Big Data

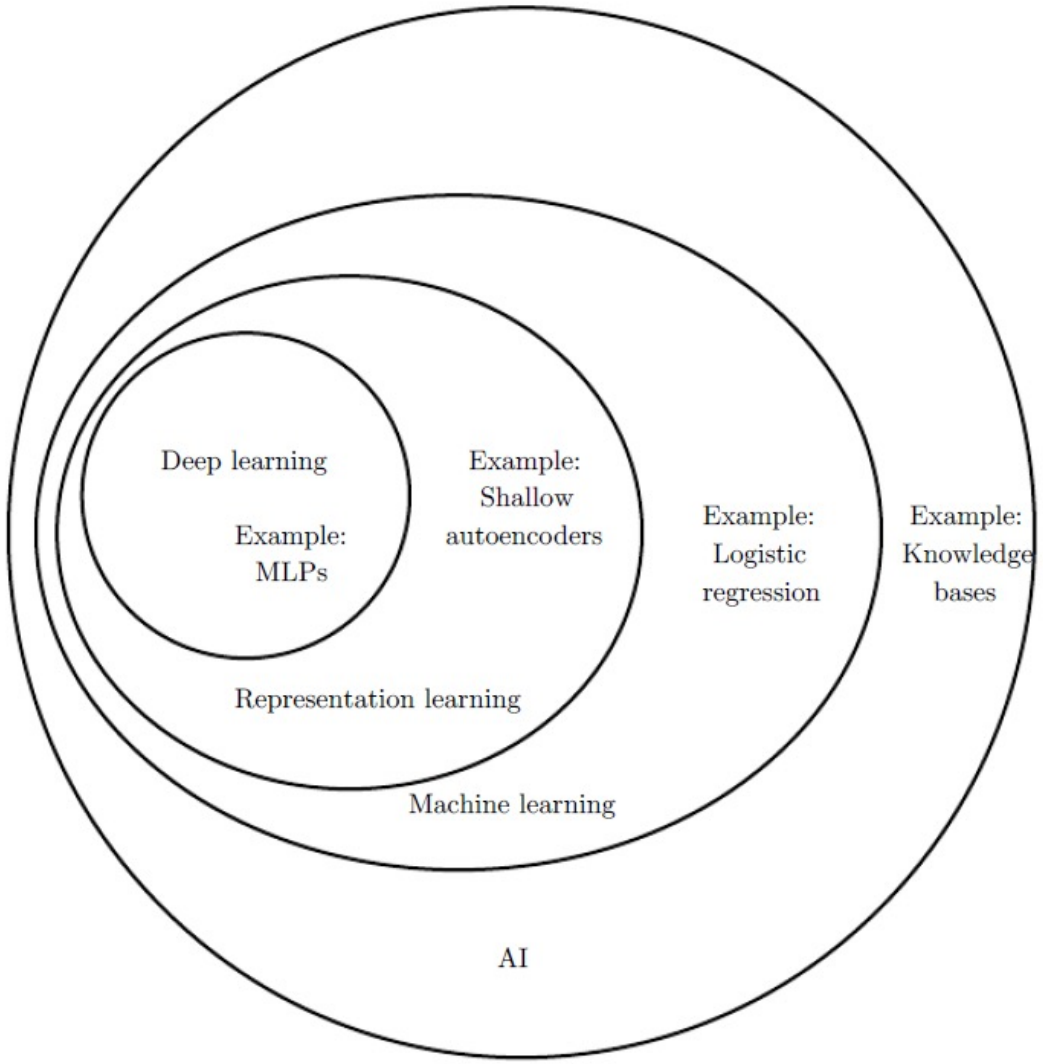
CSI4121

Ch 1. Data Representation

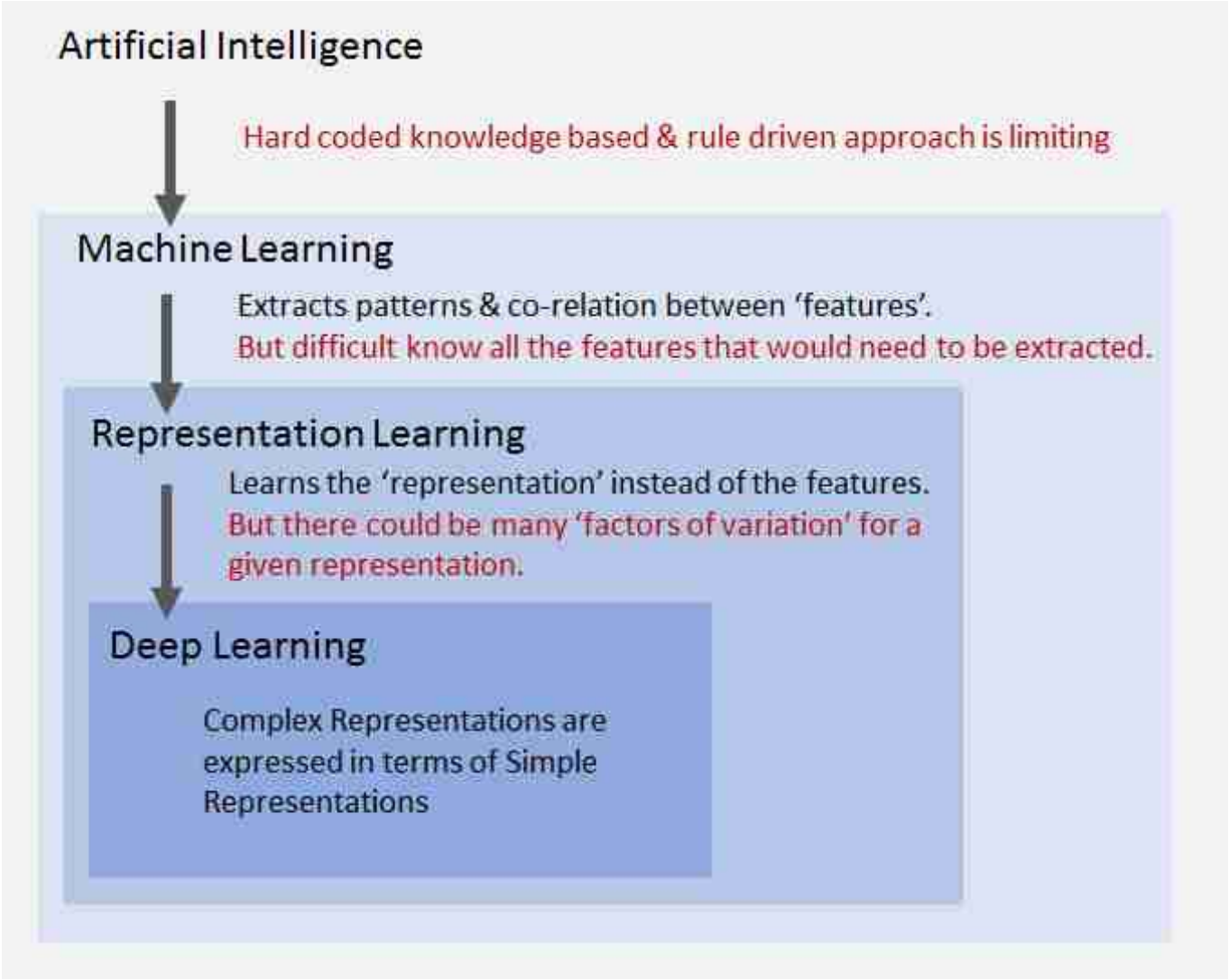
Jinyoung Yeo

Yonsei AI

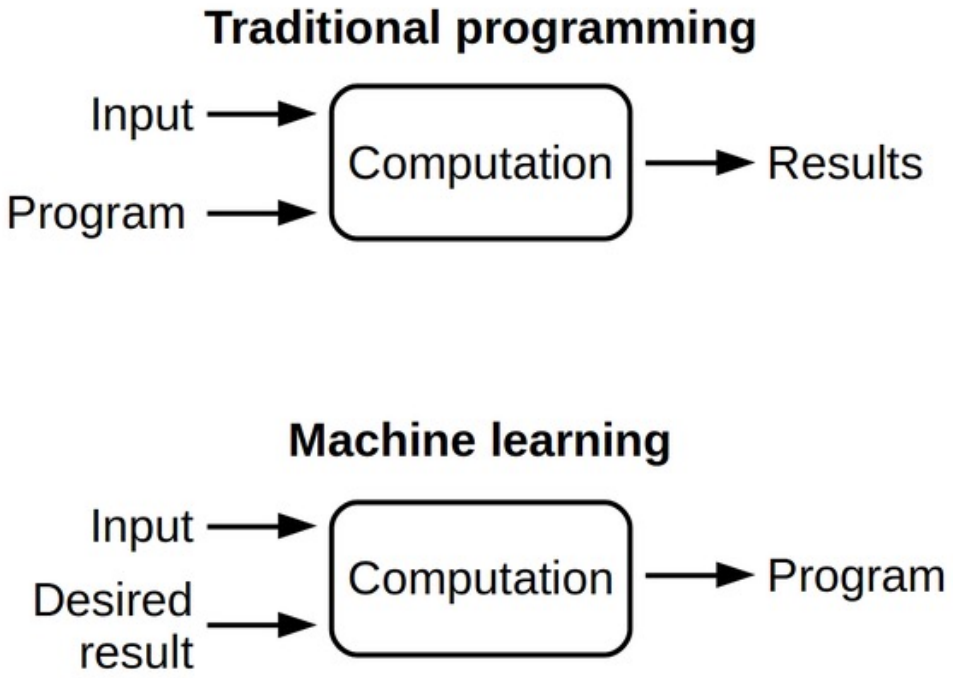
Relationships between DL, representation learning, ML, and AI



Relationships between DL, representation learning, ML, and AI



Machine Learning



Machine Learning

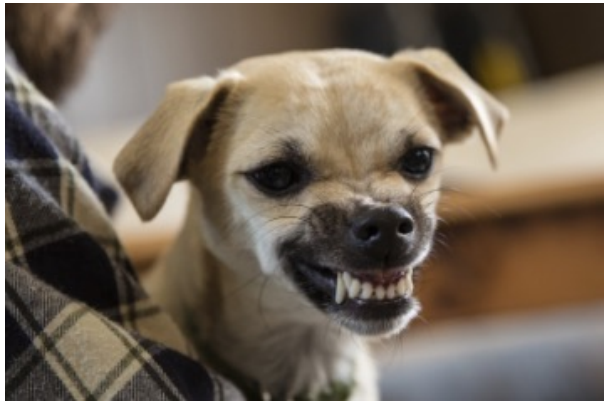
- Task
- Experience
- Performance Measure
- Program = Model

Overview [edit]

The name *machine learning* was coined in 1959 by Arthur Samuel.[5] Tom M. Mitchell provided a widely quoted, more formal definition of the algorithms studied in the machine learning field: "A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P* if its performance at tasks in *T*, as measured by *P*, improves with experience *E*."[6] This definition of the tasks in which machine learning is concerned offers a fundamentally operational definition rather than defining the field in cognitive terms. This follows Alan Turing's proposal in his paper "Computing Machinery and Intelligence", in which the question "Can machines think?" is replaced with the question "Can machines do what we (as thinking entities) can do?".[7] In Turing's proposal the various characteristics that could be possessed by a *thinking machine* and the various implications in constructing one are exposed.

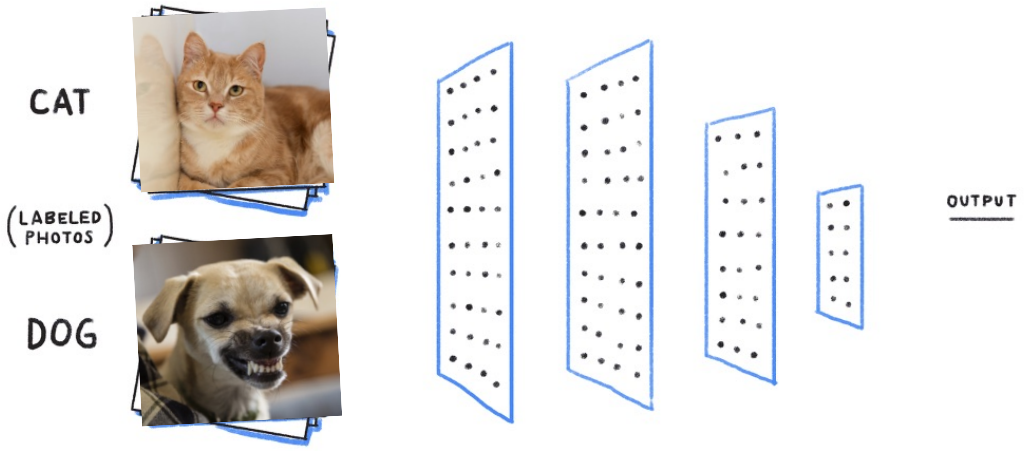
Task and Experience: Training Data

- Everything starts with a dataset!
- Inputs: $x = \{x_1, x_2, x_3, \ldots, x_N\}$
- Output labels: $\mathrm{y} = \{y_1, y_2, y_3, \ldots, y_N\}$
- Dataset $\mathrm{D} = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_N, y_N)\}$



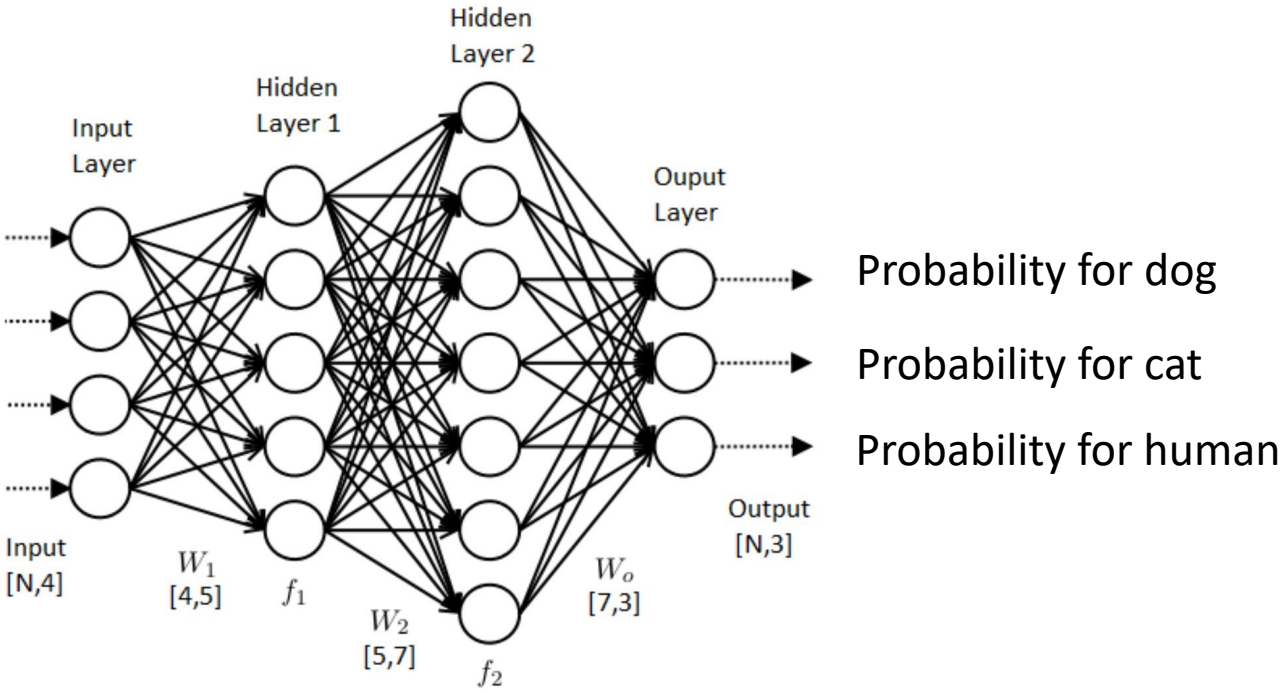
Program (Model)

