AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

Feed Forward Neural Networks

Hyper Parameter Tunings

Feed Forward Neural Networks

- Hyper Parameter Tunings

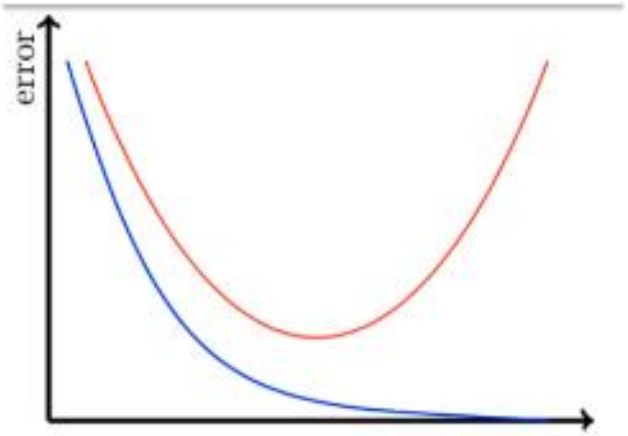
Feed Forward NN - Hyper Parameter Tunings

Network Architectures

- Number of layers
- Number of neurons

Algorithms

- Vanilla/Momentum /Nesterov GD
- AdaGrad
- RMSProp
- Adam



Activation Functions

- tanh (RNNs)
- relu (CNNs, DNNs)
- leaky relu (CNNs)

Strategies

- Batch
- Mini-Batch (32, 64, 128)
- Stochastic
- Learning rate schedule

Regularization

- L2
- Early stopping
- Dataset augmentation
- Drop-out
- Batch Normalizat

Initialization Methods

- *Xavier*
- *He*

Recap-Gradient Descent

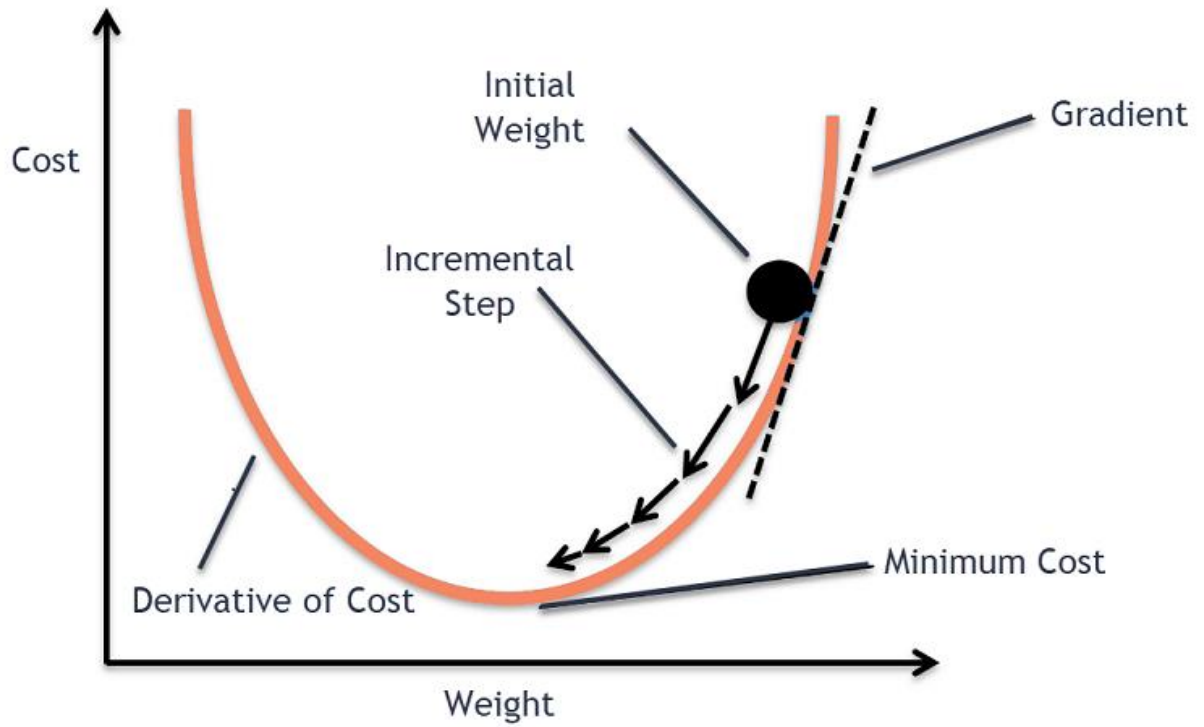
Gradient descent is a way to minimize an objective function parameterized by a model's parameters by updating the parameters in the opposite direction of the gradient of the objective function

The learning rate η determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley

Steps

1.Compute the slope (gradient) that is the first-order derivative of the function at the current point

2.Move-in the opposite direction of the slope increase from the current point by the computed amount



$$w_{t+1} = w_t - \eta \Delta w_t$$
$$b_{t+1} = b_t - \eta \Delta b_t$$
$$where\ \Delta w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}\ at\ w=w_t,b=b_t, \Delta b_t = \frac{\partial \mathscr{L}(w,b)}{\partial b}\ atw=w_t,b=b_t$$

Gradient Descent Algorithm- Tuning

- **Gradient Descent Variants**
  - Batch gradient descent
  - Stochastic gradient descent
  - Mini-batch gradient descent

How do you compute the gradients ?

or

What data should you use for computing the gradients?

- **Gradient descent optimization algorithms**
  - **Momentum**
  - **Nesterov accelerated gradient**
  - **Adagrad**
  - Adadelta
  - **RMSprop**
  - **Adam**
  - AdaMax
  - Nadam
  - AMSGrad

How do you use the gradients?

or

Can you come up with a better update rule ?

Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

Variants of Gradient Descent

- *What is one epoch? One step?*

$$\mathscr{L} = \sum_{i=1}^{i=N} (f(x_i) - y_i)^2$$

- 1 epoch = one pass over the entire data
- 1 step = one update of the parameters
- N = number of data points
- B = mini-batch size

$$\Delta w = \frac{\partial \mathscr{L}}{\partial w} = \sum_{i=1}^{i=N} \frac{\partial}{\partial w} (f(x_i) - y_i)^2$$

Algorithm	# of steps in one epoch
Batch gradient descent	1
Stochastic gradient descent	N
Mini-Batch gradient descent	N/B

$$w_{t+1} = w_t - \eta \Delta w_t$$
$$b_{t+1} = b_t - \eta \Delta b_t$$

$$where\ \Delta w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}\ at\ w=w_t, b=b_t, \Delta b_t = \frac{\partial \mathscr{L}(w,b)}{\partial b}\ atw=w_t, b=b_t$$

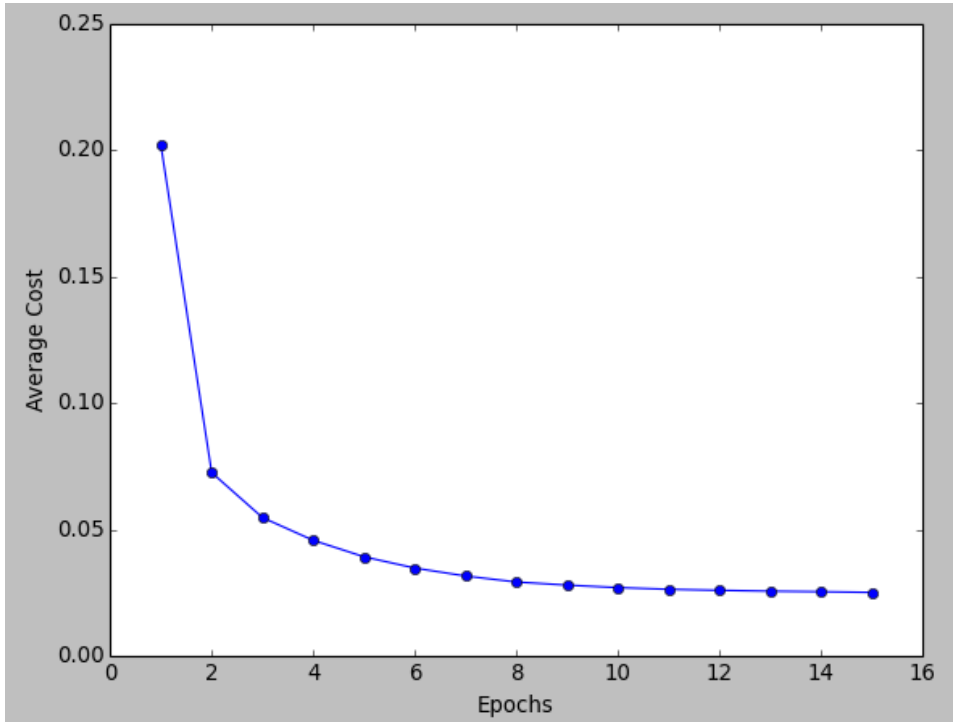
Batch Gradient Descent

1 step in 1 epoch

In Batch Gradient Descent,

- All the training data is taken into consideration to take a single step.
- We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters.
- So that's just one step of gradient descent in one epoch.

Batch gradient descent



Batch Gradient Descent is great for convex or relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution.

But what if our dataset is very huge. Suppose our dataset has 5 million examples, then just to take one step the model will have to calculate the gradients of all the 5 million examples.??

Stochastic Gradient Descent

N steps in 1 epoch

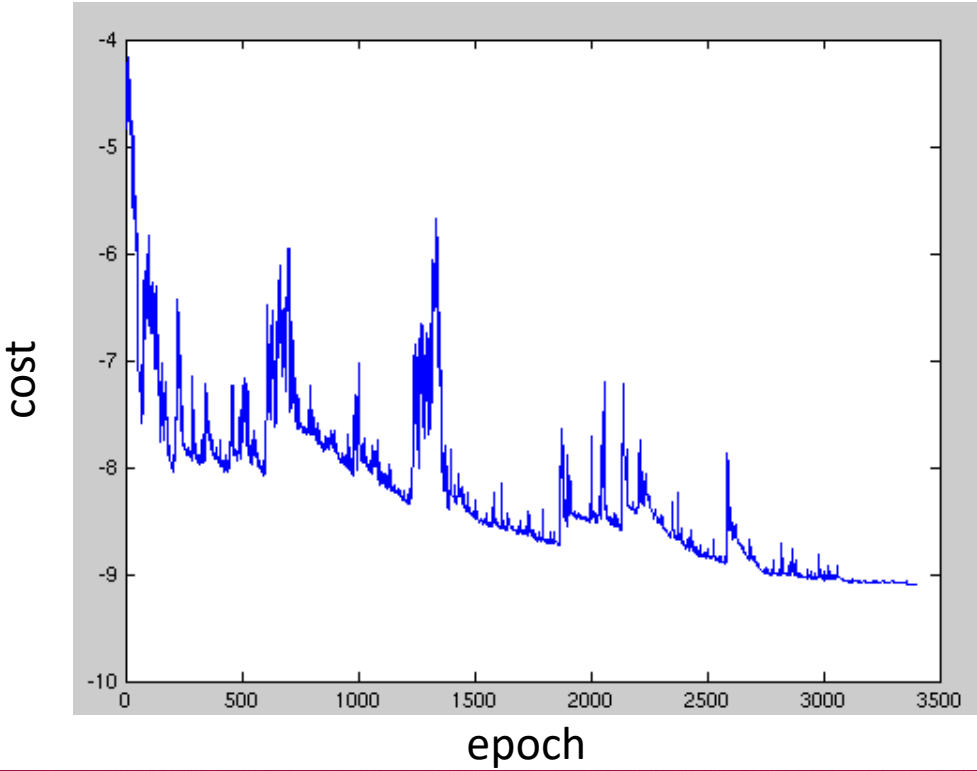
In Batch Gradient Descent we were considering all the examples for every step of Gradient Descent. But what if our dataset is very huge. Deep learning models crave for data. The more the data the more chances of a model to be good

Suppose our dataset has 5 million examples, then just to take one step the model will have to calculate the gradients of all the 5 million examples

In Stochastic Gradient Descent (SGD), we consider just one example at a time to take a single step.

Observations

- Since we are considering just one example at a time the cost will fluctuate over the training examples and it will **not** necessarily decrease.
- But in the long run, you will see the cost decreasing with fluctuations.
- Because the cost is so fluctuating, it will never reach the minima but it will keep dancing around it.
- SGD can be used for larger datasets. It converges faster when the dataset is large as it causes updates to the parameters more frequently



Mini Batch Gradient Descent

N/B steps in 1 epoch

Batch Gradient Descent can be used for smoother curves. SGD can be used when the dataset is large.

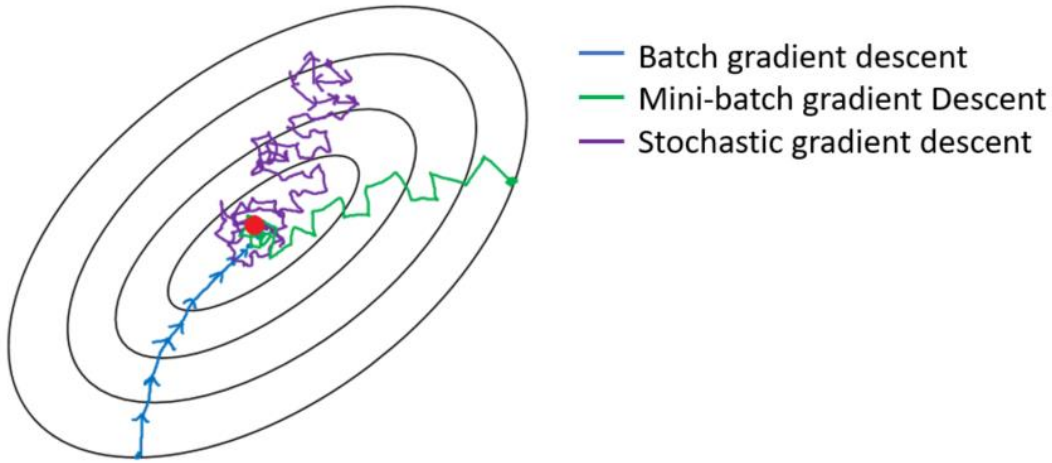
Batch Gradient Descent converges directly to minima. SGD converges faster for larger datasets.

But, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This can slow down the computations. To tackle this problem, a mixture of Batch Gradient Descent and SGD is used- **Mini Batch GD.**

Mini Batch gradient Descent

We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch. Doing this helps us achieve the advantages of both Batch and stochastic GD. So, after creating the mini-batches of fixed size, iterate over mini-batches of size B:

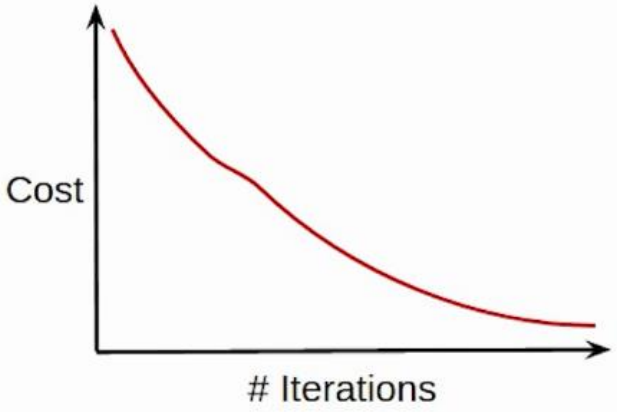
N/B step in one epoch (N : total no of data points, B batch size)



Normal Batch size- 32,64,128

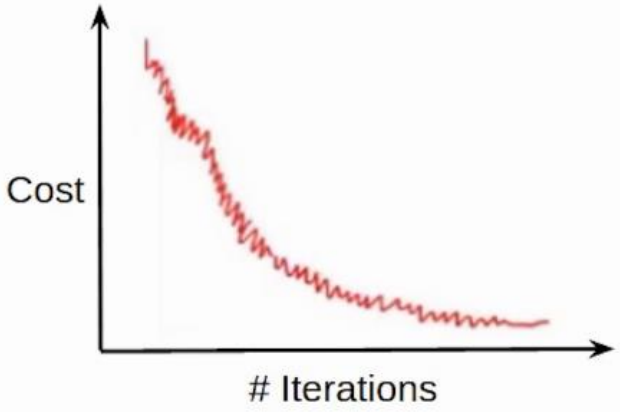
Batch Gradient Descent

- Entire dataset for updation
- Cost function reduces smoothly



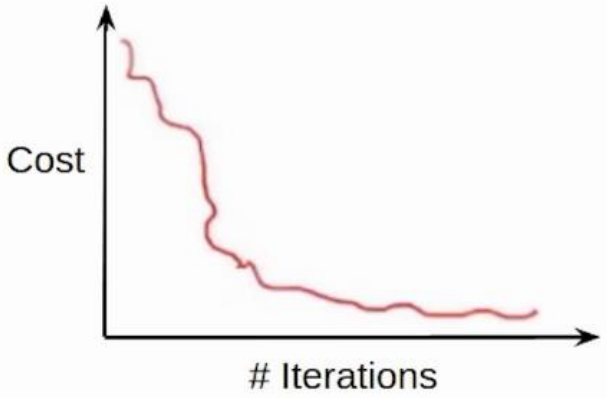
Stochastic Gradient Descent (SGD)

- Single observation for updation
- Lot of variations in cost function



Mini-Batch Gradient Descent

- Subset of data for updation
- Smoother cost function as compared to SGD



Batch gradient descent

```
def do_gradient_descent():
    w, b, eta = -2, -2, 1.0
    max_epochs = 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

Stochastic gradient descent

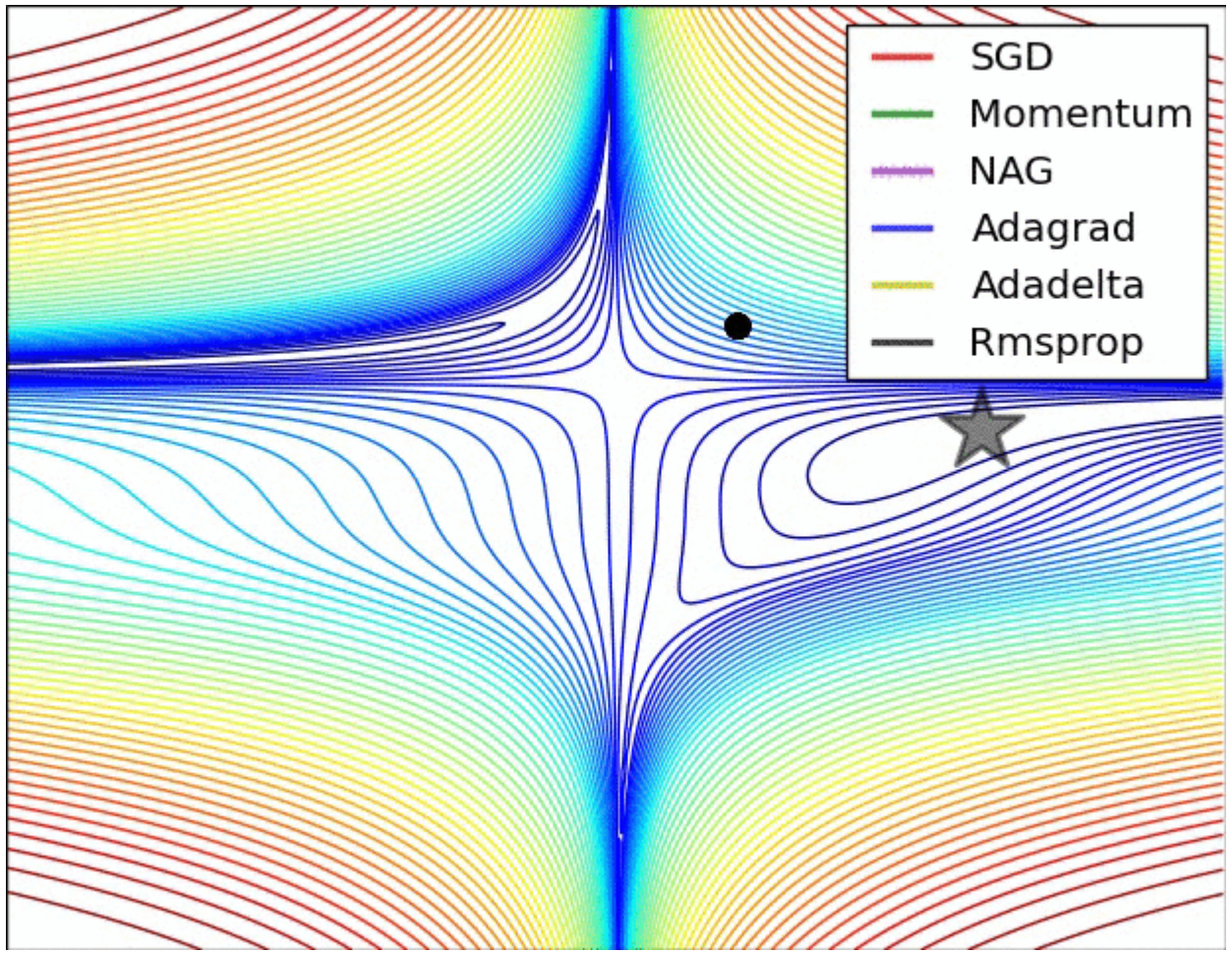
```
def do_stochastic_gradient_descent():
    w, b, eta = -2, -2, 1.0
    max_epochs = 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

