Loss Functions

Objective:

Minimize the loss between predictions and the actual output.

Mean Squared Error (L2 Loss)
- Most simple/widely used function.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

$\hat{Y}_i$ - Predicted output

Y_i - Actual Output.

n - Training samples in minibatch (or entire training samples)

Variations of Mean Squared Error.

- Half of the Mean Squared Error

$$MSE = \frac{1}{2n} \sum_{i=1}^{n} (\hat{Y}_i - Y_i)^2$$

- Root Mean Squared Error.

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (\hat{Y}_i - Y_i)^2}$$

MSE is high for large loss values and decreases as loss approaches 0

L1 Loss / Mean Absolute Error (MAE) Loss

Instead of squaring it, we take the absolute value of the error.

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |Y_i - \hat{Y}_i|$$

$\hat{Y}$:= predicted output

n = training samples

Y_i = Actual output

Outlier. – Deviation from your data point.

L1 or L2 Loss

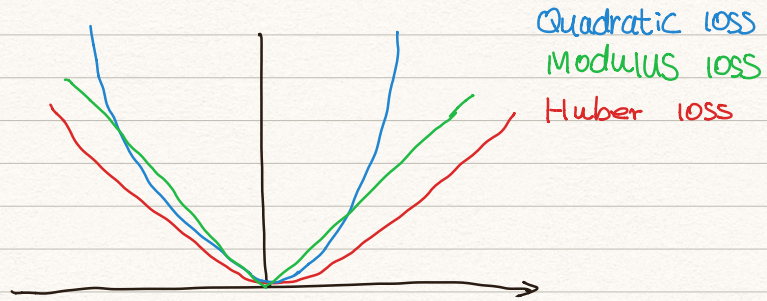
- L2 loss (MSE) is more sensitive to outliers.
- When there are large deviations, the error is big.
- Consequently, L1 loss function is more robust and is generally not affected by outliers.
- L2 loss function will try to adjust the model according to these outlier values.

Huber Loss (Smooth L1 Loss)

- Combination of L1 and L2 loss function.
- Less sensitive to outliers in data than the squared error loss
- Basically absolute error which becomes quadratic when the error is small.
- The threshhold for "small" depends on a hyperparameter δ (delta), which can be tuned.
- The choice of delta is critical because it determines

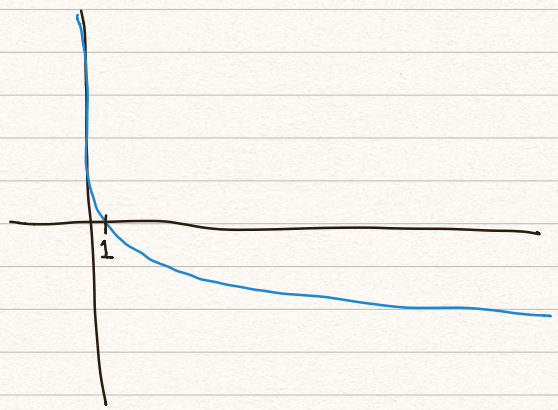
what you're willing to consider as an outlier.

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x)| \le \delta \\ \delta |y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$



Cross Entropy Loss (= Negative Logarithmic Function)

Graph for $-\log(x)$



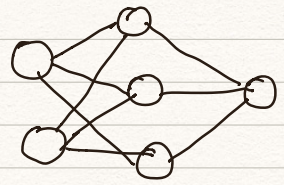
- small input large output
- Large input small output.

- Binary Cross Entropy.
  - usually used when the output labels have values 0 or 1 or between 0 and 1. (e.g. Yes/No)

$$-\frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{c}\left[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)\right]$$

y - Actual class label (0 or 1)
p - predicted probability for the class
c - number of classes
n - number of samples.

When we only have two classes



1 or 0

```
def CrossEntropy(yHat, y):
    if yHat == 1:
        return -log(y)
    else:
        return -log(1-y)
```

$$-\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(p_i) + (1-y_i)\log(1-p_i)\right]$$

If label is 1: $-y_i \log(p_i)$

If label is 0: $-\log(1-p_i)$

Multi-label classification

	Loss
0.6	1
0.9	0
0.2	0
0.8	1

Consider $n=1$ sample.

$$\text{BCE Loss} = -\sum_{i=1}^{c}\left[y_i \log(p_i) + (1-y_i)\log(1-p_i)\right]$$

$$= -\sum\left[\log(0.6) + \log(1-0.9) + \log(1-0.2) + \log(0.8)\right]$$

- If the label is 1 and the prediction is 0.1

  $-y\log(p) = -\log(0.1)$ Loss is high

- If the label is 1 and the prediction is close to 1 (0.9)

  $-y\log(p) = -y\log(0.9)$ Loss is low.

- If the label is 0 and the prediction is 0.9.

$$-(1-y)\log(1-p) = -\log(1-0.9)$$

$$-\log(0.1) \qquad \text{Loss is high.}$$

- If the label is 0 and the prediction is 0.1, Loss ↓

Ideally, we want label 1, prediction 1. Label 0, prediction 0.

Cross Entropy

$$CE = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{c} y_i \log \hat{y}_i$$

i = class number
c = number of classes
y_i = actual label
$\hat{y}_i$ = predicted label

The actual outputs should be in the form of a one-hot vector.

Suppose we have 4 labels (4 outputs), then

Class 1 → $\begin{matrix}1\\0\\0\\0\end{matrix}$ Class 2 → $\begin{matrix}0\\1\\0\\0\end{matrix}$ Class 3 → $\begin{matrix}0\\0\\1\\0\end{matrix}$

Class 4 → $\begin{matrix}0\\0\\0\\1\end{matrix}$

If label = 0 (wrong) → There is no loss calculation

If label = 1 (correct) → Loss calculation.

Loss only calculated for correct predictions.

It penalizes probabilities of correct classes only.

- You only calculate the loss for the correct classes.

For example.

- Suppose we have 4 different classes to classify
- For a single Training example:

Actual label [1 0 0 0]

Predicted label [0.1 0.4 0.2 0.3]

$$\text{Cross Entropy Loss} = -(1 \times \log(0.1+0+0+0))$$
$$= -\log(0.1)$$
$$= 2.303$$

* Ignore the loss for the 0 labels.
* The loss doesn't depend on the probabilities for the incorrect classes!

Sometimes, the cross entropy loss is averaged over the training samples n.

$$J = -\frac{1}{n}\left(\sum_{i=1}^{n} y_i \log \hat{y}_i\right)$$

If the prediction is close to the outcome.

e.g

Actual [1 0 0 0]

Predicted [0.9 0.01 0.05 0.04]

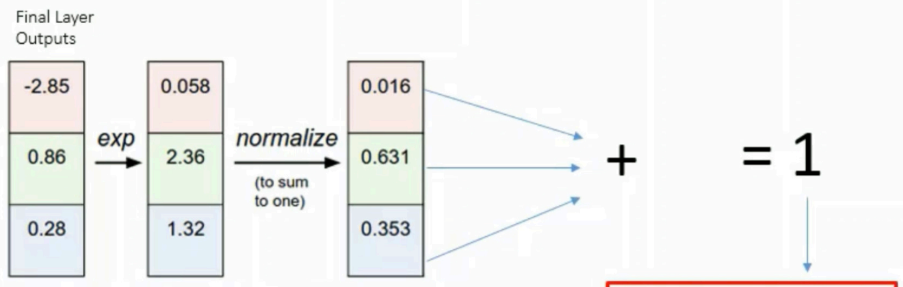
$$\text{Cross Entropy Loss} = -(1 \times \log(0.9)+0+0+0)$$
$$= -\log(0.9) = 0.04$$ LOSS is low.

Softmax Function

- Cross Entropy loss comes with the Softmax layer.
- The softmax layer turns all the class probabilities to values that sum up to 1.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

K = Number of classes
Do this for j = 1.....K (for all output neurons)



http://cs231n.github.io/linear-classify/#softmax

$$0.016 + 0.631 + 0.353 = 1$$

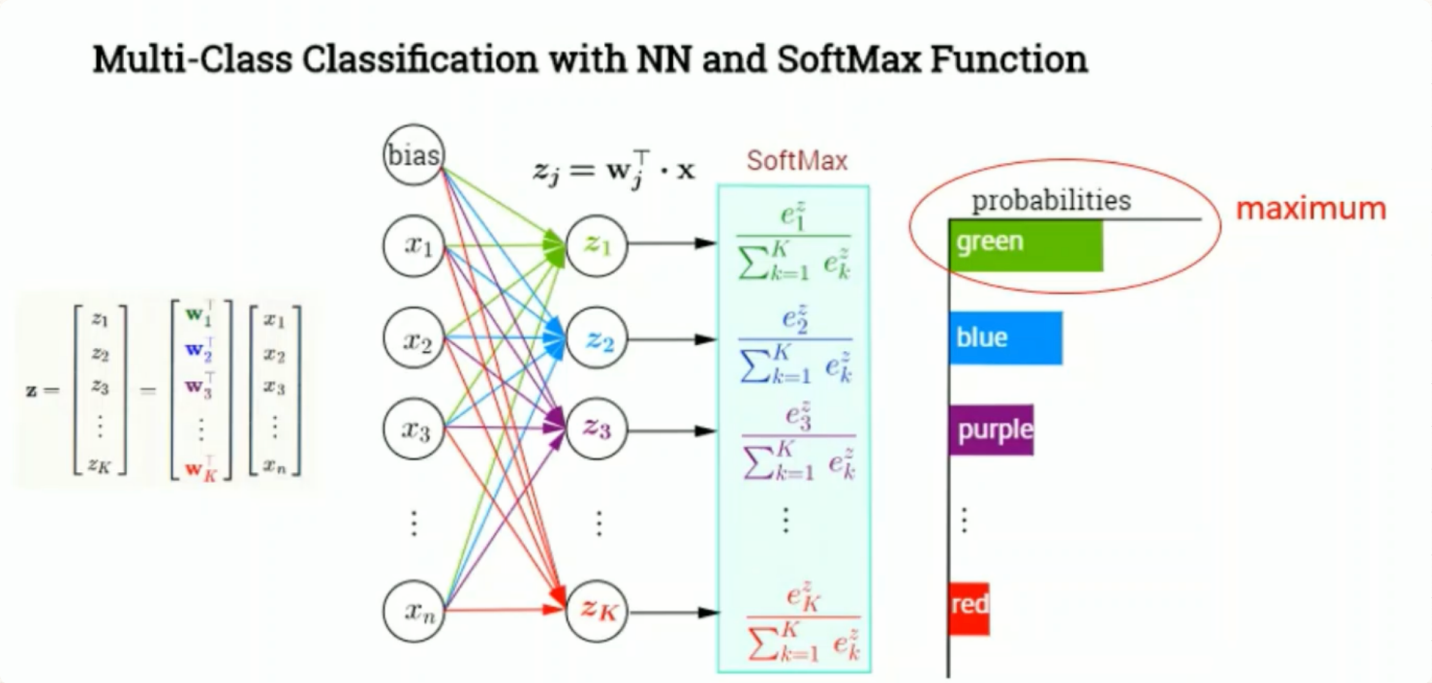
$$\sum_{k=1}^{K} e^{z_k} = 0.058 + 2.36 + 1.32 = 3.73$$

$$e^{-2.85} = 0.058 \qquad \frac{0.058}{3.73} = 0.016$$

$$e^{0.86} = 2.36 \qquad \frac{2.36}{3.73} = 0.631$$

$$e^{0.28} = 1.32 \qquad \frac{1.32}{3.73} = 0.353$$

Basically distributing each output that sums upto 1.



Why do we need to apply the softmax when we are just looking for the max value?

It's not used for the testing, but for training the cross entropy. Since cross entropy used logarithms of the probability.

* Code Implementation

From Scratch

```
In [60]: lis = []
         for value in input:
             l.append(math.exp(value))

In [61]: lis

Out[61]: [2.526967465897281,
          0.22386275805183797,
          1.3071939981655192,
          0.2663413956582393,
          0.3741513995559274,
          2.564505636175701]

In [62]: tens = torch.tensor(lis)

In [63]: sum = torch.sum(tens)

In [64]: sum

Out[64]: tensor(7.2630)

In [65]: #Verify
         print(2.5269/7.2630)
         print(2.5269/sum)
```

0.3479

```
print(tens[0]/sum.item())
```

Using PyTorch to Verify

```
In [59]: m = nn.Softmax()
         input = torch.randn(2, 3)
         input = input.view(6)
         print("Input Tensor",input)
         output = m(input)
         print("Output Tensor",output)

Input Tensor tensor([ 0.9270, -1.4967,  0.2679, -1.3230, -0.9831,  0.9418])
Output Tensor tensor([ 0.3479,  0.0308,  0.1800,  0.0367,  0.0515,  0.3531])
```

KL divergence

Measures how two probabilities $P(X)$ and $Q(X)$ are different. If these two distributions are different, KL divergence gives a high value.

- If they are similar, returns a low value.
- If they are equal, returns zero.

It is the difference between entropy and cross entropy loss:

$$KL(y \| \hat{y}) = \sum_i y_i \log \frac{1}{\hat{y}} - \sum_i y_i \log \frac{1}{y_i}$$

$$= \sum_i y_i \log \frac{y_i}{\hat{y}_i}$$

Same equation.

It is never negative and only 0 when $y_i = \hat{y}_i$ since $\log 1 = 0$

$$D_{KL}(P \| Q) = \sum_{i=0}^{n} p(x_i) \log(p(x_i)) - \sum_{i=0}^{n} p(x_i) \log(q(x_i))$$

One major use of KL divergence is in Variational Autoencoders

Q. Is KL-divergence same as cross entropy for image classification?

A. Yes.

$$KL(y \| \hat{y}) = \sum_i y_i \log \frac{1}{y_i} - \sum_i y_i \log \frac{1}{y_i} = \sum_i y_i \log \frac{y_i}{\hat{y}_i}$$

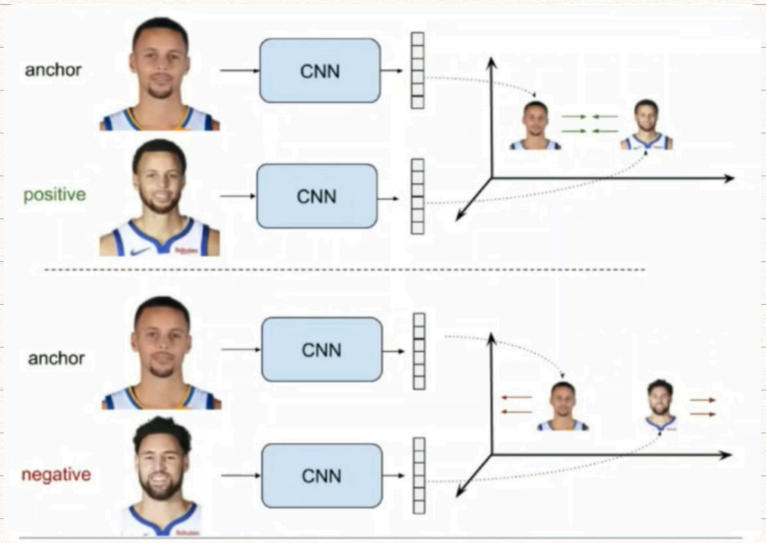
In image classification, we use one-hot encoding for our labels. Therefore, when y_i is the actual label, it equals 1 → $\log(1) = 0$, and the whole term is cancelled.
when y_i is not correct label, it equals 0 and the whole term is also cancelled out.

$$KL(y \| \hat{y}) = \underbrace{\sum_i y_i \log \frac{1}{\hat{y}}}_{\text{Cross Entropy}} - \underbrace{\cancel{\sum_i y_i \log \frac{1}{y_i}}}_{\text{Entropy}}$$

Therefore, KL divergence = Cross Entropy in image classification tasks.

Contrastive Loss

Measure the distance or similarity between two inputs.



← In this case, the distance is close.

← In this case, the distance is far from each other

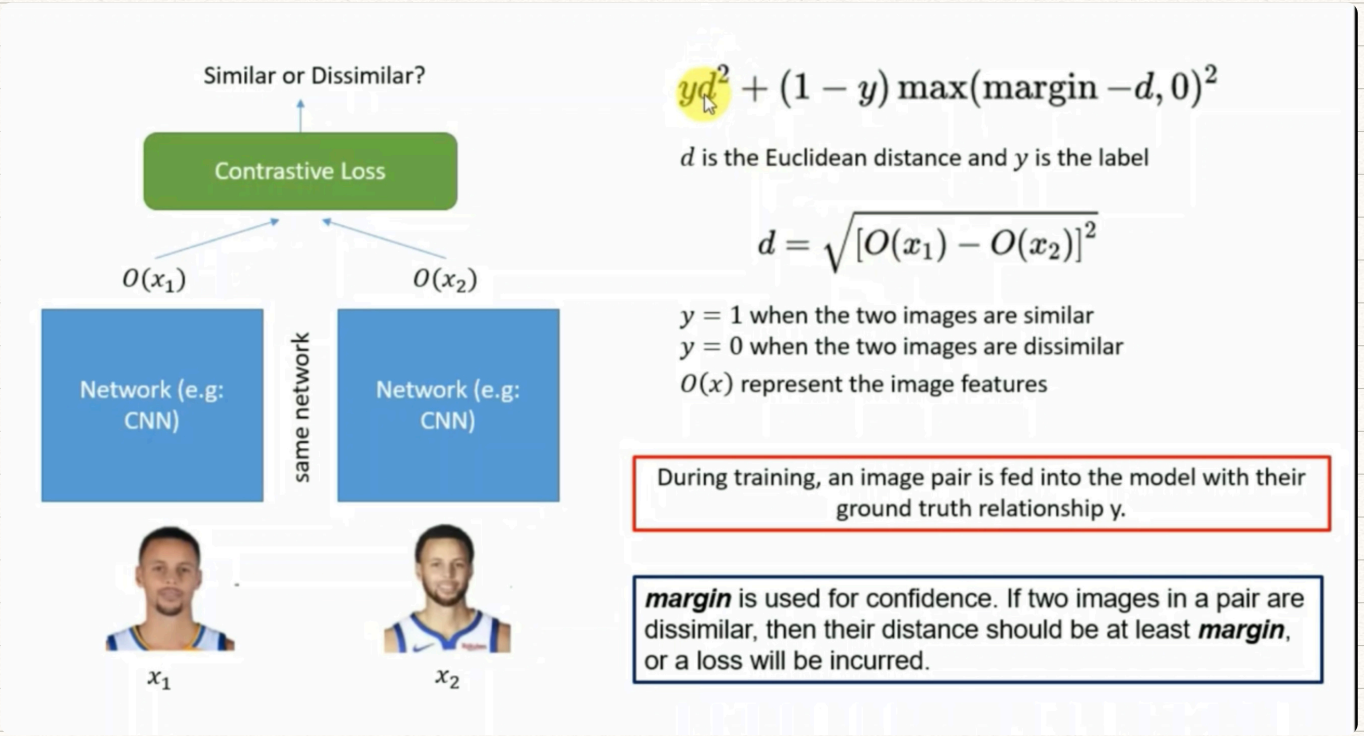
- Contrasive loss is a distance-based loss function (as opposed to prediction error-based losses like cross entropy) used to learn discriminative features for images.

