DSE 3121 DEEP LEARNING

Neural Network Optimization

Rohini Rao & Abhilash K Pai

Department of Data Science and Computer Applications

MIT Manipal

Credits: Most of the slides are adapted from CS7015 Deep Learning, Dept. of CSE, IIT Madras

Gradient Descent Algorithm

Algorithm: gradient_descent()

$t \leftarrow 0;$

$max\_iterations \leftarrow 1000;$

$Initialize \quad \theta_0 = [W_1^0, ..., W_L^0, b_1^0, ..., b_L^0];$

while $t{+}{+} < max\_iterations$ **do**

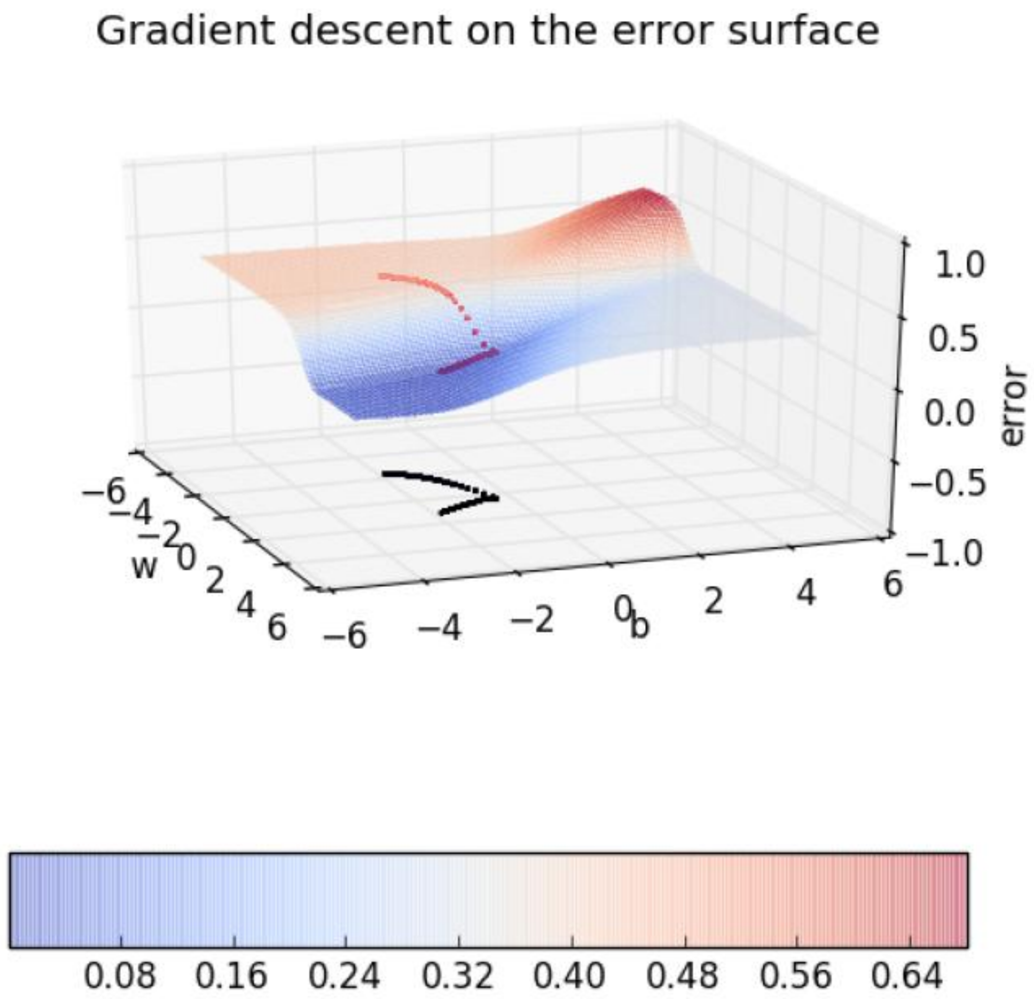
$\quad h_1, h_2, ..., h_{L-1}, a_1, a_2, ..., a_L, \hat{y} = forward\_propagation(\theta_t);$

$\quad \nabla\theta_t = backward\_propagation(h_1, h_2, ..., h_{L-1}, a_1, a_2, ..., a_L, y, \hat{y});$

$\quad \theta_{t+1} \leftarrow \theta_t - \eta\nabla\theta_t;$

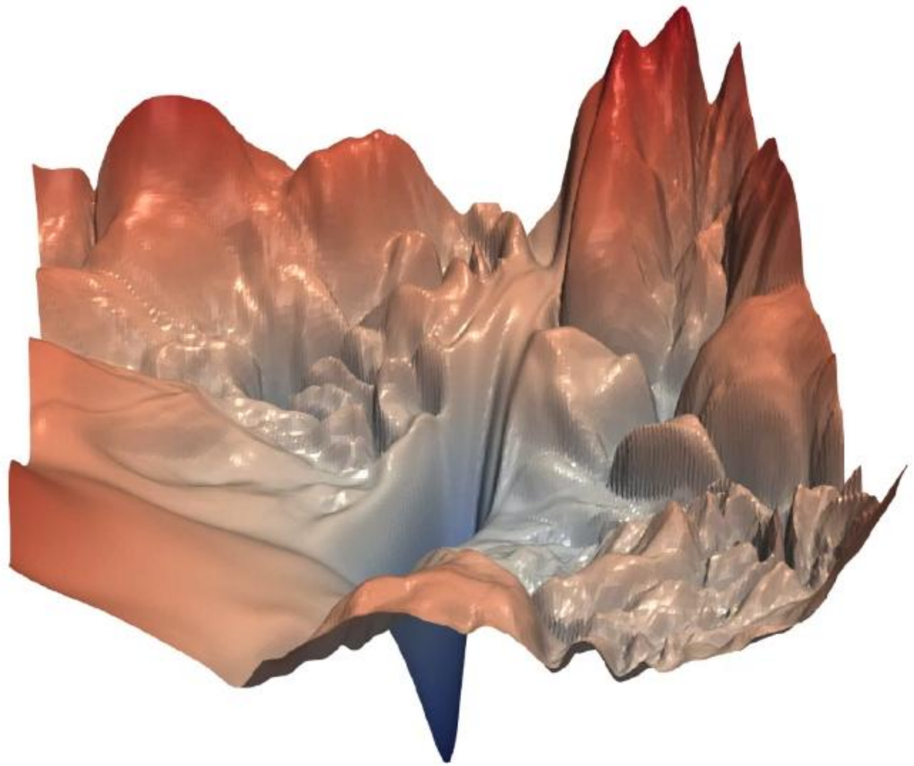
end

Gradient descent: Error surface



Gradient descent: Error surface

Visualization of error surface of a neural network (ResNet-56)



Gradient descent and its variants

Vanilla (Batch) Gradient Descent

Require: Learning rate α, initial parameters θ_t, training dataset $\mathcal{D}_{tr}$

while stopping criterion not met **do**
    Initialize parameter updates $\Delta\theta_t = 0$
    for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ **do**
        Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
        Aggregate gradient $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t}\mathcal{L}$
    end for
    Apply update $\theta_{t+1} = \theta_t - \alpha\frac{1}{|\mathcal{D}_{tr}|}\Delta\theta_t$
end while

What is the issue here?

Think of how the algorithm will work for large Dataset (Eg: ImageNet with Billions of Data)

Gradient descent and its variants

Stochastic Gradient Descent

Require: Learning rate α, initial parameters θ_t, training dataset $\mathcal{D}_{tr}$

while stopping criterion not met **do**
    for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ **do**
        Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
        Gradient $\Delta\theta_t = \nabla_{\theta_t}\mathcal{L}$
        Apply update $\theta_{t+1} = \theta_t - \alpha\Delta\theta_t$
    end for
end while

Gradient descent and its variants

Mini-Batch Stochastic Gradient Descent

Require: Learning rate α, initial parameters θ_t, mini-batch size m, training dataset $\mathcal{D}_{tr}$

```
while stopping criterion not met do
    Initialize gradients Δθ_t = 0
    Sample m examples from D_tr (call it D_mini)
    for each (x^(i), y^(i)) in D_mini do
        Compute gradient using backpropagation ∇_{θ_t} L(θ_t; x^(i), y^(i))
        Aggregate gradient Δθ_t = Δθ_t + ∇_{θ_t} L
    end for
    Apply update θ_{t+1} = θ_t − αΔθ_t
end while
```

Optimizers

- Optimizer
  - is a function or an algorithm that modifies the attributes of the neural network, such as weights and learning rate.
- Terminology
  - Weights/ Bias – The learnable parameters in a model that controls the signal between two neurons.
  - Epoch – The number of times the algorithm runs on the whole training dataset.
  - Sample – A single row of a dataset.
  - Batch –denotes the number of samples to be taken for updating the model parameters.
  - Learning rate –defines a scale of how much model weights should be updated.
  - Cost Function/Loss Function -is used to calculate the cost that is the difference between the predicted value and the actual value.

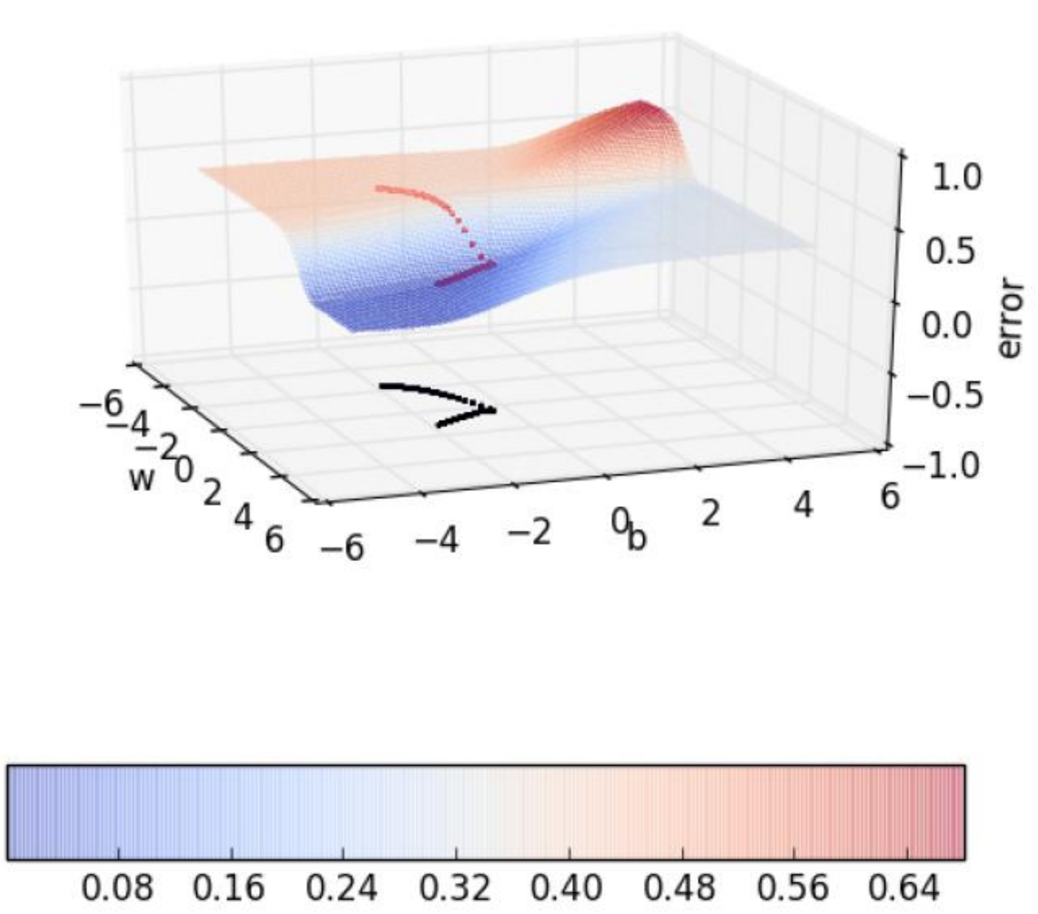
Mini Batch Gradient Descent Deep Learning Optimizer

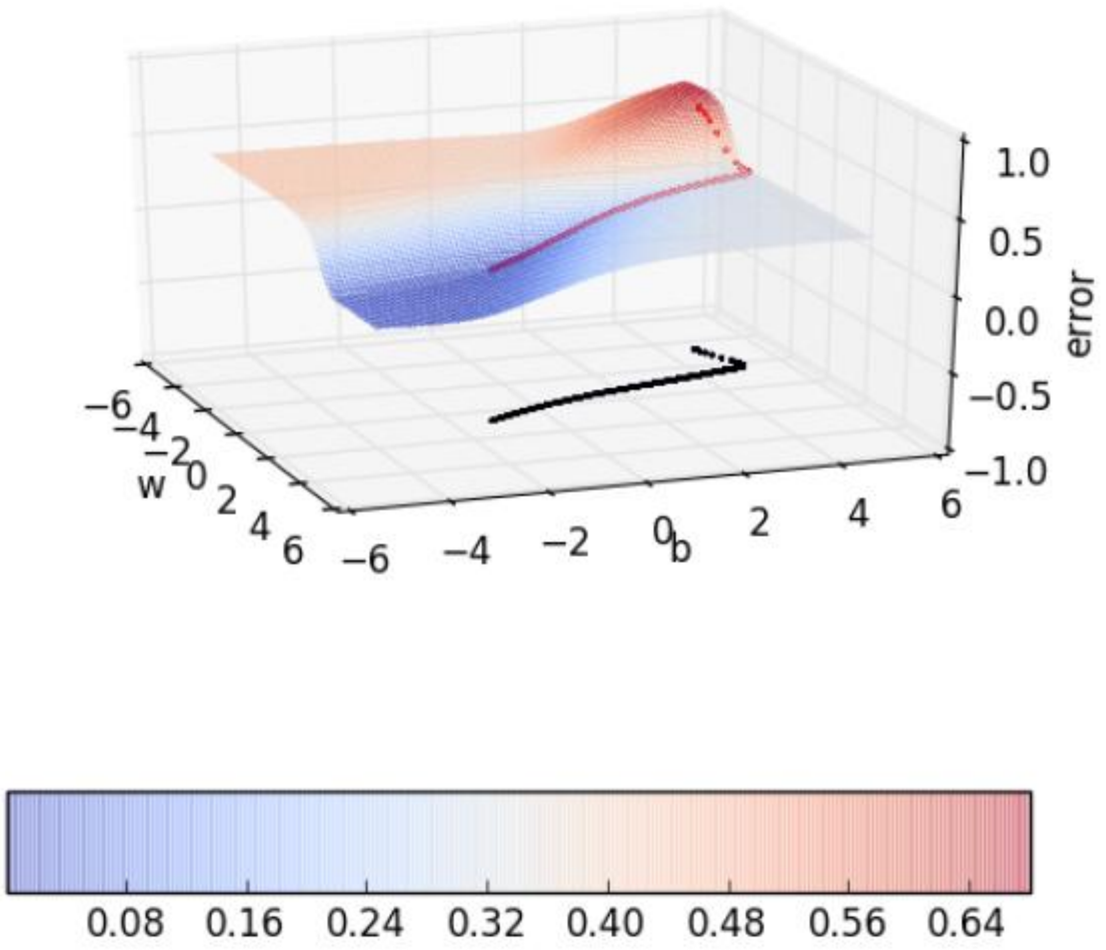
- **Batch gradient descent**:
  - gradient is average of gradients computed from ALL the samples in dataset
- Mini Batch GD:
  - subset of the dataset is used for calculating the loss function, therefore fewer iterations are needed.
  - batch size of 32 is considered to be appropriate for almost every case.
  - Yann Lecun (2018) – “Friends don’t let friends use mini batches larger than 32”
- is faster , more efficient and robust than the earlier variants of gradient descent.
- the cost function is noisier than the batch GD but smoother than SDG.
- Provides a good balance between speed and accuracy.
- It needs a hyperparameter that is “mini-batch-size”, which needs to be tuned to achieve the required accuracy.

