COMPUTING PRIME FACTORIZATIONS WITH NEURAL NETWORKS

By

Dakota Dragomir, M.S.

A Project Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Statistics

University of Alaska Fairbanks

December 2022

APPROVED:

Scott Goddard, Committee Chair
Margaret Short, Committee Member
Ronald Barry, Committee Member
Leah Berman, Chair
Department of Mathematics & Statistics
Karsten Hueffer, Interim Dean
College of Natural Science & Mathematics
Richard Collins, *Director of the Graduate School*

Abstract

When dealing with sufficiently large integers, even the most cutting-edge existing algorithms for computing prime factorizations are impractically slow. In this paper, we explore the possibility of using neural networks to approximate prime factorizations in the hopes of providing an alternative factorization method which trades accuracy for speed. Due to the intrinsic difficulty associated with this task, the focus of this paper is largely concentrated on the obstacles encountered in the training of the neural net, rather than on the viability of the method itself.

Table of Contents

Page

Title Page i

Abstract iii

Table of Contents v

List of Figures vii

Acknowledgments ix

Chapter 1 Introduction 1

Chapter 2 Preliminary Theory 3
2.1 Machine Learning 3
2.2 Neural Networks 14

Chapter 3 Set-Up 24
3.1 Prime Numbers Review 24
3.2 Problem Formulation, Part 1 25
3.3 Problem Formulation, Part 2 27
3.4 Experimental Design and Neural Network Architecture 32

Chapter 4 Training 38
4.1 Training Approach 38
4.2 Training Results 41

Chapter 5 Discussion 56

References 60

Appendix 62

List of Figures

Page

Figure 2.1 A visualization of the decomposition of excess risks $\mathcal{E}(\hat{f}_1, f), \mathcal{E}(\hat{f}_2, f)$ for two estimates $\hat{f}_1$, $\hat{f}_2$ which use hypothesis spaces $\mathcal{H}_1$, $\mathcal{H}_2$ respectively. Red denotes excess risk, green denotes estimation error, while blue denotes approximation error. Notice how the estimate in the smaller hypothesis space $\mathcal{H}_1$ exhibits small estimation error with large approximation error, while the estimate in the larger hypothesis space $\mathcal{H}_2$ exhibits large estimation error with small approximation error, but ultimately $\hat{f}_1$ exhibits lower excess risk than $\hat{f}_2$. 9

Figure 2.2 A partial visualization of the model selection process involving four potential estimates $\hat{f}_1$, $\hat{f}_2$, $\hat{f}_3$, $\hat{f}_4$ which use (nested, for simplicity) hypothesis spaces $\mathcal{H}_1$, $\mathcal{H}_2$, $\mathcal{H}_3$, $\mathcal{H}_4$ respectively. Red denotes excess risk, while the various shades of green distinguish between hypothesis spaces. Notice that $\hat{f}_2$ produces the lowest excess risk, since $\mathcal{H}_2$ provides the optimal estimation-approximation error trade-off of the hypothesis spaces considered, and would ideally be the estimate selected during model selection. Not pictured is the fact that the true excess risk is not known to us and is therefore not available for comparisons; instead, we must estimate which of the $\hat{f}_1$, $\hat{f}_2$, $\hat{f}_3$, $\hat{f}_4$ provides the lowest excess risk using in-sample or out-of-sample methods. 12

Figure 2.3 The composition diagram of a neural net with an input layer consisting of 3 nodes, two hidden layers each consisting of 4 nodes, and an output layer consisting of 2 nodes. Blue denotes input nodes, red denotes hidden nodes, and green denotes output nodes. The diagram flows from left to right, as the input layer is fed into the first hidden layer, which in turn is fed into the second hidden layer, and so on. Notice that the input layer consists of 3 nodes, indicating 3 input variables, while the output layer consists of 2 nodes, indicating 2 output variables. Due to the diagram's lack of cycles and the connectedness between nodes, this is also an example of a fully connected feedforward neural net. 14

Figure 2.4 (a) The composition diagram of a recurrent neural net with 3 recurrent hidden layers, where recurrent connections are denoted by loops, and (b) the composition diagram of a residual neural net with 3 hidden layers, where skip-connections are denoted by dashed lines and residual connections are denoted by squiggly lines. In both figures, blue denotes the input layer, red denotes hidden layers, and green denotes the output layer; nodes within layers are excluded for simplicity. 17

Figure 2.5 The composition diagram of a multilayer perceptron including bias nodes with an input layer consisting of 3 nodes, two hidden layers each consisting of 4 nodes, and an output layer consisting of 2 nodes. Blue denotes input nodes, red denotes hidden nodes, yellow denotes bias nodes, and green denotes output nodes. Notice that bias nodes can be visualized as nodes which take the value 1 constantly. While edges from the previous layer's nodes to a bias node could be drawn, ultimately these edges would be superfluous as the bias node is a constant and thus does not depend on previous layers meaningfully. Moreover, while bias nodes are found in the input and hidden layers, we consider them distinct from input and hidden nodes; in particular, aspects of the network architecture like layer width or connectedness do not apply to bias nodes. 20

Figure 3.1 An example of a potential multilayer perceptron architecture which could be used for estimating p_{10}, the restriction of the prime factorization map p to $\mathbb{Z}_{2\leq z\leq 10}$. Blue denotes the input layer, red denotes hidden layers, yellow denotes bias nodes, and green denotes the output layer. Notice that the input layer consists of 1 node, since we have 1 input variable, while the output layer consists of 4 nodes, as there are 4 primes in $\mathbb{Z}_{2\leq z\leq 10}$ and thus 4 sequence entries to predict. Moreover, there are 2 hidden layers, each with a width of 4; other architectures under consideration could have different hidden layer depths and widths, but their input and output layers will remain the same. 35

Figure 4.1 A visualization of the effect of ReLU-induced dead neurons on the neural net architecture. Black denotes dead neurons while red denotes active hidden neurons. Note how the dead neurons effectively narrow the network. While this is typically a good thing, as networks are oftentimes wider than they need to be, too much narrowing can restrict the neural net's ability to vary with respect to inputs. At the extreme end of the spectrum, a layer can consist entirely of dead neurons (visualized as a layer of only black nodes), in which case the neural network is in fact a constant function..... 51

Acknowledgments

I'd like to thank my advisor, Scott Goddard, for his suggestion of this project topic. I'd also like to thank my parents, John Dragomir and Collene Brady-Dragomir, my fiancée, Akashia Martinez, and my cat, Ella, for all the support they have given me while I worked to complete my project.

Chapter 1 Introduction

In practice, computing the prime factorization of a large integer is considered to be a difficult task. The difficulty stems from the fact that while there are many existing procedures which can in theory completely factor an integer, ranging from the simple brute-force method of Trial Division to the cutting-edge General Number Field Sieve algorithm (*Lenstra et al.* [1993]), the time required for factoring a sufficiently large integer under even the fastest known algorithm is so large as to make the exercise practically impossible. While the non-existence of a fast prime factorization algorithm has not been formally established, and thus the question of whether or not prime factorization is provably difficult remains an open problem, the fact that no existing approach is fast means that for all intents and purposes prime factorization is hard.

Since current factorization algorithms are all computationally expensive when dealing with sufficiently large integers, the existence of an alternative factorization method which trades off lower accuracy (i.e. the computed prime factorizations are no longer exactly correct) for increased computational speed could prove useful in some situations. The aim of this project is to create such a method by using the tools of machine learning to train a neural network to compute prime factorizations. Ideally, the neural net would compute slightly inaccurate prime factorizations but do so far quicker than other available factorization methods. Unfortunately, due to the apparent complexity of prime factorization, as well as the known difficulties of using neural nets, this ideal is not realistic. Hence, rather than focusing seriously on whether the neural net performs well or not, this project takes a more expository tone, using the problem of approximately computing prime factorizations using neural nets as an illustrative context for understanding some of the statistical theory, practices, and problems encountered in machine learning.

In more detail, the structure of this paper is as follows. In Chapter 2, a description of the goals and theory of machine learning/statistical learning and neural nets is given. In Chapter 3, a formalization of the prime factorization problem in the language of machine learning is provided and our notion of approximating prime factorizations is defined, supplying the framework within which the rest of the paper takes place. Then, Chapter 4 follows our efforts to train neural nets to compute prime factorizations and the issues encountered along the way, with a particular emphasis on overcoming the Dying ReLU problem. Afterwards, Chapter 5 concludes the paper with a discussion of the remaining issues we encountered in the process of training neural nets, and some potential solutions to those problems that could be implemented in the future.

Note that while all of the theory detailed in Chapter 2 can be found within typical statistical learning/machine learning/deep learning texts such as *Hastie et al.* [2009], *Shalev-Shwartz and Ben-David* [2014], and *Goodfellow et al.* [2016], the language used in this paper to express these concepts differs (at times significantly) from the language used in these resources. This is because the approach taken in this project is to explain the ideas behind machine learning explicitly in the framework of inferential statistics, so that a statistician who knows nothing about machine learning could read this paper and get something out of it. In other words, this project is intended to provide a sort of dictionary which translates the standard presentation of machine learning into standard statistical language. Although inferential statistics and statistical decision theory provide the foundations for every idea involved in machine learning, most resources on this subject tend to leave the statistical considerations implicit (due to the field's focus on practice and application) or use terminology which makes it difficult to see what's going on statistically. Thus, while the ideas discussed in Chapter 2 are identical to those found in standard machine learning texts, in spirit our presentation is novel since we try to explicitly describe every idea in familiar statistical language.

Chapter 2 Preliminary Theory

2.1 Machine Learning

Although the exact definition of machine learning is a contentious matter amongst practitioners, it could be said that the defining characteristic of machine learning is the usage of data to make or improve predictions; this is the "learning" refered to in machine learning, while "machine" refers to the fact that oftentimes a computer is used to automate the learning process. Due to its early and widespread adoption by computer scientists for use in disciplines like artificial intelligence, the field of machine learning is often viewed as a computer science topic. However, due to its reliance on data, machine learning can perhaps more accurately be viewed as a branch of statistics concerned with using data to estimate the "optimal" way of using some variables to predict others (though the terminology used in machine learning is often different from that of statistics, due to its historical roots in computer science).

In more mathematical detail, the statistical framework behind machine learning centers around prediction and thus begins with a joint distribution (X, Y) between random variables X, Y which take values in $\mathcal{X}$, $\mathcal{Y}$ respectively. For one reason or another, we would like to use the value of X to make a guess or prediction about the value of Y; in more formal language, we seek a *prediction function* $f : \mathcal{X} \to \mathcal{Y}$ which takes a value of X and spits out the associated prediction for the value of Y. Of course, since some guessing methods are obviously better than others, we aren't seeking just any prediction function – what we really want is a prediction function f which makes good guesses.

To define how good f's predictions are, we need to specify the additional structure of a *loss function* $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^{\geq 0}$. The loss function is a quantitative measure whose essential purpose is to codify which aspects of Y we want our predictions to focus on pinning down; in

other words, $L(f(x), y)$ defines how well we consider the prediction $f(x)$ to approximate the possible realization y of Y. Commonly used loss functions in contexts where Y is quantitative include squared loss ($L(a, b) = (a - b)^2$) and absolute loss ($L(a, b) = |a - b|$), while the use of $0 - 1$ loss ($L(a, b) = \mathbf{1}_{x \neq y}(a, b)$) is typical when Y is categorical, though ultimately the choice of loss function is determined based off of external considerations such as the intended purpose of the predictions (as well as mathematical convenience). Oftentimes, loss functions are required to satisfy certain axioms in order to be considered genuine loss functions, but there is no universally agreed upon set of axioms which define them. For our purposes, we only require that the loss function L satisfies

$$L(y_1, y_2) = 0 \text{ if and only if } y_1 = y_2 \text{ for all } y_1, y_2 \in \mathcal{Y};$$

this requirement essentially guarantees that a loss of 0 corresponds to being "exactly the same" and vice versa, which is a natural property to require of the loss function.

Generally speaking, $\mathcal{Y}$ will be too large for a single prediction $f(x)$ to provide a low-loss approximation to every possible value of Y, thus making it impossible for $f(x)$ to perform well in all circumstances. Nonetheless, while there are many possible values that Y can take, some values are more likely to occur than others. Hence, by aiming $f(x)$ towards the likelier possibilities, we can make guesses which, while not necessarilly performing well in all situations, do perform well most of the time. This understanding, which is at the core of prediction, can be intuitively formalized by saying our prediction function f makes good guesses if it is a *conditional risk minimizer* (also known as a *conditional expected loss minimizer*); that is, for every $x \in \mathcal{X}$,

$$f(x) := \arg\min_c \mathbb{E}_{Y|X=x}[L(c, Y)].$$

Thus, the underlying goal of prediction (and therefore machine learning) is to find a conditional risk minimizing function $f : \mathcal{X} \to \mathcal{Y}$ for the joint distribution (X, Y).

Note that while our formulation of a good prediction rule f is defined in terms of conditional risk minimization, there are other possible criteria; one particularly common alternative is that f is instead required to be a *joint risk minimizer*, i.e. $f : \mathcal{X} \to \mathcal{Y}$ minimizes $\mathbb{E}_{(X,Y)}[L(f(X), Y)]$. A priori, one might expect that the task of joint risk minimization differs meaningfully from that of conditional risk minimization, as joint risk minimization explicitly involves the distribution of X while conditional risk minimization does not. However, due to the law of total expectation, it turns out that a function f minimizes the joint risk if and only if it minimizes the conditional risk almost everywhere (where "almost everywhere" is taken with respect to the probability measure of X). In other words, joint risk minimization is just conditional risk minimization where we only have to minimize conditional risk at "most values of X" rather than "every value of X". Since joint risk minimization is essentially just conditional risk minimization (and is therefore, somewhat surprisingly, almost completely independent of the distribution of X), we use the term *risk minimizer* to refer unambiguously to the solutions of both conditional and joint risk minimization.

Furthermore, despite how restrictive risk minimization appears to be, nearly any function of interest related to (X, Y) can be reformulated as a risk minimizer under an appropriate choice of loss function. For example, the conditional mean and median functions can be recast as the risk minimizers under squared and absolute loss, respectively. Similarly, if $Y = g(X)$ (i.e. Y is a function of X), then g can be equivalently thought of as the risk minimizer under any loss function (provided the loss function satisfies our assumption that $L(y_1, y_2) = 0$ if and only if $y_1 = y_2$ for all $y_1, y_2 \in \mathcal{Y}$). Thus, while risk minimization may seem niche, in fact nearly any relevant function can be conceived of as an "optimal" prediction rule under a suitable loss function. In particular, this implies that just about any function estimation problem in statistics can be rephrased in the language of risk minimizer estimation (though it may not necessarily be the best way of viewing the problem).

