Randomization in Machine Learning: Weight Initialization and Training[*]

Caleb Manicke

1 Abstract

In recent times, no field of computer science has seen as much growth or attention than machine learning. From self-driving cars to chat bots giving homework answers for students, machine learning models has reached unprecedented levels of sophistication. However, these impressive feats have come at a cost. Training algorithms have become so complicated that we have not been able to assign them to a time complexity class. Furthermore, recent models have become so resource intensive that even on state-of-the-art hardware they take weeks to train. For these reasons, randomization has become a standard practice in model training. Specifically, randomizing initial model weights then updating those weights on randomly sampled training examples has shown to produce better performing models. Therefore, this paper will explore randomization in weight initialization and training by giving a formal introduction to machine learning classifiers, a working intuition of classifier loss, a rigorous proof for our training algorithm as well as results from experiments that utilize these methods.

[*]Source: `https://www.overleaf.com/project/63d07d57e3c07006b484dc9d`

2 Formal Definitions and Objective

Before we uncover how randomization helps optimize machine learning model training, we need to formally define what it is we are try to optimizing.

Definition 1 (Image)**.** *An image is a row vector with d_x real values, or pixels, denoted $x_i = (x_{i,1}, ..., x_{i,d_x}) \in \mathbb{R}^{d_x}$.*

Definition 2 (Dataset of Images)**.** *A data set is a finitely countable set of points $\{x_i\}_{i=1}^n$.*

Definition 3 (Label)**.** *A label y is a discrete value belonging to a finite set of integers $\{1, 2, ..., n\}$. Each integer corresponds to a discrete class.*

A set of labels is a countably finite set of $\underline{\text{integers}}$ denoted $\{y_i\}_{i=1}^n$. Each value y_i denotes a class label for its corresponding image x_i.

Example - Suppose our dataset of images could be categorized into 10 labels {Cat, Dog, Plane, Truck, ...}. Each x_i is an *image* of an object in one of these classes. Then the label of x_i, denoted y_i, should be the integer which denotes which class x_i belongs to, so if x_3 is a picture of an airplane, then $y_i = 3$.

Definition 4 (Training Set)**.** *Given a set of images 2 and labels 3 for these images, a training set is a set of pairs of each image and their corresponding label $\{(x_1, y_1), (x_2, y_2), ...\}$.*

Suppose we have a training set of images and labels. A machine learning model would use this training set in order to find a set of parameters that would allow it to accurately choose class labels for new images. This is an optimization problem: we want to find a classifier, i.e. a model that picks a label for an input image, that minimizes the number of times it mislabels an image [3].

Definition 5 (Classifier)**.** *A classifier is a function that maps an image $x_i \in \mathbb{R}^{d_x}$ to a label $y_i = \{1, 2, 3, ...\}$, so $h : \mathbb{R}^{d_x} \to \{1, 2, 3, ...\}$.*

Definition 6 (New Classifier Objective)**.** *We want to find a classifier h such that $h(x_i) = y_i$ for most i, $1 \le i \le n$. Formally, we want to solve*

$$\min_n \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}(h(x_i) \neq y_i)$$

i.e. minimize the number of examples for which the label our classifier outputs is wrong. Recall that the indicator function $\mathbb{1}$ takes events as input, so $\mathbb{1}(A) = 1$ if A is true and $\mathbb{1}(A) = 0$ otherwise.

This is our formal optimization problem. However, we need to make some tweaks to this: if we want to optimize the number of examples our classifier gets right, then we need to find the right set of *parameters* for our classifier. Our problem is exactly like trying to find the best fit-line in a plane: if $h(x) = ax + b$ is our best fit line, then we want to find **weights** $w = \{a, b\}$ such that for each example x_i, $h(x_i)$ is as close to each y_i as possible. We generalize this to saying the weights for any classifier h exist as a d-dimensional real vector, $w \in \mathbb{R}^d$. Notice that d is not necessarily our number of pixels d_x!

Another problem is that since $\mathbb{1}$ only returns either a 1 or 0, our minimization function only returns Boolean values. This is not the best for optimization problems: it's much easier to optimize a continuous and differentiable function since we can use its derivative to update its weights. We define a continuous **loss** function $\mathcal{L} : \mathbb{R} \times \mathbb{R} \to [0, \infty)$ that approximates $\mathbb{1}$. Intuitively, $\mathcal{L}(x, y)$ will tell us how "close" x and y are to one another. This allows us to redefine our classifier objective as follows:

Definition 7 (Classifier Objective)**.** *We want to solve...*

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(h(x_i; w), y_i)$$

i.e. we want to find the best weights $w \in \mathbb{R}^d$ such that for each example image x_i, the value our loss function returns after comparing our model's output with the correct label is minimally small.

Consider a weight vector $w \in \mathbb{R}^d$ such that $d >> d_x$. We can break-apart the weight vector into sub-matrices that when vectorized and concatenated vertically give us our original weight: $w = w_a \circ w_b \circ \ldots$. Our classifier will then become $h(x) = w_a \cdot w_b \cdot \ldots \cdot x \in \mathbb{R}^m$. We can attribute each sub-weight matrix w_i as a layer in our model.

Suppose our labels y lived in the range 1 to some integer m. Since each submatrix $w_a, w_b, ...$ is a real matrix, our output is a m-dimensional real vector. We can easily turn this into a probability vector by applying a normalized exponential transformation as seen below [4]:

$$\frac{e^{h(x)}}{\sum_{i=1}^{m} e^{h_i(x)}} = \begin{bmatrix} \mathcal{P}(y = 1 | X = x) \\ \cdots \\ \mathcal{P}(y = m | X = x) \end{bmatrix}$$

The normalized exponential transformation, commonly called the Soft-Max transformation, will clampe each real value outputted by $h(x)$ to a 0-1 data range. This allows us to interpret each entry $h_j \in h(x)$, $1 \leq j \leq m$ as the likelihood that our input x_i belongs to our j-th label.

This completes our formal introduction of machine learning classifiers. Informally, a classifier uses a set of weights to output a vector where each entry is the probability an input x_i belongs to a class. Note that we have not yet defined *how* we can find the right set of weights $w \in \mathbb{R}^d$ to solve our minimization problem 7 or what our loss function $\mathcal{L} : \mathbb{R} \times \mathbb{R} \to [0, \infty)$ should be. We will keep these as black-box terms for now and introduce them later.

3 Vanishing and Exploding Gradients Problem

We defined a classifier over several layers as a product of sub-matrices with arbitrary dimensions w_i, $h(x) = w_a \cdot w_b \cdot \ldots \cdot x \in \mathbb{R}^m$, that produces an m-dimensional probability vector. Since we are taking the product over several matrices, there are two scenarios that can occur.

Case #1: For simplicity, suppose we have a binary input $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ and all of our sub-weight matrices were 2×2 matrices $w_i = \begin{bmatrix} c & 0 \\ 0 & c \end{bmatrix}$ for some real $c > 1$. Our first product gives us $w_z \cdot x = \begin{bmatrix} c & 0 \\ 0 & c \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} c \cdot x_1 \\ c \cdot x_2 \end{bmatrix} = c \cdot x$. Then our output past the l-th layer becomes $c^l x$, an exponentially large output!

This is our exploding gradients problem [9]. To perceptual this, consider the following: all of our images exist in an “input space”. This input space is a subset of the reals where our pixel values live in. The purpose of a classifier is to convert information in the input space to a “feature space” where it can decide the likelihood an image belongs to each class label.