- Like any-distance-based loss, it tries to ensure that semantically similar examples are embedded close together. It is calculated on Pairs.

- The loss measures the similarity between two inputs.

- Each sample is composed of two images (positive pairs or negative pairs). Our goal is to maximize the distance between negative pairs and minimize the distance between positive pairs

- We want small distance between the positive pairs (because they are similar images/inputs) and great distance than some margin m for negative pairs.



So, when $y=1$. (The pairs are similar),

$yd^2 + (1-y)\max(\text{margin} - d, 0)^2$ becomes yd^2. Because $(1-y)$ becomes zero. So, we want to minimize the distance d^2.

If $y=0$ (not similar), so yd^2 cancels away and stay with $(1-y)\max(\text{margin} - d, 0)^2$.

This function doesn't only penalize when it is wrong but

also correct classification that is not confident enough.

Loss becomes zero when distance is bigger than margin.

Hinge Loss

- Associated usually with support vector machines.
  - support vector machines.
    - machine learning technique (supervised model) that is usually used for classification.

- The 0/1 Loss

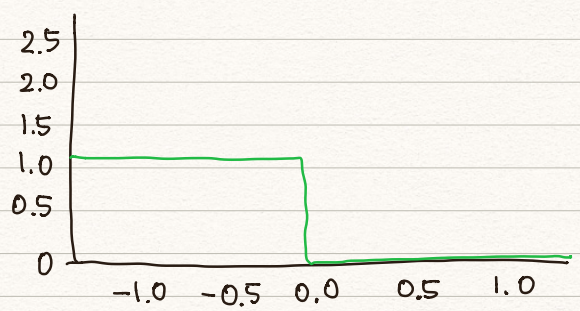
Consider y to be the actual label (-1 or 1)

and $\hat{y}$ to be the prediction.

If we multiply y and $\hat{y}$ together

	$y=1$	$y=-1$
$\hat{y}=1$	Pos	Neg
$\hat{y}=-1$	Neg	Pos

Pos = 0 loss
Neg = 1 loss



* Hinge Loss

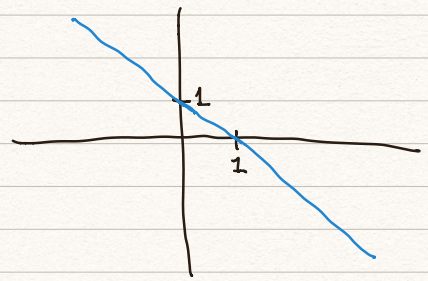
The hinge loss uses the similar idea, but the loss grows proportionally to the error.

Rather than penalizing with just 1 ☹, we make the penalization linear/proportional to the error

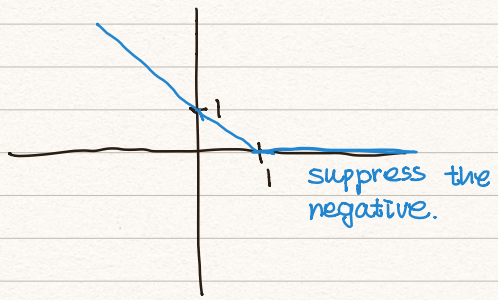
Also, we want to penalize the prediction that is correct but not confident enough. Here we will again introduce the idea of margin

* Consider the plot of $1-x$.

Graph for $1-x$



Graph for $\max(0, 1-x)$



- A marginal loss, usually used for SVMs.
- Used when labels are [-1, 1]
- It penelizes not only wrong predictions, but correct predictions which are not confident enough.
- Faster than cross entropy, but accuracy is degraded.

For all samples : $\sum \max(0, 1 - y * \hat{y})$

Where y is the actual label (-1 or 1) and $\hat{y}$ is the prediction.

The loss is 0 when the sign of the label and prediction match.

Code Implementation.

```
def Hinge(yHat, y):
    return np.max(0, 1 - yHat * y)
```

Consider the prediction when the actual label is -1:

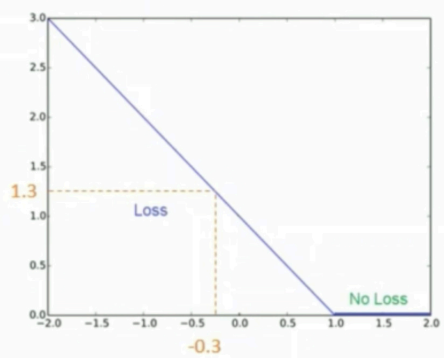
max[0, 1-(-1*0.3)] = max[0, 1.3] = 1.3 → Loss is high

max[0, 1-(-1*-0.8)] = max[0, 0.2] = 0.2 → Loss is low

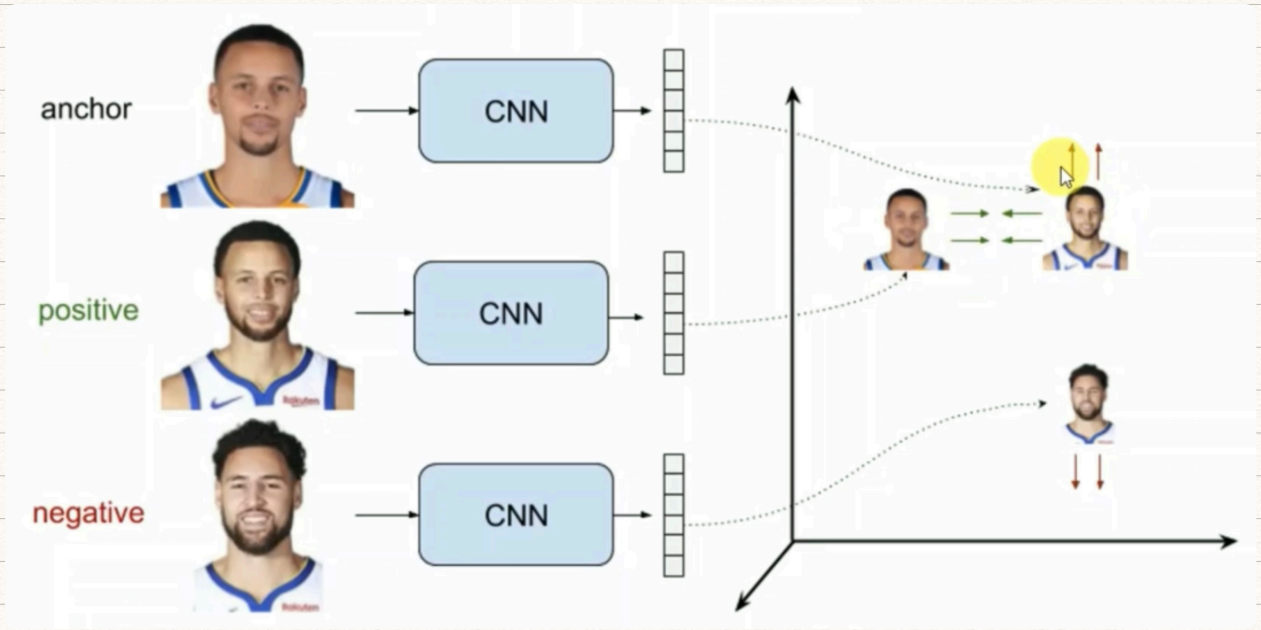
max[0, 1-(-1*-1.1)] = max[0, -0.1] = 0 → No Loss!

max[0, 1-(-1*-1)] = max[0, 0] = 0 → No Loss!

max[0, 1-(-1*1)] = max[0, 2] = 0 → Loss is very high



Triplet Ranking Loss



Our goal is to minimize the distance between the anchor and positive images and maximize the distance between the anchor and negative images.

I_a (anchor) : Reference Image.

I_p (positive) : Positive Image

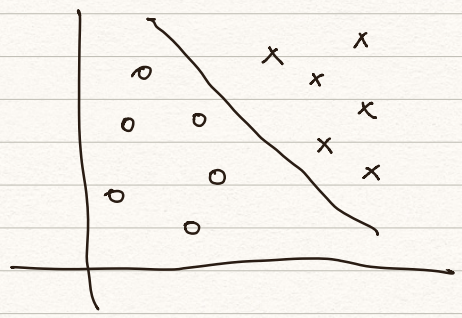
I_n (negative) : Negative Image.

$$\max(\|f_a - f_p\|^2 - \|f_a - f_n\|^2 + m, 0)$$

Activation Function

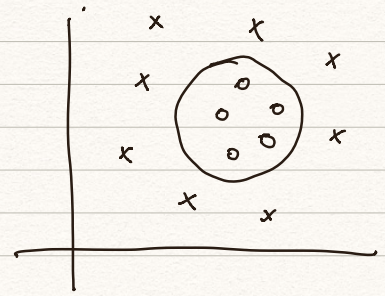
- Recap.

Linearly seperable.



The 2 classes can be seperated by a straight line.

Linearly unseperable



The 2 classes can be seperated by a curve or a more complex function than a straight line.

Unfortunately, real world data is non-linear. We need more complex functions in order to predict / classify the data.

Sigmoid Activation

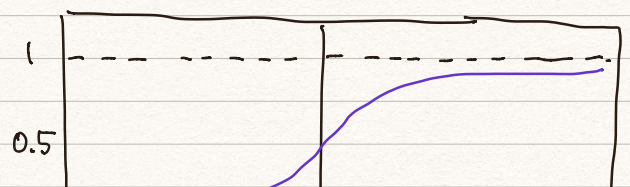
$$\phi(z) = \frac{1}{1+e^{-z}}$$

Any input will be scaled to a value between 0-1.

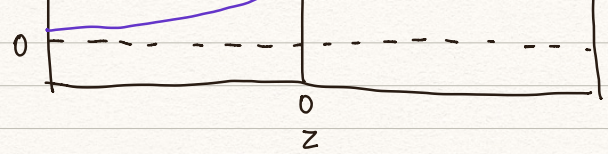
Ex:

$$x = 2 \rightarrow f(x) = \frac{1}{1+e^{-2}} = 0.88080$$

$$x = -1 \rightarrow f(x) = \frac{1}{1+e^{1}} = 0.26894$$



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



Sigmoid Function Derivative

$$y = \frac{1}{1+e^{-x}}$$

$$\frac{dy}{dx} = -\frac{1}{(1+e^{-x})^2}(-e^{-x})$$

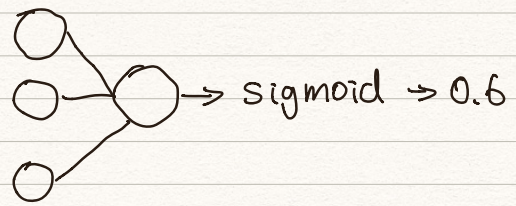
$$= \frac{e^{-x}}{(1+e^{-x})^2}$$

$$= \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right)$$

$$= y(1-y)$$

Sigmoid can act as a gate as well.

When we only have two classes (binary classification)



class 1 p: 0.6

class 2 p: 1 − 0.6 = 0.4

We do this procedure for all samples n and take the average.

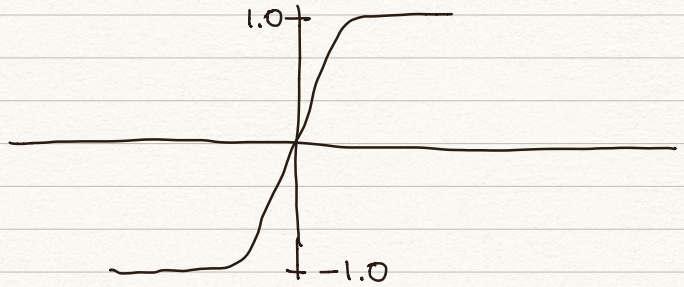
$$-\frac{1}{n}\sum_{i=1}\underbrace{[y_i \log(p_i) + (1-y_i)\log(1-p_i)]}_{\text{For each sample.}}$$

If label is 1: $-\log(p_i)$

If label is 0: $-\log(1-p_i)$

Hyperbolic Tan Activation (Tanh Activation)

tanh(x) Hyperbolic tangent function

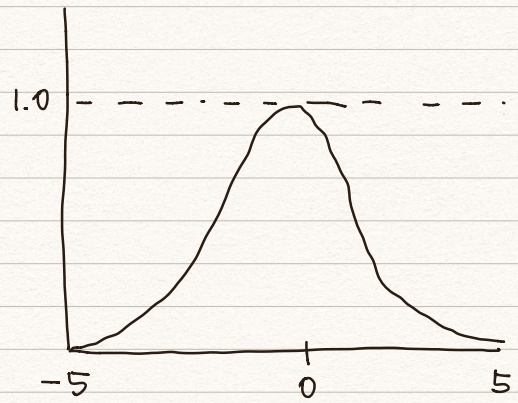


$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Examples:
$f(0) = 0$
$f(1) = 0.761$
$f(-0.5) = -0.462$
$f(1.2) = 0.833$

The tanh activation ranges between 1 and -1.

Derivative of Tanh

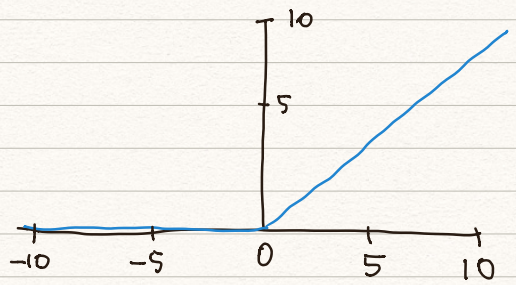


$$\frac{d}{dx} \tanh(x) = 1 - \tanh(x)^2$$

ReLU and PReLU

ReLU (Rectified Linear Unit)

$R(z) = \max(0, z)$



If input is negative, output is zero.

If input is positive, stays positive.

Derivative of ReLU

$$f(x) = \begin{cases} 0 & \text{for } x<0 \\ x & \text{for } x \geq 0 \end{cases}$$

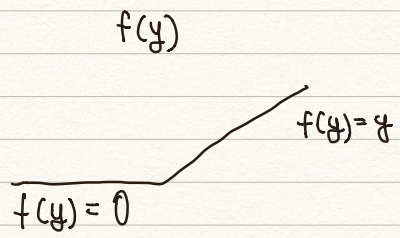
$$f'(x) = \begin{cases} 1, & \text{if } x>0 \\ 0, & \text{otherwise} \end{cases}$$

Problem:
when the input is negative, the output is 0 and the gradient will die.

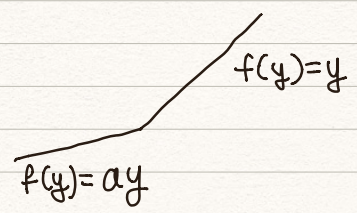
Example:
when initializing from the from the normal distribution $N(0,1)$, half of the values are negative.
Activating with ReLU means setting half of the values to 0.

Parametric ReLU (PReLU)

PReLU solves the dying ReLU problem (ReLU suffers from dead neurons of output 0), where gradients become zero.



ReLU



PReLU

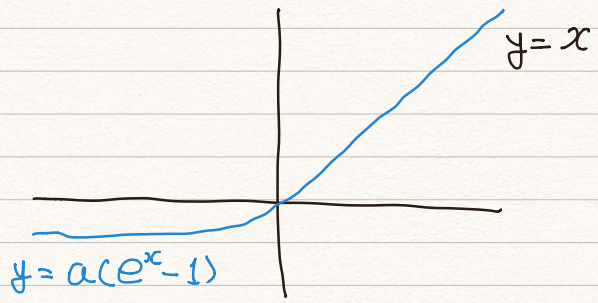
If $a=0.01$,
Leaky ReLU (static)

a is a learnable parameter

PReLU derivative:

$$f'(x) = \begin{cases} a & \text{for } x<0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

Exponentially Linear Units (ELUs)

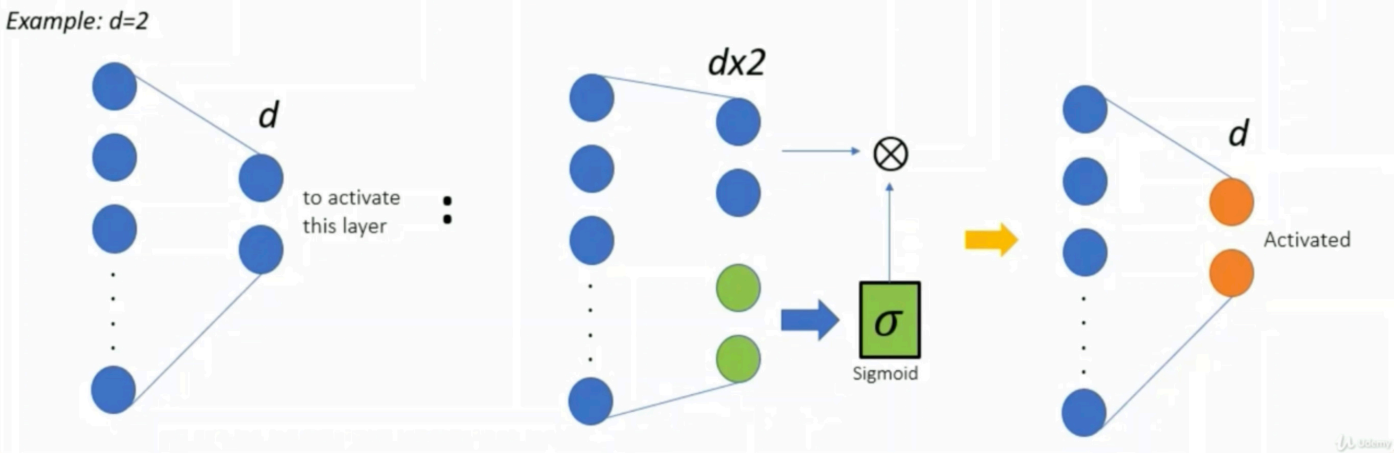


$$f(x) = \begin{cases} x & \text{if } x > 0 \\ a(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

Gated Linear Units (GLU)

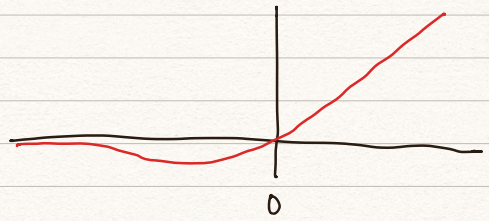
To activate a layer of dimension d, we make it output double its dimension (2xd). Then we split it into 2 halves. The first half acts as the original layer output, and the second half acts as a gating layer. The gating layer is followed by a sigmoid activation function which squashes each value to a range of 0-1. The gate values are then element-wise multiplied with the first half.

$$(XW_1 + b_1) \otimes sigmoid(XW_2 + b_2)$$



Swish Activation

Multiplication of the input x with the sigmoid function for x.



$$f(x) = x * \text{sigmoid}(x)$$
$$= x * (1 + e^{-x})^{-1}$$

- Swish is a smooth function.
  - does not abruptly change direction like ReLU
- Non-monotonic
  - → does not remain stable or move in one direction.