So far in this discussion, the framework for machine learning has been no different from that of prediction. Nonetheless, there is a key difference between them; the classical prediction setting assumes the joint distribution (X, Y) is known, whereas in machine learning contexts (X, Y) is unknown. By assuming the form of (X, Y), the task of determining a risk minimizer f reduces to solving an explicit optimization problem. In contrast, when the form of (X, Y) is left unknown, f is made completely indeterminate from available information. In such circumstances, a statistical approach must be employed, whereby data is collected in order to estimate f. Thus, what separates prediction and machine learning is the fact that prediction assumes (X, Y) and thus solves a specific optimization problem to determine f (though, whether or not the optimization problem can actually be explicitly solved is another matter) while machine learning leaves (X, Y) unspecified and thus must employ the usage of inferential statistics to estimate f. Put another way, prediction answers the question "how we should use X values to predict Y values" while machine learning answers the more open-ended question "from the available data, what's our best guess on how we should use X values to predict Y values".

Due to its reliance on data, machine learning comes in a few different flavors according to what kind of data is considered. The most common setting is *supervised learning*, which refers to when the available data is *labeled*, meaning it is of the form $\{(x_i, y_i)\}_{i=1}^n$ (i.e. our sample consists of pairs of inputs and associated outputs). In contrast, *unsupervised learning* refers to when the data is *unlabeled*, meaning it looks like $\{x_i\}_{i=1}^n$ (i.e. our sample consists purely of inputs). The general setting, where some combination of labeled and unlabeled data is produced, is refered to as *semi-supervised learning*. These different data types imply different statistical models, which in turn place different restrictions on how well we are able to estimate risk minimizers; for example, the lack of Y observations in unsupervised learning typically implies we ought to be less ambitious in our estimation goals than in the context of supervised learning, due to problems like identifiability becoming more of an issue. This

leads to each flavor of learning having their own typical aims and techniques; however, since the computation of prime factorizations using neural networks will ultimately be expressed as a supervised learning problem, for this rest of this paper we restrict our focus to supervised learning.

Considering the supervised learning context in more detail, let $(X, Y) \sim P$ (i.e. (X, Y) are random variables with joint distribution P), where $P \in \mathcal{P}(\mathcal{X} \times \mathcal{Y})$ is fixed but unknown (here, $\mathcal{P}(\mathcal{X} \times \mathcal{Y})$ denotes the set of probability measures over $\mathcal{X} \times \mathcal{Y}$). Let $f : \mathcal{X} \to \mathcal{Y}$ be the risk minimizer associated to P under some chosen loss function L, and let $\mathcal{F}$ denote the set of risk minimizers derived from distributions in $\mathcal{P}(\mathcal{X} \times \mathcal{Y})$ (i.e. $\mathcal{F}$ is the space f lives in, so $f \in \mathcal{F}$). Then the statistical model for our data D in the supervised setting is simply $D = \{(X_i, Y_i)\}_{i=1}^n \sim_{iid} P$, and our goal is to construct an estimator $\hat{f} : (\mathcal{X} \times \mathcal{Y})^n \to \mathcal{F}$ of f which performs well in the sense that $\hat{f}$ hopefully spits out a function which has a close-to-minimal risk. More specifically, we hope that $\hat{f}$ calculated on a given observation has low *excess risk*, meaning

$$\mathcal{E}(\hat{f}, f) := R(\hat{f}) - R(f)$$

is small, where $R(\hat{f}), R(f)$ are the joint risks of $\hat{f}, f$ under the true distribution P. Phrased in formal statistical language, excess risk serves as the loss function for our estimation problem (in contrast to L, which serves as the loss function for our underlying prediction problem), and our (frequentist) goal is for $\hat{f}$ to be a *minimax* estimator for f, i.e. achieve the smallest possible worst-case expected estimation loss (though Bayesian methods are appropriate too).

So, our goal is to construct an estimator $\hat{f}$ of f, but how should we do that? Due to the great generality of our parameter space, we are operating within a nonparametric context and thus common estimation techniques used in parametric situations (like, say, Maximum Likelihood Estimation) no longer perform well. Thus, we must rely on general principles of nonparametric estimation to construct our estimator. Perhaps the most common ad-hoc

technique for constructing an estimator in nonparametric statistics is the *plug-in principle* which roughly says "find some property or purpose which characterizes the parameter, then find something which satisfies an analogous property over the data and use it as your estimate". Since our goal is to estimate f, and f's defining feature is that it minimizes risk, according to the plug-in principle a reasonable estimate $\hat{f}$ will be a function which minimizes an *estimate of the risk* (instead of the true risk, which is unknown to us since we don't know P); in other words,

$$\hat{f} := \arg\min_{h \in \mathcal{H}} \hat{R}(h),$$

where $\hat{R}(h)$ is an estimator of the true risk of h and $\mathcal{H}$ is an appropriate space of functions from which we select our estimate. We will refer to this paradigm for estimation as *estimated risk minimization*, while the action of solving the associated optimization problem to determine the estimate $\hat{f}$ is referred to as the *training/fitting/learning process.*

There are a few things to note about the estimated risk minimization process. First, it requires the specification of a risk estimator $\hat{R}$. The most commonly used risk estimator is *empirical risk*, i.e.

$$\hat{R}_{\text{empirical risk}}(f) := \frac{\sum_{i=1}^{n} L(f(x_i), y_i)}{n};$$

when empirical risk is used as the chosen risk estimator, we refer to the associated estimation technique as *empirical risk minimization.* Note that empirical risk can be naturally derived via the plug-in principle, by noting that the data analog of expected loss under the true joint distribution of (X, Y) is simply the expected loss under the *empirical distribution* of the data. Furthermore, most risk estimators are in fact some combination of empirical risk with an appropriate notion of regularization, such as the inclusion of a penalty term or the use of a surrogate loss function $L_{\text{surrogate}}$ in place of L. Hence, due to its role as the "workhorse" of machine learning, empirical risk serves as our choice of risk estimator in this paper.

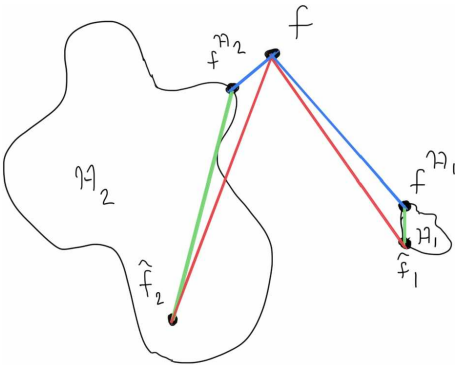

Figure 2.1: A visualization of the decomposition of excess risks $\mathcal{E}(\hat{f}_1, f), \mathcal{E}(\hat{f}_2, f)$ for two estimates $\hat{f}_1$, $\hat{f}_2$ which use hypothesis spaces $\mathcal{H}_1$, $\mathcal{H}_2$ respectively. Red denotes excess risk, green denotes estimation error, while blue denotes approximation error. Notice how the estimate in the smaller hypothesis space $\mathcal{H}_1$ exhibits small estimation error with large approximation error, while the estimate in the larger hypothesis space $\mathcal{H}_2$ exhibits large estimation error with small approximation error, but ultimately $\hat{f}_1$ exhibits lower excess risk than $\hat{f}_2$.

Secondly, estimated risk minimization requires a choice of function space $\mathcal{H}$ (also known as the *hypothesis space*) for our estimator $\hat{f}$ to take values in; for example, $\mathcal{H}$ could be the set of GLMs under some choice of link function and linear predictor. To understand the importance of $\mathcal{H}$, let $f^{\mathcal{H}}$ denote a function within $\mathcal{H}$ which achieves the smallest possible risk within $\mathcal{H}$, and note that the excess risk can be decomposed as

$$\mathcal{E}(\hat{f}, f) = \underbrace{\left[R(\hat{f}) - R(f^{\mathcal{H}})\right]}_{\text{estimation error}} + \underbrace{\left[R(f^{\mathcal{H}}) - R(f)\right]}_{\text{approximation error}}.$$

The *estimation error* describes how close our specific guess actually comes to attaining the smallest possible risk within $\mathcal{H}$, and thus depends on factors dictating how well we can estimate $f^{\mathcal{H}}$ (i.e. how much data we have, how good our estimate of the risk is, how well we can optimize over $\mathcal{H}$, and so on). The *approximation error* describes how close we can theoretically come to attaining minimal risk using only functions in $\mathcal{H}$ (and therefore only depends on the choice of $\mathcal{H}$). Thus, excess risk depends on both how well $\hat{f}$ estimates $f^{\mathcal{H}}$ and how well $f^{\mathcal{H}}$ approximates the actual risk minimizer f (see Figure 2.1). Moreover, as it turns out, there is generally a trade-off between estimation error and approximation error.

The key to understanding the trade-off is the fact that in general $\hat{R}$ will differ from R, as $\hat{R}$ is only an estimate of R. This discrepancy between estimated risk and true risk leads to the possibility of functions existing in $\mathcal{H}$ which "fool" the risk estimator, in the sense that they have a low estimated risk but actually have a large true risk (this phenomenon is also known as *overfitting*). When such functions exist in $\mathcal{H}$, $\hat{f}$ will typically not estimate $f^{\mathcal{H}}$ very well, leading to a large estimation error; this occurs due to the fact that while $f^{\mathcal{H}}$ will oftentimes have a small risk estimate, the "fooling" functions will nonetheless exist more numerously in $\mathcal{H}$ and can typically attain smaller risk estimates (like, say, a risk estimate of 0, i.e. a perfect fit to the data), thus making it more likely that $\hat{f}$ (which is chosen based on minimizing the risk estimate) ends up an overfitting function rather than something resembling $f^{\mathcal{H}}$.

When $\mathcal{H}$ is small, the risk of including "fooling" functions into $\mathcal{H}$ is lessened but we also run the risk of not including enough functions to allow for $f^{\mathcal{H}}$ to well-approximate f (this is known as *underfitting*); thus, estimation error is typically low while approximation error is typically high. On the other hand, when $\mathcal{H}$ is large, the potential for including "fooling" functions into $\mathcal{H}$ is higher but we now have a better chance of including enough functions to allow $f^{\mathcal{H}}$ to approximate f; thus, estimation error is typically high while approximation error is typically low. Hence, the goal when using the estimated risk minimization paradigm is to find a hypothesis space $\mathcal{H}$ and risk estimator $\hat{R}$ which optimally balance the trade-off between estimation and approximation error; specifically, we want $\mathcal{H}$ to be large enough to include functions so that $f^{\mathcal{H}}$ approximates f, while simultaneously being small enough so that $\hat{f}$ provides a good estimate of $f^{\mathcal{H}}$ (see Figure 2.1).

Once a risk estimator and hypothesis space are chosen, a risk minimizer estimator is defined; however, there are many different risk estimators and hypothesis spaces that could

be chosen, and thus many different risk minimizer estimators that could be defined. So, how do we choose our estimator? Ideally, we would construct an estimator which is approximately minimax, but it can be difficult to explicitly construct such an estimator. Hence, we resort to informed guessing, where we construct a set of potential estimators $\{\hat{f}_1, ..., \hat{f}_n\}$ by trying various promising combinations of risk estimators and hypothesis spaces in the hopes that at least one of the $\hat{f}_i$'s is suitable. Unfortunately, due to how general the estimation problem is, most of the $\hat{f}_i$'s will not perform anywhere near minimax; nonetheless, we can employ an *ensemble method*, which takes the set of potential estimators and combines them in some fashion to build a more complex, but hopefully improved, estimator. In doing so, ensemble methods thus provide a general technique for successively improving estimators.

Of the ensemble methods, perhaps the most fundamental is *model selection*, which at every observation in the sample space considers the guesses made by each potential estimator in $\{\hat{f}_1, ..., \hat{f}_n\}$ and selects the guess which results in the smallest estimated risk. In other words, model selection takes a set of potential estimators $\{\hat{f}_1, ..., \hat{f}_n\}$ and constructs a new estimator $\hat{f}_{\text{model selection}}$ defined by

$$\hat{f}_{\text{model selection}}(d) := \arg\min_{\hat{f}_i(d) \in \{\hat{f}_1(d), ..., \hat{f}_n(d)\}} \hat{R}(\hat{f}_i(d)),$$

where d is an arbitrary observation in the sample space and $\hat{R}$ is a risk estimator (which is not necessarily the same risk estimator used to define the $\hat{f}_i$'s, as will be discussed later).

The intuition behind model selection is that given a set of potential estimators $\{\hat{f}_1, ..., \hat{f}_n\}$, the obvious way to construct an improved estimator would be to consider at each observation d the guesses $\{\hat{f}_1(d), ..., \hat{f}_n(d)\}$ made by each of the potential estimators and choose the guess which has the lowest excess risk, i.e. at each observation d, choose $\hat{f}_i(d)$ which minimizes $\mathcal{E}(\hat{f}_i(d), f) = R(\hat{f}_i(d)) - R(f)$ (see Figure 2.2). Unfortunately, since f is unknown, there is

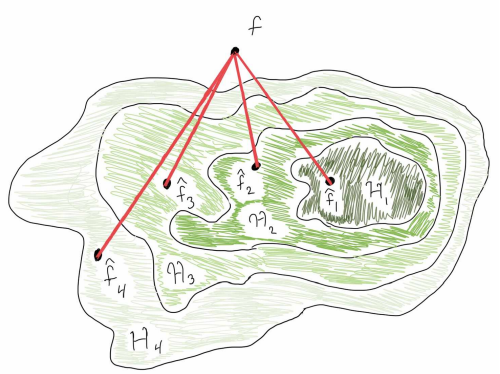

Figure 2.2: A partial visualization of the model selection process involving four potential estimates $\hat{f}_1$, $\hat{f}_2$, $\hat{f}_3$, $\hat{f}_4$ which use (nested, for simplicity) hypothesis spaces $\mathcal{H}_1$, $\mathcal{H}_2$, $\mathcal{H}_3$, $\mathcal{H}_4$ respectively. Red denotes excess risk, while the various shades of green distinguish between hypothesis spaces. Notice that $\hat{f}_2$ produces the lowest excess risk, since $\mathcal{H}_2$ provides the optimal estimation-approximation error trade-off of the hypothesis spaces considered, and would ideally be the estimate selected during model selection. Not pictured is the fact that the true excess risk is not known to us and is therefore not available for comparisons; instead, we must estimate which of the $\hat{f}_1$, $\hat{f}_2$, $\hat{f}_3$, $\hat{f}_4$ provides the lowest excess risk using in-sample or out-of-sample methods.

no way to determine or estimate $\mathcal{E}(\hat{f}_i(d), f)$. However, it is easy to verify that minimizing $\mathcal{E}(\hat{f}_i(d), f)$ is equivalent to minimizing $R(\hat{f}_i(d))$, and $R(\hat{f}_i(d))$ is actually something we can estimate. Thus, while we can't estimate the excess risks themselves, we can estimate the ordering of the excess risks which therefore allows us to estimate which of the potential risk minimizer estimators performs the best at each observation. By choosing the guess which is estimated to perform the best at every observation, the model selection estimator hopefully improves upon the set of potential estimators (though whether or not it actually succeeds in doing so depends crucially on how well we can estimate $R(\hat{f})$).

Note that there is a subtle problem with estimating $R(\hat{f})$: since $\hat{f}$ itself is chosen specifically to minimize the risk estimator $\hat{R}$, it follows that $\hat{f}$ and $\hat{R}$ are not independent of one another, and this dependency results in $\hat{R}(\hat{f})$ generally producing overly optimistic estimates of $R(\hat{f})$. Thus, we can't just use $\hat{R}(\hat{f})$ as our risk estimator for model selection and must consider an alternative. There are two common ways to account for this: in-sample and out-of-sample adjustments.

