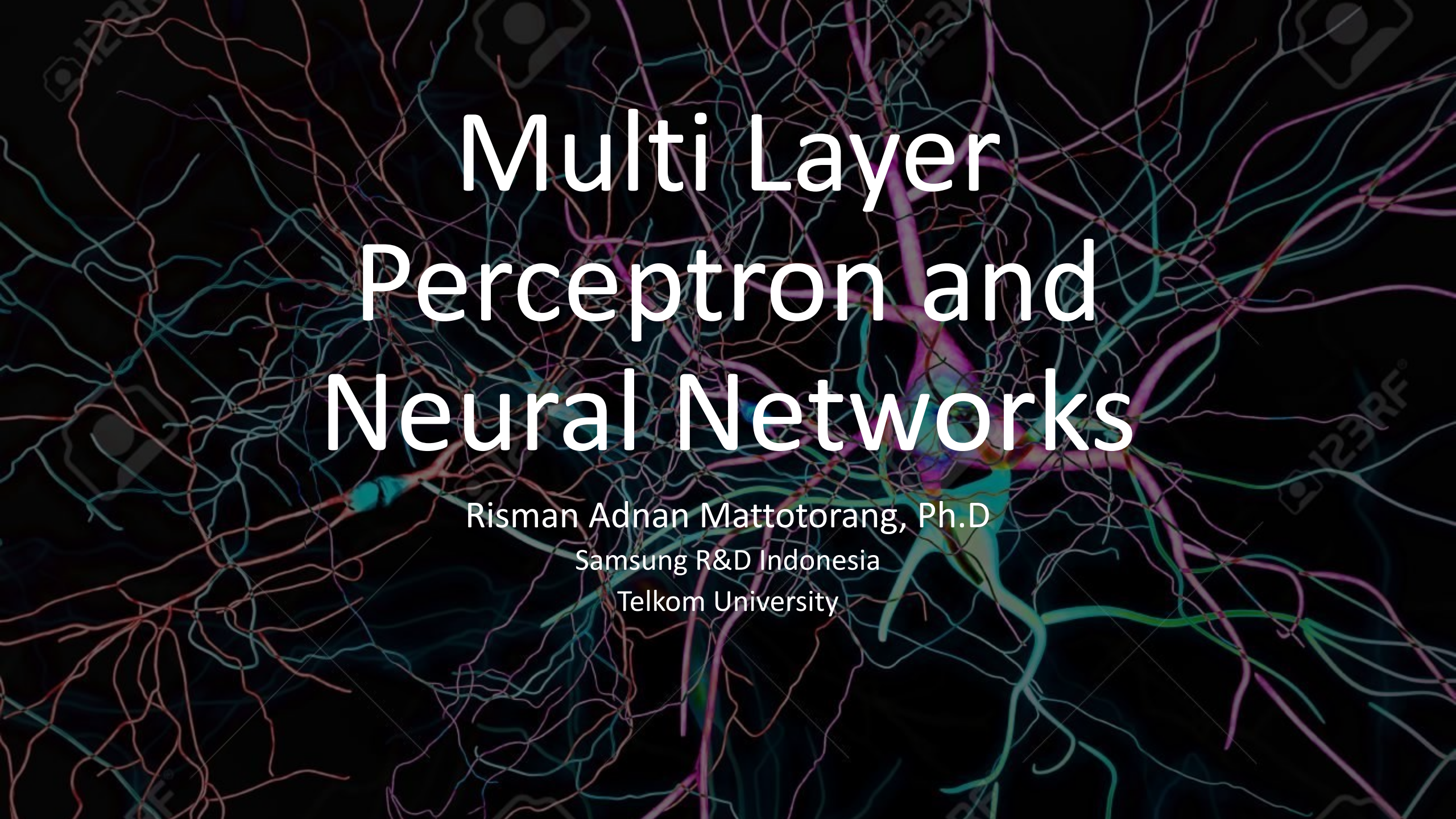
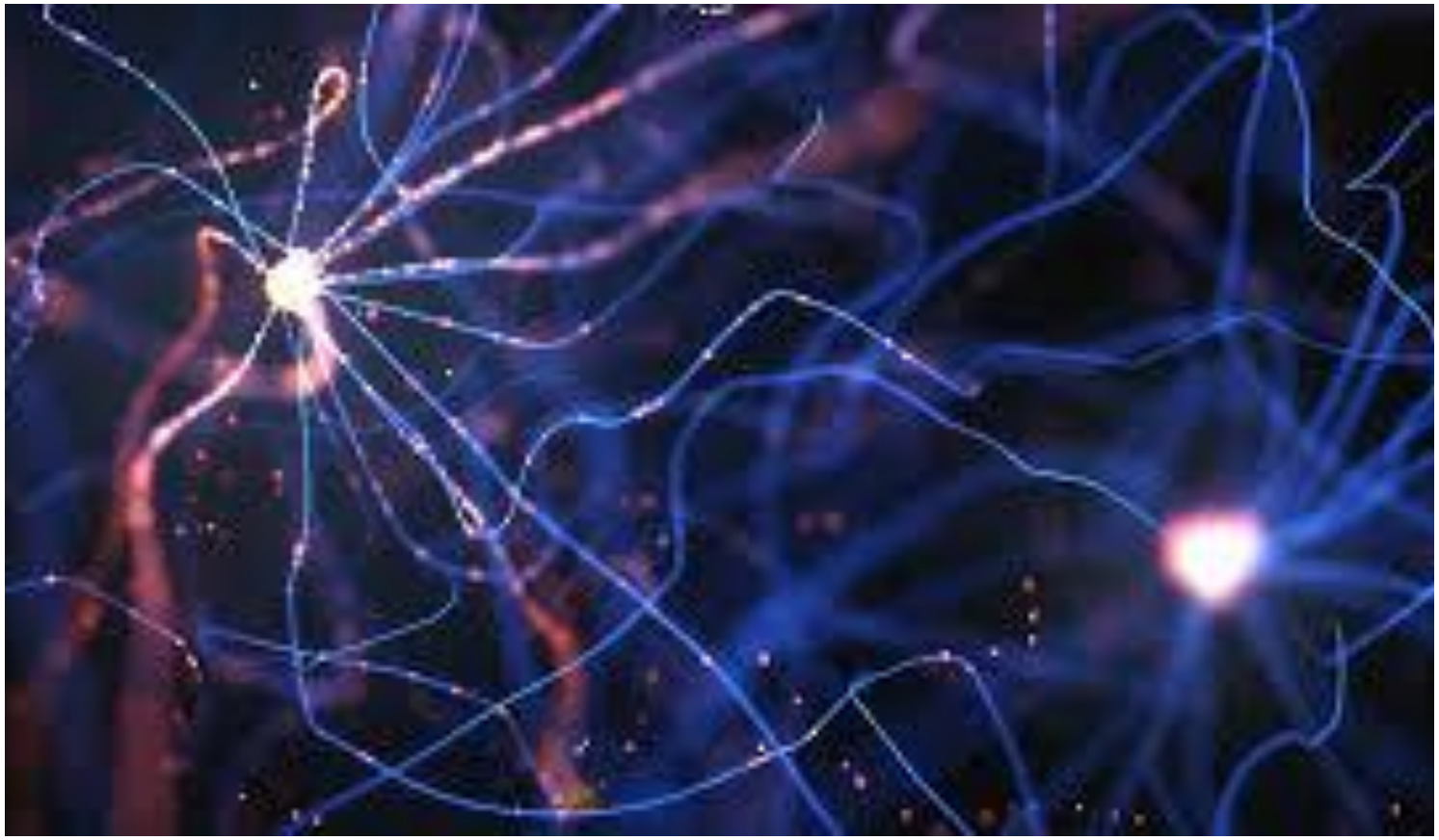


Multi Layer Perceptron and Neural Networks

Risman Adnan Mattotorang, Ph.D
Samsung R&D Indonesia
Telkom University

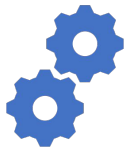


Outline

- Understanding Problem
- How Neural Network Works
- Error / Loss Function
- Back Propagation Algorithm
- Heuristic Training Dynamic
- Implementation
- Q&A

Thinking Framework – Supervised Learning

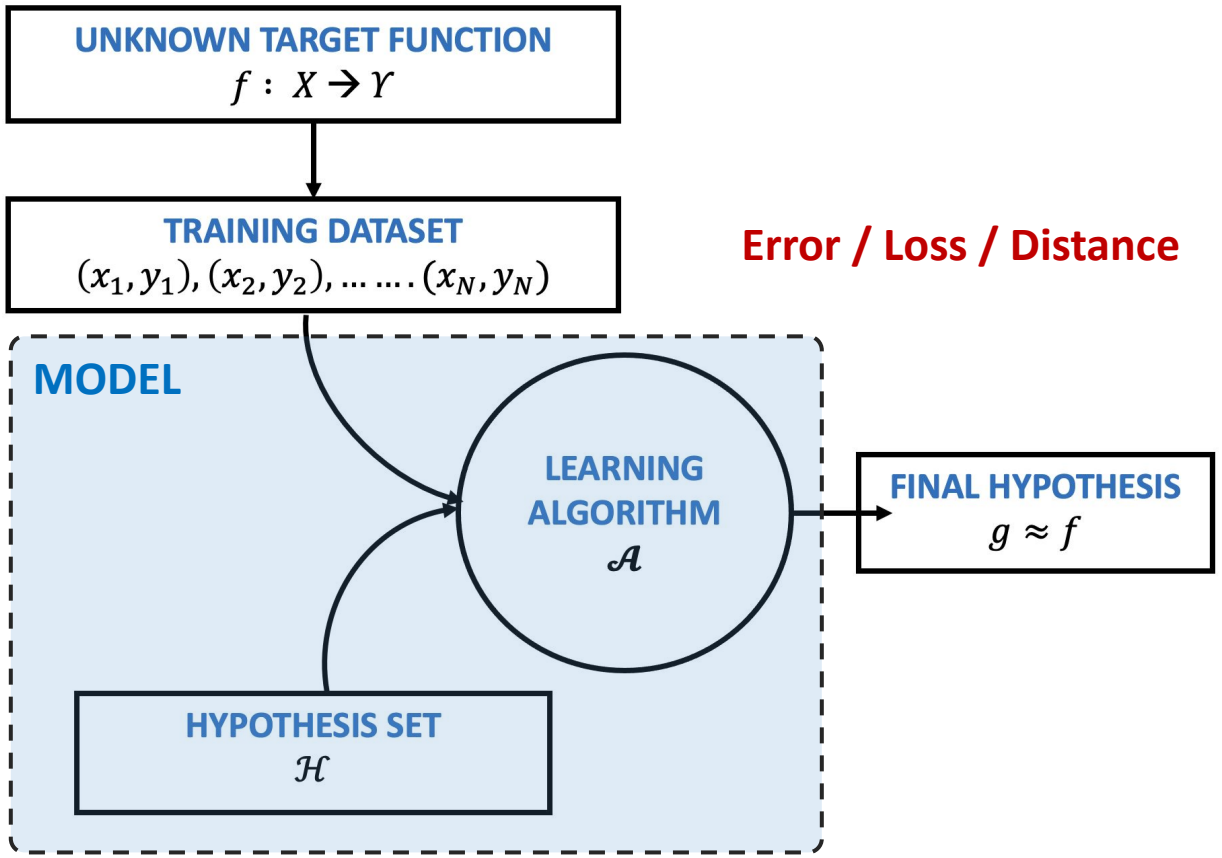
The **data** that we can learn from.



A **model** of how to transform the data.

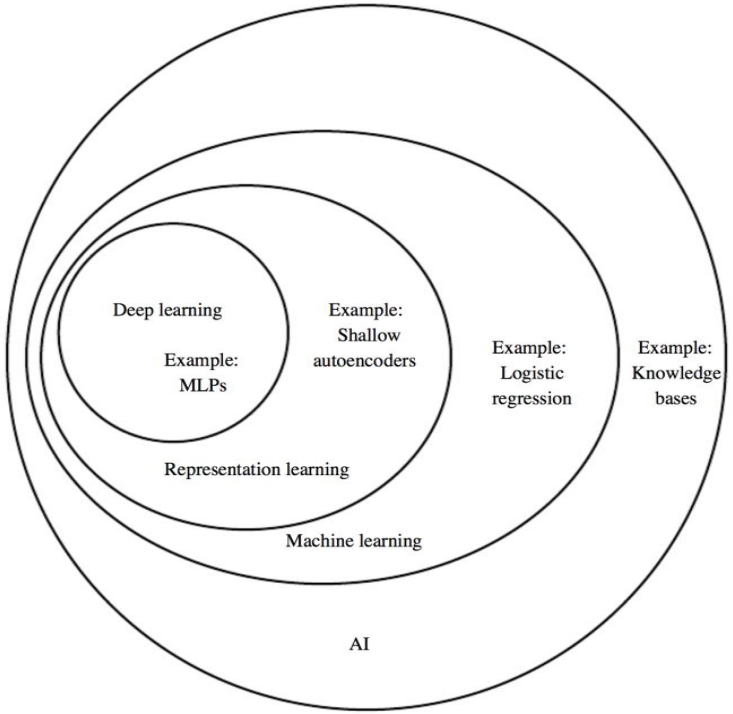
An **objective function** that quantifies how well (or badly) the model is doing.

An **algorithm** to adjust the model's parameters to optimize the objective function.

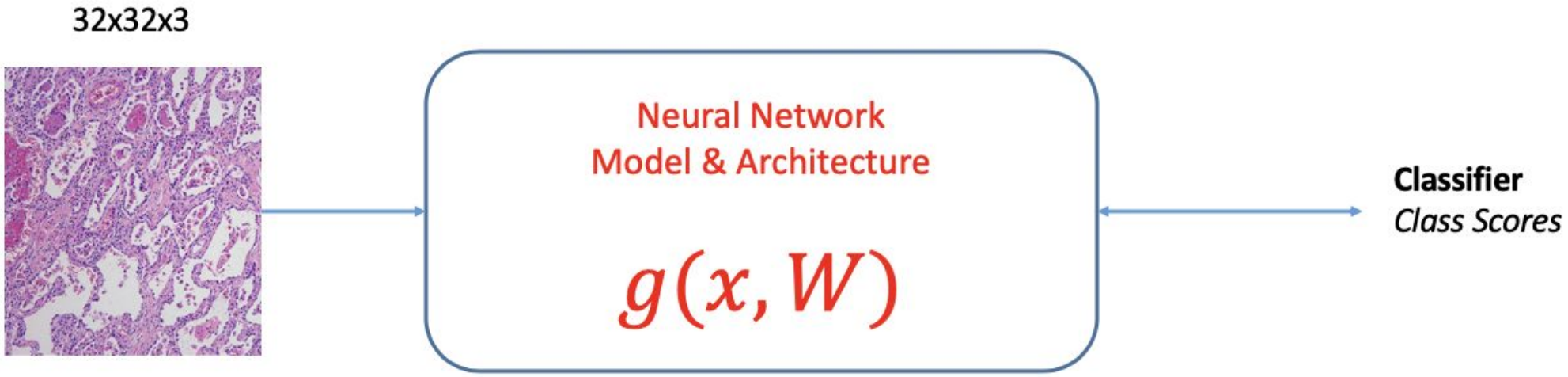


Machine Learning Landscape

•Machine learning become a jungle of models and methods!

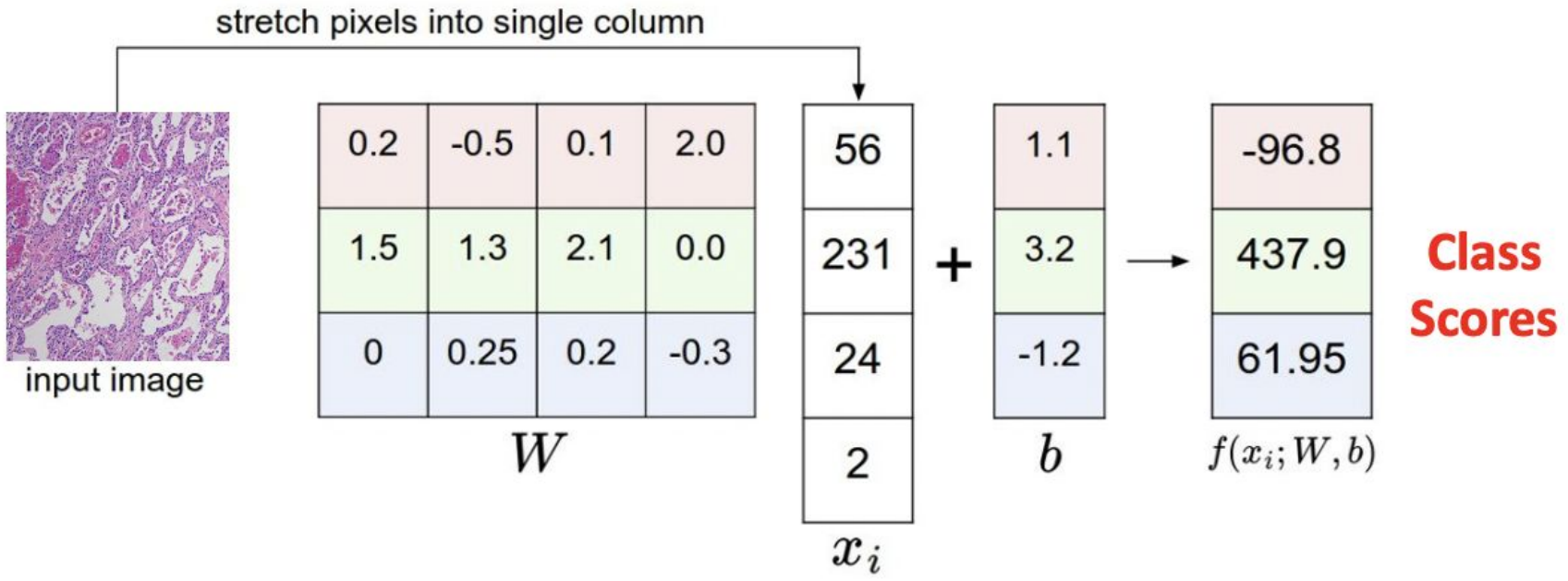


THEORY	TECHNIQUES		PARADIGMS
	MODELS	METHODS	
VC	Linear	**Regularization**	**Supervised**
Bias-Variance	**Neural Networks**	**Validation**	Unsupervised
Complexity	SVM	**Aggregation**	Reinforcement
Bayesian	Nearest Neighbors	**Input Processing**	Active
	RBF		Online
	Gaussian Processes		
	SVD		



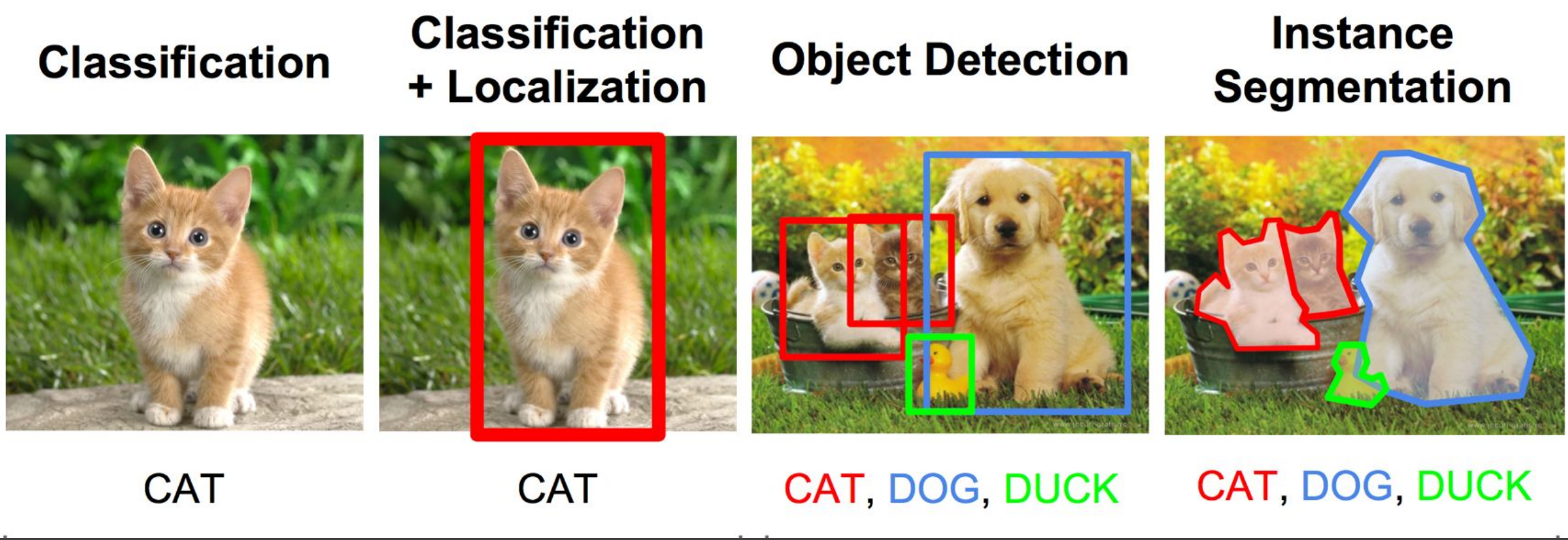
Essence of ML Model

- There is Pattern in Data
- We Don’t Have Analytical Solution
- We Have Data (+Label)

