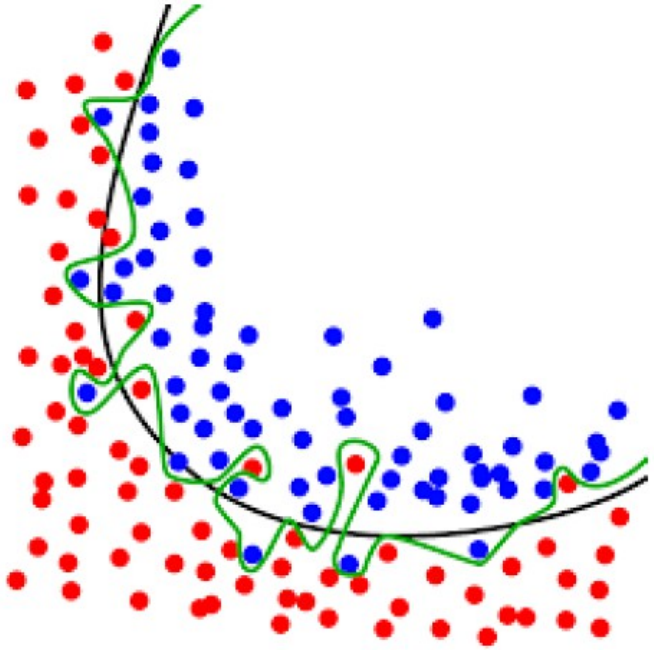
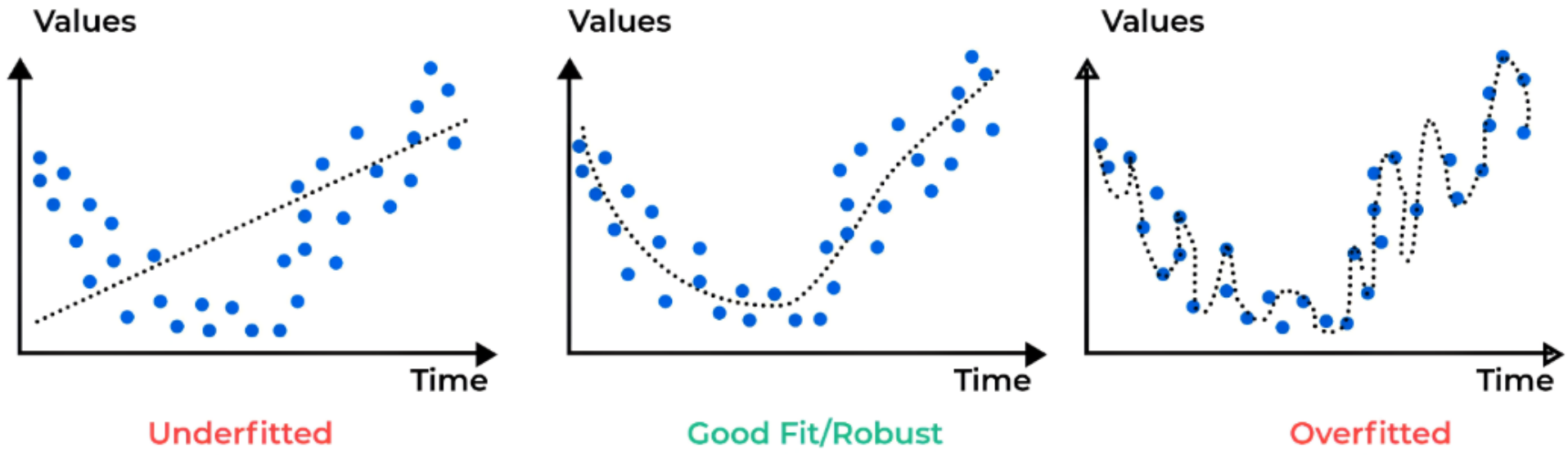
Problem: OverFitting in a Neural Network

- Why does overfitting happen in a neural network?
  - There are Too many free parameters.

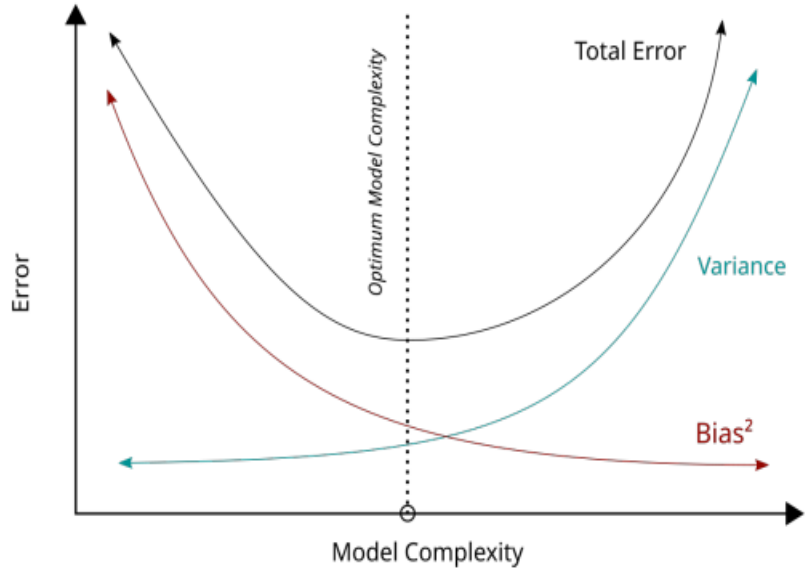


- **Recall from previous lectures:** Overfitting vs Underfitting
- **Overfitting:** Learning the underlying patterns and noise in the training data.
  - High accuracy on training data but poor generalization on unseen data.
- **Underfitting:** The model is too simple to capture the patterns in the data.
  - Poor performance on both training and test data.



Bios variance trade off

- **Tradeoff in Neural Networks:** Increasing layers and parameters can reduce bias but increase variance.
- **Finding the Balance: Regularization** methods help control this tradeoff by **adjusting model complexity**.



Regularization

- It is possible that some features that is actually irrelevant appears by chance to be useful, resulting in overfitting
- Use regularization to avoid overfitting
- Regularization:
  - Minimize the Empirical Loss and the Complexity of the model
  - $Cost = Empirical\ Loss + \lambda\ Complexity$
  - Technique that constrains our optimization problem to discourage complex models
  - Why we need it ? Improve generalization of our model on unseen data

Regularization

- **Goal:** Prevent overfitting.
- **Key Idea:** Favor simpler models to avoid fitting noise in the data.

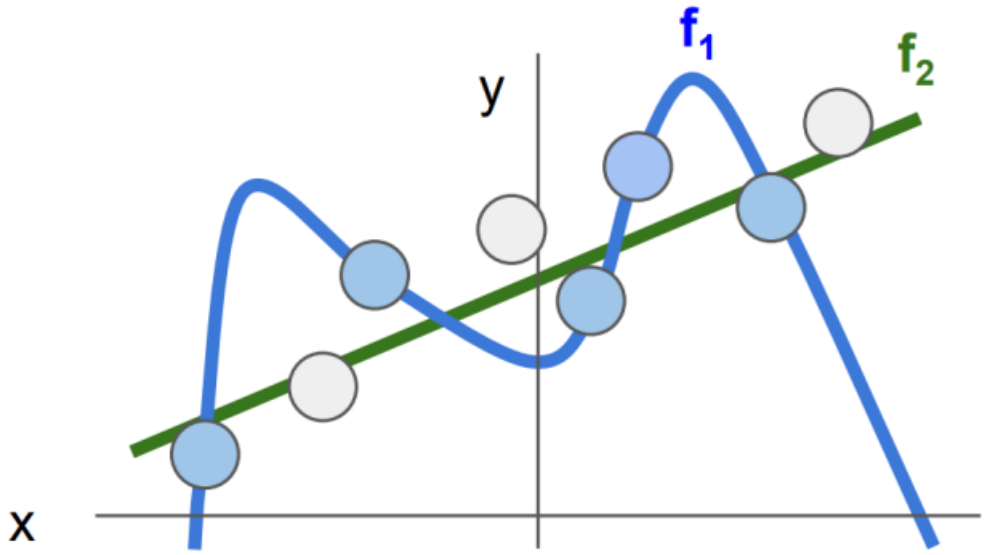


Figure 5: Regularization prevents overfitting by avoiding fitting noise in the data.