In-sample adjustments proceed by replacing $\hat{R}(\hat{f})$ with $\hat{R}_{\text{modified}}(\hat{f})$, where $\hat{R}_{\text{modified}}$ is a modification of the risk estimator which incorporates some kind of regularization (oftentimes in the form penalty which punishes complex models) intended to improve the estimation of $R(\hat{f})$. Common examples of in-sample adjustment include AIC, BIC, and Mallow's C_p. In contrast, *out-of-sample adjustments* proceed by calculating $\hat{f}$ on a subset of the available data known as a *training set*, and then calculate $\hat{R}(\hat{f})$ on a subset of data unused for training known as a *testing set*. By using an independent data set to calculate $\hat{R}(\hat{f})$, the problematic dependency between $\hat{f}$ and $\hat{R}$ is severed, allowing better estimation of $R(\hat{f})$. Common examples of out-of-sample adjustments include all the various cross-validation methods and bootstrap corrections. Ultimately, while much more could be said about this topic, the point is that whether you ought to take an in-sample or out-of-sample approach is itself a complicated statistical problem which depends on what kind of data is available to you, how much data is available, and other aspects of the estimation problem.

In summary, machine learning is about using data to estimate risk minimizers. To construct such an estimator, the estimated risk minimization paradigm is usually employed. This process essentially involves two choices: a choice of function space $\mathcal{H}$ for the estimator to take values in, and a choice of risk estimator $\hat{R}$ to use as a criterion to select a function from $\mathcal{H}$. The goal is to choose $\mathcal{H}$ and $\hat{R}$ such that we achieve an optimal trade-off between estimation and approximation error and thus avoid overfitting and underfitting. Due to the difficulty of this task, the typical strategy proceeds by constructing a set of potential estimators and then combining them according to some ensemble method to build an improved estimator. For more detailed but less statistically-explicit resources on these ideas, see *Hastie et al.* [2009] or *Shalev-Shwartz and Ben-David* [2014]. For reasons we now discuss, one particularly popular choice of hypothesis space is to have $\mathcal{H}$ be an appropriate class of neural nets.

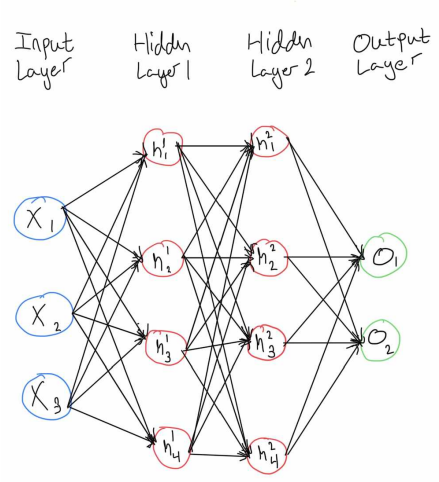

Figure 2.3: The composition diagram of a neural net with an input layer consisting of 3 nodes, two hidden layers each consisting of 4 nodes, and an output layer consisting of 2 nodes. Blue denotes input nodes, red denotes hidden nodes, and green denotes output nodes. The diagram flows from left to right, as the input layer is fed into the first hidden layer, which in turn is fed into the second hidden layer, and so on. Notice that the input layer consists of 3 nodes, indicating 3 input variables, while the output layer consists of 2 nodes, indicating 2 output variables. Due to the diagram's lack of cycles and the connectedness between nodes, this is also an example of a fully connected feedforward neural net.

2.2 Neural Networks

Named for their resemblance to networks of biological neurons, *neural nets* are, roughly speaking, functions formed through the repeated composition of many constituent functions (which are known as *neurons* in the neural net literature). Due to their compositional structure, neural nets are best expressed through their function composition diagram, which captures (in convenient graph-theoretic language) the *network architecture*, i.e. how the neural net composes its various component neurons together (see Figure 2.3). In more detail, the *diagram* of a neural net is a directed graph whose *nodes* represent the outputs of individual neurons and whose *directed edges* indicate function composition (where the order of composition is specified by the direction of the edge). The nodes can be subdivided into three categories: *input nodes*, which represent the values of input variables, *hidden nodes*, which represent the outputs of neurons part-way through the family of compositions, and *output nodes*, which represent the outputs of neurons at the end of the function compositions

(i.e. the values of output variables).

Oftentimes, the diagram of a neural net can be organized into *layers* (see Figure 2.3), which roughly correspond to sets of neurons that all operate at the same level of the function composition hierarchy. In particular, we have an *input layer* (consisting of all input nodes), some number of *hidden layers* (each consisting of hidden nodes), and an *output layer* (consisting of all output nodes). The number of hidden layers a neural net possesses is referred to as the *depth* of the network, while the number of nodes within a layer is referred to as the *width* of the layer. While one could in principle have varying widths across the hidden layers, we make the common simplifying assumption that each hidden layer is of the same fixed width.

It is often said that while every function has an associated composition diagram equipped with an input layer and an output layer, what characterizes neural nets is the presence of hidden layers. But that is not exactly correct, as there are many functions which are not neural nets (or at least, not in a meaningful sense) which nonetheless can be thought of as having hidden layers. This is because the hidden layers in a composition diagram simply represent a transformation or pre-processing of the inputs into something more suitable to be fed into the output layer, and since many functions besides neural nets invoke transformations of the inputs, it follows that many functions have hidden layers. For example, the composition diagram of any GLM $g(\eta(\mathbf{x}))$ with *activation function* (i.e. inverse link function) g and linear predictor $\eta(\mathbf{x}) = \sum_{i=1}^{k} \beta_i f_i(\mathbf{x})$ can be realized with a hidden layer whose k nodes are given by the $f_i(\mathbf{x})$'s, even though GLMs are typically considered neural nets only in a trivial/degenerate sense. What characterizes neural nets, then, is not just the presence of hidden layers, but rather the fact that within a class of neural nets the exact form of the hidden layers is allowed to vary from net to net. In other words, while many classes of functions take the form of a fixed transformation of inputs followed by a parameterized

output layer (i.e. $g_\beta(f(\mathbf{x}))$), classes of neural nets provide a generalization by taking the form of a parameterized transformation of inputs followed by a parameterized output layer (i.e. $g_\beta(f_\theta(\mathbf{x}))$).

Since neural nets consider both the input transformation and output layer as parameters, when neural nets are fit to data they jointly learn how to transform the inputs for better use in the output layer and how the output layer should optimally act on the transformed inputs. It is this ability to automatically learn a transformation of the input data to help achieve the task at hand which defines the purpose of neural nets, and thus the reason why they are used in practice (including in unsupervised learning, where they are used as a sort of generalized "non-linear" PCA; for more details see *Kramer* [1991]). Sometimes, this defining attribute of neural nets is referred to as *automatic feature extraction*, since the learned input transformation can be thought of as "extracting features" of the inputs which are useful for use in the output layer. However, it is important to note that the "features" referred to in this context do not refer to human-interpretable features of the data, like say the shapes our eyes can detect in an image. Instead, they refer to semi-arbitrary numerical features intended to make things work out; this is analogous to how, say, PCA extracts the principal components of a data set, and while the principal components are useful features of the data, they oftentimes don't have any particularly intuitive or interpretable meaning. Consequently, in the process of designing a neural net architecture, the goal is not to extract specific features of the data at each hidden layer (as it is sometimes described in popular descriptions), but rather to ensure that the structure of the hidden layers allows an efficient and effective transformation of the inputs to be learned.

Designing a neural net, then, boils down to designing the hidden layers (as the output layers are usually specified to be something very simple, like a layer of GLMs), which in turn is equivalent to designing the input pre-processing transformation with the goal of ensuring

(a) A recurrent neural net. (b) A residual neural net.

Figure 2.4: (a) The composition diagram of a recurrent neural net with 3 recurrent hidden layers, where recurrent connections are denoted by loops, and (b) the composition diagram of a residual neural net with 3 hidden layers, where skip-connections are denoted by dashed lines and residual connections are denoted by squiggly lines. In both figures, blue denotes the input layer, red denotes hidden layers, and green denotes the output layer; nodes within layers are excluded for simplicity.

that a useful transformation of the input data is readily learnable. Of course, since there is an immense number of ways to structure the input transformation, there are thus many different flavors of neural nets, all differing from one another based on the structure of their hidden layers. These structural differences are intended to either make the learned input transformation perform better or make the process of learning a suitable input transformation more efficient for a certain application. Moreover, different choices of hidden layer structure can result in significantly different learned input transformations and approaches to training.

Some neural net variants employ specially-designed constructions in the hidden layers to help take advantage of or preserve important structures associated with a specific data type, and in doing so either improve performance or reduce the computational load. For example, *recurrent neural nets* (*Hochreiter and Schmidhuber* [1997]) employ cycles in their composition diagram in order to incorporate a kind of "temporal memory" to better deal with the dependencies in time-series data (see Figure 2.4a). Likewise, *convolutional neural nets* (*Lecun et al.* [1998]) implement convolutions instead of matrix multiplications to take advantage of the local correlations between pixels in an image as a means of reducing the number of parameters defining the neural net.

Furthermore, many neural nets utilize constructions which are intended to stabilize, speed-up, simplify, or otherwise improve the training process. For example, *residual neural nets* (*He et al.* [2016]) employ the use of *skip-connections*, which feed a layer into the following layer untouched through the identity map, alongside *residual connections*, which feed a layer into the next layer altered through a non-trivial mapping (see Figure 2.4b). As a consequence, the output at each layer is the output from the previous layer (due to the skip connection) coupled with a light modification of the previous layer (due to the residual connection). This allows residual neural nets to express successive layers as incremental modifications of previous layers, thereby reducing the extremity of the change exhibited from layer to layer which in turn simplifies the training process. Similarly, other neural nets employ *batch normalization layers* (*Ioffe and Szegedy* [2015]) to normalize some of their hidden layers. This forces the outputs of each node within the normalized layer to be on the same scale, which makes the optimization involved in training easier to perform (this is actually more or less the same reasoning as to why inputs are oftentimes normalized, too).

In this paper, the network architecture considered is that of neural nets which are both *fully connected*, i.e. in every layer of the composition diagram each node has an edge connecting it to each node in the following layer, and *feedforward*, meaning their diagram includes no cycles (see Figure 2.3). Specifically, we are concerned with a subclass of fully connected feedforward neural nets known as *multilayer perceptrons* (or MLPs), which satisfy the additional property of having at least one hidden layer and whose hidden and output neurons are all GLMs (except the activation functions are not required to be invertible or smooth, as they typically are in GLM theory). While one could allow varying activation functions from hidden node to hidden node in an MLP, we make the simplifying assumption that each hidden node utilizes the same activation function g_{hidden}. Similarly, since the output nodes in our context will each be the same "type" of variable, we also assume that each output node utilizes the same activation function g_{output}.

As this is all rather abstract, for concreteness we now describe in detail the explicit form of an MLP (referred to as nn) with an input layer consisting of $\mathcal{I}$ nodes, $\mathcal{D}$ hidden layers of width $\mathcal{W}$, and an output layer consisting of $\mathcal{O}$ nodes. First, the nodes $\{h_n^1\}_{n=1}^{\mathcal{W}}$ of the 1st hidden layer H^1 each take the form

$$h_n^1 := g_{\text{hidden}}(\sum_{i=1}^{\mathcal{I}} w_{ni}^1 x_i)$$

for some activation function g_{hidden}, where $\{x_i\}_{i=1}^{\mathcal{I}}$ is the set of input variables. Using vectorization notation (and using transposes to save page space), we can compactly represent the 1st hidden layer as

$$H^1 := \begin{bmatrix} h_1^1 & h_2^1 & \dots & h_{\mathcal{W}}^1 \end{bmatrix}^T = g_{\text{hidden}}\left(W^1 X\right),$$

where $W^1 := [w_{ni}^1]$ is the $\mathcal{W} \times \mathcal{I}$ matrix of weights and $X := \begin{bmatrix} x_1 & x_2 & \dots & x_{\mathcal{I}} \end{bmatrix}^T$ is the $\mathcal{I} \times 1$ column vector of input variables (here, g_{hidden} acting on a column vector is understood to mean g_{hidden} is applied to each entry of the column vector). More generally, the lth hidden layer H^l is given by

$$H^l = g_{\text{hidden}}(W^l H^{l-1}),$$

where W^l is a $\mathcal{W} \times \mathcal{W}$ matrix of weights. Similarly, the output layer O is given by

$$O = g_{\text{output}}(W^O H^{\mathcal{D}})$$

for some activation function g_{output}, where W^O is a $\mathcal{O} \times \mathcal{W}$ matrix of weights and $H^{\mathcal{D}}$ is the $\mathcal{W} \times 1$ column vector representing the final hidden layer. Thus, in full compositional glory, the MLP nn is defined by

$$\text{nn}(X) := g_{\text{output}}(W^O g_{\text{hidden}}(W^{\mathcal{D}} g_{\text{hidden}}(\dots g_{\text{hidden}}(W^1 X)\dots))).$$

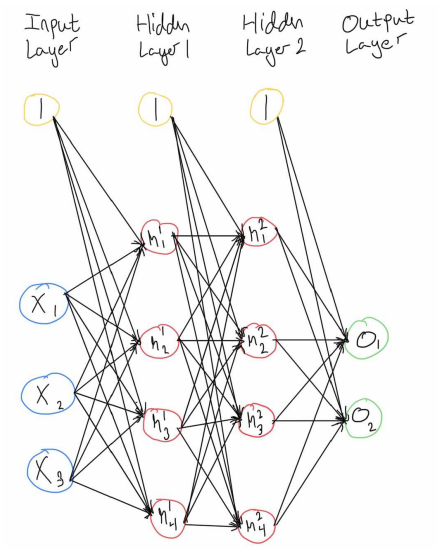

Figure 2.5: The composition diagram of a multilayer perceptron including bias nodes with an input layer consisting of 3 nodes, two hidden layers each consisting of 4 nodes, and an output layer consisting of 2 nodes. Blue denotes input nodes, red denotes hidden nodes, yellow denotes bias nodes, and green denotes output nodes. Notice that bias nodes can be visualized as nodes which take the value 1 constantly. While edges from the previous layer's nodes to a bias node could be drawn, ultimately these edges would be superfluous as the bias node is a constant and thus does not depend on previous layers meaningfully. Moreover, while bias nodes are found in the input and hidden layers, we consider them distinct from input and hidden nodes; in particular, aspects of the network architecture like layer width or connectedness do not apply to bias nodes.

There is a minor caveat to the picture we have thus far painted, which we must now discuss: the presence of bias nodes. Roughly speaking, a *bias node* is an extra node attached to the input layer and/or a hidden layer of an MLP (see Figure 2.5), whose purpose is to introduce an "intercept" or "shift" term to the linear predictor of each GLM in the next layer. Put plainly, in our previous discussion the lth hidden layer was given by

$$H^l = g_{\text{hidden}}(W^l H^{l-1}),$$

but if a bias node is present in the $l-1$th hidden layer then we instead have that

$$H^l = g_{\text{hidden}}(B^l + W^l H^{l-1}),$$

where B^l is the $\mathcal{W} \times 1$ column vector of biases. Thus, the effect of a bias node in layer $l-1$ is that the linear transformation $W^l H^{l-1}$ is made into an affine transformation $B^l + W^l H^{l-1}$; hence, the presence of a bias node is useful for the same reason an intercept term in a linear regression is useful.