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

Algorithm	# of steps in 1 epoch
Vanilla (Batch) Gradient Descent	1
Stochastic Gradient Descent	N
Mini-Batch Gradient Descent	$\frac{N}{B}$

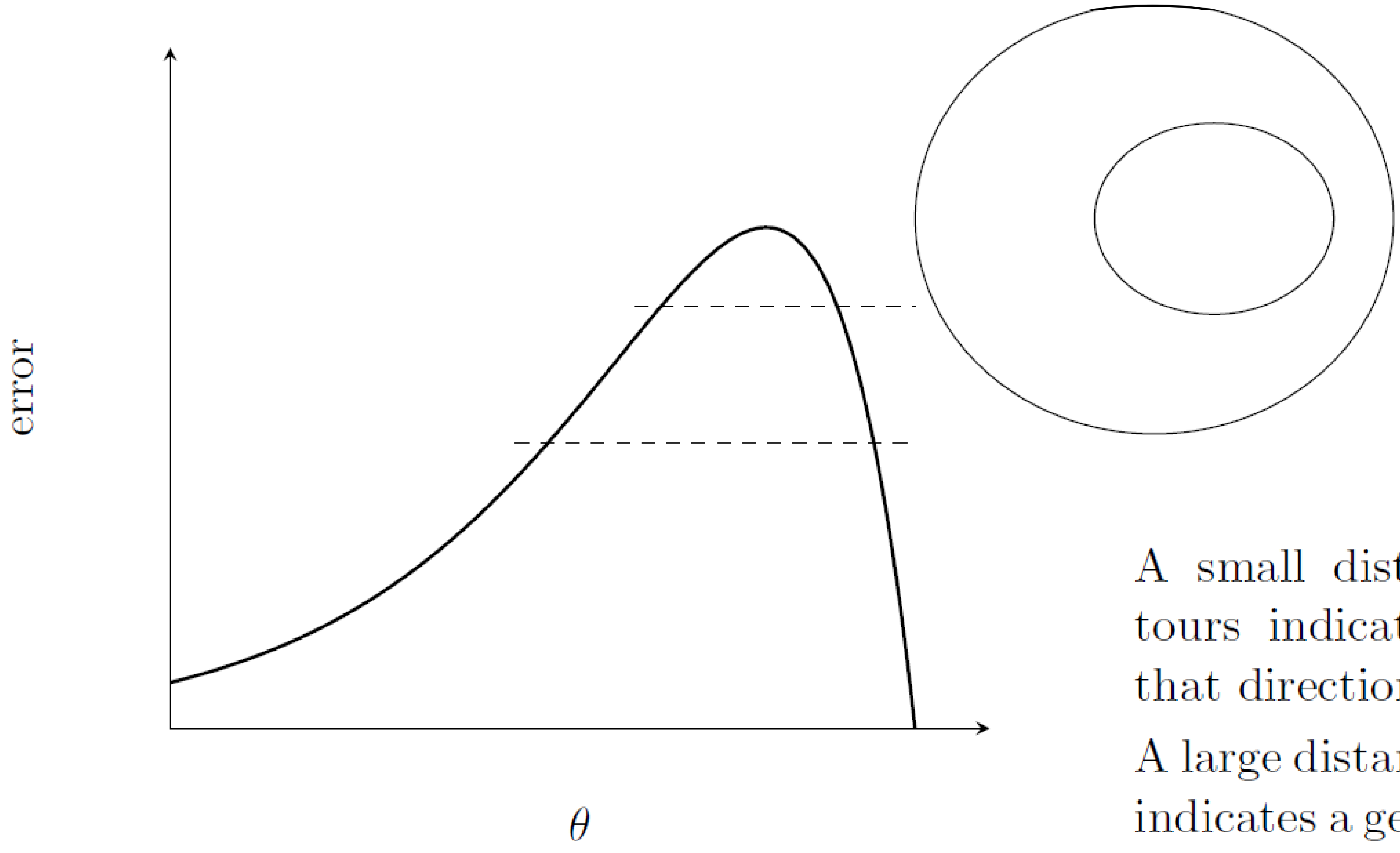
Gradient descent: Behaviour for different initializations





Slow in flat regions and fast in steep slopes

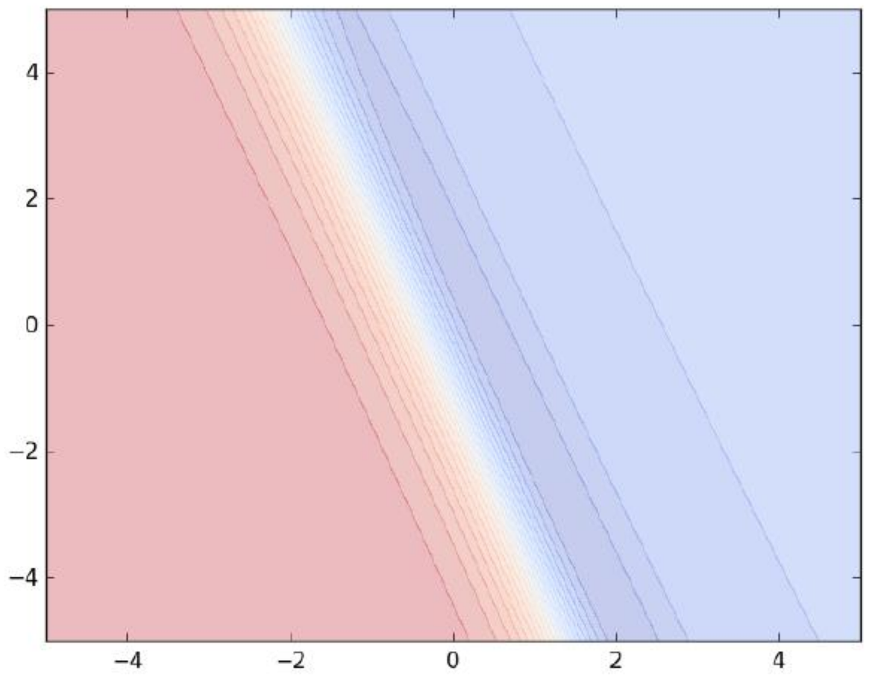
Contour Plot: a better way of visualizing error surface in 2D

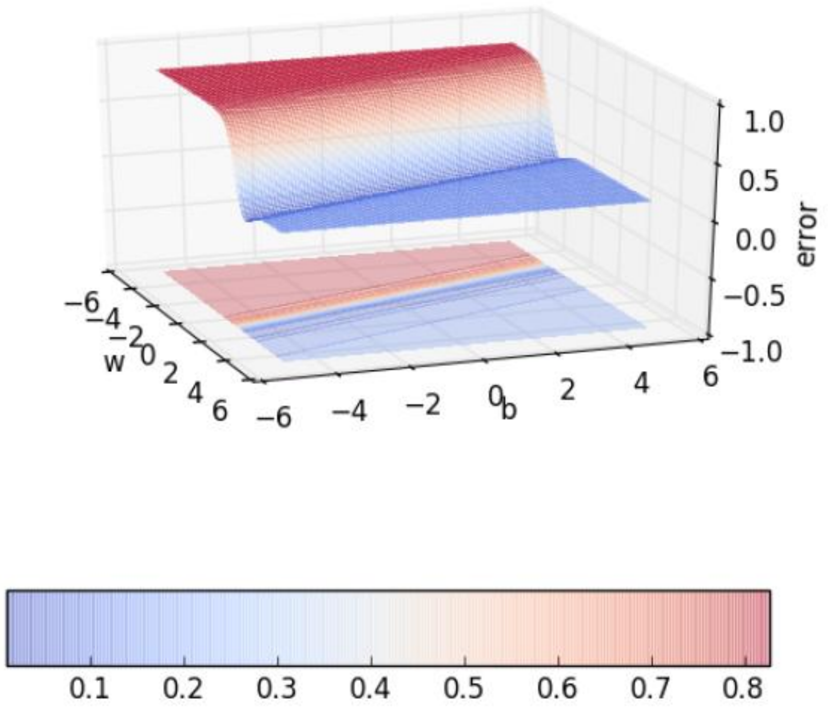


A small distance between the contours indicates a steep slope along that direction

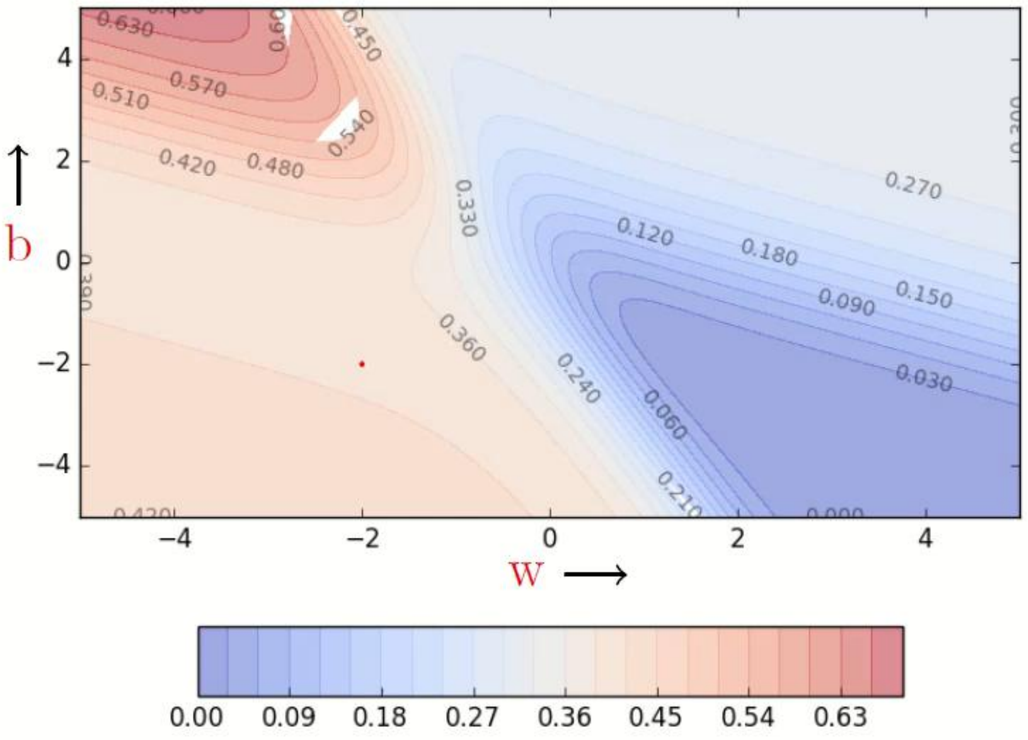
A large distance between the contours indicates a gentle slope along that direction

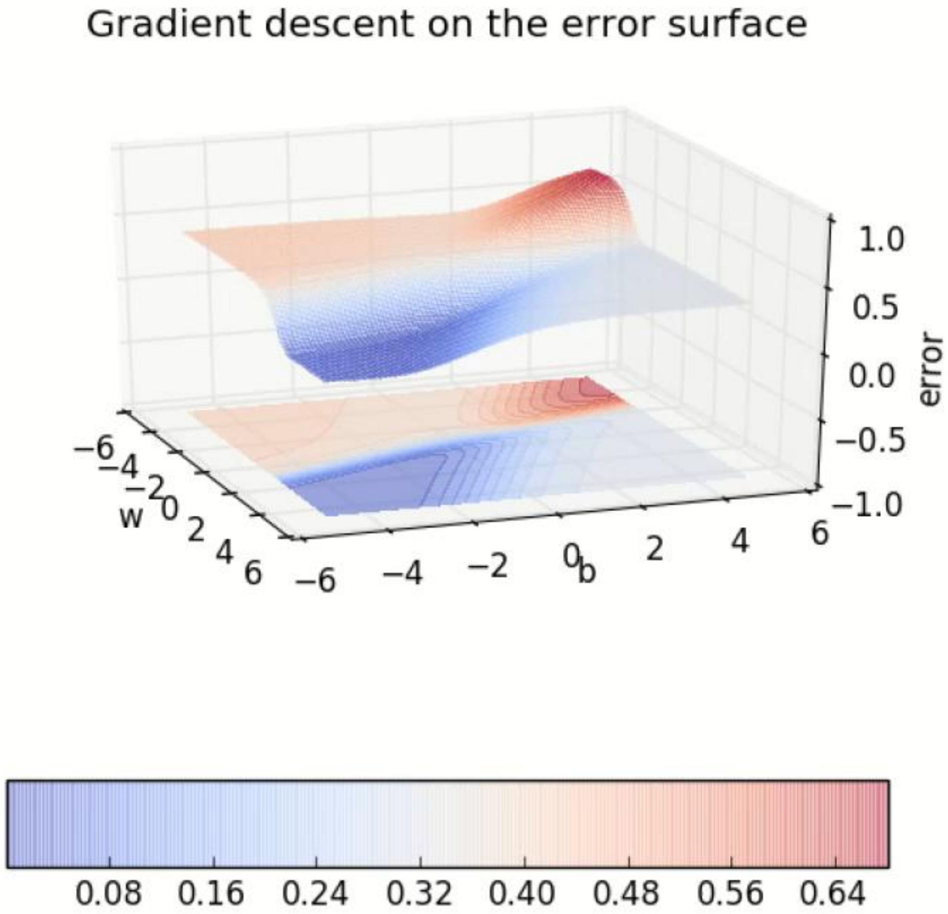
Contour Plot vs 3D surface plot





Gradient descent and its variants





Gradient descent: Problems

Plateaus and Flat Regions

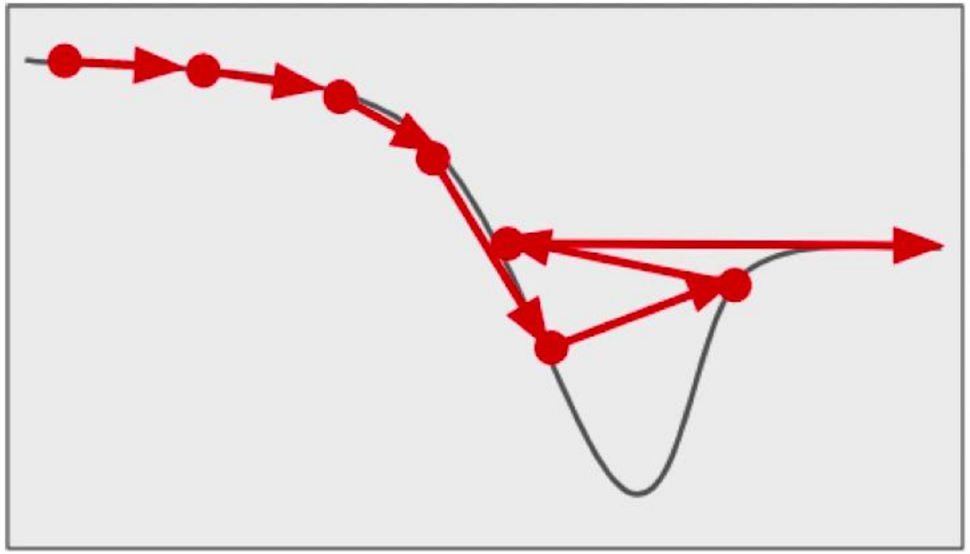
They constitute portions of error surface where gradient is highly non-spherical

Gradient descent spends a long time traversing in these regions as the updates are small

Can we expedite this process?