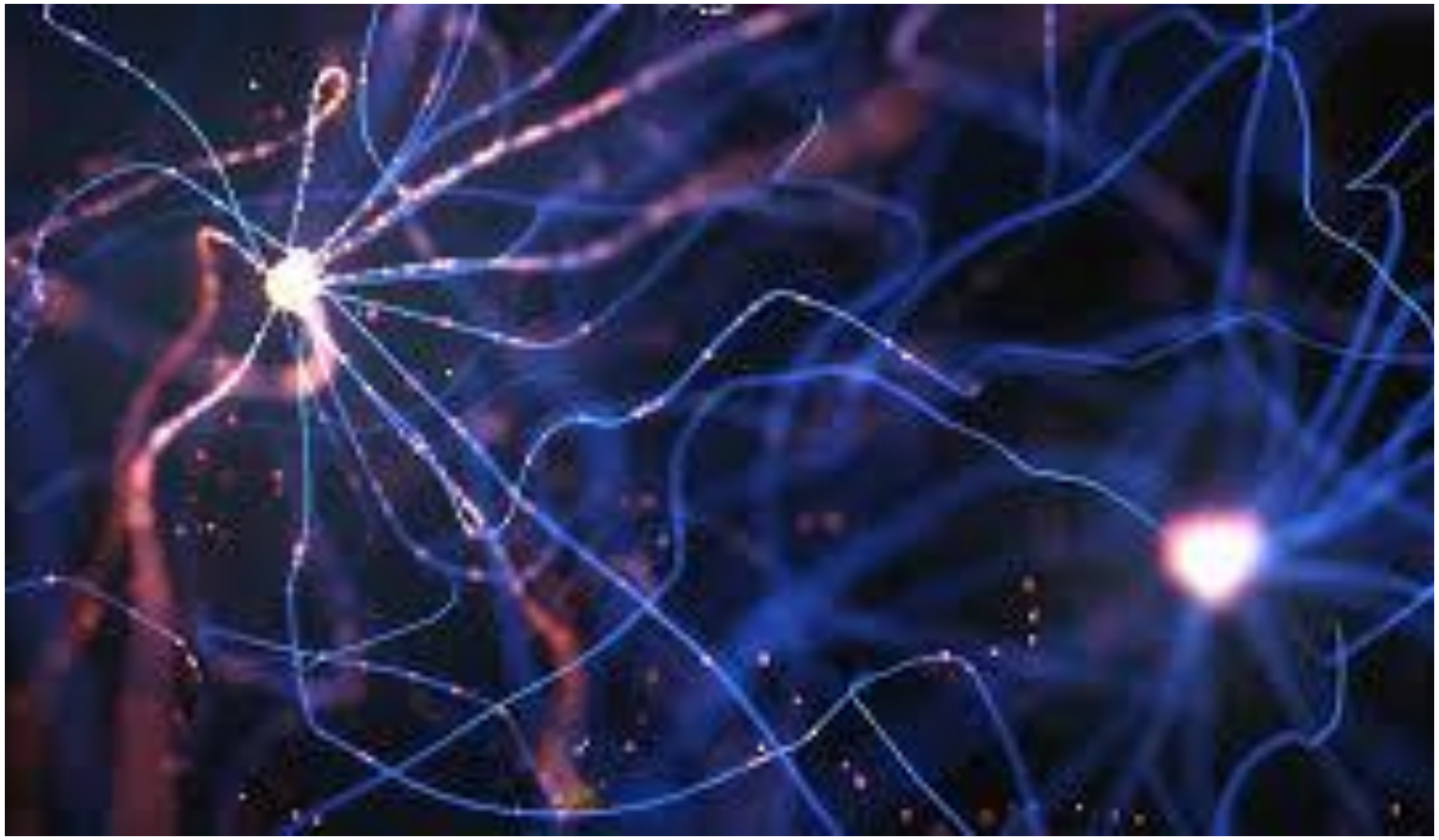


Trained Model

- What We Expect
  - Good Accuracy
  - Good Generalization
- What We Can Do
  - Inference



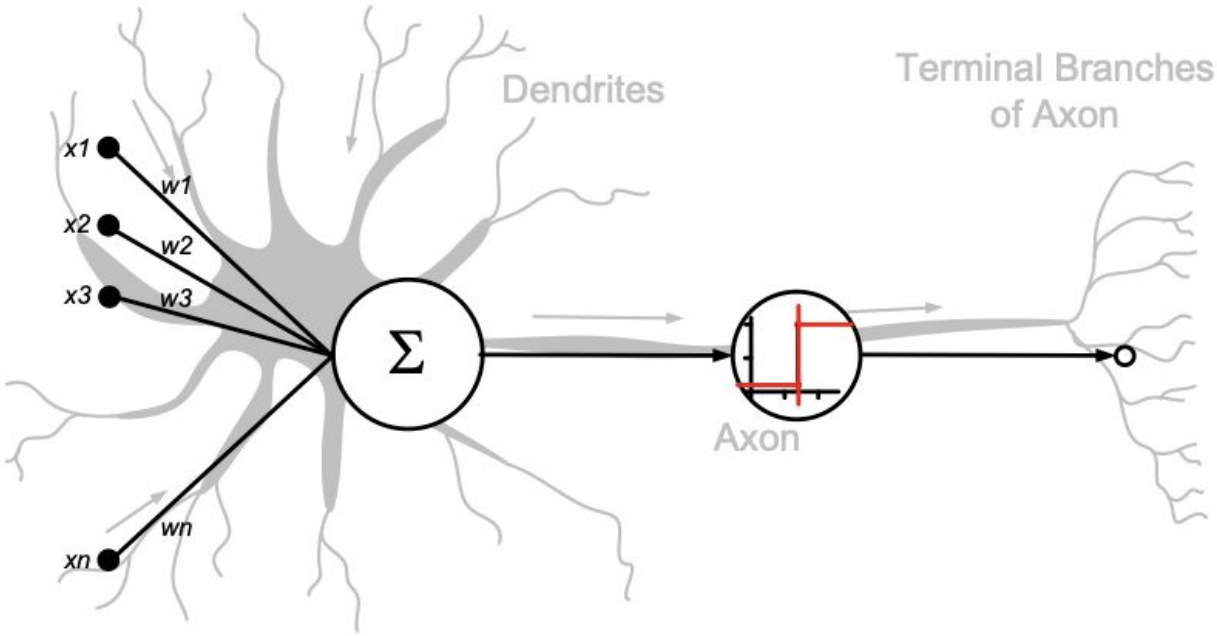
Computer Vision Domain



Outline

- Understanding Problem
- How Neural Network Works
- Error / Loss Function
- Back Propagation Algorithm
- Heuristic Training Dynamic
- Implementation
- Q&A

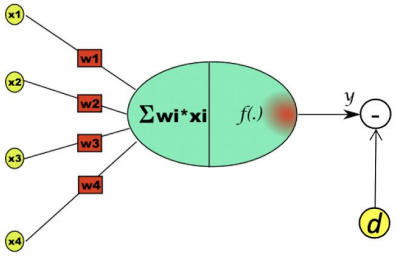
Artificial Neural Networks



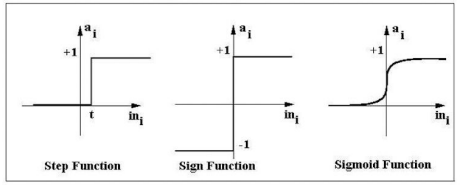
Slide credit : Andrew L. Nelson

A Simple Model - Perceptron

Perceptron:



Activation functions:



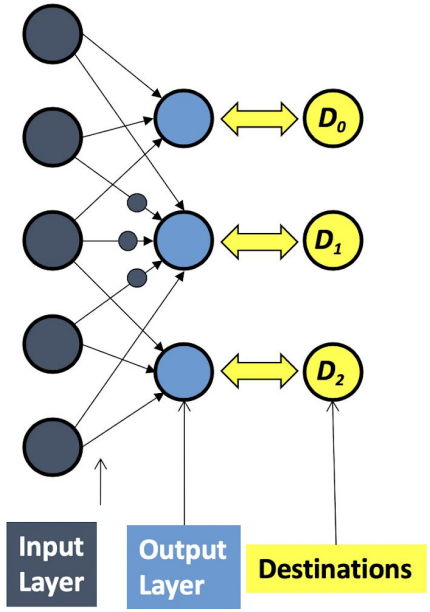
Learning:

$$y^{(t)} = f\left\{\sum_i w_i^{(t)} x_i^{(t)}\right\}$$

Update

$$\Delta w_i^{(t)} = \varepsilon(d^{(t)} - y^{(t)}) x_i^{(t)}$$

$$w_i^{(t+1)} = w_i^{(t)} + \Delta w_i^{(t)}$$



Slide credit : Geoffrey Hinton

Key Concepts:

1. Perceptron
2. Activation Function
3. Learning Algorithm

A Simple Hypothesis Set

1. Simple Threshold Function
2. Simple Linier Model
3. Only in Two Dimension

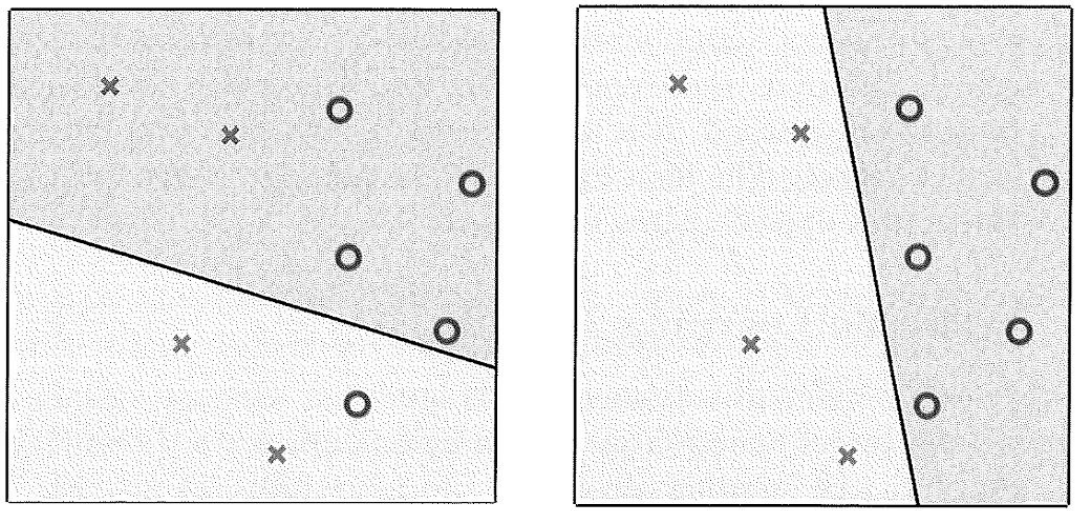
- For input **x** = (x_1, …., x_d) ' attributes for a customer'

$$\text{Approve credit if} \quad \sum_{i=1}^{d} w_i x_i > \text{threshold},$$

$$\text{Deny credit if} \quad \sum_{i=1}^{d} w_i x_i < \text{threshold}.$$

- This formula can be written more compact as

$$h(\mathbf{x}) = \text{sign}\left(\left(\sum_{i=1}^{d} w_i x_i\right) + b\right),$$



(a) Misclassified data (b) Perfectly classified data

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x}).$$

Perceptron Learning Algorithm

- The perceptron implement:

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x}).$$

- Given the training set:

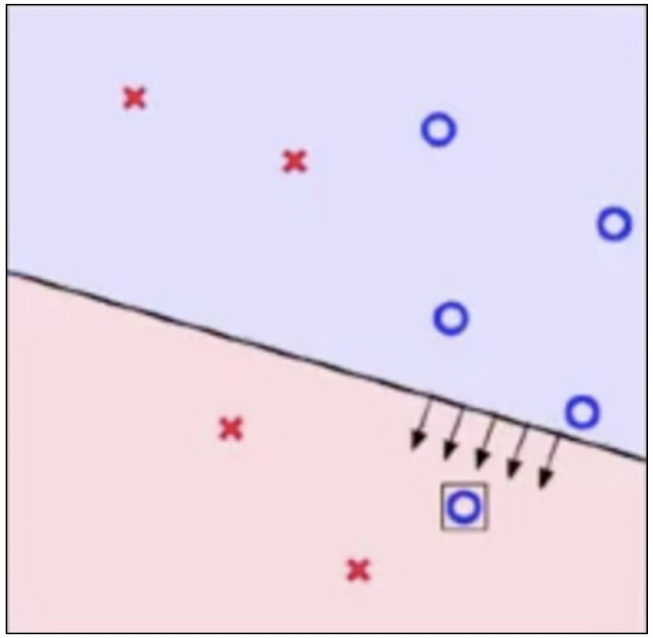
$$(\mathbf{x}_1, y_1) \cdots (\mathbf{x}_N, y_N)$$

- Pick a misclassified point:

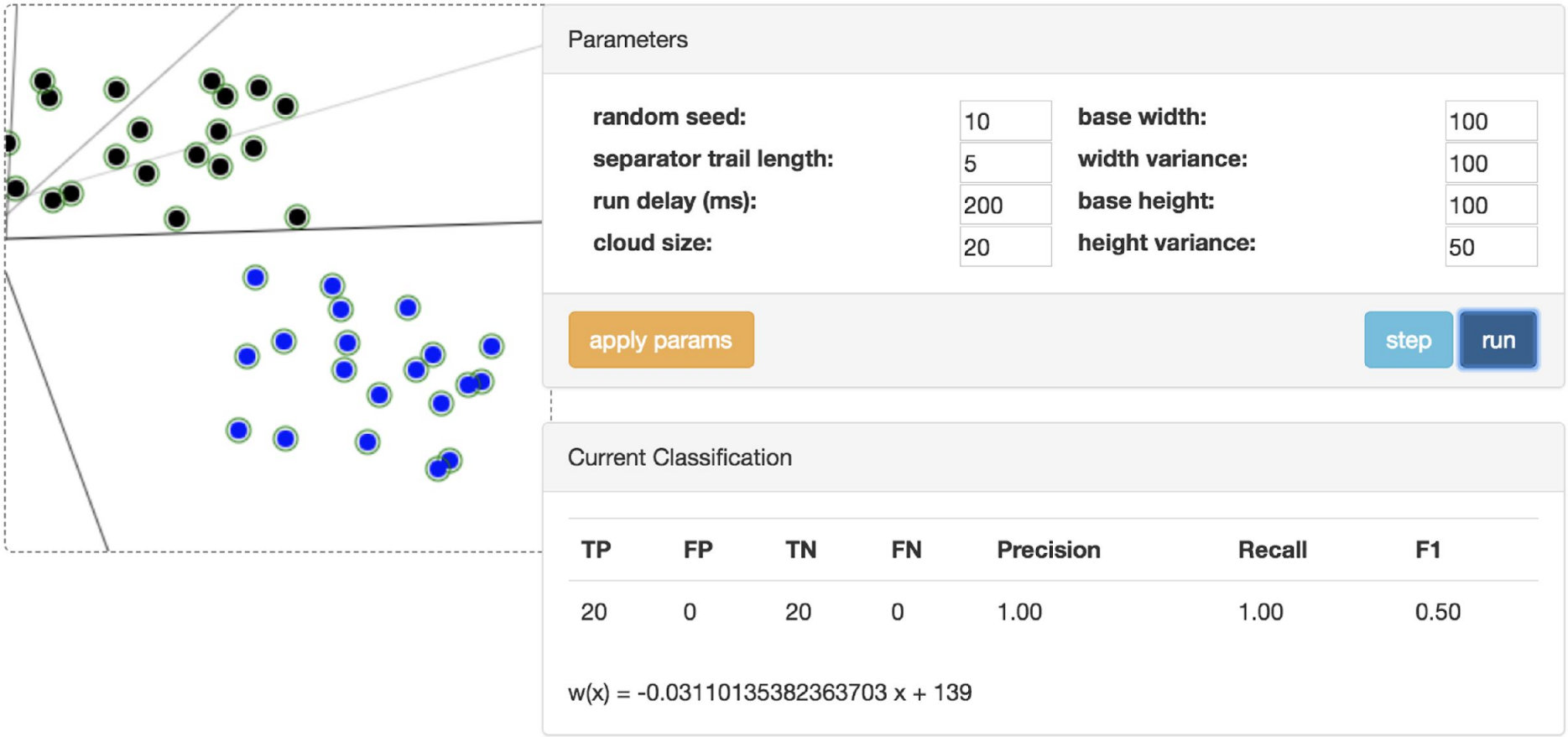
$$(\mathbf{x}(t), y(t))$$

- And update the weight vector:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + y(t)\mathbf{x}(t).$$

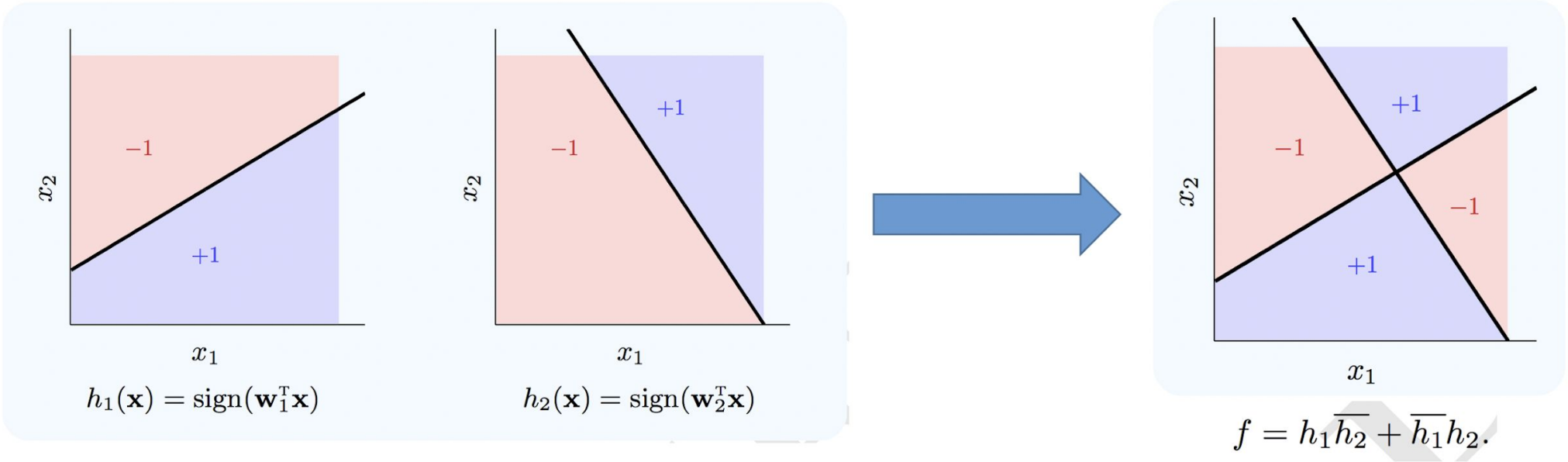


- Simple Rule to Update Parameters
- Iterative Update on Parameters
- No Stopping Criteria



Code credit : https://github.com/ditam/perceptron-demo

Perceptron Learning Algorithm

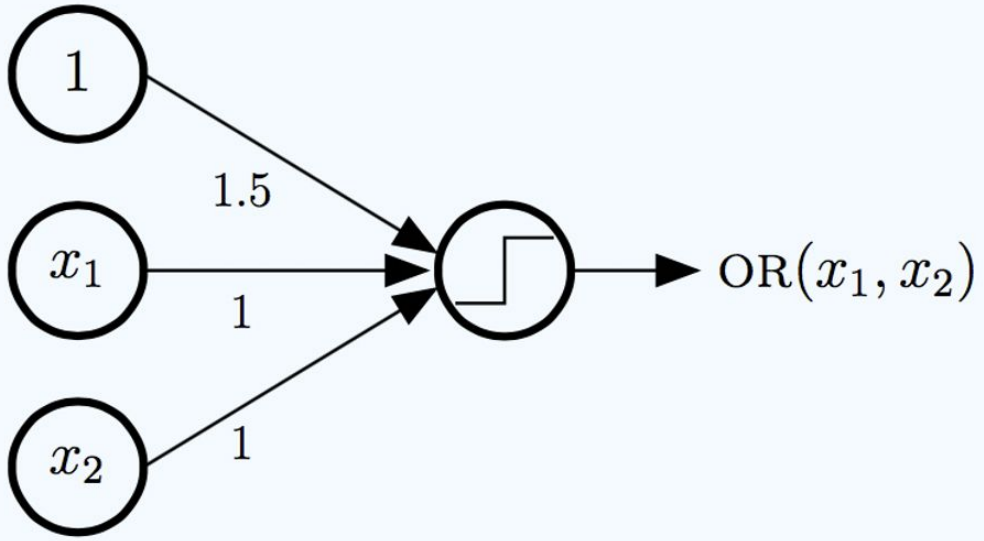


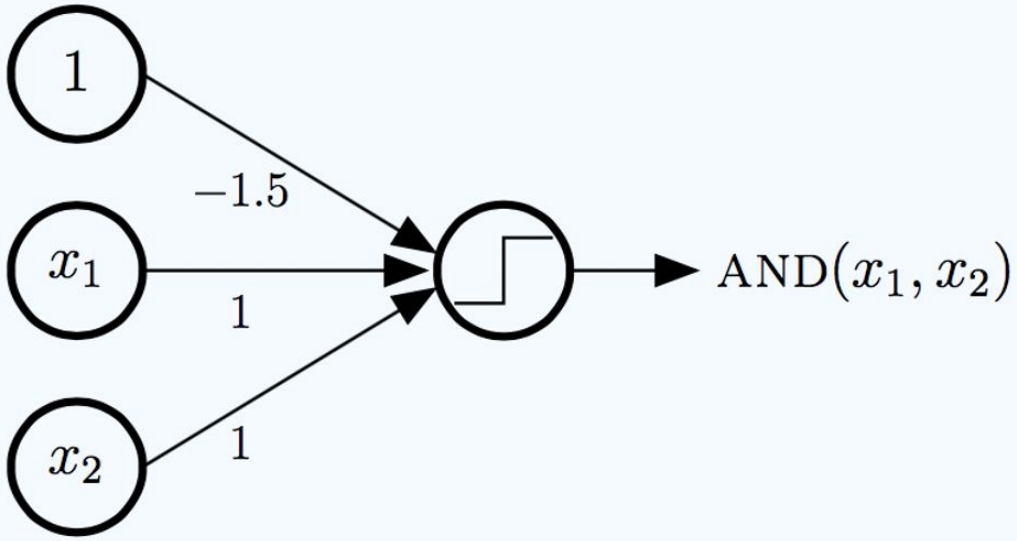
Note: Using standard Boolean notation (multiplication for AND, addition for OR, and overbar for negation)

