LECTURE 2:

OPTIMIZATION IN DEEP LEARNING

University of Washington, Seattle

Fall 2025

Previously in EEP 596…

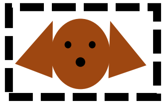
Fixed model

$$x_n \rightarrow \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \rightarrow X_k$$

Learned model

$$x_n \rightarrow \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \rightarrow X_k$$

Fixed vs Learned model



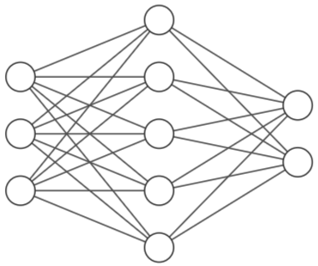
Model

☹

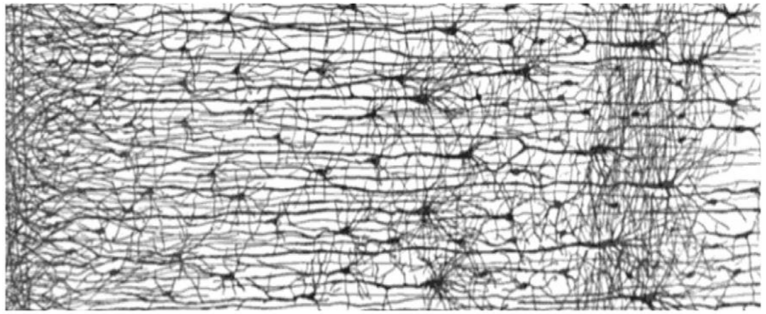
1: Café Poodle
0: Not Café Poodle

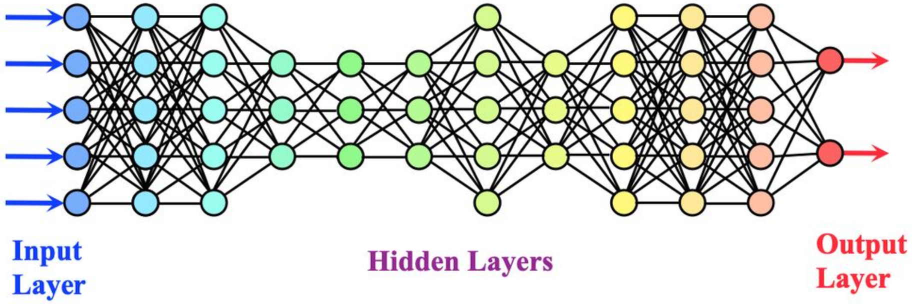
Classical ML methods

Previously in EEP 596…



1: Café Poodle
0: Not Café Poodle



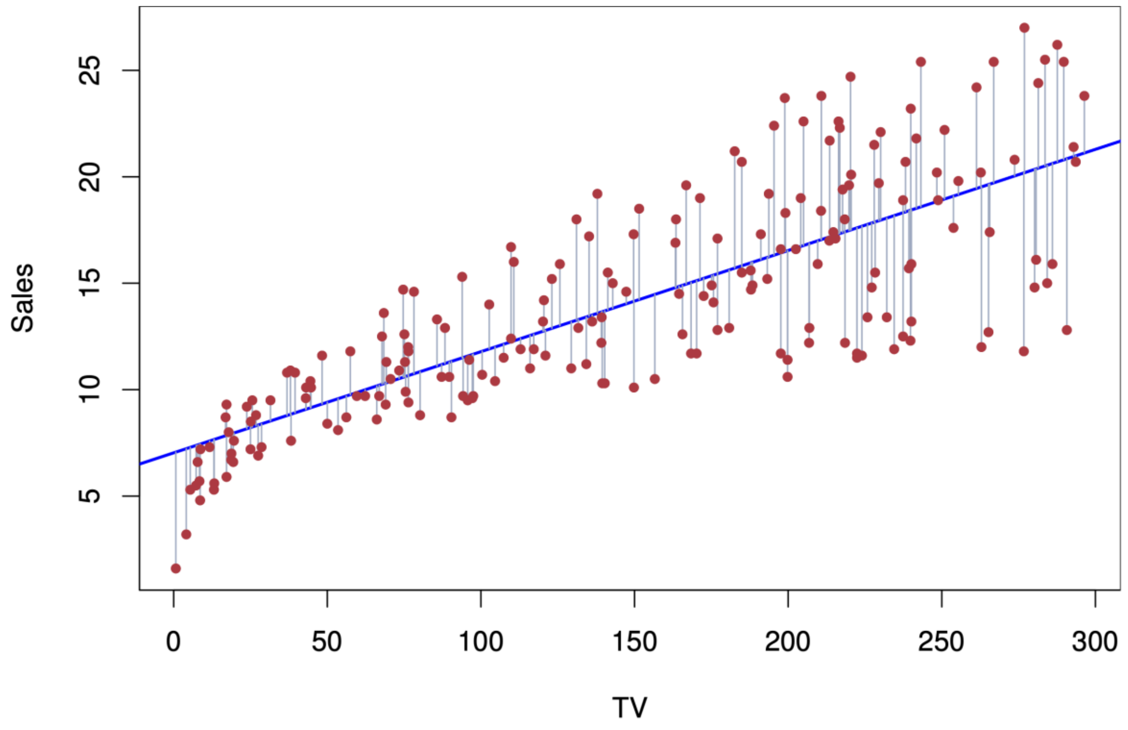


ANN as a model

Deep ANN as brain analogue

Previously in EEP 596…

$$Y = f(X, W)$$



Linear Regression

OUTLINE

Part 1: Binary Classification

- Binary Logistic Regression
- Linear vs Logistic regression
- Logistic regression and Neuron

Part 2: Training and Optimization of a Neuron

- Binary Cross Entropy Loss function
- Training Logistic Regression
- Gradient descent
- Back-propagation algorithm

Part 3: Stochastic Gradient Descent

- GD vs SGD
- Convergence of SGD
- Learning rate and convergence
- Comparing GD variants

Part 4: Optimization Techniques in DL

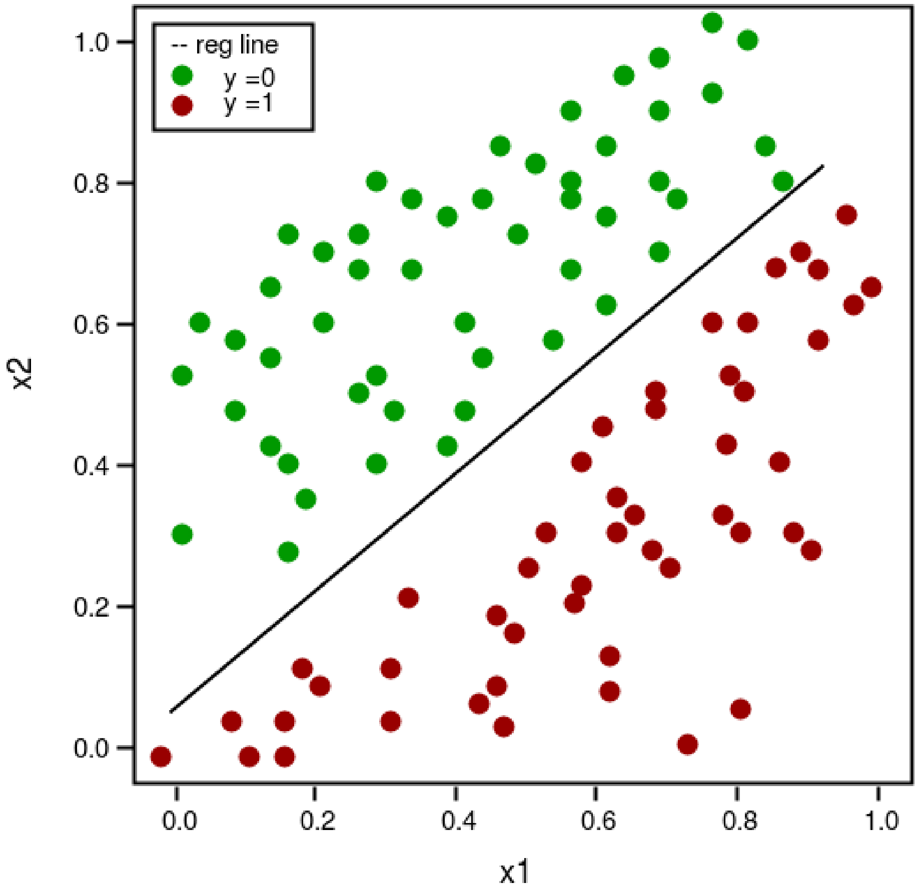
- Variable learning rate
- Advanced methods
- Cross validation
- Regularization/Normalization/Initialization
- Hyperparameter tunings

PART 1:

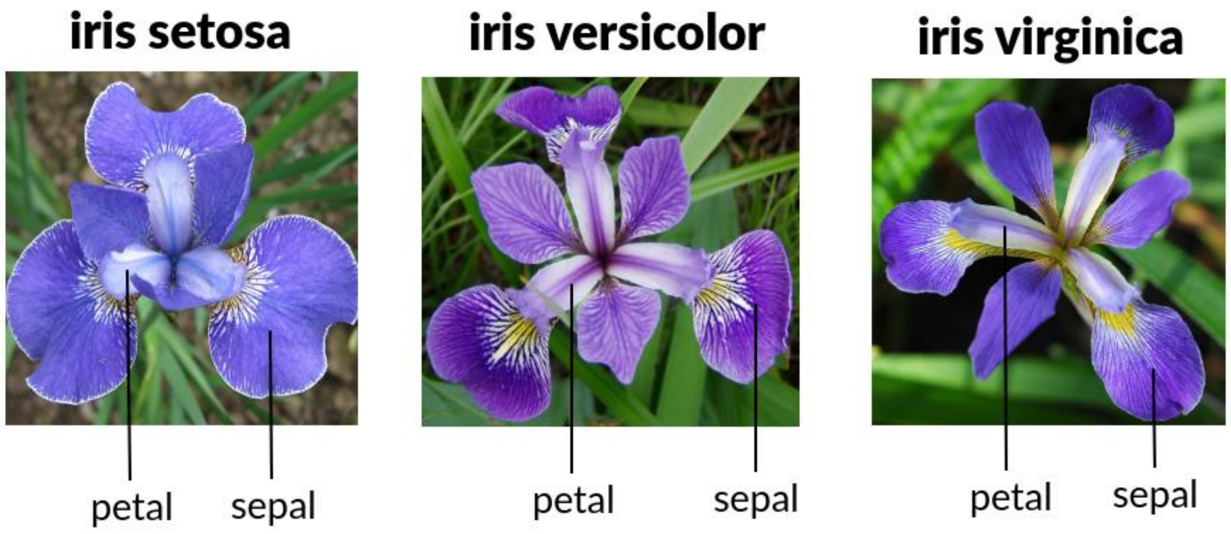
Binary Classification

Binary classification problem

$$y = \sigma(\vec{w}^T \vec{x} + b)$$



Iris Flower Data



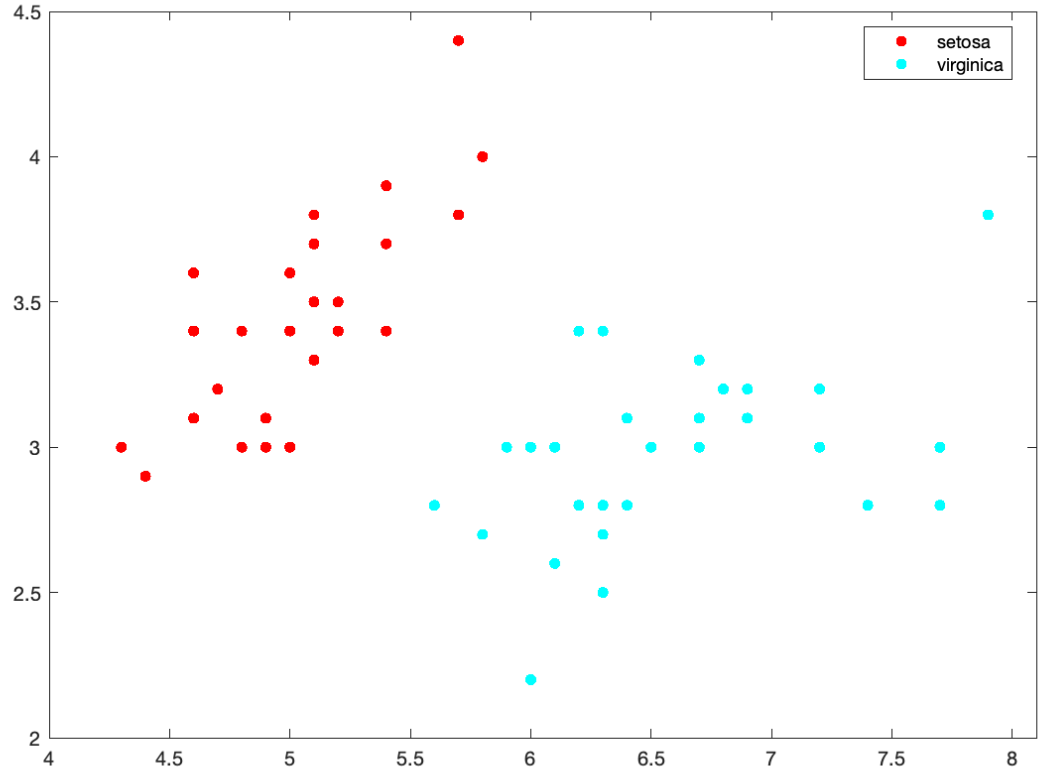
	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa
5	6	5.4	3.9	1.7	0.4	Iris-setosa
6	7	4.6	3.4	1.4	0.3	Iris-setosa
7	8	5.0	3.4	1.5	0.2	Iris-setosa
8	9	4.4	2.9	1.4	0.2	Iris-setosa
9	10	4.9	3.1	1.5	0.1	Iris-setosa

Features

Labels (Targets)

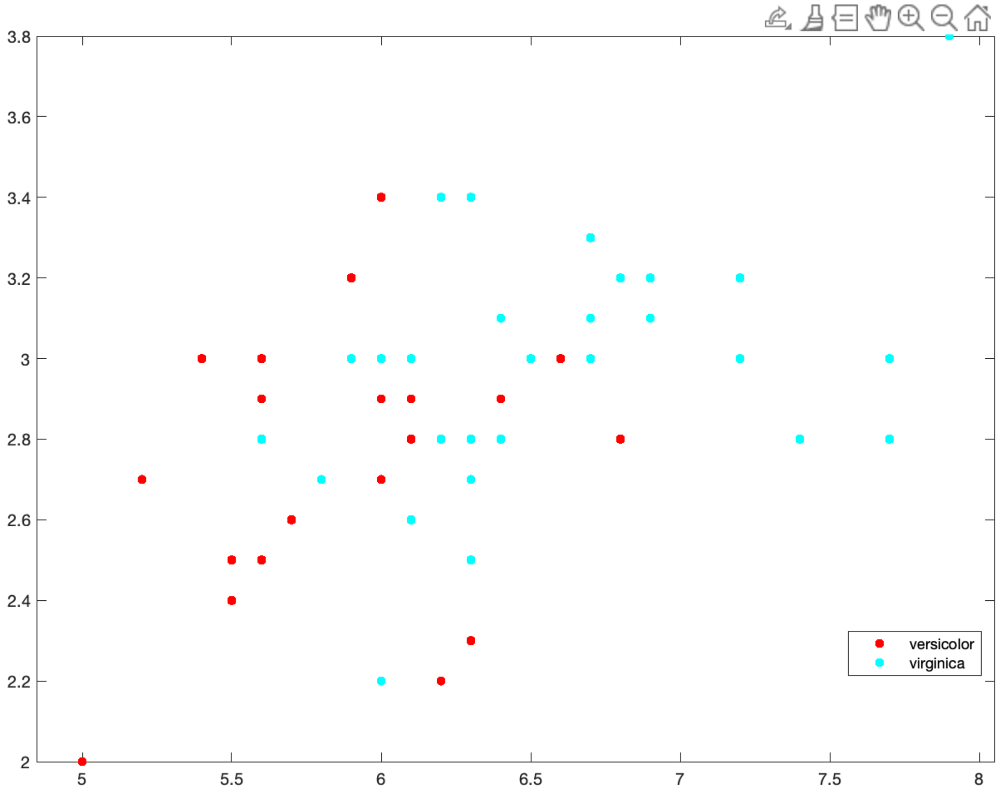
Linear vs Logistic regression

Sepal length (x_2)



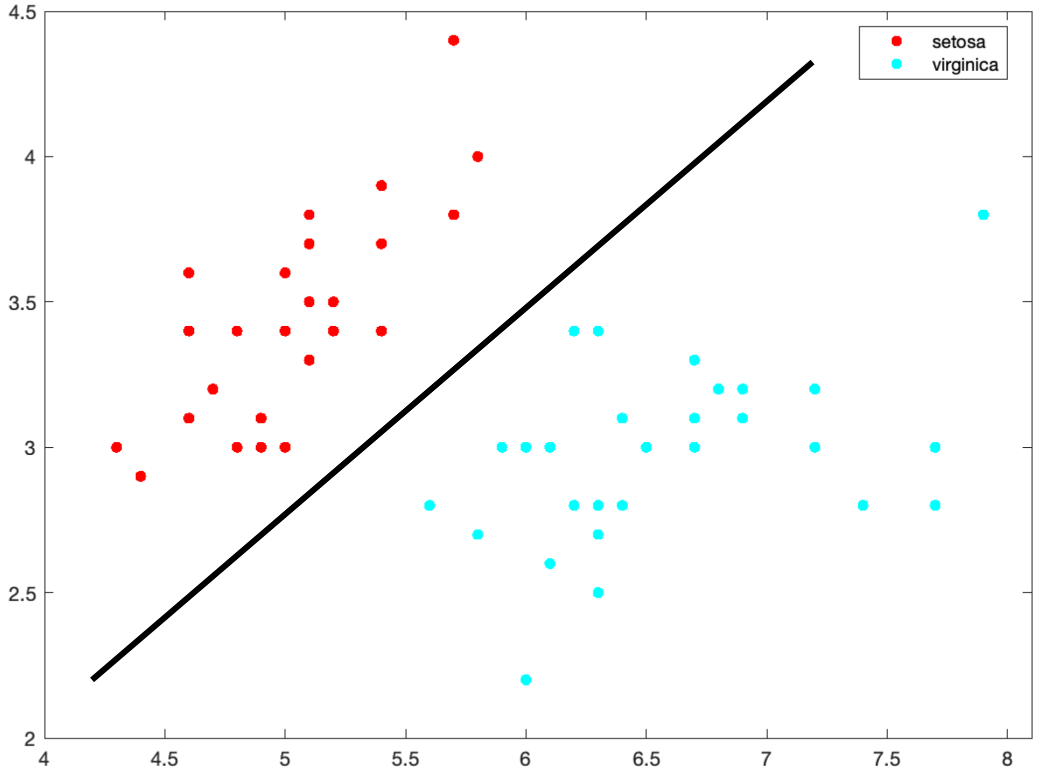
Sepal width (x_1)