When the majority of sub-weight matrices are “larger” than the identity matrix for their corresponding dimensions, our original data is sent from a defined real input space to an exponentially larger feature space. This means we loose the original input information at an exponential pace! Because our input information is spread thin with much less correlation with one another, it is much larger for our classifier to draw parallels between each pixel and predict what class label our original image belonged to. By this intuition then, our classifier will perform very poorly.

Case #2: Consider the opposite scenario, where for every weight matrix $w_i = \begin{bmatrix} c & 0 \\ 0 & c \end{bmatrix}$ we have $0 < c < 1$. Then our output past the l-th layer becomes $c^l x << x$, which is exponentially small!

This is our vanishing gradient problem [9]. The intuition is opposite to that of the exploding gradient problem. Because each sub-matrix decreases the range of values where our input image exists in, the feature space is exponentially smaller than the input space. We are sending our input information to a fraction of its real space, so we are loosing information with each layer, which would make for worse classifier performance.

Vanishing and exploding gradients make particular sets of weights terrible for classifiers. Although one may assume we can “learn” better weights via training, the worst part of vanishing and exploding gradients is that they circumvent model training. For that, we must consider *how* models learn.

3.1 Gradient Descent

Algorithm 1 Gradient Descent [1]

```
1. Receive initial weight $w \in \mathbb{R}^d$, dataset $\{x_i\}_1^{n=1}$ with labels $\{x_i\}_1^{n=1}$, max iterations $T \in \mathbb{N}$, and constant step sizes $\{a_k\}_{k=1}^T$
2. For $k = 1, ..., T$ do:
3.   For $1 \leq i \leq n$ do:
4.    Add $\mathcal{L}_{\text{Total}} := \mathcal{L}_{\text{Total}} + \mathcal{L}(h(x_i; w), y_i)$
4.   Update $w := w - \alpha_k \cdot \Delta\mathcal{L}_{\text{Total}}$
```

Before it can be explained what Gradient Descent does,like our explanation for vanishing and exploding gradients, we must intuitively define what *space* it works in. We can imagine our Parameter $\times$ Loss space as a bowl in $\mathbb{R}^d \times \mathbb{R}$ dimensions. Here, each possible set of parameters in our parameter space $\mathbb{R}^d$ maps to an "accuracy" in the loss space in $\mathbb{R}$ via our loss function $\mathcal{L} : \mathbb{R} \times \mathbb{R} \to [0, \infty)$. Because we are dealing with a "bowl", we can imagine that there is an "optimal" set of parameters with the lowest possible loss in the middle of our parameter space.

We can find this "optimal", minimal loss weight by descending *against* the direction of our score's gradient. At each step k, our weight is "hugging" the convex bowl at a particular loss point $\mathcal{L}_{\text{Total}}$. We can find the direction of the tangent line that our weight creates with the bowl, $\delta\mathcal{L}_{\text{Total}}$, and go in the *opposite* direction to converge towards the center.

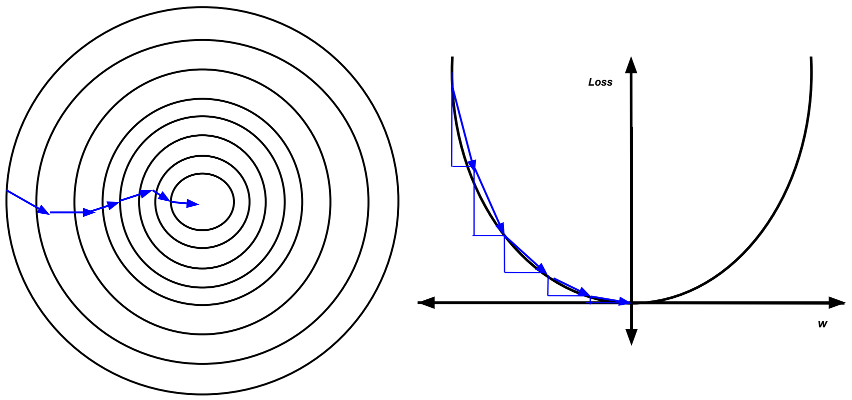

Figure 1: Image that our parameter $\times$ loss space is in two-dimensions, where each ring in our diagram above represents a cross-section of our bowl. Gradient Descent will converge to the minimal loss by following the direction of the tangent line our current weights create with respect to the convex bowl.

Gradient descent is the most vanilla method of training models, with the update rule $w := w - \alpha_k \cdot \Delta\mathcal{L}_{\text{Total}}$ serving as our most integral component of every "weight optimization" algorithm. But it is exactly this gradient term in our update rule that makes vanishing and exploding gradients a larger problem.

3.2 Vanishing & Exploding Gradients in Gradient Descent

Consider the gradient sum $\Delta\mathcal{L}_{\text{Total}}$, which can be extended as follows: $\Delta\left[\sum_{i=1}^{n} \mathcal{L}(h(x_i, w), y_i)\right] = \sum_{i=1}^{n} \Delta\mathcal{L}(h(x_i, w), y_i)$. Since we derive with respect to our model weights, by the chain rule, we get that $\Delta\mathcal{L}(h(x_i, w), y_i) = \dfrac{\partial\mathcal{L}(h(x_i, w), y_i)}{\partial w} = \dfrac{\partial\mathcal{L}(h(x_i, w), y_i)}{\partial h(x_i, w)} \cdot \dfrac{\partial h(x_i, w)}{\partial w}$.

Consider our classifier $h(x_i) = f(w_a \cdot w_b \cdot \ldots \cdot x_i)$, where f is our normalized exponential transformation defined above 2. Deriving with respect to any label y_j will give us $\dfrac{\partial h(x_i, w)}{\partial w} = \dfrac{\partial h(x_i, w)}{\partial y_j} \cdot \dfrac{\partial y_j}{\partial w} = \dfrac{\partial y_i}{\partial w} \cdot f(w_a \cdot w_b \cdot \ldots \cdot x_{i,j}) \cdot (1 - f(w_a \cdot w_b \cdot \ldots \cdot x_i))$ [8].

For simplicity, suppose $w_a \cdot w_b \cdot \ldots \cdot x_i \approx c \cdot x_i$ for some $c > 0$ where $d_x = m$, so our input and output dimensions match. Then our term above becomes the following: $\dfrac{\partial h(x_i, w)}{\partial w} = \dfrac{\partial y_i}{\partial w} \cdot f(c \cdot x_{i,j}) \cdot (1 - f(c \cdot x_i)) = \dfrac{\partial y_i}{\partial w} \cdot \dfrac{e^{\,c \cdot x_{i,j}}}{e^{\,c \cdot x_{i,1}} + \ldots + e^{\,c \cdot x_{i,m}}} \cdot \left(1 - \dfrac{e^{\,c \cdot x_i}}{e^{\,c \cdot x_{i,1}} + \ldots + e^{\,c \cdot x_{i,m}}}\right)$. Since $e^{\,c \cdot x_{i,j}}$ and $e^{\,c \cdot x_{i,1}} + \ldots + e^{\,c \cdot x_{i,m}}$ are reals, this is a vector multiple of $e^{\,c \cdot x_i}$.

What we have just proved is that there is a direct proportional relationship between our vanishing/exploding constant c and the derivative of classifier loss with respect to weight used in our update rule. This means that if c is small enough, $e^{c \cdot x_i} = e^c \cdot e^{x_i}$ will be extremely small, meaning regardless of what step size α we choose at step k, our gradient will be so small that our weights will barely change. If our weight w barely changes, we can expect the same loss, same rate of change of loss, and approximately the same weight for the next iteration.

