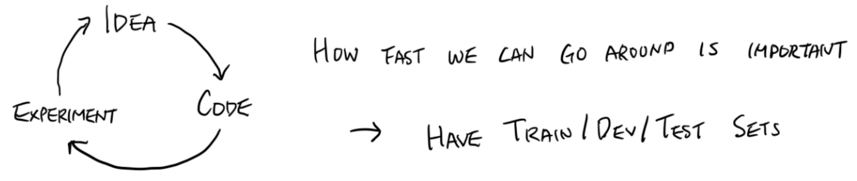
Optimization of Deep Neural Networks

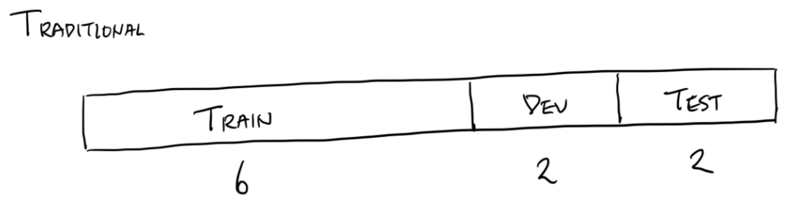
Swapnoneel Kayal

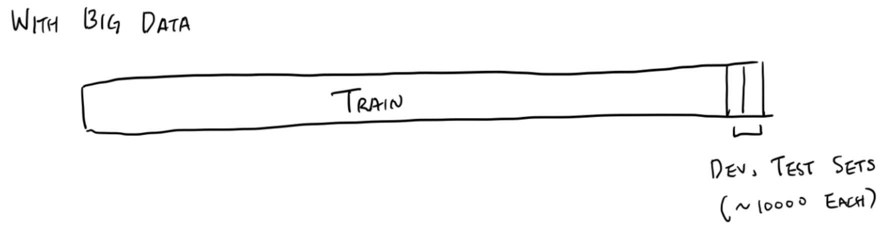
8 June - 14 June 2021

1 Train/Dev/Test Sets

Deep learning is a very iterative process looking for the right set of **hyper-parameters**. It is important to repeat the process of having an idea, coding, and experimenting. To do so, we need to set up dataset properly into **train**, **dev**, and **test** sets. **Train set** is used to train the model. **dev** and **test sets** are data that the model had never seen before and are used to analyze the model. Traditionally, we had the golden ratio of 6:2:2 = Train:Dev:Test. However with **big data**, that distribution will allocate too much data for **dev** and **test sets**. Considering that **deep learning** is very data hungry, it is wiser to use most of the data for **training** and just have around 10,000 samples each for **dev** and **test sets**. Also, it is okay to just have the **dev set**. If we have both **dev** and **test sets**, it is extremely important that they come from the same distribution. For example, one cannot be high-def images while the other is low-def images.