Regularization in NN

$$J_\lambda(w) = \underbrace{J(w)}_{\text{Data loss}} + \underbrace{\lambda R(w)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Regularization

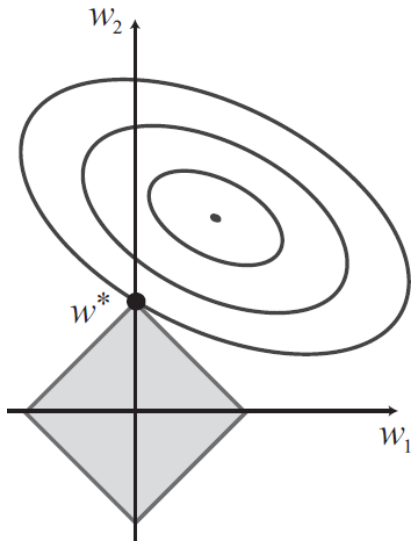
- In NN the complexity can be specified as a function of the weights
  - For e.g. $Complexity = \sum_i |w_i|^q$
- With q=1 we have L1 regularization which minimize the sum of the absolute values
- With q=2 we have L2 regularization which minimize the sum of squares
- Which regularization should pick ?
  - Depend on the specific problem
  - L1 regularization tend to produce sparse model (often set many weights to zero)
- Minimizing $Loss(w) + \lambda\, Complexity$(w) is equivalent to minimizing Loss(w) subject to constraint that $Complexity\,(w) =; c$ for some constant c that is related to λ

L1 Regularization

- Figure shows the diamond-shaped box represents the set of points **w** in two-dimensional weight space that have L1 complexity less than c
- The solution is somewhere inside the box
- The concentric ovals represent contours of the loss

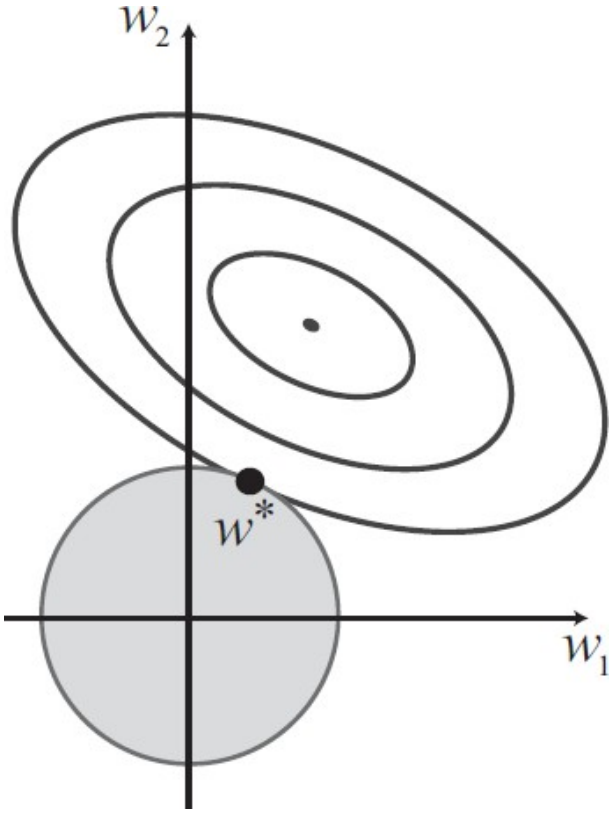
function, with the minimum loss at the center

- We want the point in the box that is closest to the minimum;
- For an arbitrary position of the minimum and its contours, it will be common for the corner of the box to find its way closest to the minimum



L2 Regolarization

- L2 complexity measure, represent a circle instead of diamond
- in general, there is no reason for the intersection to appear on one of the axes
- L2 regularization does not tend to produce zero weights



- What is advantage of the each regularization ?

- L1 (Lasso): Sparse weights, feature selection

$$\text{L1 Regularization} = \lambda \sum_{j=1}^{m} |w_j|$$

- L2 (Ridge): Adds a penalty for large weights to the loss function

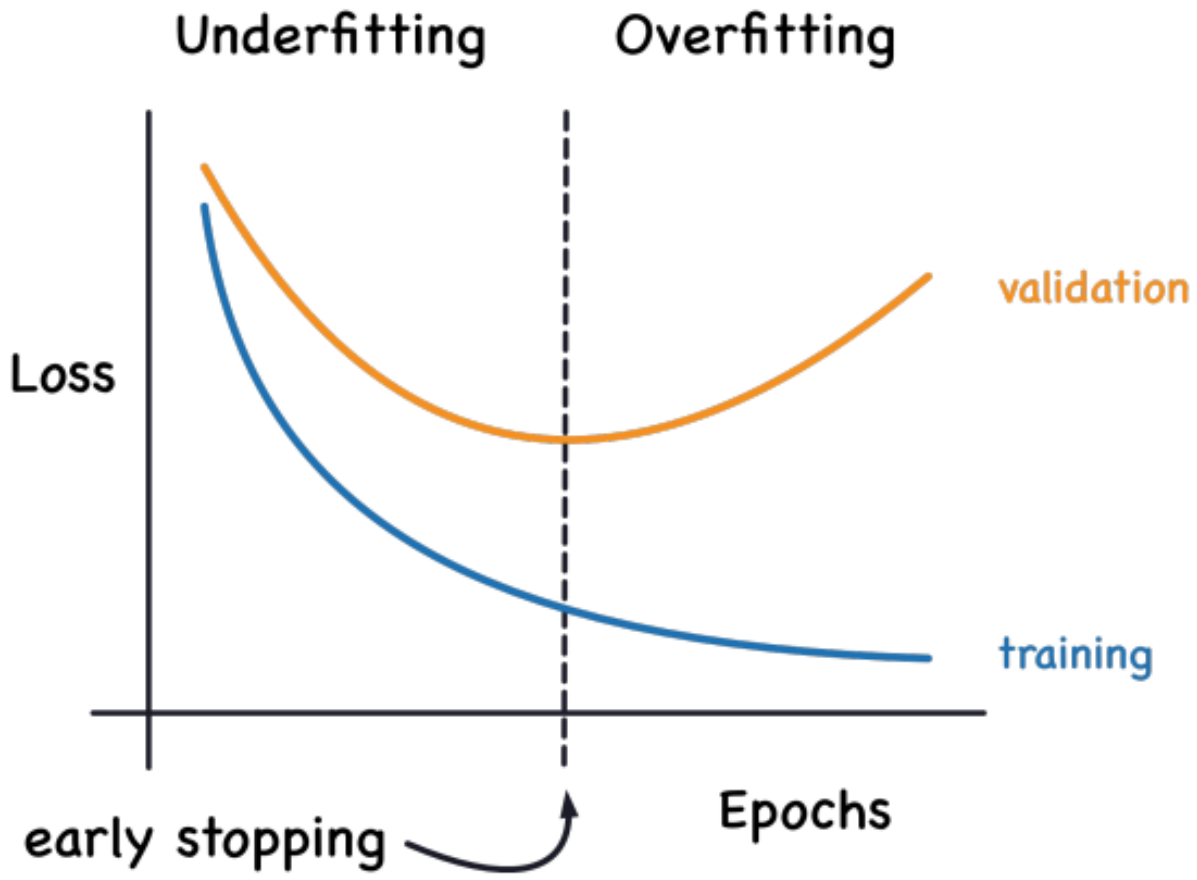
$$\text{L2 Regularization} = \lambda \sum_{j=1}^{m} w_j^2 = \lambda \mathbf{W}^T \mathbf{W}$$

- Elastic Net (L1 + L2): Combines both L1 and L2 penalties, where β controls the balance between L1 and L2 regularization.

$$\text{Elastic net Regularization} = \lambda \sum_{j=1}^{m} (\beta |w_j| + \frac{1-\beta}{2} w_j^2)$$