Why is it better than ReLU?

- sparsity: Very negative weights are zeroed out.
- For very large values, the outputs do not saturate to the maximum value.
- Small negative values are zeroed out in ReLU. However, those negative values may still be relevant for capturing patterns underlying the data, which are still preserved in Swish.

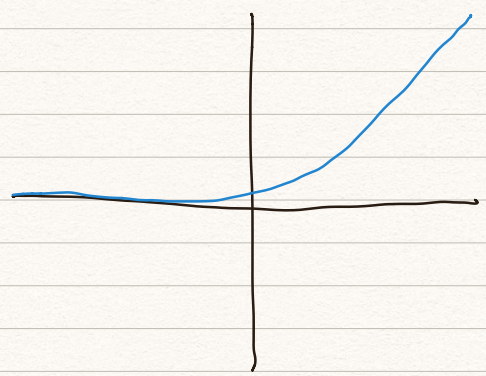
Mish Activation

Before learning mish activation, we need to learn softplus activation function.

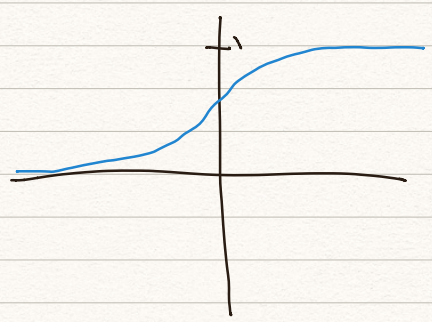
$$f(x) = \frac{1}{\beta} * \log(1 + \exp(\beta * x))$$

if $\beta = 1$: (you can scale β).

$$f(x) = \log(1 + e^{x})$$



Softplus Derivative



The sigmoid function

$$f'(x) = \frac{1}{1 + e^{-x}}$$

Mish Activation Function

- modified gated form of Softplus Activation Function.

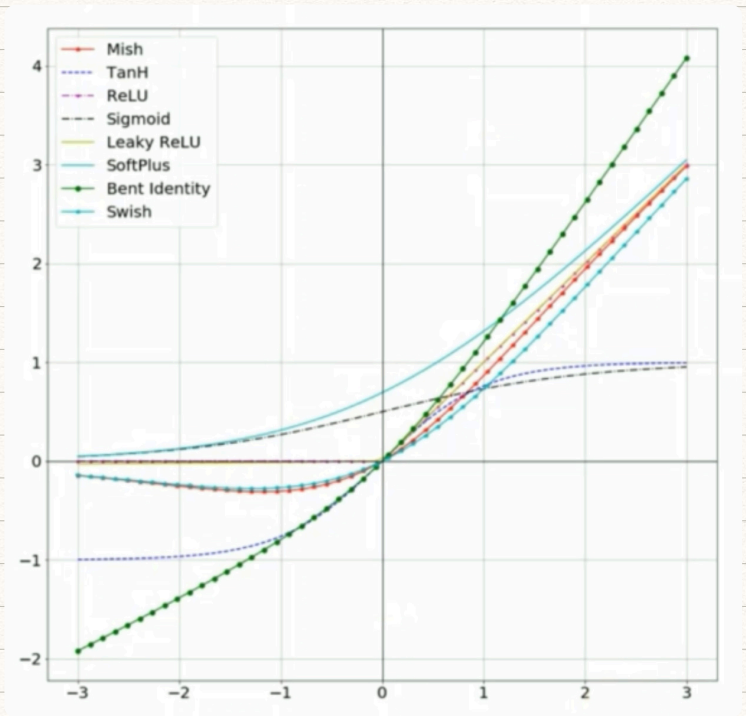
$$f(x) = x \cdot \tanh(\ln(1+e^x))$$

Code

```
x * (torch.tanh(F.softplus(x))
```

- unbounded above
  → No saturation
- Small negative values are not zero.
  → allows better gradient flow vs a hard zero bound as in ReLU
- Continuous.
  → unlike ReLU which is discontinuous at zero, which helps in effective optimization and generalization.

Compared with other activation functions.



Mish activation function:

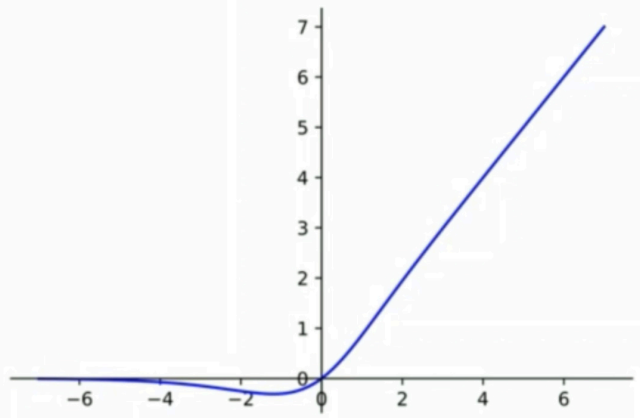


Table 6. CIFAR-100 Results (Test Top-1 Accuracy)

Model	Mish	Swish	ReLU
ResNet v2-110	**74.41%**	74.13%	73%
WRN 22-10	**72.32%**	71.89%	72.2%
WRN 40-4	69.52%	**69.59%**	69.35%
DenseNet - 121	**66.31%**	65.91%	65.50%
DenseNet - 169	65.38%	**65.69%**	64.99%
ResNext-50	**67.58%**	66.72%	67.52%
MobileNet v1	**50.09%**	49.95%	49.20%
SE Net-18	**64.38%**	63.89%	62.71%
Shuffle Net v2	**59.35%**	58.91%	58.56%
Squeeze Net	**63.07%**	62.11%	60.92%

2.15% increase from ReLU!

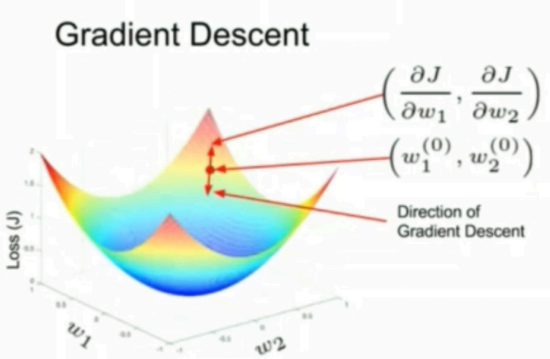
Optimization Algorithms in Neural Networks.

The way you update weights. (method of learning).

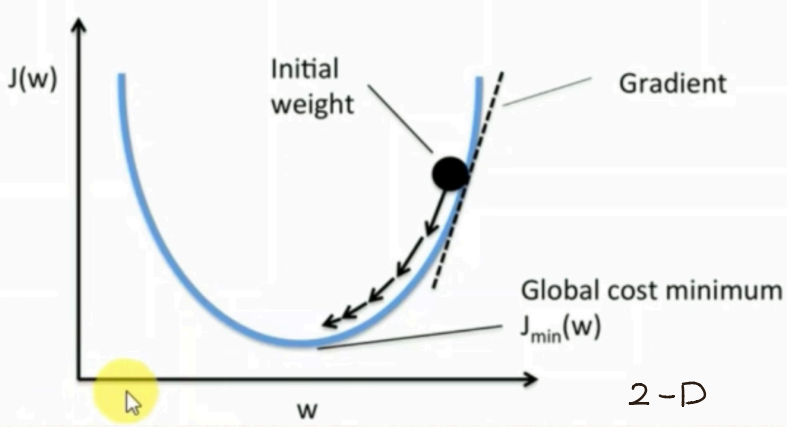
How it uses the gradient descent.

Gradient Descent

- The base algorithm that is used to minimize the error with respect to the weights of the neural network.
- The learning rate determines the step size of the update used to reach the minimum.
- An epoch is one complete pass through the entire samples.



3-D



2-D

- Changing the weights so that the error will decrease.

Batch Gradient Descent.

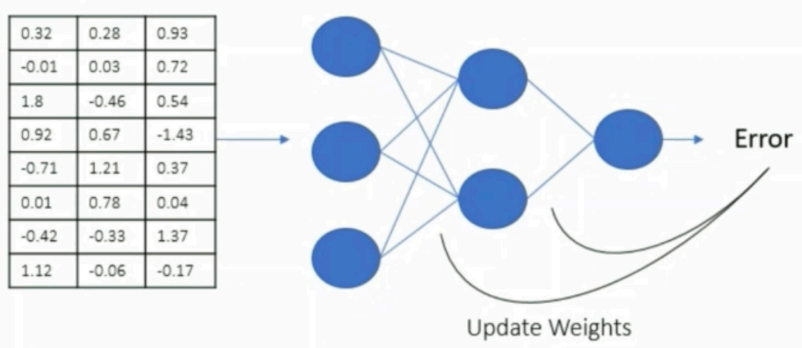
- Take all the samples in one iteration. In this case, iteration equals epoch.
- All the samples are passed to the neural network at one time.
- Computes the gradient of the loss function with respect to the network's weights for the entire training dataset, and the performs one update.
- n = all samples per iteration / epoch.
- In literature, weights w are sometimes denoted as θ

Weight Update Rule.

$$\theta = \underbrace{\theta}_{\text{old weights}} - \eta \cdot \nabla_\theta \underbrace{J(\theta)}_{\text{Loss Function of all samples.}}$$

For example,

	w_1	w_2	w_3
Sample 1	0.32	0.28	0.93
Sample 2	-0.01	0.03	0.72
Sample 3	1.82	-0.46	0.54
Sample 4	0.92	0.67	-1.43
Sample 5	-0.71	1.21	0.37
Sample 6	0.01	0.78	0.04
Sample 7	-0.42	-0.33	1.37
Sample 8	1.12	-0.06	-0.17



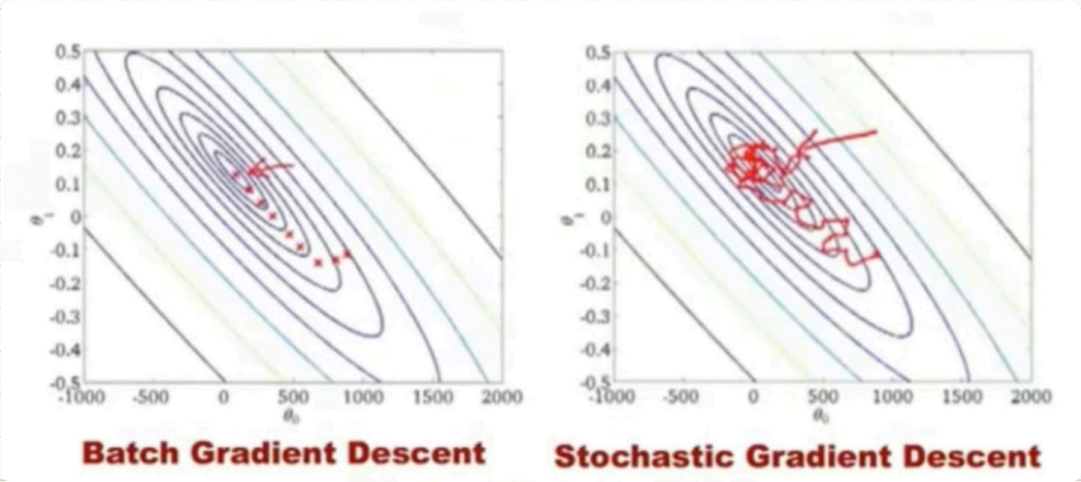
Stochastic gradient descent.

- Takes in one sample at each iteration.
- Performs the weights update for each sample at a time.
- Number of iteration per epoch is the number of samples.
- It has high variance of updates and causes the function to fluctuate.

For each sample x^i and y^i

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

$n =$ per iteration.



Take one sample and feed it to the network. Calculate error based on one sample and update weights. Repeat this steps for all samples.

Stochastic Gradient Descent has lots of noises, but sometimes outperforms the other optimization functions.

Mini-batch Gradient Descent

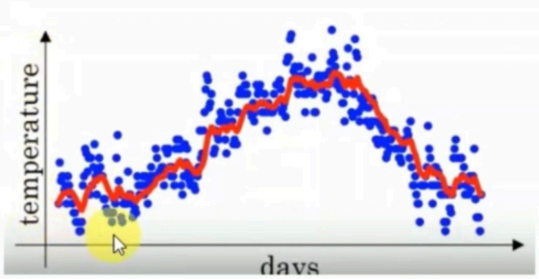
- Takes the best of both the Batch and Stochastic Gradient Descent.
- Perform weight update for a batch of samples.
- Take n training samples → Feed to the network

  $n =$ mini-batch size.
- Number of iterations per epoch = samples / batch size.

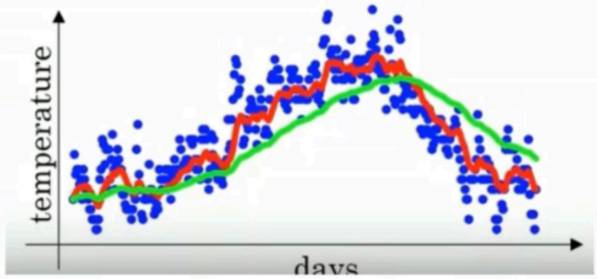
for each batch $x^{(i:i+n)}$ and $y^{(i:i+n)}$:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

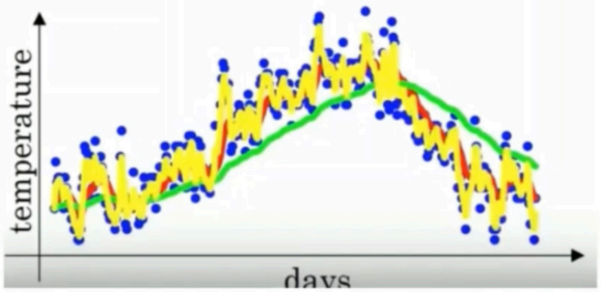
Exponetially weighted Average



$\beta = 0.9$ (*Averaging Over* 10 *points*)



$\beta = 0.98$ (*Averaging Over* 50 *points*)



$\beta = 0.5$ (*Averaging Over* 2 *points*) → *More noisy*

We are averaging over $\frac{1}{1-\beta}$ points

If current data points rapidly change → it won't get affected

Implementation

Formula.

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

if $\beta = 0.90$

$$v_t = 0.9 v_{t-1} + 0.1\,\theta_t$$

In this case, more weights to previous values and less weights to the current values.

$$v_{50} = 0.9v_{49} + 0.1(\theta_{50})$$
$$v_{49} = 0.9v_{48} + 0.1(\theta_{49})$$
$$v_{48} = 0.9v_{47} + 0.1(\theta_{48})$$
$$v_{47} = 0.9v_{46} + 0.1(\theta_{47})$$
$$v_{46} = 0.9v_{45} + 0.1(\theta_{46})$$

It is always from the previous weights.

$$v_{50} = 0.1\theta_{50} + 0.9\left[0.9\left(0.9v_{47} + 0.1(\theta_{48})\right) + 0.1(\theta_{49})\right]$$
$$v_{50} = 0.1\theta_{50} + (0.9)^3 v_{47} + (0.9)^2(0.1)\theta_{48} + 0.9(0.1)\,\theta_{49}$$
$$v_{50} = 0.1\theta_{50} + 0.9(0.1)\,\theta_{49} + (0.9)^2(0.1)\theta_{48} + (0.9)^3 v_{47}$$

$$v_{50} = 0.1\theta_{50} + 0.9\left[0.9\left(0.9\left(0.9v_{46} + 0.1(\theta_{47})\right) + 0.1(\theta_{48})\right) + 0.1(\theta_{49})\right]$$
$$v_{50} = 0.1\theta_{50} + (0.9)^4 v_{46} + (0.9)^3(0.1)\theta_{47} + (0.9)^2 0.1(\theta_{48}) + 0.9\left(0.1(\theta_{49})\right)$$
$$v_{50} = \underline{0.1\theta_{50}} + \underline{0.9(0.1)}\theta_{49} + \underline{(0.9)^2 0.1}(\theta_{48}) + \underline{(0.9)^3(0.1)}\theta_{47} + \underline{(0.9)^4}v_{46}$$

Sum up to 1. (**0.1 only** for current, and **all the rest** for previous)

$$\boxed{v_t = 0.1\theta_t + 0.1\sum_{n=1,t=t}^{n=t,t=1} (0.9)^n\,\theta_{t-1}}$$

Bias Correction of Exponentially Weighted Averages

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

Since if we assume $v_0 = 0$, then the starting value is too small.

For e.g. if we set $\beta = 0.98$, $v_1 = \beta v_0 + (1-\beta)\theta_1$

$$v_1 = 0 + 0.02\theta_1$$
$$v_2 = 0.98(0.02\theta_1) + 0.02\theta_2$$
$$= 0.0196\theta_1 + 0.02\theta_2$$

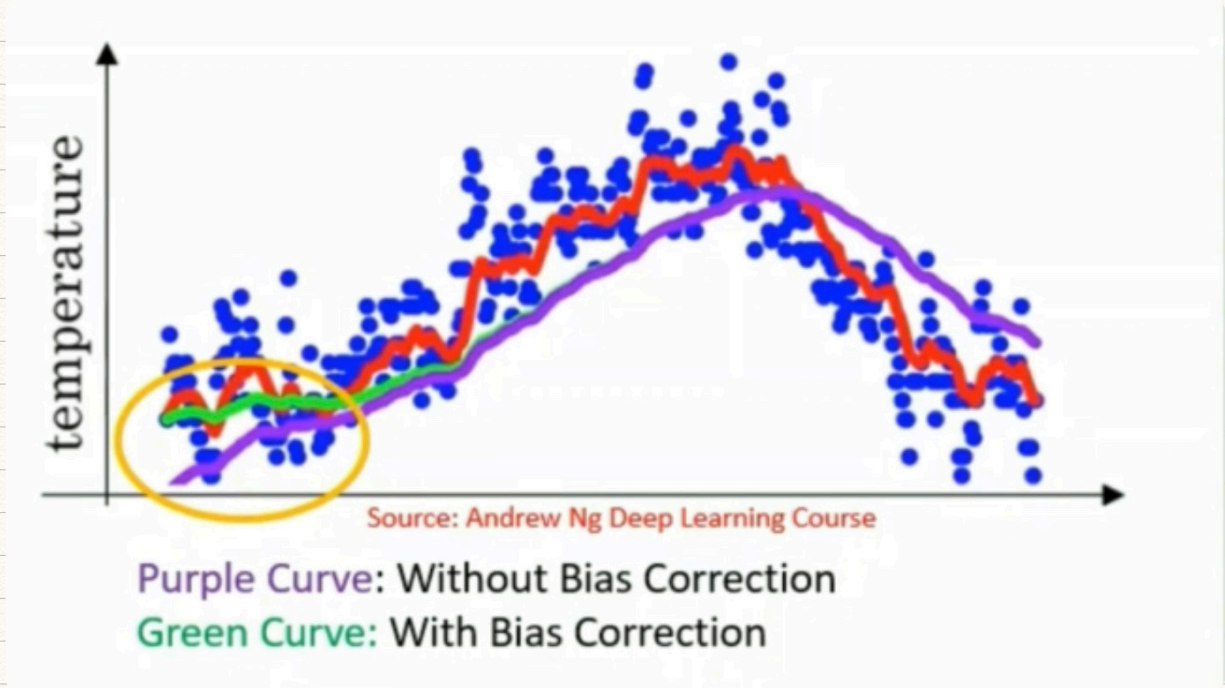
Initial values get smaller.