How about increasing the learning rate?

Though traversal becomes faster in plateaus, there is a risk of divergence



Gradient descent and its variants

Momentum-based Gradient Descent

Range : [0,1] (typically closer to 1)

Accumulated history of weight updates

$$\boxed{update_t} = \gamma \cdot update_{t-1} + \eta \nabla w_t$$

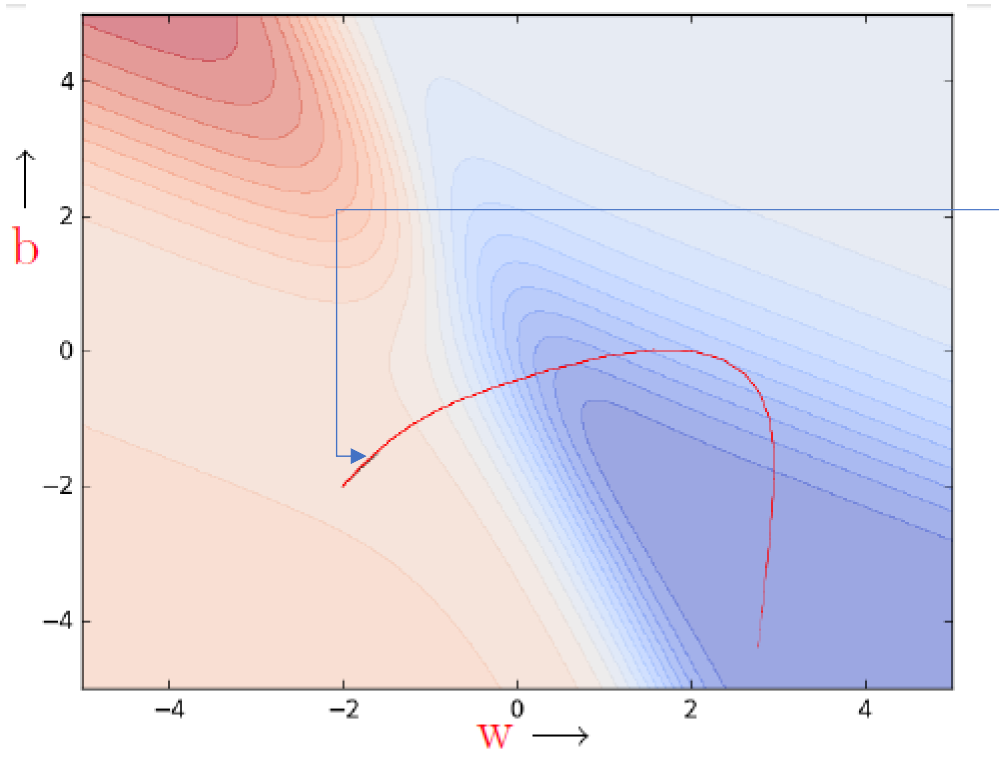
$$w_{t+1} = w_t - update_t$$

$$\begin{aligned}
update_0 &= 0 \\
update_1 &= \gamma \cdot update_0 + \eta \nabla w_1 = \eta \nabla w_1 \\
update_2 &= \gamma \cdot update_1 + \eta \nabla w_2 = \gamma \cdot \eta \nabla w_1 + \eta \nabla w_2 \\
update_3 &= \gamma \cdot update_2 + \eta \nabla w_3 = \gamma(\gamma \cdot \eta \nabla w_1 + \eta \nabla w_2) + \eta \nabla w_3 \\
&= \gamma \cdot update_2 + \eta \nabla w_3 = \gamma^2 \cdot \eta \nabla w_1 + \gamma \cdot \eta \nabla w_2 + \eta \nabla w_3 \\
update_4 &= \gamma \cdot update_3 + \eta \nabla w_4 = \gamma^3 \cdot \eta \nabla w_1 + \gamma^2 \cdot \eta \nabla w_2 + \gamma \cdot \eta \nabla w_3 + \eta \nabla w_4 \\
&\vdots \\
update_t &= \gamma \cdot update_{t-1} + \eta \nabla w_t = \gamma^{t-1} \cdot \eta \nabla w_1 + \gamma^{t-2} \cdot \eta \nabla w_1 + \ldots + \eta \nabla w_t
\end{aligned}$$

Update = An Exponential weighted average of gradients (more weightage to recent updates and less weightage to old updates)

Gradient descent and its variants

Momentum-based Gradient Descent



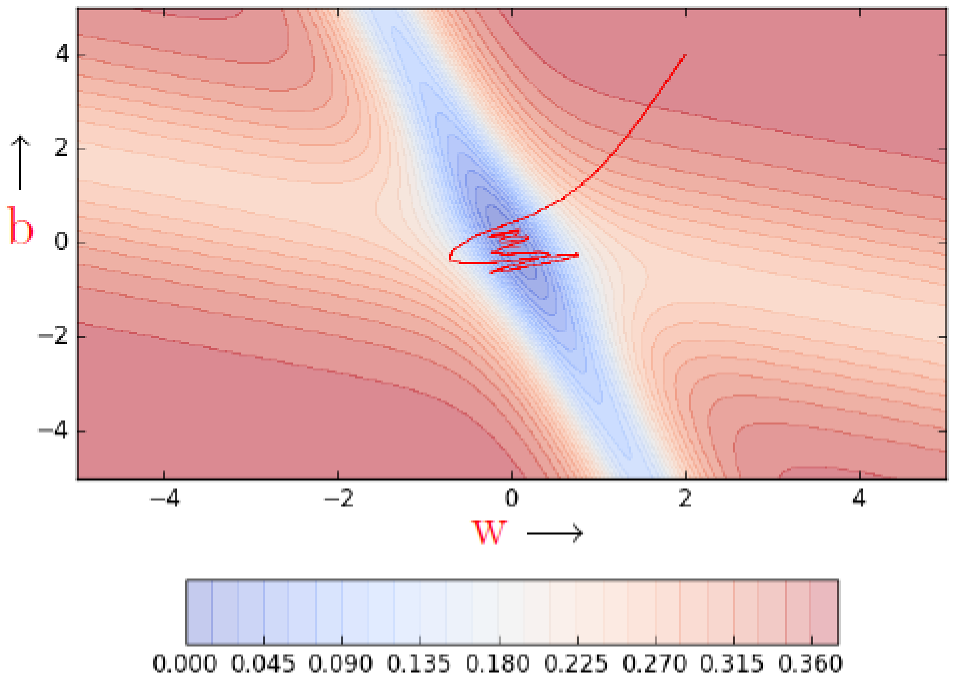
Black curve (GD) updates are slow in the flat regions.

At same no. of iterations, the red curve (momentum-based GD) converges faster

Is moving fast always good? Would there be a situation where momentum would cause us to run pass the goal?

Gradient descent and its variants

Momentum-based Gradient Descent



- Momentum based gradient descent oscillates in and out of the minima valley as the momentum carries it out of the valley
- Takes a lot of u-turns before finally converging
- Despite these u-turns it still converges faster than vanilla gradient descent
- After 100 iterations momentum based method has reached an error of 0.00001 whereas vanilla gradient descent is still stuck at an error of 0.36
- How to reduce the oscillations?

Gradient descent and its variants

Nesterov Accelerated Gradient Descent

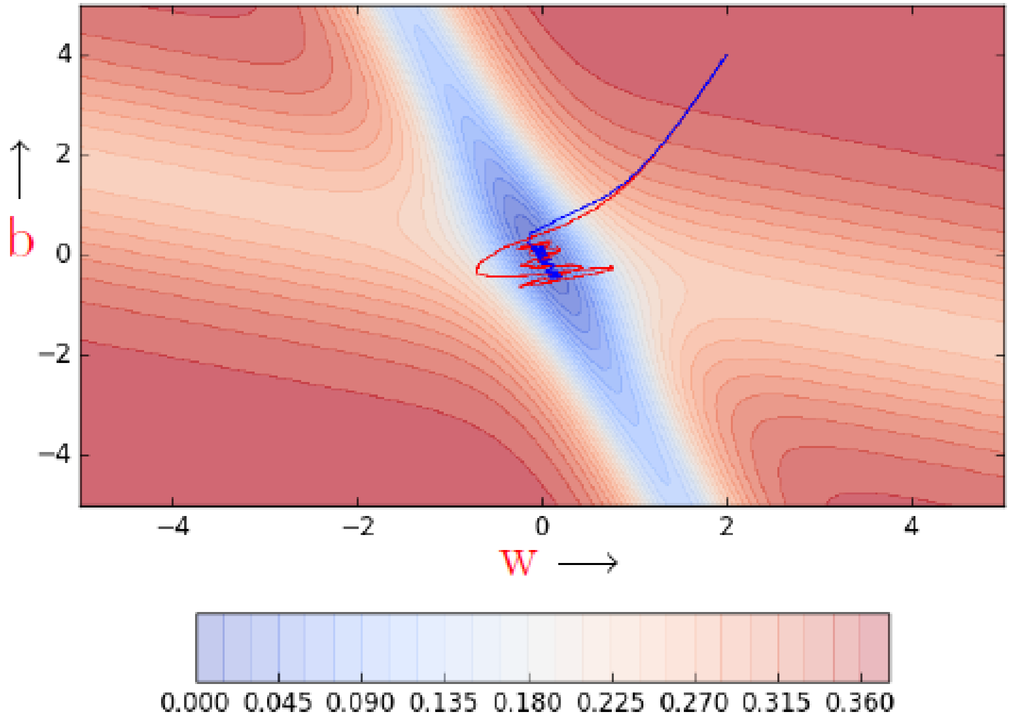
Intuition: **Look before you leap!**

- Recall that: $update_t = \gamma . update_{t-1} + \eta \nabla w_t$
- So, the movement is at least $\gamma . update_{t-1}$ and then a bit more by $\eta \nabla w_t$

$$\boxed{\begin{aligned} w_{look\_ahead} &= w_t - \gamma \cdot update_{t-1} \\ update_t &= \gamma \cdot update_{t-1} + \eta \nabla w_{look\_ahead} \\ w_{t+1} &= w_t - update_t \end{aligned}}$$

Gradient descent and its variants

Nesterov Accelerated Gradient Descent

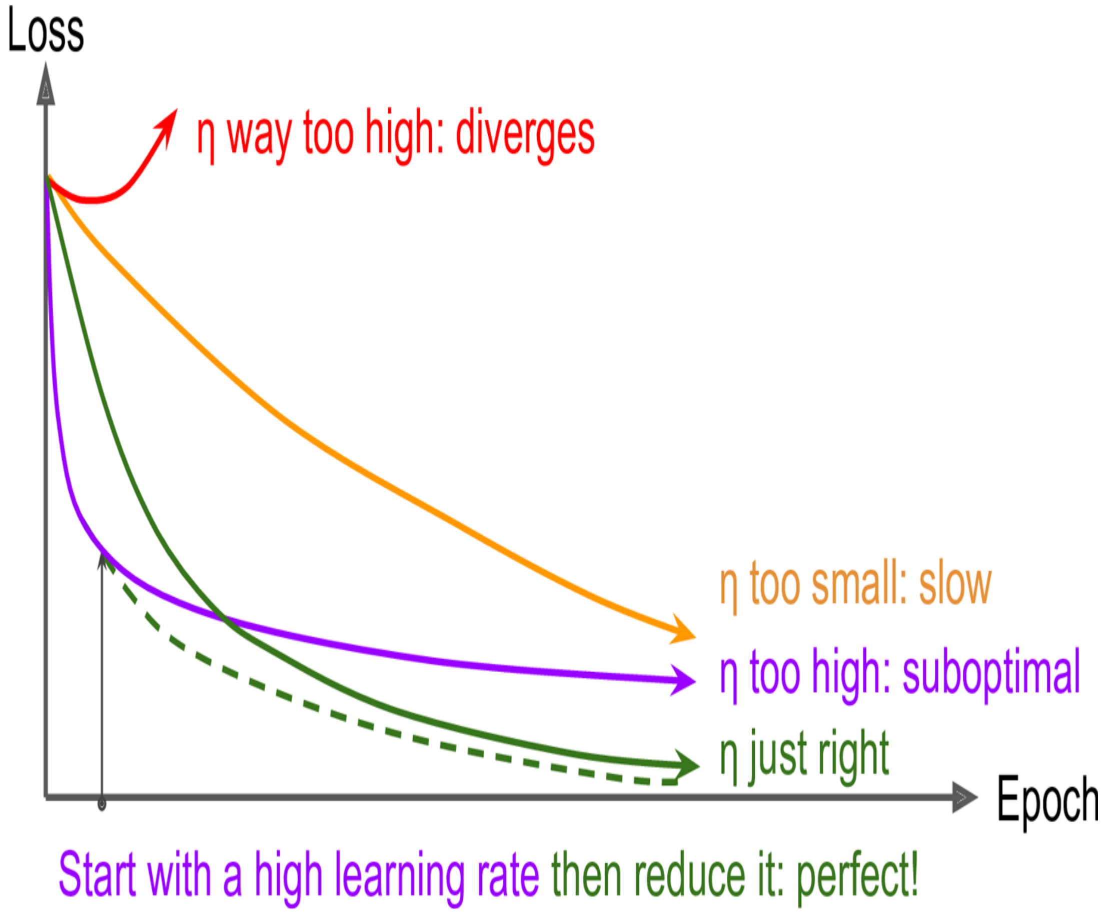


Red curve – momentum-based GD
Blue curve – Nesterov accelerated GD

Constant learning rate not ideal

- Better to start with a high learning rate
- then reduce it once it stops making fast progress
- can reach a good solution faster

- Learning Schedule strategies can be applied
  - Power Scheduling
  - Exponential Scheduling
  - Piecewise Constant Scheduling
  - Performance Scheduling



How about optimizing GD by varying learning rate ?

- If learning rate is too small, it takes long time to converge.
  If learning rate is too large, the gradients explode.

Some techniques for choosing learning rate:

- **Linear Search**

Tune learning rate [Try different values on a log scale: 0.0001, 0.001, 0.01, 0.1. 1.0]
Run a few epochs with each of these and figure out a learning rate which works best
Now do a finer search around this value [for example, if the best learning rate was 0.1 then now try some values around it: 0.05, 0.2, 0.3]

- *Piecewise constant scheduling*
  - Constant learning rate for a number of epochs
    - e.g., η_0 = 0.1 for 5 epochs
  - then a smaller learning rate for another number of epochs
    - e.g., η_1 = 0.001 for 50 epochs and so on
- *Performance scheduling*
  - Measure the validation error every *N* steps (just like for early stopping)
  - reduce the learning rate by a factor of λ when the error stops dropping

Choosing a learning rate

- **Annealing-based methods:**
  - **Step Decay:**
    - Halve the learning rate after every 5 epochs or
    - Halve the learning rate after an epoch if the validation error is more than what it was at the end of the previous epoch
  - **Exponential Decay:** $\eta = \eta_0^{-kt}$ where, η_0 and k are hyperparameters and t is the step number
  - **1/t Decay:** $\eta = \dfrac{\eta_0}{1 + kt}$ where, η_0 and k are hyperparameters and t is the step number

GD with adaptive learning rate

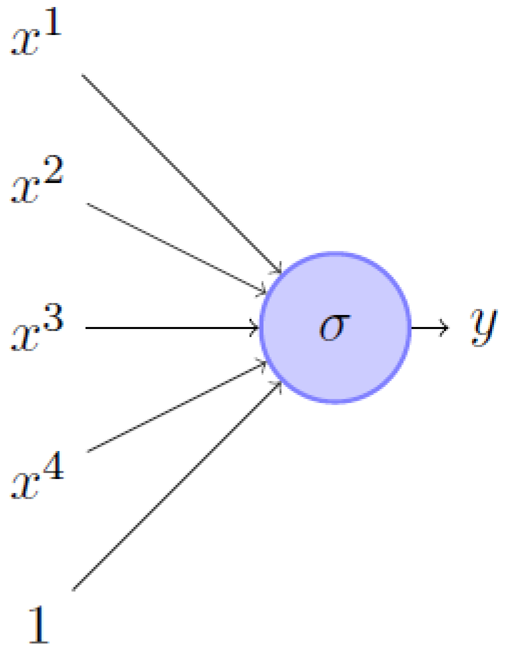
- Motivation: Can we have a different learning rate for each parameter which takes care of the frequency of features ?