Setosa vs Virginica

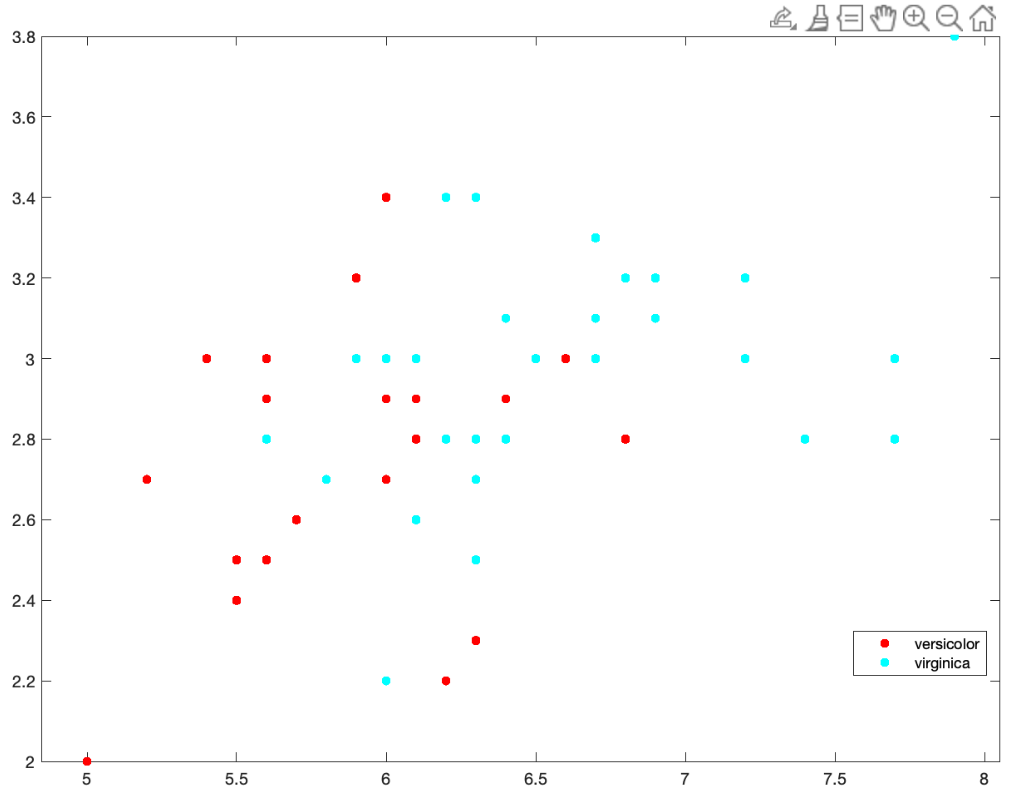


Virsicolor vs Virginica

Linear vs Logistic regression

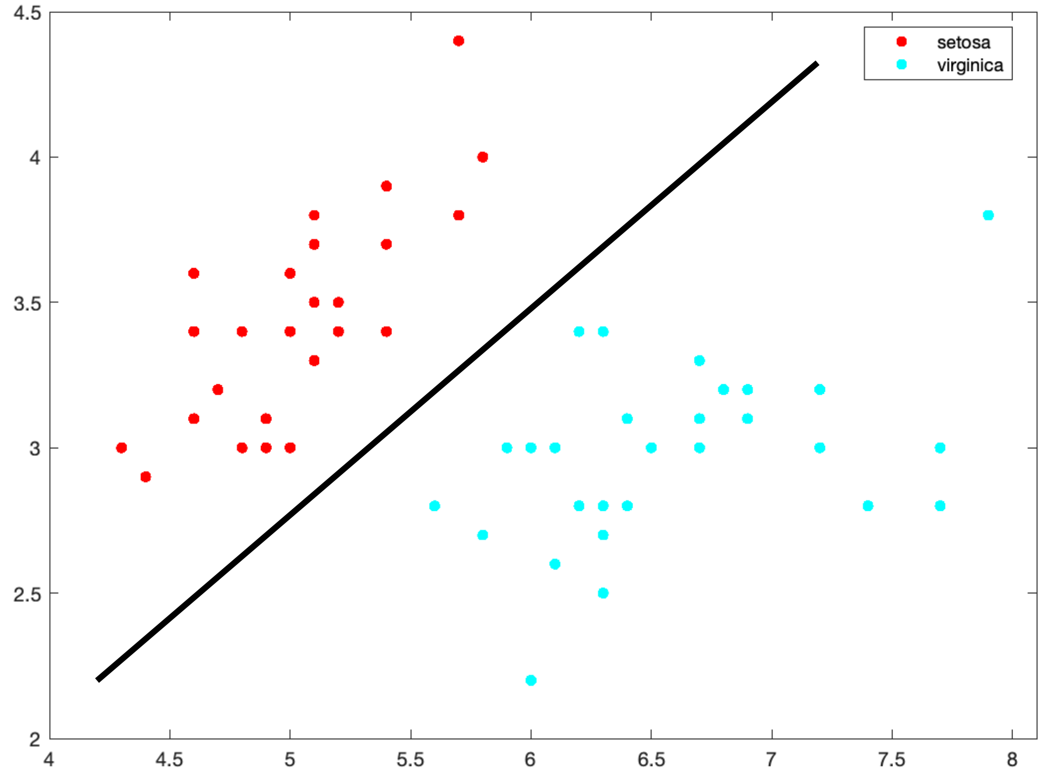


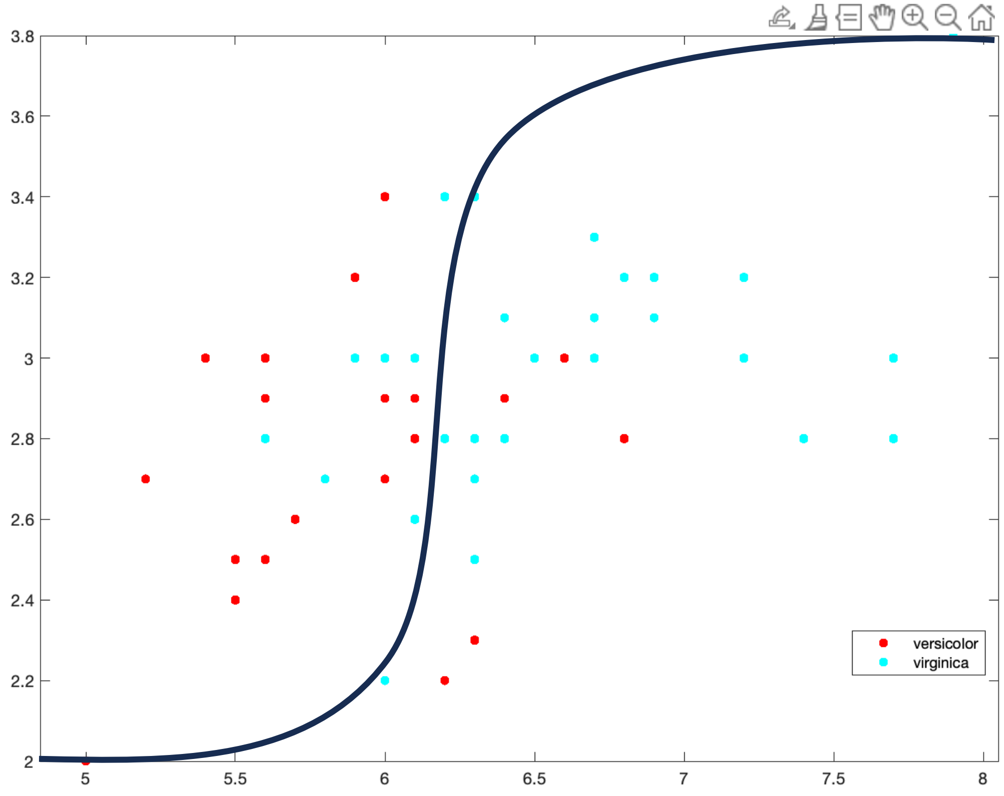
Setosa vs Virginica



Virsicolor vs Virginica

Linear vs Logistic regression

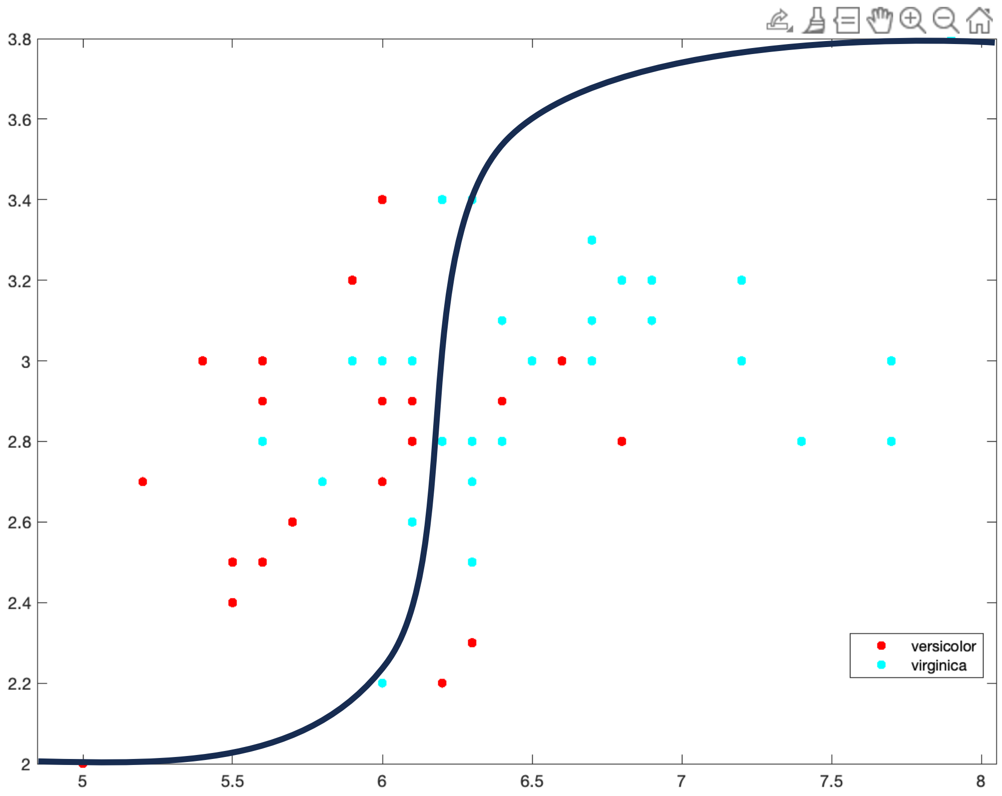




Setosa vs Virginica

Virsicolor vs Virginica

Linear vs Logistic regression



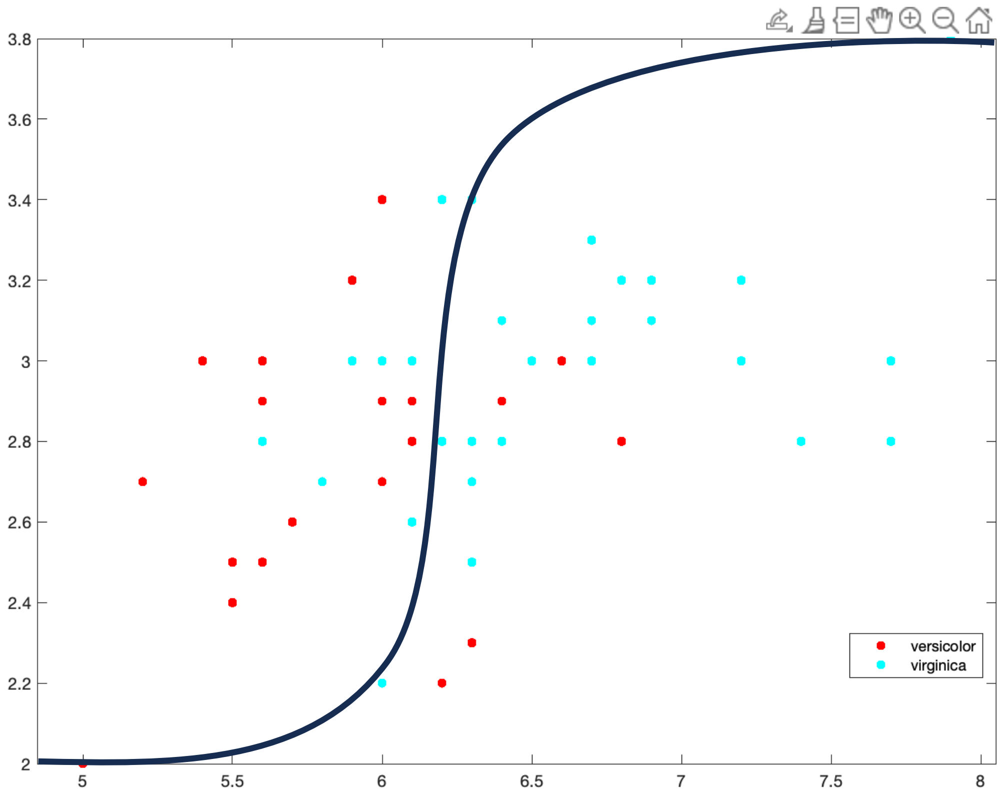
$$p = \frac{1}{1 + e^{-z}}$$

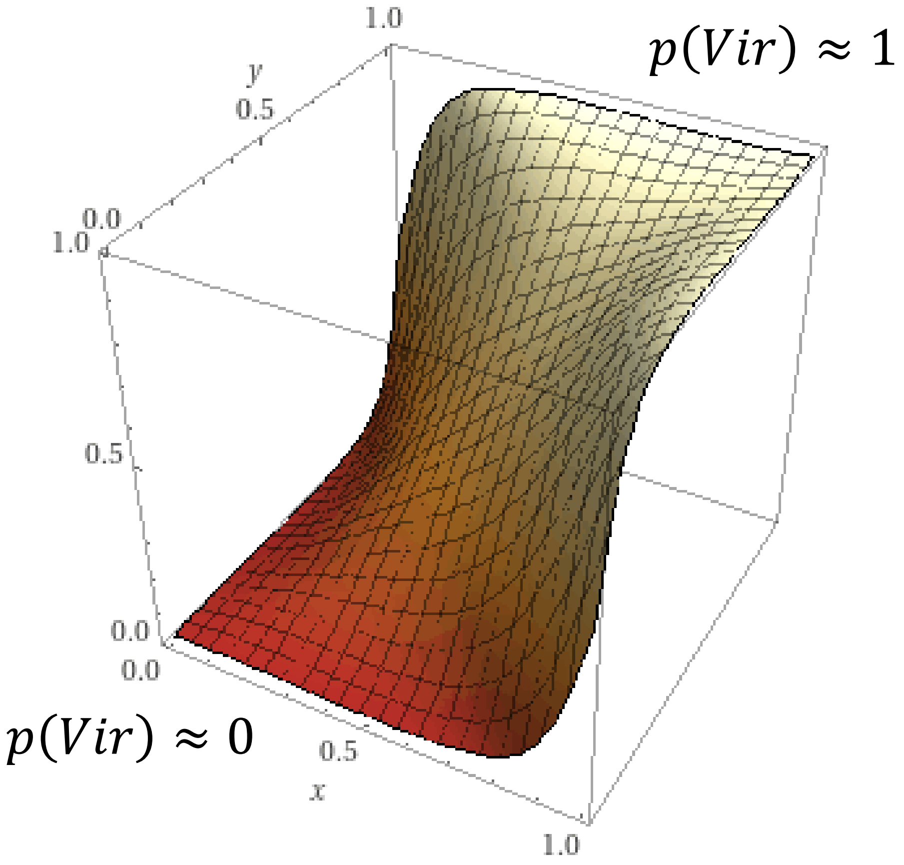
$$0 \leq p \leq 1$$

$$z = \vec{\beta}^T \vec{x} + \beta_0$$

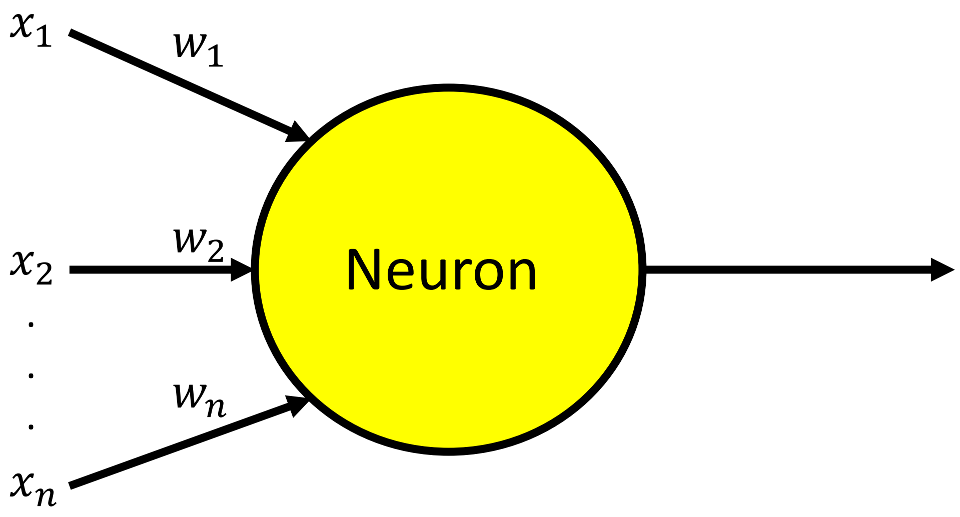
$$\vec{x} = [x_1, x_2] \qquad \vec{\beta} = [\beta_1, \beta_2]$$

Linear vs Logistic regression

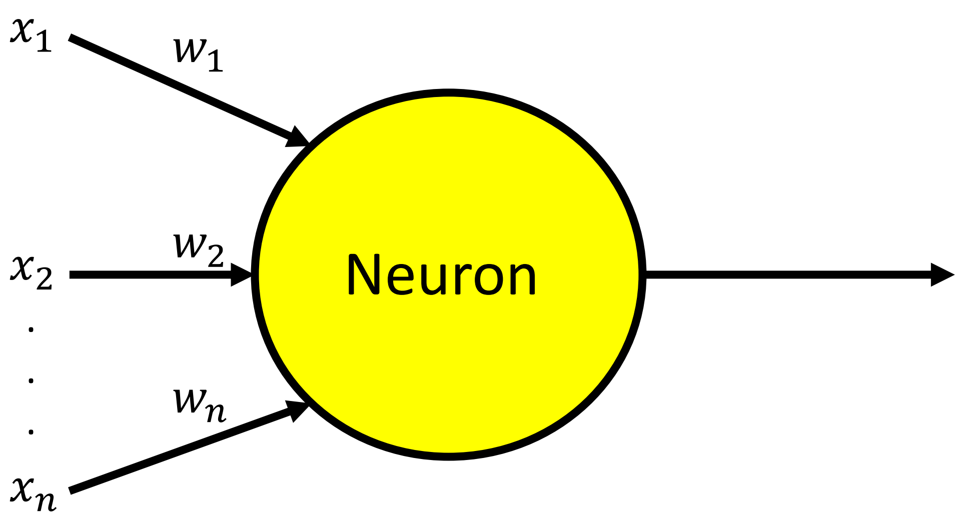




Artificial Neuron as Logistic Regression Model



Artificial Neuron as Logistic Regression Model



Input

$$z = \vec{w}^T \vec{x} + b$$

$$\vec{x} \in \mathbb{R}^n$$

Activation

$$\sigma = \frac{1}{1 + e^{-z}}$$

Output

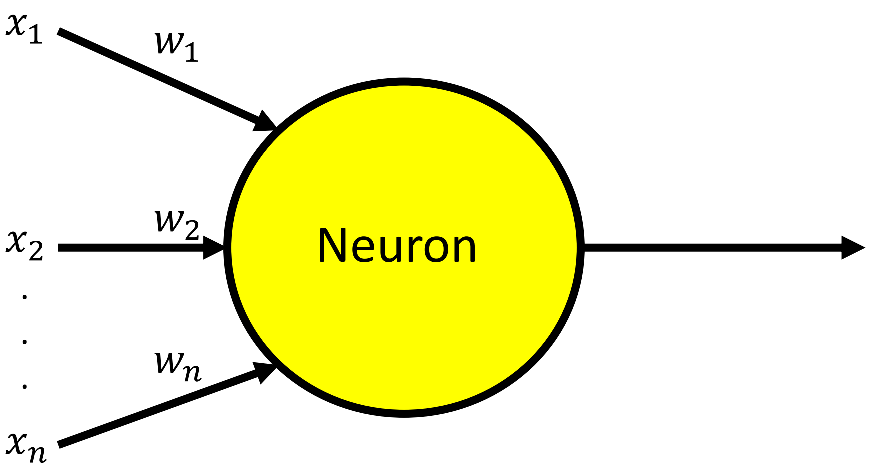
$$y = \sigma(\vec{w}^T \vec{x} + b)$$

$$y = P(y = 1|\vec{x}), 0 \leq y \leq 1$$

PART 2:

Training and Optimization of a Neuron

Cross Entropy Loss function (Binary)



Input

Activation

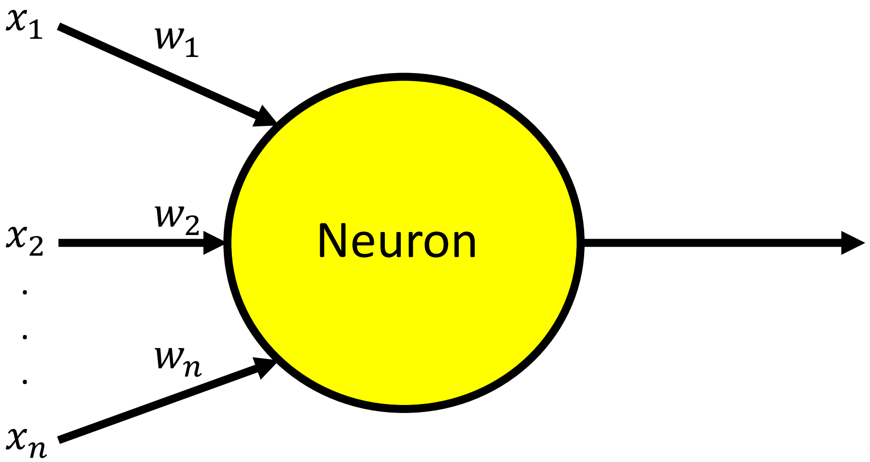
Output

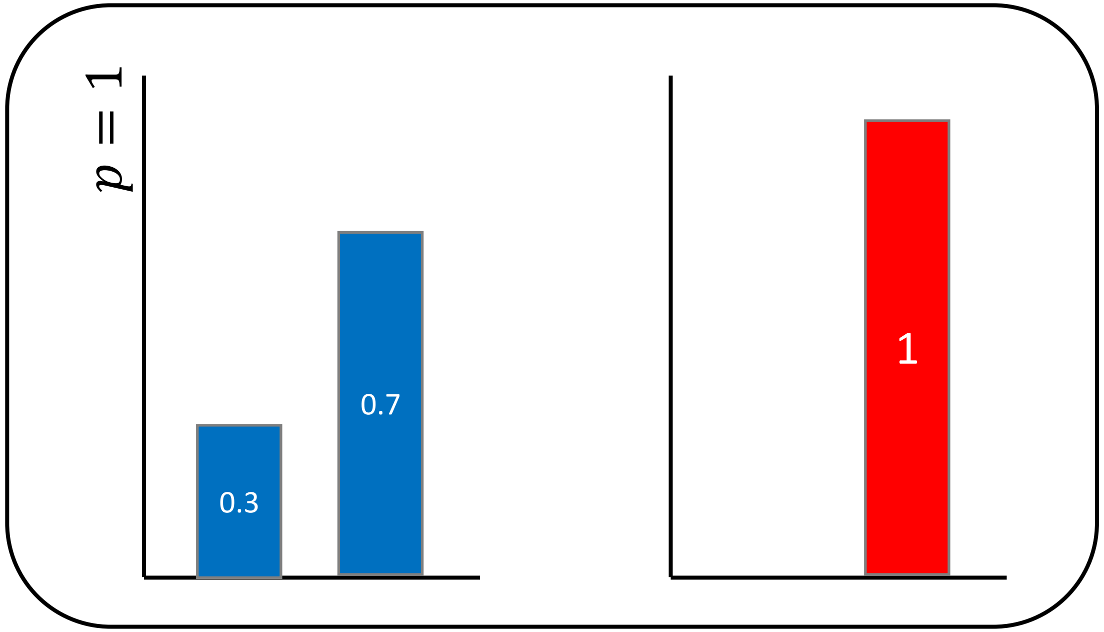
$$z = \vec{w}^{T}\vec{x}$$

$$\sigma = \frac{1}{1 + e^{-z}}$$

$$y = \sigma(\vec{w}^{T}\vec{x} + b)$$

Cross Entropy Loss function (Binary)





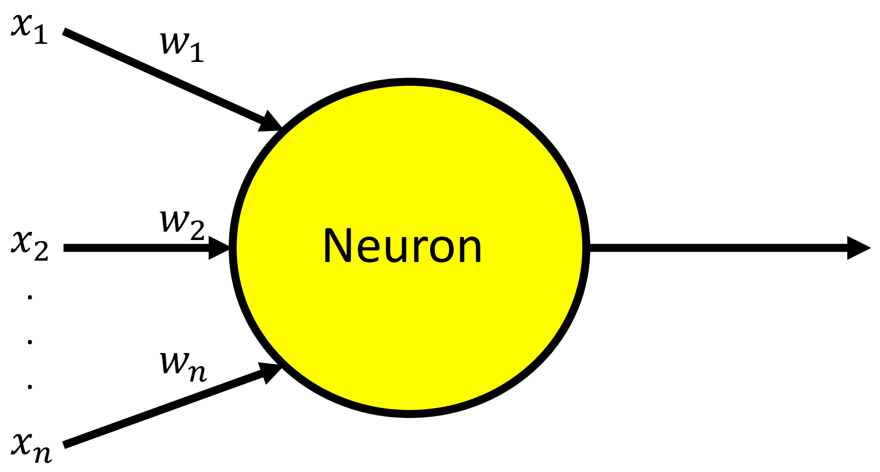
Input | Activation | Output | Model | Ground truth data

$$z = \vec{w}^T \vec{x}$$

$$\sigma = \frac{1}{1 + e^{-z}}$$

$$y = \sigma(\vec{w}^T \vec{x} + b)$$

Cross Entropy Loss function (Binary)

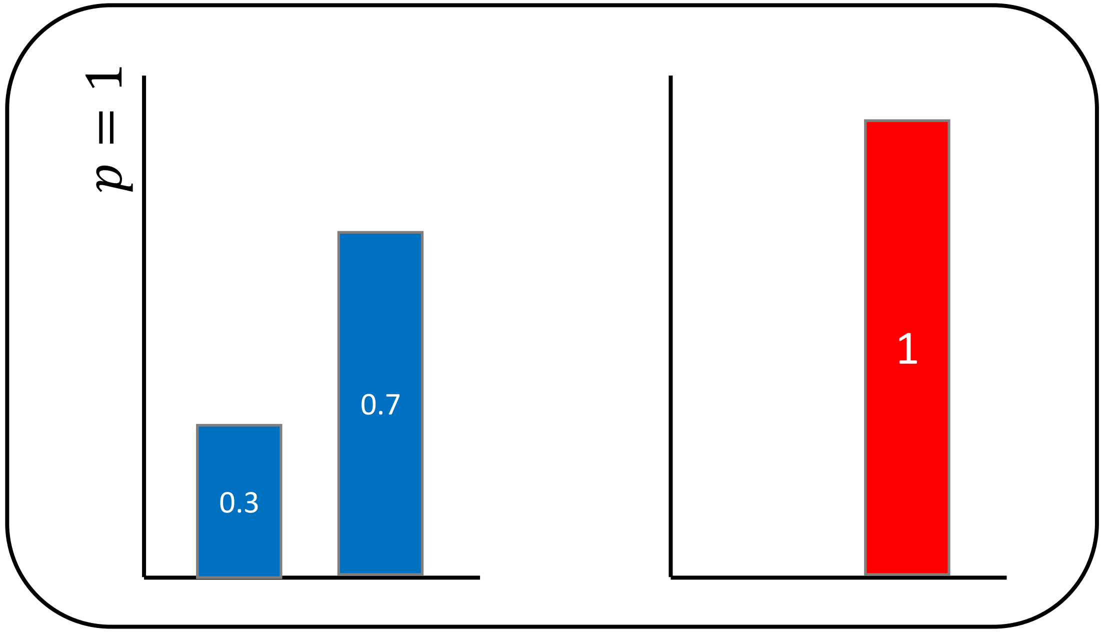


Input | Activation | Output

$$z = \vec{w}^T \vec{x}$$

$$\sigma = \frac{1}{1 + e^{-z}}$$

$$y = \sigma(\vec{w}^T \vec{x} + b)$$



Model | Ground truth data

$J \approx$ Measure of difference in distributions

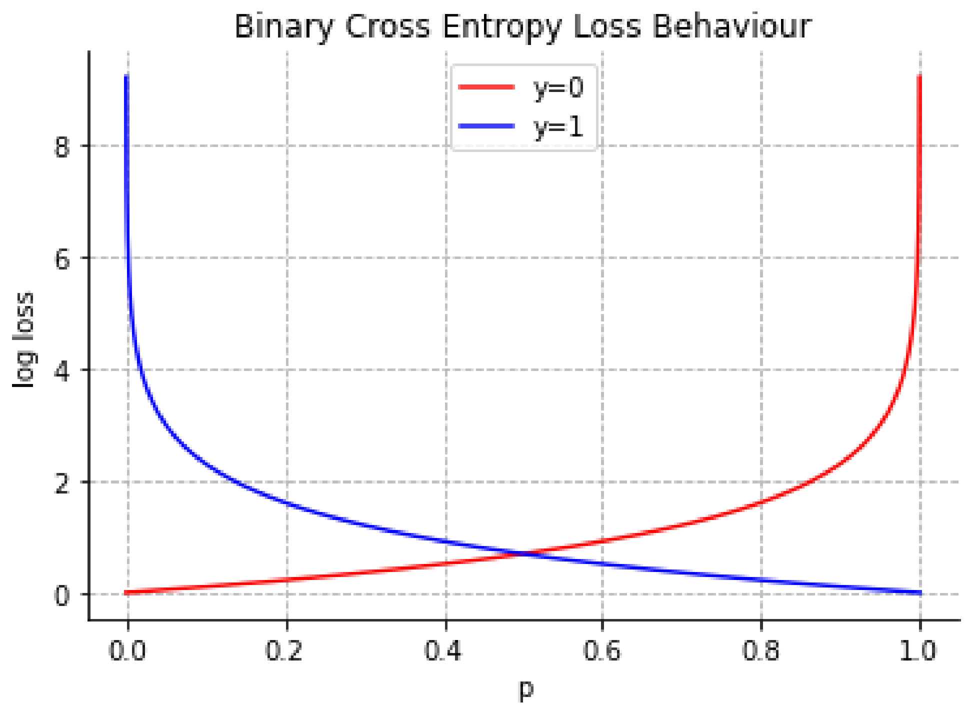
Cross Entropy Loss function (Binary)

$$L(\hat{y}, y) = f(x) = \begin{cases} -\,log(\hat{y})\,, & y = 1 \\ -log(1 - \hat{y}), & y = 0 \end{cases}$$

Cross Entropy Loss function (Binary)

$$L(\hat{y}, y) = f(x) = \begin{cases} -\,log(\hat{y})\,, & y = 1 \\ -log(1-\hat{y}), & y = 0 \end{cases}$$