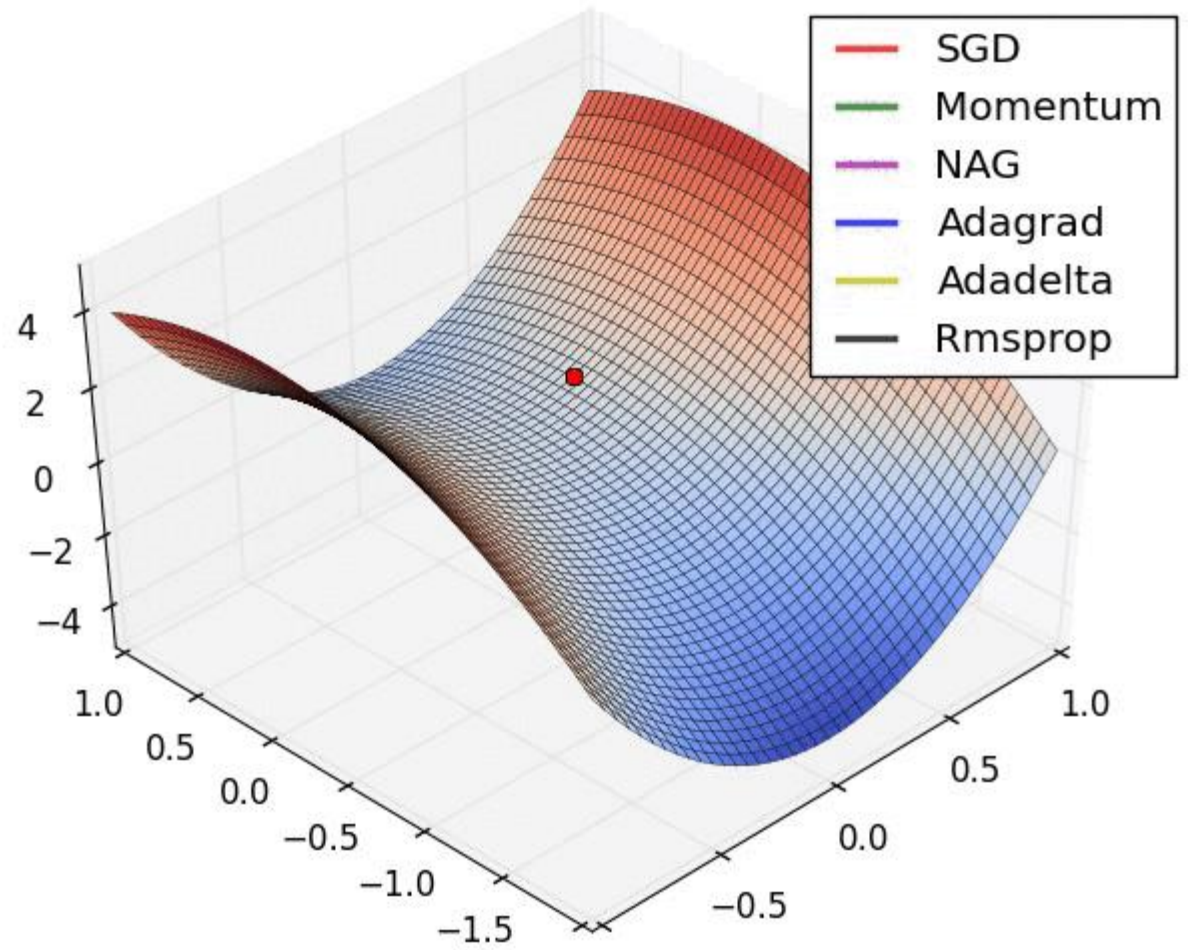
Mini-batch gradient descent

```
def do_mini_batch_gradient_descent():
    w, b, eta = -2, -2, 1.0
    max_epochs = 1000
    mini_batch_size = 0
    num_points_seen = 0
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
            num_points_seen += 1

            if num_points_seen % mini_batch_size == 0:
                w = w - eta * dw
                b = b - eta * db
```

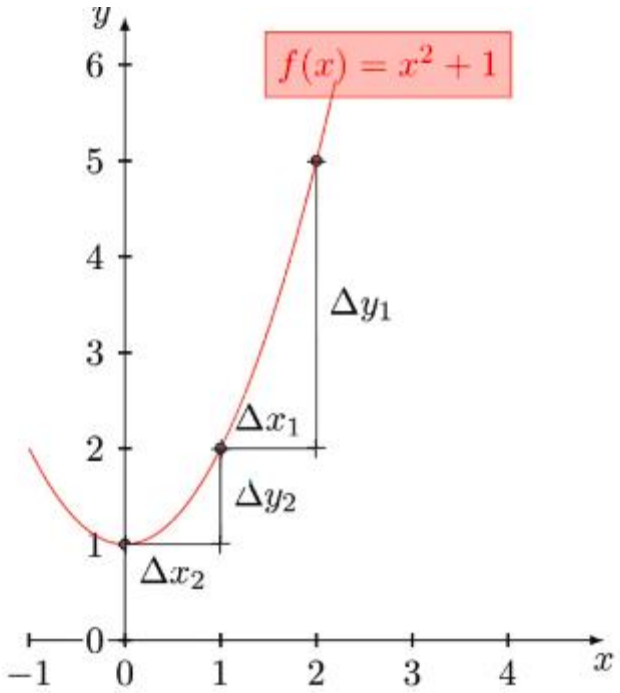
Comparison- Visualisation





Issue with Gradient Descent optimisation

vanilla gradient descent



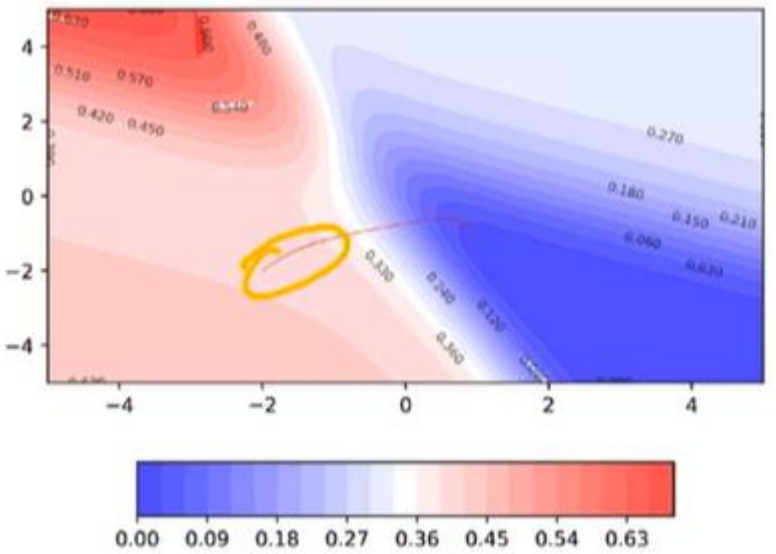
Gradient Descent Update Rule

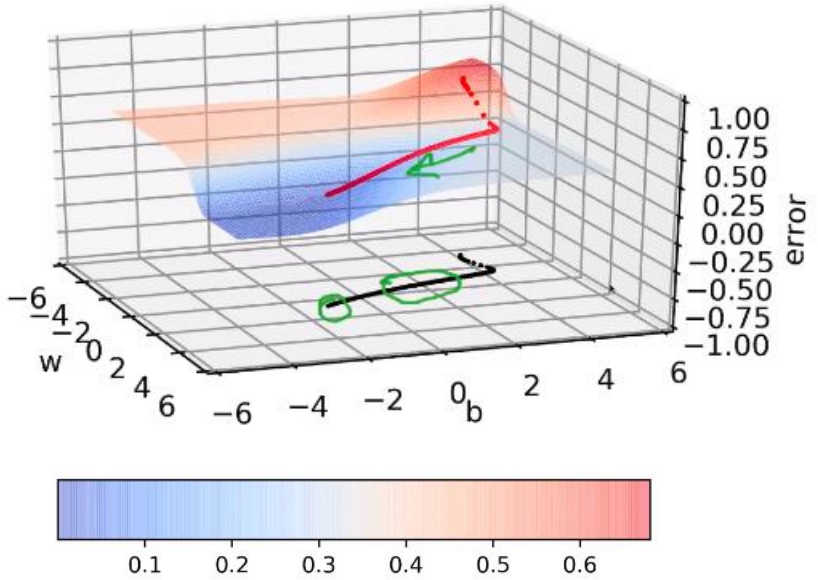
$$w = w - \eta \frac{\partial \mathscr{L}(w)}{\partial w}$$

Issues

It takes a lot of time to navigate regions having gentle slope (because the gradient in these regions is very small)

2D contour Maps-3D surface on 2D map





Momentum-Based Gradient Descent

Intuition : *If I am repeatedly being asked to move in the same direction, then I should probably gain some confidence and start taking bigger steps in that direction. Just as a ball gains momentum while rolling down a slope.*

- In addition to the current update, we also look at the history of updates.
- You can see that the current update is proportional to not just the present gradient but also gradients of previous steps, although their contribution reduces every time step by γ(gamma) times. And that is how we boost the magnitude of the update at gentle regions.

Momentum based Gradient Descent Update Rule

$$v_t = \gamma * v_{t-1} + \eta \nabla w_t$$

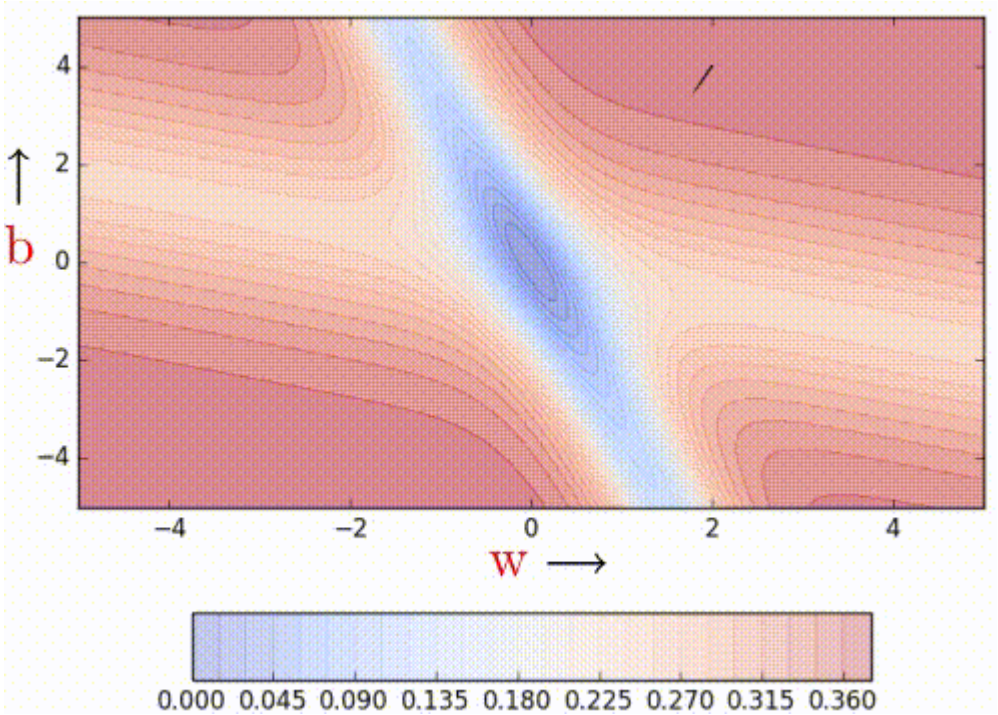
$$w_{t+1} = w_t - v_t$$

Gradient Descent Update Rule

$$w_{t+1} = w_t - \eta \nabla w_t$$

$$\begin{aligned}
v_0 &= 0 \\
v_1 &= \gamma \cdot v_0 + \eta \nabla w_1 = \eta \nabla w_1 \\
v_2 &= \gamma \cdot v_1 + \eta \nabla w_2 = \gamma \cdot \eta \nabla w_1 + \eta \nabla w_2 \\
v_3 &= \gamma \cdot v_2 + \eta \nabla w_3 = \gamma(\gamma \cdot \eta \nabla w_1 + \eta \nabla w_2) + \eta \nabla w_3 \\
&= \gamma \cdot v_2 + \eta \nabla w_3 = \gamma^2 \cdot \eta \nabla w_1 + \gamma \cdot \eta \nabla w_2 + \eta \nabla w_3 \\
v_4 &= \gamma \cdot v_3 + \eta \nabla w_4 = \gamma^3 \cdot \eta \nabla w_1 + \gamma^2 \cdot \eta \nabla w_2 + \gamma \cdot \eta \nabla w_3 + \eta \nabla w_4 \\
&\vdots \\
v_t &= \gamma \cdot v_{t-1} + \eta \nabla w_t = \gamma^{t-1} \cdot \eta \nabla w_1 + \gamma^{t-2} \cdot \eta \nabla w_1 + \ldots + \eta \nabla w_t
\end{aligned}$$

Advantage and disadvantage of momentum based GD



momentum would cause us to overshoot and run past our goal!

We can observe that momentum-based gradient descent oscillates in and out of the minima valley as the momentum carries it out of the valley. This makes us take a lot of U-turns before finally converging. **Despite these U-turns, it still converges faster than vanilla gradient descent**

It is evident that even in the regions having gentle slopes, momentum-based gradient descent can take substantial steps because the momentum carries it along.

Can we do something to reduce the oscillations/U-turns? Yes, Nesterov Accelerated Gradient Descent helps us do just that.

Nesterov Accelerated Gradient Descent (NAG)

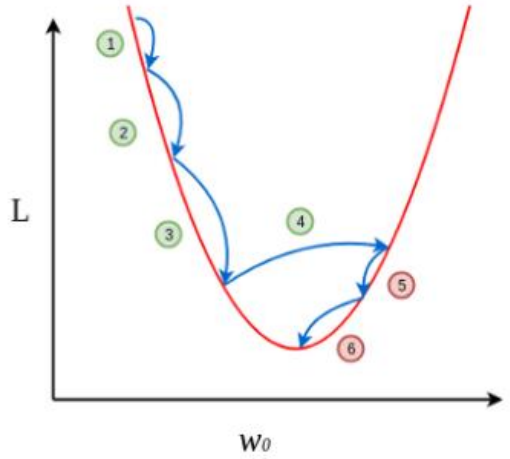
Intuition : *Look ahead before you leap!* → Effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

NAG Update Rule

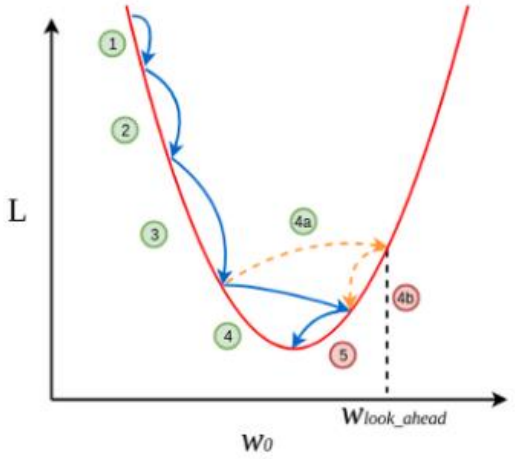
$$w_{temp} = w_t - \gamma * v_{t-1}$$

$$w_{t+1} = w_{temp} - \eta \nabla w_{temp}$$

$$v_t = \gamma * v_{t-1} + \eta \nabla w_{temp}$$



(a) Momentum-Based Gradient Descent



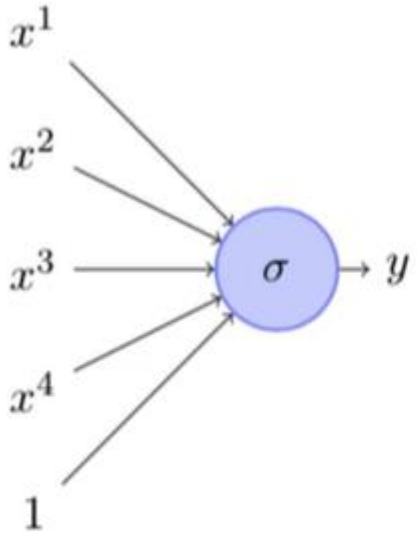
(b) Nesterov Accelerated Gradient Descent

Calculate the gradient at the partially updated value w_{temp} instead of calculating using the current value w_t

NAG's case, every update happens in two steps — first, a partial update, where we get to the *look_ahead* point and then the final update which is negative

This negative final update slightly reduces the overall magnitude of the update, still resulting in an overshoot but a smaller one when compared to the vanilla momentum-based gradient descent

Why adaptive learning rate needed



$$y = f(x) = \frac{1}{1+e^{-(\mathbf{w}\cdot\mathbf{x}+b)}}$$

$$\mathbf{x} = \{x^1, x^2, x^3, x^4\}$$

$$\mathbf{w} = \{w^1, w^2, w^3, w^4\}$$

$$\nabla w_1^1 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^1$$
$$\nabla w^2 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^2$$

$$\nabla w \propto x$$

Updated $w \propto \nabla w$ $\quad w_{t+1} = w_t - \eta \Delta w_t$

Can we have a different learning rate for each parameter which takes care of the frequency of features ?

In real data some features will be sparse and some dense.
We want η to be aggressive in case of sparse features and less aggressive in case of dense features- adaptive learning rate (I don't mind learning slow when feature is ON all the time. But when feature is ON less frequently then learn aggressively when it is ON)

Adagrad

Advantage

- Parameters corresponding to sparse features get better updates

Adagrad

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

Intuition: Decay the learning rate for parameters in proportion to their update history (fewer updates, lesser decay)

Performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. For this reason, it is well-suited for dealing with sparse data

- Denominator will be less for sparse features (History taken) and will be high for dense features. Hence high fraction for sparse features – learn aggressive. Low fraction for dense features- learn less aggressive

$$\frac{\eta}{\sqrt{(v_t)} + \epsilon}$$

Disadvantage

- The learning rate decays very aggressively as the denominator grows (not good for parameters corresponding to dense features)

RMSProp

- Adagrad got stuck when it was close to convergence (it was no longer able to move in the vertical (b) direction because of the decayed learning rate)
- RMSProp overcomes this problem by being less aggressive on the decay

Intuition: Why not decay the denominator and prevent its rapid growth ?

RMSProp

$$v_t = \beta * v_{t-1} + (1-\beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

Adagrad

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

- RMSprop resolve Adagrad's radically diminishing learning rates.
- It divides the learning rate by an exponentially decaying average of squared gradients

RMSprop

$V_0 = 0$

$V_1 = 0.1(\nabla\omega_1)^2$

$V_2 = \beta V_1 + (1-\beta)\nabla\omega_2^2$

$= (0.9)(0.1)\nabla\omega_1^2 + 0.1\nabla\omega_2^2$

$V_3 = (0.9)^2(0.1)\nabla\omega_1^2 + (0.9)(0.1)\nabla\omega_2^2$

$V_4 = (0.9)^3(0.1)\nabla\omega_1^2 + (0.9)^2(0.1)\nabla\omega_2^2$

$+ (0.9)(0.1)\nabla\omega_3^2 + (0.1)\nabla\omega_4^2$

Adagrad

$V_4 = \nabla\omega_1^2 + \nabla\omega_2^2 + \nabla\omega_3^2 + \nabla\omega_4^2$

RMSProp Will prevent blowup of denominator for dense features

Adam

In addition to storing an exponentially decaying average of past squared gradients vtvt like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients mtmt, similar to momentum.