The opposite problem occurs when c is large. Recall that we want our weight to converge to a local minimum in the Parameter $\times$ Loss space. If our gradient in our update step is extremely large, it becomes possible to "overstep" the minimum and end up on the other side of the bowl. This may increase the error in our next step, increasing the chance of a greater loss, and thus reducing the possibility of converging to the minimum.

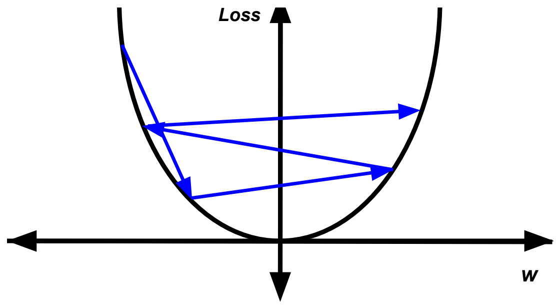

Figure 2: Exploding gradients causes our loss and gradient with respect to loss at every step to be extremely large. Because our weight and loss pair at any step is a point on the bowl, if our loss gradient increases, gradient descent may start to climb "up" the convex bowl, which will prevent us from converging to the minimal point of loss.

We have proven that vanishing and exploding gradients cannot be resolved from further training. If we cannot "learn" a better set of weights from a bad set of weights, then we need our starting set of weights to be ideal. For this reason, we turn to randomization.

4 Randomizing Model Weights

Vanishing and exploding gradients will persist when classifier weights allow for them to. Therefore, we must choose the right set of initial weights if we want to minimize our training loss. We want to investigate several weight initialization techniques in an applied setting. In practice, machine learning is an art form: choosing your model, loss function, optimization algorithm, external parameters (such as our learning parameters α_k) are all subject to change on your dataset. Here, we will see how constant and randomized weight initialization affect model performance in one setting.

We will train a model on CIFAR-10: a well-known image dataset consisting of 50,000 32×32 training examples balanced across 10 distinct labels [11]. We will be using a Convolutional Neural Network, or CNN for short: a type of classifier specifically designed to make backpropagation easier for image recognition [2]. We can skip the technical details on model architectures since our formal definitions allow us to conceptualize a classifier as a black box that use trainable weights to output a set of labels given a set of input images.

We will use the Carlini network: a shallow, three layer CNN with 30,000 trainable parameters total designed as a benchmark model in ML security [12]. We use a standard loss function and optimization algorithm for training a CNN, so Cross Entropy Loss and an Adam optimizer [16], [10]. We use a fixed learning rate of $\alpha = 0.0001$ and train over 50 epochs with a batch size of 128.

4.1 Weight Initialization Techniques

There are three weight initialization techniques we will investigate:

Constant Initialization: We will initialize all weights in our Carlini network to either a 0 or a 1. This will serve as a benchmark for training on non-randomized, uniformly distributed weights.

Kaiming Initialization: Before we can explain Kaiming initialization, we need to branch out our formal understanding to allow for another interpretation of machine learning models. We had discussed how the weight vector $w \in \mathbb{R}^d$ can be broken into sub-weight matrices $w = w_a \circ w_b \circ \ldots$ which represent different layers in a neural network. We can interpret each column in our sub-weight matrix as a "node" in a layer that accepts how many entries are in each column as an input that outputs a real number.

We can define n_l as the number of *inputs* into a layer l. Formally, since in matrix multiplication, we find $w_{j-1} \cdot w_j$ by multiplying the rows of w_{j-1} by the columns of w_j, so n_j is the number of columns in our sub-matrix up to w_j at that point in our forward computation.

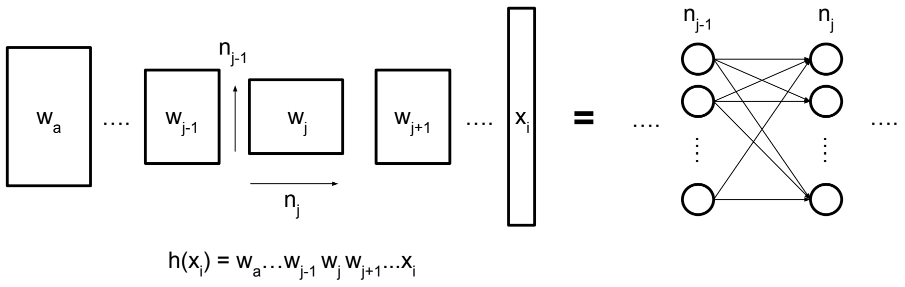

Figure 3: Imagine each of our sub-weight matrices corresponded to a layer with nodes. Each node from a previous layer feeds into each node in the next layer. The number of nodes that feed into the next layer is exactly the number of columns in our previous weight matrix or number of rows in our current weight matrix.

Kaiming Initialization sets weights in each layer l according to a normal variable with mean 0 and standard deviation $\sqrt{\frac{2}{n_l}}$ [7]. This is designed to make any variance between the input and output the same in order to mitigate vanishing gradients.

Xavier Initialization: Xavier Initialization is the industry standard for initializing convolutional layers in neural networks. This can be best explained using our notion of machine linear models as graphs as discussed above. Consider a layer l where every node takes in n_l inputs, performs some computation, and outputs n_{l+1}. For each node in this layer, Xavier Initialization samples a weight from a uniform distribution $U\left[\frac{-\sqrt{6}}{\sqrt{n_l + n_{l+1}}}, \frac{\sqrt{6}}{\sqrt{n_l + n_{l+1}}}\right]$ [5].

4.2 Experiment & Data

We will train four separate Carlini networks with weights from our initialization techniques above. We will record the total loss value over each training epochs. An ideal set of weights should allow for a smooth decrease in total loss recorded over each epoch. The smoother the curve, the more gradual our weights' descent towards the minimal loss is. If our model has no loss after a certain amount of epochs, our set of weights for that model cannot improve because our update derivative in gradient descent will be close to zero.

After training, we will evaluate all of our models on how many examples it correctly classifies in our training set, as well as 10,000 other CIFAR-10 images that we did not train on. This is our validation set, and will test how well our model generalizes to examples outside the training set. The best initialization technique should achieve the highest training and validation accuracy with ideal training behavior.

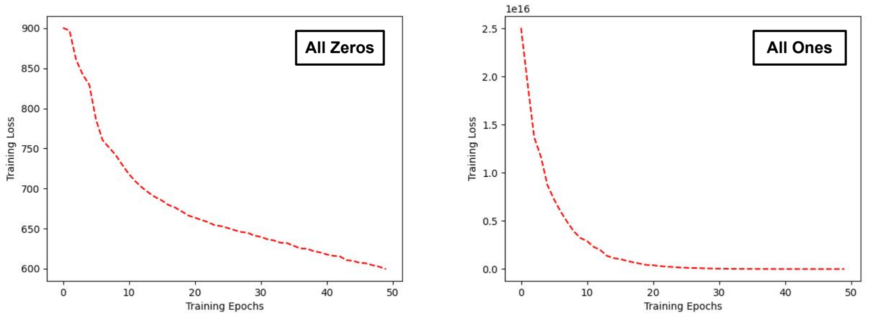

Figure 4: Resulting training loss vs. epoch graphs from when we initialize all weights in our model to a constant 0 or 1.

