DS-UA 301
Advanced Topics in Data Science
Advanced Techniques in ML and Deep Learning

LECTURE 4

Parijat Dube

Agenda

- Artificial Neural Network
- Deep Learning
- Deep Learning Training
- Deep Learning Hyperparameters
- ML performance improvement techniques
  - Regularization techniques
  - Data augmentation
  - Input scaling

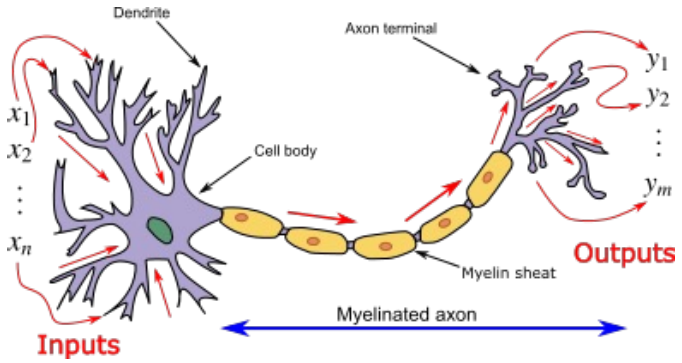
Artificial Neuron

- Artificial Neuron:

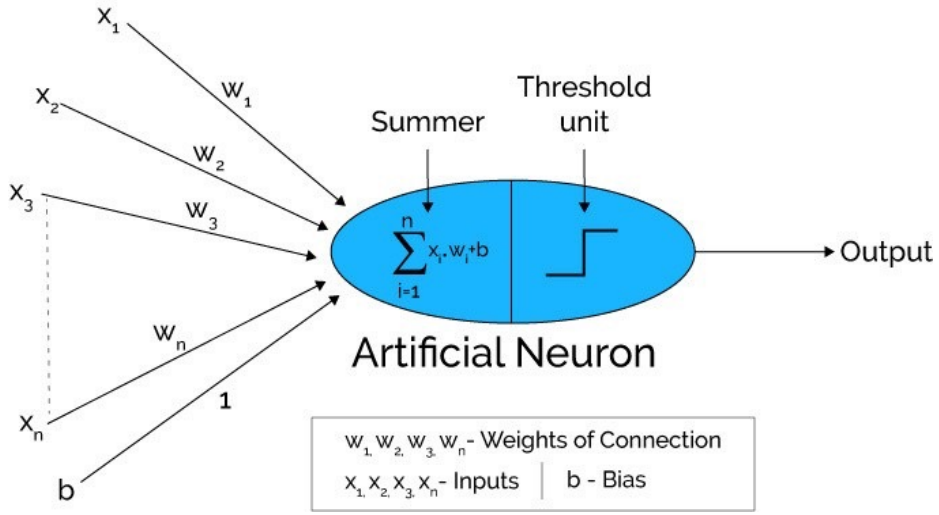
$$Z = W^T X + b$$

$$\hat{Y} = \mathrm{A(Z)}$$

- W^T *:* weights vector
- X *:* input features vector:
  - **1 sample has multiple features**
- $\hat{Y}$: prediction scalar
- b: bias scalar
- $A(Z)$: activation function (threshold scalar)



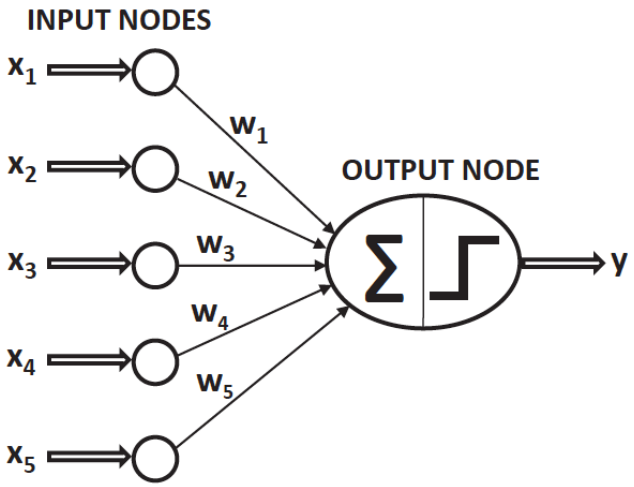
A Biological neuron (source: Wikipedia)



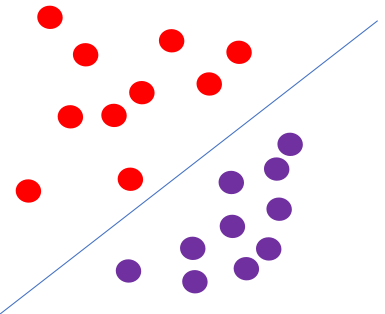
Artificial Neural Nets are an old idea…

- **Frank Rosenblatt** (NY) invents the perceptron in **1958**
- Simulated the perceptron on an IBM 704 computer at Cornell University
- IBM 704 -a 5-ton computer the size of a room
- Described as the first machine "capable of having an original idea."

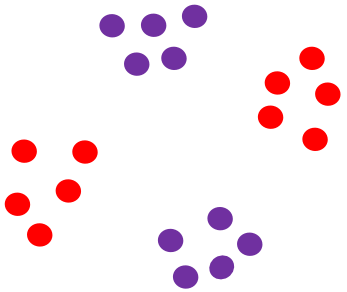
Single Layer Perceptron



Linearly separable



Not linearly separable



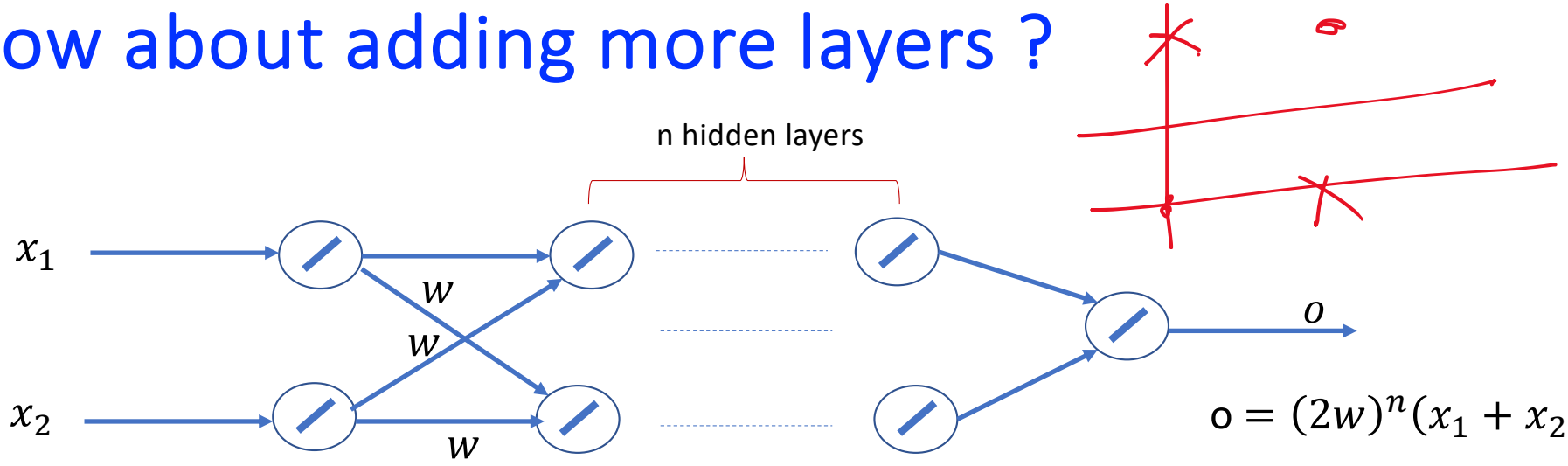
$\bar{W}.\bar{X} = 0$

$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X}\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j\}$$

$$\overline{W} \Leftarrow \overline{W} + \alpha \underbrace{(y - \hat{y})}_{\text{Error}} \overline{X}$$

Perceptron training uses one training data at each update
One cycle through the entire training data set is referred to as an epoch ⇒ Multiple epochs required
Perceptron training will not converge for not linearly separable dataset

How about adding more layers ?



Multi-layer neural network with linear activation functions ⇔ single layer neural network with linear activation

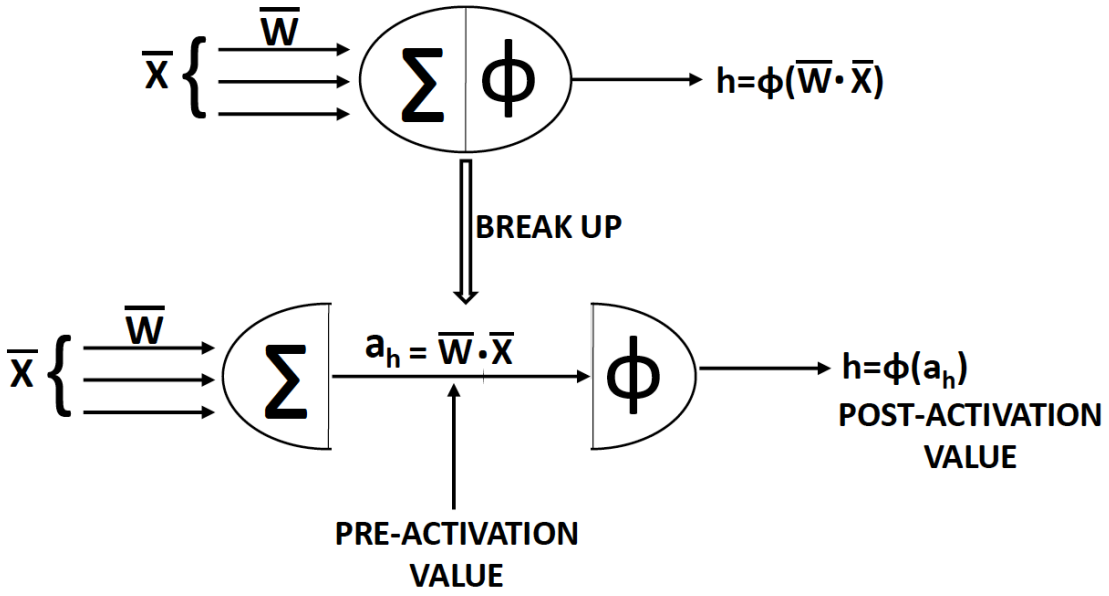
- A neural network with any number of layers but only linear activations can be shown to be equivalent to a single-layer network
  - True for any activation function in the output node

- Cannot solve XOR problem

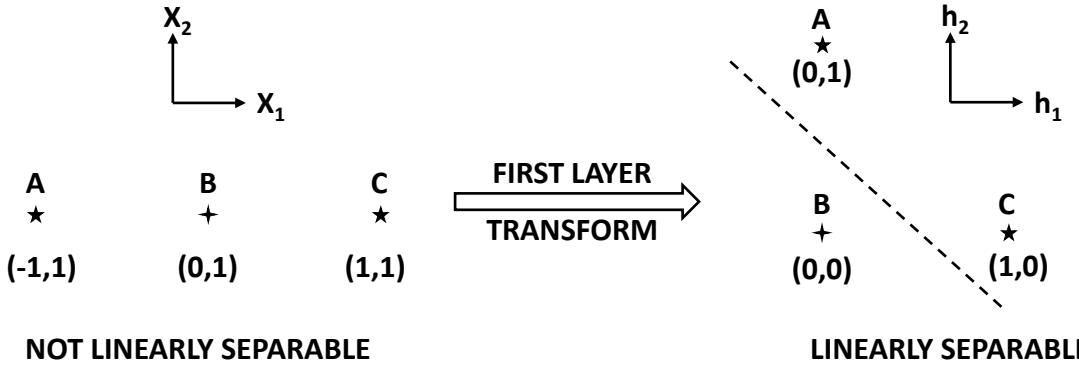
x_1	x_2	u
0	0	0
0	1	1
1	0	1
1	1	0

$$0w_1 + 0w_2 + b \leq 0 \iff b \leq 0$$
$$0w_1 + 1w_2 + b > 0 \iff b > -w_2$$
$$1w_1 + 0w_2 + b > 0 \iff b > -w_1$$
$$1w_1 + 1w_2 + b \leq 0 \iff b \leq -w_1 - w_2$$