Momentum based Gradient Descent Update Rule

$$v_t = \gamma * v_{t-1} + \eta \nabla w_t$$

$$w_{t+1} = w_t - v_t$$

RMSProp

$$v_t = \beta * v_{t-1} + (1 - \beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

Adam

$$m_t = \beta_1 * v_{t-1} + (1 - \beta_1)(\nabla w_t)$$

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

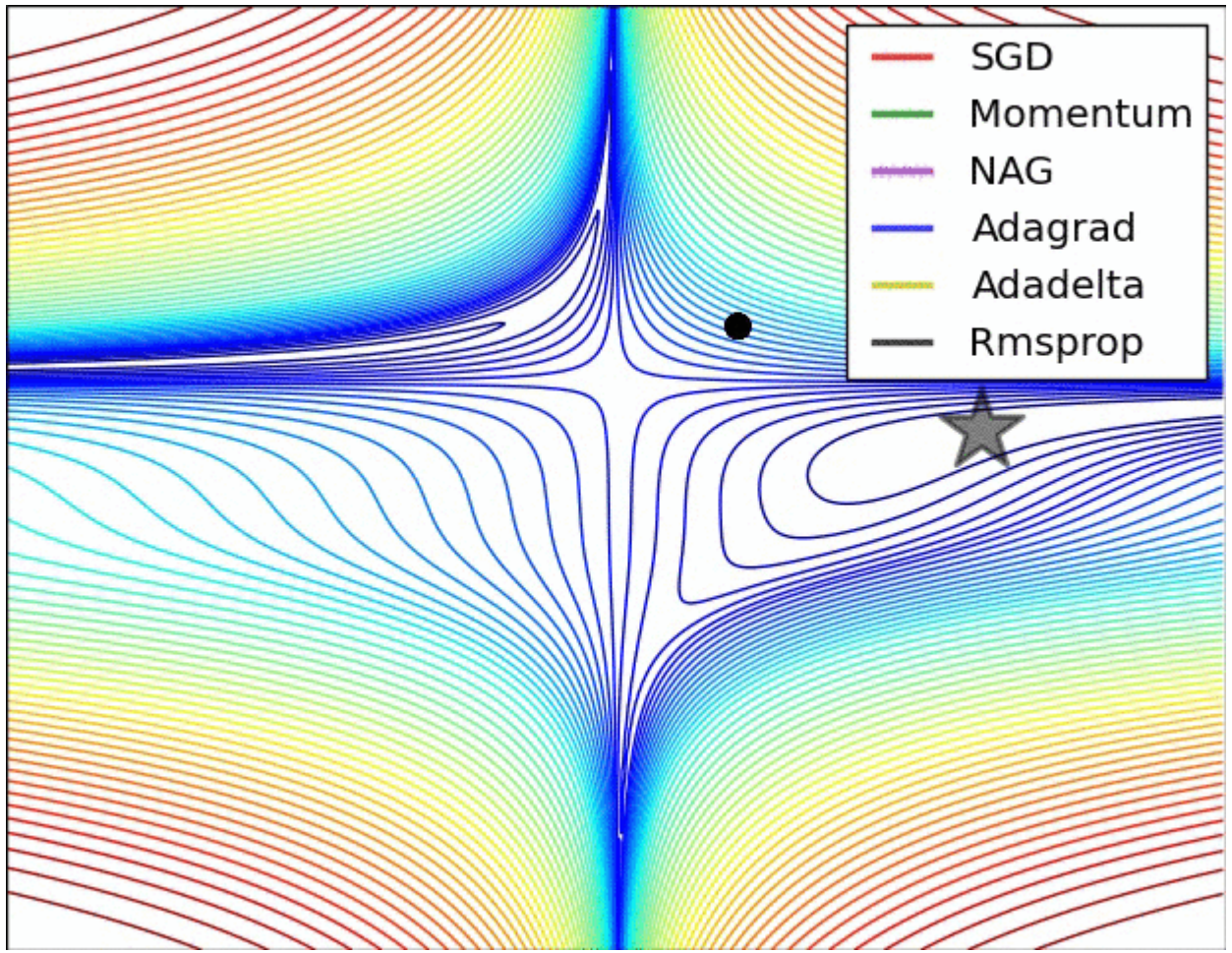
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} m_t$$

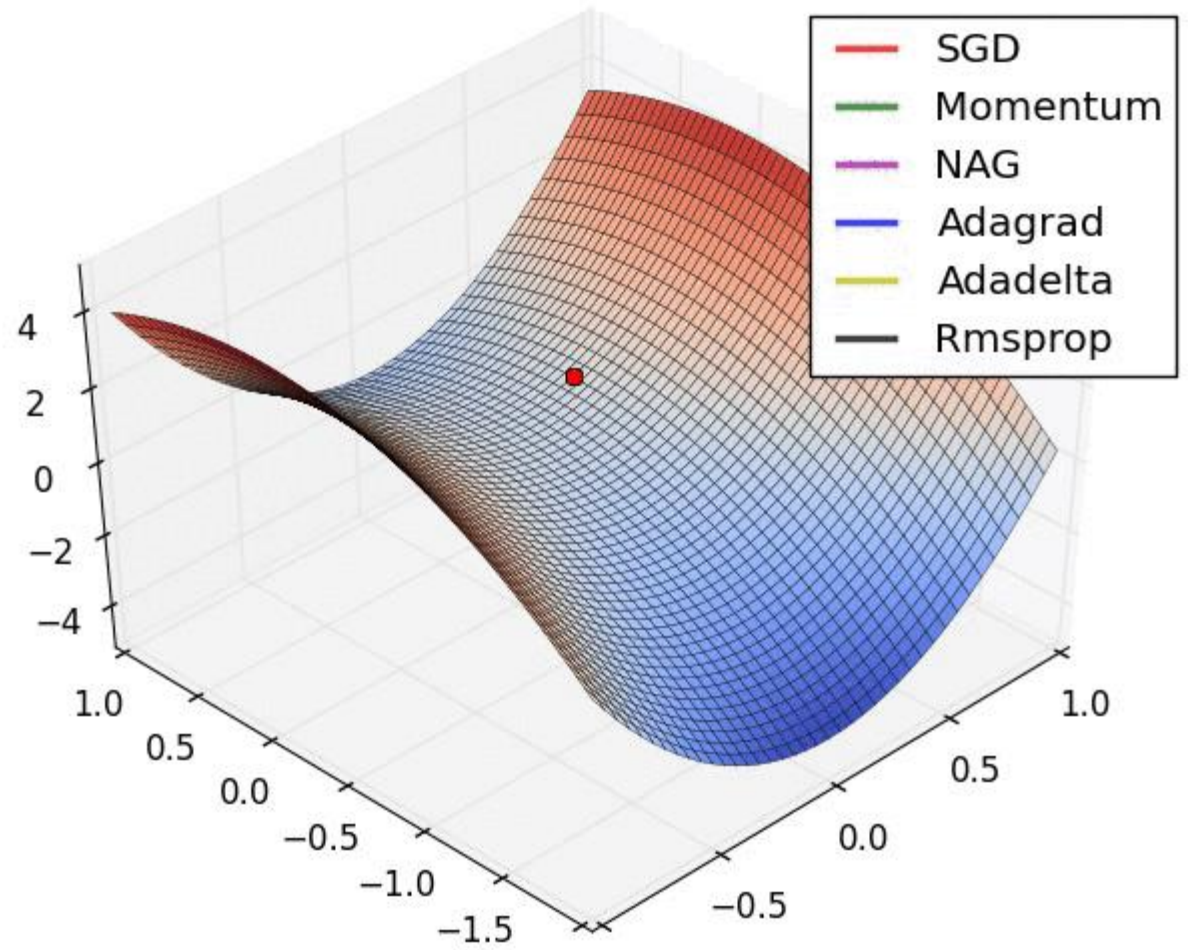
$$m_t = \frac{m_t}{1 - \beta_1^t}$$

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

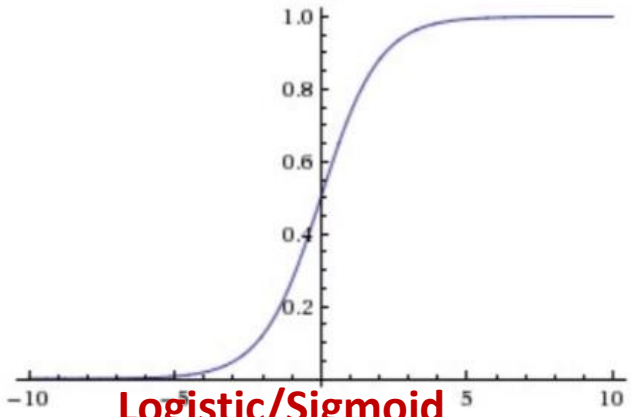
Combines the advantage of momentum and RMSprop

Comparison- Visualisation

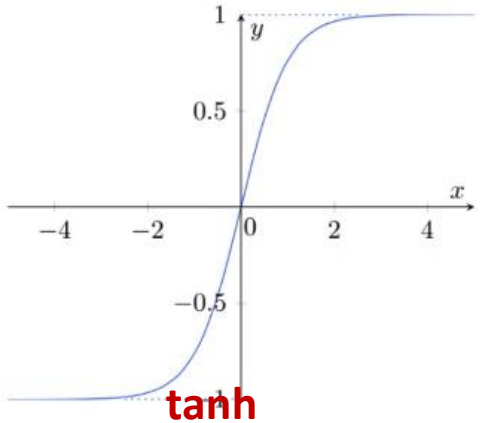




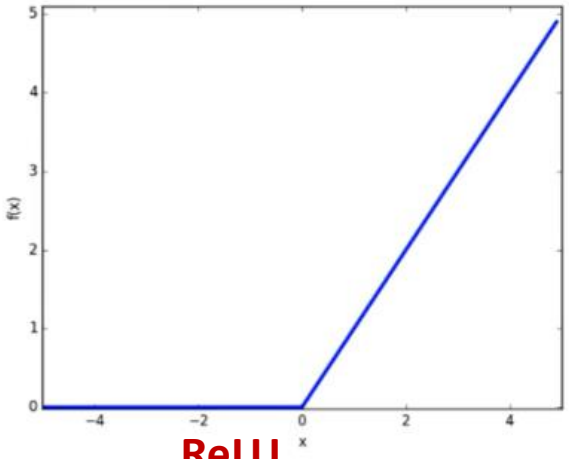
Activation functions



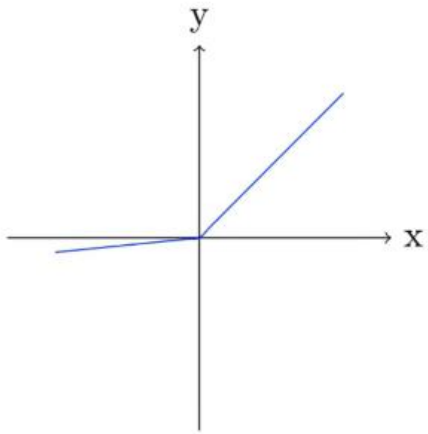
Logistic/Sigmoid



tanh



ReLU



Leaky ReLU

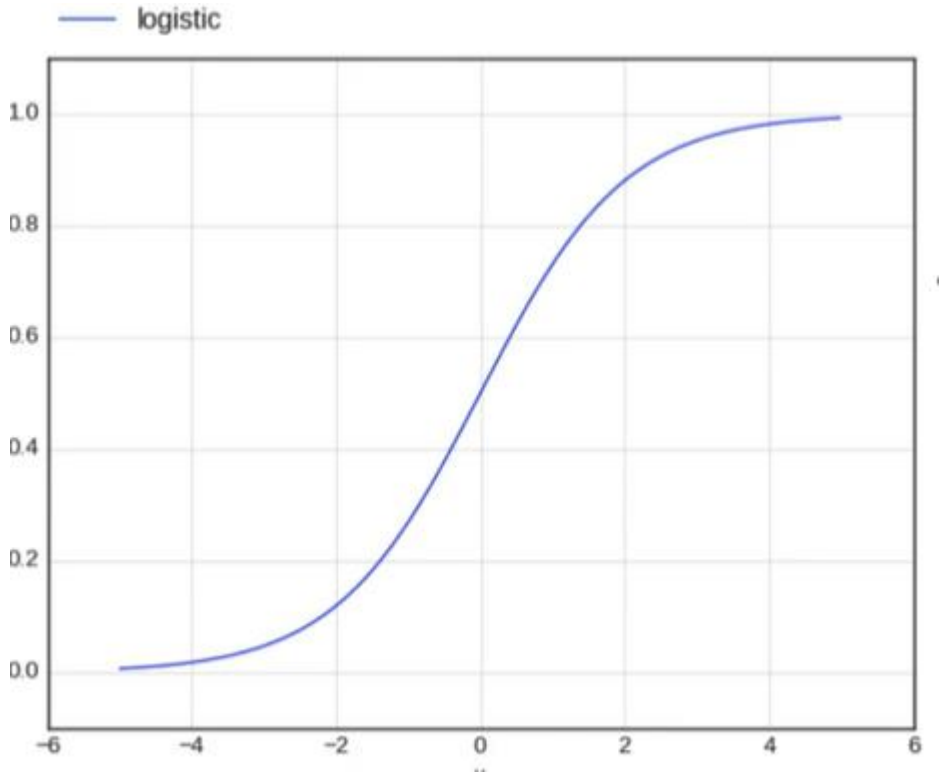
Logistic/Sigmoid

Vanishing Gradient problem

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

always normalize the inputs (so that they lie between 0 to 1)

$$f'(x) = \frac{\partial f(x)}{\partial x} = f(x) * (1 - f(x))$$



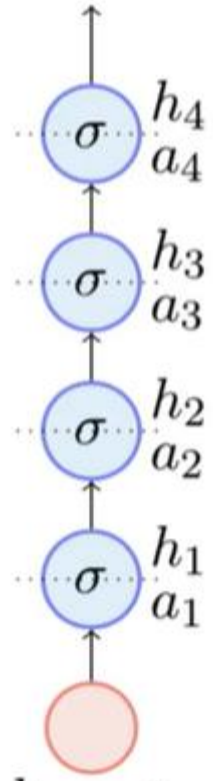
Saturated neurons cause the gradients to vanish

Saturation:

when $f(x) = 0$ *or* 1

and hence $f'(x) = 0$

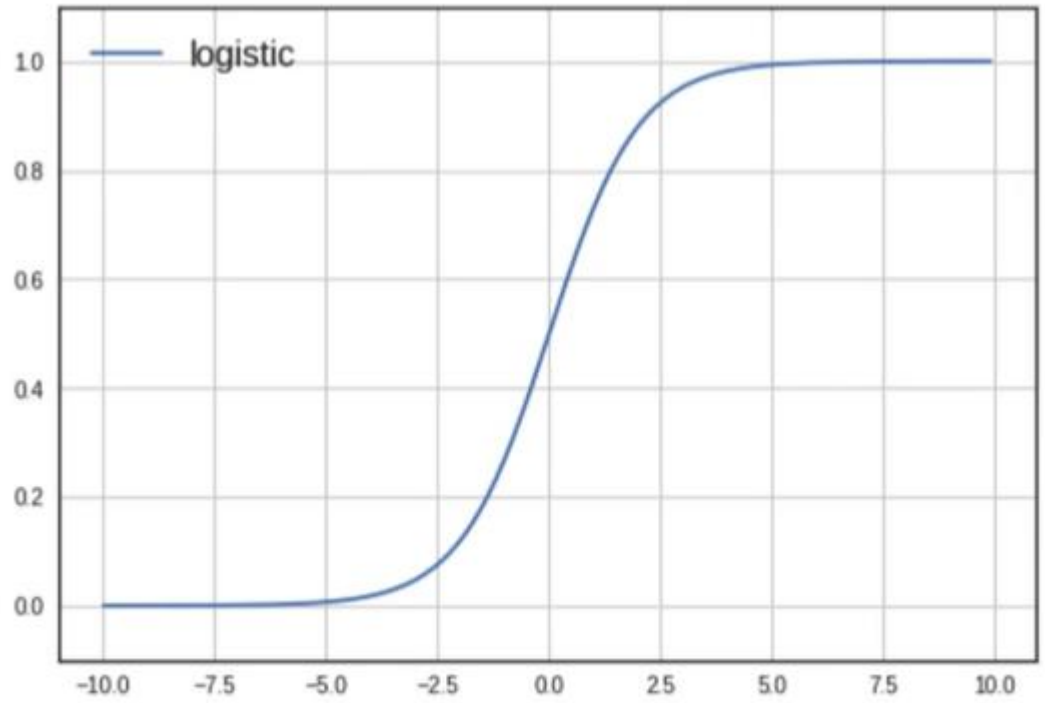
$$\underbrace{\frac{\partial \mathscr{L}(\theta)}{\partial W_{111}}}_{\text{Talk to the weight directly}} = \underbrace{\frac{\partial \mathscr{L}(\theta)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_3}}_{\text{Talk to the output layer}} \underbrace{\frac{\partial a_3}{\partial h_2} \frac{\partial h_2}{\partial a_2}}_{\text{Talk to the previous hidden layer}} \underbrace{\frac{\partial a_2}{\partial h_1} \frac{\partial h_1}{\partial a_1}}_{\text{Talk to the previous hidden layer}} \underbrace{\frac{\partial a_1}{\partial W_{111}}}_{\text{and now talk to the weights}}$$



$\frac{\partial h_2}{\partial a_2} = 0$ Then $\nabla w = 0$ and hence no updation in weight happens

Gradient at some neurons vanishes and no more learning happens- Vanishing gradient

Logistic/Sigmoid

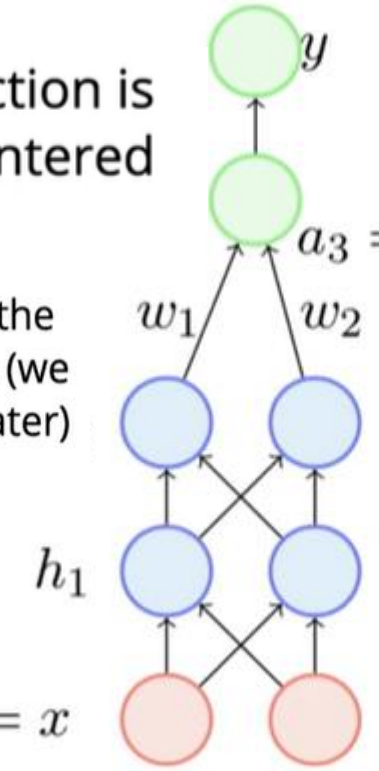


$\sum_{i=1}^{4} w_i x_i$

Logistic Neuron is not zero centered which restricts the movement of gradient on training – takes time to converge

logistic function is not zero-centered

Remember to initialize the weights to small values (we will come back to this later)



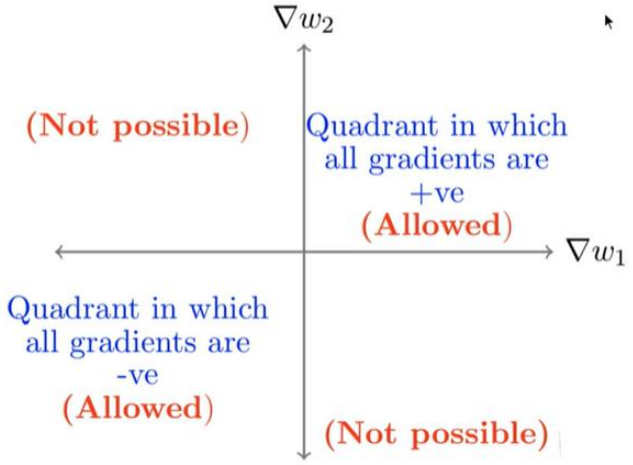
$$\nabla w_1 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \frac{\partial a_3}{\partial w_1}$$

$$\nabla w_2 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \frac{\partial a_3}{\partial w_2}$$

+/-

$$\nabla w_1 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \; + \; h_{21}$$

$$\nabla w_2 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \; h_{22}$$



∇w_1 :

∇w_2 :

Will be either all + or all-

+/- +

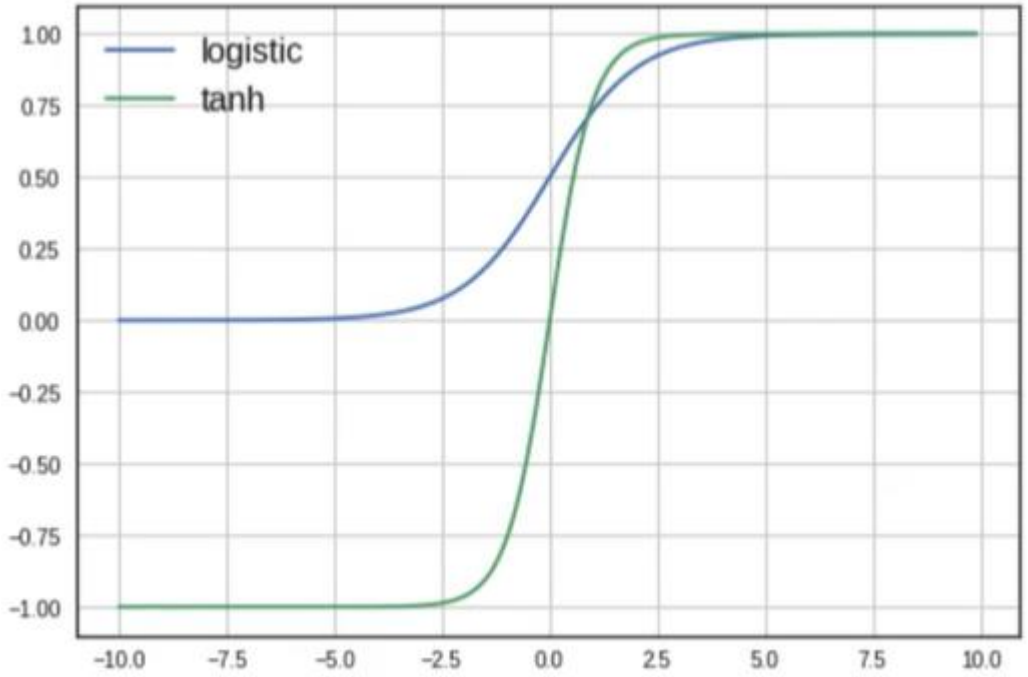
The gradients w.r.t. all the weights connected to the same neuron are either all +ve or all -ve