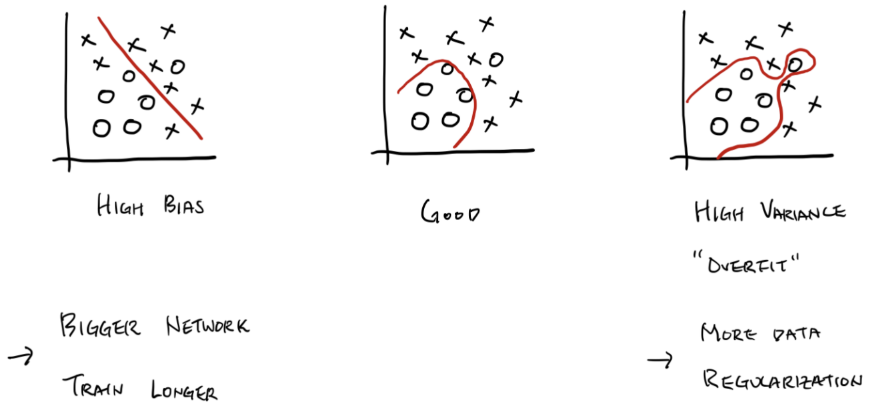

IDEA
CODE
EXPERIMENT
HOW FAST WE CAN GO AROUND IS IMPORTANT
→ HAVE TRAIN / DEV / TEST SETS





2 Bias/Variance

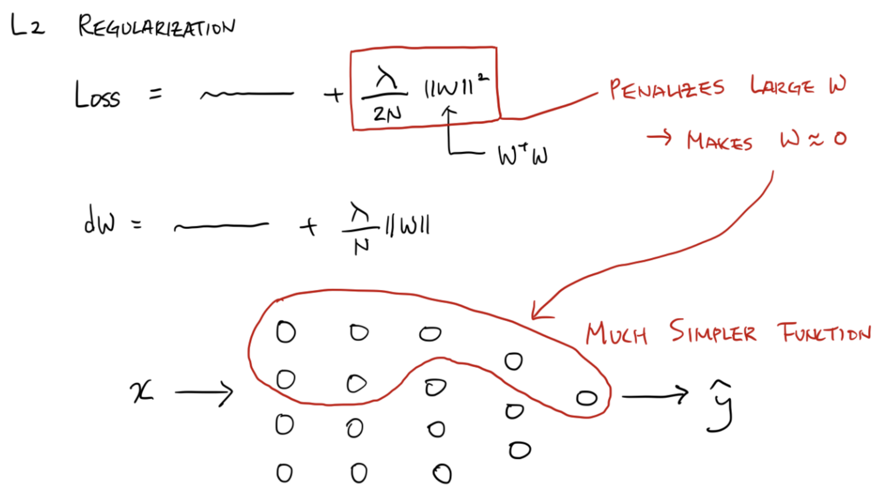
The reason we need separate **train** and **dev** set is to analyze the model's **bias** and **variance**. Model with **high bias** will be less flexible and fail to have enough complexity to classify properly. On the other hand, model with **high variance** will be too flexible and **overfit** to the **training data**, being specialized with the **train set** and not good with any data that it has not seen before. Therefore, when **dev error** is high while **train error** is low, we fix the **variance** problem and if the **train error** is high and **dev error** is about same, we fix the **bias** problem. If we have a higher **dev error** while **train error** is already high, we have both **bias** and **variance** problems to solve. Eventually, we want both **train error** and **dev error** to be low with very low gap in between them — low **bias** and low **variance**. When we have **high bias**, we need to make the network more flexible/complex or simply make it learn more. So for **bias** problem, we can build a bigger network, which almost never hurts, or train for longer. With **high variance**, we want to prevent **overfitting** by introducing more **training data** or **regularization** to the model.



TRN ERROR	1%	15%	15%	0.5%
DEV ERROR	15%	15%	30%	1%
	LOW BIAS HIGH VAR	HIGH B LOW V	HIGH B HIGH V	LOW B LOW V

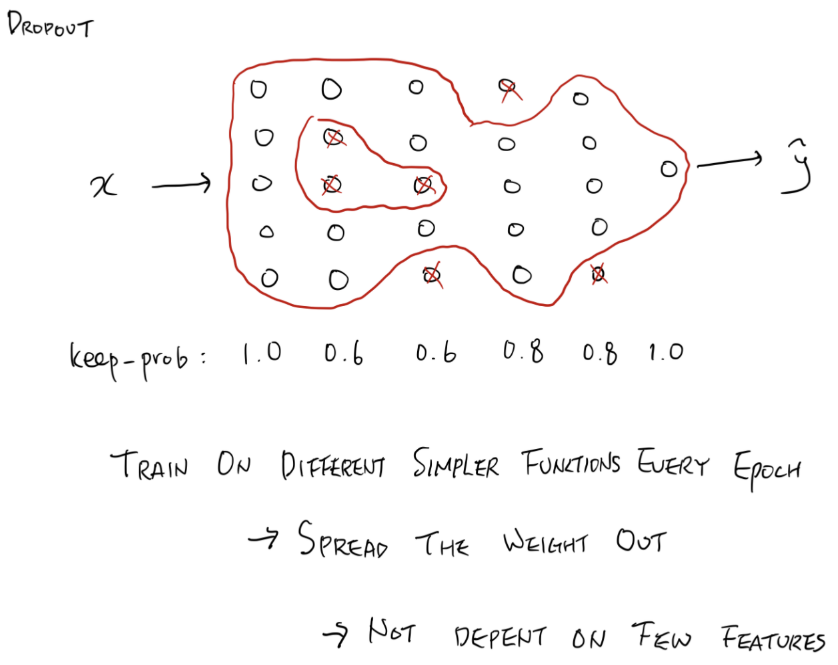
3 Regularization

There are several ways to do **regularization**. One of them, **weight decay** adds the sum of magnitudes of **weights** and **biases** to the **loss function**, so that it penalizes large **weights** (both negative and positive). **Weights** are more likely to be closer to 0 because our goal is to minimize the **loss function**. This allows us to learn much simpler mapping function. If you square the sum, it becomes the **L2 regularization** and if you don't, it becomes **L1 regularization**. $\lambda/2N$ is a term multiplied before the sum and λ is a new **hyper-parameter**. It is okay to omit **biases** because they are negligible.