- Define a function f_W to predict a corresponding output $\hat{y}_i$ for a given input x_i.
- $\hat{y}_i = f(x_i; W)$ or $\hat{y} = f_W(x_i)$
- $\hat{y}_i = f(x_i; \theta)$ or $\hat{y} = f_\theta(x_i)$



Model in Neural Networks

- $x_i \in \mathbb{R}^4$
- **Learnable** parameters $W = \{W_1, b_1, W_2, b_2, W_o, b_o\}$
- $y_i \in \mathbb{R}^3$



Performance Measure: Loss

- Find the optimized parameters W to map x into y.
- Loss $L(f(x, W), y) = L(\hat{y}, y)$
- Assumption: $L(\hat{y}, y) = \sum_i L(\hat{y}_i, y_i)$

Mean Squared Error (MSE)

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

* n is the number of data points
* Y_i represents observed values
* $\hat{Y}_i$ represents predicted values

Cross Entropy

$$\begin{bmatrix} 0.1 \\ 0.5 \\ 0.4 \end{bmatrix} \quad L(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_j y_j \ln \hat{y}_j \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$\hat{\mathbf{y}}$ $\mathbf{y}$

To fully understand why we use these as loss function, study about maximum likelihood estimation (MLE)

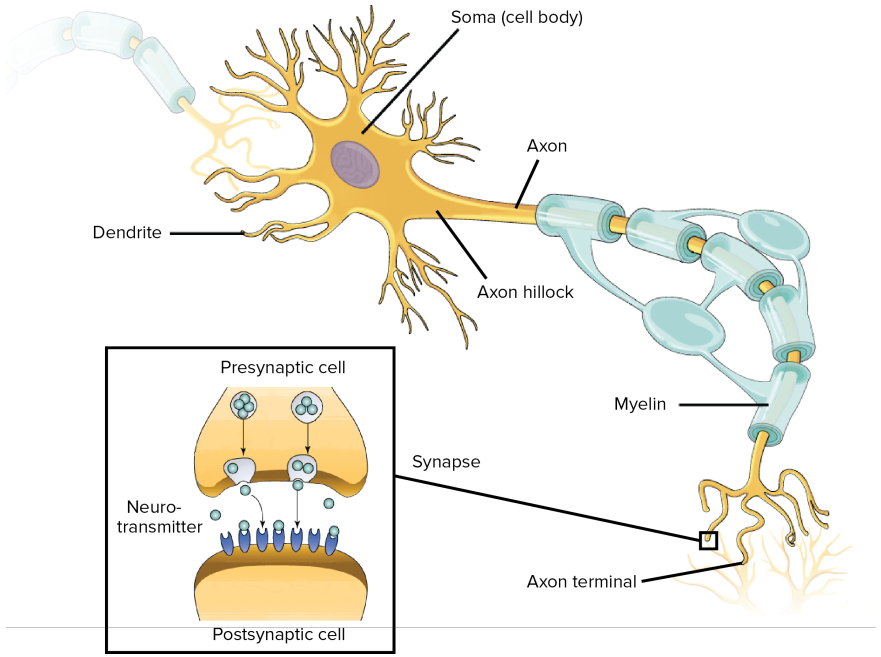
Model Training

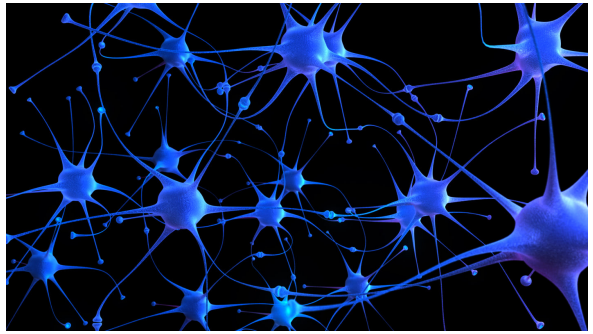
- $W^* = argmin_\theta L(f(x; W), y)$
  - $y \sim \hat{y}$
- Update $W \rightarrow W + \Delta W$ only if $L(\theta + \Delta W) < L(W)$
- Finish it when $L(W + \Delta W) == L(W)$
- How can we find ΔW so that $L(W + \Delta W) < L(W)$?
  - Gradient Descent

Neural Networks

Neuron

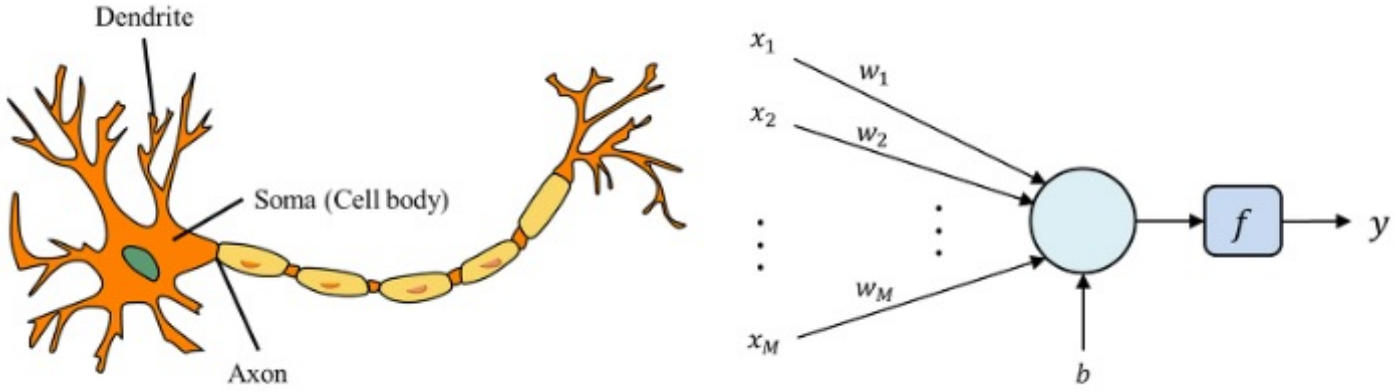
- Neurons are the fundamental units of the brain and nervous system.
- A dendrite receives input from other cells.
- A axon sends an electrical message to other neurons.





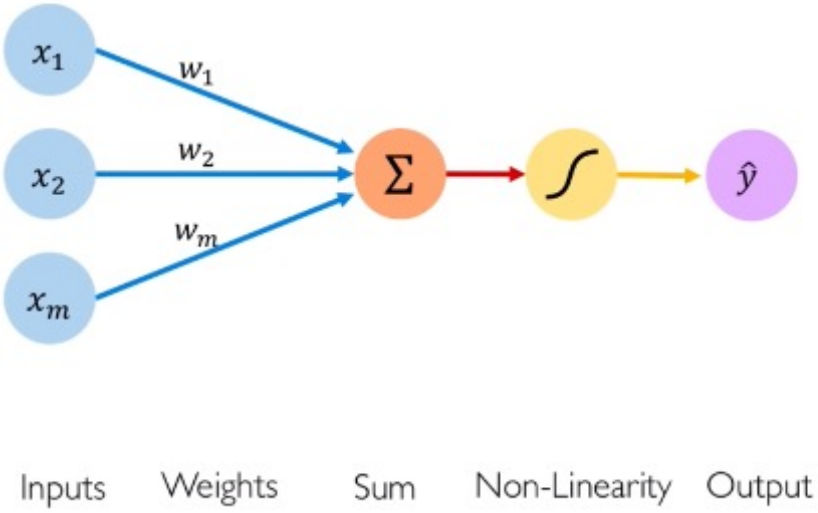
Artificial Neuron (Perceptron)

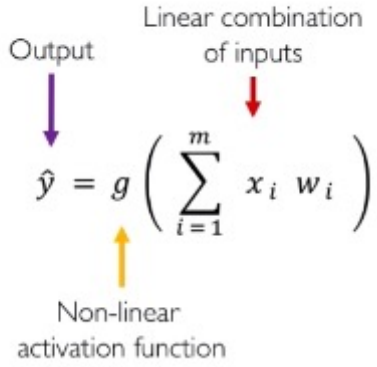
- A fundamental building block of a neural network
- Perceptron is a simplified model of a biological neuron.
- Dendrite → Weights, Axon → Activation function



The Perceptron: Forward Propagation

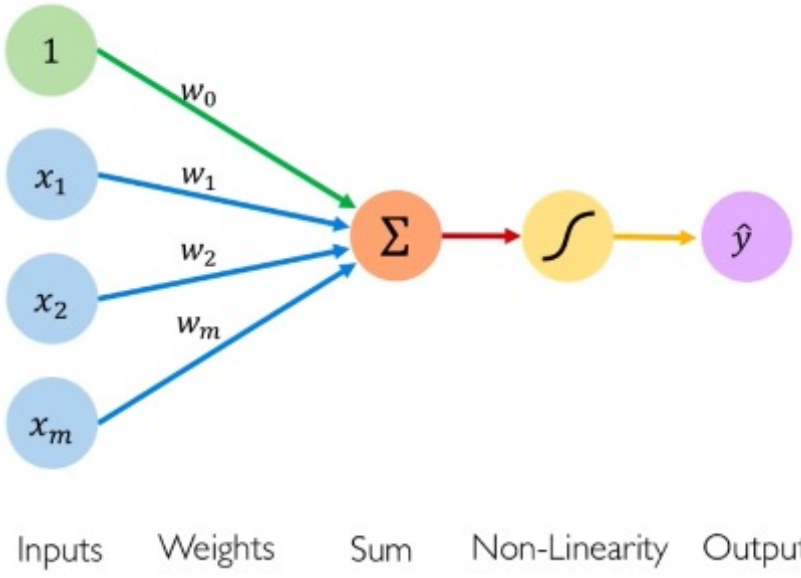
- Computation in perceptron: the forward propagation of information through a single neuron
- Input size m should be pre-defined and fixed before the computation.

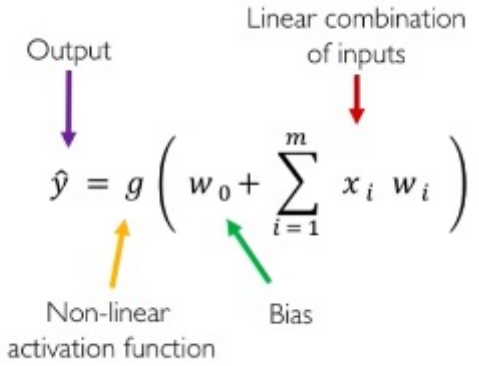




The Perceptron: Forward Propagation

- Dot product of the transpose of an input vector X and the weight vector W
- And then, add a bias term and apply non-linearity g



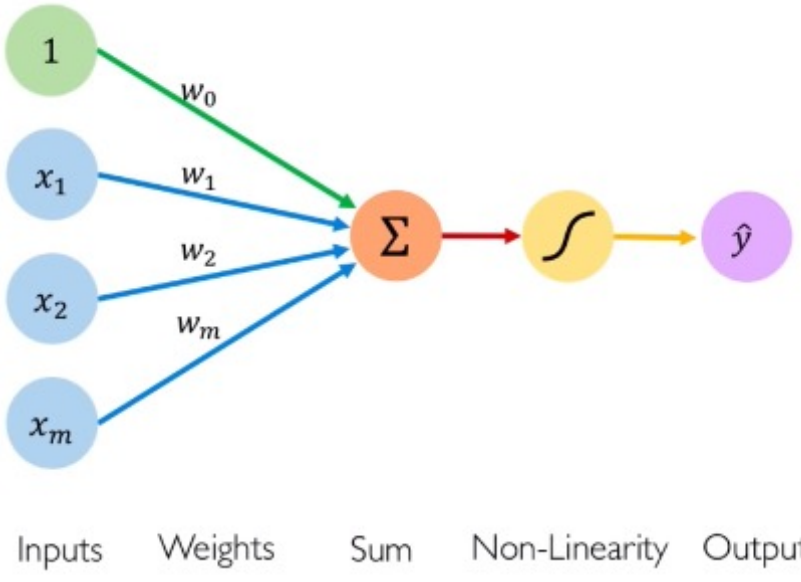


$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

Activation Function

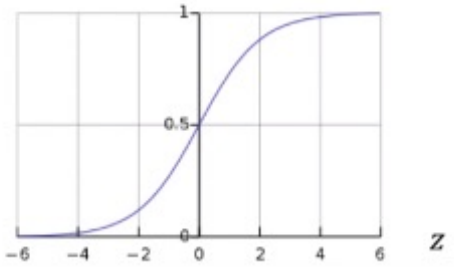
- Sigmoid takes any real number as input on the x-axis and it transforms that real number into a scalar output between 0 and 1.
- Commoly used when you're dealing with probabilities bounded between 0 and 1



$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$

- Example: sigmoid function

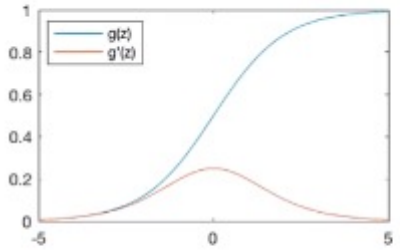
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Activation Functions

Is it Time to Swish? Comparing Deep Learning Activation Functions Across NLP tasks, EMNLP'19

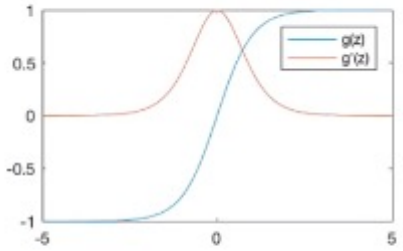
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

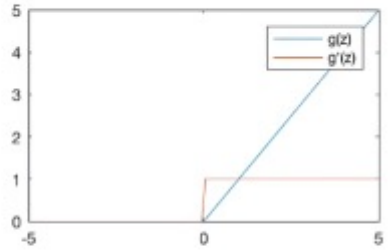
Hyperbolic Tangent



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)

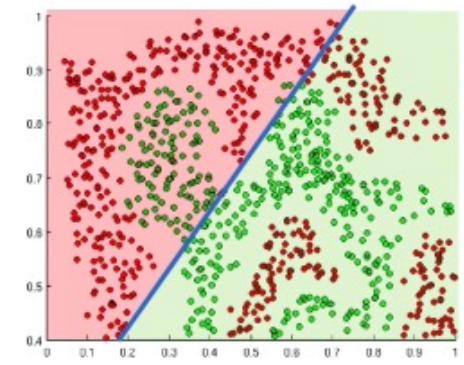


$$g(z) = \max(0, z)$$

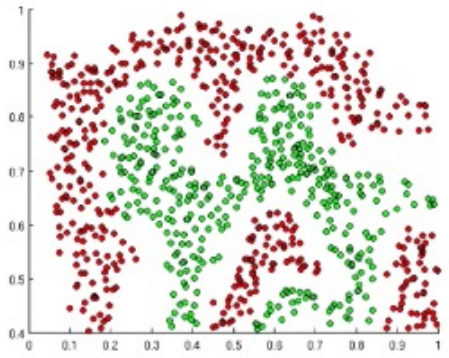
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Importance of Activation Functions

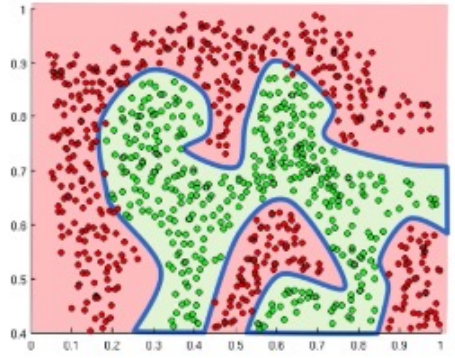
The purpose of activation function is to introduce non-linearities into the network.