As is, the presence of bias nodes leads to some vagueness on what the architecture of an MLP refers to – for example, does width refer to the number of nodes including the bias node, or without the bias node? In this paper, we take the convention that bias nodes, despite belonging to the input or hidden layers, are considered distinct from input or hidden nodes (since bias nodes play different roles and exhibit behavior functionally different from the input and hidden nodes). Hence, when we specify the architecture of an MLP, we are really specifying the core "non-bias" part of the MLP; in particular, the width of a hidden layer refers to the number of hidden nodes and thus does not account for the bias node. Moreover, we implicitly assume the presence of a bias node in the input layer and every hidden layer. So, for example, an MLP with a hidden layer of width 4 really means that it has 4 hidden nodes, with the understanding that there is also 1 bias node lurking in the layer too.

Note that while the nodes of the hidden layers and output layer of an MLP are all GLMs, the hidden layer activation function g_{hidden} is almost always distinct from the output layer activation function g_{output}. This is due to the previously discussed fact that the hidden layers and the output layer serve qualitatively different purposes; as a consequence, the hidden layer activation functions are chosen to help facilitate the learning of a suitable input transformation, while the output layer activation functions are chosen to ensure that the outputs are always in an appropriate range. Furthermore, it should be noted that the purpose of the activation functions within the hidden layers of an MLP is to provide non-linearity; without them, the hidden layers are just compositions of linear functions and are therefore linear, but linear data transformations are typically not flexible enough to capture interesting behavior.

Due to their simplicity, multilayer perceptrons often serve as a "baseline" or "vanilla" genre of neural nets to test out before increasing the complexity of the situation by adding bells and whistles to the hidden layers, and it is for this reason that this paper focuses on them. While the exact specifications of our architecture will be discussed in the next chapter, the key thing to note is that we will be implementing *deep* MLPs, meaning the neural net has at least a few hidden layers (though oftentimes they have far more than "a few"). By stacking multiple hidden layers together, the input transformation to be learned takes the form of a repeated composition of functions; for example, an MLP with multiple hidden layers corresponds to expressing the input transformation as a bunch of vector-valued GLMs composed one after another. Choosing to structure the input transformation as a sequence of function compositions is an approach referred to as the *deep learning paradigm* (*Goodfellow et al.* [2016]), and though it has seen wild success across a wide range of applications in recent years, the use of repeated composition is admittedly unintuitive at first blush.

Although repeated function composition may appear unnatural, it's actually not so far-fetched. First, since composition is one of the fundamental operations defined between functions (alongside pointwise addition and multiplication, for real-valued functions at least), it's at least reasonable (albeit uncommon in typical models) to add complexity to a function by repeatedly composing functions with it, analogous to how one might add or multiply more and more functions together to accomplish an increase in complexity. Beyond simply being a reasonable thing to try, though, repeated function composition is perhaps best thought of as a *discrete dynamic system* (*Chen et al.* [2018]), where the input values are thought of as initial conditions and the repeated composition serves as the "flow" or "evolution" of the system. This may seem unusual, as dynamic systems are typically an object of study for their own purposes rather than as means of accomplishing a specific task, but in fact this understanding is wrong; to see otherwise, note that every iterative algorithm for solving a

problem (like, say, the gradient descent technique for optimization) can be thought of as a discrete dynamic system which takes an initial point and (hopefully) converges to a solution.

Thus, dynamic systems serve as a surprisingly natural tool for evolving raw input into something more in-line with a given goal, and this is the mindset deep learning takes. That is, when we employ deep neural networks, what we are doing is forcing the input transformation to take the form of a dynamic system. Hence, deep learning can be thought of as learning a dynamic system which optimally transforms our inputs for use in the output layer. While the benefits of this approach over other possible structurings of the hidden layers are not exactly known, the establishment of deep network *universal approximation theorems* (which roughly state that any Borel-measurable function can be approximated arbitrarily well by sufficiently deep networks; for more details see *Hornik et al.* [1989]) and the empirical success of deep networks in terms of both accuracy and speed imply that the approach is certainly capable theoretically and practically. Moreover, the task of learning a dynamic system to transform inputs can be studied from the perspective of dynamic systems and control theory, allowing a huge mathematical framework to potentially be applied in deep learning contexts to help us more effectively learn a dynamic system for a given task.

Chapter 3 Set-Up

3.1 Prime Numbers Review

With a basic understanding of machine learning and neural nets established, we turn now to the problem considered in this paper: using neural networks to approximately compute prime factorizations. However, while the computation of prime factorizations is a familiar task to most, it's not immediately clear how exactly to use neural nets to perform these computations. Thus, we first need to express the task of prime factorization in the language of machine learning; that is, we need to rephrase prime factorization as a risk minimizing prediction function. Then, the goal of approximately computing prime factorizations can be re-interpreted as the estimation of a risk minimizer, allowing us to bring the tools of statistical learning discussed in Chapter 2 to bear on this problem.

Before diving into the task of formulating prime factorization as a risk minimization problem, we must go over a few facts about prime numbers. First, recall that a *prime number* is an integer greater than 1 whose only positive divisors are 1 and itself. Note that we exclude 1 as a prime, as otherwise prime factorizations are not unique (which would defeat the purpose of computing prime factorizations in many applications). Since the prime factorization of a negative integer $-z$ is essentially the same as the prime factorization of the positive integer z, we can ignore negative integers; moreover, since 0 and 1 do not admit prime factorizations, we can also ignore 0 and 1. Hence, for the purposes of this paper we restrict our attention to computing the prime factorizations of integers greater than or equal to 2, i.e. integers belonging to the subset $\mathbb{Z}_{2\leq}$, as these are the integers with meaningful prime factorizations.

Furthermore, recall that by the Fundamental Theorem of Arithmetic, any integer $z \geq 2$ can be factorized into a unique (up to the order of multiplication) product of powers of prime

numbers. In other words,

$$z = \prod_{i=1}^{n} p_i^{k_i}$$

for some distinct primes $\{p_i\}_{i=1}^n$ and positive integers $n, \{k_i\}_{i=1}^n$, and this factorization is unique up to the order of multiplication. To move beyond the non-uniqueness caused by the commutativity of multiplication, we assume that the p_i's are *ordered sequentially*, i.e. $p_1 < p_2 < ... < p_n$; this forces the ith prime factor in the multiplication to be the ith prime factor in magnitude, thus specifying a canonical order to the multiplication.

Finally, note that we can extend the prime factorization $\prod_{i=1}^n p_i^{k_i}$ to a sequentially ordered infinite product over all prime numbers. Namely,

$$\prod_{i=1}^{n} p_i^{k_i} = \prod_{j=1}^{\infty} q_j^{h_j}$$

where q_j is the jth prime number (i.e. $q_1 = 2$, $q_2 = 3$, $q_3 = 5$, and so on), and the only q_j's which have a non-zero exponent h_j are the q_j's which correspond to prime factors p_i in the prime factorization. In other words, the infinite product representation is simply the original prime factorization multiplied by an infinite number of 1's. Moreover, since the infinite product is also sequentially ordered (i.e. $q_1 < q_2 < ...$), this infinite product representation is unique. Thus, we can think of the infinite product representation as a unique "embedding" of the prime factorization into a bigger factorization which involves every prime number.

3.2 Problem Formulation, Part 1

Now, with all the relevant prime factorization facts touched upon, consider the task of formulating prime factorization as a risk minimization problem. The first step of this process is to envision prime factorization as a map which takes integers as inputs and spits out prime factorizations as outputs. To do this, we need to find a way to encode prime factorizations

as "objects" themselves, so that our mapping actually has something to output. To be a genuine encoding, we require that the encoding method be *injective*, i.e. if two prime factorizations result in the same encoding then they are in fact the same factorization. This ensures that the encodings of distinct prime factorizations remain distinct, and thus no loss of distinguishing information occurs in the encoding process.

So, how exactly should we encode a prime factorization $\prod_{i=1}^{n} p_i^{k_i}$? One naive possibility is to represent the factorization by its sequence of powers, i.e.

$$\prod_{i=1}^{n} p_i^{k_i} \sim (k_1, k_2, ..., k_n),$$

where the ith entry is the power of the ith prime in the factorization. However, this representation is problematic because it is not injective. More specifically, the problem with this encoding method is that while the ith entry of the sequence representation is the power of the ith prime in the factorization, the primes $\{p_i\}_{i=1}^{n}$ which actually compose the factorization are not accounted for themselves. Thus, any prime factorization which involves the same sequence of prime powers will result in the same encoding, regardless of which primes were involved, leading to non-injectivity. For example, consider the integers 2 and 3. Then the prime factorizations are $2 = 2^1$ and $3 = 3^1$, so the sequence of powers associated to both prime factorizations is (1).

But, suppose instead that we extend the prime factorization $\prod_{i=1}^{n} p_i^{k_i}$ to its unique infinite product representation $\prod_{j=1}^{\infty} q_j^{h_j}$, and then represent the infinite product by its sequence of powers, i.e.

$$\prod_{j=1}^{\infty} q_j^{h_j} \sim (h_1, h_2, ...),$$

where the ith entry of the sequence is the power of the ith prime. Since the ith entry of this sequence is the power of the ith prime, rather than the power of the ith prime

in the factorization (as was the case under the naive encoding method), it follows that this encoding method implicitly takes into account which prime factors compose a factorization and is thus injective. Hence, we will use this method to encode prime factorizations, and we will refer to the infinite sequence representation associated to a prime factorization as the *prime factorization sequence*. As a concrete example, since the prime factorization of 8 is 2^3, the associated prime factorization sequence is simply $(3, 0, 0, 0, ...)$ (since the power of 2 which appears in the prime factorization is 3, the power of 3 which appears is 0, the power of 5 which appears is 0, and so on). Similarly, since the prime factorization of 6 is $2^1 3^1$, the corresponding prime factorization sequence is $(1, 1, 0, 0, ...)$.

With a method for encoding prime factorizations decided upon, we can now represent prime factorization as a function $p : \mathbb{Z}_{2\leq} \to (\mathbb{Z}_{0\leq})^\infty$, which we shall refer to as the *prime factorization map*. The prime factorization map takes an integer $z \geq 2$ and spits out the associated prime factorization sequence $p(z)$; hence, the task of computing prime factorizations can be equivalently thought of as determining the output of an integer under p. Now, we must consider the second step in our reformulation: recasting p as a risk minimizer.

3.3 Problem Formulation, Part 2

Recall that a risk minimizer for a joint distribution (X, Y) is a function $f : \mathcal{X} \to \mathcal{Y}$ which minimizes the joint risk $E_{(X,Y)}[L(f(X), Y)]$ (or equivalently conditional risk, but for our purposes we will focus on joint risk). To recast p as a risk minimizer, we therefore need to find a joint distribution (Z, S) over $\mathbb{Z}_{2\leq} \times (\mathbb{Z}_{0\leq})^\infty$ whose joint risk minimizer under some loss function L corresponds to p. To do this, we will take advantage of an observation made in Chapter 2 which noted that if $Y = g(X)$, then g can be thought of as the risk minimizer of (X, Y) under any choice of loss function (with the understanding that loss functions must satisfy the $L(y_1, y_2) = 0$ if and only if $y_1 = y_2$ condition). For simplicity, we state and prove the result as a lemma.

Lemma. *Let Z be a random variable taking values in $\mathbb{Z}_{2\leq}$ and let $S := p(Z)$, where p is the prime factorization map (i.e. S is the prime factorization sequence of the random integer Z). Then p is the joint risk minimizer for the joint distribution (Z, S), where the distribution of Z and loss function L are arbitrary.*

Proof. Let Z be distributed according to some distribution over $\mathbb{Z}_{2\leq}$. Since $S := p(Z)$, it follows that the joint distribution of (Z, S) is $(Z, p(Z))$. In particular, we have that

$$S \mid Z = z \sim \delta_{p(z)},$$

where $\delta_{p(z)}$ denotes the Dirac-delta distribution centered on the value $p(z)$ (i.e. a distribution where probability 1 is given to the value $p(z)$).

Now, pick some loss function L and recall from Chapter 2 that joint risk minimization is equivalent to minimizing conditional risk almost everywhere with respect to the distribution of Z. Hence, to determine the joint risk minimizer of (Z, S) under L, it is sufficient to determine the conditional risk minimizer. That is, for each $z \in \mathbb{Z}_{2\leq}$ we need to determine the value c_z which minimizes $E[L(c_z, S) \mid Z = z]$.

Since $S \mid Z = z \sim \delta_{p(z)}$, it follows that $L(c_z, S) \mid Z = z \sim \delta_{L(c_z, p(z))}$, and so

$$E[L(c_z, S) \mid Z = z] = L(c_z, p(z)).$$

Hence, at each $z \in \mathbb{Z}_{2\leq}$ the optimization problem reduces to finding c_z which minimizes $L(c_z, p(z))$. But then, since we require that the loss function L satisfies $L(y_1, y_2) = 0$ if and only if $y_1 = y_2$, it follows that $c_z = p(z)$, as this is the unique value which results in 0 (and therefore minimal) loss. Thus, p is the joint risk minimizer of the joint distribution $(Z, p(Z))$. Moreover, since the distribution of Z and loss function L were arbitrary, the result holds for any choice of distribution Z and loss function L. □

While we could leave the distribution of Z unspecified (or equivalently, leave Z completely distributionless) since the joint risk minimizer only weakly depends on how Z is distributed, in order to simplify things we let

$$Z \sim \text{Uniform}(\mathbb{Z}_{2\leq z\leq k})$$

for some user-specified integer k. The primary reason for doing this is because later on in our experimental design, we will gather data $\{(Z_i, p(Z_i))\}_{i=1}^n$ by repeatedly sampling integers Z_i from $\mathbb{Z}_{2\leq}$ according to the distribution $Z_i \sim \text{Uniform}(\mathbb{Z}_{2\leq z\leq k})$ and then calculating their prime factorization sequences $p(Z_i)$. Since each $Z_i \sim \text{Uniform}(\mathbb{Z}_{2\leq z\leq k})$, we are limited in the information we can estimate about $(Z, p(Z))$ as we are only going to be gathering Z_i's over the subset $\mathbb{Z}_{2\leq z\leq k}$. Thus, by just assuming $Z \sim \text{Uniform}(\mathbb{Z}_{2\leq z\leq k})$ from the get-go, we explicitly acknowledge the inferential limitations of our experimental design and restrict our attention to what we can attempt to reasonably infer.