Better Activation Functions- tanh

- Saturated logistic neurons cause the gradients to vanish
- logistic function is not zero-centered
- logistic function is computationally expensive (because of e^x

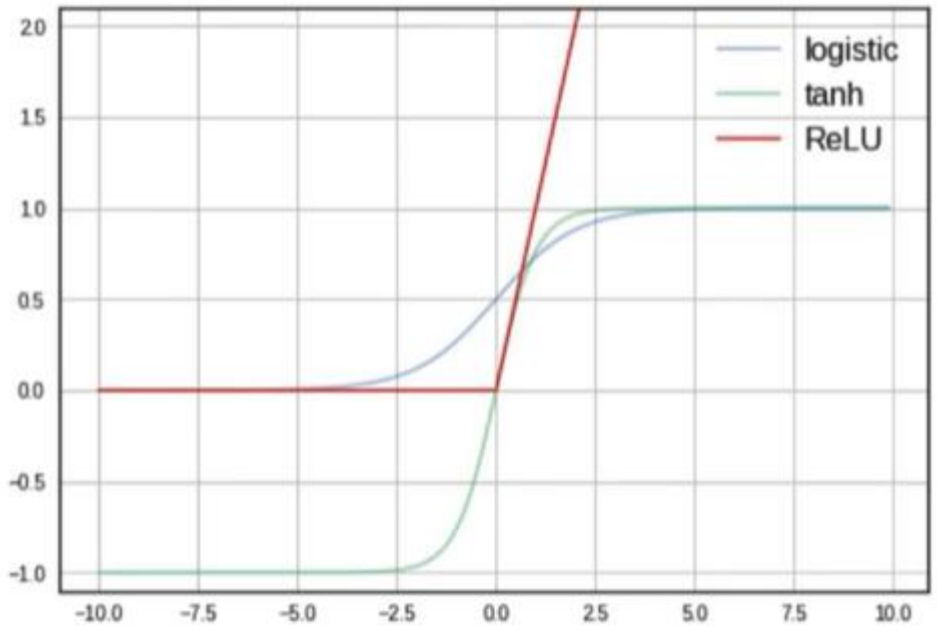
$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$f'(x) = \frac{\partial f(x)}{\partial x} = (1 - (f(x))^2)$$



- Saturated tanh neurons cause the gradients to vanish
- tanh is zero-centered
- tanh is computationally expensive (because of e^x

Better Activation Functions- ReLU (Rectified Linear activation Unit)

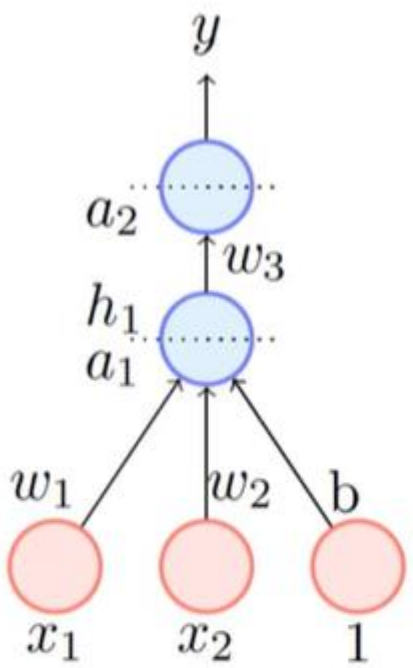


$$f(x) = max(0, x)$$

$$f'(x) = \frac{\partial f(x)}{\partial x} = \begin{cases} 0 & if \quad x < 0 \\ 1 & if \quad x > 0 \end{cases}$$

- Does not saturate in the positive region
- Not zero-centered
- Easy to compute (no expensive e^x)

Issues with ReLU



$$h_1 = ReLU(a_1) = max(0, a_1)$$
$$= max(0, w_1 x_1 + w_2 x_2 + b)$$

What happens if b takes on a large negative value due to a large negative update (∇b) at some point ?

$$w_1 x_1 + w_2 x_2 + b < 0 \quad [if \quad b << 0]$$
$$\implies h_1 = 0 \quad [dead\ neuron]$$
$$\implies \frac{\partial h_1}{\partial a_1} = 0$$

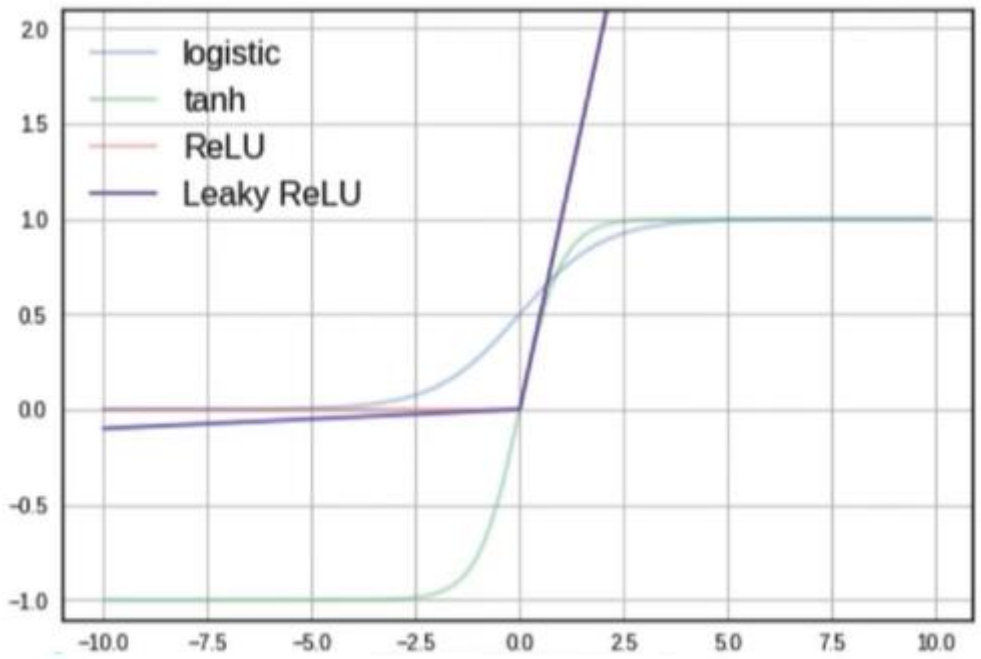
$$\nabla w_1 = \frac{\partial \mathscr{L}(\theta)}{\partial y} \cdot \frac{\partial y}{\partial a_2} \cdot \frac{\partial a_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1}$$

- A large fraction of ReLU units can die if the learning rate is set too high

- It is advised to initialize the bias to a positive value

- Use other variants of ReLU

Leaky ReLU

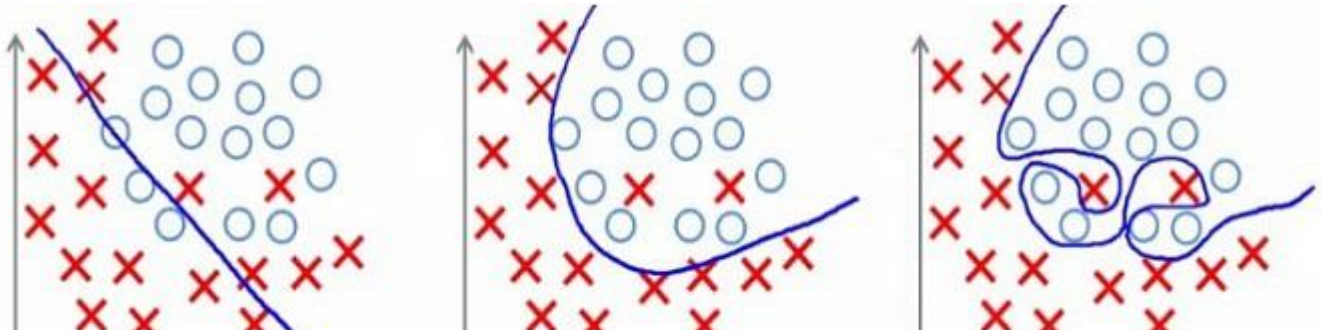


- Does not saturate in positive or negative region
- Will not die (0.01 x ensures that at least a small gradient will flow through)

$$f(x) = max(0.01x, x)$$

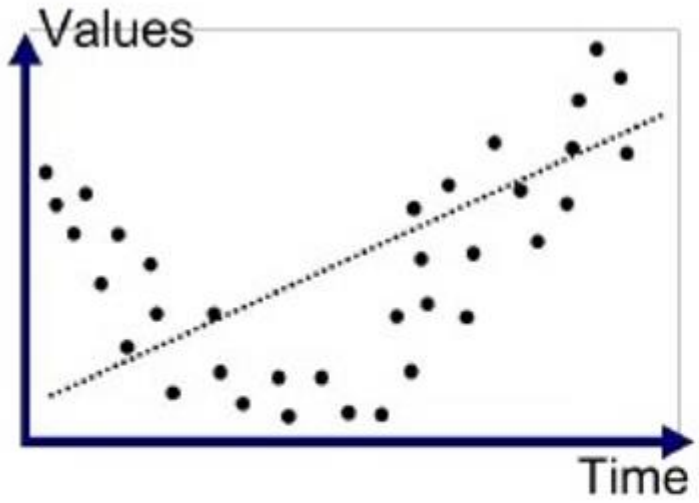
$$f'(x) = \frac{\partial f(x)}{\partial x} = \begin{cases} 0.01 & if \quad x < 0 \\ 1 & if \quad x > 0 \end{cases}$$

Overfitting - Underfitting

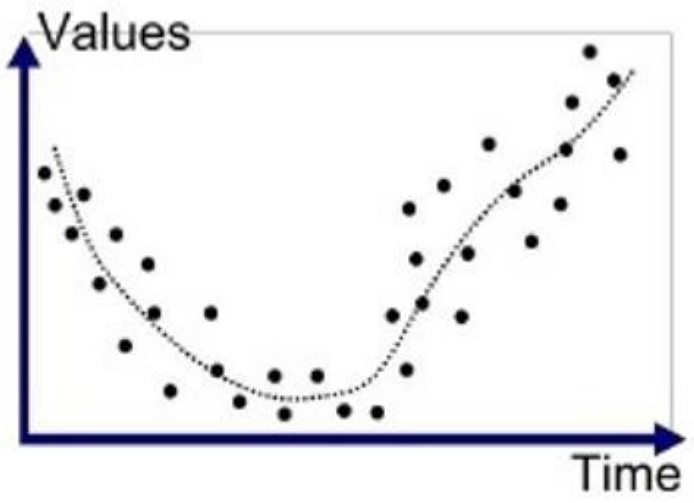


In supervised learning, **underfitting** happens when a model unable to capture the underlying pattern of the data (Not able to learn)

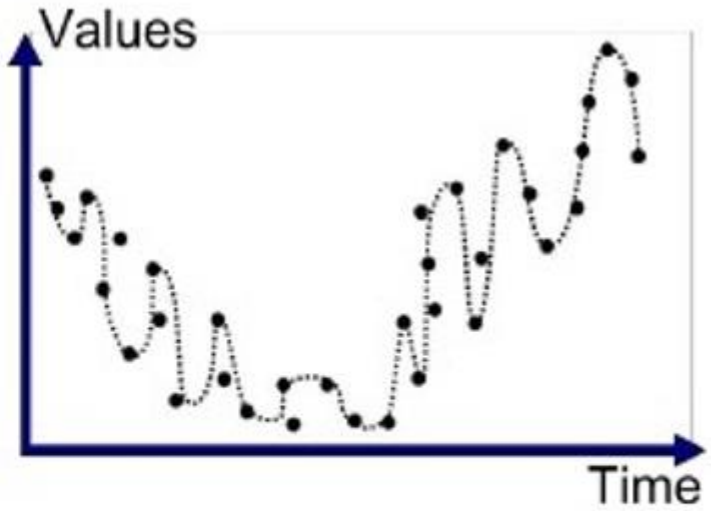
overfitting happens when our model captures the noise along with the underlying pattern in data.



Underfitted

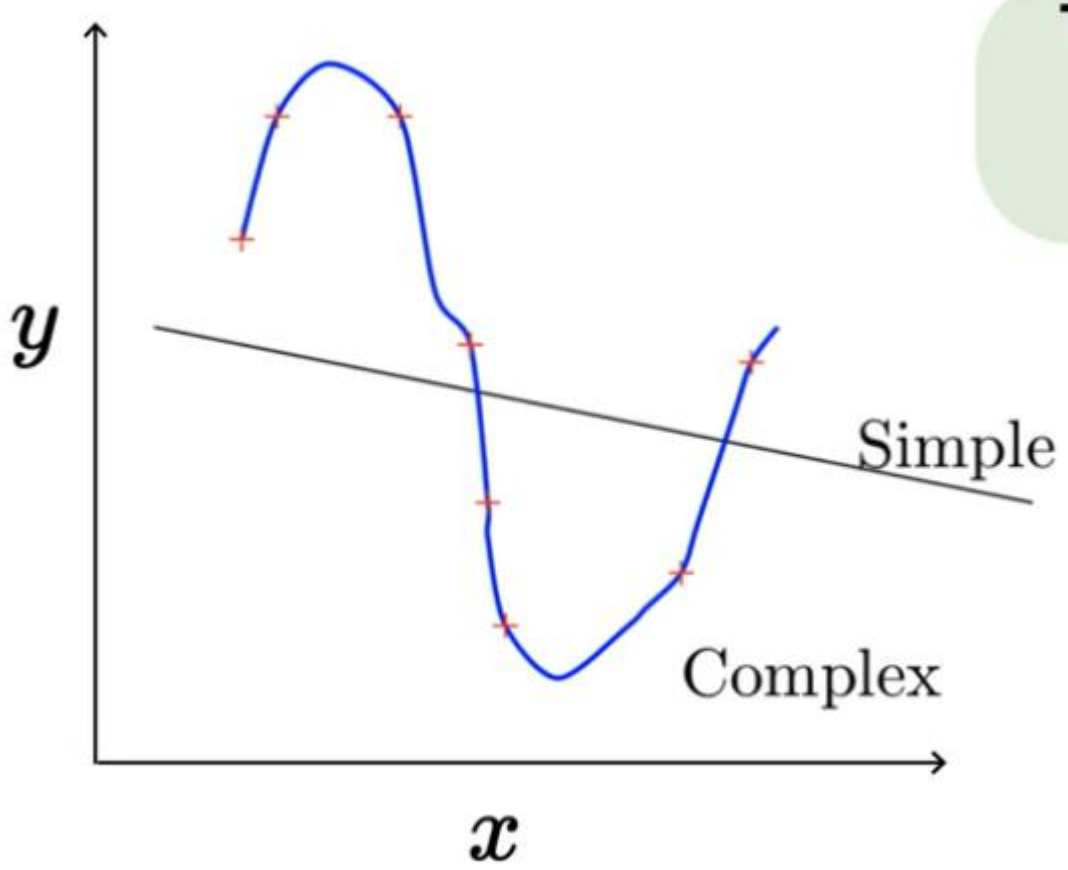


Good Fit/Robust



Overfitted

Bias- Variance



True Relation*

$$y = f(x)$$

Our Approximation(model):

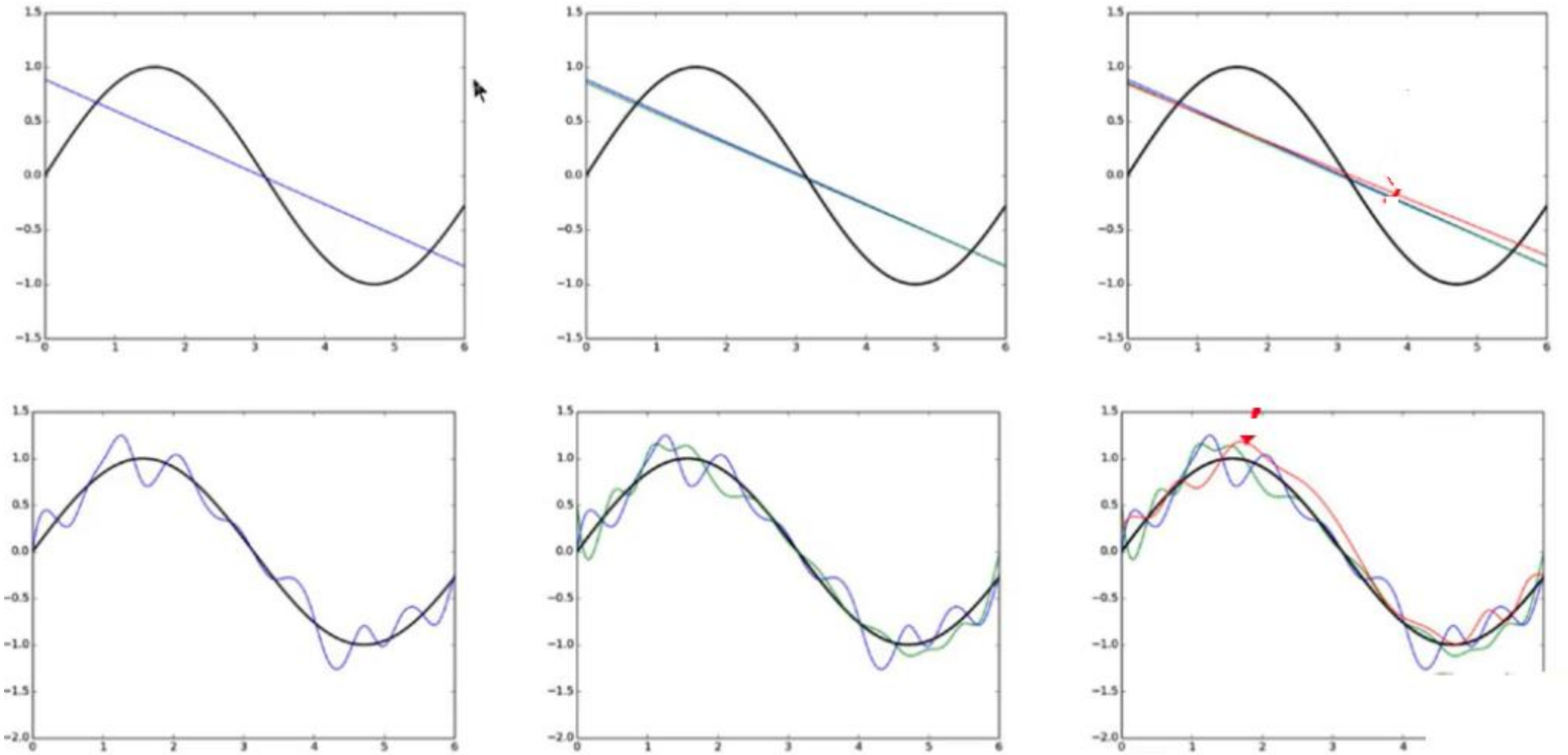
$$\underset{(degree:1)}{Simple} \quad y = \hat{f}(x) = w_1 x + w_0$$

$$\underset{(degree:25)}{Complex} \quad y = \hat{f}(x) = \sum_{i=1}^{25} w_i x^i + w_0$$

*In this case I know that $f(x) = sin(x)$

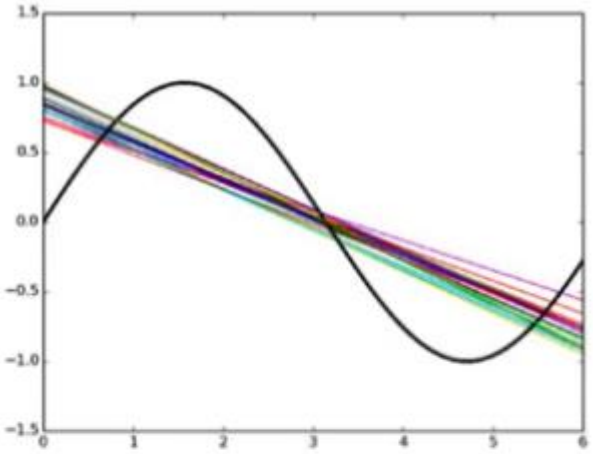
Bias and Variance trade-off

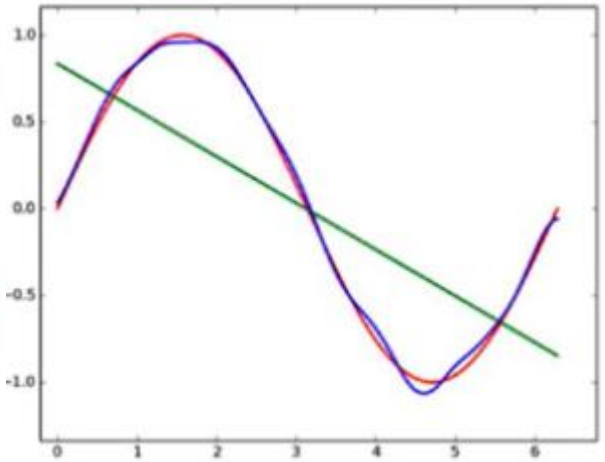
What happens if you train with different subsets of training data