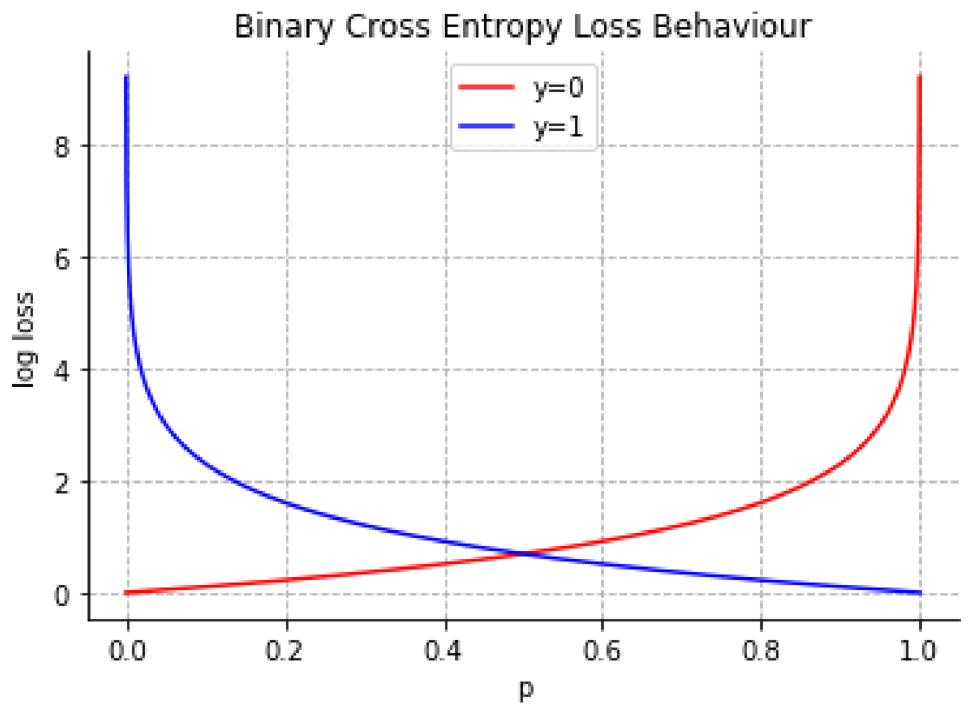
$$L(\hat{y}, y) = -(y\log\hat{y} + (1-y)\log(1-\hat{y}))$$



Cross Entropy Loss function (Binary)

$$L(\hat{y}, y) = f(x) = \begin{cases} -log(\hat{y}), & y = 1 \\ -log(1-\hat{y}), & y = 0 \end{cases}$$

$$L(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log (1-\hat{y}))$$



Cross Entropy Loss is Convex

$\leftrightarrow L(\hat{y}, y)'' \geq 0$

The line segment between any two points does not lie below the graph

Logistic Regression Training

Training Set D

$$D : \{(\vec{x}^{(1)}, y^{(1)}), .., (\vec{x}^{(i)}, y^{(i)}), .., (\vec{x}^{(m)}, y^{(m)})\}$$

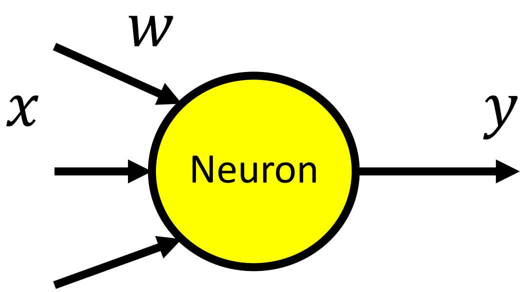
Cost J

$$J(\{\hat{y}\}^m, \{y\}^m; \{\vec{x}\}^m) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

$$= -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)}))$$

Minimizing Loss using Gradient Descent

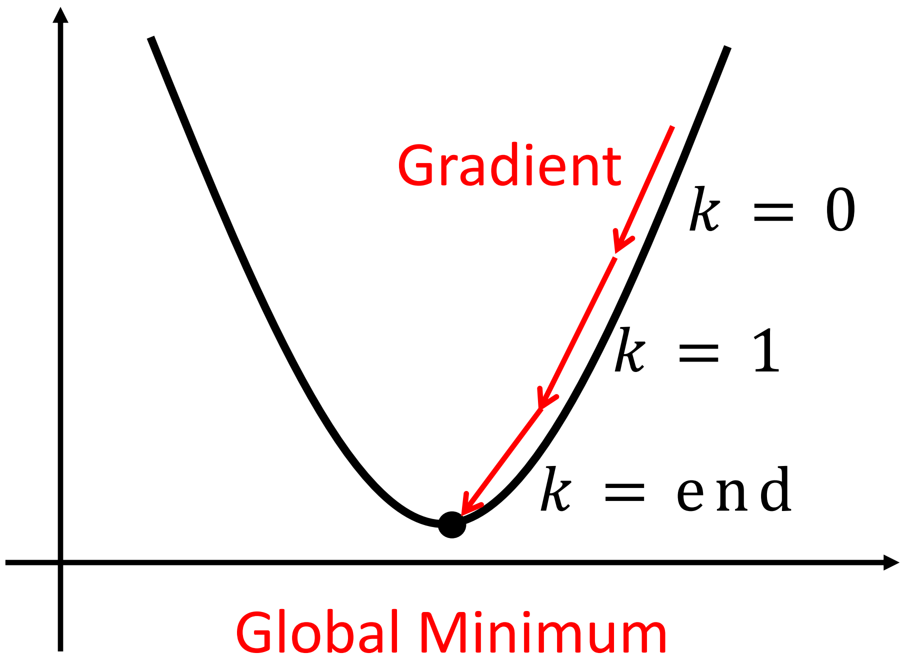
$$y = \sigma(\overrightarrow{w}^T \overrightarrow{x} + b)$$



$$J = L((\overrightarrow{w}, b), y)$$

$$\vec{w}_{k+1} = \vec{w}_k - \alpha \nabla_{\vec{w}} J(\vec{w}_k; b)$$

$$b_{k+1} = b_k - \alpha \frac{\partial}{\partial b} J(\vec{w}; b_k)$$



Minimizing Loss using Gradient Descent

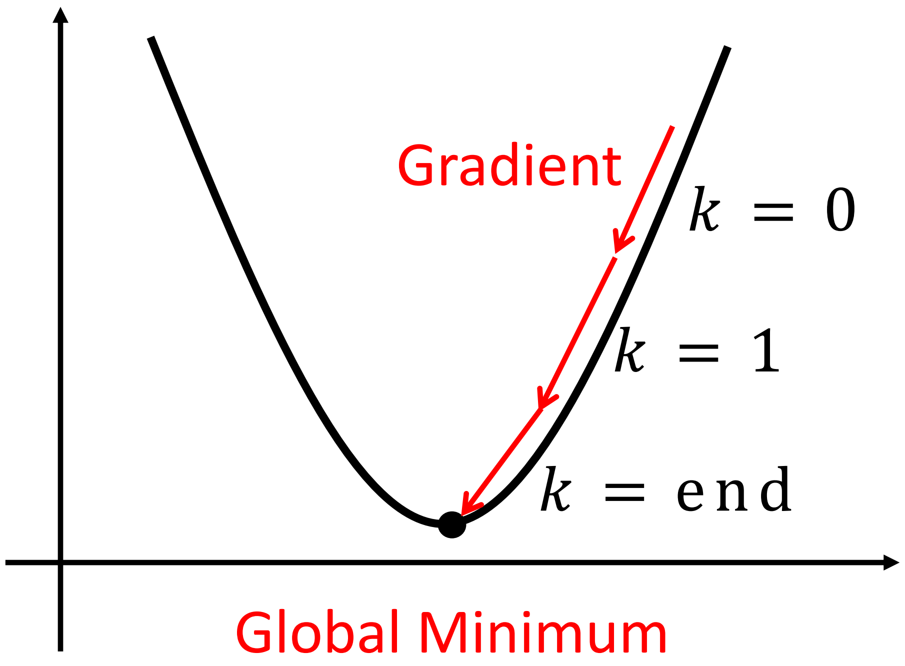
$$y = \sigma(\vec{w}^T\vec{x} + b)$$

Learning rate

$$\vec{w}_{k+1} = \vec{w}_k - \alpha\nabla_{\vec{w}}J(\vec{w}_k; b)$$

$$b_{k+1} = b_k - \alpha\frac{\partial}{\partial b}J(\vec{w}; b_k)$$

$$J = L((\vec{w}, b), y)$$



Iteratively adjust $(\vec{w}, b)$ until we reach the global minimum

Finding Gradient w.r.t. weights and biases

$$J = L(\hat{y}, y)$$

Finding Gradient w.r.t. weights and biases

$$J = L(\underbrace{\hat{y}}_{\hat{y} = \sigma(z)}, y)$$

$\hat{y} = \sigma(z)$ Activation function

Finding Gradient w.r.t. weights and biases

$$J = L(\overbrace{\hat{y}}, y)$$

$$\hat{y} = \sigma(\overbrace{z})$$ Activation function

$$z = \vec{w}^T \vec{x} + b$$ Integrate Inputs

Finding Gradient w.r.t. weights and biases

$$J = L(\overbrace{\hat{y}}^{}, y)$$

$$\hat{y} = \sigma(\overbrace{z}^{})$$

$$z = \vec{w}^T \vec{x} + b$$

$$J = L(\sigma(\vec{w}^T \vec{x} + b), y)$$

Finding Gradient w.r.t. weights and biases

$$J = L(\underbrace{\hat{y}}_{\hat{y} = \sigma(\underbrace{z}_{z = \vec{w}^T \vec{x} + b})}, y)$$

$$J = L(\sigma(\boxed{\vec{w}}^T \vec{x} + b), y)$$

Using Chain Rule to Compute Gradients

$$J = L(\hat{y}, y)$$

$$\hat{y} = \sigma(z)$$

$$z = \vec{w}^T\vec{x} + b$$

$$J = L(\sigma(\boxed{\vec{w}}^T\vec{x} + b), y)$$

$$P = f(g(h(x)))$$

$$\frac{dP}{dx} = \frac{df}{dg} * \frac{dg}{dh} * \frac{dh}{dx}$$

Using Chain Rule to Compute Gradients

$$J = L(\overbrace{\sigma(\underbrace{\vec{w}^T\vec{x} + b}_{z})}^{\hat{y}}, y)$$

$$P = f(g(h(x)))$$

$$\frac{dP}{dx} = \frac{df}{dg} * \frac{dg}{dh} * \frac{dh}{dx}$$

Using Chain Rule to Compute Gradients

$$J = L(\overbrace{\sigma(\underbrace{\vec{w}^T\vec{x} + b}_{z})}^{\hat{y}}, y) \qquad P = f(g(h(x)))$$

$$\frac{\partial J}{\partial \vec{w}} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial \vec{w}} \qquad \frac{dP}{dx} = \frac{df}{dg} * \frac{dg}{dh} * \frac{dh}{dx}$$

$$\frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \nabla_{\vec{w}}\, z$$

Using Chain Rule to Compute Gradients

$$\nabla_{\vec{w}} L(\hat{y}, y) = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \nabla_{\vec{w}} z$$

FWD

$$z = \vec{w}^T \vec{x} + b \longrightarrow \hat{y} = \sigma(z) \longrightarrow L(\hat{y}, y)$$

BWD

$$\nabla_{\vec{w}} z \longleftarrow \frac{\partial \hat{y}}{\partial z}(z) \longleftarrow \frac{\partial L}{\partial \hat{y}}(\hat{y}, y)$$

Using Chain Rule to Compute Gradients

BWD	$\nabla_{\vec{w}} z$	$\leftarrow$	$\frac{\partial \hat{y}}{\partial z}(z)$	$\leftarrow$	$\frac{\partial L}{\partial \hat{y}}(\hat{y}, y)$
$\nabla_{\vec{w}} L$	$\vec{x}$	$\leftarrow$	$\sigma(z)(1 - \sigma(z))$	$\leftarrow$	$-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$
$\frac{\partial L}{\partial b}$	1	$\leftarrow$	$\sigma(z)(1 - \sigma(z))$	$\leftarrow$	$-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$

Training Terminologies

- **Forward Propagation:**
  Computing the **loss through forward pass** for a single training example

- **Backward Propagation:**
  Computing **gradients of parameters** through backward pass for a single training example

- **Batch:**
  Training set could be divided into **smaller sets** called batches

- **Iteration:**
  When an **entire batch** is passed both **forward and backward**

- **Epoch:**
  When an **entire dataset** is passed both **forward and backward** through the NN once

Batch Gradient Descent

Training Set D

$$D : \{(\vec{x}^{(1)}, y^{(1)}), .., (\vec{x}^{(i)}, y^{(i)}), .., (\vec{x}^{(m)}, y^{(m)})\}$$

Cost J

$$J(\{\hat{y}\}^m, \{y\}^m; \{\vec{x}\}^m) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

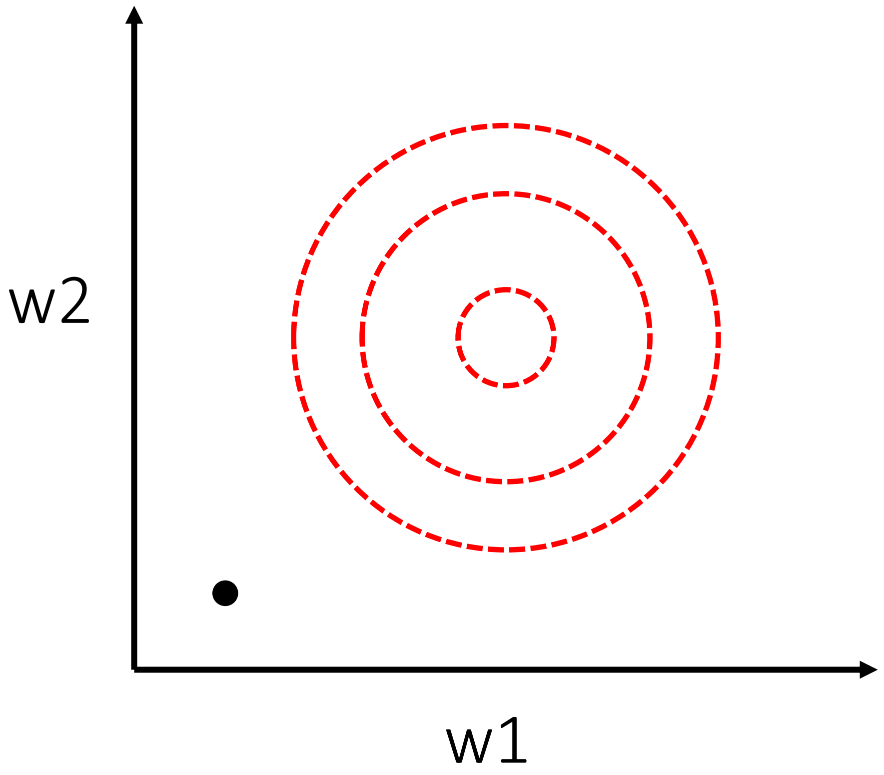
$$= -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)}))$$

Sum the gradients for all m-examples where m = total number of samples in training set (Single epoch)

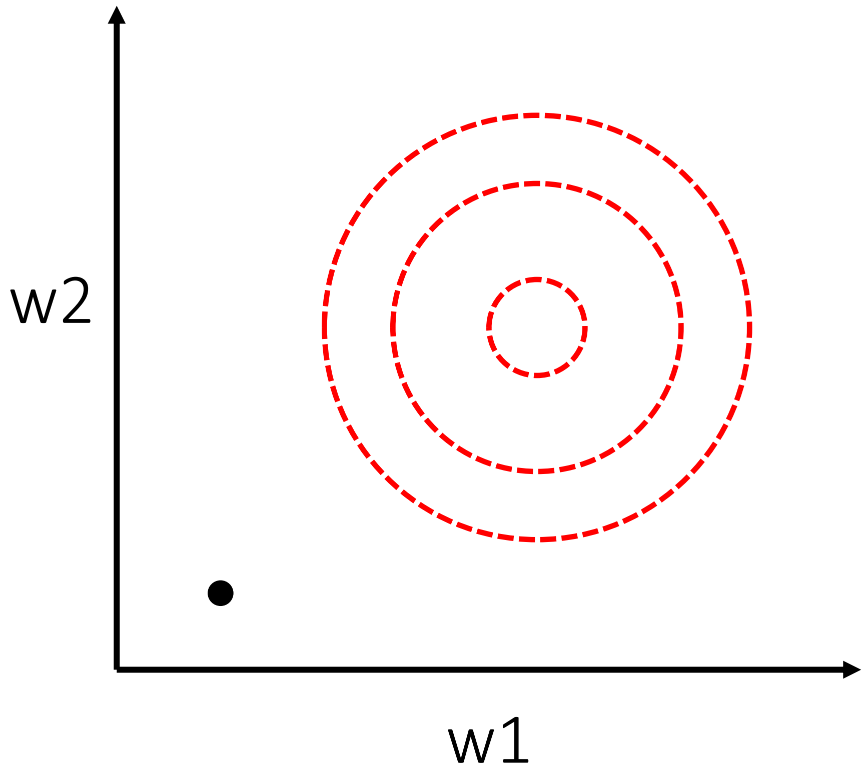
PART 3:

Stochastic Gradient Descent (SGD)

Gradient descent (GD) vs Stochastic Gradient Descent (SGD)

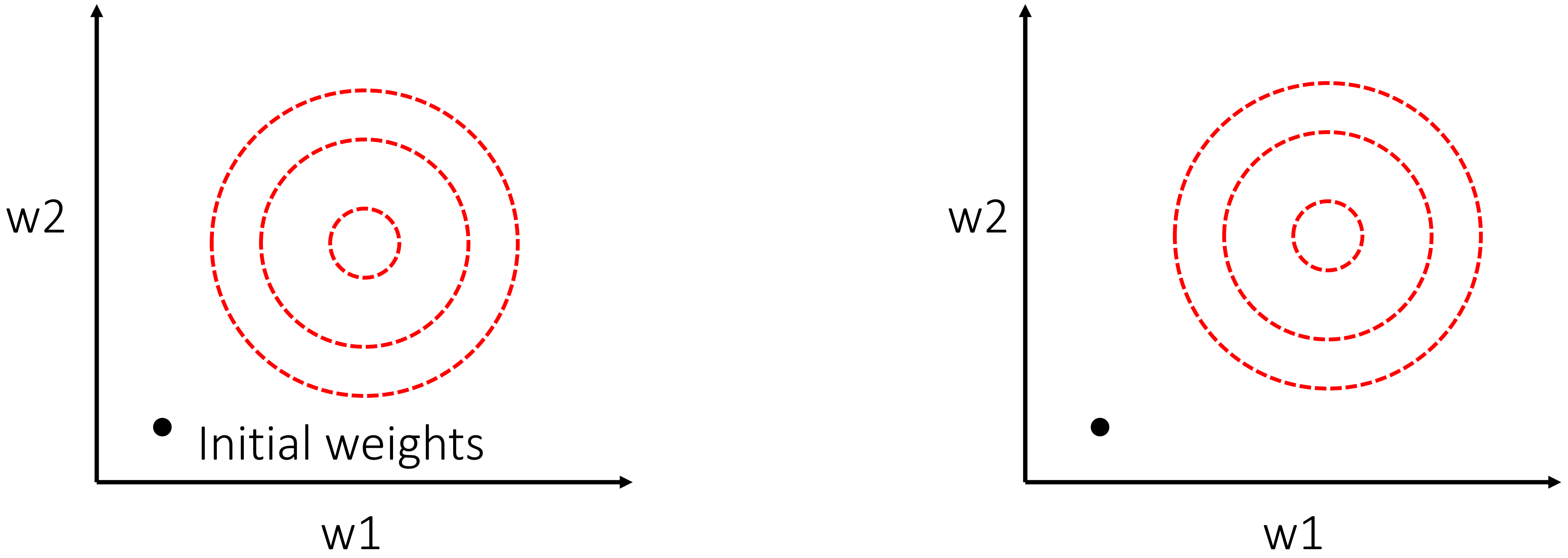


Batch Gradient Descent
1 iteration: FWD pass and BWD pass on **whole training set**



Stochastic Gradient Descent
1 iteration: FWD pass and BWD pass on **subset of training set**

