass. With this in mind, we can describe the entire network above in a very simple way. The input x is a length 2 column vector. The hidden layer has a 4×2 weight matrix W_0 and a length 4 bias column vector b_0. Even though there is only one output neuron, we can still think of it as a layer with 1×4 weight matrix W_1 and length 1 bias vector b_1. Putting it all together, the output of the network is computed as follows.

$$\hat{y} = a\left(W_1 a\left(W_0 x + b_0\right) + b_1\right)$$

We're abusing notation a little bit by giving the activation function vector inputs. This simply indicates that we're applying the activation to each entry of the vector individually.

It will be very helpful to break the computation of $\hat{y}$ into steps and give names to the values computed at each step. First, let's number our layers, starting our count at zero. In the small network above, the hidden layer is Layer 0 and the output layer is Layer 1. Then we can say that Layer i has weight matrix W_i and bias vector b_i.

Call the input vector x_0 because it's the input to Layer 0. In other words, $x_0 = x$. Then define $x_i = a\left(W_{i-1}x_{i-1} + b_{i-1}\right)$. In other words, x_i is the ouput of Layer $i-1$ and the input to Layer i. For additional convenience, define $z_i = W_i x_i + b_i$. Then we can simply write $x_i = a(z_{i-1})$.

With this notation, we can break down the action of an m-layer network as follows.

$$\begin{aligned}
x_0 &= x \text{ (input)} \\
z_0 &= W_0 x_0 + b_0 \\
x_1 &= a(z_0) \\
z_1 &= W_1 x_1 + b_1 \\
x_2 &= a(z_1) \\
&\vdots \\
z_{m-1} &= W_{m-1} x_{m-1} + b_{m-1} \\
x_m &= a(z_{m-1}) \\
\text{(output) } \hat{y} &= x_m
\end{aligned}$$

Writing out all of the network operations in this way will make it much easier to compute all the derivatives needed for gradient descent.

The Backward Pass. The algorithm for calculating all of the derivatives in a neural network is called backpropagation because it moves backwards, starting with the output layer and then addressing the hidden layers one at a time in reverse order.

As in the case of a single-neuron model, the first step is to use the loss function to compute $\partial\ell/\partial\hat{y} = \partial\ell/\partial x_m$. With this derivative in hand, it's simple to compute $\partial\ell/\partial z_i$ and $\partial\ell/\partial x_i$ for each i according to these rules:

$$\begin{aligned}
\frac{\partial x_{i+1}}{\partial z_i} &= \frac{\partial}{\partial z_i} a(z_i) = a'(z_i) \\
\frac{\partial z_i}{\partial x_i} &= \frac{\partial}{\partial x_i} W_i x_i + b_i = W_i^T
\end{aligned}$$

As before, we use $a'(z)$ to denote the operation of applying a' separately to each entry of the vector z. These two formulas are applied to calculate all of the $\partial\ell/\partial z_i$ and $\partial\ell/\partial x_i$

sequentially in this way:

$$
\begin{aligned}
\frac{\partial \ell}{\partial x_m} &= \frac{\partial \ell}{\partial \hat{y}} \\
\frac{\partial \ell}{\partial z_{m-1}} &= \frac{\partial x_m}{\partial z_{m-1}} \frac{\partial \ell}{\partial x_m} = a'\left(z_{m-1}\right) \circ \frac{\partial \ell}{\partial x_m} \\
\frac{\partial \ell}{\partial x_{m-1}} &= \frac{\partial z_{m-1}}{\partial x_{m-1}} \frac{\partial \ell}{\partial z_{m-1}} = W_{m-1}^T \frac{\partial \ell}{\partial z_{m-1}} \\
\frac{\partial \ell}{\partial z_{m-2}} &= \frac{\partial x_{m-1}}{\partial z_{m-2}} \frac{\partial \ell}{\partial x_{m-1}} = a'\left(z_{m-2}\right) \circ \frac{\partial \ell}{\partial x_{m-1}} \\
\frac{\partial \ell}{\partial x_{m-2}} &= \frac{\partial z_{m-2}}{\partial x_{m-2}} \frac{\partial \ell}{\partial z_{m-2}} = W_{m-2}^T \frac{\partial \ell}{\partial z_{m-2}} \\
&\vdots \\
\frac{\partial \ell}{\partial z_0} &= \frac{\partial x_1}{\partial z_0} \frac{\partial \ell}{\partial x_1} = a'\left(z_0\right) \circ \frac{\partial \ell}{\partial x_1} \\
\frac{\partial \ell}{\partial x_0} &= \frac{\partial z_0}{\partial x_0} \frac{\partial \ell}{\partial z_0} = W_0^T \frac{\partial \ell}{\partial z_0}
\end{aligned}
$$

Here the $\circ$ symbol denotes the Hadamard product — the element-wise product of two vectors. Because the activation function is applied individually to each component of z_i, it makes sense that the derivatives work the same way.

It is instructive to verify that the above algorithm produces derivatives with the correct dimensions. That is, that each $\partial \ell / \partial z_i$ and $\partial \ell / \partial x_i$ is a vector with the same length as the same length as the corresponding z_i or x_i. In particular, this shows why the weight matrices are transposed by differentiation.

This is only half of the backpropagation algorithm. We now need to work out the derivatives $\partial \ell / \partial W_i$ and $\partial \ell / \partial b_i$ that we'll actually be using to adjust parameters during gradient descent.

To derive $\partial \ell / \partial W_i$ and $\partial \ell / \partial b_i$, it's easier to look at one row at a time. Let $w_{i,r}$ be the rth row of W_i, $b_{i,r}$ the rth entry of b_i, and $z_{i,r}$ the rth entry of z_i. Thus $z_{i,r} = w_{i,r} x_i + b_{i,r}$. Note that $w_{i,r}$ is a row vector and x_i is a column vector, so their product is a scalar. This is consistent with the dot product formulation used previously. Now we can easily take some derivatives.

$$\begin{aligned}\frac{\partial \ell}{\partial w_{i,r}} &= \frac{\partial z_{i,r}}{\partial w_{i,r}} \frac{\partial \ell}{\partial z_{i,r}} \\ &= \frac{\partial}{\partial w_{i,r}} \left(w_{i,r} x_i + b_{i,r} \right) \frac{\partial \ell}{\partial z_{i,r}} \\ &= x_i^T \frac{\partial \ell}{\partial z_{i,r}} \\ &= \frac{\partial \ell}{\partial z_{i,r}} x_i^T .\end{aligned}$$

The derivative with respect to $b_{i,r}$ is even simpler.

$$\begin{aligned}\frac{\partial \ell}{\partial b_{i,r}} &= \frac{\partial z_{i,r}}{\partial b_{i,r}} \frac{\partial \ell}{\partial z_{i,r}} \\ &= \frac{\partial}{\partial b_{i,r}} \left(w_{i,r} x_i + b_{i,r} \right) \frac{\partial \ell}{\partial z_{i,r}} \\ &= \frac{\partial \ell}{\partial z_{i,r}} .\end{aligned}$$

Note that we can reorder the product for $\partial \ell / \partial w_{i,r}$ because $\partial \ell / \partial z_{i,r}$ is a scalar. When we stack the rows $\partial \ell / \partial w_{i,r}$ to form $\partial \ell / \partial W_i$, this is not the case. We have to write $\partial \ell / \partial W_i = (\partial \ell / \partial z_i) x_i^T$ to capture what we just proved above: that each row of $\partial \ell / \partial W_i$ is simply x_i^T multiplied by the corresponding entry of $\partial \ell / \partial z_i$.

Now we have the rules needed to compute all of the $\partial \ell / \partial W_i$ and $\partial \ell / \partial b_i$:

$$\begin{aligned}\frac{\partial \ell}{\partial W_i} &= \frac{\partial \ell}{\partial z_i} x_i^T \\ \frac{\partial \ell}{\partial b_i} &= \frac{\partial \ell}{\partial z_i}\end{aligned}$$

Training a neural network then amounts to repeating two operations:

- The forward pass — computing the z_i and x_i en route to the network output $x_m = \hat{y}$
- The backward pass — computing the derivatives $\partial L/\partial W_i$ and $\partial L/\partial b_i$ with backpropagation and using them to update the W_i and b_i via gradient descent.

These two steps are alternated until some stopping criterion is reached.

2.3 Hyperparameters

When designing and training a neural network, one must make a number of choices. Many, such as the number of neurons in a layer or the learning rate to be used for gradient descent, might reasonably be called parameters of the network. To distinguish these human-determined quantities from the weights and biases, which are adjusted with gradient descent, the term *hyperparameters* is used.

Initialization. Before we can adjust the parameters of a network, we need to choose initial values for them. In the past, this was a difficult task and initialization algorithms were a significant area of research. Today, thanks to various innovations in network design and optimization, successful training is much less dependent on careful initialization. In fact, most networks are simply initialized with random values.

Most commonly, biases are initially set to zero while weights are randomly sampled from distributions such as normal, uniform, or truncated normal. All of these distributions are consistently chosen to have mean zero, while the variance is usually chosen to be l/k_i where l is a constant (popular values for l include 1, 2, and 6) and k_i is the length of x_i. The scaling is done to keep the derivatives of the loss function with respect to different parameters from having wildly differing magnitudes, as this can cause problems for gradient descent.

Pre-processing. Like other machine learning models, neural networks perform appreciably better when data is processed so as to have certain properties. In particular, it is usually helpful to normalize inputs — shifting and scaling each feature (dimension) of x_0 so as to give it zero mean and unit variance. Like the previously described initialization scheme, this makes gradient descent more effective by ensuring that different parameters have derivatives of similar magnitudes.

Categorical data is usually reshaped so as to consist of one-hot vectors. For example, if a feature can take on three different values, it is better to represent these as $[1\,0\,0]$, $[0\,1\,0]$, and $[0\,0\,1]$ than to use a single dimension with values in $\{0, 1, 2\}$. Additional, specialized pre-processing techniques exist for structured data types, such as images and text.

Activation Functions. So far, we've only discussed two activation functions — sigmoid (logistic) and identity. While both may be appropriate for output layers, neither is a good choice for hidden layers. Sigmoid activation functions lead to the *vanishing gradient problem*. Because backpropagation involves repeated multiplication by $a'(z)$, an activation function with a small derivative can cause gradients to shrink exponentially as they move backward through the network. When this happens, updates to the corresponding parameters are too small and the network fails to learn at a reasonable speed. It's also possible, though much less common, for gradients to become large early in the network and cause instability. This is called the *exploding gradient problem*. Activation functions are thus chosen with their derivatives in mind, especially in the case of deep networks with many hidden layers.

From this perspective, the identity function sounds like an ideal activation. Its derivative shouldn't cause any problems; multiplying by one never hurt anybody! In fact, the identity function is an even worse choice. If the activation function is linear, then the network simply performs a series of linear transformations on the data. The composition of two linear transformations is itself linear, so such a network would have the same fundamental limitation as a single neuron — it could only learn linear patterns. A good activation function needs to be nonlinear while having a derivative that won't cause gradients to vanish or explode.

One of the first alternatives to the sigmoid function was the hyperbolic tangent $\tanh(z)$. This represented a modest improvement. A much larger improvement, and arguably one

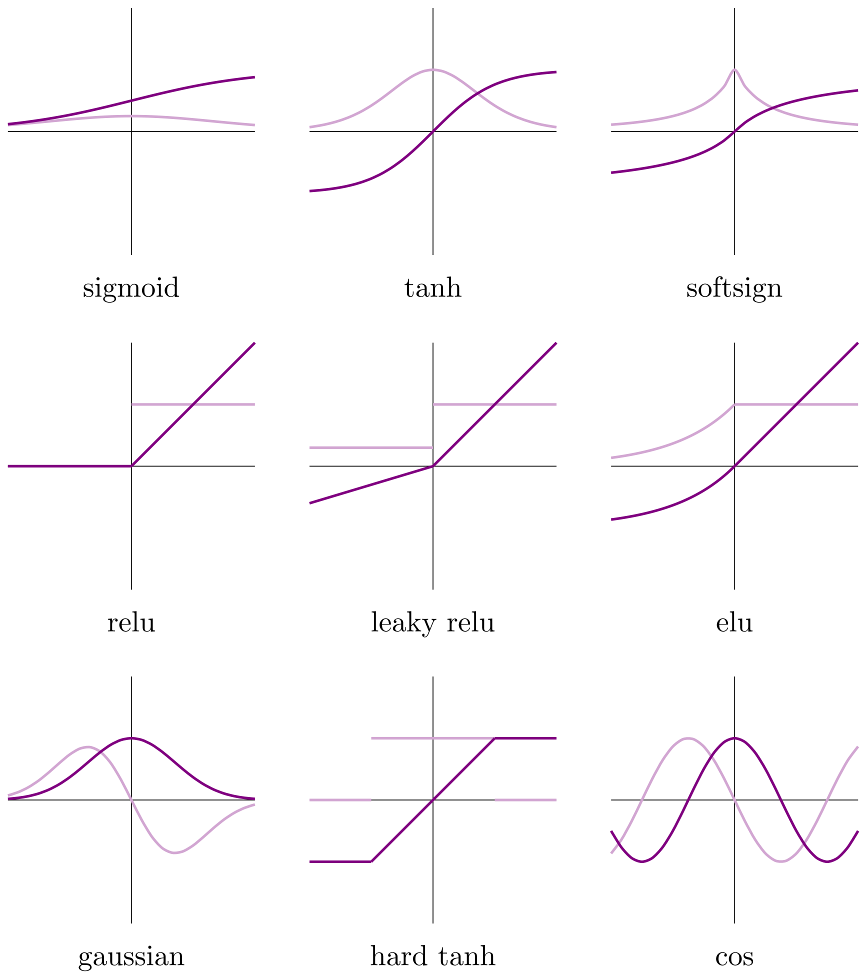

Figure 2.10: These are just some of the many activation functions (and their derivatives) that are used in neural networks. Relu is by far the most popular due to its combination of simplicity and effectiveness. Variants such as leaky relu and elu are similarly effective. Other activations are sometimes used in special applications.

of the most important developments in the history of the field, came with the introduction of the relu (short for ‘rectified linear unit’ and sometimes styled ReLU) activation function $a(z) = \max(0, z)$. Currently, relu is by far the most common choice of activation function for hidden layers. Several variants have been proposed with the intention of improving on relu, but none has performed convincingly better in practice.

Regularization. The appeal of neural networks lies largely in their ability to learn complex patterns in data. However, this flexibility naturally comes with the capacity to overfit the training data. Regularization methods are simple and popular ways of trying to reduce overfitting. Parameter regularization adds a penalty to the loss function to keep the weights of a network from growing too large, as large weights have been observed to be a sign of overfitting in some cases.

$$L = \frac{1}{n} \sum_{i=1}^{n} \ell \left(\hat{y}^{(i)}, y^{(i)} \right) + c \sum_{w} w^2.$$

Here, the second sum runs over all the weights in the network individually. Biases are typically not penalized. This specific example shows L_2 regularization, so called because it penalizes the L_2 norm of the weights. Another common choice is L_1 regularization, which uses the absolute value $|w|$ instead of w^2.

Activity regularization does essentially the same thing, but it penalizes the layer outputs x_i by adding an x_i^2 or $|x_i|$ to the loss function. In both cases, it is difficult to choose an appropriate regularization constant c. If c is too small, it won't impact training. If too large, the network may keep too many weights at zero because it prioritizes regularization over the original learning task. As is often the case in this field, appropriate regularization constants are generally found empirically.

Dropout. Dropout is another anti-overfitting tool, more interesting and more powerful than regularization. With dropout, some neurons are randomly 'dropped' from the network at each training iteration. Sepcifically, dropout requires a probability p. Then each hidden neuron is ignored with probability p in the sense that their outputs are replaced with zero and the associated parameters are not updated. Thus each training iteration looks at a random sub-network, asking it to perform the network's task despite the missing units.

This method compels the network to learn redundancy. When neurons can be dropped, the network must ensure that critical information is learned by sets of neurons. At test time, the whole network is used. Due to the training process, the complete network behaves somewhat like an ensemble of smaller models. This tends to result in better generalization.

Here are two examples of sub-models that dropout might produce at different training iterations. Output neurons are not dropped because in general it doesn't make sense for $\hat{y}$ to have fewer dimensions than y.

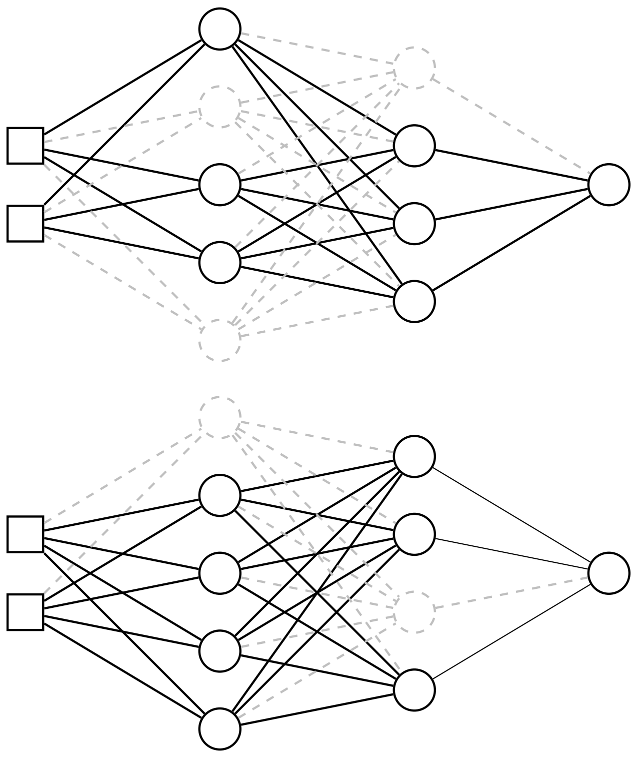

Figure 2.11: Example subnetworks active during a given training step with dropout.

Batching. Because the loss function for a network is defined as an average over the training set, so are its derivatives. For large data sets, this can make gradient descent very expensive. In practice, neural networks are almost always trained with *stochastic* gradient descent. Instead of computing derivatives for every data point at every iteration, only a small subset (called a 'mini-batch' or just a 'batch') is used each time. The resulting updates aren't exactly the same, but even for small batches they tend to be close to the true ones. While it may take a few more iterations to get down to a given loss, the individual iterations are so much faster that training is significantly more efficient overall.

A common approach is to randomly divide a data set into batches of equal size and then use each batch for a step of gradient descent. The number of steps needed to iterate through each batch is called an epoch. Networks are usually trained for multiple epochs, with the data set randomly divided into new batches each time.

Further, recent research indicates that the noisy nature of stochastic gradient descent may partially explain why neural networks generalize well to test data in practice despite having tremendous potential to overfit.

Momentum. Gradient descent is often described with the metaphor of a man descending a mountain by repeatedly taking steps downhill — specifically, in the direction of steepest descent. Momentum is then explained by replacing the man with a boulder that picks up speed as it goes.

It's an instructive metaphor, even though the momentum method used in neural network training doesn't work exactly like real-world inertia. In pseudocode, standard gradient descent looks like the following. Here we use `p` for a network parameter, `dp` for $\partial L/\partial p$ (or its mini-batch approximation), and `r` for the learning rate. To add momentum, we also need a decay rate `m` and a velocity `vp`.

```
while training:
    for p in parameters:
        dp = get_gradient(batch,p)
        p -= r*dp
```

Figure 2.12: Vanilla gradient descent.

```
for p in parameters:
    vp = 0
while training:
    for p in parameters:
        dp = get_gradient(batch,p)
        vp = m*vp + dp
        p -= r*vp
```

Figure 2.13: Gradient descent with momentum.

The velocity term v_p is a decaying sum of the past gradients with respect to the parameter p. The decay rate of this sum, $0 \leq m < 1$, determines how quickly momentum will diminish. If $m = 0$, the result is the same as regular gradient descent.

Choosing the right m is, of course, usually an empirical process. Momentum tends to speed up learning significantly, but too much can slow it down again by introducing oscillations. However, it is not uncommon to see rates as high as $m = .99$ in practice.

Learning Rates. Choosing a good learning rate is difficult. If it's too large, gradient descent can become unstable and even diverge. If it's too small, not only can training take forever, but the network is more likely to overfit. To complicate matters further, the standard practice is to change the learning rate during training. Many authors decrease the learning rate periodically according to a fixed schedule, resulting in loss graphs that look like this:

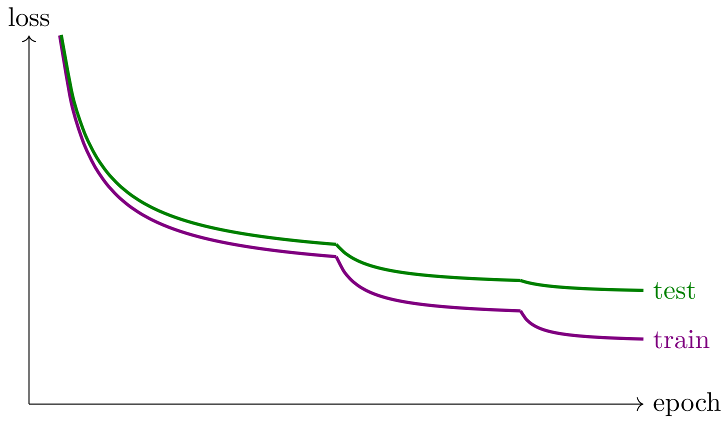

Figure 2.14: Example neural network loss curves with a decreasing learning rate schedule.

Decreasing the learning rate allows gradient descent to get closer to a minimum of the loss function by taking smaller steps. However, smaller steps tend to result in improvements that don't translate as well to test data. Usually, there comes a point after which the training loss will continue to decrease while the test loss stops or even increases a bit.

Hyperparameter Validation. In addition to the parameters that are optimized by gradient descent — the weights and biases — a neural network has many *hyperparameters* that are found experimentally. These include depth (number of layers), layer width (number of neurons in a given layer), regularization constants, dropout probability, learning rates, momentum decay rates, batch sizes, and many more that we haven't discussed here. For this reason, it is prudent to have three data sets — training, validation, and testing. Gradient descent only ever looks at the training set. The person (or machine) choosing the hyperparameters can see how they affect performance on the validation set and adjust them accordingly. This means that the validation data has been used for optimization and so can't be used for testing. The three-way split is necessary to ensure scientifically valid results. If all of the available data is used for training and validation, then there is no way of knowing whether the results will generalize to new data.

2.4 Convolutions

The neural networks discussed so far are often called fully connected neural networks because every neuron in a given layer depends on every neuron in the previous layer. Here we discuss convolutional neural networks, which restrict the pattern of connections between layers in order to take advantage of structured data.

Filters. Neural networks as described so far do poorly on common types of real-world data because they don't account for relationships between features. For example, it's easy to represent an image as a vector by allocating one entry for each pixel value, but the resulting representation fails to capture the relationships between pixels that result from their arrangement in the image plane. For this reason, we introduce the convolution — an operation from image processing that we will use to build a new type of neural network layer suited to processing images.