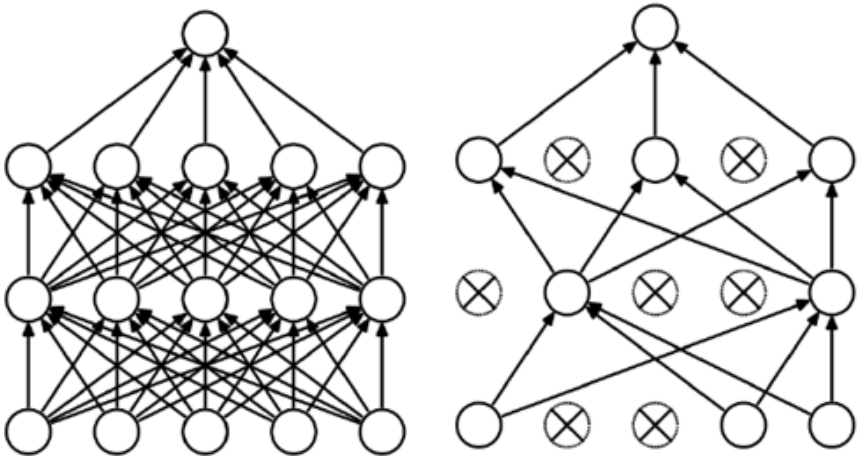
Solution 2: Early Stopping

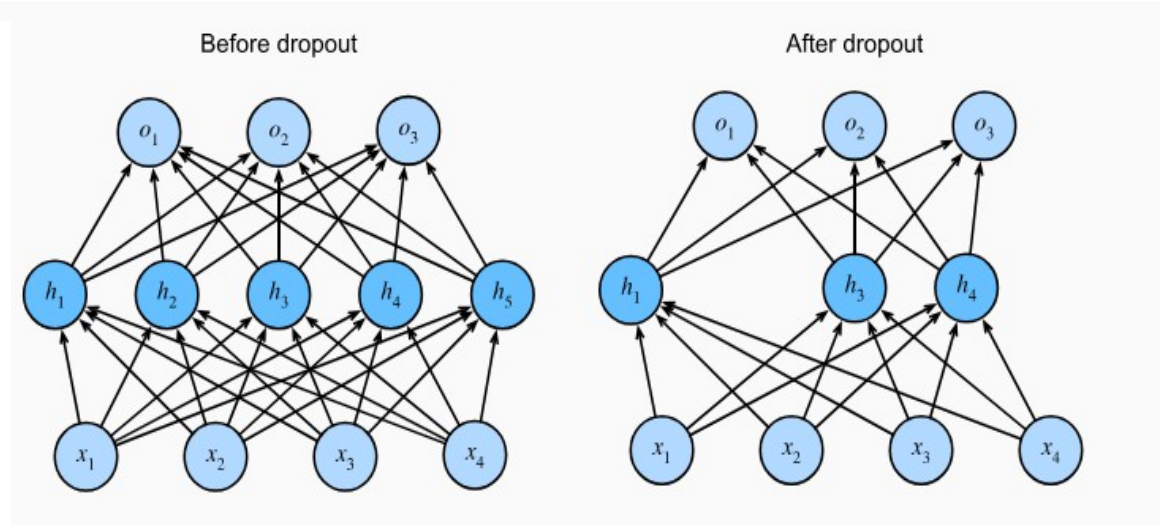
- Stop the training procedure when the validation error is minimum.



Dropout

- Dropout is a regularization technique that reduces overfitting in neural networks.
  - **Randomly deactivates** neurons in each layer during training.
  - Effectively trains **multiple "thinned" sub-networks**, then averages them at test time.
- This makes neurons more robust, as they cannot rely on specific other neurons to perform well.

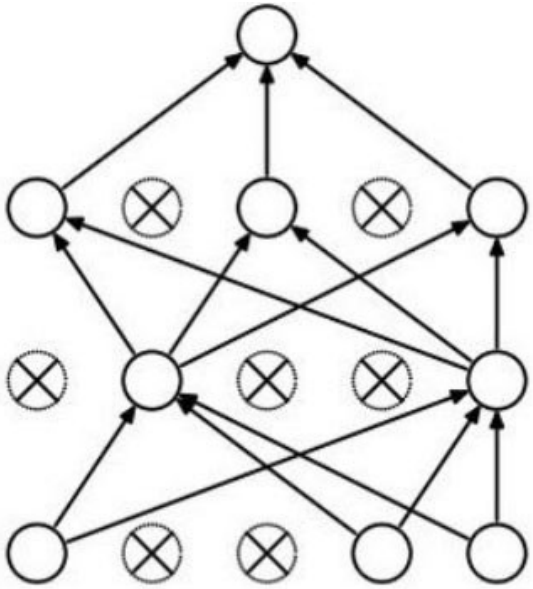




Dropout: Why can this possibly be a good idea?

- Dropout-trained neurons are unable to co-adapt with their surrounding neurons.
- They also can't depend too heavily on a small number of input neurons.
- They become less responsive to even little input changes.
- The result is a stronger network that generalizes better.

Forces the network to have a redundant representation



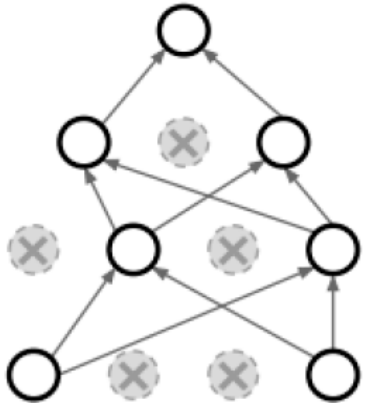
Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model, gets trained on only ~one datapoint.

Dropout: Why can this possibly be a good idea?

- Dropout trains a large ensemble of models that share parameters.
- Every possible dropout state for neurons of a network, which is called a mask, is one model.



Dropout: Test Time

- Dropout makes our output random at training time.

$$y = f_W(x, \underbrace{z}_{\text{random mask}})$$

- We want to average out the randomness at test time,

$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

- But this integral seems complicated.
- Let's approximate the integral for a superficial layer where dropout rate is 0.5.

Dropout: Test Time

$$E_{train}[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + w_2 y) + \frac{1}{4}(0x + 0y)$$

$$= \frac{1}{2}(w_1 x + w_2 y)$$

$$E_{test}[a] = w_1 x + w_2 y$$

$$\Rightarrow E_{train}[a] = \underbrace{0.5}_{\text{keep probability}} \quad E_{test}[a]$$

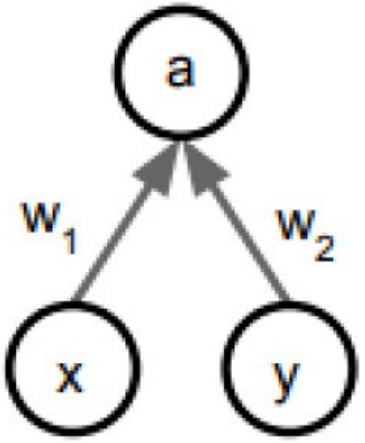


Figure: Simple neural network. [6]