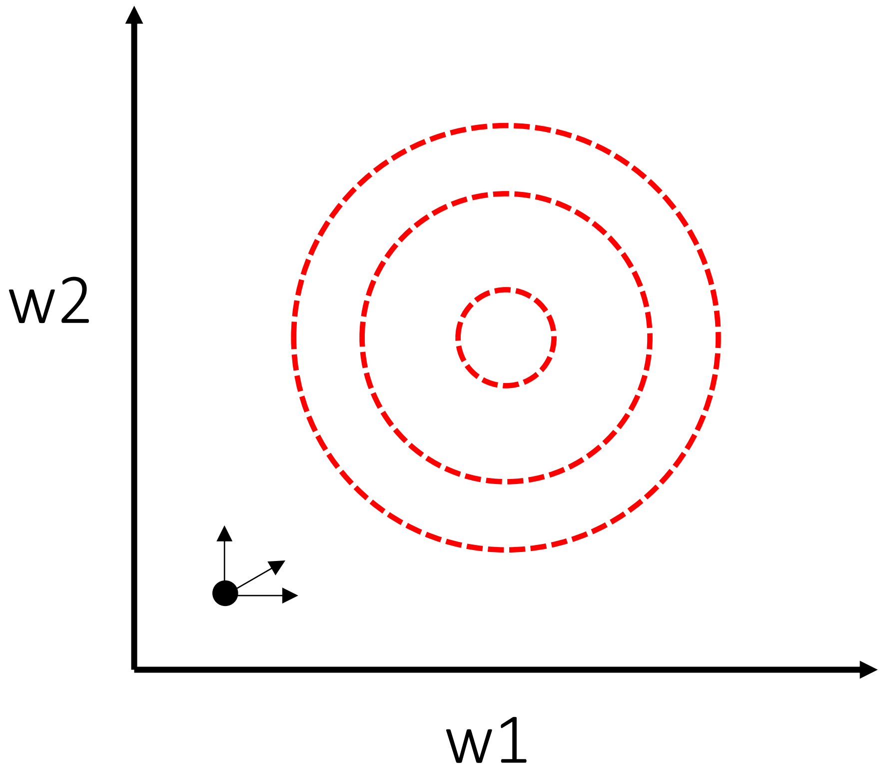
GD vs SGD



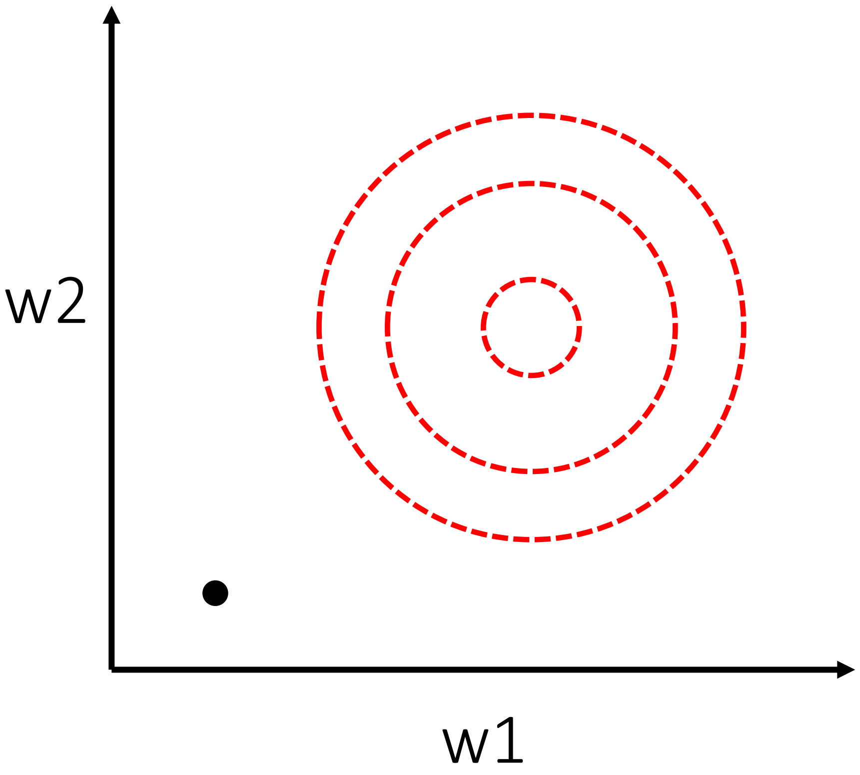
Batch Gradient Descent

Stochastic Gradient Descent

GD vs SGD

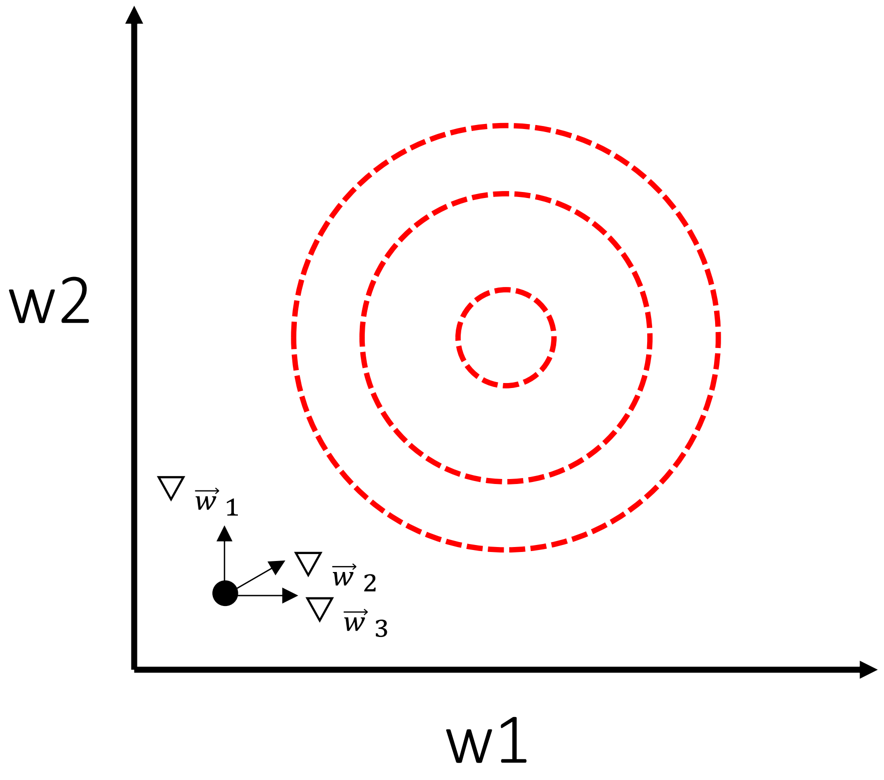


Batch Gradient Descent

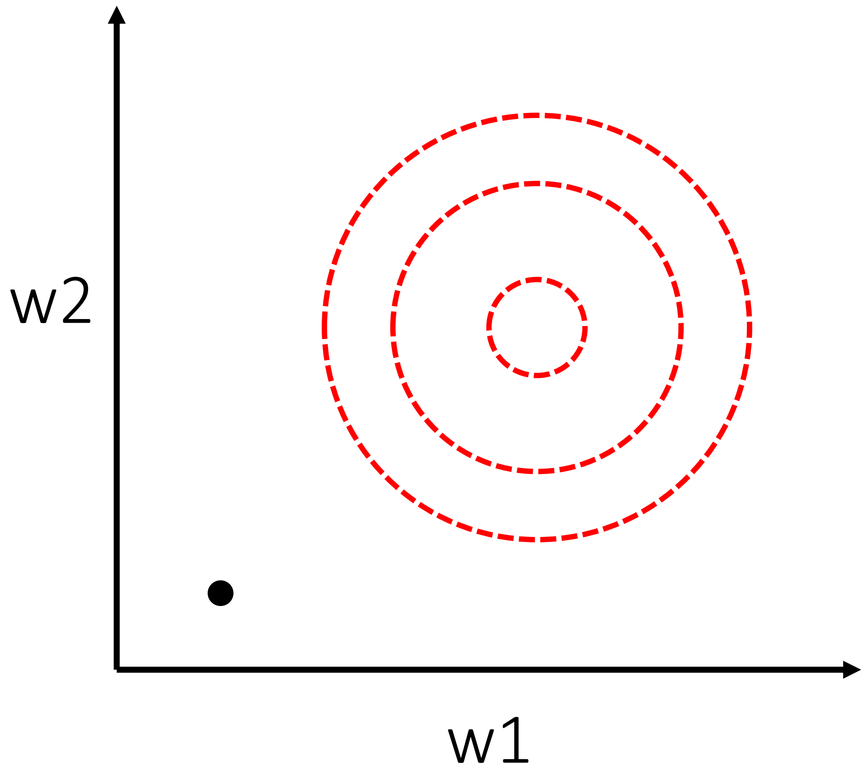


Stochastic Gradient Descent

GD vs SGD

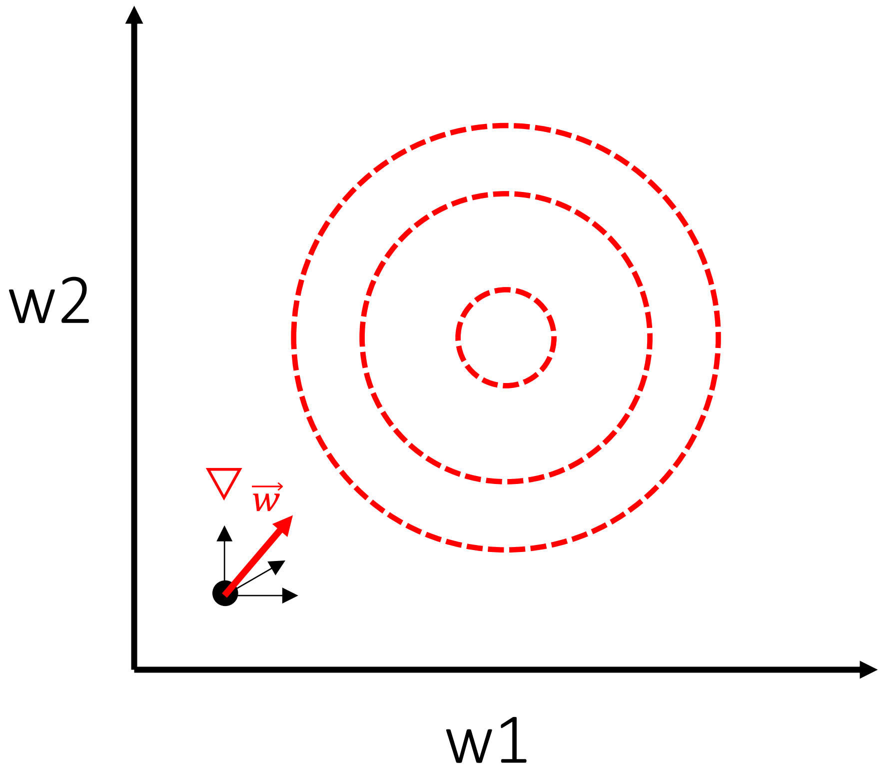


Batch Gradient Descent

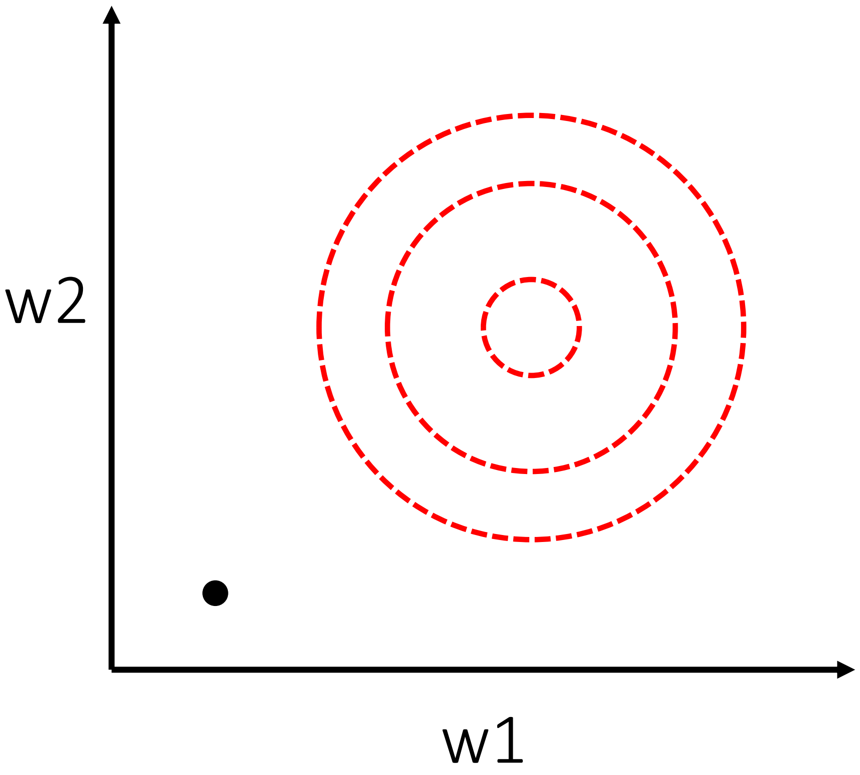


Stochastic Gradient Descent

GD vs SGD

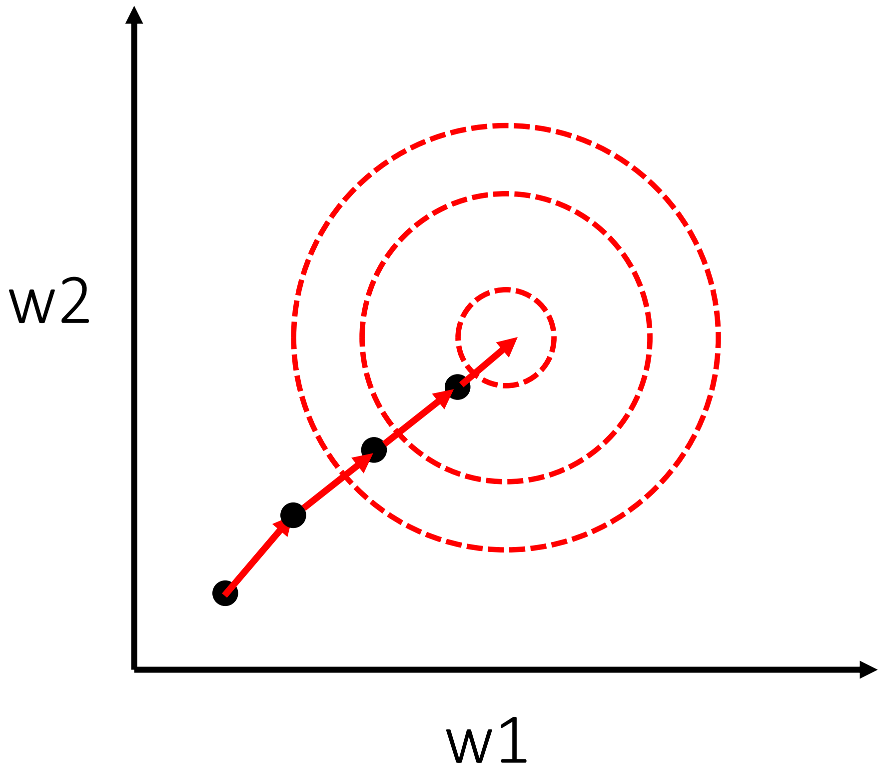


Batch Gradient Descent

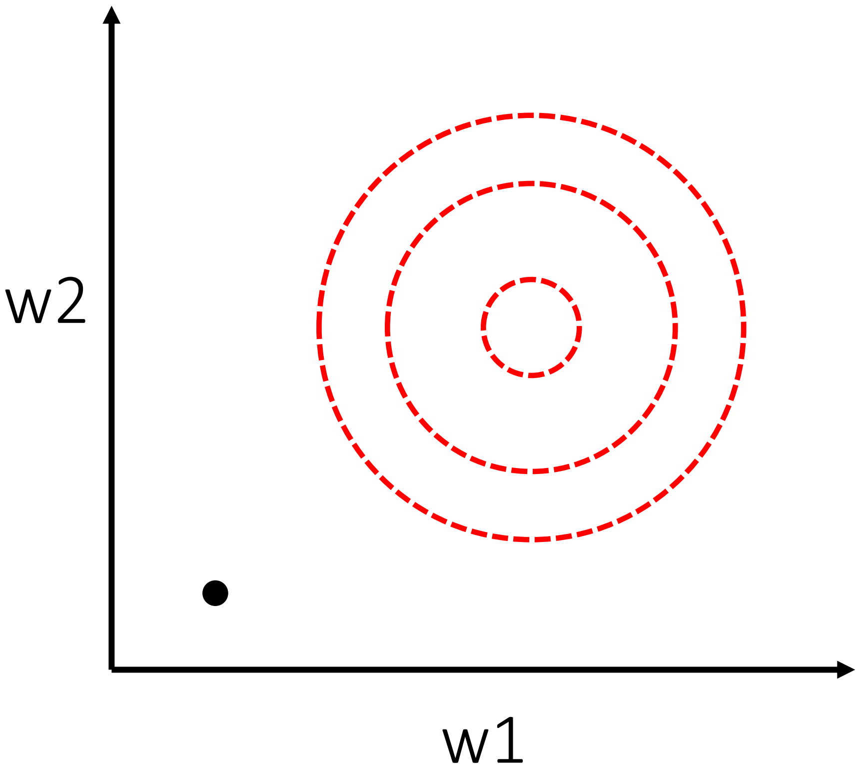


Stochastic Gradient Descent

GD vs SGD

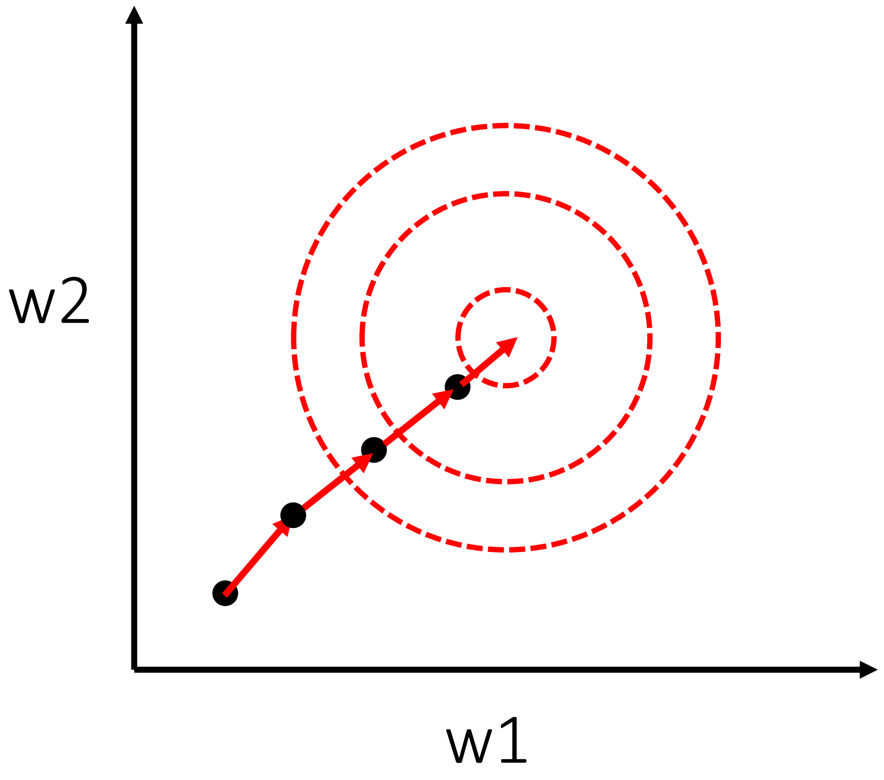


Batch Gradient Descent

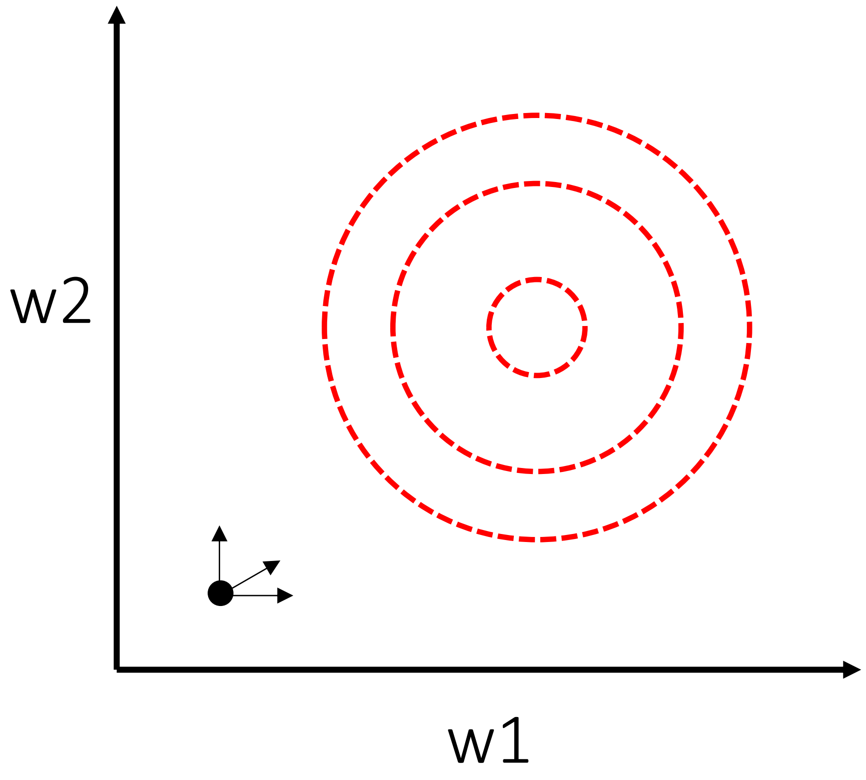


Stochastic Gradient Descent

GD vs SGD

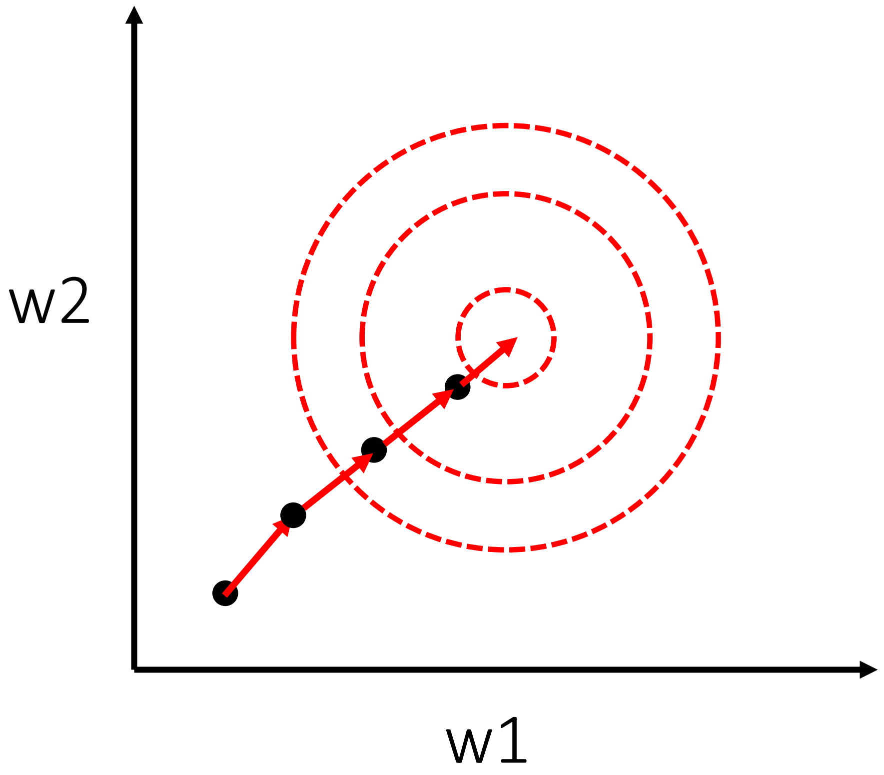


Batch Gradient Descent

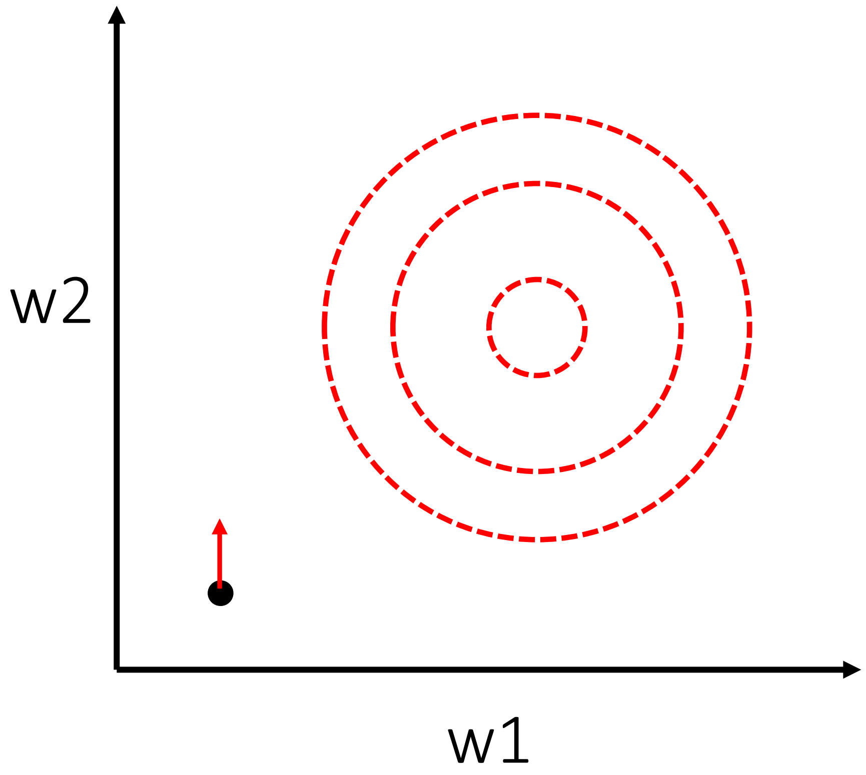


Stochastic Gradient Descent

GD vs SGD

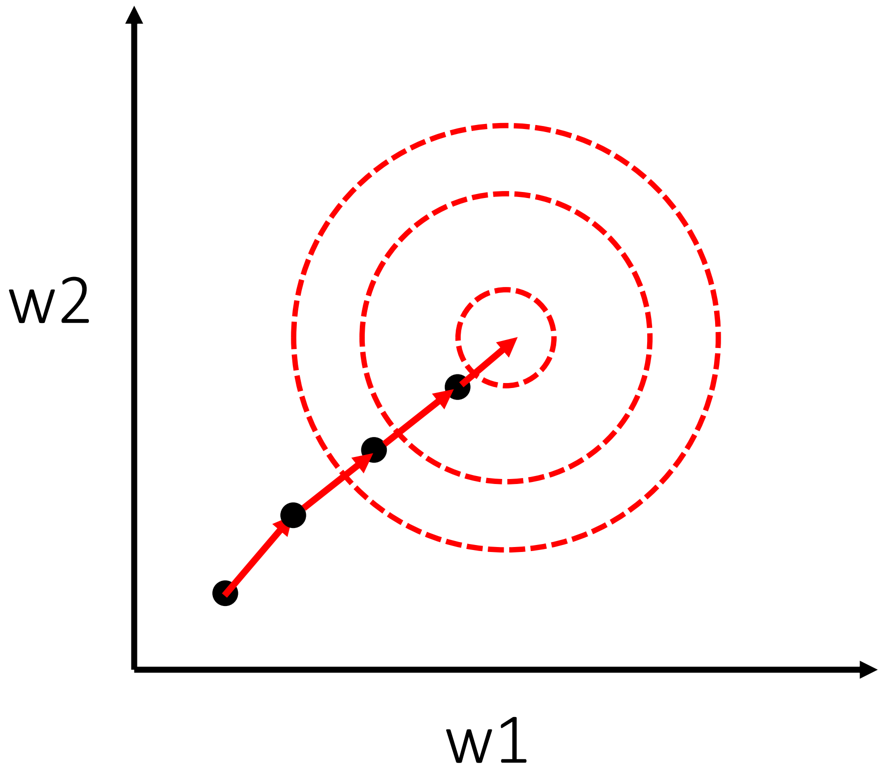


Batch Gradient Descent

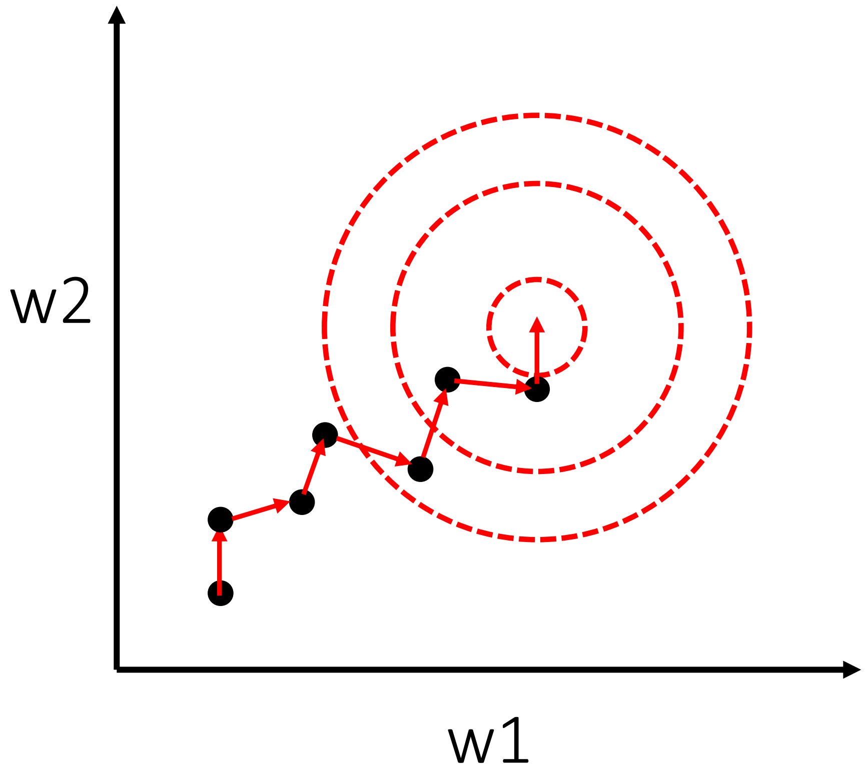


Stochastic Gradient Descent

GD vs SGD

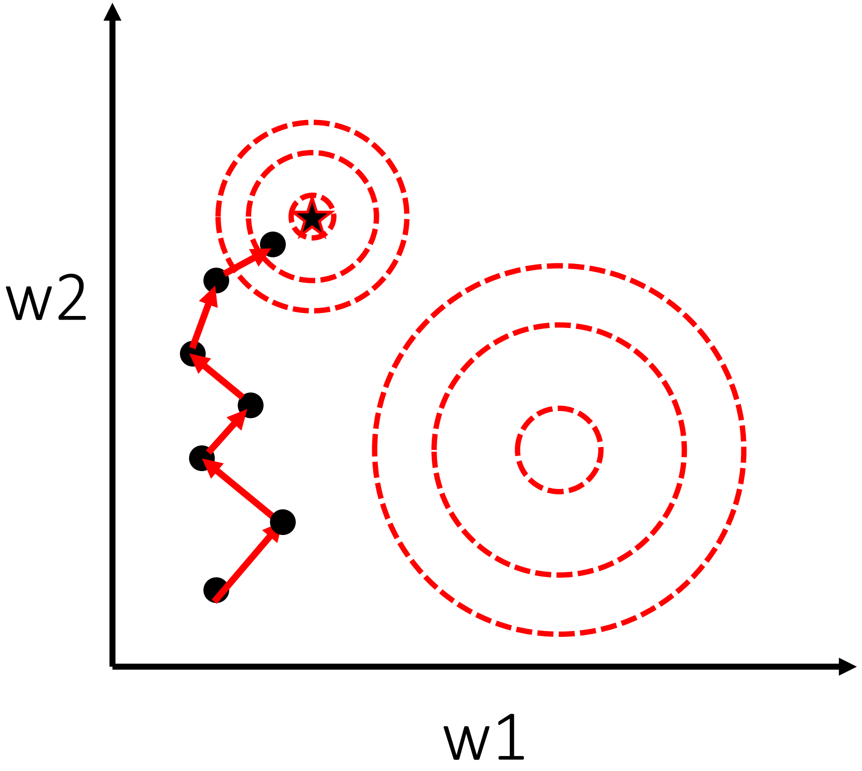


Batch Gradient Descent



Stochastic Gradient Descent

Pros and Cons of SGD

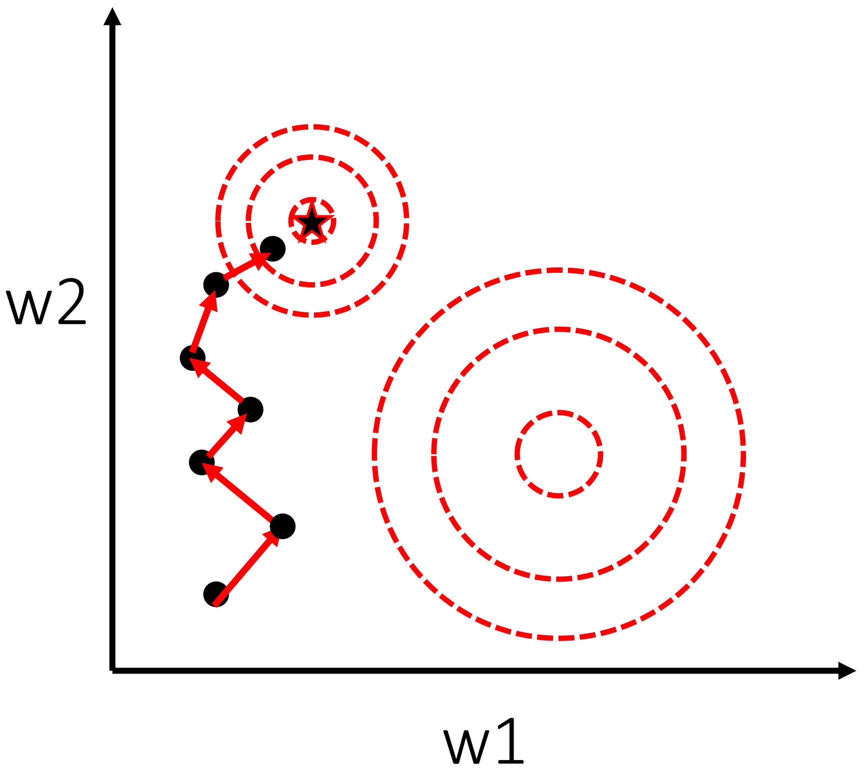


Pros:

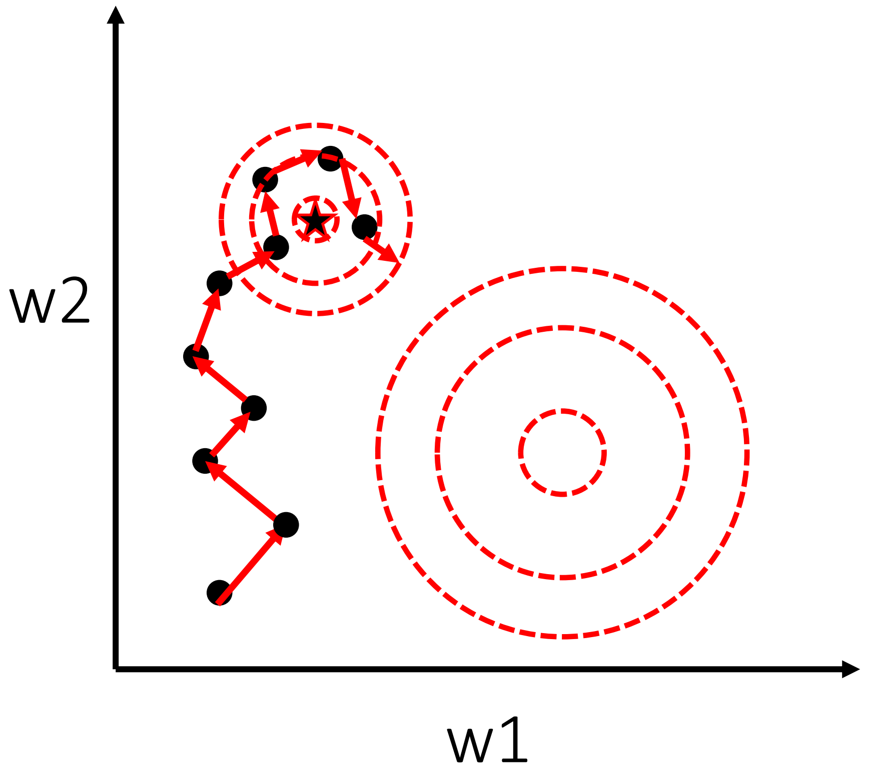
Still consistently converges to minimum

May take shortcut to minimum

Pros and Cons of SGD

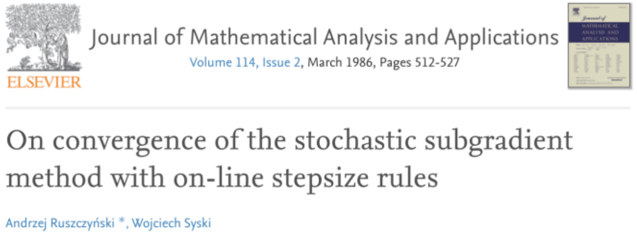


Pros:
Still consistently converges to minimum
May take shortcut to minimum



Cons:
Not useful when we are already close to minimum
Hard to parallelize

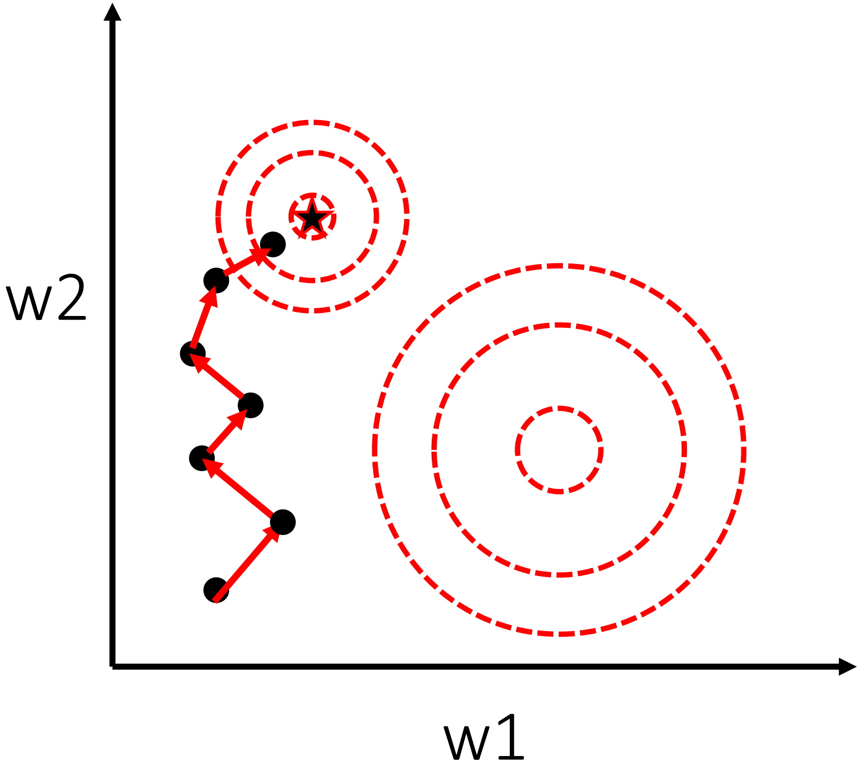
GD vs SGD

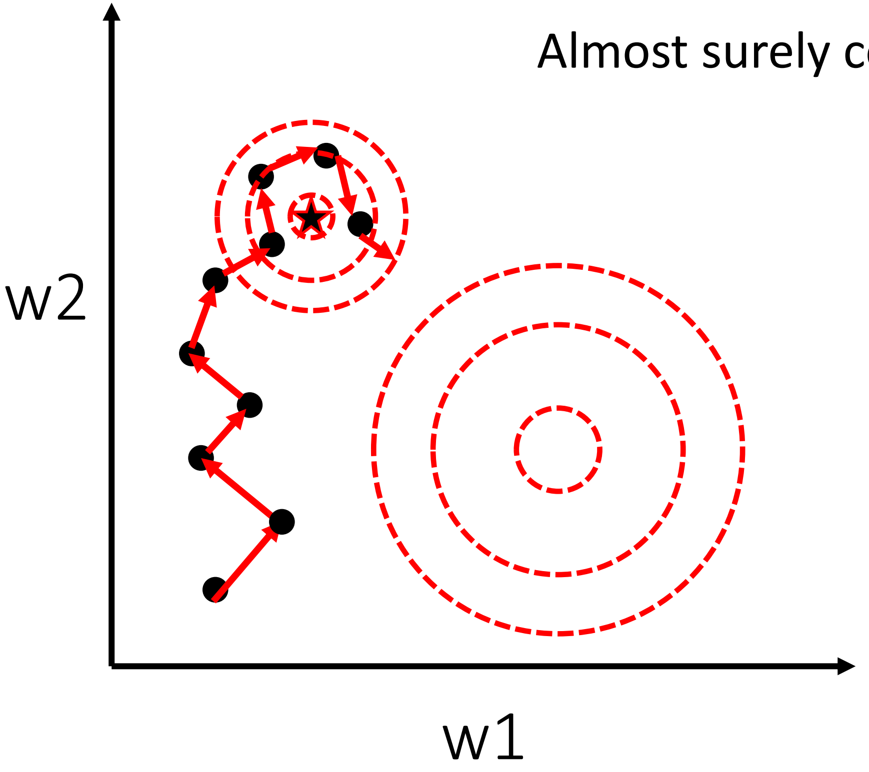

Journal of Mathematical Analysis and Applications
Volume 114, Issue 2, March 1986, Pages 512-527

On convergence of the stochastic subgradient method with on-line stepsize rules

Andrzej Ruszczyński *, Wojciech Syski

Almost surely convergence

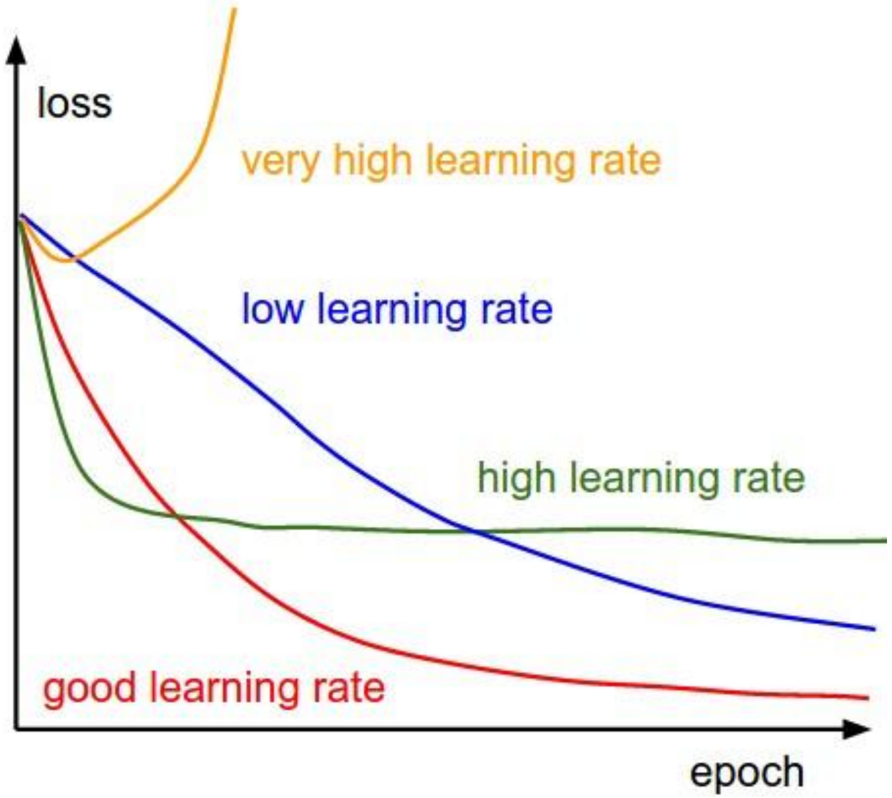




Pros:
Still consistently converges to minimum
May take shortcut to minimum

Cons:
Not useful when we are already close to minimum
Hard to parallelize

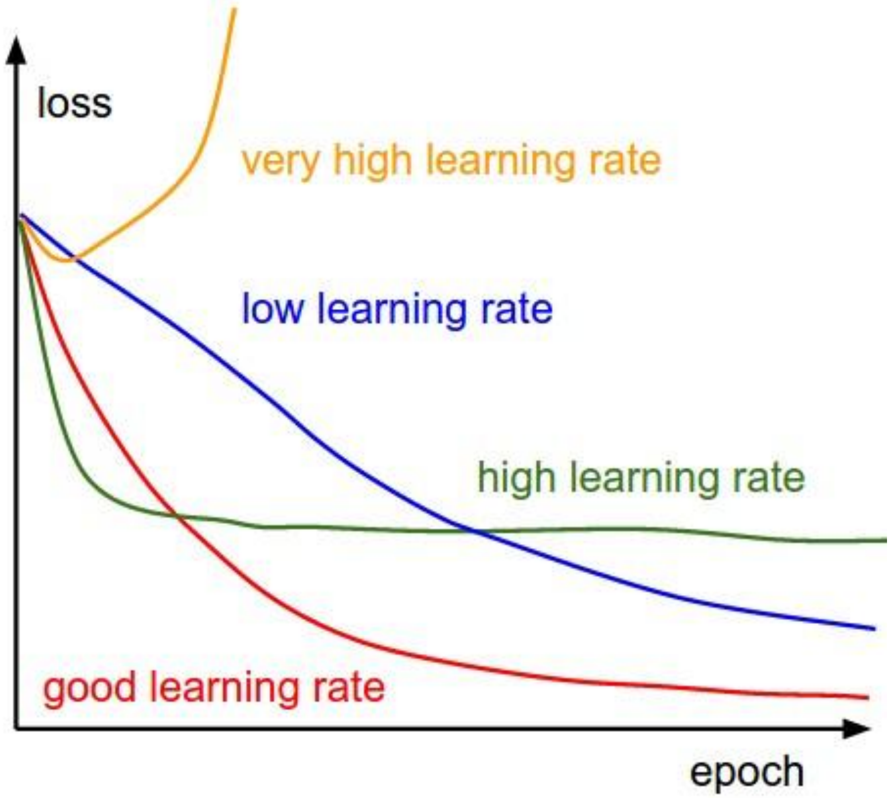
Effects of learning rate (α) on SGD



$$\vec{w}_{k+1} = \vec{w}_k - \boxed{\alpha} \nabla_{\vec{w}} J(\vec{w}_k; b)$$

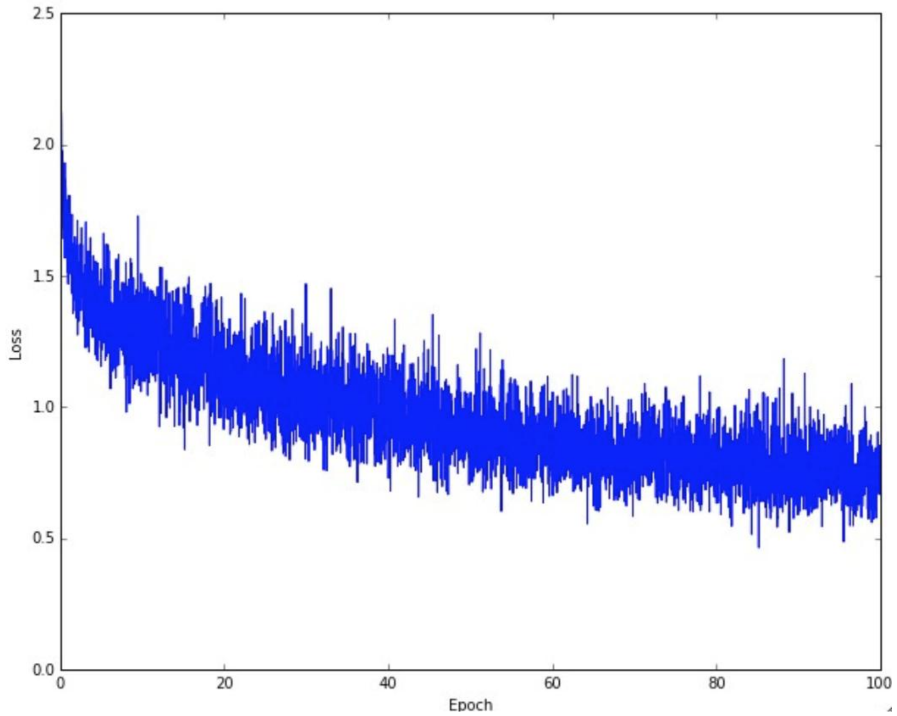
$$b_{k+1} = b_k - \boxed{\alpha} \frac{\partial}{\partial b} J(\vec{w}; b_k)$$