Linear activation functions produce linear decisions no matter the network size



What if we wanted to build a neural network to distinguish green vs red points?

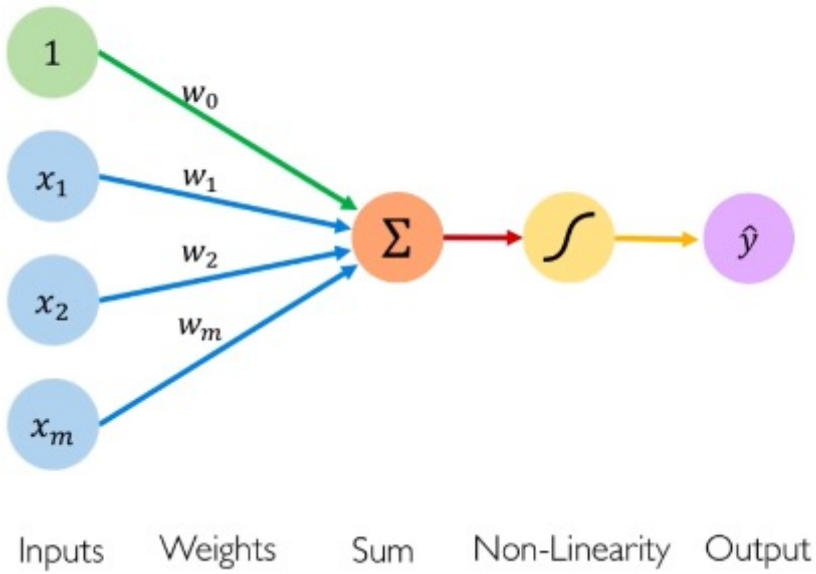


Non-linearities allow us to approximate arbitrarily complex functions

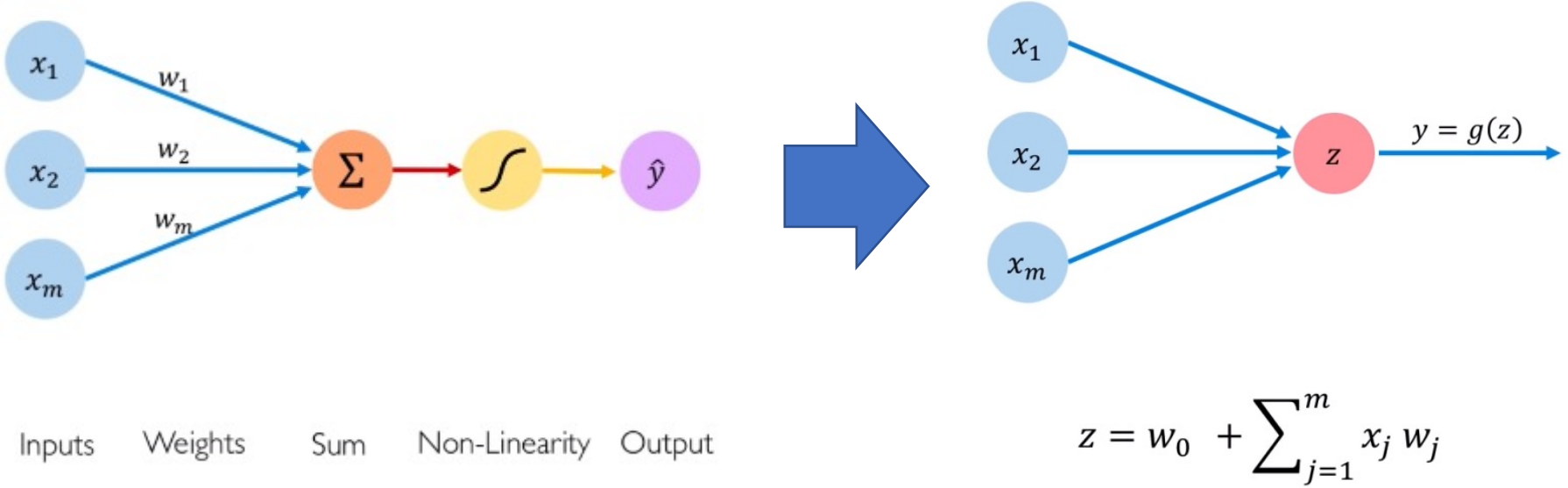
The Perceptron: Simplified

Model: weight parameters (to learn) and hyper-parameters (pre-defined by you)

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W} \right)$$



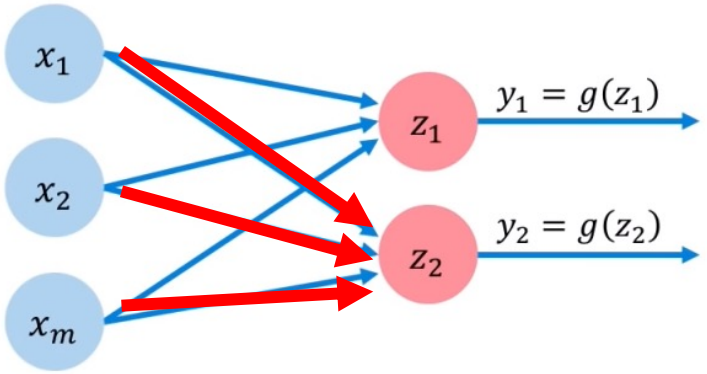
The Perceptron: Simplified



$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$

Multi Output Perceptron

- Because all inputs are densely connected to all outputs, these layers are called Dense layers.

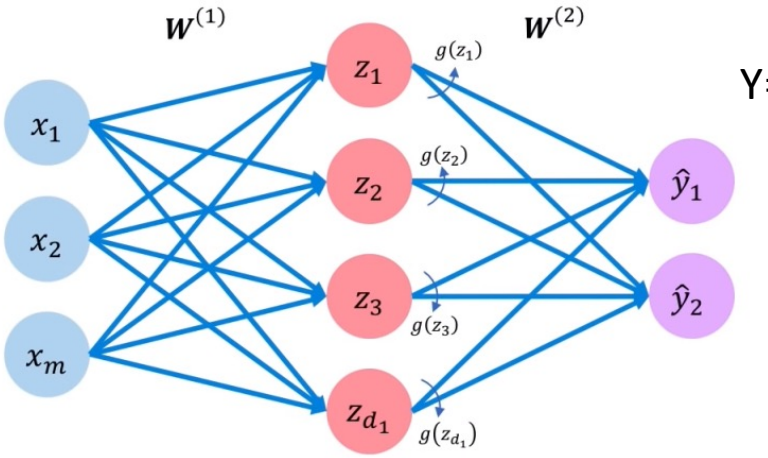


$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

Multi Output Perceptron

$Z = X^T W^{(1)})$

$W^{(1)} \in \mathbb{R}^{m \times d1}$



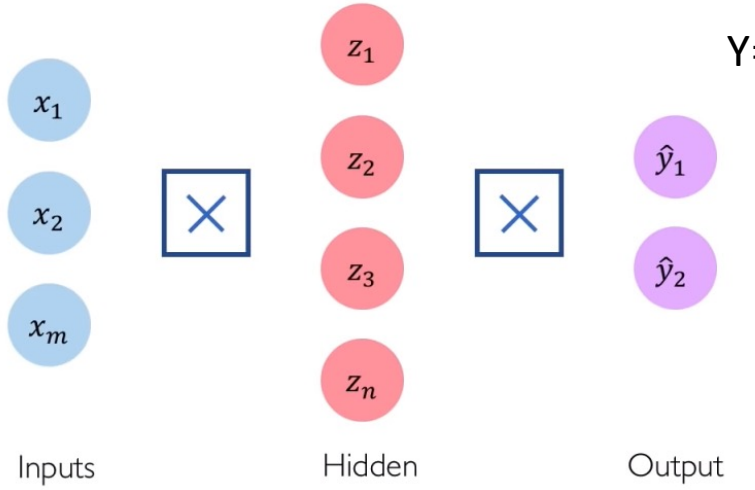
Y= $softmax(g(Z)^T W^{(2)})$

$W^{(1)} \in \mathbb{R}^{d1 \times 2}$

Multi Output Perceptron

$$Z = X^T W^{(1)})$$

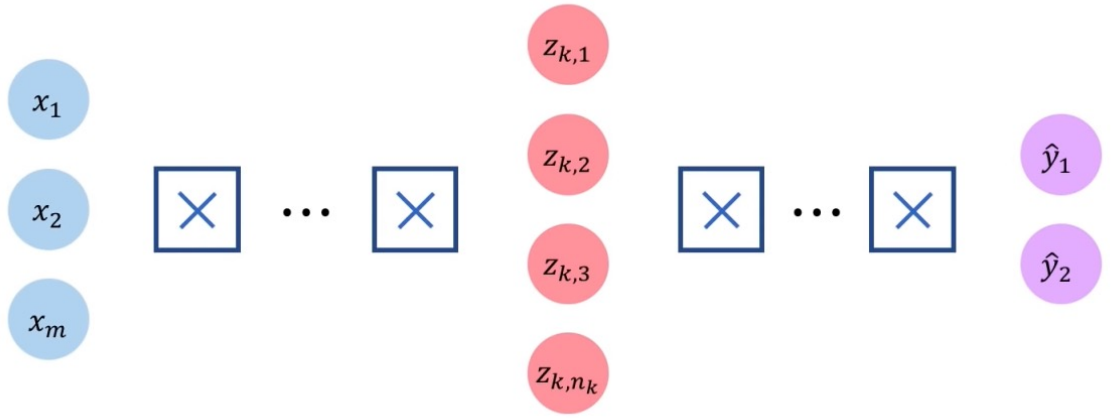
$$W^{(1)} \in \mathbb{R}^{m \times d1}$$



$$\text{Y} = softmax(g(Z)^T W^{(2)})$$

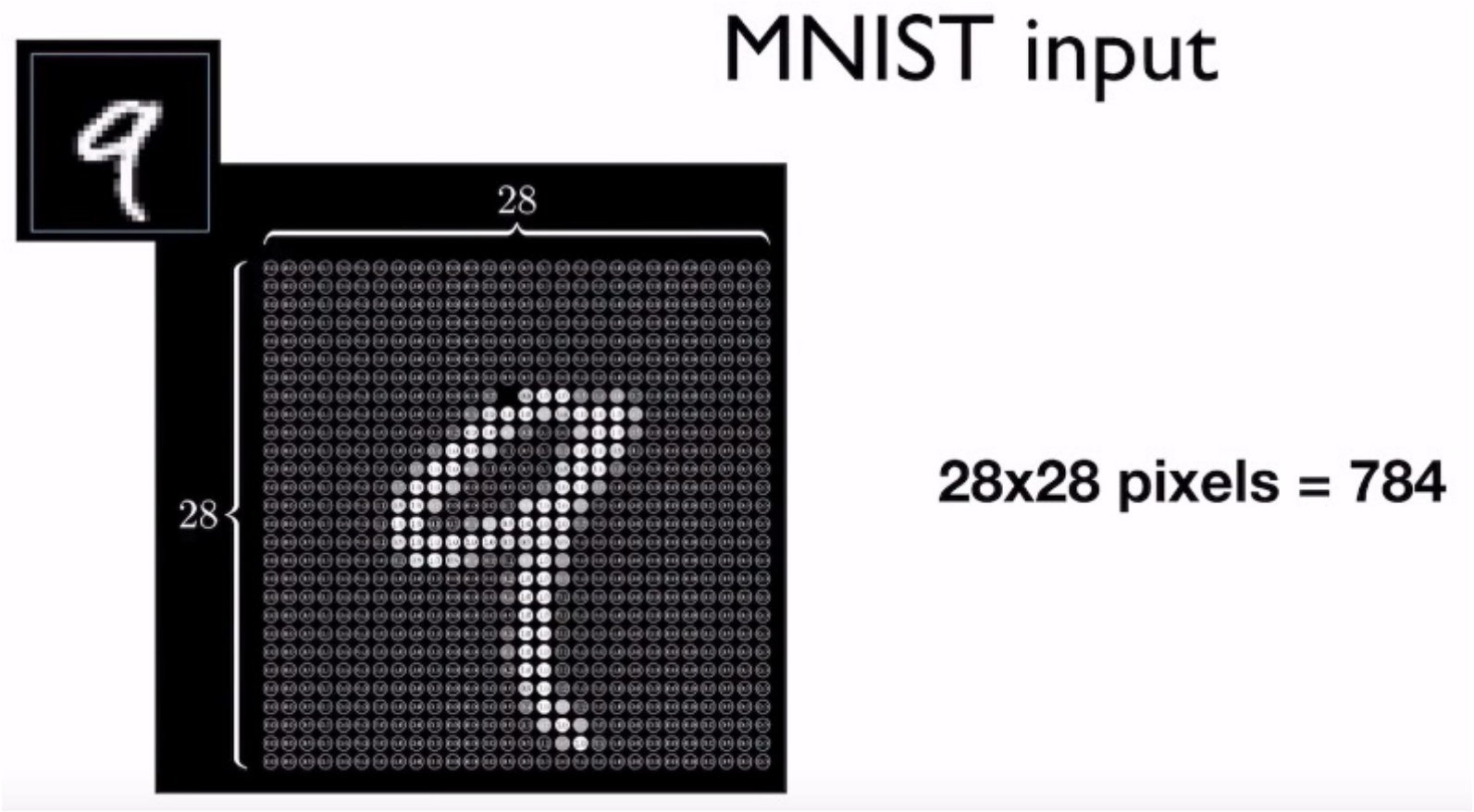
$$W^{(1)} \in \mathbb{R}^{d1 \times 2}$$

Multi Output Perceptron



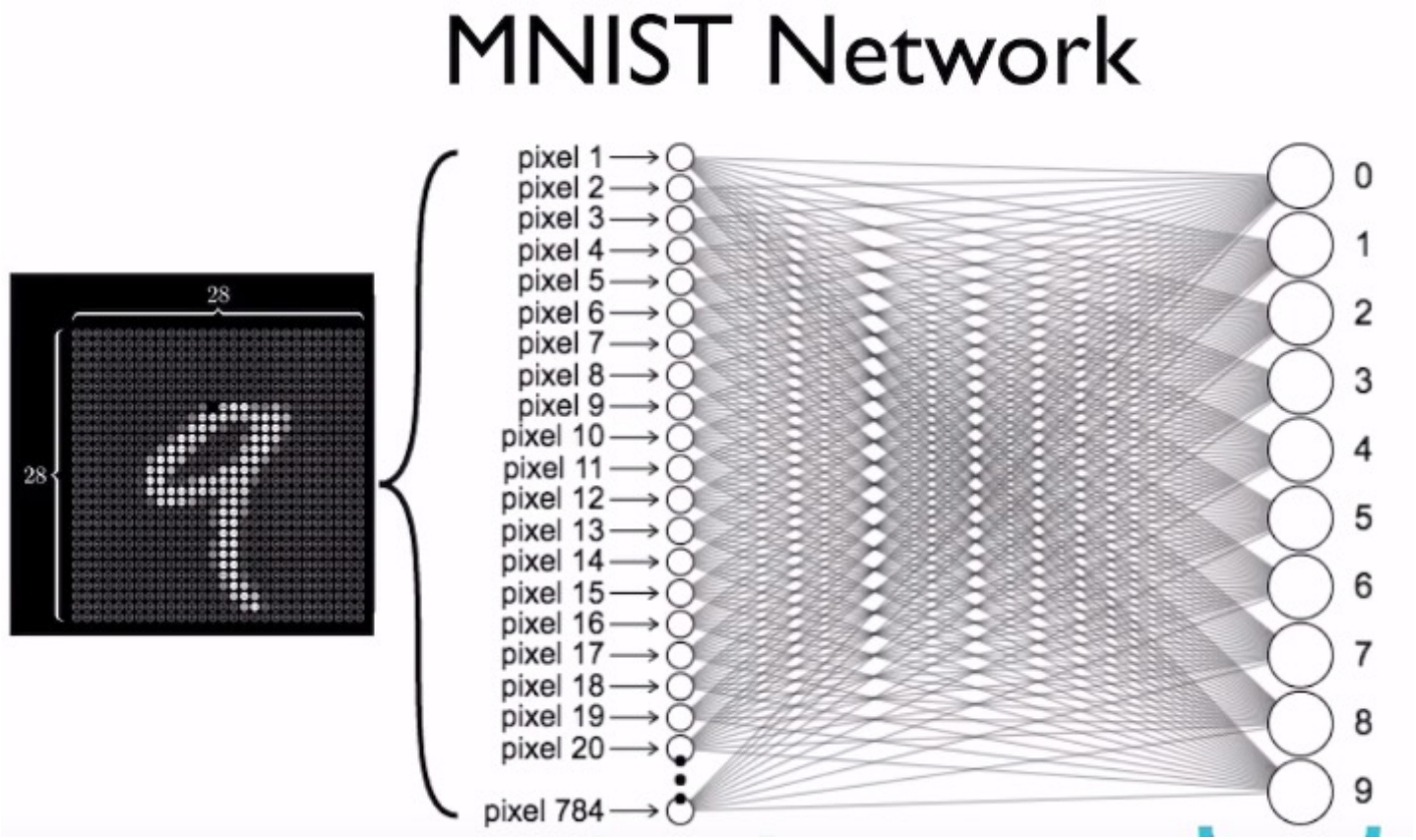
$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