Bringing Non-linearity with Activation Functions



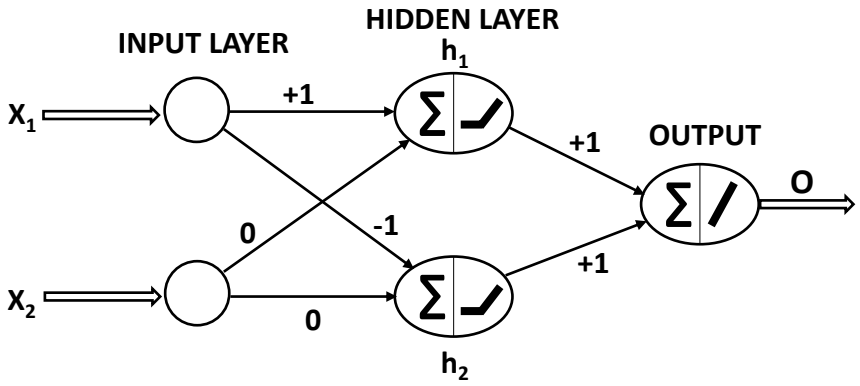
Non-linear activations in hidden layers



NOT LINEARLY SEPARABLE

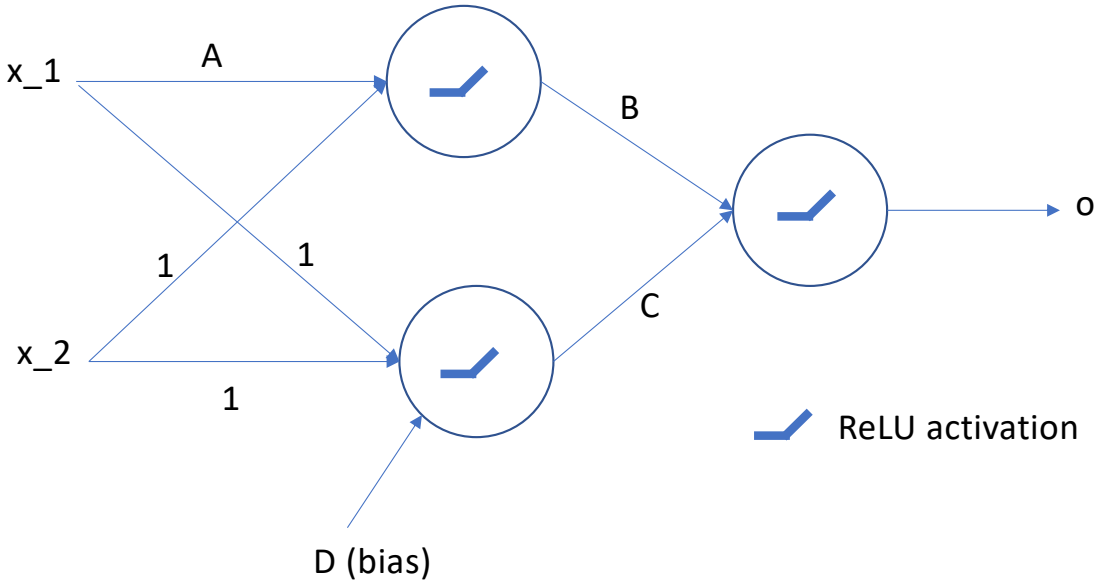
LINEARLY SEPARABLE

$$h_1 = \max\{x_1, 0\} \quad h_2 = \max\{-x_1, 0\}$$



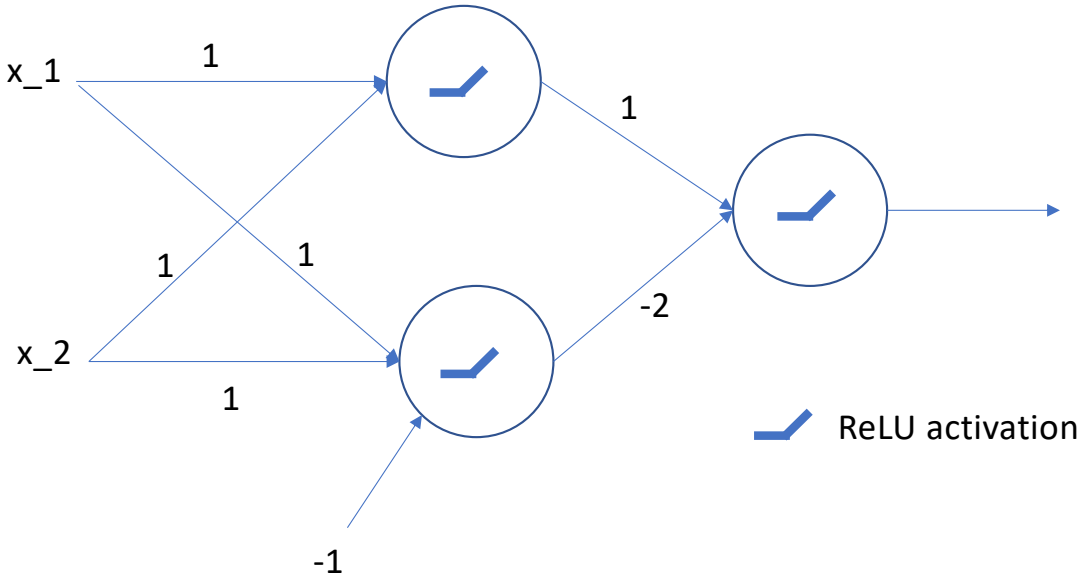
X1	X2	h1	h2	h1+h2
-1	1	0	1	1
0	1	0	0	0
1	1	1	0	1

What should be the values of A, B, C for XOR implementation ?

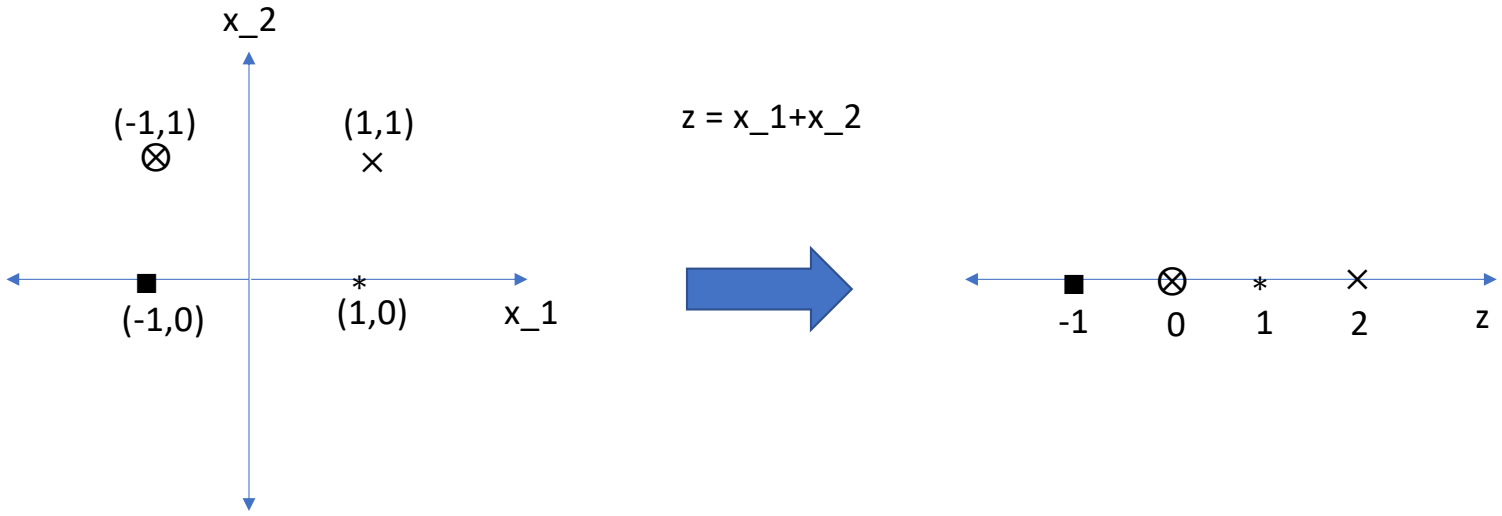


x_1	x_2	o
0	0	0
1	0	1
0	1	1
1	1	0

XOR implementation

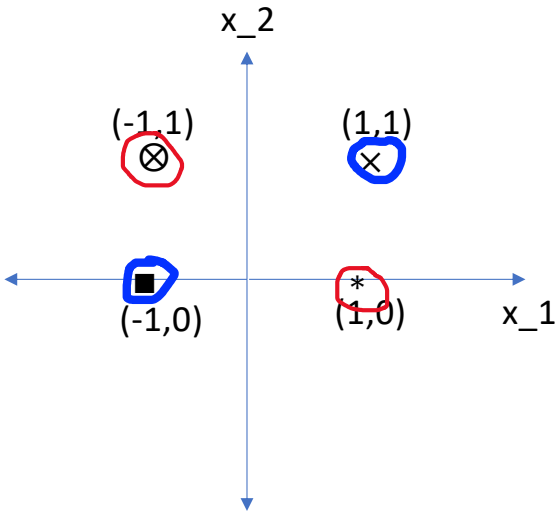


2D to 1D transformation example



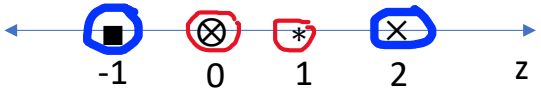
2D to 1D transformation example

Points in 2-D are not linearly separable



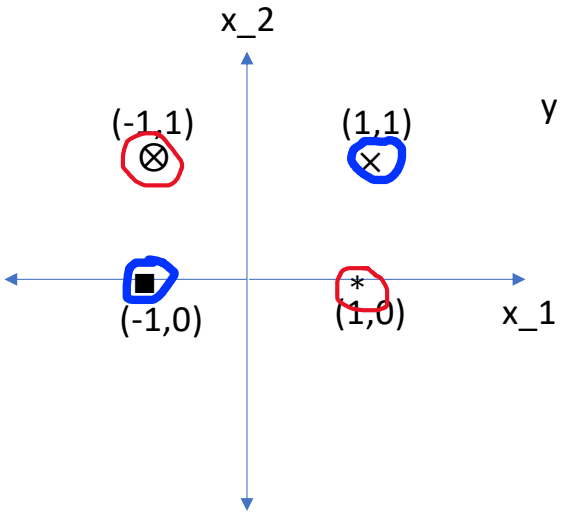
z = x_1+x_2

Points in 1-D are not linearly separable



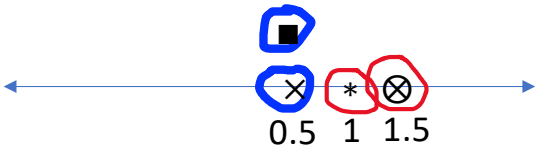
2D to 1D transformation example

Points in 2-D are not linearly separable

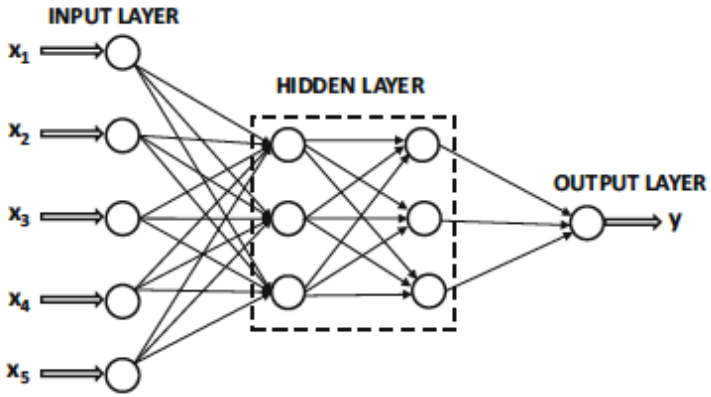


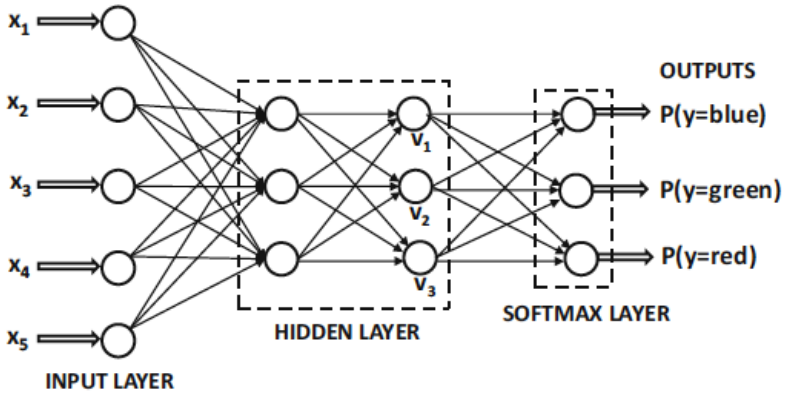
y = max(x_1,x_2)-0.5*min(x_1,x_2)