To mitigate this problem:

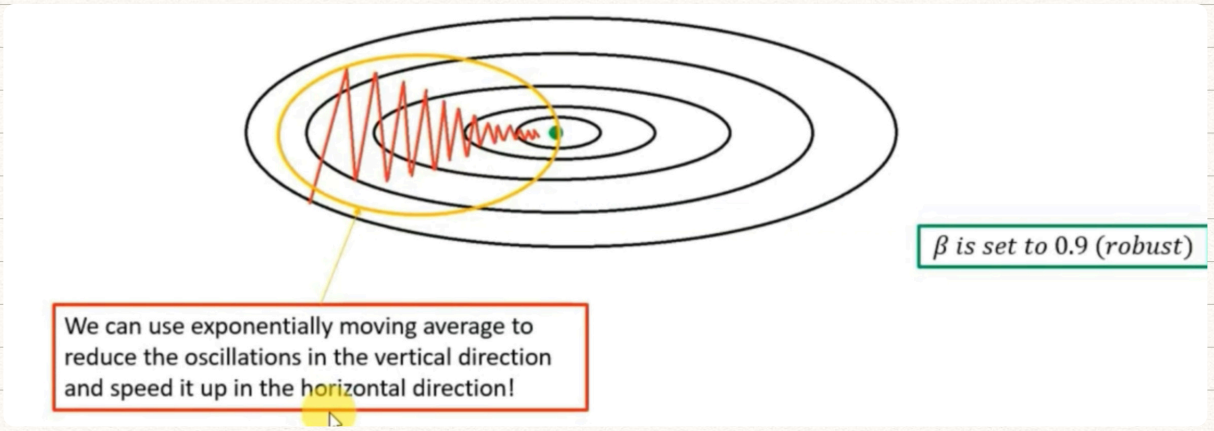
Divide v_t by $(1-\beta^t)$

Example: At $t=2$:

$$1-\beta^t = 1 - 0.98^2 = 0.0396$$
$$= \frac{v_2}{0.0369} \Rightarrow 0.494\theta_1 + 0.505\theta_2$$



Momentum



Compute dw and db on your mini-batch, and then:

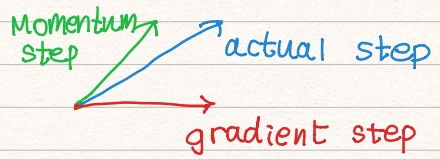
$$V_{dw} = \beta V_{dw} + (1-\beta)\, dw$$
$$V_{db} = \beta V_{db} + (1-\beta)\, db$$

Then perform weight update;

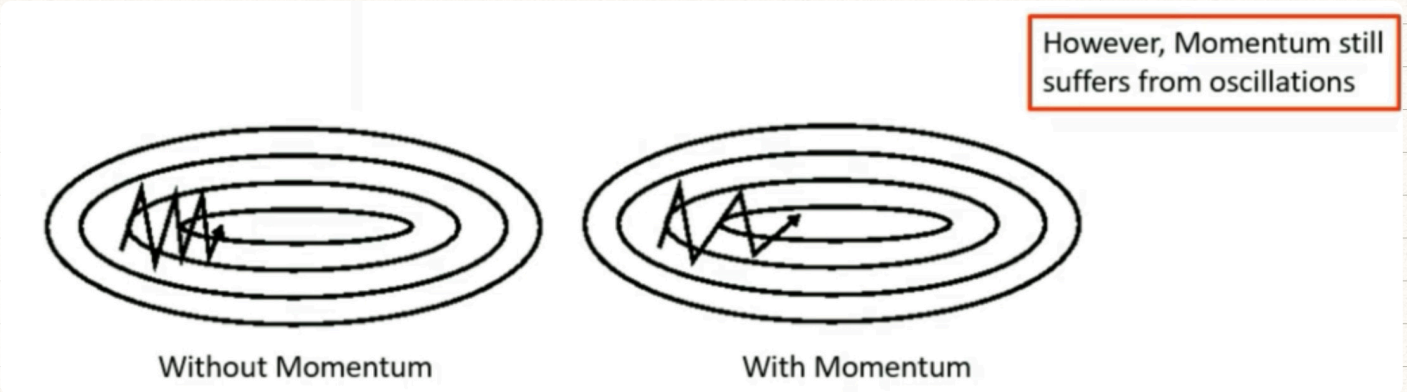
$$w = w - \alpha V_{dw}$$
$$b = b - \alpha V_{db}$$

we are updating through the exponentially weighted average.

Momentum update



It is easily accompanied with exponentially weighted average. Accelerates the SGD in the correct direction and weakens the oscilliations by adding a fraction γ of the past update vector to the new update vector. The momentum term increases for dimensions whose gradients point in the same directions and reduces update for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation. So the history is getting accumulated, and the more we accumulate in one direction, the faster the faster is in the same direction.



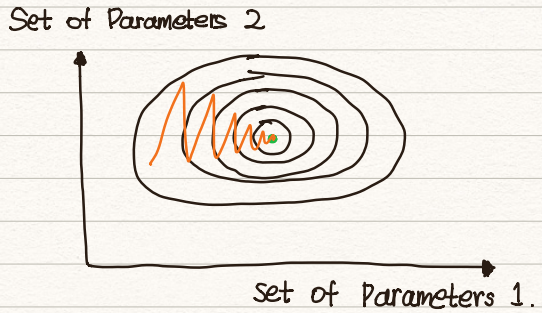
RMSProp

- An adaptive Learning Rate Method.
- RMSProp divides the learning rate by exponentially decaying average of squared gradients.
- γ is suggested to be 0.9 and $\eta = 0.001$.

Each time, the learning rate is being changed based on $E[g^2]_t$.

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

$$\theta_{t+1} = \underbrace{\boxed{\theta_t}}_{\text{old weight}} - \underbrace{\boxed{\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}}}}_{\text{Learning Rate Changing!}} g_t$$



we want to minimize the oscillation in the vertical direction.

we want to increase the speed in the horizontal direction.

Denote set of Parameters 1 as dw.

Denote set of Parameters 2 as db.

dw is small (has a small variation)

db is large (has a large variation)

$\boxed{dw^2 \text{ and } db^2 \text{ are element-wise squared.}}$

$S_{dw} = \beta S_{dw} + (1-\beta) dw^2$ small

$S_{db} = \beta S_{db} + (1-\beta) db^2$ large

ϵ is a small number to prevent division by zero.

$$w_{new} = w_{old} - \frac{\alpha}{\sqrt{S_{dw}} + \epsilon} dw$$

Dividing it by small number → gets larger

$$b_{new} = b_{old} - \frac{\alpha}{\sqrt{S_{db}} + \epsilon} db$$

Dividing it by a large number → gets smaller.

We will compute exponetially weighted average on both sets of parameters.

Adam Optimization

ADAM (= Adaptive Moment Estimation)

- Learning Rate is adaptive.
- Stores an exponentially decaying average of past square gradients (v_t)
- Stores an exponentially decaying average of past gradients (m_t)

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \leftarrow \text{Momentum}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \leftarrow \text{RMS Prop}$$

To prevent m_t and v_t from becoming zero during the initial steps, an adjustment is made:

Bias Correction

$$m_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

And finally the update

Learning Rate changing

$$\theta_{t+1} = \boxed{\theta_t} - \boxed{\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}} \boxed{\hat{m}_t}$$

Taking exponentially weighted average.

Old weights

Proposed values:

$\beta_1 = 0.9$

$\beta_2 = 0.9999$

$\epsilon = 10^{-8}$

Switching from Adam to SGD, SWATS

Despite superior training time, Adam in some areas does not converge to an optimal solution, so for some tasks (such as image classification on popular CIFAR datasets), state-of-art results are only achieved by applying SGD with momentum.

→ They proposed a simple fix which uses a very simple idea. Earlier stages of training, Adam still outperforms SDG, but later the learning saturates. They proposed simple strategy, called SWATS in which they start training deep neural network with Adam but then switch to SDG when certain criteria hits.

$$\text{SWATS:} \quad \boxed{\text{Adam}} \xrightarrow[\text{if criteria hits.}]{} \boxed{\text{SGD}}$$

Weight Decay and Regularization.

* when GSD. is applied only.

The weight update rule is given as:

$$w_i \leftarrow w_i - \eta \underbrace{\frac{\partial E}{\partial w_i}}_{\text{Gradient.}}$$

Adding the term to the cost function.

$$\tilde{E}(w) = E(w) + \frac{\lambda}{2} w^2$$

$$\boxed{\frac{\lambda}{2} \frac{d(w^2)}{w} = \frac{\lambda}{\cancel{2}}(\cancel{2}w) = \lambda w}$$

Taking the differentiation of the above equation and substituting it in the weight update rule, we end up with update rule.

$$w_i \leftarrow w_i - \eta \frac{\partial E}{\partial w_i} - \eta \lambda w_i$$

$$\boxed{\begin{aligned} \frac{d\tilde{E}}{dw} &= \frac{dE}{dw} + \lambda w \\ w &= w - \eta \left(\frac{dE}{dw} + \lambda w\right) \end{aligned}}$$

This new term coming from regularization causes the weight to decay in proportion to its size.

So we subtract a little portion of the weight at each step. hence named decay. We directly modify the weight update rule rather than modifying the loss function. We don't want to add more computations by modifying the loss when there is an easier way.

Decoupling Weight Decay.

Adding Weight Decay to Momentum and ADAM.

The SGD with momentum and weight decay (SGDW) update:

$$v_t = \gamma v_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - v_t - \boxed{\eta w_t \theta_t}$$ decoupled weight decay.

θ: weights
v: momentum

Similarly, for Adam with weight decay (AdamW) we obtain:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \underbrace{\theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t}_{\text{usual update rule.}} \boxed{- \eta \lambda \theta_t}$$ decoupled weight decay.

AMSGrad (On the Convergence of Adam) 2018

- In some cases (object recognition, machine translation), they fail to converge to an optimal solution and are outperformed by SGD with momentum.

Problems with ADAM:

- short term memory of the gradients becomes an obstacle.
- Exponential averaging diminishes their influence, which leads to poor convergence.

Solution.

- Since the values of the step size are often decreasing over time, they proposed a fix of keeping the maximum of values V and use it instead of the moving average to update parameters.

AMSGrad uses the maximum of past squared gradients v_t rather than the exponential average to update the parameters.

v_t is defined the same as in Adam:

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

Instead of using v_t (or its bias-corrected version $\hat{v}_t$) directly, we now employ the previous v_{t-1} if it is larger than the current one.

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

The full AMSGrad update without bias-corrected estimates can be seen as following:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} m_t$$

Hyperparameter Tuning and Learning Rate Scheduling

Hyperparameter Tuning

- choosing the optimization settings
  - Batch size
  - Learning rate
  - scheduling the learning rate.
  - the optimizer
  - etc.
- usually settings are set manually with predetermined value before starting the training.

Hyperparameter Optimization

- To find the model hyperparameters that yield the best score on the validation set metric.

→ These are main components to minimize the error of neural network.

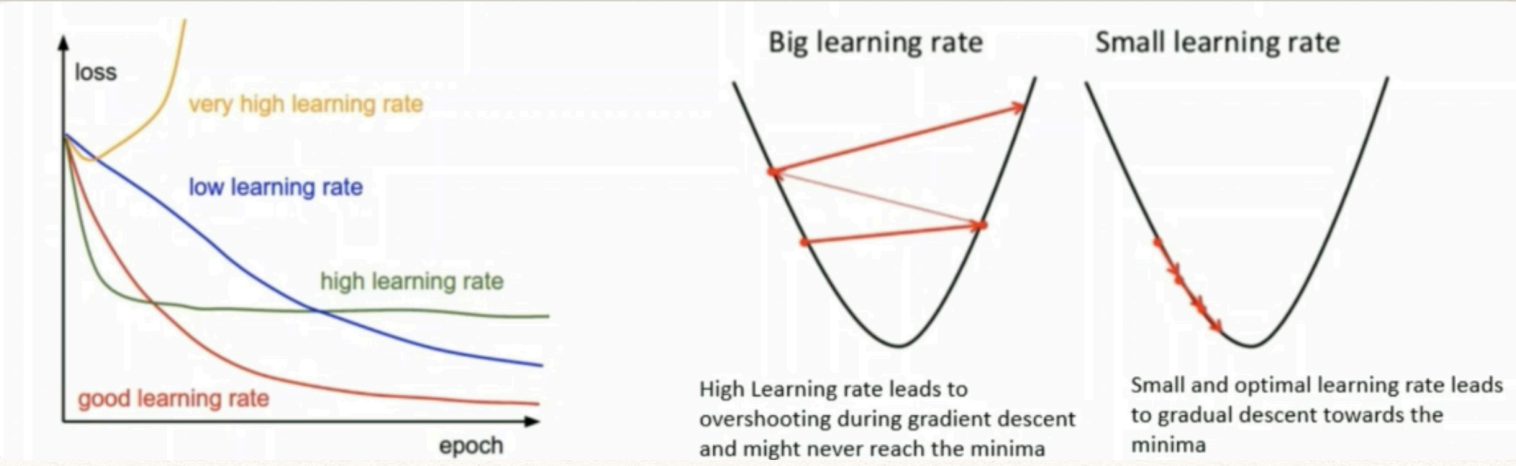
Learning Rate

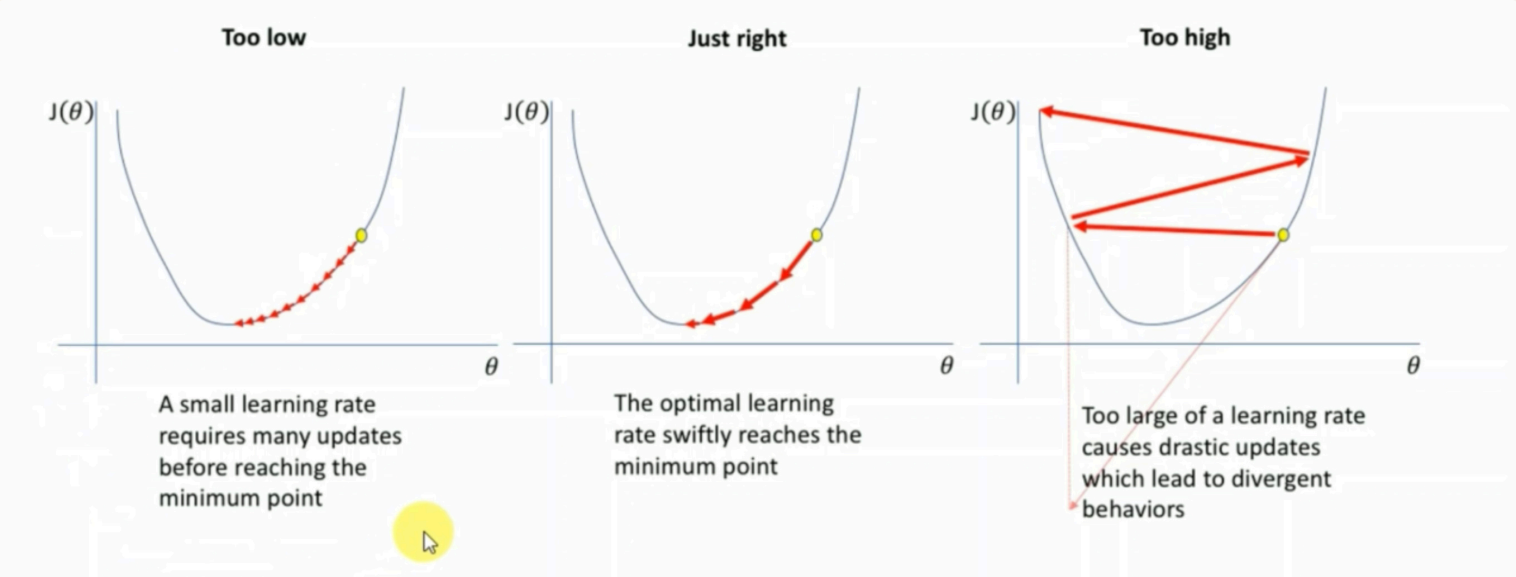
The learning rate is very high
→ Network will never learn

The learning rate is very low

Usually, a learning rate in between is good.

(0.001, 0.0001, $5e^{-4}$)





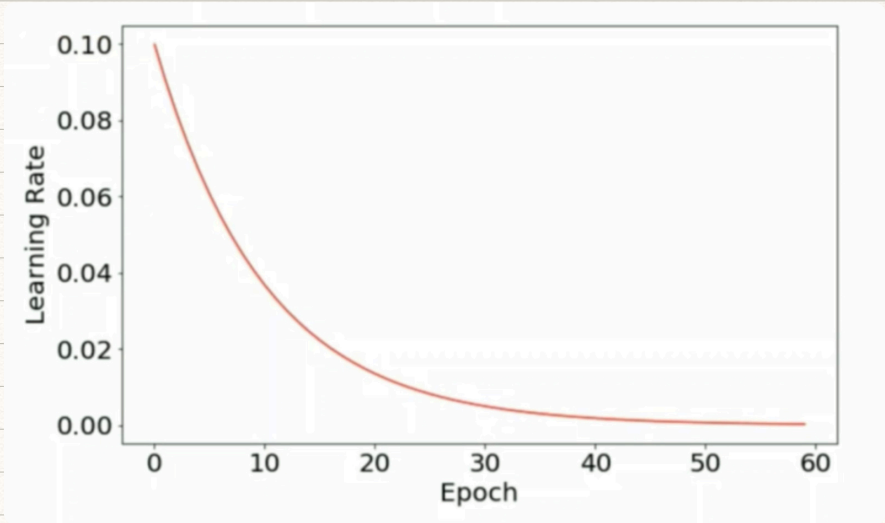
Learning Rate Scheduling

It is important that the learning rate is decayed each number of epochs.

- Decay the learning rate by a factor of 0.8 every 3 epochs.