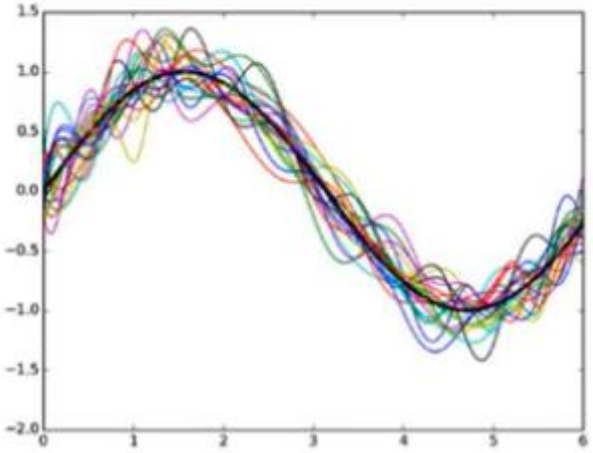


sine curve(Black color) is the actual model

Bias and Variance trade-off







Simple Model: high bias, low variance

Complex Model: low bias, high variance

Ideal Model: low bias, low variance

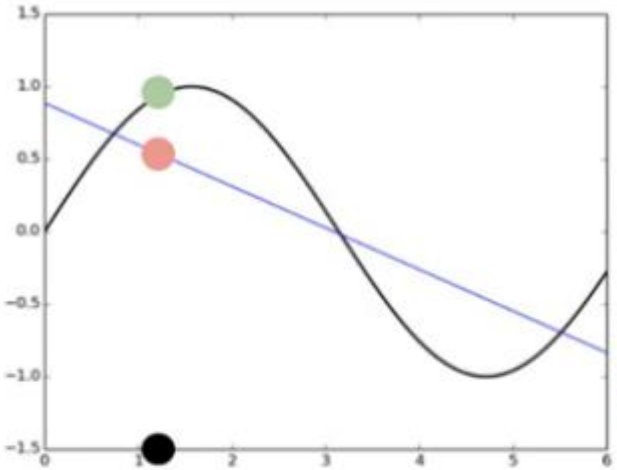
$$\text{Bias } (\hat{f}(x)) = E[\hat{f}(x)] - f(x)$$

$$\text{Variance } (\hat{f}(x)) = E[(\hat{f}(x) - E[\hat{f}(x)])^2]$$

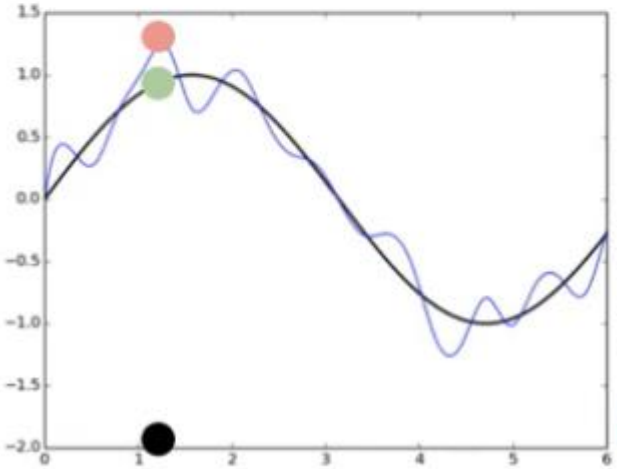
Bias is the **difference between the average prediction of our model and the correct value** which we are trying to predict. Model with high bias pays very little attention to the training data and oversimplifies the model. It always leads to **high error on training and test data**.

Variance is the **variability of model prediction for a given data point or a value** which tells us spread of our data. Model with high variance pays a lot of attention to training data and does not generalize on the data which it hasn't seen before. As a result, such models **perform very well on training data** but has **high error rates on test data.**

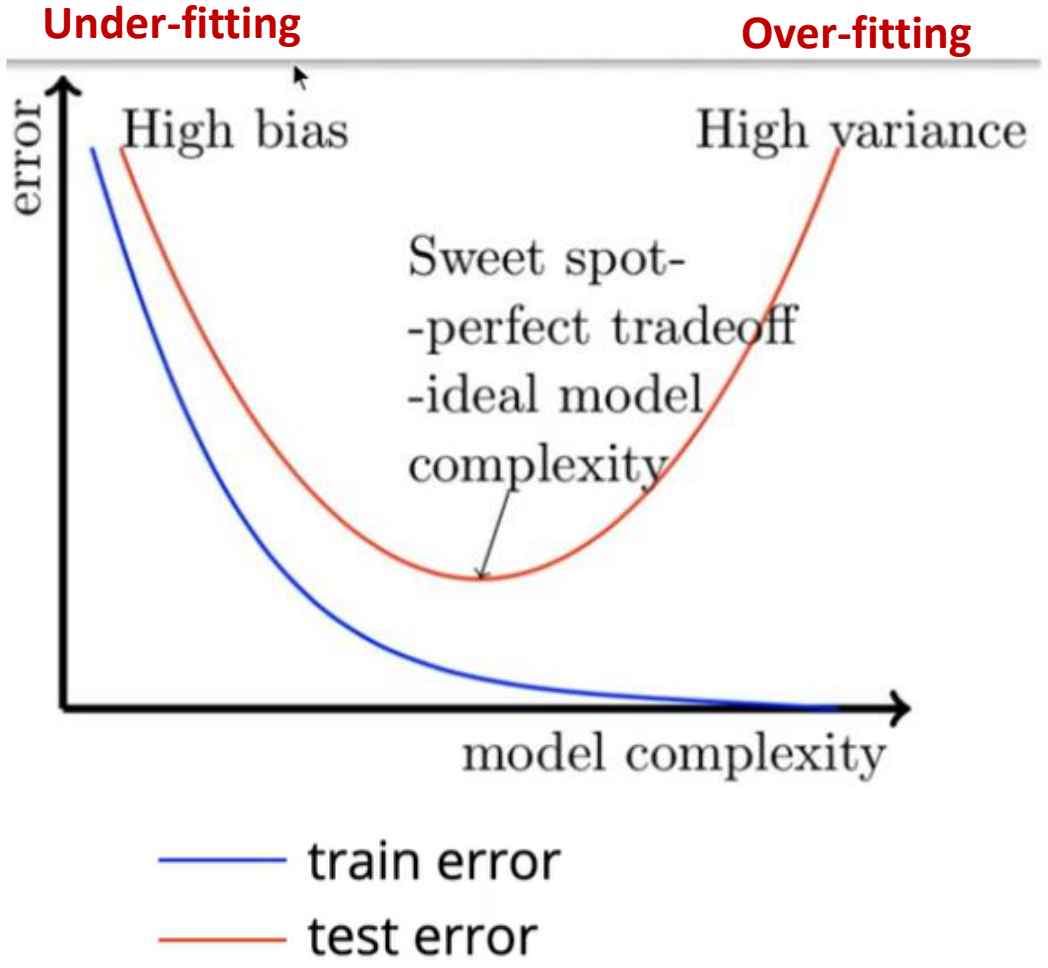
Bias and Variance trade-off



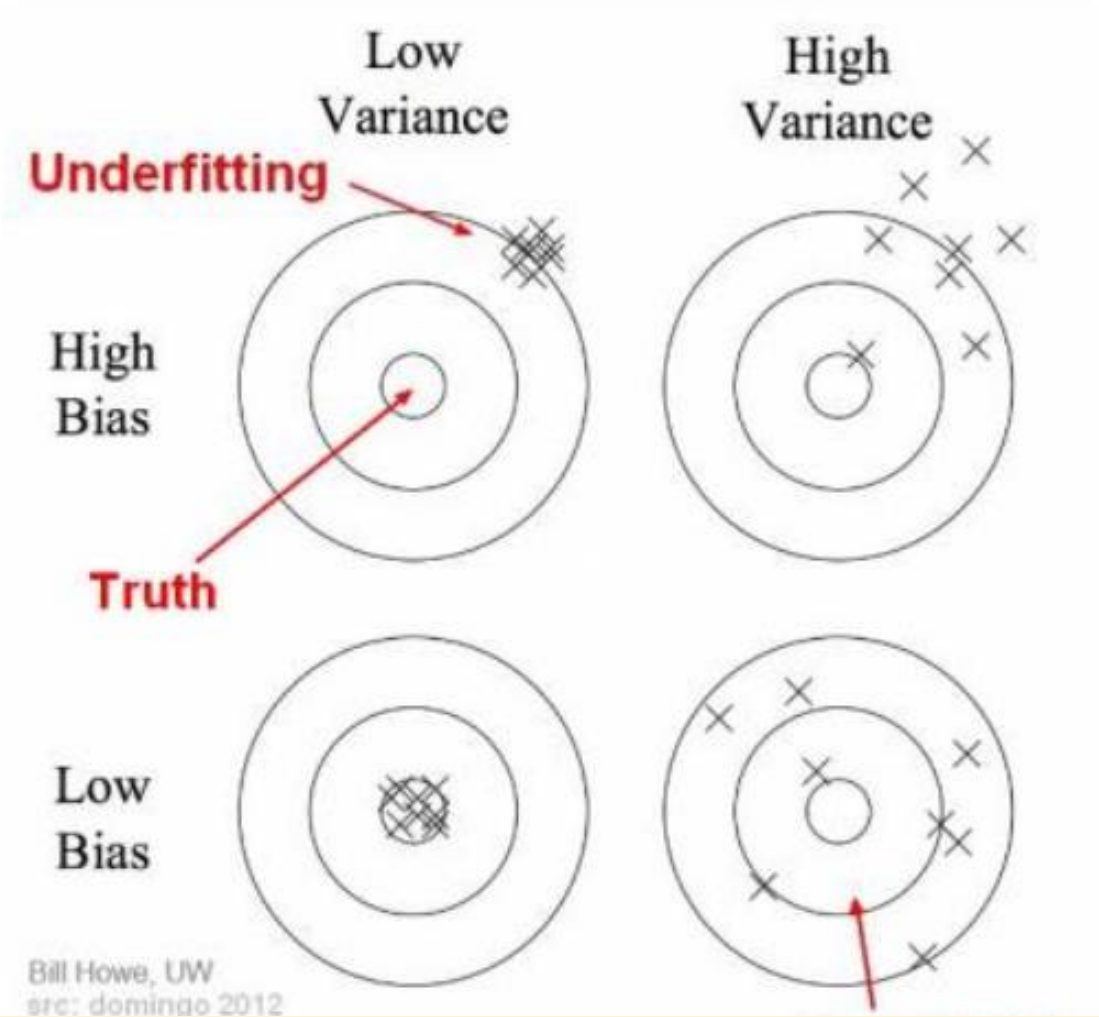
High test error due to high bias (under-fitting)



High test error due to high variance (over-fitting)



Overfitting Underfitting



In supervised learning, **underfitting** happens when a model unable to capture the underlying pattern of the data. These models usually have **high bias and low variance**. It happens when we have very less amount of data to build an accurate model or when we try to build a linear model with a nonlinear data

In supervised learning, **overfitting** happens when our model captures the noise along with the underlying pattern in data. It happens when we train our model a lot over noisy dataset. These models have **low bias and high variance**. These models are very complex like Decision trees which are prone to overfitting.

Bias and Variance trade-off

How to deal with it practically in DLL

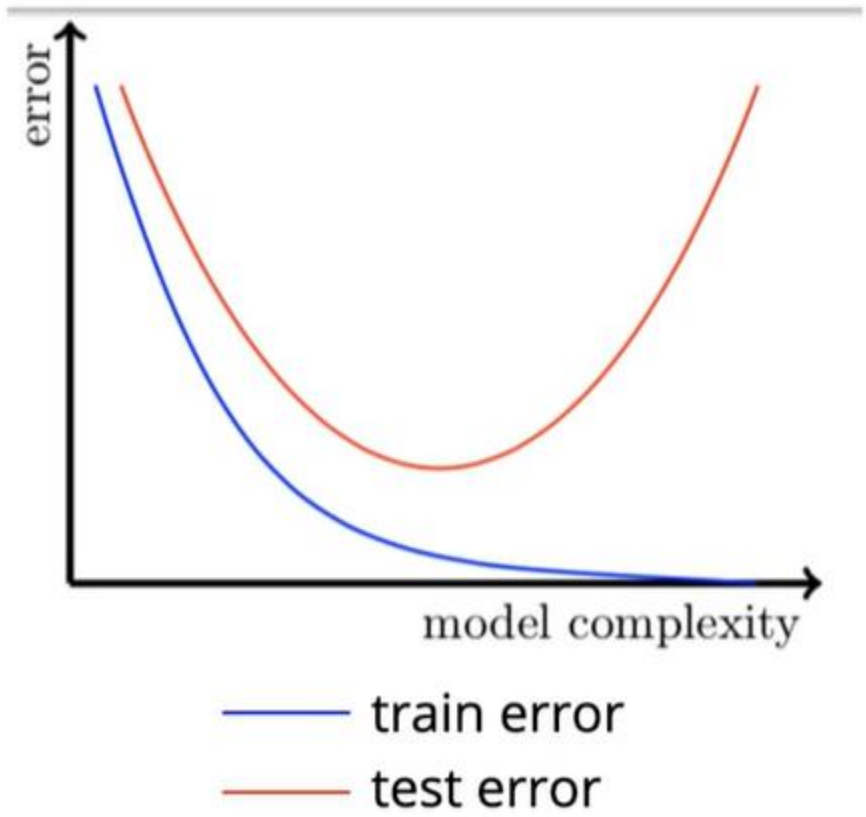
- Divide data into train, test and validation/development splits
- Start with some network configuration (say, 2 hidden layers, 50 neurons each)
- Make sure that you are using the
  - right activation function (tanh, ReLU, leaky ReLU)
  - right initialization method (He, Xavier) and
  - right optimization method (say, Adam)
- Monitor training and validation error (do not touch the test data)

Training Error	Valid Error	Cause	Solution
High	High	High bias	- Increase model complexity - Train for more epochs
Low	High	High variance	- Add more training data (e.g., **dataset augmentation**) - Use **regularization** - User **early stopping** (train less)
Low	Low	Perfect tradeoff	- You are done!

Under-fitting

Over-fitting

Data Augmentation



- Easy to drive training error to zero if data is less (too many parameters for very little data)
- Augmenting with more data will make it harder to drive the training data to zero
- By augmenting more data, we might also end up seeing data which is similar to valid/test data (hence, effectively reduce the valid/test data)

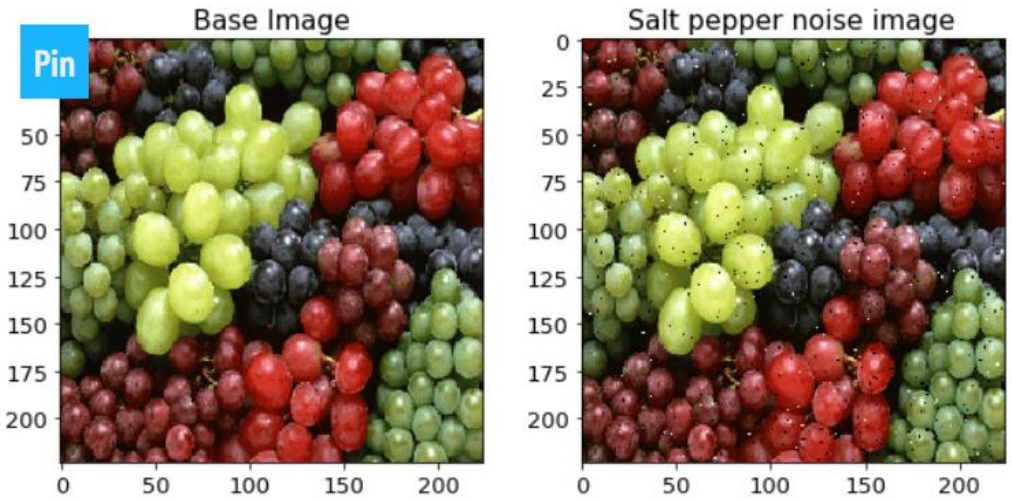
Data Augmentation

Data augmentation techniques generate different versions of a real dataset artificially to increase its size. Computer vision and natural language processing (NLP) models use data augmentation strategy to handle with data scarcity and insufficient data diversity.

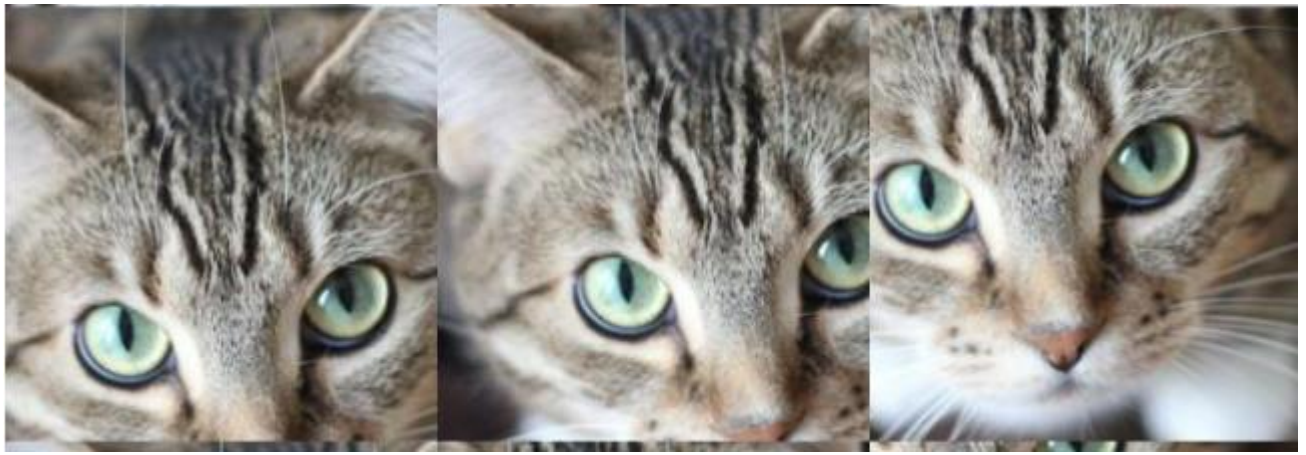
Data augmentation algorithms can increase accuracy of machine learning models. According to an experiment, a deep learning model after image augmentation performs better in training loss (i.e. penalty for a bad prediction) & accuracy and validation loss & accuracy than a deep learning model without augmentation for image classification task.

