

“NEURAL NETWORKS: ZERO TO HERO”: A MATHEMATICIAN’S COMPANION

AARON MAZEL-GEE

ABSTRACT. We summarize the mathematical content contained in Andrej Karpathy’s video lecture series *Neural Networks: Zero to Hero*, emphasizing the journey of developing these neural networks (as opposed to merely the end result). We assume a basic understanding of partial derivatives, linear algebra, and probability. **[incomplete draft]**

each section corresponds to a single lecture. colored material is to-be-written/finalized, and can be ignored.

CONTENTS

1. An Introduction to Neural Networks: Backpropagation, Multilayer Perceptrons, and Gradient Descent	2
1.0. Overview	2
1.1. Backpropagation	2
1.2. Multilayer Perceptrons	3
1.3. Training a Neural Network via Gradient Descent	5
1.4. Hyperparameters	7
2. Bigram Models: Combinatorial and Trained	8
2.0. Overview	8
2.1. The Combinatorial Bigram Model	8
2.2. The Trained Bigram Model	11
3. Fundamental Principles of Machine Learning: Embeddings, n -Gram Models, Overfitting/Underfitting, Stochastic Gradient Descent, and Hyperparameter Tuning	14
3.0. Overview	14
3.1. Embeddings	14
3.2. The Quadgram Model	15
3.3. Overfitting	15
3.4. Stochastic Gradient Descent	16
3.5. Learning Rate Tuning	17
3.6. Test Loss	18
3.7. More Hyperparameter Tuning	19
3.8. Final Training and Challenge	20
4. Initialization Optimization, Batch Normalization, and Diagnostics	20
4.0. Overview	20
4.1. Initialization Optimization	20
4.2. Batch Normalization	25
4.3. Diagnostics	29

1. AN INTRODUCTION TO NEURAL NETWORKS: BACKPROPAGATION, MULTILAYER PERCEPTRONS, AND GRADIENT DESCENT

1.0. Overview. In this section, we introduce *neural networks*. A neural network is a particular sort of function, whose behavior is controlled by the values of its *parameters*, just as e.g. the behavior of the function $f(x) := ax^2 + bx + c$ is controlled by its parameter values $a, b, c \in \mathbb{R}$.¹

Of course, in order for our neural network to be useful, it is essential that we find “good” values for the parameters. The process of doing so is referred to as *training* (or *optimizing*) the neural network.

The most mathematically sophisticated step in the training process amounts to the computation of certain partial derivatives. This computation is accomplished through a repeated application of the chain rule, which in the context of neural networks is implemented through an algorithm called *backpropagation*.

We begin in §1.1 with a prelude on backpropagation. We then discuss a particularly simply sort of neural network in §1.2, and in §1.3 we explain how to train such neural networks with a concrete example. We conclude in §1.4 by briefly discussing the notion of *hyperparameters*.

1.1. Backpropagation. Consider the function

$$w(x, y, z) := \left(x + \frac{x+y}{y \cdot z} \right)^2,$$

which assigns a real number to each point $(x, y, z) \in \mathbb{R}^3$ satisfying the inequality $y \cdot z \neq 0$. We can decompose this function as an iterated composite of functions according to the directed graph shown in Figure 1, which we refer to as a **computational graph** for the function. Given fixed input values for the variables x , y , and

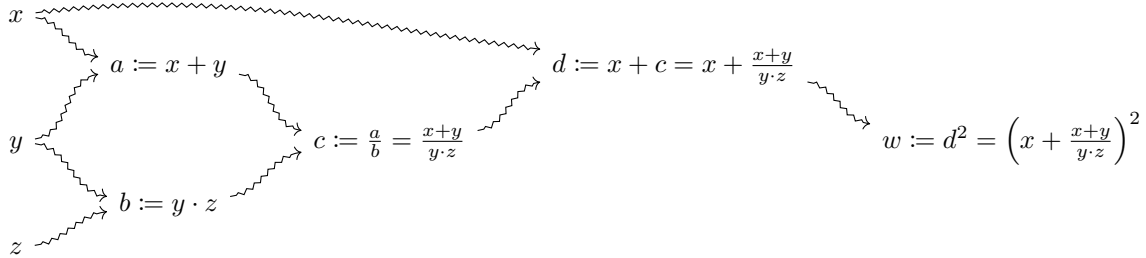


FIGURE 1. We decompose the function $w(x, y, z)$ as an iterated composition of functions, as depicted.

z , a **forward pass** (with respect to this choice of computational graph) is the calculation of $w = w(x, y, z)$ via the calculation of the intermediate variables a , b , c , and d . For example, if our input values are $x = 10$, $y = 2$, and $z = 3$, then a forward pass consists of computing that $a = 12$, $b = 6$, $c = 2$, $d = 14$, and $w = 196$.

Now, let us compute the partial derivative $\frac{dw}{dx}$ directly using the chain rule:

$$\begin{aligned} \frac{dw}{dx} &= \frac{dw}{dd} \cdot \left(\frac{dd}{dx} + \frac{dd}{dc} \cdot \left(\frac{dc}{da} \cdot \frac{da}{dx} + \frac{dc}{db} \cdot \frac{db}{dy} \right) \right) \\ &= 2d \cdot \left(1 + 1 \cdot \left(\frac{1}{b} \cdot 1 + \frac{-a}{b^2} \cdot 0 \right) \right) \\ &= 2d \cdot \left(1 + \frac{1}{b} \right). \end{aligned}$$

Just as we factorized the computation of w using a forward pass, we can likewise factorize the computation of $\frac{dw}{dx}$ using a **backward pass**: the computation of *all* partial derivatives $\frac{dw}{dt}$ for the variables $t \in \{x, y, z, a, b, c, d, w\}$ – that is, the nodes of the computational graph – in such an order that if a variable

¹In full generality, the (not completely precise) term “neural network” may be used to refer to any function whose definition involves “neurons” and connections between them (as in §1.2).

u depends on a variable t (which (as in Figure 1) we denote by $t \rightsquigarrow u$) then we compute $\frac{dw}{du}$ before computing $\frac{dw}{dt}$.² In general, if $\{u_i\}_i$ is the set of variables that directly depend on a variable t , then we have $\frac{dw}{dt} = \sum_i \frac{dw}{du_i} \cdot \frac{du_i}{dt}$. This formula explains the alternative term **backpropagation**: the values of the partial derivatives $\frac{dw}{du_i}$ propagate backward through the computational graph, all contributing to the value of the partial derivative $\frac{dw}{dt}$.

In our example, we perform a backward pass by computing (again using the chain rule):

$$\begin{aligned} \frac{dw}{dw} &= 1, \\ \frac{dw}{dd} &= 2d, \\ \frac{dw}{dc} &= \frac{dw}{dd} \cdot \frac{dd}{dc} = 2d \cdot 1 = 2d, \\ \frac{dw}{da} &= \frac{dw}{dc} \cdot \frac{dc}{da} = 2d \cdot \frac{1}{b} = \frac{2d}{b}, \quad \text{and finally} \\ \frac{dw}{dx} &= \frac{dw}{dd} \cdot \frac{dd}{dx} + \frac{dw}{da} \cdot \frac{da}{dx} = 2d \cdot 1 + \frac{2d}{b} \cdot 1 = 2d \cdot \left(1 + \frac{1}{b}\right). \end{aligned}$$

Again, this is nothing but a decomposition of the above direct computation of $\frac{dw}{dx}$. However, it is computationally advantageous if we intend to compute all three of the partial derivatives $\frac{dw}{dx}$, $\frac{dw}{dy}$, and $\frac{dw}{dz}$: for instance, $\frac{dw}{da}$ is also implicitly computed in the computation of $\frac{dw}{dy}$, and so it is more efficient to compute $\frac{dw}{da}$ just a single time. And indeed, for training neural networks we will want to efficiently compute the partial derivatives $\frac{dw}{dt}$ for *many* independent variables t which share *many* intermediate variables that depend on them.

1.2. Multilayer Perceptrons. We now work up to defining *multilayer perceptrons*, or MLPs for short. In general, a **neural network** is just a particular sort of function, and an MLP is a neural network with a particularly simple architecture; one is illustrated via a computational graph in Figure 2. The neural networks that we will actually study (after that of §1.3) will be iterative generalizations of MLPs (see e.g. Footnote 36).

First of all, a **neuron** with n inputs is a function $\mathbb{R}^n \xrightarrow{N} \mathbb{R}$ of the form $N(x_1, \dots, x_n) := f((\sum_i x_i w_i) + b)$ for some scalars $w_i, b \in \mathbb{R}$ and some function $\mathbb{R} \xrightarrow{f} \mathbb{R}$. We refer to the w_i as the **weights** and to b as the **bias** of the neuron; collectively these are called its **parameters**, and there are $n+1$ of them. We refer to f as its **activation function** (or alternatively its “nonlinearity function”); this is generally taken to be nonlinear (except possibly at the last layer, depending on the desired output of the MLP),³ and for the moment we will always just take this to be the function $f(t) := \tanh(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}}$ illustrated in [temp figure – illustrate a handful of activation functions here, for future reference]⁴

²Said differently, we can compute the partial derivatives $\frac{dw}{dt}$ as we let t run reversely through a topological ordering of the computational graph. (A *topological ordering* of a directed graph is a linear ordering of its vertices such that if there exists an edge $v_1 \rightsquigarrow v_2$ then v_1 precedes v_2 ; such an ordering exists if and only if the graph is directedly acyclic.) For example, one topological ordering (of many) of the computational graph depicted in Figure 1 is $y < z < b < x < a < c < d < w$. (Similarly, a forward pass must run (forward) through a topological ordering.)

³Indeed, if f is an affine function (i.e. the sum of a linear function and a constant function) then the neuron itself is an affine function, so that f can be absorbed into the weights and bias (i.e. it can be taken to be the identity function $f(t) := t$). Affine functions are closed under composition, so that e.g. an MLP whose neurons are all affine is itself affine (and hence practically useless in the context of machine learning).

⁴here, add a figure with a graph of $\tanh$. describe it (it’s a smooth and increasing diffeomorphism $\mathbb{R} \rightarrow (-1, 1)$). also mention (and graph) ReLU here, as another common choice. (this appears e.g. in §4.1.3.) [this is activation-functions.png \(a screenshot\)](#): use this for now, make my own later. say also: “We just use $\tanh$ (at least for now...?); we will discuss different possible choices of activation functions (and their relative merits?) in [ref later].”

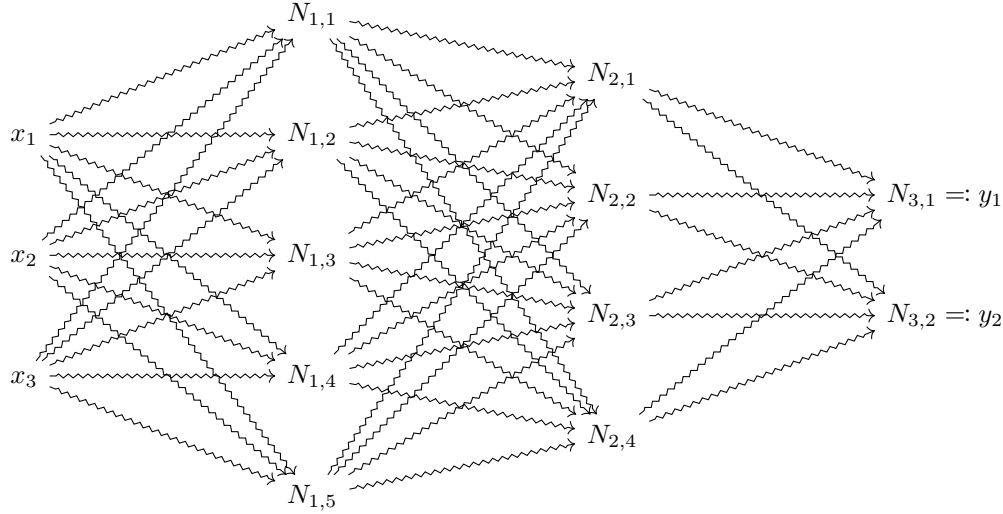


FIGURE 2. This diagram depicts a computational graph for a multilayer perceptron, which in this case is a function $\mathbb{R}^3 \xrightarrow{P} \mathbb{R}^2$: its inputs are the x_i and its outputs are the y_i . The nodes $N_{i,j}$ are its *neurons*, which are scalar functions with the indicated inputs. It has three layers $L_i = \{N_{i,j}\}_j$; the layers L_1 and L_2 are called *hidden layers*, while the layer L_3 is called its *output layer*.

Activation Functions

Sigmoid
 $\sigma(x) = \frac{1}{1+e^{-x}}$



Leaky ReLU
 $\max(0.1x, x)$



tanh
 $\tanh(x)$



Maxout
 $\max(w_1^T x + b_1, w_2^T x + b_2)$



ReLU
 $\max(0, x)$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Next, a **layer** is a function $\mathbb{R}^n \xrightarrow{L} \mathbb{R}^k$ defined by a set of k neurons $N_1, \dots, N_k$ that each have n inputs, namely $L(x_1, \dots, x_n) := (N_1(x_1, \dots, x_n), \dots, N_k(x_1, \dots, x_n))$. One may refer to k as the number of outputs of the layer. Although it is of course not strictly necessary, one generally assigns the same activation function to all neurons of a layer. We note for future reference that such a layer L has $(n+1) \cdot k$ parameters.⁵

Finally, a **multilayer perceptron** is a function $\mathbb{R}^n \xrightarrow{P} \mathbb{R}^m$ defined by a set of composable layers $L_1, \dots, L_{j+1}$ where L_1 has n inputs and L_{j+1} has m outputs, namely $P(x_1, \dots, x_n) := L_j \circ \dots \circ L_1(x_1, \dots, x_n)$. Of course, for the layers to be composable means that the number of outputs of L_i equals the number of inputs of L_{i+1} for all $1 \leq i \leq j$. The layers $L_1, \dots, L_j$ are the *hidden layers* of the MLP. So altogether, the shape of P is specified by the number n of inputs and the sizes $k_1, \dots, k_{j+1}$ of the layers; for instance, the MLP illustrated in Figure 2 has $n = 3$, $k_1 = 5$, $k_2 = 4$, and $k_3 = 2$ (with $j = 2$). Together, the shape of the MLP along with the choices of activation functions for its neurons are collectively referred to as its *architecture*. Thus, in order to completely specify P given its architecture, what remains is to specify the values of its parameters (i.e. those of its neurons).

We pause to condense our notation using linear algebra. Let us simply write $\vec{x} := (x_1, \dots, x_n)$ and $\vec{w} := (w_1, \dots, w_n)$. Then, the above neuron $\mathbb{R}^n \xrightarrow{N} \mathbb{R}$ can be written as $N(\vec{x}) = f(\vec{x}\vec{w}^T + b)$.⁶ Thereafter, suppose that a layer $\mathbb{R}^n \xrightarrow{L} \mathbb{R}^k$ consists of neurons $N_1, \dots, N_k$ with common activation function f , and that

⁵eventually, we'll use the term "layer" more loosely: for instance, we'll split up such a layer into an affine layer and a tanh layer, and intersperse a batchnorm layer between them.

⁶fix a notation for transpose, and use it here. maybe also replace all instances of dot product using it, too.

each N_i is specified by the vector $\vec{w}_i$ of weights and the bias b_i . Then, writing $W := (\vec{w}_1^T, \dots, \vec{w}_k^T)$ for the n -by- k matrix whose columns are the vectors $\vec{w}_i^T$ and writing $\vec{b} := (b_1, \dots, b_n)$, this layer can be written as $L(\vec{x}) = f(\vec{x}W + \vec{b})$ (where we apply f componentwise).⁷

1.3. Training a Neural Network via Gradient Descent.

1.3.0. *Overview.* We now discuss the process of *training* an MLP, which will apply essentially without change to the process of training more general neural networks. As a concrete example, we consider a three-layer MLP

$$P : \mathbb{R}^3 \xrightarrow{L_1} \mathbb{R}^4 \xrightarrow{L_2} \mathbb{R}^4 \xrightarrow{L_3} \mathbb{R}^1$$

in which all hidden neurons have activation function $\tanh$ and the output layer is linear: said differently, we have

$$P(\vec{x}) := \tanh(\tanh(\vec{x}W_1 + \vec{b}_1)W_2 + \vec{b}_2)W_3 + \vec{b}_3$$

for some matrices W_i and vectors $\vec{b}_i$ (where we implicitly apply $\tanh$ coordinatewise⁸).⁹ Given this fixed architecture, there are $(3+1) \cdot 4 + (4+1) \cdot 4 + (4+1) \cdot 1 = 41$ parameters that must be specified in order to specify P . In other words, writing Params for this 41-element set, each such MLP is uniquely specified by a point in the space $\text{ParamVals} := \mathbb{R}^{\text{Params}} \cong \mathbb{R}^{41}$ of functions from the set of parameters to the space of real numbers.¹⁰

We initialize these 41 parameter values as random numbers sampled from the uniform distribution $\mathcal{U}(-1, 1)$.¹¹ From here, we will *train* the MLP, meaning that we will iteratively update – or *tune* – these values: in other words, we will vary P by varying its corresponding point in ParamVals .¹² We will do so using the 4-element set of *training data* illustrated in Figure 3 (left),¹³ with the goal of making its *predictions* $\vec{y}_i^{\text{pred}} := P(\vec{x}_i)$ closely approximate the respective *ground truth* values $\vec{y}_i^{\text{gt}}$.