- Intuition: Decay the learning rate for parameters in proportion to their update history.
  - For sparse features, accumulated update history is small
  - For dense features, accumulated update history is large

Make learning rate inversely proportional to the update history i.ie, if the feature has been updated fewer times, give it a larger learning rate and vice versa

Let's consider an example .....

GD with adaptive learning rate



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

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

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

- Given this network, it should be easy to see that given a single point $(\mathbf{x}, y)$...
- $\nabla w^1 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^1$
- $\nabla w^2 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^2$ ... so on
- If there are n points, we can just sum the gradients over all the n points to get the total gradient
- What happens if the feature x^2 is very sparse? (*i.e.*, if its value is 0 for most inputs)
- ∇w^2 will be 0 for most inputs (see formula) and hence w^2 will not get enough updates
- If x^2 happens to be sparse as well as important we would want to take the updates to w^2 more seriously
- Can we have a different learning rate for each parameter which takes care of the frequency of features ?

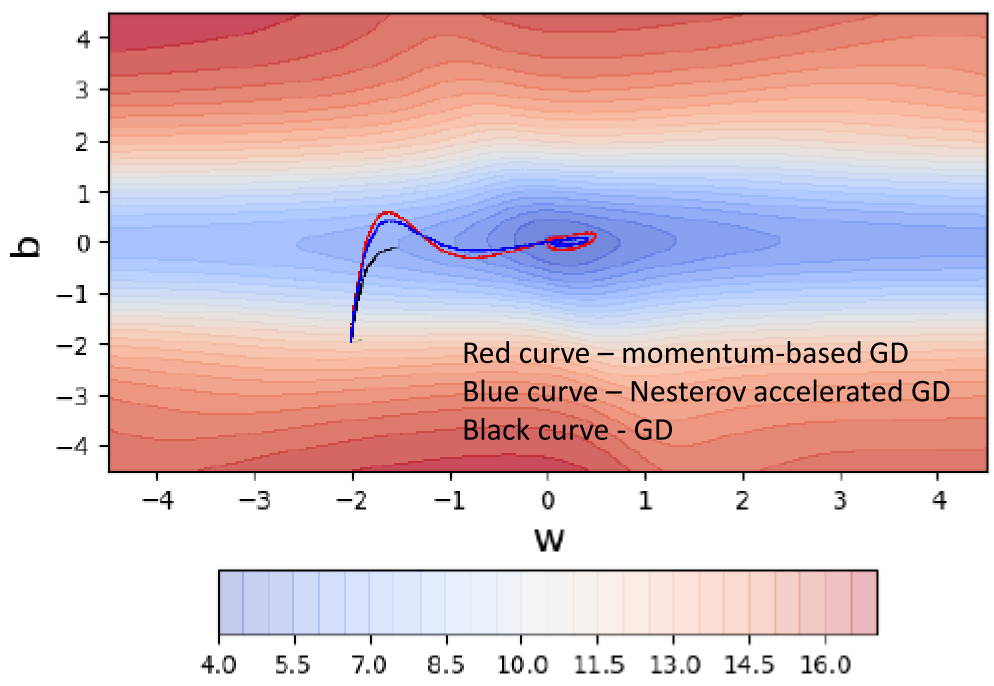
Adagrad

- Update rule for Adagrad

$$v_t = v_{t-1} + (\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t$$

If the feature has been updated fewer times, give it a larger learning rate and vice versa

GD (black), momentum (red) and NAG (blue)



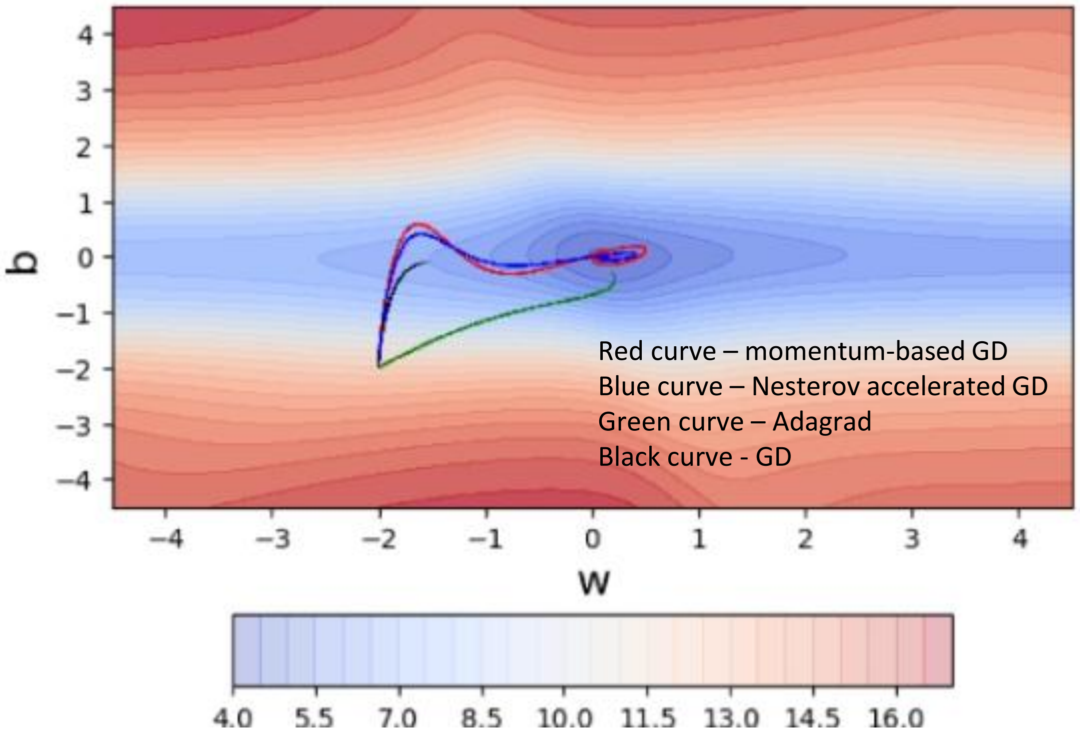
Initially, all three algorithms are moving mainly along the vertical (b) axis and there is very little movement along the horizontal (w) axis

Why? Because in our data, the feature corresponding to w is sparse and hence w undergoes very few updates

Such sparsity is very common in large neural networks containing $1000s$ of input features and hence we need to address it

Adagrad

- By using a parameter specific learning rate it ensures that despite sparsity w gets a higher learning rate and hence larger updates
- Further, it also ensures that if b undergoes a lot of updates its effective learning rate decreases because of the growing denominator
- In practice, this does not work so well if we remove the square root from the denominator (something to ponder about)
- What's the flipside? over time the effective learning rate for b will decay to an extent that there will be no further updates to b
- Can we avoid this?



RMS Prop

- Intuition: Adagrad decays the learning rate very aggressively (as the denominator grows).

- To avoid this why not decay the denominator and prevent its rapid growth.

    - Update rule for RMS Prop:

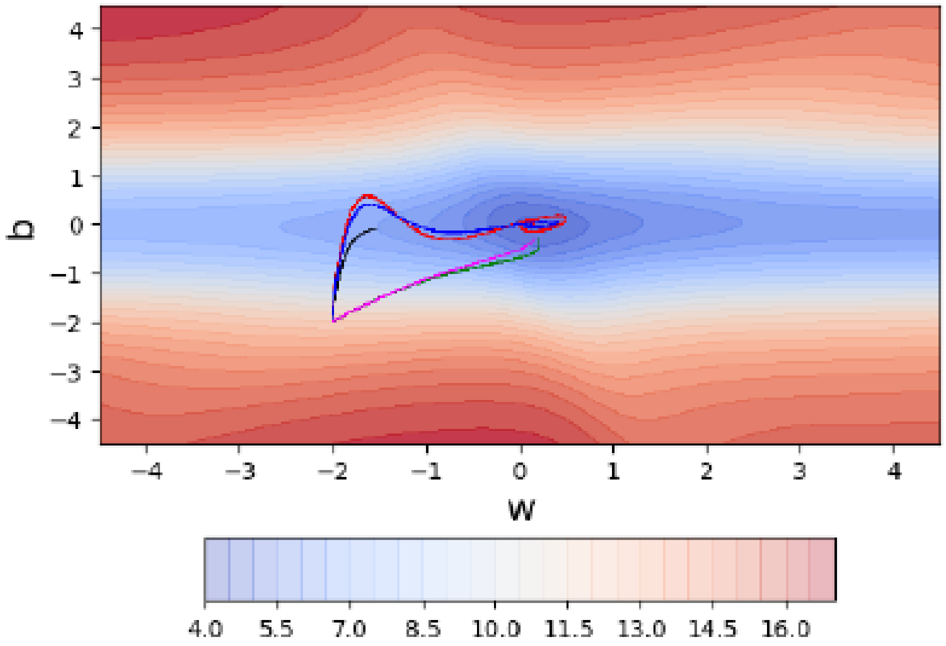
Exponential weighted moving average (weighted decay)

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

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

RMS Prop

- Intuition: Adagrad decays the learning rate very aggressively (as the denominator grows).
- Update rule for RMS Prop:



Red curve – momentum-based GD
Blue curve – Nesterov accelerated GD
Green curve – Adagrad
Black curve – GD
Pink curve – RMS Prop

Adam

- Intuition: Do everything RMSProp does to solve the decay problem of Adagrad + Use a cumulative history of gradients.

- Update rule for Adam:

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

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

$$\boxed{\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}} \rightarrow \text{Bias correction}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t$$

In practice, $\beta_1 = 0.9$ and $\beta_2 = 0.999$

Regularization

Different forms of regularization

- L2 regularization
- Dataset augmentation
- Parameter Sharing and tying
- Adding Noise to the inputs
- Adding Noise to the outputs
- Early stopping
- Dropout

L2 regularization

- For l_2 regularization we have,

$$\widetilde{\mathscr{L}}(w) = \mathscr{L}(w) + \frac{\alpha}{2}\|w\|^2$$

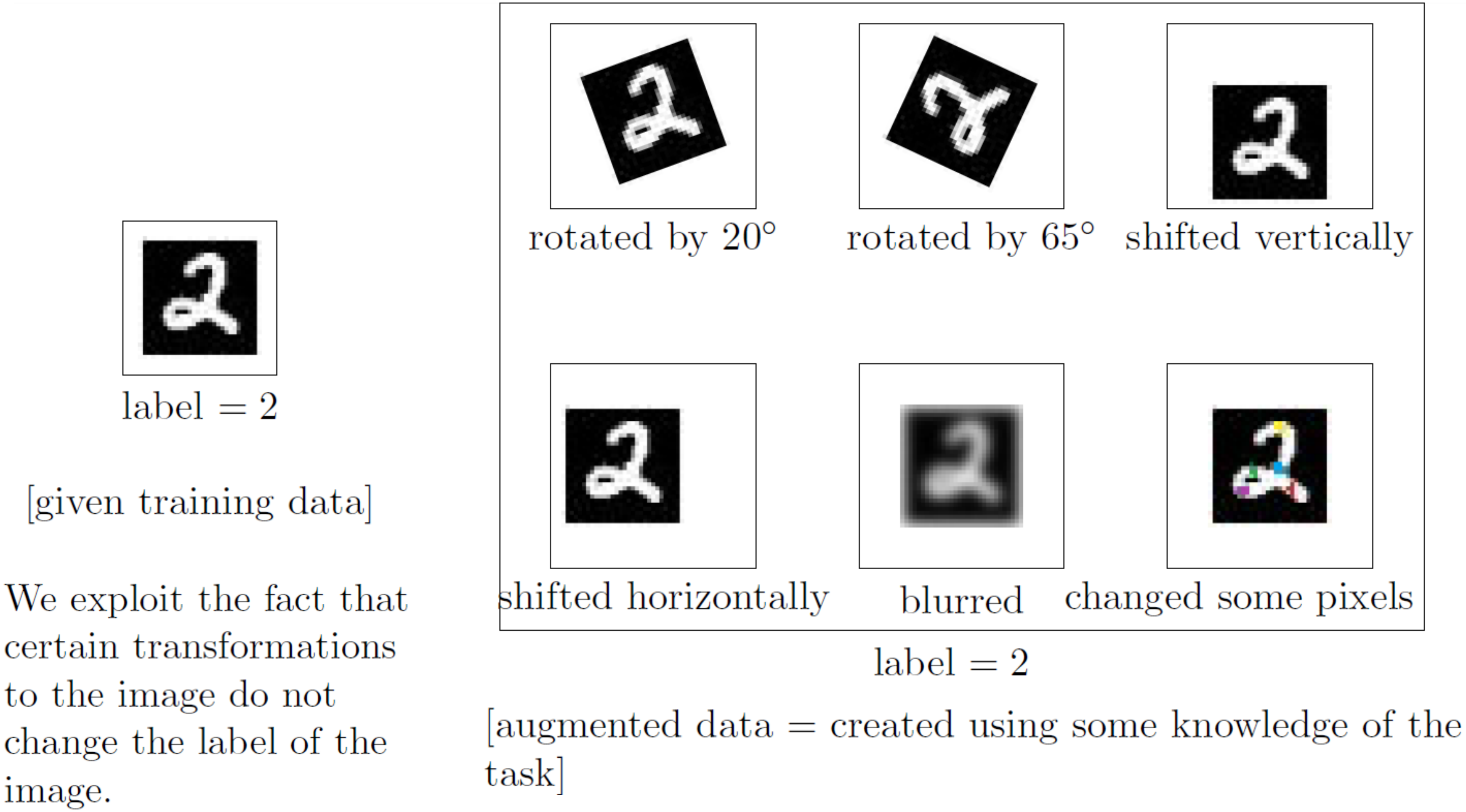
- For SGD (or its variants), we are interested in

$$\nabla\widetilde{\mathscr{L}}(w) = \nabla\mathscr{L}(w) + \alpha w$$

- Update rule:

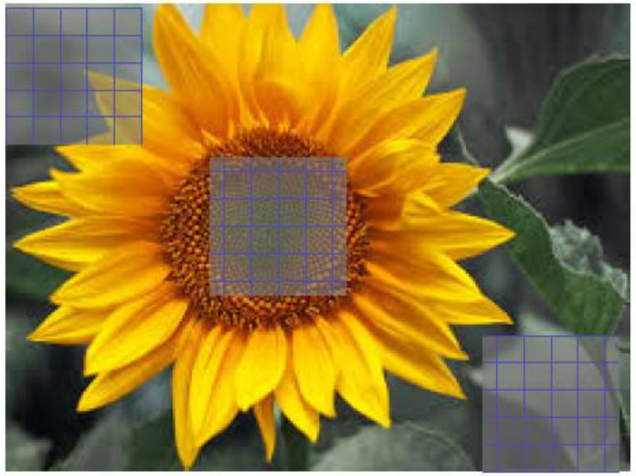
$$w_{t+1} = w_t - \eta\nabla\mathscr{L}(w_t) - \eta\alpha w_t$$