Reason.

After some epochs of training, a neural network would have learnt something that is much better than the previous epochs. Therefore, we want to reserve these weights as much as possible. If we want to use constant learning rate. these good weights will keep on getting overwritten. Instead, we want to avoid changing the values too much.



Learning Rate Scheduling.

- Reducing the learning rate during training.

Exponential Decay.

Constant Learning Rate

- Learning rate is set constant for all the training epochs and not decayed.
- The model might not converge.

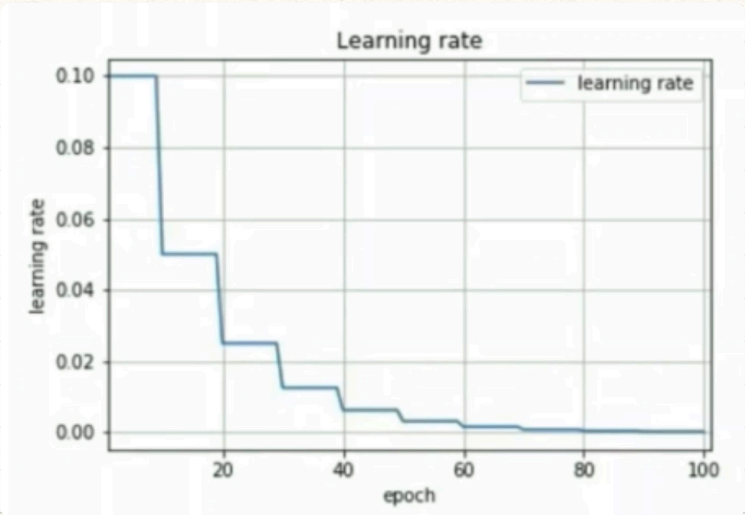
Step Decay

The new learning rate is calculated as

initial learning rate * decay factor.

decay factor = decay rate^{fraction}

fraction = epoch - start_decay_epoch // decay_every



In this case, it decays on every 10 epochs.

Example

initial_lr = 0.001, decay rate = 0.8, start decay at epoch 0, decay every 3

- **Epoch 0**: 0//3 = 0, fraction = 0, decay factor = 1, new lr = 0.001 * 1 = 0.001
- **Epoch 1**: 1//3 = 0, fraction = 0, decay factor = 1, new lr = 0.001 * 1 = 0.001
- **Epoch 2**: 2//3 = 0, fraction = 0, decay factor = 1, new lr = 0.001 * 1 = 0.001
- **Epoch 3**: 3//3 = 1, fraction = 1, decay factor = 0.8, new lr = 0.001 * 0.8 = 0.0008
- **Epoch 4**: 4//3 = 1, fraction = 1, decay factor = 0.8, new lr = 0.001 * 0.8 = 0.0008
- **Epoch 5**: 5//3 = 1, fraction = 1, decay factor = 0.8, new lr = 0.001 * 0.8 = 0.0008
- **Epoch 6**: 6//3 = 2, fraction = 2, decay factor = 0.64, new lr = 0.001 * 0.64 = 0.00064
- **Epoch 7**: 7//3 = 2, fraction = 2, decay factor = 0.64, new lr = 0.001 * 0.64 = 0.00064
- **Epoch 8**: 8//3 = 2, fraction = 2, decay factor = 0.64, new lr = 0.001 * 0.64 = 0.00064
- **Epoch 9**: 9//3 = 3, fraction = 3, decay factor = 0.512, new lr = 0.001 * 0.512 = 0.000512
- **Epoch 10**: 10//3 = 3, fraction = 3, decay factor = 0.512, new lr = 0.001 * 0.512 = 0.000512

Python Implementation

```
# Decay the learning rate every 3 epochs by a factor of 0.8
decay = 0
for epoch in range(num_epochs):
    if (epoch+1) % 3 == 0:
        decay += 1.
        new_lr = initial_lr * (0.8 ** decay)
```

Pytorch Library Implementation.

CLASS `torch.optim.lr_scheduler.StepLR(optimizer, step_size, gamma=0.1, last_epoch=-1)` [SOURCE]

Decays the learning rate of each parameter group by gamma every step_size epochs. Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler. When last_epoch=-1, sets initial lr as lr.

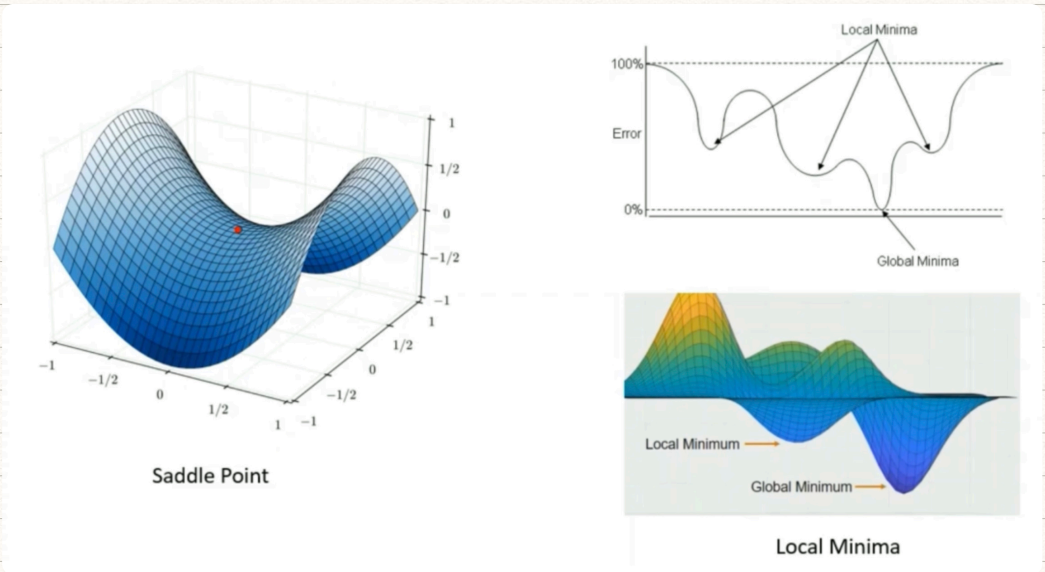
Parameters

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **step_size** (*python:int*) – Period of learning rate decay.
- **gamma** (*python:float*) – Multiplicative factor of learning rate decay. Default: 0.1.
- **last_epoch** (*python:int*) – The index of last epoch. Default: -1.

Example

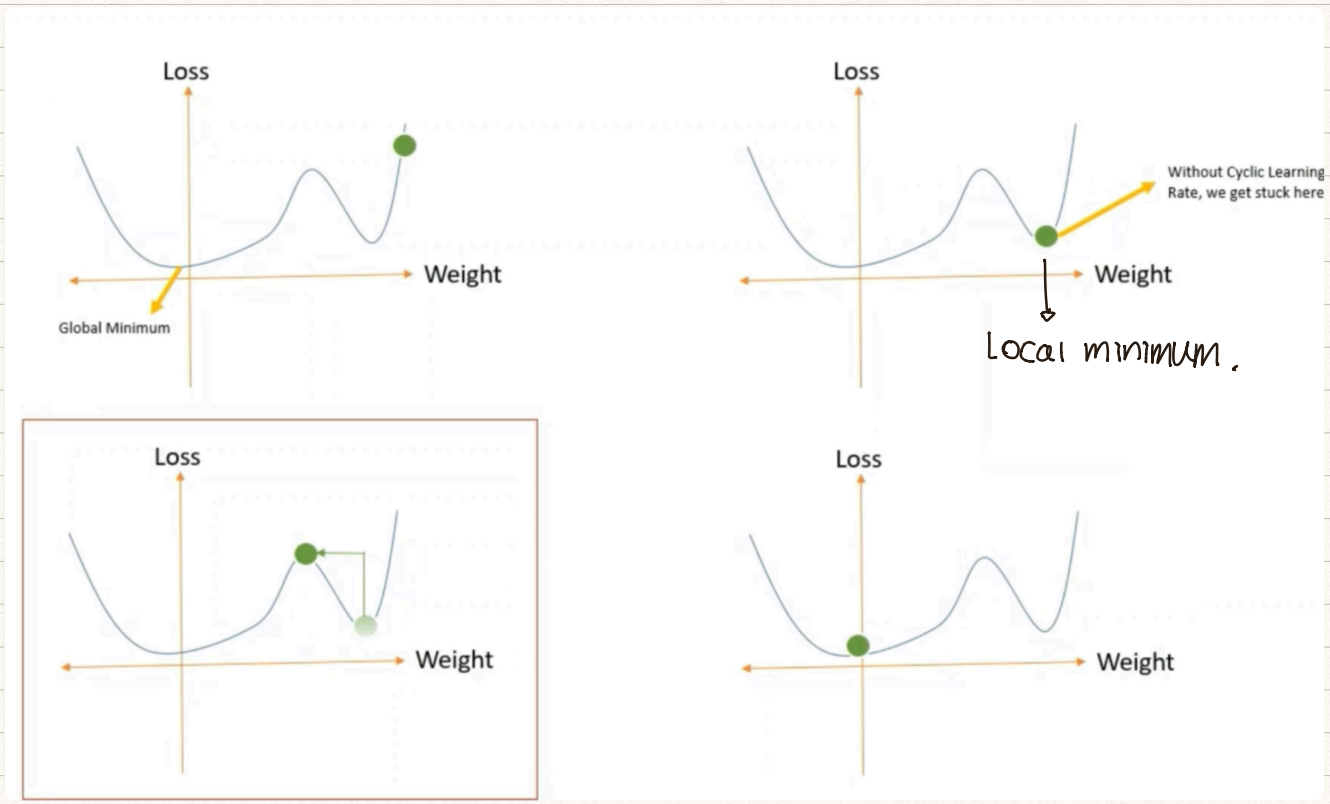
```
>>> # Assuming optimizer uses lr = 0.05 for all groups
>>> # lr = 0.05     if epoch < 30
>>> # lr = 0.005    if 30 <= epoch < 60
>>> # lr = 0.0005   if 60 <= epoch < 90
>>> # ...
>>> scheduler = StepLR(optimizer, step_size=30, gamma=0.1)
>>> for epoch in range(100):
>>>     train(...)
>>>     validate(...)
>>>     scheduler.step()
```

Cyclic Learning Rate and Restarts



Saddle Point.
- Where your gradient point is zero.

Our object is to reach the global minimum.

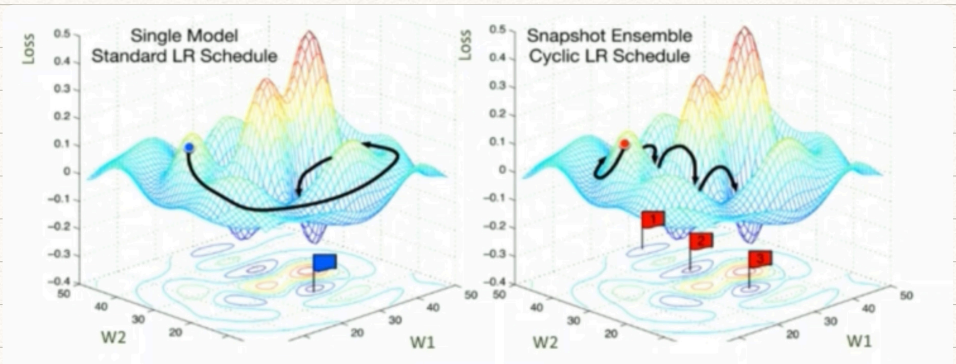


↳ The only way to reach this point is by increasing the learning rate.

Cyclic Learning Rate

The learning rate changes in a cyclical way (increases and decreases back within two bounds, periodically).

- We can escape any saddle points or local minima.
- If saddle point happens to be elaborated plateau, lower learning rates will probably never generate enough gradient to escape, resulting in difficulty in minimizing the loss.
- The restart will help to escape saddle points or local minima.
- If we pick improper learning rate, our model might get stuck from the start.



Example of loss curve for 2 parameters W1 and W2.

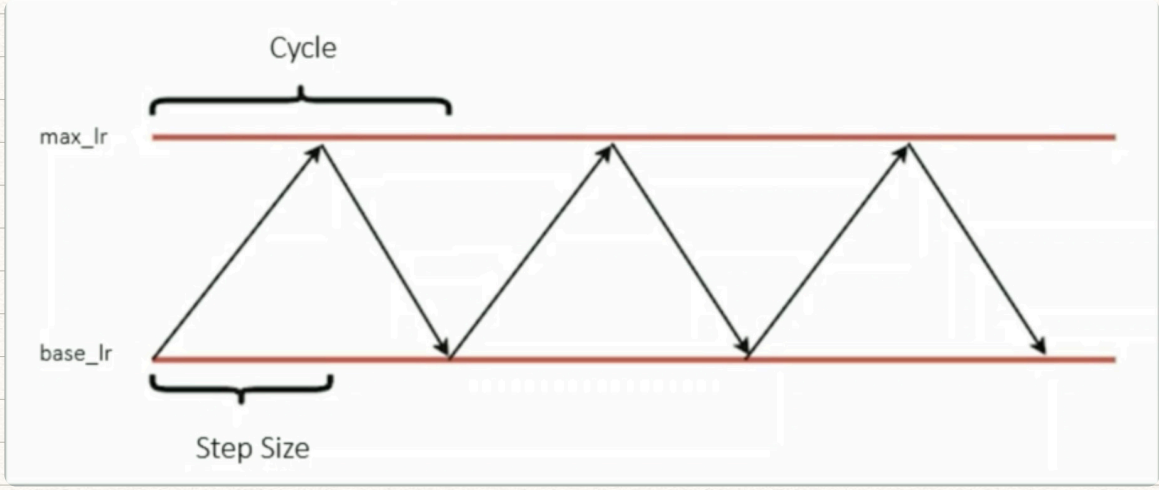
Parameters for using the Cyclic Learning Rate

- We need to set the bounds between which learning rate will vary:
  - base learning rate
  - maximum learning rate.

Cycle: Number of iterations we want for our learning rate to go from lower bound to upperbound, and then back to lower bound.

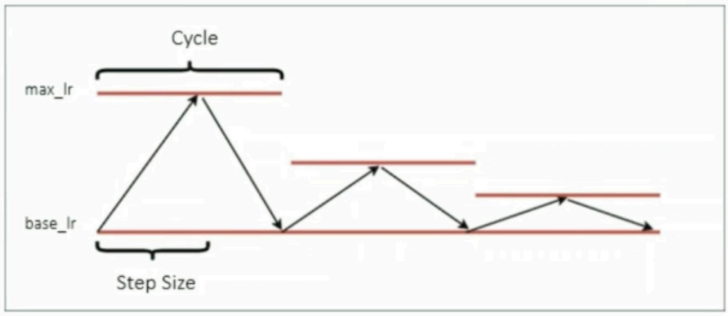
Step size: Number of iterations to complete half of a cycle.

Once the maximum learning rate is hit, we then decrease the learning rate back to base learning rate, which takes a half cycle.

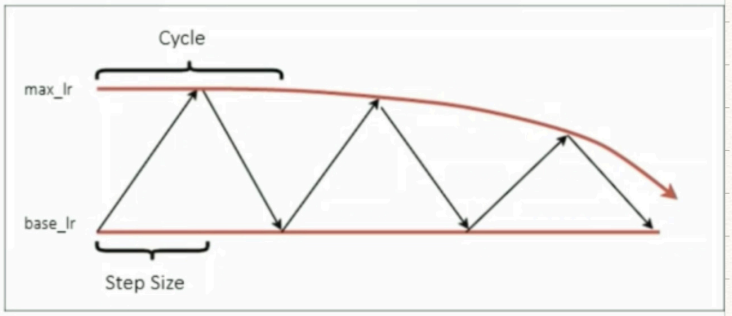


Variations.

Triangular2 : Here the *max_lr* is halved after every cycle.

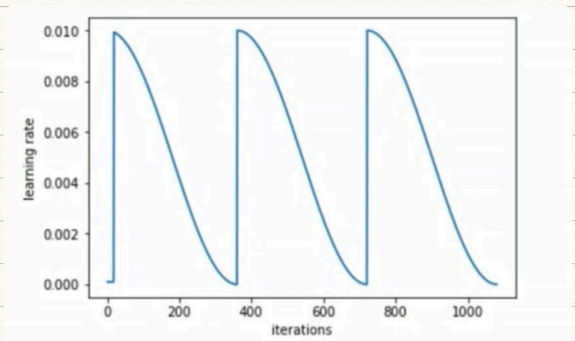


Exponential Range : Here *max_lr* is reduced exponentially



Cosine Annealing with Warm Restarts

Follows the same concept as cyclic learning rate, but anneals the learning rate using a cosine function.



$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_i}\pi\right)\right)$$

when $T_{cur} == T_i$
$\Rightarrow \eta_t = \eta_{min}$.

$$\boxed{\begin{array}{l}\cos(\pi) = -1 \\ \cos(0) = 0\end{array}}$$

when $T_{cur} == 0$ after restart
$\Rightarrow \eta_t = \eta_{max}$

η_{max} = Initial / Maximum learning rate.

η_{min} = Minimum learning rate.

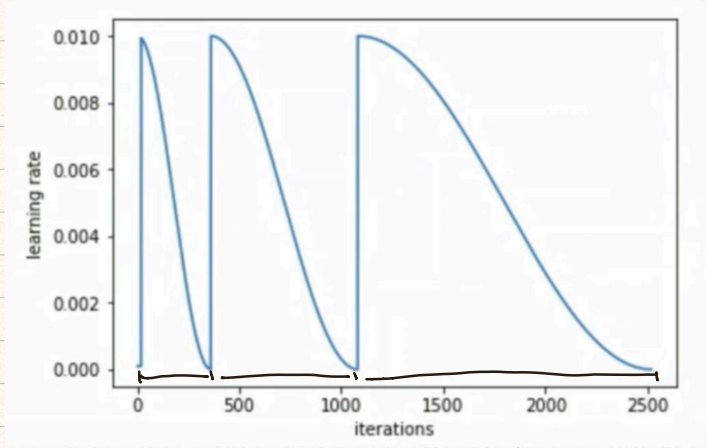
T_{cur} = number of epochs since the last restart

T_i = number of epochs between two warm restarts in SGDR

T_{cur} is updated at each batch iteration, so it can take discredited values such as 0.1, 0.2... etc.

- We will most likely get closer to global minimum the more iterations we do through our dataset. Therefore instead of restarting every epoch, we can lengthen the number of epochs until restart.

Example: first cycle will decrease over 1 epoch, the second 2 epochs and the third over 4 epochs, etc.