Example with Pytorch



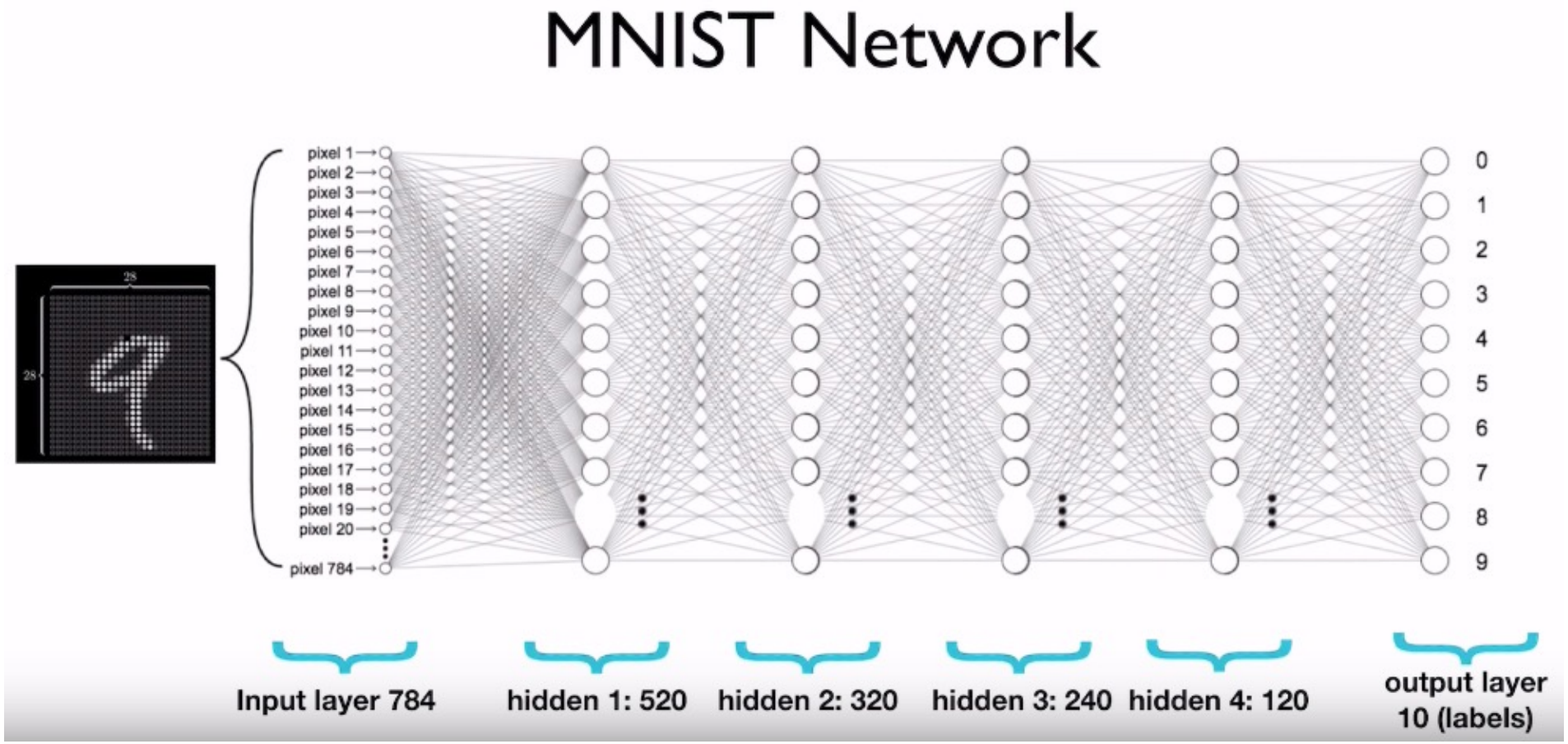
https://ml4a.github.io/ml4a/looking_inside_neural_nets/
https://www.youtube.com/watch?v=lvNdl7yg4Pg&list=PLlMkM4tgfjnJ3I-dbhO9JTw7gNty6o_2m&index=9

Example with Pytorch



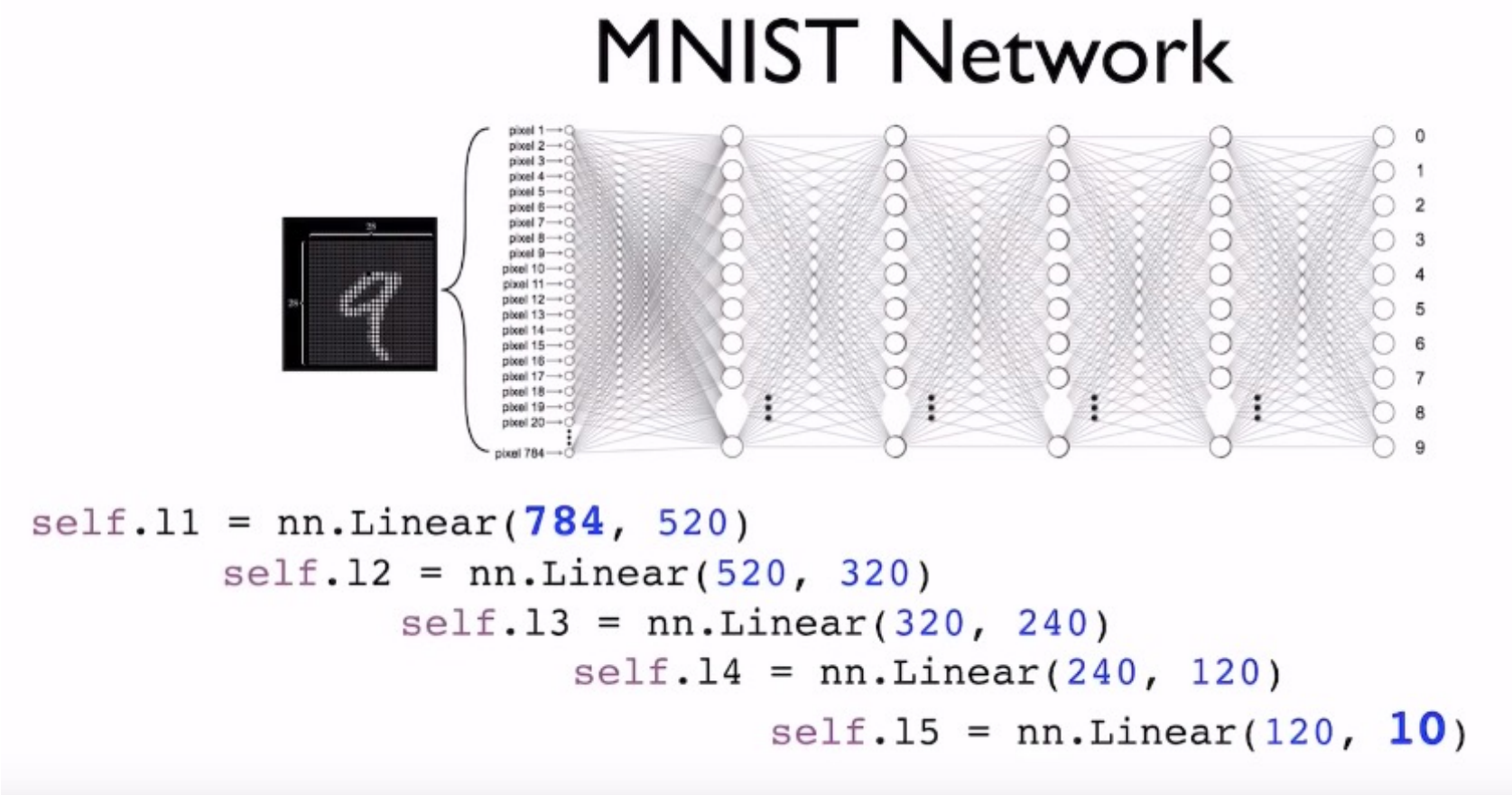
https://ml4a.github.io/ml4a/looking_inside_neural_nets/
https://www.youtube.com/watch?v=lvNdl7yg4Pg&list=PLlMkM4tgfjnJ3I-dbhO9JTw7gNty6o_2m&index=9

Example with Pytorch



https://ml4a.github.io/ml4a/looking_inside_neural_nets/
https://www.youtube.com/watch?v=lvNdl7yg4Pg&list=PLlMkM4tgfjnJ3I-dbhO9JTw7gNty6o_2m&index=9

Example with Pytorch



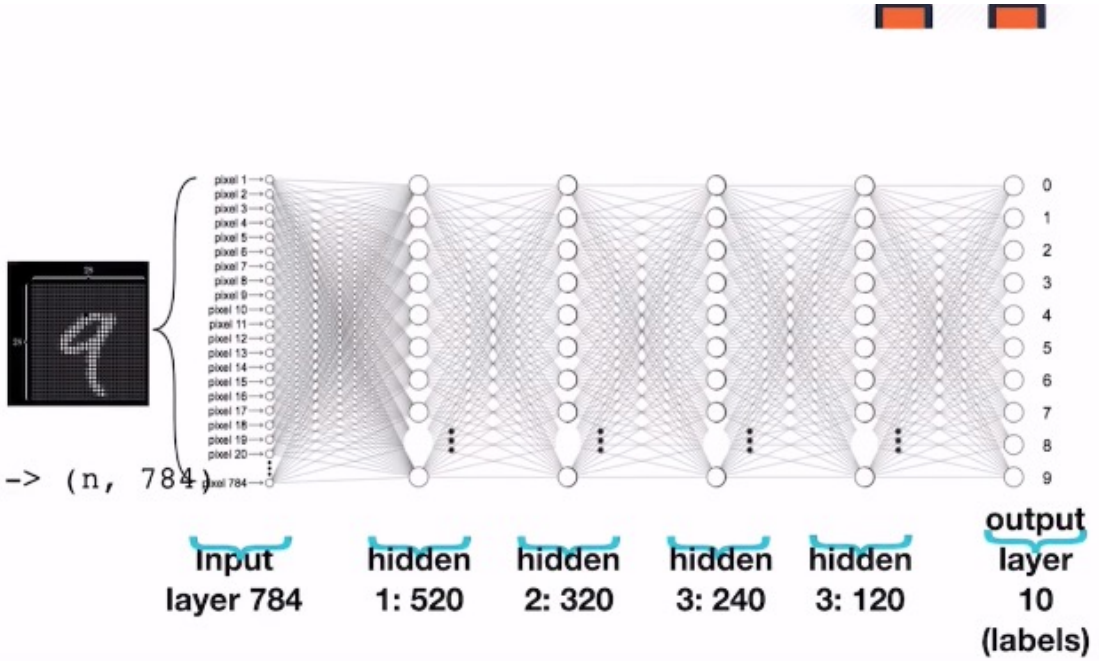
https://ml4a.github.io/ml4a/looking_inside_neural_nets/
https://www.youtube.com/watch?v=lvNdl7yg4Pg&list=PLlMkM4tgfjnJ3I-dbhO9JTw7gNty6o_2m&index=9

Example with Pytorch

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.l1 = nn.Linear(784, 520)
        self.l2 = nn.Linear(520, 320)
        self.l3 = nn.Linear(320, 240)
        self.l4 = nn.Linear(240, 120)
        self.l5 = nn.Linear(120, 10)

    def forward(self, x):
        # Flatten the data (n, 1, 28, 28)-> (n, 784)
        x = x.view(-1, 784)
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = F.relu(self.l3(x))
        x = F.relu(self.l4(x))
        return self.l5(x) # No need activation
```



https://ml4a.github.io/ml4a/looking_inside_neural_nets/
https://www.youtube.com/watch?v=lvNdl7yg4Pg&list=PLlMkM4tgfjnJ3I-dbhO9JTw7gNty6o_2m&index=9

Example with Pytorch

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.l1 = nn.Linear(784, 520)
        self.l2 = nn.Linear(520, 320)
        self.l3 = nn.Linear(320, 240)
        self.l4 = nn.Linear(240, 120)
        self.l5 = nn.Linear(120, 10)

    def forward(self, x):
        # Flatten the data (n, 1, 28, 28)-> (n, 784)
        x = x.view(-1, 784)
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = F.relu(self.l3(x))
        x = F.relu(self.l4(x))
        return self.l5(x) # No need activation
```

```
criterion = nn.CrossEntropyLoss()
…
for batch_idx, (data, target) in enumerate(train_loader):
    data, target = Variable(data), Variable(target)
    optimizer.zero_grad()
    output = model(data)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()
```

Softmax + cross_entropy

https://ml4a.github.io/ml4a/looking_inside_neural_nets/
https://www.youtube.com/watch?v=lvNdl7yg4Pg&list=PLlMkM4tgfjnJ3I-dbhO9JTw7gNty6o_2m&index=9

Example Problem

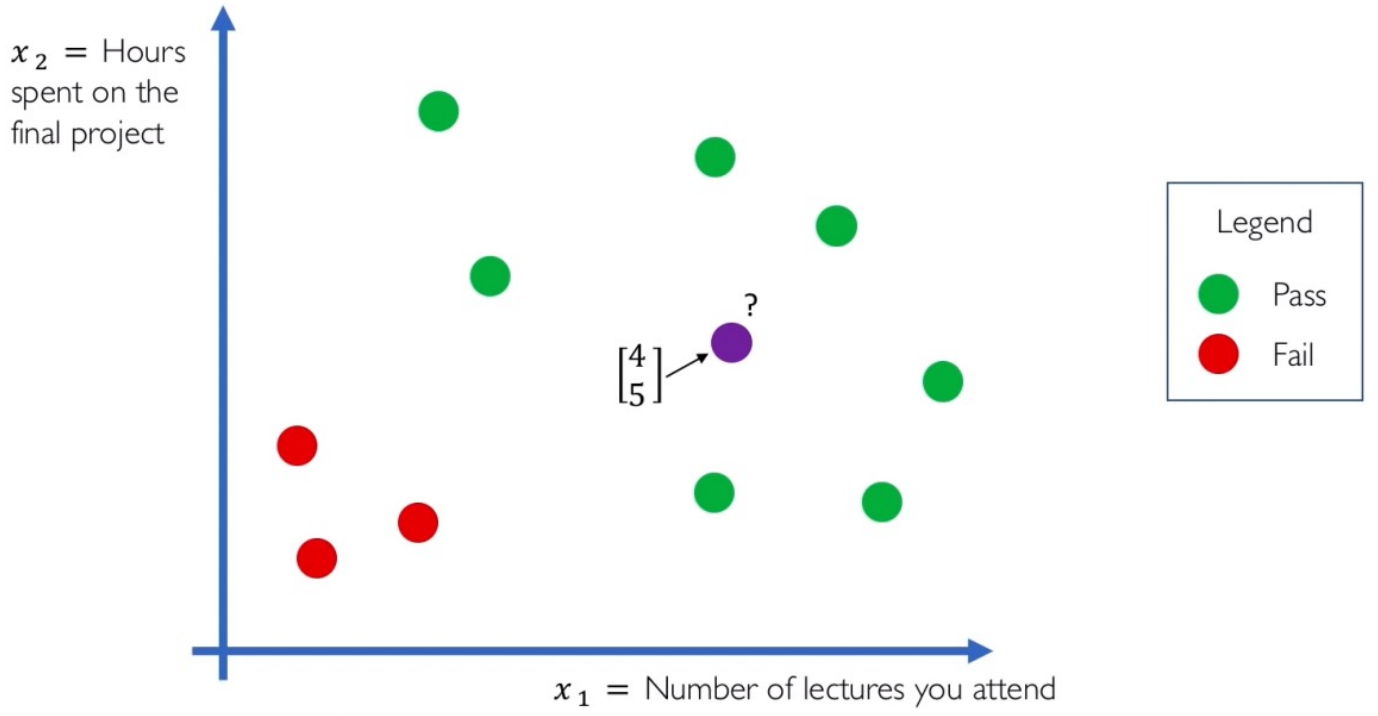
Will I pass this class?