An important consequence of the assumption that $Z \sim \text{Uniform}(\mathbb{Z}_{2\leq z\leq k})$ is the fact that Z has a support over $\mathbb{Z}_{2\leq z\leq k}$. The effect this has on the joint risk minimizer is that it only has to equal the conditional risk minimizer over the set $\mathbb{Z}_{2\leq z\leq k}$ (as outside this set everything is measure 0 according to the distribution of Z and is thus irrelevant according to our discussion in Chapter 2). That is, the joint risk minimizer under this distributional assumption is really just

$$p_k := p\big|_{\mathbb{Z}_{2\leq z\leq k}},$$

i.e. the restriction of the prime factorization map p to the subset $\mathbb{Z}_{2\leq z\leq k}$. Hence, rather than estimating the entire prime factorization map p (which is more or less a lost cause, since we aren't assuming a ton of known structure about p and thus will not be able to extrapolate the form of p very well beyond the range of integers sampled) we instead only estimate p_k, which is a much more reasonable task.

Furthermore, note that the prime factorizations of integers over $\mathbb{Z}_{2\leq z\leq k}$ will only involve prime factors which are also in $\mathbb{Z}_{2\leq z\leq k}$, since an integer can't be divisible by something bigger than itself. In particular, the largest prime factor that can possibly divide an integer in $\mathbb{Z}_{2\leq z\leq k}$ is the largest prime in $\mathbb{Z}_{2\leq z\leq k}$. This implies that every prime factorization sequence of an integer in $\mathbb{Z}_{2\leq z\leq k}$ must have their last non-zero entry occur at or before the entry corresponding to the largest prime in $\mathbb{Z}_{2\leq z\leq k}$. In other words, the prime factorization sequences for integers in $\mathbb{Z}_{2\leq z\leq k}$ can be truncated to finite-length sequences whose lengths are given by the total number of primes in the subset $\mathbb{Z}_{2\leq z\leq k}$ without loss of information. In doing so, we avoid the need to use infinite length sequences, which can be difficult to deal with on a computer.

Thus, rather than treating p_k as a map $p_k : \mathbb{Z}_{2\leq z\leq k} \to (\mathbb{Z}_{0\leq})^{\infty}$, we can instead think of it as a map

$$p_k : \mathbb{Z}_{2\leq z\leq k} \to (\mathbb{Z}_{0\leq})^{\#\{\text{primes in } \mathbb{Z}_{2\leq z\leq k}\}}.$$

For concreteness, consider the integers in $\mathbb{Z}_{2\leq z\leq 10}$. Within $\mathbb{Z}_{2\leq z\leq 10}$, there are only 4 primes, with the largest prime being 7. So, the prime factorization sequences of integers in $\mathbb{Z}_{2\leq z\leq 10}$ can each be truncated to sequences of length 4. For instance, the prime factorization sequence of 2 can be thought of as $(1, 0, 0, 0)$ rather than $(1, 0, 0, 0, ...)$, while the prime factorization sequence of 7 can be thought of as $(0, 0, 0, 1)$ rather than $(0, 0, 0, 1, ...)$. Thus, our map $p_{10} : \mathbb{Z}_{2\leq z\leq 10} \to (\mathbb{Z}_{0\leq})^{\infty}$ can be reduced to $p_{10} : \mathbb{Z}_{2\leq z\leq 10} \to (\mathbb{Z}_{0\leq})^4$.

Finally, it should be noted that while the loss function L used to define our notion of risk was arbitrary, some loss functions are easier to deal with in practice than others. Since our interest is in the computation of prime factorization sequences, a choice of L which reflects the "multiplicative" nature of this task would be ideal. One such candidate is the

loss function

$$L_{\text{multiplicative}}(z_1, z_2) := \frac{z_1}{\gcd(z_1, z_2)} + \frac{z_2}{\gcd(z_1, z_2)} - 2,$$

where z_1, z_2 are integers. While this is a loss function defined between integers, we can extend it to a loss function on prime factorization sequences by simply defining $L_{\text{multiplicative}}(s_1, s_2) := L_{\text{multiplicative}}(z_1^{s_1}, z_2^{s_2})$, where $z_1^{s_1}, z_2^{s_2}$ are the integers whose prime factorization sequences are given by s_1, s_2 respectively. In other words, the multiplicative loss between prime factorization sequences provides a measure of how close sequences are to one another based on how close the integers factorized by those sequences are in a multiplicative sense.

It is easy to verify that if m, n are *co-prime* integers (i.e. they share no common divisor beyond 1) then

$$L_{\text{multiplicative}}(mx, nx) = L_{\text{multiplicative}}(m, n) = m + n - 2,$$

meaning $L_{\text{multiplicative}}$ respects some aspects of the divisibility relations between integers and thus really does provide a measure of how close two integers are in a multiplicative sense. Unfortunately, $L_{\text{multiplicative}}$ is not differentiable and is therefore hard to use under the empirical risk minimization paradigm, which typically requires the use of a differentiable loss function so that some form of gradient descent can be used to solve the associated optimization problem. Thus, for practical considerations we instead use an extension of the classic squared loss function to vectors, known as *squared vector loss*

$$L_2(s_1, s_2) := ||s_1 - s_2||_2^2,$$

where $||\cdot||_2$ denotes the vector 2-norm. Despite not being particularly sensitive to the multiplicative structure of our problem, L_2 is at least a reasonable choice of loss function (since our choice of loss function is ultimately arbitrary) and it is has the benefit of being differentiable.

To summarize, determining prime factorizations is equivalent to computing the outputs of the prime factorization map p. Moreover, p can be recast somewhat trivially as the joint risk minimizer for the joint distribution $(Z, p(Z))$, where the distribution of Z and loss function L are arbitrary. Nonetheless, due to practical considerations, we assume $Z \sim \text{Uniform}(\mathbb{Z}_{2\leq z\leq k})$ and $L = L_2$. This allows us to redefine our inferential objective as the more tractable, reduced aim of estimating p_k, the restriction of p to $\mathbb{Z}_{2\leq z\leq k}$, where p_k is thought of as the joint risk minimizer of the joint distribution $(Z, p_k(Z))$ under loss function L_2.

3.4 Experimental Design and Neural Network Architecture

So, our finalized goal is to estimate the restricted prime factorization map

$$p_k : \mathbb{Z}_{2\leq z\leq k} \to (\mathbb{Z}_{0\leq})^{\#\{\text{primes in } \mathbb{Z}_{2\leq z\leq k}\}}$$

for some user-specified k. With p_k phrased in the language of risk minimization, all that remains is to specify the statistical model detailing our data collection process and the estimation procedure $\hat{p}_k$ we will use to estimate p_k. Recalling from Chapter 2 that we are taking a supervised learning approach centered around training deep multilayer perceptrons via empirical risk minimization, it follows that we need only to explain how we will acquire our labeled data, the form empirical risk takes in our context, and which MLP architecture we will be considering.

Since we are in the fortunate position of having full control over how we choose to collect data for estimation, the approach we take is to simply have a computer simulate data for us. In more detail, we will begin by using a computer program to sample some number n of integers Z_i from $\mathbb{Z}_{2\leq z\leq k}$ independently and uniformly-at-random. Then, using another computer program, we will compute their (finite-length) prime factorization sequences $p_k(Z_i)$.

The end result is that we will have computer-generated labeled data $\{(Z_i, p_k(Z_i))\}_{i=1}^n$, which we will then use to estimate p_k. Thus, the statistical model for our data D is

$$D := \{(Z_i, p_k(Z_i))\}_{i=1}^n \sim_{iid} (Z, p_k(Z)),$$

where $Z \sim \text{Uniform}(\mathbb{Z}_{2\leq z\leq k})$.

For the sake of clarity, note that the reason we sample from the subset $\mathbb{Z}_{2\leq z\leq k}$ for some user-specified k rather than the entirety of $\mathbb{Z}_{2\leq}$ is because $\mathbb{Z}_{2\leq}$ is unbounded and p is complicated. More specifically, since p is a complicated function, we don't have any understanding a priori of how p will behave out in the tail of $\mathbb{Z}_{2\leq}$. This means that to get an idea of what p does asymptotically, we have to sample arbitrarily large integers. While we could grab integers all throughout $\mathbb{Z}_{2\leq}$, the problem is that we will only be able to sample finitely many integers and thus will be unable to sample integers beyond a certain bound. As a consequence, we have no hope of estimating the behavior of p for arbitrarily large integers. Hence, we restrict our interest to some user-specified subset $\mathbb{Z}_{2\leq z\leq k}$, which allows us to focus on estimating p_k and thus sidestep the headache caused by our inability to extrapolate beyond whatever integers we end up sampling.

Moreover, note that the reason we sample the integers according to $Z_i \sim \text{Uniform}(\mathbb{Z}_{2\leq z\leq k})$ is not deep; it's just a very simple sampling scheme which allows us to observe integers from all throughout $\mathbb{Z}_{2\leq z\leq k}$ without having to think too much, provided the sample size n is large enough. One could instead perform a deterministic sampling of the integers, but it's not immediately clear how to do that without introducing some kind of "sampling bias" into the experimental design. For example, if the integers were naively sampled deterministically in equally-spaced increments, depending on the increment size it's possible that we might miss out on all the even or odd integers.

Now, consider our estimator $\hat{p}_k$, which we construct from data D by training multilayer perceptrons under the empirical risk minimization paradigm. Due to the character of our data and our choice to use L_2 as our loss function, it follows that empirical risk takes the form

$$\hat{R}_{\text{empirical risk}}(f) := \frac{\sum_{i=1}^{n} L_2(f(z_i), p_k(z_i))}{n} = \frac{\sum_{i=1}^{n} ||f(z_i) - p_k(z_i)||_2^2}{n}.$$

Hence, $\hat{p}_k$ is given by

$$\hat{p}_k := \arg\min_{\text{nn} \in \mathcal{NN}} \hat{R}_{\text{empirical risk}}(\text{nn}) = \arg\min_{\text{nn} \in \mathcal{NN}} \frac{\sum_{i=1}^{n} ||\text{nn}(z_i) - p_k(z_i)||_2^2}{n},$$

where $\mathcal{NN}$ is the family of MLPs under consideration.

So, which architecture should be used for the family $\mathcal{NN}$ of MLPs we will be training? Recall from Chapter 2 that the form of an MLP (with bias nodes) is given by

$$\text{nn}(X) := g_{\text{output}}(B^O + W^O g_{\text{hidden}}(B^{\mathcal{D}} + W^{\mathcal{D}} g_{\text{hidden}}(\dots g_{\text{hidden}}(B^1 + W^1 X)\dots))).$$

Hence, to specify the architecture of a family of MLPs, we need to specify the following: the input layer X, the depth $\mathcal{D}$ and width $\mathcal{W}$ of the hidden layers, an activation function g_{hidden} for the nodes of the hidden layers, an output layer O, and an activation function g_{output} for the output layer nodes. Note that X and O are designated by the form of the function we are interested in estimating (in this case, p_k), while $\mathcal{D}$, $\mathcal{W}$, g_{hidden}, g_{output} are up to us to decide (though g_{output} could also be considered as partially specified by p_k, in the sense that the co-domain of p_k suggests characteristics g_{output} should satisfy).

Since p_k is a map which takes an integer as input and outputs a sequence of length $\#\{\text{primes in } \mathbb{Z}_{2 \leq z \leq k}\}$, it follows that our MLP architecture must have an input layer X with 1 node (since we have 1 input variable) while the output layer O must have $\#\{\text{primes in } \mathbb{Z}_{2 \leq z \leq k}\}$

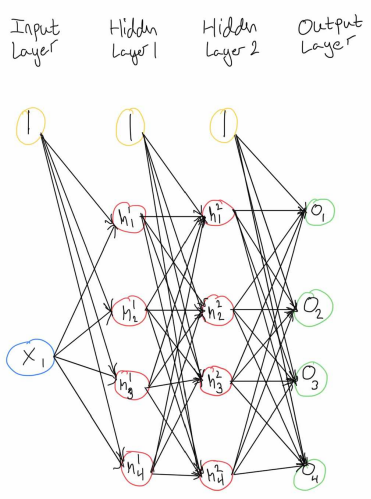

Figure 3.1: An example of a potential multilayer perceptron architecture which could be used for estimating p_{10}, the restriction of the prime factorization map p to $\mathbb{Z}_{2\leq z\leq 10}$. Blue denotes the input layer, red denotes hidden layers, yellow denotes bias nodes, and green denotes the output layer. Notice that the input layer consists of 1 node, since we have 1 input variable, while the output layer consists of 4 nodes, as there are 4 primes in $\mathbb{Z}_{2\leq z\leq 10}$ and thus 4 sequence entries to predict. Moreover, there are 2 hidden layers, each with a width of 4; other architectures under consideration could have different hidden layer depths and widths, but their input and output layers will remain the same.

nodes (since we have $\#\{\text{primes in } \mathbb{Z}_{2\leq z\leq k}\}$ output variables to account for; see Figure 3.1). Moreover, since the outputs of the neural net are intended to be approximate prime factorization sequences, it would seem reasonable to use a rounding activation function in the output layer to force the sequences to have the integer-valued entries they ought to have. Unfortunately, the rounding function is not differentiable and would thus obstruct our ability to train the neural net, so we instead let the output layer activation function g_{output} be the identity map (as a consequence, our neural nets are going to output real-valued sequences rather than integer-valued sequences, but that's fine).

Moving on, all that remains is to specify the depth, width, and activation function for the hidden nodes. However, rather than determining a specific family $\mathcal{NN}$ of MLPs by fixing $\mathcal{D}$ and $\mathcal{W}$, we will instead consider a collection of MLP families $\{\mathcal{NN}_{\mathcal{D}_i,\mathcal{W}_i}\}_i$ which differ from one another only by their depth $\mathcal{D}_i$ and width $\mathcal{W}_i$. We do this because we aren't quite

sure what depths and widths are appropriate a priori and thus would like to trial-run a few potential choices to see which combinations end up performing better.

Note that this is an example of model selection; we aren't quite sure what the optimal hypothesis space $\mathcal{H}$ is, so we will try a few different candidate hypothesis spaces $\{\mathcal{H}_i\}_i$ and see what works best (where in this context, the hypothesis spaces under consideration are families of MLPs which have different hidden layer depths and widths). It should also be noted that due to our ability to simulate data as needed, our model selection process is very simple; instead of reserving some of the data D to use as a testing set, we can just run an appropriate computer program to generate an additional data set D' for use as our testing set. In other words, we have access to the simplest out-of-sample method for estimating the risk of our fitted neural nets, making the risk performance comparisons between the fitted neural nets a straightforward process.

Thus, all that truly remains to be specified is our choice of activation function for the hidden nodes. As we are particularly interested in fitting deep MLPs, we need to choose an activation function for the hidden nodes which simultaneously creates non-linearity while retaining enough simplicity to allow for efficient training even when many hidden layers are involved. To do this, we will let g_{hidden} be the *Rectified Linear Unit* (ReLU) activation function, which is defined by

$$\text{ReLU}(x) := \max(0, x).$$

Our decision to use the ReLU function stems from the fact that nowadays it is the standard choice of hidden activation function for deep learning (*Glorot et al.* [2011]). To understand why, notice that the ReLU function is just the identity mapping with negative values sent to 0, and thus is probably the simplest piecewise-linear function that can be conceived of. As piecewise-linear functions themselves can be considered the most basic class of non-linearities, it therefore seems natural to view ReLU as an attractive activation function for deep learning.