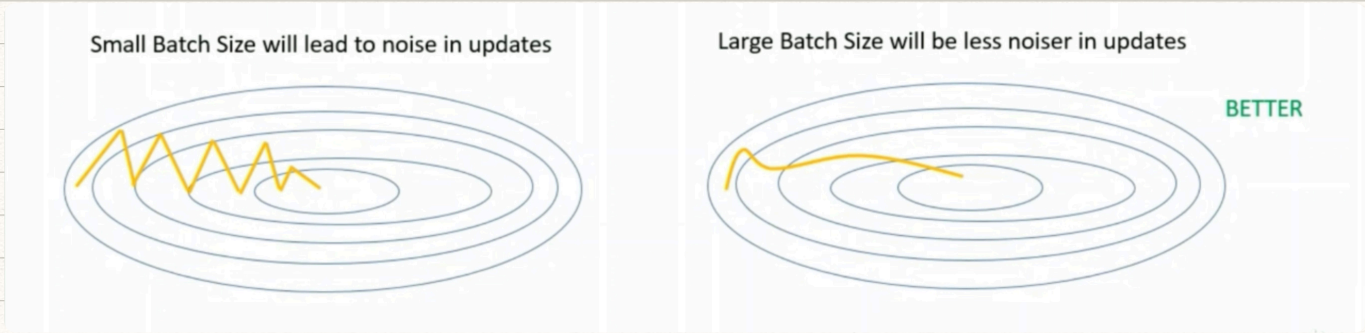


Effect of Batch Size in the Training Accuracy

The batch size is highly dependent on the dataset and the application. The larger the batch size, in general, the better performance there is.

Why?

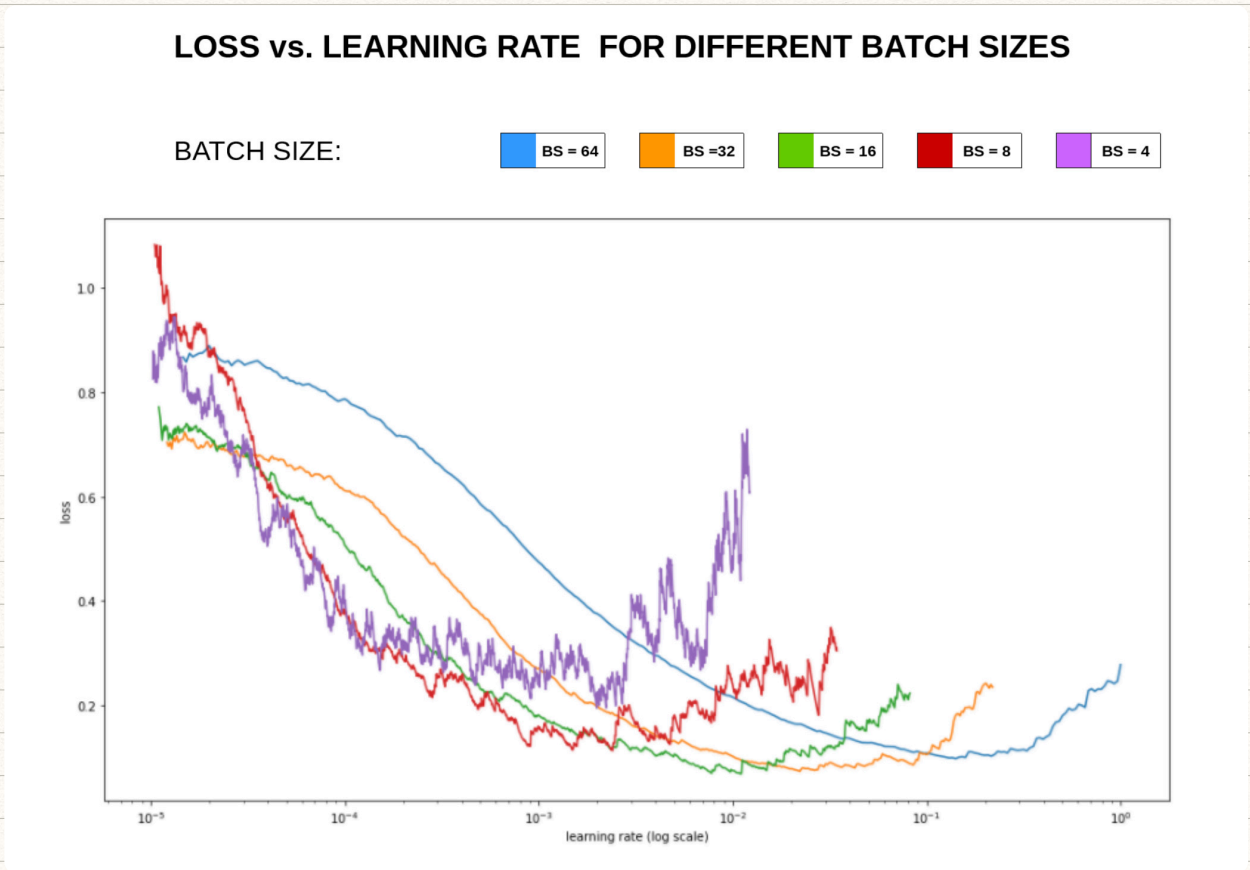
The bigger the batch size is, the less noisier are your updates. By using a large batch size, they can be ignored since other samples will be more dominant. Therefore, bigger batch size means more confidence in the direction of the descent of the error surface while the smaller a batch size is the closer you are to "stochastic" descent (batch size 1)



Batch Size vs Learning Rate

If we increase the batch size, we should also increase the learning rate.

Generally: when multiplying the batch size by k, one should multiply the learning rate by $\sqrt{k}$ to keep the variance in the gradient expectation constant.



Weights Initialization in Neural Network

Intialize all weights to 0?

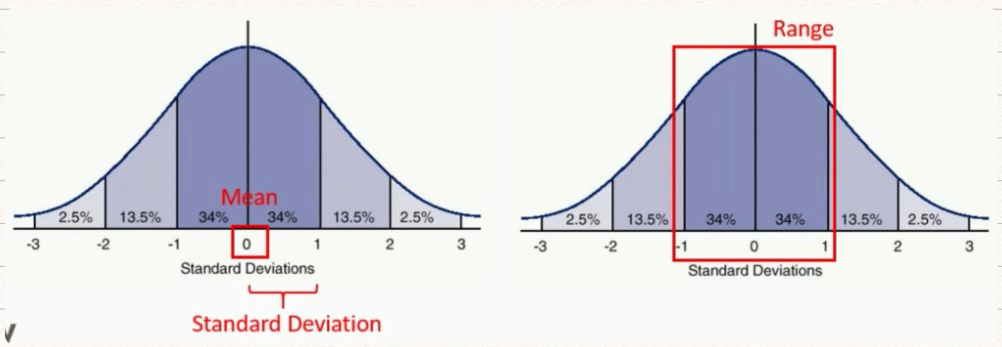
While Backpropagation is performed, the updates to the weights is dependent on the previous weights. And since all of our weights. And since all of our weights are 0, we will never be able to update these weights and no learning will occur. If you remember the chain rule, the net input value coming to the neuron is input * weight. Then the result becomes all zeroes.

Random Numbers?

- close, but not correct.
- Gradients might vanish and explode with just random thrown numbers. small gradients lead to smaller gradients while backpropagation, as well as large ones lead to larger gradients.
- Different outputs will have different ranges.

Solution

- Initialize with specific range.
- Take them from a normal distribution with a mean of zero and a standard deviation of 1.



If all weights are equal, then, no learning will occur.

$$z_1 = x_1 w_{11} + x_2 w_{12}$$
$$z_2 = x_1 w_{21} + x_2 w_{22}$$
$$a_1 = \sigma(z_1)$$
$$a_2 = \sigma(z_2)$$
$$z = a_1 w_1 + a_2 w_2$$
$$a = \sigma(z)$$

Weight Update for w_1 :

$$\frac{dE}{dw_1} = \frac{dE}{da}\frac{da}{dz}\frac{dz}{dw_1}$$

Weight Update for w_2 :

$$\frac{dE}{dw_2} = \frac{dE}{da}\frac{da}{dz}\frac{dz}{dw_2}$$

If we initialize with 0s

$$\left.\begin{aligned} z_1 &= x_1 w_{11} + x_2 w_{12} = x_1(0) + x_2(0) = 0 \\ z_2 &= x_1 w_{21} + x_2 w_{22} = x_1(0) + x_2(0) = 0 \end{aligned}\right\} \text{Same Inputs}$$

$$\left.\begin{aligned} a_1 &= \sigma(0) = 0.5 \\ a_2 &= \sigma(0) = 0.5 \end{aligned}\right\} \text{Same Activations}$$

$$z = a_1 w_1 + a_2 w_2 = a_1(0) + a_2(0) = 0$$
$$a = \sigma(0) = 0.5$$

↳ feed forward.

Back propagation

$$\frac{dE}{dw_1} = \frac{dE}{da}\frac{da}{dz}\frac{dz}{dw_1} \qquad \frac{da}{dz} = a(1-a) = 0.5(0.5) = \boxed{0.25}$$

Same

$$\frac{dE}{dw_2} = \frac{dE}{da}\frac{da}{dz}\frac{dz}{dw_2} \qquad \frac{da}{dz} = a(1-a) = 0.5(0.5) = \boxed{0.25}$$

$$\frac{dE}{dw_1} = \frac{dE}{da}\frac{da}{dz}\frac{dz}{dw_1} \qquad \frac{dz}{dw_1} = a_1 = \boxed{0.5}$$

Same

$$\frac{dE}{dw_2} = \frac{dE}{da}\frac{da}{dz}\frac{dz}{dw_2} \qquad \frac{dz}{dw_2} = a_2 = \boxed{0.5}$$

$$\frac{dE}{dw_1} = \frac{dE}{da}\frac{da}{dz}\frac{dz}{dw_1} \qquad \frac{dE}{da} = -(expected - output) = -(1 - 0.5) = \boxed{-0.5}$$

Same

$$\frac{dE}{dw_2} = \frac{dE}{da}\frac{da}{dz}\frac{dz}{dw_2} \qquad \frac{dE}{da} = -(expected - output) = -(1 - 0.5) = \boxed{-0.5}$$

assume expected = 1

$$\frac{dE}{dw_1} = \frac{dE}{dw_2}$$

Both weight will get updated with the same value and they will be equal.

It's like trying the weights on each other.

Xavier initializer

Normal initialization is good. But we need to be able to control the variance of the output.

- The objective is to have weights that can produce outputs that follow a similar distribution across all neurons. This will help convergence during training and train quickly and efficiently.

Step-by-Step walk through:

The outputs of a linear layer (fully connected layer):

$$Y = W_1 X_1 + W_2 X_2 + \cdots + W_n X_n$$

We want the weights to remain in specific range, so we expect them to have the same variance as the inputs:

$$Var(Y) = Var(X_i) \rightarrow nVar(W_i)\cancel{Var(X_i)} = \cancel{Var(X_i)}$$

The variance of the product of two variables:

$$Var(W_i X_i)$$
$$= E[X_i]^2 Var(W_i) + E[W_i]^2 Var(X_i) + Var(W_i) Var(X_i)$$

Since the mean (E) of the weights and inputs are zero, the previous equation simplifies to:

$$Var(W_i X_i) = Var(W_i) Var(X_i)$$

If X_i and W_i are all independently and identically distributed:

$$Var(Y) = Var(W_1 X_1 + W_2 X_2 + \cdots + W_n X_n)$$
$$= n Var(W_i) Var(X_i)$$

To satisfy the condition: $Var(Y) = Var(X_i)$

Then the term $nVar(W_i)$ should be equal to 1.

$$nVar(W_i) = 1.$$

That means: $Var(W_i) = \frac{1}{n} = \frac{1}{n_{in}}$

If we go through the same steps for the backpropagated signal, then also: $Var(W_i) = \frac{1}{n_{out}}$

Compromising: $Var(W_i) = \frac{2}{n_{in} + n_{out}} \rightarrow std = \sqrt{Var} = \sqrt{\frac{2}{n_{in} + n_{out}}}$

if n_{in} and n_{out} are equal, it is good, but if they are not equal, we need to find the compromising value, such as average.

Harmonic Mean

Where n_{in} in the number of input neurons/weights coming to the neuron and n_{out} is number of output neurons/weights going out of the neuron

Note: The initialization occurs on a per-layer basis.

HE Norm Initialization

- A ReLU (Rectifying Linear Unit) is zero for half of its input (negative input), so we need to double the size of the weight variance to keep the signal's variance constant. Therefore, we multiply our previous equation of $\frac{1}{n_{in}}$ by 2.

$$Var(W) = \frac{2}{n_{in}} \qquad std = \sqrt{Var} = \sqrt{\frac{2}{n_{in}}}$$

Where n_{in} in the number of input neurons/weights coming to the neuron.

So when to use which?

When the layers have ReLU activation
→ HE initialization.

When the layers have sigmoid activation
→ Xavier intialization.

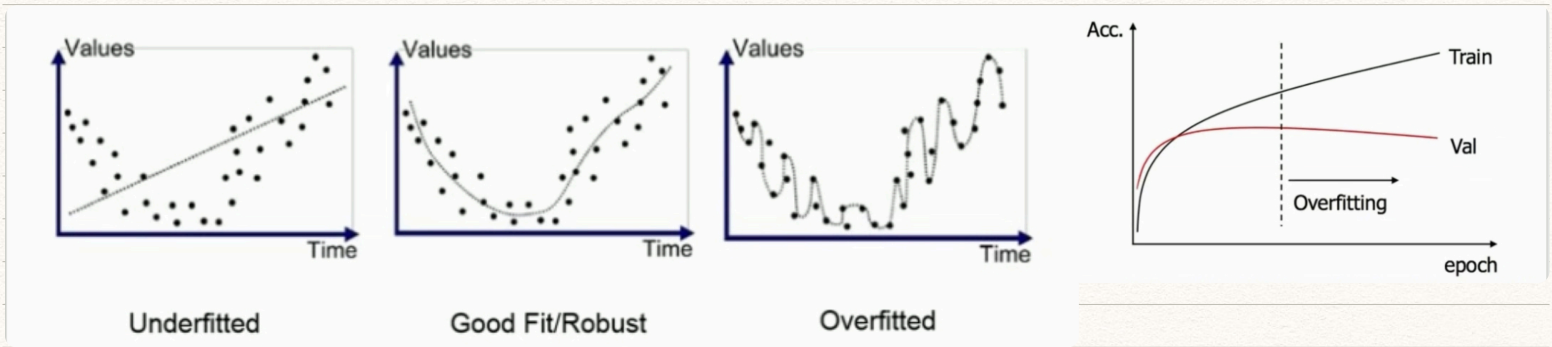
Regularization in Neural Network

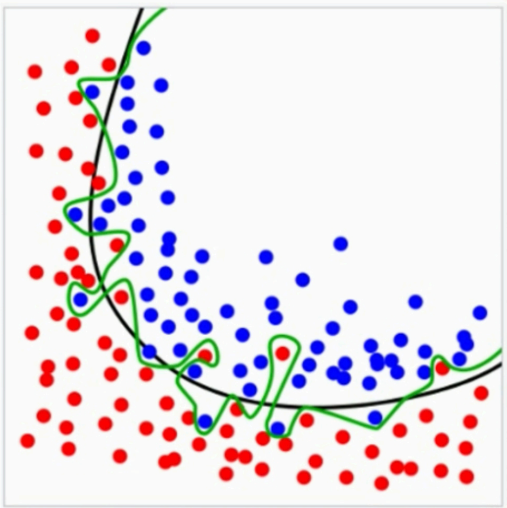
Overfitting

- A situation where your network has only learned the training data, but cannot cope up with the test data. It only performs well on the training data.

- If you get a high accuracy on the training data, but a low accuracy on the test data,

  → Your model has overfitted.





The green line represents an overfitted model and black line represents a regularized model.

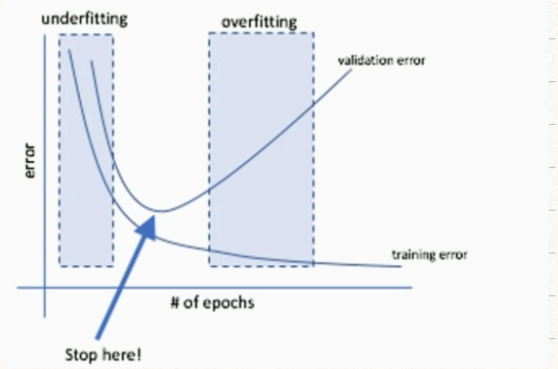
While the green line best follows the training data, it is too dependent on the data and it is likely to have a higher error rate on new unseen data, compared to the black line.

How can we reduce overfitting?

Without adding any Regularization:

- Train on more data.
- use data agumentation.
- Use early stopping:

  when validation error starts increasing, or validation error not improving.



Regularization Techniques

L1 Regularization:

adds "absolute value of magnitude" of coefficient as penalty term to the loss function.

The Loss Function: $\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j) + \lambda \sum_{j=1}^{p} |\beta_j|$

(λ) If this part is 0, we just get the error only.

L2 Regularization:

adds "squared magnitude" of coefficient as penalty term to the loss function. Here the highlighted represents L2 regularization element.

The Loss Function:

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

→ We are adding a fraction of the sum of the squared values of the weights. So larger weight values lead to larger error, and therefore the training algorithm favors and generates small weight values.

(λ) If this part equals 0, we just get the error only.

Dropout Regularization

Training time

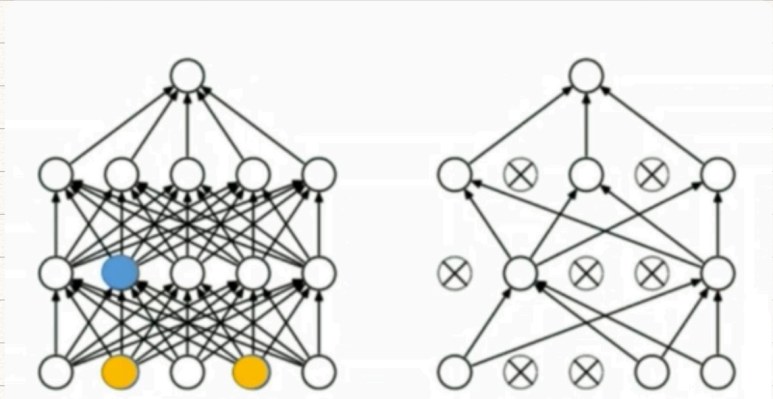
- During training of the network, neurons tend to heavily depend on each other, hence the power of each neuron is lost and the model might overfit.
- Dropout forces a neural network to learn more robust features that are useful in conjunction with many different

random subsets of the other neurons.

- Dropout roughly doubles the number of iterations required to converge. However, training time for each epoch is less, since some neurons are disabled.

In the training phase:

For a specific hidden layer, for each iteration, randomly disable a fraction, p, of neurons (and corresponding activations).



(a) Standard Neural Net (b) After applying dropout.

The Result:

Neurons don't depend on each other anymore, and hence can perform well on other data. The network is able to learn several independent correlations in the data. Therefore, overfitting is reduced.