Let's start with a simple two feature model

x_1 = Number of lectures you attend

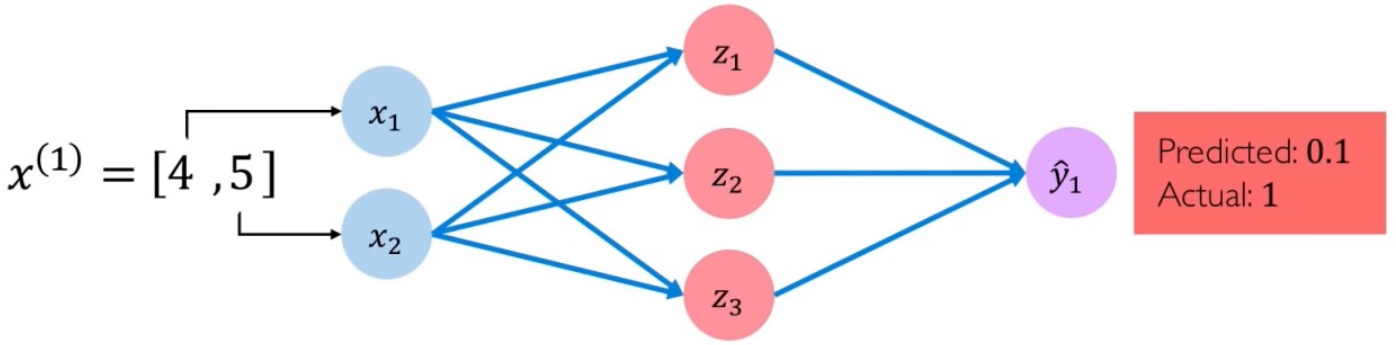
x_2 = Hours spent on the final project

Example Problem: Will I pass this class?



Quantifying Loss

- The loss of our network measures the cost incurred from inccorect predictions.



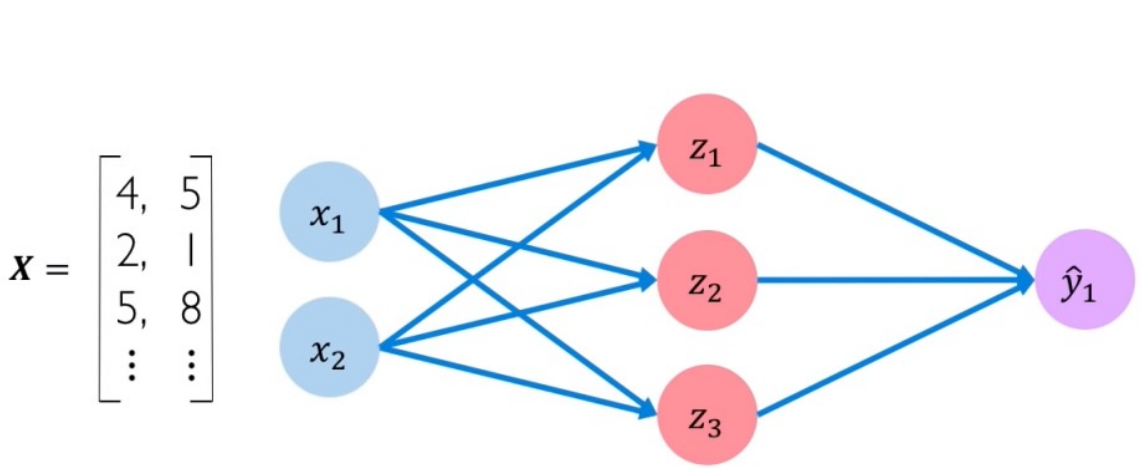
$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

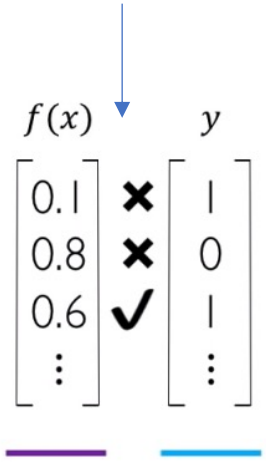
Empirical Loss

- The empirical loss measures the total loss over our entire datasest

$$\hat{y} = 1 \; if \; f(x) \geq 0.5$$
$$\hat{y} = 0 \; if \; f(x) < 0.5$$



$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



Also known as:
- Objective function
- Cost function
- Empirical Risk

$$\boldsymbol{J(W)} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

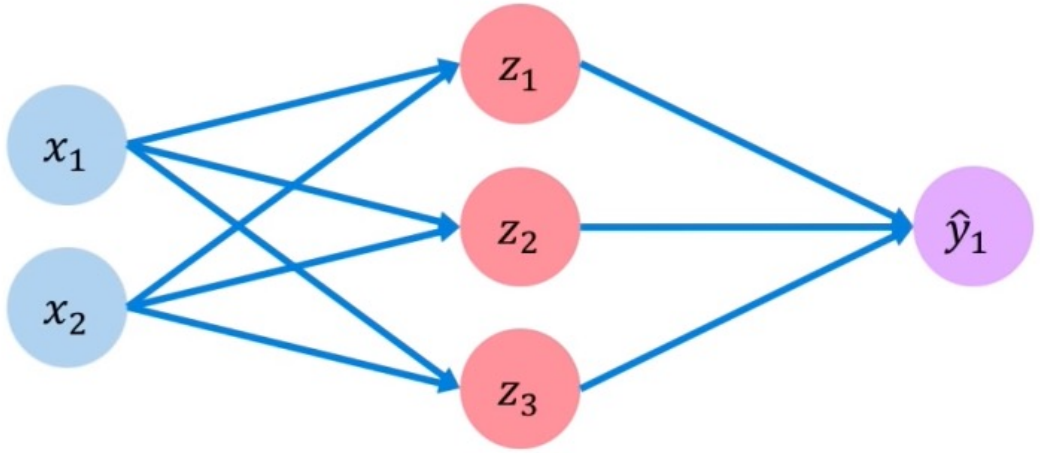
Predicted Actual

Binary Cross Entropy Loss

- Cross entropy loss can be used with models that output a probability between 0 and 1.

$$\hat{y} = 1 \; if \; f(x) \geq 0.5$$
$$\hat{y} = 0 \; if \; f(x) < 0.5$$

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$

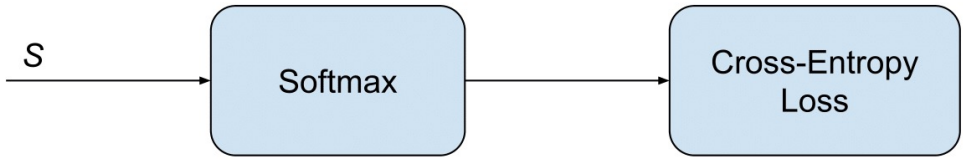


$f(x)$		y
0.1	✗	1
0.8	✗	0
0.6	✓	1
⋮		⋮

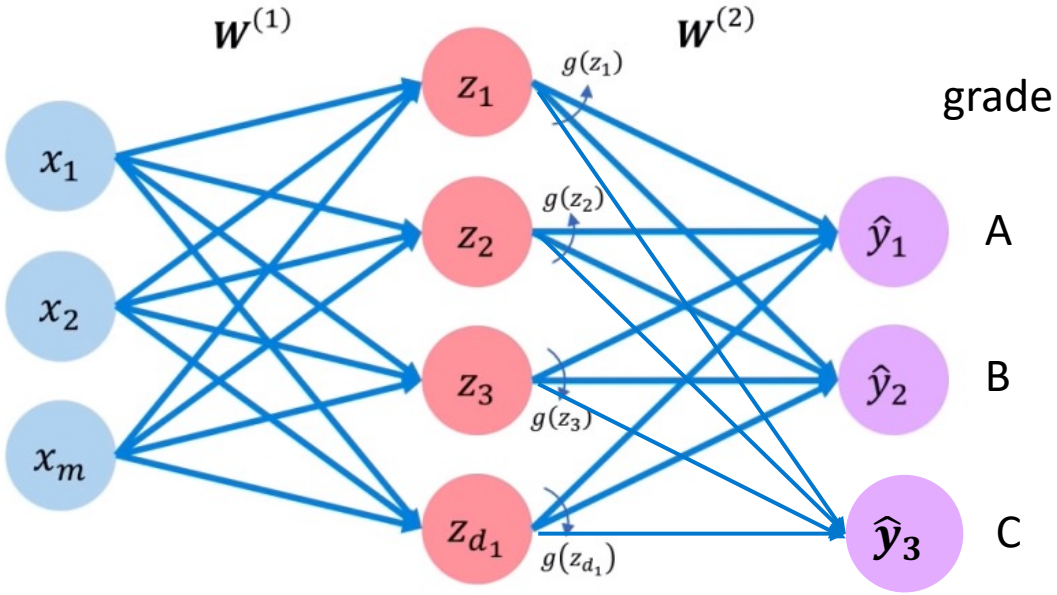
$$\boldsymbol{J(W)} = \frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log\left(f\left(x^{(i)}; \boldsymbol{W}\right)\right) + (1 - y^{(i)}) \log\left(1 - f\left(x^{(i)}; \boldsymbol{W}\right)\right)$$

Actual — Predicted — Actual — Predicted

(General) Cross Entropy Loss

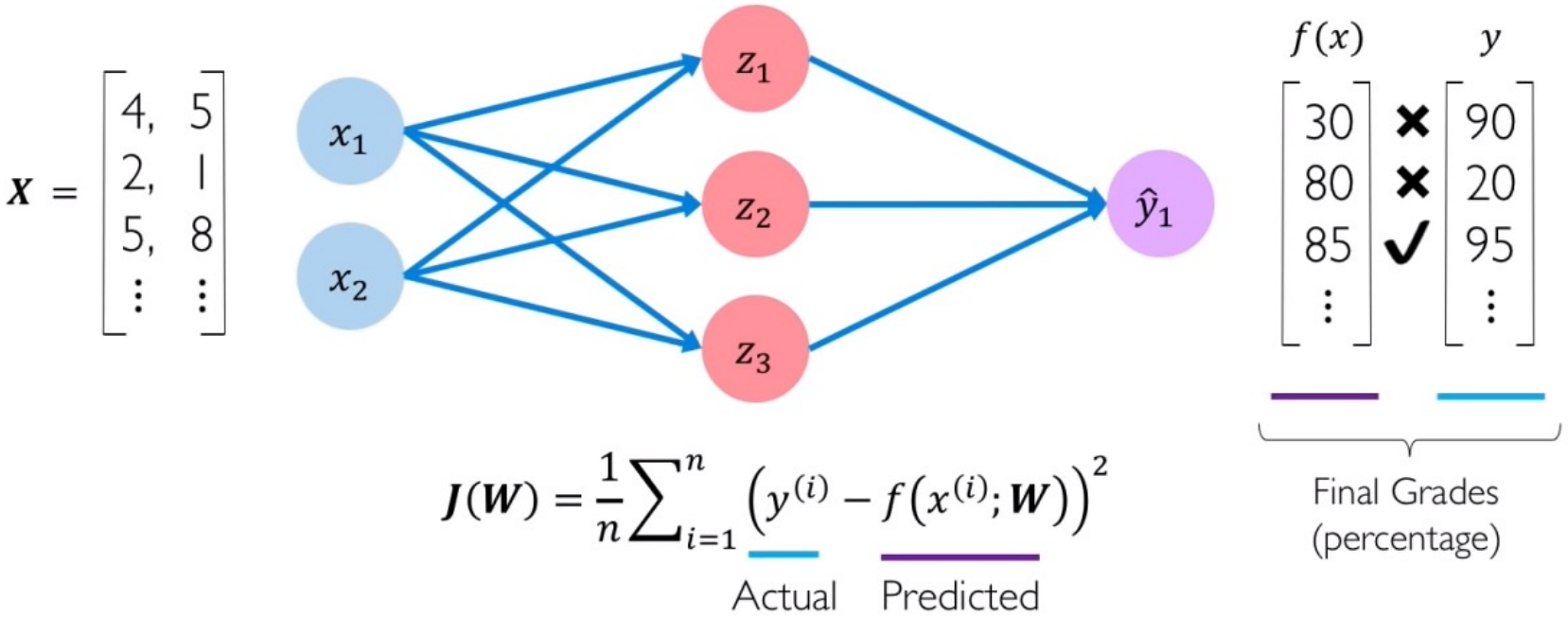


$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$



Mean Squared Error Loss

- Mean squared error loss can be used with regression models that output continuous real numbers



Loss Optimization

- We want to find the network weights that achieve the lowest loss.