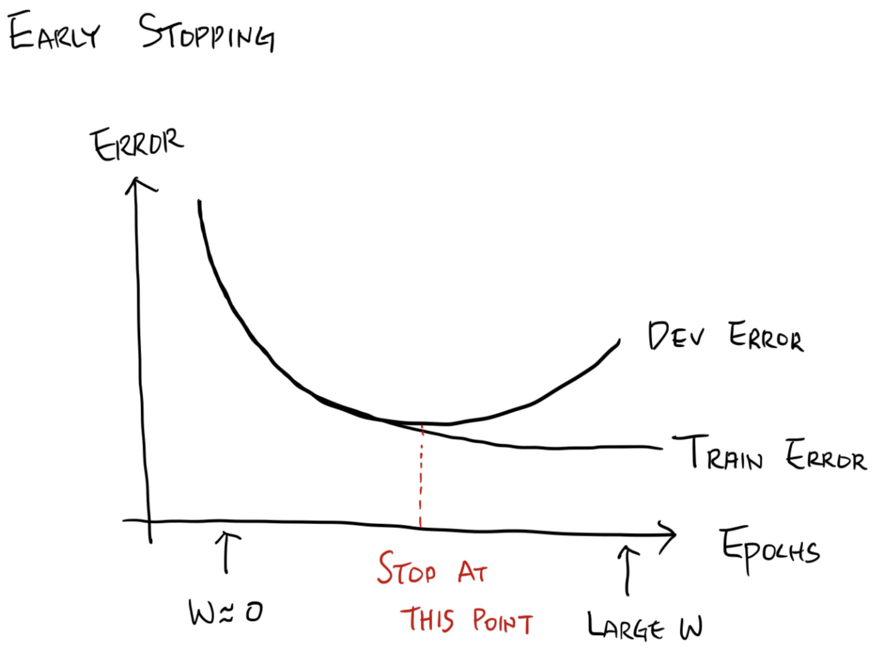
4 Dropout Regularization

Another **regularization** method is **dropout**. With certain chance, we shut some **neurons** off forcing the network to learn much simpler function. It is important to **dropout** different **neurons** every iteration. We might want to use different **dropout** chances for each layer because **earlier layers** have higher chance of overfitting compared to the **later layers**. So we would normally **dropout** more **neurons** in **earlier layers** by using lower **keep_prob**, which tells us how much **neurons** to keep. We usually do not use **dropout** for the **input layer** and definitely not during **test time**. The intuition behind **dropout** is in trying to learn simpler mapping function but also to spread out the **weights** so that we do not specialize in few features. One downside of **dropout** is that it kind of makes the **loss function** less well defined.


Dropout
x
ŷ
keep-prob : 1.0 0.6 0.6 0.8 0.8 1.0
Train On Different Simpler Functions Every Epoch
→ Spread The Weight Out
→ Not depent on Few Features

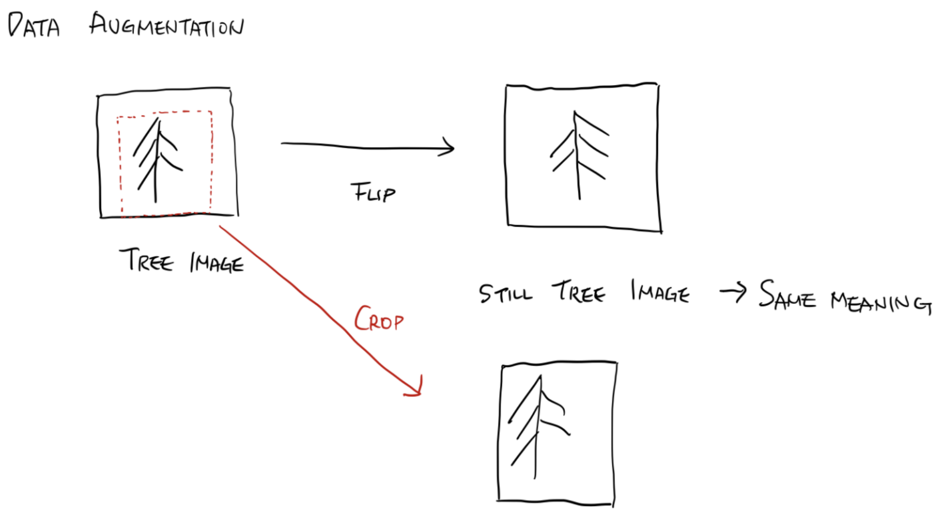
5 Early Stopping

Lastly, **early stopping** is something worth trying. If **weights** are initialized to small random numbers, they will start to grow too big and **overfit** as we train for longer time. So, we monitor both **train** and **dev error** during **train time** and end **training** when things seem to be just right. However, we mentioned that training for longer period affects **bias**, which means that shortening training time will affect bias as well. Therefore, **early stopping** is not an ideal option for **orthogonalization** which means to deal with **bias** and **variance** separately.