Points in 1-D are linearly separable

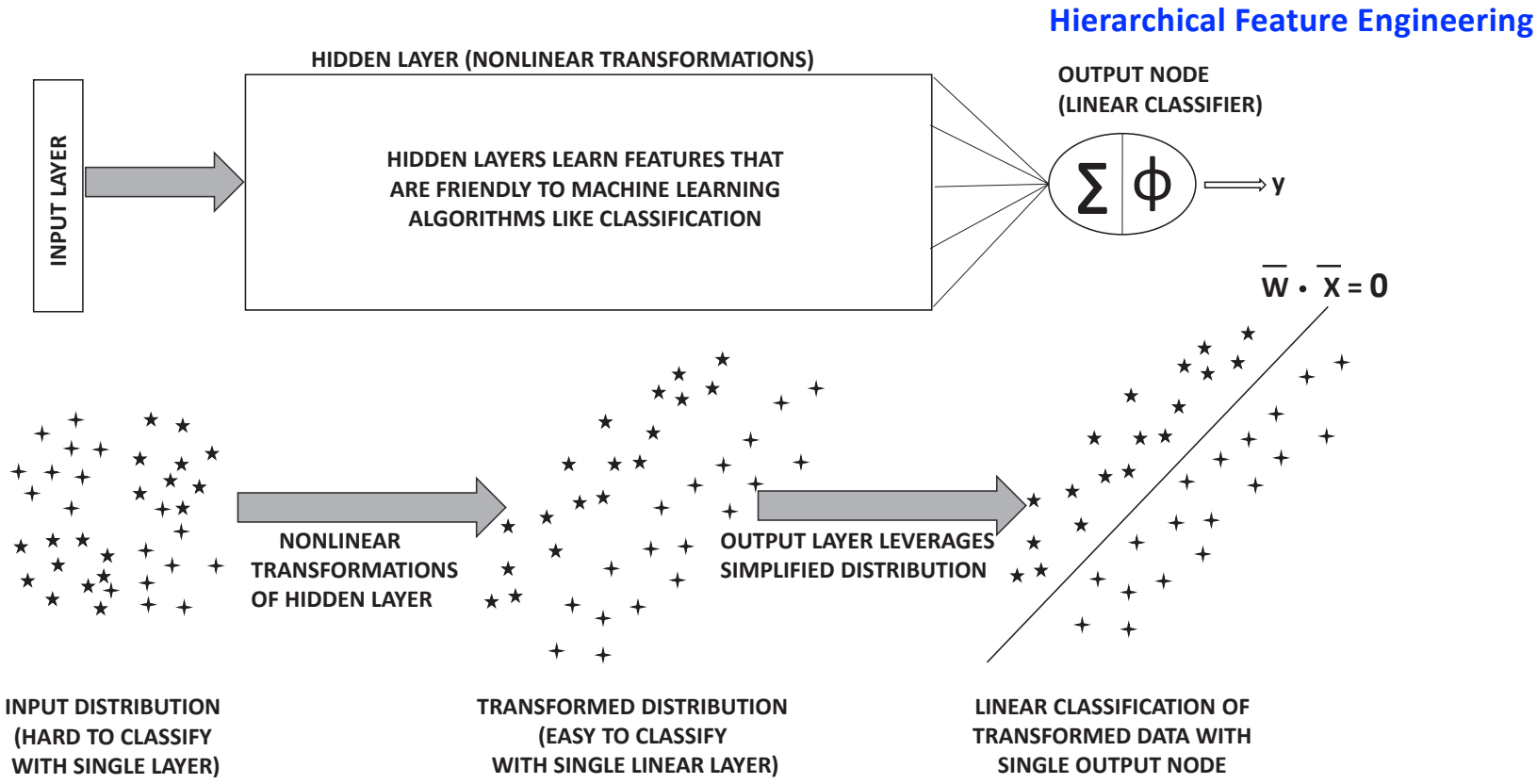


Multilayer Neural Networks





Hidden Layers Role



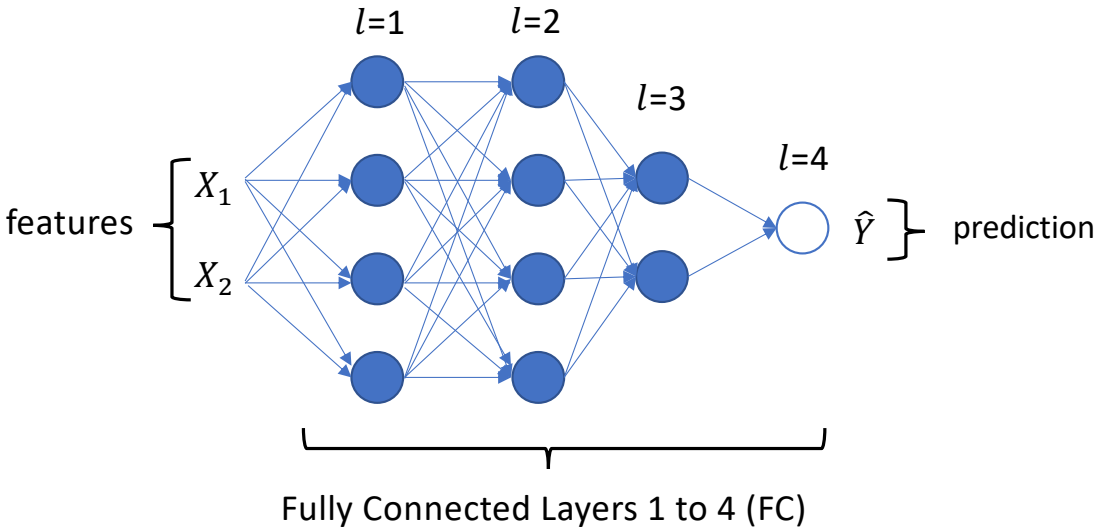
C. Aggarwal. Neural Networks and Deep Learning

Neural Networks Lifecycle

1. **Definition phase:**
   Define Number, Structure and Type of Layers
   Define other algorithmic and structural parameters (Hyperparameters)
2. **Training phase:** discover neuron's weights and biases
3. **Inference phase:** use model to make predictions (or classify)

Neural Network Definition

- Feed-Forward and Fully Connected NN:



- Each output of the layer l is connected to all inputs of layer $l+1$

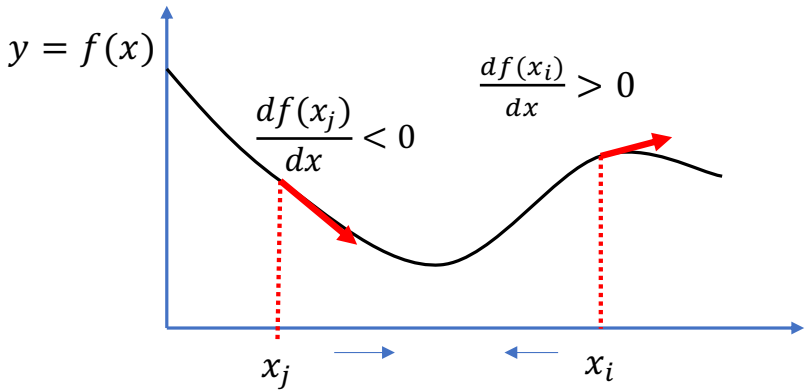
- Two types of nodes:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = \phi(Z^{[l]})$$

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$
$$\hat{Y} = sigmoid(Z^{[l]})$$

Training - Gradient Descent

- Find x_k such that $y_k = f(x_k)$ is a (local) minima
  - f is defined and differentiable
- Gradient Descent:
  - Start with random x_o
  - Repeat:
    - x_{n+1} <- $x_n - \alpha \frac{df(x_n)}{dx}$
    - Until you don't see any improvement



- α is the **Learning Rate:** determines how fast we descend the curve
  - α is usually very small: 0.01 or less

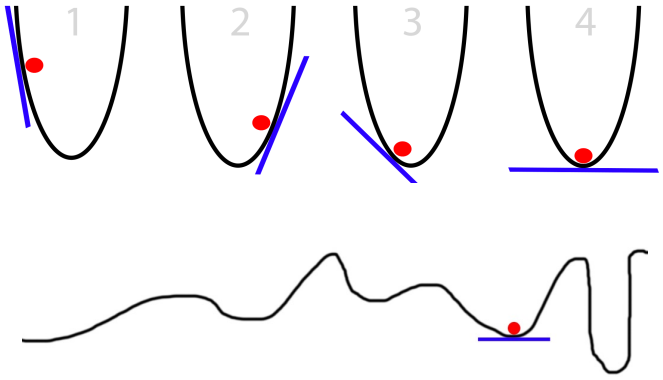
Training - Gradient Descent

- Used to minimize the **Cost Function** (loss/error across all samples)
- Gradient descent is that it will more often than not get stuck into the first local minimum that it encounters
  - There is no guarantee that this local minimum it finds is the best (global) one (bottom figure)

Algorithm 1 Gradient Descent

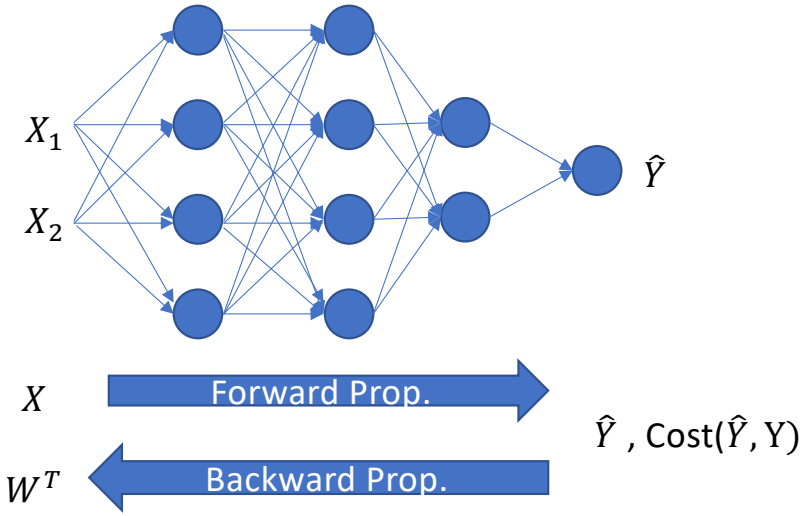
Input: Differentiable function $\mathbf{f(x)}$ where $\mathbf{f(x)} : \mathbf{R^n} \rightarrow \mathbf{R}$
Start point $\mathbf{x_{old}}$
Output: The local minima $\mathbf{x}^*$ that minimize $\mathbf{f(x)}$

```
while TRUE do
    tmpDelta ← x_old − α.(∇f(x_old))
    if abs(tmpDelta − x_old) < CRITERIA then
        break
    end if
    x_old ← tmpDelta
end while
```

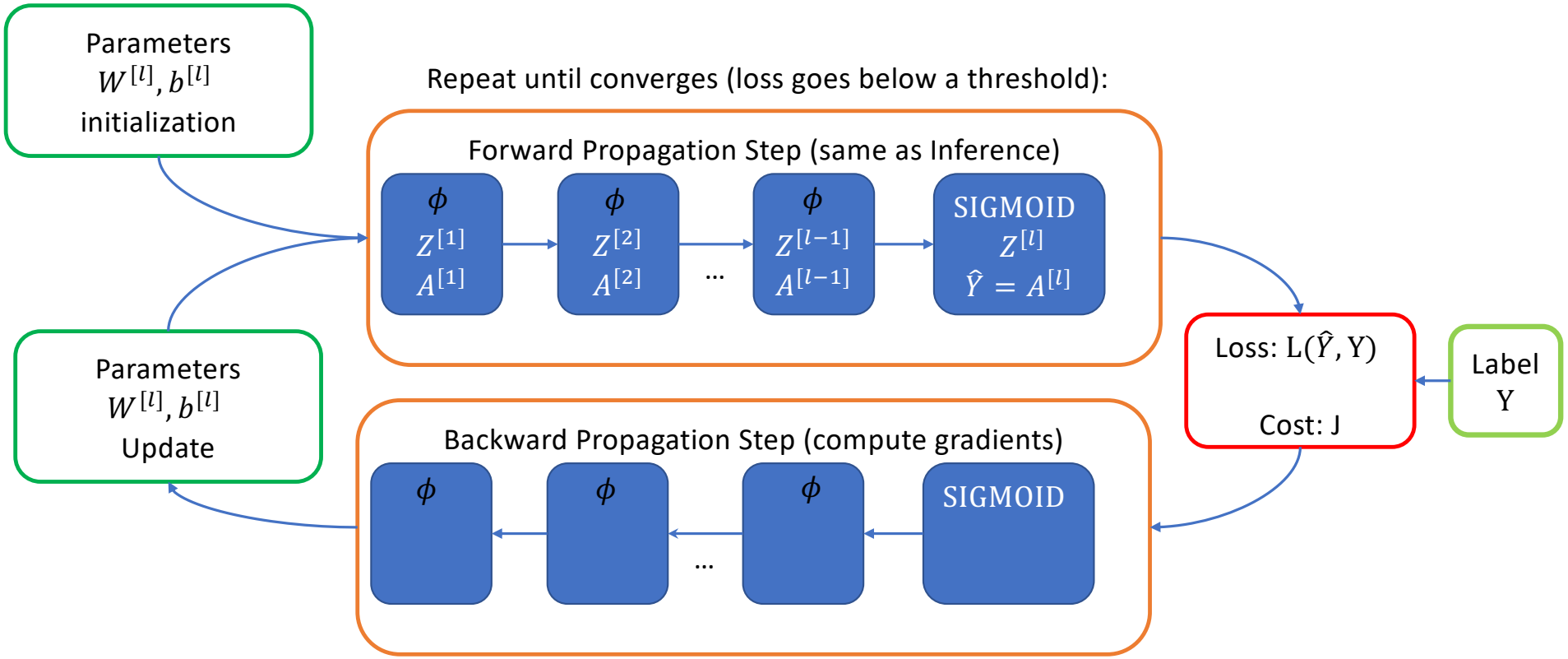


Training - Gradient Descent

- Prerequisite: Network structure is already defined (hyperparameters)
  - Type and number of layers (FC or other types)
  - Number of neurons on each layer
  - Activation functions of each layer
- **Batch Gradient Descent: use all samples**
- While (COST < threshold)
  - **Forward Propagation**:
    - compute prediction
    - Same formulas as Inference
  - **Backward Propagation**
    - compute gradient
    - adjust weights