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

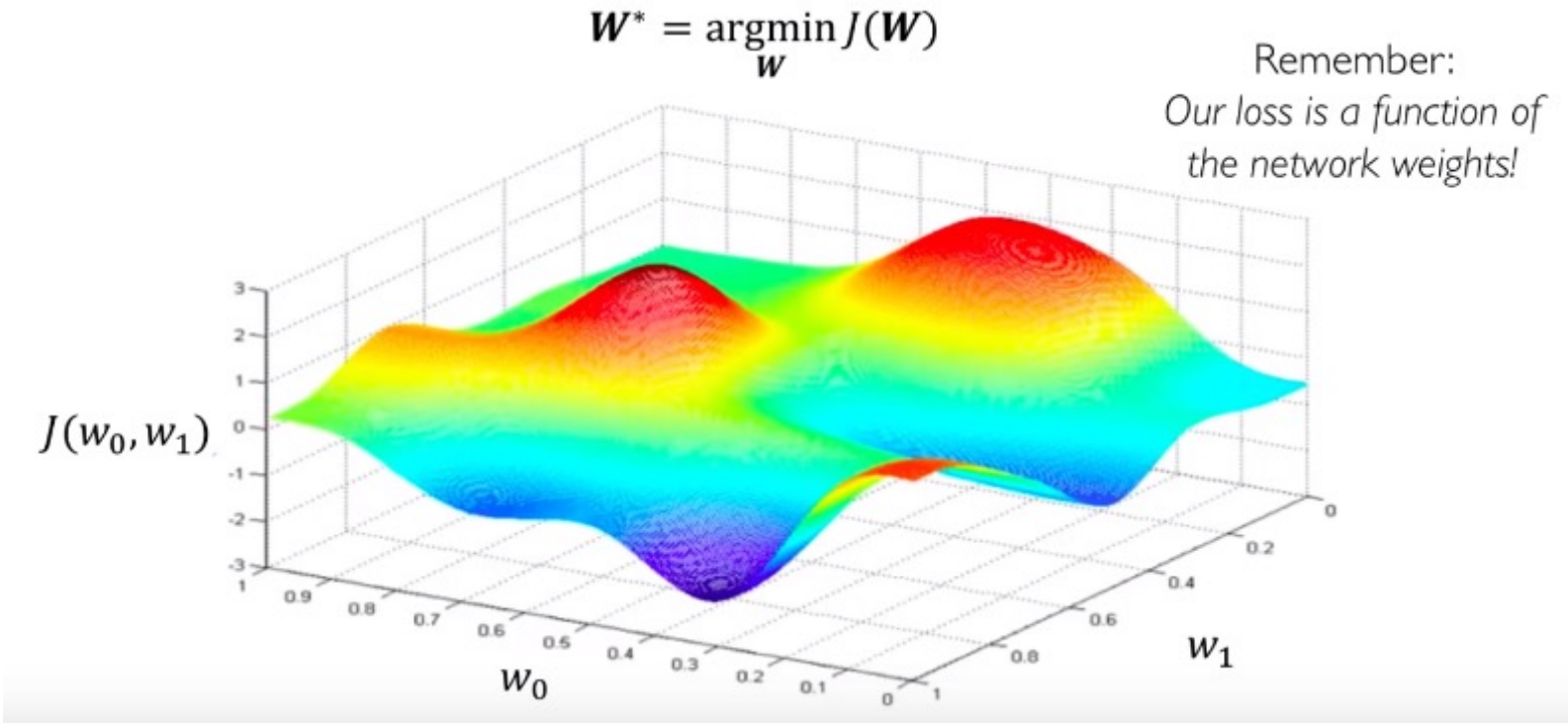
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

Remember:
$\boldsymbol{W} = \{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \dots\}$

Loss Optimization

$$W^* = \underset{W}{\text{argmin}}\, J(W)$$

Remember:
Our loss is a function of the network weights!

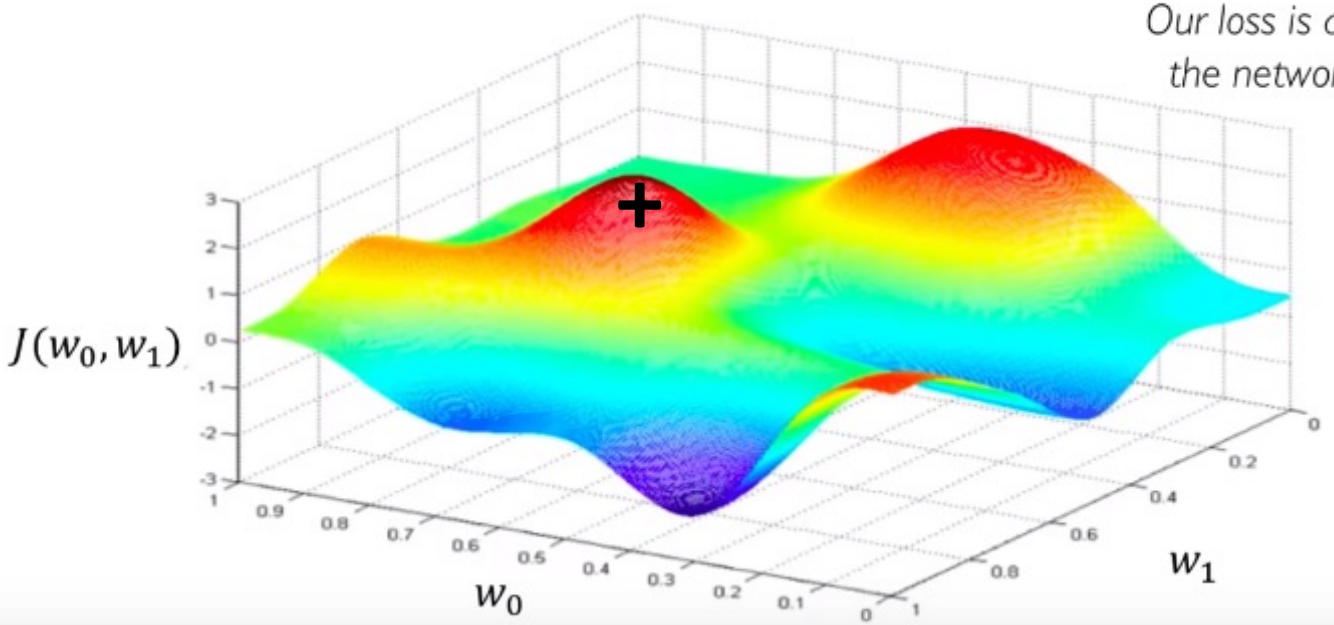


Loss Optimization

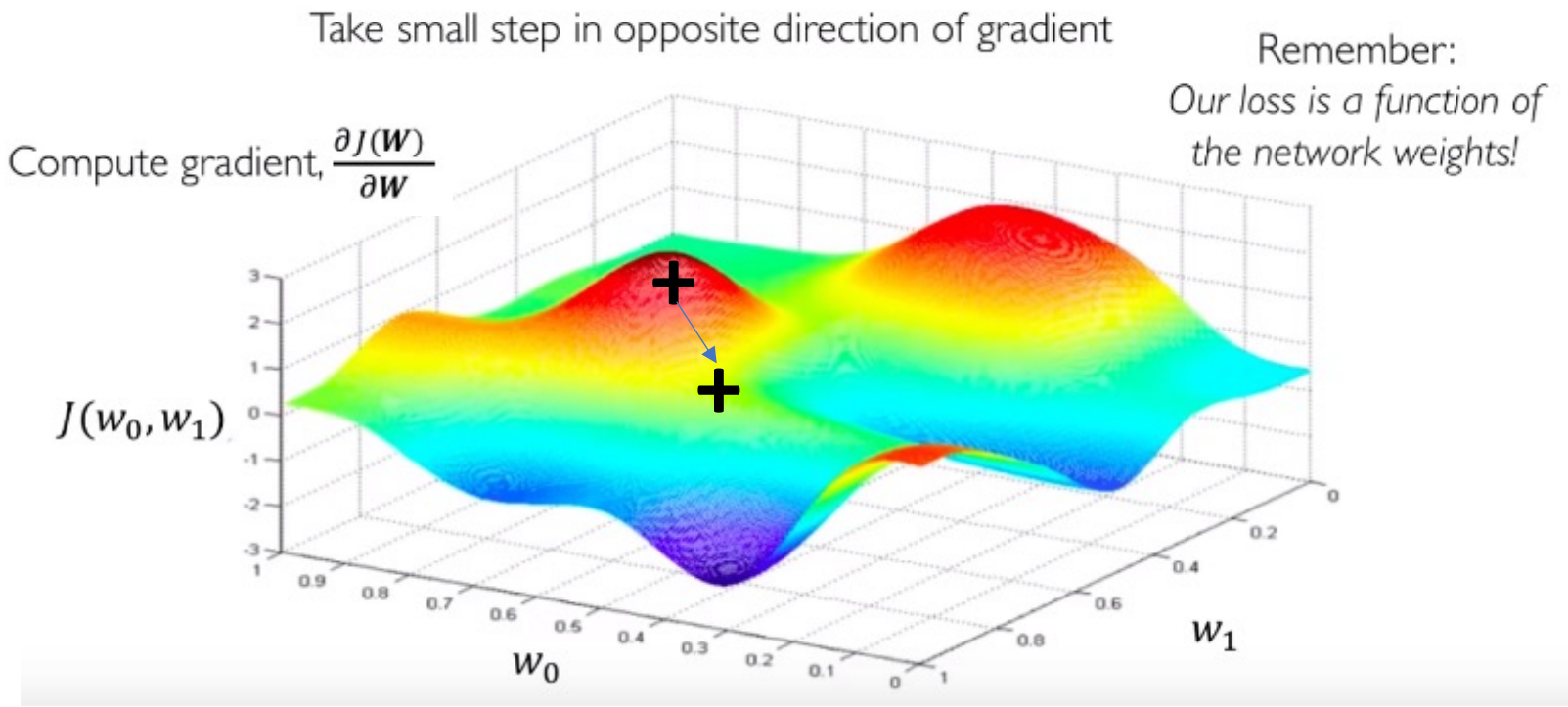
Randomly pick an initial (w_0, w_1)

$$W^* = \underset{W}{\text{argmin}}\, J(W)$$

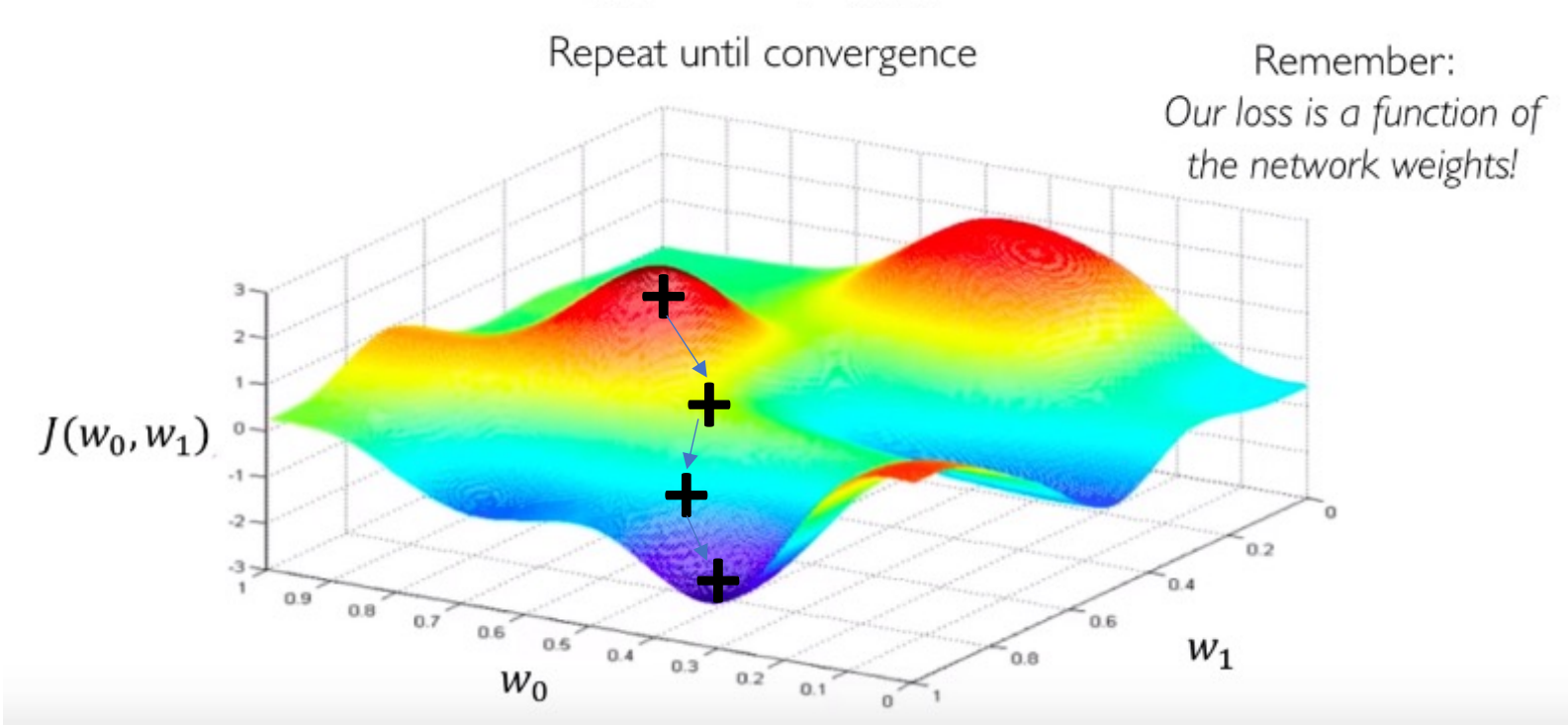
Remember:
Our loss is a function of the network weights!



Loss Optimization



Gradient Descent



Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

Computing Gradeints: Backpropagation

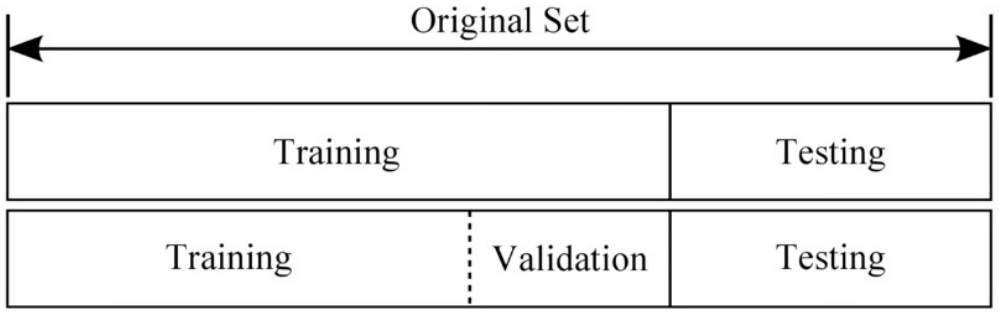
- See this video lecture by Prof. Sung Kim
  - https://www.youtube.com/watch?v=ma2KXWbllIc&t=647s

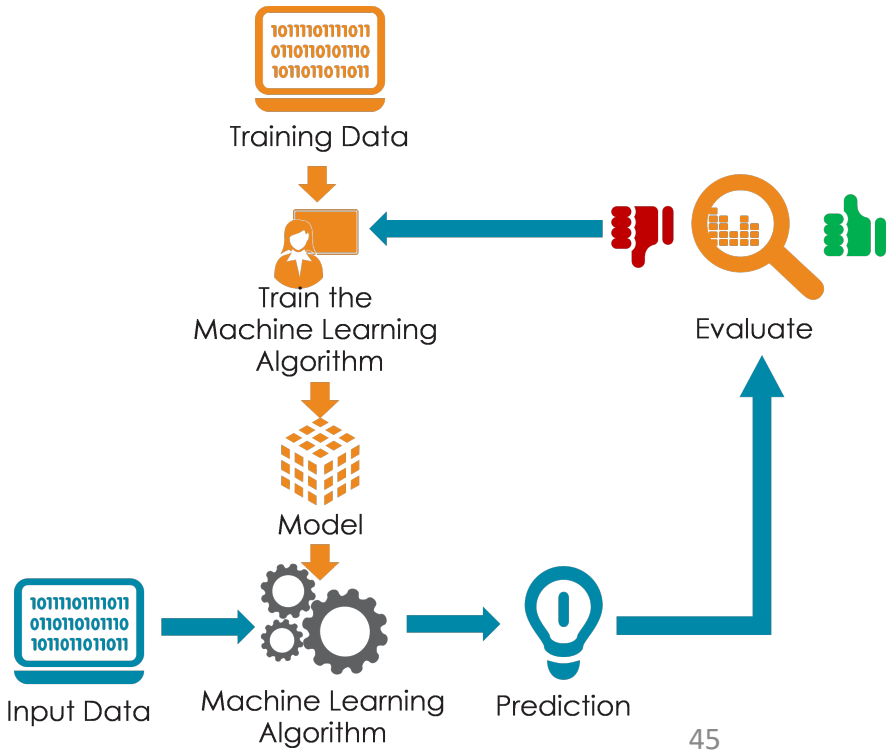
Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

- We will discuss it more later in the chapter of RNN.