Combining Perceptron

- Can perceptron combined to perform more complex target function?
- Yes if we can write *f* using simple AND and OR operations:

$$\begin{aligned} \mathrm{OR}(x_1, x_2) &= \mathrm{sign}(x_1 + x_2 + 1.5); \\ \mathrm{AND}(x_1, x_2) &= \mathrm{sign}(x_1 + x_2 - 1.5). \end{aligned}$$

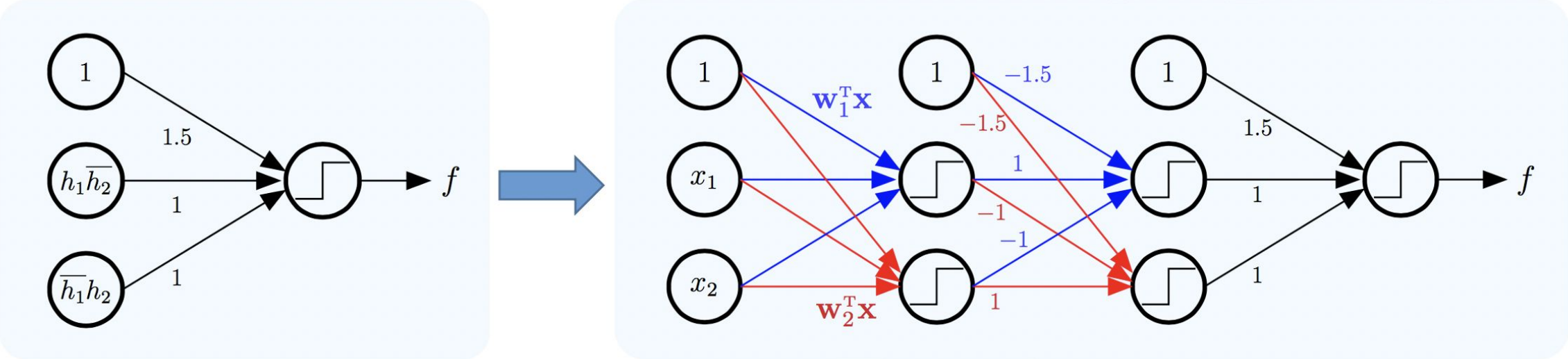




Combining Perceptron

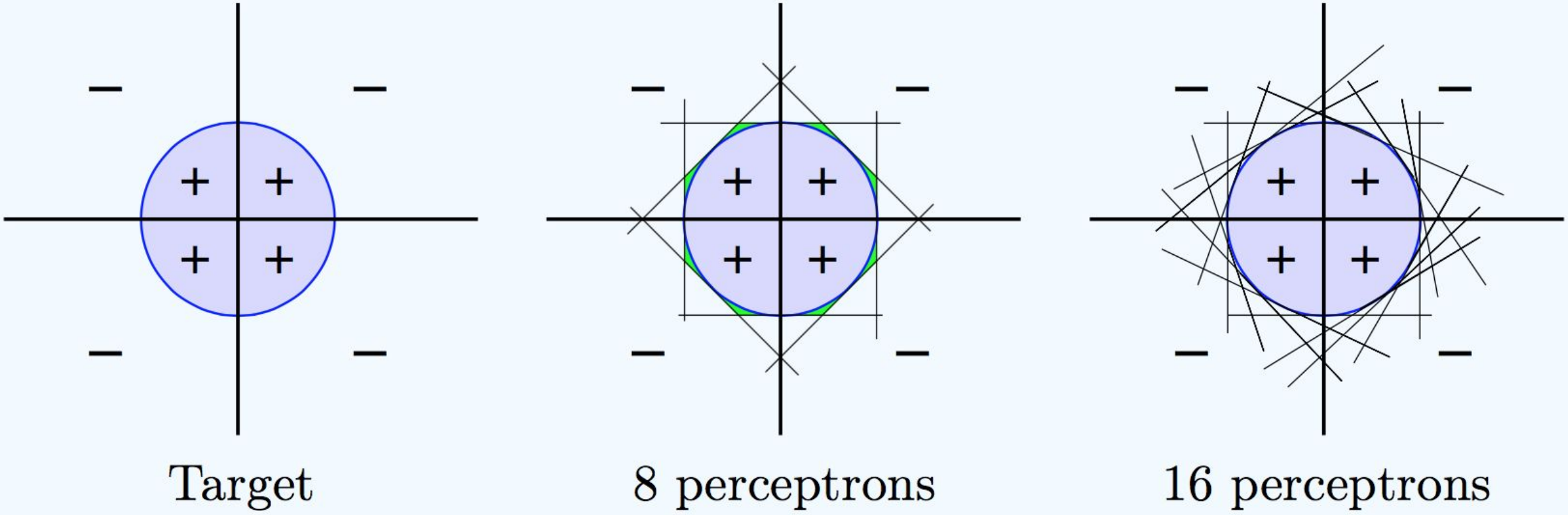
Can perceptron implement AND and OR operation?

- We can visualize $f = h_1\overline{h_2} + \overline{h_1}h_2$ as the following:



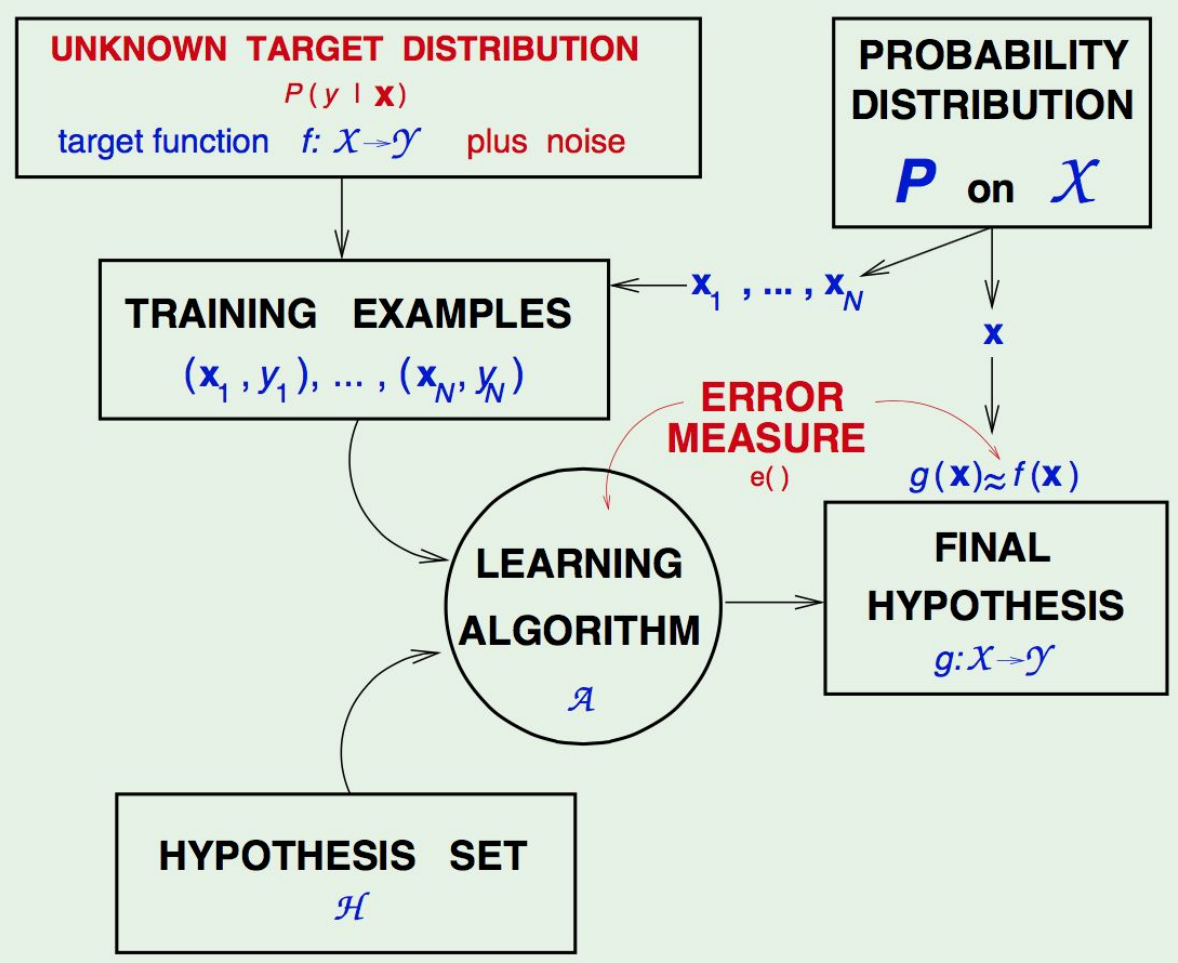
Combining Perceptron

- **Key Insight**: More layers of nodes are used between input and output to implement more complex target function *f*. We called **Multi-Layer Perceptron**. Additional layers are called ***hidden layers***.



A Powerful Model

- If *f* can be decomposed into perceptrons using an OR of ANDs, then it can be implemented by a 3-layer perceptron. If *f* is not strictly decomposable into perceptrons, but the decision boundary is smooth, then a 3-layer perceptron can come arbitrarily close to implementing *f*. **Visual Proof**:



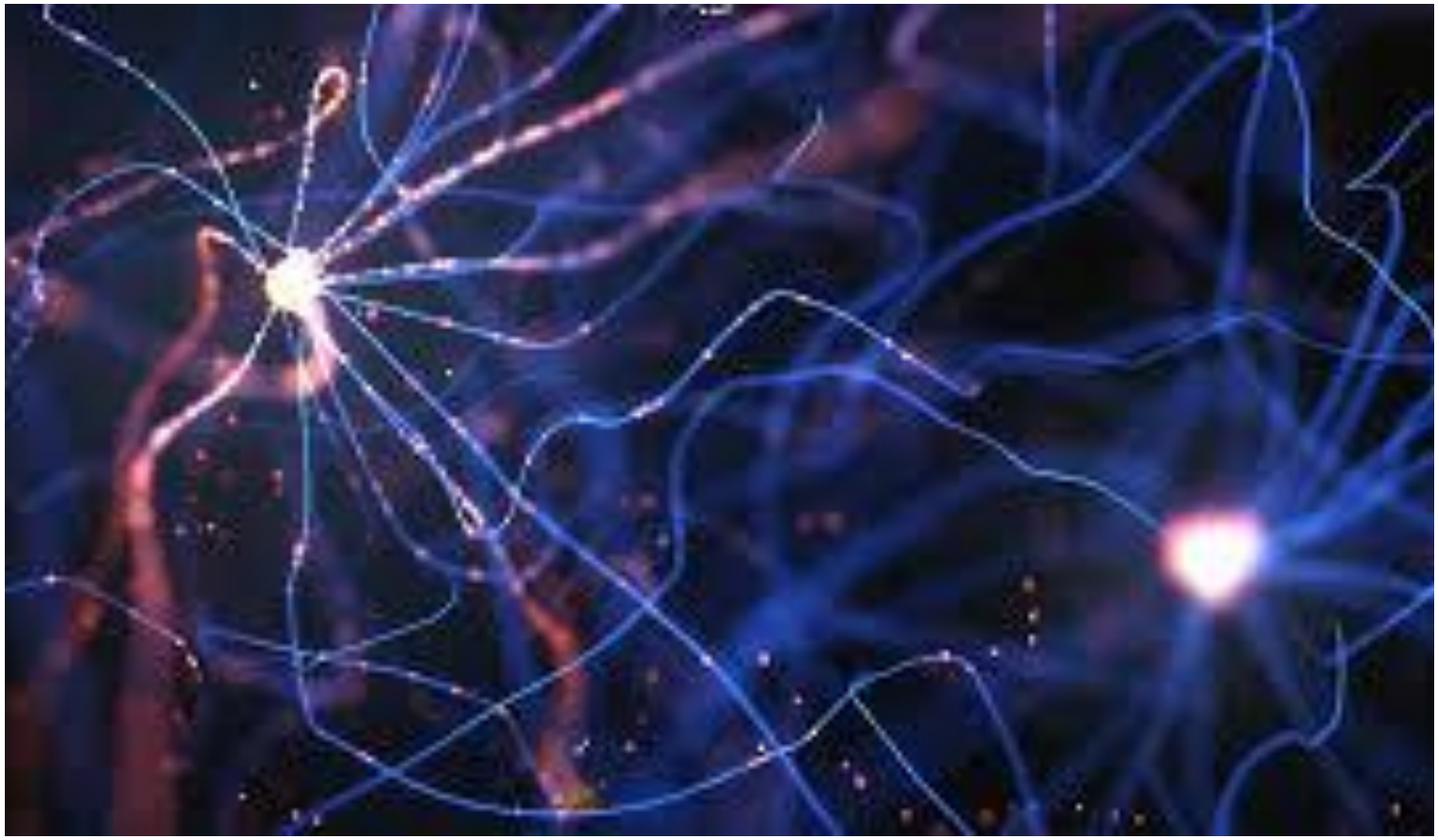
- **How to Learn Weights?**
  - **Error or Loss or Cost Function**
  - Optimization Technique
  - Regularization Technique
  - Goal: Accuracy & Generalization

- Error function quantifying what it means to have a "good" **weights**.
- We have to come up with a way to minimize error/loss.

Two type of error/loss:

- In-sample Error E_{in}
- Out-of-sample Error E_{out}

The Learning Framework



Outline

- Understanding Problem
- How Neural Network Works
- Error / Loss Function
- Back Propagation Algorithm
- Heuristic Training Dynamic
- Implementation
- Q&A

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathrm{e}_n$$

How to Learn the Weights?

Goal: Learning all weights by minimizing error

- **One Hidden Layer:** Simply aggregation of perceptrons.

- **Deep Neural Networks**? Not Easy. Why?
  - Challenging to Train.
  - High Model Complexity.
  - Sophisticated Structural Decisions.
  - Hard Optimization Problem
  - Computational Complexity

Error Measures

What does $h \approx f$ means?

- Error (sometime called ***loss***) measure: $E(h, f)$
- Almost always pointwise definition: $e(h(\boldsymbol{x}), f(\boldsymbol{x}))$

Examples:

- Squared error : $e\big(h(\boldsymbol{x}), f(\boldsymbol{x})\big) = \big(h(\boldsymbol{x}) - f(\boldsymbol{x})\big)^2$
- Binary error : $e\big(h(\boldsymbol{x}), f(\boldsymbol{x})\big) = [\![h(\boldsymbol{x}) \neq f(\boldsymbol{x})]\!]$

Note: There are drawbacks with squared error.

Question: How to choose overall error function?

From Pointwise to Overall

- Overall error $E(h, f)$ = average of pointwise error $e(h(\boldsymbol{x}), f(\boldsymbol{x}))$

- In sample error : $E_{in}(h) = \frac{1}{N} \sum_{n-1}^{N} e(h(\boldsymbol{x}_n), f(\boldsymbol{x}_n))$
- Out of sample error : $E_{out}(h) = E_x[e(h(\boldsymbol{x}), f(\boldsymbol{x}))]$

Learning is feasible. It is likely that $E_{out}(\boldsymbol{g}) = E_{in}(\boldsymbol{g})$

Main Questions:

- Can we make sure that $E_{out}(g)$ is close enough to $E_{in}(\boldsymbol{g})$?
- Can we make $E_{in}(\boldsymbol{g})$ small enough?

How to Choose Error Function

- More or less same with other parametric models.
- The squared error measure has some drawbacks.
- **Question**: Is there a different error function that works better?

- **Yes**: Use SVM classifier or force the outputs to represent **a probability distribution** across discrete alternatives.

- **We will learn the following:**
  - Multiclass SVM Loss
  - Softmax Classifier

Softmax Classifier

- Multinomial Logistic Regression

cat	**3.2**
car	5.1
frog	-1.7

scores = *unnormalized log probabilities of the classes.*

$$s = f(x_i; W) \text{ and } P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ Softmax function

Want to maximize **the log likelihood**, or (for error/loss function) to minimize the negative log likelihood of the correct class:

$$E_i = -\log P(Y = y_i | X = x_i) = -\log(\frac{e^{s_k}}{\sum_j e^{s_j}})$$

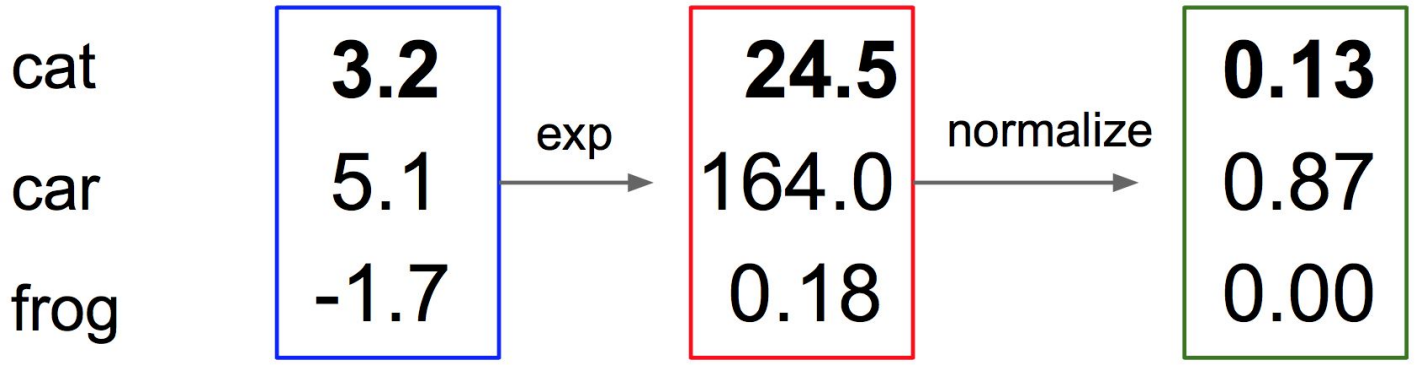
Softmax Classifier

- Multinomial Logistic Regression

$$E_i = -\log\left(\frac{e^{s_k}}{\sum_j e^{s_j}}\right)$$

Softmax function

Unnormalized Probabilities



$$E_i = -\log(0.13) = 0.89$$

Unnormalized Log Probabilities

Probabilities

Multiclass SVM Loss

- Suppose: 3 training examples, 3 classes
- With some W the scores $f(x, W) = Wx$ are:



cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

Multiclass SVM Error/Loss:

Given an example (x_i, y_i)

where x_i is image and

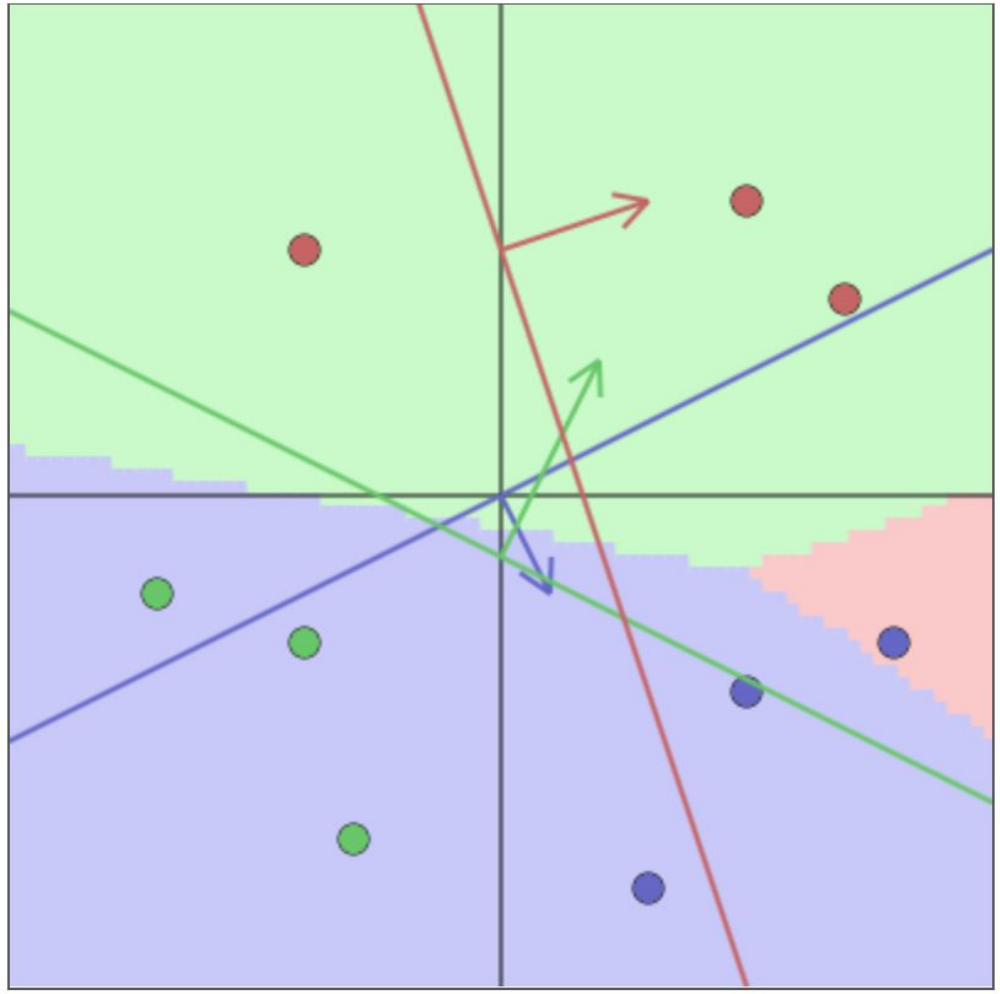
where y_i is the (integer) label

Shorthand for score vector $s = f(W, x_i)$

SVM loss has the form:

$$E_i = \sum_{j \neq yi} \max(0, s_j - s_{yi} + 1)$$

Interactive Web Demo



W[0,0]	W[0,1]	b[0]
1.00 *-0.38*	2.00 *0.07*	0.00 *0.11*
W[1,0]	**W[1,1]**	**b[1]**
2.00 *0.51*	-4.00 *-0.58*	0.50 *-0.11*
W[2,0]	**W[2,1]**	**b[2]**
3.00 *0.17*	-1.00 *0.36*	-0.50 *0.00*