Effects of learning rate (α) on SGD



$$\vec{w}_{k+1} = \vec{w}_k - \boxed{\alpha} \nabla_{\vec{w}} J(\vec{w}_k; b)$$

$$b_{k+1} = b_k - \boxed{\alpha} \frac{\partial}{\partial b} J(\vec{w}; b_k)$$



Loss curve is typically noisy with SGD

Effects of learning rate on SGD

	SGD	Mini-batch GD	Batch GD
Data batch size per iteration	**1**		
	(-) Can loose speedup from oscillations (-) hard to parallelize		

N = Total # of datapoints in training set

m = Number of mini-batches for training set

Effects of learning rate on SGD

	SGD	Mini-batch GD	Batch GD
Data batch size per iteration	**1**	**N/m**	
	(-) Can loose speedup from oscillations (-) hard to parallelize	(+) The whole mini-batch is evaluated in parallel (+) Mostly consistent convergence	

N = Total # of datapoints in training set

m = Number of mini-batches for training set

Effects of learning rate on SGD

	SGD	**Mini-batch GD**	**Batch GD**
Data batch size per iteration	**1**	**N/m**	**n**
	(-) Can loose speedup from oscillations (-) hard to parallelize	(+) The whole mini-batch is evaluated in parallel (+) Mostly consistent convergence	(+) Consistent convergence (+) Maximum parallelization (-) Too long per iteration (-) Hardware memory limit (RAM, VRAM)

N = Total # of datapoints in training set

m = Number of mini-batches for training set

PART 4:

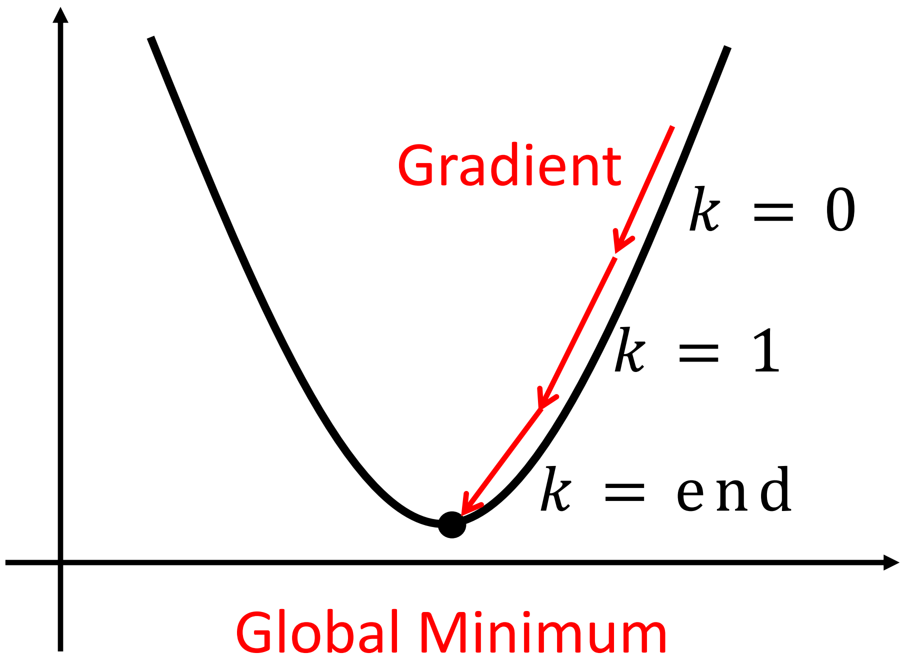
Optimization Techniques in Deep Learning

Variable Learning Rates

$$\vec{w}_{k+1} = \vec{w}_k - \alpha \nabla_{\vec{w}} J(\vec{w}_k; b)$$

$$b_{k+1} = b_k - \alpha \frac{\partial}{\partial b} J(\vec{w}; b_k)$$

$J = L((\vec{w}, b), y)$

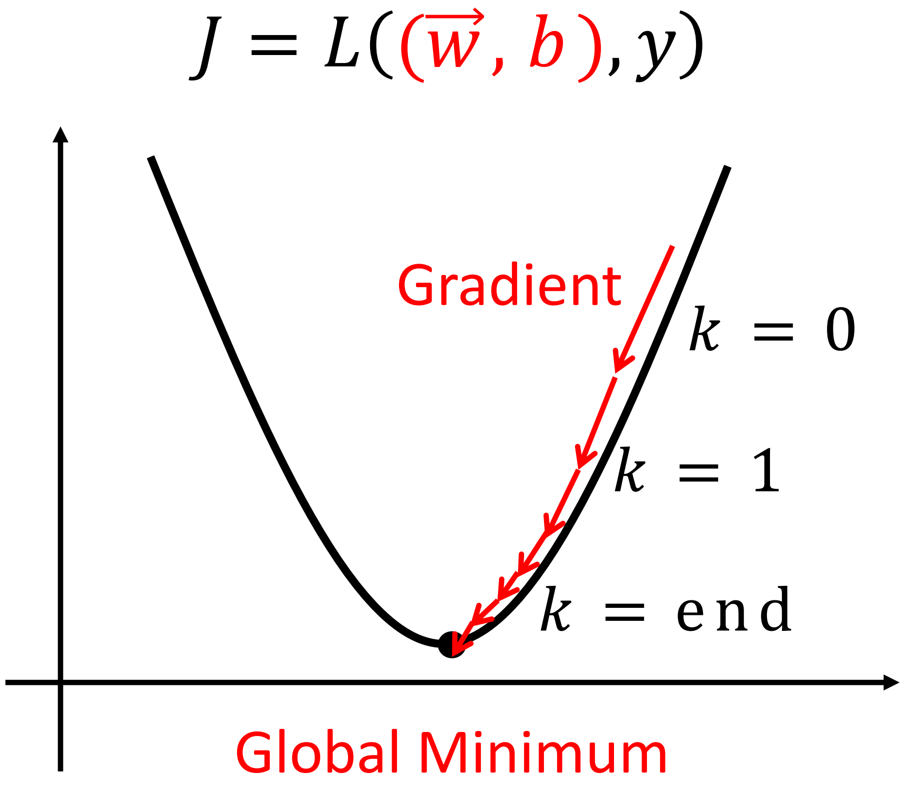


Variable Learning Rates

$$\vec{w}_{k+1} = \vec{w}_k - \alpha \nabla_{\vec{w}} J(\vec{w}_k; b)$$

$$b_{k+1} = b_k - \alpha \frac{\partial}{\partial b} J(\vec{w}; b_k)$$

$$\alpha = f(hp_1, hp_2, \dots)$$



Variable Learning Rates

$$\vec{w}_{k+1} = \vec{w}_k - \alpha \nabla_{\vec{w}} J(\vec{w}_k; b)$$

$$b_{k+1} = b_k - \alpha \frac{\partial}{\partial b} J(\vec{w}; b_k)$$

$$\alpha = f(hp_1, hp_2, \ldots)$$

$$\alpha = \frac{1}{1 + decr \cdot epnum} \alpha_0$$

$$\alpha = d^{epnum} \cdot \alpha_0$$

$$\alpha = \frac{d}{\sqrt{epnum}} \cdot \alpha_0$$

Advanced Methods

Momentum

“Accelerate” gradients vectors in the right directions, to lead to faster converging.