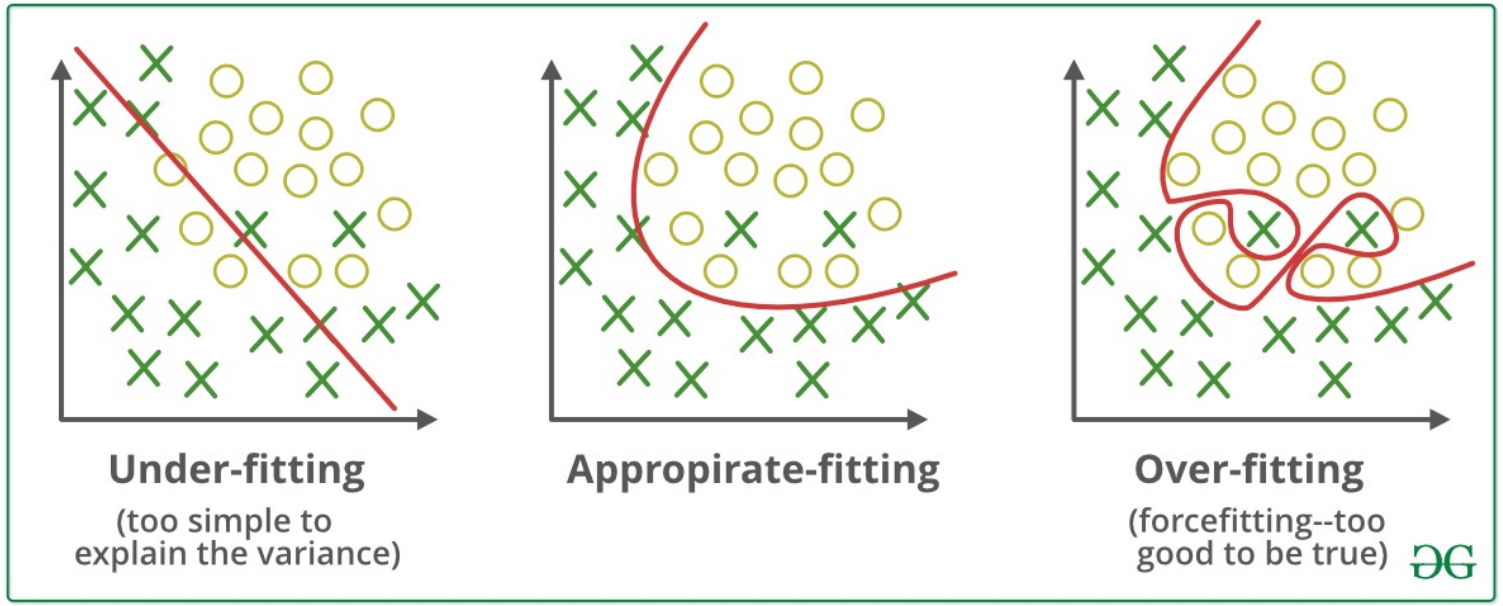
Training and Test Data





Overfitting

- https://en.wikipedia.org/wiki/Overfitting



Regularization: Adding L2-norm

- Reducing magnitude/values of weight paramters W
- Works well when we have a lot of features, each of which contributes a bit to predicting y.

L1 Regularization

$$\text{Cost} = \sum_{i=0}^{N} \boxed{\text{Ordinary loss}} + \lambda \sum_{j=0}^{M} |W_j|$$

L2 Regularization

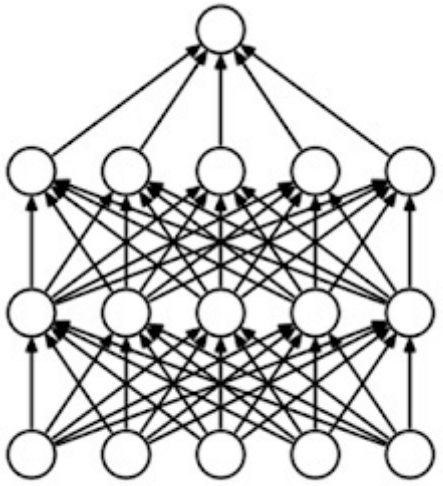
$$\text{Cost} = \sum_{i=0}^{N} \boxed{\text{Ordinary loss}} + \lambda \sum_{j=0}^{M} W_j^2$$

Loss function

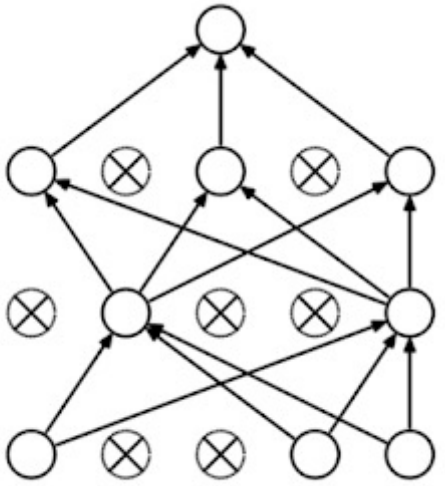
Regularization Term

Regularization: Dropout

- During training, outputs of a layer to zero randomly with probability p
  - Prevents units from co-adapting too much
  - Forces network to learn more robust features
- At test time, dropout is disabled.

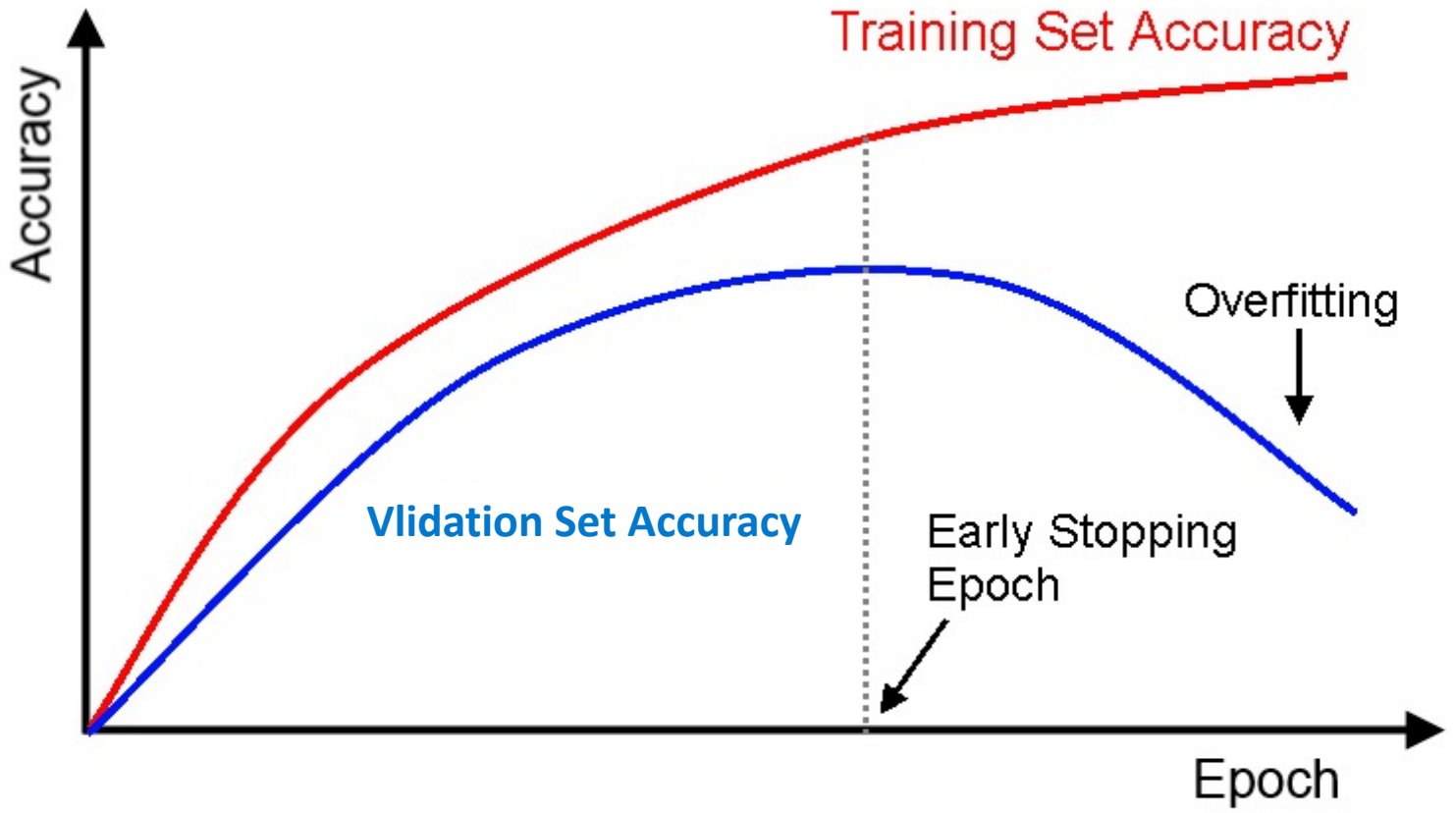


(a) Standard Neural Net

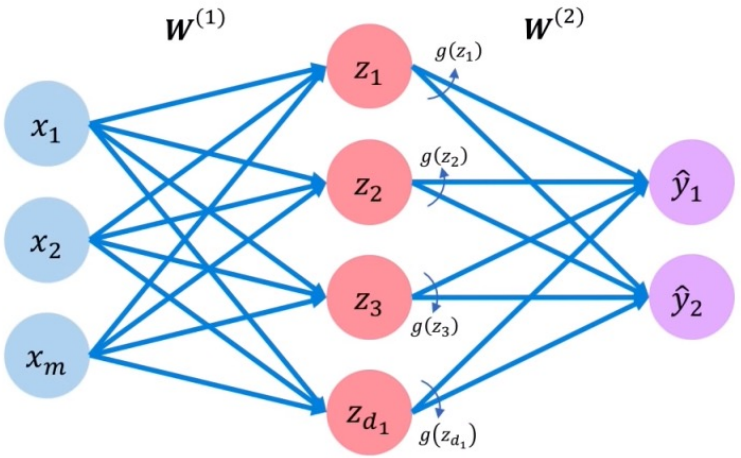


(b) After applying dropout.

Regularization: Early Stop



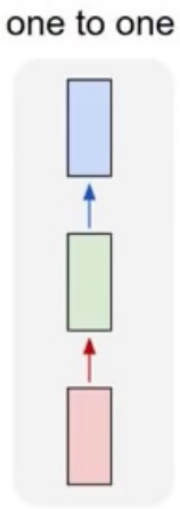
“Vanilla” Feed-forward Neural Network

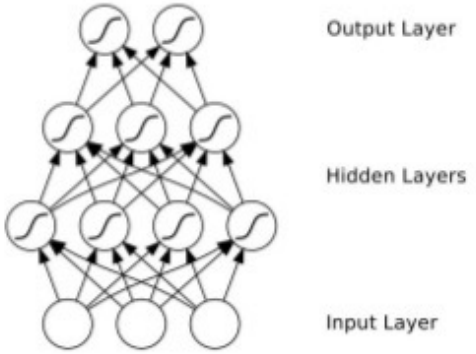


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)}\right)$$

“Vanilla” Feed-forward Neural Network

- Vanilla feed forward network receives input with a fixed size (e.g., image), and the input is fed through hidden layers and produces outputs.

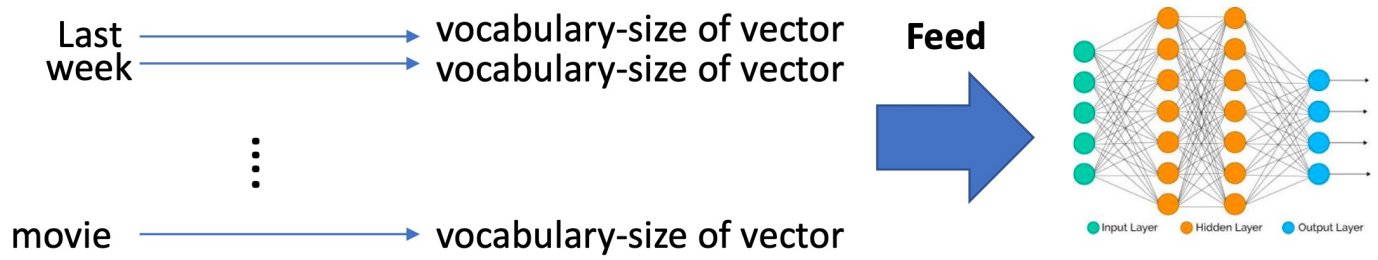




One-hot Vector/Encoding

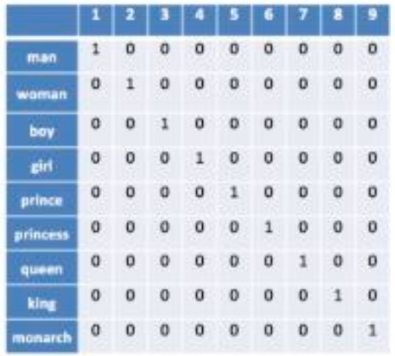
- $v \in \{0,1\}^{|V|}, where$ v is one-hot vector and |V| is vocaburary size

	1	2	3	4	5	6	7	8	9
man	1	0	0	0	0	0	0	0	0
woman	0	1	0	0	0	0	0	0	0
boy	0	0	1	0	0	0	0	0	0
girl	0	0	0	1	0	0	0	0	0
prince	0	0	0	0	1	0	0	0	0
princess	0	0	0	0	0	1	0	0	0
queen	0	0	0	0	0	0	1	0	0
king	0	0	0	0	0	0	0	1	0
monarch	0	0	0	0	0	0	0	0	1



Drawbacks of One-hot Representation

- Curse of Dimensionality



	1	2	3	4	5	6	7	8	9
man	1	0	0	0	0	0	0	0	0
woman	0	1	0	0	0	0	0	0	0
boy	0	0	1	0	0	0	0	0	0
girl	0	0	0	1	0	0	0	0	0
prince	0	0	0	0	1	0	0	0	0
princess	0	0	0	0	0	1	0	0	0
queen	0	0	0	0	0	0	1	0	0
king	0	0	0	0	0	0	0	1	0
monarch	0	0	0	0	0	0	0	0	1

Mostly zeros

Information quality ↓

Memory use ↑

Computation time ↑

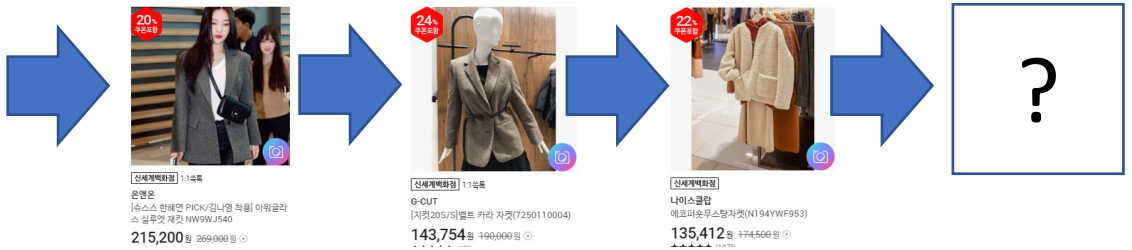
- Orthogonality

$$[0,1,0,\ldots,0] \text{ X } [0,1,0,\ldots,0]^T = 0$$

Distance(“Dog”, “Puppy”) = Distance(“Dog”, “Computer”) = 0

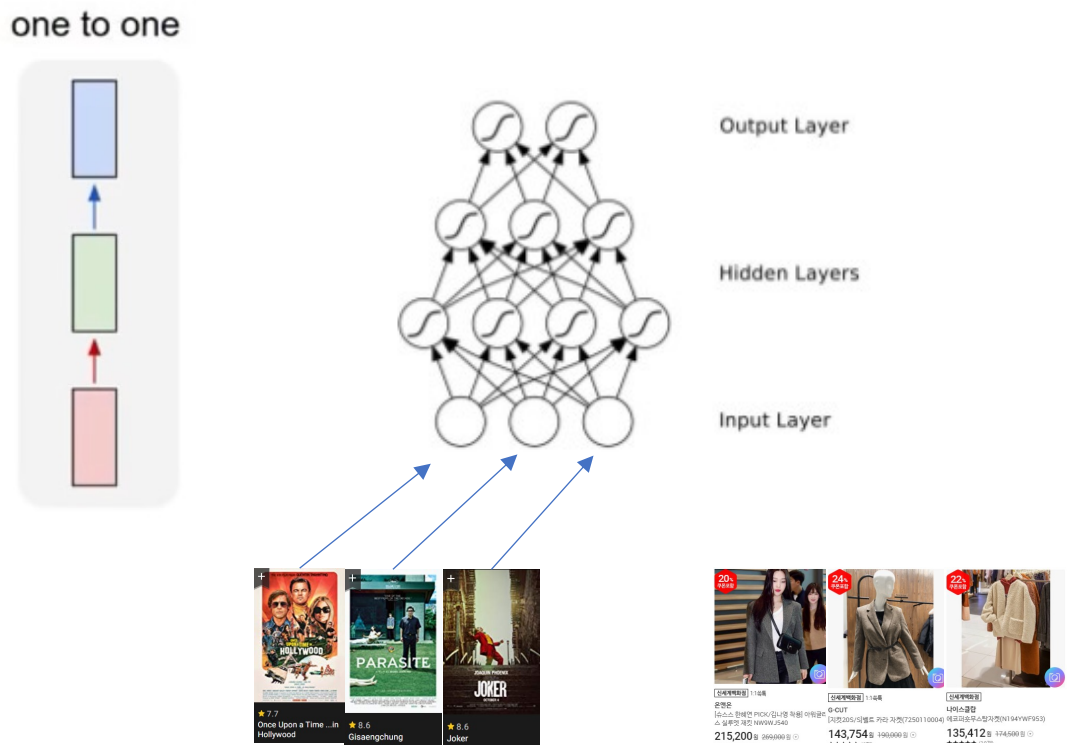
How can we handle sequence data?