$\vec{x}_i$	$\vec{y}_i^{\text{gt}}$	$\vec{y}_i^{\text{pred}}, 0 \text{ epochs}$	$\vec{y}_i^{\text{pred}}, 20 \text{ epochs}$	$\vec{y}_i^{\text{pred}}, 80 \text{ epochs}$
(2.0, 3.0, −1.0)	1.0	−0.2940	0.8794	0.9559
(3.0, −1.0, 0.5)	−1.0	0.0535	−0.9324	−0.9705
(0.5, 1.0, 1.0)	−1.0	−0.0850	−0.8303	−0.9410
(1.0, 1.0, −1.0)	1.0	−0.1067	0.8429	0.9374

FIGURE 3. The table on the left displays the training data for our MLP $\mathbb{R}^3 \xrightarrow{P} \mathbb{R}^1$, which consists of four input vectors $\vec{x}_i \in \mathbb{R}^3$ along with their corresponding *ground truth* values $\vec{y}_i^{\text{gt}} \in \mathbb{R}^1$ – the desired output vectors of P . The table on the right displays the respective predictions $\vec{y}_i^{\text{pred}} \in \mathbb{R}^1$ after training the MLP with learning rate 0.05 for 0, 20, and 80 epochs; the respective values of the loss function ℓ are 4.8463, 0.0726, and 0.0102.

⁷or say this once and for all in intro, perhaps when mentioning tensors (and our general avoidance of them).

⁸or say this once and for all in the intro, perhaps when mentioning tensors (and our general avoidance of them).

⁹For now, we remain in a purely mathematical realm: the inputs and outputs of P have no specific meaning. We will see our first “meaningful” neural network (and the process of training it) in §2.

¹⁰Beware that it is common not to distinguish between the *parameters* (which here form a 41-element set) and their *values* (each of which is a real number) that collectively determine the MLP: that is, it is common to refer to a parameter value simply as a parameter. The notation ParamVals emphasizes that this is the space of parameter *values*, but in order to avoid awkward language we henceforth often elide this distinction as well.

¹¹Such (pseudo)random numbers can be generated using a “random seed”, which completely determines them and thus allows for reproducibility (e.g. of the results in [these videos]). However, beware that the effect of a given random seed can vary e.g. across different versions of Python. [probably move this to intro.]

¹²In particular, beware of the potentially confusing terminology: when one speaks of training *an* MLP, one is in fact moving through a *space* of MLPs. (More elaborately, one could write e.g. $P_0, P_1, P_2, \dots$ for the sequence of MLPs generated during the training process, each of which (after the randomly chosen P_0) is obtained from the previous by a single update of the parameter values.)

¹³Here and henceforth we may refer to a “set” of data (or to a “dataset”), but beware that this need not literally be a set since it may contain repetitions. Rather, a dataset may be thought of as a *list* of datapoints, or alternatively as a *multiset* (to avoid any semblance of an ordering). We may also use the term “list” if we wish to emphasize the possibility of repetitions.

1.3.1. *Gradient Descent.* Of course, we must first quantify the notion of “closeness” between a set $\{\bar{y}_i^{\text{pred}}\}_i$ of predictions (resulting from specific values of the parameters) and the set $\{\bar{y}_i^{\text{gt}}\}_i$ of ground truth values. For this, we define the **loss** for this set of predictions to be $\ell := \sum_i |\bar{y}_i^{\text{pred}} - \bar{y}_i^{\text{gt}}|^2$; note that this is always nonnegative, and is zero precisely when we have equalities $\bar{y}_i^{\text{pred}} = \bar{y}_i^{\text{gt}}$ for all i .¹⁴ Allowing the parameters to vary, we may consider this as defining a loss function $\text{ParamVals} \xrightarrow{\ell} \mathbb{R}$ (inasmuch as the predictions $\bar{y}_i^{\text{pred}}$ depend on the parameter values). Our goal, then, is to find the parameter values that minimize the value of ℓ .

Recall that if a smooth function $\mathbb{R}^n \xrightarrow{f} \mathbb{R}$ has a minimum at some point $\vec{x} \in \mathbb{R}^n$, then it must be the case that its corresponding **gradient vector** $\nabla f(\vec{x}) := (\frac{df}{dx_i}(\vec{x}))_{1 \leq i \leq n}$ (whose i^{th} entry is the partial derivative $\frac{df}{dx_i}$ evaluated at $\vec{x}$) must be zero.¹⁵ This leads to the standard method of minimization that one is taught in multivariable calculus class, which begins by first finding all local minima by solving the equation $\nabla f(\vec{x}) = \vec{0}$ (with respect to the variable $\vec{x} \in \mathbb{R}^n$). However, this method is not feasible if the function f is even moderately complicated, as is certainly the case with our loss function $\mathbb{R}^{41} \cong \text{ParamVals} \xrightarrow{\ell} \mathbb{R}$.

An alternative approach is the method of **gradient descent**. Starting at some point $\vec{x} \in \mathbb{R}^n$, the gradient vector $\nabla f(\vec{x})$ points in the direction in which f increases most rapidly;¹⁶ likewise, its negative $-\nabla f(\vec{x})$ points in the direction in which f decreases most rapidly. Thus, if we wish to decrease the value of f , we can simply move from $\vec{x}$ in the direction of the negative gradient vector $-\nabla f(\vec{x})$. Note that this does *not* require the solution of any complicated equations, but rather merely the computation of the partial derivatives of f at $\vec{x}$.

And indeed, training our MLP amounts to performing gradient descent for the corresponding point $\vec{\theta} \in \text{ParamVals} \cong \mathbb{R}^{41}$ (which encodes the parameters’ current values) with respect to the loss function ℓ . This is implemented by repeatedly performing the following three-step procedure.

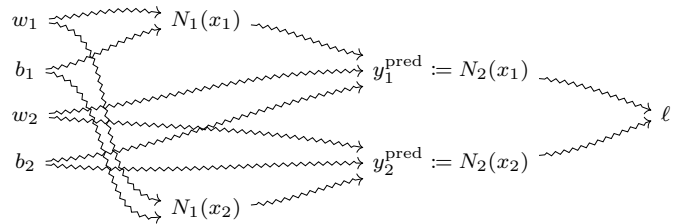
- (1) A **forward pass** amounts to computing the loss ℓ (and in particular the predictions $\bar{y}_i^{\text{pred}}$), based on the current value of $\vec{\theta} \in \text{ParamVals}$.
- (2) A **backward pass** amounts to computing the partial derivative values $\frac{d\ell}{dp}(\vec{\theta})$ with respect to the parameters $p \in \text{Params}$ via backpropagation.^{17,18}

¹⁴This loss is (up to a factor of $\frac{1}{4}$) called the *mean squared error* of the predictions. Of course, we could instead reasonably take our loss to be the sum $\sum_i |\bar{y}_i^{\text{pred}} - \bar{y}_i^{\text{gt}}|$. Note too that we have made the choice to weight all of these squared error terms equally. More generally, we could reasonably take our loss to be the sum $\sum_i c_i \cdot |\bar{y}_i^{\text{pred}} - \bar{y}_i^{\text{gt}}|^{e_i}$ for any positive coefficients c_i and exponents e_i .

¹⁵Indeed, if some partial derivative $\frac{df}{dx_i}(\vec{x})$ is nonzero, then slightly changing the i^{th} coordinate of $\vec{x}$ can yield a lower value of f . (If $\frac{df}{dx_i}(\vec{x})$ is positive (resp. negative) then this is achieved by slightly lowering (resp. raising) the i^{th} coordinate of $\vec{x}$.)

¹⁶More precisely, assuming the gradient vector is nonzero, it points in the direction in which the infinitesimal rate of change (i.e. the directional derivative) of f is largest, and its magnitude is precisely that largest rate.

¹⁷Note that this backpropagation proceeds through the computational graph of ℓ , considering the parameters Params as independent variables and with each neuron contributing one dependent variable for each datapoint – in stark contrast with the computational graph of the MLP itself. For a very simple example, consider a two-layer MLP $\mathbb{R} \xrightarrow{N_1} \mathbb{R} \xrightarrow{N_2} \mathbb{R}$ defined by neurons $N_i(x) := f_i(xw_i + b_i)$. This MLP is naturally described by the computational graph $x \rightsquigarrow N_1 \rightsquigarrow y := N_2$. Suppose that we have two datapoints (x_1, y_1^{gt}) and (x_2, y_2^{gt}) in our dataset, and that we take our loss function to be $\ell := \sum_i |y_i^{\text{pred}} - y_i^{\text{gt}}|^2$. Then, our loss function $\mathbb{R}^4 \cong \text{ParamVals} \xrightarrow{\ell} \mathbb{R}$ is naturally described by the computational graph



¹⁸A computational convenience of using $\tanh$ as an activation function is the fact that $\frac{d}{dt} \tanh(t) = 1 - \tanh^2(t)$. [or probably actually emphasize that this is *built in* to our “backprop engine”. Python does *not* secretly know how to do calculus; rather, it can compute partial derivatives based on the backpropagation algorithm.]

- (3) An **update step** amounts to changing the value of $\vec{\theta} \in \text{ParamVals}$ according to the prescription $\Delta\vec{\theta} = -r \cdot \nabla\ell(\vec{\theta})$, where r is some specified **learning rate**.

Each run-through of this procedure is called an **epoch**. Arbitrarily choosing our learning rate to be $r = 0.05$, we obtain predictions from the MLP after 0, 20, and 80 epochs as illustrated in Figure 3 (right).

1.3.2. Learning Rates. Determining an appropriate learning rate r is an important and subtle problem, which we will address more systematically in §3.5 [or also again later?]. Nevertheless, we can already illustrate its importance and subtlety.

For this, it can be instructive to envision the process of gradient descent for a function $\mathbb{R}^2 \xrightarrow{f} \mathbb{R}$. We can think of $\mathbb{R}^2$ as a map of some mountainous terrain, with f describing the altitude at each point of the terrain; the surface $\Gamma := \{(\vec{x}, f(\vec{x}))\} \subset \mathbb{R}^2 \times \mathbb{R} \cong \mathbb{R}^3$ (i.e. the graph of f) plays the role of the terrain itself. The negative gradient vector $\vec{v} := -\nabla f(\vec{x})$ in $\mathbb{R}^2$ points in the direction of steepest descent along Γ from the point $p := (\vec{x}, f(\vec{x}))$. In these terms, the learning rate determines the sizes of our steps as we walk around on this mountainous terrain Γ .

Now, if our learning rate is too large, we may end up moving back uphill as we step away from p : if the terrain Γ is complicated, then $\vec{v}$ may only point downhill in a relatively small neighborhood around p .¹⁹ On the other hand, if our learning rate is too small, then it may take us a very long time to move downhill. That is, it may take us a great many steps to move downhill, and correspondingly it may take us a great deal of time and computational effort (since the forward and backward passes can be quite computationally intensive).

Moreover, a learning rate that is too small may cause us to become caught in a small dip in the terrain: a small mountain lake. A larger learning rate makes it more likely that we will step past such a dip, or alternatively that if we do happen to land in it we will nevertheless step out of it (since it is unlikely that we will land *exactly* at the local minimum that it contains (where the gradient is zero)). Another way of mitigating the risk posed by mountain lakes is to instead “perturb” our steps according to some small randomly chosen vector (which changes with each step). Yet another way of introducing some helpful randomness into the process is to perform *stochastic* gradient descent, as described in §3.4.

This discussion illustrates the issue that gradient descent is by no means guaranteed to find a *global* minimum. Indeed, a priori one may be worried that it is difficult to avoid becoming trapped in a sufficiently wide mountain lake, and never discovering the lower values of f that may lie outside of its bounds. However, this unfortunate circumstance tends to become less and less likely as the number of parameters grows, first and foremost because a local minimum can only occur at $\vec{x}$ if the partial derivatives $\frac{df}{dx_i}(\vec{x})$ are zero for *all* parameters x_i , and it is reasonable to hope that these values are not too closely correlated. And thereafter, even if $\frac{df}{dx_i}(\vec{x}) = 0$ for all parameters x_i , local minimality is obstructed e.g. by the existence of a single parameter x_i for which $\frac{d^2f}{dx_i^2}(\vec{x}) < 0$.

1.4. Hyperparameters. Constructing a neural network involves a plethora of design choices. Among these choices, the numerical ones are collectively referred to as **hyperparameters**. We have already seen some hyperparameters: the number of layers in an MLP, the number of neurons in a layer, and the learning rate. In particular, note that hyperparameters can play a role in both the architecture of a neural network and in its training process. One’s choices of hyperparameters can bear greatly on the ultimate effectiveness of the neural network, as well as on the amount of time and computing power needed to train it.

Recall that the training process for a neural network involves *tuning* its parameters: changing them bit by bit, and keeping track of how these changes affect the neural network’s performance (as measured by the loss function). Analogously but one layer more abstractly, one may perform **hyperparameter tuning** in order to optimize the hyperparameters (relative to one’s various and potentially competing goals): changing the hyperparameters bit by bit, and keeping track of how these changes affect the neural network’s training process and ultimate effectiveness. This will be a recurring topic of discussion for us.²⁰

¹⁹Said differently, stepping in the direction of $\vec{v}$ with the hope of reducing the value of f involves the implicit assumption that f is well-approximated by its *closest linear approximation* at $\vec{x}$: the function whose graph is the plane that is tangent to Γ at p . (Concretely, this is given by the formula $L_{f,\vec{x}}(\vec{t}) := \nabla f(\vec{x}) \bullet (\vec{t} - \vec{x}) + f(\vec{x})$.) However, in general this is only a good approximation to f in a small neighborhood around $\vec{x}$. (The size of this neighborhood (tautologically) depends on the extent to which f deviates from being linear near $\vec{x}$, which is in turn measured by its various *second* partial derivatives at $\vec{x}$.)

²⁰[specific refs, or is this just for ever and always?](#)

2. BIGRAM MODELS: COMBINATORIAL AND TRAINED

2.0. Overview. In this section, we discuss a very simple neural network which is intended to help generate new names. We will build it based upon a dataset of 32,033 names, which begins as follows: `emma, olivia, ava, isabella, sophia`.²¹

An extremely simple method of generating names would be to randomly choose their letters one by one. Of course, we must have some way of deciding when to *stop* randomly choosing letters, and so we introduce an additional special character `*` that indicates the end of a name. Then, sampling repeatedly from the uniform distribution on the set $\text{Chars} := \{\text{a}, \text{b}, \text{c}, \dots, \text{y}, \text{z}, *\}$ (so that at each stage each character is selected with probability $\frac{1}{27}$), we obtain e.g. the “name” `qvsaayxbqrqmyqwuznivanukotdjdvdhd*`.²²

Such an absurd example makes clear two immediate deficiencies of this method. First of all, we should not give all possible choices of the next character equal probability; for example, letters that appear more frequently in actual names (i.e. those in our dataset) should appear with higher probability in our newly generated names. And moreover, it would be better if the chosen letter depends on some *context* that precedes it; for example, the letter `v` probably shouldn’t follow the letter `q`!

We will use a neural network to simultaneously address these two deficiencies in the simplest possible way. First of all, at each step, our neural network will provide a not-necessarily-uniform probability distribution on the set Chars from which we will sample in order to choose the next character. And moreover, our neural network will determine this probability distribution based upon the context of the previous letter. For this reason, it is called a **bigram model**; a *bigram* is simply a length-2 string of characters, e.g. `ax`.²³ When these probability distributions are well chosen, the bigram model will produce new “names” that look much more reasonable, such as: `dontell, khylum, camatena, aeriline, najlah`.²⁴

Note that we must also introduce a special starting character, so that there is “context” for choosing the first letter of a name. For this, there is no reason not to simply reuse the special character `*`.²⁵ Altogether, then, our dataset of names parses into a dataset of bigrams, which begins as follows: `*e, em, mm, ma, a*, *o, ol, li`.

In fact, we will discuss *two* bigram models built from our dataset of $\sim 32\text{k}$ names: a “combinatorial” one in §2.1, and thereafter a “trained” one (in the sense of §1.3) in §2.2. However, it will turn out that these two models are essentially identical, as we explain in §2.2.2.

2.1. The Combinatorial Bigram Model.

2.1.0. The Model. The construction of the combinatorial bigram model is extremely simple: in order to determine the probability distribution associated to a given context $x \in \text{Chars}$, we simply count the number of bigrams that begin with the character x and then rescale these counts to obtain a probability distribution (by dividing them all by their sum). The counts of all bigrams in our dataset are displayed in Figure 4. For example, the bigram `*a` appears 4,410 times, which means that there are 4,410 names that start with `a`. Since there are 32,033 bigrams in the dataset that begin with `*` (as each name contributes exactly one such bigram), rescaling the uppermost row to a probability distribution assigns a probability of $\frac{4410}{32033} = 13.77\%$ for the character `*` to be followed by the character `a`: this is the probability that a name generated by our model will begin with the letter `a`. The probability distributions associated to a few contexts are displayed in Figure 5.

Sampling from the model, the first name that we obtain is `mor`. Unpacking the process, this means that:

- sampling from the the probability distribution associated to the context `*` (the 0th row in Figure 5) yielded the character `m`;
- then, sampling from the probability distribution associated to the context `m` (the 1st row in Figure 5) yielded the character `o`;

²¹To slightly simplify our discussion, we consider names as consisting of all-lowercase letters.

²²This name has 32 letters (not counting the special character `*`). This may seem rather long, but in fact there is a 29.89% chance that a name so selected has 32 or more letters. (In general, the probability of such a name having at most n letters is $\frac{1}{27} + \frac{26}{27} \cdot \frac{1}{27} + \left(\frac{26}{27}\right)^2 \cdot \frac{1}{27} + \dots + \left(\frac{26}{27}\right)^n \cdot \frac{1}{27} = \sum_{i=0}^n \left(\frac{26}{27}\right)^i \cdot \frac{1}{27} = 1 - \left(\frac{26}{27}\right)^{n+1}$.)

²³In particular, we may also refer to our neural network simply as a *model*.

²⁴We employ a standard abuse of terminology: as a shorthand, we say that the model actually produces those results (when in actuality only produces probability distributions that we then sample from in order to obtain those results).

²⁵In order to avoid the possibility of a “name” having zero ordinary letters, one must simply set the probability of the character `*` following itself to be 0.