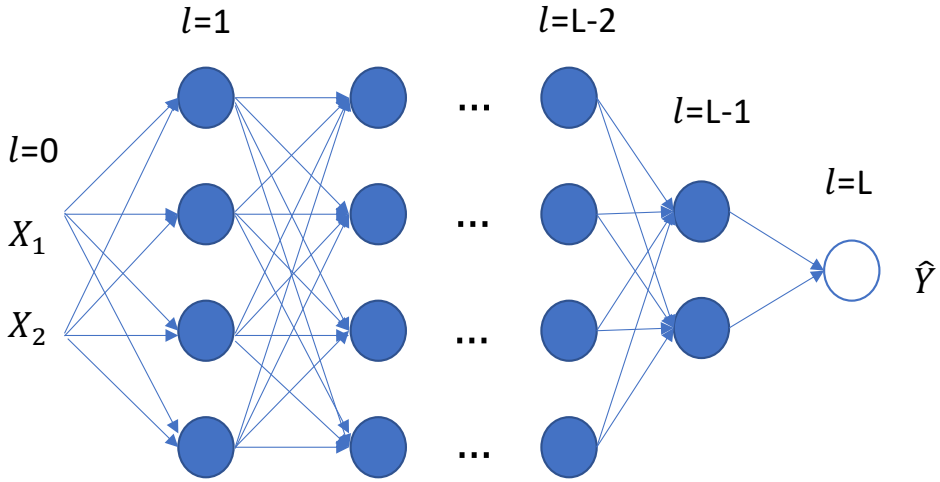


Training - Gradient Descent Algorithm



Forward Propagation (and Inference as well)

- m samples
- L layers
- $n^{[l]}$ is the number of neurons for layer l (where $l \in [0, L]$)
- $n^{[0]}$ is the number of features of each sample (layer 0)
- For each layer l compute:
  - $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$
  - $A^{[l]} = g^{[l]}(Z^{[l]})$
- Where
  - $A^{[0]} = X$
  - $g^{[L]}$ is a Sigmoid function
  - $g^{[l]}$ for $l < L$ is a ϕ function
- Matrices shapes
  - $W^{[l]}$: $(n^{[l]}, n^{[l-1]})$
  - $b^{[l]}$: $(n^{[l]}, 1)$
  - $A^{[l]}$: $(n^{[l]}, m)$
  - $Z^{[l]}$: $(n^{[l]}, m)$



Forward Propagation – Cost Function

- Cost function (cross-entropy):

$$\mathrm{J} = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)$$

  - $y^{(i)}$ is the target (label) value from the dataset of the *i*-th sample
  - $a^{[L](i)}$ is the output of the forward propagation (prediction) of the *i*-th sample
  - m is the number of samples used (dataset size)
  - **Cost function: a function of individual samples loss functions (J)**

Backward Propagation – Parameter Updates

- For each layer, we want to update the parameters with the gradients

$$W^{[l]} \Leftarrow W^{[l]} - \alpha \frac{dJ}{dW^{[l]}}$$

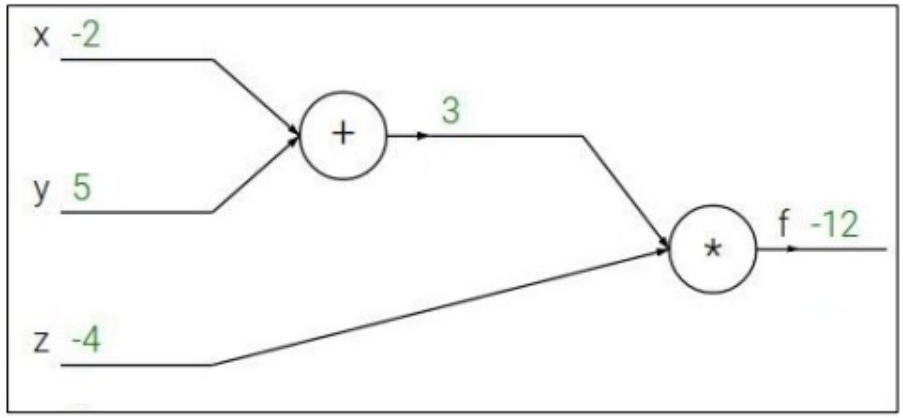
$$b^{[l]} \Leftarrow b^{[l]} - \alpha \frac{dJ}{db^{[l]}}$$

- How do we compute $\frac{dJ}{dW^{[l]}}$ and $\frac{dJ}{db^{[l]}}$?
- Go backward from the cost function J: backward propagation

Backward Propagation Example

- Initial Function $\mathrm{f(x, y, z)} = (x + y) * z$
- Computation Graph Functions:
  - $q(x, y) = x + y$
  - $f(q, z) = qz$
- Inputs: $\mathrm{x} = \text{-}2, \mathrm{y} = 5,\ \mathrm{z} = \text{-}4$
- Want to obtain:
  - $\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}$

Computation Graph for (x+y)* z

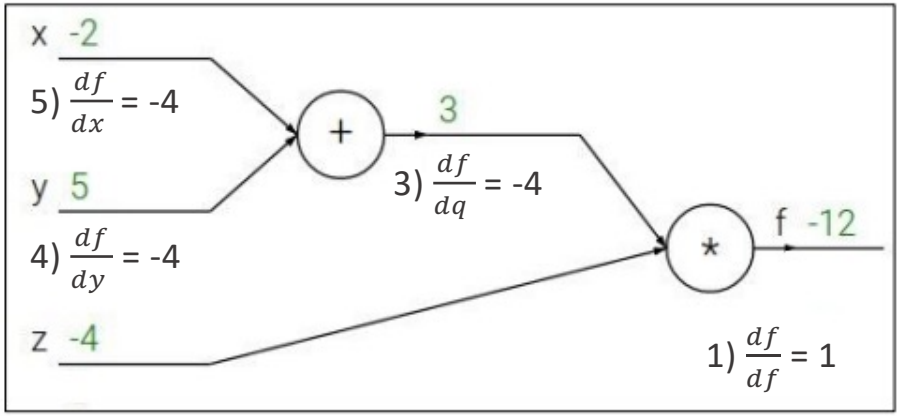


From http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf

Backward Propagation Example

- Computation Graph Functions:
  - $q(x, y) = x + y$
  - $f(q, z) = qz$
- Basic gradients :
  - $\frac{dq(x,y)}{dx} = 1, \ \frac{dq(x,y)}{dy} = 1$
  - $\frac{df(q,z)}{dq} = z, \ \frac{df(q,z)}{dz} = q$
- Compute gradients with chain rule:
  - $\frac{df}{dz} = q = 3$
  - $\frac{df}{dx} = \frac{df}{dq} * \frac{dq}{dx} = z * 1 = -4$
  - $\frac{df}{dy} = \frac{df}{dq} * \frac{dq}{dy} = z * 1 = -4$

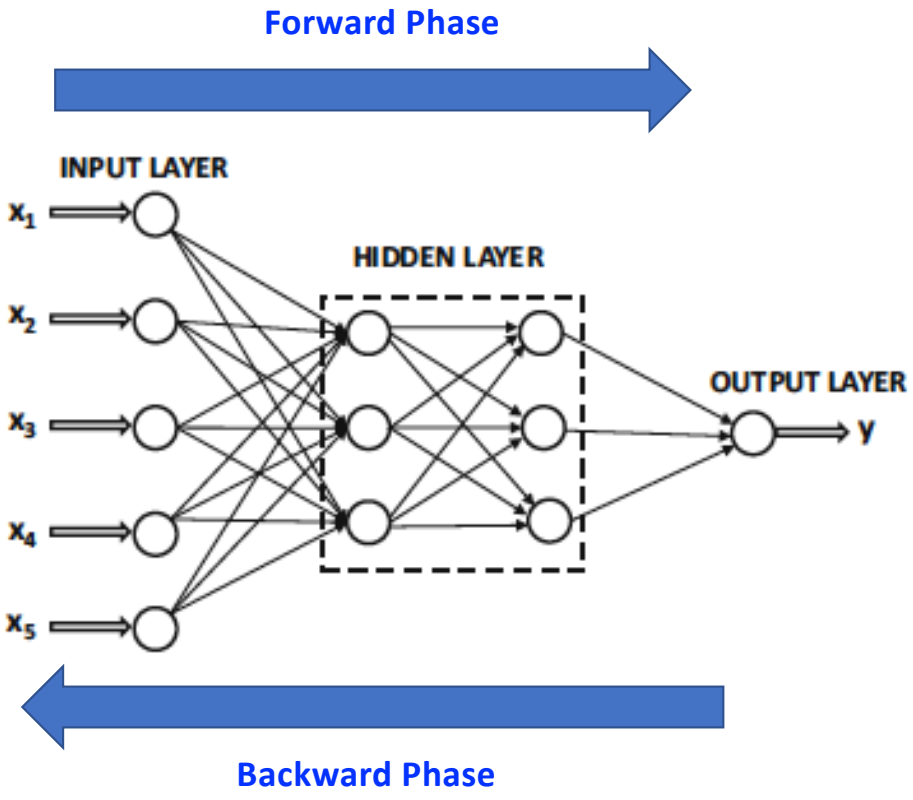
Computation Graph for (x+y)* z



From http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf

Deep Learning Training

- Forward phase
- Loss calculation
- Backward phase
- Weight update



Deep Learning Training Steps

- Forward phase:
  - compute the activations of the hidden units based on the current value of weights
  - calculate output
  - calculate loss function
- Backward phase:
  - compute partial derivative of loss function w.r.t. all the weights;
  - use *backpropagation algorithm* to calculate the partial derivatives recursively
  - backpropagation changes the weights (and biases) in a network to decrease the loss
- Update the weights using gradient descent

Loss Functions

- Form of loss functions depends on the type of output (continuous valued or discrete) and on the range of output values
- Least-squares regression for continuous valued targets
  - Least-square regression loss
    - Linear activation in output

$$Loss = \ (y - \hat{y})^2$$

- Probabilistic class prediction for discrete valued targets
  - Logistic regression loss
    - Sigmoid activation in output
    - If y is binary valued in {-1,1} and $\hat{y} \in (0,1)$

$$Loss = -\log|\frac{y}{2} - 0.5 + \hat{y}|$$

    - If y is binary valued in {0,1} and $\hat{y} \in (0,1)$

$$Loss = -\mathrm{y}\log\hat{y} - (1-y)\log(1-\hat{y})$$

Multi-class classification: Activation and Loss

- Softmax activtation: function is calculated with respect to multiple inputs
- Converts real valued predictions into output probabilities

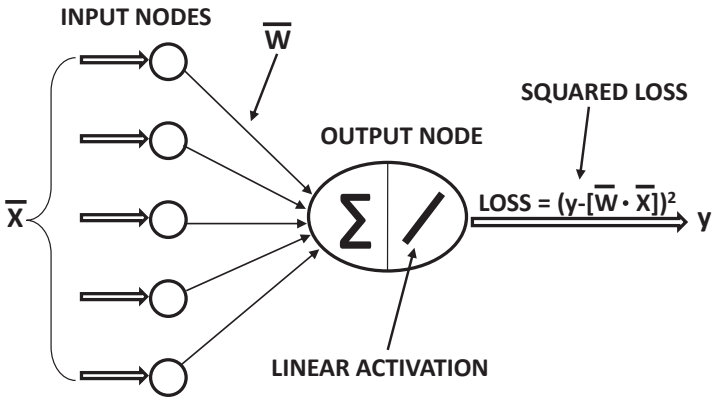
$$o_i = \frac{\exp(v_i)}{\sum_{j=1}^{k} \exp(v_j)} \quad \forall i \in \{1, \ldots, k\}$$

- Cross-entropy loss function

$$L = -\sum_{i=1}^{k} y_i \log(o_i)$$

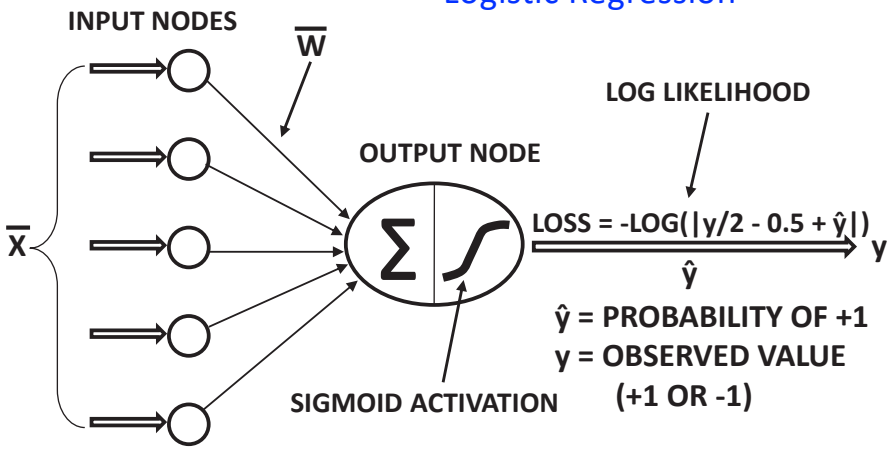
Neural Networks for Machine Learning Models