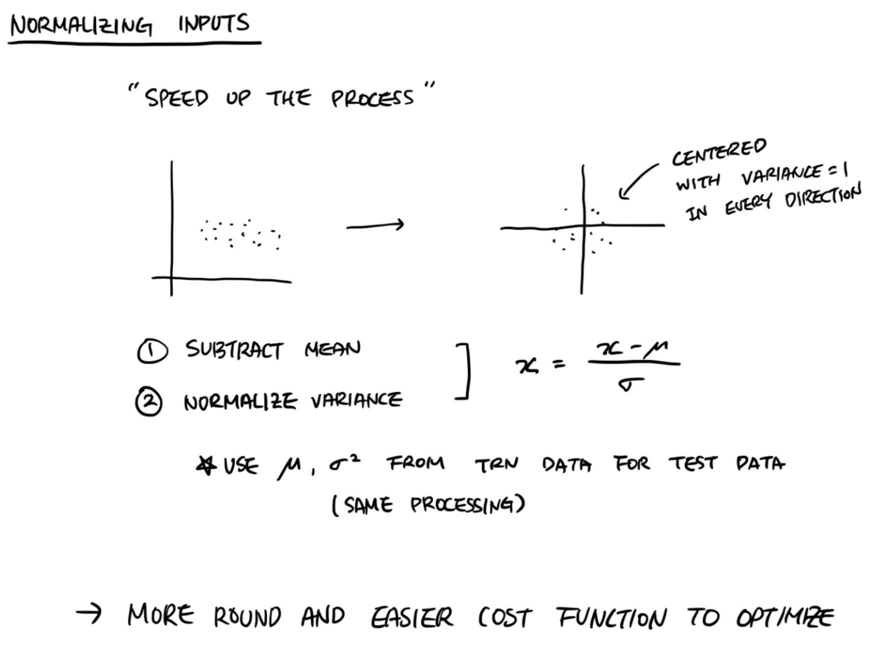
6 Data Augmentation

Coming up with more data could be very difficult and expensive. **Data augmentation** is a technique where we create new data by **flipping**, **random cropping**, and etc. to modify the original data just slightly enough to not change the meaning of it.