Step size: 0.10000

Single parameter update

Start repeated update

Stop repeated update

Randomize parameters

x[0]	x[1]	y	s[0]	s[1]	s[2]	L
0.50	0.40	0	1.30	-0.10	0.60	0.30
0.80	0.30	0	1.40	0.90	1.60	1.70
0.30	0.80	0	1.90	-2.10	-0.40	0.00
-0.40	0.30	1	0.20	-1.50	-2.00	3.20
-0.30	0.70	1	1.10	-2.90	-2.10	6.80
-0.70	0.20	1	-0.30	-1.70	-2.80	2.40
0.70	-0.40	2	-0.10	3.50	2.00	2.50
0.50	-0.60	2	-0.70	3.90	1.60	3.30
-0.40	-0.50	2	-1.40	1.70	-1.20	4.70
						mean:
						2.77

Total data loss: 2.77
Regularization loss: 3.50
Total loss: 6.27

L2 Regularization strength: 0.10000

Multiclass SVM loss formulation:
- (•) Weston Watkins 1999
- ( ) One vs. All
- ( ) Structured SVM
- ( ) Softmax

http://vision.stanford.edu/teaching/cs231n/linear-classify-demo

Bugs with Error/Loss

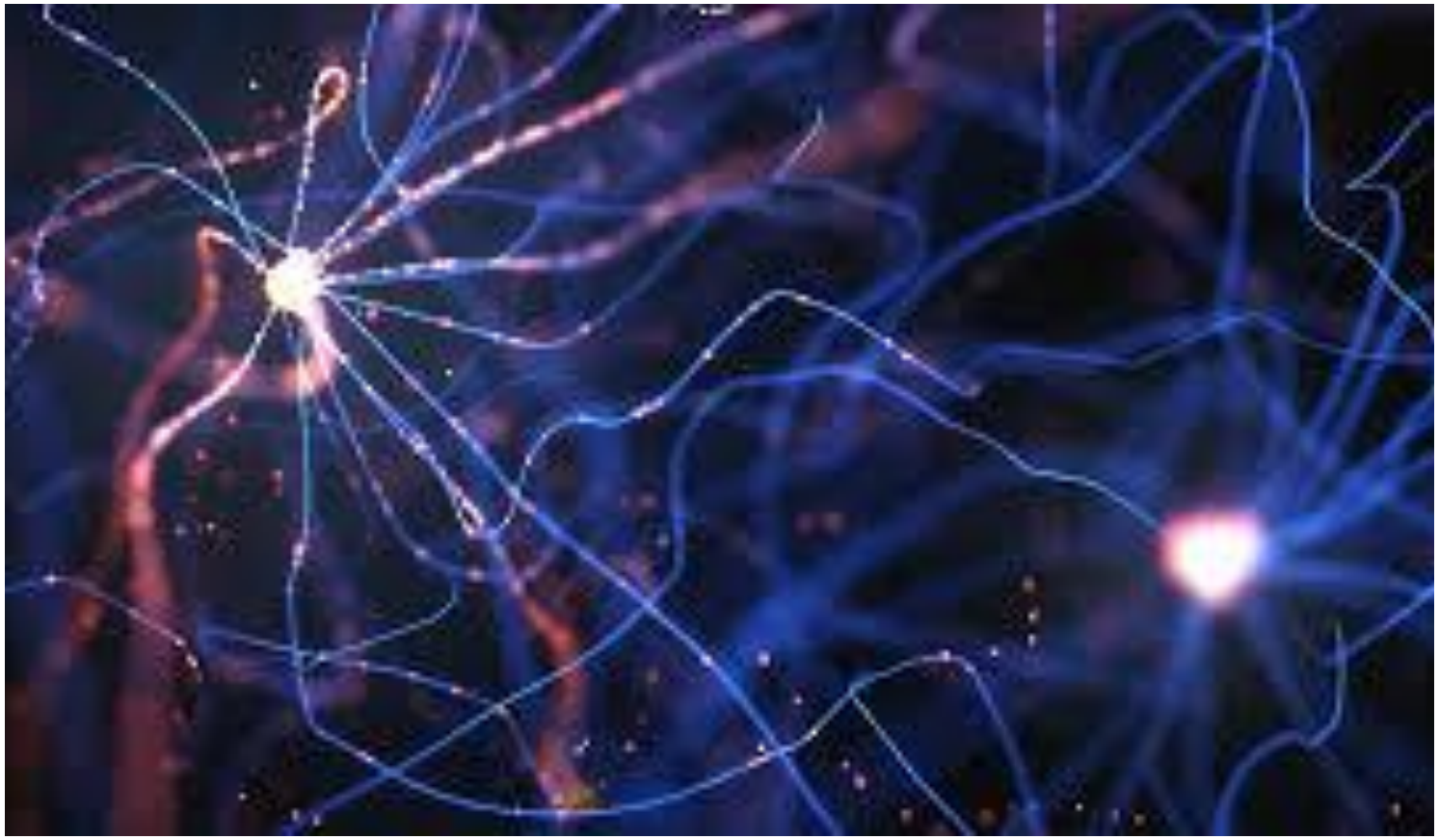
•

$$f(x, W) = Wx$$

$$E = \frac{1}{N}\sum_{i=1}^{N}\sum_{j \neq yi} \max(0, f(x_i; W)_j - f(x_i; W)_{yi} + 1)$$

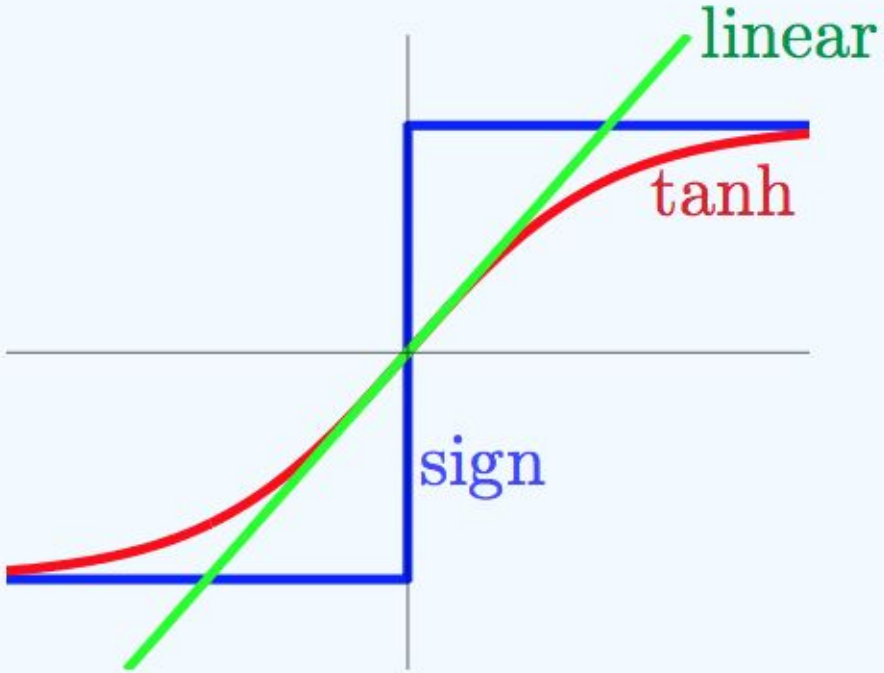
Eg. Suppose that we found a *W* such that *E* = 0.

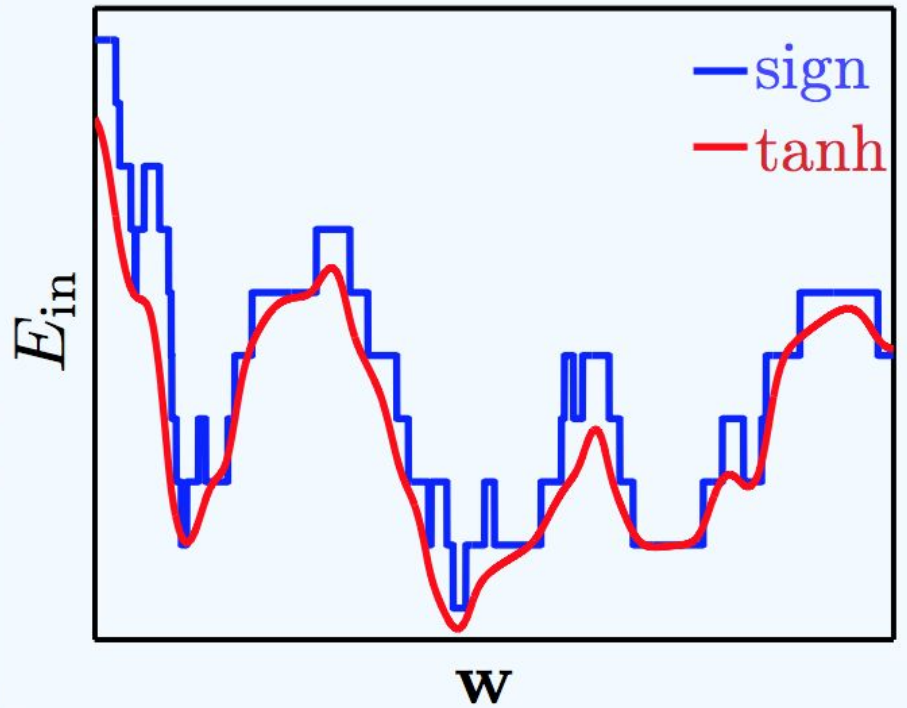
Is this *W* unique?



Outline

- Understanding Problem
- How Neural Network Works
- Error / Loss Function
- Back Propagation Algorithm
- Heuristic Training Dynamic
- Implementation
- Q&A



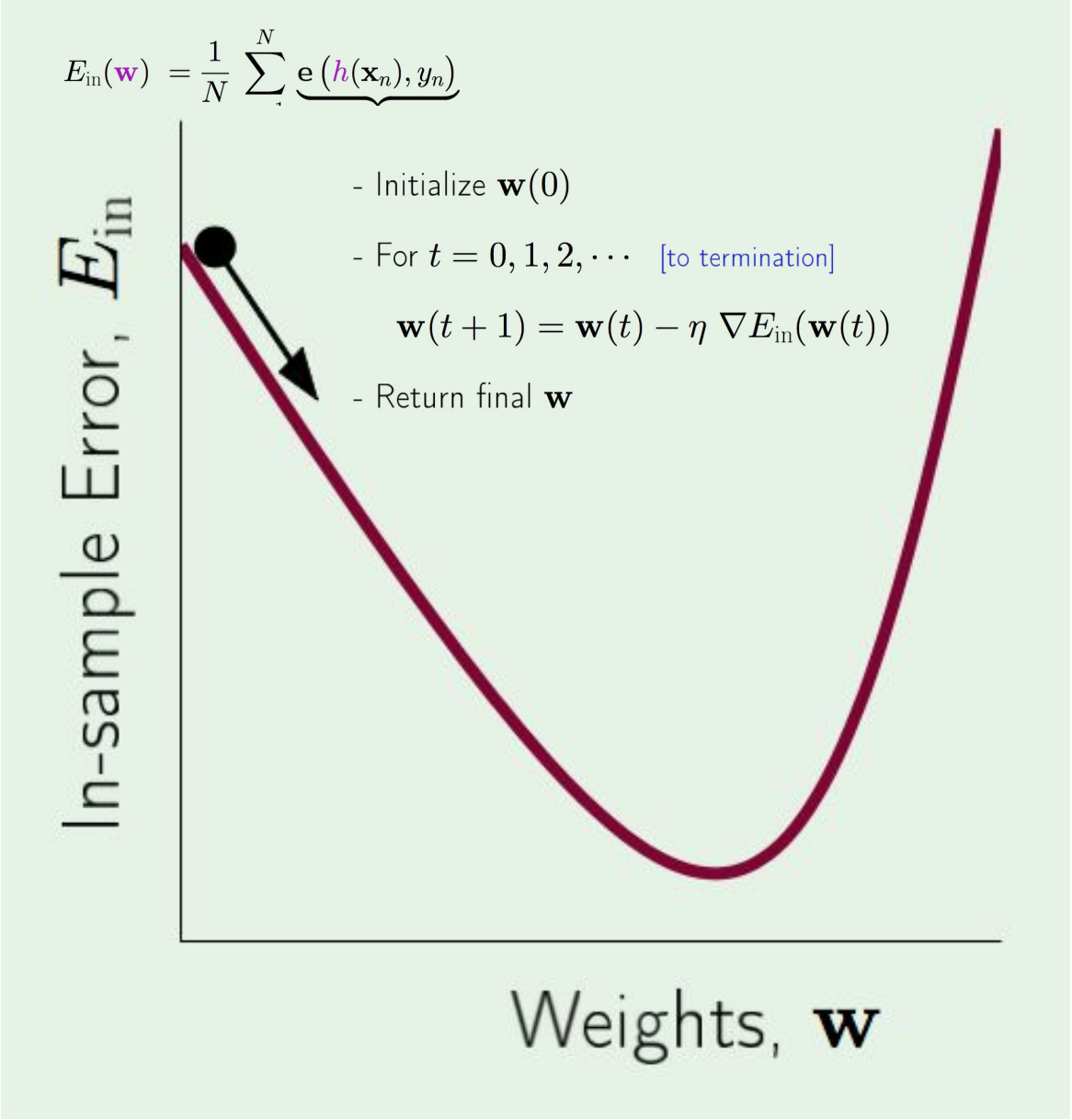


Activation Function Nonlinearity

- Using **sign**(.) turn into hard combinatorial problem, even harder for MLP.
- We a need smooth version of **sign**(.) function, a differentiable approximation to **sign**(.).
- The sigmoidal approximation captures the general shape of the in-sample error E_{in},.

Gradient Descent

- In PLA, **Gradient Descent** is a iterative optimization algorithm to find a local minimum of in-sample error function (E_{in}) or sometime called **Cost Function.**
- GD minimize E_{in} by iterative steps for all examples ($\boldsymbol{x}_n$, $\boldsymbol{y}_n$) and we called it "batch" GD.

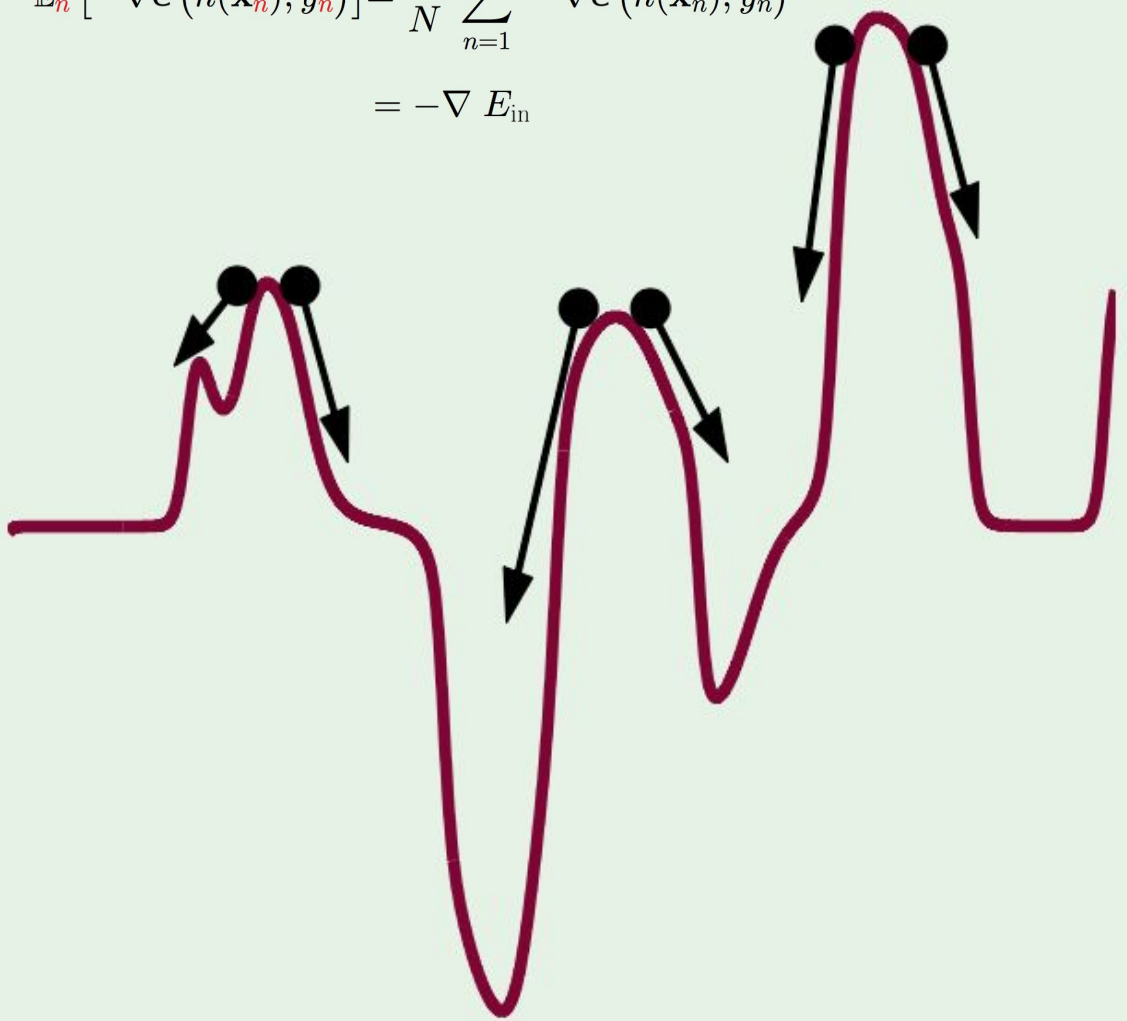


Stochastic Gradient Descent

- Stochastic Gradient Descent (SGD), is a ***stochastic*** approximation of the gradient descent optimization method for minimizing in-sample error function E_{in} . The method sometime called “mini batch” or a randomized version of GD.
- **Benefit of SGD**: Cheaper, Randomization, Simple

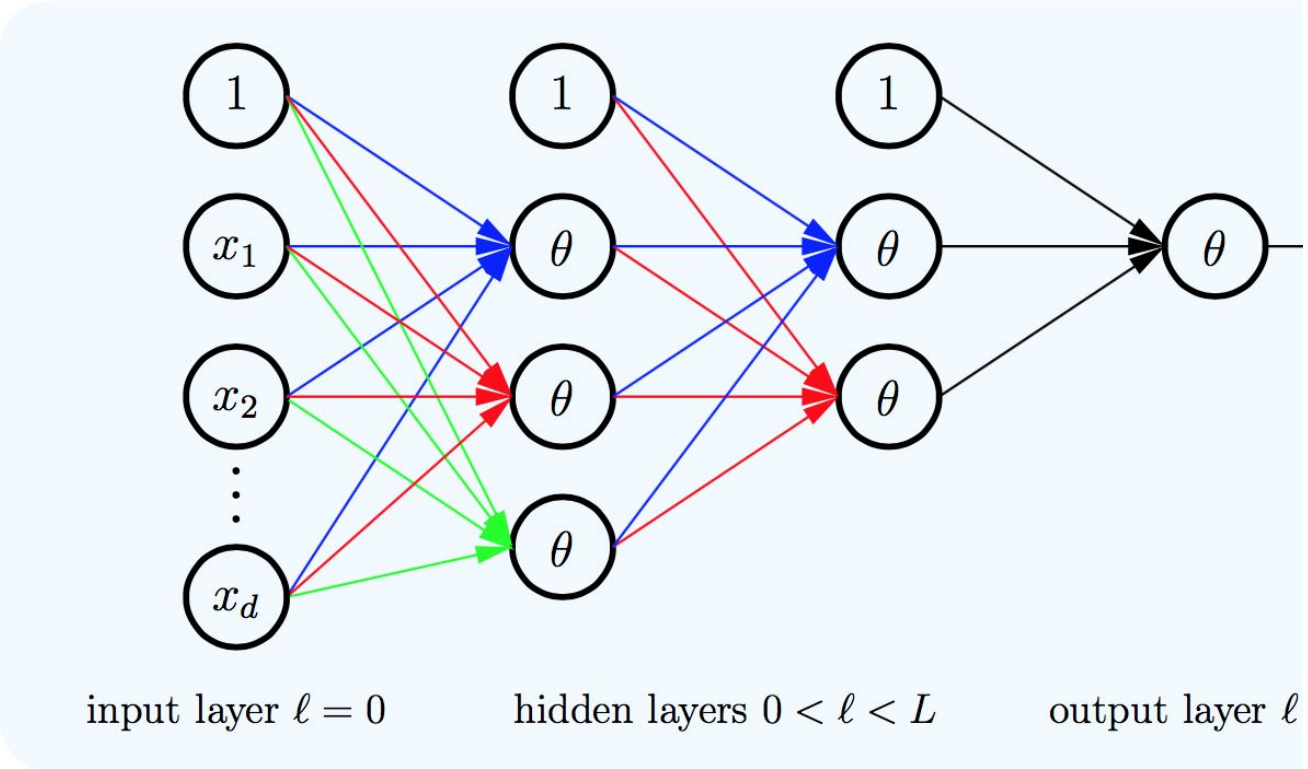
- Pick one ($\boldsymbol{x}_n$, $\boldsymbol{y}_n$) at a time. Apply GD to $\mathbf{e}(h(\boldsymbol{x}_n), \boldsymbol{y}_n)$
- Use average direction:

$$\mathbb{E}_n\left[-\nabla \mathbf{e}\left(h(\mathbf{x}_n), y_n\right)\right] = \frac{1}{N}\sum_{n=1}^{N} -\nabla \mathbf{e}\left(h(\mathbf{x}_n), y_n\right)$$
$$= -\nabla E_{\text{in}}$$