For simplicity, imagine a black and white image. This lets us represent each pixel with a single real value. We can then *convolve* this image with a *filter*, another grid of real values which is usually much smaller (in height and width) than the image. In this example, we have a 6×6 image and a 3×3 filter. The convolution computes a *filtered image* by matching up the filter with each possible patch of the image, taking the dot product of the filter and the patch (by multiplying the matched-up pairs of pixel values and then adding up all of

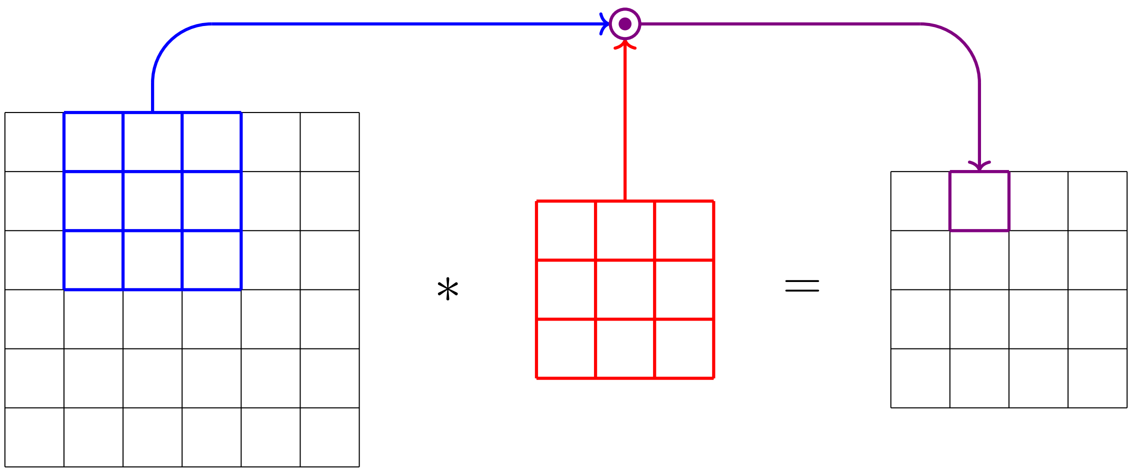

Figure 2.15: A convolution on a single-channel image.

the products), then placing the resulting value in the corresponding location in the filtered image.

There are two principle reasons for using filters in this fashion. The first is that they are designed to take advangate of locality — they 'look' for recognizable patterns in small, contiguous subregions of the image, as humans do. Second, they offer translation invariance — a feature will activate the filter in the same manner no matter where it appears in the image. Fully connected layers do not have either of these properties.

In the above example, the filtered image is smaller than the original image. In general, an $h \times w$ image and a $k \times k$ filter will produce an $h - k + 1 \times w - k + 1$ filtered image. (Non-square filters are possible, but rarely used in practice.) This size-reducing effect is often undesirable and will be addressed later in this section.

First, we turn to the issue of channels. If an image has c channels (real-world color images usually have three or four), then the filter must have c channels as well. Channels of the filter get matched up with the corresponding image channels for the product step and everything is then summed, so the filtered image has a single channel.

In convolutional networks, we'll almost always want to convolve an image with many filters. In this case, our output will have one channel per filter. This is demonstrated in the following diagram. The input image has four channels, so each filter has five channels.

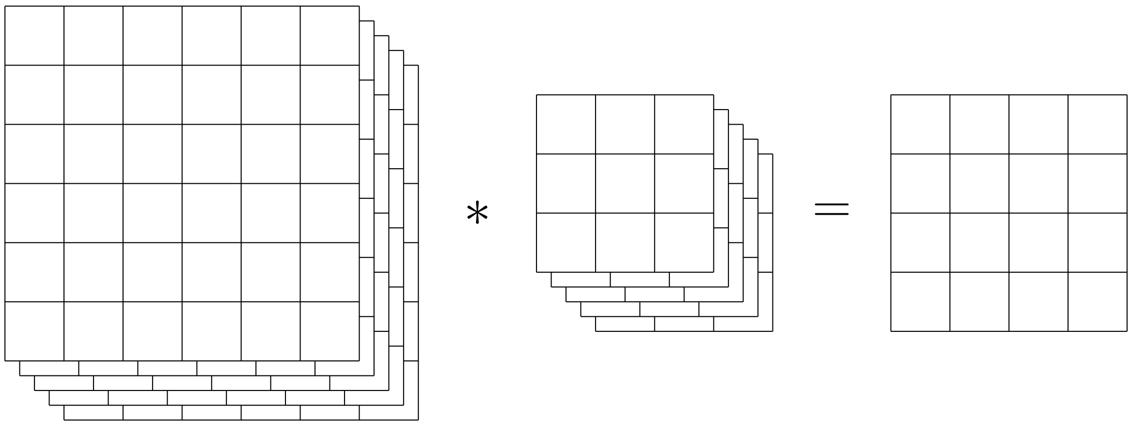

Figure 2.16: A convolution on a multi-channel image.

There are three filters, so the output image has three channels. In practice, it's common to see tens or hundreds of channels, but that would make for busy diagrams.

This is what a convolutional layer does.. It takes an input image with c_{in} channels and outputs an image with c_{out} channels. To do this, it uses c_{out} filters, each of which is $k \times k$ (k is usually 3 or 5) and has c_{in} channels so it can match up with the input images. Each filter has a bias which is added to each pixel of the corresponding output channel. Then every value of the output image is run separately through the activation function (which is usually relu). The values in the filters are usually called weights so that we can discuss convolutional layers with the same language as standard ones. The weights and biases are trained with gradient descent just as in a standard layer.

Filters can be made arbitrarily large or small, and strictly speaking they do not need to be square. In practice, networks with many layers of small, square filters have been the most effective for image processing tasks. However, convolutions can be used for other types of data with array-like structures. In these cases, different filter sizes and shapes may be appropriate.

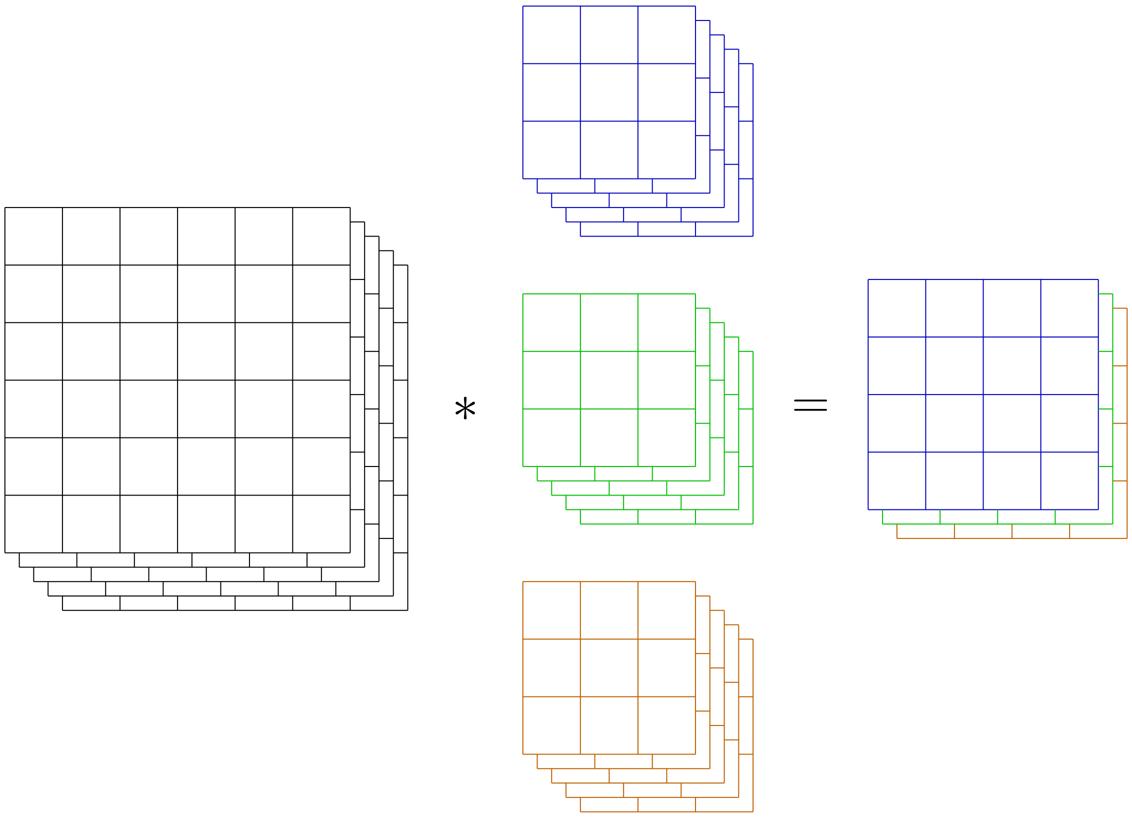

Figure 2.17: A multi-channel convolution on a multi-channel image.

Padding. Before we can build a network out of convolutional layers, we need to address the problem of images shrinking when convolved. This is done by padding the input image, putting a border around it so that the filter has as many places to go as there are pixels.

Usually, the values in this border are all set to zero. This is called, intuitively enough, zero padding. More sophisticated padding schemes have been proposed, including some that attempt to extrapolate appropriate values from the image, but these are almost never seen in practice because zero padding, the simplest possible solution, has proven sufficient for even the most sophisticated processing models.

When filters have odd sizes, the padding can be done in a symmetric fashion. A 3×3 filter needs one pixel of padding, a 5×5 filter needs two. Even-sized filters require asymmetric padding, and this is surely one of the reasons why they are rarely used in practice.

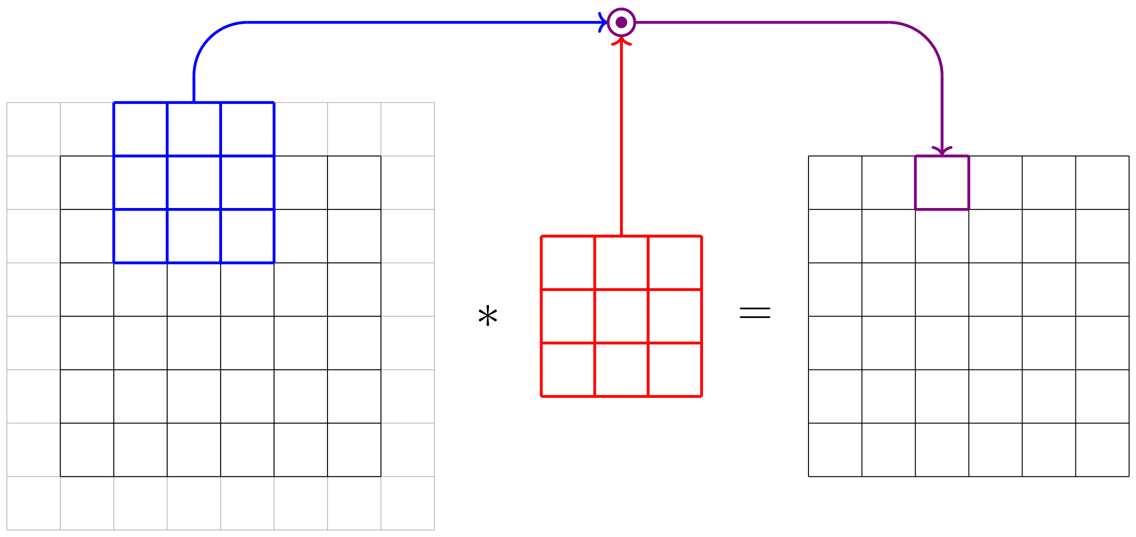
Figure 2.18: A padding scheme that preserves image size.

Pooling. Convolutional networks usually include a second type of layer called a pooling layer. These layers have no parameters. They simply serve to shrink images in a simple way. The image is divided into $k \times k$ pools (k is usually 2) which are then collapsed down to one pixel. The two most common forms are average pooling and max pooling, which take the average and maximum value of the pixels in the pool respectively. Pooling is applied to each channel separately. When an image dimension isn't divisible by k, padding is used to compensate.

Pooling is done in part for computational convenience. Convolutions are expensive. If an image can be downsized without losing too much information, it can speed things up considerably. Usually, pooling layers are used after several convolutional layers have been used to extract information about local image features. Each convolutional layer tends to have more filters than the previous. So nothing is really lost in pooling. The network takes a large grid with a few channels and turns it into a smaller grid with many channels.

Whereas the original few channels contained low-level information (usually the presence of a color in a pixel), the new channels contain higher level information about the presence of shapes and patterns. Most convolutional networks contain many convolutional layers,

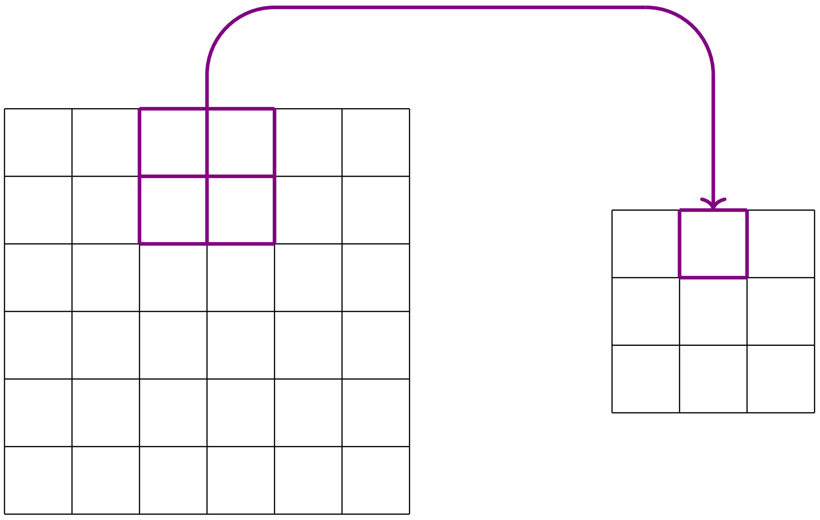

Figure 2.19: A 2×2 pooling operation.

interspersed with a few pooling layers. In fact, many convolutional networks end with a global pooling layer, which shrinks the image down to a single pixel (with many channels).

Receptive Fields. It may seem counterintuitive that convolutional networks can get away with using only small filters. In fact, state-of-the-art image recognition networks often use 3×3 filters exclusively, yet are able to detect arbitrarily large objects in images. This is because each filter with $k > 1$ increases the *receptive field* - the size of the area in the original image that affects any one pixel of a given convolutional layer's output.

Consider the following example, in which an original image is convolved with two 3×3 filters consecutively. A pixel in the second image is affected by a 3×3 region of the first, and likewise a pixel in the third image is affected by a 3×3 region of the second. So a pixel in the third image is affected by a 5×5 region of the first image.

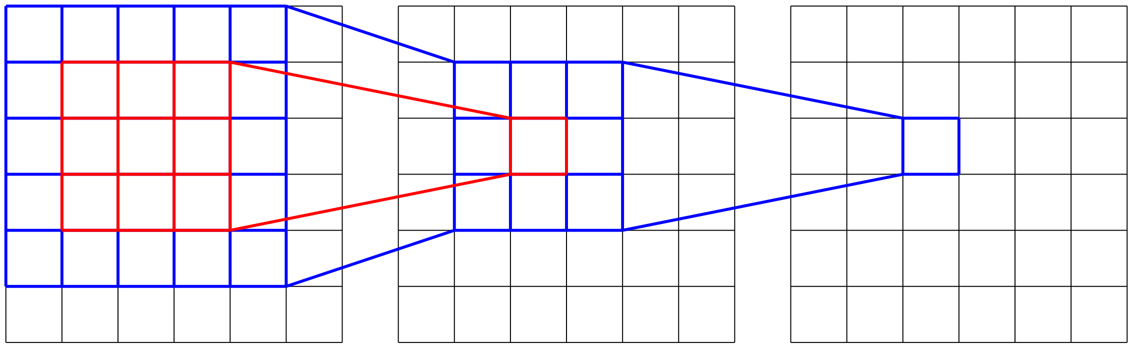

Figure 2.20: After two 3×3 convolutions, a single pixel has a 5×5 receptive field.

Depthwise Convolutions. Convolutions as described above have been used for many years and continue to be the standard tool for image processing tasks. Recently, a different type of convolution has gained popularity, particularly in applications where models with fewer parameters are desirable. Unlike traditional convolutions, these 'depthwise' convolutions have one single-channel filter for each input channel. Each input channel is convolved with its filter to produce an output channel. Thus $c_{out} = c_{in}$.

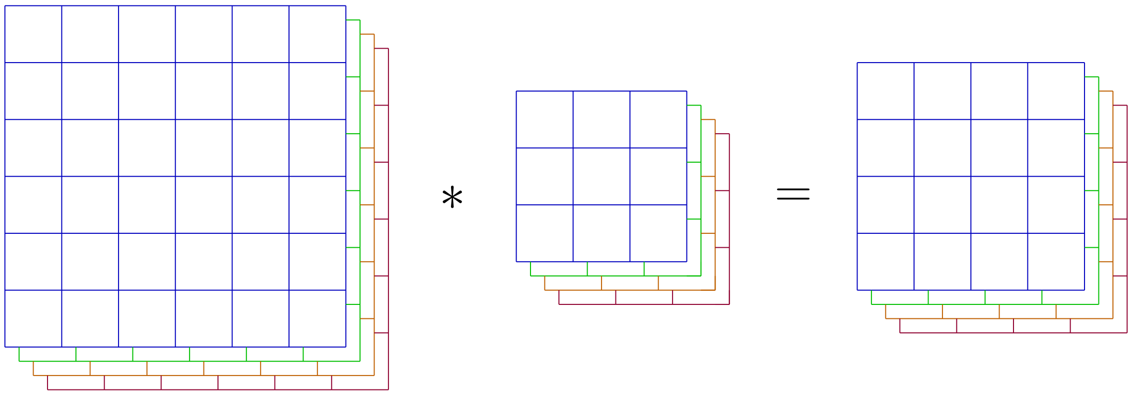

Figure 2.21: A deptwise convolution.

Depthwise convolutions are usually used in conjunction with 1×1 convolutions, which simply produce output channels that are linear combinations of their input channels and are convolutions only in a trivial sense.

1D Convolutions. Many forms of data have a natural sequence structure. They can be expressed as an ordered list of elements. All forms of time series data fit this description, as do text, audio, video, and others. Two main tools exist for exploiting this sequence structure — 1D convolutions and recurrent units.

Convolutions in one dimension work just as they do in two. If anything, they're simpler. A filter of length k is convolved with a sequence of length n to produce a filtered sequence of length $n - k + 1$.

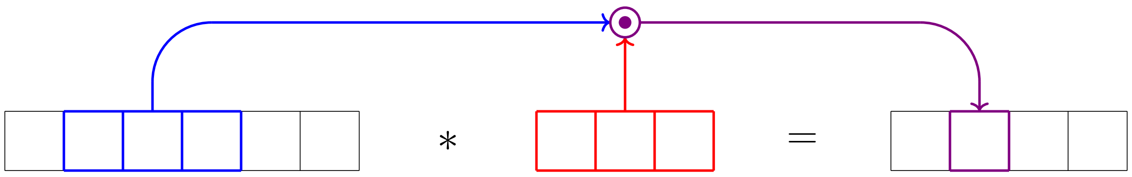

Figure 2.22: A 1D Convolution

If a data point is a sequence of vectors, then each dimension of those vectors can be thought of as a channel. Then filters must match the channel depth of the sequence with which they are convolved, and multiple filters can be used to produce a multi-channel output sequence. This approach, which is analogous to the one described above for images, is useful for many types of sequence data.

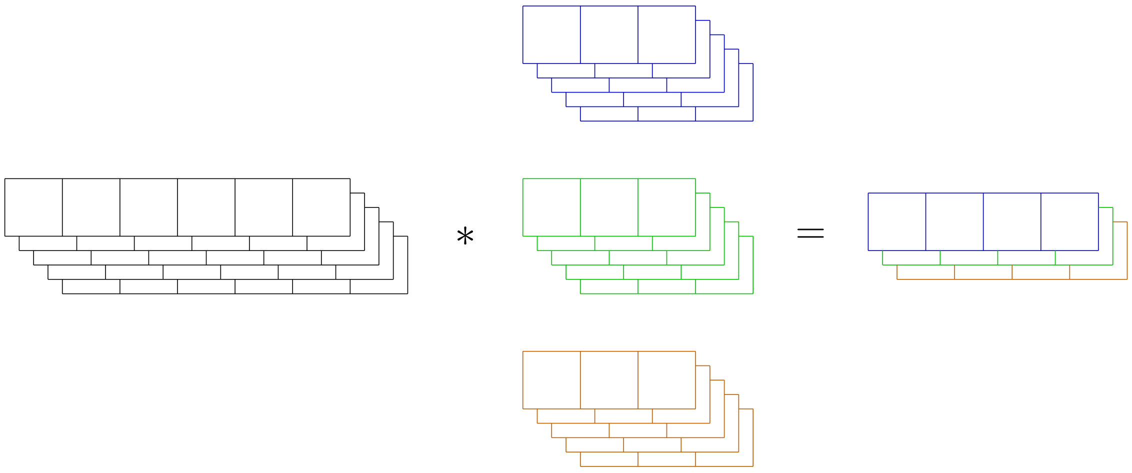

Figure 2.23: A 1D Convolution on Multi-Channeled Input

Other convolutional network techniques, such as pooling and padding, also translate easily to the one-dimensional case.

Chapter 3. Crelu Networks

In this chapter, we motivate the use of fully-connected neural networks, define and motivate the crelu activation scheme, do away with the bias parameter, define the most common loss functions, and work out the formulas for gradient descent in both recursive (for backpropagation) and explicit forms.

3.1 Depth and Universality

The goal of training a feedforward neural network is to find a set of parameters which cause the network to map its training inputs as near as possible (as quantified by the loss function) to their corresponding outputs. This raises an obvious and critical question: Does such a set of parameters exist?

It has been known for decades that neural networks with as few as one hidden layer are universal approximators — under a few easily-satisfied conditions, they can approximate any well-behaved function from one vector space to another with arbitrary precision [1, 2, 3, 4, 5, 6, 7, 8]. Several universal approximation theorems were proven in the late twentieth century, each placing different constraints on the space of functions to be approximated and the activation function used by the neural network.

Definition 3.1. In the case of a single variable $x \in \mathbb{R}$, $\mathrm{relu}(x) = \max(0, x)$. We will also refer to relu as acting on vectors in $\mathbb{R}^d$. In this case, the function is applied to each element of the vector individually. That is,

$$\mathrm{relu}\left(\begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}\right) = \begin{bmatrix} \mathrm{relu}(x_1) \\ \vdots \\ \mathrm{relu}(x_d) \end{bmatrix}.$$

Definition 3.2. A fully connected relu layer has a weight matrix $W \in \mathbb{R}^{d_{out} \times d_{in}}$ and a bias vector $b \in \mathbb{R}^{d_{out}}$. The layer takes an input $x \in \mathbb{R}^{d_{in}}$ and returns an output $\text{relu}(Wx + b) \in \mathbb{R}^{d_{out}}$. The entries of W and b are treated as parameters which can be adjusted in order to change the layer's output. We will refer to d_{out} as the layer's 'width.'

Definition 3.3. A fully connected relu network consists of a number of fully-connected relu layers, each with its own weight matrix and bias vector, and an output layer. The first relu layer acts on the network input. Each subsequent relu layer acts on the previous layer's output. The output layer also has a weight matrix and a bias vector, but does not have an activation function. It acts on the output of the last relu layer, and the value it returns is the output of the network.

Theorem 3.4. *(Hornik) Let $a : \mathbb{R} \to \mathbb{R}$ be continuous and nonpolynomial. Let $X \subset \mathbb{R}^d$ be compact, and let $C(X)$ be the set of continuous functions from X to $\mathbb{R}$. Let F be the set of functions of the form*

$$\sum_{i=1}^{n} c_i a\left(w_i \cdot x + b_i\right),$$

where $w_i \in \mathbb{R}^d$ and $b_i, c_i \in \mathbb{R}$. The closure of F with respect to the uniform topology contains $C(X)$.