AdaGrad

Adagrad uses a different learning rate for every parameter w_j at every step k. It eliminates the need to manually tune the learning rate.

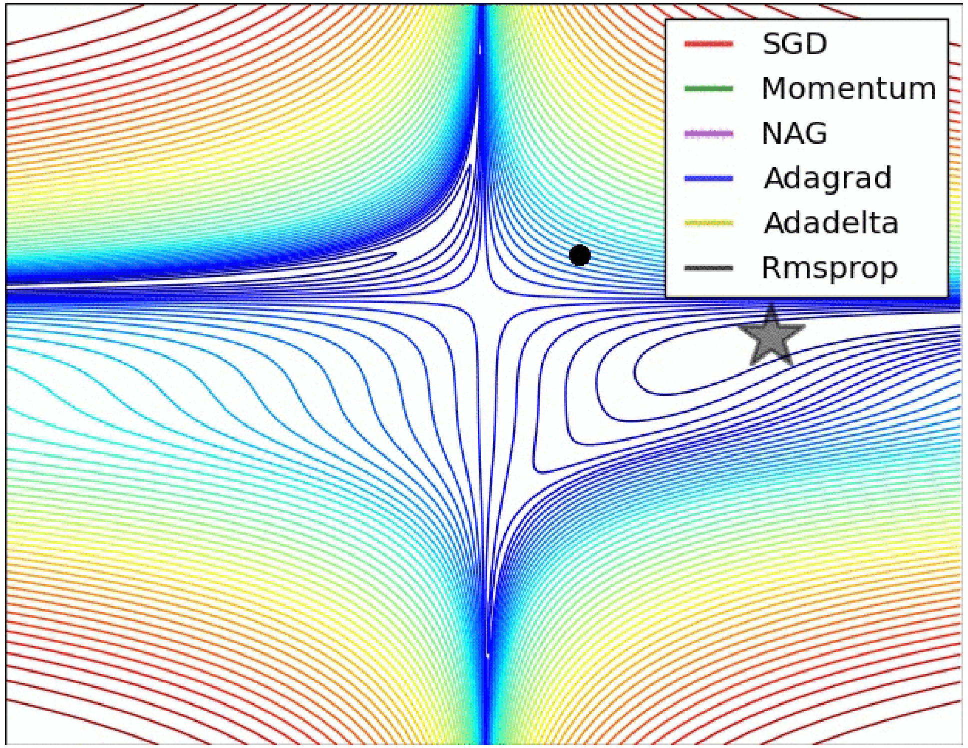
RMSProp

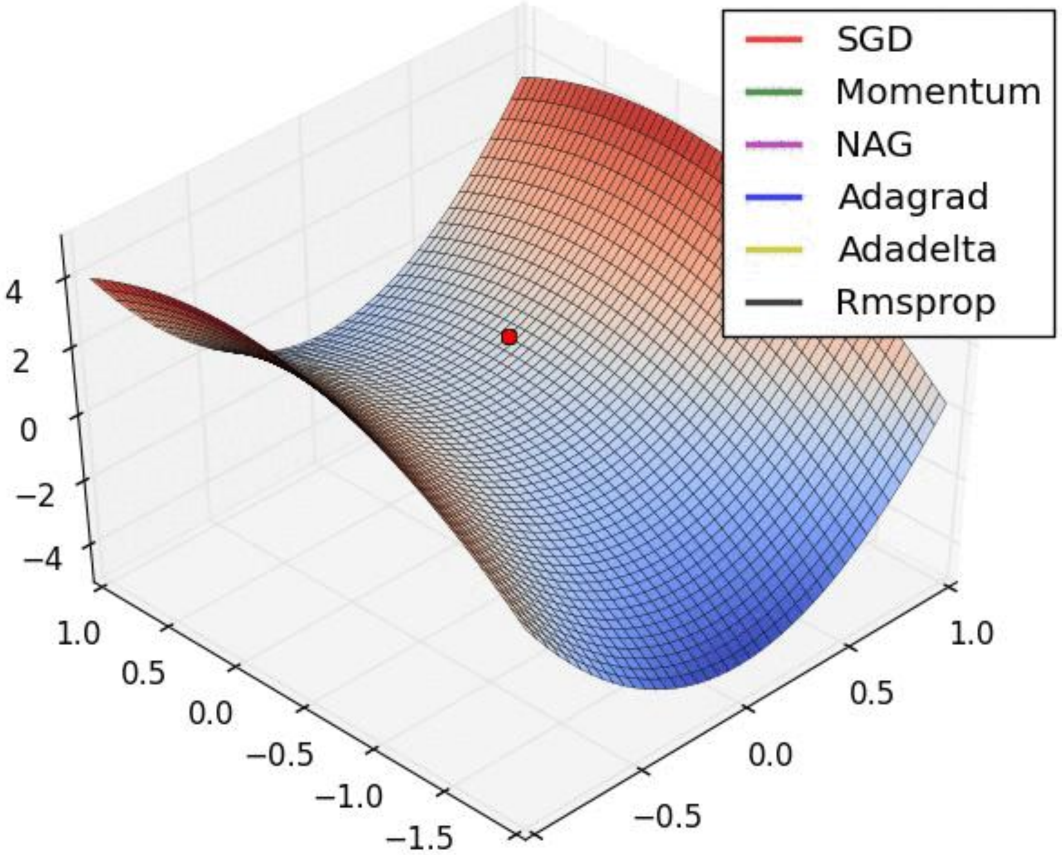
“Extended” and weighted version of AdaGrad via moving average of squared gradients

AdaM

Adaptive learning rate + Momentum

Advanced Methods





Advanced Methods

SGD

Fixed α

First pass

Advanced Methods

SGD
Fixed α

First pass

RSMProp & AdaDelta
adaptive

Worth a try if SGD fails to converge

Adam
adaptive + momentum

Standard optimizer in DL community

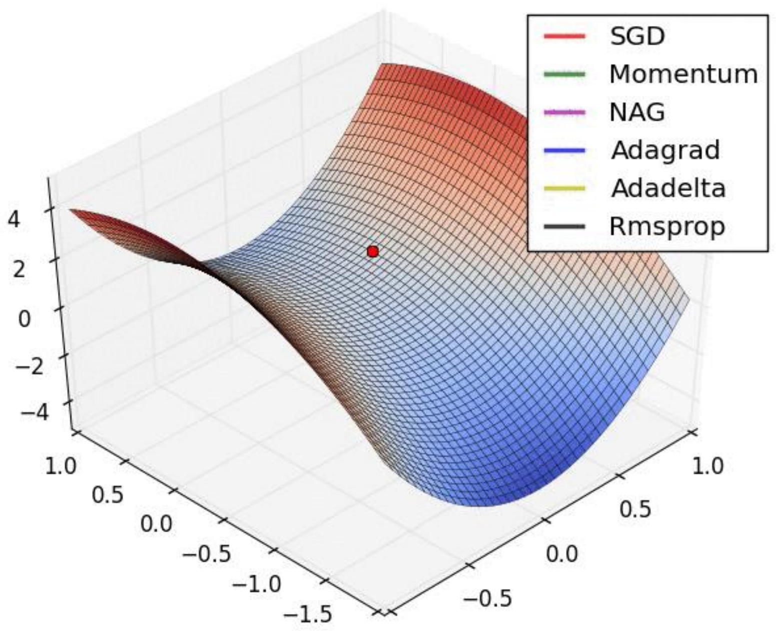
Optimizer vs Optimization Techniques

Optimizers

Optimization Techniques

Optimizer vs Optimization Techniques

Optimizers



Optimization Techniques

Optimizer vs Optimization Techniques

Optimizers

- Vanilla SGD
- Momentum
- AdaGrad
- RMSProp
- Adam

Optimization Techniques

Optimizer vs Optimization Techniques

Optimizers

- Vanilla SGD
- Momentum
- AdaGrad
- RMSProp
- Adam

Optimization Techniques

Everything else that contributes to optimization

Optimizer vs Optimization Techniques

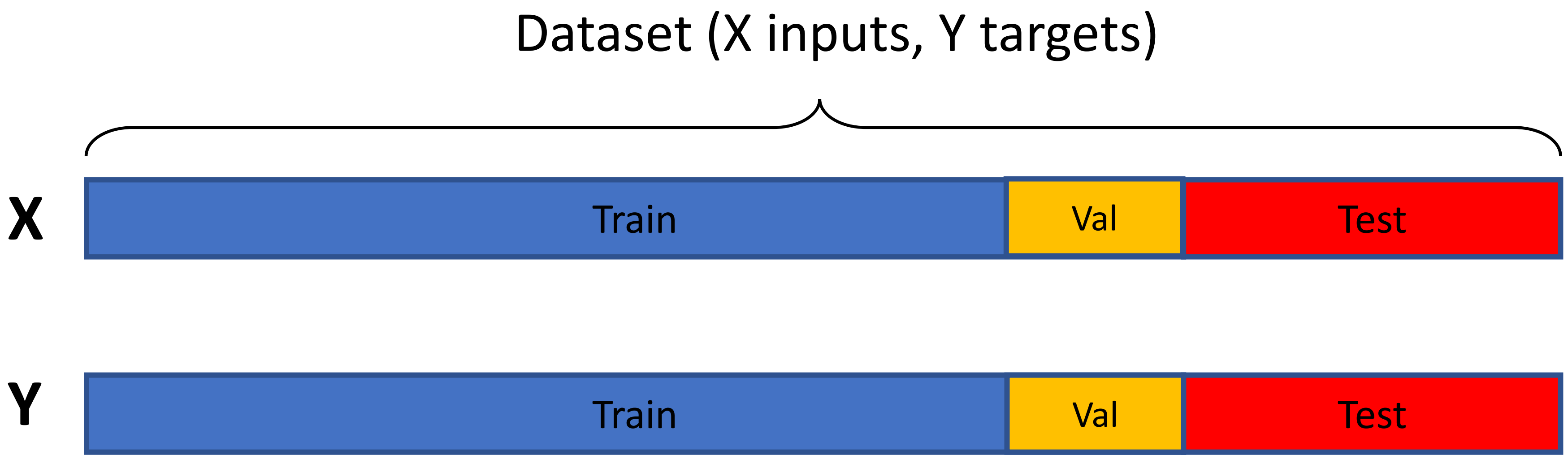
Optimizers

- Vanilla SGD
- Momentum
- AdaGrad
- RMSProp
- Adam

Optimization Techniques

- Data splitting (Train/Val/Test)
- Regularization
- Data normalization
- Batch-normalization
- Network initialization
- Hyperparameter tunings

Cross Validation in Supervised Learning



During training

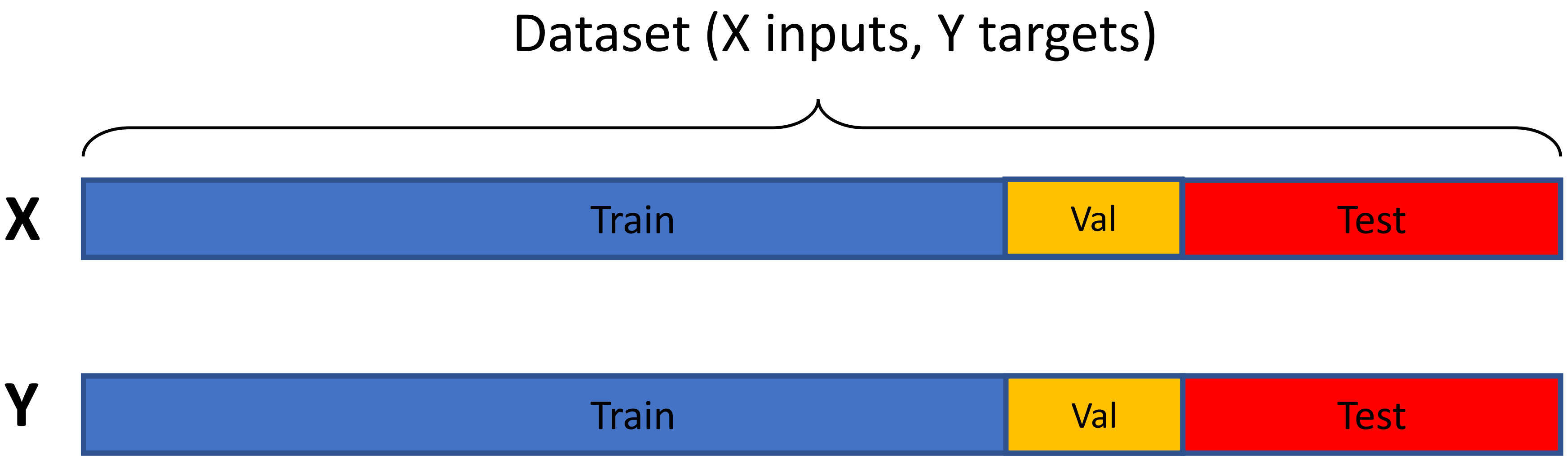
Training loss ↓

Validation performance ↑

After training

Testing performance ↑

Cross Validation in Supervised Learning



Common ratios

70, 15, 15

80, 10, 10

60, 20, 20

⋮

During training

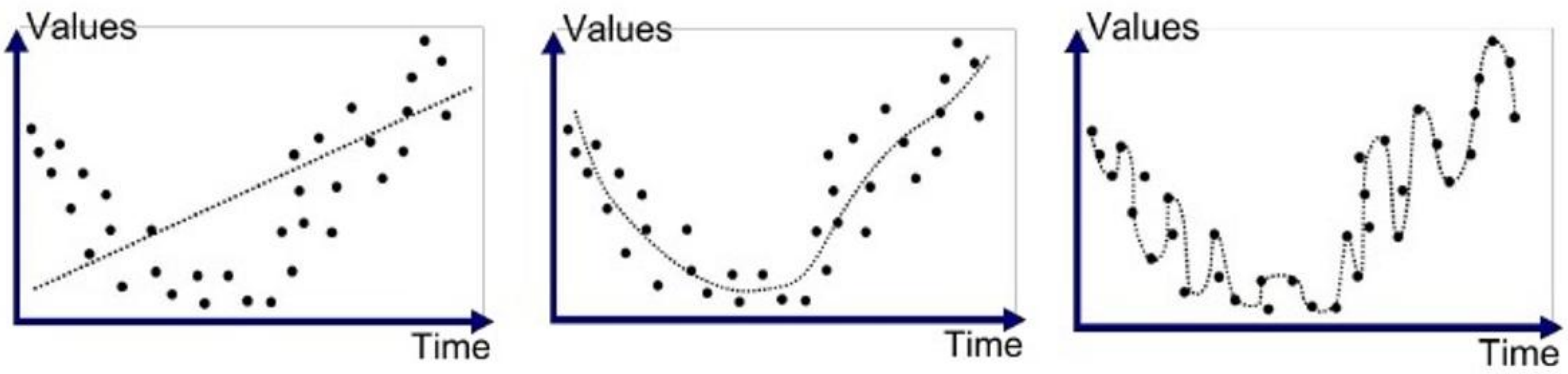
Training loss ↓

Validation performance ↑

After training

Testing performance ↑

Overfitting vs Underfitting

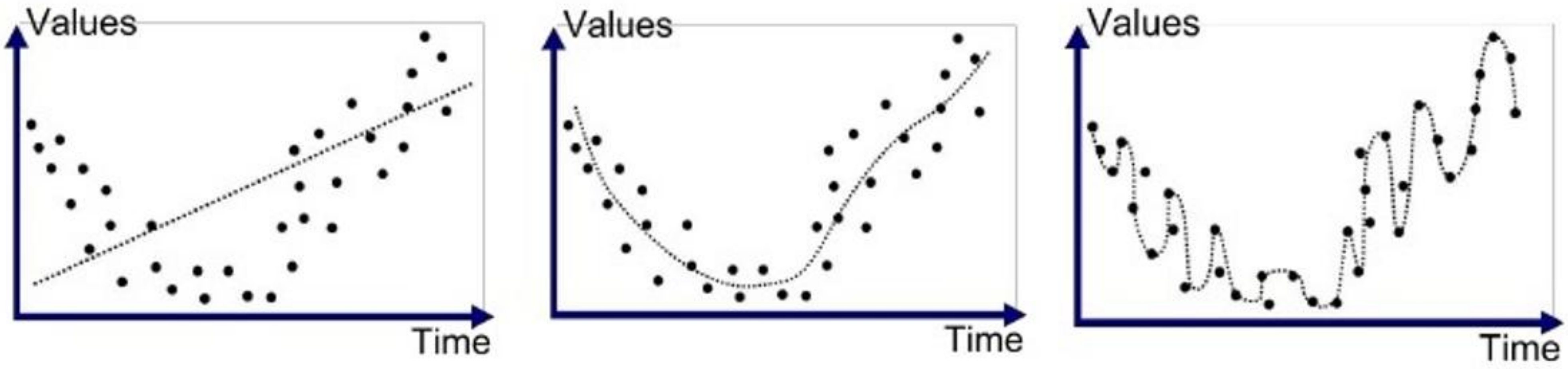


Underfitted

Good Fit/Robust

Overfitted

Overfitting vs Underfitting



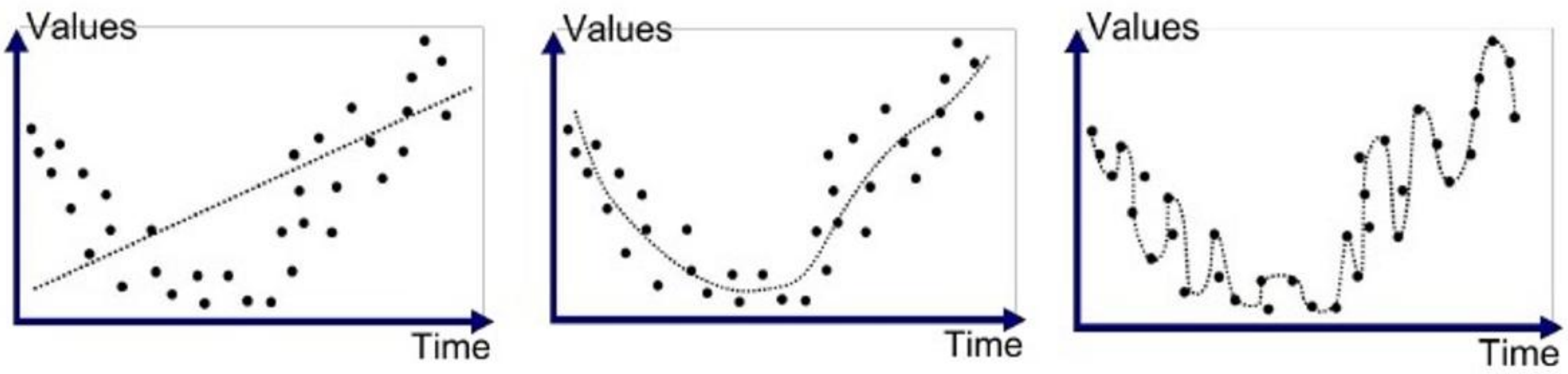
Underfitted | Good Fit/Robust | Overfitted

Bad training accuracy
Bad testing accuracy

Good training accuracy
Good testing accuracy

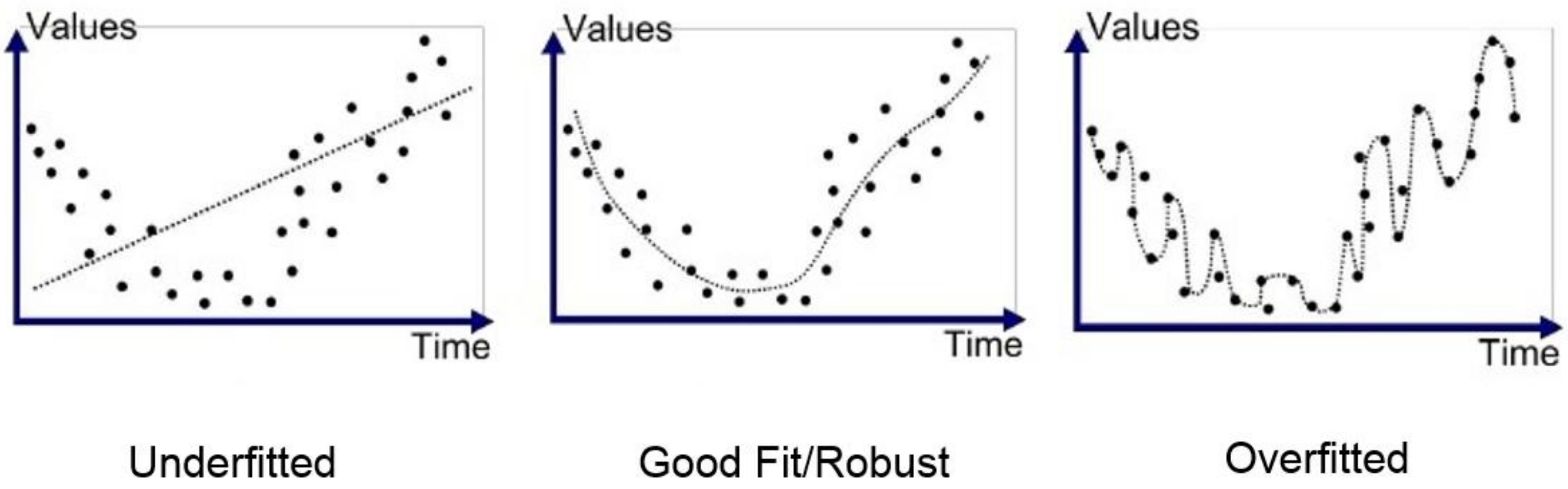
Great training accuracy
Bad testing accuracy

Overfitting vs Underfitting



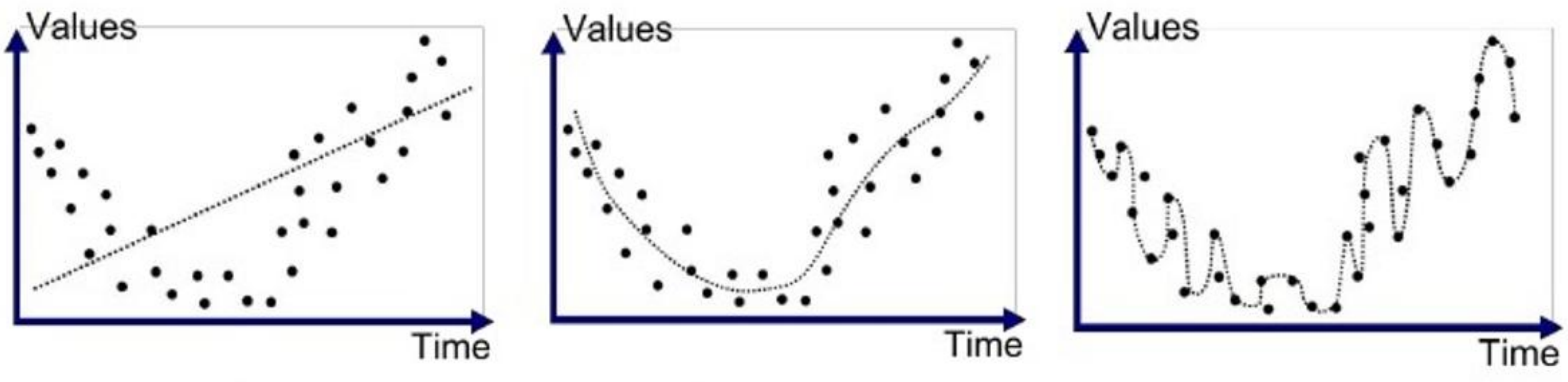
Underfitted	Good Fit/Robust	Overfitted
Bad training accuracy	Good training accuracy	Great training accuracy
Bad testing accuracy	Good testing accuracy	Bad testing accuracy
High Bias		**High Variance**