Randomization helps to escape local minimums

Feed-forward Neural Network



Architecture:

- **Input** Layer
- **Hidden** Layers
- **Output** Layer
- One Direction Flow

Deep Neural Networks = Many Hidden Layers

Two Red Flags for **Generalization** and **Optimization**

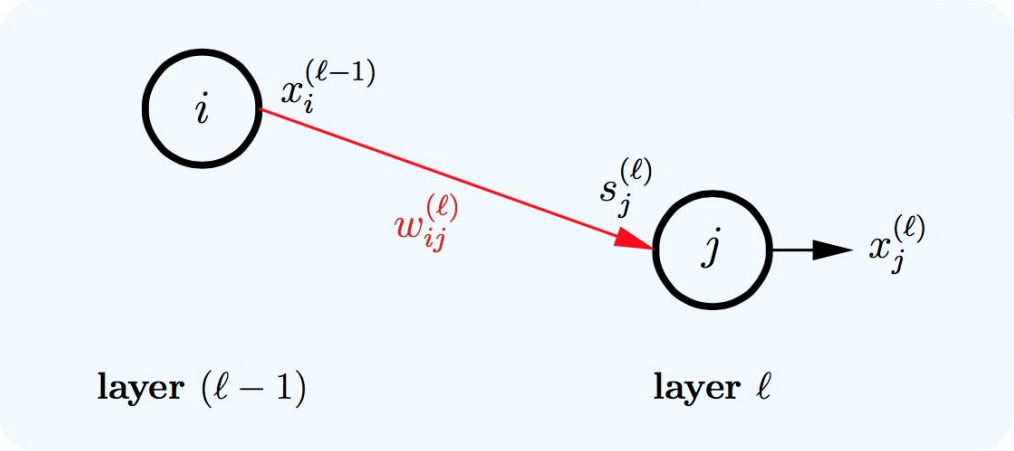
Applying SGD

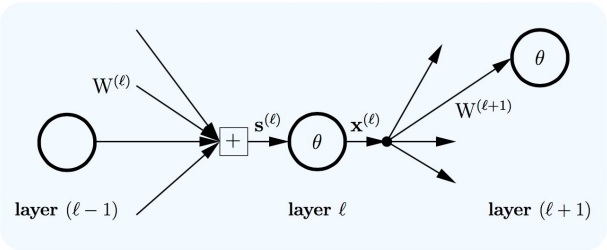
Problem Statement:

- SGD Iteration: $\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E_{\text{in}}(\mathbf{w}(t))$ where $E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathrm{e}_n$ and $\mathrm{e}_n = \mathrm{e}(h(\mathbf{x}_n), y_n)$
- Compute *h*(**x**) from known **w,** forward propagation
- To compute **e** we need *h*(**x**) and **w**
- To apply SGD we need to know **how to compute gradient** of **e** efficiently

What You Have to Know

1. Formal Notation
2. Forward Propagation
3. Back Propagation





layer ℓ parameters

signals in	$\mathbf{s}^{(\ell)}$	$d^{(\ell)}$ dimensional input vector
outputs	$\mathbf{x}^{(\ell)}$	$d^{(\ell)}+1$ dimensional output vector
weights in	$\mathrm{W}^{(\ell)}$	$(d^{(\ell-1)}+1)\times d^{(\ell)}$ dimensional matrix
weights out	$\mathrm{W}^{(\ell+1)}$	$(d^{(\ell)}+1)\times d^{(\ell+1)}$ dimensional matrix

Formal Notation

Conventions:

- Layers labeled by *l*
- *l = 0* is input
- *l = L* is output
- *L* = number of layers
- Each layer *l* has dimension *d*
- Arrow represent weight

After we fix the weights, we have specified a particular neural network hypothesis *h*(**x**, **w**).

Forward Propagation

- Hypothesis $h(\mathbf{x})$ is computed using *forward propagation*

Forward propagation to compute $h(\mathbf{x})$:

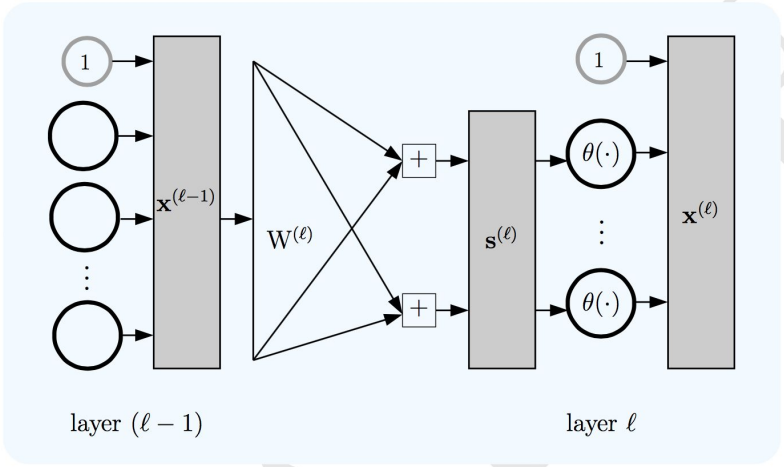
1: $\mathbf{x}^{(0)} \leftarrow \mathbf{x}$ **[Initialization]**

2: **for** $\ell = 1$ to L **do** **[Forward Propagation]**

3: $\mathbf{s}^{(\ell)} \leftarrow (\mathrm{W}^{(\ell)})^{\mathrm{T}}\mathbf{x}^{(\ell-1)}$

4: $\mathbf{x}^{(\ell)} \leftarrow \begin{bmatrix} 1 \\ \theta(\mathbf{s}^{(\ell)}) \end{bmatrix}$

5: $h(\mathbf{x}) = \mathbf{x}^{(L)}$ **[Output]**



$$\mathbf{x} = \mathbf{x}^{(0)} \xrightarrow{\mathrm{W}^{(1)}} \mathbf{s}^{(1)} \xrightarrow{\theta} \mathbf{x}^{(1)} \xrightarrow{\mathrm{W}^{(2)}} \mathbf{s}^{(2)} \xrightarrow{\theta} \mathbf{x}^{(2)} \cdots \longrightarrow \mathbf{s}^{(L)} \xrightarrow{\theta} \mathbf{x}^{(L)} = h(\mathbf{x})$$

Compute Weights Matrix:

$$\mathrm{W}^{(\ell)}: \; s_j^{(\ell)} = \sum_{i=0}^{d^{(\ell-1)}} w_{ij}^{(\ell)} x_i^{(\ell-1)}$$

$$\mathbf{s}^{(\ell)} = \left(\mathrm{W}^{(\ell)}\right)^{\mathrm{T}} \mathbf{x}^{(\ell-1)}$$

Compute Error $E_{in}(\mathbf{w})$:

$$E_{\mathrm{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}\left(h(\mathbf{x}_n;\mathbf{w}) - y_n\right)^2$$

$$= \frac{1}{N}\sum_{n=1}^{N}\left(\mathbf{x}_n^{(L)} - y_n\right)^2.$$

Problem Statement: Compute Gradient

- **Optimizer Goal**: Minimize $E_{\text{in}}(\mathbf{w})$ to obtain learned weights.

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathrm{e}_n.$$

- To compute gradient of E_{in} we need:

$$\frac{\partial E_{\text{in}}}{\partial \mathrm{W}^{(\ell)}} = \frac{1}{N}\sum_{n=1}^{N} \frac{\partial \mathrm{e}_n}{\partial \mathrm{W}^{(\ell)}},$$

- **Complexity**: $O(Q^2)$ where Q is number of weights.
- **Back Propagation**: Elegant DP algorithm with $O(Q)$.

Some Definitions

- **Chain rule** for partial derivatives of layer *l* using partial derivatives of layer (*l*+1).
- **Sensitivity vector $\boldsymbol{\delta}$** for layer *l* as gradient of error with respect to input signal ***s:***

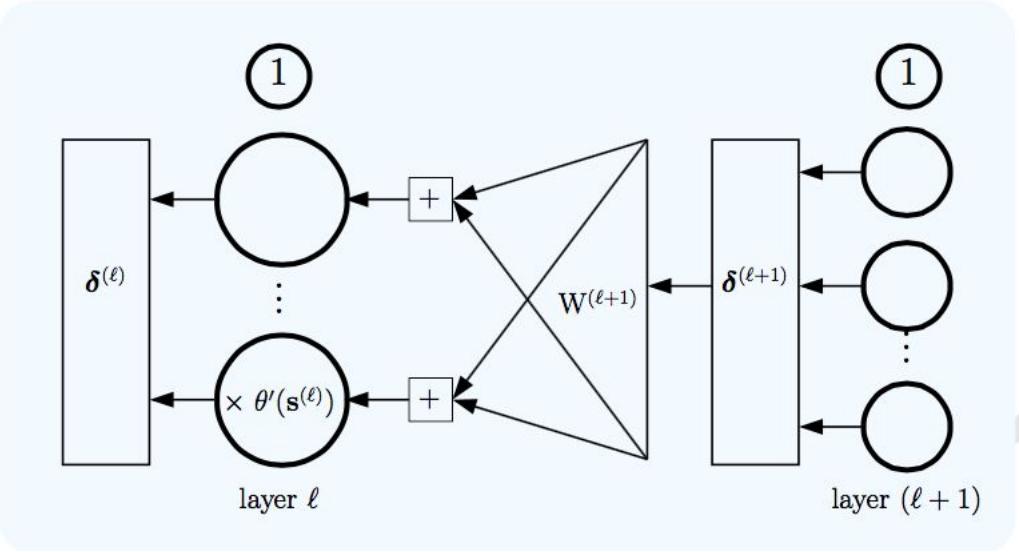
$$\boldsymbol{\delta}^{(\ell)} = \frac{\partial \mathrm{e}}{\partial \mathbf{s}^{(\ell)}}. \qquad \frac{\partial \mathrm{e}}{\partial \mathrm{W}^{(\ell)}} = \mathbf{x}^{(\ell-1)} (\boldsymbol{\delta}^{(\ell)})^{\mathrm{T}}.$$

- Partial derivatives have contributions from two components:
  - Output vector of the layer from which the weights originate;
  - Sensitivity vector of the layer into which the weights go;

- It turns out that the sensitivity vectors can be obtained by running a slightly modified version of the neural network backwards, and hence the name **back propagation**.

Essence of Back Propagation

- In forward propagation, each layer outputs the vector $\boldsymbol{x}^{(l)}$ and in back propagation, each layer outputs (backwards) the vector $\boldsymbol{\delta}^{(l)}$. In forward propagation, we compute $\boldsymbol{x}^{(l)}$ from $\boldsymbol{x}^{(l-1)}$ and in back propagation, we compute $\boldsymbol{\delta}^{(l)}$ from $\boldsymbol{\delta}^{(l+1)}$.
- For now, observe that if we know $\boldsymbol{\delta}^{(l+1)}$, then you can get $\boldsymbol{\delta}^{(l)}$. We use $\boldsymbol{\delta}^{(L)}$ to seed the backward process, and we can get that explicitly because : $\mathrm{e} = (\mathbf{x}^{(L)} - y)^2 = (\theta(\mathbf{s}^{(L)}) - y)^2$.



Forward propagation:

- Transformation was the sigmoid $\theta(\cdot)$

Back propagation:

- Transformation is multiplication by $\theta'(\boldsymbol{s}^{(l)})$

$$\boldsymbol{\delta}^{(\ell)} = \theta'(\mathbf{s}^{(\ell)}) \otimes [\mathbf{W}^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}]_1^{d^{(\ell)}}$$

$\otimes$ denotes component-wise multiplication

Back Propagation to Compute $\boldsymbol{\delta}^{(l)}$

Backpropagation to compute sensitivities $\delta^{(\ell)}$.
Input: a data point $(\mathbf{x}, y)$.

0: Run forward propagation on $\mathbf{x}$ to compute and save:

$$\begin{aligned} &\mathbf{s}^{(\ell)} \quad \text{for } \ell = 1, \ldots, L; \\ &\mathbf{x}^{(\ell)} \quad \text{for } \ell = 0, \ldots, L. \end{aligned}$$

1: $\delta^{(L)} \leftarrow 2(x^{(L)} - y)\theta'(s^{(L)})$ **[Initialization]**

$$\theta'(s^{(L)}) = \begin{cases} 1 - (x^{(L)})^2 & \theta(s) = \tanh(s); \\ 1 & \theta(s) = s. \end{cases}$$

2: **for** $\ell = L - 1$ to 1 **do** **[Back-Propagation]**

3: Let $\theta'(\mathbf{s}^{(\ell)}) = \left[1 - \mathbf{x}^{(\ell)} \otimes \mathbf{x}^{(\ell)}\right]_1^{d^{(\ell)}}$.

4: Compute the sensitivity $\boldsymbol{\delta}^{(\ell)}$ from $\boldsymbol{\delta}^{(\ell+1)}$:

$$\boldsymbol{\delta}^{(\ell)} \leftarrow \theta'(\mathbf{s}^{(\ell)}) \otimes \left[\mathrm{W}^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}\right]_1^{d^{(\ell)}}$$

Forward propagation:

- Compute $\boldsymbol{x}^{(l)}$ for $l = 0,\ldots,L$

Back propagation:

- Compute $\boldsymbol{\delta}^{(l)}$ for $l = 1,\ldots,L$

for $l = L$:

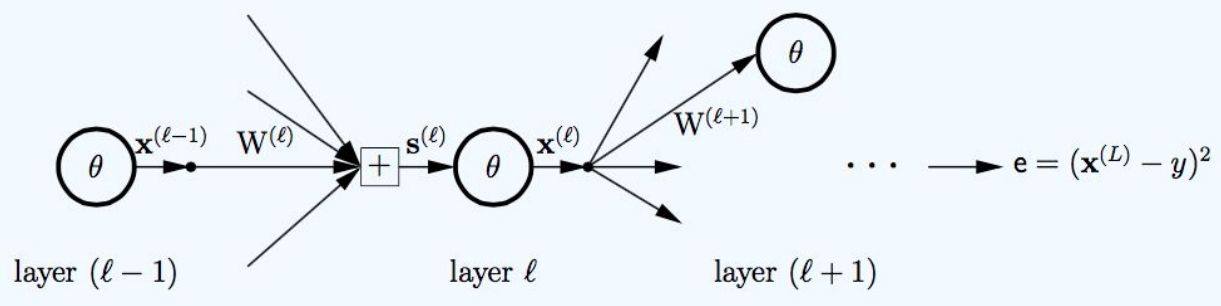
$$\begin{aligned} \boldsymbol{\delta}^{(L)} &= \frac{\partial \mathrm{e}}{\partial \mathbf{s}^{(L)}} \\ &= \frac{\partial}{\partial \mathbf{s}^{(L)}}(\mathbf{x}^{(L)} - y)^2 \\ &= 2(\mathbf{x}^{(L)} - y)\frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \\ &= 2(\mathbf{x}^{(L)} - y)\theta'(\mathbf{s}^{(L)}). \end{aligned}$$

Recursively we can see:

$$\boldsymbol{\delta}^{(1)} \longleftarrow \boldsymbol{\delta}^{(2)} \cdots \longleftarrow \boldsymbol{\delta}^{(L-1)} \longleftarrow \boldsymbol{\delta}^{(L)}.$$

Calculus of Back Propagation

- If you trust math, you can skip this ☺️. Let see:



- Chain of dependencies: $\mathrm{W}^{(\ell)} \longrightarrow \mathbf{s}^{(\ell)} \longrightarrow \mathbf{x}^{(\ell)} \longrightarrow \mathbf{s}^{(\ell+1)} \cdots \longrightarrow \mathbf{x}^{(L)} = h.$
- Chain rule:

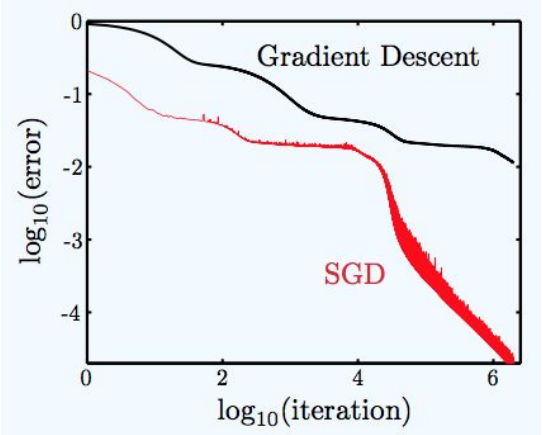
$$\frac{\partial \mathrm{e}}{\partial w_{ij}^{(\ell)}} = \frac{\partial \mathbf{s}_j^{(\ell)}}{\partial w_{ij}^{(\ell)}} \cdot \frac{\partial \mathrm{e}}{\partial \mathbf{s}_j^{(\ell)}} = \mathbf{x}_i^{(\ell-1)} \cdot \boldsymbol{\delta}_j^{(\ell)},$$

$$\boldsymbol{\delta}_j^{(\ell)} = \frac{\partial \mathrm{e}}{\partial \mathbf{s}_j^{(\ell)}} = \frac{\partial \mathrm{e}}{\partial \mathbf{x}_j^{(\ell)}} \cdot \frac{\partial \mathbf{x}_j^{(\ell)}}{\partial \mathbf{s}_j^{(\ell)}} = \theta'\left(\mathbf{s}_j^{(\ell)}\right) \cdot \frac{\partial \mathrm{e}}{\partial \mathbf{x}_j^{(\ell)}}.$$

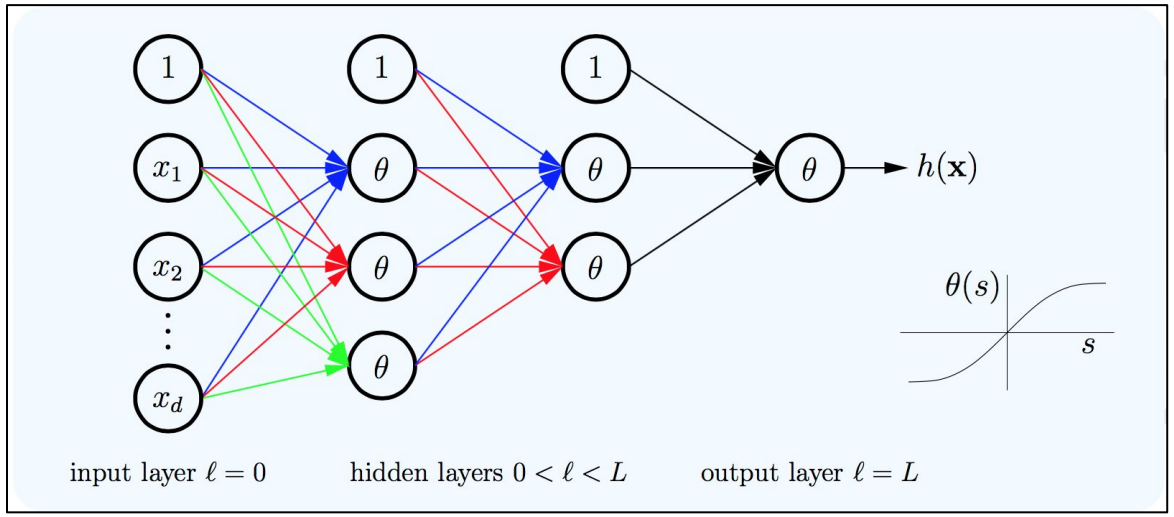
$$\frac{\partial \mathrm{e}}{\partial \mathbf{x}_j^{(\ell)}} = \sum_{k=1}^{d^{(\ell+1)}} \frac{\partial \mathbf{s}_k^{(\ell+1)}}{\partial \mathbf{x}_j^{(\ell)}} \cdot \frac{\partial \mathrm{e}}{\partial \mathbf{s}_k^{(\ell+1)}} = \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(\ell+1)} \boldsymbol{\delta}_k^{(\ell+1)}.$$

$$\boldsymbol{\delta}_j^{(\ell)} = \theta'(\mathbf{s}_j^{(\ell)}) \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(\ell+1)} \boldsymbol{\delta}_k^{(\ell+1)},$$