Proof. See [6]. $\square$

Corollary 3.5. *Let $X \subset \mathbb{R}^{d_{in}}$ be compact. Let f be a continuous function from X to $\mathbb{R}^{d_{out}}$ and let $\epsilon > 0$. There exists a fully-connected relu network N with a single hidden layer such that*

$$\sup_{x \in X} \|f(x) - f_N(x)\| \leq \epsilon.$$

Proof. Relu is continuous and nonpolynomial, so it meets the conditions on the activation function. The function f can be broken into d_{out} component functions $f_k : X \to \mathbb{R}$. By the theorem, there exists a network N_k for each k such that

$$\sup_{x \in X} \|f_k(x) - f_{N_k}(x)\| \leq \epsilon / d_{out}.$$

These can be combined into a single network N by concatenating the weight matrices and bias vectors of the first layers vertically and by assembling the weight matrices of the output layer into a diagonal block matrix. Thus, by subadditivity of suprema, N is a satisfactory approximator. $\square$

These theorems generally rely on the width of the network to give it expressive power. That is, they guarantee that a network with a single hidden layer can be made to approximate a given function arbitrarily well if that layer is given a sufficiently large number of neurons. However, the wide and shallow networks suggested by these theorems are rarely used in practice. Deep networks, on the other hand, have been the key to breakthroughs in many fields of study and have accordingly become some of the most celebrated tools in machine learning [9].

Many explanations have been given for the tremendous success of deep networks, and this continues to be an area of very active research as many in the field seek to understand why these models work so well [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]. In recent years, several authors have proven that neural networks can be given arbitrary approximation capability through increasing depth rather than increasing width [22, 23, 24, 25, 26, 27]. In particular, each of these results establishes a minimum width. So long as each hidden layer has at least this number of neurons, universal approximation can be achieved solely by adding sufficiently many layers.

Theorem 3.6. *(Hanin et. al.) Define $w_{min}(d_{in}, d_{out})$ to be the minimal value of w such that for every continuous function $f : [0,1]^{d_{in}} \to \mathbb{R}^{d_{out}}$ and every $\epsilon > 0$ there is a network N with relu activations, input dimension d_{in}, hidden layer widths at most w, and output dimension d_{out} that ϵ-approximates f in the sense that*

$$\sup_{x\in[0,1]^{d_{in}}} \|f(x) - f_N(x)\| \leq \epsilon.$$

For all d_{in}, d_{out},

$$d_{in} + 1 \leq w_{min}(d_{in}, d_{out}) \leq d_{in} + d_{out}.$$

Proof. See [27]. □

Corollary 3.7. *Let $X \subset \mathbb{R}^{d_{in}}$ be compact. Let f be a continuous function from X to $\mathbb{R}^{d_{out}}$ and let $\epsilon > 0$. There exists a fully-connected relu network N with hidden layers of width not greater than $d_{in} + d_{out}$ such that*

$$\sup_{x \in X} \|f(x) - f_N(x)\| \leq \epsilon.$$

Proof. Because X is compact, there exist a $c > 0$ and $s \in \mathbb{R}^{d_{in}}$ such that $X' = cX + s \subseteq [0,1]^{d_{in}}$. Let $f'(x) = f((x-s)/c)$ on X'. By the theorem, there exists a satisfactory network N'. Let N be a copy of N' in which the weight matrix W' and bias vector b' of the first layer have been replaced with $W = W'/c$ and $b = b' - Ws$. The new network N ϵ-approximates f as desired. □

Both the older universality-by-width and the newer universality-by-depth results have been shown to apply to networks with relu activation functions, which is critical because this is by far the most common choice of activation function in practice. It will follow immediately that these results also apply to the class of crelu networks because they contain relu networks as subnetworks and thus inherit their expressivity.

3.2 Concatenated Relu

In 2016, several different authors separately proposed the same new way of using the relu activation function [28, 29, 30]. By applying relu to both a vector and its negative, they gave their models desirable new properties and saw improvements in performance. Following Shang et. al., we will use the term crelu (short for concatenated relu) to refer to the following operation on a vector.

Definition 3.8. The crelu activation function acts on a vector in $\mathbb{R}^d$ by

$$\text{crelu}(x) = \begin{bmatrix} \text{relu}(x) \\ \text{relu}(-x) \end{bmatrix}.$$

The output of the crelu activation is a vector in $\mathbb{R}^{2d}$.

Each group of authors gave a different motivation for this proposal. Shang et. al. had noticed a tendency of convolutional networks to learn filters in positive-negative pairs. For them, crelu was a tool that eliminated unnecessary parameters and the corresponding computation [28]. Blot et. al. used crelu to avoid the information loss caused when relu maps all negative inputs to zero [29]. Kim et. al. used crelu in order to more closely imitate the way human retinas process light [30]. In each case, the authors saw improved performance on image recognition tasks.

Here, we use crelu as the activation function in general-purpose fully-connected neural networks, not image-processing convolutional networks. So motivations specific to the domain of computer vision are not relevant in this case. Instead, we choose crelu because it has all of the characteristics that have made relu so successful while also having mathematical properties that make it possible to analyze the network with probabilistic tools and choose hyperparameters according to this analysis.

Lemma 3.9. *Crelu networks are also universal approximators in the sense of both Corollary 3.5 and Corollary 3.7.*

Proof. Let $\mathcal{C}$ be an operator that acts on fully-connected relu networks by replacing relus with crelus and weight matrices W following relus with $\begin{bmatrix} W & 0 \end{bmatrix}$, where 0 represents a zero matrix with the same dimensions as W. The crelu matrix $\mathcal{C}(N)$ is equivalent to N in the sense that it returns the same output for any input. □

All of the aforementioned authors noted that crelu preserves information when processing network inputs. Without an explicit definition, this could be taken to mean simply that crelu is an invertible function, but the same can be said of sigmoid activation functions that don't double the number of dimensions of their inputs. Relu variants such as leaky relu and parametric relu also have this property. What makes crelu special is that it also preserves the relative scale of its inputs. In order for a traditional activation function (one that does not change the number of dimensions) to do this, it would have to have a constant slope. But then it would be an affine function and unsuitable as an activation per the universality theorems referenced in Section 3.1.

Lemma 3.10. *If $a : \mathbb{R} \to \mathbb{R}$ be continuous, injective, and norm-preserving in the sense that $|a(x)| = |x|$, then $a(x)$ is linear and therefore unsuitable as an activation function.*

Proof. There are only four continuous functions from $\mathbb{R}$ to $\mathbb{R}$ with $|a(x)| = |x|$: x, $-x$, $|x|$, and $-|x|$. Of these, only x and $-x$ are injective. A layer using x or $-x$ as its activation function performs an affine transformation, and the composition of affine transformations is itself affine. Such layers are therefore completely redundant. □

Lemma 3.11. *Crelu is continuous, injective, and norm-preserving (for any p-norm) while also enabling universal approximation when used as an activation function.*

Proof. The first three properties follow immediately from the definition of crelu. The last is established by Lemma 3.9. □

Maintaining scale is important in practice. Sigmoid networks are an excellent example, as they have long been known to struggle with 'saturation' — the problem that occurs when the input to a sigmoid function is very high or very low and the corresponding slope is close to zero. In such cases, the neuron can 'die' and cease to receive meaningful parameter updates.

We will also take advantage of the fact that crelu is norm-preserving during both phases of backpropagation. During the forward pass, it maintains the magnitudes of the data being processed. During the backward pass, it does the same thing for the gradient information being sent back. This will make it possible to say how changes in initialization will affect the sizes of parameter updates, which will in turn allow us to avoid a common pitfall of gradient descent.

In addition, when we consider the weights of the network as random variables (an appropriate perspective due to random initialization), it will be clear that the choice of crelu activations results in well-behaved distributions for which we can compute the moments we need in order to determine the curvature of the loss surface and thus the appropriate choice of learning rate.

Finally, we will show that crelu activations make it possible to initialize the network such that the loss surface is smooth in a way that is not possible with traditional relu architectures, resulting in better-behaved gradients and both faster and more predictable learning. This advantage greatly increases our ability to train very deep networks in practice.

In short, crelu improves upon traditional relu by enabling us to analyze our networks and make correct hyperparameter choices rather than follow heuristics or play guess-and-check. It also enables faster, more consistent learning and greater network depths. We will show that these improvements come at a negligible cost in terms of real-world computation, suggesting that crelu would be a superior choice to relu in general.

3.3 Eliminating Biases

The traditional formulation of neural networks uses two types of parameters — weights and biases. With a little rearranging, we can arrive at an equivalent formulation that uses only weights. It's a small simplification, but one that makes the math significantly cleaner. It does require that the data have an extra dimension added in preprocessing, as we will note in that section.

Theorem 3.12. *Define $p : \mathbb{R}^d \to \mathbb{R}^{d+1}$ by*

$$p(x) = \begin{bmatrix} x \\ 1 \end{bmatrix}.$$

For every relu network N, there is another relu network N' that has no bias vectors but achieves $N'(p(x)) = p(N(x))$. So relu (and therefore crelu) networks do not need bias vectors to be universal approximators.

Proof. Define $q : \mathbb{R}^{m\times n} \times \mathbb{R}^m$ by

$$q(W, b) = \begin{bmatrix} W & b \\ 0 & 1 \end{bmatrix}.$$

Observe that $\text{relu}(q(W, b)p(x)) = p(\text{relu}(Wx + b))$. This shows that, once the input space has been transformed by p, all of the relu layers in a network can be replaced by relu layers without bias vectors. These new layers are one unit wider than the layers with bias they replace. □

Now we develop an equivalent formulation for crelu networks which will be more convenient for our purposes.

Definition 3.13. Let the Heaviside step function $h : \mathbb{R} \to \mathbb{R}$ be defined by

$$h(z) = \begin{cases} 0 & z < 0, \\ 1/2 & z = 0, \\ 1 & z > 0. \end{cases} \cdot$$

Definition 3.14. Let $A : \mathbb{R}^d \to \mathbb{R}^{d\times d}$ be the diagonal matrix

$$A(x) = \begin{bmatrix} h(x_1) & & & \\ & h(x_2) & & \\ & & \ddots & \\ & & & h(x_d) \end{bmatrix}.$$

Note that $A(x)x = \text{relu}(x)$ and $A(-x)x = -\text{relu}(-x)$. We will refer to the matrices produced by A as *activation matrices.*

Definition 3.15. A *split crelu layer* takes an input from $\mathbb{R}^{d_{in}}$ and returns an output in $\mathbb{R}^{d_{out}}$. The layer has two weight matrices $P, N \in \mathbb{R}^{d_{out} \times d_{in}}$. The output of the layer for an input x is given by

$$PA(x)x + NA(-x)x.$$

Note that this is equal to

$$\begin{bmatrix} P & -N \end{bmatrix} \text{crelu}(x).$$

Definition 3.16. A *split crelu network* consists only of split crelu layers. The inputs to the network come from $X \subseteq R^{d_{in}}$. Let $d_0 = d_{in} + 1$. Let $d_1, \ldots, d_n$ be the output dimensions for the n layers of the network respectively, with d_n being equal to the dimension of the desired output space.

Let $x_0^k = p(x^k)$, where $x^k \in X$ and $p(x) = \left[\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right]$. For $1 \leq i \leq n$,

$$x_i^k = P_i A(x_{i-1}^k) x_{i-1}^k + N_i A(-x_{i-1}^k) x_{i-1}^k,$$

where $P_i, N_i \in \mathbb{R}^{d_i \times d_{i-1}}$ are the weight matrices of the ith layer.

In a computer implementation, using diagonal matrices to compute the activations would be wasteful. It might also be more efficient to concatenate P_i and N_i into one large weight matrix for reduced overhead. The point of the split formulation is to facilitate mathematical analysis, and the insights from this analysis apply regardless of the details of an implementation.

3.4 Loss Functions

We define the two most common and important loss functions and give the gradients and Hessians we'll need to do quadratic approximations down the road. We also establish the framework which one would use to apply the techniques and results in this paper to other loss functions and give the conditions under which this would be valid.

Definition 3.17. The *squared error* loss function takes a network output x_n and its corresponding target y and returns the loss value

$$L(x_n, y) = (x_n - y)^2.$$

The gradient of this loss with respect to x_n is

$$\nabla x_n = 2(x_n - y).$$

The Hessian with respect to x_n is

$$\mathrm{H}(x_n) = 2I.$$

Definition 3.18. Define logsumexp : $\mathbb{R}^d \to \mathbb{R}$ by

$$\operatorname{logsumexp}(x) = \ln\left(e^{x_1} + e^{x_2} + \cdots + e^{x_d}\right)$$

and softmax : $\mathbb{R}^d \to \mathbb{R}^d$ by

$$\operatorname{softmax}(x) = \frac{1}{e^{x_1} + e^{x_2} + \cdots + e^{x_d}} \begin{bmatrix} e^{x_1} \\ e^{x_2} \\ \vdots \\ e^{x_d} \end{bmatrix}.$$

Note that logsumexp is a smooth approximation to the max function and softmax is a smooth approximation to the argmax function.

The *categorical cross-entropy* loss function takes a network output x_n and its corresponding target y and returns the loss value

$$L(x_n, y) = \operatorname{logsumexp}(x_n) - y^T x_n.$$

The gradient of this loss with respect to x_n is

$$\nabla x_n = \operatorname{softmax}(x_n) - y.$$

The Hessian with respect to x_n is

$$\mathrm{H}(x_n) = \operatorname{diag}(\operatorname{softmax}(x_n)) - \operatorname{softmax}(x_n)\operatorname{softmax}(x_n)^T.$$

Note that y must be a one-hot vector (one entry is 1 and the rest are 0) in order for this loss function to work as intended.

3.5 Backpropagation

We derive the formulas for calculating the weight matrix updates recursively, then expand these to give the explicit formula for a given weight matrix update.

Definition 3.19. For convenience, define

$$\begin{aligned} R_i^k &= A(x_{i-1}^k) \\ L_i^k &= A(-x_{i-1}^k) \\ W_i^k &= P_i R_i^k + N_i L_i^k \end{aligned}$$

Definition 3.20. Let ∇^k denote partial derivatives of $L(x_n^k, y^k)$. For example,

$$\nabla^k P_i = \partial L(x_n^k, y^k)/\partial P_i.$$

Lemma 3.21. *For a given input x_0^k,*

$$\begin{aligned}
\nabla^k x_i^k &= (W_{i+1}^k)^T \nabla^k x_{i+1}^k = (P_{i+1}A(x_i^k) + N_{i+1}A(-x_i^k))^T \nabla^k x_{i+1}^k, \\
\nabla^k P_i &= \nabla^k x_i^k (x_{i-1}^k)^T R_i^k = \nabla^k x_i^k (x_{i-1}^k)^T A(x_{i-1}^k), \\
\nabla^k N_i &= \nabla^k x_i^k (x_{i-1}^k)^T L_i^k = \nabla^k x_i^k (x_{i-1}^k)^T A(-x_{i-1}^k).
\end{aligned}$$

Proof. These are obtained by differentiating

$$x_i^k = P_i A(x_{i-1}^k) x_{i-1}^k + N_i A(-x_{i-1}^k) x_{i-1}^k,$$

with respect to x_{i-1}^k (and then shifting the indices), P_i, and N_i respectively. □

Definition 3.22. For further convenience, define

$$W_{a,b}^k = W_a^k W_{a-1}^k \cdots W_{b+1}^k W_b^k$$

for $a \geq b$. If $a < b$, let $W_{a,b}^k = 1$.

Lemma 3.23. *The gradients with respect to P_i and N_i for a given data point x_0^k are*

$$\begin{aligned}
\nabla^k P_i &= (W_{n,i+1}^k)^T \nabla^k x_n^k (x_0^k)^T (W_{i-1,1}^k)^T R_i^k, \\
\nabla^k N_i &= (W_{n,i+1}^k)^T \nabla^k x_n^k (x_0^k)^T (W_{i-1,1}^k)^T L_i^k.
\end{aligned}$$

Proof. By applying the definition of a split crelu layer recursively, we obtain

$$x_i^k = W_{i,1}^k x_0^k.$$

Similarly, repeated application of the first backpropagation rule from Lemma 3.21 yields

$$\nabla^k x_i^k = (W_{n,i+1}^k)^T \nabla^k x_n^k.$$

Combining these two gives us

$$\nabla^k x_i^k (x_{i-1}^k)^T = (W_{n,i+1}^k)^T \nabla^k x_n^k (x_0^k)^T (W_{i-1,1}^k)^T.$$

Multiplying by L_i^k and R_i^k completes the proof. □

Definition 3.24. Given a set of data points $x_0^1, x_0^2, \ldots, x_0^s \in \mathbb{R}^{d_0}$ and corresponding targets $y^1, y^2, \ldots, y^s \in \mathbb{R}^{d_n}$, the *total network loss* for a network N is

$$L_N = \frac{1}{s} \sum_{k=1}^{s} L(x_n^k, y^k).$$

Definition 3.25. Given a *learning rate* $\lambda > 0$, the update matrices ΔP_i and ΔN_i are defined as

$$\Delta P_i = \frac{1}{s} \sum_{k=1}^{s} \nabla^k P_i$$
$$\Delta N_i = \frac{1}{s} \sum_{k=1}^{s} \nabla^k N_i,$$

for $1 \leq i \leq n$. A single step of gradient descent consists of performing the parameter updates

$$P_i \leftarrow P_i - \lambda \Delta P_i$$
$$N_i \leftarrow N_i - \lambda \Delta N_i,$$

for $1 \leq i \leq n$. The network is *trained* by repeatedly updating its weight matrices in this manner, until some stopping condition is reached.

Chapter 4. Optimal Hyperparameters

In this chapter, we answer the questions that naturally arise when a chosen architecture is to be trained: How should inputs and outputs be preprocessed so as to better condition the optimization problem? How should parameters be initialized? What should be the initial learning rate? How should the learning rate change over the course of training? In each case, we choose a straightforward metric for success and optimize accordingly.

4.1 Initialization Scale

We show that initializing with centered Gaussians and scaling according to the size (number of entries) of each weight matrix leads to proportional updates. We will refer to independent and identically distributed random variables as being i.i.d. for brevity.

Lemma 4.1. *Let $x \in \mathbb{R}^d$ consist of independent standard normal variables. That is, $x_i \sim \mathcal{N}(0,1)$ and the x_i are all independent. Let $f : \mathbb{R}^d \to \mathbb{R}$ be the probability density function for x. Then f is spherically symmetric in the sense that $f(Mx) = f(x)$ for any orthogonal matrix $M \in \mathbb{R}^{d\times d}$.*

Proof. Because the entries of x are independent, f is simply the product of the pdf for each. Hence

$$f(x) = \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi}} e^{-\frac{-x_i^2}{2}} = (2\pi)^{-\frac{d}{2}} e^{-\frac{x_1^2+x_2^2+\cdots+x_d^2}{2}} = (2\pi)^{-\frac{d}{2}} e^{-\frac{\|x\|^2}{2}}.$$

If M is orthogonal, then $M^T M = I$. So $\|Mx\|^2 = x^T M^T M x = x^T x = \|x\|^2$. Because $f(x)$ can be written as a function of $\|x\|^2$ as above, $f(Mx) = f(x)$. $\square$

The converse is also true and was proven by J.C. Maxwell.

Lemma 4.2. *(Maxwell) Let $x \in \mathbb{R}^d$ consist of i.i.d. random variables and let x have a spherically symmetric distribution. The elements of x are normally distributed with mean zero.*

Proof. See [31]. $\square$

Lemma 4.3. *If $M \in \mathbb{R}^{c\times d}$ is a matrix consisting of i.i.d. standard normal variables, then $\mathrm{E}(M^TM) = cI$. In this sense, M is expected to be an orthogonal matrix, up to a scaling factor of $\sqrt{c}$.*

Proof. Let M_i be the ith column of M. The product $M_i^TM_i$ is the sum of the squares of c independent standard normal variables, so $M_i^TM_i \sim \chi^2(c)$. This tells us that $\mathrm{E}[M_i^TM_i] = c$. Let $j \neq i$. So $M_i^TM_j = m_{1i}m_{ij} + m_{2i}m_{2j} + \cdots + m_{di}m_{di}$. Because m_{ki} and m_{kj} are independent, $\mathrm{E}[m_{ki}m_{kj}] = \mathrm{E}[m_{ki}]\,\mathrm{E}[m_{kj}] = 0$. $\square$

Lemma 4.4. *Let x_1 and x_2 be independent normal variables with means μ_1 and μ_2 and variances σ_1^2 and σ_2^2 respectively. The sum $x_1 + x_2$ is a normal variable with mean $\mu_1 + \mu_2$ and variance $\sigma_1^2 + \sigma_2^2$.*

Proof. This can be verified by convolving the probability density functions of x_1 and x_2. Alternately, see [33]. $\square$

Lemma 4.5. *Let $M \in \mathbb{R}^{c\times d}$ be a matrix consisting of i.i.d. standard normal variables. Let $x \in \mathbb{R}^d$ be a fixed unit vector. The entries of $y = Mx$ are i.i.d. standard normal variables. In particular, this implies that y is spherically symmetric, that $\|y\|^2 \sim \chi^2(c)$, and that $\mathrm{E}[\|y\|^2] = c$.*

Proof. The ith element of y is $y_i = m_{i1}x_1 + m_{i2}x_2 + \cdots + m_{id}x_d$. By Lemma 4.4, $y_i \sim \mathcal{N}(0, x_1^2 + x_2^2 + \cdots + x_d^2) = \mathcal{N}(0,1)$. The elements of y are independent from one another because they depend on disjoint subsets of the elements of M, all of which are independent from one another. $\square$

Theorem 4.6. *Let all of the entries of all of the P_i and N_i be normal random variables with mean zero. Denote by P_{ijk} the entry in the jth row and kth column of P_i, and by N_{ijk} the corresponding entry in N_i. Let all of the parameters in a given layer have the same variance $\sigma_i^2 > 0$. That is, $P_{ijk}, N_{ijk} \sim \mathcal{N}(0, \sigma_i^2)$. Let P_{ijk} be independent from P_{qrs} and N_{qrs} if $i \neq q$,*

$j \neq r$, or $k \neq s$. That is to say that P_{ijk} is independent from every other network parameter except perhaps N_{ijk}. Likewise let N_{ijk} be independent from every other network parameter except perhaps P_{ijk}. Then

$$\frac{\mathrm{E}[\|\Delta P_i\|^2]}{\mathrm{E}[\|\Delta P_j\|^2]} = \frac{\mathrm{E}[\|P_i\|^2]}{\mathrm{E}[\|P_j\|^2]} \quad \textit{and} \quad \frac{\mathrm{E}[\|\Delta N_i\|^2]}{\mathrm{E}[\|\Delta N_j\|^2]} = \frac{\mathrm{E}[\|N_i\|^2]}{\mathrm{E}[\|N_j\|^2]}$$

if and only if

$$\frac{\sigma_i^2}{\sigma_j^2} = \frac{\sqrt{d_j d_{j-1}}}{\sqrt{d_i d_{i-1}}},$$

where d_0 is the input dimension and d_i is the dimension of the output of the ith layer.