Data Augmentation

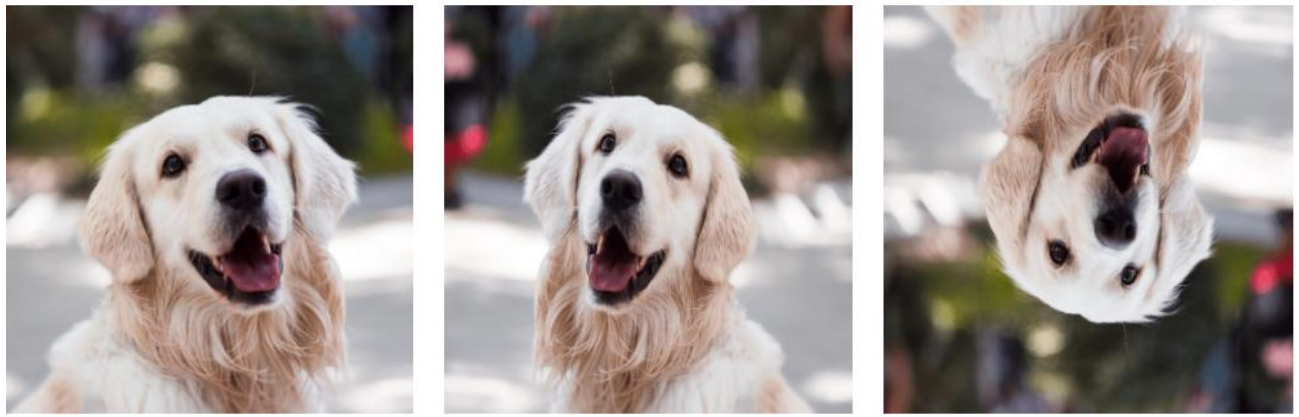
Adding noise



A section of the image is selected, cropped and then resized to the original image size

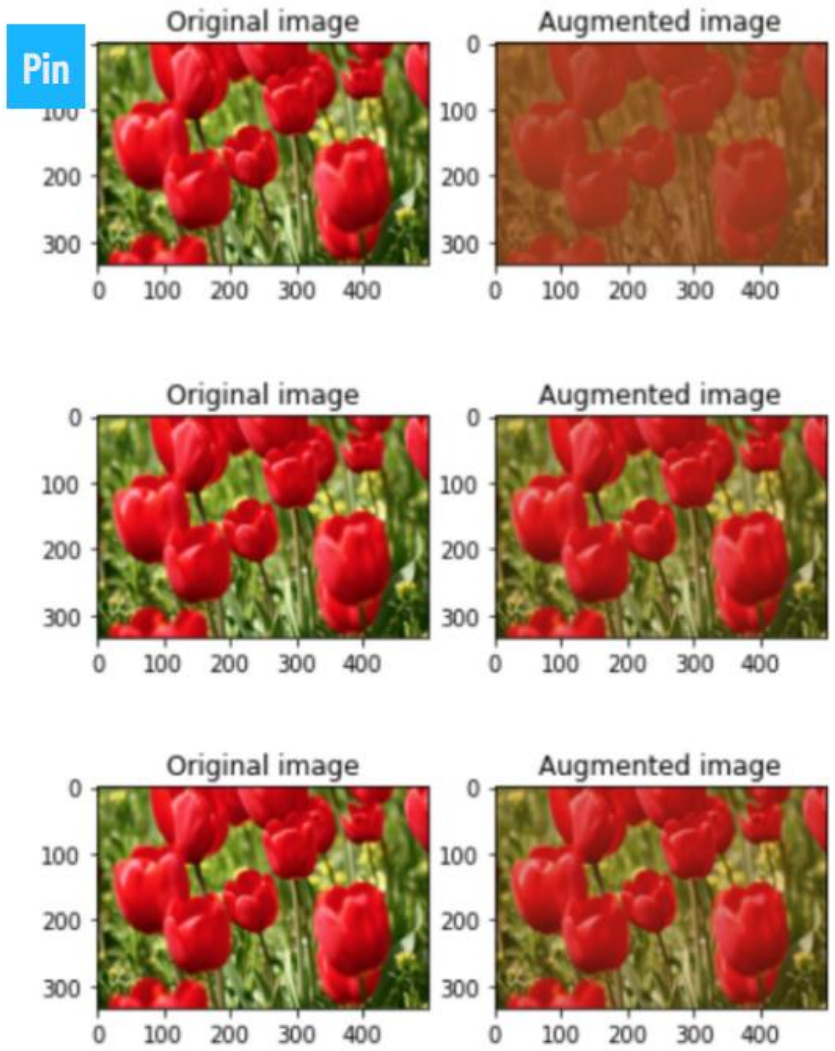


The image is flipped horizontally and vertically

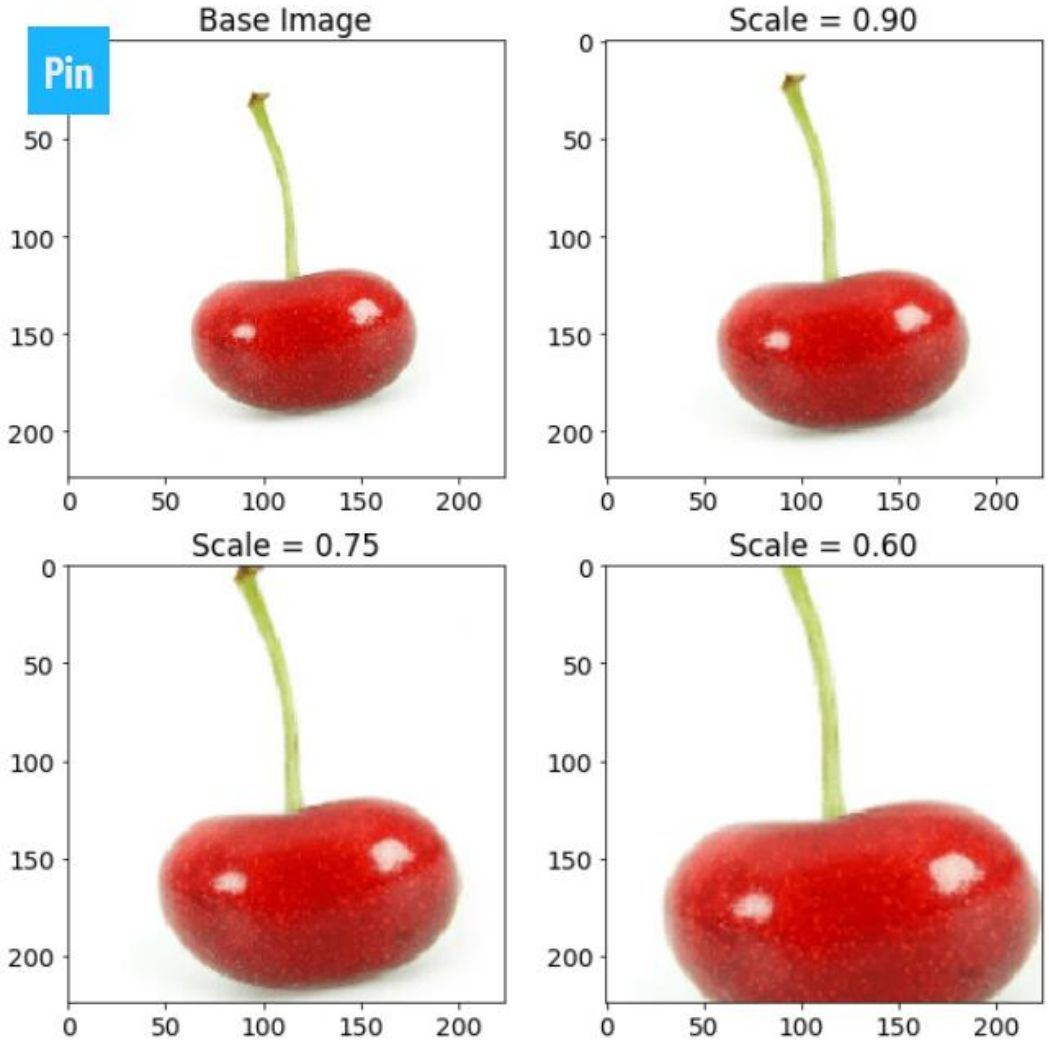


Translation

Data Augmentation



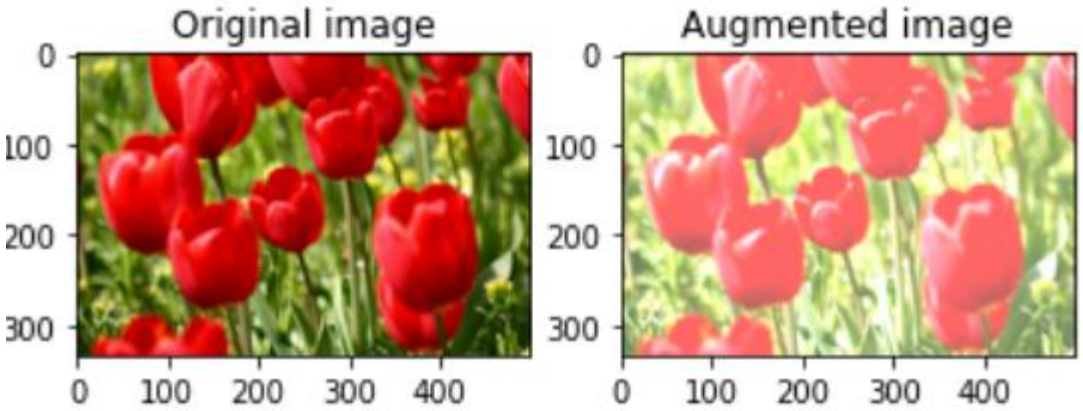
Contrast changed

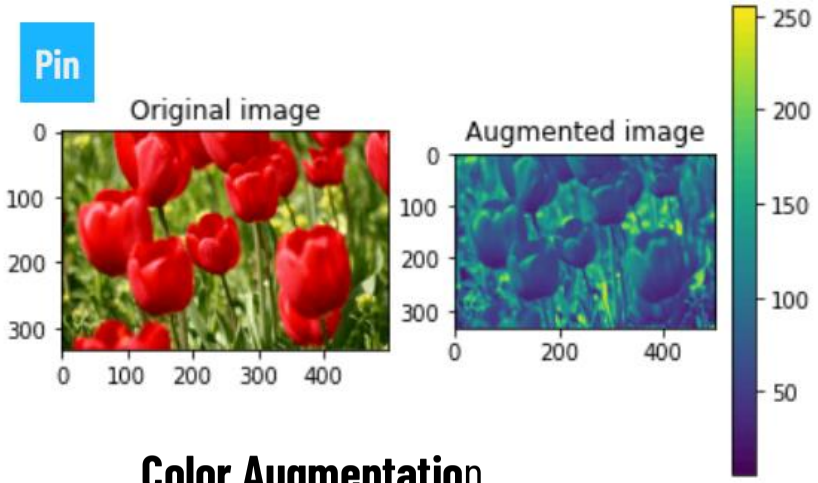


Scaling

Data Augmentation

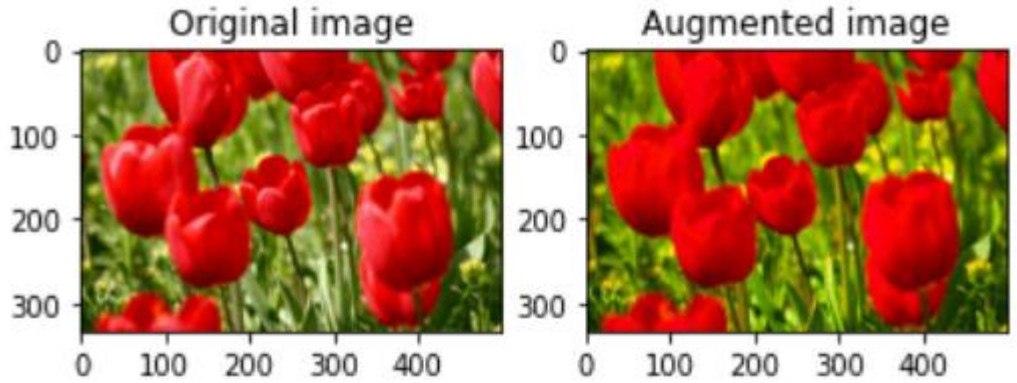
The brightness of the image is changed and new image will be darker or lighter.





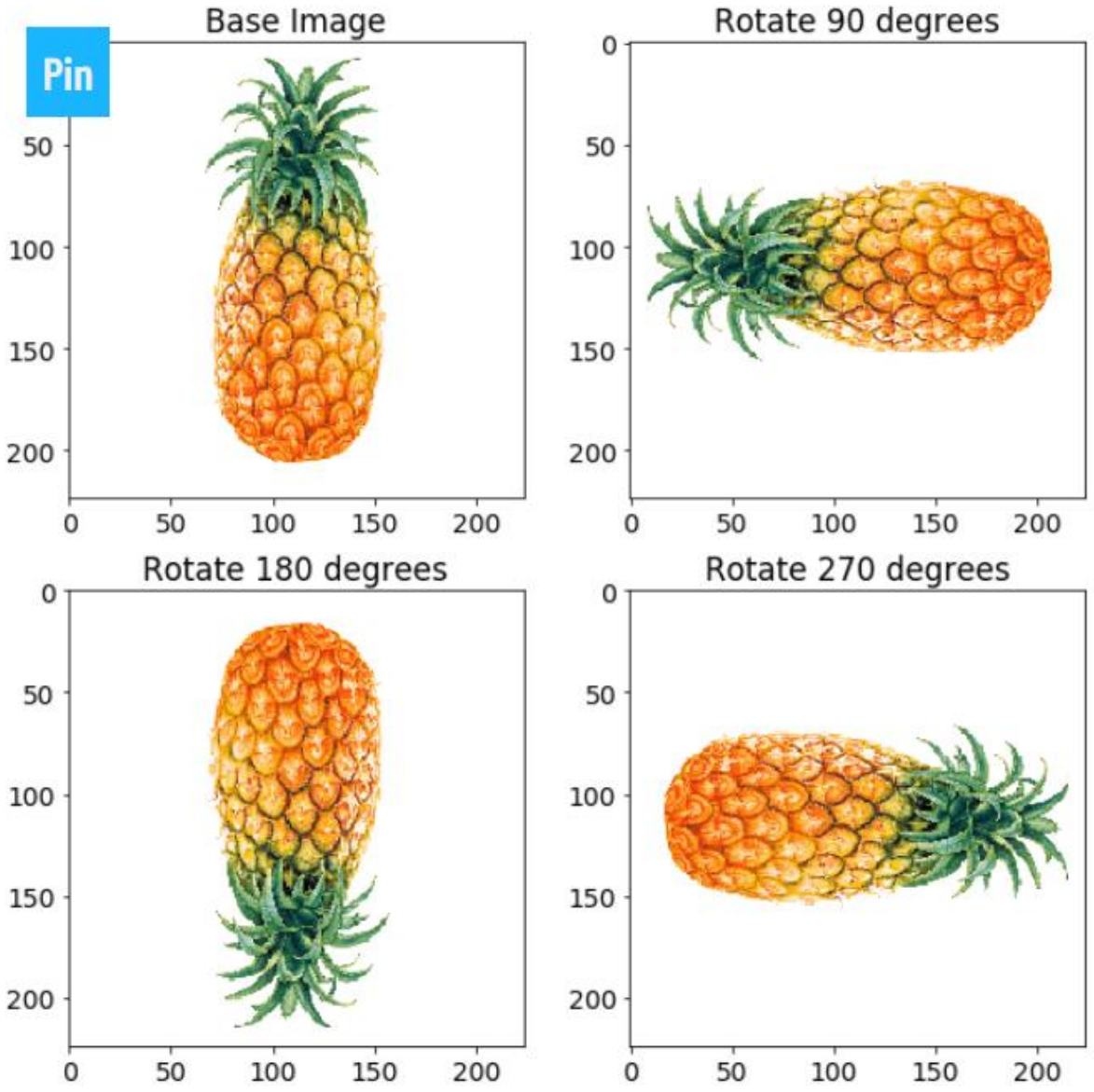
Color Augmentation

Saturation is depth or intensity of color in an image. Image is saturated

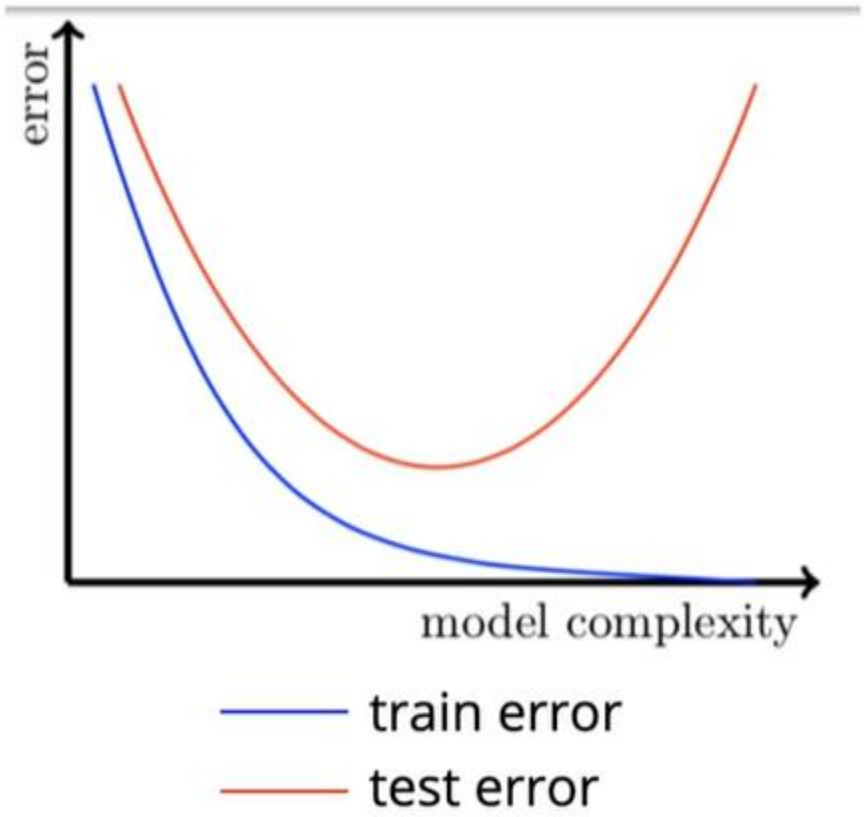


Data Augmentation

Rotation



Regularization- Intuition behind L2 regularization



$$\mathscr{L}_{train}(\theta) = \sum_{i=1}^{N}(y_i - f(x_i))^2$$

$$\min_{\theta} \mathscr{L}(\theta) = \mathscr{L}_{train}(\theta) + \Omega(\theta)$$

$$\theta = [W_{111}, W_{112} + \cdots + W_{Lnk}]$$

$$\Omega(\theta) = ||\theta||_2^2$$

$$= W_{111}^2 + W_{112}^2 + \cdots + W_{Lnk}^2$$

sgd = torch.optim.SGD(model.parameters(), weight_decay=weight_decay)

pytorch optimizers have a parameter called weight_decay which corresponds to the L2 regularization factor:

Initialise w, b

Iterate over data:

compute $\hat{y}$

compute $\mathscr{L}(w, b)$

$w_{111} = w_{111} - \eta \Delta w_{111}$

$w_{112} = w_{112} - \eta \Delta w_{112}$

....

$w_{313} = w_{313} - \eta \Delta w_{313}$

till satisfied

$$\min_{\theta} \mathscr{L}(\theta) = \mathscr{L}_{train}(\theta) + \Omega(\theta)$$

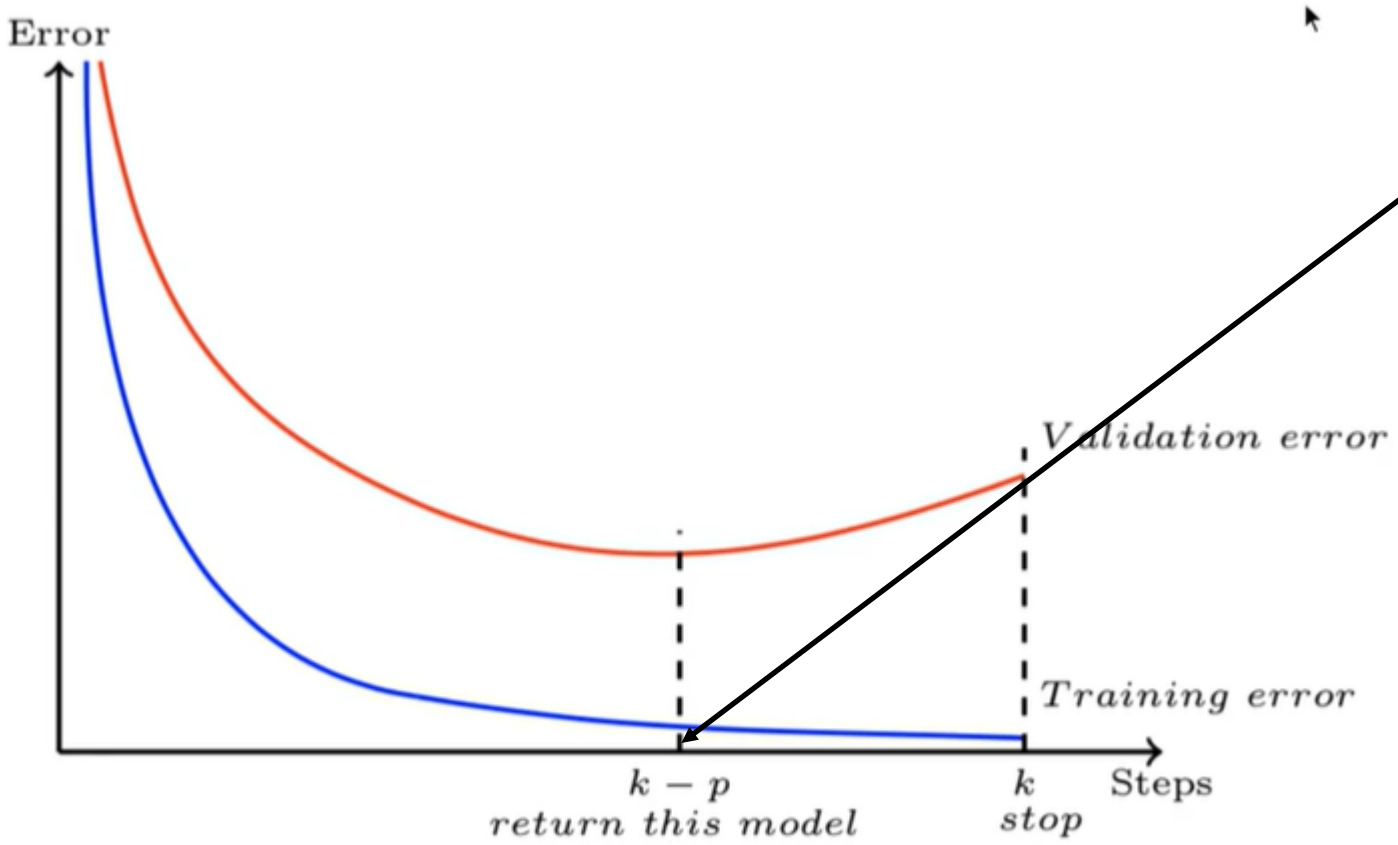
$$\mathscr{L}_{train}(\theta) = \sum_{i=1}^{N} (y_i - f(x_i))^2$$

$$\Omega(\theta) = W_{111}^2 + W_{112}^2 + \cdots + W_{Lnk}^2$$

$$\Delta W_{ijk} = \frac{\partial \mathscr{L}(\theta)}{\partial W_{ijk}}$$

$$= \frac{\partial \mathscr{L}_{train}(\theta)}{\partial W_{ijk}} + \frac{\partial \Omega(\theta)}{\partial W_{ijk}}$$

Regularization- Early Stopping



After certain no of epoch if validation error is increasing through training error decreasing, it means model is getting over fitted. So we need to stop the training

Stopping Criteria: Training is stopped as soon as the performance on the validation dataset decreases as compared to the performance on the validation dataset at the prior training epoch (e.g. an increase in loss).