Dataset Augmentation



Works well for image classification / object recognition tasks. Also shown to work well for speech

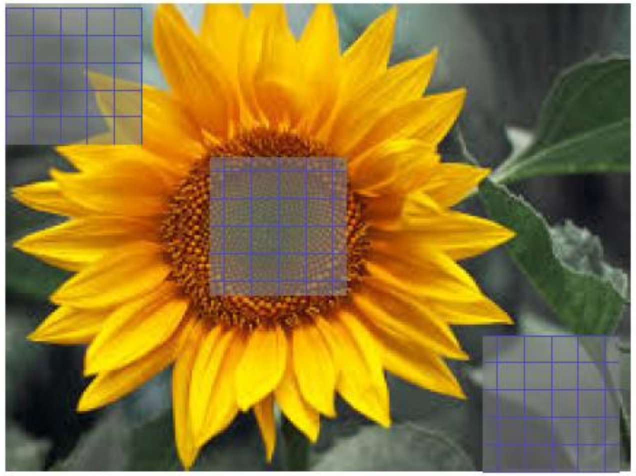
Parameter sharing and tying

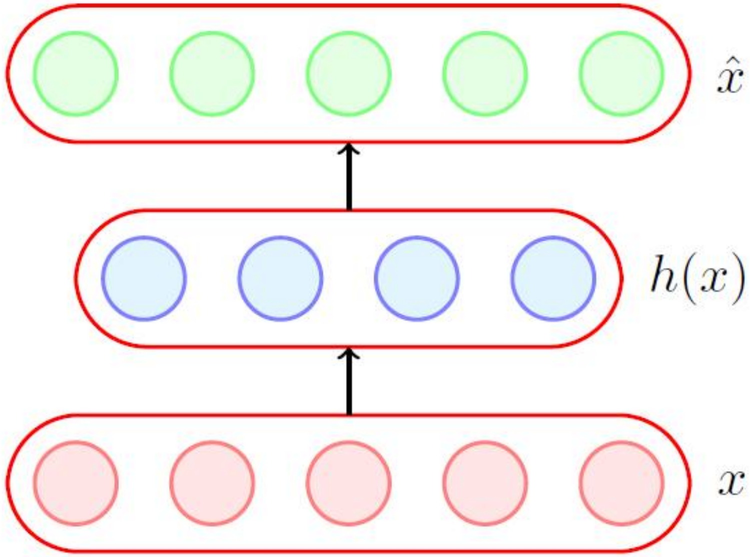


Parameter Sharing

- Used in CNNs
- Same filter applied at different positions of the image
- Or same weight matrix acts on different input neurons

Parameter sharing and tying





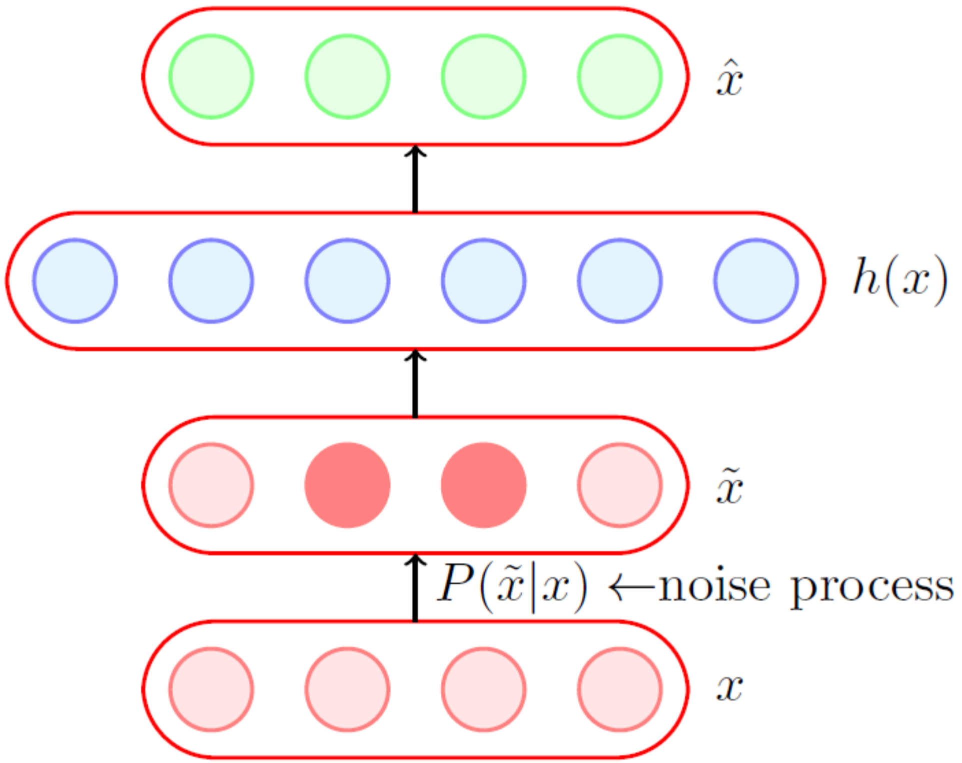
Parameter Sharing

- Used in CNNs
- Same filter applied at different positions of the image
- Or same weight matrix acts on different input neurons

Parameter Tying

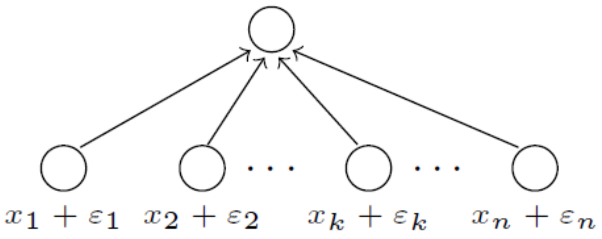
- Typically used in autoencoders
- The encoder and decoder weights are tied.

Adding noise to inputs



Same as in autoencoders

Adding noise to inputs



$$\varepsilon \sim \mathcal{N}(0, \sigma^2)$$

$$\begin{aligned}
\widetilde{x}_i &= x_i + \varepsilon_i \\
\widehat{y} &= \sum_{i=1}^{n} w_i x_i \\
\widetilde{y} &= \sum_{i=1}^{n} w_i \widetilde{x}_i \\
&= \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} w_i \varepsilon_i \\
&= \widehat{y} + \sum_{i=1}^{n} w_i \varepsilon_i
\end{aligned}$$

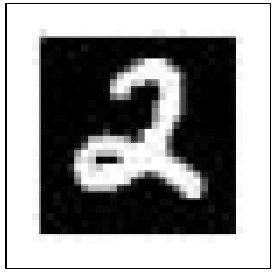
We are interested in $E[(\widetilde{y} - y)^2]$

$$\begin{aligned}
E\left[(\widetilde{y} - y)^2\right] &= E\left[\left(\widehat{y} + \sum_{i=1}^{n} w_i \varepsilon_i - y\right)^2\right] \\
&= E\left[\left(\left(\widehat{y} - y\right) + \left(\sum_{i=1}^{n} w_i \varepsilon_i\right)\right)^2\right] \\
&= E\left[(\widehat{y} - y)^2\right] + E\left[2(\widehat{y} - y)\sum_{i=1}^{n} w_i \varepsilon_i\right] + E\left[\left(\sum_{i=1}^{n} w_i \varepsilon_i\right)^2\right] \\
&= E\left[(\widehat{y} - y)^2\right] + 0 + E\left[\sum_{i=1}^{n} w_i^2 \varepsilon_i^2\right]
\end{aligned}$$

($\because$ ε_i is independent of ε_j and ε_i is independent of $(\widehat{y}$-$y)$)

$$= (E\left[(\widehat{y} - y)^2\right] + \boxed{\sigma^2 \sum_{i=1}^{n} w_i^2} \quad \text{(same as } L_2 \text{ norm penalty)}$$

Adding noise to outputs



0	0	1	0	0	0	0	0	0	0	Hard targets

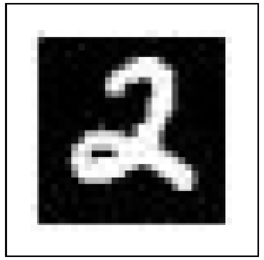
$$\text{minimize}: \sum_{i=0}^{9} p_i \log q_i$$

$$\text{true distribution}: p = \{0, 0, 1, 0, 0, 0, 0, 0, 0, 0\}$$

$$\text{estimated distribution}: q$$

Intuition: Don't trust the true labels, they may be noisy. Instead, use soft targets.

Adding noise to outputs



$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	$1-\varepsilon$	$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	$\frac{\varepsilon}{9}$	Soft targets

$$\text{minimize} : \sum_{i=0}^{9} p_i \log q_i$$

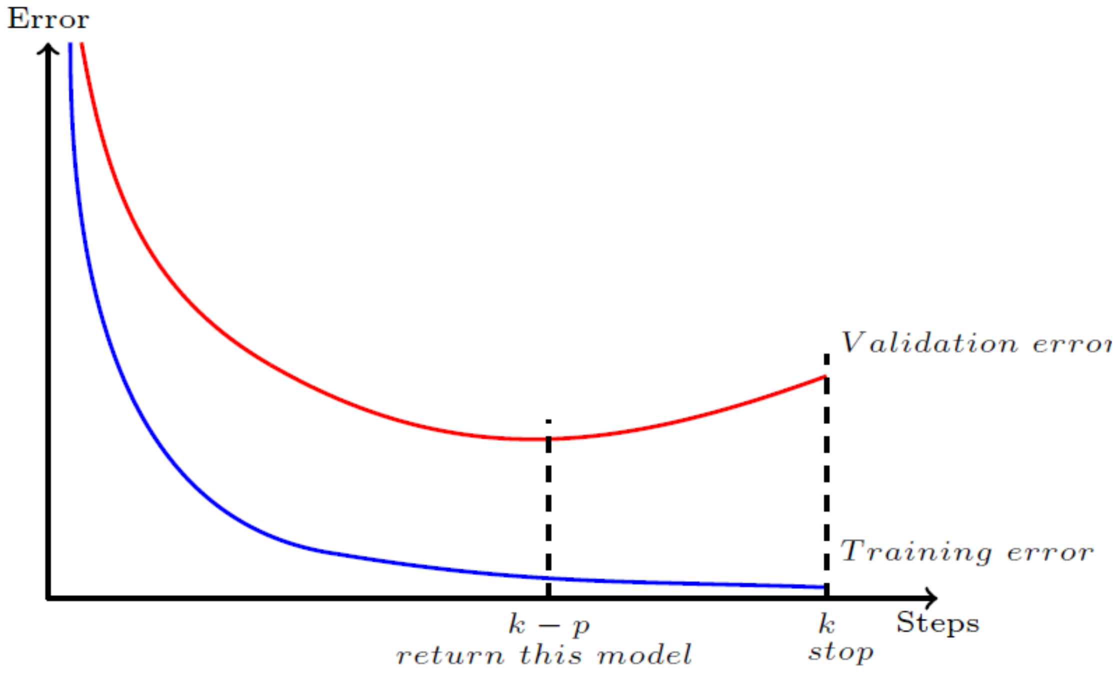
$$\text{true distribution} + \text{noise} : p = \left\{\frac{\varepsilon}{9}, \frac{\varepsilon}{9}, 1-\varepsilon, \frac{\varepsilon}{9}, \ldots\right\}$$

$$\varepsilon = \text{small positive constant}$$

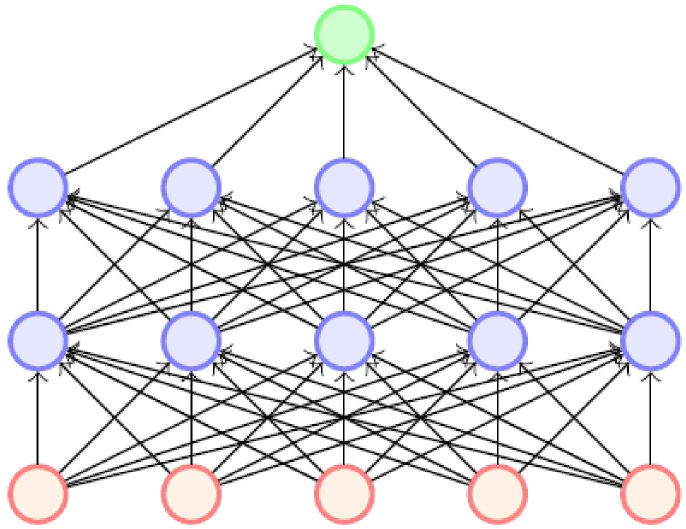
Intuition: Don't trust the true labels, they may be noisy. Instead, use soft targets.

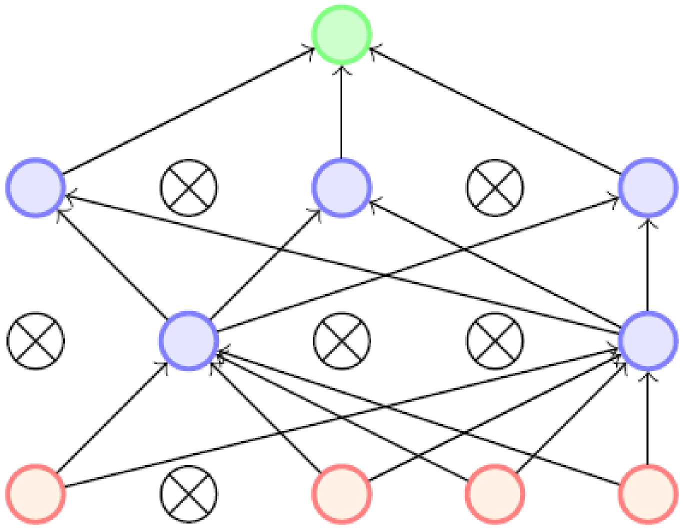
Early stopping

- Track the validation error
- Have a patience parameter p
- If you are at step k and there was no improvement in validation error in the previous p steps then stop training and return the model stored at step k - p



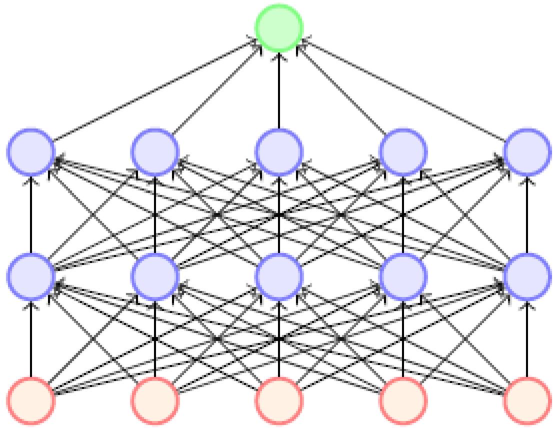
Dropout

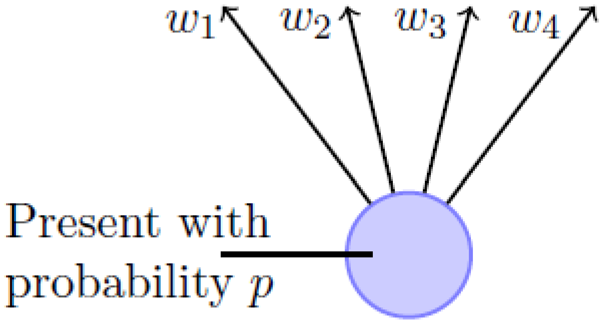




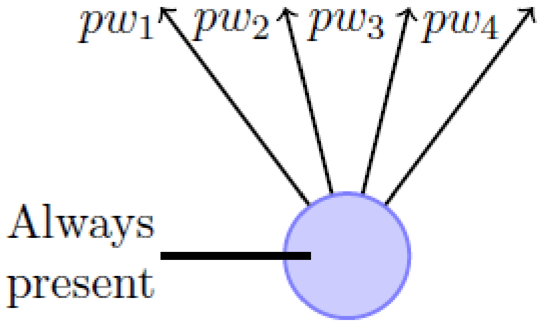
- Dropout refers to dropping out units
- Temporarily remove a node and all its incoming/outgoing connections resulting in a thinned network
- Each node is retained/dropped randomly with some probability (eg: 0.5) at each step during training.