0	*a	*b	*c	*d	*e	*f	*g	*h	*i	*j	*k	*l	*m	*n	*o	*p	*q	*r	*s	*t	*u	*v	*w	*x	*y	*z
a*	aa	ab	ac	ad	ae	af	ag	ah	ai	aj	ak	al	am	an	ao	ap	aq	ar	as	at	au	av	aw	ax	ay	az
6640	556	541	470	1042	692	134	168	2332	1650	175	568	2528	1634	5438	63	82	60	3264	1118	687	381	834	161	182	2650	435
b* 114	ba 321	bb 38	bc 1	bd 65	be 655	bf 0	bg 0	bh 41	bi 217	bj 1	bk 0	bl 103	bm 0	bn 4	bo 105	bp 0	bq 0	br 842	bs 8	bt 2	bu 45	bv 0	bw 0	bx 0	by 83	bz 0
c* 97	ca 815	cb 0	cc 42	cd 1	ce 551	cf 0	cg 2	ch 664	ci 271	cj 3	ck 316	cl 116	cm 0	cn 0	co 380	cp 1	cq 11	cr 76	cs 5	ct 35	cu 35	cv 0	cw 0	cx 3	cy 104	cz 4
d* 516	da 1303	db 1	dc 3	dd 149	de 1283	df 5	dg 25	dh 118	di 674	dj 9	dk 3	dl 60	dm 30	dn 31	do 378	dp 0	dq 1	dr 424	ds 29	dt 4	du 92	dv 17	dw 23	dx 0	dy 317	dz 1
e* 3983	ea 679	eb 121	ec 153	ed 384	ee 1271	ef 82	eg 125	eh 152	ei 818	ej 55	ek 178	el 3248	em 769	en 2675	eo 269	ep 83	eq 14	er 1958	es 861	et 580	eu 69	ev 463	ew 50	ex 132	ey 1070	ez 181
f* 80	fa 242	fb 0	fc 0	fd 0	fe 123	ff 44	fg 1	fh 1	fi 160	fj 0	fk 2	fl 20	fm 0	fn 4	fo 60	fp 0	fq 0	fr 114	fs 6	ft 18	fu 10	fv 0	fw 4	fx 0	fy 14	fz 2
g* 108	ga 330	gb 3	gc 0	gd 19	ge 334	gf 1	gg 25	gh 360	gi 190	gj 3	gk 0	gl 32	gm 6	gn 27	go 83	gp 0	gq 0	gr 201	gs 30	gt 31	gu 85	gv 1	gw 26	gx 0	gy 31	gz 1
h* 2409	ha 2244	hb 8	hc 2	hd 24	he 674	hf 2	hg 2	hh 1	hi 729	hj 9	hk 29	hl 185	hm 117	hn 138	ho 287	hp 1	hq 1	hr 204	hs 31	ht 71	hu 166	hv 39	hw 10	hx 0	hy 213	hz 20
i* 2489	ia 2445	ib 110	ic 509	id 440	ie 1653	if 101	ig 428	ih 95	ii 82	ij 76	ik 445	il 1345	im 427	in 2126	io 588	ip 53	iq 52	ir 849	is 1316	it 541	iu 109	iv 269	iw 8	ix 89	iy 779	iz 277
j* 71	ja 1473	jb 1	jc 4	jd 4	je 440	jf 0	fg 0	jh 45	ji 119	jj 2	jk 2	jl 9	jm 5	jn 2	jo 479	jp 1	jq 0	jr 11	js 7	jt 2	ju 202	jv 5	jw 6	jx 0	iy 10	jz 0
k* 363	ka 1731	kb 2	kc 2	kd 2	ke 895	kf 1	kg 0	kh 307	ki 509	kj 2	kk 20	kl 139	km 9	kn 26	ko 344	kp 0	kq 0	kr 109	ks 95	kt 17	ku 50	kv 2	kw 34	kx 0	ky 379	kz 2
l* 1314	la 2623	lb 52	lc 25	ld 138	le 2921	lf 22	lg 6	lh 19	li 2480	lj 6	lk 24	ll 1345	lm 60	ln 14	lo 692	lp 15	lq 3	lr 18	ls 94	lt 77	lu 324	lv 72	lw 16	lx 0	ly 1588	lz 10
m* 516	ma 2590	mb 112	mc 51	md 24	me 818	mf 1	mg 0	mh 5	mi 1256	mj 7	mk 1	ml 5	mm 168	mn 20	mo 452	mp 38	mq 0	mr 97	ms 35	mt 4	mu 139	mv 3	mw 2	mx 0	my 287	mz 11
n* 6763	na 2977	nb 8	nc 213	nd 704	ne 1359	nf 11	ng 273	nh 26	ni 1725	nj 44	nk 58	nl 195	nm 19	nn 1996	no 496	np 5	nq 2	nr 44	ns 278	nt 443	nu 96	nv 55	nw 11	nx 6	ny 465	nz 145
o* 855	oa 149	ob 140	oc 114	od 190	oe 132	of 34	og 44	oh 171	oi 69	oj 16	ok 68	ol 619	om 261	on 2411	oo 115	op 95	oq 3	or 1059	os 504	ot 118	ou 275	ov 176	ow 114	ox 45	oy 103	oz 54
p* 33	pa 209	pb 2	pc 1	pd 0	pe 197	pf 1	pg 0	ph 204	pi 61	pj 1	pk 1	pl 16	pm 1	pn 1	po 59	pp 39	pq 0	pr 151	ps 16	pt 17	pu 4	pv 0	pw 0	px 0	py 12	pz 0
q* 28	qa 13	qb 0	qc 0	qd 0	qe 1	qf 0	qg 0	qh 0	qi 13	qj 0	qk 0	ql 1	qm 2	qn 0	qo 2	qp 0	qq 0	qr 1	qs 2	qt 0	qu 206	qv 0	qw 3	qx 0	qy 0	qz 0
r* 1377	ra 2356	rb 41	rc 99	rd 187	re 1697	rf 9	rg 76	rh 121	ri 3033	rj 25	rk 90	rl 413	rm 162	rn 140	ro 869	rp 14	rq 16	rr 425	rs 190	rt 208	ru 252	rv 80	rw 21	rx 3	ry 773	rz 23
s* 1169	sa 1201	sb 21	sc 60	sd 9	se 884	sf 2	sg 2	sh 1285	si 684	sj 2	sk 82	sl 279	sm 90	sn 24	so 531	sp 51	sq 1	sr 55	ss 461	st 765	su 185	sv 14	sw 24	sx 0	sy 215	sz 10
t* 483	ta 1027	tb 1	tc 17	td 0	te 716	tf 2	tg 2	th 647	ti 532	tj 3	tk 0	tl 134	tm 4	tn 22	to 667	tp 0	tq 0	tr 352	ts 35	tt 374	tu 78	tv 15	tw 11	tx 2	ty 341	tz 105
u* 155	ua 163	ub 103	uc 103	ud 136	ue 169	uf 19	ug 47	uh 58	ui 121	uj 14	uk 93	ul 301	um 154	un 275	uo 10	up 16	uq 10	ur 414	us 474	ut 82	uu 3	uv 37	uw 86	ux 34	uy 13	uz 45
v* 88	va 642	vb 1	vc 0	vd 1	ve 568	vf 0	vg 0	vh 1	vi 911	vj 0	vk 3	vl 14	vm 0	vn 8	vo 153	vp 0	vq 0	vr 48	vs 0	vt 0	vu 7	vv 7	vw 0	vx 0	vy 121	vz 0
w* 51	wa 280	wb 1	wc 0	wd 8	we 149	wf 2	wg 1	wh 23	wi 148	wj 0	wk 6	wl 13	wm 2	wn 58	wo 36	wp 0	wq 0	wr 22	ws 20	wt 8	wu 25	wv 0	ww 2	wx 0	wy 73	wz 1
x* 164	xa 103	xb 1	xc 4	xd 5	xe 36	xf 3	xg 0	xh 1	xi 102	xj 0	xk 0	xl 39	xm 1	xn 1	xo 41	xp 0	xq 0	xr 0	xs 31	xt 70	xu 5	xv 0	xw 3	xx 38	xy 30	xz 19
y* 2007	ya 2143	yb 27	yc 115	yd 272	ye 301	yf 12	yg 30	yh 22	yi 192	yj 23	yk 86	yl 1104	ym 148	yn 1826	yo 271	yp 15	yq 6	yr 291	ys 401	yt 104	yu 141	yv 106	yw 4	yx 28	yy 23	yz 78
z* 160	za 860	zb 4	zc 2	zd 2	ze 373	zf 0	zg 1	zh 43	zi 364	zj 2	zk 2	zl 123	zm 35	zn 4	zo 110	zp 2	zq 0	zr 32	zs 4	zt 4	zu 73	zv 2	zw 3	zx 1	zy 147	zz 45

FIGURE 4. This table displays the counts of the $27^2 = 729$ bigrams in our dataset. The 27 rows correspond to the 27 possible contexts. The darkness of each cell correlates with the frequency of the corresponding bigram.

- then, sampling from the probability distribution associated to the context **o** (the 2nd row in Figure 5) yielded the character **r**;
- finally, sampling from the probability distribution associated to the context **r** (the 3rd row in Figure 5) yielded the character *****.

Referring back to Figure 5, we see that the *likelihood* of this occurrence is $0.079 \cdot 0.068 \cdot 0.133 \cdot 0.108 = 7.716 \cdot 10^{-5}$. The next few names that arise are **axx**, **minaymoryles**, **kondlaisah**, and **anchshizarie**. Sampling a bit further, among the “names” that we obtain are the one-letter names **h**, **f**, and **r**. Although these are rather absurd as names, they are not terribly unlikely outputs of our model. For instance, again

**	*a	*b	*c	*d	*e	*f	*g	*h	*i	*j	*k	*l	*m	*n	*o	*p	*q	*r	*s	*t	*u	*v	*w	*x	*y	*z
0.0	0.138	0.041	0.048	0.053	0.048	0.013	0.021	0.027	0.018	0.076	0.092	0.049	0.079	0.036	0.012	0.016	0.003	0.051	0.064	0.041	0.002	0.012	0.01	0.004	0.017	0.029
m*	ma	mb	mc	md	me	mf	mg	mh	mi	mj	mk	ml	mm	mn	mo	mp	mq	mr	ms	mt	mu	mv	mw	mx	my	mz
0.078	0.389	0.017	0.008	0.004	0.123	0.0	0.0	0.001	0.188	0.001	0.0	0.001	0.025	0.003	0.068	0.006	0.0	0.015	0.005	0.001	0.021	0.001	0.0	0.0	0.043	0.002
o*	oa	ob	oc	od	oe	of	og	oh	oi	oj	ok	ol	om	on	oo	op	oq	or	os	ot	ou	ov	ow	ox	oy	oz
0.108	0.019	0.018	0.014	0.024	0.017	0.004	0.006	0.022	0.009	0.002	0.009	0.078	0.033	0.303	0.015	0.012	0.001	0.133	0.063	0.015	0.035	0.022	0.014	0.006	0.013	0.007
r*	ra	rb	rc	rd	re	rf	rg	rh	ri	rj	rk	rl	rm	rn	ro	rp	rq	rr	rs	rt	ru	rv	rw	rx	ry	rz
0.108	0.185	0.003	0.008	0.015	0.133	0.001	0.006	0.01	0.235	0.002	0.007	0.033	0.013	0.011	0.068	0.001	0.001	0.033	0.015	0.016	0.02	0.006	0.002	0.0	0.061	0.002

FIGURE 5. This table displays the probability distributions in the combinatorial bigram model associated to the contexts $*$, m , o , and r .

referring back to Figure 5, we see that the likelihood of obtaining the name r is $0.051 \cdot 0.108 = 5.508 \cdot 10^{-3}$: this name is ~ 100 times more likely to occur from sampling from our model than the name mor .

We now proceed to describe our model as an MLP. For this, let us index our characters according to the bijection $\text{Chars} \xrightarrow{\text{ind}} \underline{27}^{26}$ given by $* \mapsto 0$, $a \mapsto 1$, $b \mapsto 2$, $c \mapsto 3$, etc. Then, we can associate to each character $x \in \text{Chars}$ its **one-hot encoding**: the standard basis vector $\vec{e}_{\text{ind}(x)} \in \mathbb{R}^{27}$ (which has a 1 in the $\text{ind}(x)^{\text{th}}$ coordinate and a 0 in all other coordinates).²⁷ Moreover, let us write P for the 27-by-27 matrix obtained by rescaling the rows of the table in Figure 4 to probability distributions (so that the table in Figure 5 displays its 0th, 13th, 15th, and 18th rows). Then, the probability distribution on Chars associated to each context $x \in \text{Chars}$ is encoded by the vector $\vec{e}_{\text{ind}(x)}P$: for each $y \in \text{Chars}$, the probability assigned to y is its $\text{ind}(y)^{\text{th}}$ coordinate $(\vec{e}_{\text{ind}(x)}P)_{\text{ind}(y)}$, which is of course just the matrix entry $P_{\text{ind}(x), \text{ind}(y)}$. All of this is to say that we can consider our combinatorial bigram model as a 1-layer linear MLP $\mathbb{R}^{27} \rightarrow \mathbb{R}^{27}$ whose biases are all 0 (namely multiplication by the weight matrix P), into which we only ever intend to input standard basis vectors and whose outputs (when fed such inputs) land in the space $\Delta^{26} \subset \mathbb{R}^{27}$ of probability distributions on a 27-element set, which (via the inverse of the bijection $\text{Chars} \xrightarrow{\text{ind}} \underline{27}$) we consider as a probability distribution on the set Chars .^{28,29}

2.1.1. *Loss.* Although we are not training the combinatorial bigram model, we can nevertheless contemplate an appropriate notion of “loss” for it. Indeed, this will serve as our loss function when we construct our trained bigram model in §2.2.

Specifically, we would like the loss to correlate with the likelihood of the dataset of names occurring. Writing BigramData for the list of bigrams that constitutes our dataset, this likelihood is the product $\prod_{\beta \in \text{BigramData}} P_{\text{ind}(\beta)}$; here, we simply write ind for the function $\text{Chars}^2 \xrightarrow{\text{ind}^2} \underline{27}^2$, so that $P_{\text{ind}(\beta)}$ denotes the entry of the matrix P corresponding to the bigram β (namely the probability of choosing the second character of β given its first character as context). To arrive at our formula for the loss, we make the following iterative refinements.

First of all, this likelihood will be absolutely miniscule.³⁰ Thus, trying to compute it directly on a computer would lead to floating-point errors. This leads us to instead contemplate its logarithm $\log(\prod_{\beta \in \text{BigramData}} P_{\text{ind}(\beta)}) = \sum_{\beta \in \text{BigramData}} \log(P_{\text{ind}(\beta)})$: the *log likelihood* of the dataset. Because the logarithm function $\mathbb{R}_{>0} \xrightarrow{\log} \mathbb{R}$ is strictly increasing everywhere (recall that its derivative at any point $t \in \mathbb{R}_{>0}$ is $\frac{d \log(t)}{dt} = \frac{1}{t}$ (which is always positive)), maximizing the likelihood is equivalent to maximizing the log likelihood. And crucially, this sum of logarithms will not lose any accuracy even if it is very negative.

²⁶possibly recall here (from intro notation stuff) that this is **arange**, i.e. $\underline{n} := \{0, 1, 2, \dots, n-1\}$.

²⁷The function ind already converts characters into numerical values, and so one *could* feed its values directly into a neural network. However, the values of ind are not really appropriate as inputs to a neural network, since they retain the vestigial information of the ordering of our alphabet. By contrast, one-hot encoding is independent of this ordering up to a change of basis (in fact up to a permutation matrix), so that this ordering does not bear in any way upon the expressive power of our neural network.

²⁸it’s kinda asymmetrical to connect with actual characters only at the input and not at the output. change this, or one way or the other? or if not, say so more clearly and maybe give a justification for why.

²⁹possibly describe this factorization of $\text{Chars} \xrightarrow{\vec{e}_{\text{ind}(-)}} \mathbb{R}^{27} \xrightarrow{P} \mathbb{R}^{27}$ through Δ^{26} diagrammatically. (but later, beware that $\text{softmax } \mathbb{R}^{27} \xrightarrow{\sigma} \Delta^{26}$ is *not* a retraction.)

³⁰Most of the values of $P_{\text{ind}(\beta)}$ are around 0.1 or lower (recall Figure 5), and BigramData has 228,146 entries.

On the other hand, note that a higher likelihood should correspond to a *lower* value of loss, since our aim will be to *decrease* the value of our loss function. This leads us to instead consider the *negative* log likelihood $-\sum_{\beta \in \text{BigramData}} \log(P_{\text{ind}(\beta)})$.

Finally, it will be convenient for our loss to be *scale-invariant*, so that we can make meaningful comparisons amongst the overall loss and the losses associated to individual names or more generally to lists of names (as e.g. in §3.6). This consideration leads us to our ultimate choice for our loss, namely the **average negative log likelihood** $\ell := \frac{-1}{|\text{BigramData}|} \sum_{\beta \in \text{BigramData}} \log(P_{\text{ind}(\beta)})$. More generally, for any list of bigrams B (e.g. those corresponding to a name or a list of names), we can define the corresponding loss as $\ell_B := \frac{-1}{|B|} \sum_{\beta \in B} \log(P_{\text{ind}(\beta)})$.³¹ For example, the loss associated to the name **emma** is $\frac{-1}{5} \cdot (\log(P_{*e}) + \log(P_{em}) + \log(P_{mm}) + \log(P_{ma}) + \log(P_{a*})) = \frac{-1}{5} \cdot (\log(0.0478) + \log(0.0377) + \log(0.0253) + \log(0.3899) + \log(0.1960)) = 2.5135$.³² By contrast, the loss associated to the name **andrej** is 3.0391: this name is less likely to result from sampling the model (even though it, too, appears in our dataset).

As it turns out, the loss for the entire dataset of 228,146 bigrams is 2.4541. This number will be relevant for us shortly.

2.1.2. Model Smoothing. Recall that the logarithm of 0 is undefined.³³ On the other hand, the logarithms $\log(P_{\text{ind}(\beta)})$ appearing in our loss formula *are* well-defined, because if $\beta \in \text{BigramData}$ then the bigram β appears a positive number of times in our dataset (and so $P_{\text{ind}(\beta)} > 0$).