Batch Normalisation
Drop out

Normalizing the input features

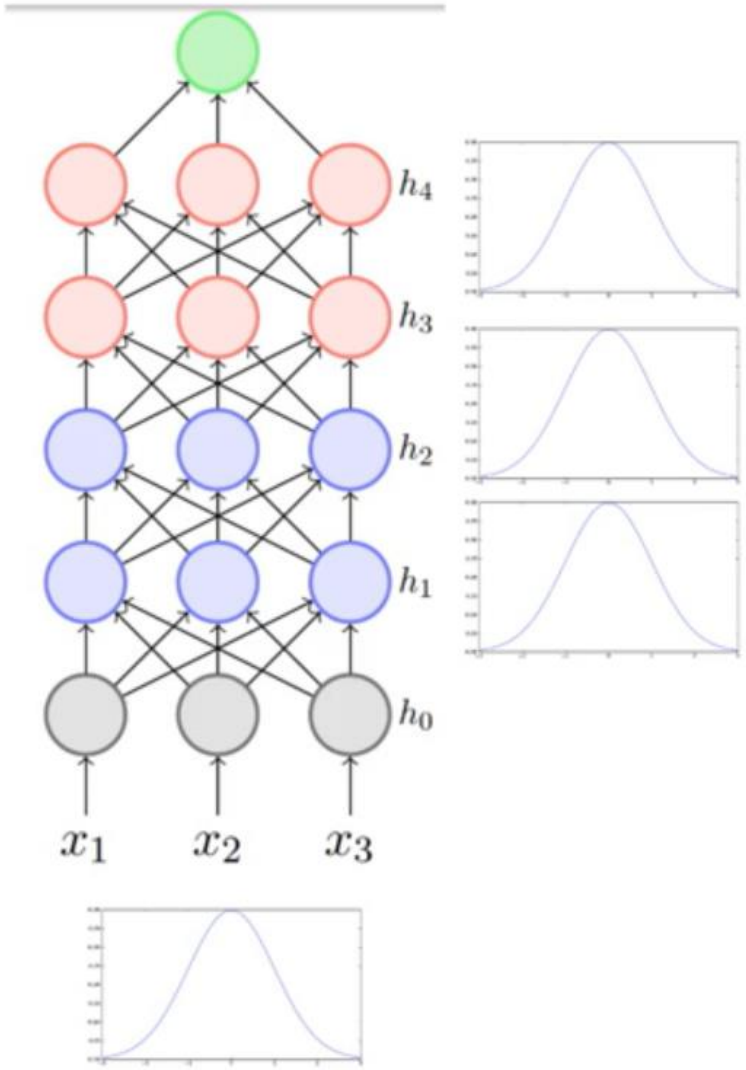
									
Launch (within 6 months)	0	1	1	0	0	1	0	1	1
Weight (g)	151	180	160	205	162	182	138	185	170
SAR Value	0.64	0.87	0.67	0.88	0.7	0.91	0	1	0.47
dual sim	1	1	0	0	0	1	0	1	0
Internal memory (>= 64 GB, 4GB RAM)	1	1	1	1	1	1	1	1	1
NFC	0	1	1	0	1	0	1	1	1
Radio	1	0	0	1	1	1	0	0	0
Battery(mAh)	3060	3500	3060	5000	3000	4000	1960	3700	3260
Price (INR)	15k	32k	25k	18k	14k	12k	35k	42k	44k
Like (*y*)	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**0**

$$x_{ij}^{norm} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij}$$

$$\sigma_j = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_{ij} - \mu_j)^2}$$

Normalisation of hidden layers



$$h_{ij}^{norm} = \frac{h_{ij} - \mu_j}{\sigma_j}$$

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} h_{ij}$$

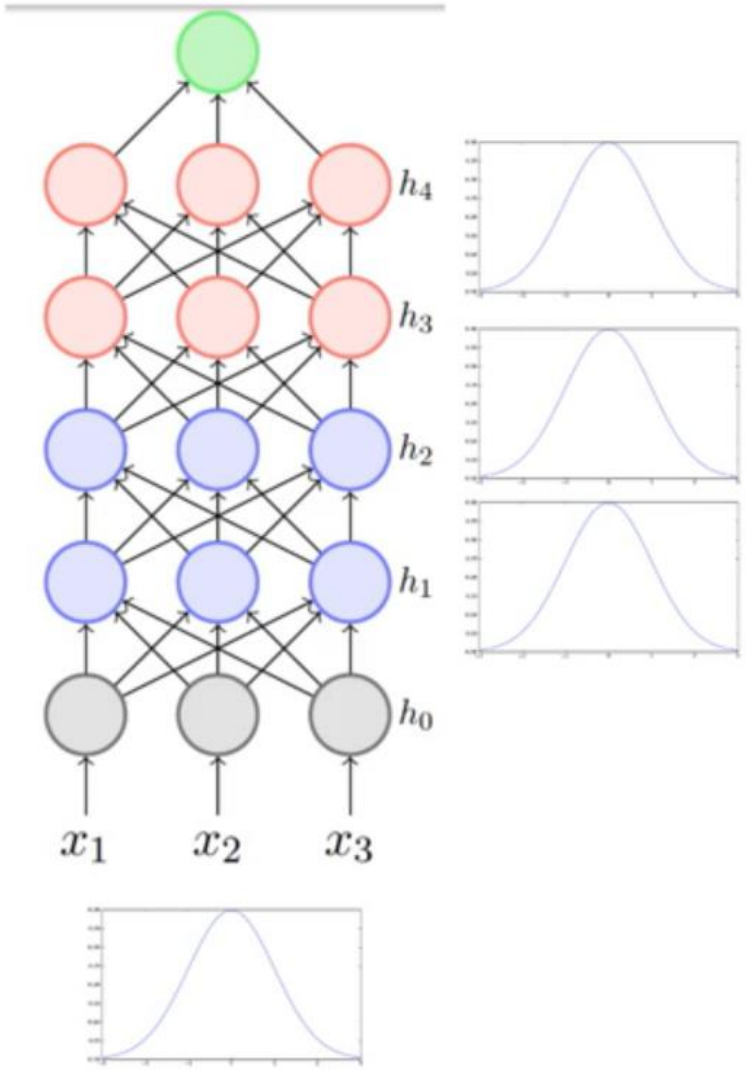
$$\sigma_j = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(h_{ij} - \mu_j)^2}$$

Why Batch normalization?

Because you compute μ and σ from a single batch as opposed to computing it from the entire data

Inputs are standardised. What about hidden layers? hij?

Flexibility to the network to decide whether normalization is required or not



γ_j and β_j are learned along with other parameters of the network

$$h_{ij}^{norm} = \frac{h_{ij} - \mu_j}{\sigma_j}$$

Give flexibility to the network to decide whether normalization is required or not. It learns accordingly parameters γ_j and β_j

$$h_{ij}^{final} = \gamma_j \cdot h_{ij}^{norm} + \beta_j$$

If

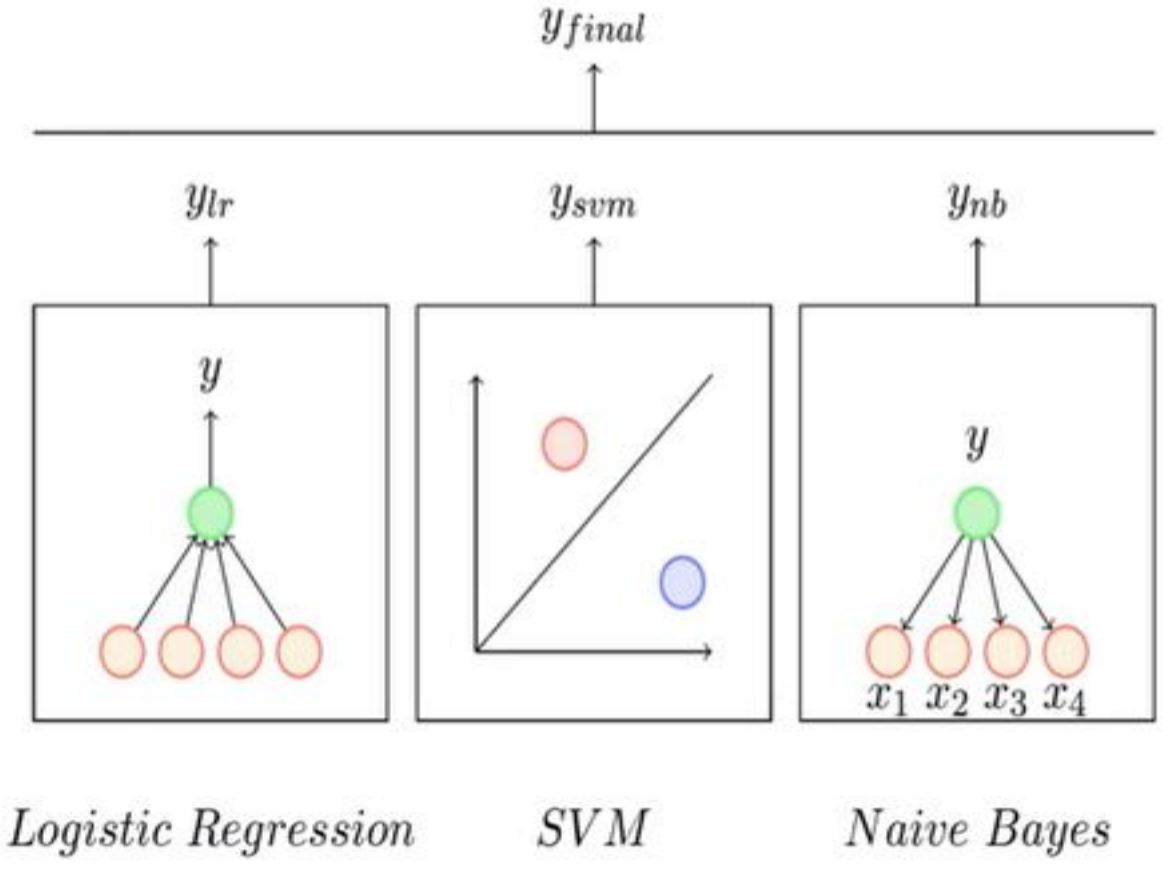
$\gamma_j = \sigma_j$

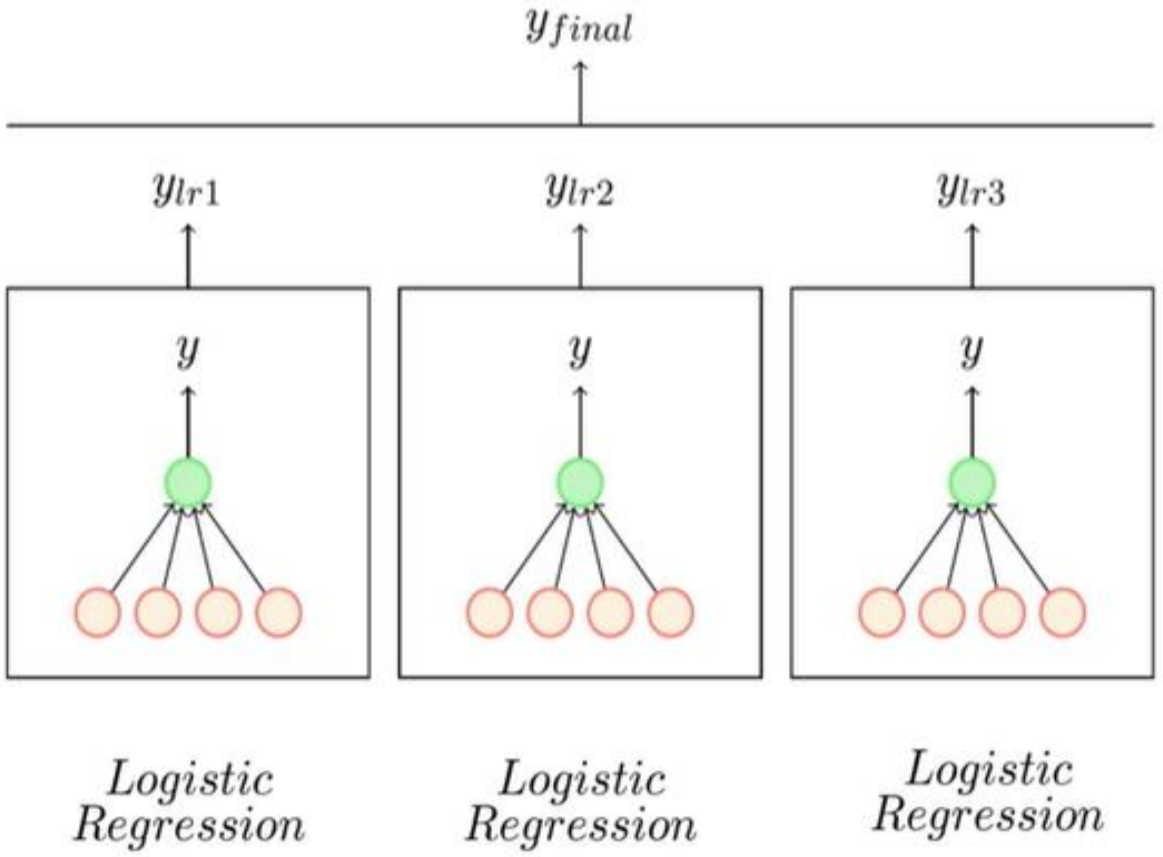
$\beta_j = \mu_j$

then

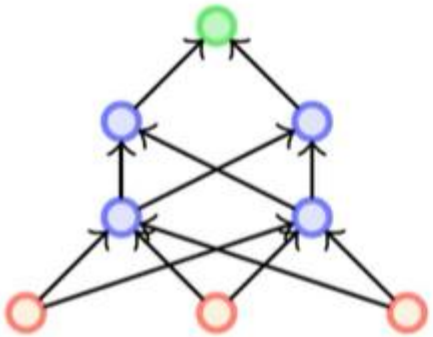
$$h_{ij}^{final} = h_{ij}^{norm}$$

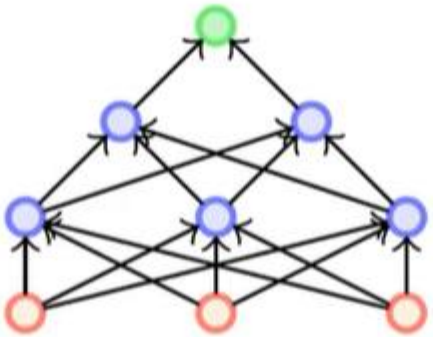
Ensemble methods (rely on multiple models than a single model)

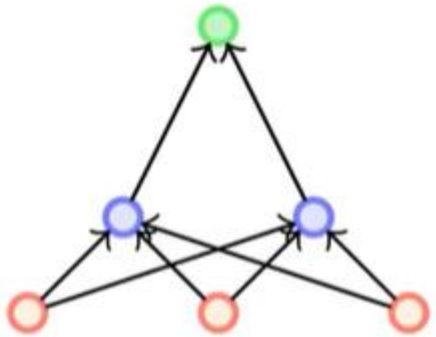




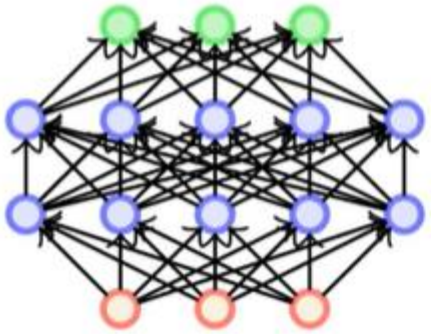
Ensemble of Neural Networks

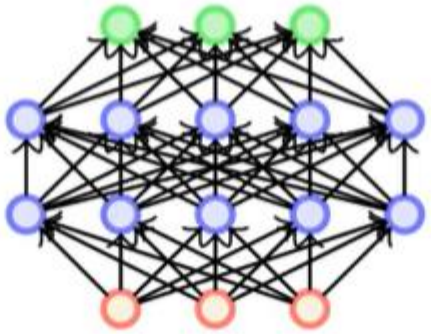


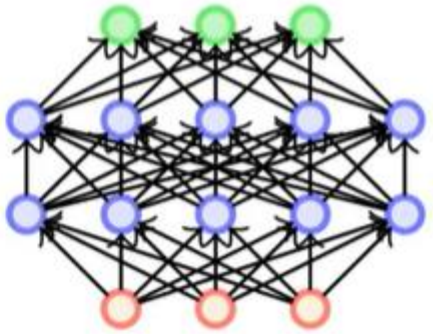




Option 1: Train different architectures (models) on the same data - **Expensive**



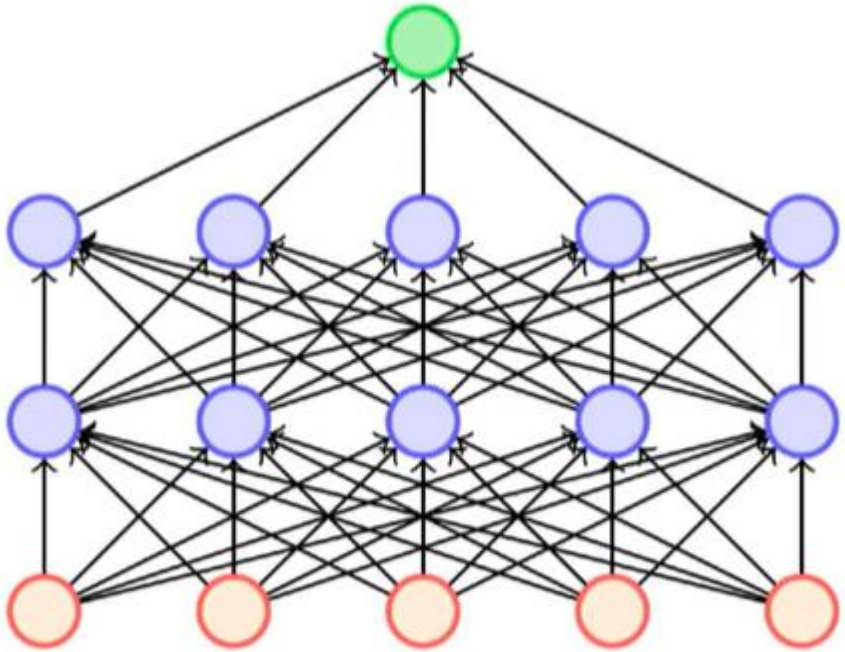




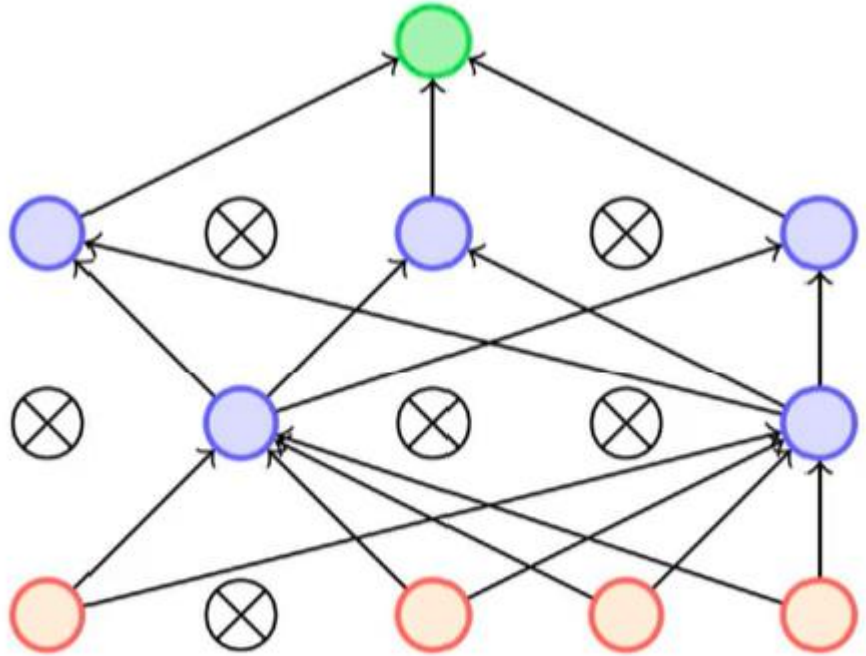
Option 2: Train same architecture (model) on different subsets of the training data - **Expensive**