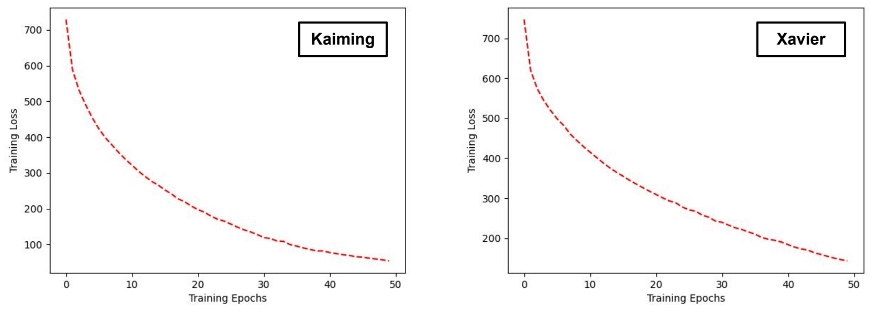

Figure 5: Resulting training loss over epoch graphs from when we initialize model weights using Kaiming or Xavier initialization

	All Ones	All Zeros	Kaiming	Xavier
Training Accuracy	10.0%	45.83%	99.33%	93.95%
Validation Accuracy	10.0%	45.15%	75.92%	75.72%

Table 1: Training & Validation Accuracy for all initialization techniques. This was found by tallying the number of correctly classified examples in training & validation set over all 50,000 training and 10,000 validation examples.

4.3 Analysis of Results

When we compare the final training and validation accuracies of each model, it is instantly clear that initializing our weights from a sophisticated randomized distribution is much better than using uniform weights. The only models that achieve a training accuracy above 90% are our randomly initialized models, whereas our constant weight models never go past a 50% training or validation accuracy.

If we consider the training loss graph of our "All Zeros" initialized model, the training loss is much less smooth than our randomly initialized models. Specifically, there are noticeably more jagged edges at our first couple of training epochs. Furthermore, around 50 epochs, the total training loss tapers off at around 600, which is much higher than the training loss of our randomized models. From this we can derive that our model with constant ones weights is experiencing exploding gradients: the training loss is flattening off at a high value, causing our update gradients to be too large to cross over into the 50% accuracy threshold.

Unlike any of our training loss graphs, our “All Ones” model reaches a training loss extremely close to zero in the fewest epochs. Despite this though, it achieved the lowest training and validation accuracy of 10%. Since our training and validation sets are balanced to have an equal number of examples from each of the ten labels, this means this model is guessing the same label every time. Because there is minimal training loss however, there is no improvement from our update rule, so our constant zero model cannot improve. Therefore, this model is suffering from vanishing gradients: an extremely small training loss means there is a small update gradient that keeps the model from improving.

Both our Kaiming and Xavier initialized models have ideal training loss graphs that smooth over and level off at a low, but non-zero training loss. Because the Kaiming model got an almost perfect training accuracy score of 99.33% percent, it would be easy to conclude that Kaiming initialization works the best for our model. However, because the training accuracy is so high, its training loss in future epochs will be much smaller. We can already see this in the training loss graphs, where the Kaiming model is already beginning to taper off at a total loss close to 100 whereas the Xavier model has just reached 100 at 50 epochs. This means that unless we introduce more training data, gradient descent will not be able to improve the ability for the Kaiming initialized model in order to generalize to our validation set. Therefore, in this training scenario, Xavier Initialization would actually be the preferred weight initialization technique.

All of this goes to show how most of applied machine learning is still limited to trying and testing different techniques and parameters. There is still no definitive method to find the right combination of initialization method, hyperparameters, and model architectures for a dataset other than experimenting. Furthermore, with randomized initialization techniques, we always run the chance of sampling and receiving a “bad” set of starting weights that can’t improve training or validation accuracy. This though can be mitigated by starting over from scratch and training again.

5 Stochastic Gradient Descent

In Section 2 above 1, we presented an algorithm that found the best set of weights by finding a point in the parameter $\times$ loss space that minimizes the loss. Although almost all weight optimization algorithms stem from the update rule in gradient descent, they almost always deviate from gradient descent in one major thing: taking the total loss over **all** examples in the data set.

Since the cost of computing scales with respect to model size, in the era of large models and big data it is simply unfeasible to go through every single example to perform a single update step. So what if we relaxed this requirement and updated our weights according to randomly picked examples? This would give us **Stochastic Gradient Descent**.

Algorithm 2 Stochastic Gradient Descent [1]

```
1. Receive initial point $w_1$, max iterations $T \in \mathbb{N}$, and step sizes $\{a_k\}_{k=1}^T$
2. For $k = 1, ..., T$ do:
3.   Choose a random image $x_i \in \{x_i\}_1^n$ and the $k$-th step size $a_k$
4.   Update $w_{k+1} = w_k - \alpha_k \Delta \mathcal{L}(h(x; w_k), y_i)$
```

Just like Gradient Descent, SGD accepts an initial set of weights and minimizes the loss by descending down the parameter $\times$ loss convex bowl. Unlike traditional Gradient Descent, at each step SGD picks a completely random image from our training set to perform an update step on.

Because we update our parameters after every example, our weight loss between each iteration acts sporadically, moving up and down our bowl in a seemingly incoherent manner. However, as our parameters start to descend down the bowl, our weight starts to oscillates around the "optimal" set of parameters in our middle of our bowl before fully converging.

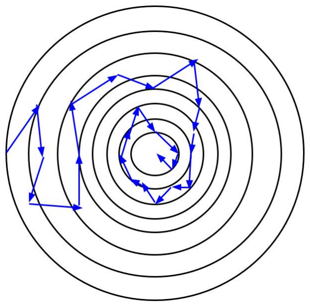

Figure 6: Unlike Gradient Descent, which at almost every step converges directly towards the optimal minimum, SGD will oscillate around the convex bowl in seemingly random directions. However, as it begins to descend down the bowl, it will eventually converge to the point of minimal loss.

6 Proving Stochastic Gradient Descent Works

We mention above that Stochastic Gradient Descent will converge to our "optimal" set of parameters, but how can we know for certainty that this will happen? In order to explain this, we need to further formalize our objectives and prove that our seemingly random process abides by certain trends. This will also give us leeway into our intuition of SGD as a stochastic process.

Definition 8 (Stationary Point)**.** *A point w^* is a stationary point for some function F if $\Delta F(w^*) = 0$.*

Definition 9 (Euclidean Norm)**.** *For $x = (x_1, ..., x_n)$ we define $||x||_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$.*

Definition 10 (Random Seed)**.** *Let $\mathcal{E}_i$ denote the random variable, or seed, which chooses our datapoint at the i-th step.*

We will work with a new, continuously differentiable function defined on random seeds. Denote $f = \mathcal{L} \circ h$ as the composition function of our loss and classifier such that $f_i(w) = f(w, \mathcal{E}_i) = \mathcal{L}(h(x_i; w), y_i)$ is the loss at the i-th step given the i-th randomly chosen image. We hope to minimize the expected loss at each step, which we denote as $F(w) = \mathbb{E}[f(w, \mathcal{E})]$.

As our number of iterations $T \to \infty$, we want to show that $\Delta F(w) \to 0$, that is, our expected loss will have found a stationary point. In the convex bowl, this can only be our minimum. In order to show that our change in expected loss converges, we will show that our losses and their derivatives are bounded through several theorems. To show this boundedness, we need our expected change in loss to have a strong continuous property, so we assume that it is Lipschitz continuous.

Definition 11 (Lipschitz Continuity)**.** *[15] Define $F : A \to \mathbb{R}$ where $A \subseteq \mathbb{R}^n$. If F is Lipschitz continuous, then for all $x, y \in A$, there exists a constant $L \in \mathbb{R}$ such that $||F(x) - F(y)||_2 \leq L \cdot ||x - y||_2$.*

Since the gradient of the expected loss $\Delta F(w)$ is its own differentiably continuous function, we assume it's Lipschitz continuous. Our first theorem shows that if this is true, we can derive a useful inequality about the bounds for our expected loss with relation to any two weights.