Nevertheless, it will be convenient for us to have a well-defined loss for *any* possible name, even one that may not have appeared in the training dataset (as e.g. in §3.6). In the present context, this can be implemented by augmenting the entries in the table of Figure 4 with “fake counts”. Even just adding 1 to each entry will make it so that all entries of the resulting matrix of probabilities are positive, so that e.g. the loss associated with the “name” **andrejq** will be 3.4834: this name remains quite unlikely (and in particular it has greater loss than **andrej**), but it is no longer literally impossible (whereas $P_{jq} = 0$). On the other hand, we if add say 1,000,000 to each of those entries, then all of the entries of the resulting matrix of probabilities will be very close to $\frac{1}{27}$, which would essentially bring us back to the “uniformly random” model discussed briefly in §2.0 (and in particular the probability distribution would essentially no longer depend on the context). In any case, this procedure is known as **model smoothing**, and we defer a more in-depth discussion of it to [ref forward (maybe to somewhere in §4? probably something about “regularization”. or actually, it seems like this is just a general theme.)].

2.2. The Trained Bigram Model.

2.2.0. The Model. The architecture of the trained bigram model will be nearly identical to that of the combinatorial bigram model: given the context of a character $x \in \text{Chars}$, we will extract a probability distribution on Chars from the product $\vec{e}_{\text{ind}(x)}W$ of its one-hot encoding with some 27-by-27 matrix W , which will of course just be the $\text{ind}(x)^{\text{th}}$ row of W . However, this vector will not generally define a probability distribution in and of itself: its entries need not be nonnegative, nor must they sum to 1.

In order to extract a probability distribution from the vector $\vec{e}_{\text{ind}(x)}W$, we apply the **softmax** function to it.³⁴ For any dimension d , this is the function $\mathbb{R}^d \xrightarrow{\sigma} \Delta^{d-1} \subset \mathbb{R}^d$ given by the formula $(t_i)_i \mapsto \left(\frac{e^{t_i}}{\sum_j e^{t_j}} \right)_i$: that is, it exponentiates all entries and then rescales the resulting vector of positive numbers down to a probability distribution.³⁵ So, the model assigns the probability distribution $\sigma(\vec{e}_{\text{ind}(x)}W) \in \Delta^{26}$ to the

³¹We can also express this as $\ell_B := \frac{-1}{|B|} \sum_{\beta \in B} \log(P_{\text{ind}(\beta)}) = -\log \left(\sqrt[|B|]{\prod_{\beta \in B} P_{\text{ind}(\beta)}} \right)$: the arithmetic mean (i.e. average) of logs is the log of the geometric mean.

³²It is worth remarking that even if we were to define our combinatorial bigram model *only* using the name **emma**, the loss associated to that name would be nonzero, i.e. it would appear with probability strictly less than 1. This is because given the context **m**, that model would return each of the characters **m** and **a** with probability $\frac{1}{2}$. Indeed, the probability of obtaining the sequence of bigrams ***e**, **em**, **mm**, **ma**, **a*** in this setting would be $1 \cdot 1 \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot 1 = \frac{1}{4}$.

³³At best, one can say that $\lim_{t \rightarrow 0^-} \log(t) = -\infty$.

³⁴here, mention a computer-friendly implementation of it: given an input vector, first subtract the maximum value among its coordinates before applying the usual function. this helps avoid floating-point errors, e.g. e^{100} is “infinity”.

³⁵When $d = 2$, this can be thought of geometrically as follows. First, the operation $(x, y) \mapsto (e^x, e^y)$ carries the (x, y) -plane diffeomorphically onto its first quadrant. And then, the function $(u, v) \mapsto \left(\frac{u}{u+v}, \frac{v}{u+v} \right)$ carries each point (u, v) in the first quadrant to the unique point on the 1-simplex Δ^1 given by its intersection with the line through (u, v) and the origin. (The

context $x \in \text{Chars}$; altogether, it is defined by the composite function

$$\text{Chars} \xrightarrow{\tilde{e}_{\text{ind}(\bullet)}} \mathbb{R}^{27} \xrightarrow{W} \mathbb{R}^{27} \xrightarrow{\sigma} \Delta^{26}.$$

Writing $\sigma(W)$ for the matrix obtained from W by applying σ to its *rows* (which choice we do not include in the notation), we have the equivalent expression $\tilde{e}_{\text{ind}(x)}\sigma(W)$ for the probability distribution $\sigma(\tilde{e}_{\text{ind}(x)}W)$.

Observe that the matrix $\sigma(W)$ will play the role for the trained bigram model that the matrix P plays for the combinatorial bigram model. In fact, this analogy extends further, since both of these matrices are obtained by rescaling the rows of a prior matrix down to probability distributions: for $\sigma(W)$ this is the matrix $\exp(W)$ obtained by exponentiating (i.e. applying the function $t \mapsto e^t$) the matrix W termwise, while for P this is simply the matrix of bigram counts displayed in Figure 4. Thus, in the present context one can *also* think of $\exp(W)$ as a matrix of “bigrams counts”, although note that while its entries are necessarily positive they will not generally be whole numbers. Hence, one can consider W as a matrix of *log counts*, which as a shorthand are generally referred to as **logits** (pronounced “low-jits”).

2.2.1. Training. We initialize the $27^2 = 729$ entries of the matrix W – our parameters – as random numbers sampled from the normal distribution $\mathcal{N}(0, 1)$ with mean 0 and variance 1. And then, we will tune these parameters via gradient descent with respect to the loss function $\mathbb{R}^{729} \xrightarrow{\ell} \mathbb{R}$ given by $\ell := \frac{-1}{|\text{BigramData}|} \sum_{\beta \in \text{BigramData}} \log(\sigma(W)_{\text{ind}(\beta)})$ (using analogous notation to that of §2.1.1).³⁷

We now run some initial tests to determine an appropriate learning rate.³⁸ Randomly initializing W , our initial loss is 3.7590. We rather arbitrarily choose a learning rate of $r = 0.1$; using this for 10 epochs, we arrive at a loss of 3.7508.³⁹ This change feels rather incremental, and so we boost our learning rate up to $r = 1$, with which 10 more epochs brings us to a loss of 3.6723. This change still feels rather incremental, and so we boost our learning rate up to $r = 10$, with which 10 more epochs brings us to a loss of 3.1768. And this change *still* feels rather incremental, and so we finally boost our learning rate up to $r = 50$, with which 10 more epochs brings us to a loss of 2.6667. This feels more reasonable, so we stick with the learning rate of $r = 50$.

We reinitialize W and train our neural network for 100 epochs using learning rate $r = 50$. This brings our loss down to 2.4729, in a way that appears to be converging: the last 10 epochs only bring down the loss from 2.4751, for a difference of $2.4751 - 2.4729 = 0.0022$ (whereas we saw previously that 10 epochs with $r = 50$ induced a decrease in loss of $3.1768 - 2.6667 = 0.5101$). Indeed, training for 100 more epochs brings our loss down to 2.4624, for a difference of $2.4729 - 2.4624 = 0.0105$.

2.2.2. Comparison of Bigram Models. The loss of 2.4541 for the combinatorial bigram model is essentially identical to the loss of 2.4624 for the trained bigram model. This reflects the fact, which we have already indicated in §2.0, that these two models are essentially identical. More specifically, as we explain presently, the matrices P and $\sigma(W)$ are essentially identical.⁴⁰

To understand why this should be the case, it is helpful to first consider a simple thought experiment. Suppose that we are studying a weighted die, whose behavior is described by some unknown probability distribution $\vec{p} := (p_1, \dots, p_6) \in \Delta^5$; here, p_i denotes the probability of rolling the number i , so that e.g. the likelihood of obtaining the sequence 2, 3, 3, 5 is $p_2 \cdot p_3 \cdot p_3 \cdot p_5$. Now, suppose we roll the die N times and

level set for any *interior* point $(p, q) \in \Delta^1$ (i.e. one with $p, q > 0$) is the line $y + \log(p) = x + \log(q)$. Of course, this geometric description persists (mutatis mutandis) in all dimensions.

³⁶Note that this is not quite an MLP, because in an MLP each neuron must have its own activation function (whereas by contrast the softmax function $\mathbb{R}^d \xrightarrow{\sigma} \mathbb{R}^d$ does *not* operate separately in each coordinate (except in the trivial case that $d = 1$)). [but Andrej (and others?) seem to refer to it as an MLP anyways – probably because it’s so close to being one, in contrast with other architectures to be discussed down the road.]

³⁷Note the freedom that the softmax function $\mathbb{R}^d \xrightarrow{\sigma} \Delta^{d-1}$ affords us: in effect, it reduces us from a constrained optimization problem (of trying to optimize certain probability distributions, i.e. points in $\Delta^{d-1} \subset \mathbb{R}^d$) to an unconstrained optimization problem (of trying to optimize vectors in $\mathbb{R}^d$ that determine those probability distributions via σ).

³⁸again ref forward for more systematic discussion.

³⁹Actually, this is the loss after 9 epochs, not 10. (The loss is only computed in the forward pass, which occurs before the update.) For simplicity, here and henceforth we elide this minor distinction.

⁴⁰maybe mention some samples from here. from the combinatorial model with +1 fake counts, he got: mor, axx, minaynnnyles, koneraisah, andhumizarie. from this one, he gets: mor, axx, minaymoryles, kondlaisah, anchthizarie. **why the heck are these consistently converging back together after diverging??!?**

obtain n_i instances of each number i . Each probability distribution determines a corresponding likelihood of these particular observations; this assignment assembles into a likelihood function

$$\Delta^5 \longrightarrow [0, 1] \quad \text{given by} \quad \vec{p} \longmapsto \prod_i p_i^{n_i},$$

which assigns to each probability distribution $\vec{p} \in \Delta^5$ the likelihood of the observed sequence of rolls. And among all possible probability distributions $\vec{p} \in \Delta^5$, the unique one that maximizes this function – that is, the optimal guess at the unknown probability distribution given the observed sequence of rolls – is obtained by setting $p_i = \frac{n_i}{N}$.⁴¹ This probability distribution is called the *maximum likelihood estimator* for the given observations.

Now, in our trained bigram model, each context $x \in \text{Chars}$ determines its own probability distribution $\vec{e}_{\text{ind}(x)} \sigma(W)$, which upon sampling is effectively intended to produce the sublist $\text{BigramData}_{x_-} \subset \text{BigramData}$ consisting of those bigrams that begin with it. Indeed, it is not hard to see that in order for the loss to be minimized, each such probability distribution $\vec{e}_{\text{ind}(x)} \sigma(W)$ must be a maximum likelihood estimator for BigramData_{x_-} .⁴²

On the other hand, the loss function $\mathbb{R}^{729} \xrightarrow{\ell} \mathbb{R}$ itself does not have a unique minimum: considering its domain as $\mathbb{R}^{729} \cong (\mathbb{R}^{27})^{27}$, its value is unchanged by adding a constant vector $(c, \dots, c) \in \mathbb{R}^{27}$ in any individual factor (because this does not change the value of the softmax (since $\frac{e^{c+t_i}}{\sum_j e^{c+t_j}} = \frac{e^c \cdot e^{t_i}}{e^c \cdot \sum_j e^{t_j}} = \frac{e^{t_i}}{\sum_j e^{t_j}}$) and so it does not change the resulting probability distribution $\vec{e}_i \sigma(W)$). Rather, the loss function admits a factorization

$$\begin{array}{ccc} (\mathbb{R}^{27})^{27} & \xrightarrow{\ell} & \mathbb{R} \\ & \searrow \sigma_{27} & \nearrow \tilde{\ell} \\ & (\Delta^{26})^{27} & \end{array}$$

(through the function σ^{27} that applies the softmax function $\mathbb{R}^{27} \xrightarrow{\sigma} \Delta^{26}$ separately in each variable), and this factorization $\tilde{\ell}$ has a unique minimum which in the $\text{ind}(x)^{\text{th}}$ coordinate is the maximum likelihood estimator for BigramData_{x_-} – namely, the vector $\vec{e}_{\text{ind}(x)} P$. So, if $W \in \mathbb{R}^{729}$ minimizes the loss, then it must be the case that $\sigma(W) = P$.⁴³

2.2.3. Regularization. Recall the discussion of model smoothing for the combinatorial bigram model in §2.1.2: we added some fixed number of “fake counts” for each bigram, with more fake counts leading to the resulting probability distributions being closer to uniform.

The trained bigram model admits an analog: given some context, it associates a uniform distribution on Chars exactly when all of the entries in the corresponding row of W are equal (since it is precisely the vectors $(c, c, \dots, c)$ that yield the uniform distribution under the softmax function). In particular, if we add to our loss a term like $\frac{1}{729} \sum_{i,j} |W_{i,j}|^2$ (the average of the squares of the entries of W), then gradient descent will simultaneously attempt to increase the likelihood of BigramData and to keep the entries of W small. (Of course, it will generally be the case that these two goals are in tension with one another.)

This procedure is called *regularization*, and such a summand is called a *regularization loss* term. As we will discuss further in [forward ref], regularization is an important procedure for preventing both overfitting and underfitting.⁴⁴ Of course, the regularization loss can be scaled up and down, leading to hyperparameters that we will discuss further in [ref].

⁴¹This assertion may be so intuitively clear that it might in fact seem to be tautological. However, it is not tautological (and it can be proved using the method of Lagrange multipliers).

⁴²Note that these probability distributions do not all contribute equally to the loss: those corresponding to characters that appear more frequently contribute more heavily. Nevertheless, they all contribute *independently* to the loss: it decomposes as the iterated sum $\ell = \frac{-1}{|\text{BigramData}|} \sum_{x \in \text{Chars}} \sum_{\beta \in \text{BigramData}_{x_-}} \log(P_{\text{ind}(\beta)})$. Hence, the proof of this claim is essentially identical to that regarding the weighted die.

⁴³With additional work, one can additionally verify that the only local minimum of $\tilde{\ell}$ is its global minimum, which implies that our gradient descent cannot be obstructed from approaching this minimum value by any other local minima (i.e. “mountain lakes”).

⁴⁴really, it can prevent underfitting? i think he said this, but e.g. this blog post seems to say that it only helps prevent overfitting (in the intro paragraph). ditto with wikipedia page on regularization.

3. FUNDAMENTAL PRINCIPLES OF MACHINE LEARNING: EMBEDDINGS, n -GRAM MODELS, OVERFITTING/UNDERFITTING, STOCHASTIC GRADIENT DESCENT, AND HYPERPARAMETER TUNING

3.0. Overview. Remaining within the realm of name generation, in this section we introduce *embeddings* in §3.1 as a means of improving our bigram models to n -gram models for $n > 2$.⁴⁵ We then set up such a quadgram (i.e. 4-gram) model in §3.2 and use it to illustrate a number of fundamental principles of machine learning: overfitting (§3.3), stochastic gradient descent (§3.4), learning rate tuning (§3.5), test loss (§3.6), and more hyperparameter tuning (§3.7). We conclude in §3.8 by indicating some “names” that result from our tuned quadgram model and issuing a challenge for the interested reader.

3.1. Embeddings. Suppose that we wish to improve our bigram model to an n -gram model: one that takes into account the previous $n - 1$ characters as context when determining the probability distribution of the next character. If we were to directly mimic the architecture of the bigram model from §2, this would involve a 27^{n-1} -by-27 matrix of weights (with rows corresponding to contexts and columns corresponding to possible next characters) for a total of 27^n parameters.⁴⁶ This number grows incredibly quickly with n (e.g. $27^5 > 10^7$), and so this approach is fundamentally unscalable.

Recall that the bigram model associates to each context $x \in \text{Chars}$ its one-hot encoding: the standard basis vector $\vec{e}_{\text{ind}(x)} \in \mathbb{R}^{27}$ (which becomes the input to the model). Likewise, the above proposal amounts to a one-hot encoding, associating to each context $x_1x_2 \cdots x_{n-1} \in \text{Chars}^{n-1}$ a standard basis vector in $\mathbb{R}^{27^{n-1}}$. One way of addressing the above unscalability, then, would be to let go of our particular attachment to standard basis vectors: instead, we could associate to each such context an *arbitrary* vector in some (possibly lower-dimensional) vector space $\mathbb{R}^m$, whose coordinates could then be included as parameters as the model (and hence learned during the training process).⁴⁷ However, as n increases, the number of contexts grows exponentially, and so the typical number of datapoints associated to any given context decreases exponentially. This poses a serious problem: if we only have a few datapoints associated to a given context $x_1x_2 \cdots x_{n-1}$, then our model will not be able to learn much from the training data about what should follow it.

Instead, as first proposed in [Bengio+03], we will associate to each individual character $x \in \text{Chars}$ a learned vector $C_x \in \mathbb{R}^m$, which enables us to associate to each context $x_1x_2 \cdots x_{n-1} \in \text{Chars}^{n-1}$ a corresponding vector $(C_{x_1}, C_{x_2}, \dots, C_{x_{n-1}}) \in (\mathbb{R}^m)^{n-1} \cong \mathbb{R}^{m(n-1)}$ obtained by concatenating these learned vectors. These learned vectors C_x are called **embedding vectors**, and the function $\text{Chars} \xrightarrow{C} \mathbb{R}^m$ is called an **embedding**.^{48,49,50} Crucially, the number of parameters needed to specify the embedding only depends on m , and in particular it is independent of the length $n - 1$ of the context: the possible embeddings assemble into the

$$\begin{array}{ccc} \text{Chars}^{n-1} & \xrightarrow{\quad} & \Delta^{26} \\ \text{pr} \downarrow & \nearrow & \\ \text{Chars}^{m-1} & & \end{array}$$

in which the functions to Δ^{26} are the models and the vertical function is given by projection onto the last $m - 1$ characters. Said differently, any m -gram model trivially defines an n -gram model via precomposition with $\text{Chars}^{n-1} \xrightarrow{\text{pr}} \text{Chars}^{m-1}$.