Linear Regression



$$\bar{W} \Leftarrow \bar{W} + \alpha(y - \hat{y})\bar{X}$$

Logistic Regression



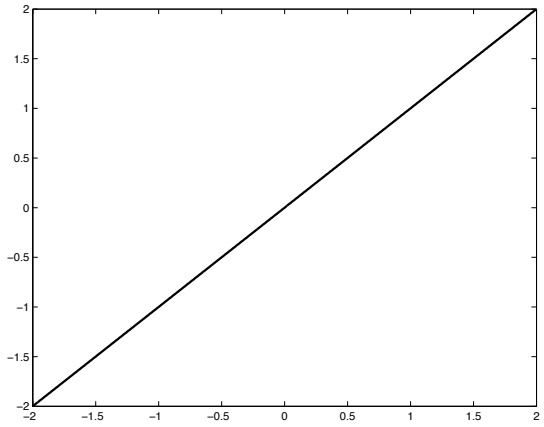
$$\overline{W} \Leftarrow \overline{W} + \alpha \frac{y_i \overline{X_i}}{1 + \exp[y_i(\overline{W} \cdot \overline{X_i})]}$$

Hyperparameters in Deep Learning

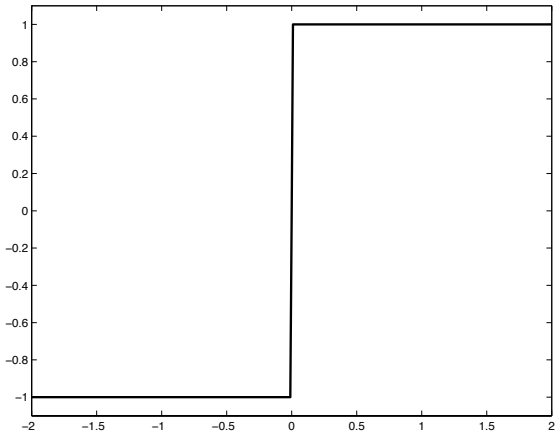
- Network architecture: number of hidden layers, number of hidden units per layer
- Activation functions
- Weight initializer
- Optimizer
- Learning rate
- Batch size
- Momentum

Activation functions

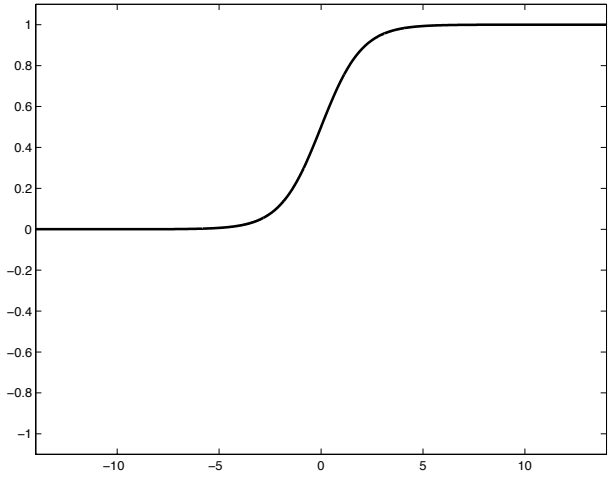
Linear $\phi(z) = z$



Sign $\phi(z) = sign(z)$

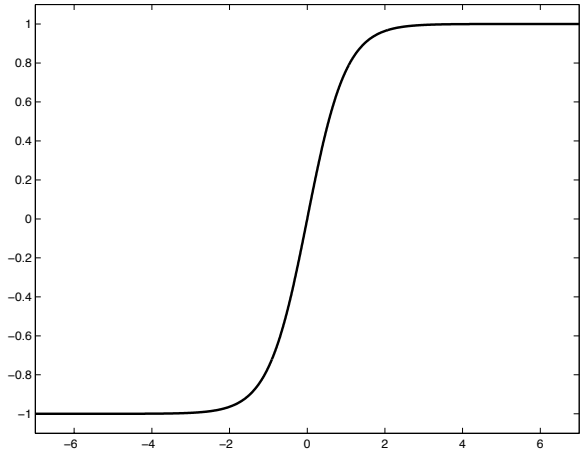


Sigmoid $\phi(z) = \frac{1}{1+e^{-z}}$



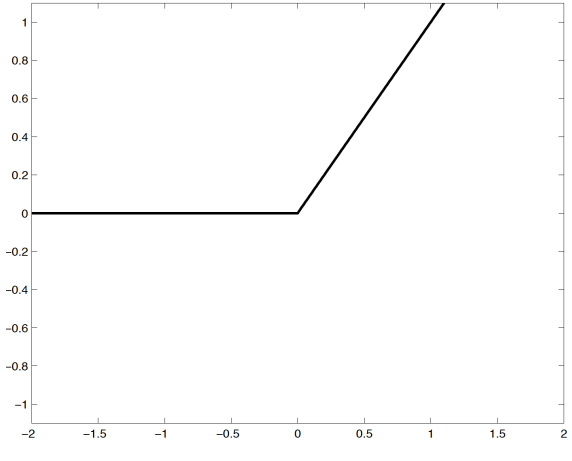
Activation functions

Tanh $\phi(z) = \frac{e^{2z}-1}{e^{2z}+1}$



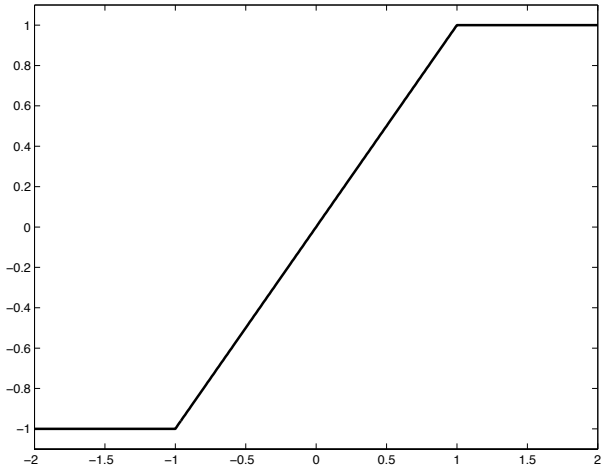
ReLU (Rectified Linear Unit)

$\phi(z) = \max\{z, 0\}$



Hard Tanh

$\phi(z) = \max\{\min[z, 1], -1\}$



- Also called *squashing* functions
- tanh(z) = 2.sigmoid(2z)-1
- ReLU is most common for hidden layers;

Activation Functions

- An activation function $\Phi(v)$ in the output layer can control the nature of the output (e.g., probability value in [0, 1])
- In multilayer neural networks, activation functions bring nonlinearity into hidden layers, which increases the complexity and representation power of the model
- Continuous, differentiable activation functions for gradient descent updates (need derivative of activation functions in weight updates during training)

Softmax Activation Function

- Function is calculated with respect to multiple inputs
- Converts real valued predictions into output probabilities

$$o_i = \frac{\exp(v_i)}{\sum_{j=1}^{k} \exp(v_j)} \quad \forall i \in \{1, \ldots, k\}$$

- Backpropagation with softmax
  - Always used in output layer, not in hidden layers
  - Always paired with cross-entropy loss

$$L = -\sum_{i=1}^{k} y_i \log(o_i) \qquad \frac{\partial L}{\partial v_i} = \sum_{j=1}^{k} \frac{\partial L}{\partial o_j} \cdot \frac{\partial o_j}{\partial v_i} = o_i - y_i$$

  - Derivatives needed for backpropagation have simple form

Universal Approximators Theorem

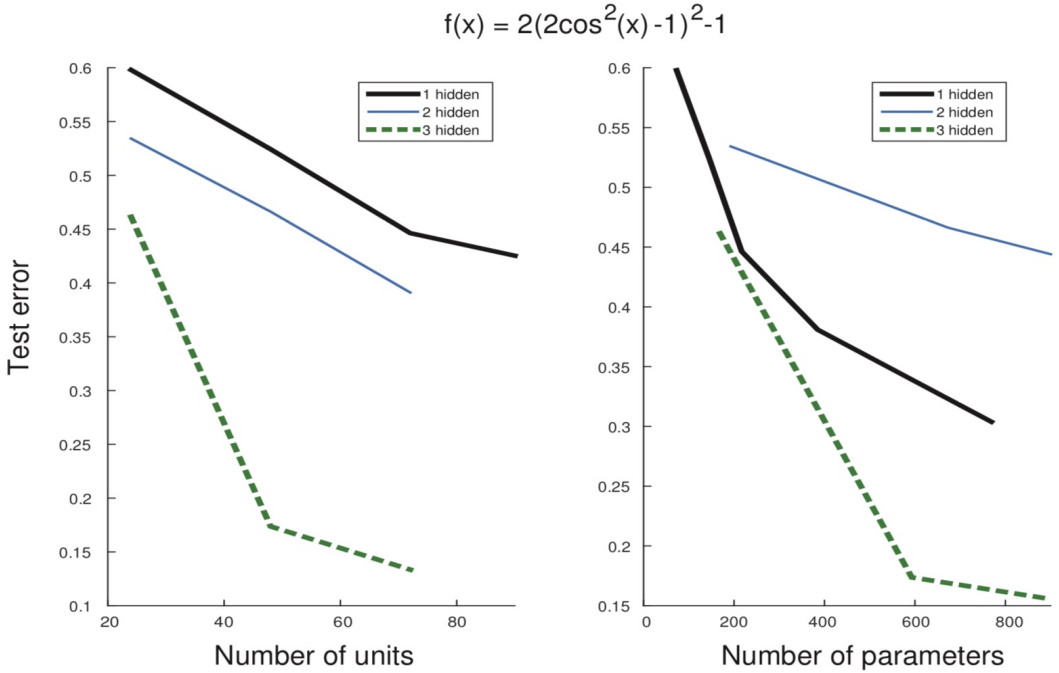
- Multilayer layer feedforward network, with as little as two layers (input and a hidden later), with arbitrary activation functions, and **sufficiently large hidden units** can approximate any continuous function with an arbitrary small error

- Why need Deep neural networks (more than one hidden layer) ?

- Going deep requires fewer units per layer; reduces the capacity of the network

https://mcneela.github.io/machine_learning/2017/03/21/Universal-Approximation-Theorem.html

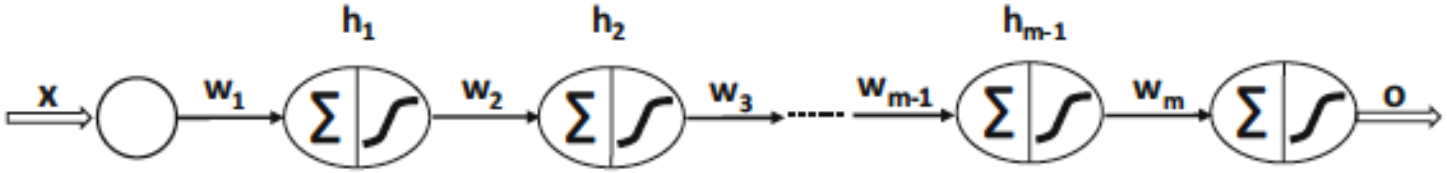
Neural Network Size: Depth vs Width



Deep Networks require exponentially lower number of parameters than shallow networks for approximating the same function

Easier to train with less data

Vanishing and Exploding Gradients



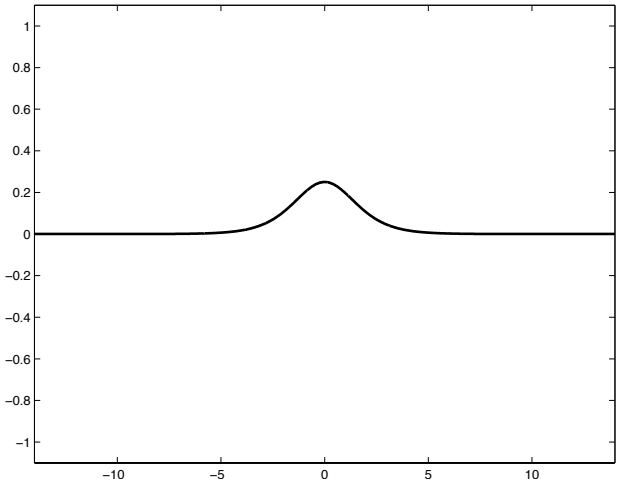
$$\frac{\partial L}{\partial h_t} = \phi'(w_{t+1}h_t).w_{t+1}.\frac{\partial L}{\partial h_{t+1}}$$

- For sigmoid activation, $\phi'(z) = \phi(z)(1-\phi(z))$, has maximum value of 0.25 at ϕ(z)=0.5

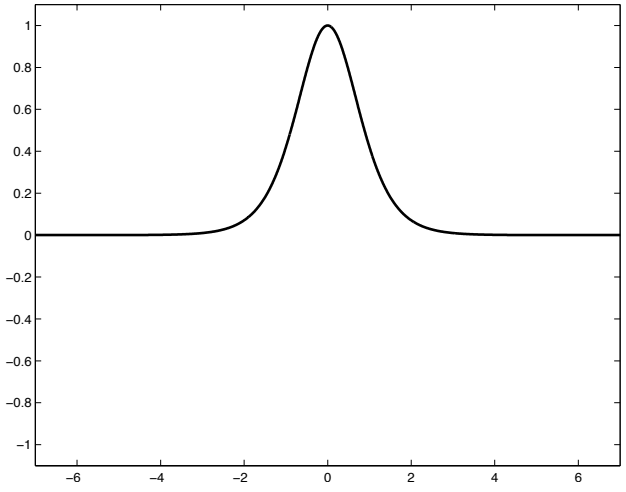
- Each $\frac{\partial L}{\partial h_t}$ will will be less than 0.25 of $\frac{\partial L}{\partial h_{t+1}}$
- As we (back) propagate further gradient keep decreasing further; After r layers the value of gradient reduces to 0.25^r (= 10^{-6} for r=10) of the original value causing the update magnitudes of earlier layers to be very small compared to later layers => vanishing gradient problem.
- If we use activation with larger gradient and larger weights=> gradient may become very large during backpropagation (exploding gradients)
- Improper initialization of weights also causes vanishing (too small weights) or exploding (too large weights) gradients