Theorem 1 (Bound of Loss)**.** *[13]*

Suppose $F : \mathbb{R}^{d_x} \to \mathbb{R}$ is continuously differentiable and $\Delta F : \mathbb{R}^{d_x} \to \mathbb{R}$ is Lipschitz continuous with Lipschitz constant L such that $||\Delta F(w) - \Delta F(v)||_2 \leq L \cdot ||w - v||_2$ for all $w, v \in \mathbb{R}^{d_x}$.

Then $F : \mathbb{R}^{d_x} \to \mathbb{R}$ satisfies $F(w) \leq F(v) + \Delta F(v)^T (w - v) + \frac{1}{2} L ||w - v||_2^2$ for all $w, v \in \mathbb{R}^{d_x}$ (where $F(v)^T$ is a transpose matrix).

Proof.

- We can define $F(w) - F(v)$ as an integral from 0 to 1 over dt of the derivative of F on the line between w and v, so $F(w) - F(v) = \int_0^1 \frac{\partial F(v + t(w-v))}{\partial t} dt$. Rearranging these terms gives us $F(w) = F(v) + \int_0^1 \frac{\partial F(v + t(w-v))}{\partial t} dt$.
- By Chain Rule for functions of several variables, if we differentiate $\frac{\partial F(v + t(w-v))}{\partial t}$ inside our integral, we get $F(w) = F(v) + \int_0^1 \Delta F(v + t(w-v))^T (w-v) dt$.
- Consider the term $\Delta F(v)^T \cdot (w - v)$. This is an inner-product, so we can add this term alongside its negation as follows: $F(w) = F(v) + [\Delta F(v)^T \cdot (w-v) - \Delta F(v)^T \cdot (w-v)] + \int_0^1 \Delta F(v + t(w-v))^T (w-v) dt = F(v) + \Delta F(v)^T \cdot (w-v) + \int_0^1 [\Delta F(v + t(w-v))^T - \Delta F(v)^T](w-v) dt = F(v) + \Delta F(v)^T \cdot (w-v) + \int_0^1 [\Delta F(v + t(w-v)) - \Delta F(v)]^T (w-v) dt$.
- We apply a Cauchy-Schwartz inequality $|\langle a, b \rangle| = a^T b \leq ||a||_2 \cdot ||b||_2$. If we apply it to our dot-product inside our integral, we get that $F(w) \leq F(v) + \Delta F(v)^T (w-v) + \int_0^1 ||\Delta F(v + t(w-v)) - \Delta F(v)||_2 \cdot ||w - v||_2 dt$.
- We can apply the definition of Lipschitz continuity on our norm $||\Delta F(v + t(w-v)) - \Delta F(v)||_2$, which gives us $F(w) \leq F(v) + \Delta F(v)^T (w-v) + \int_0^1 L||t(w-v)||_2 \cdot ||w-v||_2 dt$.
- Taking the t out of our norm in our integral will give us our final expression $F(w) \leq F(v) + \Delta F(v)^T (w-v) + \frac{1}{2} L||w-v||_2^2$.

□

We will offer an intuitive explanation: by the definition of Lipschitz continuity, the gradients for the expected change in loss for two weights w, v is bounded by a constant L, formally written as $||\Delta F(w) - \Delta F(v)||_2 \leq L \cdot ||w - v||_2$. When re-arranged, our inequality above gives us $F(w) - F(v) \leq \Delta F(v)^T (w-v) + \frac{1}{2} L||w-v||_2^2$. Thus, our theorem tells us that if the expected change in the loss for two weights is Lipschitz continuous, then their losses are bounded by the same L constant. This is why we refer to this theorem as "Bound of Loss".

Although our theorem 1 above gives us a clear notion of how the bounds of loss and change of loss are related, we need to be most specific and define our change of loss in terms of what random example we used to find our loss on, as found by our "seed".

Definition 12 (Expectation of Seed)**.** *Let $\mathbb{E}_{\mathcal{E}_k}[X]$ be the expected value taken with respect to the distribution of the random "seed" variable $\mathcal{E}_k$ that samples our k-th example, given some variable X.*

We consider the expected change of loss $F(w_{k+1})$ at our $k+1$-th step given our seed $\mathcal{E}_k$. We want to bound this term $\mathbb{E}_{\mathcal{E}_k}[F(w_{k+1})]$ by some quantity defined by our *previous* change of loss, our loss for our previous weight w_k, and our step size α_k to affirm that no matter what example our seed chooses, our weights will change *no greater* than our decreasing quantities α_k and $\Delta f(w_k, \mathcal{E}_k)$.

Theorem 2 (Bound of Expectation)**.** *[13] Assume $F : \mathbb{R}^{d_x} \to \mathbb{R}$ is continuously differentiable and $\Delta F : \mathbb{R}^{d_x} \to \mathbb{R}$ is Lipschitz continuous with Lipschitz constant L. Then $\mathbb{E}[F(w_{k+1})] - F(w_k) \leq -\alpha_k \Delta F(w_k)^T \mathbb{E}[\Delta f(w_k, \mathcal{E}_k)] + \frac{1}{2}\alpha_k^2 \cdot L \cdot \mathbb{E}_{\mathcal{E}_k}[||\Delta f(w_k, \mathcal{E}_k)||_2^2]$ for all $k = 1, ..., T$.*

Proof.

- Our assumptions for this theorem match those of our previous theorem 1, so for two points $w, v \in \mathbb{R}^{d_x}$, we have $F(w) \leq F(v) + \Delta F(v)^T(w - v) + \frac{1}{2}L||w - v||_2^2$.
- If we replace w and v with two consecutive weights w_{k+1} and w_k respectively, we get $F(w_{k+1}) - F(w_k) \leq \Delta F(w_k)^T(w_{k+1} - w_k) + \frac{L}{2}||w_{k+1} - w_k||_2^2$.
- Recall our stochastic gradient descent algorithm 2, where our update rule sets $w_{k+1} = w_l - \alpha_k \Delta h(x_k; w_k)$ where $\Delta h(x_k; w_k) = \Delta f(w_k; \mathcal{E}_k)$.
- If we replace all qualities $w_{k+1} - w_k$ in our inequality with $-\alpha_k \Delta f(w_k; \mathcal{E}_k)$, we get $F(w_{k+1}) - F(w_k) \leq -\alpha_k \Delta F(w_k)^T \cdot \Delta f(w_k; \mathcal{E}_k) + \alpha_k \cdot \frac{L}{2} \cdot ||\Delta f(w_k; \mathcal{E}_k)||_2^2$.
- Now we will take the expectation of each term in our equality with respect to $\mathcal{E}_k$. Recall that we use $\mathcal{E}_k$ to randomly choose a point **at** the k-th step, but we did not use it to determine our quantities at the k-th step. This means our expected change of loss $F(w_k)$ is independent from $\mathcal{E}_k$, but our next weights w_{k+1} were chosen using $\mathcal{E}_k$, so $F(w_{k+1})$ and the change of loss after choosing our example $\Delta f(w_k, \mathcal{E}_k)$ are dependent on $\mathcal{E}_k$.
- Thus, when we take expectations with respect to $\mathcal{E}_k$, we get $\mathbb{E}_{\mathcal{E}_k}[F(w_{k+1})] - F(w_k) \leq -\alpha_k \Delta F(w_k)^T \mathbb{E}[\Delta f(w_k, \mathcal{E}_k)] + \frac{1}{2} \cdot \alpha_k^2 L \cdot \mathbb{E}_{\mathcal{E}_k}[||\Delta f(w_k, \mathcal{E}_k)||_2^2]$.