If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time

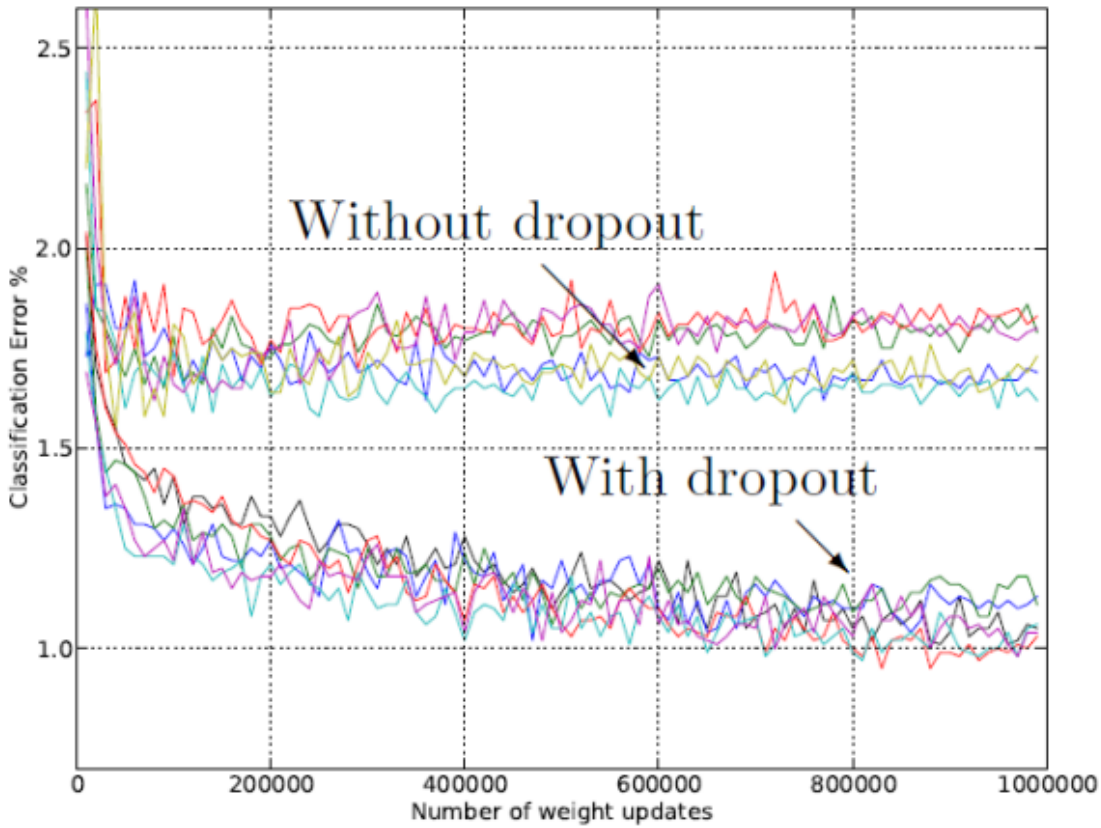


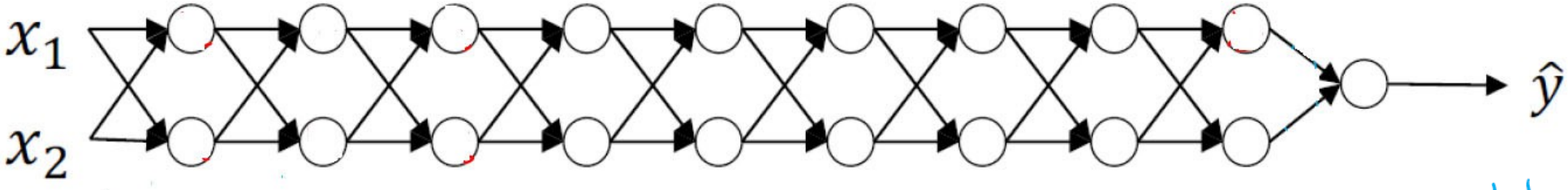
Figure 8: Effect of dropout for different architectures: adapted from *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*

Vanishing/Exploding Gradient

- Vanishing
  - Gradients often get smaller as the algorithm progresses down. As a result, gradient descent updates do not effectively change the weights of the lower layer connections, and training never converges.
  - Make learning slow especially of front layers in the network.
- Exploding
  - Gradients can get bigger and bigger, so there are very large weight updates at many levels, causing the algorithm to diverge.
  - The model is not learning much on the training data therefore resulting in a poor loss.

Exploding Gradient

- Root cause of exploding gradients
  - network architecture and the choice of activation functions
    - when multiple layers have weights greater than 1, the gradients can grow exponentially as they propagate back through the network during training
    - exacerbated when using activation functions with outputs that are not bounded (hyperbolic tangent or the sigmoid function)
  - initialization of the network's weights
    - If the initial weights are too large, even a small gradient can be amplified through the layers,

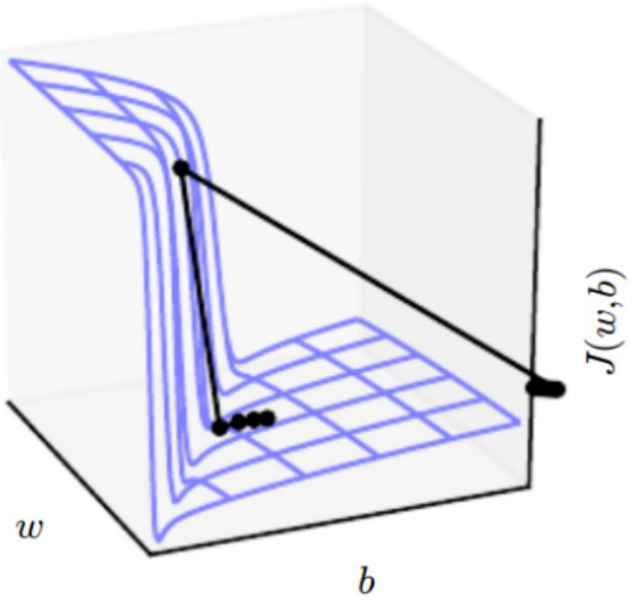


$\hat{y} = w^{[L]} w^{[L-1]} w^{[L-2]} \cdots w^{[3]} w^{[2]} w^{[1]} X$

$\hat{y} = w^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} X$

Consequences of Exploding Gradients

- weight updates during training can become so large that they cause the learning algorithm to overshoot the minima of the loss function
- This can result in model parameters diverging to infinity, causing the learning process to fail



Solutions to exploding gradient

- **Gradient clipping** :
  - This technique involves setting a threshold value, and if the gradient exceeds this threshold, it is scaled down to keep it within a manageable range. This prevents any single update from being too large.
- **Weight Initialization:**
  - Using a proper weight initialization strategy, such as Xavier or He initialization, can help prevent gradients from becoming too large at the start of training.
- **Batch Normalization:**
  - maintain the output of each layer within a certain range, reducing the risk of exploding gradients.

Gradient Clipping

- Two approaches to do so:
  - Clipping by value
  - Clipping by norm

Gradient Clipping by value

- Set a max (α) and min (β) threshold value.
- For each index of gradient $\boldsymbol{g}_i$ if it is lower or greater than your threshold clip it:

$$
\begin{aligned}
&\text{if } \boldsymbol{g}_i > \alpha : \\
&\qquad \boldsymbol{g}_i \leftarrow \alpha \\
&\text{else if } \boldsymbol{g}_i < \beta : \\
&\qquad \boldsymbol{g}_i \leftarrow \beta
\end{aligned}
$$

Gradient clipping by norm

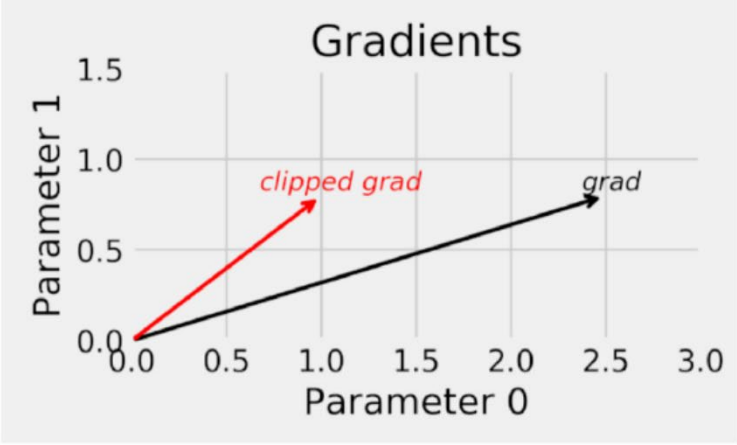


Figure: The effect of clipping by value. Source

- Clipping by value will change gradient direction.
- To preserve direction use clipping by norm.

Gradient clipping by norm

- Clip the norm $\|\boldsymbol{g}\|$ of the gradient $\boldsymbol{g}$ before updating parameters:

$$\text{if } \|\boldsymbol{g}\| > v:$$
$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}}{\|\boldsymbol{g}\|} v$$

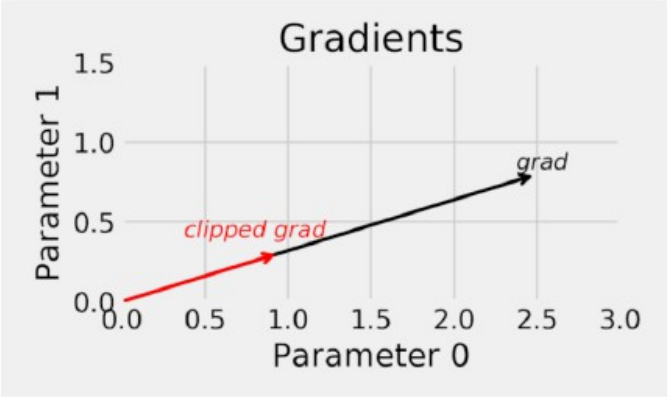


Figure: The effect of clipping by norm. source

solution: **gradient clipping**

Algorithm 1 Pseudo-code for norm clipping

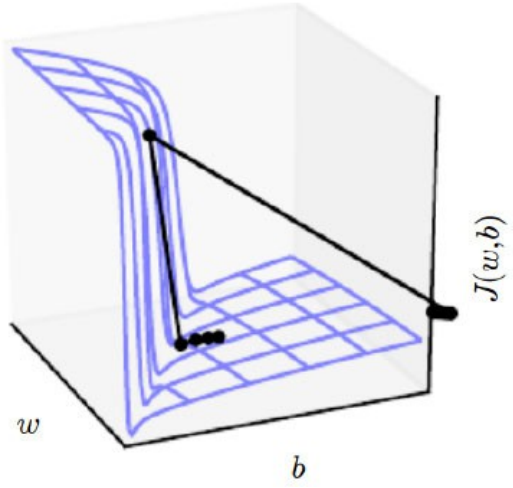
$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$
end if