⁴⁵Actually, there are not quite 27^{n-1} possible contexts, since a valid context cannot contain any instances of the special character $*$ that follow ordinary letters (so e.g. $\mathbf{a*b}$ is not a valid context). But 27^{n-1} is certainly the dominant term in the precise expression for the number of valid contexts.

⁴⁶Equivalently, this proposal amounts to using one-hot encoding but prepending a linear layer $\mathbb{R}^{27^{n-1}} \rightarrow \mathbb{R}^m$ to the model. So, it would involve $(n - 1) \cdot m + m \cdot 27 = m \cdot (27 + m - 1)$ parameters, which is now only linear (instead of exponential) in n .

⁴⁷One may also refer to the dimensions of $\mathbb{R}^m$ as *features*, and to the vectors C_x as *feature vectors*. (Optimistically, one may imagine that e.g. in the case of an embedding of words (instead of characters), the different axes might measure different features of the words (e.g. parts of speech, positive/negative valence, etc.), although of course in practice this need not be the case.)

⁴⁸Analogously to Footnote 47, this proposal amounts to applying the composite $\text{Chars} \xrightarrow{\vec{e}_{\text{ind}(\bullet)}} \mathbb{R}^{27} \xrightarrow{C} \mathbb{R}^m$ of one-hot encoding followed by a linear function separately to each character of a context $x_1x_2 \cdots x_{n-1}$ and then concatenating the results.

⁴⁹here, i'm already just writing $\text{Chars} \xrightarrow{C} \mathbb{R}^m$ for the function, instead of putting in $\vec{e}_{\text{ind}(-)}$. keep that? in any case, keep things consistent...

vector space $(\mathbb{R}^m)^{\text{Chars}} \cong (\mathbb{R}^m)^{27} \cong \mathbb{R}^{27m}$ (which will be one factor in the vector space of possible parameter values).

A powerful emergent property of this framework is that it enables the model to learn *similarities* between individual characters. For instance, in the present situation, the model may learn through training that vowels can play similar roles in names, so that e.g. if a context can be followed by one vowel then it can likely be followed by other vowels too; for example, the trigram `ell` begins the names `ella`, `ellen`, and `elliott`. And indeed, in our explorations we will generate an embedding that treats vowels somewhat interchangeably (see Figure 8). Similarly, the paper [Bengio+03] – which embeds a “vocabulary” of $\sim 17\text{k}$ words (instead of 27 characters) into $\mathbb{R}^m$ (and takes m to be 30, 60, and 100) – gives the example that the appearance of the sentence `The cat is walking in the bedroom` in the training data may make more likely the sentence `A dog was running in a room` as a possible output of the model, due to the semantic and syntactic similarities between their respective words. This ability of the model to generate sensible but genuinely new (a.k.a. “out of sample”) outputs, called **generalization**, is highly desirable. In particular, note the dramatic improvement in expressive power over one-hot encoding, which associates orthonormal input vectors to *any* pair of distinct contexts.

3.2. The Quadgram Model. Let us first describe the general n -gram model: based on a context of length $k := n - 1$ (called the *block size*), it will provide a probability distribution from which we will sample in order to determine the next character in the newly generated name. Its architecture is described by the composite function

$$\text{Chars}^k \xrightarrow{(\vec{e}_{\text{ind}(-)})^k} (\mathbb{R}^{27})^k \xrightarrow{C^k} (\mathbb{R}^m)^k \cong \mathbb{R}^{mk} \xrightarrow{\tanh((-)W_1 + \vec{b}_1)} \mathbb{R}^h \xrightarrow{(-)W_2 + \vec{b}_2} \mathbb{R}^{27} \xrightarrow{\sigma} \Delta^{26},$$

where we write $(-)^k$ to indicate the application of a function separately in each of the k factors (recall Footnote 49).⁵¹ In other words, given a k -gram $x_1 x_2 \cdots x_k \in \text{Chars}^k$, this function returns the probability distribution $\sigma(\tanh((C_{x_1}, C_{x_2}, \dots, C_{x_k})W_1 + \vec{b}_1)W_2 + \vec{b}_2) \in \Delta^{26}$, where we write $C_x := \vec{e}_{\text{ind}(x)}C$ for the $\text{ind}(x)^{\text{th}}$ row of C – the embedding vector corresponding to $x \in \text{Chars}$. The matrices C , W_1 , $\vec{b}_1$, W_2 , and $\vec{b}_2$ respectively contribute $27m$, mkh , h , $27h$, and 27 parameters to the neural network.^{52,53}

We will train and play with a quadgram model. More specifically, we will take our block size to be $k = 3$ (so that $n = 4$), and for now (until § 3.7) we will fix our embedding dimension to be $m = 2$ and the number of neurons in the hidden tanh layer to be $h = 100$. Thus, our neural network will have $27 \cdot 2 + 2 \cdot 3 \cdot 100 + 100 + 27 \cdot 100 + 27 = 3481$ parameters.

We initialize all 3,481 parameters as random numbers sampled from the normal distribution $\mathcal{N}(0, 1)$. We will train our neural network on the same dataset of $\sim 32\text{k}$ names as before. Of course, now we will parse these names not into bigrams but into quadgrams, so that e.g. the name `emma` contributes the quadgrams `***e`, `**em`, `*emm`, `emma`, and `mma*` (which in turn yield the input/output pairs `(***, e)`, `(**e, m)`, etc.). Specifically, our dataset parses into a list `QuadgramData` of 228,146 quadgrams.⁵⁴ Analogously to when we trained our bigram model in § 2.2.1, we will take the average negative log likelihood of the model over `QuadgramData` as our loss function $\mathbb{R}^{3481} \xrightarrow{\ell} \mathbb{R}$.⁵⁵

3.3. Overfitting. For illustrative purposes, we actually first train our model not on the entire dataset of 228,146 quadgrams, but rather on just the 32 quadgrams associated to the first five names in the dataset: `emma`, `olivia`, `ava`, `isabella`, and `sophia`. That is, we perform gradient descent with respect to a much simpler loss function $\mathbb{R}^{3481} \rightarrow \mathbb{R}$, which assigns to each choice of 3,481 parameter values the resulting average negative log likelihood of these 32 quadgrams.

⁵¹or say this once and for all in Notation/Conventions above

⁵²More generally, with a “vocabulary” of size v (no longer necessarily 27), this architecture has $vm + mkh + h + vh + v = v \cdot (m + h + 1) + mkh$ parameters.

⁵³possibly here, for future reference introduce the notation $\mathfrak{h}(x_1 x_2 \cdots x_k) := \tanh((C_{x_1}, C_{x_2}, \dots, C_{x_k})W_1 + \vec{b}_1) \in \mathbb{R}^h$ for the hidden layer and $\mathfrak{l}(x_1 x_2 \cdots x_k) := \sigma(\mathfrak{h}(x_1 x_2 \cdots x_k)W_2 + \vec{b}_2) \in \mathbb{R}^{27}$ for the logits (which are both functions of the context).

⁵⁴There are exactly as many quadgrams as there were bigrams; indeed, each name contributes one n -gram for each of its characters (including the terminating character `*`), regardless of the value of n .

⁵⁵possibly here but actually probably in the previous section, add a (foot?)note about cross-entropy. see appendix for notes on it. (e.g. it comes with similar speedups analogous to $\frac{d}{dt} \tanh(t)$, and it also implements the same fix to avoid floating-point errors as does the built-in softmax function.)

For now, we fix our learning rate to be $r = 0.1$. After 10 epochs, our loss drops from 17.7697 down to 4.4147. Training for 100 more epochs brings our loss down to 0.3244, and training for 100 more epochs after that brings our loss down to 0.2774. Training for 1,000 more epochs brings our loss to 0.2552.

This loss is dramatically lower than that of our bigram model (which was around 2.5). However, this is explained by the fact that we are *overfitting* our data: with so many parameters, our model can almost perfectly learn such a small dataset (akin to how a sufficiently high-degree polynomial can be made to have its graph pass exactly through almost any finite set of points in the plane).⁵⁶ In this situation, one may say that the model is “memorizing” the dataset. So, we should not consider this as necessarily desirable in and of itself: our model will likely not generalize very well. We will address this issue more systematically in §3.6.

We return to the full dataset `QuadgramData` of 228,146 quadgrams and reinitialize our parameters. Now, our loss begins at 19.5052; after 10 epochs it decreases to 11.0519, and after 10 more epochs it decreases to 8.5218. This loss value is again decreasing – our model *is* learning – but much more gradually than it did just above, since now it is much harder to fit the parameters to the dataset (which is now so much more diverse).

3.4. Stochastic Gradient Descent. Whereas training the model on just 32 bigrams was essentially instantaneous, training the model on the full dataset is substantially slower – taking a few seconds for each round of 10 epochs.⁵⁷ Obviously, this issue only intensifies with a larger dataset and/or a larger neural network.

In order to address it, we can use a subset of `QuadgramData` to *approximate* the entire dataset. Of course, this comes with a tradeoff: a larger subset will more accurately approximate the entire dataset, but each epoch will take more time and computational power to run. Said differently, we can run more epochs at the expense of sampling fewer datapoints in each one (which comes with concomitant disadvantages). However, it turns out that this tradeoff is well worth it, at least to a point: so long as it is randomly chosen, a surprisingly small subset of the dataset can nevertheless often provide a reasonably representative sample.

Altogether, this leads us to the training method of *stochastic gradient descent*. In stochastic gradient descent, we begin each epoch by randomly selecting (with replacement) a small subset of `QuadgramData`, which is called a *batch*.^{58,59,60} Then, in that epoch we perform gradient descent – that is, the forward pass, the backward pass, and the update step – merely with respect to the loss function corresponding to the batch. We can now put a finer point on the above tradeoff: the size b of the batch will generally affect the *quality* of the gradient vector that we use in our update step, i.e. the extent to which it approximates the gradient vector of the true loss function.

We fix our batch size to be $b = 32$, although of course this is a hyperparameter that can be tuned. The process of randomly selecting a batch is essentially instantaneous, and as a result it is essentially instantaneous for us to perform many thousands of epochs of stochastic gradient descent.

Of course, we are ultimately trying to reduce the value of the true loss function, and so in stochastic gradient descent it is essential to compute this (at least periodically) in addition to our batch loss values (which are computed as the forward passes). In our experimentation, we repeatedly run 1,000 epochs of stochastic gradient descent and then compute the true loss; the resulting values are recorded in Figure 6. Observe that the batch loss may be either lower or higher than the true loss, and that the difference between

⁵⁶However, it is not possible to achieve a loss of zero, due to the fact that the trigram `***` is followed by all five characters `e`, `o`, `a`, `i`, and `s` in the dataset (analogously to Footnote 32).

⁵⁷This is because the complexity of a loss function – and in particular the size of its computational graph – grows with the number of datapoints used to define it (recall Footnote 17).

⁵⁸The term “stochastic” refers to the randomness that we are injecting into the process.

⁵⁹A batch is also sometimes referred to as a *minibatch*, although the meanings of these terms are somewhat fluid. Here, we take the latter term to refer to smaller subsets of a batch. Decomposing a batch into minibatches can be useful for processing very large batches: in order to avoid using too much memory at once, one can then perform the forward and backward passes on each minibatch separately, aggregating their resulting contributions to the gradient vector before proceeding to the update step.

⁶⁰Note that it is not entirely standard to continue to use the term “epoch” in the context of stochastic gradient descent. Whereas we are sampling with replacement, one could alternatively sample without replacement, i.e. decompose the entire dataset into batches and then process each batch; and in this case, a single pass through the entire dataset might be referred to as a single epoch.

batch loss	true loss
2.7011	2.7744
2.8076	2.5981
2.4725	2.5713
2.5761	2.5370
2.1793	2.6405
2.6405	2.5857

FIGURE 6. This table records the batch loss and true loss values computed during the process of stochastic gradient descent. The passage from each row to the next represents 1,000 epochs.

these two values can be fairly large. Of course, one should generally expect the difference between them to be smaller if the batch size is larger.

3.5. Learning Rate Tuning. The loss values displayed in Figure 6 suggest that with the current hyperparameters, our method of stochastic gradient descent may be bottoming out: it probably won’t be able to get the true loss below 2.5. However, we should not be satisfied with these loss values, since they are worse than that of our bigram model.⁶¹ Thus, we should expect that some hyperparameter tuning is in order. To that end, we presently discuss the process of tuning the learning rate.

Roughly speaking, we would like for the learning rate to be as large as possible, subject to the constraint that the loss is generally decreasing when we use it for stochastic gradient descent.⁶² So, we will perform stochastic gradient descent with respect to a range of learning rates. Of course, in order to do this we must decide what range of learning rates to try, and so the first step is to seek lower and upper bounds on reasonable learning rates.

To find a lower bound, we first try $r = 0.0001 = 10^{-4}$. After 100 epochs, the (batch) loss decreases from 21.4716 to 20.2003. This feels too slow, so we instead try $r = 0.001 = 10^{-3}$. Now, 100 more epochs brings our (batch) loss down to 16.2544. This is still rather slow, but not nearly so much as before. So we take $r = 10^{-3}$ as our lower bound.

To find an upper bound, we try $r = 1$. Now, over the course of 100 epochs, the loss drops from 20.4092 down to as low as 4.8548, but thereafter it continues to jump up and down and in particular reaches back up as high as 9.1018. These dramatic fluctuations indicate that this learning rate is too high: we are stepping too far in the direction of the gradient vector, beyond the range where the loss function is well-approximated by its linear approximation (recall Footnote 19). Just to get an even starker illustration of this phenomenon, we also try $r = 10$. Now, the loss begins at 21.3501, and over the course of 100 epochs jumps to as high as 152.7129. So this value of r is *way* too big.

So, we expect that the “optimal” learning rate lies somewhere between $0.001 = 10^{-3}$ and $1 = 10^0$. In order to find it, we will approach it from below, starting with $r = 10^{-3}$ and slowly increasing r with each epoch until our loss is no longer generally decreasing.⁶³ Specifically, since the learning rate functions as a multiplicative (rather than additive) hyperparameter, we construct a sequence $\{e_i\}_{1 \leq i \leq 1000}$ of 1,000 evenly-spaced exponents starting with $e_1 = -3$ and $e_{1000} = 0$, and in the i^{th} epoch we use learning rate $r_i := 10^{e_i}$. The resulting batch losses are depicted in Figure 7. Altogether, we find that a learning rate of $r = 10^{-1.0} = 0.1$ is in fact quite reasonable: it is lower by a safe margin than the regime in which the loss begins to behave erratically.

⁶¹If $n > m$, then an n -gram model that factors through an m -gram model (in the sense of Footnote 45) also inherits its average negative log likelihood, which implies that the loss of a trained n -gram model should be no greater than that of a trained m -gram model.

⁶²In particular, the optimal value for the learning rate r may well depend on the batch size b .

⁶³Observe that this is substantially more efficient than performing an entire round of training for many different values of r . Of course, it is necessary to approach the “optimal” value of r from below rather than from above.

⁶⁴Because the loss drops so quickly, it can also be helpful to graph not the loss but rather its logarithm.

⁶⁵The fact that both of these graphs are “fuzzy” – i.e. that they repeatedly jump above and below the smooth curves that they seem to be approximating – can be explained by the fact that we are performing *stochastic* gradient descent, so that in our update steps we are using relatively low-quality gradient vectors. (Of course, this phenomenon could still occur in ordinary (i.e. non-stochastic) gradient descent, if the learning rates are too large.)

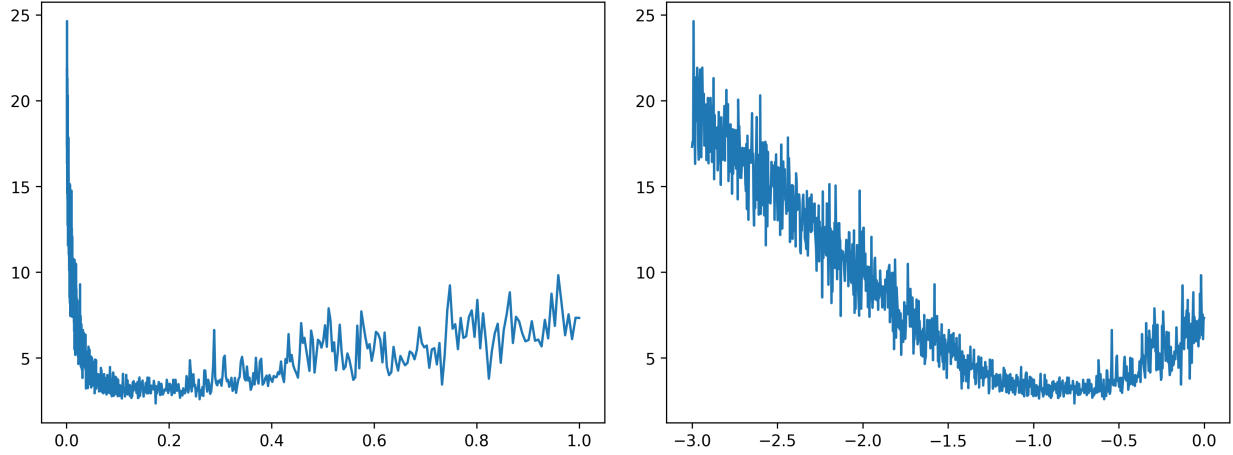


FIGURE 7. On the left is graphed the loss against the learning rate. Since the learning rate r_i increases multiplicatively (namely by a factor of $10^{3/1000}$ from each epoch to the next), most of its datapoints are clustered towards the left side. So, for a more uniform illustration (in which the datapoints are evenly horizontally spaced), on the right is graphed the loss against the learning rate exponent e_i .^{64,65}

Having convinced ourselves that $r = 0.1$ does in fact constitute a “reasonably optimal” learning rate, we reinitialize our parameters to train the model from scratch using it. Running 10,000 epochs at a time and then computing our true (not batch) loss, we obtain losses of 2.4896, 2.4666, 2.4036, and 2.3963. So, we have successfully bested the loss value of our bigram model! In particular, it seems that we were simply not patient enough when considering the loss values displayed in Figure 6 (which were likewise obtained with $r = 0.1$, but only running 1,000 epochs of training at a time).