Summary of Back Propagation

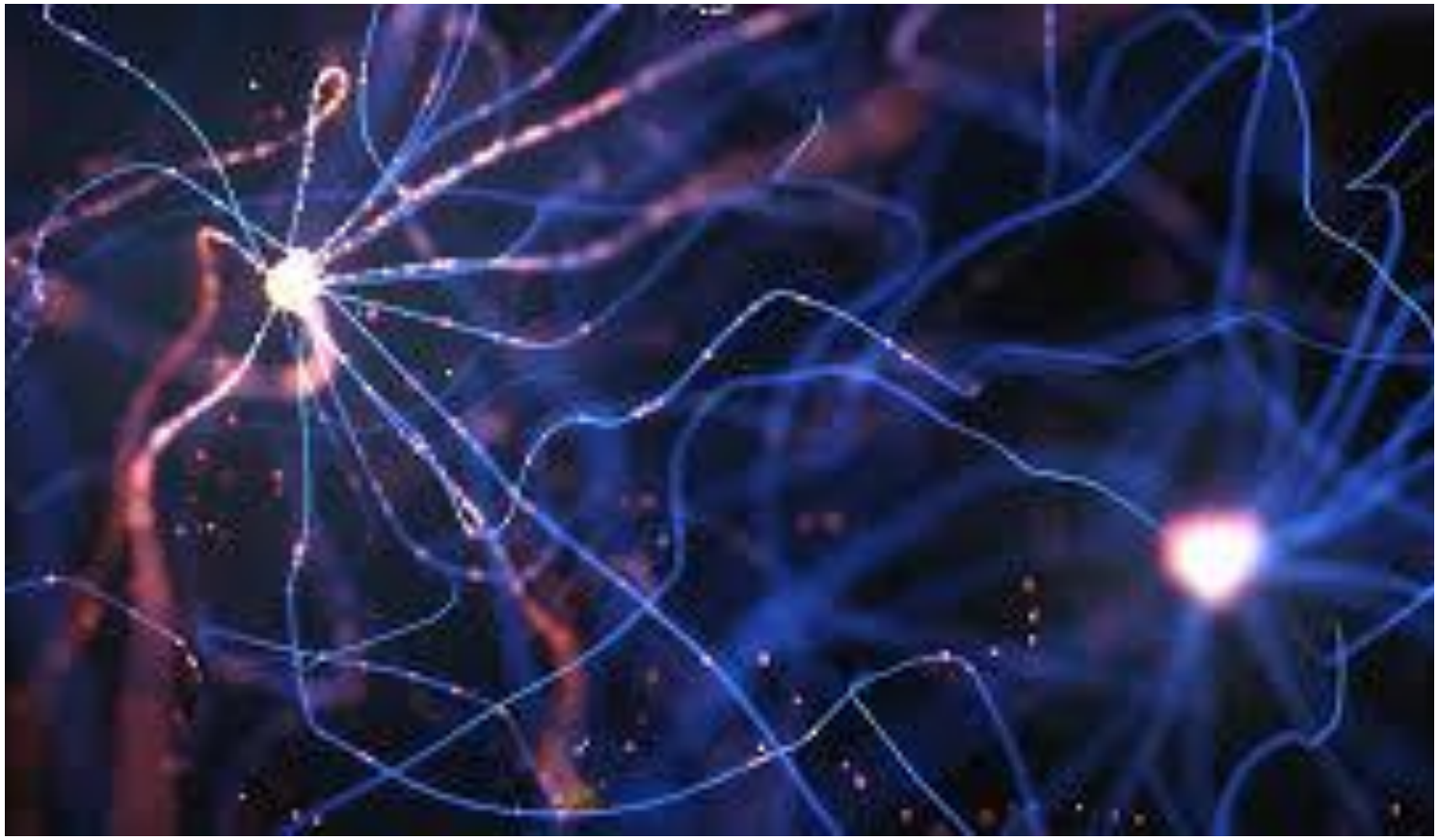


- Setup network architecture & hyper parameters
- Compute $\boldsymbol{x}^{(l)}$ using forward propagation
- Compute sensitivities $\boldsymbol{\delta}^{(l)}$ using back propagation
- Compute in-sample error and gradient on data point $\mathbf{x}_n$ as $\mathrm{G}^{(l)}(\mathbf{x}_n)$



Algorithm to Compute $E_{\text{in}}(\mathbf{w})$ and $\mathbf{g} = \nabla E_{\text{in}}(\mathbf{w})$.
Input: $\mathbf{w} = \{\mathrm{W}^{(1)}, \ldots, \mathrm{W}^{(L)}\}$; $\mathcal{D} = (\mathbf{x}_1, y_1) \ldots (\mathbf{x}_N, y_n)$.
Output: error $E_{\text{in}}(\mathbf{w})$ and gradient $\mathbf{g} = \{\mathrm{G}^{(1)}, \ldots, \mathrm{G}^{(L)}\}$.

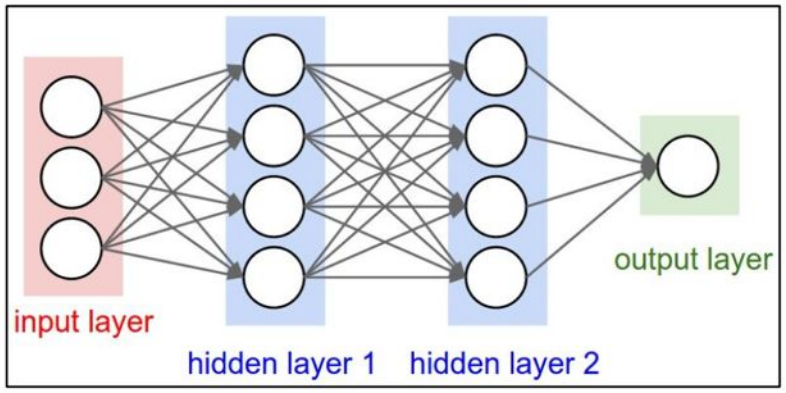
1: Initialize: $E_{\text{in}} = 0$ and $\mathrm{G}^{(\ell)} = 0 \cdot \mathrm{W}^{(\ell)}$ for $\ell = 1, \ldots, L$.
2: **for** Each data point $(\mathbf{x}_n, y_n)$, $n = 1, \ldots, N$, **do**
3: Compute $\mathbf{x}^{(\ell)}$ for $\ell = 0, \ldots, L$. [forward propagation]
4: Compute $\boldsymbol{\delta}^{(\ell)}$ for $\ell = L, \ldots, 1$. [backpropagation]
5: $E_{\text{in}} \leftarrow E_{\text{in}} + \frac{1}{N}(\mathbf{x}^{(L)} - y_n)^2$.
6: **for** $\ell = 1, \ldots, L$ **do**
7: $\mathrm{G}^{(\ell)}(\mathbf{x}_n) = [\mathbf{x}^{(\ell-1)}(\boldsymbol{\delta}^{(\ell)})^{\mathrm{T}}]$
8: $\mathrm{G}^{(\ell)} \leftarrow \mathrm{G}^{(\ell)} + \frac{1}{N}\mathrm{G}^{(\ell)}(\mathbf{x}_n)$



Agenda

- Understanding Problem
- How Neural Network Works
- Error / Loss Function
- Back Propagation Algorithm
- Heuristic Training Dynamic
- Implementation
- Q&A

Training Deep Neural Networks

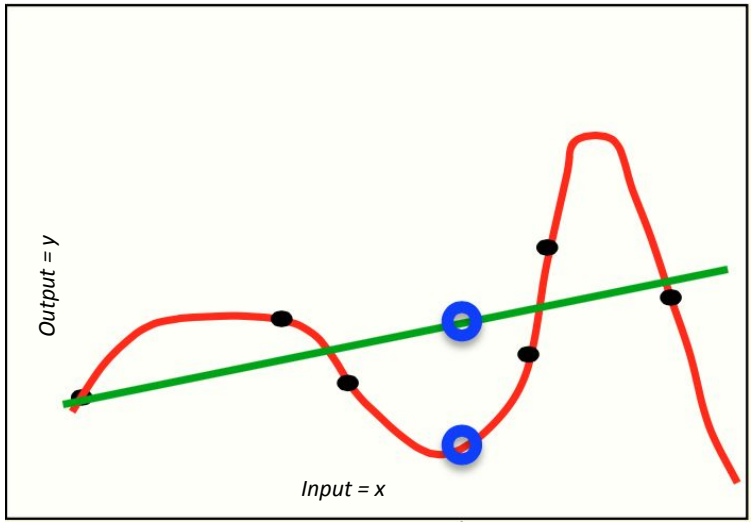


Mini-Batch SGD Loop:

1. Sample a batch of data
2. **Forward prop** it through the network
3. Get the loss / in-sample error
4. **Backprop** to calculate gradient
5. Update parameters using gradient

Heuristics

1. **Software Packages**
2. **One Time Setup**
   - Activation Functions
   - Data Preprocessing
   - Weight Initialization
   - Regularization
   - Gradient Descent Optimizers
3. **Training Dynamics**
   - Babysitting the Learning Process
   - Monitor and Visualize
   - Learning Rate Schedules
   - Hyperparameter Optimization
4. **Evaluation**
   - Model Ensembles



Which output value should you predict

for this test input?

Overfitting Problem

- **Overfitting** : the downside of complex model
- Training data contains two type of noise:
  - **Unreliable target values**
  - **Sampling error**
- When we fit the model, it cannot tell which regularities are real and which are caused by sampling error.
- **Which model we trust?**
  - Complicated model?
  - Simple model?
  - If the model is very flexible it can model the sampling error really well. This is a disaster.

Limit the size of weights by penalizing large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty).

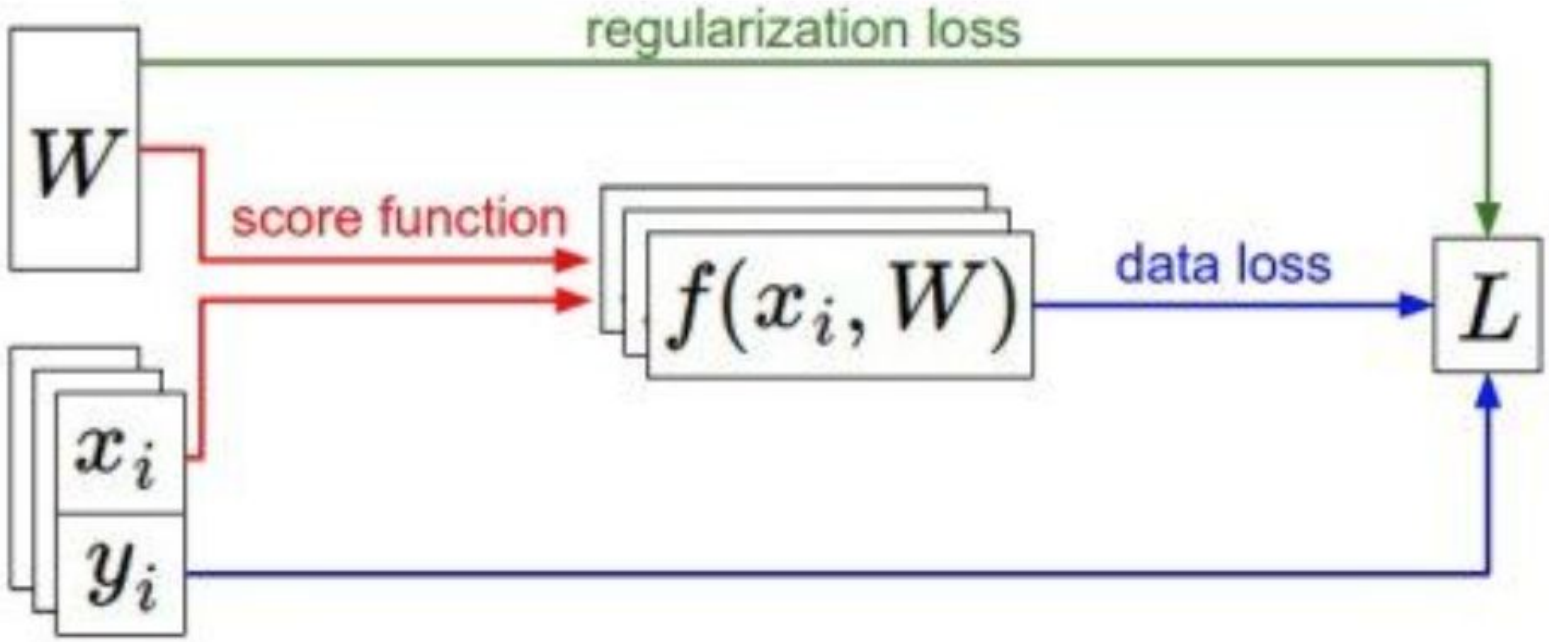
How to Handle Overfitting

Many Techniques Available

- Get more data
- Use right capacity model
- **Weight-decay / regularization**
- Weight-sharing
- Early stopping
- Model averaging
- Bayesian fitting of neural nets
- Dropout
- Generative pre-training

Weights Initialization

- **Initial Question**: What happen when $W = 0$ init is used?
- **First idea**:
  - Small random numbers (gaussian with zero mean and 0.01 sd)
  - Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.
- **Proper Initialization is active area of research:**



We have some dataset of (x, y)

We have a score function: $s = f(x; W) = Wx$

Softmax: $E_i = -\log(\frac{}{\sum_j e^{s_j}})$

SVM: $E_i = \sum_{j \neq y_i} \max(0, sj - syi + 1)$

Full Loss: $E = L = \frac{1}{N} \sum_{i=1}^{N} E_i + R(W)$

Regularization

Weight Regularization

$$E = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq yi}\max(0, f(x_i; W)_j - f(x_i; W)_{yi} + 1) + \lambda R(W)$$

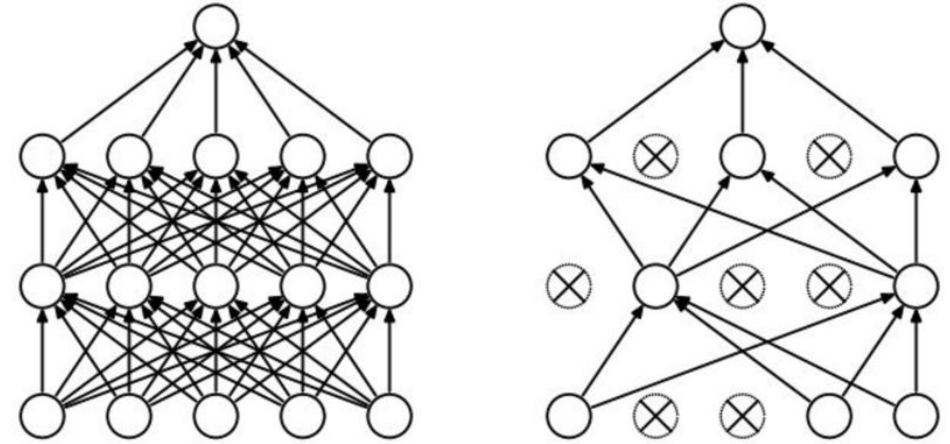
Regularization Strength

In common use:

- L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$
- L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$
- Elastic Net (L1+L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$
- Max Norm Regularization
- Dropout

Regularization (Dropout)

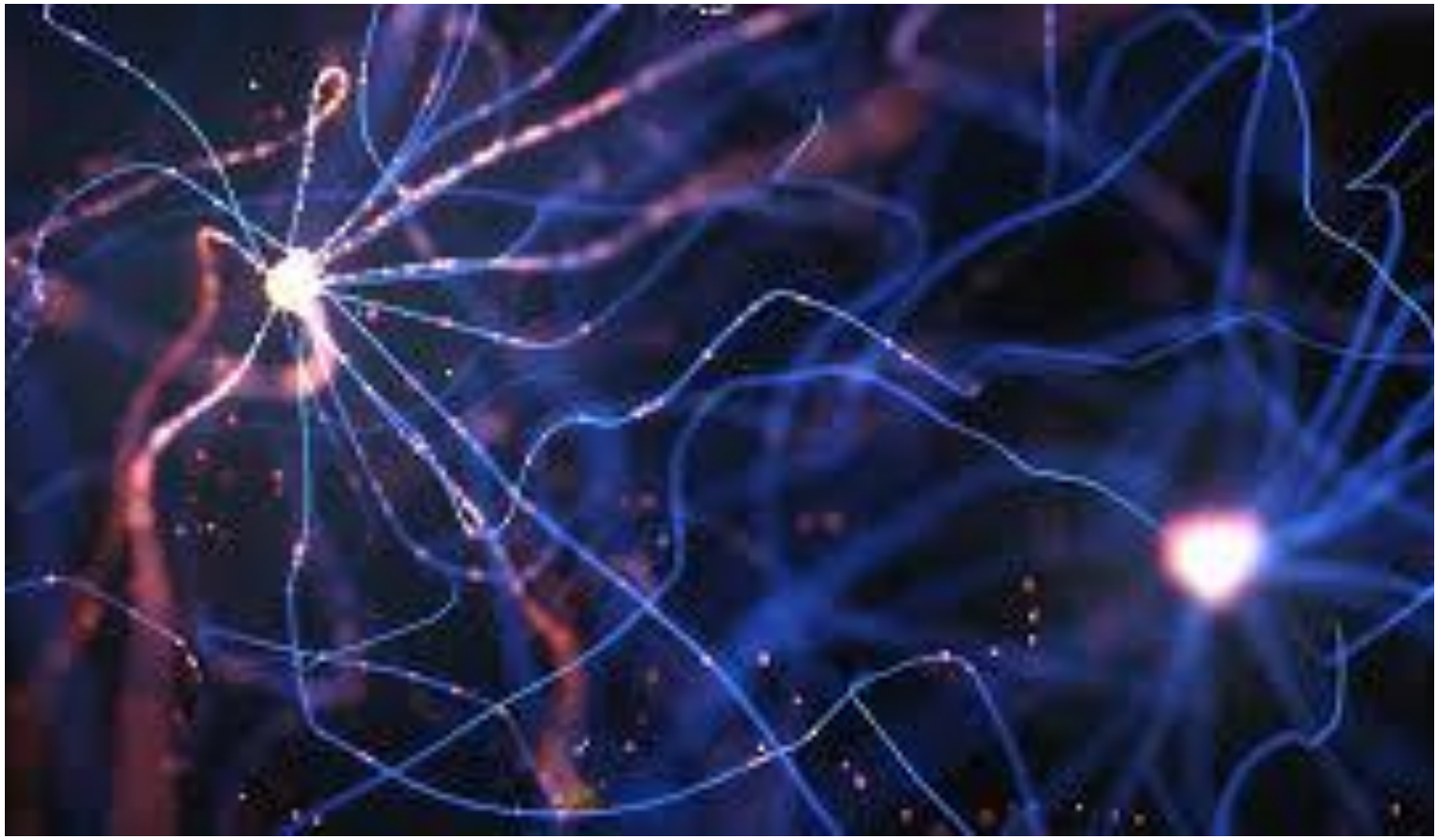
- Randomly set some neurons to zero in the forward pass.
- Monte Carlo Approximation: do many forward passes with different dropout masks, average all predictions.
- How could this possibly be a good idea?



(a) Standard Neural Net (b) After applying dropout.

Srivastava et al., 2014

Forces the network to have a redundant representation.