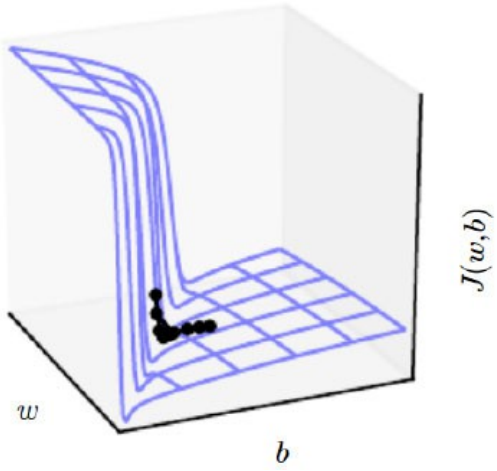
v is the threshold for clipping which is a hyperparameter.

- Gradient clipping saves the direction of gradient and controls its norm.

Gradient Clipping

- Clipping by value will change the direction of the gradient, so it will send us to a bad neighborhood.
- Clipping by norm will preserve the direction and just control the value.
- So it is better to use clipping by norm.





Weight initialization

- A bad initialization may increase convergence time or even make optimization diverge
- How to initialize?
  - Zero initialization
  - Random initialization

Weight Initialization: Zero Initialization

Zero Initialization method:

$$\begin{cases} W^{[l]} = \mathbf{0}, \\ b^{[l]} = \mathbf{0} \end{cases}$$

- Simple but perform very poorly. (why?)
- Zero initialization will lead each neuron to learn the same feature
- This problem is known as network failing to break symmetry
- In fact any constant initialization suffers from this problem.

Weight Initialization: Zero Initialization

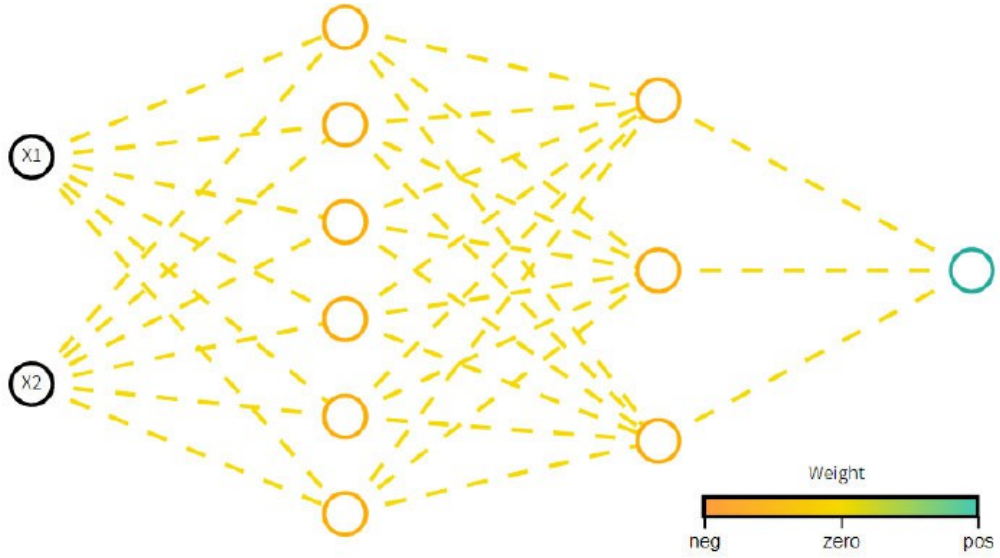


Figure: As we can see network has failed to break symmetry. There has been no improvement in weights after about 600 epochs of training

- We need to break symmetry. How? using randomness.

Weight Initialization: Random Initialization

- To use randomness in our initialization we can use uniform or normal distribution:

General Uniform Initialization:

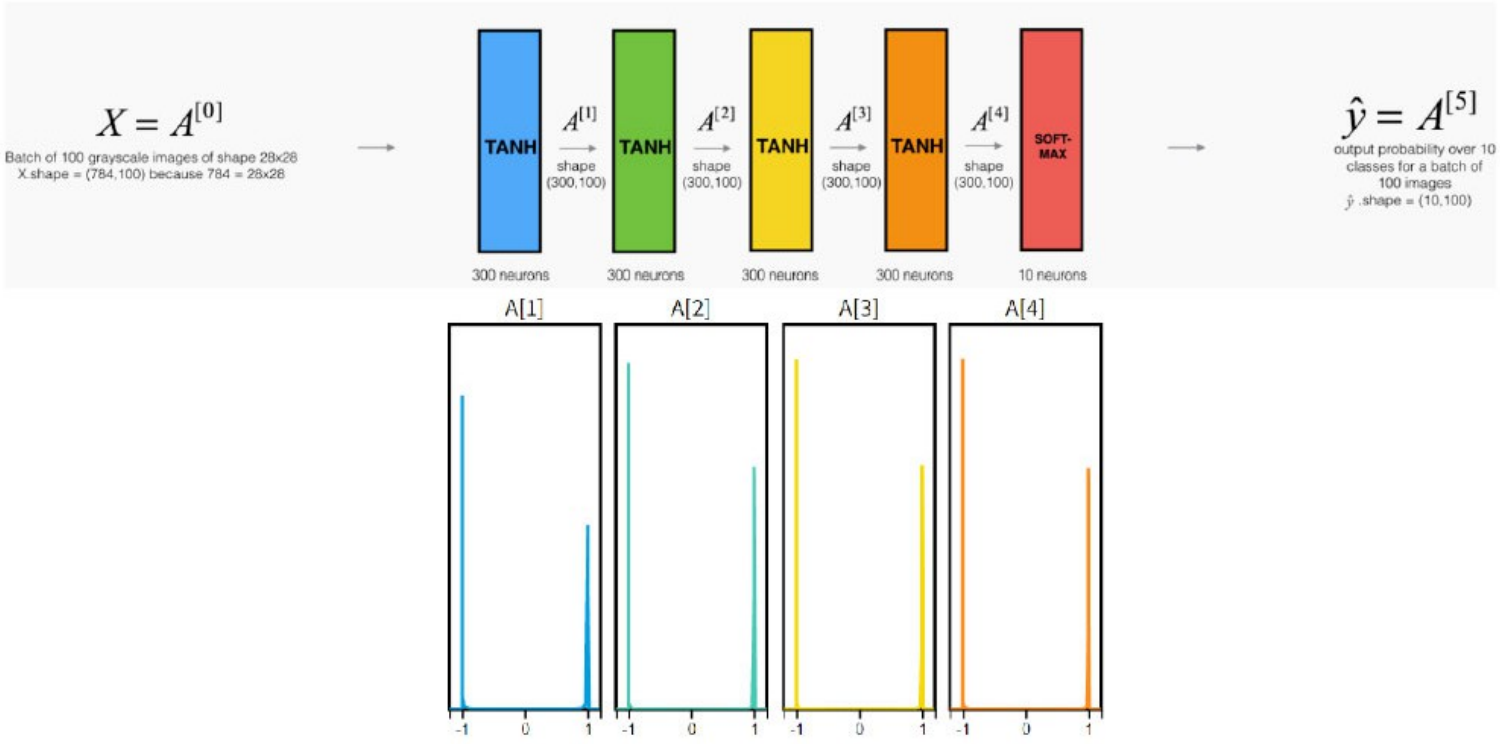
$$\begin{cases} W^{[l]} \sim U(-r, +r), \\ b^{[l]} = 0 \end{cases}$$

General Normal Initialization:

$$\begin{cases} W^{[l]} \sim \mathcal{N}(\mu = 0, \sigma^2), \\ b^{[l]} = 0 \end{cases}$$

- But this is really crucial to choose r or σ properly.