Putting this all together, our goal is to estimate p_k for some user-specified k, where p_k is thought of as the joint risk minimizer of the joint distribution $(Z, p_k(Z))$ under the loss function L_2 and $Z \sim \text{Uniform}(\mathbb{Z}_{2 \leq z \leq k})$. To perform the estimation, we use a computer to simulate supervised learning data $D = \{(Z_i, p_k(Z_i))\}_{i=1}^n \sim_{iid} (Z, p_k(Z))$. Then, we will train a few different families (the exact number of which will be determined based on how smoothly training goes) of appropriate multilayer perceptrons $\{\mathcal{NN}_{\mathcal{D}_i, \mathcal{W}_i}\}_i$ (with a particular emphasis on deepness) on the data D by minimizing empirical risk in order to build a couple estimates of p_k. We will then simulate an additional data set D' over which we will estimate the risks (continuing to use empirical risk as our risk estimator) of each fitted neural net, in order to get an honest measure of which neural nets performed the best.

Chapter 4 Training

4.1 Training Approach

With our goal and strategy explicitly outlined in Chapter 3, we are now finally ready to procede with training neural nets to compute prime factorization sequences. To get started, recall that our task of training a neural net is machine learning terminology for us attempting to solve the optimization problem

$$\min_{\text{nn} \in \mathcal{NN}_{\mathcal{D}_i, \mathcal{W}_i}} \frac{\sum_{i=1}^{n} ||\text{nn}(z_i) - p_k(z_i)||_2^2}{n}.$$

Since every family $\mathcal{NN}_{\mathcal{D}_i, \mathcal{W}_i}$ of neural nets we will consider is parameterized by the weight matrices and bias vectors which compose every layer of the architecture, it follows that we can identify a neural net from such a family with a particular (non-unique) choice of weights and biases. Thus, our goal of training a neural net can be more concretely thought of as trying to find weights and biases which yield a neural net (of the appropriate architecture) best fitting our data (where "best fit" is measured by empirical risk).

While we will not discuss the intricacies of our optimization problem in too much depth (as doing so would require another paper due to its complexity), there are some important observations to make. First, any issues we have from here on out will be due to training difficulties, i.e. problems associated with our ability to solve the empirical risk minimization optimization problem. While some choices of hypothesis space result in fairly easy, if not trivial, training procedures, training neural networks (and in particular, deep neural networks) is unfortunately an extremely challenging problem (*Goodfellow et al.* [2016]). This is because, roughly speaking, the overall behavior of a neural net depends on its weight and bias values in a very complicated, high-dimensional fashion due to the hierarchical nature of the network and the sheer number of parameters involved. As a consequence, the

loss landscape (or rather, the empirical risk landscape) of neural nets typically ends up being a very bumpy, highly non-convex surface in a high-dimensional space, thus making it incredibly difficult to find weight and bias values which sufficiently minimize empirical risk.

Secondly, although it is difficult to see, the loss landscape associated to our neural nets is in fact smooth (or rather, piecewise smooth). Due to this smoothness, some variant of gradient descent will be employed to perform the optimization. As it turns out, applying gradient descent is the standard approach to training deep neural nets (*Goodfellow et al.* [2016]), despite the prevelance of other possible optimization methods (such as, say, Quasi-Newton methods). This is due largely to the number of highly efficient gradient calculation/estimation algorithms that are available for use when training neural nets, which take advantage of the hierarchical structure of the network to cut down on the number of redundant sub-calculations involved with gradient computation and thus speed things up. Even so, it should be noted that the gradient descent's "valley descending" nature will typically experience serious difficulty globally minimizing the bumpy high-dimensional objective function involved with training neural nets. In particular, the solution converged upon by gradient descent will depend strongly on the algorithm's point of initialization, the step sizes used during the descent, and other aspects of the algorithm.

Finally, since we lack an in-depth theoretical analysis for our specific optimization problem, it's not clear a priori what specific issues we will encounter during training; due to the complexity of the problem, the only thing we can be sure of is that there will be complications. While the software we will use to train neural nets will attempt to partially account for generic problems that tend to crop up during training, or at least has built-in capabilities to handle them, ultimately we can't be certain of what to expect and therefore can't be sure of what the best "settings" are for our problem. In particular, it's not clear how much data we ought to collect before hand, nor which network depths and widths we should use, nor

where we ought to initialization our gradient descent, nor how big our descent step sizes should be, etc. Thus, the approach we are forced to take in the process of training is to simply try some initial specifications, see how well they work (or rather, how badly they fail to work), identity the problems and adjust accordingly, and rinse and repeat until we have something which seems to work well.

In terms of computer implementation, all programming will be done using the Python language. To generate our training and testing data sets, we will use homemade code which appropriately samples integers and computes their prime factorization sequences. To develop our neural nets, the software we will use is *PyTorch* (*Paszke et al.* [2019]), a free open source Python library designed specifically for easily designing neural nets. Beyond providing a flexible framework for defining neural net architectures, the PyTorch framework also comes complete with all the standard optimization algorithms involved in the training of neural nets, thus greatly reducing the work we have to do ourselves to train our networks. Note that due to PyTorch's reliance on "tensors" (which, for our purposes, are just matrices with fancy names), we will think of our data D in matrix form. That is, rather than thinking of our data D as a set of ordered pairs, we will instead represent D as an $n \times (\#\{\text{primes in } \mathbb{Z}_{2\leq z\leq k}\}+1)$ matrix $D = \begin{bmatrix} Z_1 & Z_2 & \dots & Z_n \\ p_k(Z_1) & p_k(Z_2) & \dots & p_k(Z_n) \end{bmatrix}^T$ (where transposes are used to save page space). For the same reasons, most of our calculations will be expressed in matrix form too.

For no particular reason, we let our user-specified choice of k be $k = 64000$. As there are 6413 primes in $\mathbb{Z}_{2\leq z\leq 64000}$, it follows that p_k takes the form $p_{64000} : \mathbb{Z}_{2\leq z\leq 64000} \to (\mathbb{Z}_{0\leq})^{6413}$. Since we are approaching training in an essentially blind, fix-as-you-go fashion, we arbitrarily let our data sample size n be $n = 10000$. Hence, our data D will be a 10000×6414 matrix consisting of 10000 integers sampled from $\mathbb{Z}_{2\leq z\leq 64000}$ coupled with their corresponding length 6413 prime factorization sequences. Moreover, for the sake of simplicity we will begin our

experiments with training a single family of multilayer perceptrons (with the ReLU-based architecture specified in Chapter 3). As we are particularly interested in training deep networks, the initial architecture we will arbitrarily consider is $\mathcal{NN}_{90,3}$, the family of MLPs with depth 90 and width 3 (as well as an input layer of size 1 and an output layer of size 6413). To get an accurate measure of performance, we will simulate an additional data set D' to use as our testing set, where D' will consist of another 10000 integers sampled from $\mathbb{Z}_{2\leq z\leq 64000}$ (so D' is a 10000×6414 matrix).

4.2 Training Results

Running our data generation code, we generate D. Since D is a 10000×6414 matrix, we can't list the entire matrix; instead, we provide the first and last 3 rows with partial listing of the prime factorization sequences below. Note that the 1st entry of each row is an integer while the remaining entries correspond to the entries of the prime factorization sequence.

```
Data
tensor([[4.019800e+04, 1.000000e+00, 0.000000e+00,  ..., 0.000000e+00,
         0.000000e+00, 0.000000e+00],
        [5.885900e+04, 0.000000e+00, 0.000000e+00,  ..., 0.000000e+00,
         0.000000e+00, 0.000000e+00],
        [5.459000e+03, 0.000000e+00, 0.000000e+00,  ..., 0.000000e+00,
         0.000000e+00, 0.000000e+00],
        ...,
        [4.669800e+04, 1.000000e+00, 1.000000e+00,  ..., 0.000000e+00,
         0.000000e+00, 0.000000e+00],
        [2.279100e+04, 0.000000e+00, 1.000000e+00,  ..., 0.000000e+00,
         0.000000e+00, 0.000000e+00],
        [6.083600e+04, 2.000000e+00, 0.000000e+00,  ..., 0.000000e+00,
         0.000000e+00, 0.000000e+00]])
```

Next, we run our neural net training scheme on our generated data. We use 2000 iterations of a popular gradient descent variant known as *Adam* (*Kingma and Ba* [2014]), using a standard choice of 10^{-3} for the value of the step length hyperparameter and a randomized weight/bias value initialization known as *He Initialization* (*He et al.* [2015]) which is

designed to work well with the structure of our neural net (in particular, with our use of ReLU). For ease of tracking the descent's progress, every 50th iteration we print the current empirical risk value, of which we list the first, middle, and last 3 50 iterations below. We also list the total training time in seconds, to see how costly our optimization was to perform.

```
Training progress
0
tensor(815.186829)
50
tensor(559.308838)
100
tensor(392.627197)
...
950
tensor(4.772607)
1000
tensor(4.760278)
1050
tensor(4.751269)
...
1900
tensor(4.727158)
1950
tensor(4.727150)
2000
tensor(4.727146)

Time
672.345993
```

So, our initial neural net produced an empirical risk value of 815.187, but after around 1000 iterations we achieved an empirical risk of 4.760. For the remaining 1000 iterations the empirical risk remained largely unchanged, decreasing only slightly to a final value of 4.727 after a total training time of 672.346 seconds (roughly 11.2 minutes). Note that our lack of real improvement for 1000 iterations suggests that either our gradient descent algorithm successfully zero'd in on some sort of critical point, or it reached a very flat part of the objective function where training procedes slowly.

With an empirical risk value of 4.727, it would seem that our trained neural net is doing a pretty good job, as ultimately the lowest possible empirical risk that could be achieved would be 0 (i.e. a perfect fit). But, recall the form of our prime factorization sequences – they are very, very sparse sequences, and the non-zero entries they do possess are typically small integers. This suggests that a squared vector loss which takes a value in the single digits may not actually indicate a good fit, as single digit losses are in some sense roughly the same "size" as the sequences themselves. To better understand, we need a benchmark performance with which to compare our neural net against.

Since the neural net is intended to be a complicated, flexible function approximating the prime factorization map, an obvious benchmark for comparison would be the performance of the constant vector function $c_{\text{best fit}}$ which minimizes empirical risk on our data. This seems reasonable, as a constant function captures none of the information detailing how the prime factorization map varies with respect to its integer inputs and thus represents the most basic function we could fit to our data (and the "antithesis" to our neural nets). Due to our choice of L_2 as our loss function, $c_{\text{best fit}}$ takes an incredibly simple form. Namely,

$$c_{\text{best fit}} = \frac{\sum_{i=1}^{n} p_k(z_i)}{n},$$

i.e. $c_{\text{best fit}}$ is the vector-valued sample mean of our data's prime factorization sequences. Note that this observation is analogous to the well known fact that the sample mean is the constant which minimizes empirical risk using the standard squared loss function (which in turn is simply a specific instance of the more general fact that expected values minimize risk under squared loss, as discussed in Chapter 2).

Similarly, the empirical risk associated to $c_{\text{best fit}}$ is given by the simple expression

$$\hat{R}_{\text{empirical risk}}(c_{\text{best fit}}) = \sum_{j=1}^{\#\{\text{primes in } \mathbb{Z}_{2\leq z\leq k}\}} \frac{\sum_{i=1}^{n}(p_k^j(z_i) - c_{\text{best fit}}^j)^2}{n},$$

i.e. $\hat{R}_{\text{empirical risk}}(c_{\text{best fit}})$ is the sum of the (non-Bessel corrected) sample variances for each entry of our data's prime factorization sequences. Hence, we can easily determine the empirical risk of our best fitting constant. To compare the empirical risk of our constant function to the empirical risk of our fitted neural net, we will consider the ratio $\frac{\hat{R}_{\text{empirical risk}}(\text{fitted neural net})}{\hat{R}_{\text{empirical risk}}(c_{\text{best fit}})}$.

On our data, the empirical risk of the best fitting constant value and the empirical risk ratio of our fitted neural net to the best fitting constant is given below.

```
Empirical risk of best fitting constant to training set
tensor(4.727291)

Empirical risk ratio
tensor(0.999969)
```

Thus, the empirical risk of our fitted neural net is only ever so slightly smaller than the empirical risk of our best fitting constant value. This indicates that, despite how small the empirical risk seems to be, our neural net is in fact providing a terrible fit; rather than training a neural net for 11 to 12 minutes, we could instead instantly train a constant function and achieve essentially the same goodness of fit! As this is obviously indicative of a problem, we must now look "under the hood" to see what's going wrong with our neural net training process.

The first place to look is at the outputs of our fitted neural net, which we can then compare to the true prime factorization sequences composing our data. The first and last 3 prime factorization sequences, along with the associated sequence approximations generated by our fitted neural net, are given below.

```
Factorization sequences
tensor([[1., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [1., 1., 0.,  ..., 0., 0., 0.],
        [0., 1., 0.,  ..., 0., 0., 0.],
        [2., 0., 0.,  ..., 0., 0., 0.]])

Fitted neural net approximate factorization sequences
tensor([[ 9.802641e-01,  5.052068e-01,  2.483061e-01,  ...,
         -1.147389e-06,  1.654029e-06, -5.215406e-07],
        [ 9.802641e-01,  5.052068e-01,  2.483061e-01,  ...,
         -1.147389e-06,  1.654029e-06, -5.215406e-07],
        [ 9.802641e-01,  5.052068e-01,  2.483061e-01,  ...,
         -1.147389e-06,  1.654029e-06, -5.215406e-07],
        ...,
        [ 9.802641e-01,  5.052068e-01,  2.483061e-01,  ...,
         -1.147389e-06,  1.654029e-06, -5.215406e-07],
        [ 9.802642e-01,  5.052068e-01,  2.483061e-01,  ...,
         -1.147389e-06,  1.654029e-06, -5.140901e-07],
        [ 9.802642e-01,  5.052068e-01,  2.483061e-01,  ...,
         -1.147389e-06,  1.654029e-06, -5.140901e-07]])
```

Clearly, our fitted neural net outputs look nothing like the true prime factorization sequences they are intended to approximate. In fact, it appears as though our neural net is outputting the same prime factorization sequence for every single integer in our generated data set; in other words, the neural net appears to be constant. Technically, our particular neural net can't be exactly constant, as it achieves an empirical risk that is slightly smaller than the smallest empirical risk a constant function can attain (and upon closer inspection of the neural net outputs listed above we do indeed see slight differences between some of the outputs, in particular on the first and last sequence entries). Nonetheless, the "essentially constant" nature of our network seems like a plausable explanation as to why our neural net is giving a terrible performance, but raises another question – why is our neural net more or less a constant function?