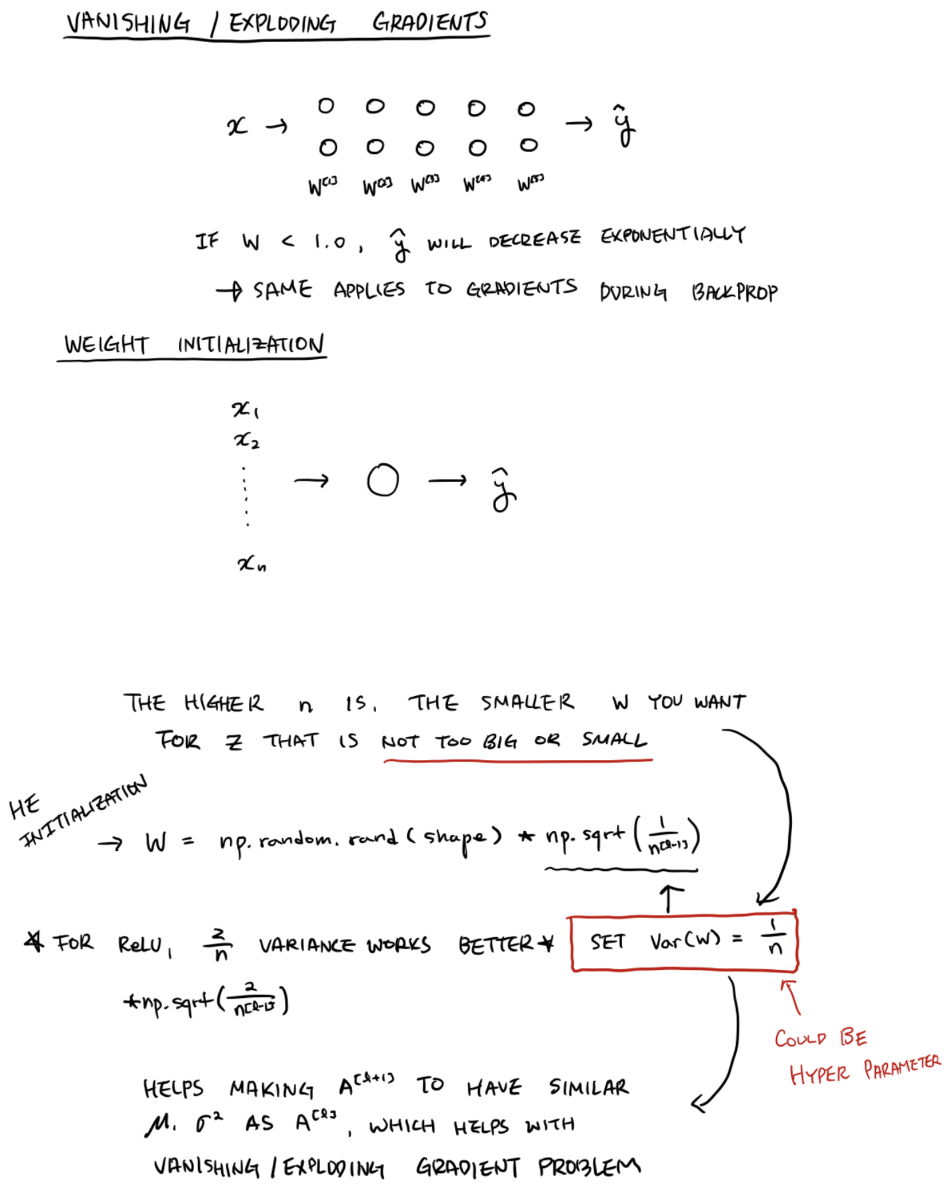
7 Normalizing Inputs

Normalizing inputs is always a good idea because it speeds up the process of learning. **Normalization** centers the inputs to (0, 0) with the **variance** of **1** in every direction. To achieve this, we first subtract the **mean (µ)** of **X** from **X** then divide it by the square root of its **variance** (σ^2). This allows to have a more round and easier **cost function** to optimize with. **Normalization** becomes more important when input features are in various scales. For example, with x_1 ranging from **1.0** to **-1.0** and x_2 ranging from **100** to **-100**, we really want to **normalize**.


NORMALIZING INPUTS
"SPEED UP THE PROCESS"
CENTERED WITH VARIANCE = 1 IN EVERY DIRECTION
① SUBTRACT MEAN
② NORMALIZE VARIANCE
$x = \frac{x - \mu}{\sigma}$
★ USE μ, σ^2 FROM TRN DATA FOR TEST DATA (SAME PROCESSING)
→ MORE ROUND AND EASIER COST FUNCTION TO OPTIMIZE

8 Vanishing/Exploding Gradients And Weight Initialization

Vanishing gradient has been a problem for **deeper networks**. If all the **weights** are less than **0**, all the **gradients** flowing back during **back propagation** will be getting smaller and smaller as they pass through **layers**. One of the ways to alleviate this problem is initializing weights properly. As if we multiplied **0.01** to initialize **weights** to be small random numbers for **sigmoid functions**, we set the **variance** of **weights** to be **2 / (number of inputs)** for **ReLU activation**. This method is called the **He initialization**.