Weight Initialization: Random Initialization



Uniform initialization problem. On the top, you can see the model architecture, and on the bottom, you can see the density of each layer's output. Model has trained on MNIST dataset for 4 epochs. Weights are initialized randomly from $U(\frac{-1}{\sqrt{n^{[l-1]}}}, \frac{1}{\sqrt{n^{[l-1]}}})$

Weight Initialization: Random Initialization

- How to choose r or σ?
- We need to follow these rules:
  - keep the mean of the activations zero.
  - keep the variance of the activations same across every layer.

Xavier Initialization:

- For Uniform distribution use:

$$r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$$

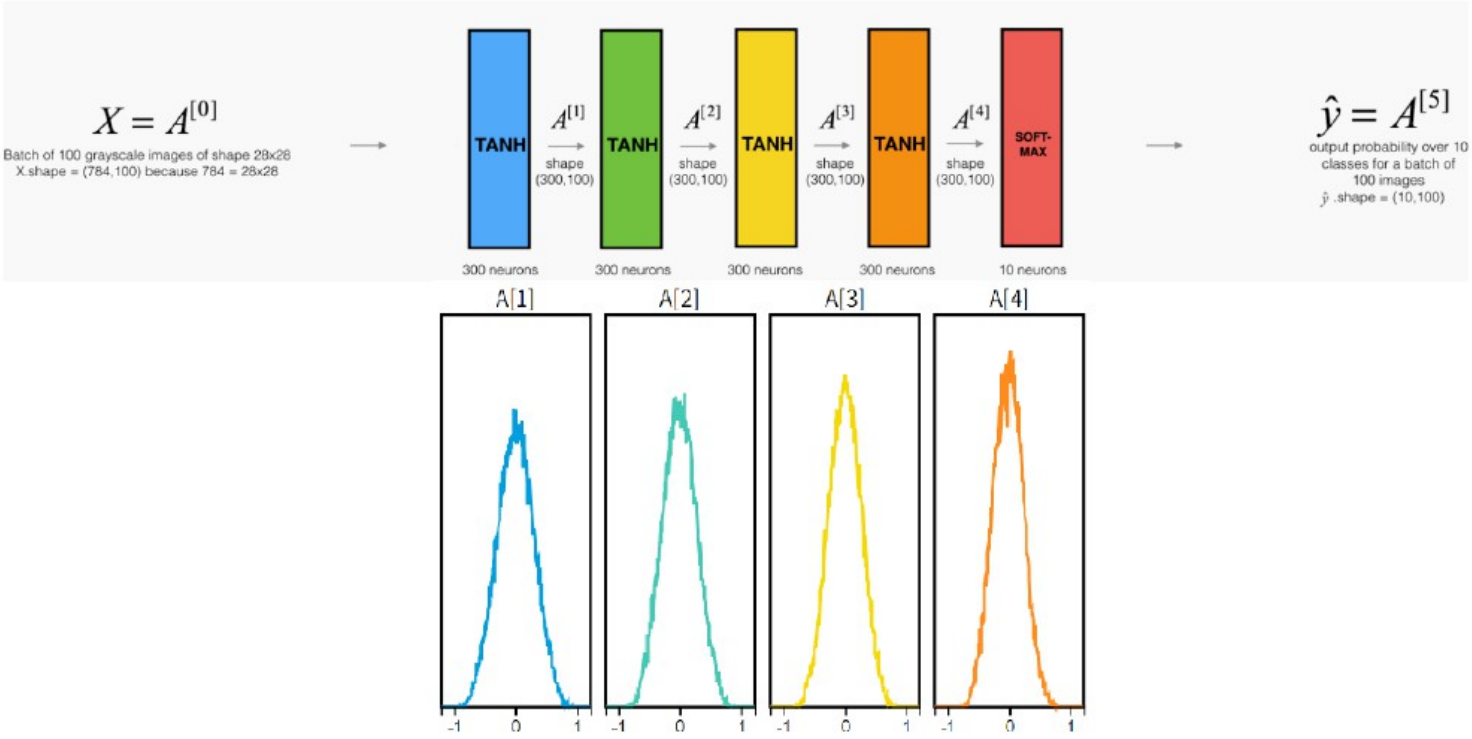
$$\begin{cases} \text{fan}_{\text{in}}^{[l]} = n^{[l-1]} & \text{(layer } l \text{ number of inputs)}, \\ \text{fan}_{\text{out}}^{[l]} = n^{[l]} & \text{(layer } l \text{ number of outputs)}, \\ \text{fan}_{\text{avg}}^{[l]} = \frac{n^{[l-1]}+n^{[l]}}{2} \end{cases}$$

- For Normal distribution use:

$$\sigma^2 = \frac{1}{\text{fan}_{\text{avg}}}$$

Weight Initialization: Xavier Initialization

- Xavier initialization works well on Tanh, Logistic or Sigmoid activation function.



Vanishing gradient is no longer problem using Xavier initialization. Model has trained on MNIST dataset for 4 epochs.

Weight Initialization: He Initialization

- Different method has proposed for different activation functions.

He Initialization:

- For Normal distribution:

$$\sigma^2 = \frac{2}{n^{[l]}}$$

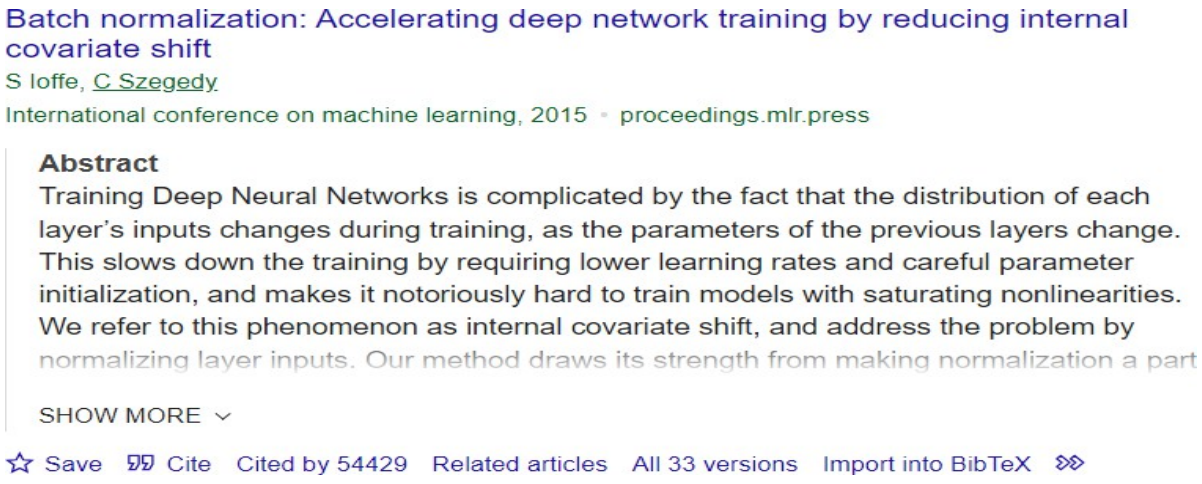
- For Uniform distribution:

$$r = \sqrt{3\sigma^2}$$

- He initialization works well on ReLU and its variants.

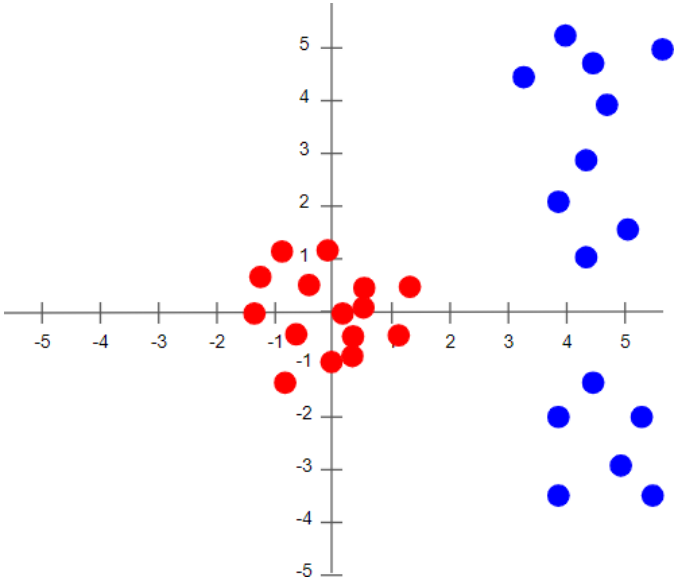
Bach normalization

- batch normalization has made it possible for practitioners to routinely train networks with over 100 layers
- A secondary benefit of batch normalization lies in its inherent regularization.