Note that the differences in our losses arising from each of these rounds of 10k epochs of training are quite small. Thus, we may expect that we are getting close to the minimum value of the loss. As such, it is now appropriate to implement *learning rate decay*; otherwise, we may be “stepping past” the minimum with each epoch.⁶⁶ Specifically, we scale our learning rate down by a factor of 10 to $r = 0.01$. Now, 10k more epochs of training bring our learning rate to 2.3113, and 10k more epochs beyond that bring our learning rate to 2.3079.

3.6. Test Loss. Now that we have bested the loss value of our bigram model, we may be tempted to conclude that our new quadgram model is “better”. However, this conclusion is not truly justified: with so many more parameters (3,481 instead of 729), we should worry about the possibility that we are overfitting the data. This issue inspires us to refine our process: rather than training our model on all of the data, we instead only train it on some fraction of the data, and then later test whether we are overfitting by evaluating its loss on some other datapoints that it has not yet seen.

In fact, we separate the data into three different “splits”: the *training split*, the *development split* (or *dev split* for short),⁶⁷ and the *test split*. Following standard practice, we take these to respectively constitute 80%, 10%, and 10% of the entire dataset (after randomly shuffling it); the training split is used for training, the dev split is used for hyperparameter tuning, and the test split is used for final evaluation and reporting (e.g. in a research paper). Note that we should compute the test loss (i.e. the loss value with respect to the test split) very sparingly, since otherwise we may begin to overfit our model on the test split (which would then invalidate its usage for final evaluation of the model).

To illustrate the usage of these splits, we reinitialize our neural network and train it on the training split via stochastic gradient descent: first for 60k epochs at a learning rate of $r = 0.1$, and then for 10k epochs

⁶⁶This is despite the fact that as we converge towards a (local or global) minimum of a function, the magnitude of its gradient vectors approaches zero. One way to detect this “stepping past” behavior would be to compute the variation in direction of the various gradient vectors (as measured e.g. by the angles between them), at least using a sufficiently large batch size in order to improve their quality.

⁶⁷The dev split is also sometimes called the *validation split*.

at a decayed learning rate of $r = 0.01$. Thereafter, we compute that the training loss (not batch loss) is 2.3313, whereas our dev loss is 2.3318. The fact that these numbers are very close suggests that we are *not* overfitting the training set.

3.7. More Hyperparameter Tuning. In fact, the close similarity of the training loss and dev loss that we have just computed suggests that we are actually *underfitting*: we should expect there to be *some* amount of memorization of the training split, which would then be reflected by the training loss being a fair bit lower than the dev loss. Altogether, this suggests that our neural network may be too small – that it may benefit from having more parameters. In particular, we may choose to increase the number of hidden neurons and/or the embedding dimension, which are currently set at $h = 100$ and $m = 2$ respectively.⁶⁸

We begin by growing h from 100 to 300. Now, our model has 10,281 parameters. Running 60k epochs at $r = 0.1$, 30k epochs at $r = 0.05$, and then 60k epochs at $r = 0.01$, we obtain a training loss of 2.2382 and a dev loss of 2.2464.⁶⁹ So once again, the dev loss is slightly higher than the training loss, as expected. The difference is now 0.0082, as opposed to the mere difference of 0.0005 obtained above, so adding more neurons to the hidden layer does seem to be helping us not underfit; but the difference is relatively slight.

Before growing our embedding dimension m from 2, we pause to graph our embedding (while it can still be readily visualized); see Figure 8.

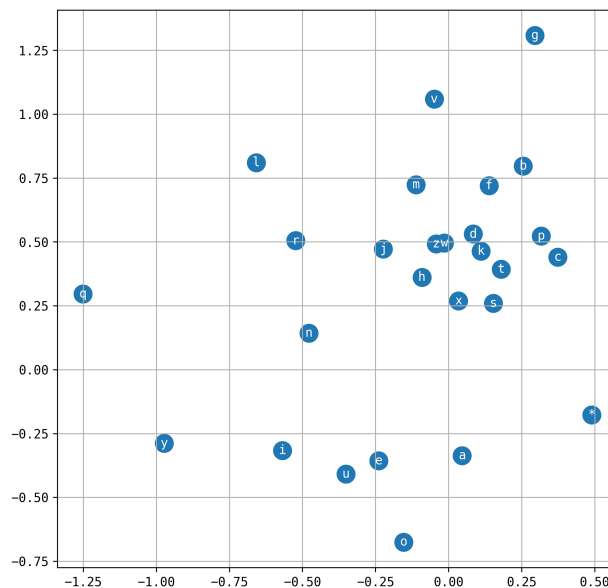


FIGURE 8. This graph depicts our 2-dimensional embedding. Note the clustering of the vowels, which indicates that the model treats them somewhat interchangeably. By contrast, the characters q and x (and to a lesser extent g) are rather isolated: appropriately, the model treats each of them as playing a rather unique role in name generation.

We now grow r from 2 to 10, and dial h down from 300 to 200. Now, our model has 11,897 parameters – roughly the same number as above.⁷⁰ We now rather arbitrarily implement stochastic gradient descent and learning rate decay: after 50k epochs with $r = 0.1$ we obtain a training loss of 2.3617 and dev loss of 2.3845; after 50k more epochs with $r = 0.01$ we obtain a training loss of 2.1742 and dev loss of 2.2003; and after 50k more epochs with $r = 0.01$ we obtain a training loss of 2.1674 and dev loss of 2.1939. In particular, note

⁶⁸Analogously to Footnote 45, adding more hidden neurons or more embedding dimensions can only improve the ultimate performance of the model: if nothing else, these additional flexibilities can remain unused.

⁶⁹andrej suggests here that with more parameters we should expect it to take more epochs to achieve the same level of loss. but why? do we now expect the gradient vector to be smaller? (we’ve kept the (initial) learning rate unchanged, after all.)

⁷⁰To a first approximation, with all else equal the amount of time and computational power that is needed to train a neural network is determined by its total number of parameters. So, this will give us a roughly “apples to apples” comparison of whether it is worth sacrificing hidden neurons in exchange for additional embedding dimensions.

that we have achieved a larger gap between our training loss and dev loss, which suggests that we are now beginning to move from the underfitting regime towards the overfitting regime.

3.8. Final Training and Challenge. Keeping all hyperparameters in place, we reinitialize our parameters and train for 100k epochs with $r = 0.1$ and thereafter for 100k epochs with $r = 0.01$. This yields a training loss of 2.1260 and a dev loss of 2.1701. Sampling from the model 20 times, we obtain the “names”: `carmahela`, `jhovi`, `kimrin`, `thil`, `halanna`, `jazhien`, `amerynci`, `aqui`, `nellara`, `chaiiv`, `kaleigh`, `ham`, `joce`, `quinton`, `lilea`, `jamilio`, `jeron`, `jaryni`, `jace`, and `chrudeley`. These are certainly more “namelike” than the outputs of our bigram model, corresponding to the fact that we are now sampling from a more powerful model.

This dev loss of 2.1701 is the best that Andrej⁷¹ was able to achieve when playing for about half an hour. He challenges us to try and do better: to achieve a lower dev loss, and perhaps even to obtain faster convergence (i.e. to lower the amount of time and computational power needed to train the model). For this, we can modify the various hyperparameters: the number h of hidden neurons, the embedding dimension m , the context size $k := n - 1$, the batch size b , the learning rate r and its decay, and the number of iterations of gradient descent.⁷² Furthermore, the paper [Bengio+03] describes a number of additional ways that we might improve our performance, which we should feel free to leverage as well.

4. INITIALIZATION OPTIMIZATION, BATCH NORMALIZATION, AND DIAGNOSTICS

4.0. Overview. In this section, we take a closer look at the initialization and training of our neural networks, and we expand our toolkit for addressing some of the challenges that these tasks present.

We begin in §4.1 by explaining certain inefficiencies and undesirable features of our initialization of the parameters of the quadgram model that we studied in §3, improving the situation by hyperparameter tuning. However, this hyperparameter tuning is a delicate process, and as a result it does not generalize well to very deep neural networks.

In §4.2, we introduce a new technique called *batch normalization*. This amounts to the interspersing of new “batch normalization” layers in our neural network, which dramatically robustifies the above hyperparameter tuning process and thus enables the effective training of much deeper neural networks.

We conclude in §4.3 by introducing certain numerical and visual diagnostic tools (i.e. statistics and graphs) for evaluating our training process. We apply these to gather further evidence that interspersing batch normalization layers allows us to effectively train a much deeper neural network than we did previously – even without employing much care in our hyperparameter tuning.

4.1. Initialization Optimization.

4.1.0. Overview. In this subsection, we discuss the optimization of the initialization of the parameters of neural networks. We do so in the context of the quadgram model that we arrived at in §3.

In general, the initialization of parameters is an extremely subtle problem, and poor choices thereof can cause the neural network to be effectively unable to learn anything at all during the training process. For instance, this would almost certainly occur if we were to attempt to train a neural network with 50 layers. This is because the effects of poor initialization compound through the successive layers of a neural network (as we’ll confirm using our diagnostic tools in §4.3). Indeed, it is only because our quadgram model has so few layers that we were able to train it reasonably successfully without addressing such considerations. Nevertheless, we will be able to illustrate these ideas using our quadgram model: the improvements in initialization will manifest as improvements in our loss values after training.

To set the stage, we recall the setup for the neural network that we arrived at in §3. This was a quadgram model

$$\text{Chars}^3 \xrightarrow{(\vec{e}_{\text{ind}(-)})^3} (\mathbb{R}^{27})^3 \xrightarrow{C^3} (\mathbb{R}^{10})^3 \cong \mathbb{R}^{30} \xrightarrow{\tanh((-)W_1 + \vec{b}_1)} \mathbb{R}^{200} \xrightarrow{(-)W_2 + \vec{b}_2} \mathbb{R}^{27} \xrightarrow{\sigma} \Delta^{26}$$

with embedding dimension 10 and with 200 neurons in its hidden layer.⁷³ Here, C denotes the matrix of character embeddings (so that $C_x := \vec{e}_{\text{ind}(x)} C \in \mathbb{R}^{10}$ is the embedding vector corresponding to the character $x \in \text{Chars}$), while the W_i and $\vec{b}_i$ are respectively its weight matrices and bias vectors. We initialized all

⁷¹how to phrase all of this?

⁷²give number this a (semi)stable notation? if so, add it both above and below.

⁷³throughout this paragraph, use the notations established? m , h , b , r .

entries of these matrices as random numbers sampled from the normal distribution $\mathcal{N}(0, 1)$. We then trained this neural network using stochastic gradient descent with batches of size 32, using a learning rate of 0.1 for 100k epochs and a decayed learning rate of 0.01 for 100k more epochs.

In §4.1.1 we discuss the issue of *misplaced confidence* of our neural network, which leads us to improve our initialization of W_2 and $\vec{b}_2$. In §4.1.2 we discuss the issue of *dead neurons* in our neural network, which leads us to improve our initialization of W_1 and $\vec{b}_1$. In those subsections we merely indicate the helpfulness of shrinking those initializations; in §4.1.3, we conclude by discussing optimum values for these initializations. For convenient comparison, we collect all of the loss values discussed in this subsection in Figure 9.⁷⁴

context	σ_{W_1}	$\sigma_{\vec{b}_1}$	σ_{W_2}	$\sigma_{\vec{b}_2}$	training loss	validation loss
starting point (§3)	1	1	1	1	2.1245	2.1682
reducing misplaced confidence (§4.1.1)	1	1	0.01	0	2.0696	2.1307
avoiding dead neurons (§4.1.2)	0.2	0.01	0.01	0	2.0356	2.1027
employing Kaiming initialization (§4.1.3)	0.3043	0.01	0.01	0	2.0377	2.1070

FIGURE 9. This table records the various training and validation loss values that we obtain in §4.1 by varying our initializations, as indicated: we initialize each $P \in \{W_1, \vec{b}_1, W_2, \vec{b}_2\}$ as $P \sim \mathcal{N}(0, \sigma_P^2)$. These improve upon the loss values ultimately obtained in §3 (from which all other choices regarding parameters and hyperparameters remain unchanged), which are also recorded here for comparison.

4.1.1. *Misplaced Confidence.* When we first initialize our neural network, its (batch) loss is 27.8817. However, this drops very quickly in the training process, as illustrated in Figure 10 (left). This phenomenon is due to

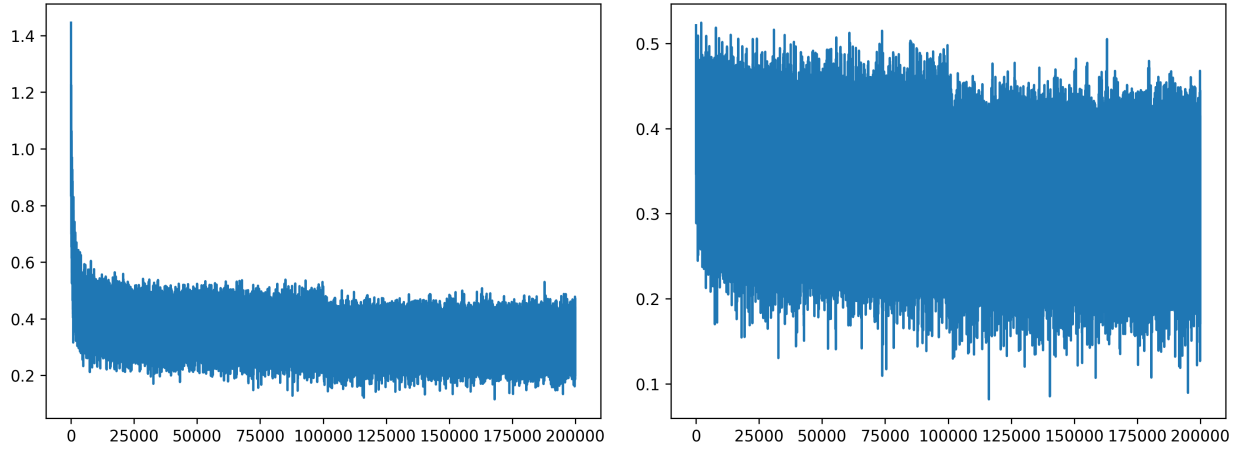


FIGURE 10. These graphs show the base-10 log of the batch loss as it evolves over the course of 200k epochs of training.⁷⁵ The graph on the left shows these values when we initialize the entries of W_2 and $\vec{b}_2$ as random numbers sampled from $\mathcal{N}(0, 1)$ (as in §3); this poor initialization leads to our loss having a “hockey stick” shape. The graph on the right shows these values when we initialize $\vec{b}_2$ as the zero vector and W_2 to have entries sampled from $\mathcal{N}(0, 0.01^2)$ (as in §4.1.1); this improved initialization allows us to skip the “shaft” of the hockey stick and instead focus our training on extending the “blade”.

the following fact: at initialization, our neural network tends to be extremely “confidently incorrect” in its predictions of the “correct” next character (given a trigram context). Indeed, the uniformly random model

⁷⁴also say: “we will illustrate all of these with our diagnostic tools in §4.3.” assuming this is true? otherwise, say this particularly in §4.1.3, since that stuff is definitely illustrated with the diagnostics.

⁷⁵As in Footnote 65, the “fuzziness” of these graphs can be explained by the fact that we are performing *stochastic* gradient descent.

(which assigns a probability of $\frac{1}{27}$ to each possible next character regardless of the context) would have a loss of only $-\log(\frac{1}{27}) = 3.2958$.

Recall that given some trigram context as input, our model would return a uniform probability distribution on the possible next characters if and only if the corresponding logits are all equal. Thus, to assess the deviation of our model from a uniformly random one, we examine some of the values of its logits. And indeed, we find that at initialization our model associates to the context ******* logits that range from as low as -23.2778 associated to the possible next character **r** to as high as 25.8767 associated to the possible next character **t**. This discrepancy leads the model to be $\frac{e^{25.8767}}{e^{-23.2778}} = e^{49.1545} = 2.2260 \cdot 10^{21}$ times more likely to begin a name with **t** than with **r** at initialization! By contrast, one can read off from Figure 4 that for a perfectly trained (or equivalently, combinatorial) quadgram model, the value of this ratio would be $\frac{1308}{1639} = 0.7980$.

Even without knowing this optimal value, however, we can nevertheless aim to reduce the model’s misplaced confidence at initialization by aiming to initialize it much closer to the uniformly random model. For this, since the values of $\tanh$ lie in the interval $(-1, 1)$, it suffices to shrink the entries of W_2 and $\vec{b}_2$. And indeed, for starters we can simply initialize $\vec{b}_2$ to be the zero vector. Then, rescaling the initial values of W_2 by a factor of 0.1 (so that they are effectively sampled not from $\mathcal{N}(0, 1)$ but from $\mathcal{N}(0, 0.1^2)$), we obtain an initial loss of 4.2326.⁷⁶ This is certainly much better than 27.8817, but it is still appreciably worse than the uniformly random loss value of 3.2958. Thus, we instead rescale the initial values of W_2 by a factor of 0.01 (so that they are effectively sampled from $\mathcal{N}(0, 0.01^2)$), and this brings our initial loss down to 3.3221.

With this improved initialization, our loss over the course of training is illustrated in Figure 10 (right), and our correspondingly improved loss values are recorded in Figure 9. In effect, our previous training began by squashing down the values of the logits to obtain some “easy” gains from reducing misplaced confidence; our improved loss values result from skipping that portion of the training, so that we can spend more epochs on the “hard” part of the training.

4.1.2. Dead Neurons. We now turn our attention from the values of the logits to the values of the hidden neurons. We refer to these values as the **activations** of the hidden neurons, and to the values from which they are obtained by applying $\tanh$ as the **preactivations** of the hidden neurons.⁷⁷ Since the activations are values of $\tanh$, they necessarily lie in the interval $(-1, 1)$. As we now explain, it is important for the activations’ absolute values to be both not too large (close to 1) and not too small (close to 0).