Proof. Because P_i is a $d_i \times d_{i-1}$ matrix of independent normal variables with mean zero and variance σ_i^2, $\sigma_i^{-2}\|P_i\|^2 \sim \chi^2(d_i d_{i-1})$ and $\mathrm{E}[\|P_i\|^2] = \sigma_i^2 d_i d_{i-1}$. This tells us that

$$\frac{\mathrm{E}[\|P_i\|^2]}{\mathrm{E}[\|P_j\|^2]} = \frac{\sigma_i^2 d_i d_{i-1}}{\sigma_j^2 d_j d_{j-1}}.$$

Recall that

$$\nabla^k P_i = (W_{n,i+1}^k)^T \nabla^k x_n^k (x_0^k)^T (W_{i-1,1}^k)^T R_i^k.$$

By applying Lemma 4.5 repeatedly, we obtain

$$\begin{aligned}
\mathrm{E}[\|(W_{n,i+1}^k)^T \nabla^k x_n^k\|^2] &= \mathrm{E}[\|(W_{i+1}^k)^T (W_{i+2}^k)^T \cdots (W_n^k)^T \nabla^k x_n^k\|^2] \\
&= \sigma_{i+1}^2 d_i \, \mathrm{E}[\|(W_{i+2}^k)^T (W_{i+3}^k)^T \cdots (W_n^k)^T \nabla^k x_n^k\|^2] \\
&\ \ \vdots \\
&= (\sigma_{i+1}^2 \sigma_{i+2}^2 \cdots \sigma_n^2)(d_i d_{i+1} \cdots d_{n-1}) \|\nabla^k x_n^k\|^2.
\end{aligned}$$

By the same method, we obtain

$$\begin{aligned}
\mathrm{E}[(x_0^k)^T(W_{i-1,1}^k)^T] &= \mathrm{E}[W_{i-1,1}^k x_0^k] \\
&= \mathrm{E}[W_{i-1}^k W_{i-2}^k \cdots W_1^k x_0^k] \\
&= \sigma_{i-1}^2 d_{i-1}\,\mathrm{E}[W_{i-2}^k W_{i-3}^k \cdots W_1^k x_0^k] \\
&\vdots \\
&= (\sigma_1^2 \sigma_2^2 \cdots \sigma_{i-1}^2)(d_1 d_2 \cdots d_{i-1})\|x_0^k\|^2.
\end{aligned}$$

Because $(W_{n,i+1}^k)^T \nabla^k x_n^k$ and $(x_0^k)^T(W_{i-1,1}^k)^T$ are independent and $\|uv^T\|^2 = \|u\|^2\|v\|^2$ for column vectors u and v, we can simply multiply to obtain

$$\mathrm{E}[\|(W_{n,i+1}^k)^T \nabla^k x_n^k (x_0^k)^T (W_{i-1,1}^k)^T\|^2] = \frac{1}{\sigma_i^2}(\sigma_1^2\sigma_2^2\cdots\sigma_n^2)(d_1 d_2 \cdots d_{n-1})\|x_0^k\|^2\|\nabla^k x_n^k\|^2.$$

Because $\nabla^k P_i = (W_{n,i+1}^k)^T \nabla^k x_n^k (x_0^k)^T (W_{i-1,1}^k)^T R_i^k$,

$$\mathrm{E}[\|\nabla^k P_i\|^2] = \frac{1}{2\sigma_i^2}(\sigma_1^2\sigma_2^2\cdots\sigma_n^2)(d_1 d_2 \cdots d_{n-1})\|x_0^k\|^2\|\nabla^k x_n^k\|^2,$$

as long as the elements of x_{i-1} have median zero. This is certainly the case for $i > 1$ because these elements are normal and centered per Lemma 4.5. In the case of the raw inputs x_0, this can be guaranteed with preprocessing.

Because

$$\Delta P_i = \frac{1}{s}\sum_{k=1}^{s} \nabla^k P_i,$$

it follows that

$$\frac{\mathrm{E}[\|\Delta P_i\|^2]}{\mathrm{E}[\|\Delta P_j\|^2]} = \frac{\sigma_j^2}{\sigma_i^2}.$$

By setting this ratio equal to the ratio of the expected squared magnitudes of the weight matrices themselves, we get

$$\frac{\sigma_j^2}{\sigma_i^2} = \frac{\sigma_i^2 d_i d_{i-1}}{\sigma_j^2 d_j d_{j-1}}.$$

This simplifies to

$$\frac{\sigma_i^2}{\sigma_j^2} = \frac{\sqrt{d_j d_{j-1}}}{\sqrt{d_i d_{i-1}}}$$

as desired. All of the same logic applies to the N_i and ΔN_i as well, completing the proof. □

Corollary 4.7. *If there is a constant $c > 0$ such that*

$$\sigma_i = \frac{c}{(d_i d_{i-1})^{1/4}},$$

for all $1 \leq i \leq n$, then

$$\mathrm{E}[\|x_n^k\|^2] = c^{2n}\sqrt{\frac{d_n}{d_0}}\|x_0^k\|^2.$$

Hence $c = 1$ is the only choice that doesn't cause the network output's magnitude to scale exponentially with network depth.

Proof. Again by applying Lemma 4.5,

$$\begin{aligned}
\mathrm{E}[\|x_n^k\|^2] &= (\sigma_1^2\sigma_2^2\cdots\sigma_n^2)(d_1 d_2\cdots d_n)\|x_0^k\|^2\\
\mathrm{E}[\|x_n^k\|^2] &= \left(\frac{c^2}{\sqrt{d_1 d_0}}\frac{c^2}{\sqrt{d_2 d_1}}\cdots\frac{c^2}{\sqrt{d_n d_{n-1}}}\right)(d_1 d_2\cdots d_n)\|x_0^k\|^2\\
\mathrm{E}[\|x_n^k\|^2] &= \left(\frac{c^{2n}}{\sqrt{d_0}d_1 d_2\cdots d_{n-1}\sqrt{d_n}}\right)(d_1 d_2\cdots d_n)\|x_0^k\|^2\\
\mathrm{E}[\|x_n^k\|^2] &= c^{2n}\sqrt{\frac{d_n}{d_0}}\|x_0^k\|^2.
\end{aligned}$$

The c^{2n} term vanishes when $c = 1$. □

Definition 4.8. In a *proportional independent initialization* for a crelu network, the entries of each P_i and N_i are sampled independently from a normal distribution with mean zero and variance $\sigma_i^2 = (d_i d_{i-1})^{-1/2}$.

In a *proportional symmetric initialization* for a crelu network, the entries of each P_i are sampled independently from a normal distribution with mean zero and variance $\sigma_i^2 = (d_i d_{i-1})^{-1/2}$ and then each N_i is set equal to the corresponding P_i.

In the rest of the paper, we will prefer the symmetric initialization. Note that this method causes the network to be linear upon initialization, in the sense that the outputs are linear functions of the inputs. When $P_i \neq N_i$, this is no longer the case. Balzduzzi et. al. have shown that this property makes gradient descent more effective [32]. In addition, having all of the W_i^k to be equal upon initialization is very helpful for analysis.

Remark. Looking at unsquared magnitudes instead of squared magnitudes yields almost exactly the same result. The only difference stems from the fact that the chi distribution appears instead of the chi-squared distribution. The mean of $\chi(d)$ is

$$\sqrt{2}\frac{\Gamma\left(\frac{d+1}{2}\right)}{\Gamma\left(\frac{d}{2}\right)},$$

which is very well approximated by $\sqrt{d}$ [33].

4.2 Preprocessing

After adding the extra dimension required to eliminate biases, we use a simple form of normalization to ensure that features contribute equally. We show how the scale to which data is normalized affects the scale of the outputs.

Lemma 4.9. *Let N be an n-layer crelu network and let $c > 0$. Let x_0 be a network input and x_n the corresponding network output $N(x_0)$. Then $N(cx_0) = cN(x_0) = cx_n$. Crelu networks preserve positive scalar multiplication.*

Proof. It is sufficient to prove that a single crelu layer preserves scalar multiplication. Let $x_i = L_i(x_{i-1})$ denote output of the ith layer for an input x_{i-1}. First, note that the step function $h(x)$ obeys $h(cx) = h(x)$ for $c > 0$. Thus $A(cx) = x$ and $A(-cx) = A(-x)$. Finally,

$$\begin{aligned} L(cx_{i-1}) &= P_i A(cx_{i-1}) cx_{i-1} + N_i A(-cx_{i-1}) cx_{i-1} \\ &= c(P_i A(x_{i-1}) x_{i-1} + N_i A(-x_{i-1}) x_{i-1}) \\ &= cx_i. \end{aligned}$$

Thus the entire crelu network preserves scaling in this fashion. □

Definition 4.10. We say that a dataset $\{x_0^k\}_{k=1}^s \subset R^{d_0}$ has been *standardized* when it meets the following conditions:

(i) There exists a constant $c \neq 0$ such that x_{0,d_0}^k, the d_0th entry of x_0^k, is equal to c for all $1 \leq k \leq s$.

(ii) The sum $\sum_{k=1}^s x_{0,i}^k = 0$ for all $1 \leq i < d_0$.

(iii) The sum $\sum_{k=1}^s (x_{0,i}^k)^2 = sc^2$ for all $1 \leq i < d_0$.

In the terminology of machine learning, the coordinates of the data points are called features. Standardization may be equivalently defined as follows:

(i) The last feature has the same nonzero value for all data points.

(ii) All of the features except the last have mean zero across the dataset.

(iii) All of the features have the same variance across the dataset.

Remark. In a standardized dataset, no two data points can be scalar multiples of each other unless they are identical. Thus Lemma 4.9 does not contradict Theorem 3.12, which asserts that crelu networks without bias are universal approximators under the constant-feature condition.

Lemma 4.11. *For an n-layer crelu network N with proportional initialization and a standardized dataset* $\{x_0^k\}_{k=1}^s$,

$$\mathrm{E}\left[\|x_n^k\|^2\right] = \sqrt{\frac{d_n}{d_0}} \|x_0^k\|^2.$$

Specifically, the coordinates of the x_n^k are normally distributed with mean zero and expected variance

$$\mathrm{E}\left[\frac{1}{s}\sum_{k=1}^{s}(x_{n,j}^k)^2\right] = \frac{c^2}{\sqrt{d_0 d_n}},$$

for $1 \leq j \leq d_n$, where c is the constant from Definition 4.10.

Proof. The first equation follows immediately from Corollary 4.7 and Definition 4.8. The second follows from the first and Lemma 4.5. □

Remark. The choice of the constant c in Definition 4.10 determines the scale of the network outputs, so the correct choice of c depends on both the loss function and the targets y^k.

4.3 Quadratic Approximation

We compute the second-order Taylor polynomial of the loss function as a function of the learning rate, then compute the expected values of the coefficients with respect to the random initialization scheme described previously. This will facilitate a discussion of the optimal learning rate and how it changes with network width and depth.

Definition 4.12. Let N be an n-layer crelu network to be trained on a dataset $\{x_0^k\}_{k=1}^s$ with targets $\{y^k\}_{k=1}^s$ and loss function L. Let the update matrices ΔP_i and ΔN_i be calculated as in Definition 3.25. Define N^λ to be the updated network produced when the weight matrices P_i and N_i are replaced with $P_i^\lambda = P_i + \lambda\Delta P_i$ and $N_i^\lambda = N_i + \lambda\Delta N_i$ respectively. Define $x_n^k(\lambda) = N^\lambda(x_0^k)$ and define $L_N(\lambda)$, the total network loss as a function of the learning rate, as

$$L_N(\lambda) = \frac{1}{s}\sum_{k=1}^{s} L(x_n^k(\lambda), y^k).$$

In other words, $L_N(0)$ is the current total loss and $L_N(\lambda)$ is what the new total loss would be after a single step of gradient descent with learning rate λ. These allow us to approximate

$L_N(\lambda)$ with a parabola by computing the second-degree Taylor polynomial

$$L_N(\lambda) \approx L_N(0) + \lambda \frac{d}{d\lambda} L_N(0) + \frac{1}{2}\lambda^2 \frac{d^2}{d\lambda^2} L_N(0).$$

Definition 4.13. The following shorthands will be useful when discussing the derivatives of $L_N(\lambda)$. Note that all of these are scalar quantities.

$$\begin{aligned}
\alpha_{i,k,l} &= (x_0^l)^T (W_{i-1,1}^l)^T \left(R_i^l R_i^k + L_i^l L_i^k\right) W_{i-1,1}^k x_0^k, \\
\omega_{i,k,l} &= (\nabla x_n^k)^T W_{n,i+1}^k (W_{n,i+1}^l)^T \nabla x_n^l, \\
\gamma_{i,j,k,l,m} &= (x_0^l)^T (W_{i-1,1}^l)^T \left(R_i^l R_i^k + L_i^l L_i^k\right) W_{i-1,j+1}^k (W_{n,j+1}^m)^T \nabla x_n^m, \\
\chi_{i,j,k,l,m} &= (\nabla x_n^m)^T W_{n,j+1}^m (W_{n,j+1}^k)^T \,\mathrm{H}(x_n^k) W_{n,i+1}^k (W_{n,i+1}^l)^T \nabla x_n^l.
\end{aligned}$$

Lemma 4.14. *In the notation of Definition 4.13, the first derivative of the total loss with respect to the learning rate is*

$$\frac{d}{d\lambda} L_N(0) = -\frac{1}{s^2} \sum_{i=1}^{n} \sum_{k=1}^{s} \sum_{l=1}^{s} \alpha_{i,k,l}\,\omega_{i,k,l}.$$

Proof. Differentiating the formula in Definition 4.12 gives

$$\frac{d}{d\lambda} L_N(0) = \frac{1}{s} \sum_{k=1}^{s} (\nabla x_n^k)^T \left(\frac{d}{d\lambda} x_n^k(0)\right).$$

Expanding this formula gives

$$\begin{aligned}
&\frac{d}{d\lambda}L_N(0)\\
&\quad= -\frac{1}{s}\sum_{k=1}^{s}(\nabla x_n^k)^T\sum_{i=1}^{n}W_{n,i+1}^k\Delta W_i^k W_{i-1,1}^k x_0^k\\
&\quad= -\frac{1}{s}\sum_{k=1}^{s}(\nabla x_n^k)^T\sum_{i=1}^{n}W_{n,i+1}^k\left(\Delta P_i R_i^k + \Delta N_i L_i^k\right)W_{i-1,1}^k x_0^k\\
&\quad= -\frac{1}{s^2}\sum_{k=1}^{s}(\nabla x_n^k)^T\sum_{i=1}^{n}W_{n,i+1}^k\sum_{l=1}^{s}\left(\nabla P_i^l R_i^k + \nabla N_i^l L_i^k\right)W_{i-1,1}^k x_0^k\\
&\quad= -\frac{1}{s^2}\sum_{k=1}^{s}(\nabla x_n^k)^T\sum_{i=1}^{n}W_{n,i+1}^k\sum_{l=1}^{s}(W_{n,i+1}^l)^T\nabla x_n^l(x_0^l)^T(W_{i-1,1}^l)^T\left(R_i^l R_i^k + L_i^l L_i^k\right)W_{i-1,1}^k x_0^k\\
&\quad= -\frac{1}{s^2}\sum_{i=1}^{n}\sum_{k=1}^{s}\sum_{l=1}^{s}(\nabla x_n^k)^T W_{n,i+1}^k(W_{n,i+1}^l)^T\nabla x_n^l(x_0^l)^T(W_{i-1,1}^l)^T\left(R_i^l R_i^k + L_i^l L_i^k\right)W_{i-1,1}^k x_0^k.
\end{aligned}$$

In terms of the shorthand from Definition 4.13, this becomes

$$\frac{d}{d\lambda}L_N(0) = -\frac{1}{s^2}\sum_{i=1}^{n}\sum_{k=1}^{s}\sum_{l=1}^{s}\alpha_{i,k,l}\omega_{i,k,l}.$$

Note that this value is necessarily nonpositive. □

Theorem 4.15. *When a crelu network is initialized with the proportional symmetric initialization, the expected first derivative of the loss function with respect to the learning rate is*

$$\mathrm{E}\left[\frac{d}{d\lambda}L_N(0)\right] = -\sum_{i=1}^{n}\frac{\sqrt{d_{i-1}d_i}}{\sqrt{d_0 d_n}}\cdot\frac{1}{s^2}\sum_{k=1}^{s}\sum_{l=1}^{s}(x_0^l)^T(x_0^k)(\nabla x_n^l)^T(\nabla x_n^k).$$

Proof. Corollary A.10 allows us to calculate $\mathrm{E}[\alpha_{i,k,l}\omega_{i,k,l}]$, while most of the work of derivation is done in Theorem A.9 and the lemmata it cites. □

Lemma 4.16. *In the notation of Definition 4.13, the second derivative of the total loss with respect to the learning rate is*

$$\begin{aligned}\frac{d^2}{d\lambda^2}L_N(0) = \frac{2}{s^3}\sum_{i=1}^{n}\sum_{j=1}^{i-1}\sum_{k=1}^{s}\sum_{l=1}^{s}\sum_{m=1}^{s}\alpha_{j,k,m}\gamma_{i,j,k,l,m}\omega_{i,k,l}\\ + \frac{1}{s^3}\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{l=1}^{s}\sum_{k=1}^{s}\sum_{m=1}^{s}\alpha_{j,k,m}\chi_{i,j,k,l,m}\alpha_{i,k,l}.\end{aligned}$$

Proof. Differentiating the formula in Definition 4.12 twice gives

$$\begin{aligned}\frac{d^2}{d\lambda^2}L_N(0) &= \frac{1}{s}\sum_{k=1}^{s}(\nabla x_n^k)^T\left(\frac{d^2}{d\lambda^2}x_n^k(0)\right) + \left(\frac{d}{d\lambda}x_n^k(0)\right)^T \mathrm{H}(x_n^k)\left(\frac{d}{d\lambda}x_n^k(0)\right)\\ &= \frac{1}{s}\sum_{k=1}^{s}(\nabla x_n^k)^T\left(\frac{d^2}{d\lambda^2}x_n^k(0)\right) + \frac{1}{s}\sum_{k=1}^{s}\left(\frac{d}{d\lambda}x_n^k(0)\right)^T \mathrm{H}(x_n^k)\left(\frac{d}{d\lambda}x_n^k(0)\right).\end{aligned}$$

We expand these two sums separately for convenience. The first becomes

$$\begin{aligned}&\frac{1}{s}\sum_{k=1}^{s}(\nabla x_n^k)^T\left(\frac{d^2}{d\lambda^2}x_n^k(0)\right)\\ &= \frac{2}{s}\sum_{i=1}^{n}\sum_{j=1}^{i-1}\sum_{k=1}^{s}(\nabla x_n^k)^T W_{n,i+1}^k \Delta W_i^k W_{i-1,j+1}^k \Delta W_j^k W_{j-1,1}^k x_0^k\\ &= \frac{2}{s^2}\sum_{i=1}^{n}\sum_{j=1}^{i-1}\sum_{k=1}^{s}\sum_{l=1}^{s}\omega_{i,k,l}(x_0^l)^T(W_{i-1,1}^l)^T\left(R_i^l R_i^k + L_i^l L_i^k\right)W_{i-1,j+1}^k \Delta C_j^k W_{j-1,1}^k x_0^k\\ &= \frac{2}{s^3}\sum_{i=1}^{n}\sum_{j=1}^{i-1}\sum_{k=1}^{s}\sum_{l=1}^{s}\sum_{m=1}^{s}\omega_{i,k,l}(x_0^l)^T(W_{i-1,1}^l)^T\left(R_i^l R_i^k + L_i^l L_i^k\right)W_{i-1,j+1}^k (W_{n,j+1}^m)^T\nabla x_n^m \alpha_{j,k,m}.\end{aligned}$$

So we can write

$$\frac{1}{s}\sum_{k=1}^{s}(\nabla x_n^k)^T\left(\frac{d^2}{d\lambda^2}x_n^k(0)\right) = \frac{2}{s^3}\sum_{i=1}^{n}\sum_{j=1}^{i-1}\sum_{k=1}^{s}\sum_{l=1}^{s}\sum_{m=1}^{s}\alpha_{j,k,m}\gamma_{i,j,k,l,m}\omega_{i,k,l}.$$

The second sum expands to become

$$\begin{aligned}
&\frac{1}{s}\sum_{k=1}^{s}\left(\frac{d}{d\lambda}x_n^k(0)\right)^T \mathrm{H}(x_n^k)\left(\frac{d}{d\lambda}x_n^k(0)\right)\\
&\quad = -\frac{1}{s^2}\sum_{i=1}^{n}\sum_{l=1}^{s}\sum_{k=1}^{s}((x_n^k)')^T \mathrm{H}(x_n^k)W_{n,i+1}^k(W_{n,i+1}^l)^T\nabla x_n^l\alpha_{i,k,l}\\
&\quad = \frac{1}{s^3}\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{l=1}^{s}\sum_{k=1}^{s}\sum_{m=1}^{s}\alpha_{j,k,m}(\nabla x_n^m)^T W_{n,j+1}^m(W_{n,j+1}^k)^T \mathrm{H}(x_n^k)W_{n,i+1}^k(W_{n,i+1}^l)^T\nabla x_n^l\alpha_{i,k,l}.
\end{aligned}$$

So we can write

$$\frac{1}{s}\sum_{k=1}^{s}\left(\frac{d}{d\lambda}x_n^k(0)\right)^T \mathrm{H}(x_n^k)\left(\frac{d}{d\lambda}x_n^k(0)\right) = \frac{1}{s^3}\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{l=1}^{s}\sum_{k=1}^{s}\sum_{m=1}^{s}\alpha_{j,k,m}\chi_{i,j,k,l,m}\alpha_{i,k,l}.$$

Note that the indexing is different for the two sums, with j running from 1 to $i-1$ in the former but from 1 to n in the latter. $\square$

Lemma 4.17. *When a crelu network is initialized with the proportional initialization, the expected value of the first sum from Lemma 4.16 is zero. That is,*

$$\mathrm{E}\left[\frac{1}{s}\sum_{k=1}^{s}(\nabla x_n^k)^T\left(\frac{d^2}{d\lambda^2}x_n^k(0)\right)\right] = 0.$$

Proof. This sum was shown in Lemma 4.16 to be equal to

$$\frac{2}{s^3}\sum_{i=1}^{n}\sum_{j=1}^{i-1}\sum_{k=1}^{s}\sum_{l=1}^{s}\sum_{m=1}^{s}\alpha_{j,k,m}\gamma_{i,j,k,l,m}\omega_{i,k,l},$$

with $\alpha_{j,k,m}$, $\gamma_{i,j,k,l,m}$, and $\omega_{i,k,l}$ defined as in Definition 4.13. It suffices to prove that

$$\mathrm{E}\left[\alpha_{j,k,m}\gamma_{i,j,k,l,m}\omega_{i,k,l}\right] = 0,$$

for all indices. This is done by defining

$$u = \alpha_{j,k,m}(x_0^l)^T(W_{i-1,1}^l)^T\left(R_i^lR_i^k + L_i^lL_i^k\right)W_{i-1,j+1}^k(W_{i-1,j+1}^m)^T,$$
$$v = (W_{n,i}^m)^T\nabla x_n^m\omega_{i,k,l}.$$

Note that $uv = \alpha_{j,k,m}\gamma_{i,j,k,l,m}\omega_{i,k,l}$. We can rewrite $v = (W_i^m)^T(W_{n,i+1}^m)^T\nabla x_n^m\omega_{i,k,l}$. Because $(W_i^m)^T$ and $(W_{n,i+1}^m)^T\nabla x_n^m\omega_{i,k,l}$ are independent and W_i is a matrix of Gaussians, v is easily seen to have a spherically symmetric distribution. Because u and v are independent, we have $\mathrm{E}[uv] = 0$. □

Definition 4.18. Let N be an n-layer crelu network with input dimension d_0 and layer widths $d_1, d_2, \ldots, d_n$. Let $1 \leq i, j \leq n$.

$$\varphi_{cd} = \frac{2}{3}\prod_{q=c}^{d-1}\left(1+\frac{2}{d_q}\right) + \frac{1}{3}\prod_{q=c}^{d-1}\left(1-\frac{1}{d_q}\right),$$
$$\psi_{cd} = \frac{1}{3}\prod_{q=c}^{d-1}\left(1+\frac{2}{d_q}\right) - \frac{1}{3}\prod_{q=c}^{d-1}\left(1-\frac{1}{d_q}\right).$$

Let $a = \min(i,j)$ and $b = \max(i,j)$. Further define

$$A_{ijklm} = \varphi_{1a}(x_0^k)^Tx_0^l(x_0^k)^Tx_0^m + \psi_{1a}(x_0^k)^Tx_0^k(x_0^l)^Tx_0^m,$$
$$B_{ijklm} = \varphi_{ab} - \psi_{ab},$$
$$C_{ijklm} = \varphi_{bn}(\nabla x_n^l)^T\,\mathrm{H}(x_n^k)(\nabla x_n^m) + \psi_{bn}\,\mathrm{Tr}(\mathrm{H}(x_n^k))(\nabla x_n^l)^T(\nabla x_n^m).$$

Theorem 4.19. *When a crelu network is initialized with the proportional symmetric initialization, the expected second derivative of the loss function with respect to the learning rate is*

$$\mathrm{E}\left[\frac{d^2}{d\lambda^2}L_N(0)\right] = \sum_{i=1}^n\sum_{j=1}^n\frac{\sqrt{d_{i-1}d_id_{j-1}d_j}}{d_0d_n}\cdot\frac{1}{s^3}\sum_{l=1}^s\sum_{k=1}^s\sum_{m=1}^s A_{ijklm}B_{ijklm}C_{ijklm}.$$

Proof. Because of Lemma 4.17, we can ignore the first sum from Lemma 4.16 and focus on the second. We do this by finding the expected value of $\alpha_{j,k,m}\chi_{i,j,k,l,m}\alpha_{i,k,l}$. This requires an entire appendix, which ends with Theorem A.14 and Corollary A.15. The latter applies directly to this calculation in a crelu network with proportional symmetric initialization. □

4.4 The Learning Rate Schedule

The loss surface of a crelu network is neither convex nor smooth in general. Nearly all of the mathematical results on gradient descent require that the function to be minimized meet one or both of these criteria. A particularly well-known result due to Nesterov establishes the convergence of gradient descent for Lipschitz-smooth functions without requiring convexity.