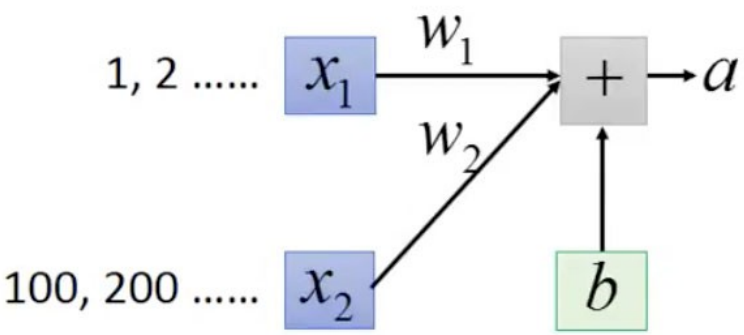
Why normalization is needed

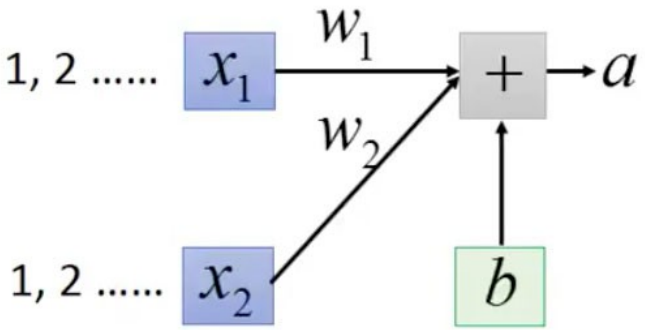
- feature might have a different range of values
  - values for feature x1 might range from 1 through 5 ,
  - values for feature x2 might range from 1000 to 99999.
- In the picture, the effect of normalizing data is shown.
- The original values (in blue) are centered around zero (in red).
- This ensures that all the feature values are now on the same scale.

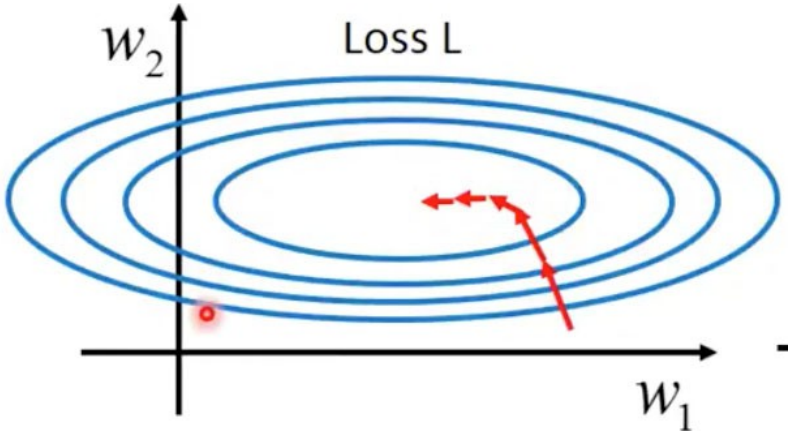


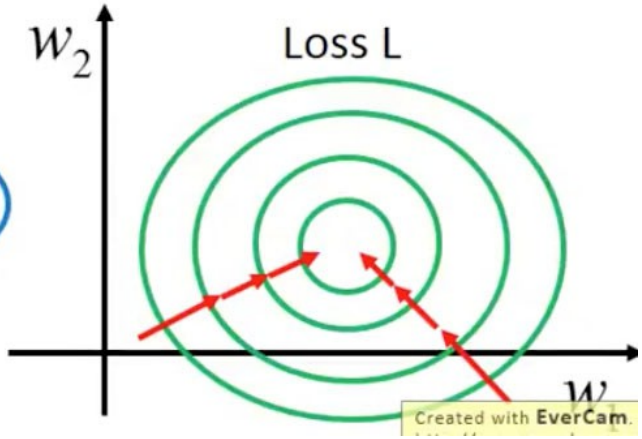
Feature Scaling

Make different features have the same scaling



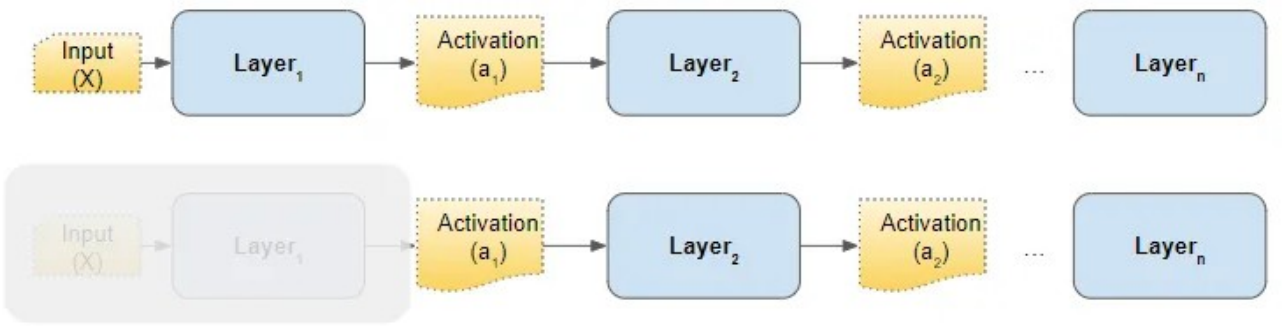






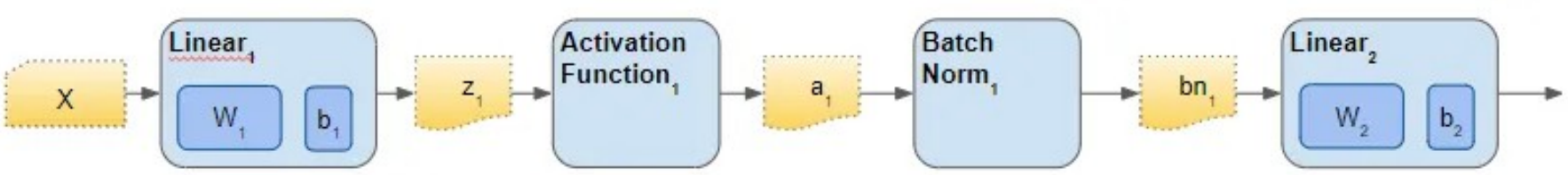
The need for batch norm

- The activations from the previous layer are simply the inputs to current layer.
- The same logic that requires us to normalize the input for the first layer will also apply to each of these hidden layers



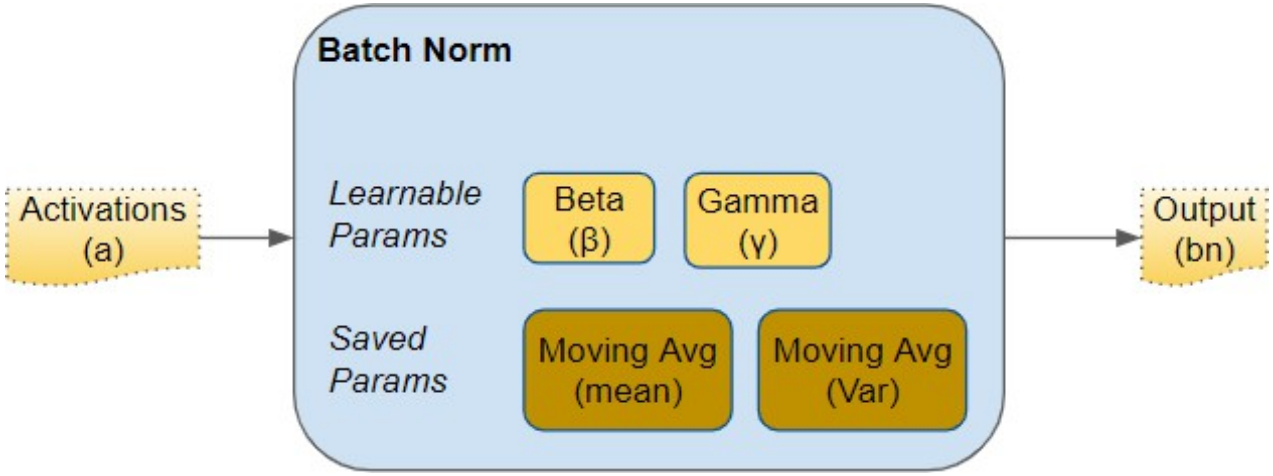
How does batch norm work?