In Figure 11 (top-left and middle-left) are shown histograms of the preactivations and activations of the 200 hidden neurons under the current initialization with respect to the initial batch of 32 inputs (for a total of 6400 values). Note that the activations are very strongly concentrated near ± 1 , corresponding to the fact that the preactivations are fairly large (approximately distributed as $\mathcal{N}(0, 5^2)$).

Now, recall from [ref]⁷⁸ that the slope of $\tanh$ approaches 0 as its value approaches ± 1 (or equivalently as its input approaches $\pm\infty$). Crucially, this implies that if for a given neuron N and a given input $\vec{x}$ the activation $N(\vec{x})$ is very close to ± 1 , then in the backward pass, gradients will effectively *not* flow through the node $N(\vec{x})$ in the computational graph for the loss function.⁷⁹ And given our chosen neuron N , if this is true for *all* inputs $\vec{x}$, then the parameters (i.e. weights and biases) of N will effectively *not change at all* in the update step. In this case, we refer to N as a **dead neuron**, since it is incapable of learning.⁸⁰ Obviously, this is undesirable: we would like for our neural network to be populated by neurons that are capable of learning!

⁷⁶We could of course also initialize the entries of W_2 to all be zero (so that our model would be initialized as the uniformly random model), but this is only because these weights are associated to the output layer: as we will see in [ref forward – dead neurons?], in general one should not initialize the weights of a neural network to be zero. **or actually, why? the dead neurons stuff (and the issue with activations being too small) isn’t about symmetry-breaking, just about scale (which is primarily a question of the learning rate). surely symmetry is already broken by the dataset? but he does say around 24:50 that he likes to set $\vec{b}_1$ to $\mathcal{N}(0, 0.01^2)$ instead of 0 so that there’s a bit of “variation and diversity” in the initialization, he finds that in practice it can help the optimization a little bit.**

⁷⁷So, given a trigram context $x_1x_2x_3 \in \text{Chars}^3$, the hidden neurons’ corresponding preactivations are the coordinates of $((C_{x_1}, C_{x_2}, C_{x_3})W_1 + \vec{b}_1) \in \mathbb{R}^{200}$ and their corresponding activations are the coordinates of $\tanh((C_{x_1}, C_{x_2}, C_{x_3})W_1 + \vec{b}_1) \in \mathbb{R}^{200}$.

⁷⁸graph of $\tanh$ and/or the fact that its derivative is $\frac{d \tanh}{dt} = 1 - \tanh(t)^2$

⁷⁹See Footnote 17 for a reminder on the computational graph of the loss function.

⁸⁰Although mathematically a neuron can never be *completely* dead since the derivative of $\tanh$ never actually reaches zero, these derivatives can literally be zeroed out in the backward pass on a computer due to floating-point errors.

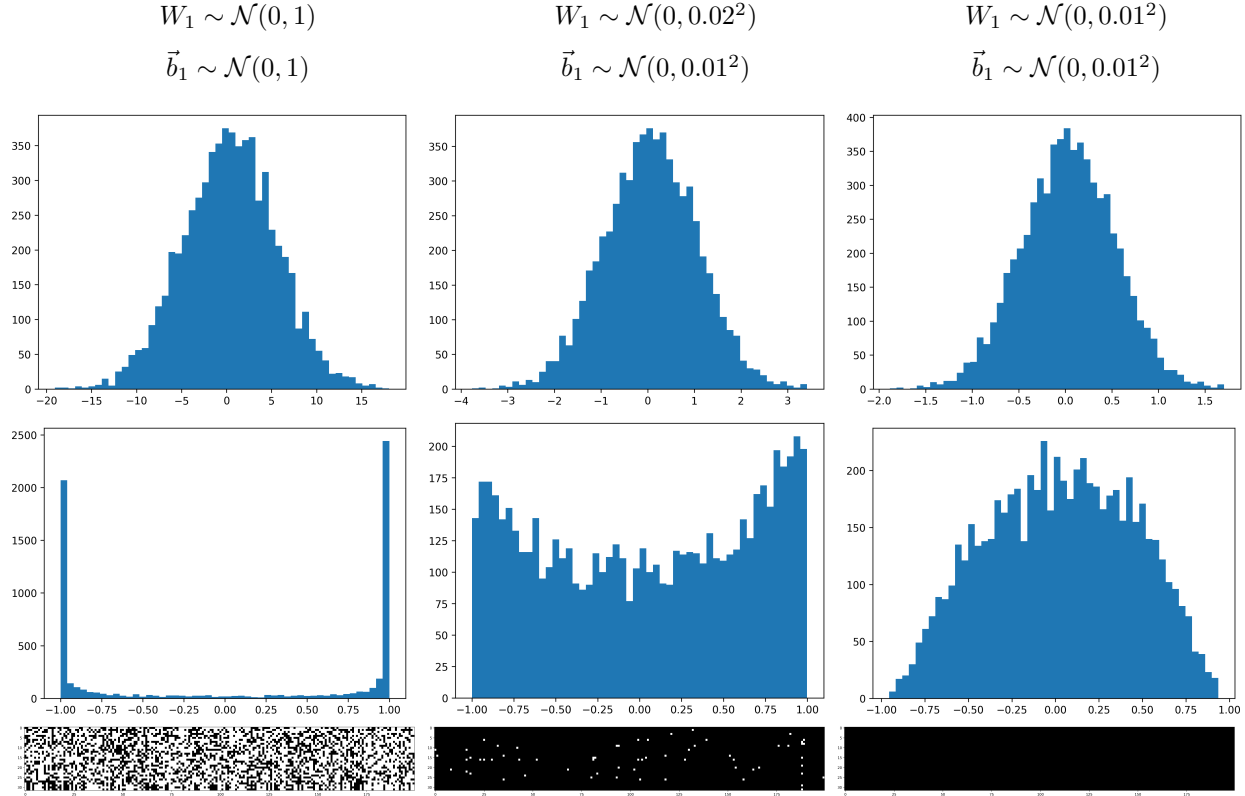


FIGURE 11. These nine images depict the values of the preactivations and activations of the hidden neurons with respect to various initializations; each column of three images corresponds to a single initialization, whose relevant hyperparameters (namely the standard deviations of the normal distributions from which the entries of W_1 and $\vec{b}_1$ are sampled) are indicated above it. The histograms depict the values of the preactivations (upper row) and activations (lower row) of the hidden neurons. In the bottom row are 32-by-200 tables depicting booleans: an entry is white if and only if the corresponding activation has absolute value greater than 0.99.

In Figure 11 (bottom-left) is shown a table. Each of the 32 rows corresponds to an input $\vec{x}$, and each of the 200 columns corresponds to a hidden neuron N , so that each entry corresponds to an activation $N(\vec{x})$. This entry is colored white if $|N(\vec{x})| > 0.99$ and black otherwise. Observe that a substantial fraction of this table’s entries are white, corresponding to the extreme values of the activations (approaching ± 1) depicted just above it. A dead neuron (which we take to mean a neuron N whose activations all satisfy $|N(\vec{x})| > 0.99$) would correspond to an entirely white column (although note that we are only displaying the activations for a single 32-datapoint batch – far from the whole dataset of 228,146 datapoints). There are no entirely white columns, and so none of our neurons are dead (even just with respect to the chosen batch). Nevertheless, there is more deadness among our neurons than we would prefer.⁸¹

⁸¹here, andrej goes on a brief tangent regarding the fact that this same phenomenon can occur for other activation functions. notes on that are here. the functions themselves are current (2024-03-03) graphed in §1.2.

- sigmoid $\sigma(t) := \frac{1}{1+e^{-t}}$ (also S-curve, but now range is $(0, 1)$)⁸²
same issue here: squashes (i.e. derivative is close to 0) for large inputs.
- ReLU, $t \mapsto \max\{0, t\}$
this too has a completely flat region, for $t < 0$. so, it’s easy to be dead: if all input examples in the training set have the preactivation value at this neuron being < 0 , then ReLU just zeros them out and so the neuron never learns (i.e. its parameters never change). this can be a problem at initialization, but it can *also* be a problem during optimization: e.g. if the learning rate is too high, then it can suddenly *become* dead even if it “shouldn’t” be. and from then on, it has “permanent brain damage”.

To address this, observe that shrinking the activations is tantamount to shrinking the preactivations,⁸³ for which it suffices to shrink the values of W_1 and $\vec{b}_1$. So, we change our initialization by rescaling $\vec{b}_1$ by a factor of 0.01 and rescaling W_1 by a factor of 0.1. This change leads to the preactivations and activations illustrated in the right column of Figure 11.

Note that the corresponding table at the bottom is entirely black. This means that *none* of the activations $N(\vec{x})$ with respect to our chosen batch satisfy $|N(\vec{x})| > 0.99$. In and of itself, this is not a problem. However, it *is* a problem if our activations are tiny: in this case, $\tanh$ essentially becomes the identity function (i.e. this is its linear approximation at 0), and as a result our “nonlinearity” function is not really adding any nonlinearity! In this situation, our layer would not really be augmenting the expressive power of our neural network (recall Footnote 3). In order to rest assured that this does not occur, we can choose to require that at initialization some small but nonzero fraction of our activations $N(\vec{x})$ satisfy $|N(\vec{x})| > 0.99$.⁸⁴ Thus, we change the rescaling factor of W_1 from 0.1 to 0.2. The resulting preactivations and activations are illustrated in the middle column of Figure 11. In particular, note that the table at the bottom is no longer entirely black – though it is much closer to being so than the table to its left.

With this improved initialization, our improved loss values resulting from 200k epochs of training are recorded in Figure 9. As in §4.1.1, these further improvements result from the fact that due to our improved initialization we are spending yet more of our 200k epochs on “productive” training (as opposed to e.g. “bringing back from the dead” any neurons whose gradients are nearly vanishing (since their activations are nearly ± 1)).

4.1.3. Kaiming Initialization. In §§4.1.1 and 4.1.2, we discussed rescalings of W_i and $\vec{b}_i$ at initialization that improved our neural network (as measured by its loss values after 200k epochs of training). However, we merely observed that shrinking these values improved the initialization: we did not address the question of what rescaling factors might in fact be optimal. We now discuss such optimal rescaling factors for hidden layers (but not output layers), which are collectively referred to as **Kaiming initialization** (since they were introduced in the paper [ref He–Zhang–Ren–Sun]).⁸⁵

As indicated in §4.1.0, these issues of initialization – or at least those for hidden layers discussed in §4.1.2 – are essentially issues of *compounding effects* through a large number of hidden layers. We can now put a finer point on them. On the one hand, if each of our successive hidden layers tends to shrink the standard deviations of the activations, then the activations of the later hidden layers will tend to be too small for these layers to be meaningfully nonlinear. On the other hand, if each of our successive hidden layers tends to grow the standard deviations of the activations, then the later hidden layers will tend to have many dead neurons. Thus, we wish to find rescaling factors that *preserve* these standard deviations.

Given a hidden layer, both the dimensionality of its input and the choice of activation function contribute to the standard deviation of its activations. We illustrate these two effects in turn. For this, we introduce

to combat this, with a NN with ReLU neurons, after doing the training, one can pass all the examples through again and try to find which neurons never activate. [see around 22:30.]

(mention that in building our backprop engine we are of course actually *defining* the derivative, and we can define it to be either 0 or 1 at 0 – it almost never happens so nbd.)

- leaky ReLU, say $t \mapsto \max\{0.1t, t\}$. (so derivative for $t < 0$ is 0.1, not 0.)
an improvement over ReLU: no flatness anywhere.
- he didn’t mention this one, but also “PReLU” = “parametric rectified linear unit” – a generalization of LReLU. see He–Zhang–Ren–Sun [here](#). they take $t \mapsto \max\{\alpha t, t\}$ for small positive α , but take α *itself* as a parameter! (and allow it to vary across the various neurons.)
- maxout: $\max\{w_1^T t + b_1, w_2^T t + b_2\}$ [he doesn’t mention this one, so probably don’t include it here. (or at least, he doesn’t mention it in the BatchNorm lecture, where the rest of these are shown/discussed...)]
- ELU: t when $t \geq 0$, and $\alpha \cdot (e^t - 1)$ for $t < 0$ (so decays towards $-\alpha$ as $t \rightarrow -\infty$).
this is also frequently used, but can also suffer from this issue: it’s fairly flat on the left (just a little below zero).

⁸³Indeed, $|\tanh(t)| > 0.99$ if and only if $|t| > \tanh^{-1}(0.99) = 2.6467$.

⁸⁴but again, it seems like this is primarily a question of learning rate, more than initialization?

⁸⁵The paper [ref] also studies the effect of initialization on backpropagation. (If certain gradients are too small then the corresponding neurons won’t learn, while if they are too large then the corresponding neurons’ parameters may change too quickly.) The conclusion of this study is that properly initializing the forward pass effectively addresses this issue as an added bonus.

the terms *fan-in* and *fan-out*⁸⁶ of the hidden layer: these respectively refer to the dimensionalities of its input and output.⁸⁷

As for the fan-in (i.e. the dimensionality of the input), we perform a simple experiment which suggests that the dot product of two random vectors in $\mathbb{R}^n$ whose entries are initialized from $\mathcal{N}(0, 1)$ is a random variable with standard deviation $\sqrt{n}$.^{88,89} So, we should rescale the output of the hidden layer by a factor of $\frac{1}{\sqrt{n}}$ in order to account for this phenomenon, where n denotes the fan-in.

As for the choice of activation function, the essential observation is that when fed a random variable these tend to change its standard deviation. The choice of scale factor to account for this phenomenon is called the *gain*, and the Kaiming initialization associates to each of a host of standard activation functions a corresponding optimal gain. For example, that of $\tanh$ is $\frac{5}{3}$; it should be expected that this is greater than 1, since $\tanh$ compresses the entire real line down to the interval $(-1, 1)$. Similarly, the optimal gain of the ReLU function $t \mapsto \max\{0, t\}$ is $\sqrt{2} = 1.4142$ (which should likewise be expected to be greater than 1).⁹⁰

Let us return to the particular example of our quadgram model. In §4.1.2, we ultimately chose to rescale the weights of its hidden layer by a factor of 0.2. Since its neurons have activation function $\tanh$ and its fan-in is 30, the Kaiming initialization prescribes a scale factor of $\frac{5/3}{\sqrt{30}} = 0.3043$. So, our somewhat arbitrarily made choice was quite close to that of the Kaiming initialization; and in fact, it just so happens that the former performs very slightly better after 200k epochs of training, as recorded in Figure 9. In any case, the main salient point is that the Kaiming initialization gives us a principled method of choosing these scale factors that does not require any experimentation to determine.⁹¹

Although the Kaiming initialization remains very commonly employed, its necessity – that is, the sensitivity of the training process to the hyperparameters that it prescribes – has greatly diminished since it was first introduced in 2015. This is due to further subsequent innovations, including residual connections [coming up later, supposedly?], normalization layers (e.g. batch normalization (discussed in §4.2), layer normalization, instance normalization, and group normalization [any of these later?]), and improved optimizers [e.g. RMSprop and especially Adam – either of these later?].

4.2. Batch Normalization.

4.2.0. *Overview.* In this subsection, we introduce the technique of *batch normalization*, or *batchnorm* for short. First proposed in [Ioffe-Szegedy], this amounts to the interspersing of *batch normalization layers* in our neural network. As we will verify using our diagnostic tools in §4.3, these layers make our training process much more robust to small changes in the (previously very delicate) scale factor hyperparameters discussed in §4.1.

Recall from §4.1.2 that at initialization we would like for our hidden neurons’ activations – or equivalently their preactivations – to be neither too small nor too large. The idea of batch normalization is to simply enforce this balance “by hand”: we manually adjust the preactivations to have an explicitly specified mean and standard deviation! On the other hand, we then take these values – the mean and standard deviation of the adjusted preactivations – as parameters, which is to say that we tune them during the training process.

We describe the batch normalization layer in §4.2.1. As we will see, batch normalization necessitates a divergence between the training and inference processes. In effect, working with batches of size b , our batchnorm training will consist of training a neural network $(\text{Chars}^3)^b \rightarrow (\Delta^{26})^b$, from which we must then extract a neural network $\text{Chars}^3 \rightarrow \Delta^{26}$ in order to perform inference. We discuss the batchnorm training process in §4.2.2 and the batchnorm inference process in §4.2.3.

⁸⁶what does the word “fan” mean here?

⁸⁷So for instance, in our quadgram model, the hidden layer has a fan-in of 30 and a fan-out of 200.

⁸⁸Specifically, we initialize a 1,000-by-10 matrix X and a 10-by-200 matrix W , both with entries sampled from $\mathcal{N}(0, 1)$, and we find that the entries of the product XW have standard deviation of $3.0996 \approx 3.1623 = \sqrt{10}$.

⁸⁹Beware however that the product of two normally distributed random variables is *not* generally normally distributed.

⁹⁰i could even include a list of more of these recommended gains, perhaps as a table. they’re listed in the docs for `nn.init.kaiming_normal`. as for the activation functions we’ve discussed: for a linear function (or maybe just the identity) it’s 1, for sigmoid it’s 1, for $\tanh$ it’s $5/3$, for ReLU it’s $\sqrt{2}$, for Leaky ReLU it’s $\sqrt{\frac{2}{1+a^2}}$ where a denotes the negative slope.

⁹¹he doesn’t do anything about the initialization of $\vec{b}_1$. what does kaiming say?

4.2.1. *The Batch Normalization Layer.* As the name suggests (and as indicated in §4.2.0), batch normalization applies when training in batches. In fact, it adjusts the preactivations of each hidden neuron separately. So, we describe the batchnorm layer by first describing its action on a list of b scalar (i.e. 1-dimensional) values, which in §4.2.2 we will take to be the preactivations for a single hidden neuron corresponding to a batch of b inputs.