To understand what's going on, we have to delve deeper. Rather than just looking at the outputs of our neural net, we need to take a look at what happens "inside" the network, i.e. what happens along the hidden layers. As our neural nets are identified by their weight and bias values, it seems reasonable to first inspect the parameter values of our fitted neural net. Unfortunately, as mentioned in Chapter 2, neural nets are generally not very interpretable; in particular, it's unclear how the weight and bias values of a given layer impact the overall behavior of the network. Hence, rather than looking at the parameter values associated to our fitted neural net, we will instead consider the output produced at each layer of the network. This allows us to see how the input layer is transformed, layer by layer, into the finalized output we have been observing, which may shed some light on the situation. Of course, since there are 90 hidden layers, each of width 3, it follows that there will be 90 10000×3 matrices of hidden layer outputs to examine, which is too much to list. Instead, we provide a sample of the layer outputs below, enough to see some patterns emerging.

```
Layer 1 output
tensor([[0.000000, 0.000000, 0.524058],
        [0.000000, 0.000000, 0.347817],
        [0.231101, 0.000000, 0.852146],
        ...,
        [0.000000, 0.000000, 0.462670],
        [0.062580, 0.000000, 0.688456],
        [0.000000, 0.000000, 0.329146]])

Layer 2 output
tensor([[0.000000, 0.163791, 0.000000],
        [0.000000, 0.090141, 0.000000],
        [0.000000, 0.201714, 0.000000],
        ...,
        [0.000000, 0.138137, 0.000000],
        [0.000000, 0.205634, 0.000000],
        [0.000000, 0.082338, 0.000000]])

Layer 4 output
tensor([[0.471476, 0.000000, 0.000000],
        [0.471320, 0.000000, 0.000000],
        [0.471556, 0.000000, 0.000000],
```

```
        ...,
        [0.471422, 0.000000, 0.000000],
        [0.471565, 0.000000, 0.000000],
        [0.471303, 0.000000, 0.000000]])

Layer 8 output
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        ...,
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])

Layer 9 output
tensor([[0.219632, 0.224696, 0.021378],
        [0.219632, 0.224696, 0.021378],
        [0.219632, 0.224696, 0.021378],
        ...,
        [0.219632, 0.224696, 0.021378],
        [0.219632, 0.224696, 0.021378],
        [0.219632, 0.224696, 0.021378]])

Layer 90 output
tensor([[0.000000, 0.266697, 0.524548],
        [0.000000, 0.266697, 0.524548],
        [0.000000, 0.266697, 0.524548],
        ...,
        [0.000000, 0.266697, 0.524548],
        [0.000000, 0.266697, 0.524548],
        [0.000000, 0.266697, 0.524548]])
```

From the outputs of the hidden layers, two patterns seem to emerge. First, the integer outputs at most hidden layers appear to be vectors consisting mostly of 0's. Second, the integer outputs rapidly become indistinguishable from one another as we move deeper through the network. The key observation tying these two patterns together is the behavior exhibited in layer 8 and the layers following it – namely, layer 8 output $\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$ for every integer, and from that point onwards every hidden layer gave the same output to every integer.

From these observations, we can piece together what seems to be happening; layers which yield outputs consisting mostly of 0s "partially collapse" the outputs of integers together. While such a layer may yield outputs which differ from integer to integer, the variability in that layer's outputs is significantly reduced because the only source of variability is in the non-zero entries, which are few in number. As a consequence, the layers beyond the "bottlenecking" layer will receive inputs which exhibit small levels of variability, and will therefore generate outputs which also generate small (if not smaller) levels of variability (since each layer is ultimately a continuous function of the preceeding layer). Thus, a layer whose outputs consist mostly of 0's essentially makes it more likely that the next layer's outputs are fairly close together, and so forth and so on, making an eventual progression to approximately constant outputs deeper in the neural net more likely. Moreover, the earlier a layer of mostly 0's is encountered in the network, the more dramatic this effect is (since there are more layers which will be affected by this bottleneck effect, and also more potential for additional bottlneck layers to exist).

In the extreme case, where a layer yields outputs consisting entirely of 0's, the layer "completely collapses" the outputs of integers together to the 0 vector. This forces every succeeding layer to return an identical output for every integer. In other words, a layer consisting entirely of 0's completely severes the dependence of the network's final output on the integer input, thus rendering the network a constant function (at least, on our sampled data). Note that this is exactly what occurs in layer 8 of our network. In prior layers, we only had partially 0'd out layers, corresponding to our observation that the variability in integer outputs was mildly decreasing as we went through the hidden layers. But in layer 8, we encountered a layer which gave only 0's, after which every hidden layer gave the exact same output to each of our sampled integers.

Now, note that our new understanding seems to suggest that our network truly is constant over our data (due to having a layer whose outputs were entirely 0's), despite the fact that the output layer appears to experience some level of variability. One explanation for the observed variability is that our limited ability for precision (due to our use of floating-point numbers) is somehow resulting in issues akin to rounding errors occuring in the output layer; in fact, this is exactly what is happening. While we will not delve into this in too much detail, for completeness we sketch the explanation below.

First, recall that the value of an output layer node is given by an affine combination of the 90th hidden layer's outputs and thus involves a calculation consisting of a bunch of multiplications and additions. Due to how rounding works, floating point arithmetic is not associative, so performing the additions and multiplications comprising the output layer node calculation in different orders will yield slightly different results. Next, note that PyTorch automatically parallelizes computations when it can, meaning it uses all the available cores of a CPU to perform multiple tasks simultaneously. In particular, rather than calculating the outputs of each integer in our data set individually (i.e. one after the other), our neural net actually calculates the outputs of integers simultaneously. Surprisingly, when computations are run in parallel with one another, sometimes the order in which the operations are performed will differ. This is due largely to the fact that the CPU cores will perform whatever tasks happen to be available at that moment, but (due to random factors) the order in which tasks become available will differ from computation to computation, resulting in sub-calculations sometimes being performed in different orders. Thus, the fact that our neural net sometimes performs parallel computations using different orders of operations, coupled with the fact that floating point arithmetic is not associative, leads to occasional differences between outputs which should theoretically be the same. Hence, the output layer variation we are observing between the integers is seemingly just an artifact of the quirks of parallel computing and our limited precision, and thus our network is actually constant.

So, it seems like we've finally discovered the root of our problem: because we have a layer which outputs entirely 0's for every integer, our network is actually a constant function (it's just that precision issues made it hard to initially see that). Even if we hadn't encountered a layer consisting exclusively of 0's, the layers consisting almost entirely of 0's are prolific, meaning we likely would end up dealing with a non-constant network that varies so little as to be effectively constant. But either way, what's causing all the problematic 0's?

In formal terms, the problem we have stumbled upon is known as the *Dying ReLU* problem (*Lu* [2020]). The idea is that since the ReLU function is defined by $\text{ReLU}(x) = \max(0, x)$, any negative inputs will get sent to 0. Since it is possible (upon initialization and during training) to attain certain configurations of weights and biases at various hidden neurons which end up producing a negative linear predictor value for all of our inputs, it follows that after an application of the ReLU function those neurons will output 0 for all of our sampled inputs; such neurons are referred to as *dead neurons*.

Dead neurons correspond to nodes which "do nothing" in the network and thus effectively act as if they aren't really present in the architecture, i.e. a dead neuron essentially decreases the width of a hidden layer (see Figure 4.1). Perhaps surprisingly, dead neurons are not a problem in and of themselves; in fact, it's actually very useful to have dead neurons (*Glorot et al.* [2011]). This is because neural networks are typically highly over-parameterized (our neural net is actually an example of this – our network has 26726 parameters, but we only gathered 10000 training samples), and the ReLU-generated dead neurons allows us to automatically prune off unnecessary neurons, reducing the effective number of parameters and thus simplifying the model. In other words, the automatic sparsity provided by the ReLU function is useful because it provides a data-driven method of "reducing" the initial neural net hypothesis space (which is typically more complex than it needs to be) down to a more

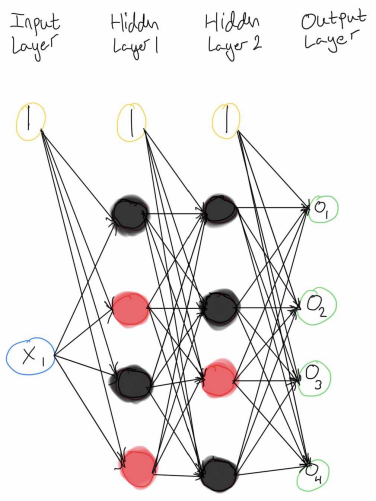

Figure 4.1: A visualization of the effect of ReLU-induced dead neurons on the neural net architecture. Black denotes dead neurons while red denotes active hidden neurons. Note how the dead neurons effectively narrow the network. While this is typically a good thing, as networks are oftentimes wider than they need to be, too much narrowing can restrict the neural net's ability to vary with respect to inputs. At the extreme end of the spectrum, a layer can consist entirely of dead neurons (visualized as a layer of only black nodes), in which case the neural network is in fact a constant function.

appropriate space for the problem at hand.

While some sparsity is thus extremely beneficial, the Dying ReLU problem arises when we have too much sparsity. When this happens, we pass the "sweet spot" of model complexity reduction and end up with a model which suffers from underfitting. In the extreme case, where the network has a layer consisting entirely of dead neurons, the result is a constant network. Moreover, while the Dying ReLU problem can be encountered during the course of training, it also happens frequently upon initialization; in particular, it is entirely possible that due to a bad choice of weight and bias initialization, the neural net can be "born dead".

In fact, this is what happened to us – layer 8 of our network was initialized as a dead layer. Once a neuron dies, it stays dead because gradient descent can no longer update the weights of that neuron (as the ReLU function has a slope of 0 over negative inputs). Thus, our network was initialized as a constant function, and remained a constant function. The

training we were witnessing was just the updating of weights and bias values beyond the final dead layer, in an attempt to attain the weights and bias values which allow our network to output $c_{\text{best fit}}$. This can be verified by noting that $c_{\text{best fit}}$ and the output of our network's output layer are almost exactly the same (especially once you account for the precision issues described previously), which we provide below.

```
Best fitting constant
tensor([0.982600, 0.505200, 0.248300,  ..., 0.000000, 0.000000, 0.000000])

Fitted neural net approximate factorization sequences
tensor([[ 9.802641e-01,  5.052068e-01,  2.483061e-01,  ...,
         -1.147389e-06,  1.654029e-06, -5.215406e-07],
        [ 9.802641e-01,  5.052068e-01,  2.483061e-01,  ...,
         -1.147389e-06,  1.654029e-06, -5.215406e-07],
        [ 9.802641e-01,  5.052068e-01,  2.483061e-01,  ...,
         -1.147389e-06,  1.654029e-06, -5.215406e-07],
        ...,
        [ 9.802641e-01,  5.052068e-01,  2.483061e-01,  ...,
         -1.147389e-06,  1.654029e-06, -5.215406e-07],
        [ 9.802642e-01,  5.052068e-01,  2.483061e-01,  ...,
         -1.147389e-06,  1.654029e-06, -5.140901e-07],
        [ 9.802642e-01,  5.052068e-01,  2.483061e-01,  ...,
         -1.147389e-06,  1.654029e-06, -5.140901e-07]])
```

Thankfully, there are a few work-arounds to the Dying ReLU problem. The key thing to note is that the Dying ReLU problem is not really an issue with the ReLU function per se, as the simplicity of ReLU and its automatic sparsity is generally beneficial. Instead, it is a consequence of a neural network architecture being too narrow for its depth (*Lu* [2020]). The narrowness makes it easier to have all the neurons within a layer be dead (as there are few neurons in each layer), and the deepness creates more opportunities to encounter a dead layer (as there are more layers), making it fairly likely that a dead layer will pop up somewhere on the network either upon initialization or during training. Hence, the fundamental solution is to consider a network with wider hidden layers; by doing so, we make it harder for every neuron in a layer to wind up dead and thus reduce our chances of initializing or training into a constant network.

Besides widening the network, other avenues to solving the Dying ReLU problem could be implemented. For example, a ReLU variant intended to avoid dead neurons could be used in place of ReLU as the hidden activation function. Alternatively, tons of batch normalization layers could be added to the network to prevent the hidden layers from attaining negative values too often. However, the problem with these methods is that they tend to replace the Dying ReLU problem with another issue. While we will not discuss these possibilities in detail, the problems associated with using these alternative solutions can be summarized as resulting in more expensive training processes than what would otherwise be experienced by intelligently widening the network (*Lu* [2020]). Hence, the approach we take is to consider a wider network. Of course, widening a network can also be expensive, so how wide do we need to go?

According to *Lu* [2020], if a network is of depth $\mathcal{D}$, then having the hidden layers be of width $\mathcal{W} = \log_2(\frac{\mathcal{D}+1}{\delta})$ ensures a probability of at least $1 - \delta$ of avoiding Dying ReLU upon initialization. Note that by appropriately changing our weight/bias initialization scheme, we can attain the desired probability of avoiding Dying ReLU upon initialization with even smaller widths; that is, if we utilize a slightly modified initialization scheme, we can achieve our desired level of Dying ReLU risk reduction without having to widen as much. However, implementing a new initialization scheme requires a lot of modifications to the code we've been using, so we leave the initialization scheme alone and simply widen the network. Since our network is of depth $\mathcal{D} = 90$, it follows that we should consider an architecture of width $\mathcal{W} = \log_2(\frac{90+1}{0.1}) \approx 10$ to reduce the risk of being born dead to an at most 10% chance. Hence, we consider $\mathcal{NN}_{90,10}$ as our next family of neural nets to train on D, with the training results given below.

```
Training progress
0
tensor(283.197357)
```

```
50
tensor(111.110939)
100
tensor(46.294769)
...
950
tensor(4.727143)
1000
tensor(4.727141)
1050
tensor(4.727141)
...
1900
tensor(4.727141)
1950
tensor(4.727141)
2000
tensor(4.727143)

Time
804.1967175006866

Fitted wide neural net approximate factorization sequences
tensor([[9.825309e-01, 5.051892e-01, 2.482701e-01,  ..., 3.732741e-06,
         9.871088e-06, 8.961186e-06],
        [9.825309e-01, 5.051892e-01, 2.482701e-01,  ..., 3.732741e-06,
         9.871088e-06, 8.961186e-06],
        [9.825309e-01, 5.051892e-01, 2.482701e-01,  ..., 3.732741e-06,
         9.871088e-06, 8.961186e-06],
        ...,
        [9.825309e-01, 5.051892e-01, 2.482701e-01,  ..., 3.732741e-06,
         9.871088e-06, 8.961186e-06],
        [9.825309e-01, 5.051892e-01, 2.482701e-01,  ..., 3.732741e-06,
         9.871088e-06, 8.961186e-06],
        [9.825309e-01, 5.051892e-01, 2.482701e-01,  ..., 3.732741e-06,
         9.871088e-06, 8.961186e-06]], grad_fn=<AddmmBackward0>)
```

Alas, it seems our attempt to avoid Dying ReLU failed; while our new family of neural nets initialized with a lower empirical risk, ultimately during training it converged to the same finalized empirical risk as our previous family, except it took about an extra 2 minutes to do

so. Moreover, the neural net became constant, yet again approximating $c_{\text{best fit}}$. For the sake of completeness, we now finally generate our testing set D' and compare the performances of our original neural net and the wider neural net to the best fitting constant to the test data.

```
Empirical risk of original network on test set
tensor(4.742164)

Empirical risk of wider network on test set
tensor(4.742031)

Empirical risk of best fitting constant to test set
tensor(4.740665)
```

As expected, the two networks gave more or less identical performances. Furthermore, the two networks performed only slightly worse than the best fitting constant to the test set, with empirical risk ratios of 1.000316 and 1.000288 given by the original and wider network, respectively. So, as far as the performance of constant functions goes, our networks seem to do well in that regard at least.