Activation Functions Derivatives

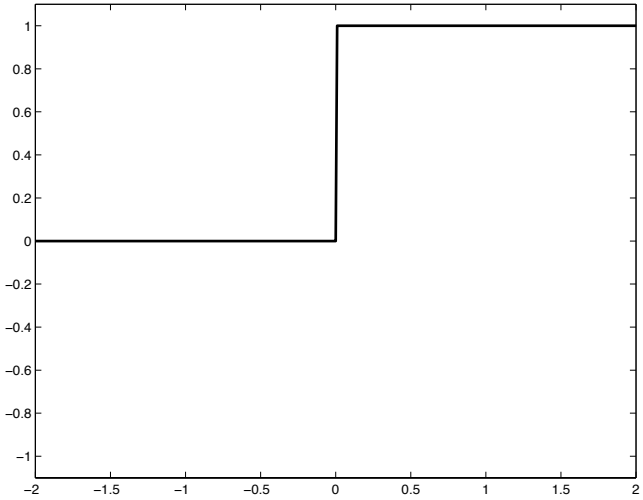
sigmoid



tanh



ReLU



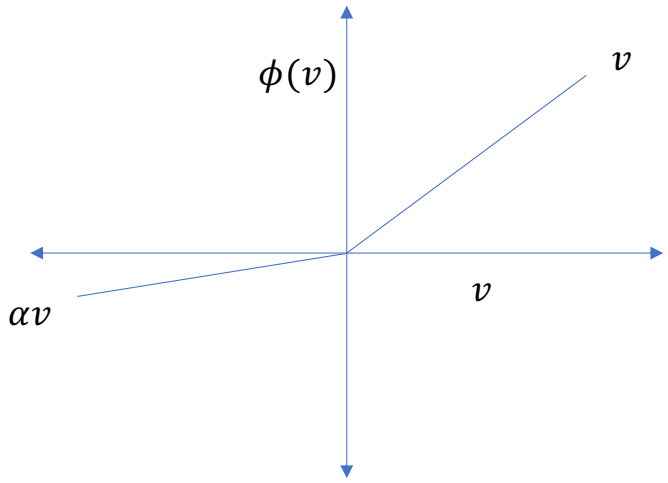
- Sigmoid and tanh derivatives vs ReLU
- Sigmoid and tanh gradients saturate at large values of argument; very susceptible to vanishing gradient problem
- ReLU is faster to train; most commonly used activation function in deep learning

Preventing vanishing gradients

- Use piece-wise linear functions like ReLU as activation. Gradients are not close to 0 for higher values of input.
- Piece-wise linear can cause dead neuron
  - Causes: improper weight initialization, high learning rates
  - Hidden unit will not fire for any input
  - Weights of the neuron will not be updated
- Leakv ReLU activation

$$\Phi(v) = \begin{cases} \alpha \cdot v & v \leq 0 \\ v & \text{otherwise} \end{cases}$$

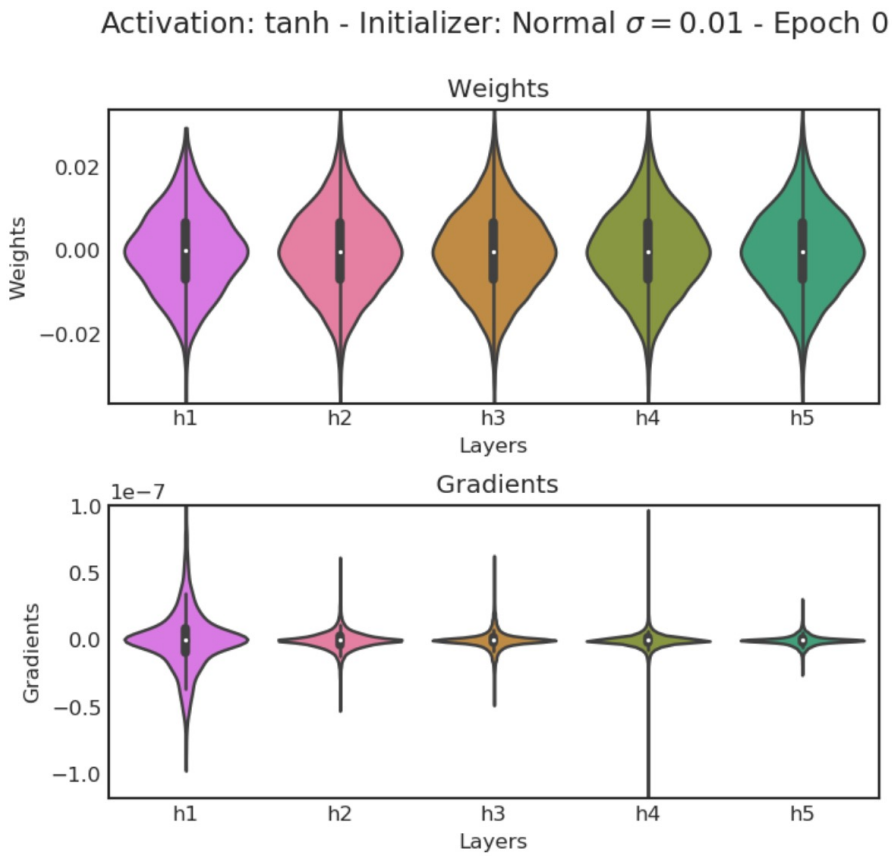
$$\alpha \in (0, 1)$$

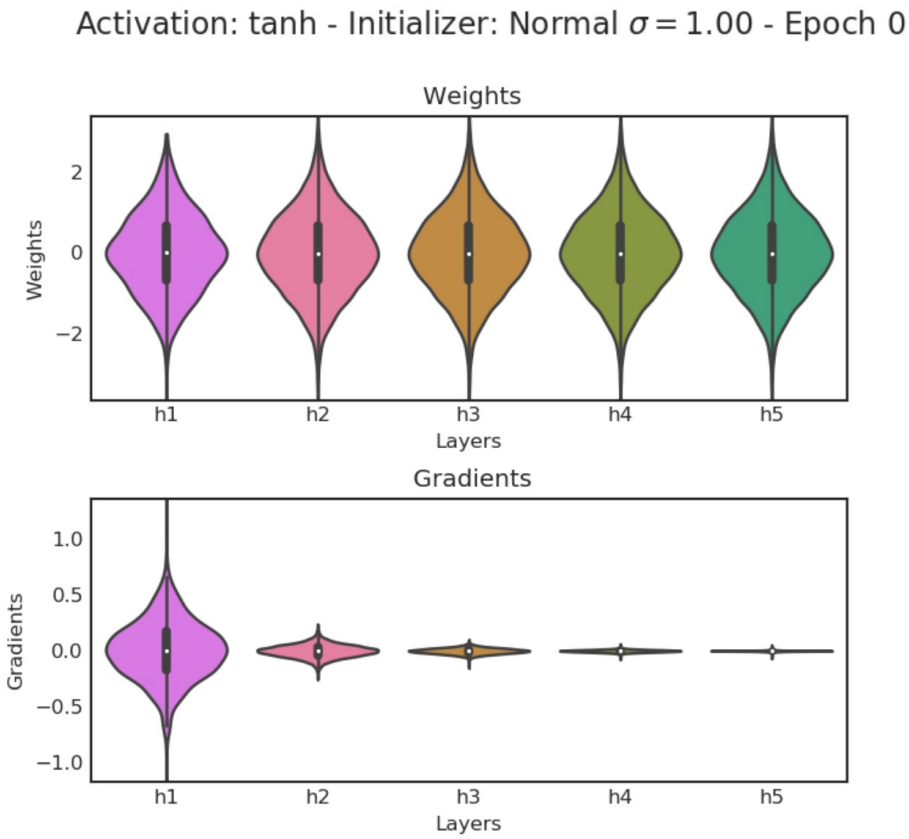


Weight Initialization

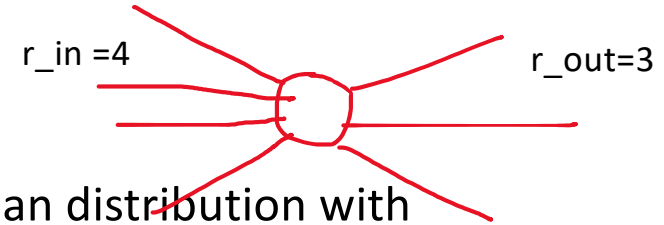
- Initializing all weights to same value will cause neurons to evolve symmetrically
- Generally, biases are initialized with 0 values and weights with random numbers; Initializing weights to random values breaks symmetry and enables different neurons to learn different features
- Initial value of weights is important.
  - Poor initializations can lead to bad convergence or no learning.
  - Instability across different layers (vanishing and exploding gradients).

Vanishing and Exploding Gradients





Popular Weight Initializers



- Xavier initialization
  - Each neuron weight is sampled from 0 mean Gaussian distribution with standard deviation $\sqrt{2/(r_{in} + r_{out})}$

    when r_{in} and r_{out} are number of input and output weights for the neuron
  - Used with Sigmoid or Tanh activations
  - Also referred to as Glorot initialization (like in *Keras*)
- He initialization
  - Sample weights from 0 mean Gaussian distribution with standard deviation

    $\sqrt{2/r}$

    r can be r_{in} or r_{out}
  - Used with ReLU activation

https://www.deeplearning.ai/ai-notes/initialization/

Normalizing Input Data

- Min-max normalization (for feature j of input datapoint i)

$$x_{ij} \Leftarrow \frac{x_{ij} - min_j}{max_j - min_j}$$

  - Data with smaller standard deviation; scaled to be in the range [0,1]
  - Lessen the effect of outliers
- Standardization

$$x_{ij} \Leftarrow \frac{x_{ij} - mean_j}{std\_dev_j}$$

- Normalization helps in the convergence of optimization algorithm
- Should apply same normalization parameters to both train and test set
- Normalization parameters are calculated using train data
- Training converges faster when the inputs are normalized

Dataset Augmentation

- Artificially enlarge the training set by adding transformations/perturbations of the training data
- Domain-specific transformations
- Provides more training data
- Helps in model generalization and prevents overfitting
- Augmentation techniques (images):
  - Horizontal and Vertical shift
  - Horizontal and Vertical flip
  - Rotation
  - Brightness
  - Zooming
  - Noising
- Only applied to training set, not to validation and test set

Dataset Augmentation Using Keras

- ImageDataGenerator class
- Keras examples: https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/

L_2 and L_1 Regularization in Neural Network

L_2 Regularization Loss

$$L = \sum_{(x,y)\in\mathcal{D}}(y-\hat{y})^2 + \underbrace{\lambda \cdot \sum_{i=0}^{d} w_i^2}_{L_2-\text{Regularization}}$$

$$w_i \Leftarrow w_i(1-2\alpha\lambda) - \alpha\frac{\partial L}{\partial w_i}$$ Weight decay

L_1 Regularization Loss

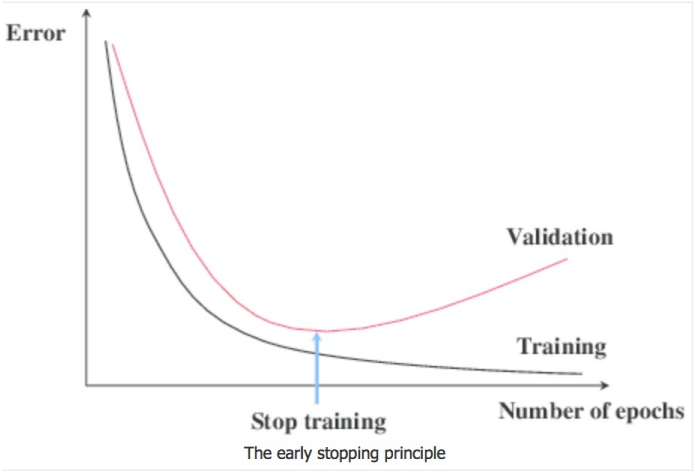
$$L = \sum_{(x,y)\in\mathcal{D}}(y-\hat{y})^2 + \lambda \cdot \sum_{i=0}^{d} |w_i|_1$$

$$w_i \Leftarrow w_i - \alpha\lambda s_i - \alpha\frac{\partial L}{\partial w_i} \qquad s_i = \begin{cases} -1 & w_i < 0 \\ +1 & w_i > 0 \end{cases}$$

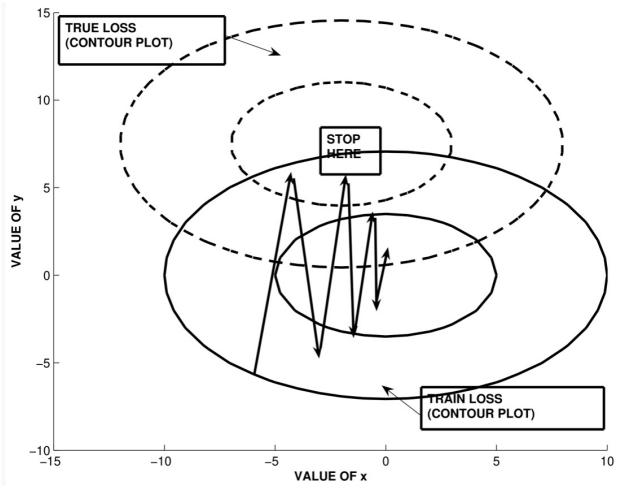
L_2 vs L_1 Regularization in Neural Network