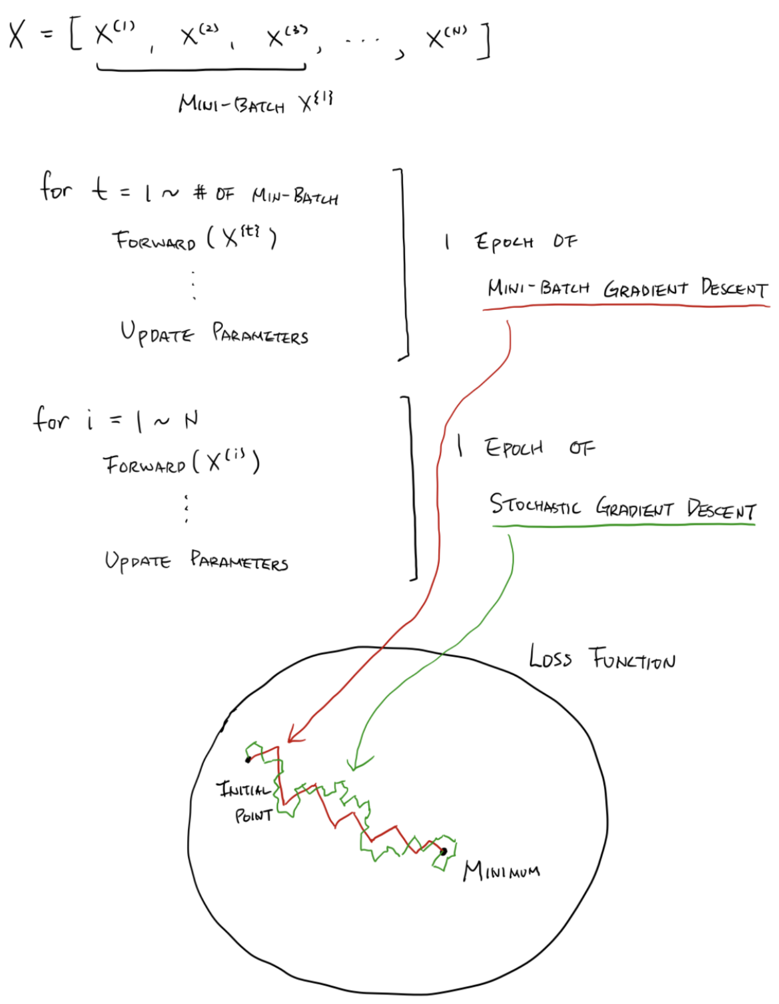


VANISHING / EXPLODING GRADIENTS
$x \rightarrow$
$\rightarrow \hat{y}$
$W^{[1]}$ $W^{[2]}$ $W^{[3]}$ $W^{[4]}$ $W^{[5]}$
IF $W < 1.0$, $\hat{y}$ WILL DECREASE EXPONENTIALLY
→ SAME APPLIES TO GRADIENTS DURING BACKPROP
WEIGHT INITIALIZATION
x_1
x_2
x_n
$\rightarrow \bigcirc \rightarrow \hat{y}$
THE HIGHER n IS, THE SMALLER W YOU WANT FOR Z THAT IS NOT TOO BIG OR SMALL
HE INITIALIZATION
→ W = np.random.rand(shape) * np.sqrt($\frac{1}{n^{[l-1]}}$)
SET $\boxed{Var(W) = \frac{1}{n}}$
★ FOR ReLU, $\frac{2}{n}$ VARIANCE WORKS BETTER ★
*np.sqrt($\frac{2}{n^{[l-1]}}$)
COULD BE HYPER PARAMETER
HELPS MAKING $A^{[l+1]}$ TO HAVE SIMILAR μ, σ^2 AS $A^{[l]}$, WHICH HELPS WITH VANISHING / EXPLODING GRADIENT PROBLEM

9 Optimization

There are several ways to **optimize** the model. Methods that we have already seen before are **stochastic gradient descent** and **mini-batch gradient descent**. **Stochastic gradient**

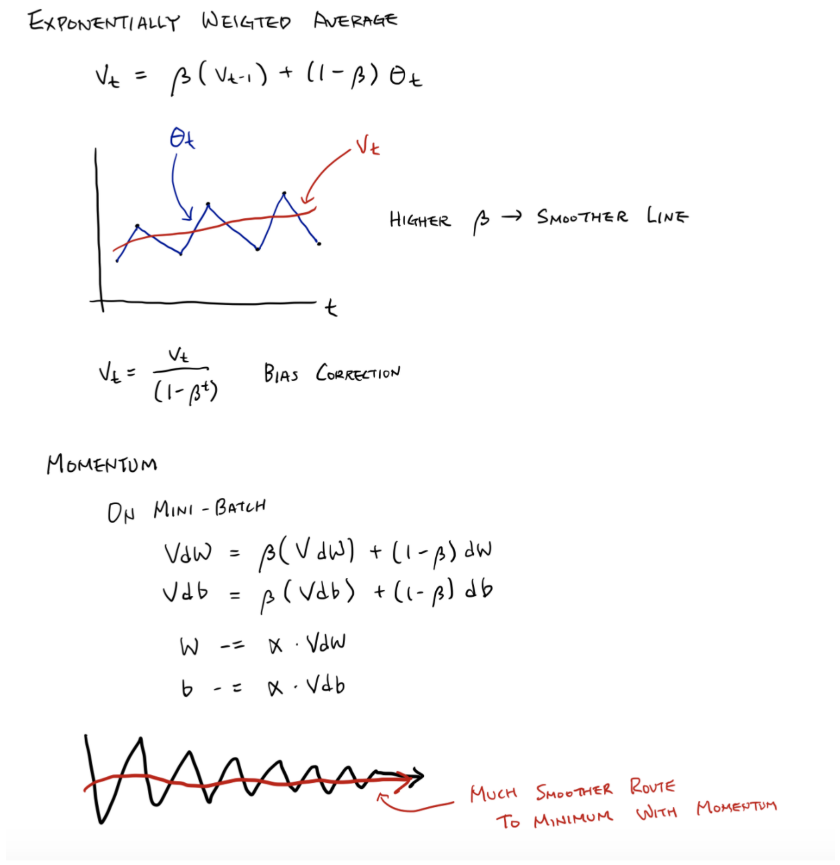
descent optimizes **parameters** by using **one sample** at a time and **mini-batch gradient descent** uses an assortment of **multiple samples** called a **mini-batch** at a time. We can consider **stochastic gradient descent** as **mini-batch gradient descent** with the **mini-batch size** of **1**. We call a cycle through training data, an **epoch**. Because of larger **mini-batch size**, **mini-batch gradient descent** has less noise than **stochastic gradient descent**. In other words, **mini-batch gradient descent** takes more direct route towards the **minimum** than **stochastic gradient descent**. **Mini-batch size** is another **hyper-parameter** that we have to figure out empirically. However, it is known that numbers like **64, 128, 256, and 512** that are computer memory sized work well for **mini-batch size**.