Theorem 4.20. *(Nesterov 98) Let $F : \mathbb{R}^d \to \mathbb{R}$ be the function to be minimized and let ∇F be the gradient of F. Suppose that there exists a finite $L > 0$ such that $\|\nabla F(x) - \nabla F(y)\| \leq L\|x - y\|$. Gradient descent with fixed step size $\lambda < 1/L$ is guaranteed to converge to a point where $\nabla F = 0$.*

Proof. Given in source [34]. □

Although this result does not technically apply to non-smooth functions, for which such a Lipschitz constant does not exist, it gives a useful intuition. Consider a loss function F with such a Lipschitz constant L. Given a starting point x_0, the loss after a step of gradient descent is $F(x_1)$ where $x_1 = x_0 - \lambda\nabla F(x_0)$. We know that $\|\nabla F(x_1) - \nabla F(x_0)\| \leq L\|x_1 - x_0\| = \lambda L\|\nabla F(x_0)\|$. If $\lambda < 1/L$, this becomes simply $\|\nabla F(x_1) - \nabla F(x_0)\| < \|\nabla F(x_0)\|$. By the triangle inequality, this implies $\|\nabla F(x_1)\| < \|\nabla F(x_0)\|$. So the gradient is guaranteed to decrease with each step.

In the non-smooth case, gradient descent with a constant learning rate is not guaranteed to converge. This can be seen even in the trivial case of minimizing $F(x) = \|x\|$, where any fixed learning rate will ultimately result in an oscillating loss. To guarantee convergence, a decreasing learning rate is required. Say that the learning rate used at step i is λ_i. We have the following classical result.

Theorem 4.21. *(Monro and Robbins 51) Let $F : \mathbb{R}^d \to \mathbb{R}$ be the function to be minimized. If the learning rate λ_i satisfies*

$$\sum_{i=1}^{\infty} \lambda_i = \infty \quad \text{and} \quad \sum_{i=1}^{\infty} \lambda_i^2 < \infty,$$

then descent is guaranteed to converge as $i \to \infty$.

Proof. Given in source [35]. □

A simple learning rate schedule such as $\lambda_i = \lambda_1/(1+ai)$, with $a > 0$, therefore guarantees convergence. However, the convergence may be too slow to matter in practice. Further, it is possible that the loss will actually increase at the outset if λ_1 is too large, only coming down when λ_i has gotten sufficiently small. Not only is this behavior undesirable in theory, but it can cause computer implementations to fail in practice. So it is important to choose both an appropriate initial learning rate and an appropriate schedule.

Here, we concern ourselves with the initial learning rate. In Section 4.3, we look at the loss as a function of the learning rate, approximate this function with a parabola, and then calculate the expected values of the coefficients of this parabola. A simple, greedy choice is to choose our initial learning rate so as to minimize the value of the loss after the first step of gradient descent. In terms of our approximation, this means choosing the value of λ which puts us at the vertex of this parabola. This motivates the following definition.

Definition 4.22. Let $F : \mathbb{R}^d \to \mathbb{R}$ be a function to be minimized with gradient descent. For $x \in \mathbb{R}^d$, let $F_x(\lambda) = F(x + \lambda \nabla F(x))$. If $F_x''(0) > 0$, the *greedy learning rate at x* is

$$\lambda_x = -\frac{F_x'(0)}{2F_x''(0)}.$$

This is a greedy choice in that it maximizes the single-step decrease in the second-order Taylor appoximation of F_x. Note that this is equivalent to using a single iteration of Newton's method to solve $F'_x(\lambda) = 0$.

In practice, an initial learning rate is always chosen before neural network training begins. So when selecting a learning rate for a crelu network with random initialization, a sensible approach is to consider the average case produced by that initialization. Of particular practical interest is the way the greedy learning rate changes depending on network architecture. Looking at the expressions in Theorem 4.15 and Theorem 4.19, we see that the expected first derivative is proportional to

$$\sum_{i=1}^{n} \sqrt{d_{i-1}d_i}$$

and the expected second derivative is approximately proportional to

$$\sum_{i=1}^{n}\sum_{j=1}^{n} \sqrt{d_{i-1}d_i d_{j-1}d_j} \prod_{k=1}^{n-1}\left(1 + \frac{2}{d_k}\right) = \left(\sum_{i=1}^{n} \sqrt{d_{i-1}d_i}\right)^2 \prod_{k=1}^{n-1}\left(1 + \frac{2}{d_k}\right).$$

This leads us to the following definition.

Definition 4.23. The *scaling factor* for a network N with dimensions $d_0, d_1, \ldots, d_n$ is

$$S_N = \sum_{i=1}^{n} \sqrt{d_{i-1}d_i} \prod_{k=1}^{n-1}\left(1 + \frac{2}{d_k}\right).$$

If we have two networks N_1 and N_2 to be trained on the same dataset, we get the following approximate relationship between their greedy learning rates λ_{N_1} and λ_{N_2}:

$$\lambda_{N_1} S_{N_1} \approx \lambda_{N_2} S_{N_2}.$$

The scaling factor is useful when training multiple architectures on the same dataset. Rather than find an effective learning rate for each architecture, one can find a good learning rate for a single architecture and use it to predict which learning rates will be similarly effective for others.

Once an initial learning rate has been established, it remains to be determined how the learning rate will change over time. Rather than choosing a schedule before training, it is common in practice to change the learning rate adaptively. For example, a simple approach is to use a fixed learning rate until the loss fails to decrease for some set number of steps, then begin a decreasing schedule that guarantees convergence. However, more sophisticated methods exist and have recently been the subject of rigorous analysis by Ward et. al. and others [36, 37].

4.5 Experiments

To show how the choice of learning rate affects training in practice, we consider two simple problems to be solved by full-connected networks.

The first is to approximate the cosine function. Specifically, we sample the x^k uniformly from $[0, 2\pi)$ and compute $y_k = \cos(x^k)$. Then we standardize the inputs by computing

$$x_0^k = \begin{bmatrix} \frac{\sqrt{3}}{\pi}(x^k - \pi) \\ 1 \end{bmatrix}.$$

For this regression application, we use squared error loss.

The second is to classify points according to a checkerboard pattern. In this case, the x^k are distributed uniformly on $[-2, 2]^2$ and assigned labels $y^k \in \{0, 1\}$ based on which 1x1 square they fall in. The x^k are normalized and given an extra dimension as before. For this classification task, we use binary cross-entropy loss.

We train crelu networks of several sizes (given in Table 4.1) on both data sets, using the proportional symmetric initialization. First, we try a wide range of candidate learning rates and train the networks for a single step of gradient descent. To get an accurate picture, we average the results across 20,000 initializations for each architecture. As expected, the loss

Name	$d_0, d_1, \ldots, d_n$
3x10	$2, 10, 10, 10, 1$
3x20	$2, 20, 20, 20, 1$
5x10	$2, 10, 10, 10, 10, 10, 1$
5x20	$2, 20, 20, 20, 20, 20, 1$

Table 4.1: We use networks with the following dimensions to show how the optimal learning rate changes with architecture.

initially decreases with the learning rate but soon begins to increase with growing speed. The results are shown in Figure 4.1.

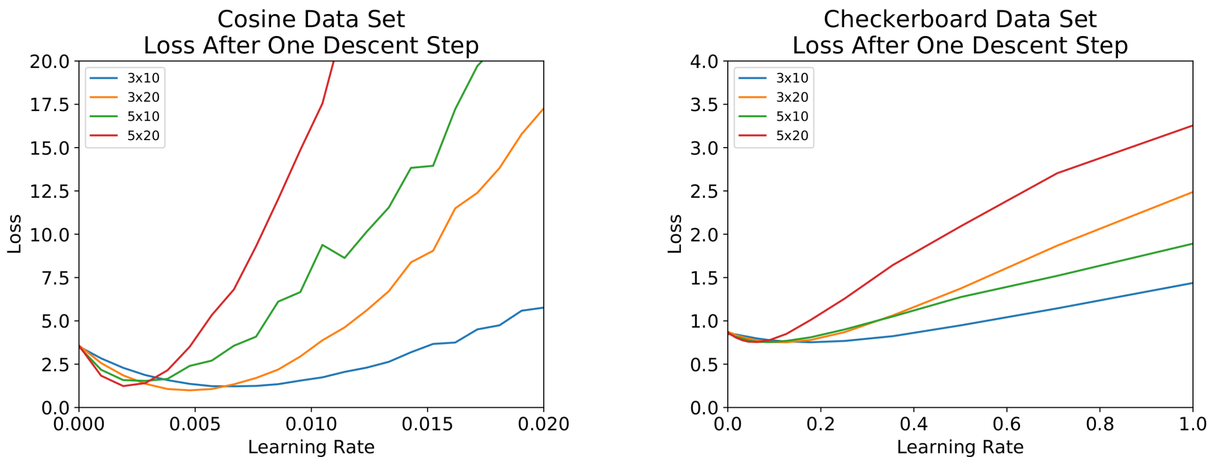

Figure 4.1: The average loss across 20,000 trials after a single step of gradient descent. Note that these values are on a log scale because they grow so quickly.

In order to get a more informative visual, we restrict ourselves to the range of learning rates that caused the loss to decrease or stay the same. Figure 4.2 shows that the loss, as a function of of the learning rate, strongly resembles a parabola over this interval for all four architectures. The parabolas have their vertices at different learning rates, indicating that the optimal value is indeed different for each architecture.

However, the fact that a learning rate yields a largest improvement on the first step does not necessarily imply that it will yield the largest improvement in the loss over many steps. For this reason, we train each model for 50 steps from 1,000 different initializations. The averages across these trials are shown in Figure 4.3. Interestingly, the results are nearly flat in ranges around the values that produced the largest drops on the first step, suggesting some

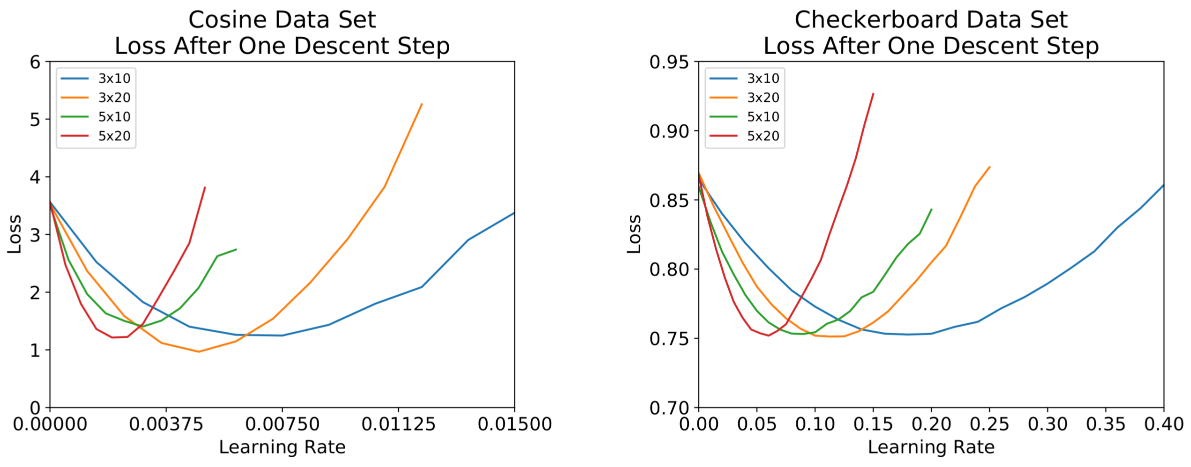

Figure 4.2: Trimming the higher values from the previous plot and using a linear axis, we see that the loss does indeed resemble a quadratic function of the learning rate.

robustness to the choice of learning rate. The loss continues to decrease slowly through this range until the models suddenly cease to learn. For all four architectures, there is a point somewhere between the one-step optimum and its double at which gradient descent starts to diverge. This suggests that the optimal learning rate for long-term training is somewhere near the one-step optimum because such a value will cause a faster decrease in loss than smaller ones while avoiding the risk of divergence that comes with larger ones.

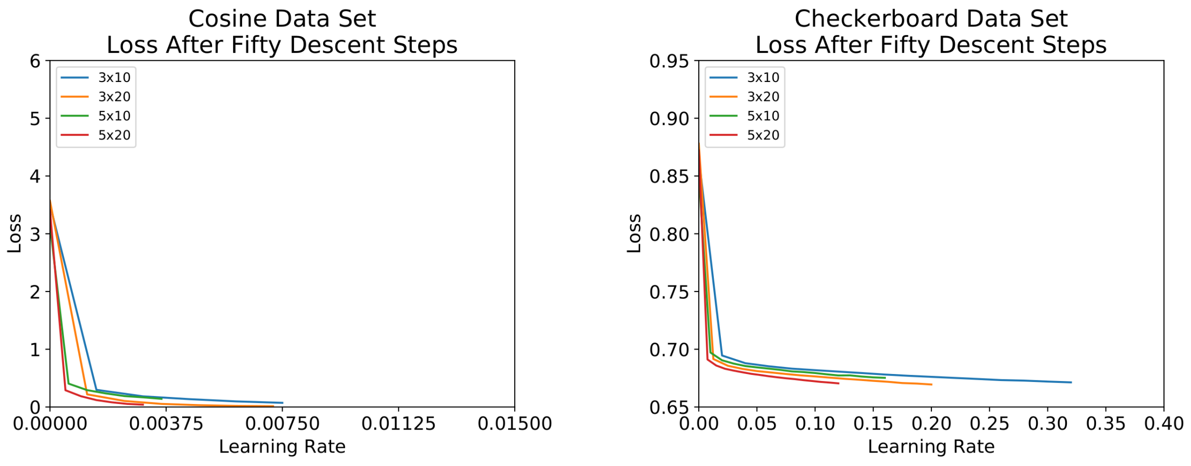

Figure 4.3: When we let the models train for fifty steps and average across 1,000 trials, we see that there is a range around the optimal learning rates in which performance is good. Above this range, gradient descent diverges.

To see whether the learning rates scale as predicted, we scale these plots by multiplying the learning rate by the scaling factor for each architecture. Sure enough, in Figure 4.4, we see that the single-step loss curves nearly line up under this scaling. The slight mismatching likely stems from the fact that the scaling factor is an approximation of the true ratio and ignores some of the smaller terms from the expressions for the expected derivatives.

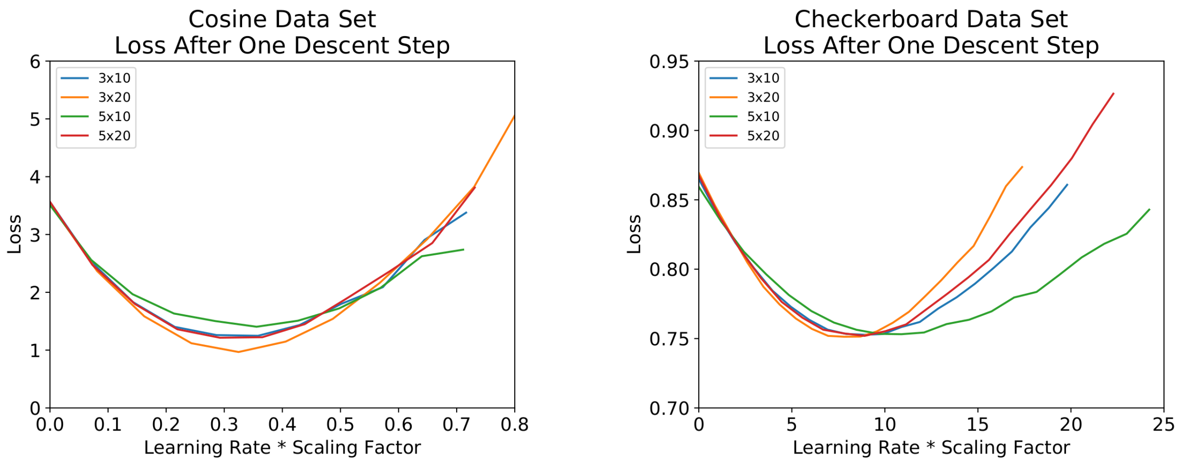

Figure 4.4: When we scale each model's curve horizontally by the scaling factor for its archtecture, they line up almost perfectly. This shows that the curvature does vary as predicted.

When the fifty-step plots are rescaled in the same manner in Figure 4.5, they too line up very well. It becomes particularly easy to see that divergence becomes a problem somewhere between the one-step optimum and its double. Recall that, on a parabola, divergence happens at precisely double the optimal learning rate.

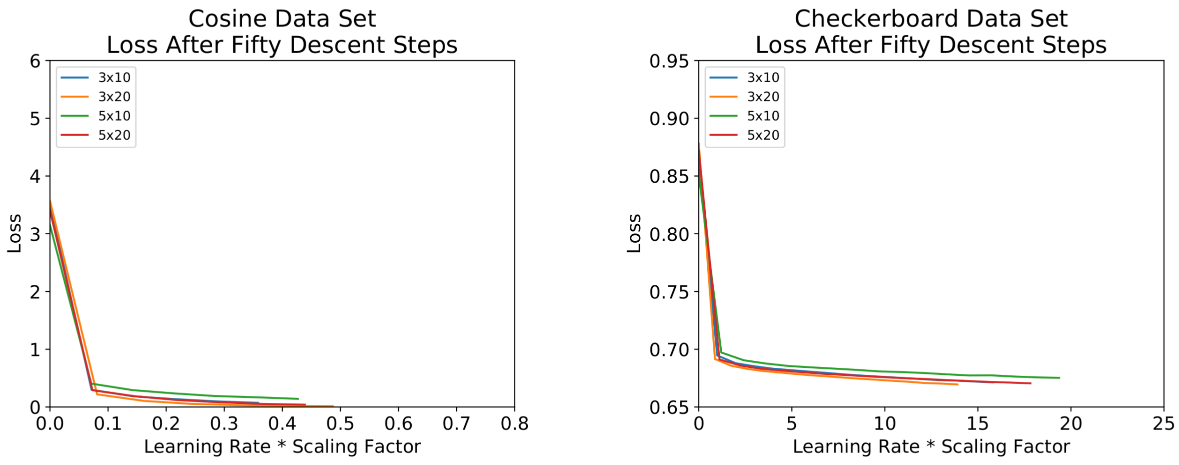

Figure 4.5: As before, the plots line up when multiplied by the scaling factor, suggesting that the optimal learning rate and the ideal range both scale in the same way.

These experiments show that it is not necessary to start the learning rate search from scratch when trying a new network architecture on a known dataset. Because the optimal learning rates scale predictably, the computational cost of this process can be avoided.

Chapter 5. Looking Forward

There are many ways in which this work can be expanded upon in the future, both for the sake of greater practical applicability and in the interest of pure mathematical understanding. We highlight just a few natural areas for exploration.

5.1 Neural Network Tools

As discussed in Chapter 2, many tools are used to aid in the training of neural networks. It would be particularly interesting to see how these theorems generalize to address:

- Stochastic Gradient Descent - While we focused here on full-batch gradient descent, neural networks are almost always trained with mini-batches in practice. Depending on how the data is distributed, the resulting noise can affect the optimal learning rate.
- Momentum - Adding momentum to gradient descent sometimes leads to significantly faster convergence that cannot be achieved by increasing the learning rate. It would be fascinating to work out a theory of optimal learning rate - momentum constant combinations, especially one that also took into account methods for adapting the learning rate.
- Dropout - While here we only worry about training performance, most practical applications are also concerned with how that performance will generalize to test data. Techniques for inserting noise into the training process to promote generalization likely have an effect on training speed and stability.
- Batch Normalization - This would pose a particular challenge, as batch normalization effectively changes parameters in between steps of gradient descent. Given its popularity and effectiveness in some cases, it certainly merits analysis.

5.2 Special Layers

Highly versatile tools are rarely the best for a specific task. As universal approximators, fully-connected networks are as versatile as tools come, so it is unsurprising that they are easily outperformed by custom layers such as convolutions for image data and recurrent units for natural language processing. Extending these results to models with such layers would be extremely useful.

5.3 Optimal Architectures

Being able to predict a learning rate given an architecture is quite helpful, but it doesn't inform the choice of architecture itself. Nor are universality theorems particularly helpful in that regard. Theorems relating the number of layers in a network, and their widths, to performance on a given data set would pair spectacularly with optimization-focused works like this one, potentially leading to an automated process of creating and training networks tailored to their data. This would decrease the need for so-called 'graduate student descent' — the process of coming up with and testing many combinations of hyperparameters manually.

5.4 Conclusion

Ali Rahimi famously compared machine learning to alchemy, encouraging machine learning researchers to bring more mathematical analysis and scientific rigor to their work. By constructing a network to have useful mathematical properties and pinning down some basic properties of its loss surface, we have made the process of training a fully-connected neural network a little less alchemical.

Appendix A. Expectations

Throughout this appendix, we use $\mathrm{E}[x]$ to denote the expected value of x, where x is a function of one or more random variables.

Lemma A.1. *Let A be an $l \times m$ matrix of independent and identically distributed elements with $\mathrm{E}[a_{ij}] = 0$, $\mathrm{E}[a_{ij}^2] = 1$, and $\mathrm{E}[a_{ij}^4] = \rho_a$.*

Let B be an $m \times n$ matrix, independent from A, of identically distributed elements with $\mathrm{E}[b_{ij}^4] = \rho_b$. Suppose that there exist constants γ_b, ν_b, τ_b, and η_b such that

$$\begin{aligned}
\gamma_b &= \mathrm{E}[b_{ij}^2 b_{kj}^2], \\
\nu_b &= \mathrm{E}[b_{ij}^2 b_{il}^2], \\
\tau_b &= \mathrm{E}[b_{ij}^2 b_{kl}^2], \\
\eta_b &= \mathrm{E}[b_{ij} b_{il} b_{kj} b_{kl}],
\end{aligned}$$

for all $1 \le i < k \le m$ and $1 \le j < l \le n$.