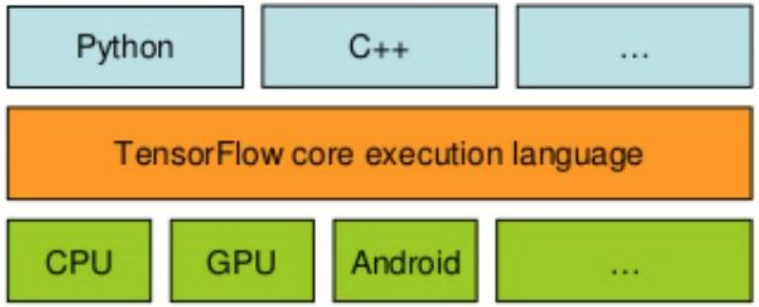


Agenda

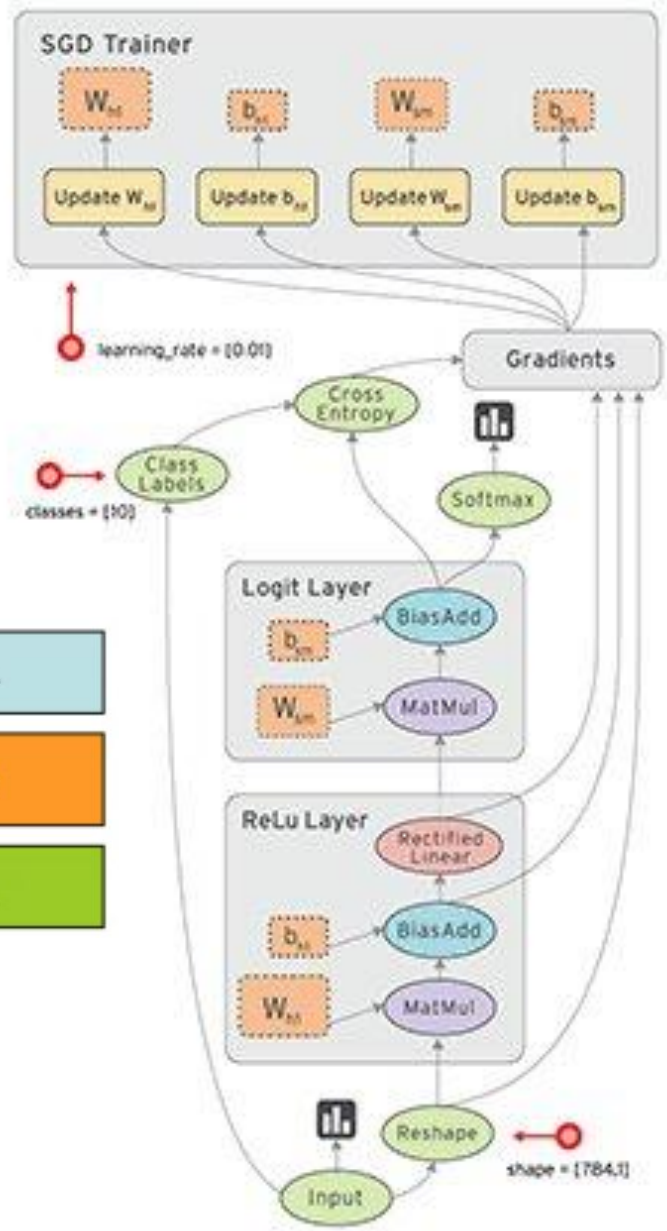
- Understanding Problem
- How Neural Network Works
- Error / Loss Function
- Back Propagation Algorithm
- Heuristic Training Dynamic
- Implementation
- Q&A

Introduction to TensorFlow

- **Tensor**
  - **Definition**: An array with more than two axes
  - Arbitrary dimensionality array
- **Directed Graph**
  - Used describes computation
  - **Node**: Instantiation of an operation
- **Operation**
  - An abstract computation
  - Have attributes
- **Kernel**
  - Particular implementation of an operation
  - Run on a type of device (e.q. CPU, GPU)
- **Variable**
  - Special operation to persistent mutable tensor
- **Session**
  - Created to interact with TensorFlow system

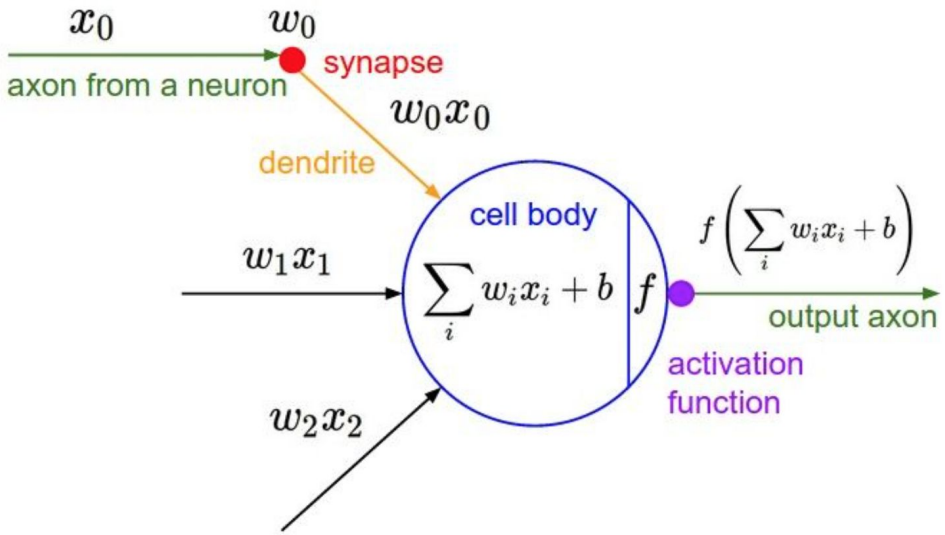


[Download at www.tensorflow.org](www.tensorflow.org)



Activation Functions

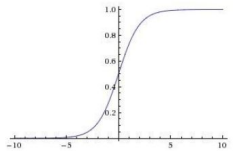
- **Smooth Nonlinearities:**
  - Sigmoid
  - Tanha
  - Elu
  - Softplus
  - SoftSign
- **Continuous Differentiable:**
  - ReLu
  - Relu6
  - CreLu
  - Relu-X
- **Random Regularization:**
  - Dropout



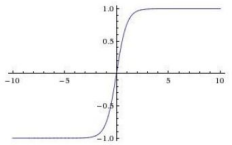
Activation Functions

Sigmoid

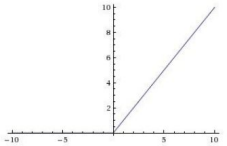
$$\sigma(x) = 1/(1+e^{-x})$$



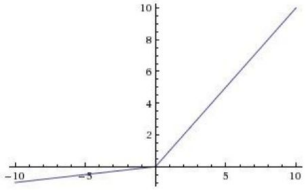
tanh tanh(x)



ReLU max(0,x)

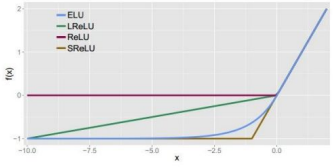


Leaky ReLU
max(0.1x, x)



Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$

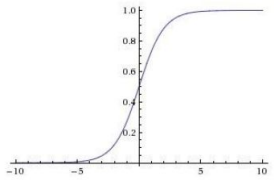


TensorFlow Activation Functions

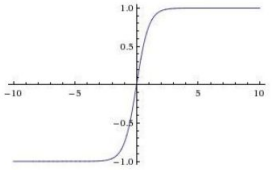
Activation Functions

Sigmoid

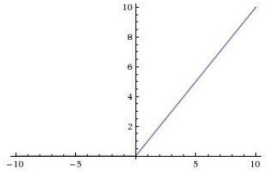
$\sigma(x) = 1/(1 + e^{-x})$



tanh tanh(x)

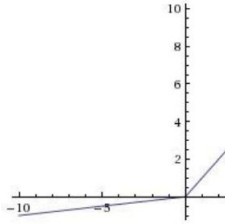


ReLU max(0,x)



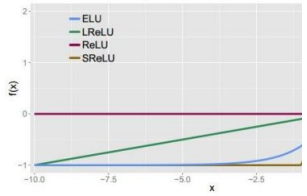
Leaky Rel

max(0.1x,



Maxout $\max(w_1^T x + b$

ELU $f(x) = \begin{cases} x \\ \alpha\,(\exp(x) - 1) \end{cases}$



- **Smooth Nonlinearities:**
  - Sigmoid
  - Tanh
  - Elu
  - Softplus
  - SoftSign
- **Continuous Differentiable:**
  - ReLu
  - Relu6
  - CreLu
  - Relu-X
- **Random Regularization:**
  - Dropout

How to Choose Activation Function

RECOMMENDATION:

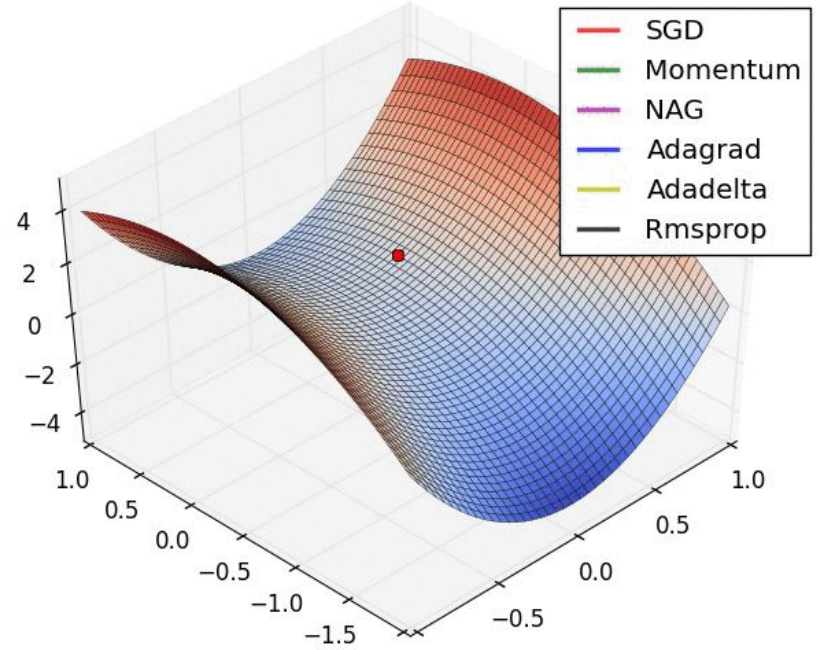
1. Use ReLU. Be carefull with learning rate.
2. Try out Leaky ReLU / Maxout / ELU.
3. Try out tanh but don't expect much.
4. Don't use sigmoid.

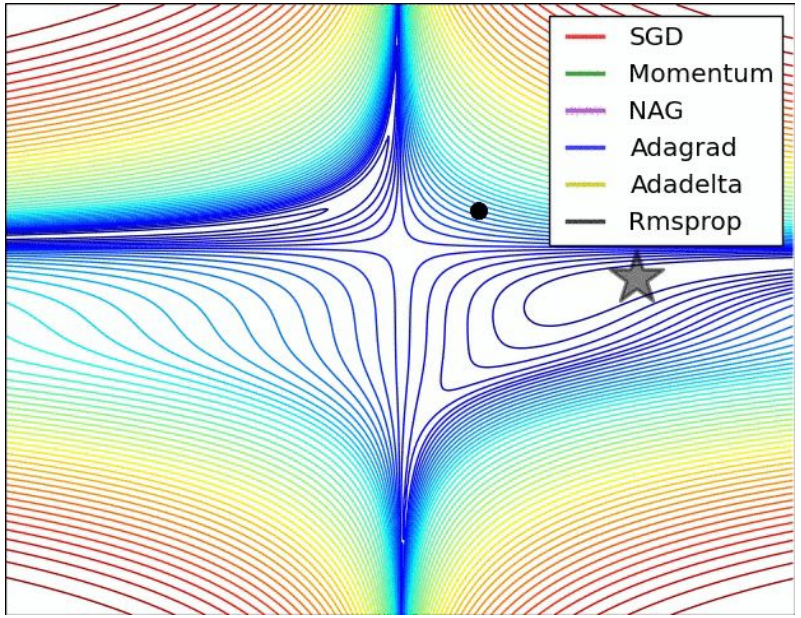
- **Sigmoid**. Historically popular since they have nice interpretation as saturating "firing rate" of a neuron.
  - (-) Saturated neurons kill gradients.
  - (-) Sigmoid outputs are not zero centered.
  - (-) Exponential is expensive computation.
- **Tanh**. Still kill gradients when saturated.
- **ReLU** (Rectifier Linear Unit).
  - (+) Doesn't saturate (in +region).
  - (+) Computationally efficient.
  - (+) Converges faster than sigmoid/tanh in practices (e.g.6x)
  - (-) Not zero centered output.
- **Leaky ReLU**. All benefits of ReLU but better. It will not "die".
- **ELU**. All benefits of ReLU. Doesn't die. Closer to zero mean outputs.
- **Maxout Neuron**. All benefits of ReLU and Leaky ReLU but doubles the number of parameters.

TensorFlow Optimizers

- **Gradient Descent Variants**:
  - Batch, Stochastic, Mini-Batch
- **Gradient Descent Optimization Algorithms:**
  1. **GD**: Basic Gradient Descent Algorithm
  2. **AdaDelta**: Ada Delta Algorithm
  3. **AdaGrad**: Adaptive Subgradient Algorithm
  4. **AdaGradDA**: Adagrad Dual Averaging Algorithm
  5. **Momentum**: SGD Momentum Algorithm
  6. **Adam**: Adam Algorithm
  7. **FTRL**: FTRL Algorithm
  8. **RMSProp**: RMSProp Algorithm

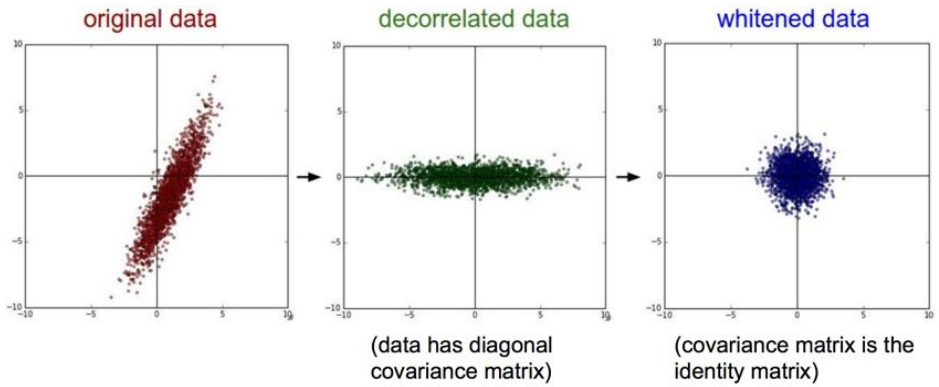
Papers listed in TF documentation

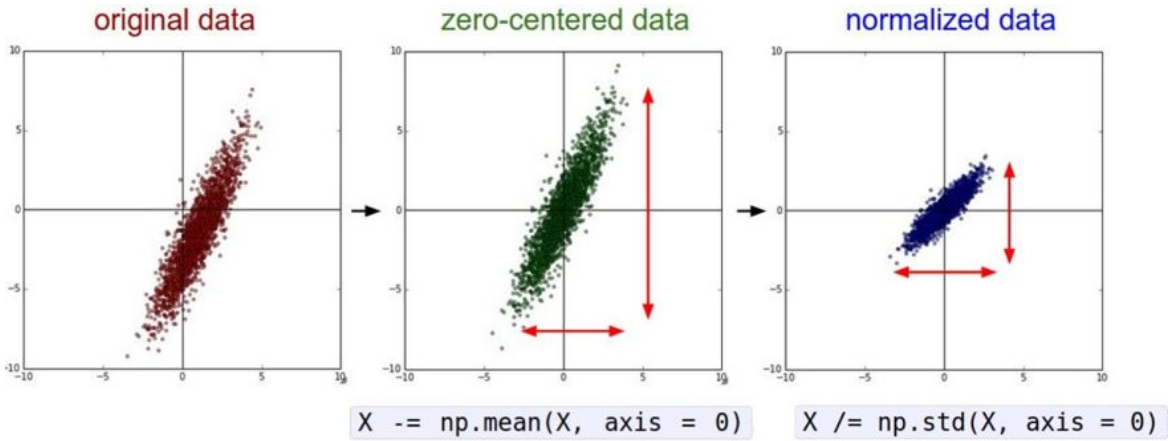




Data Preprocessing

- Normalization
- PCA and Whitening

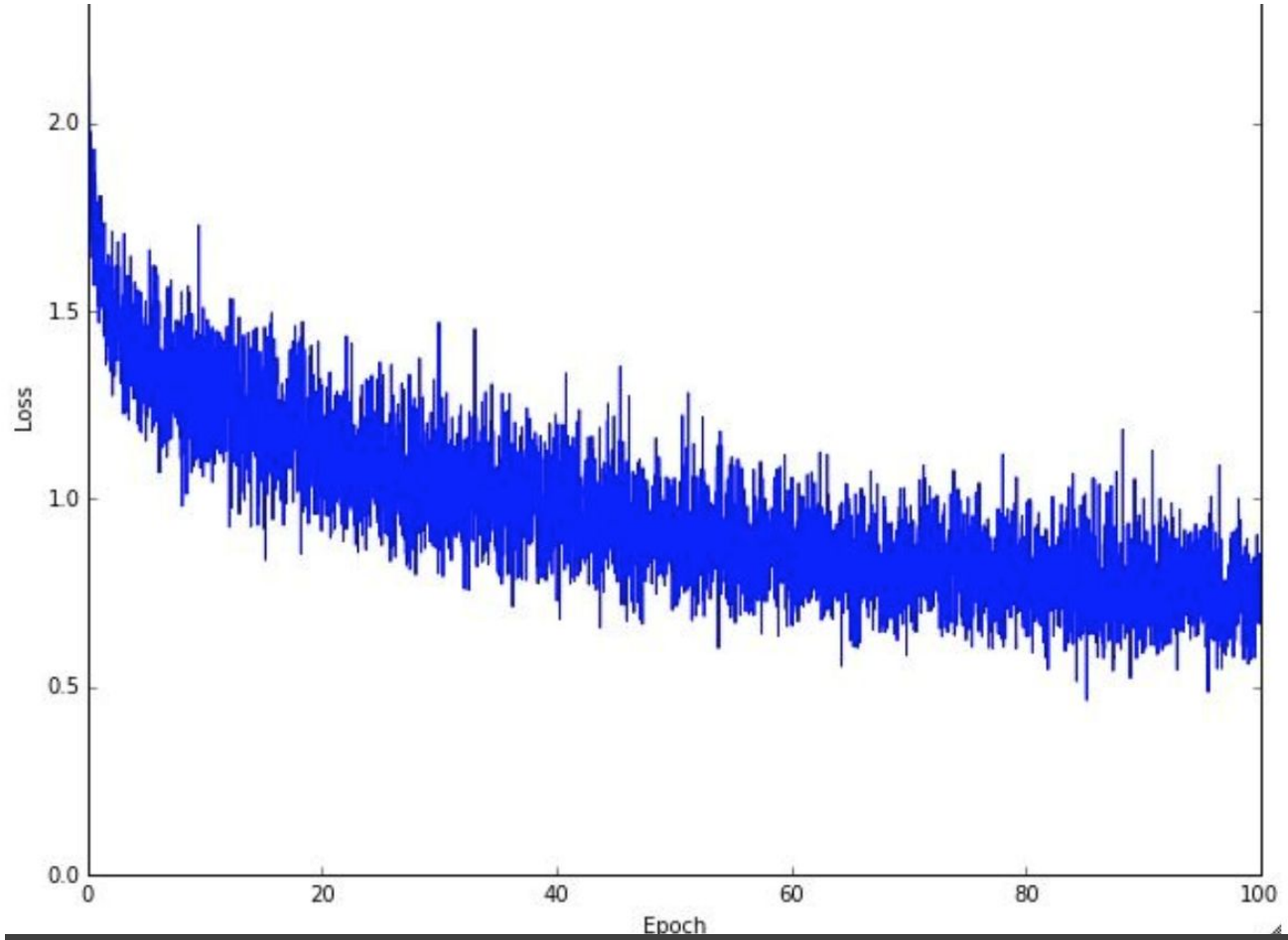


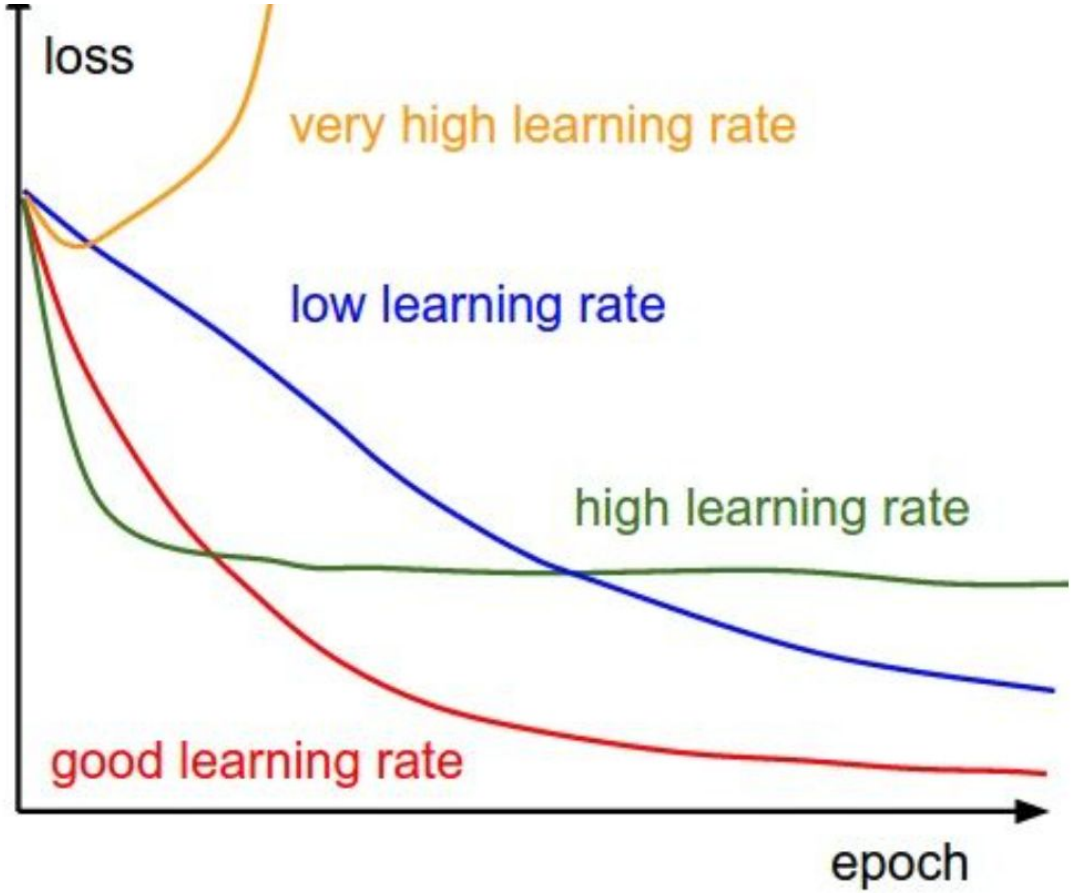


Note: We will have session about this!