We begin with some abstract considerations. Suppose we are given some point $\vec{t} := (t_i)_i := (t_1, \dots, t_b) \in \mathbb{R}^b$. Then, we can define the mean and standard deviation

$$\mu := \frac{1}{b} \sum_i t_i \quad \text{and} \quad \sigma := \sqrt{\frac{1}{b} \sum_i |t_i - \mu|^2}$$

of its coordinates.⁹² These definitions assemble into functions

$$\mathbb{R}^b \xrightarrow{\mu} \mathbb{R} \quad \text{and} \quad \mathbb{R}^b \xrightarrow{\sigma} \mathbb{R}_{\geq 0}.$$

Observe that we have $\sigma(\vec{t}) = 0$ if and only if $t_1 = t_2 = \dots = t_b$. In particular, if we assume that the point $\vec{t} \in \mathbb{R}^b$ does not lie on the diagonal $\Delta := (1, \dots, 1) \cdot \mathbb{R} \subseteq \mathbb{R}^b$, then the standard deviation $\sigma(\vec{t})$ of its coordinates is strictly positive. Hence, we can define the *standard batch normalization function*

$$\mathbb{R}^b \setminus \Delta \xrightarrow{\text{BN}_{1,0}} \mathbb{R}^b \quad \text{given by} \quad \text{BN}_{1,0}(\vec{t}) := \left(\frac{t_i - \mu(\vec{t})}{\sigma(\vec{t})} \right)_i.$$

By construction, all values of this function have coordinates with mean 0 and standard deviation 1: for any $\vec{t} \in \mathbb{R}^b \setminus \Delta$, we have that

$$\mu(\text{BN}_{1,0}(\vec{t})) := \frac{1}{b} \sum_i \text{BN}_{1,0}(\vec{t})_i := \frac{1}{b} \sum_i \frac{t_i - \mu(\vec{t})}{\sigma(\vec{t})} = \frac{1}{\sigma(\vec{t})} \cdot \left(\frac{1}{b} \sum_i t_i - \frac{1}{b} \sum_i \mu(\vec{t}) \right) = \frac{1}{\sigma(\vec{t})} \cdot \left(\mu(\vec{t}) - \frac{1}{b} \cdot b \cdot \mu(\vec{t}) \right) = 0$$

and that

$$\sigma(\text{BN}_{1,0}(\vec{t})) := \sqrt{\frac{1}{b} \sum_i |\text{BN}_{1,0}(\vec{t})_i - \mu(\text{BN}_{1,0}(\vec{t}))|^2} = \sqrt{\frac{1}{b} \sum_i \left| \frac{t_i - \mu(\vec{t})}{\sigma(\vec{t})} \right|^2} = \frac{1}{\sigma(\vec{t})} \cdot \sqrt{\frac{1}{b} \sum_i |t_i - \mu(\vec{t})|^2} = 1.$$

More generally, for any parameters $\gamma, \beta \in \mathbb{R}$ we can define a corresponding *batch normalization function*

$$\mathbb{R}^b \setminus \Delta \xrightarrow{\text{BN}_{\gamma,\beta}} \mathbb{R}^b \quad \text{given by} \quad \text{BN}_{\gamma,\beta}(\vec{t}) := \text{BN}_{1,0}(\vec{t}) \cdot \gamma + \beta := \left(\frac{t_i - \mu(\vec{t})}{\sigma(\vec{t})} \cdot \gamma + \beta \right)_i,$$

whose values all have coordinates with mean β and standard deviation $|\gamma|$. We refer to γ as the **gain** and to β as the **bias** of the batch normalization function $\text{BN}_{\gamma,\beta}$.⁹³

Now, let us write h for the number of neurons in the hidden layer into which we will intersperse our batchnorm layer in §4.2.2. As indicated above, this interspersing will amount to the interspersing of a batch normalization function separately in each neuron. In particular, the i^{th} neuron will come equipped with its own gain and bias parameters $\gamma_i, \beta_i \in \mathbb{R}$; collectively, these parameters assemble into vectors $\vec{\gamma}, \vec{\beta} \in \mathbb{R}^h$. Altogether, then, our **batch normalization layer** amounts to the function

$$(\mathbb{R}^b \setminus \Delta)^h \xrightarrow{\text{BN}_{\vec{\gamma}, \vec{\beta}} := (\text{BN}_{\gamma_i, \beta_i})_i} (\mathbb{R}^b)^h$$

given by applying the batch normalization function $\text{BN}_{\gamma_i, \beta_i}$ in the i^{th} coordinate.

⁹²Obviously, we assume here and throughout that $b > 0$. (It must surely be a vacuous operation to train a neural network on an empty batch!)

⁹³One can slightly modify this function so that it is defined over all of $\mathbb{R}^b$ (and not just $\mathbb{R}^b \setminus \Delta$) by fixing some small parameter $\varepsilon > 0$ and defining

$$\text{BN}_{\gamma,\beta}^\varepsilon(\vec{t}) := \left(\frac{t_i - \mu(\vec{t})}{\sqrt{\sigma(\vec{t})^2 + \varepsilon}} \cdot \gamma + \beta \right)_i.$$

For instance, this appears in the PyTorch implementation of batchnorm [discuss this sort of thing?], where the default value is taken to be $\varepsilon = 10^{-5}$. (Clearly, $\lim_{\varepsilon \rightarrow 0^+} \text{BN}_{\gamma,\beta}^\varepsilon(\vec{t}) = \text{BN}_{\gamma,\beta}(\vec{t})$ for any $\vec{t} \in \mathbb{R}^b \setminus \Delta$.)

4.2.2. *Batch Normalization Training.* Recall that our quadgram model is ultimately just a function $\text{Chars}^3 \xrightarrow{F} \Delta^{26}$ (depending on certain parameters). Previously, when training it on a batch of b quadgram datapoints $(\vec{x}_i, y_i^{\text{gt}}) \in \text{Chars}^3 \times \text{Chars} \cong \text{Chars}^4$, we simply applied this same function b times – once to each trigram input $\vec{x}_i \in \text{Chars}^3$. Specifically, each of these inputs determined a probability distribution $F(\vec{x}_i) \in \Delta^{26}$, which contributed to our loss function according to the probability that it assigns to the corresponding ground truth value $y_i^{\text{gt}} \in \text{Chars}$.

By contrast, in batch normalization training, we take the entire batch $\{(\vec{x}_i, y_i^{\text{gt}})\}_i$ as a single input of a function $(\text{Chars}^3)^b \xrightarrow{F_{\text{BN}}} (\Delta^{26})^b$. To obtain this function, we augment our (randomly initialized) quadgram model F by first replicating it b times⁹⁴ to obtain a function $(\text{Chars}^3)^b \xrightarrow{F^b} (\Delta^{26})^b$ and then interspersing a batchnorm layer in its hidden layer. More precisely, the b -fold replication of our hidden layer factors as a composite

$$\begin{array}{ccc} (\mathbb{R}^{30})^b & \xrightarrow{(\tanh((-)W_1 + \vec{b}_1))^b} & (\mathbb{R}^{200})^b \\ & \searrow^{((-)W_1 + \vec{b}_1)^b} & \nearrow^{\tanh^b} \\ & (\mathbb{R}^{200})^b & \end{array}$$

of the b -fold replications of an affine layer followed by a tanh layer, and within this factorization we intersperse a batchnorm layer to obtain the composite

$$(\mathbb{R}^{30})^b \xrightarrow{((-)W_1 + \vec{b}_1)^b} (\mathbb{R}^{200})^b \cong (\mathbb{R}^b)^{200} \xrightarrow{\text{BN}_{\vec{\gamma}, \vec{\beta}}} (\mathbb{R}^b)^{200} \cong (\mathbb{R}^{200})^b \xrightarrow{\tanh^b} (\mathbb{R}^{200})^b$$

for some vectors of gain and bias parameters $\vec{\gamma}, \vec{\beta} \in \mathbb{R}^{200}$.^{95,96,97} We initialize these vectors as $\vec{\gamma} = (1, \dots, 1)$ and $\vec{\beta} = (0, \dots, 0)$, so that our batchnorm layer is initialized as the standard batchnorm function $\mathbb{R}^b \xrightarrow{\text{BN}_{1,0}} \mathbb{R}^b$ in each of the 200 coordinates.

We then train our neural network $(\text{Chars}^3)^b \xrightarrow{F_{\text{BN}}} (\Delta^{26})^b$ (including tuning the parameters $\gamma_i, \beta_i \in \mathbb{R}$, and with all hyperparameters (e.g. learning rate and number of epochs) unchanged), and obtain a training loss of 2.0668 and a validation loss 2.1048. These are not really better than the loss values obtained without the batchnorm layer (see Figure 9); but again, we shouldn’t expect the batchnorm layer to give us much improvement in this situation, since we only have one hidden layer and have already implemented the Kaiming initialization (which controls the mean and standard deviation of the hidden activations).

It is worth noting that whereas previously our neural network’s output for any given input trigram was entirely deterministic, with batchnorm its output depends not just on the given input trigram but also on the other input trigrams in the batch. This nondeterministic behavior may seem like a bug, but in fact it can be seen as a feature: it serves as a sort of regularization for our training process, preventing the model from overfitting by “memorizing” the training data.⁹⁸

⁹⁴i.e. take the whole thing to the b^{th} (cartesian) power

⁹⁵Here, we make the notational distinction between $(\mathbb{R}^{200})^b$ and $(\mathbb{R}^b)^{200}$ to emphasize the difference in how these functions operate: whereas the affine and tanh layers operate separately on each vector arising from a single input trigram in our batch, the batchnorm layer operates on preactivations occurring across the entire batch but separately in each of the 200 coordinates.

⁹⁶Having taken care to indicate in §4.2.1 that the batch normalization function is *not* defined on the diagonal $\Delta \subseteq \mathbb{R}^b$ (Footnote 93 notwithstanding), we presently ignore this subtlety in order to simplify our notation. Indeed, as long as $b > 1$, the preactivations will almost certainly not lie on the diagonal. (We address the case that $b = 1$ – i.e. the inference process – in §4.2.3.)

⁹⁷Note that this interspersing renders the bias vector $\vec{b}_1 \in \mathbb{R}^{200}$ effectively obsolete. Specifically, adding the bias $(\vec{b}_1)_i \in \mathbb{R}$ to all preactivations of the i^{th} neuron is completely redundant: for any $\vec{t} \in \mathbb{R}^b$, writing $\vec{t}' := (t_j + (\vec{b}_1)_i)_j \in \mathbb{R}^b$, we have $\text{BN}_{\gamma, \beta}(\vec{t}) = \text{BN}_{\gamma, \beta}(\vec{t}')$ for any $\gamma, \beta \in \mathbb{R}$ (since $t_j - \mu(\vec{t}) = t'_j - \mu(\vec{t}')$ and $\sigma(\vec{t}) = \sigma(\vec{t}')$). Hence, in this case we have $\frac{d\ell}{d(\vec{b}_1)_i} = 0$, so that these bias parameters will *never* change through training. So in practice, a batchnorm function should only follow a linear layer, not an affine one. (Note too that our neural network will still be able to bias the outputs of the batchnorm layer via the biases of the batchnorm function.)

⁹⁸he says something like: “sociologically, this makes it hard for us to remove the use of batchnorm, even though we dislike the nondeterministicness of it from an aesthetic perspective.” double-check this. “people have tried to deprecate it, and move to other normalization techniques: linear, instance, group normalizations – we will (maybe?) cover some of them later. but batchnorm was the first to be introduced.”

4.2.3. Batch Normalization Inference. As we already asserted in §4.2.0, in the presence of a batchnorm layer, the process of inference must now diverge from the process of training. Indeed, we can now no longer feed a single input into our neural network, since for any point $\vec{t} := (t_1) \in \mathbb{R}^1$ the standard deviation $\sigma(\vec{t})$ of its single coordinate is always zero (and correspondingly we now have an equality $\Delta = \mathbb{R}^1$: the diagonal consists of the entire real line). And similarly, it is a rather degenerate operation to adjust the coordinate t_1 by subtracting the mean $\mu(\vec{t}) = t_1$!

The most obvious way to handle this issue is to *calibrate* the batchnorm statistics: after training, we can simply calculate the means and standard deviations of the hidden preactivations across the entire training set (or some sufficiently large representative subset thereof). Let us respectively denote by μ_i and σ_i the mean and standard deviation of the preactivations of the i^{th} hidden neuron. Then, we can define a corresponding *batch normalization inference function*

$$\mathbb{R} \xrightarrow{\text{BN}_{\gamma_i, \beta_i}^{\text{inf}, \mu_i, \sigma_i}} \mathbb{R} \quad \text{given by} \quad \text{BN}_{\gamma_i, \beta_i}^{\text{inf}, \mu_i, \sigma_i}(t) := \frac{t - \mu_i}{\sigma_i} \cdot \gamma_i + \beta_i, \quad 99$$

where $\gamma_i, \beta_i \in \mathbb{R}$ are the gain and bias parameters that were tuned during the training process. These statistics assemble into vectors $\vec{\mu}, \vec{\sigma} \in \mathbb{R}^h$, and altogether we obtain a **batch normalization inference layer**

$$\mathbb{R}^h \xrightarrow{\text{BN}_{\vec{\gamma}, \vec{\beta}}^{\text{inf}, \vec{\mu}, \vec{\sigma}} := (\text{BN}_{\gamma_i, \beta_i}^{\text{inf}, \mu_i, \sigma_i})_i} \mathbb{R}^h$$

given by applying the batch normalization inference function $\text{BN}_{\gamma_i, \beta_i}^{\text{inf}, \mu_i, \sigma_i}$ in the i^{th} coordinate. Using this, we can now define a quadgram model involving our batchnorm inference layer that takes in a single input (rather than a batch of them) as the composite

$$\text{Chars}^3 \xrightarrow{(\vec{e}_{\text{ind}(-)})^3} (\mathbb{R}^{27})^3 \xrightarrow{C^3} (\mathbb{R}^{10})^3 \cong \mathbb{R}^{30} \xrightarrow{(-)W_1 + \vec{b}_1} \mathbb{R}^{200} \xrightarrow{\text{BN}_{\vec{\gamma}, \vec{\beta}}^{\text{inf}, \vec{\mu}, \vec{\sigma}}} \mathbb{R}^{200} \xrightarrow{\tanh} \mathbb{R}^{200} \xrightarrow{(-)W_2 + \vec{b}_2} \mathbb{R}^{27} \xrightarrow{\sigma} \Delta^{26},$$

where throughout the composite we use the specific parameters that were tuned during the training process. Of course, we should compute our loss values using this modified neural network, and doing so gives us a training loss of 2.0668 and a validation loss of 2.1049 – which values are essentially identical to those obtained in §4.2.2.

On the other hand, it may be desirable to avoid this explicit calibration step. As an alternative, we can instead approximate these exact statistics via **exponential (ly weighted) moving averages**. This proceeds as follows. For the i^{th} hidden neuron, we initialize values $\mu_i^{\text{ema}} = 0$ and $\sigma_i^{\text{ema}} = 1$ (which are indeed respectively the mean and standard deviation at initialization, by construction). Then, in each forward pass, if its preactivations are given by the vector $\vec{t} \in \mathbb{R}^b$, we “drag” these values towards the computed values $\mu(\vec{t})$ and $\sigma(\vec{t})$, by resetting μ_i^{ema} to equal $(1 - m) \cdot \mu_i^{\text{ema}} + m \cdot \mu(\vec{t})$ and resetting σ_i^{ema} to equal $(1 - m) \cdot \sigma_i^{\text{ema}} + m \cdot \sigma(\vec{t})$ for some small positive *momentum* value m .^{100, 101}

To illustrate the essential interchangeability of these two techniques for inference, we reinitialize our neural network and train it for 200k epochs, computing our exponential moving averages with momentum 0.001.¹⁰² The exact and approximated statistics of the first few hidden neurons’ preactivations after this training are recorded in Figure 12; as desired, all approximations are very close to the corresponding exact values. And indeed, we obtain essentially identical loss values using these approximated statistics: a training loss of 2.0666 and a validation loss of 2.1051.¹⁰³

⁹⁹Almost certainly it will be the case that $\sigma_i > 0$!

¹⁰⁰Note that the exponential moving average of a sequence $a_0, \dots, a_k$ is equal to $(1 - m)^k \cdot a_0 + m \cdot \sum_{j=1}^k (1 - m)^{k-j} \cdot a_j$ (which explains the terminology). In particular, aside from the initial value a_0 this assigns higher weight to later values (i.e. those a_j for larger j), at least assuming that $m < 1$ (which is certainly reasonable). Such behavior is desirable in our situation, since we expect these batch-level statistics $\mu(\vec{t})$ and $\sigma(\vec{t})$ to be better and better approximations to the exact values μ_i and σ_i (computed after training is complete) as our training proceeds.

¹⁰¹In particular, these values are *not* parameters: we do not compute their infinitesimal contribution to the loss, nor do we modify them in the update steps of our training process. To emphasize this distinction, one may refer to these values as *buffers*.

¹⁰²in PyTorch, the default value is 0.1. a nice point: the larger your batch size, the more you can trust these values, so a higher momentum is fine. but 32 is tiny. (also, obviously these two momenta could be decoupled, no need to take them to be equal.)

¹⁰³after this, andrej goes on a long digression illustrating how the various things we’ve discussed appear in the `torch.nn` code for resnet50. this seems irrelevant for the present purposes; no new ideas are discussed. (and it’s also confusing: it’s about

μ_i	μ_i^{ema}	σ_i	σ_i^{ema}
−2.3355	−2.3539	2.3929	2.3689
0.6775	0.6872	2.1364	2.0998
−0.9132	−0.9000	2.2663	2.2499
1.0163	1.0159	1.9944	1.9599
1.0865	1.0894	2.1860	2.1515

FIGURE 12. The i^{th} rows of these tables record the exact and approximated statistics of the i^{th} hidden neuron’s preactivations after 200k epochs of training.

4.3. Diagnostics.

convolutions, but we haven’t discussed those, so he just talks about linear layers instead even when displaying that code.) also, they initialize from *uniform* distribution, instead of normal. also btw, in “summary” he claims we’ll eventually do RNNs, but none so far (after 8 videos)...