Let $C = AB$. Then C is an $l \times n$ matrix of identically distributed elements with $\mathrm{E}[c_{ij}] = 0$ and $\mathrm{E}[c_{ij}^2] = m\,\mathrm{E}[b_{ij}^2]$. There exist constants ν_c, γ_c, τ_c, and η_c for C analogous to those for B. These constants and $\rho_c = \mathrm{E}[c_{ij}^4]$ can be computed as follows:

$$\begin{aligned}
\rho_c &= m\rho_a\rho_b + 3m(m-1)\gamma_b, \\
\gamma_c &= m\rho_b + m(m-1)\gamma_b, \\
\nu_c &= m\rho_a\nu_b + m(m-1)\tau_b + 2m(m-1)\eta_b, \\
\tau_c &= m\nu_b + m(m-1)\tau_b, \\
\eta_c &= m\nu_b + m(m-1)\eta_b.
\end{aligned}$$

Proof. The elements of C are defined as

$$c_{ij} = \sum_{q=1}^{m} a_{iq} b_{qj}.$$

Because the elements of A and B are identically distributed (within their respective matrices — elements of A do not necessarily have the same distribution as elements of B), the elements of C are as well. Because a_{iq} and b_{qj} are independent, $\mathrm{E}[a_{iq} b_{qj}] = \mathrm{E}[a_{iq}] \, \mathrm{E}[b_{qj}] = 0$. So $\mathrm{E}[c_{ij}] = 0m = 0$.

We compute $\mathrm{E}[c_{ij}^2]$ explicitly.

$$\begin{aligned}
c_{ij}^2 &= \left(\sum_{q=1}^{m} a_{iq} b_{qj} \right)^2 \\
&= \sum_{q=1}^{m} a_{iq} b_{qj} \sum_{r=1}^{m} a_{ir} b_{rj} \\
&= \sum_{q=1}^{m} \sum_{r=1}^{m} a_{iq} a_{ir} b_{qj} b_{rj} \\
\mathrm{E}[c_{ij}^2] &= \sum_{q=1}^{m} \sum_{r=1}^{m} \mathrm{E}[a_{iq} a_{ir}] \, \mathrm{E}[b_{qj} b_{rj}].
\end{aligned}$$

In order to compute $\mathrm{E}[a_{iq} a_{ir}]$, we need to know if $q = r$. We break the sum up accordingly.

$$\begin{aligned}
\mathrm{E}[c_{ij}^2] &= \sum_{q=1}^{m} \mathrm{E}[a_{iq}^2] \, \mathrm{E}[b_{qj}^2] + \sum_{q=1}^{m} \sum_{r \neq q} \mathrm{E}[a_{iq}] \, \mathrm{E}[a_{ir}] \, \mathrm{E}[b_{qj} b_{rj}] \\
&= \sum_{q=1}^{m} \mathrm{E}[b_{qj}^2] \\
&= m \, \mathrm{E}[b_{qj}^2].
\end{aligned}$$

Now we derive ρ_c.

$$\begin{aligned} c_{ij}^4 &= \left(\sum_{q=1}^{m} a_{iq}b_{qj}\right)^4 \\ &= \sum_{q=1}^{m} a_{iq}b_{qj} \sum_{r=1}^{m} a_{ir}b_{rj} \sum_{s=1}^{m} a_{is}b_{sj} \sum_{t=1}^{m} a_{it}b_{tj} \\ &= \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} a_{iq}a_{ir}a_{is}a_{it}b_{qj}b_{rj}b_{sj}b_{tj} \\ \rho_c &= \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} \mathrm{E}[a_{iq}a_{ir}a_{is}a_{it}]\,\mathrm{E}[b_{qj}b_{rj}b_{sj}b_{tj}]. \end{aligned}$$

Most of the terms of the sum vanish because $\mathrm{E}[a_{iq}a_{ir}a_{is}a_{it}] = 0$ unless one of the following is true: $q = r = s = t$, $q = r \neq s = t$, $q = s \neq r = t$, or $q = t \neq r = s$. Because the indices are interchangeable, the last three cases produce the same sum.

$$\begin{aligned} \rho_c &= \sum_{q=1}^{m} \mathrm{E}[a_{iq}^4]\,\mathrm{E}[b_{qj}^4] + 3\sum_{q=1}^{m}\sum_{r\neq q} \mathrm{E}[a_{iq}^2]\,\mathrm{E}[a_{ir}^2]\,\mathrm{E}[b_{qj}^2 b_{rj}^2] \\ &= m\rho_a\rho_b + 3m(m-1)\gamma_b. \end{aligned}$$

Next, we derive γ_c. Assume $i \neq k$.

$$\begin{aligned} c_{ij}^2 c_{kj}^2 &= \left(\sum_{q=1}^{m} a_{iq}b_{qj}\right)^2 \left(\sum_{s=1}^{m} a_{ks}b_{sj}\right)^2 \\ &= \sum_{q=1}^{m} a_{iq}b_{qj} \sum_{r=1}^{m} a_{ir}b_{rj} \sum_{s=1}^{m} a_{ks}b_{sj} \sum_{t=1}^{m} a_{kt}b_{tj} \\ &= \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} a_{iq}a_{ir}a_{ks}a_{kt}b_{qj}b_{rj}b_{sj}b_{tj} \\ \gamma_c &= \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} \mathrm{E}[a_{iq}a_{ir}a_{ks}a_{kt}]\,\mathrm{E}[b_{qj}b_{rj}b_{sj}b_{tj}]. \end{aligned}$$

In this case, $\mathrm{E}[a_{iq}a_{ir}a_{ks}a_{kt}]$ is only nonzero if $q = r = s = t$ or $q = r \neq s = t$.

$$\begin{aligned}\gamma_c =& \sum_{q=1}^{m} \mathrm{E}[a_{iq}^2]\,\mathrm{E}[a_{kq}^2]\,\mathrm{E}[b_{qj}^4] + \sum_{q=1}^{m}\sum_{s\neq q} \mathrm{E}[a_{iq}^2]\,\mathrm{E}[a_{ks}^2]\,\mathrm{E}[b_{qj}^2 b_{sj}^2] \\ =& m\rho_b + m(m-1)\gamma_b.\end{aligned}$$

The derivation for ν_c is similar. Assume $j \neq l$.

$$\begin{aligned}c_{ij}^2 c_{il}^2 =& \left(\sum_{q=1}^{m} a_{iq}b_{qj}\right)^2 \left(\sum_{s=1}^{m} a_{is}b_{sl}\right)^2 \\ =& \sum_{q=1}^{m} a_{iq}b_{qj} \sum_{r=1}^{m} a_{ir}b_{rj} \sum_{s=1}^{m} a_{is}b_{sl} \sum_{t=1}^{m} a_{it}b_{tl} \\ =& \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} a_{iq}a_{ir}a_{is}a_{it}b_{qj}b_{rj}b_{sl}b_{tl} \\ \nu_c =& \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} \mathrm{E}[a_{iq}a_{ir}a_{is}a_{it}]\,\mathrm{E}[b_{qj}b_{rj}b_{sl}b_{tl}].\end{aligned}$$

Now $\mathrm{E}[a_{iq}a_{ir}a_{is}a_{it}]$ is only nonzero if $q = r = s = t$, $q = r \neq s = t$, $q = s \neq r = t$, or $q = t \neq r = s$. Only the last two of those cases are equivalent.

$$\begin{aligned}\nu_c =& \sum_{q=1}^{m} \mathrm{E}[a_{iq}^4]\,\mathrm{E}[b_{qj}^2 b_{ql}^2] + \sum_{q=1}^{m}\sum_{s\neq q} \mathrm{E}[a_{iq}^2]\,\mathrm{E}[a_{is}^2]\,\mathrm{E}[b_{qj}^2 b_{sl}^2] + 2\sum_{q=1}^{m}\sum_{r\neq q} \mathrm{E}[a_{iq}^2]\,\mathrm{E}[a_{ir}^2]\,\mathrm{E}[b_{qj}b_{rj}b_{ql}b_{rl}] \\ =& m\rho_a\nu_b + m(m-1)\tau_b + 2m(m-1)\eta_b.\end{aligned}$$

Next, we derive τ_c.

$$\begin{aligned}
c_{ij}^2 c_{kl}^2 =& \left(\sum_{q=1}^{m} a_{iq} b_{qj}\right)^2 \left(\sum_{s=1}^{m} a_{ks} b_{sl}\right)^2 \\
=& \sum_{q=1}^{m} a_{iq} b_{qj} \sum_{r=1}^{m} a_{ir} b_{rj} \sum_{s=1}^{m} a_{ks} b_{sl} \sum_{t=1}^{m} a_{kt} b_{tl} \\
=& \sum_{q=1}^{m} \sum_{r=1}^{m} \sum_{s=1}^{m} \sum_{t=1}^{m} a_{iq} a_{ir} a_{ks} a_{kt} b_{qj} b_{rj} b_{sl} b_{tl} \\
\tau_c =& \sum_{q=1}^{m} \sum_{r=1}^{m} \sum_{s=1}^{m} \sum_{t=1}^{m} \mathrm{E}[a_{iq} a_{ir} a_{ks} a_{kt}] \, \mathrm{E}[b_{qj} b_{rj} b_{sl} b_{tl}].
\end{aligned}$$

As with γ_c, $\mathrm{E}[a_{iq} a_{ir} a_{ks} a_{kt}]$ is only nonzero if $q = r = s = t$ or $q = r \neq s = t$.

$$\begin{aligned}
\tau_c =& \sum_{q=1}^{m} \mathrm{E}[a_{iq}^2] \, \mathrm{E}[a_{kq}^2] \, \mathrm{E}[b_{qj}^2 b_{ql}^2] + \sum_{q=1}^{m} \sum_{s \neq q} \mathrm{E}[a_{iq}^2] \, \mathrm{E}[a_{ks}^2] \, \mathrm{E}[b_{qj}^2 b_{sl}^2] \\
=& m\nu_b + m(m-1)\tau_b.
\end{aligned}$$

Finally, we derive η_c.

$$\begin{aligned}
c_{ij} c_{il} c_{kj} c_{kl} =& \left(\sum_{q=1}^{m} a_{iq} b_{qj}\right) \left(\sum_{r=1}^{m} a_{ir} b_{rl}\right) \left(\sum_{s=1}^{m} a_{ks} b_{sj}\right) \left(\sum_{t=1}^{m} a_{kt} b_{tl}\right) \\
=& \sum_{q=1}^{m} \sum_{r=1}^{m} \sum_{s=1}^{m} \sum_{t=1}^{m} a_{iq} a_{ir} a_{ks} a_{kt} b_{qj} b_{rl} b_{sj} b_{tl} \\
\eta_c =& \sum_{q=1}^{m} \sum_{r=1}^{m} \sum_{s=1}^{m} \sum_{t=1}^{m} \mathrm{E}[a_{iq} a_{ir} a_{ks} a_{kt}] \, \mathrm{E}[b_{qj} b_{rl} b_{sj} b_{tl}].
\end{aligned}$$

Once again, $\mathrm{E}[a_{iq} a_{ir} a_{ks} a_{kt}]$ is only nonzero if $q = r = s = t$ or $q = r \neq s = t$.

$$\begin{aligned}
\eta_c =& \sum_{q=1}^{m} \mathrm{E}[a_{iq}^2] \, \mathrm{E}[a_{kq}^2] \, \mathrm{E}[b_{qj}^2 b_{ql}^2] + \sum_{q=1}^{m} \sum_{s \neq q} \mathrm{E}[a_{iq}^2] \, \mathrm{E}[a_{ks}^2] \, \mathrm{E}[b_{qj} b_{ql} b_{sj} b_{sl}] \\
=& m\nu_b + m(m-1)\eta_b.
\end{aligned}$$

Thus all five equations are proven. □

Corollary A.2. *Let A, B, and C be as in Lemma A.1. If $g \neq i$, then* $\mathrm{E}[c_{de}c_{df}c_{gh}c_{ij}] = 0$.

Proof. Suppose $g \neq i$.

$$\begin{aligned} c_{de}c_{df}c_{gh}c_{ij} &= \left(\sum_{q=1}^{m} a_{dq}b_{qe}\right)\left(\sum_{r=1}^{m} a_{dr}b_{rf}\right)\left(\sum_{s=1}^{m} a_{gs}b_{sh}\right)\left(\sum_{t=1}^{m} a_{it}b_{tj}\right) \\ &= \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} a_{dq}a_{dr}a_{gs}a_{it}b_{qe}b_{rf}b_{sh}b_{tj} \\ \mathrm{E}[c_{de}c_{df}c_{gh}c_{ij}] &= \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} \mathrm{E}[a_{dq}a_{dr}a_{gs}a_{it}]\,\mathrm{E}[b_{qe}b_{rf}b_{sh}b_{tj}]. \end{aligned}$$

Because $g \neq i$, g and i cannot both be equal to d. Without loss of generality, say $i \neq d$. This gives $\mathrm{E}[a_{dq}a_{dr}a_{gs}a_{it}] = \mathrm{E}[a_{dq}a_{dr}a_{gs}]\,\mathrm{E}[a_{it}] = 0$. □

Corollary A.3. *Let A, B, and C be as in Lemma A.1. Suppose that* $\mathrm{E}[b_{de}b_{if}b_{gh}b_{kj}] = 0$ *if one of e, f, h, and j is distinct from the other three. Then C has this property as well.*

Proof. Suppose that one of e, f, h, and j is distinct from the other three.

$$\begin{aligned} c_{de}c_{if}c_{gh}c_{kj} &= \left(\sum_{q=1}^{m} a_{dq}b_{qe}\right)\left(\sum_{r=1}^{m} a_{ir}b_{rf}\right)\left(\sum_{s=1}^{m} a_{gs}b_{sh}\right)\left(\sum_{t=1}^{m} a_{kt}b_{tj}\right) \\ &= \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} a_{dq}a_{ir}a_{gs}a_{kt}b_{qe}b_{rf}b_{sh}b_{tj} \\ \mathrm{E}[c_{de}c_{df}c_{gh}c_{gj}] &= \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} \mathrm{E}[a_{dq}a_{ir}a_{gs}a_{kt}]\,\mathrm{E}[b_{qe}b_{rf}b_{sh}b_{tj}]. \end{aligned}$$

By the assumption, $\mathrm{E}[b_{qe}b_{rf}b_{sh}b_{tj}] = 0$. So we see that $\mathrm{E}[c_{de}c_{df}c_{gh}c_{gj}] = 0$. □

Corollary A.4. *Let A, B, and C be as in Lemma A.1. Suppose that if $i \neq j$, then* $\mathrm{E}[b_{ej}b_{gf}b_{hf}b_{ki}] = 0$. *Then C has this property.*

Proof. Suppose that $i \neq j$.

$$\begin{aligned}
c_{ej}c_{gf}c_{hf}c_{ki} &= \left(\sum_{q=1}^{m} a_{eq}b_{qj}\right)\left(\sum_{r=1}^{m} a_{gr}b_{rf}\right)\left(\sum_{s=1}^{m} a_{hs}b_{sf}\right)\left(\sum_{t=1}^{m} a_{kt}b_{ti}\right) \\
&= \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} a_{eq}a_{gr}a_{hs}a_{kt}b_{qj}b_{rf}b_{sf}b_{ti} \\
\mathrm{E}[c_{ej}c_{gf}c_{hf}c_{ki}] &= \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} \mathrm{E}[a_{eq}a_{gr}a_{hs}a_{kt}]\,\mathrm{E}[b_{qj}b_{rf}b_{sf}b_{ti}].
\end{aligned}$$

By the assumption, $\mathrm{E}[b_{qj}b_{rf}b_{sf}b_{ti}] = 0$. □

Corollary A.5. *Let A, B, and C be as in Lemma A.1. Then $\mathrm{E}[c_{ei}c_{gf}c_{hf}c_{ki}] = 0$ if one of e, g, h, and k is distinct from the other three.*

Proof. Suppose that one of e, g, h, and k is distinct from the other three.

$$\begin{aligned}
c_{ei}c_{gf}c_{hf}c_{ki} &= \left(\sum_{q=1}^{m} a_{eq}b_{qi}\right)\left(\sum_{r=1}^{m} a_{gr}b_{rf}\right)\left(\sum_{s=1}^{m} a_{hs}b_{sf}\right)\left(\sum_{t=1}^{m} a_{kt}b_{ti}\right) \\
&= \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} a_{eq}a_{gr}a_{hs}a_{kt}b_{qi}b_{rf}b_{sf}b_{ti} \\
\mathrm{E}[c_{ei}c_{gf}c_{hf}c_{ki}] &= \sum_{q=1}^{m}\sum_{r=1}^{m}\sum_{s=1}^{m}\sum_{t=1}^{m} \mathrm{E}[a_{eq}a_{gr}a_{hs}a_{kt}]\,\mathrm{E}[b_{qi}b_{rf}b_{sf}b_{ti}].
\end{aligned}$$

Without loss of generality, say e is not equal to any of g, h, and k. This gives $\mathrm{E}[a_{eq}a_{gr}a_{hs}a_{kt}] = \mathrm{E}[a_{eq}]\,\mathrm{E}[a_{gr}a_{hs}a_{kt}] = 0$. □

Lemma A.6. *Let $m_0, \ldots, m_n \in \mathbb{N}$. For $1 \leq q \leq n$, let A_q be an $m_q \times m_{q-1}$ matrix. Let all of the elements of all of the A_q be independent and identically distributed. Denote by a_{qij} the element in the ith row and jth column of A_q. Let $\mathrm{E}[a_{qij}] = 0$ and $\mathrm{E}[a_{qij}^2] = 1$. Define $\rho_1 = \mathrm{E}[a_{qij}^4]$.*

Let $B_q = A_q \cdots A_1$. Then B_q is an $m_q \times m_0$ matrix of identically distributed elements with $\mathrm{E}[b_{qij}] = 0$ and $\mathrm{E}[b_{qij}^2] = m_1 \cdots m_{q-1}$. For all $1 \leq q \leq n$, there exist constants ρ_q, γ_q,

ν_q, τ_q, and η_q such that

$$\begin{aligned}
\rho_q &= \mathrm{E}[b_{qij}^4], \\
\gamma_q &= \mathrm{E}[b_{qij}^2 b_{qkj}^2], \\
\nu_q &= \mathrm{E}[b_{qij}^2 b_{qil}^2], \\
\tau_q &= \mathrm{E}[b_{qij}^2 b_{qkl}^2], \\
\eta_q &= \mathrm{E}[b_{qij} b_{qil} b_{qkj} b_{qkl}],
\end{aligned}$$

for all $1 \le i < k \le m_q$ and $1 \le j < l \le m_0$. These constants obey the relations

$$\begin{aligned}
\rho_{q+1} &= m_q \rho_1 \rho_q + 3 m_q (m_q - 1) \gamma_q, \\
\gamma_{q+1} &= m_q \rho_q + m_q (m_q - 1) \gamma_q, \\
\nu_{q+1} &= m_q \rho_1 \nu_q + m_q (m_q - 1) \tau_q + 2 m_q (m_q - 1) \eta_q, \\
\tau_{q+1} &= m_q \nu_q + m_q (m_q - 1) \tau_q, \\
\eta_{q+1} &= m_q \nu_q + m_q (m_q - 1) \eta_q,
\end{aligned}$$

with $\gamma_1 = \nu_1 = \tau_1 = 1$ and $\eta_1 = 0$.

Proof. Everything here follows by repeatedly applying Lemma A.1 to $B_{q+1} = A_{q+1} B_q$. □

Corollary A.7. *Let B_q be as in Lemma A.6. Then B_q has all of the properties ascribed to C in Corollary A.2, Corollary A.3, Corollary A.4, and Corollary A.5.*

Proof. In each case, the corollary is applied by induction to A_q and B_{q-1} to show that B_q has the desired property. □

Lemma A.8. *If $\rho_1 = 3$ in Lemma A.6, then the constants in question can be calculated directly as follows:*

$$\begin{aligned}
\rho_q &= 3\prod_{r=1}^{q-1} m_r(m_r+2),\\
\gamma_q &= \prod_{r=1}^{q-1} m_r(m_r+2),\\
\nu_q &= \prod_{r=1}^{q-1} m_r(m_r+2),\\
\tau_q &= \frac{1}{3}\prod_{r=1}^{q-1} m_r(m_r+2) + \frac{2}{3}\prod_{r=1}^{q-1} m_r(m_r-1),\\
\eta_q &= \frac{1}{3}\prod_{r=1}^{q-1} m_r(m_r+2) - \frac{1}{3}\prod_{r=1}^{q-1} m_r(m_r-1).
\end{aligned}$$

In particular, these formulas hold when the a_{qij} are Gaussian.

Proof. If $\rho_1 = 1$, then the recurrence relations for ρ_{q+1} and γ_{q+1} become

$$\begin{aligned}
\rho_{q+1} &= 3m_q\rho_q + 3m_q(m_q-1)\gamma_q,\\
\gamma_{q+1} &= m_q\rho_q + m_q(m_q-1)\gamma_q.
\end{aligned}$$

Since we have $\gamma_1 = 1$, this means that $\rho_q = 3\gamma_q$ for all $1 \le q \le n$. This allows us to simplify the expression for ρ_{q+1} to

$$\begin{aligned}
\rho_{q+1} &= 3m_q\rho_q + m_q(m_q-1)\rho_q,\\
&= m_q(m_q+2)\rho_q.
\end{aligned}$$

By applying this simple rule repeatedly, we get

$$\rho_q = 3\prod_{r=1}^{q-1} m_q(m_q+2),$$
$$\gamma_q = \prod_{r=1}^{q-1} m_q(m_q+2).$$

Similarly, if $\rho_1 = 3$, the recurrence relations for ν_{q+1}, τ_{q+1}, and η_{q+1} become

$$\begin{aligned}
\nu_{q+1} &= 3m_q\nu_q + m_q(m_q-1)\tau_q + 2m_q(m_q-1)\eta_q,\\
\tau_{q+1} &= m_q\nu_q + m_q(m_q-1)\tau_q,\\
\eta_{q+1} &= m_q\nu_q + m_q(m_q-1)\eta_q.
\end{aligned}$$

Since we have $\nu_1 = \tau_1 = 1$ and $\eta_1 = 0$, this means that $\nu_q = \tau_q + 2\eta_q$ for all $1 \le q \le n$. This allows us to simplify the expression for ν_{q+1} to

$$\begin{aligned}
\nu_{q+1} &= 3m_q\nu_q + m_q(m_q-1)\tau_q + 2m_q(m_q-1)\eta_q\\
&= 3m_q\nu_q + m_q(m_q-1)(\tau_q + 2\eta_q)\\
&= 3m_q\nu_q + m_q(m_q-1)\nu_q\\
&= m_q(m_q+2)\nu_q.
\end{aligned}$$

By applying this rule repeatedly, we get

$$\nu_q = \prod_{r=1}^{q-1} m_r(m_r+2).$$

Interestingly, this means that $\nu_q = \gamma_q$ for all $1 \le q \le n$.