Remedies for Overfitting/Underfitting



- More Layers/Neurons
- Longer Training
- Architecture
- Hyperparameter tunings

Remedies for Overfitting/Underfitting

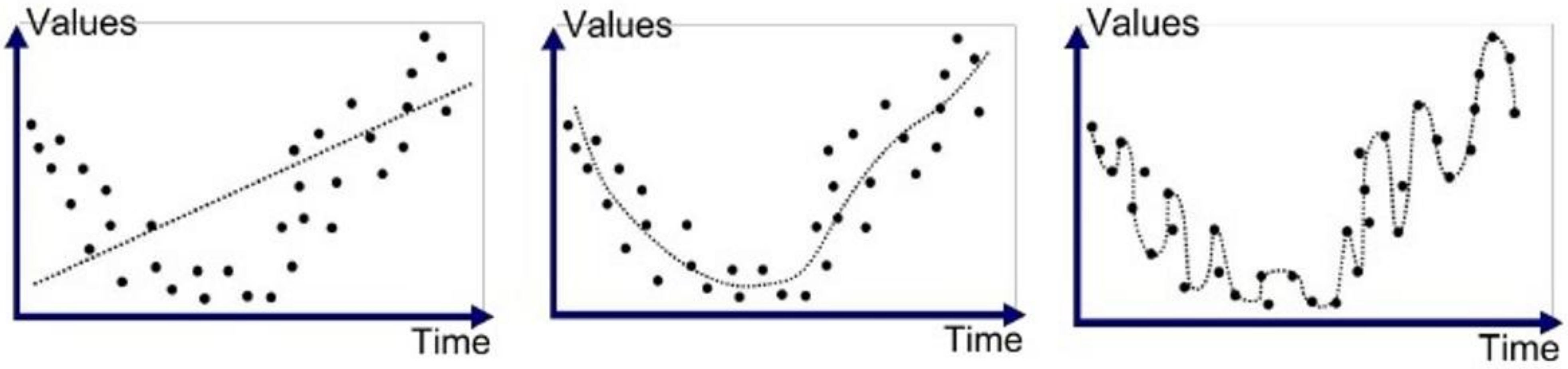


Underfitted   Good Fit/Robust   Overfitted

- More Layers/Neurons
- Longer Training
- Architecture
- Hyperparameter tunings

- More training data
- Regularization
- Dropout
- Initialization

Remedies for Overfitting/Underfitting



Underfitted | Good Fit/Robust | Overfitted

- More Layers/Neurons
- Longer Training
- Architecture
- Hyperparameter tunings

- More training data
- Regularization
- Dropout
- Initialization

L1, L2 Regularizations

L1 Regularization

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

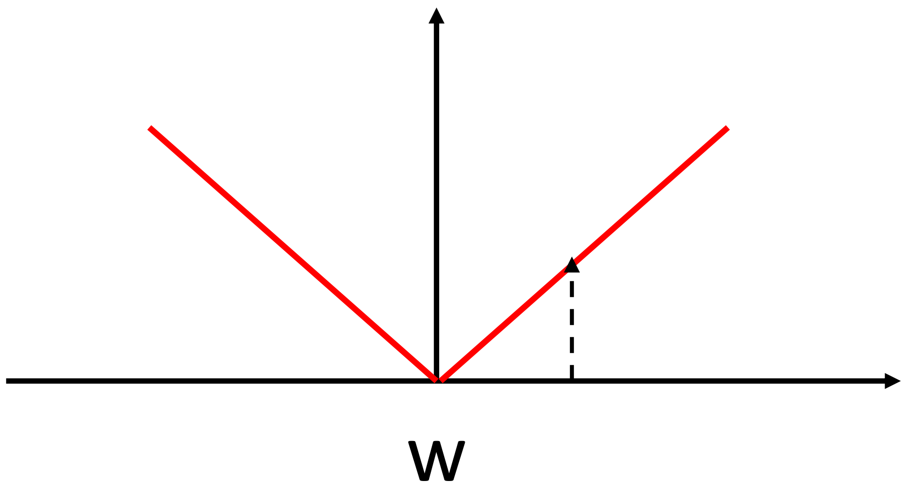
L2 Regularization

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

L1, L2 Regularizations

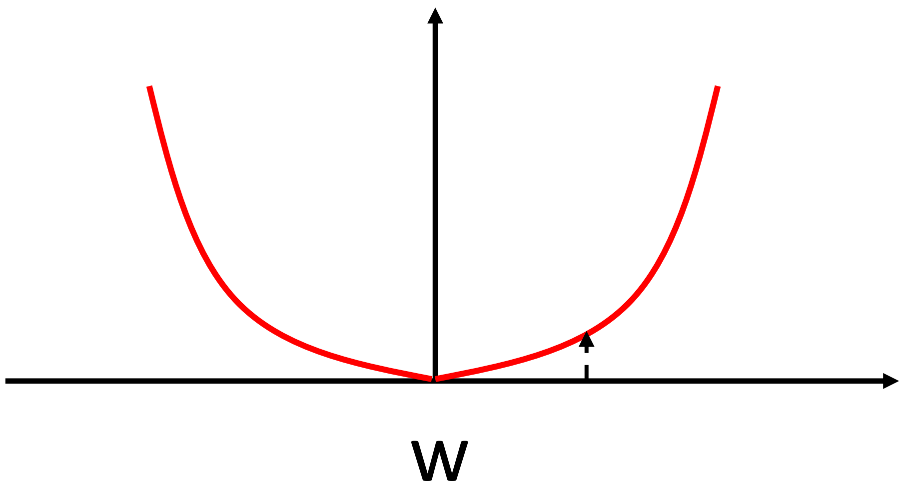
L1 Regularization

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$



L2 Regularization

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$



L1, L2 Regularizations

L1 Regularization

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

Penalizes sum of absolute values of weights

Results in a sparse model

Not suitable for learning complex patterns

Robust to outliers

L2 Regularization

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

Penalizes sum of squared values of weights

Results in a dense model

Learns complex patterns

Sensitive to outliers

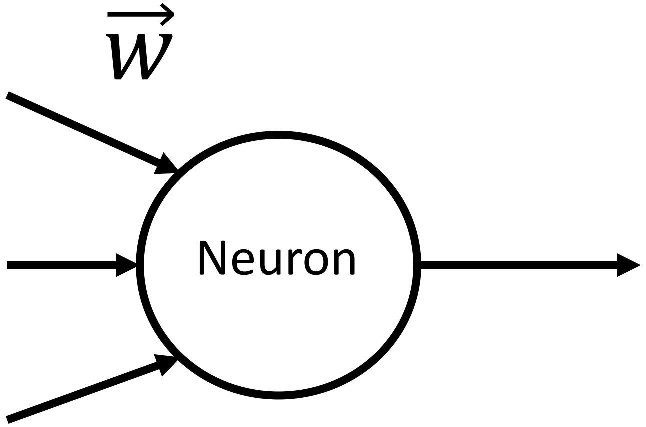
Single-neuron Regularization

$$J(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}||\vec{w}||_2^2$$

$$+\frac{\lambda}{m}||\vec{w}||_1$$

Cost function

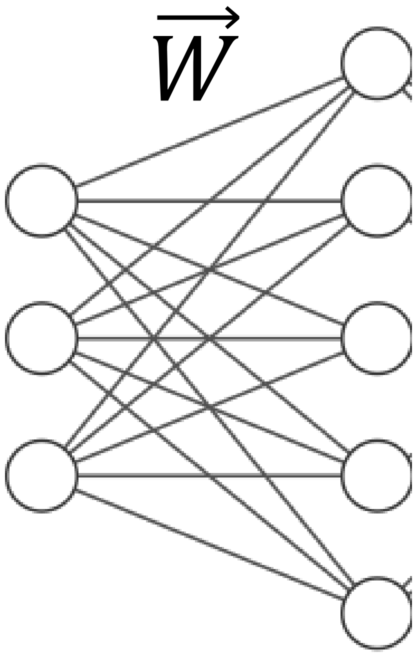
Weight regularization terms



Multi-neuron Regularization

$$J(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} ||W^{[l]}||_F^2$$

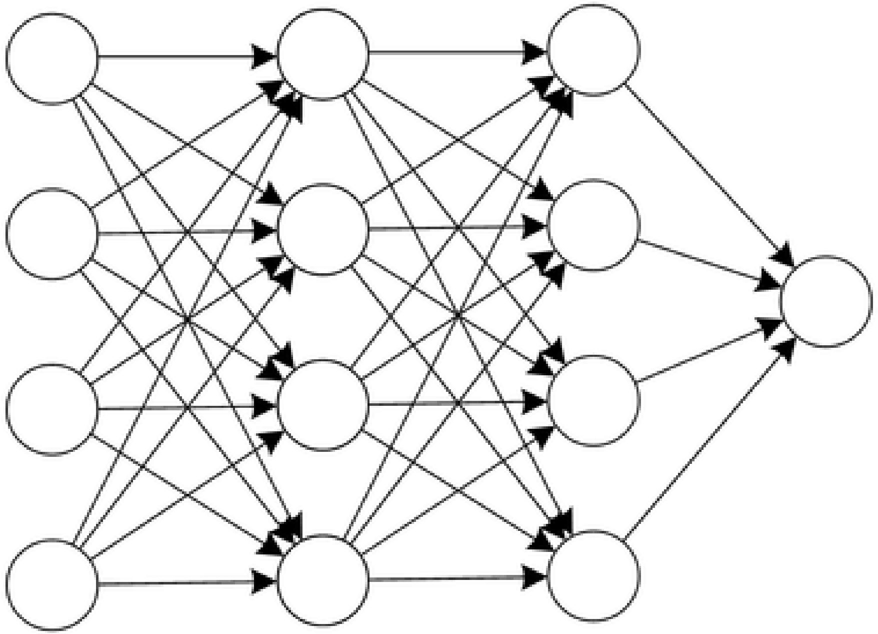
Cost function



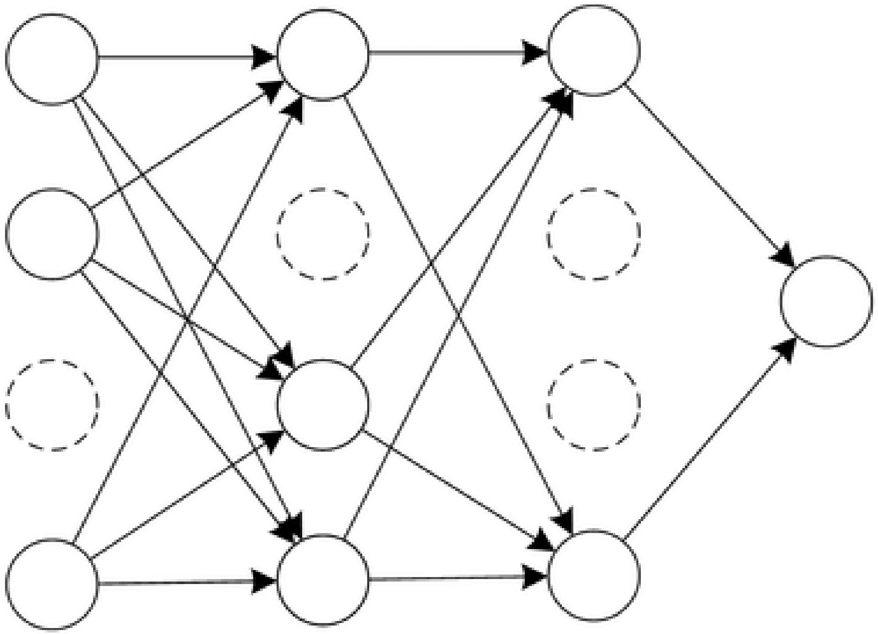
$$||W^{[l]}||_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$$

Weight regularization term over multiple layer (Frobenius norm)

Dropout Regularization

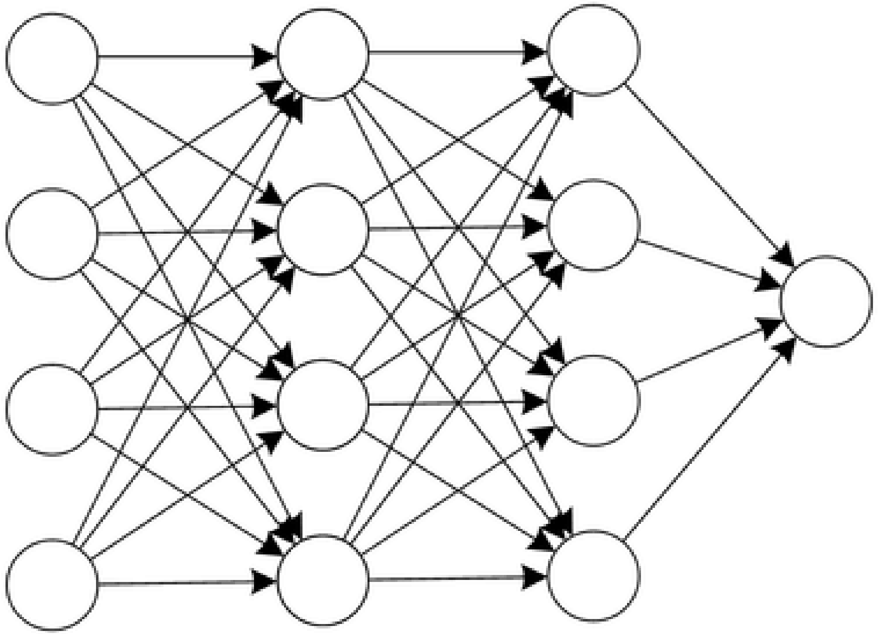


Standard Neural Network

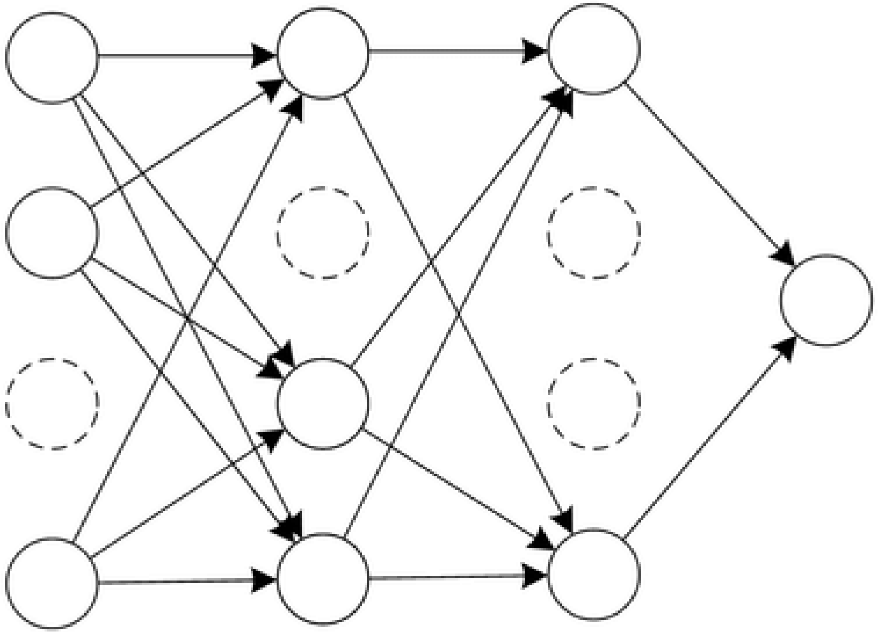


Network with Dropout

Dropout Regularization



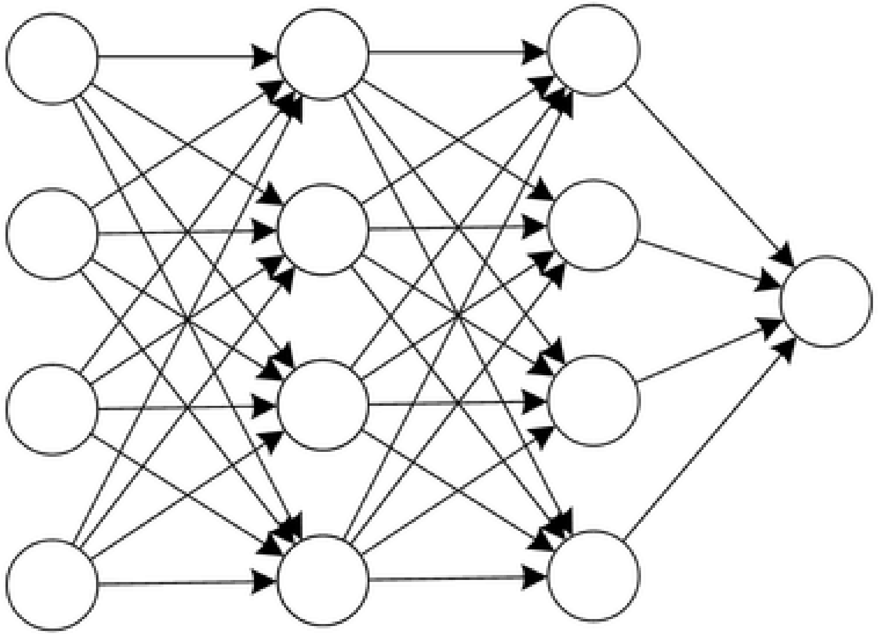
Standard Neural Network



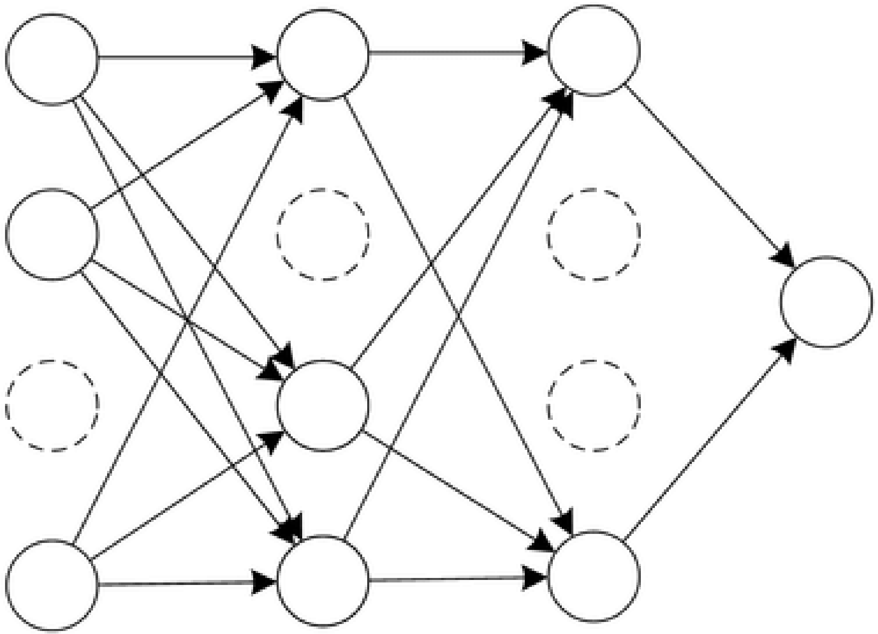
Network with Dropout

Dropout forces the network to learn more robust features + different random subsets of other neurons

Dropout Regularization



Standard Neural Network



Network with Dropout

- Effectively spreading the weights
- Similar to L2 reg
- Testing with dropout p_d=0

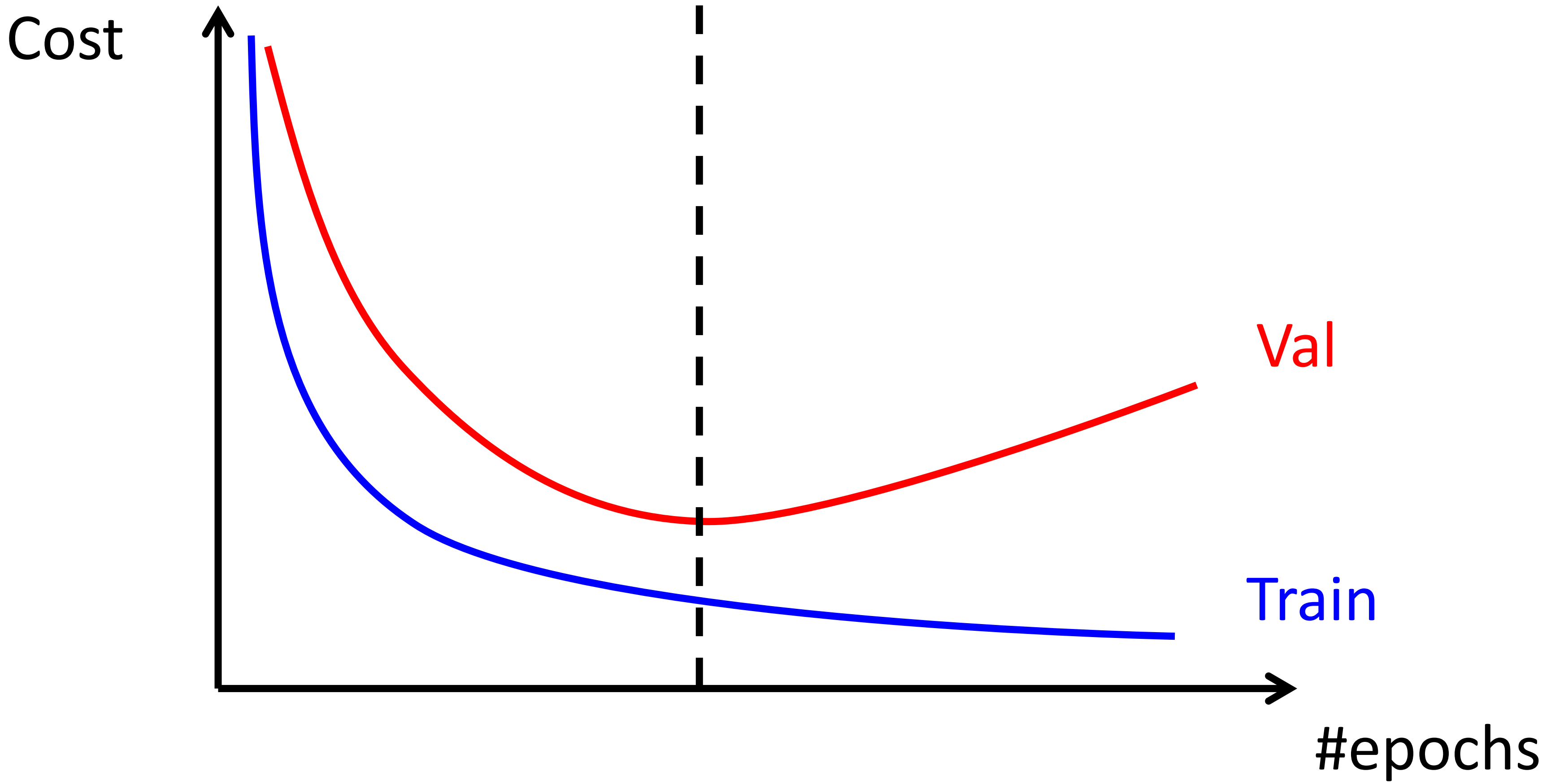
- Can depend on weights (W)
- J could not be well defined in each pass

Data Augmentation

Data Augmentation

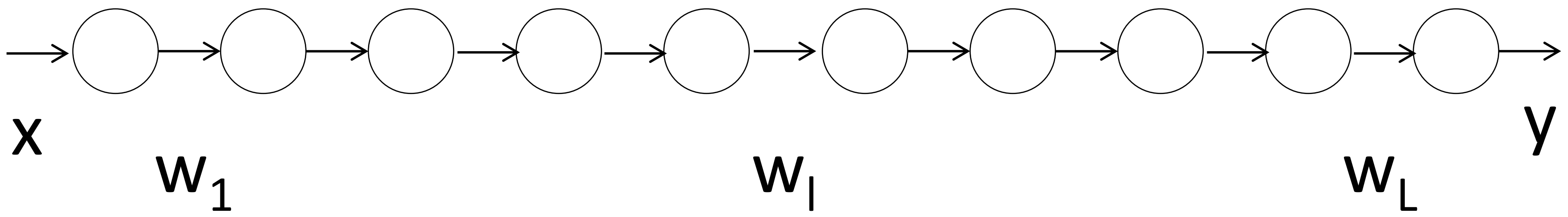
- Add noise
- Distortions
- Synthetic images
- Resize resolutions
- Rotation
- Add symmetries

Early Stopping



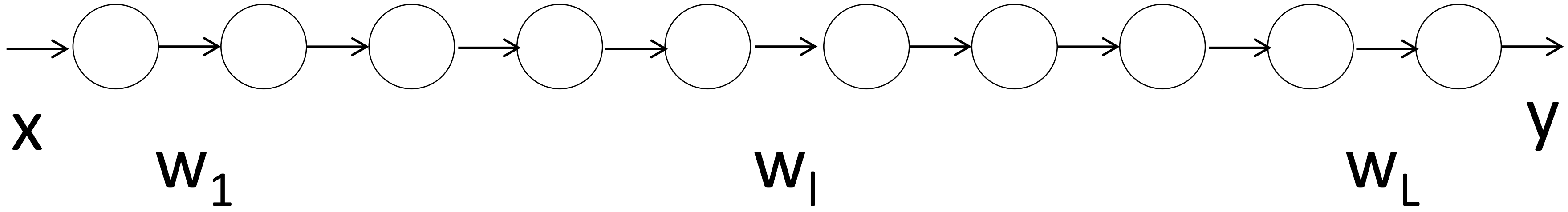
Exploding/Vanishing Gradients

Very deep neural network



Exploding/Vanishing Gradients

Very deep neural network



$$\hat{y} = w_L \cdot \ldots w_l \cdot \ldots w_2 \cdot w_1 \cdot x$$

$$w_l = w > 1; \qquad \hat{y} = w^L x \to \infty$$

$$w_l = w < 1; \qquad \hat{y} = w^L x \to 0$$

Exploding/Vanishing Gradients

With **activation**:

$$\ldots w_3\sigma_3(w_2\sigma_2(\sigma_1'(w_1x))$$

For **gradients**:

$$\frac{\partial J}{\partial w_1} = \sigma_3'(z_3)w_3\sigma_2'(z_2)w_2\sigma_1'(z_1)x$$