- Value of lambda (hyperparameter) can be tuned using the validation set
- L_1 regularization leads to sparse weight matrices; Used to determine edges to prune
- Both L_2 and L_1 regularization move the weights progressively towards 0
- Multiplicative vs additive weight decay

Early Stopping

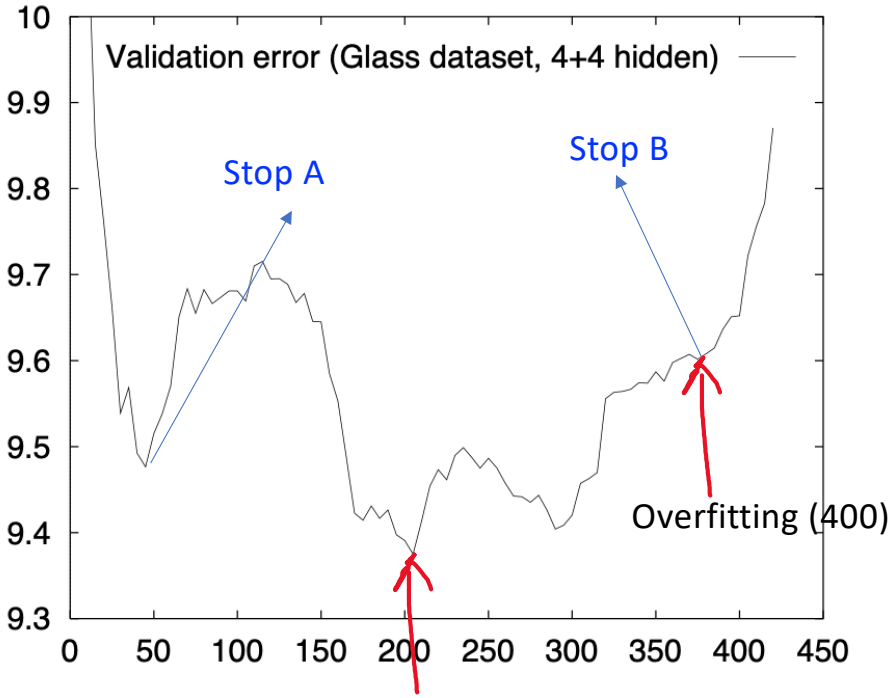


The early stopping principle



- Stop the training when validation error starts rising to prevent overfitting
- Early stopping is an implicit regularization technique
- Done in hindsight; define a performance criteria and checkpoint the latest “best” model
- May not help with large datasets with less likelihood of overfitting
- No principled approach to early stop. Can be tricky when validation error has multiple local minima
- $\boldsymbol{L_2}$ regularization can achieve similar or better performance than early stopping

Reality is Ugly



- 16 local minima before epoch 400
- 4 are global minimum up to where they occur
- Stop A: you get model with validation error > 9.5
- Stop B: you get model with validation error < 9.4
- Stop A to Stop B is ~7x increase in training time and
- 1.1% decrease in validation error

Examples Early Stopping Tests

- **Stopping criterion 1**: stop as soon as the generalization loss (GL) exceeds a certain threshold

$$GL(t) = 100 \cdot \left(\frac{E_{va}(t)}{E_{opt}(t)} - 1 \right)$$

Validation error till epoch t

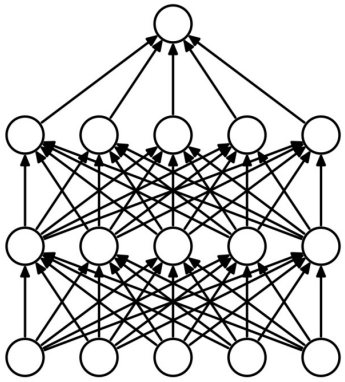
Minimum validation error till epoch t

stop after first epoch t with $GL(t) > \alpha$

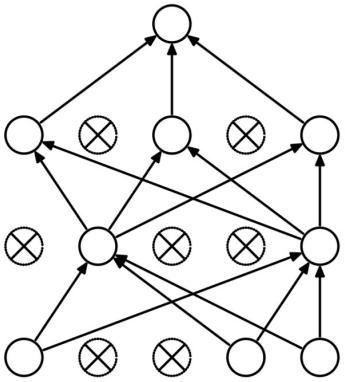
- **Stopping criterion 2**: suppress stopping if training is progressing rapidly
  - When training error decreases rapidly, generalization loss has higher chance of getting repaired
  - Overfitting usually begins when training error decreases slowly
  - Use the quotient of generalization loss and training progress
- **Stopping criterion 3**: stop when generalization error increased in s successive intervals

Dropout

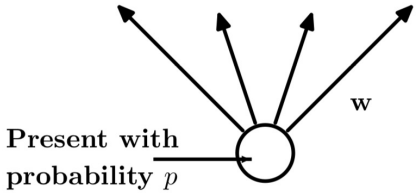
- Dropout is a regularization technique to deal with overfitting problem and improve generalization
- Prevents co-adaptation of activation units
- Probabilistically drop input features or activation units in hidden layers
- Layer dependent dropout probability (~0.2 for input, ~0.5 for hidden)



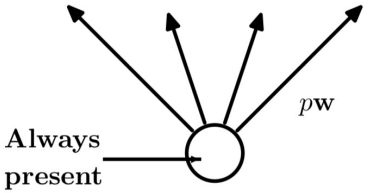
(a) Standard Neural Net



(b) After applying dropout.

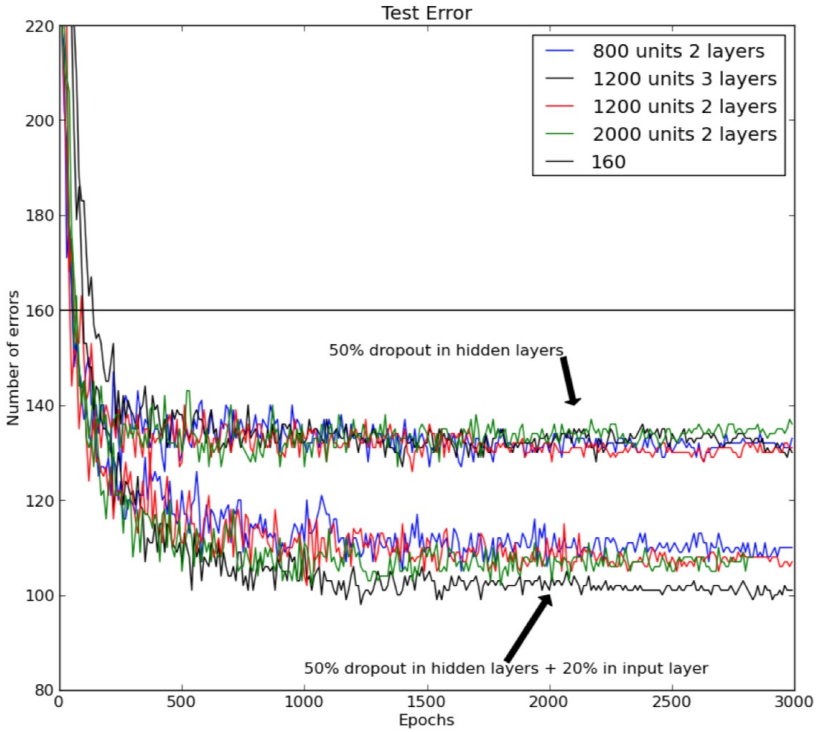


(a) At training time



(b) At test time

Dropout with MNIST Image Classification



MNIST dataset (images of handwritten digits)
60,000 train images
10,000 test images
160 errors (without dropout)
130 errors (with 50% dropout in hidden layers)
110 errors (with 50% dropout in hidden layers + 20% dropout in input layer)
Improvement seen across different networks with different number of layers and units

MNIST dataset available at http://yann.lecun.com/exdb/mnist/

Batch normalization

- Internal covariance shift – change in the distribution of network activations due to change in network parameters during training
- Idea is to reduce internal covariance shift by applying normalization to inputs of each layer
- Achieve fix distribution of inputs at each layer
- Normalization *for each training mini-batch*.
- Batch normalization enables training with larger learning rates
  - Reduces the dependence of gradients on the scale of the parameters
  - Faster convergence and better generalization

Batch normalization

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

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

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

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

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

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

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

Why this step ?

Gradient Descent

$$L = \sum_{i=1}^{n} L_i$$

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L}{\partial \overline{W}}$$

- Loss is calculated over all the training points at each weight update
- Memory requirements may be prohibitive

Stochastic Gradient Descent (SGD)

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L_i}{\partial \overline{W}}$$

- Loss is calculated using one training data at each weight update
- Stochastic gradient descent is only a randomized approximation of the true loss function.

Mini-batch Gradient Descent

$$\overline{W} \Leftarrow \overline{W} - \alpha \sum_{i \in B} \frac{\partial L_i}{\partial \overline{W}}$$

- A batch B of training points is used in a single update of weights
- Increases the memory requirements. Layer outputs are matrices instead of vectors. In backward phase, matrices of gradients are calculated.

Batch and Stochastic Gradient descent

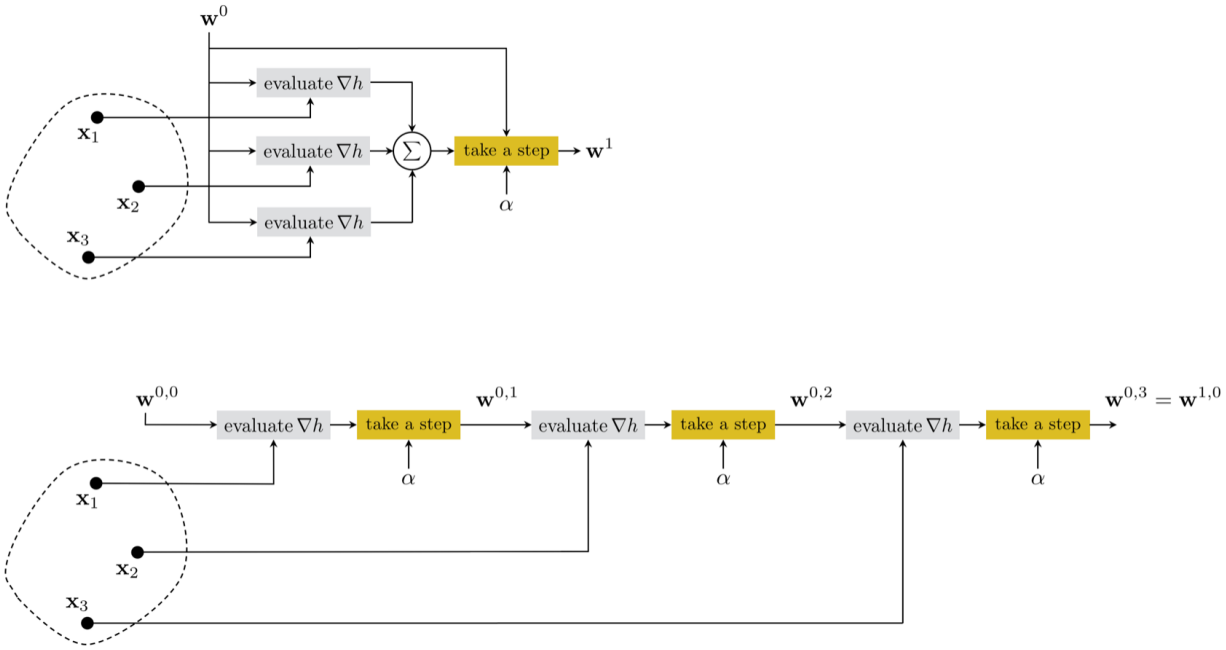
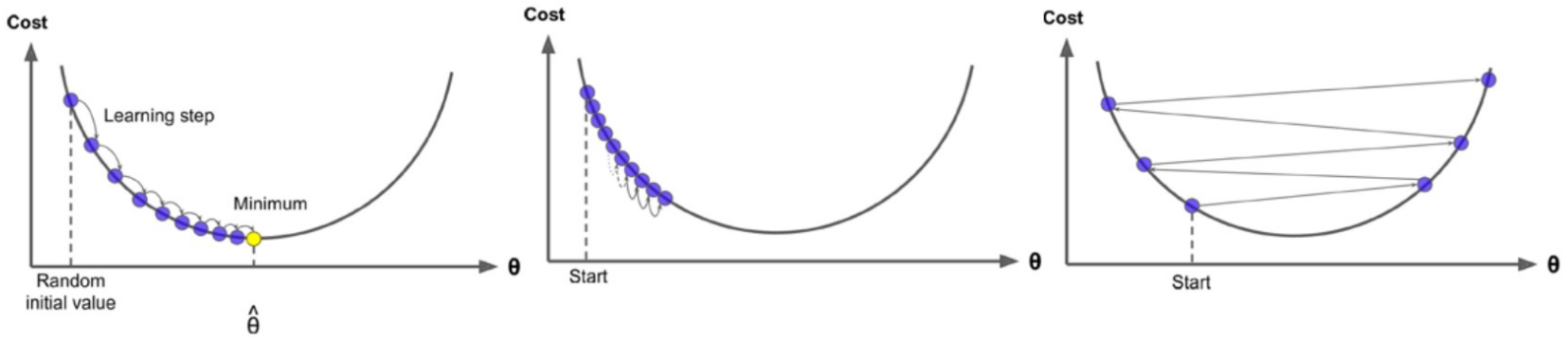


Figure 1: *Schematic comparison of first iteration of (top) batch and (bottom) stochastic gradient descent, using a dataset with three data points.*

https://kenndanielso.github.io/mlrefined/blog_posts/13_Multilayer_perceptrons/13_6_Stochastic_and_minibatch_gradient_descent.html

Learning rate: large value vs small value



Optimum learning rate : The model adjusts weights (θ) in subsequent training loops to arrive at cost minima.