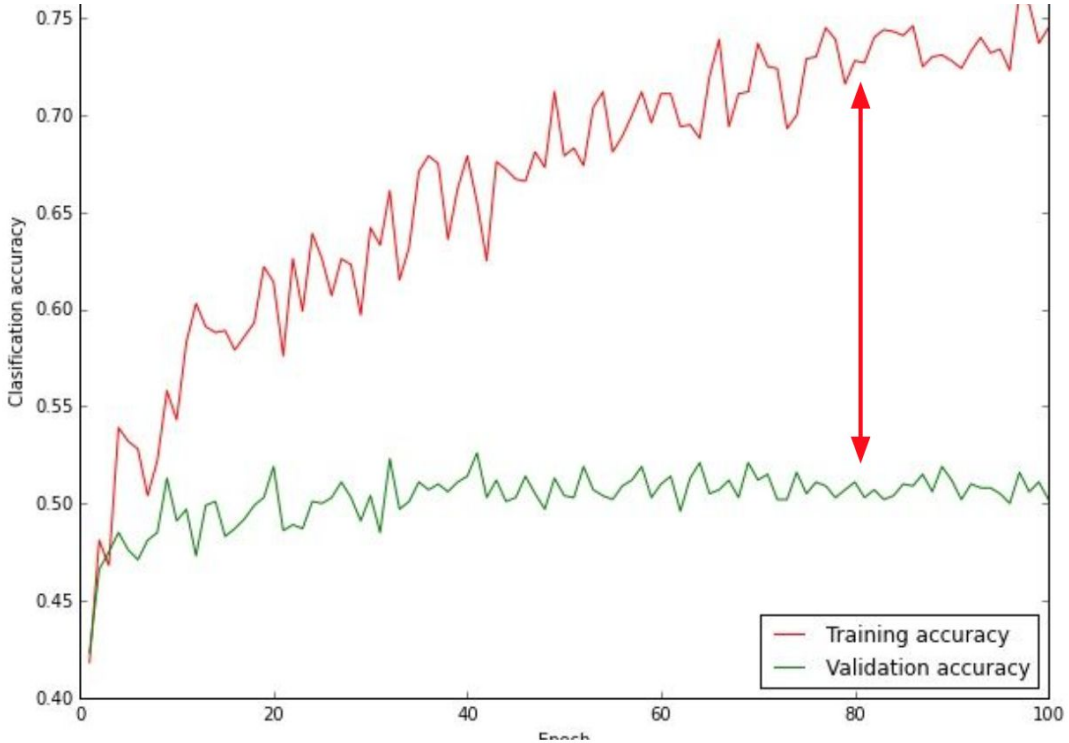
Babysitting the Learning Process

- Pre-process the data
- Choose network architecture (DNN, CNN, RNN, etc)
- Double check the cost/loss is reasonable
- Adjust regularization (L1, L2, etc)
- Find learning rate that makes loss down
  - If loss not going down or barely changing, learning rate too low
  - If loss exploding, learning rate too high
- Monitor and visualize loss curve
- Monitor and visualize accuracy





Monitoring Loss Values



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

Monitor and Visualize Accuracy

Hyperparameter Optimization

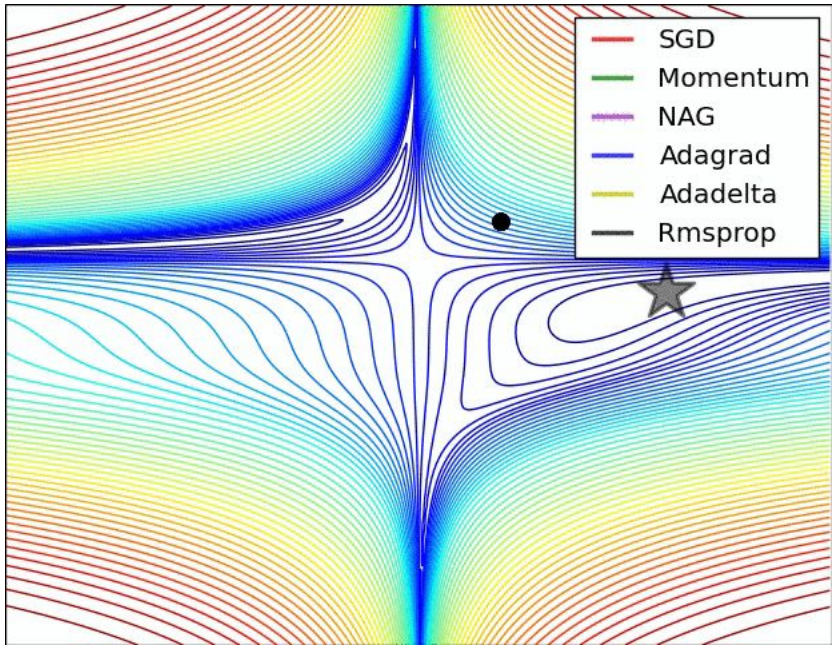
- Coarse-Fine Cross Validation Strategy
  - **First stage**: only a few epochs to get rough idea of what params work
  - **Second stage**: longer running time, finer search …. (repeat as necessary)
- Hyperparameters to play with:
  - Network Architecture
  - Learning Rate, Its Decay Schedule, Update Type
  - Regularization (L2/Dropout Strength)

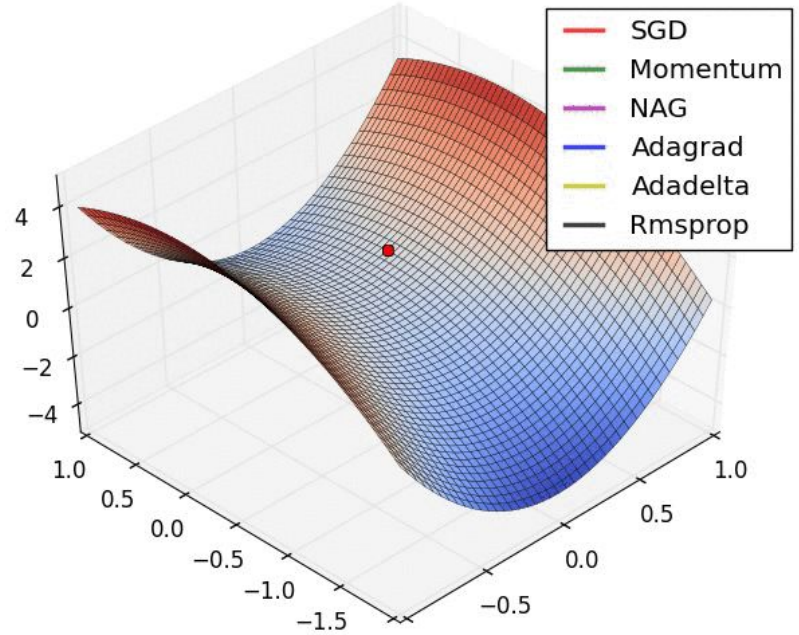
Gradient Descent Optimizers

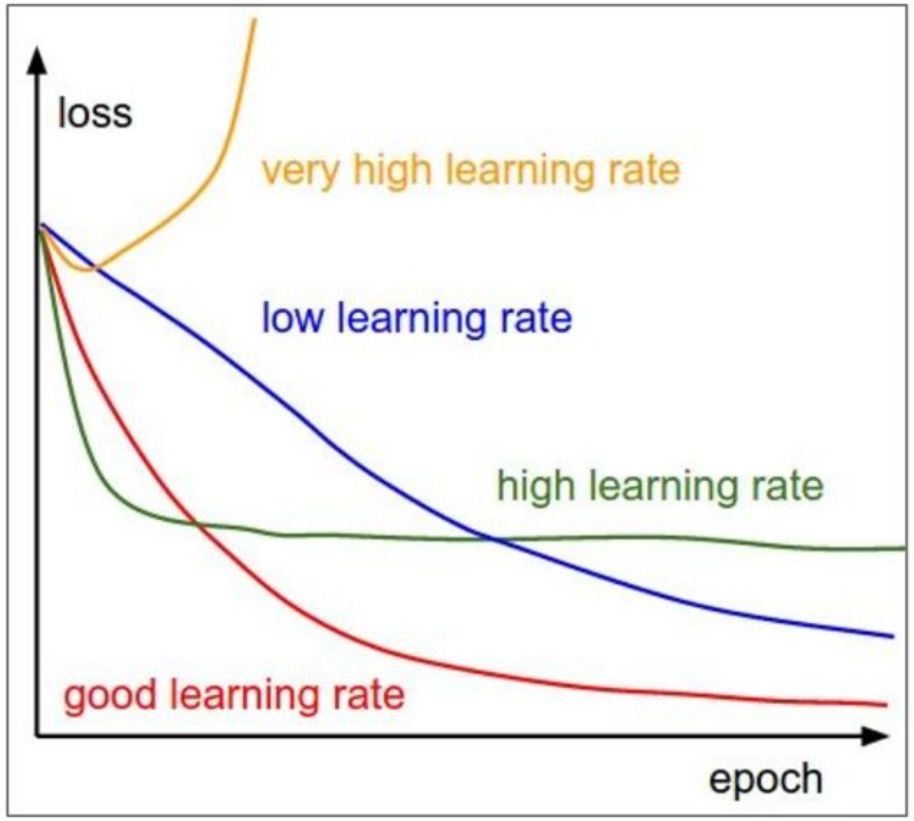
- **Gradient Descent Variants**:
  - Batch, Stochastic, Mini-Batch
- **Gradient Descent Optimization Algorithms:**
  1. **GD**: Basic Gradient Descent Algorithm
  2. **AdaDelta**: Ada Delta Algorithm
  3. **AdaGrad**: Adaptive Subgradient Algorithm
  4. **AdaGradDA**: Adagrad Dual Averaging Algorithm
  5. **Momentum**: SGD Momentum Algorithm
  6. **Adam**: Adam Algorithm
  7. **FTRL**: FTRL Algorithm
  8. **RMSProp**: RMSProp Algorithm

Papers listed in TF documentation. Adam is good default choice.

Read More: [Gradient Descent Optimization Algorithms]



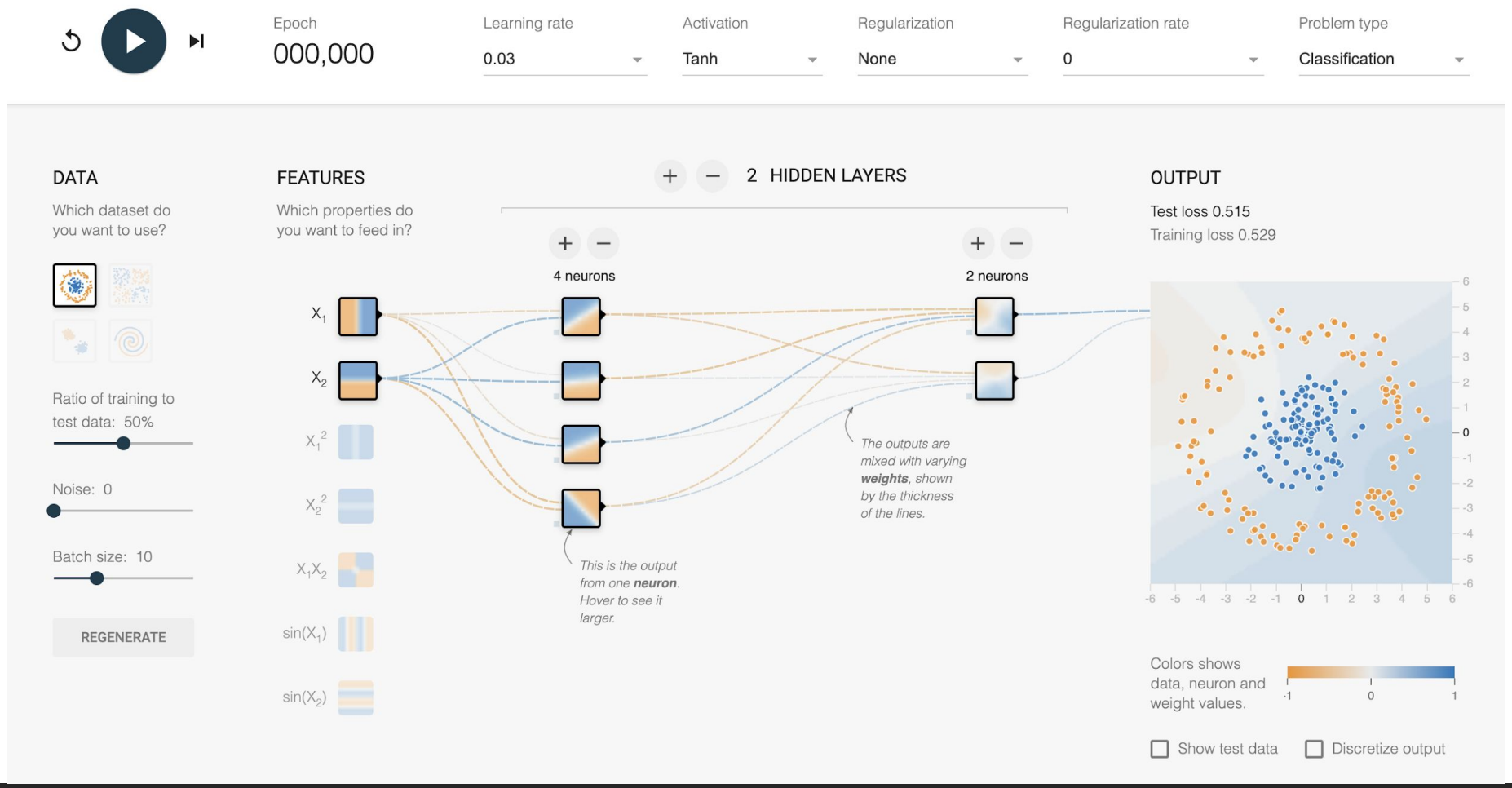





```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
```

Choosing Learning Rate

- All optimizers all have learning rate as hyperparameter.
- Learning rate decay over time.
- We have to try the learning rate values.
- Which one of these learning rates is best?
- Find the good learning rate. Read more papers.



Neural Network Playground

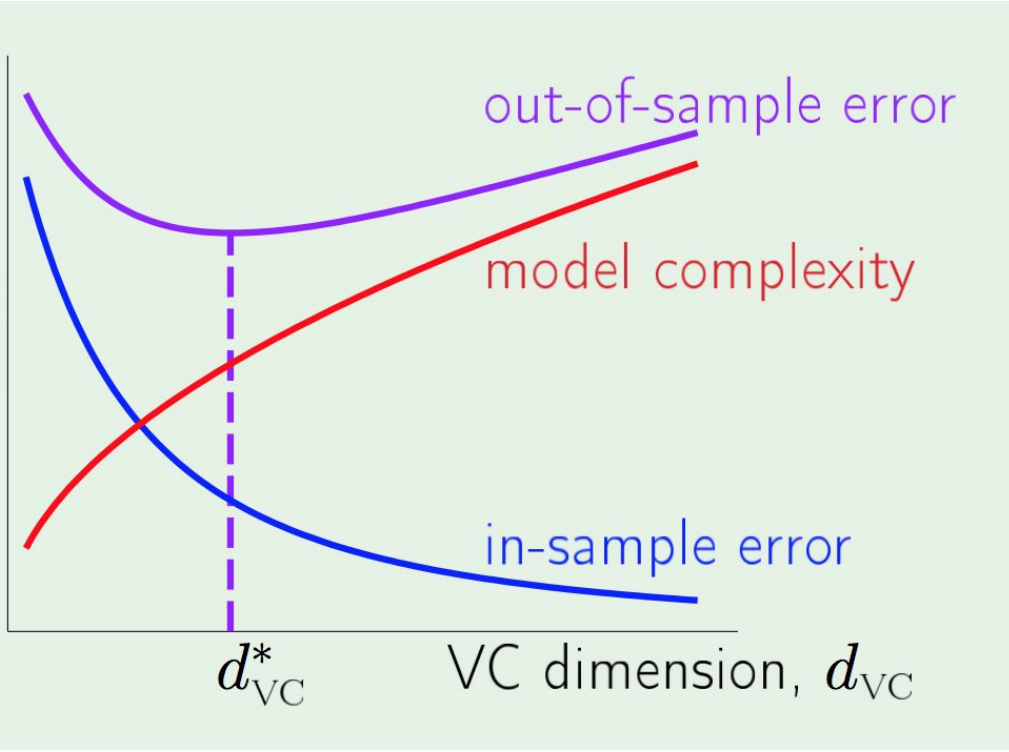
What The Theory Tell Us

Characterizing The Following:

- The feasibility of learning *infinite M*.
- The tradeoff:

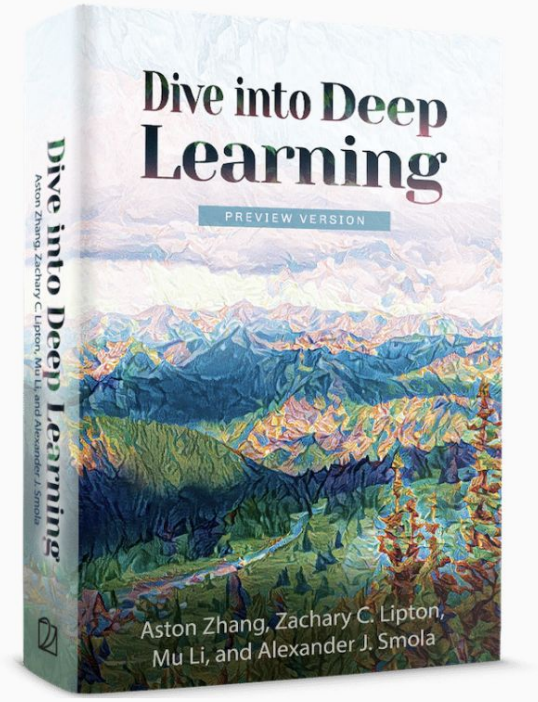
Model complexity ↑	E_{in} ↓
Model complexity ↑	$E_{out} - E_{in}$ ↑

See you in next lecture! Stay Tuned!



Home Work

Please reproduce Chapter 3 and 4 of www.d2l.ai

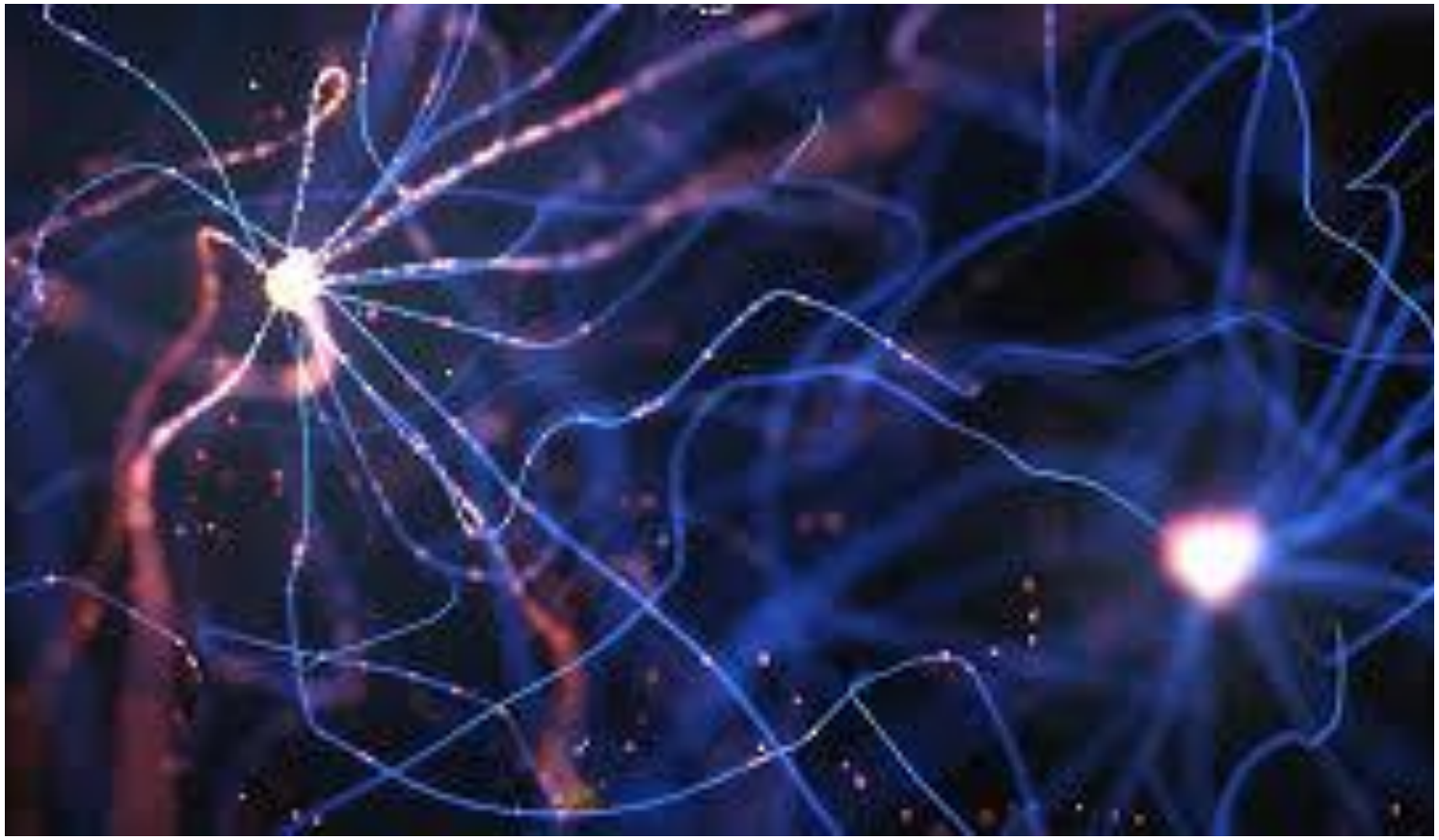


Dive into Deep Learning

Interactive deep learning book with code, math, and discussions

Implemented with **NumPy/MXNet**, **PyTorch**, and **TensorFlow**

Adopted at 200 universities from 50 countries



Q&A

- Understanding Problem
- How Neural Network Works
- Error / Loss Function
- Back Propagation Algorithm
- Heuristic Training Dynamic
- Implementation
- Q&A