Chapter 5 Discussion

Unfortunately, our attempts to avoid the Dying ReLU problem failed. Even if we hadn't ended up with a constant network due to completely dead layers, we likely would have ended up with an approximately constant function due to mostly dead layers (or our limited ability for precision), so either way our networks would have provided unsatisfactory approximations to the prime factorization map. Nonetheless, there's a ton of components that went into formalizing and attempting to solve our problem, and thus there's no shortage of things we can modify in an attempt to make a network which does provide reasonable approximate prime factorization sequences for integers.

With respect to the network itself, it seems as though the deep feedforward neural nets we tried were not suitable for the task of computing prime factorizations. In the future, perhaps even wider networks could be trained, possibly offering improved performance (especially when coupled with an initialization scheme which specficially attempts to avoid Dying ReLU). Moreover, as we still ended up with a constant network after widening, maybe an alternative hidden activation which avoids the Dying ReLU problem could be employed instead of ReLU. Alternatively, some kind of smooth approximation to the rounding function could be employed as the activation function in the output layer instead of the identity mapping, as this would be natural to try given the type of outputs we are generating (namely, integer-valued sequences). Another option to consider is that rather than dealing with a vanilla multilayer perceptron, it's possible that some other network like a residual neural net would yield better results (due to their use of skip-connections, which typically ease the training process).

Beyond modifying the network structure, which corresponds to tweaking the hypothesis space, we could also consider modifying the risk estimator we used to define our estima-

tor. For example, rather than using the L_2 loss, perhaps a differentiable approximation to $L_{\text{multiplicative}}$ could be used; this would allow the multiplicative structure of our problem to more easily "get heard" by our estimation technique, though it's unclear how exactly one might do this. Alternatively, some kind of regularization penalty term intended to punish functions for deviating from known properties of the prime factorization map could be tacked onto the empirical risk estimator; since we know the prime factorization map should have very sparse outputs, regularization intended to promote sparsity could work nicely.

Ultimately, all the tweaks prescribed above correspond to modifications to the optimization problem defining our estimated risk minimization estimator. However, due to the complicated nature of the training process, it is perhaps more accurate to think of an estimated risk minimization estimator as consisting of not only a choice of hypothesis space and risk estimator (which was outlined in Chapter 2), but also a choice of optimization algorithm. In other words, an estimated risk minimization estimator is most truthfully envisioned as a choice of optimization problem (specified by a choice of hypothesis space and risk estimator) coupled with a choice of optimization algorithm. This understanding makes it obvious that while the choice of hypothesis space and risk estimator is key, the choice of optimization algorithm is just as important to the construction of a good risk minimizer estimator; in particular, tweaks and modifications to the optimization algorithm could result in forming better risk minimizer estimates.

While some variant of gradient descent is still likely to be the right optimization technique to employ on our problem, more bells and whistles could be added to it or different choices of hyperparameter values could be made. For example, while we used He Initialization, which is intended to be used with ReLU as the hidden activation function, the scheme ultimately does not account for the Dying ReLU problem. Hence, a scheme which does explicitly account for the Dying ReLU problem may be of use. Similarly, we used a step length hyperparameter

value of 10^{-3}, but perhaps a smaller (or larger) value would give the Adam algorithm an improved ability to adaptively choose appropriate descent step lengths. This could allow our gradient descent to converge to a more appropriate critical point (rather than the constant network we seemed to consistently end up at). Alternatively, rather than picking specific hyperparameter values for our optimization method ahead of time, we could instead think of the different choices of optimization hyperparameter values as defining different potential estimators (since we really should be thinking of the choice of optimization algorithm as a defining aspect of our estimator). Then, we could take a data-driven approach to selecting the hyperparameter values by using ensemble methods like model selection to improve upon the associated set of potential estimators, i.e. we could choose the optimization hyperparameter values based on which ones result in risk minimizer estimates that are estimated to perform the best.

It is also worth entertaining the possibility that our method of representing prime factorizations as a function may not be the best way to do so. As is, our chosen encoding method results in the prime factorization map yielding outputs which are very high-dimensional and incredibly sparse; by this observation alone, it makes sense that training any function (let alone a neural net) to approximate the prime factorization map is an incredibly challenging task. While there aren't any other possibilities which immediately come to mind, it's possible that an alternative method of encoding prime factorizations may result in an easier training process. In particular, perhaps some kind of unsupervised learning technique applied to the prime factorizations, with the intent of extracting relevant features, could generate a more appropriate encoding method (see *Goodfellow et al.* [2016] for more details). Alternatively, while there could be easier representations to deal with, it's also very possible that this difficulty will persist in all formulations of the problem, lending support to the intuition that computing prime factorizations is fundamentally a difficult problem.

Nonetheless, one possible avenue to overcoming this (possibly) intrinsic difficulty could be to break the goal of computing prime factorizations down into many sub-problems. Then, one could train neural nets to solve the sub-problems, and then ensemble them together to build a neural net which computes prime factorizations. For example, perhaps one could train a neural net to predict which powers of 2 are in an integer's prime factorization. Then, separately train another network to predict the powers of 3 in the prime factorization, and so on. Once all the sub-problem solving networks are sufficiently trained, they could then be stitched together to create a network which hopefully does a better job of computing prime factorizations than a network who approaches the problem head-on. While this approach is theoretically no different from the approach we took, in practice it may work better because training is our biggest issue, and training many networks to solve simple sub-problems is likely easier than training one network to solve the entire complex problem in one go.

References

Chen, R. T., Y. Rubanova, J. Bettencourt, and D. K. Duvenaud (2018), Neural ordinary differential equations, *Advances in neural information processing systems*, *31*.

Glorot, X., A. Bordes, and Y. Bengio (2011), Deep sparse rectifier neural networks, in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323, JMLR Workshop and Conference Proceedings.

Goodfellow, I., Y. Bengio, and A. Courville (2016), *Deep Learning*, MIT Press, `http://www.deeplearningbook.org`.

Hastie, T., R. Tibshirani, and J. Friedman (2009), *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer series in statistics, Springer.

He, K., X. Zhang, S. Ren, and J. Sun (2015), Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, doi:10.48550/ARXIV.1502.01852.

He, K., X. Zhang, S. Ren, and J. Sun (2016), Deep residual learning for image recognition, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, doi:10.1109/CVPR.2016.90.

Hochreiter, S., and J. Schmidhuber (1997), Long Short-Term Memory, *Neural Computation*, *9*(8), 1735–1780, doi:10.1162/neco.1997.9.8.1735.

Hornik, K., M. Stinchcombe, and H. White (1989), Multilayer feedforward networks are universal approximators, *Neural Networks*, *2*(5), 359–366, doi:https://doi.org/10.1016/0893-6080(89)90020-8.

Ioffe, S., and C. Szegedy (2015), Batch normalization: Accelerating deep network training by reducing internal covariate shift, in *International conference on machine learning*, pp. 448–456, PMLR.

Kingma, D. P., and J. Ba (2014), Adam: A method for stochastic optimization, *arXiv preprint arXiv:1412.6980*.

Kramer, M. A. (1991), Nonlinear principal component analysis using autoassociative neural networks, *AIChE Journal*, *37*(2), 233–243, doi:https://doi.org/10.1002/aic.690370209.

Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998), Gradient-based learning applied to document recognition, *Proceedings of the IEEE*, *86*(11), 2278–2324, doi:10.1109/5.726791.

Lenstra, A. K., H. W. Lenstra, M. S. Manasse, and J. M. Pollard (1993), The number field sieve, in *The development of the number field sieve*, edited by A. K. Lenstra and H. W. Lenstra, pp. 11–42, Springer Berlin Heidelberg, Berlin, Heidelberg.

Lu, L. (2020), Dying ReLU and initialization: Theory and numerical examples, *Communications in Computational Physics*, *28*(5), 1671–1706, doi:10.4208/cicp.oa-2020-0165.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala (2019), Pytorch: An imperative style, high-performance deep learning library, in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc.

Shalev-Shwartz, S., and S. Ben-David (2014), *Understanding Machine Learning: From Theory to Algorithms*, Cambridge University Press, doi:10.1017/CBO9781107298019.

Appendix

```
### Code For Data Generation; Data.py ###

import numpy as np
from random import randint
import torch

# Sieves for the primes <= k, returns a list of the
# primes between 2 and k
def sieve(k):
    primes = [True for i in range(0, k + 1)]
    primes[0] = False
    primes[1] = False
    p = 2
    while (p*p <= k):
        if (primes[p] == True):
            for i in range(p**2, k + 1, p):
                primes[i] = False
        p = p + 1
    return np.nonzero(primes)[0]

# Generates n samples of integers from 2 to k;
# returns list of primes between 2 and k as well as
# n independent draws from integers from 2 to k along with their
# prime factorization sequences as a float tensor,
# so that the data is compatible with PyTorch...
# comes with optional argument ‘‘primes", which exists so that
# if you already have the list of primes from a previous run of sieve,
# the program won’t waste time calculating primes again for no reason
# (particularly useful when you already generated training data and
# now need to generate test data)
def sampler(n, k, primes = []):
    if np.array_equal(primes, []):
        primes = sieve(k)
    data = np.zeros((n, len(primes) + 1))
    for i in range(0, n):
        s = randint(2, k)
        data[i, 0] = s
        q = s
        j = 0
```

```
        while (q != 1):
            p = primes[j]
            while (q % p == 0):
                data[i, j + 1] = data[i, j + 1] + 1
                q = q/p
            j = j + 1
    data = torch.from_numpy(data).float()
    return primes, data

### Code For Defining Networks, Loss, and Training Process; Classes.py ###

import time
import torch
nn = torch.nn
F = torch.nn.functional

# Net defines the class of neural nets we are using,
# i.e. it specifies all the built-in properties of our neural net objects;
# init specifies the attributes of the neural net class,
# which basically corresponds to defining the
# neural net architecture for each neural net by describing
# the maps defining each layer of the neural net, while
# forward just defines how each neural net should map inputs to outputs...
# hsizes specifies the width of the input layer and each hidden layer, while
# osizes specifies the width of the output layer
# (so for us, osizes will equal the number of primes in the range of
# integers being sampled)
# the printer argument of forward() just specifies
# whether we want to print out the hidden layer outputs or leave
# them hidden
class Net(nn.Module):
    def __init__(self, hsizes, osize):
      super(Net, self).__init__()
      self.hidden = nn.ModuleList()
      for i in range(len(hsizes) - 1):
          self.hidden.append(nn.Linear(hsizes[i], hsizes[i + 1]))
      self.out = nn.Linear(hsizes[len(hsizes) - 1], osize)

    def forward(self, x, printer = False):
        if printer == True:
            for layer in self.hidden:
                x = layer(x)
```

```
                x = F.relu(x)
                print(x)
            x = self.out(x)
        else:
            for layer in self.hidden:
                x = layer(x)
                x = F.relu(x)
            x = self.out(x)
        return x

# Defines our loss function directly as a class,
# like how nn.MSELoss() is defined, so that it is
# easier to use with features of PyTorch;
# ncol is the attribute you specify when you instantiate an object
# from the Loss class, it is what you need to multiply MSELoss()
# by to get the correct loss function for our purposes, because
# MSELoss() averages over the number of entries of a tensor, but we
# just want to average over the number of rows of a tensor
class Loss(nn.Module):
    def __init__(self, ncol):
        super().__init__()
        self.ncol = ncol
        self.criterion = nn.MSELoss()

    def forward(self, target, output):
        return self.ncol*self.criterion(target, output)

# Sets up training routine function, which takes
# input data, target data, a neural net object, a loss function object,
# the number of desired iterations, and the learning rate hyperparameter,
# and then optimizes the neural net fit,
# printing the final loss and training time once finished
def fit(input, target, net, loss, numiters, lr):
    # initializes optimizer
    optimizer = torch.optim.Adam(net.parameters(), lr = lr)
    start = time.time()
    for iters in range(0, numiters):
        # zeros out gradient
        optimizer.zero_grad()
        # calculates output of current neural net
        output = net(input)
```

```
        if iters % 50 == 0:
            print(iters)
            print(loss(target, output))
        # backpropogates
        loss(target,output).backward()
        # updates the neural net parameters
        optimizer.step()
    end = time.time()
    print(numiters)
    print(loss(target, net(input)))
    print(end - start)

### Code For Training, Testing Project Networks; TrainerTester.py ###

import Data
import Classes
import copy
import torch
import numpy as np
torch.set_printoptions(precision = 6)

# Generates data (namely, 10000 observations of integers collected between
# 2 and 64000)
[primes, data] = Data.sampler(10000, 64000)
input = data[:, [0]]
target = data[:, 1:]

# standardizes inputs, as is typical for use with neural nets
input = (input - min(input))/(max(input) - min(input))

# Instantizes (90, 3) neural nets and loss function
deephsizes = [1]
for i in range(1, 91):
    deephsizes.append(int(3))
deepnet = Classes.Net(deephsizes, len(primes))
loss = Classes.Loss(len(primes))

# Fits neural nets and prints run time for fitting procedure
fitdeepnet = copy.deepcopy(deepnet)
Classes.fit(input, target, fitdeepnet, loss, 2000, 1e-3)

# Compares network performance to best fitting constant performance
```

```
torch.mean(target, axis = 0)
print(sum(torch.var(target, axis = 0, unbiased = False)))
print(loss(fitdeepnet(input), target)/sum(torch.var(target, axis = 0, unbiased = False)))

# Spits out true factorization sequences and
# predicted output under fitted neural net
print(target)
print(fitdeepnet(input))

# Spits out fitted neural net hidden layer outputs
print(fitdeepnet(input, printer = True))

# Generates wider (90, 10) neural net and prints hidden layer outputs
widedeephsizes = [1]
for i in range(1, 91):
    widedeephsizes.append(int(np.ceil(np.log2((91)/0.1))))
widedeepnet = Classes.Net(widedeephsizes, len(primes))
print(widedeepnet(input, printer = True))

# Fits wider neural nets and prints run time for fitting procedure
# and outputs
fitwidedeepnet = copy.deepcopy(widedeepnet)
Classes.fit(input, target, fitwidedeepnet, loss, 2000, 1e-3)
print(fitwidedeepnet(input))

# Generates a test set and evaluates the fitted neural nets
# and compares to best fitting constant to test data
[newprimes, testdata] = Data.sampler(10000, 64000, primes = primes)
testinput = testdata[:, [0]]
testinput = (testinput - min(testinput))/(max(testinput) - min(testinput))
testtarget = testdata[:, 1:]

print(loss(fitdeepnet(testinput), testtarget))
print(loss(fitwidedeepnet(testinput), testtarget))
print(sum(torch.var(testtarget, axis = 0, unbiased = False)))
print(loss(fitdeepnet(testinput), testtarget)/sum(torch.var(testtarget, axis = 0, unbiased = False)))
print(loss(fitwidedeepnet(testinput), testtarget)/sum(torch.var(testtarget, axis = 0, unbiased = False)))
```