□

Although we try to be as precise as possible by taking expectations and gradients with respect to our random seed $\mathcal{E}_k$, there will always be some "noise" from our random process. This means if we want to create more precise boundaries for our weights between each iteration of SGD, we need to include a general metric for our "noise". In our next theorem, we will bound our expected change of loss *after* choosing our k-th example *with respect* to our k-th seed $\mathcal{E}_k$, formally written as $\mathbb{E}_{\mathcal{E}_k}[||\Delta f(w_k, \mathcal{E}_k)||_2^2]$, by constants M and M_G that represent the "noise" of our gradients that are independent from our process.

Theorem 3 (Bound of Iteration)**.** *[14] Suppose there exist constants $M \geq 0$ and $M_G \geq 1$ such that $\mathbb{E}_{\mathcal{E}_k}[||\Delta f(w_k, \mathcal{E}_k)||_2^2] \leq M + M_G||\Delta F(w_k)||_2^2$. Furthermore, assume $F : \mathbb{R}^{d_x} \to \mathbb{R}$ is continuously differentiable and $\Delta F : \mathbb{R}^{d_x} \to \mathbb{R}$ is Lipschitz continuous.*

Then the iterations of SGD satisfy $\mathbb{E}_{\mathcal{E}_k}[F(w_{k+1})] - F(w_k) \leq -(1 - \frac{1}{2}\alpha_k L M_G)\alpha_k$ $||\Delta F(w_k)||_2^2 + \frac{1}{2}\alpha_k LM$ for all $k = 1, ..., T$.

Proof.

- Our suppositions match those of our previous theorem 2. Therefore, by this theorem, we get $\mathbb{E}_{\mathcal{E}_k}[F(w_{k+1})] - F(w_k) \leq -\alpha_k \Delta F(w_k)^T \mathbb{E}_{\mathcal{E}_k}[\Delta f(w_k, \mathcal{E}_k)] + \frac{1}{2}\alpha_k^2 \cdot L \cdot \mathbb{E}_{\mathcal{E}_k}[||\Delta f(w_k, \mathcal{E}_k)||_2^2]$.

- At our k-th seed $\mathcal{E}_k$, our expected gradient or change of loss $\mathbb{E}_{\mathcal{E}_k}[\Delta f(w_k, \mathcal{E}_k)]$ at a single datapoint x_i is just $\Delta F(w_k)$, so replacing this term gives us with $F(w_k)$ gives us $\mathbb{E}_{\mathcal{E}_k}[F(w_{k+1})] - F(w_k) \leq -\alpha_k \Delta F(w_k)^T F(w_k) + \frac{1}{2}\alpha_k^2 \cdot L \cdot F(w_k)$

- Since $F(w_k)^T \cdot F(w_k)$ is a dot product of the same term, it is equal to the squared Euclidean norm of $F(w_k)$, so $\mathbb{E}_{\mathcal{E}_k}[F(w_{k+1})] - F(w_k) \leq -\alpha_k||F(w_k)||_2^2 + \frac{1}{2}\alpha_k^2 \cdot L \cdot \mathbb{E}_{\mathcal{E}_k}[||\Delta f(w_k, \mathcal{E}_k)||_2^2]$.

- Plugging in our bound noise assumption for $\mathbb{E}_{\mathcal{E}_k}[\Delta f(w_k, \mathcal{E}_k)]$ gives us $\mathbb{E}_{\mathcal{E}_k}[F(w_{k+1})] - F(w_k) \leq -\alpha_k||F(w_k)||_2^2 + \frac{1}{2}\alpha_k^2 L(M + M_G||\Delta F(w_k)||_2^2)$. Simplifying this gives us our final statement $\mathbb{E}_{\mathcal{E}_k}[F(w_{k+1})] - F(w_k) \leq -(1 - \frac{1}{2}\alpha_k L M_G)\alpha_k||\Delta F(w_k)||_2^2 + \frac{1}{2}\alpha_k LM$.

□

We have established solid ground for how our weights are bounded according to the gradients of our loss function, step sizes, and even independent noise. With this we can explore how SGD converges. Recall that we want to find a stationary point where $\Delta F(w_k) = 0$ after some number of iterations k. Our next algorithm shows that when we have a converging sequence of diminishing step sizes, so $\sum_{i=1}^{\infty} a_k = \infty$ but $\sum_{i=1}^{\infty} a_k^2 < \infty$, as $T \to \infty$, $\Delta F(w_k) \to 0$.

Theorem 4 (Convergence of Gradients)**.** *[14] Suppose there exist constants $M \geq 0$ and $M_G \geq 1$ such that $\mathbb{E}_{\mathbb{E}_k}[||\Delta f(w_k, \mathcal{E}_k)||_2^2] \leq M + M_G||\Delta F(w_k)||_2^2$. Furthermore, assume $F : \mathbb{R}^{d_x} \to \mathbb{R}$ is continuously differentiable and $\Delta F : \mathbb{R}^{d_x} \to \mathbb{R}$ is Lipschitz continuous.*

Suppose we have a diminishing step size sequence such that $\sum_{i=1}^{\infty} a_k = \infty$ but $\sum_{i=1}^{\infty} a_k^2 < \infty$. Define $A_T = \sum_{i=1}^{T} a_k$. Then $\lim_{T\to\infty} \mathbb{E}\left[\sum_{k=1}^{T} a_k||\Delta F(w_k)||_2^2\right] < \infty$ and therefore $\mathbb{E}\left[\frac{1}{A_T}\sum_{k=1}^{T} a_k \cdot ||\Delta F(w_k)||_2^2\right] \to 0$ as $T \to \infty$. [14]

Proof.

- The assumptions we make for this theorem match those in our previous theorem 3. Therefore, we know that $\mathbb{E}_{\mathcal{E}_k}[F(w_{k+1})] - F(w_k) \leq -(1 - \frac{1}{2}\alpha_k LM_G)\alpha_k ||\Delta F(w_k)||_2^2 + \frac{1}{2}\alpha_k LM$.

- We know that $k \to \infty$, $\alpha_k \to 0$, so without loss of generality for our k we can assume that $a_k \leq \frac{1}{LM_G}$ (since α_k becomes arbitrarily small). This inequality allows us to simplify our term $-(1 - \frac{1}{2}\alpha_k LM_G) \leq \frac{-1}{2}$, so $\mathbb{E}_{\mathcal{E}_k}[F(w_{k+1})] - F(w_k) \leq -\frac{1}{2}\alpha_k ||\Delta F(w_k)||_2^2 + \frac{1}{2}\alpha_k LM$.

- Denote $F_{\text{inf}} = F(w_T)$ as a constant where $T \to \infty$. When we sum over every inequality above for each iteration $k = 1, ..., T$, we get $F_{\text{inf}} - F(w_1) \leq \mathbb{E}[F(w_T)] - F(w_1) \leq -\frac{1}{2}\sum_{k=1}^{T} a_k \mathbb{E}[||\Delta F(w_k)||_2^2] + \frac{1}{2}LM\sum_{k=1}^{T} a_k^2$.