A specific neuron might always learn to depend on one (or more) of the previous (features) in order to extract information. When we remove that feature and prevent the neuron to depend on it, it will be forced to learn the information needed from the other features. Thus, at testing time, it can perform better on unseen data since it has learned how to extract information from features that are not present at testing time.

Testing time.

consider p to be the probability that a neuron is kept

- higher probabilities mean less dropout.

At training time:

Drop the neuron activated outputs (consider $p = 0.5$)

0.2	0.3	0.9	0.7	0.4	0.1
0.4	0.7	0.4	1.1	0.6	-0.2
0.9	0.1	0.1	1.2	0.1	-0.7
0.6	0.6	0.8	0.7	0.6	-0.4

0	0.3	0.9	0	0.4	0.1
0	0	0	1.1	0	0
0	0.1	0.1	1.2	0	-0.7
0.6	0	0.8	0	0.6	0

At the test time, dropout is not applied (all inputs are considered), so we want the outputs of neurons at test time to be identical to their expected outputs at training time. otherwise, at testing time, it will be very unusual since there will be high numbers (and during training the network saw smaller values since we zeroed out some)

$$\text{neuron output (dropout)} = px + (1-p)0 = px$$

Neurons that are not kept.

0.2	0.3	0.9	0.7	0.4	0.1
0.4	0.7	0.4	1.1	0.6	-0.2
0.9	0.1	0.1	1.2	0.1	-0.7
0.6	0.6	0.8	0.7	0.6	-0.4

0.2 (0.5)	0.3 (0.5)	0.9 (0.5)	0.7 (0.5)	0.4 (0.5)	0.1 (0.5)
0.4 (0.5)	0.7 (0.5)	0.4 (0.5)	1.1 (0.5)	0.6 (0.5)	-0.2 (0.5)
0.9 (0.5)	0.1 (0.5)	0.1 (0.5)	1.2 (0.5)	0.1 (0.5)	-0.7 (0.5)
0.6 (0.5)	0.6 (0.5)	0.8 (0.5)	0.7 (0.5)	0.6 (0.5)	-0.4 (0.5)

So at test time, we must scale x by px to get the same result.

Another way to do this is using Inverted Dropout

Performs the scaling at the training time, leaving the forward pass at test time untouched. →

At training time, divide the activations by keeping probability p

0.2	0.3	0.9	0.7	0.4	0.1
0.4	0.7	0.4	1.1	0.6	-0.2
0.9	0.1	0.1	1.2	0.1	-0.7
0.6	0.6	0.8	0.7	0.6	-0.4

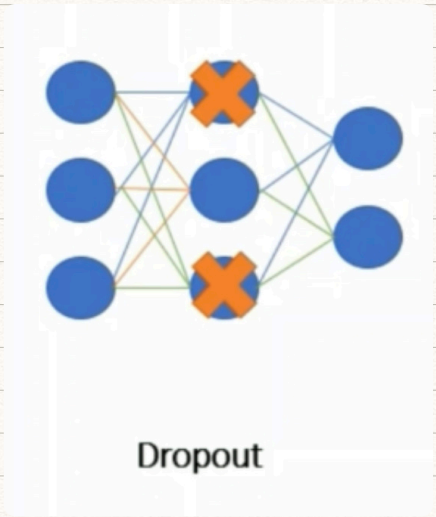
0	0.3/0.5	0.9/0.5	0	0.4/0.5	0.1/0.5
0	0	0	1.1/0.5	0	0
0	0.1/0.5	0.1/0.5	1.2/0.5	0	-0.7/0.5
0.6/0.5	0	0.8/0.5	0	0.6/0.5	0

DropBlock in CNNs

For CNN Architectures, Dropout is not effective. DropBlock is used instead.

Drop Connect

The only difference with dropout is that we drop the weights (set them to zero) rather than dropping the neurons output.



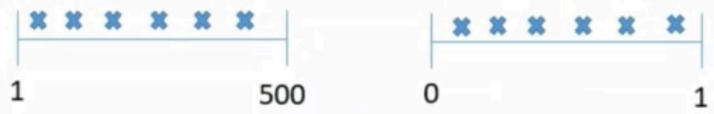
Normalization and Standardization

Normalization : setting all of the data to have the same range of the values

Standard Devaition:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

- Normalization:



Min-Max Normalization (Scales data to have values between 0-1):

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Example: [44,23,56]
44 → (44 – 23) / (56 – 23) = 0.63
23 → (23 – 23) / (56 – 23) = 0
56 → (56 – 23) / (56 – 23) = 1

- Standardization: $x = \frac{x - mean}{std}$

The data will have a mean of 0 and standard deviation of 1

Example: [44,23,56]
Mean: (44 + 23 + 56) / 3 = 41
Standard Deviation: 13.63
44 → (44 – 41) / 13.63 = 0.22
23 → (23 – 41) / 13.63 = -1.32
56 → (56 – 41) / 13.63 = 1.10

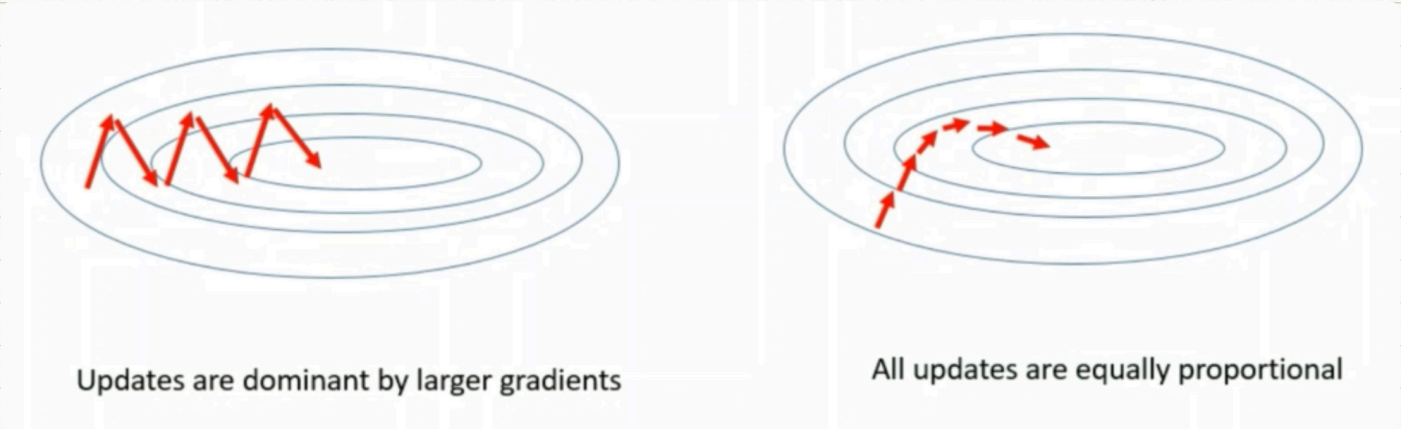
$$\sigma^2 = \frac{\Sigma(x_i - \mu)^2}{N} = \frac{(44-41)^2 + \ldots + (56-41)^2}{3} = \frac{558}{3} = 186$$

$$\sigma = \sqrt{186} = 13.638181696986$$

Batch Normalization

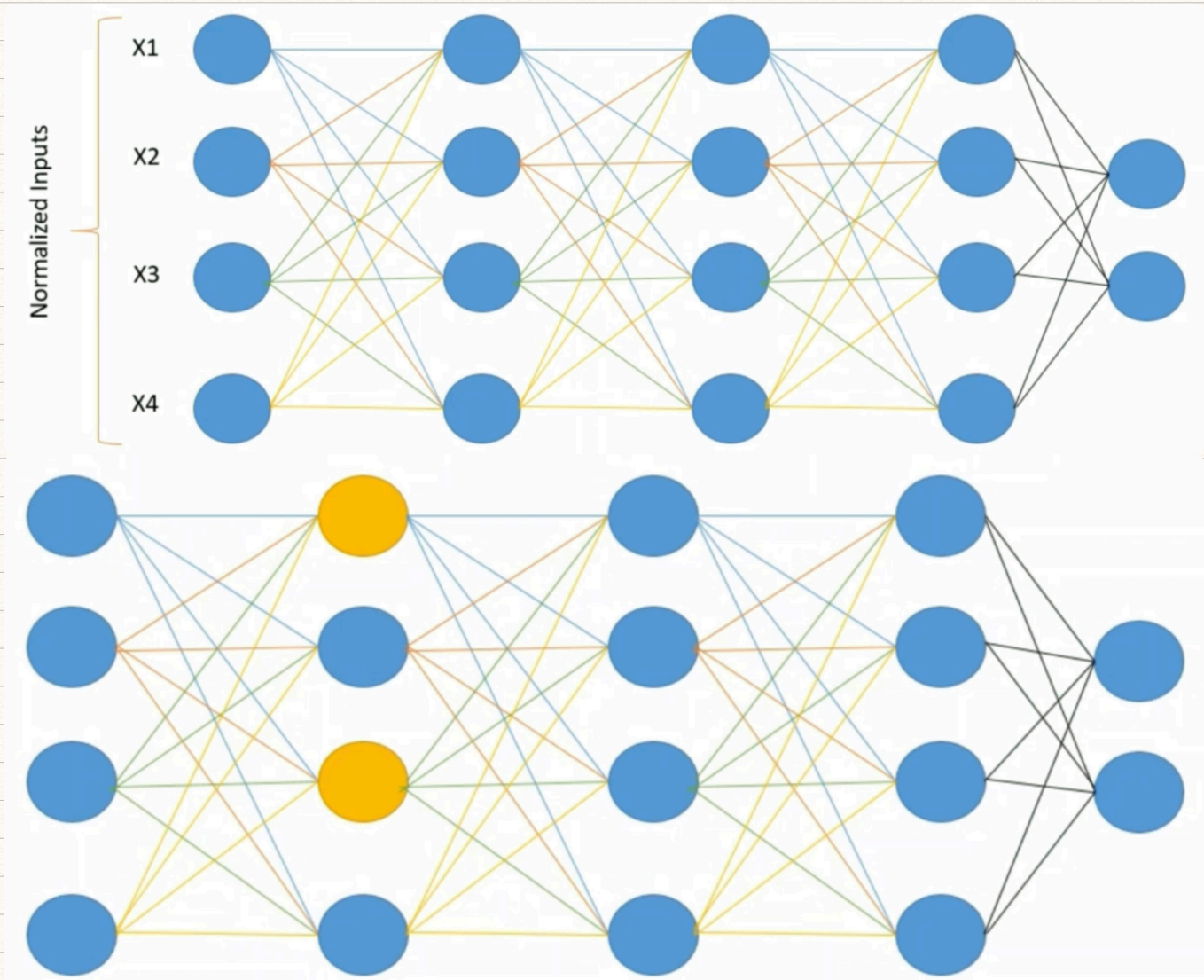
Why do we need to use either Normalization or standardization?

If we simply have a simple neural network with two inputs: The first input value varies 0 to 1 while the second input value varies from 0 to 0.01. Through a series of linear combinations and non-linear activations of a neural network, the parameters associated with each input will also exist on different scales. Therefore, the larger values will be dominant and the smaller ones will be ignored in weight update.



By normalizing all of our inputs to a standard scale, we're allowing the network to more quickly learn the optimal parameters for each input node.

Batch Normalization

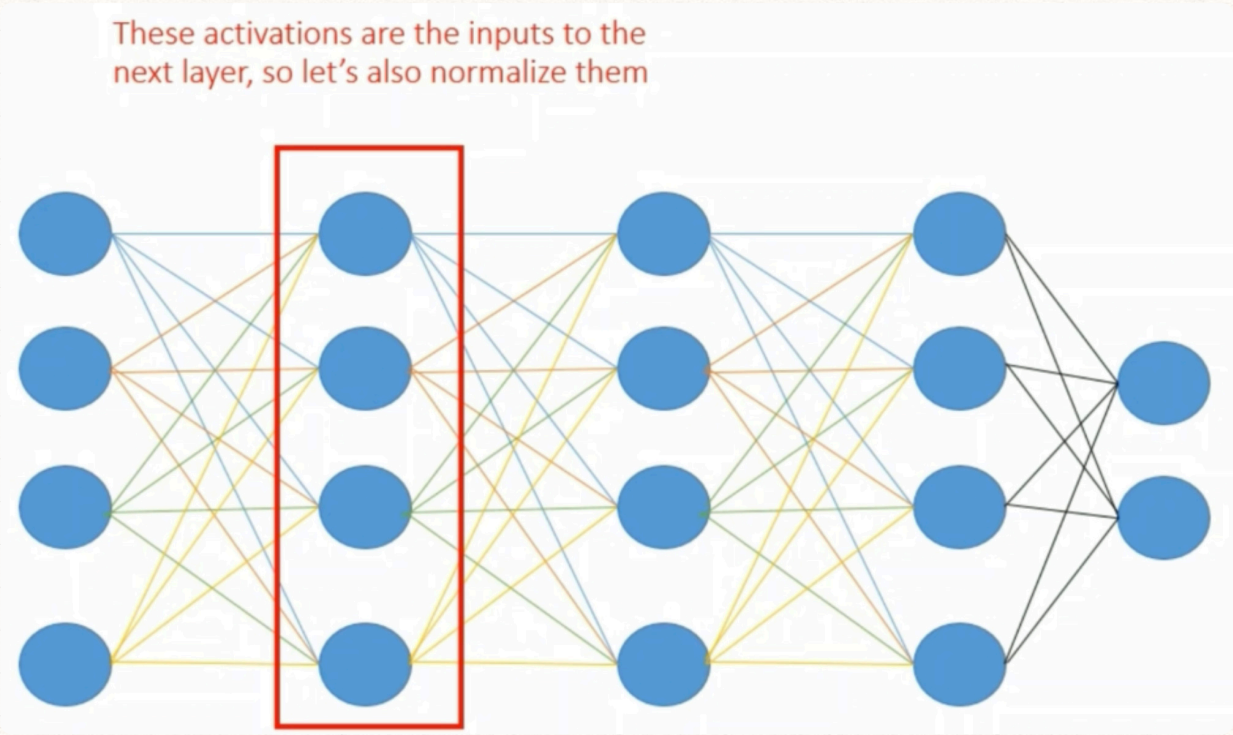


Q. what if one (or some) of the neurons have high values?

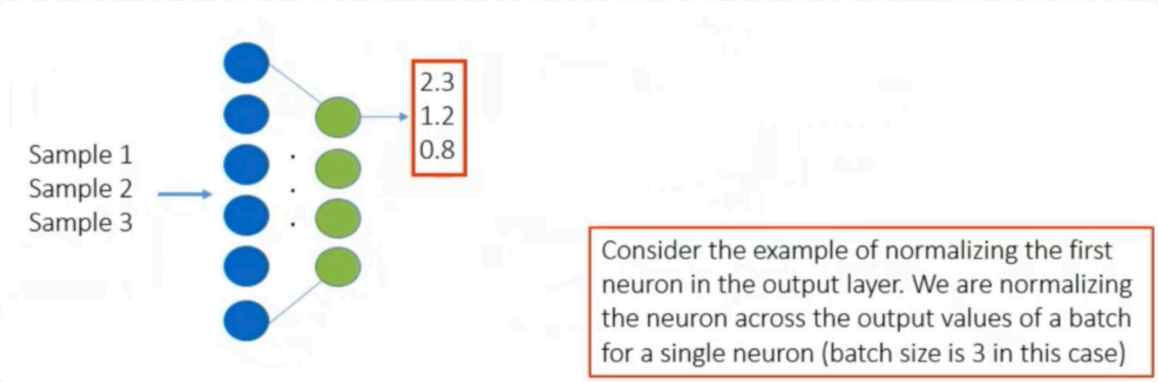
They would also cause the neurons depending on it to have a

larger value thus the network would still be unstable.

Solution



We normalize each feature (each neuron) in a layer according to the batch of samples. The mean and standard deviation are calculated across a batch of samples.



Definition

- Normalization of each output neuron in each hidden layer, rather than only for the input layer, for a mini-batch of the data. In other words, it reduces the amount that the hidden layer values shift around.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

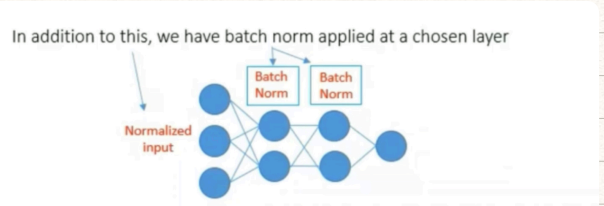
Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

https://arxiv.org/abs/1502.03167

Learnable Parameters

ϵ: Very Small number to prevent dividing by zero

If we set γ=1 and β =0
We just have standardization



The mean and standard deviation are calculated **across a batch of samples.**

We don't always want to normalize our value to have zero mean and unit variance. This would perform poorly for some activation functions (ex. Sigmoid). Thus, we learn the best distribution by scaling our normalized values by γ and shifting by β

Benefits of Batch Normalization.

- converges faster and reduces the need for dropout.
- allow the network to normalize a layer into whichever distribution is most optimal for learning (since we are learning the scaling and shifting parameters).
- Eliminates the need for a bias when batch normalization is applied to it, since we are already shifting the normalized values with the β parameter.

Layer Normalization

- we normalize each feature (each neuron) in a layer according to the features in that layer. The mean and standard deviation are calculated across the features of a layer (across the outputs of a layer).

At the output of a feedforward layer:

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

Mean over all neurons in the layer

m: the number of neurons in that layer

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

Variance over all neurons in the layer

For each neuron in the output layer:

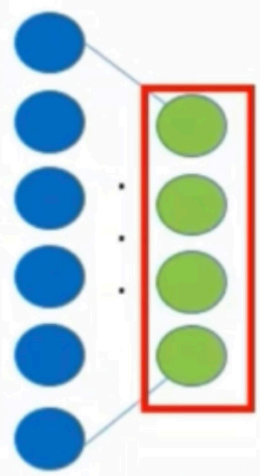
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

Normalize: subtract mean and divide by standard deviation

$$y_i \leftarrow \gamma \hat{x}_i + \beta$$

Scale by a learnable value and shift by a learnable value

Learnable Parameters

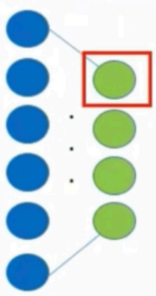


Example.