$X = [\underbrace{X^{(1)}, X^{(2)}, X^{(3)}}_{\text{Mini-Batch } X^{\{1\}}}, \dots, X^{(N)}]$
for t = 1 ~ # of Min-Batch
Forward ($X^{\{t\}}$)
Update Parameters
1 Epoch of
Mini-Batch Gradient Descent
for i = 1 ~ N
Forward ($X^{(i)}$)
Update Parameters
1 Epoch of
Stochastic Gradient Descent
Loss Function
Initial Point
Minimum

10 Exponentially Weighted Averages And Gradient Descent With Momentum

By using **exponentially weighted averages** of **gradients**, we could take even more direct route. **Exponentially weighted average** is computed by mixing **exponentially weighted average** till now (v_{t-1}) and **current value** (θ_t): $v_t = \beta v_{t-1} + (1\text{-}\beta)\theta_t$. β is another **hyper-parameter**, and 0.9 usually works well. You can notice from the formula that, v_t with low t will not be close to the original θ_t. We could use **bias correction** to fix such problem. After computing normal v_t, we divide it by $(1 - \beta^t)$. Overall, **exponentially weighted average** will give smoother curve than the original values. Having oscillations during **gradient descent** made it hard for us use large **learning rates**. To solve this problem, we can use **gradient descent with momentum** that uses **exponentially weighted averages** of **dW's** and **db's** (V_{dW} and V_{db}) to update the **parameters**. Things almost always work better with **momentum**.


EXPONENTIALLY WEIGTED AVERAGE
$V_t = \beta(V_{t-1}) + (1-\beta)\theta_t$
θ_t
V_t
t
HIGHER $\beta \rightarrow$ SMOOTHER LINE
$V_t = \frac{V_t}{(1-\beta^t)}$
BIAS CORRECTION
MOMENTUM
ON MINI-BATCH
$VdW = \beta(VdW) + (1-\beta)dW$
$Vdb = \beta(Vdb) + (1-\beta)db$
$W \mathrel{-}= \alpha \cdot VdW$
$b \mathrel{-}= \alpha \cdot Vdb$
MUCH SMOOTHER ROUTE TO MINIMUM WITH MOMENTUM

11 RMS Prop and Adam Optimization

RMS Prop is an **optimization** method with the similar concept except that it takes **exponentially weighted average** on **gradient squared** (S_{dW} and S_{db}) instead of normal **gradient**. To update **parameters**, it uses dW/$\sqrt{(\mathrm{S}_{dW})}$ and db/$\sqrt{(\mathrm{S}_{db})}$. Another **optimization method** that uses **exponentially weighted average** is **Adam**. It computes **exponentially weighted average** for both **gradient** and **gradient squared**. So, we need two extra hyper-parameters β_1 and β_2. **0.9** works well for β_1, and **0.999** works well for β_2. With **Adam optimization**, we update **parameters** with $V_{dW}/\sqrt{(\mathrm{S}_{dW})}$ and $V_{db}/\sqrt{(\mathrm{S}_{db})}$.

RMS PROP

$$SdW = \beta_2(SdW) + (1-\beta_2)\, dW^2$$

$$Sdb = \beta_2(Sdb) + (1-\beta_2)\, db^2$$

$$W \mathrel{-}= \alpha \cdot \frac{dW}{\sqrt{SdW}}$$

$$b \mathrel{-}= \alpha \cdot \frac{db}{\sqrt{Sdb}}$$

ADAM

COMPUTE VdW, SdW, Vdb, Vdb

BIAS CORRECTION

$$W \mathrel{-}= \alpha \frac{VdW}{\sqrt{SdW}}$$

$$b \mathrel{-}= \alpha \frac{Vdb}{\sqrt{Sdb}}$$

12 Learning Rate Decay

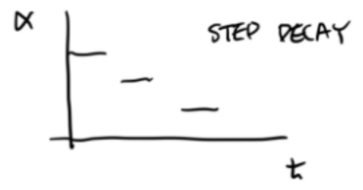
Last thing we have to know about **optimization** is **learning rate decay**. As its name suggests, we reduce **learning rate** as we get closer to the **minimum** in order to make converging to it easier. We can use several different functions to decay the original **learning rate**.

$$\alpha = \frac{1}{1 + \underline{\text{decay-rate}} * \text{epoch}} \cdot \underline{\alpha_0}$$

TWO HYPER PARAMETERS

$$\alpha = \text{decay-rate}^{\text{epoch}} \cdot \alpha_0$$

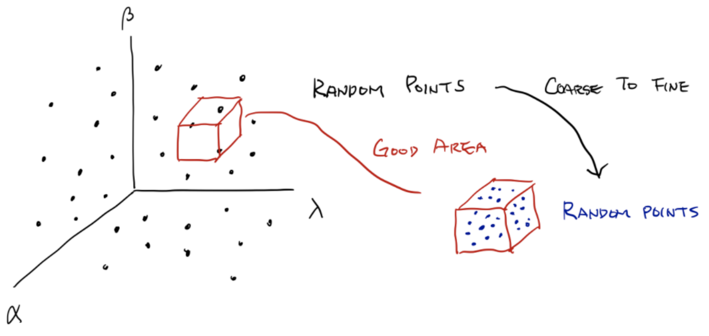
$$\alpha = \frac{k}{\sqrt{\text{epoch}}} \cdot \alpha_0 .$$