- We can simplify this inequality by multiplying both sides of our inequality by 2, moving our sum inside our expectation, and moving our expectation to the left-hand side. This will give us $\mathbb{E}\left[\sum_{k=1}^{T} ||\Delta F(w_k)||_2^2\right] \leq 2(F(w_1) - F_\infty) + LM\sum_{k=1}^{T} \alpha_k^2$.

- Dividing both sides by $\frac{1}{A_T}$ gives us our total equation $\frac{\mathbb{E}\left[\sum_{k=1}^{T} ||\Delta F(w_k)||_2^2\right]}{A_T} \leq \frac{2(F(w_1) - F_\infty) + LM\sum_{k=1}^{T} \alpha_k^2}{A_T}$.

- Since as $T \to \infty$, $A_T \to \infty$, we can conclude that $\frac{\mathbb{E}\left[\sum_{k=1}^{T} ||\Delta F(w_k)||_2^2\right]}{A_T}$ $\leq \frac{2(F(w_1) - F_\infty) + LM\sum_{k=1}^{T} \alpha_k^2}{A_T} \to 0$ (since $F(w_1) - F_\infty$, L, M are constants, and $\sum_{k=1}^{\infty} \alpha_k^2 < \infty$, our numerator term is bounded by ∞).

$\square$

Our theorem shows us that as $T \to \infty$, the squared norm of the change of expected loss $||\Delta F(w_k)||_2^2$ will converge to 0. This means our norm for the change of expected loss becomes so small that even though our step sizes α_k also approaches 0, the ratio of loss to step size also approaches 0, so the change to weight made at our k-th step becomes so minuscule it might as well have left our previous weights untouched. We have shown that with SGD, our change to weight will reach 0, which can only happen at a stationary point. Therefore, we can conclude that we are bound to find our optimal minimum as long as our number of iterations is large enough.

7 Interpretting SGD as a Stochastic Process

We have proven that SGD finds a stationary minimum when we train using independent, identically distributed random variables which we call "seeds" for picking our next data point. This allows us to formalize Stochastic Gradient Descent as a stochastic process defined on a clear state space.

Our state space is clearly the weight parameters $\times$ loss real space. From here, we can define a process $(X_t)_{t\geq 0}$ such that each X_t is a random variable that represents our weights at time t, where our initial weight X_0 is a randomly chosen set of starting parameters and we use our update rule to find $X_{t+1} = X_t - \alpha_t \Delta h(x; X_t)$ using the t-th step-size α_t and a randomly chosen image x.

We are interested only in the long term behavior of our process, and we have painstakingly proven that as $t \to \infty$, $\Delta h(x; X_i) \to 0$ for any image x. Since our state space is a subset of $\mathbb{R}^d \times \mathbb{R}^+$, our long-term transition matrix $\lim_{n\to\infty} \mathcal{P}$ will be entirely empty in the row where our optimal minimum $((a, b, c, ...), l_{\text{Final}}) \in \mathbb{R}^d \times \mathbb{R}^+$ is located in, so $\lim_{n\to\infty} p^n_{((a,b,c,...),)l_{\text{Final}}),((a,b,c,...),l_{\text{Final}})} =$ $P_{\text{Start from optimal minimum}}(\text{Return to optimal minimum starting in 1 step}) = 1$ since by our update rule, we won't be able to change our weight once we reach minimum, which is a the stationary point.

Every row in our transition matrix has to add up to 1. To explain this, we need some more intuition: imagine that the closer a point on a parabola is to the vertex, the smaller the gradient of its tangent line will be. This means if our points x on a parabola could take a step of the size of its derivative to another point on the parabola, the closer it is to the vertex, the more likely it will stay relatively close to the vertex. The same intuition holds for our long-term transition matrix: the closer the entries are to our optimal minimum row-wise, the less spread our their probabilities will be, so our non-zero entries will be concentrated more closer together while being centered around the optimal minimum.

We can split apart our state space into several classes. The most obvious example is our optimal minimum: once we reach our stationary point, our weights cannot change in future iterations, so we are bound to return to the same set of parameters for all future steps. This means our optimal minimum represents a positive-recurrent class with only one entry. Assuming there are no other stationary points, because we have proved that we expect our process to abandon every state that is not our optimal minimum as our number of iterations $T \to \infty$, the rest of our state space is a transient class.

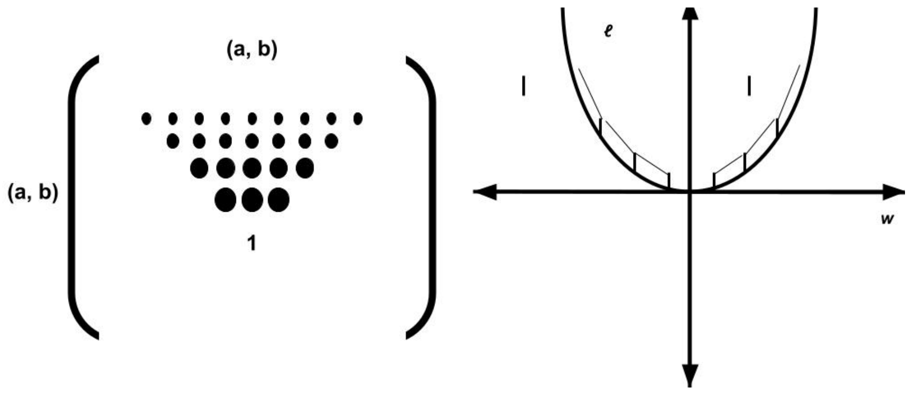

Figure 7: Consider a linear classifier $h(x) = ax + b$ in $\mathbb{R}^2$, and suppose we find some optimal weight pair (a, b) that minimizes the loss. Then the only entry that is 1 in our long-term transition matrix will be our optimal minimum. Furthermore, the closer our probabilities are to our optimal minimum, the larger and more concentrated they will be. This looks similar to how the gradients for the tangent line of points in a parabola are smaller when those points are closer to our vertex. Therefore, once we descend down our slope, it's much less probable that our current weight can climb back up since it can't take larger steps.

8 SGD in Large Model Training

We have proven that SGD will converge to an optimal minimal point for loss in a parameter space. It remains to show that this works in an experimental setting as well. However, unlike our previous experiment, our objective won't be to show that SGD outperforms other optimization techniques like normal Gradient Descent or ADAM.

Rather, because SGD updates model weights after each example, SGD may converge to a minimal training loss in less epochs than other optimization techniques. This is more pronounced in larger models, where computations on inputs and finding derivatives (via back-propagation) are extremely computationally expensive.

To test the performance of stochastic gradient descent in complicated parameter spaces, we will train a Carlini network as well as a substantially larger model: the Residual Network. The Residual Network, or ResNet for short, uses forward connections that send calculations in previous layers to later layers [6]. This makes the ResNet a great candidate for our higher complexity large image recognition classifier since its architecture helps preserve information in the feature space which helps to mitigate vanishing and exploding gradients.

It is important to note that the regular SGD algorithm we introduced above 2 does not work well with convolutional neural networks. Therefore, in order to mimic the randomized weight update of SGD in our optimization technique, we will use an Adam optimizer but vary the batch size. The batch size corresponds to how many examples we take the loss over before we perform an update rule. A low batch size would mean we are updating our weights based on our classifiers' performance for few examples, which would give us the desired stochastic behavior we want in our weight update.

8.1 Experiment & Data

Like our last experiment, we will train both the Carlini and ResNet56 on CIFAR-10 using a learning rate of 0.0001 over 50 epochs. We use Cross Entropy loss and an Adam optimizer. The only thing we vary in our experiment is the batch size. We will train two separate Carlini networks and ResNet56s using a batch size of 10 and 300. Since there are $50,000$ training examples in CIFAR-10, using a batch size of 10 means there will be $5,000$ updates per epoch and a batch size of 300 means there are 167 updates. We will plot the training loss with respect to epochs and evaluate all of our models on the training and validation set after training.