Dropout





At training time

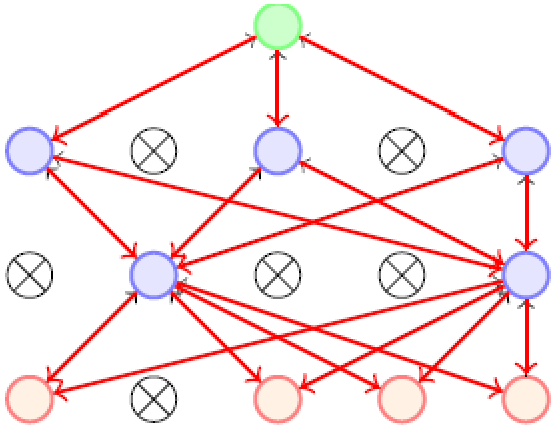


At test time

During Testing (Inference):

- Dropout is turned off, and all neurons are active.
- To keep the expected output at each layer the same as during training, the weights of each neuron are scaled by the dropout rate.
- For example, if the dropout rate is 0.5, the weights are scaled by multiplying by 0.5 during inference.
- This effectively reduces the contribution of each neuron, so the output magnitude remains consistent with the training phase.

Dropout



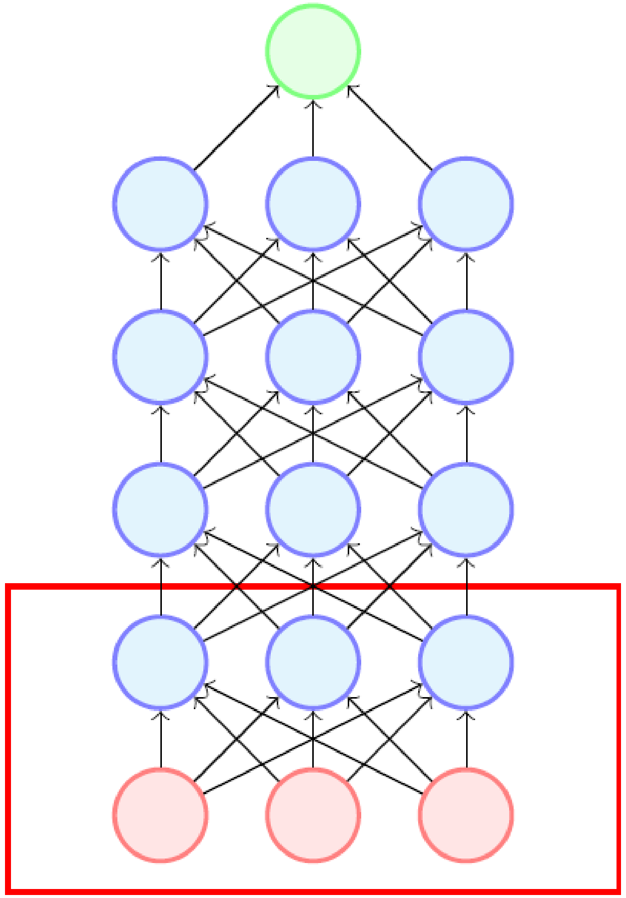
- Dropout essentially applies a masking noise to the hidden units
- Prevents hidden units from co-adapting.
- Essentially a hidden unit cannot rely too much on other units as they may get dropped out any time
- Each hidden unit has to learn to be more robust to these random dropouts

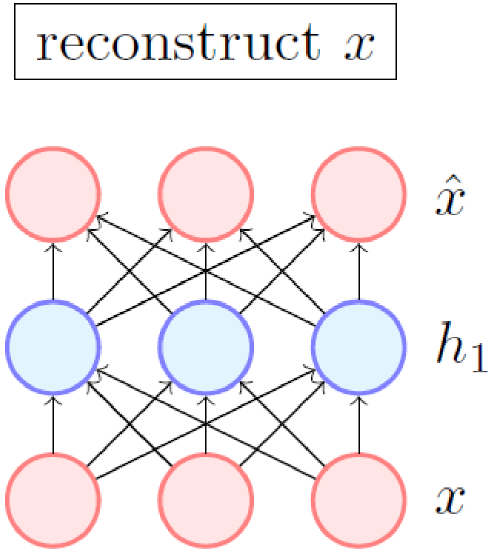
The Vanishing/Exploding Gradient Problems

- Algorithm computes the gradient of the cost function with regard to each parameter in the network
- **Problems include**
- **Vanishing Gradients**
  - Gradients become very small as algorithm progresses down to lower layers
  - So connection weights remain unchanged and training never converges to a solution
- **Exploding Gradients**
  - Gradients become so large until layers get huge weight updates and algorithm diverges
- Deep Neural Networks suffer from unstable gradients, different layers may learn at different speeds.
- **Reasons for unstable gradients** Glorot & Bengio (2010)
  - Because of combination of Sigmoid activation and weight initialization (normal distribution with mean 0 and std dev 1)
  - Variance of outputs of each layer is much greater than the variance of its inputs
  - Variance goes on increasing after each layer
  - until the activation function saturates(0 or 1, with derivative close to 0) at the top layers

Solutions include : 1. Weight Initialization

Weight Initialization through unsupervised pre-training

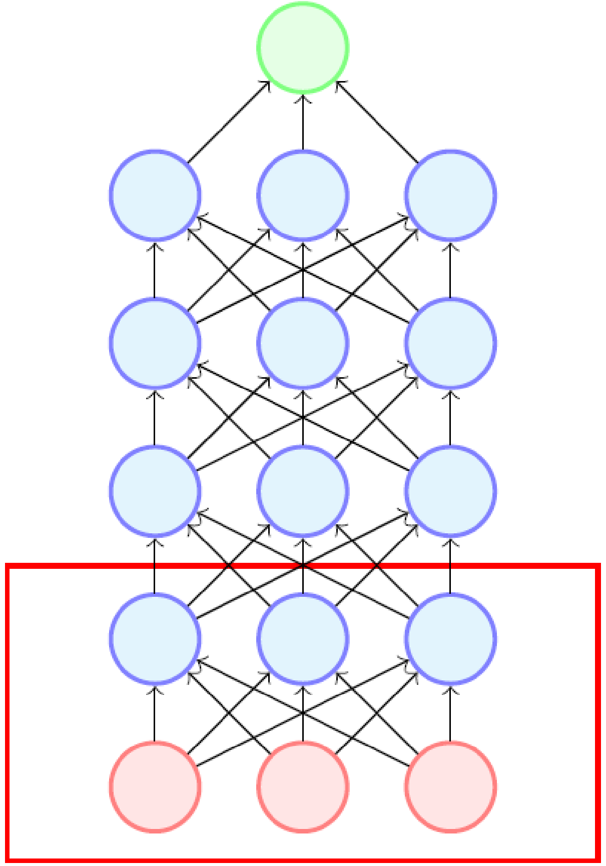


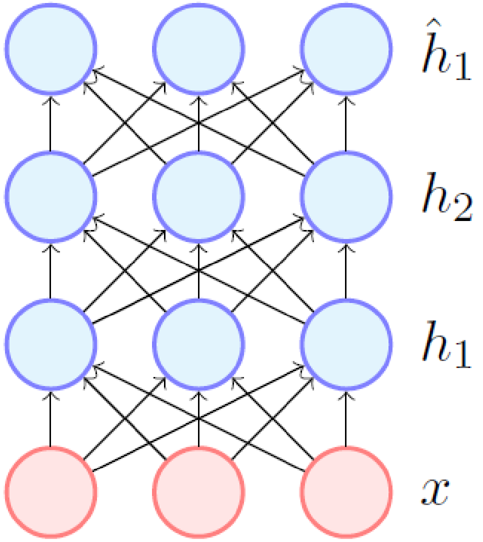


$$\min \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} (\hat{x}_{ij} - x_{ij})^2$$

Solutions include : 1. Weight Initialization

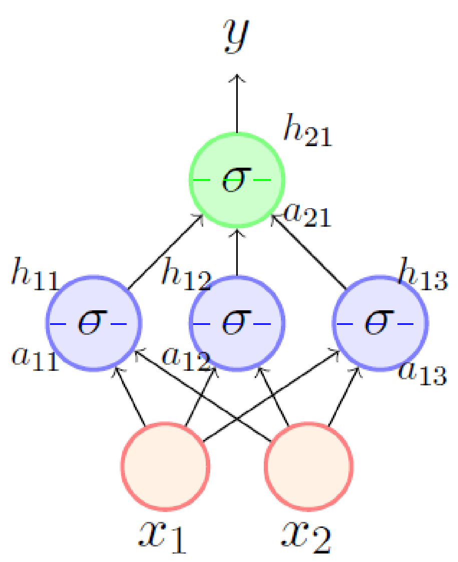
Weight Initialization through unsupervised pre-training





$$\min \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} (\hat{h}_{1_{ij}} - h_{1_{ij}})^2$$

Weight Initialization



$$a_{11} = w_{11}x_1 + w_{12}x_2$$
$$a_{12} = w_{21}x_1 + w_{22}x_2$$
$$\therefore a_{11} = a_{12} = 0$$
$$\therefore h_{11} = h_{12}$$

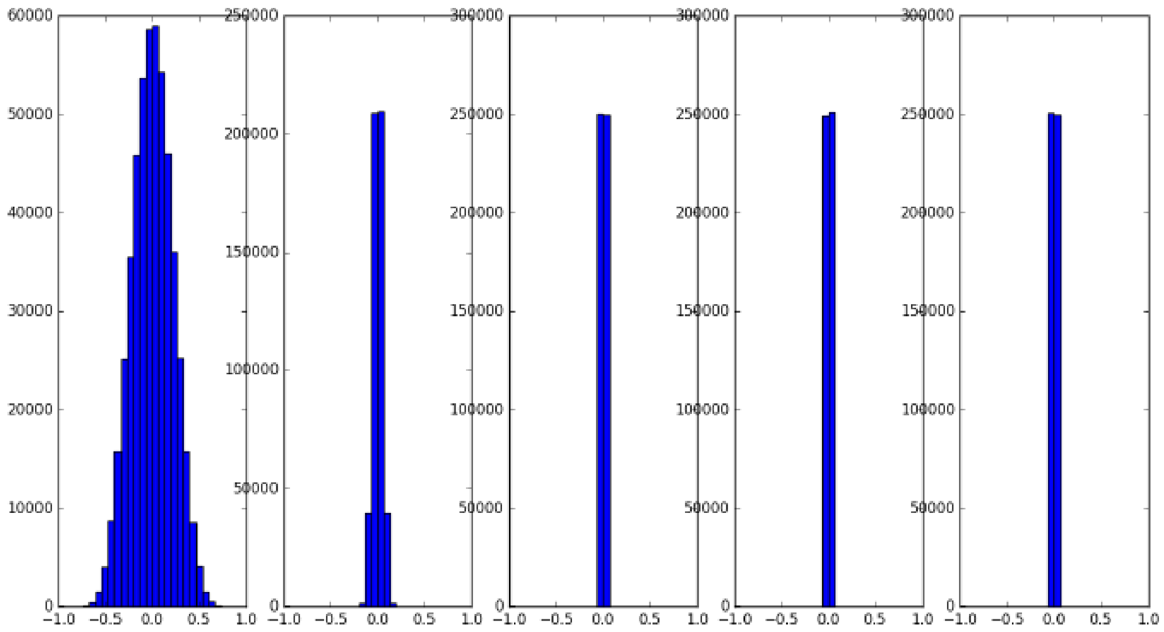
- What happens if we initialize all weights to 0?
  - All neurons in layer 1 will get the same activation.
  - So during backpropagation:

$$\nabla w_{11} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \cdot \frac{\partial y}{\partial h_{11}} \cdot \frac{\partial h_{11}}{\partial a_{11}} . x_1$$
$$\nabla w_{21} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \cdot \frac{\partial y}{\partial h_{12}} \cdot \frac{\partial h_{12}}{\partial a_{12}} . x_1$$
$$but \quad h_{11} = h_{12}$$
$$and \quad a_{12} = a_{12}$$
$$\therefore \nabla w_{11} = \nabla w_{21}$$

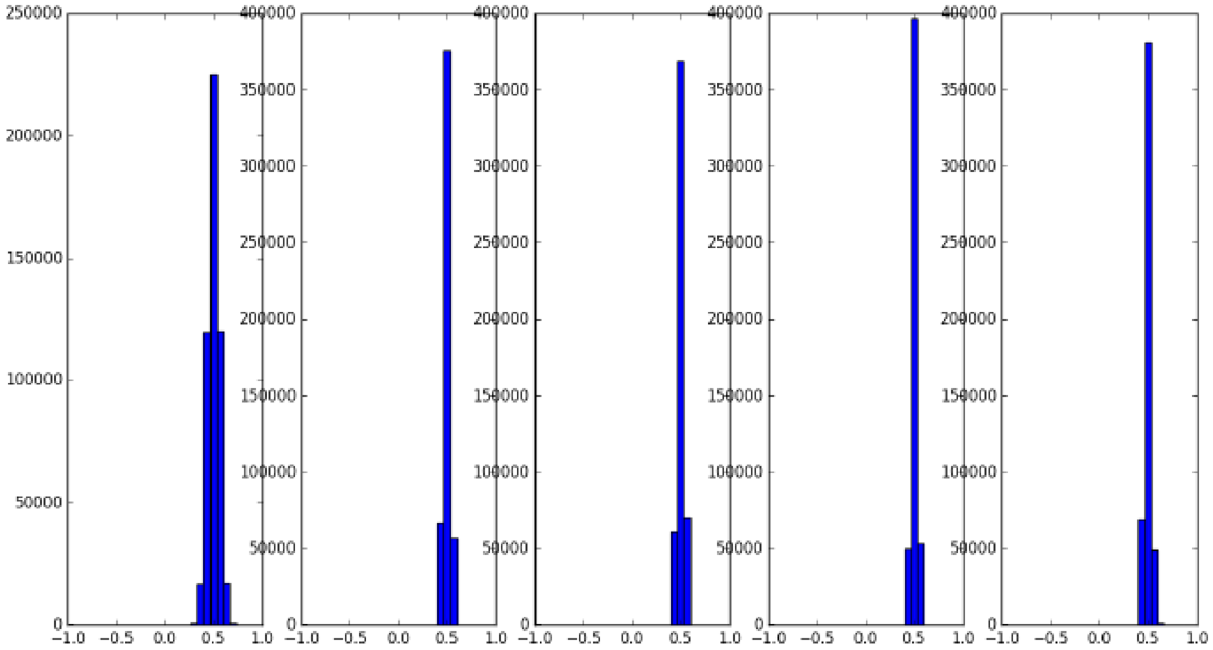
  - Hence both the weights will get the same update and remain equal