Idea of Drop Out

How to choose which nodes to be included?
For each node find np.random(0,1) if u get a value <0.5 drop the node. If u get value>0.5 include the node



Original network



Network with some nodes dropped out

If I have n nodes in a neural network I can create 2^n number of neural networks with some weights dropped out

Trick :

- Share parameters across all these neural networks
- Sample a different neural network for each training instance

Training Procedure

How to choose which nodes to be included?
For each node find np.random(0,1) if u get a value <0.5 drop the node. If u get value>0.5 include the node

Dropout

Initialise parameters

Iterate over data:

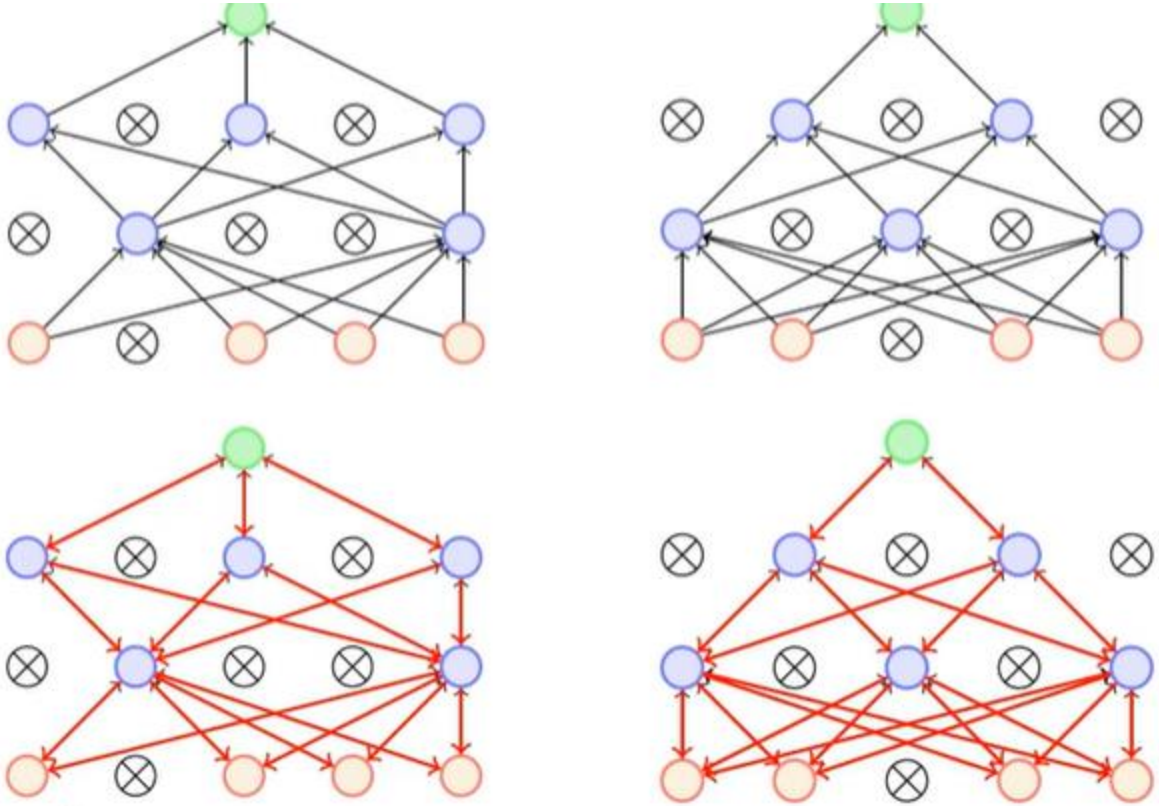
X_i, Y_i = current mini_batch

NN = dropout Network

$\mathcal{L}(\theta)$ = compute_loss (NN, X_i, Y_i)

back_propagate(NN)

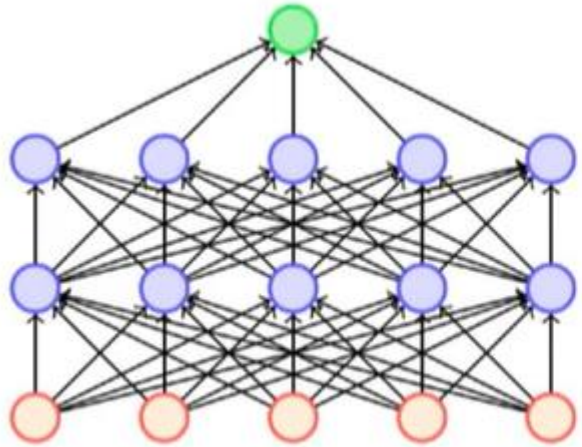
till satisfied

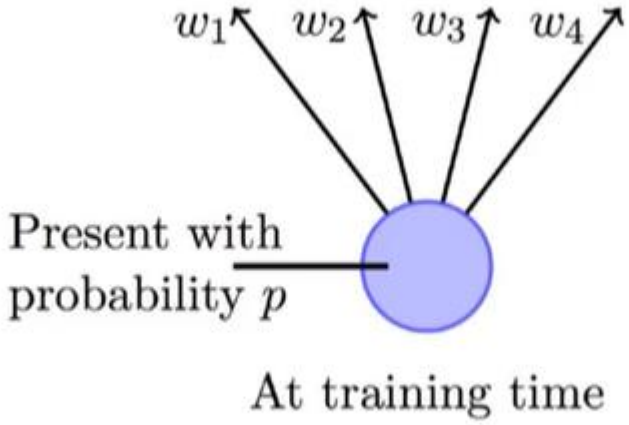


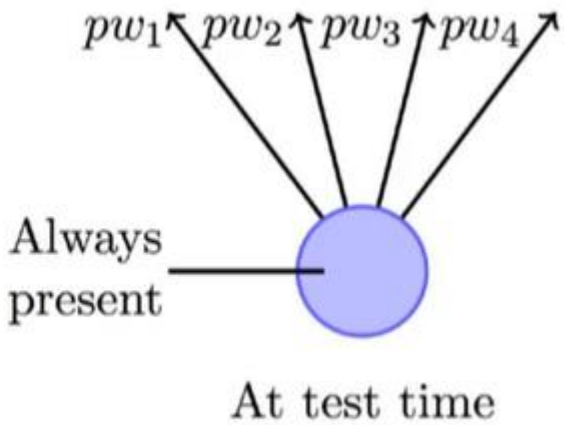
Weights are being shared!

Using drop out at test time

Intuition:
Scale the output of each neuron by p







It is computationally expensive to get the output of all network and aggregate. Each node will be present with a probability p Hence in testing, we scale up output of each neuron by p so that the contribution of that node towards the output can be controlled. Hence complete the inference

Intuition:
- Acts as a regularizer by introducing noise
- Prevents co-adaptation

Pytorch : Dropout

```
model_dropout = torch.nn.Sequential(
    torch.nn.Linear(1, N_h),
    torch.nn.Dropout(0.2),
    torch.nn.ReLU(),
    torch.nn.Linear(N_h, N_h),
    torch.nn.Dropout(0.2),
    torch.nn.ReLU(),
    torch.nn.Linear(N_h, 1),
)
```

Adding dropout to your PyTorch models with the torch.nn.Dropout class, which takes in the dropout rate – the probability of a neuron being deactivated – as a parameter.

self.dropout = nn.Dropout(0.25)

Pytorch BatchNorm

Two-dimensional Batch Normalization is made available by nn.BatchNorm2d.

For one-dimensional Batch Normalization, you can use nn.BatchNorm1d.

CNN

```
class CNN_BN(nn.Module):
    def __init__(self):
        super(MyNetBN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 3, 5),
            nn.ReLU(),
            nn.AvgPool2d(2, stride=2),
            nn.Conv2d(3, 6, 3),
            nn.BatchNorm2d(6)
        )
        self.features1 = nn.Sequential(
            nn.ReLU(),
            nn.AvgPool2d(2, stride=2)
        )
        self.classifier = nn.Sequential(
            nn.Linear(150, 25),
            nn.ReLU(),
            nn.Linear(25,10)
        )
```

6 no of input

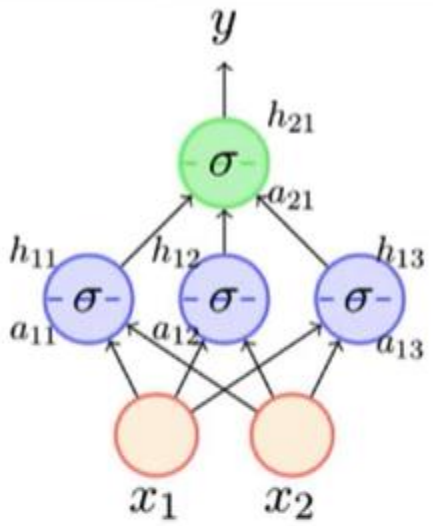
MLP

```
class MLP(nn.Module):
  '''
    Multilayer Perceptron.
  '''
  def __init__(self):
    super().__init__()
    self.layers = nn.Sequential(
      nn.Flatten(),
      nn.Linear(32 * 32 * 3, 64),
      nn.BatchNorm1d(64),
      nn.ReLU(),
      nn.Linear(64, 32),
      nn.BatchNorm1d(32),
      nn.ReLU(),
      nn.Linear(32, 10)
    )
```

One-dimensional BatchNormalization (nn.BatchNorm1d) applies Batch Normalization over a 2D or 3D input (a batch of 1D inputs with a possible channel dimension).

Two-dimensional BatchNormalization (nn.BatchNorm2d) applies it over a 4D input (a batch of 2D inputs with a possible channel dimension).

Initialization of Weights and Bias



Initialise w, b

Iterate over data:

compute $\hat{y}$

compute $\mathscr{L}(w, b)$

$w_{11} = w_{11} - \eta \Delta w_{11}$

$w_{12} = w_{12} - \eta \Delta w_{12}$

... ...

till satisfied

$$\nabla w_{11} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \cdot \frac{\partial y}{\partial h_{11}} \cdot \frac{\partial h_{11}}{\partial a_{11}} \cdot x_1$$

$$\nabla w_{21} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \cdot \frac{\partial y}{\partial h_{12}} \cdot \frac{\partial h_{12}}{\partial a_{12}} \cdot x_1$$

$$but \quad h_{11} = h_{12}$$

$$and \quad a_{12} = a_{12}$$

$$\therefore \nabla w_{11} = \nabla w_{21}$$

✓ never initialize all weights to same value

$$a_{11} = w_{11}x_1 + w_{12}x_2$$

$$a_{12} = w_{21}x_1 + w_{22}x_2$$

$$\therefore a_{11} = a_{12} = 0$$

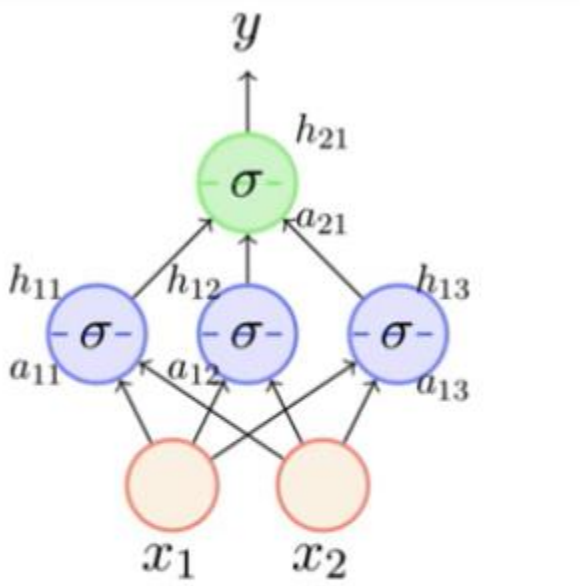
$$\therefore h_{11} = h_{12}$$

✗ This symmetry will never break during training (**symmetry breaking problem**)

✗ Hence **weights** connected to the same neuron should **never be initialized to the same value**

✓ never initialize all weights to 0

Initialization of Weights and Bias

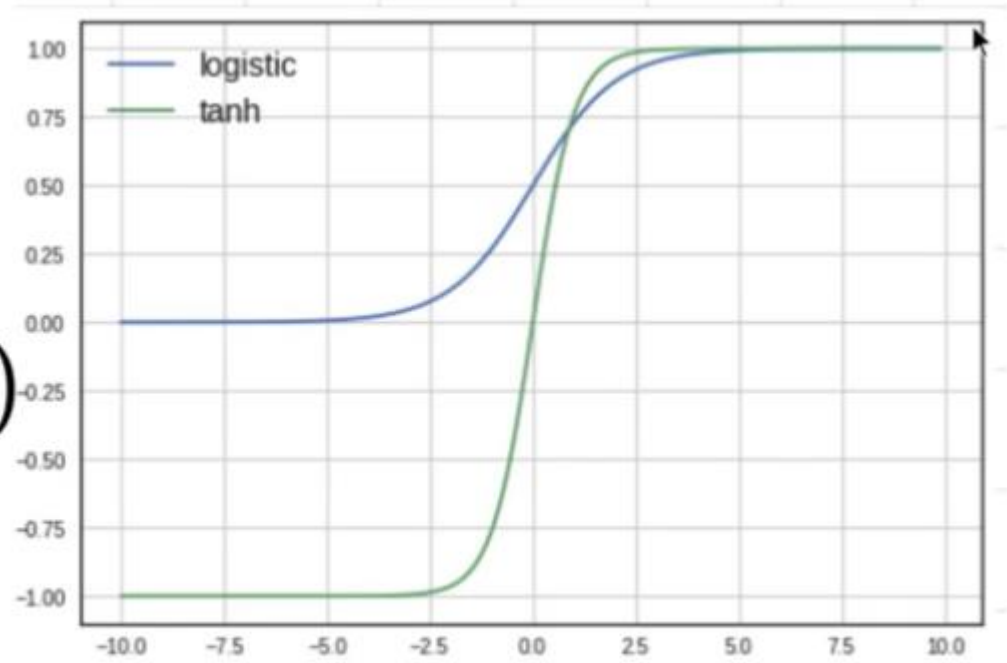


$$a_{11} = w_{11}x_1 + w_{12}x_2$$

$$a_{12} = w_{21}x_1 + w_{22}x_2$$

- always normalize the inputs (so that they lie between 0 to 1)
- never initialize weights to large values

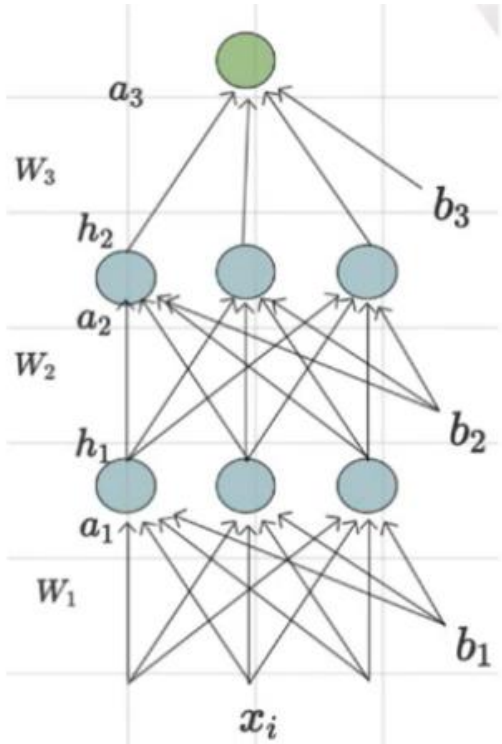
$$h_{11} = \sigma(a_{11})$$



a_{11}

Blowing up a11 cause vanishing gradient

Xavier Initialization – Softmax and tanh activation



$$a_2 = W_2 x + b_2$$

$$h_2 = \sigma(a_2)$$

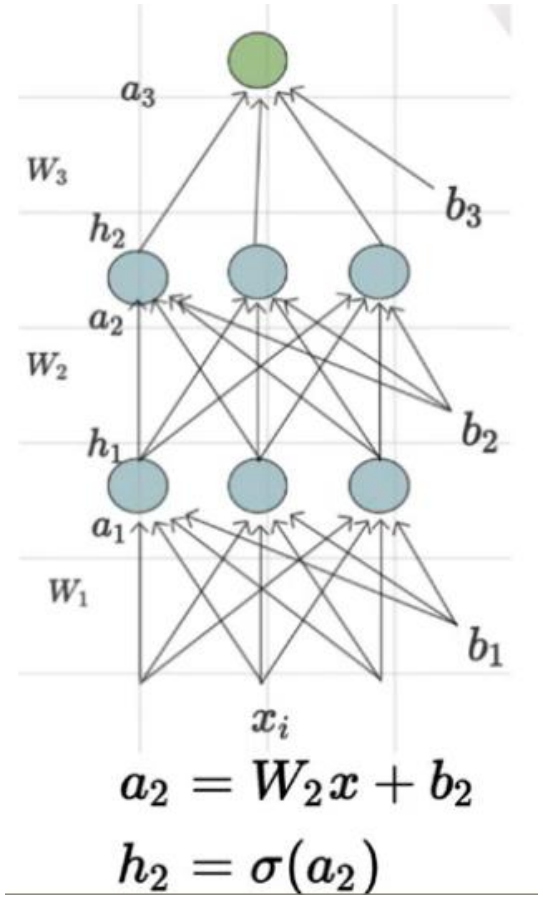
Xavier initialization sets a layer's weights to values chosen from a random uniform distribution that's bounded between

$$\pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}$$

where n_i is the number of incoming network connections, or "fan-in," to the layer, and n_{i+1} is the number of outgoing network connections from that layer, also known as the "fan-out."

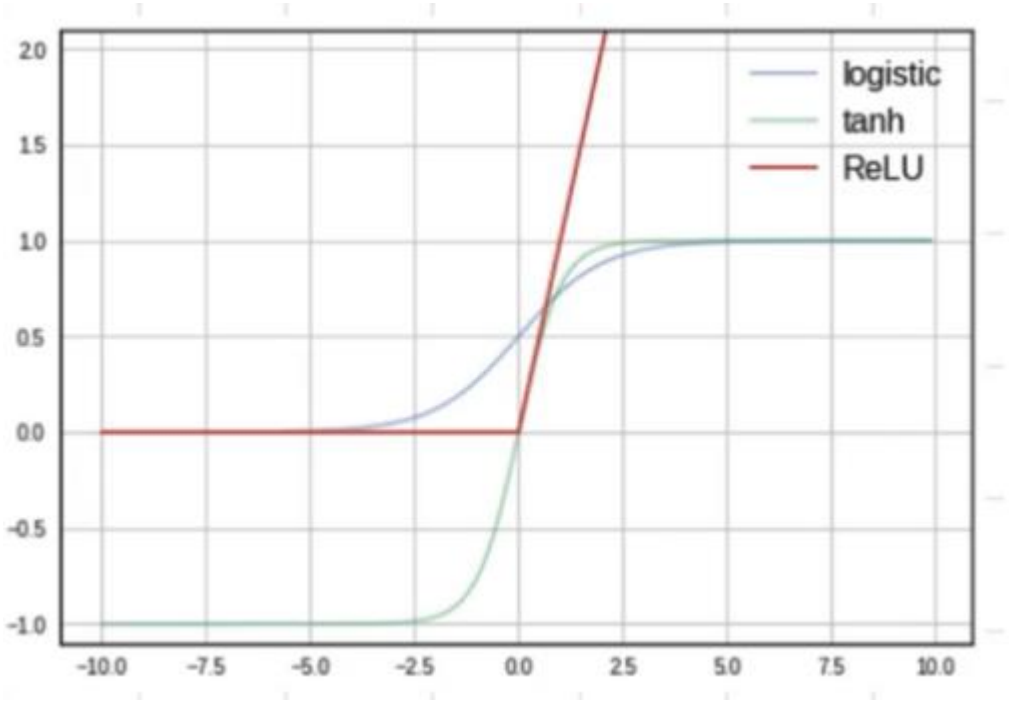
He Initialization ReLu activation

The **he** initialization method is calculated as a random number with a Gaussian probability distribution (G) with a mean of 0.0 and a standard deviation of $\sqrt{\frac{2}{n}}$, where *n* is the number of inputs to the node.



$$a_2 = W_2 x + b_2$$

$$h_2 = \sigma(a_2)$$



In ReLU half the neurons are dead hence n/2

Namah Shivaya