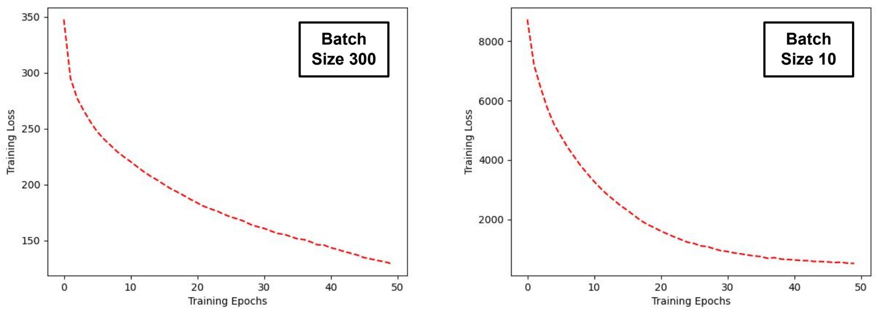

Figure 8: Resulting training loss over epochs for **Carlini** network when we vary the batch size.

If we look at the training loss graphs for both our Carlini and ResNet56, we can notice that the graphs produced from using a batch size of 10 are more convex and smooth to the point where they start to level off.

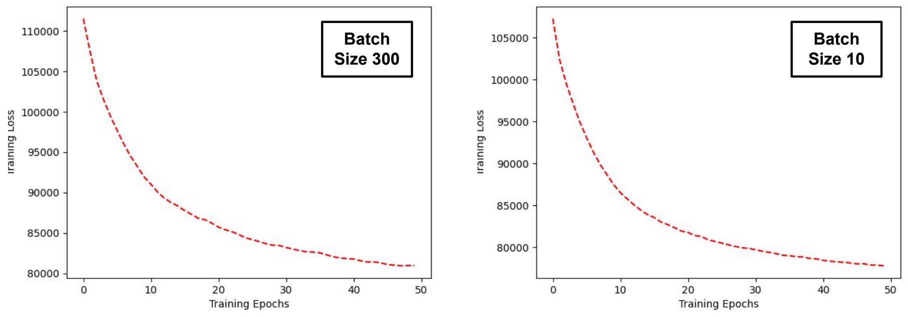

Figure 9: Resulting training loss over epochs for **ResNet56** when we vary the batch size.

	Carlini (BS = 300)	Carlini (BS = 10)	ResNet56 (BS = 300)	ResNet56 (BS = 10)
Training Accuracy	76.87%	99.40%	73.94%	84.50%
Validation Accuracy	71.23%	78.71%	47.32%	69.93%

Table 2: Training & Validation Accuracy for Carlini and ResNet56 models trained with different batch sizes (BS).

For Carlini, although the training loss for using a batch size 10 was substantially higher than using a batch size of 300, the training loss graph for the Carlini with batch size of 10 reached a point where the loss flattened our in less epochs. We want to reach this point where our training loss tampers off in less epochs since this indicates that we are converging to our minimal training loss. Thus, the batch size 10 model has the more ideal behavior.

For the ResNet56, although these graphs are extremely similar, the ResNet56 with a batch size of 300 is slightly jagged around the middle and end of the curve, whereas the ResNet56 with batch size 10 produced a slightly more smooth graph. We want to minimize any instances where our training loss might increase, so using a batch size of 10 is also more ideal when we only compare the training loss graphs.

If we consider the training and validation accuracies, for both the Carlini and ResNet56, using a batch size of 10 produced better accuracies across the board. The most noticeable difference is that we see a 20% increase in training accuracy for the Carlini and a 20% increase in validation accuracy for the ResNet56. This tells us that because using a smaller batch size creates a stochastic behavior for model weight update, it may help weights in smaller models converge sooner to an ideal minimal training loss as well as help weights in larger parameter spaces for larger models generalize more.

9 Conclusion

We have introduced an extremely formal, yet vague enough notion of machine learning classifiers that has allowed us to deconstruct classic problems and algorithms in machine learning such as vanishing & exploding gradients and stochastic gradient descent. The concensus from the theory we built was clear: using properly initialized weights and using a stochastic process to train those weights produces better models with minimal loss. The experimental sections helped cement this intuition further, showing that introducing randomized weight initialization and taking more sporadic weight updates produced higher raw accuracies and ideal training loss graphs.

We live in the age of smart machines, and as the cutting-edge models of now become the norm of tomorrow, the field will have to innovate. However, this ability to innovate is already becoming more limited: you can only train benchmark models today if you have access to high-performance GPUs. The models trained for the experimental sections presented here were completed in days on a cloud server.

This all ties into why randomization is key. Large machine learning model training has become one of the most demanding computational tasks of today. Any incentive to reduce training time while achieving high classifier confidence *will* be taken. Thus, we are bound to see randomization embraced in every possible machine learning practice. If there is a will, there is a way - or should I say, a weight.

9.1 Repository

All code can be found here:

References

[1] *An overview of gradient descent optimization algorithms.* https://arxiv.org/pdf/1609.04747.pdf.

[2] Handwritten digit recognition with a back-propagation network. https://proceedings.neurips.cc/paper_files/paper/1989/file/53c3bce66e43be4f209556518c2fcb54-Paper.pdf.

[3] Stochastic gradient descent: An intuitive proof. https://medium.com/oberman-lab/proof-for-stochastic-gradient-descent-335bdc8693d0.

[4] Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. https://proceedings.neurips.cc/paper_files/paper/1989/file/0336dcbab05b9d5ad24f4333c7658a0e-Paper.pdf.

[5] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. https://arxiv.org/abs/1412.6980.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *2015 IEEE International Conference on Computer Vision (ICCV)*, page 1026–1034, 2016. https://ieeexplore.ieee.org/document/7410480.

[8] Prakash Jay. Back-propagation is very simple. who made it complicated?, Apr 2018. https://medium.com/@14prakash/back-propagation-is-very-simple-who-made-it-complicated-97b794c97e5c.

[9] Kian Katanforoosh and Daniel Kunin. Ai notes: Initializing neural networks, 2019. https://www.deeplearning.ai/ai-notes/initialization/index.html#:~:text=Initializing%20all%20the%20weights%20with,weights%20with%20some%20constant%20%CE%B1.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. https://arxiv.org/abs/1412.6980.

[11] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*, Apr 2009. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[12] Kaleel Mahmood, Phuong Ha Nguyen, Lam M Nguyen, and Marten Van Dijk. Besting the black-box: Barrier zones for adversarial example defense. *IEEE*, 10:1451–1474, Dec 2021. `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&amp;arnumber=9663375`.

[13] Özgür Martin. Introduction to stochastic optimization for large-scale machine learning part 1. `https://www.youtube.com/watch?v=Q_DahzeZhLs`.

[14] Özgür Martin. Introduction to stochastic optimization for large-scale machine learning part 2. `https://www.youtube.com/watch?v=fBeOV_xrmaI`.

[15] Andrew McCrady. Lipschitz functions: Intro and simple explanation for usefulness in machine learning, Jul 2021. `https://www.youtube.com/watch?v=UjvFFXakMks`.

[16] Zhilu Zhang and Mert R. Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels. *CoRR*, abs/1805.07836, 2018. `http://arxiv.org/abs/1805.07836`.