Weight Initialization

- What happens if we initialize the weights to small numbers



tanh activation functions

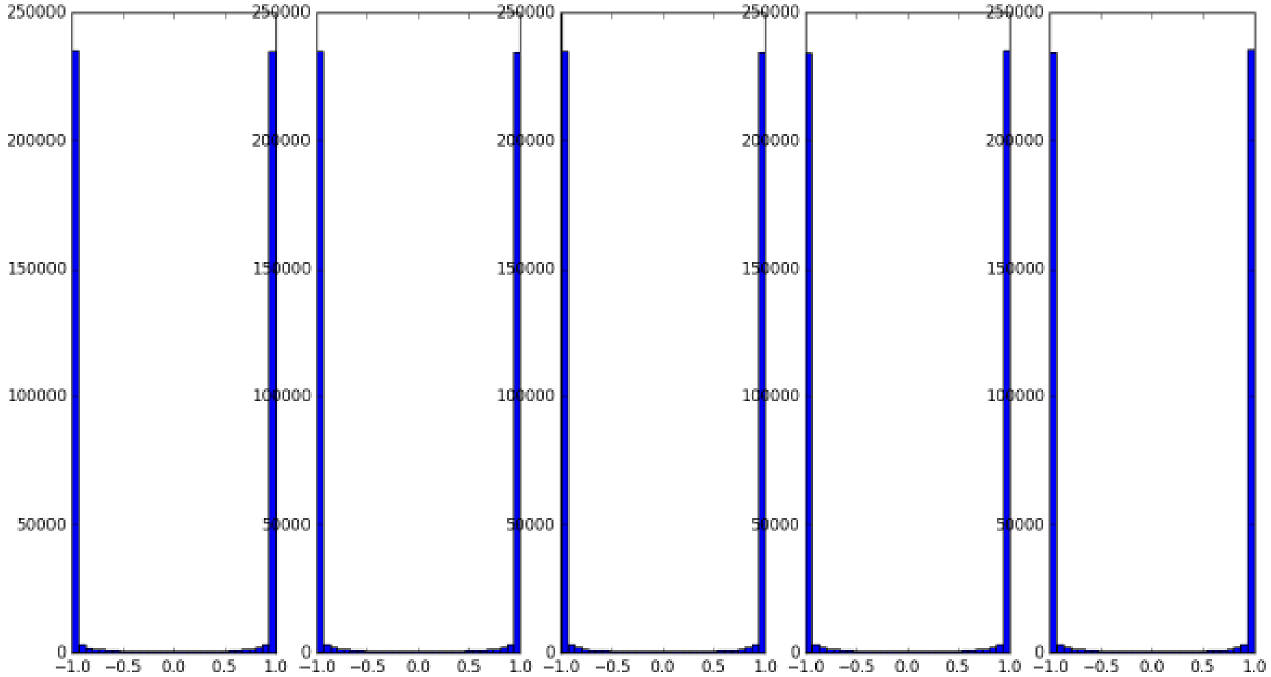


sigmoid activation functions

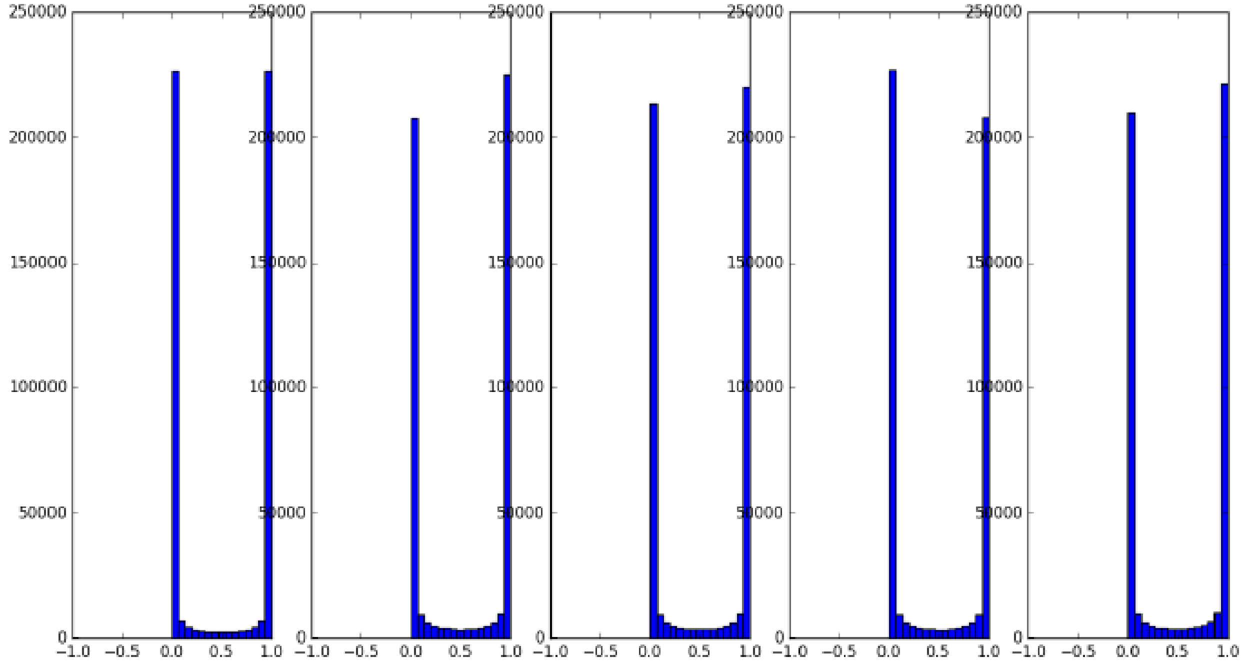
As the training progresses, all the weights are close to zero- vanishing gradient problem.

Weight Initialization

- What happens if we initialize the weights to large numbers



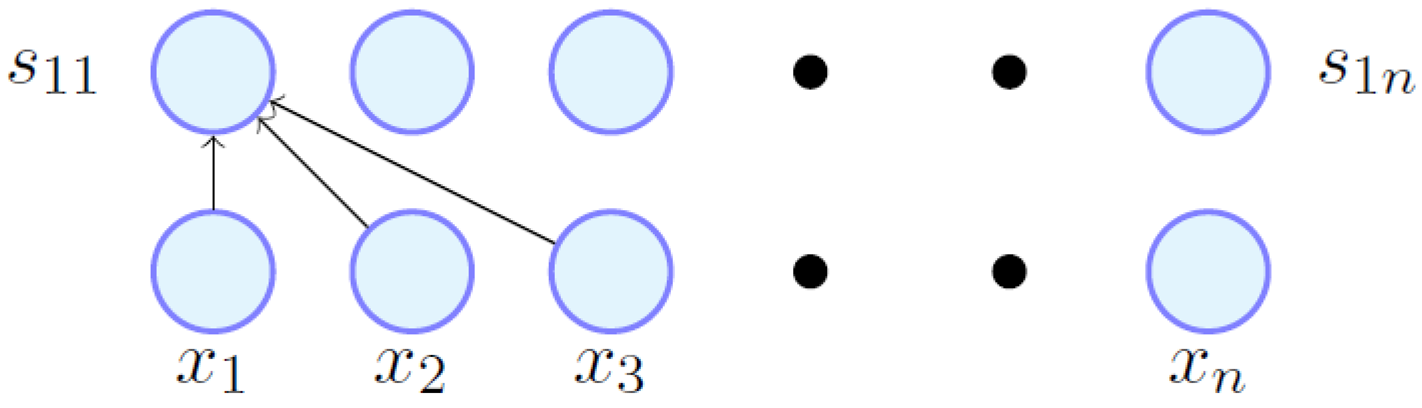
tanh activation functions



sigmoid activation functions

As the training progresses, most activations are saturated -gradients at this point is close to zero.

Xavier and He's Weight Initialization



$$s_{11} = \sum_{i=1}^{n} w_{1i} x_i$$

For k-layers:

$$\boxed{Var(s_{ki}) = [nVar(w)]^k Var(x)}$$

$$Var(s_{11}) = Var(\sum_{i=1}^{n} w_{1i} x_i) = \sum_{i=1}^{n} Var(w_{1i} x_i) \quad = (nVar(w))(Var(x))$$

$$Var(S_{1i}) = (nVar(w))(Var(x))$$

Xavier and He's Weight Initialization

To ensure that variance in the output of any layer does not blow up or shrink we want:

$$nVar(w) = 1$$

```
W = np.random.randn(fan_in, fan_out) / sqrt(fan_in)
```

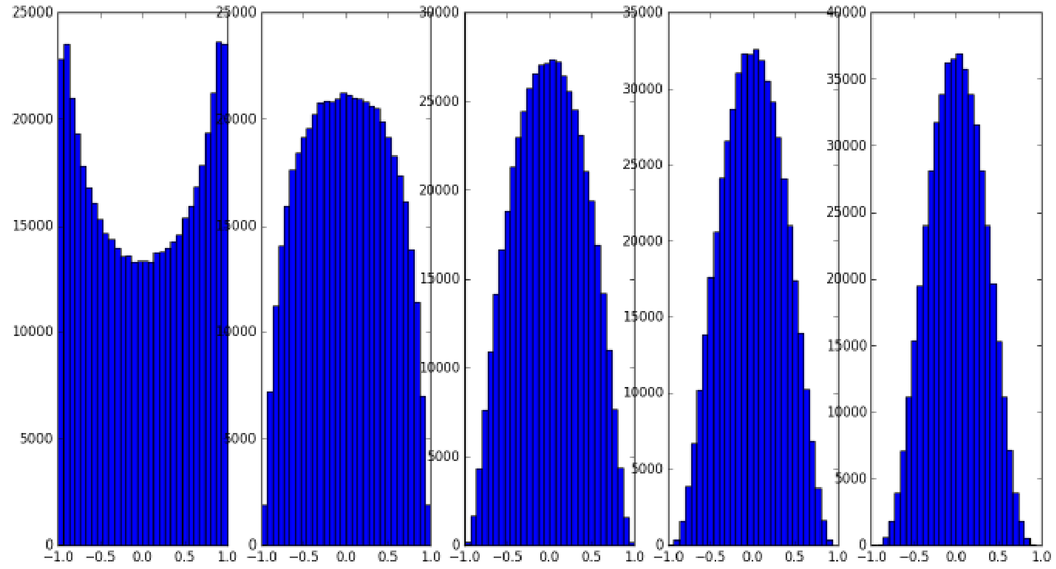
If we draw the weights from a unit Gaussian and scale them by $\frac{1}{\sqrt{n}}$ then, we have :

$$nVar(w) = nVar(\frac{z}{\sqrt{n}}) \quad = n * \frac{1}{n} Var(z) = 1$$

$$\boxed{Var(az) = a^2(Var(z))}$$

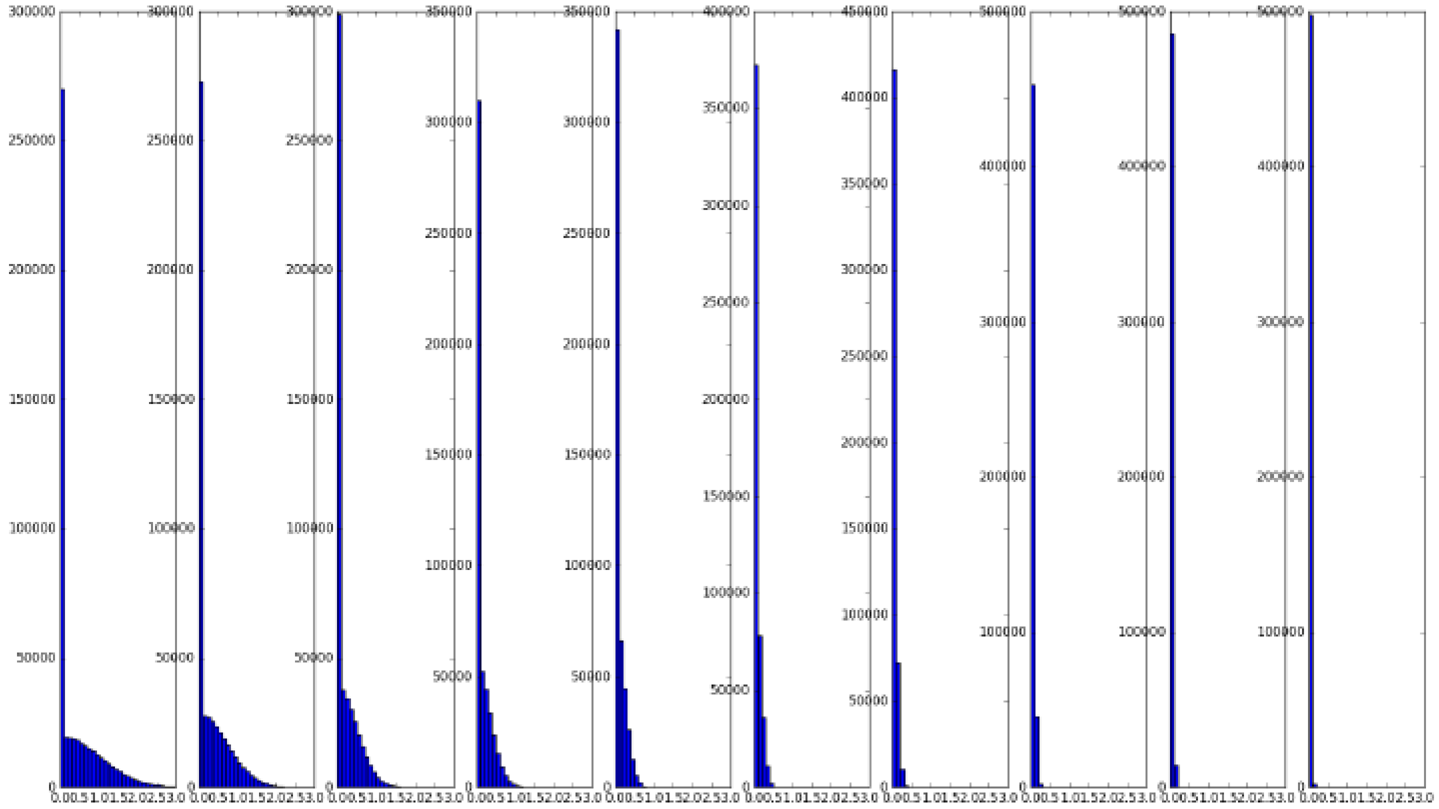
Xavier and He's Weight Initialization

```
W = np.random.randn(fan_in, fan_out) / sqrt(fan_in)
```