Slow learning rate : Converges to cost minima but very slowly.

*Fast learning rate : **may** not converge to cost minima and the cost might keep increasing with further training loops.*

Image Credit : "Hands-on Machine Learning with Scikit-Learn and TensorFlow " by Aurelien Geron

Constant Learning Rate

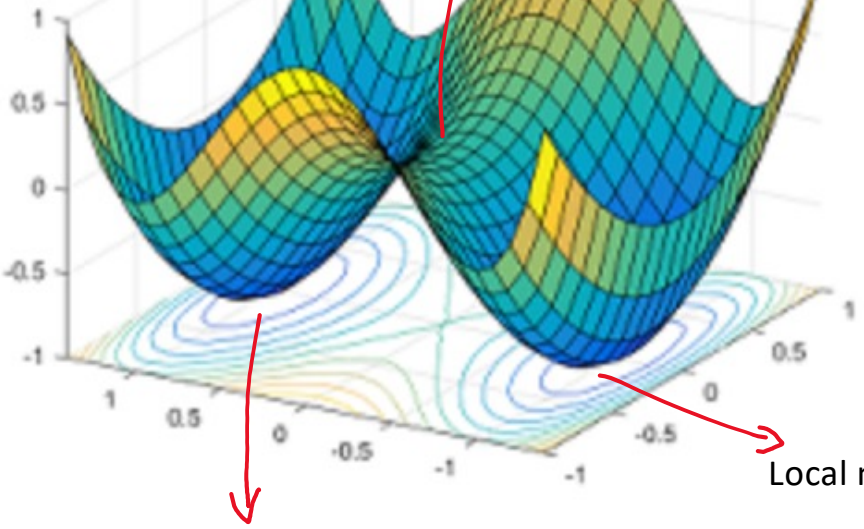
- Low learning rate may cause the algorithm to take too long time to come even close to an optimal solution
- Large learning rate may allow the algorithm to come close to a good solution but will then oscillate around the point or even diverge

Learning rate schedule

- Start with a higher learning rate to explore the loss space => find a good starting values for the weights
- Use smaller learning rates in later steps to converge to a minima =>tune the weights slowly
- Different choices of decay functions:
  - exponential, inverse, multi-step, polynomial
  - babysitting the learning rate
- Training with different learning rate decay
  - Keras learning rate schedules and decay
- Other new forms: cosine decay

Decay functions	Decay equation
Inverse	$\alpha_t = \frac{\alpha_0}{1+\gamma.t}$
exponential	$\alpha_t = \alpha_0 exp(-\gamma.t)$
polynomial n=1 gives linear	$\alpha_t = \alpha_0 \left(1 - \frac{t}{\max\_t}\right)^n$
multi-step	$\alpha_t = \frac{\alpha_0}{\gamma^n}$ at step n

Non-convex Loss Functions

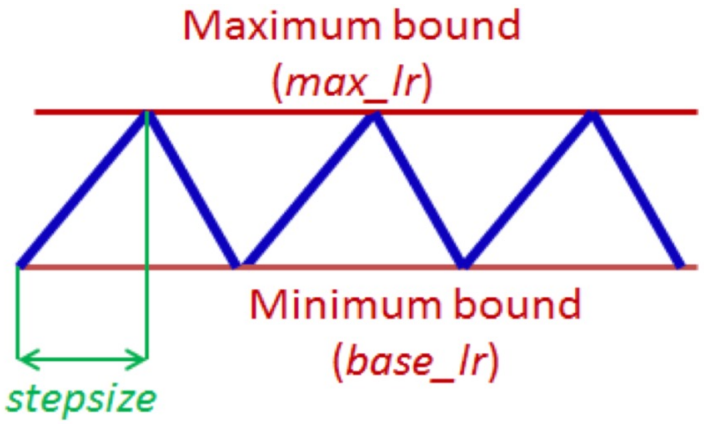


$$y = x_1^4 - 2x_1^2 + x_2^2$$

$$f(y) \approx f(x) + \langle \nabla f(x), y - x \rangle + \frac{1}{2}(y - x)^\top \nabla^2 f(x)(y - x)$$

https://www.offconvex.org/2016/03/22/saddlepoints/

Cyclical Learning Rate



Dataset	LR policy	Iterations	Accuracy (%)
CIFAR-10	*fixed*	70,000	81.4
CIFAR-10	*triangular2*	**25,000**	81.4
CIFAR-10	*decay*	25,000	78.5
CIFAR-10	*exp*	70,000	79.1
CIFAR-10	*exp_range*	42,000	**82.2**
AlexNet	*fixed*	400,000	58.0
AlexNet	*triangular2*	400,000	**58.4**
AlexNet	*exp*	300,000	56.0
AlexNet	*exp*	460,000	56.5
AlexNet	*exp_range*	300,000	56.5
GoogLeNet	*fixed*	420,000	63.0
GoogLeNet	*triangular2*	420,000	**64.4**
GoogLeNet	*exp*	240,000	58.2
GoogLeNet	*exp_range*	240,000	60.2

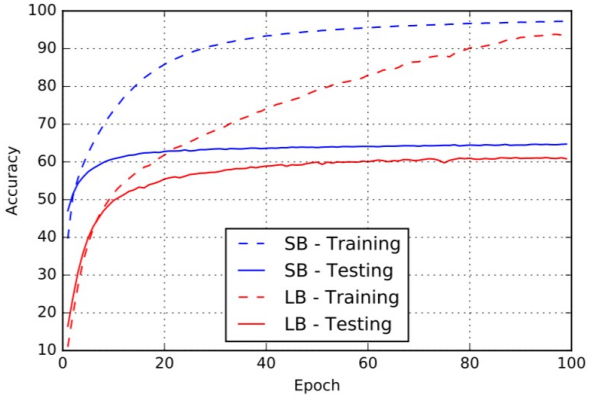
- Idea is to have learning rate continuously change in cyclical manner with alternate increase and decrease phases
- Keras implementation available; Look at example Cyclical Learning Rates with Keras and Deep Learning

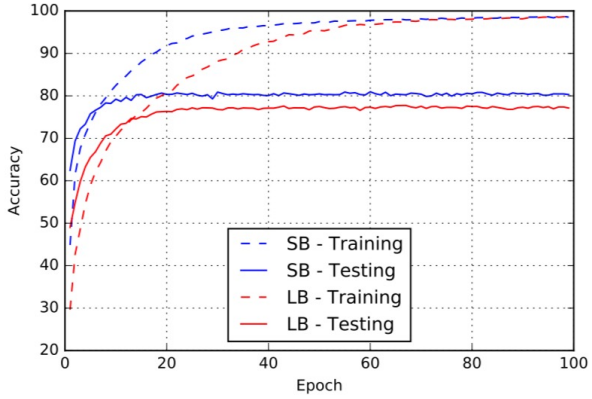
Batch size

- Effect of batch size on learning
- Batch size is restricted by the GPU memory (12GB for K40, 16GB for P100 and V100) and the model size
  - Model and batch of data needs to remain in GPU memory for one iteration
- Are you restricted to work with small size mini-batches for large models and/or GPUs with limited memory
  - No, you can simulate large batch size by delaying gradient/weight updates to happen every n iterations (instead of n=1) ; supported by frameworks

What Batch size to choose ?

- Hardware constraints (GPU memory) dictate the largest batch size
- Should we try to work with the largest possible batch size ?
  - Large batch size gives more confidence in gradient estimation
  - Large batch size allows you to work with higher learning rates, faster convergence
- Large batch size leads to poor generalization (Keskar et al 2016)
  - Lands on sharp minima wheareas small batch SGD find flat minimas which generalize better





Learning rate and Batch size relationship

- "Noise scale" in stochastic gradient descent (Smith et al 2017)

$$g = \epsilon\left(\frac{N}{B} - 1\right)$$

$$g \approx \frac{\epsilon N}{B} \quad as\ B \ll N$$

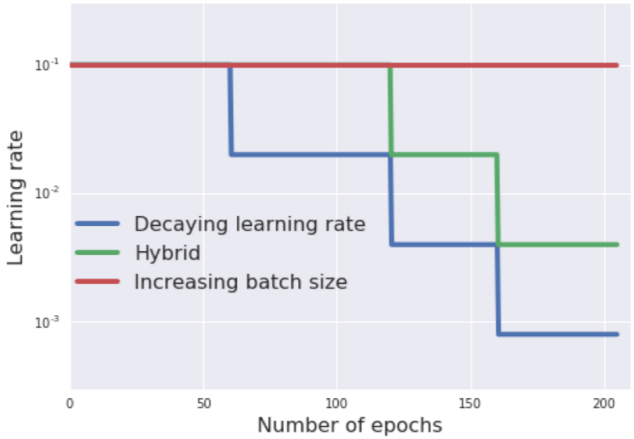
N: training dataset size

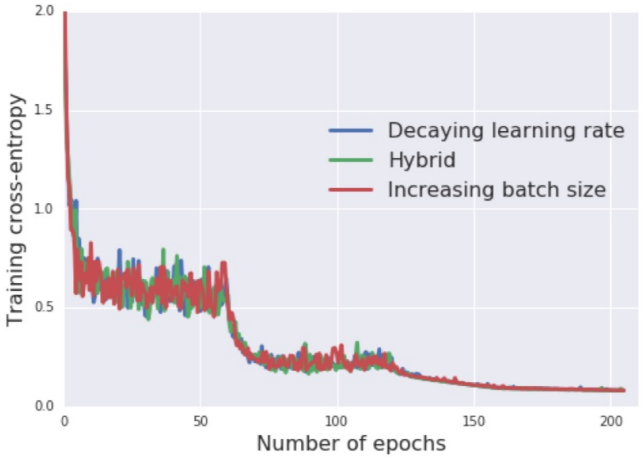
B: batch size

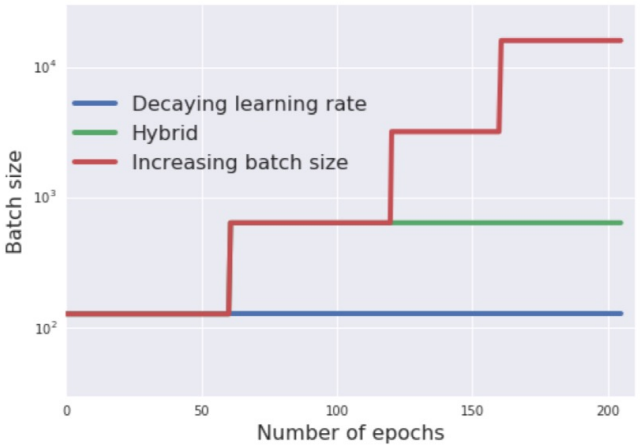
ϵ: learning rate

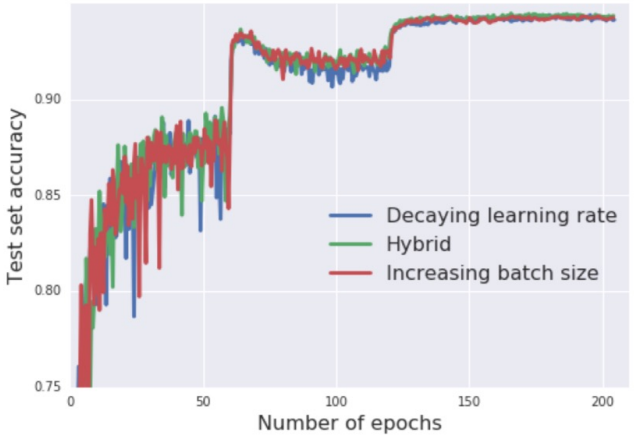
- There is an optimum noise scale g which maximizes the test set accuracy (at constant learning rate)
  - Introduces an optimal batch size proportional to the learning rate when B ≪ N
- Increasing batch size will have the same effect as decreasing learning rate
  - Achieves near-identical model performance on the test set with the same number of training epochs but significantly fewer parameter updates

Learning rate decrease Vs Batch size increase









Reading List

- **Learning rate and Batch size**
  - Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying & Quoc V. Le DON'T DECAY THE LEARNING RATE, INCREASE THE BATCH SIZE
  - Keskar et al On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima
  - Leslie N. Smith Cyclical Learning Rates for Training Neural Networks
  - Elad Hoffer et al Augment your batch: better training with larger batches
- **Weight initialization**
  - Glorot and Bengio. Understanding the difficulty of training deep feedforward neural networks

Blogs/Code Links

- David Morton Increasing Mini-batch Size without Increasing Memory
- Adrian Rosebrock Keras learning rate schedules and decay