Remedies for exploding/vanishing gradients: Data Normalization

Zero mean:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$x^{(i)\mu} = x^{(i)} - \mu$$

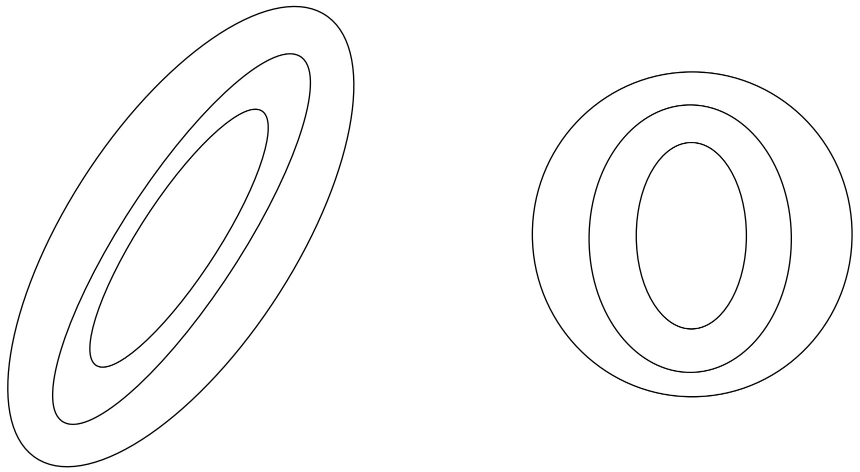
Normalized Variances

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} {x^{(i)}}^2$$

$$x^{(i)\mu,\sigma^2} = x^{(i)\mu} ./ \sigma^2$$

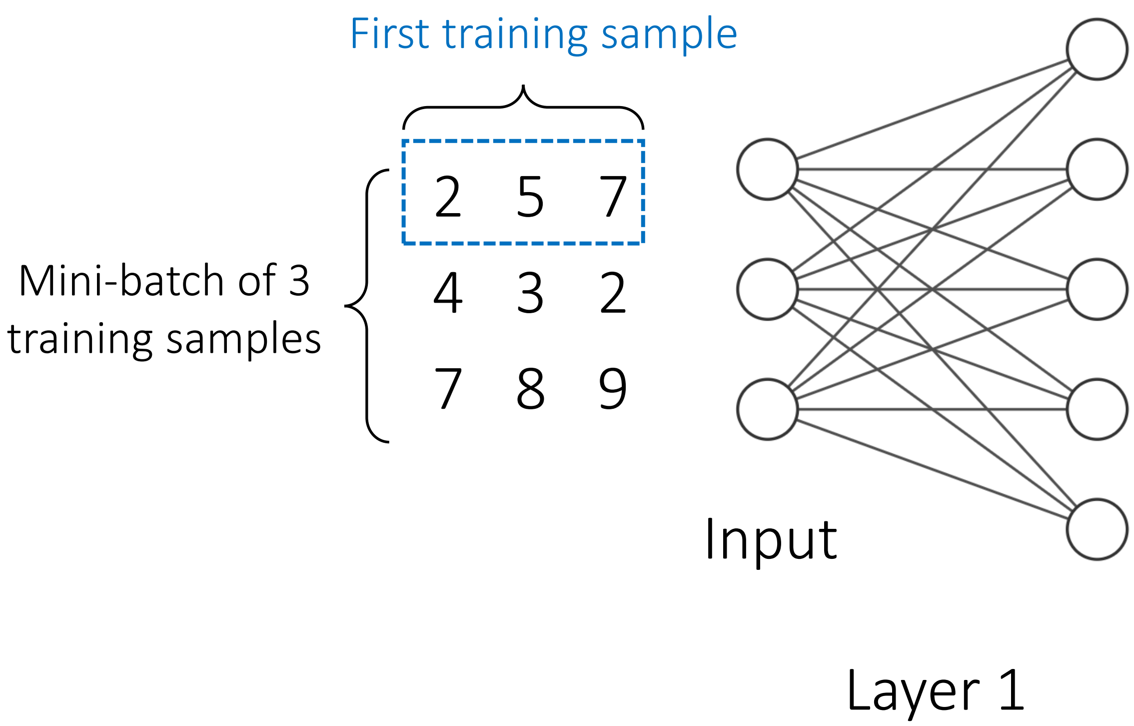
Intuition for data normalization

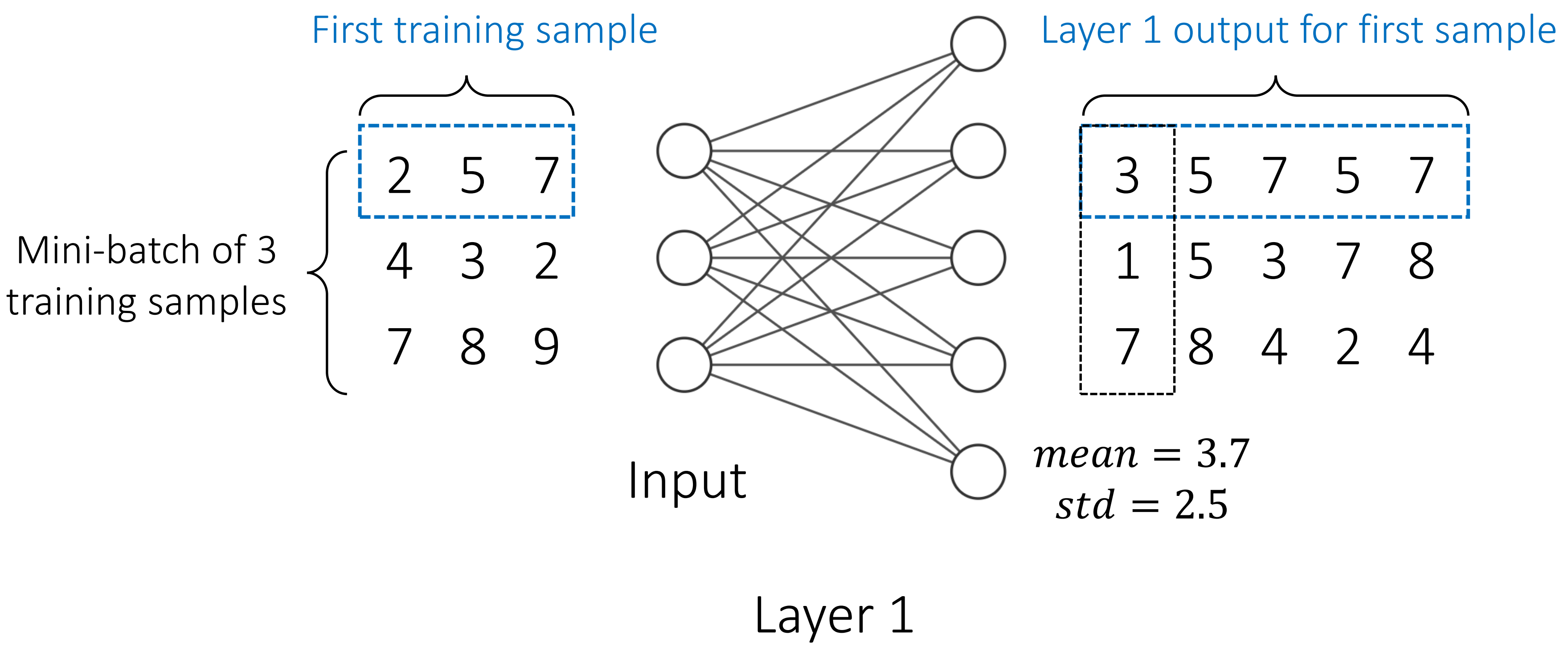
If **inputs have different scales**, the **cost function** will also have to include different scales → increased likelihood of instability



Remember to **normalize all sets**: training, validation, testing

Remedies for Vanishing/Exploding Gradients: Batch Normalization

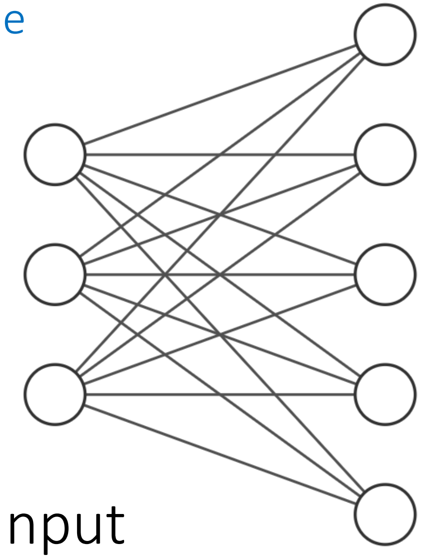



First training sample
Layer 1 output for first sample
Mini-batch of 3 training samples
2 5 7
4 3 2
7 8 9
3 5 7 5 7
1 5 3 7 8
7 8 4 2 4
$mean = 3.7$
$std = 2.5$
Input
Layer 1

First training sample

Mini-batch of 3 training samples

2	5	7
4	3	2
7	8	9



Input

Layer 1

Layer 1 output for first sample

3	5	7	5	7
1	5	3	7	8
7	8	4	2	4

$mean = 3.7$
$std = 2.5$

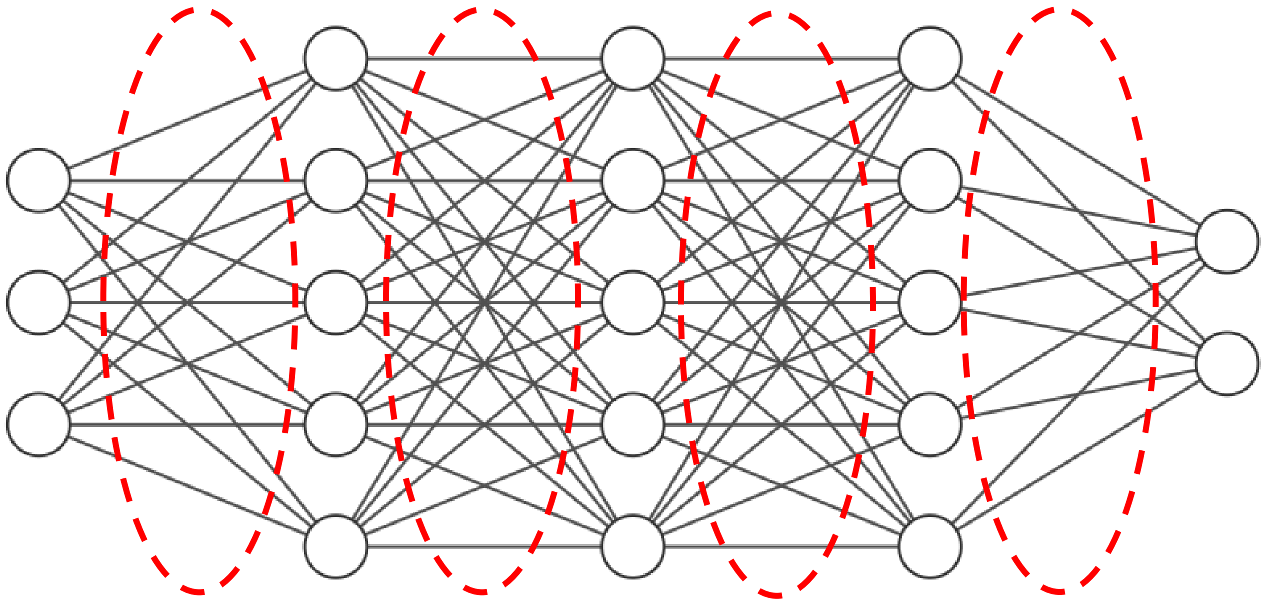
Normalized layer 1 output

-0.3	-0.7	1.4	0.2	0.4
-1.1	-0.7	-1.0	1.1	1.0
1.3	1.4	-0.4	-1.3	-1.4

$mean \approx 0$
$std \approx 1$

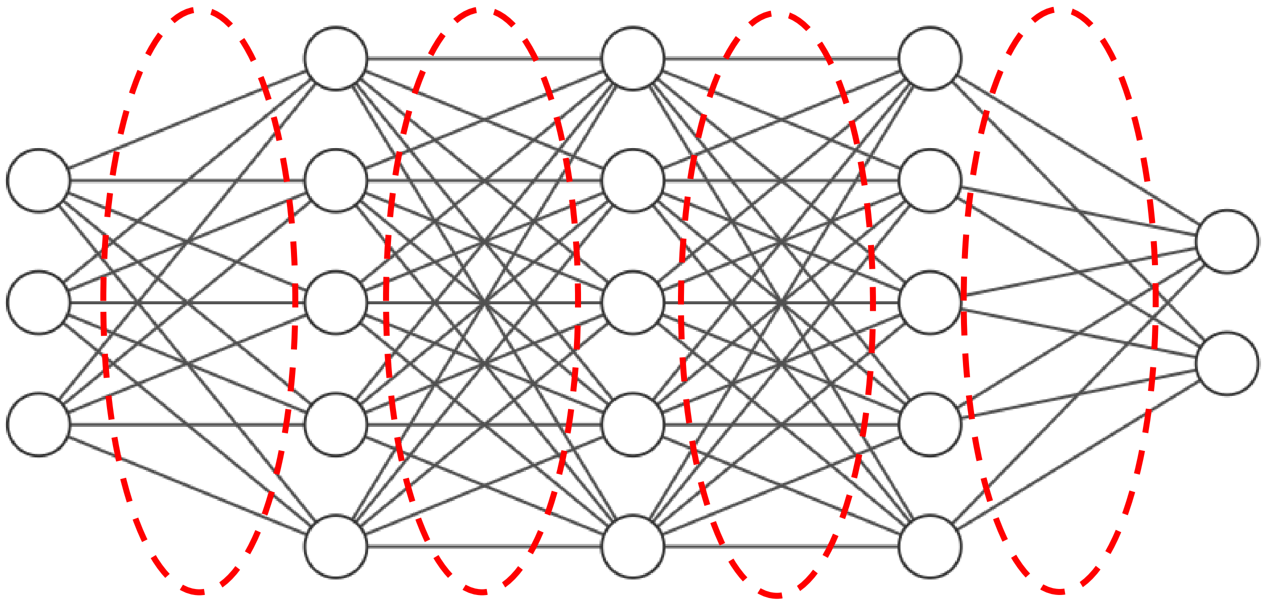
Batch normalization

Remedies for Vanishing/Exploding Gradients: Weight Initialization



Proper weight initialization plays essential roles in preventing exploding/vanishing gradients

Remedies for Vanishing/Exploding Gradients: Weight Initialization



Proper weight initialization plays essential roles in preventing exploding/vanishing gradients

↓

Faster convergence

Network Initialization

- Zero → Problematic
- Random Normal (0,1) -> Problematic
- Xavier (tanh):

$$Var(w^{[l]}) : 1/n^{[l-1]}$$

$$w^{[l]} = N(0,1) \cdot \sqrt{\frac{1}{n^{[l-1]}}}$$

Network Initialization

- He (ReLU):

$$Var(w^{[l]}) : 2/n^{[l-1]}$$

- Other: $$w^{[l]} = N(0,1) \cdot \sqrt{\frac{2}{n^{[l-1]}}}$$

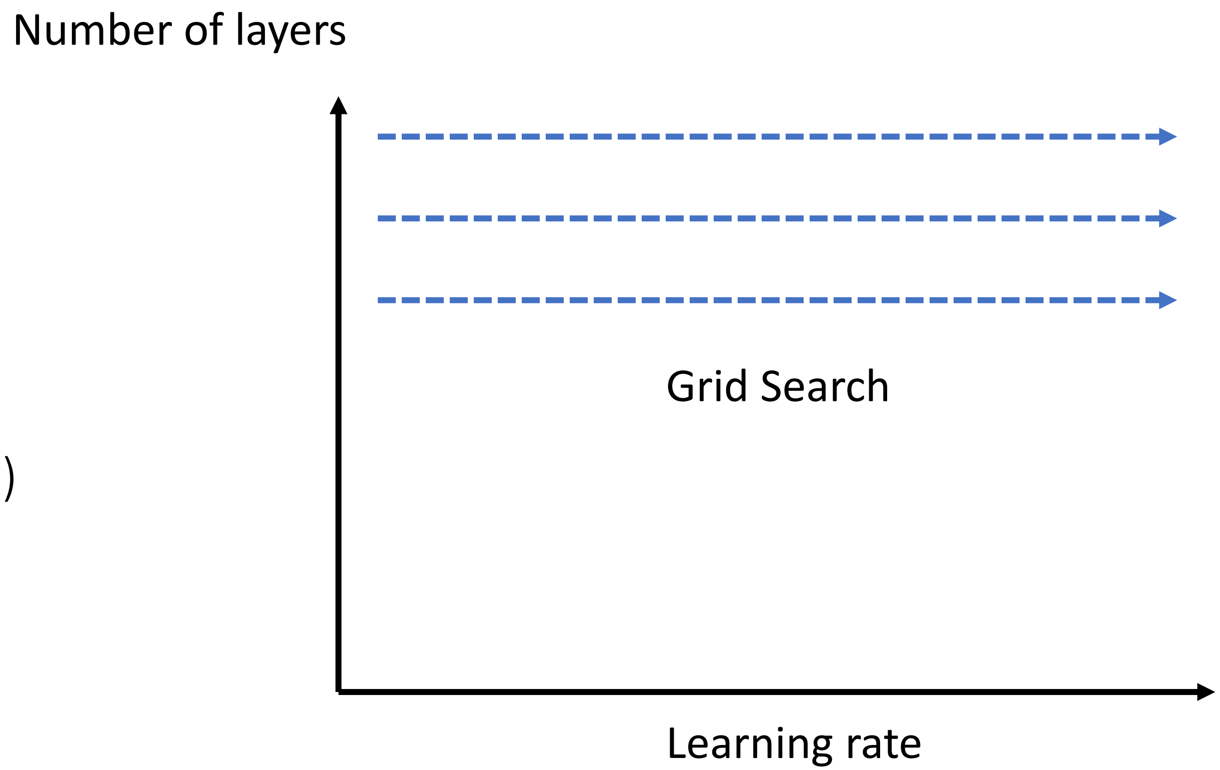
$$Var(w^{[l]}) : \frac{2}{n^{[l-1]} + n^{[l]}}$$

Hyperparameters

- Learning rate
- Number of layers
- Neurons in each layer
- Activation function
  (ReLU, Tanh, sigmoid)
- Training batch size
  (SGD, Mini-batch, Batch Gradient)
- Optimizer
  (SGD, Adam, RMS Prop etc)
- Number of training epochs

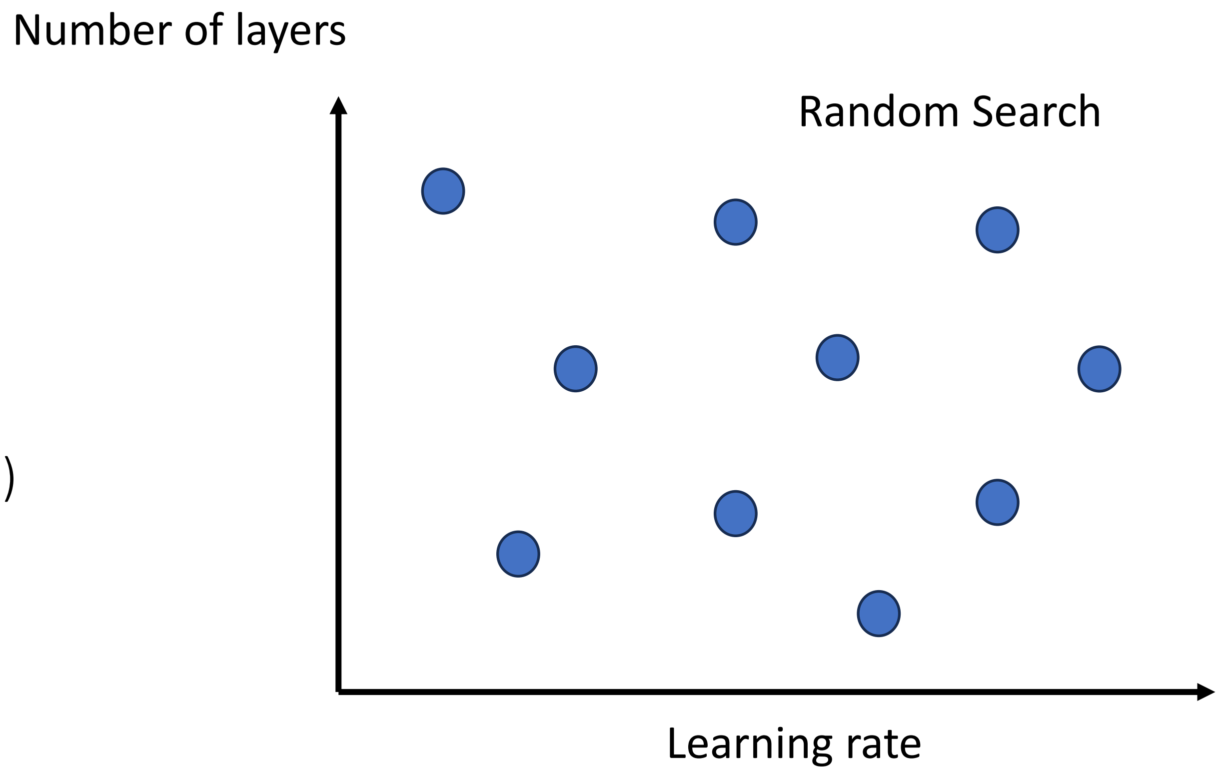
Hyperparameters

- Learning rate
- Number of layers
- Neurons in each layer
- Activation function
  (ReLU, Tanh, sigmoid)
- Training batch size
  (SGD, Mini-batch, Batch Gradient)
- Optimizer
  (SGD, Adam, RMS Prop etc)
- Number of training epochs



Hyperparameters

- Learning rate
- Number of layers
- Neurons in each layer
- Activation function
  (ReLU, Tanh, sigmoid)
- Training batch size
  (SGD, Mini-batch, Batch Gradient)
- Optimizer
  (SGD, Adam, RMS Prop etc)
- Number of training epochs



Summary

Optimizers

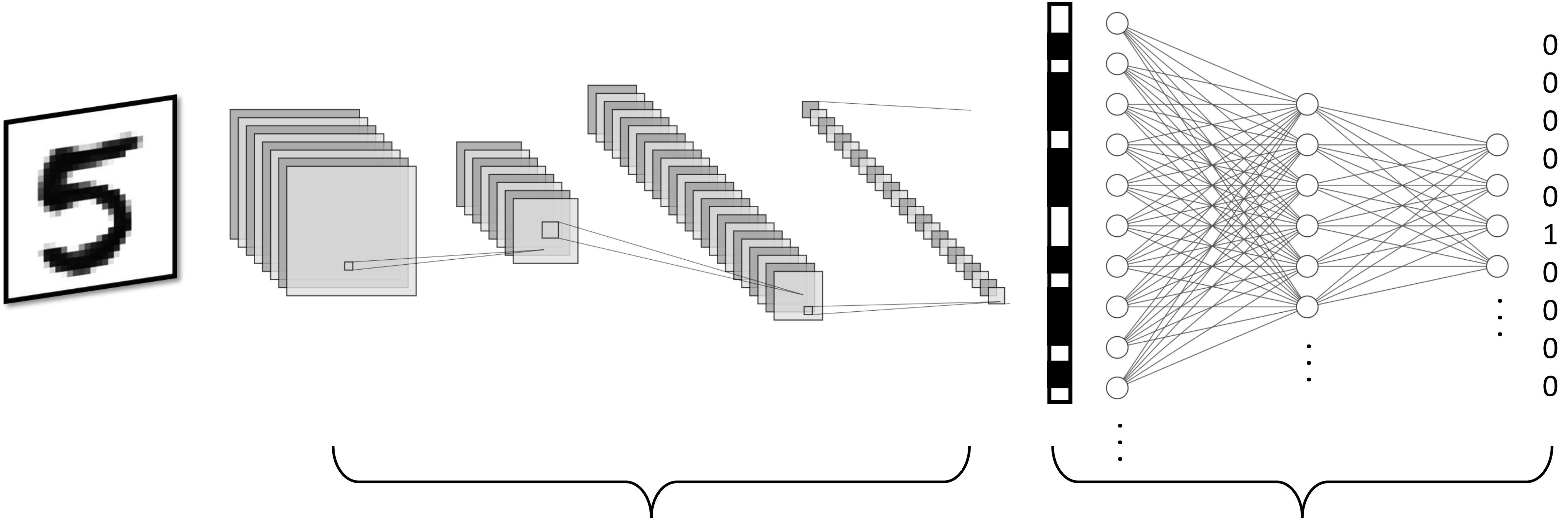
- Vanilla SGD
- Momentum
- AdaGrad
- RMSProp
- Adam

Balance

Optimization Techniques

- Data splitting (Train/Val/Test)
- Regularization
- Data normalization
- Batch-normalization
- Network initialization
- Hyperparameter tunings

Next episode in EEP 596



Convolution Layers + Pooling Layers
(Image feature extraction)

Fully connected layers
(Classifier)