tanh activation

Xavier and He’s Weight Initialization

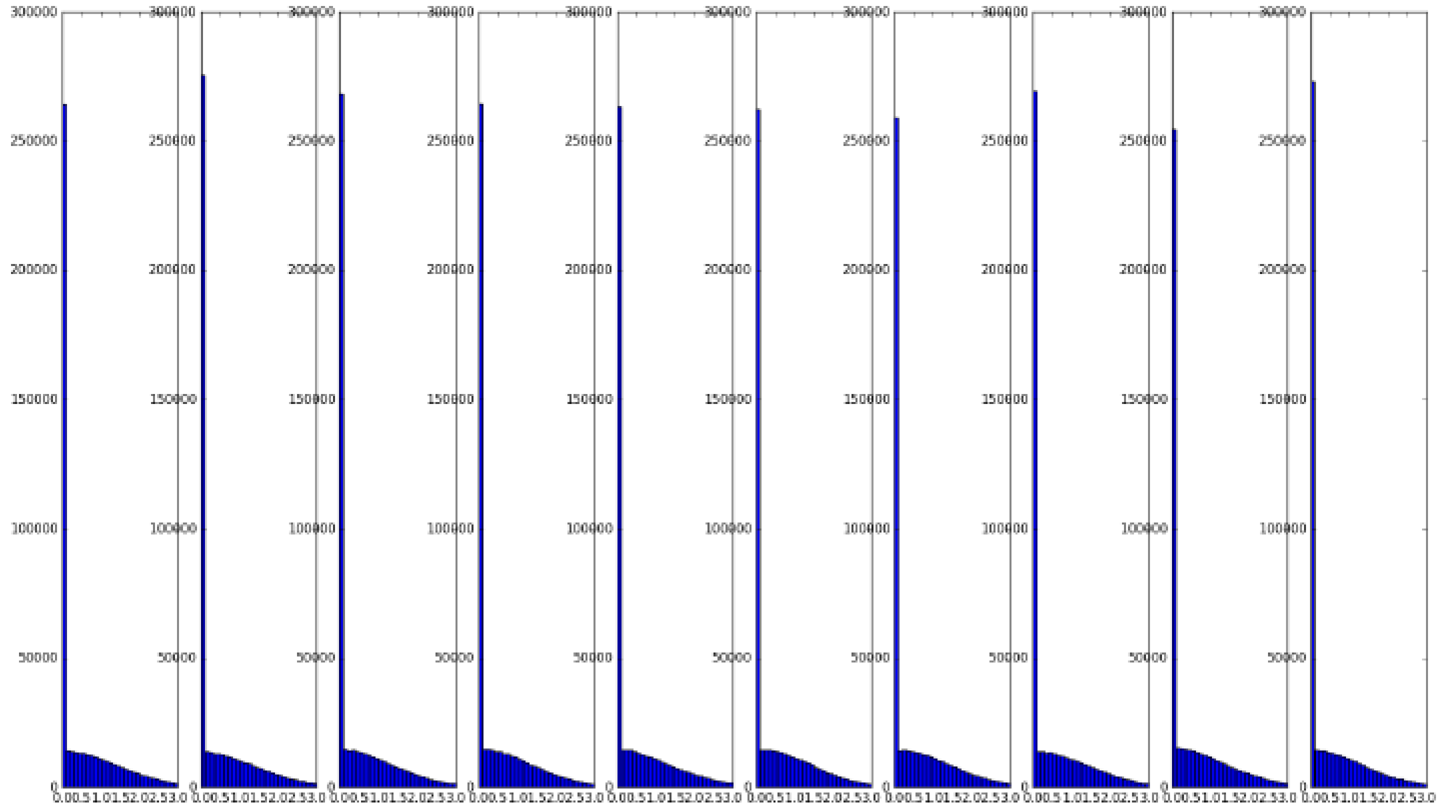


However, this initialization doesn't perform well with ReLU

Xavier and He's Weight Initialization

```
W = np.random.randn(fan_in, fan_out) / sqrt(fan_in/2)
```

A small tweak in the equation to improve the performance with ReLU



Solutions include

2. Using Non-saturating Activation functions

1. ReLU does not saturate for +ve values
2. Suffer from problem of dying ReLU- neurons output only 0
   - Weighted sum of its inputs are negative for all instances in training set
3. Use Leaky ReLU instead – (only go into coma don't die, may wake up)
   1. α = 0.01 , sometimes 0.2
4. Flavors include

   Randomized leaky ReLU(RReLU)

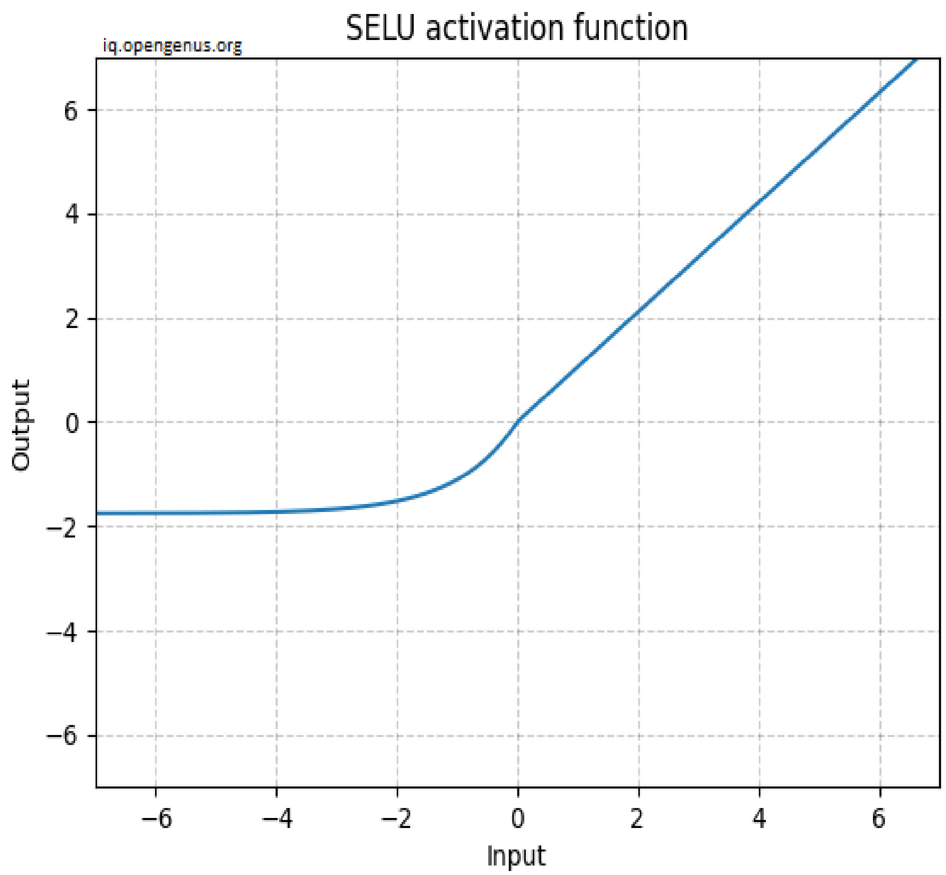
   α is picked randomly in a given range during training and is fixed to an average value during testing

   Parametric Leaky Relu (PReLU)

   α is authorised to be learned during training as a parameter

SELU

- **Scaled Exponential Linear Units**
  - Induce self-normalization
  - the output of each layer will tend to preserve mean 0 and standard deviation 1 during training
  - f(x) = λx if x>= 0
  - f(x) = λα(exp(x)-1) if x < 0
  - α = 1.6733 , λ= 1.0507
- conditions for self-normalization to happen:
  - The input features must be standardized (mean 0 and standard deviation 1).
  - Every hidden layer’s weights must also be initialized using normal initialization.
  - The network’s architecture must be sequential.



Solutions Include

3**. Batch Normalization -Ioffe and Szegedy** (2015)

- designed to solve the vanishing/exploding gradients problems, is also a good regularizer
- BN layer performs the standardizing and normalizing operations on the input of a layer coming from a previous layer.
- Normalization
  - brings the numerical data to a common scale without distorting its shape.
  - (mean = 0, std dev= 1)
- BN adds extra operations in the model , before activation
  - Operation zero centres and normalizes each input
  - Then scale and shift the result using two new parameter vectors per layer
  - Each BN Layer learn 4 parameter vectors
    - Output Scale Vector
    - Output offset vector
    - Input mean vector
    - Input standard deviation
- To zero-centre and normalize the inputs , mean and standard deviation of input needs to be computed
- Current mini-batch is used to evaluate mean and standard deviation

Solutions Include

3. Batch Normalization -Ioffe and Szegedy (2015)

1. $$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

2. $$\sigma_B^{\,2} = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mu_B\right)^2$$

3. $$\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^{\,2} + \epsilon}}$$

4. $$\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$$

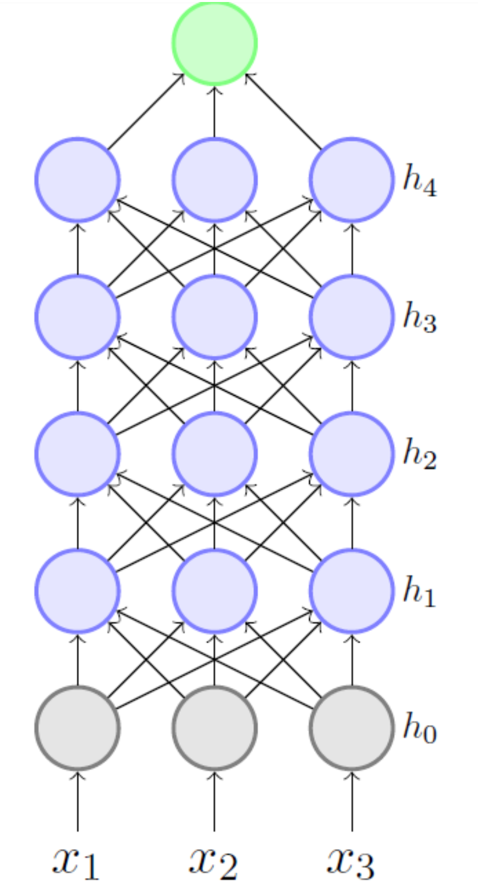
μ_B is the vector of input means, evaluated over the whole mini-batch B (it contains one mean per input).

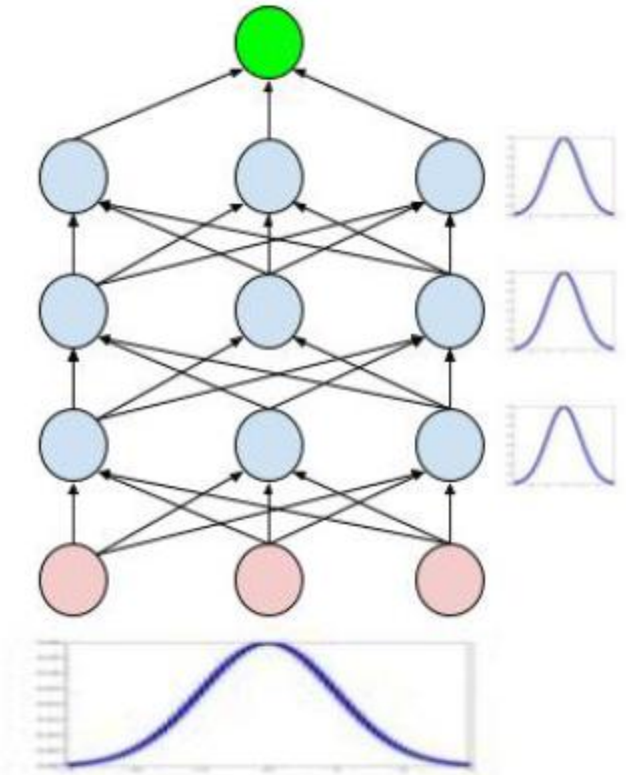
- σ_B is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
- m_B is the number of instances in the mini-batch.
- $\hat{\mathbf{x}}^{(i)}$ is the vector of zero-centered and normalized inputs for instance i.
- γ is the output scale parameter vector for the layer (it contains one scale parameter per input).
- $\otimes$ represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).
- β is the output shift (offset) parameter vector for the layer (it contains one offset parameter per input). Each input is offset by its corresponding shift parameter.
- ϵ is a tiny number to avoid division by zero (typically 10^{-5}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation: it is a rescaled and shifted version of the inputs.

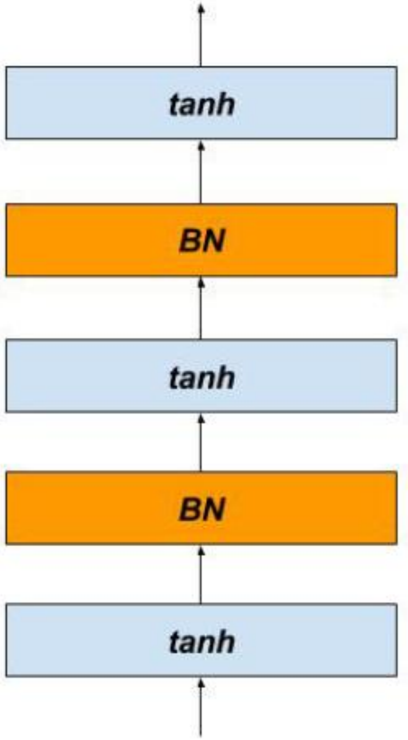
Aurelien Geron, “Hands-On Machine Learning with Scikit-Learn , Keras & Tensorflow, OReilly Publications

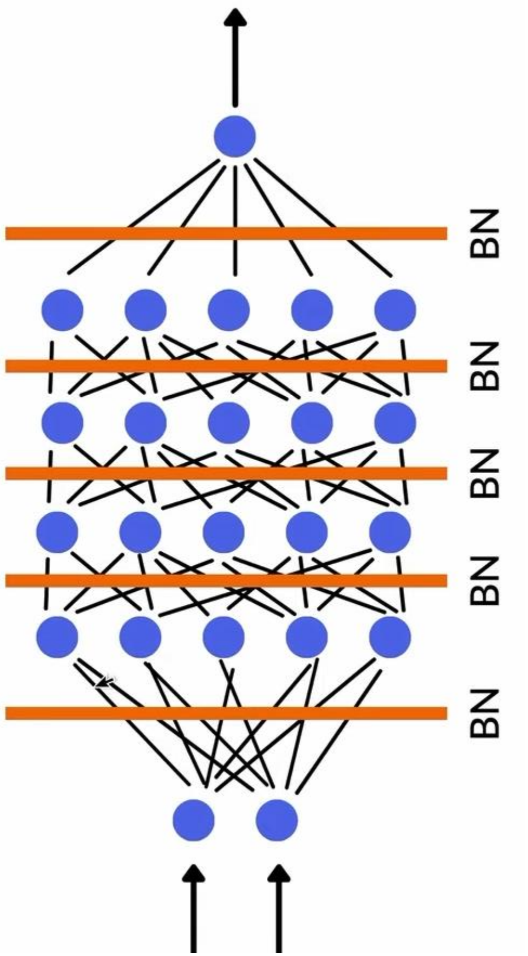
Batch Normalization

$$\hat{s_{ik}} = \frac{s_{ik} - E[s_{ik}]}{\sqrt{var(s_{ik})}}$$









$$y^{(k)} = \gamma^k \hat{s_{ik}} + \beta^{(k)}$$

$$\gamma^k = \sqrt{var(x^k)}$$

$$\beta^k = E[x^k]$$

Implementing Batch Normalization in Keras

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

```
>>> model.summary()
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization_v2 (Batc (None, 784)               3136
_________________________________________________________________
dense_50 (Dense)             (None, 300)               235500
_________________________________________________________________
batch_normalization_v2_1 (Ba (None, 300)               1200
_________________________________________________________________
dense_51 (Dense)             (None, 100)               30100
_________________________________________________________________
batch_normalization_v2_2 (Ba (None, 100)               400
_________________________________________________________________
dense_52 (Dense)             (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

Solutions include

4. Gradient Clipping

- Clip gradient during back propagation so that they never exceed some threshold

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

- All the partial derivatives of the loss will be clipped between -0.1 to 0.1
- Threshold can also be a hyperparameter to tune

5. Reusing Pretrained Layers

- Transfer Learning