Finally, taking the difference of τ_{q+1} and η_{q+1} gives a useful relation:

$$\tau_{q+1} - \eta_{q+1} = m_q(m_q-1)(\tau_q - \eta_q).$$

It follows that

$$\tau_q - \eta_q = \prod_{r=1}^{q-1} m_r(m_r - 1).$$

This allows us to obtain the following formulas for τ_{q+1} and η_{q+1}:

$$\begin{aligned}
\tau_q &= \frac{1}{3}(\tau_q + 2\eta_q) + \frac{2}{3}(\tau_q - \eta_q) \\
&= \frac{1}{3}\prod_{r=1}^{q-1} m_r(m_r + 2) + \frac{2}{3}\prod_{r=1}^{q-1} m_r(m_r - 1), \\
\eta_q &= \frac{1}{3}(\tau_q + 2\eta_q) - \frac{1}{3}(\tau_q - \eta_q) \\
&= \frac{1}{3}\prod_{r=1}^{q-1} m_r(m_r + 2) - \frac{1}{3}\prod_{r=1}^{q-1} m_r(m_r - 1).
\end{aligned}$$

This completes the proof. □

Lemma A.9. *Let $m_0, \ldots, m_n \in \mathbb{N}$. For $1 \leq q \leq n$, let A_q be an $m_q \times m_{q-1}$ matrix. Let all of the elements of all of the A_q be independent and identically distributed. Denote by a_{qij} the element in the ith row and jth column of A_q. Let $\mathrm{E}[a_{qij}] = 0$, $\mathrm{E}[a_{qij}^2] = 1$, and $\mathrm{E}[a_{qij}^4] = 3$. Let $B_{q,r} = A_q \cdots A_r$ for $q \geq r$. Let $1 \leq i \leq n$. Let $x, y \in \mathbb{R}^{m_0}$. Let $w, v \in \mathbb{R}^{m_n}$. Thus*

$$\mathrm{E}\left[x^T(B_{i-1,1})^T B_{i-1,1} y w^T B_{n,i+1}(B_{n,i+1})^T v\right] = m_1 m_2 \cdots m_{n-1} x^T y w^T v.$$

Proof. First, note that $x^T(B_{i-1,1})^T B_{i-1,1} y$ and $w^T B_{n,i+1}(B_{n,i+1})^T v$ are independent. This allows us to factor the expectation as

$$\mathrm{E}\left[x^T(B_{i-1,1})^T B_{i-1,1} y w^T B_{n,i+1}(B_{n,i+1})^T v\right] = \mathrm{E}\left[x^T(B_{i-1,1})^T B_{i-1,1} y\right] \mathrm{E}\left[w^T B_{n,i+1}(B_{n,i+1})^T v\right],$$

and work out the two simpler expectations separately. Further, because the two have fundamentally the same structure, we only have to work out the details for the first.

To keep the subscripts manageable, define the shorthands $B = B_{i-1,1}$, $n = m_0$, and $m = m_{i-1}$. Note that x and y, are $n \times 1$ while B is $m \times n$.

First, we derive an explicit formula for $x^T B^T By$.

$$\begin{aligned} x^T B^T By &= \sum_{d=1}^{m} (Bx)_d (By)_d \\ &= \sum_{d=1}^{m} \sum_{e=1}^{n} b_{de} x_e \sum_{f=1}^{n} b_{df} y_f \\ &= \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f=1}^{n} b_{de} b_{df} x_e y_f. \end{aligned}$$

Then we take the expected value.

$$\mathrm{E}[x^T B^T By] = \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f=1}^{n} \mathrm{E}[b_{de} b_{df}] x_e y_f.$$

Because $\mathrm{E}[b_{de} b_{df}] = 0$ when $e \neq f$, this simplifies readily.

$$= \sum_{d=1}^{m} \sum_{e=1}^{n} \mathrm{E}[b_{de}^2] x_e y_e.$$

By Lemma A.6, $\mathrm{E}[b_{de}^2] = m_1 m_2 \cdots m_{i-2}$. Recall that $m = m_{i-1}$.

$$= m_1 m_2 \cdots m_{i-1} x^T y.$$

In the language of the lemma statement, this tells us that

$$\mathrm{E}\left[x^T (B_{i-1,1})^T B_{i-1,1} y\right] = m_1 m_2 \cdots m_{i-1} x^T y.$$

Applying the same logic to the other expectation in question gives

$$\mathrm{E}\left[w^T B_{n,i+1} (B_{n,i+1})^T v\right] = m_i m_{i+2} \cdots m_{n-1} w^T v.$$

The desired result is obtained by multiplying the two.

$$\mathrm{E}\left[x^T(B_{i-1,1})^T B_{i-1,1} y w^T B_{n,i+1}(B_{n,i+1})^T v\right] = m_1 m_2 \cdots m_{n-1} x^T y w^T v.$$

It is noteworthy that this expression does not depend on i. $\square$

Corollary A.10. *Let all be as in Theorem A.9 except with each A_q scaled by a factor of $\sigma_q > 0$, causing $\mathrm{E}[a_{qij}^2] = \sigma_q^2$. Then*

$$\mathrm{E}\left[x^T(B_{i-1,1})^T B_{i-1,1} y w^T B_{n,i+1}(B_{n,i+1})^T v\right] = \frac{\sigma_1^2\sigma_2^2\cdots\sigma_n^2}{\sigma_i^2} m_1 m_2 \cdots m_{n-1} x^T y w^T v.$$

If $\sigma_i = (m_{i-1}m_i)^{-1/4}$, then this simplifies to

$$\frac{\sqrt{m_{i-1}m_i}}{\sqrt{m_0 m_n}} x^T y w^T v.$$

Proof. Each of the A_q occurs twice in the product except for A_i, which does not occur. Hence the scaling by

$$\frac{\sigma_1^2\sigma_2^2\cdots\sigma_n^2}{\sigma_i^2}.$$

When $\sigma_i = (m_{i-1}m_i)^{-1/4}$, this becomes

$$\frac{\sqrt{m_{i-1}m_i}}{\sqrt{m_0}m_1 m_2\cdots m_{n-1}\sqrt{m_n}} m_1 m_2 \cdots m_{n-1} x^T y w^T v.$$

Cancellation gives the simple form above. $\square$

Lemma A.11. *Let $m_0, \ldots, m_n \in \mathbb{N}$. For $1 \le q \le n$, let A_q be an $m_q \times m_{q-1}$ matrix. Let all of the elements of all of the A_q be independent and identically distributed. Denote by a_{qij} the element in the ith row and jth column of A_q. Let $\mathrm{E}[a_{qij}] = 0$, $\mathrm{E}[a_{qij}^2] = 1$, and $\mathrm{E}[a_{qij}^4] = 3$.*

Let $B_{q,r} = A_q \cdots A_r$ for $q \geq r$. Let $1 \leq i \leq j \leq n$. Let $x, y, z \in \mathbb{R}^{m_0}$. Then

$$\begin{aligned}
&\mathrm{E}\left[\left(x^T B_{i-1,1}^T B_{i-1,1} y\right)\left(x^T B_{j-1,1}^T B_{j-1,1} z\right) \mid A_i, \ldots, A_{j-1}\right] \\
&\quad = \frac{\|B_{j-1,i}\|_F^2}{m_{i-1}} \left(\frac{2x^T y x^T z + x^T x y^T z}{3} \prod_{r=1}^{i-1} m_r(m_r+2) + \frac{x^T y x^T z - x^T x y^T z}{3} \prod_{r=1}^{i-1} m_r(m_r-1) \right).
\end{aligned}$$

Here $\|\cdot\|_F$ denotes the Frobenius norm.

Proof. To keep the notation manageable, let $B = B_{i-1,1}$, $C = B_{j-1,i}^T B_{j-1,i}$, $n = m_0$, and $m = m_{i-1}$. With this new notation, we derive

$$\mathrm{E}\left[\left(x^T B^T B y\right)\left(x^T B^T C B z\right) \mid C\right].$$

Note that x, y, and z are $n \times 1$, B is $m \times n$, and C is $m \times m$.

First, we derive an explicit formula for $x^T B^T B y$.

$$\begin{aligned}
x^T B^T B y &= \sum_{d=1}^{m} (Bx)_d (By)_d \\
&= \sum_{d=1}^{m} \sum_{e=1}^{n} b_{de} x_e \sum_{f=1}^{n} b_{df} y_f \\
&= \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f=1}^{n} b_{de} b_{df} x_e y_f.
\end{aligned}$$

We do the same for $x^T B^T C B z$.

$$\begin{aligned}
x^T B^T C B z &= \sum_{g=1}^{m} (Bx)_g (CBz)_g \\
&= \sum_{g=1}^{m} \sum_{h=1}^{n} b_{gh} x_h \sum_{i=1}^{m} c_{gi} (Bz)_i \\
&= \sum_{g=1}^{m} \sum_{h=1}^{n} b_{gh} x_h \sum_{i=1}^{m} c_{gi} \sum_{j=1}^{n} b_{ij} z_j \\
&= \sum_{g=1}^{m} \sum_{h=1}^{n} \sum_{i=1}^{m} \sum_{j=1}^{n} b_{gh} b_{ij} c_{gi} x_h z_j .
\end{aligned}$$

We combine the two to obtain a formula for $x^T B^T B y x^T B^T C B z$.

$$\begin{aligned}
x^T B^T B y x^T B^T C B z &= \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f=1}^{n} b_{de} b_{df} x_e y_f \sum_{g=1}^{m} \sum_{h=1}^{n} \sum_{i=1}^{m} \sum_{j=1}^{n} b_{gh} b_{ij} c_{gi} x_h z_j \\
&= \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f=1}^{n} \sum_{g=1}^{m} \sum_{h=1}^{n} \sum_{i=1}^{m} \sum_{j=1}^{n} b_{de} b_{df} b_{gh} b_{ij} c_{gi} x_e x_h y_f z_j .
\end{aligned}$$

By Corollary A.7, $\mathrm{E}[b_{de} b_{df} b_{gh} b_{ij}] = 0$ if $g \neq i$. This lets us simplify a little bit.

$$\begin{aligned}
\mathrm{E}[x^T B^T B y x^T B^T C B z \mid C] &= \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f=1}^{n} \sum_{g=1}^{m} \sum_{h=1}^{n} \sum_{i=1}^{m} \sum_{j=1}^{n} \mathrm{E}[b_{de} b_{df} b_{gh} b_{ij}] c_{gi} x_e x_h y_f z_j \\
&= \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f=1}^{n} \sum_{g=1}^{m} \sum_{h=1}^{n} \sum_{j=1}^{n} \mathrm{E}[b_{de} b_{df} b_{gh} b_{gj}] c_{gg} x_e x_h y_f z_j .
\end{aligned}$$

By Corollary A.7, $\mathrm{E}[b_{de} b_{df} b_{gh} b_{gj}]$ is nonzero in four cases: $e = h \neq f = j$, $e = j \neq f = h$, $e = f \neq h = j$, and $e = f = h = j$. We break the sum up accordingly.

$$\begin{aligned}
&\mathrm{E}[x^T B^T B y x^T B^T C B z \mid C] \\
&\quad = \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f \neq e} \sum_{g=1}^{m} \mathrm{E}[b_{de} b_{df} b_{ge} b_{gf}] c_{gg} x_e x_e y_f z_f + \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f \neq e} \sum_{g=1}^{m} \mathrm{E}[b_{de} b_{df} b_{gf} b_{ge}] c_{gg} x_e x_f y_f z_e \\
&\qquad + \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{g=1}^{m} \sum_{h \neq e} \mathrm{E}[b_{de} b_{de} b_{gh} b_{gh}] c_{gg} x_e x_h y_e z_h + \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{g=1}^{m} \mathrm{E}[b_{de}^2 b_{ge}^2] c_{gg} x_e x_e y_e z_e .
\end{aligned}$$

We break the sum apart again to handle the cases $d = g$ and $d \neq g$.

$$\begin{aligned}
&\mathrm{E}[x^T B^T B y x^T B^T C B z \mid C] \\
&\quad = \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f \neq e} \mathrm{E}[b_{de}^2 b_{df}^2] c_{dd} x_e x_e y_f z_f + \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f \neq e} \mathrm{E}[b_{de}^2 b_{df}^2] c_{dd} x_e x_f y_f z_e \\
&\qquad + \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{h \neq e} \mathrm{E}[b_{de}^2 b_{dh}^2] c_{dd} x_e x_h y_e z_h + \sum_{d=1}^{m} \sum_{e=1}^{n} \mathrm{E}[b_{de}^4] c_{dd} x_e x_e y_e z_e \\
&\qquad + \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f \neq e} \sum_{g \neq d} \mathrm{E}[b_{de} b_{df} b_{ge} b_{gf}] c_{gg} x_e x_e y_f z_f + \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{f \neq e} \sum_{g \neq d} \mathrm{E}[b_{de} b_{df} b_{gf} b_{ge}] c_{gg} x_e x_f y_f z_e \\
&\qquad + \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{g \neq d} \sum_{h \neq e} \mathrm{E}[b_{de}^2 b_{gh}^2] c_{gg} x_e x_h y_e z_h + \sum_{d=1}^{m} \sum_{e=1}^{n} \sum_{g \neq d} \mathrm{E}[b_{de}^2 b_{ge}^2] c_{gg} x_e x_e y_e z_e.
\end{aligned}$$

We substitute in the constants from Lemma A.1 and sum over d and g to separate $\mathrm{Tr}(C)$ from each sum.

$$\begin{aligned}
&\mathrm{E}[x^T B^T B y x^T B^T C B z \mid C] \\
&\quad = \nu_b \operatorname{Tr}(C) \sum_{e=1}^{n} x_e x_e \sum_{f \neq e} y_f z_f + \nu_b \operatorname{Tr}(C) \sum_{e=1}^{n} x_e z_e \sum_{f \neq e} x_f y_f \\
&\qquad + \nu_b \operatorname{Tr}(C) \sum_{e=1}^{n} x_e y_e \sum_{h \neq e} x_h z_h + \rho_b \operatorname{Tr}(C) \sum_{e=1}^{n} x_e x_e y_e z_e \\
&\qquad + \eta_b (m-1) \operatorname{Tr}(C) \sum_{e=1}^{n} x_e x_e \sum_{f \neq e} y_f z_f + \eta_b (m-1) \operatorname{Tr}(C) \sum_{e=1}^{n} x_e z_e \sum_{f \neq e} x_f y_f \\
&\qquad + \tau_b (m-1) \operatorname{Tr}(C) \sum_{e=1}^{n} x_e y_e \sum_{h \neq e} x_h z_h + \gamma_b (m-1) \operatorname{Tr}(C) \sum_{e=1}^{n} x_e x_e y_e z_e.
\end{aligned}$$

Defining $w = \sum_{e=1}^{n} x_e^2 y_e z_e$ allows us to simplify further.

$$\begin{aligned}
&\mathrm{E}[x^T B^T B y x^T B^T C B z \mid C] \\
&\quad = \nu_b \operatorname{Tr}(C) \left(x^T x y^T z - w\right) + \nu_b \operatorname{Tr}(C) \left(x^T z x^T y - w\right) \\
&\qquad + \nu_b \operatorname{Tr}(C) \left(x^T y x^T z - w\right) + \rho_b \operatorname{Tr}(C) (w) \\
&\qquad + \eta_b (m-1) \operatorname{Tr}(C) \left(x^T x y^T z - w\right) + \eta_b (m-1) \operatorname{Tr}(C) \left(x^T z x^T y - w\right) \\
&\qquad + \tau_b (m-1) \operatorname{Tr}(C) \left(x^T y x^T z - w\right) + \gamma_b (m-1) \operatorname{Tr}(C) (w) .
\end{aligned}$$

Recall from the proof of Corollary 2 that $\rho_b = 3\nu_b$ and $\gamma_b = \tau_b + 2\eta_b$. This allows us to combine terms and arrive at a much simpler expression.

$$\begin{aligned}
&\mathrm{E}[x^T B^T B y x^T B^T C B z \mid C] \\
&\quad = \operatorname{Tr}(C) \left(\nu_b \left(x^T x y^T z + 2 x^T y x^T z\right) + \eta_b (m-1) \left(x^T x y^T z + x^T y x^T z\right) + \tau_b (m-1) \left(x^T y x^T z\right)\right) .
\end{aligned}$$

Substitute in the explicit expressions for ν_b, η_b, and τ_b from Corollary 2.

$$\begin{aligned}
&\mathrm{E}[x^T B^T B y x^T B^T C B z \mid C] \\
&\quad = \operatorname{Tr}(C) \left(x^T x y^T z + 2 x^T y x^T z\right) \left(\prod_{r=1}^{i-2} m_r (m_r + 2)\right) \\
&\qquad + \operatorname{Tr}(C) \left(x^T x y^T z + x^T y x^T z\right) (m_{i-1} - 1) \left(\frac{1}{3} \prod_{r=1}^{i-2} m_r (m_r + 2) - \frac{1}{3} \prod_{r=1}^{i-2} m_r (m_r - 1)\right) \\
&\qquad + \operatorname{Tr}(C) \left(x^T y x^T z\right) (m_{i-1} - 1) \left(\frac{1}{3} \prod_{r=1}^{i-2} m_r (m_r + 2) + \frac{2}{3} \prod_{r=1}^{i-2} m_r (m_r - 1)\right) .
\end{aligned}$$

With some rearranging, we find an even nicer form.

$$\begin{aligned}
&\mathrm{E}[x^T B^T B y x^T B^T C B z \mid C] \\
&\quad = \frac{\operatorname{Tr}(C)}{m_{i-1}} \left(\frac{2 x^T y x^T z + x^T x y^T z}{3} \prod_{r=1}^{i-1} m_r (m_r + 2) + \frac{x^T y x^T z - x^T x y^T z}{3} \prod_{r=1}^{i-1} m_r (m_r - 1)\right) .
\end{aligned}$$

Finally, note that $\mathrm{Tr}(C) = \|B_{j-1,i}\|_F^2$.

$$\begin{aligned}
&\mathrm{E}[x^T B^T B y x^T B^T C B z \mid C] \\
&\quad = \frac{\|B_{j-1,i}\|_F^2}{m_{i-1}} \left(\frac{2x^T y x^T z + x^T x y^T z}{3} \prod_{r=1}^{i-1} m_r(m_r+2) + \frac{x^T y x^T z - x^T x y^T z}{3} \prod_{r=1}^{i-1} m_r(m_r-1) \right).
\end{aligned}$$

This completes the proof. $\square$

Lemma A.12. *Let $m_0, \ldots, m_n \in \mathbb{N}$. For $1 \le q \le n$, let A_q be an $m_q \times m_{q-1}$ matrix. Let all of the elements of all of the A_q be independent and identically distributed. Denote by a_{qij} the element in the ith row and jth column of A_q. Let $\mathrm{E}[a_{qij}] = 0$, $\mathrm{E}[a_{qij}^2] = 1$, and $\mathrm{E}[a_{qij}^4] = 3$. Let $B_{q,r} = A_q \cdots A_r$ for $q \ge r$. Let $1 \le i \le j \le n$. Then*

$$\mathrm{E}\left[\frac{\|B_{j-1,i}\|_F^2}{m_{i-1}} \frac{\|B_{j,i+1}\|_F^2}{m_j} \right] = \frac{1}{3} \prod_{r=i}^{j-1} m_r(m_r+2) + \frac{2}{3} \prod_{r=i}^{j-1} m_r(m_r-1).$$

Here $\|\cdot\|_F$ denotes the Frobenius norm.

Proof. To keep the notation manageable, let $B = B_{j-1,i+1}$, $C = A_i$, $D = A_j$, $q = m_{i-1}$, $r = m_i$, $s = m_{j-1}$, and $t = m_j$. With this new notation, we derive

$$\mathrm{E}\left[\|BC\|_F^2 \|DB\|_F^2 \right].$$

Note that B is $s \times r$, C is $r \times q$, and D is $t \times s$.

First, we derive $\|BC\|_F^2$ as a sum.

$$\begin{aligned}
\|BC\|_F^2 &= \sum_{e=1}^{s} \sum_{f=1}^{q} (BC)_{ef}^2 \\
&= \sum_{e=1}^{s} \sum_{f=1}^{q} \left(\sum_{g=1}^{r} b_{eg} c_{gf} \right)^2 \\
&= \sum_{e=1}^{s} \sum_{f=1}^{q} \sum_{g=1}^{r} \sum_{h=1}^{r} b_{eg} b_{eh} c_{gf} c_{hf}.
\end{aligned}$$

We take the expectation given B to eliminate the c terms.

$$\begin{aligned}
\mathrm{E}\left[\|BC\|_F^2 \mid B\right] =& \sum_{e=1}^{s}\sum_{f=1}^{q}\sum_{g=1}^{r}\sum_{h=1}^{r} b_{eg}b_{eh}\,\mathrm{E}[c_{gf}c_{hf}] \\
=& \sum_{e=1}^{s}\sum_{f=1}^{q}\sum_{g=1}^{r} b_{eg}^2\,\mathrm{E}[c_{gf}^2] \\
=& q\sum_{e=1}^{s}\sum_{g=1}^{r} b_{eg}^2.
\end{aligned}$$

We do the same for $\|DB\|_F^2$

$$\begin{aligned}
\|DB\|_F^2 =& \sum_{i=1}^{t}\sum_{j=1}^{r}(DB)_{ij}^2 \\
=& \sum_{i=1}^{t}\sum_{j=1}^{r}\left(\sum_{k=1}^{s} d_{ik}b_{kj}\right)^2 \\
=& \sum_{i=1}^{t}\sum_{j=1}^{r}\sum_{k=1}^{s}\sum_{l=1}^{s} b_{kj}b_{lj}d_{ik}d_{il}.
\end{aligned}$$

As with the c terms, the independence of the d terms causes most of them to vanish.

$$\begin{aligned}
\mathrm{E}\left[\|DB\|_F^2 \mid B\right] =& \sum_{i=1}^{t}\sum_{j=1}^{r}\sum_{k=1}^{s}\sum_{l=1}^{s} b_{kj}b_{lj}\,\mathrm{E}[d_{ik}d_{il}] \\
=& \sum_{i=1}^{t}\sum_{j=1}^{r}\sum_{k=1}^{s} b_{kj}^2\,\mathrm{E}[d_{ik}d_{ik}] \\
=& t\sum_{j=1}^{r}\sum_{k=1}^{s} b_{kj}^2.
\end{aligned}$$

We combine the previous two expressions to obtain the object in question.

$$\frac{1}{qt}\,\mathrm{E}\left[\|BC\|_F^2\|DB\|_F^2\right] = \sum_{e=1}^{s}\sum_{g=1}^{r}\sum_{j=1}^{r}\sum_{k=1}^{s}\mathrm{E}[b_{eg}^2 b_{kj}^2].$$

As always, it is essential to break up the sum based on which indices might be equal.

$$\begin{aligned}
\mathrm{E}\left[\|DB\|_F^2 \mid B\right] =& \sum_{e=1}^{s}\sum_{g=1}^{r} \mathrm{E}[b_{eg}^4] \\
&+ \sum_{e=1}^{s}\sum_{g=1}^{r}\sum_{j\neq g} \mathrm{E}[b_{eg}^2 b_{ej}^2] \\
&+ \sum_{e=1}^{s}\sum_{g=1}^{r}\sum_{k\neq e} \mathrm{E}[b_{eg}^2 b_{kg}^2] \\
&+ \sum_{e=1}^{s}\sum_{g=1}^{r}\sum_{j\neq g}\sum_{k\neq e} \mathrm{E}[b_{eg}^2 b_{kj}^2].
\end{aligned}$$

Substitute in the constants from Lemma A.1.

$$\mathrm{E}\left[\|DB\|_F^2 \mid B\right] = sr\rho_b + sr(r-1)\nu_b + sr(s-1)\gamma_b + sr(s-1)(r-1)\tau_b.$$

Now we can return to our original notation and use the expressions from Corollary 2.

$$\begin{aligned}
\mathrm{E}\left[\frac{\|B_{j-1,i}\|_F^2}{m_{i-1}}\frac{\|B_{j,i+1}\|_F^2}{m_j}\right] =& 3m_i m_{j-1}\prod_{r=i+1}^{j-2} m_r(m_r+2) \\
&+ m_{j-1}m_i(m_i-1)\prod_{r=i+1}^{j-2} m_r(m_r+2) \\
&+ m_{j-1}m_i(m_{j-1}-1)\prod_{r=i+1}^{j-2} m_r(m_r+2) \\
&+ \frac{1}{3}m_{j-1}m_i(m_{j-1}-1)(m_i-1)\prod_{r=i+1}^{j-2} m_r(m_r+2) \\
&+ \frac{2}{3}m_{j-1}m_i(m_{j-1}-1)(m_i-1)\prod_{r=i+1}^{j-2} m_r(m_r-1).
\end{aligned}$$

All of this simplifies in a straightforward way to a nicer expression.

$$\mathrm{E}\left[\frac{\|B_{j-1,i}\|_F^2}{m_{i-1}}\frac{\|B_{j,i+1}\|_F^2}{m_j}\right] = \frac{1}{3}\prod_{r=i}^{j-1} m_r(m_r+2) + \frac{2}{3}\prod_{r=i}^{j-1} m_r(m_r-1).$$

This completes the proof. $\square$

Lemma A.13. *Let $m_0, \ldots, m_n \in \mathbb{N}$. For $1 \le q \le n$, let A_q be an $m_q \times m_{q-1}$ matrix. Let all of the elements of all of the A_q be independent and identically distributed. Denote by a_{qij} the element in the ith row and jth column of A_q. Let $\mathrm{E}[a_{qij}] = 0$, $\mathrm{E}[a_{qij}^2] = 1$, and $\mathrm{E}[a_{qij}^4] = 3$. Let $B_{q,r} = A_q \cdots A_r$ for $q \ge r$. Let $1 \le i \le j \le n$. Let $w, v \in \mathbb{R}^{m_n}$ and let P be $m_n \times m_n$ and symmetric. Then*

$$\begin{aligned}
&\mathrm{E}\left[w^T B_{n,j+1} B_{n,j+1}^T P B_{n,i+1} B_{n,i+1}^T v \mid A_{i+i}, \ldots, A_j\right] \\
&= \frac{\|B_{j,i+1}\|_F^2}{m_j}\left(\frac{2w^T P v + \mathrm{Tr}(P) w^T v}{3}\prod_{r=j}^{n-1} m_r(m_r+2) + \frac{w^T P v - \mathrm{Tr}(P) w^T v}{3}\prod_{r=j}^{n-1} m_r(m_r-1)\right).
\end{aligned}$$

Here $\|\cdot\|_F$ denotes the Frobenius norm.