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \frac{-2.3 + 1.9 + 2.7 - 3.4}{4} = -0.275$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

$$\frac{(-2.3 + 0.275)^2 + (1.9 + 0.275)^2 + (2.7 + 0.275)^2 + (-3.4 + 0.275)^2}{4} = 6.861$$

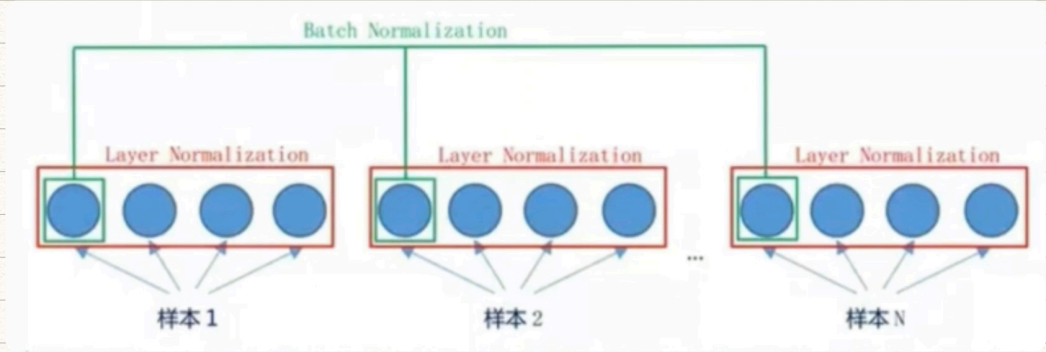


$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \frac{-2.3 + 0.275}{\sqrt{6.861 + 10^{-9}}} = -0.77$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad -0.77\theta_1 + \theta_2$$

Summary

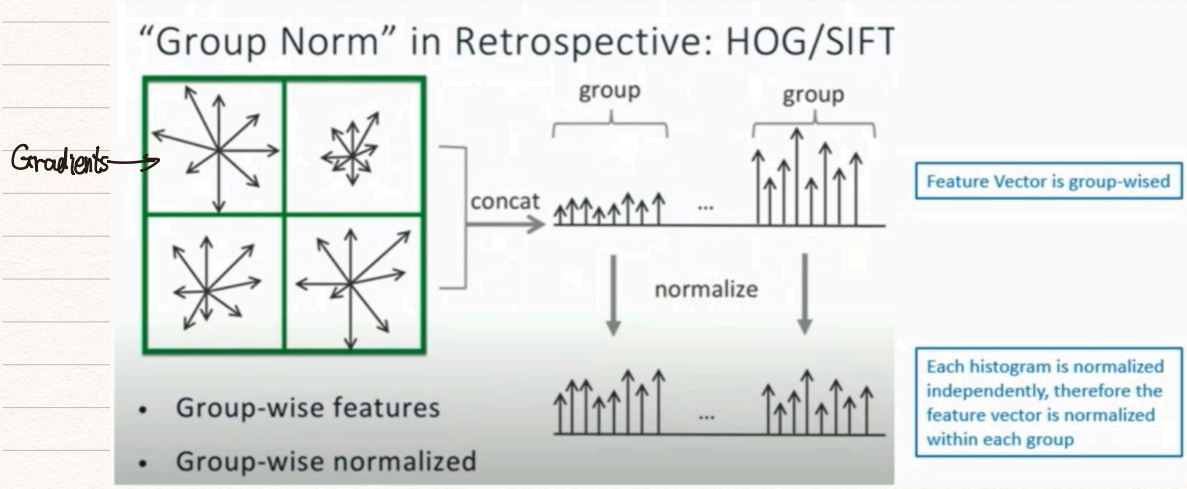
- All the normalization is calculated using: $x = \frac{x - \text{mean}}{\text{std}}$
- Batch Normalization and Layer Normalization are performed in different directions (it only differs in how mean is calculated).
- For batch normalization, input values of the same neuron from from different images in one mini batch are normalized. In layer normalization, input values for different neurons in the same layer are normalized without consideration of mini batch.



Group Normalization

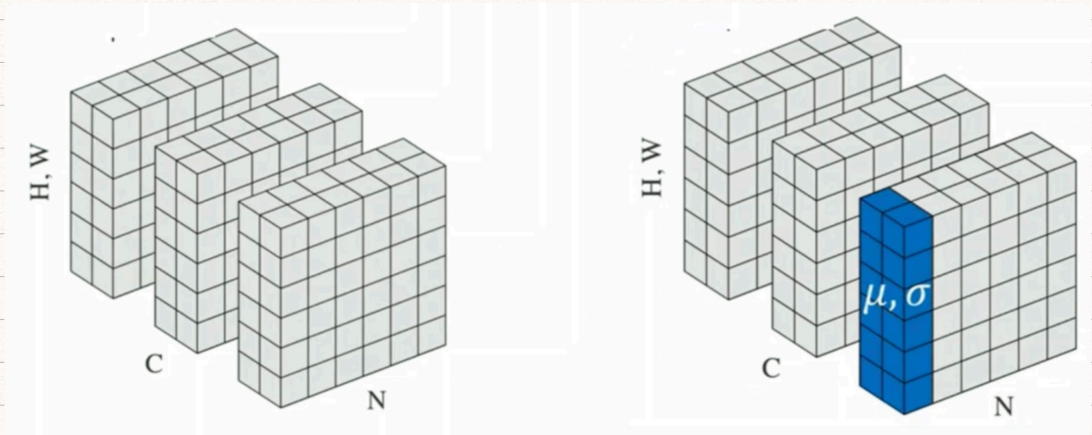
Motivation of Group Normalization.

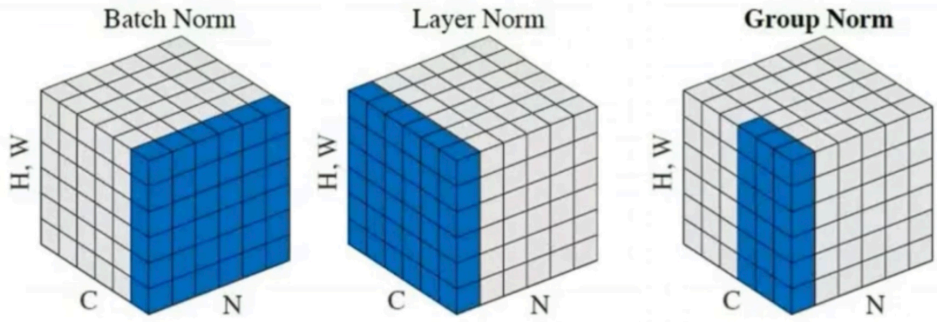
- From HOG/SIFT Feature extraction method in image processing



The deep features can have internal group-wise sub-structures

→ Normalize within the group.





$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

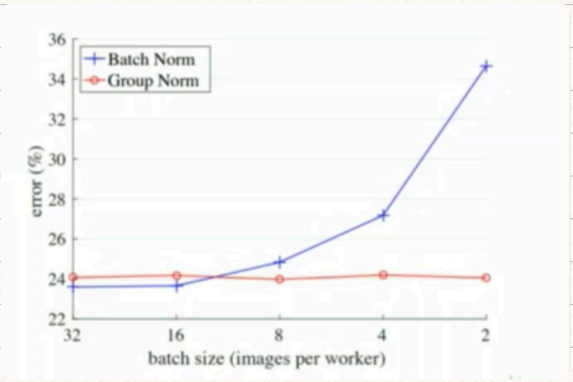
The input channels are separated into num_groups groups, each containing num_channels / num_groups channels. The mean and standard-deviation are calculated separately over the each group.

In the figure above for Group Norm: Separate 6 channels into 2 groups

If we put all 6 channels into a single group → equivalent with LayerNorm

When is group normalization more helpful than Batch Normalization?

- In Batch Normalization, if we use a small batch size, the error rate increases.
- However, this problem is avoided in Group Normalization.

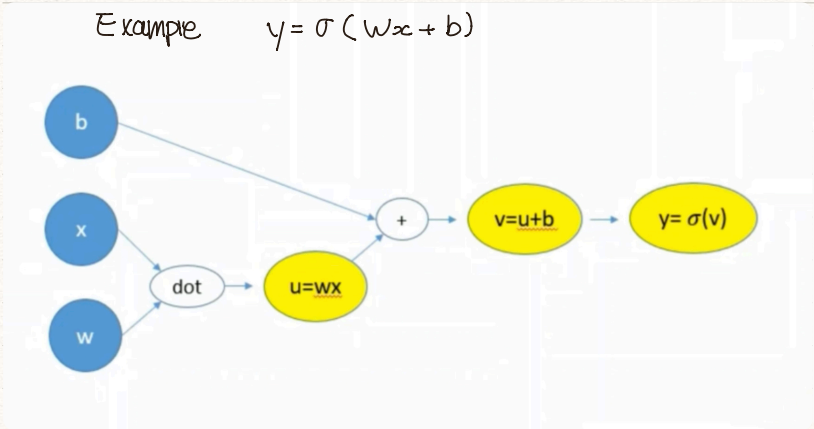


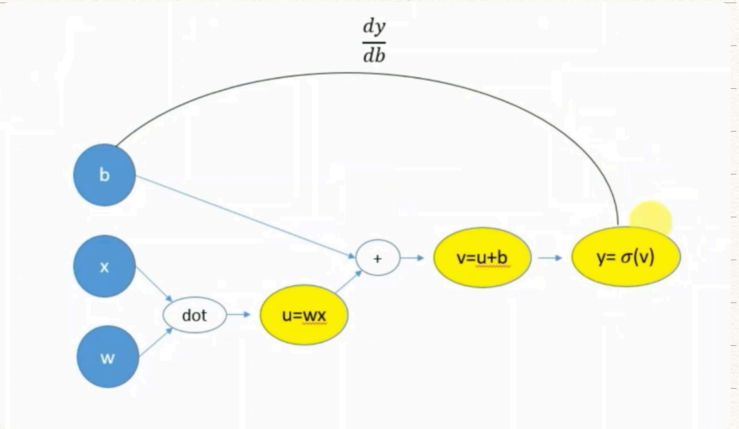
Computation Graphs and Automatic Differentiation

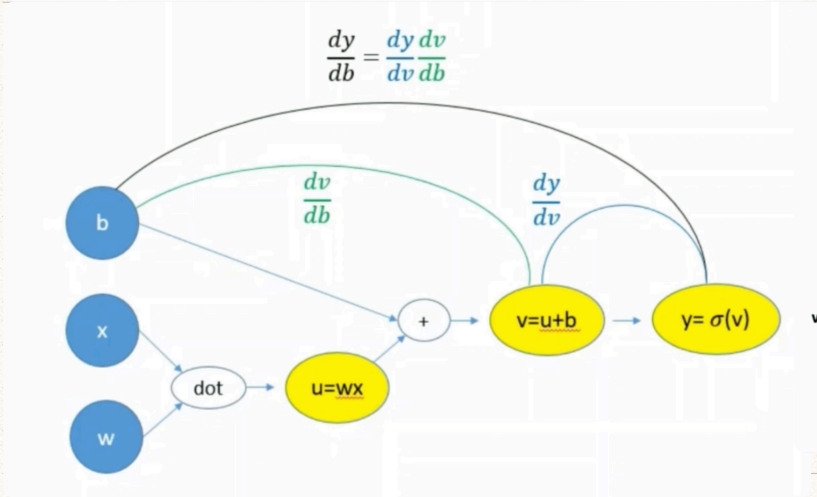
What is a computational graph?

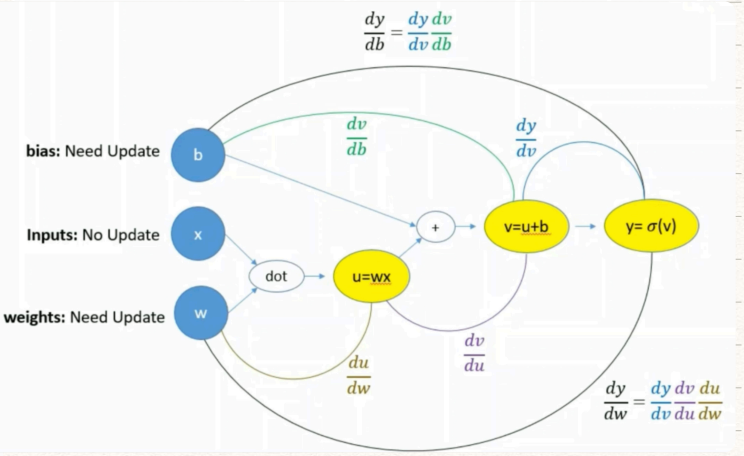
- A computational graph is a directed graph where the nodes correspond to operations or variables. The values that are fed into the nodes and come out of the nodes are called tensors (multi-dimensional matrices).

Example $y = \sigma(wx + b)$









Most deep learning frameworks use automatic differentiation, meaning that we do not need to define the derivative of a function. The frameworks will automatically derive them for gradient and back propagation computation.

Torch: A GPU programming language.

Key Idea: Uses Dynamic Computation Graphs.

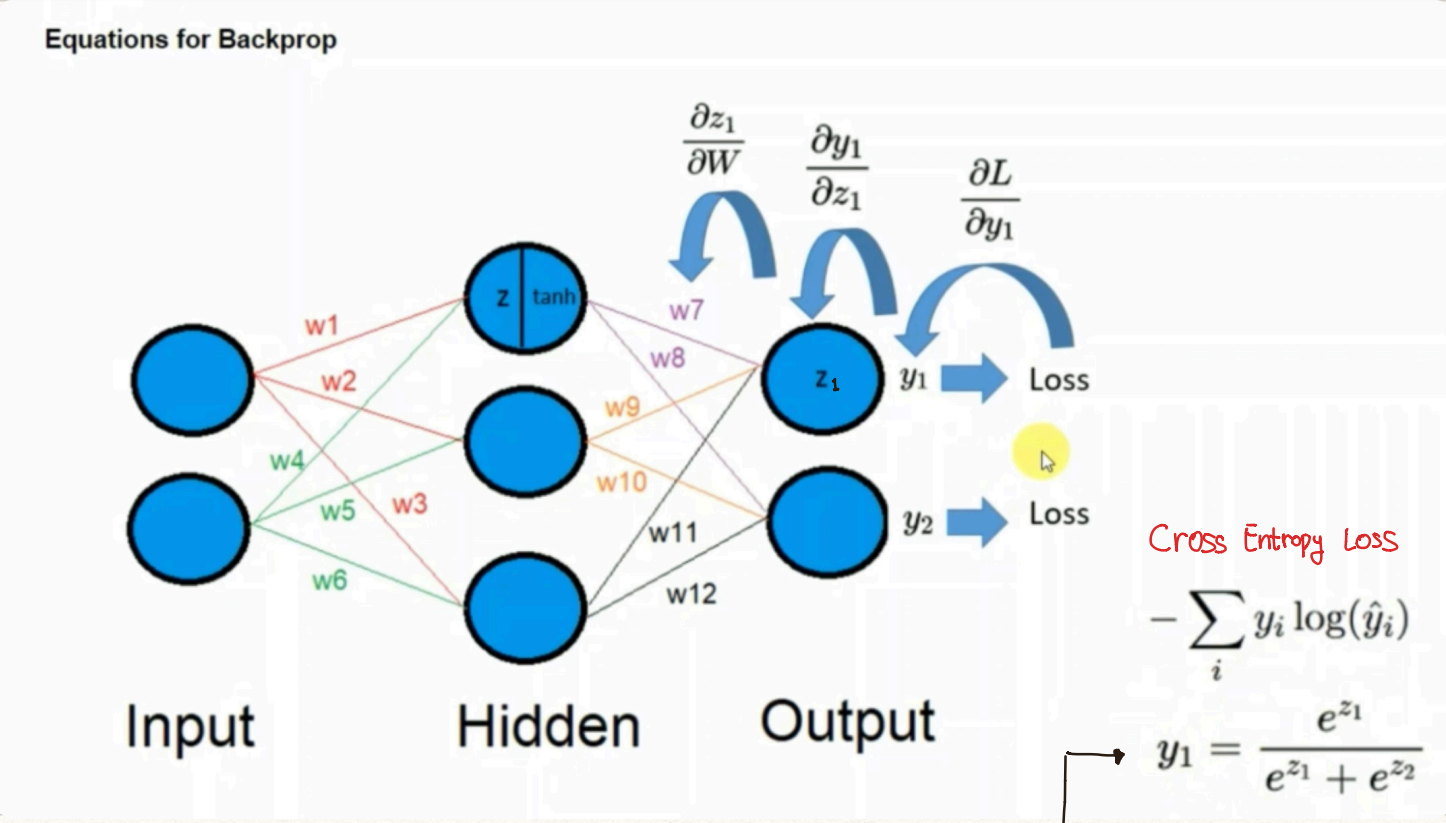
- Graph is defined on the fly, as opposed to static graphs which are defined before running the code. In static graphs, we define the whole architecture first and then compile & run it.

- This can be helpful when using RNNs when we have variable sequence length. In this case, we can process our inputs in different batch size! Also, lots of conditions to adjust training/prediction during the actual run is possible.

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```



Equation for z_1 would be

$$z_1 = a_1 w_{11} + a_2 w_{21} + a_3 w_{11}$$

Using the back propagation algorithm, we are going to update our weights.

Chain Rule.

⇒ We need to define the derivative of Loss with respect to y_1, and y_1 to z_1 and z_1 to w_{11}.
(we will be updating all the weights all using a single matrix)

$$\frac{\partial z_1}{\partial w} \cdot \frac{\partial y_1}{\partial z_1} \cdot \frac{\partial L}{\partial y_1}$$

Derivative of the loss with respect to the weights.

$$\text{Loss} = -\sum_i y_i \log(\hat{y}_i)$$

$$\frac{\partial L}{\partial y_1} = -\hat{y}_1 \left[\frac{(y_1)'}{y_1}\right] = -\hat{y}_1 \left[\frac{1}{y_1}\right] \frac{\partial y_1}{\partial z_1}$$

Actual prediction

Network prediction

Softmax function (Equation of y)

$$y_1 = \frac{e^{z_1}}{e^{z_1}+e^{z_2}}$$

$$\frac{\partial y_1}{\partial z_1} = \frac{\partial}{\partial z_1} \cdot \frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}}$$

$$= y_1(1-y_1)$$

Network prediction.

$$\frac{\partial L}{\partial y_1} = -\hat{y}_1\left[\frac{(y_1)'}{y_1}\right] = -\hat{y}\left[\frac{y_1(1-y_1)}{y_1}\right] = -\hat{y}(1-y_1) = \hat{y}_1 - y_1$$

We will just call $\delta_3 = \hat{y} - y$ that is $\frac{\partial L}{\partial y_1}$

We are now only left with derivative of z_1 with respect to weights.

$$\frac{\partial z}{\partial w} = a_1 + a_2 + a_3.$$

$$\frac{\partial L}{\partial W_2} = a_1^T \delta_3$$

$$\frac{\partial L}{\partial b_2} = \delta_3$$

$$\delta_2 = (1-\tanh^2 z_1) \circ \delta_3 W_2^T$$

$$\frac{\partial L}{\partial W_1} = x^T \delta_2$$

$$\frac{\partial L}{\partial b_1} = \delta_2$$

$$\frac{\partial z_1}{\partial W}$$