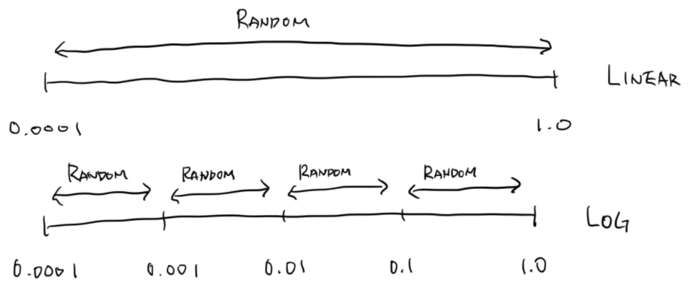


MANUAL DECAY IS AN OPTION TOO

13 Tuning Process

We have discovered many new **hyper-parameters**. How do we tune **hyper-parameters**? First, we should try to find good **learning rate** because it is the most important **hyper-parameters**. Then, we should work on finding **the number of hidden units, mini-batch size**, and β's. How do we find good value for **hyper-parameters**? We should choose random points within certain ranges of **hyper-parameters**. As we try them out, we should find the good area with smaller ranges to sample the random points more densely and keep searching from coarse to fine. There are two search scales we can use: **linear scale** and **log scale**. **linear scale** samples things within the range evenly, while log scale samples things evenly on the orders of magnitude. For example, **linear scale** will sample 0.0001 to 1.0 evenly/linearly, but **log scale** will sample evenly within the orders of magnitude: from 0.0001 to 0.001, 0.001 to 0.01, and 0.01 to 0.1, and 0.1 to 1.0. In practice, even after finding the good **hyper-parameters** for our model, we should tune our **hyper-parameters** once in a while.





14 Batch Norm

Batch normalization is like having **input normalization** for every **layer**. So it normalizes every activation to **train** every **parameters** faster. But in practice, we actually normalize **Z** which has the same effect as normalizing **A**. We first normalize **Z** into Z_{norm}. Then, we compute the $Z_{new} = \gamma \cdot Z_{norm} + \beta$. γ and β are learnable **parameters** that basically allows **Z** to have other **mean** and **variance** that are more suitable for **activation functions**. Because Z_{new} is centered to 0 with **variance** 1, it is very likely for **Z** to hit the sweet spot of **activation functions**. **Batch norm** allows the model to be more robust to **covariate shift** when the distribution of input change. Moreover **Batch norm** has slight **regularization** effect because it kind of can cancel out large W's, adding noise. Its **regularization** effect gets weaker as **batch size** gets larger because there is less noise as **batch size** gets larger. If **batch norm** is not beneficial, the model can always learn γ to be the **variance of Z** and β to be the **mean of Z** to cancel out **batch norm** and make Z_{new} equal to **Z**. During **back propagation**, we need to compute dγ and dβ to update γ and β. Each layer has its own γ and β, and their shapes are **(number of units in the layer, 1)** because γ and β has to multiply and add to every single z's from the **layer**. Obviously, dγ and dβ will have the same size. During **test time**, we use **exponentially weighted average** of **means** and **variances** from **training** across different **mini-batches** and across different **layers**. This is as if we are learning general **mean** and **variance** during training.

15 Softmax Regression

We used **sigmoid activation function** in **output layer** for our **binary classifier**. What if we have more than two **classes**? We use **Softmax function** which is really good for picking the maximum probability out of many. As its name suggests, it is a soft version of **max function**. Instead of selecting the maximum, it distributes probability in between 0 and 1 so that the maximum gets the largest portion and the minimum gets the least portion, which makes **Softmax function** suitable for **multi-class classification**. Those probabilities add up to 1. For **loss**

function, we use **cross-entropy loss**. For **backprop, dL/dZ** of the output layer beautifully works out to be **A - Y**. For implementing **Softmax function**, there is a naive way that directly follows the mathematical function. Because exponentials can easily explode beyond float64 capacity, the naive version is numerically unstable. Therefore, we generally implement the stable version of **Softmax function** that basically normalizes the values before putting them through exponentials.

$$a = \text{SOFTMAX}(z) \longrightarrow a_i = \frac{e^{z_i}}{\Sigma e^{z}}$$

STABLE SOFTMAX

$$a_i = \frac{C e^{z_i}}{C \Sigma e^{z}} = \frac{e^{z_i + \log(c)}}{\Sigma e^{z + \log(c)}}$$

WE SET $\log(c) = -\max(z)$

$$\text{CROSS ENTROPY LOSS} = -\Sigma\, y \log(a)$$