- Batch Norm is just another network layer that gets inserted between a hidden layer and the next hidden layer
- Its job is to take the outputs from the first hidden layer and normalize them before passing them on as the input of the next hidden layer.
- Just like the parameters (eg. weights, bias) of any network layer, a Batch Norm layer also has parameters of its own:
  - Two learnable parameters called beta and gamma.



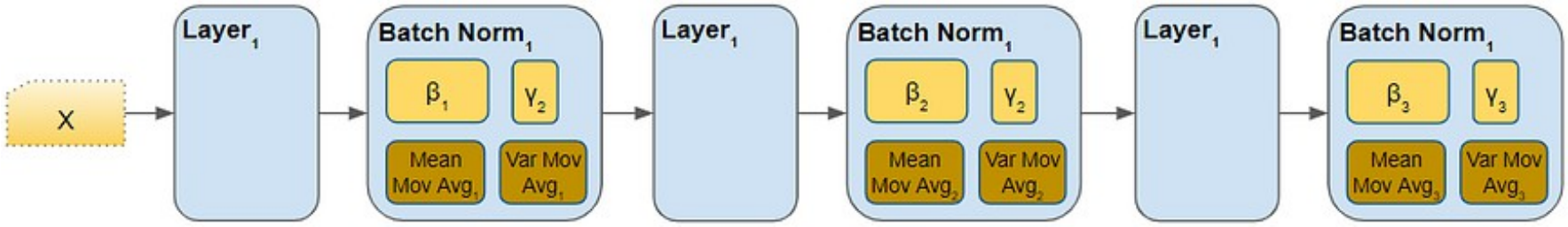
Batch norm layer

- Two learnable parameters called beta and gamma.
- Two non-learnable parameters (Mean Moving Average and Variance Moving Average) are saved as part of the ‘state’ of the Batch Norm layer.
-



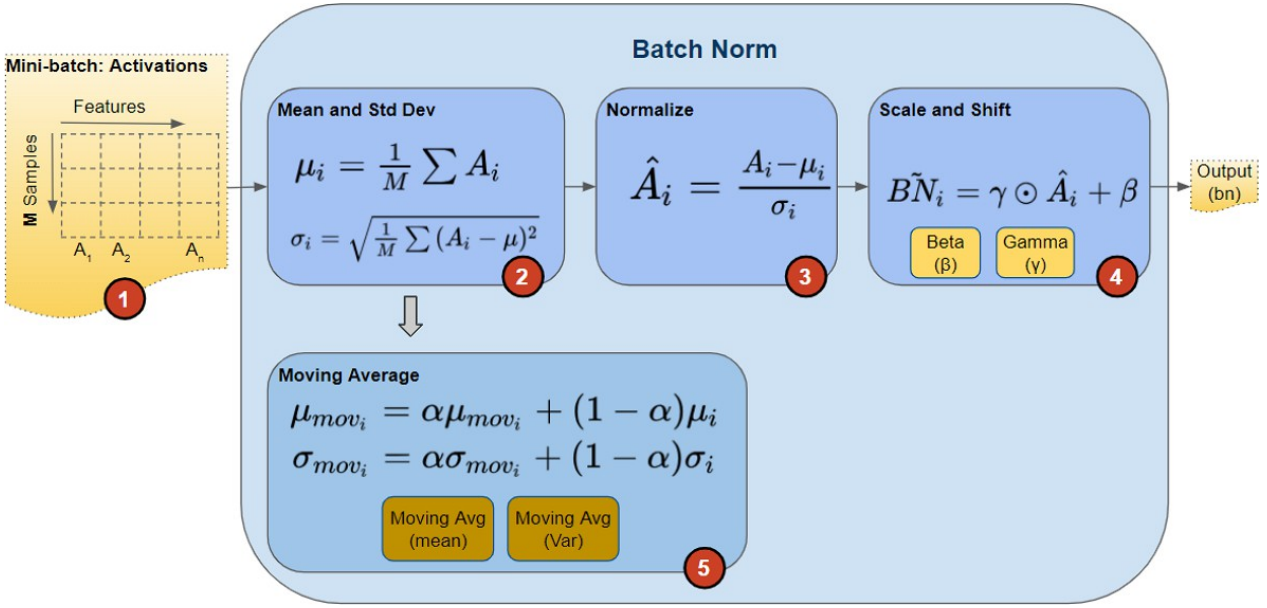
Batch norm layer

- parameters are per Batch Norm layer.
- if we have, say, three hidden layers and three Batch Norm layers in the network, we would have three learnable beta and gamma parameters for the three layers. Similarly for the Moving Average parameters.

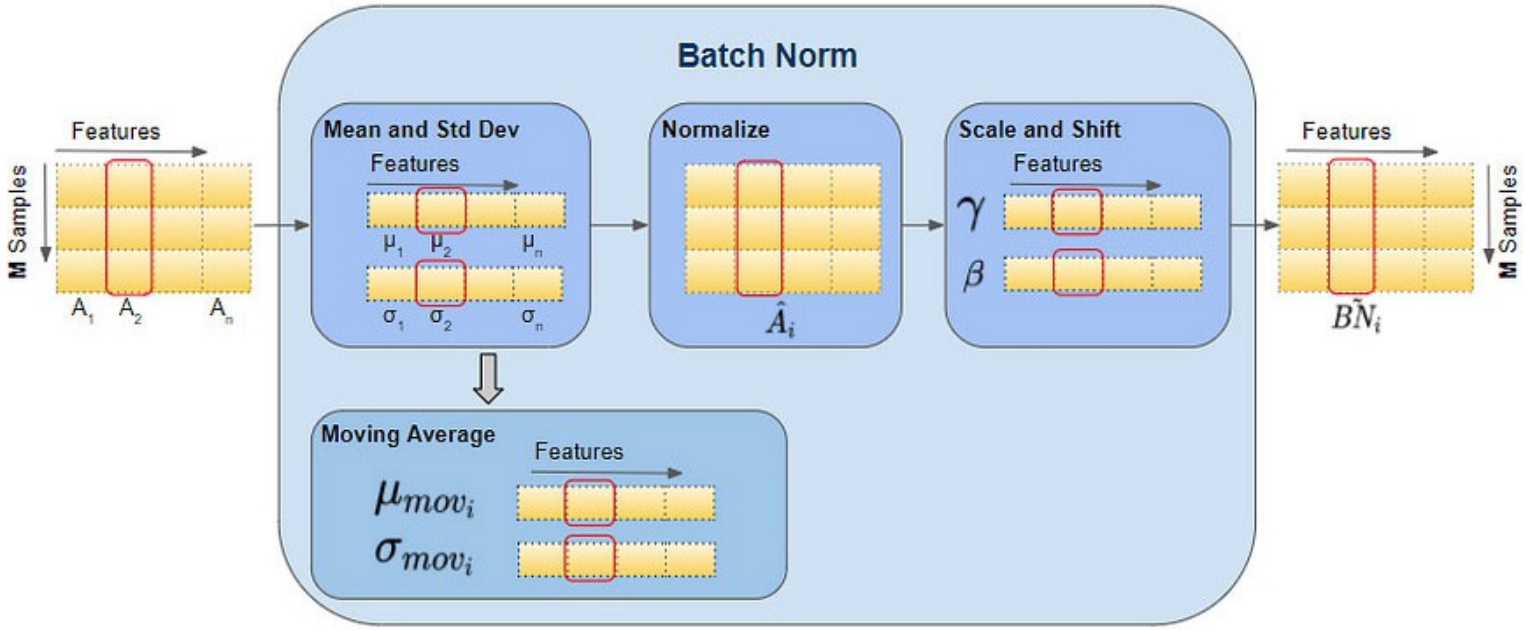


Batch norm during traing

- During training, we feed the network one mini-batch of data at a time.
- The Batch Norm layer processes its data as follows:
-



Batch norm layer vector shape



Batch norm during inference

- during Inference, we have a single sample, not a mini-batch. How do we obtain the mean and variance in that case?

Batch norm

- Pros
  - Vanishing and exploding gradient problem is reduced by a considerable amount
  - The network is much less sensitive to initial weight
  - We are able to use larger learning rates, which speeds up the training
  - It also acts as a regularizer
- Cons
  - It increase model parameter and prediction latency