Proof. To keep the notation manageable, let $B = B_{j+1,n}$, $D = B_{j,i+1}^T B_{j,i+1}$, $n = m_n$, and $m = m_j$. With this new notation, we derive

$$\mathrm{E}\left[w^T B B^T P B D B^T v \mid D\right].$$

Note that x, y, and z are $n \times 1$, B is $n \times m$, P is $n \times n$, and D is $m \times m$.

First, we derive an explicit formula for the elements of $P^T B B^T w$.

$$\begin{aligned}
(P^T B B^T w)_e &= \sum_{f=1}^{m} (P^T B)_{ef} (B^T w)_f \\
&= \sum_{f=1}^{m} \sum_{g=1}^{n} p_{ge} b_{gf} \sum_{h=1}^{n} b_{hf} w_h \\
&= \sum_{f=1}^{m} \sum_{g=1}^{n} \sum_{h=1}^{n} b_{gf} b_{hf} p_{ge} w_h.
\end{aligned}$$

We do the same for BDB^Tv.

$$\begin{aligned}(BDB^Tv)_e &= \sum_{i=1}^{m}(BD)_{ei}(B^Tv)_i \\ &= \sum_{i=1}^{m}\sum_{j=1}^{m} b_{ej}d_{ji}\sum_{k=1}^{n} b_{ki}v_k \\ &= \sum_{i=1}^{m}\sum_{j=1}^{m}\sum_{k=1}^{n} b_{ej}b_{ki}d_{ji}v_k.\end{aligned}$$

These formulas let us compute $w^TBB^TPBDB^Tv = (P^TBB^Tw)^T(BDB^Tv)$.

$$w^TBB^TPBDB^Tv = \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{g=1}^{n}\sum_{h=1}^{n}\sum_{i=1}^{m}\sum_{j=1}^{m}\sum_{k=1}^{n} b_{ej}b_{gf}b_{hf}b_{ki}d_{ji}p_{ge}w_hv_k.$$

By Corollary A.7, $\mathrm{E}[b_{ej}b_{gf}b_{hf}b_{ki}] = 0$ if $i \neq j$. We get to remove one index.

$$\mathrm{E}\left[w^TBB^TPBDB^Tv \mid D\right] = \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{g=1}^{n}\sum_{h=1}^{n}\sum_{i=1}^{m}\sum_{k=1}^{n} \mathrm{E}[b_{ei}b_{gf}b_{hf}b_{ki}]d_{ii}p_{ge}w_hv_k.$$

By Corollary A.7, $\mathrm{E}[b_{ei}b_{gf}b_{hf}b_{ki}]$ is nonzero in four cases: $e = k \neq g = h$, $e = g \neq h = k$, $e = h \neq g = k$, and $e = g = h = k$. We break the sum up accordingly.

$$\begin{aligned}&\mathrm{E}[w^TBB^TPBDB^Tv \mid D] \\ &\quad = \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{g\neq e}\sum_{i=1}^{m} \mathrm{E}[b_{ei}^2bYo_{gf}^2]d_{ii}p_{ge}w_gv_e + \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{h\neq e}\sum_{i=1}^{m} \mathrm{E}[b_{ei}b_{ef}b_{hf}b_{hi}]d_{ii}p_{ee}w_hv_h \\ &\quad\quad + \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{g\neq e}\sum_{i=1}^{m} \mathrm{E}[b_{ei}b_{gf}b_{ef}b_{gi}]d_{ii}p_{ge}w_ev_g + \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{i=1}^{m} \mathrm{E}[b_{ei}^2b_{ef}^2]d_{ii}p_{ee}w_ev_e.\end{aligned}$$

We break the sum apart again to handle the cases $i = f$ and $i \neq f$.

$$
\begin{aligned}
&\mathrm{E}[w^T BB^T PBDB^T v \mid D] \\
&\quad = \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{g\neq e} \mathrm{E}[b_{ef}^2 b_{gf}^2] d_{ff} p_{ge} w_g v_e + \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{h\neq e} \mathrm{E}[b_{ef}^2 b_{hf}^2] d_{ff} p_{ee} w_h v_h \\
&\qquad + \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{g\neq e} \mathrm{E}[b_{ef}^2 b_{gf}^2] d_{ff} p_{ge} w_e v_g + \sum_{e=1}^{n}\sum_{f=1}^{m} \mathrm{E}[b_{ef}^4] d_{ff} p_{ee} w_e v_e \\
&\qquad + \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{g\neq e}\sum_{i\neq f} \mathrm{E}[b_{ei}^2 b_{gf}^2] d_{ii} p_{ge} w_g v_e + \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{h\neq e}\sum_{i\neq f} \mathrm{E}[b_{ei} b_{ef} b_{hf} b_{hi}] d_{ii} p_{ee} w_h v_h \\
&\qquad + \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{g\neq e}\sum_{i\neq f} \mathrm{E}[b_{ei} b_{gf} b_{ef} b_{gi}] d_{ii} p_{ge} w_e v_g + \sum_{e=1}^{n}\sum_{f=1}^{m}\sum_{i\neq f} \mathrm{E}[b_{ei}^2 b_{ef}^2] d_{ii} p_{ee} w_e v_e.
\end{aligned}
$$

We can now identify the constants from Lemma A.1 and separate out $\mathrm{Tr}(D)$.

$$
\begin{aligned}
&\mathrm{E}[w^T BB^T PBDB^T v \mid D] \\
&\quad = \gamma_b \,\mathrm{Tr}(D) \sum_{e=1}^{n}\sum_{g\neq e} p_{ge} w_g v_e + \gamma_b \,\mathrm{Tr}(D) \sum_{e=1}^{n}\sum_{h\neq e} p_{ee} w_h v_h \\
&\qquad + \gamma_b \,\mathrm{Tr}(D) \sum_{e=1}^{n}\sum_{g\neq e} p_{ge} w_e v_g + \rho_b \,\mathrm{Tr}(D) \sum_{e=1}^{n} p_{ee} w_e v_e \\
&\qquad + \tau_b (m-1) \,\mathrm{Tr}(D) \sum_{e=1}^{n}\sum_{g\neq e} p_{ge} w_g v_e + \eta_b (m-1) \,\mathrm{Tr}(D) \sum_{e=1}^{n}\sum_{h\neq e} p_{ee} w_h v_h \\
&\qquad + \eta_b (m-1) \,\mathrm{Tr}(D) \sum_{e=1}^{n}\sum_{g\neq e} p_{ge} w_e v_g + \nu_b (m-1) \,\mathrm{Tr}(D) \sum_{e=1}^{n} p_{ee} w_e v_e.
\end{aligned}
$$

Use the relations from the proof of Corollary 2 to simplify. Replace h with g so that terms match.

$$
\begin{aligned}
&\mathrm{E}[w^T BB^T PBDB^T v \mid D] \\
&\quad = \mathrm{Tr}(D) \left(\gamma_b \sum_{e=1}^{n}\sum_{g=1}^{n} p_{ge} w_g v_e + \gamma_b \sum_{e=1}^{n}\sum_{g=1}^{n} p_{ee} w_g v_g + \gamma_b \sum_{e=1}^{n}\sum_{g=1}^{n} p_{ge} w_e v_g \right). \\
&\qquad + (m-1) \,\mathrm{Tr}(D) \left(\tau_b \sum_{e=1}^{n}\sum_{g=1}^{n} p_{ge} w_g v_e + \eta_b \sum_{e=1}^{n}\sum_{g=1}^{n} p_{ee} w_g v_g + \eta_b \sum_{e=1}^{n}\sum_{g=1}^{n} p_{ge} w_e v_g \right)
\end{aligned}
$$

These sums can now be expressed conveniently in the language of linear algebra.

$$\begin{aligned}
&\mathrm{E}[w^T B B^T P B D B^T v \mid D] \\
&\quad = \mathrm{Tr}(D)\left(w^T P v \gamma_b + \mathrm{Tr}(P) w^T v \gamma_b + v^T P w \gamma_b\right) \\
&\qquad + (m-1)\,\mathrm{Tr}(D)\left(w^T P v \tau_b + \mathrm{Tr}(P) w^T v \eta_b + v^T P w \eta_b\right).
\end{aligned}$$

Substitute in the expressions from Lemma A.8.

$$\begin{aligned}
&\mathrm{E}[w^T B B^T P B D B^T v \mid D] \\
&\quad = \frac{\mathrm{Tr}(D)}{m}\left(w^T P v + \mathrm{Tr}(P) w^T v + v^T P w\right) m_j \left(\prod_{r=j+1}^{n-1} m_r(m_r+2)\right) \\
&\qquad + \frac{\mathrm{Tr}(D)}{m}\left(w^T P v\right) m_j(m_j-1)\left(\frac{1}{3}\prod_{r=j+1}^{n-1} m_r(m_r+2) + \frac{2}{3}\prod_{r=j+1}^{n-1} m_r(m_r-1)\right) \\
&\qquad + \frac{\mathrm{Tr}(D)}{m}\left(\mathrm{Tr}(P) w^T v + v^T P w\right) m_j(m_j-1)\left(\frac{1}{3}\prod_{r=j+1}^{n-1} m_r(m_r+2) - \frac{1}{3}\prod_{r=j+1}^{n-1} m_r(m_r-1)\right).
\end{aligned}$$

Rearrange.

$$\begin{aligned}
&\mathrm{E}[w^T B B^T P B D B^T v \mid D] \\
&\quad = \frac{\mathrm{Tr}(D)}{m_j}\left(\frac{w^T P v + \mathrm{Tr}(P) w^T v + v^T P w}{3}\prod_{r=j}^{n-1} m_r(m_r+2)\right) \\
&\qquad + \frac{\mathrm{Tr}(D)}{m_j}\left(\frac{2w^T P v - \mathrm{Tr}(P) w^T v - v^T P w}{3}\prod_{r=j}^{n-1} m_r(m_r-1)\right).
\end{aligned}$$

Use the symmetry of P to simplify.

$$\begin{aligned}
&\mathrm{E}[w^T B B^T P B D B^T v \mid D] \\
&\quad = \frac{\mathrm{Tr}(D)}{m_j}\left(\frac{2w^T P v + \mathrm{Tr}(P) w^T v}{3}\prod_{r=j}^{n-1} m_r(m_r+2) + \frac{w^T P v - \mathrm{Tr}(P) w^T v}{3}\prod_{r=j}^{n-1} m_r(m_r-1)\right).
\end{aligned}$$

Recall that $D = B_{j,i+1}^T B_{j,i+1}$ and so $\mathrm{Tr}(D) = \|B_{j,i+1}\|_F^2$.

$$\begin{aligned}
&\mathrm{E}[w^T B B^T P B D B^T v \mid D] \\
=&\frac{\|B_{j,i+1}\|_F^2}{m_j}\left(\frac{2w^T P v + \mathrm{Tr}(P) w^T v}{3}\prod_{r=j}^{n-1} m_r(m_r+2) + \frac{w^T P v - \mathrm{Tr}(P) w^T v}{3}\prod_{r=j}^{n-1} m_r(m_r-1)\right).
\end{aligned}$$

This completes the proof. $\square$

Theorem A.14. *Let $m_0, \dots, m_n \in \mathbb{N}$. For $1 \le q \le n$, let A_q be an $m_q \times m_{q-1}$ matrix. Let all of the elements of all of the A_q be independent and identically distributed. Denote by a_{qij} the element in the ith row and jth column of A_q. Let $\mathrm{E}[a_{qij}] = 0$, $\mathrm{E}[a_{qij}^2] = 1$, and $\mathrm{E}[a_{qij}^4] = 3$. Let $B_{q,r} = A_q \cdots A_r$ for $q \ge r$. Let $1 \le i \le j \le n$. Let $x, y, z \in \mathbb{R}^{m_0}$. Let $w, v \in \mathbb{R}^{m_n}$ and let P be $m_n \times m_n$ and symmetric. Define*

$$\begin{aligned}
f_{ij} &= \frac{2x^T y x^T z + x^T x y^T z}{3}\prod_{r=1}^{i-1} m_r(m_r+2) + \frac{x^T y x^T z - x^T x y^T z}{3}\prod_{r=1}^{i-1} m_r(m_r-1), \\
g_{ij} &= \frac{1}{3}\prod_{r=i}^{j-1} m_r(m_r+2) + \frac{2}{3}\prod_{r=i}^{j-1} m_r(m_r-1), \\
h_{ij} &= \frac{2w^T P v + \mathrm{Tr}(P) w^T v}{3}\prod_{r=j}^{n-1} m_r(m_r+2) + \frac{w^T P v - \mathrm{Tr}(P) w^T v}{3}\prod_{r=j}^{n-1} m_r(m_r-1).
\end{aligned}$$

Then

$$\mathrm{E}\left[\left(x^T B_{i-1,1}^T B_{i-1,1} y\right)\left(x^T B_{j-1,1}^T B_{j-1,1} z\right)\left(w^T B_{n,j+1} B_{n,j+1}^T P B_{n,i+1} B_{n,i+1}^T v\right)\right] = f_{ij} g_{ij} h_{ij}.$$

Proof. Let

$$c = \left(x^T B_{i-1,1}^T B_{i-1,1} y\right)\left(x^T B_{j-1,1}^T B_{j-1,1} z\right)\left(w^T B_{n,j+1} B_{n,j+1}^T P B_{n,i+1} B_{n,i+1}^T v\right).$$

Because each A_q is independent of all the others, we can factor the expectation as

$$\mathrm{E}[c] = \mathrm{E}[c \mid A_i, A_{i+1}, \dots, A_n]\,\mathrm{E}[c \mid A_1, A_2, \dots A_{i-1}, A_j, A_{j+1}, \dots, A_n]\,\mathrm{E}[c \mid A_1, A_2, \dots, A_{j-1}].$$

Lemma A.11, Lemma A.12, and Lemma A.13 establish that

$$\begin{aligned} \mathrm{E}[c \mid A_i, A_{i+1}, \ldots, A_n] &= f_{ij}, \\ \mathrm{E}[c \mid A_1, A_2, \ldots A_{i-1}, A_j, A_{j+1}, \ldots, A_n] &= g_{ij}, \\ \mathrm{E}[c \mid A_1, A_2, \ldots, A_{j-1}] &= h_{ij}, \end{aligned}$$

respectively. □

Corollary A.15. *Let all be as in Theorem A.14 except with each A_q scaled by a factor of $\sigma_q > 0$, causing $\mathrm{E}[a_{qij}^2] = \sigma_q^2$, and $\mathrm{E}[a_{qij}^4] = 3\sigma_q^4$. Then*

$$\begin{aligned} &\mathrm{E}\left[\left(x^T B_{i-1,1}^T B_{i-1,1} y\right)\left(x^T B_{j-1,1}^T B_{j-1,1} z\right)\left(w^T B_{n,j+1} B_{n,j+1}^T P B_{n,i+1} B_{n,i+1}^T v\right)\right] \\ &= \frac{\sigma_1^4 \sigma_2^4 \cdots \sigma_n^4}{\sigma_i^2 \sigma_j^2} f_{ij} g_{ij} h_{ij}. \end{aligned}$$

In particular, if $\sigma_q = (m_q m_{q-1})^{-1/4}$, then the expectation becomes

$$\frac{(m_{i-1} m_i m_{j-1} m_j)^{1/2}}{m_0 m_1^2 m_2^2 \cdots m_{n-1}^2 m_n} f_{ij} g_{ij} h_{ij}.$$

Proof. If $i < j$, then each of the A_q appear four times in the above product except for A_i and A_j, which each appear twice. If $i = j$, then each of the A_q appear four times except for A_i, which does not appear at all. □

Bibliography

[1] Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2(3):183 – 192, 1989.

[2] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

[3] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989.

[4] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

[5] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861 – 867, 1993.

[6] K. Hornik. Some new results on neural network approximation. *Neural Networks*, 6(8):1069 – 1072, 1993.

[7] Andrew R. Barron. Approximation and estimation bounds for artificial neural networks. *Machine Learning*, 14(1):115–133, Jan 1994.

[8] Franco Scarselli and Ah Chung Tsoi. Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results. *Neural Networks*, 11(1):15 – 37, 1998.

[9] Henry W. Lin, Max Tegmark, and David Rolnick. Why does deep and cheap learning work so well? *Journal of Statistical Physics*, 168(6):1223–1247, Sep 2017.

[10] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2924–2932. Curran Associates, Inc., 2014.

[11] Matus Telgarsky. Representation benefits of deep feedforward networks. *CoRR*, abs/1509.08101, 2015.

[12] Hrushikesh Mhaskar, Qianli Liao, and Tomaso A. Poggio. Learning real and boolean functions: When is deep better than shallow. *CoRR*, abs/1603.00988, 2016.

[13] Shiyu Liang and R. Srikant. Why deep neural networks? *CoRR*, abs/1610.04161, 2016.

[14] H. N. Mhaskar and T. Poggio. Deep vs. shallow networks: An approximation theory perspective. *Analysis and Applications*, 14(06):829–848, 2016.

[15] Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. *CoRR*, abs/1611.01491, 2016.

[16] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl Dickstein. On the expressive power of deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 2847–2854. JMLR.org, 2017.

[17] David Rolnick and Max Tegmark. The power of deeper networks for expressing natural functions. *CoRR*, abs/1705.05502, 2017.

[18] Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. In Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir, editors, *29th Annual Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pages 907–940, Columbia University, New York, New York, USA, 23–26 Jun 2016. PMLR.

[19] Nadav Cohen, Or Sharir, and Amnon Shashua. On the expressive power of deep learning: A tensor analysis. In Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir, editors, *29th Annual Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pages 698–728, Columbia University, New York, New York, USA, 23–26 Jun 2016. PMLR.

[20] Matus Telgarsky. Benefits of depth in neural networks. In Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir, editors, *29th Annual Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pages 1517–1539, Columbia University, New York, New York, USA, 23–26 Jun 2016. PMLR.

[21] Dmytro Perekrestenko, Philipp Grohs, Dennis Elbrächter, and Helmut Bölcskei. The universal approximation power of finite-width deep relu networks. *CoRR*, abs/1806.01528, 2018.

[22] Sho Sonoda and Noboru Murata. Neural network with unbounded activations is universal approximator. *CoRR*, abs/1505.03654, 2015.

[23] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. *CoRR*, abs/1709.02540, 2017.

[24] Nick Harvey, Christopher Liaw, and Abbas Mehrabian. Nearly-tight vc-dimension bounds for piecewise linear neural networks. In Satyen Kale and Ohad Shamir, editors, *Proceedings of the 2017 Conference on Learning Theory*, volume 65 of *Proceedings of Machine Learning Research*, pages 1064–1068, Amsterdam, Netherlands, 07–10 Jul 2017. PMLR.

[25] Dmitry Yarotsky. Error bounds for approximations with deep relu networks. *Neural Networks*, 94:103 – 114, 2017.

[26] Boris Hanin. Universal function approximation by deep neural nets with bounded width and relu activations. *arXiv preprint arXiv:1708.02691*, 2017.

[27] Boris Hanin and Mark Sellke. Approximating continuous functions by relu nets of minimal width. *arXiv preprint arXiv:1710.11278*, 2017.

[28] Wenling Shang, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. Understanding and improving convolutional neural networks via concatenated rectified linear units. *CoRR*, abs/1603.05201, 2016.

[29] Michael Blot, Matthieu Cord, and Nicolas Thome. Maxmin convolutional neural networks for image classification. *CoRR*, abs/1610.07882, 2016.

[30] Jonghong Kim, O. Sangjun, Yoonnyun Kim, and Minho Lee. Convolutional neural network with biologically inspired retinal structure. *Procedia Computer Science*, 88:145 – 154, 2016. 7th Annual International Conference on Biologically Inspired Cognitive Architectures, BICA 2016, held July 16 to July 19, 2016 in New York City, NY, USA.

[31] James Clerk Maxwell. V. illustrations of the dynamical theory of gases.part i. on the motions and collisions of perfectly elastic spheres. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 19(124):19–32, 1860.

[32] David Balduzzi, Marcus Frean, Lennox Leary, J. P. Lewis, Kurt Wan-Duo Ma, and Brian McWilliams. The shattered gradients problem: If resnets are the answer, then what is the question? *CoRR*, abs/1702.08591, 2017.

[33] M. Evans, N. Hastings, and B. Peacock. *Statistical Distributions.* Wiley Series in Probability and Statistics. Wiley, 2000.

[34] Yurii Nesterov. Introductory lectures on convex programming volume i: Basic course. *Lecture notes*, 3(4):5, 1998.

[35] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[36] Rachel Ward, Xiaoxia Wu, and Leon Bottou. Adagrad stepsizes: Sharp convergence over nonconvex landscapes, from any initialization. *arXiv preprint arXiv:1806.01811*, 2018.

[37] Xiaoxia Wu, Simon S Du, and Rachel Ward. Global convergence of adaptive gradient methods for an over-parameterized neural network. *arXiv preprint arXiv:1902.07111*, 2019.