- Many things are sequence.
  - When you watch movies (Next basket prediction)
  - When you enjoy online-shopping (Next basket prediction)
  - Speech recognition, Actions in video, …
  - When you write down something



Vanilla NN X Sequential Data

- What if the sequence length is not fixed?
- Sequential order is useless information?



Vanilla NN X Sequential Data

- What if the sequence length is not fixed?
  - Naïve solution: Average! (RecSys'16)

Deep Neural Networks for YouTube Recommendations

Paul Covington, Jay Adams, Emre Sargin
Google
Mountain View, CA
{pcovington, jka, msargin}@google.com

ABSTRACT

YouTube represents one of the largest scale and most sophisticated industrial recommendation systems in existence. In this paper, we describe the system at a high level and focus on the dramatic performance improvements brought by deep learning. The paper is split according to the classic two-stage information retrieval dichotomy: first, we detail a deep candidate generation model and then describe a separate deep ranking model. We also provid… actical lessons and insights derive… m designing, i… and maintain… massive rec… ion s… normous…

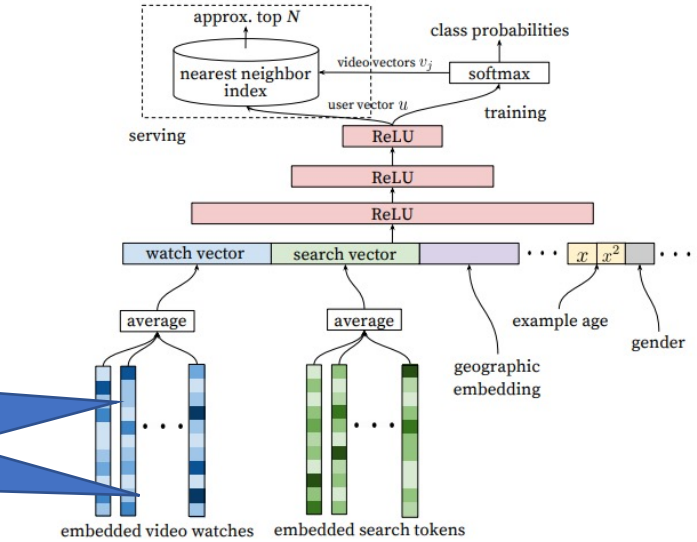
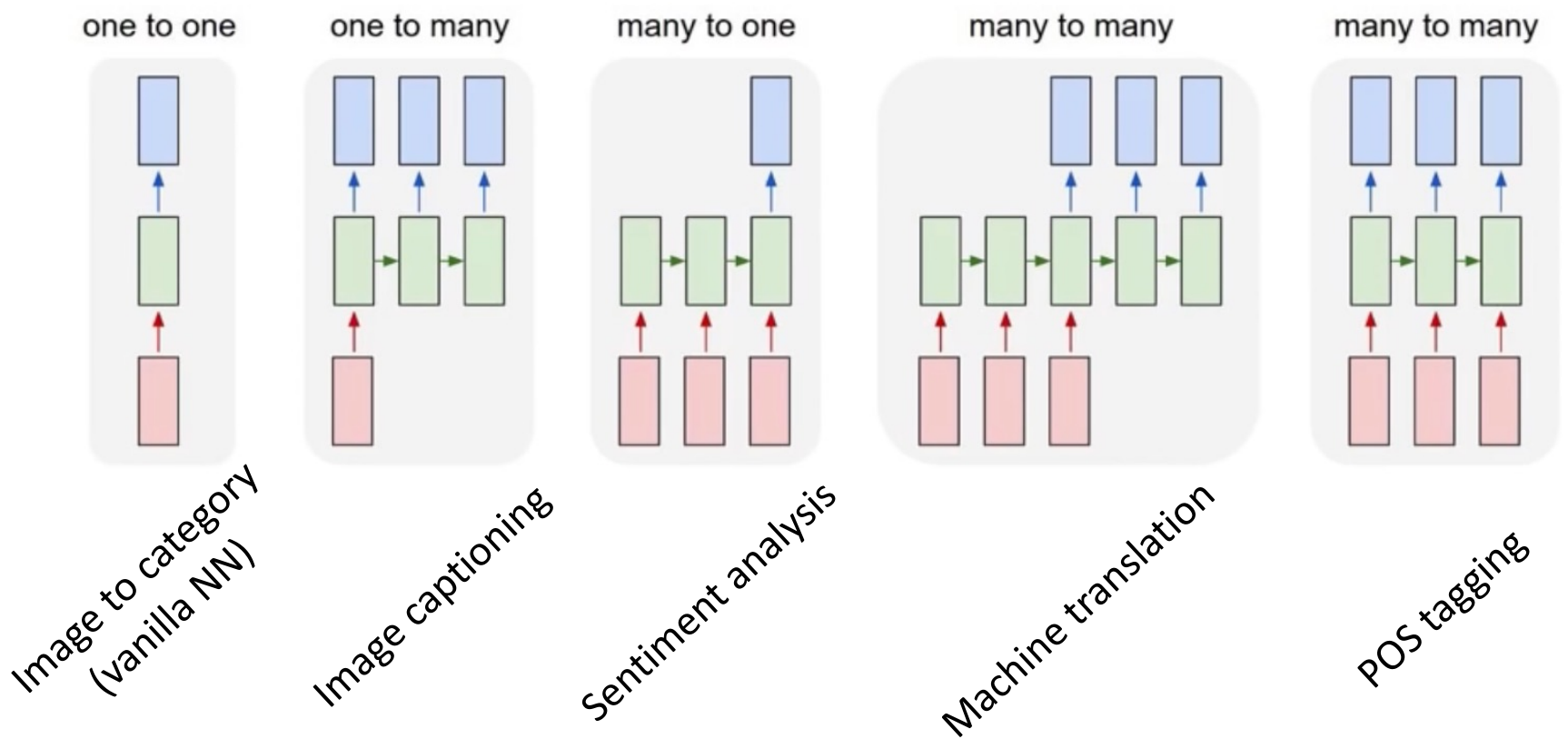


Figure 3: Deep candidate generation model architecture showing embedded sparse features concatenated with dense features. Embeddings are averaged before concatenation to transform variable sized bags of sparse IDs into fixed-width vectors suitable for input to the hidden layers. All hidden layers are fully connected. In training, a cross-entropy loss is minimized with gradient descent on the output of the sampled softmax. At serving, an approximate nearest neighbor lookup is performed to generate hundreds of candidate video recommendations.

Sequential order is useless information?

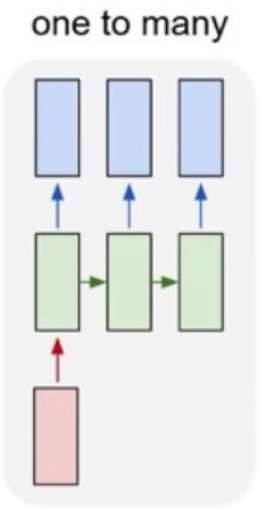
Recurrent Neural Networks: Process Sequences

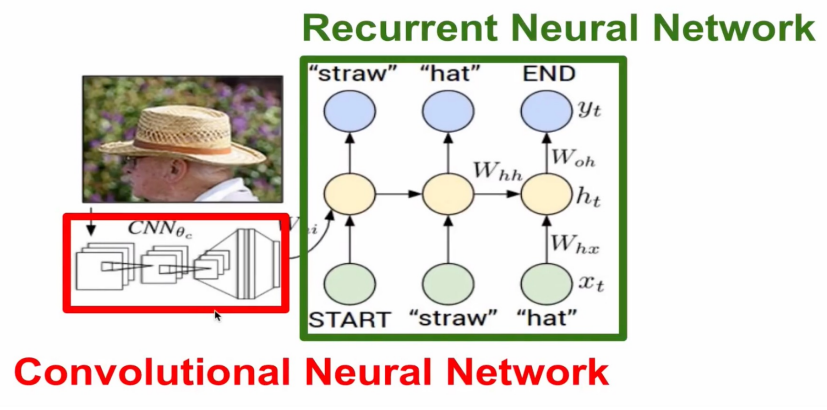
- Features in many tasks have temporal (sequential) dependency.



Recurrent Neural Networks: Process Sequences

- Image captioning
  - Image encoding (CNN) + Language model (RNN)





"man in black shirt is playing guitar."

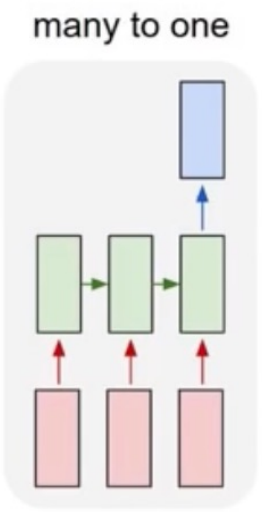
"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

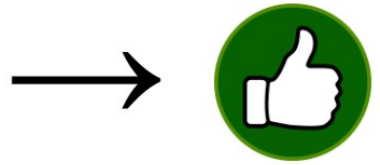
"boy is doing backflip on wakeboard."

Recurrent Neural Networks: Process Sequences

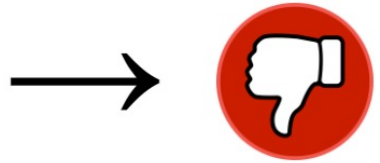
- Sentiment Analysis
  - ex) review text to a class {positive, negative} or rating



"I love this movie.
I've seen it many times
and it's still awesome."



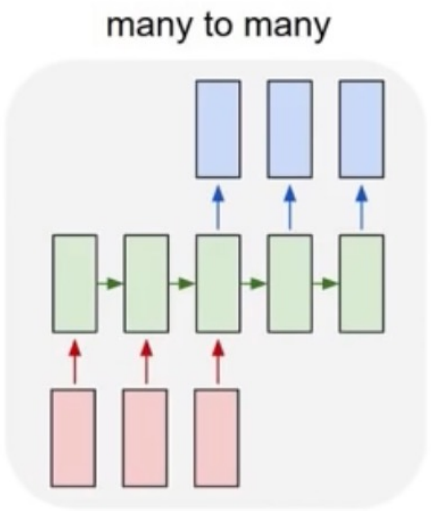
"This movie is bad.
I don't like it it all.
It's terrible."

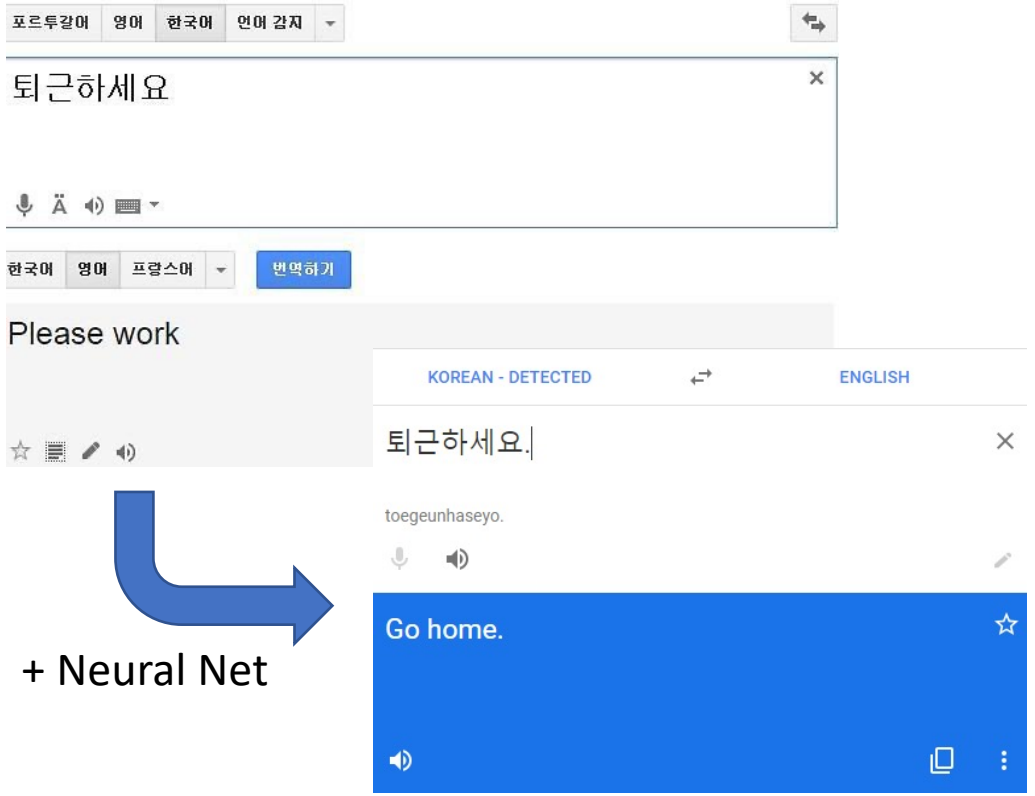


Review (X)	Rating (Y)
"This movie is fantastic! I really like it because it is so good!"	★★★★☆
"Not to my taste, will skip and watch another movie"	★★☆☆☆
"This movie really sucks! Can I get my money back please?"	★☆☆☆☆

Recurrent Neural Networks: Process Sequences

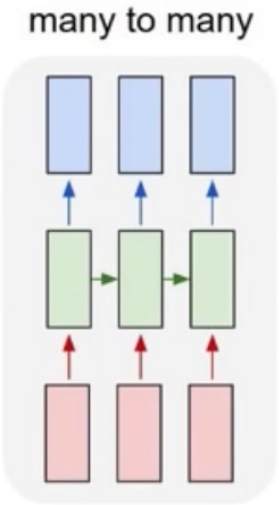
- Machine translation
  - Source language to target language (e.g., Korean to English)

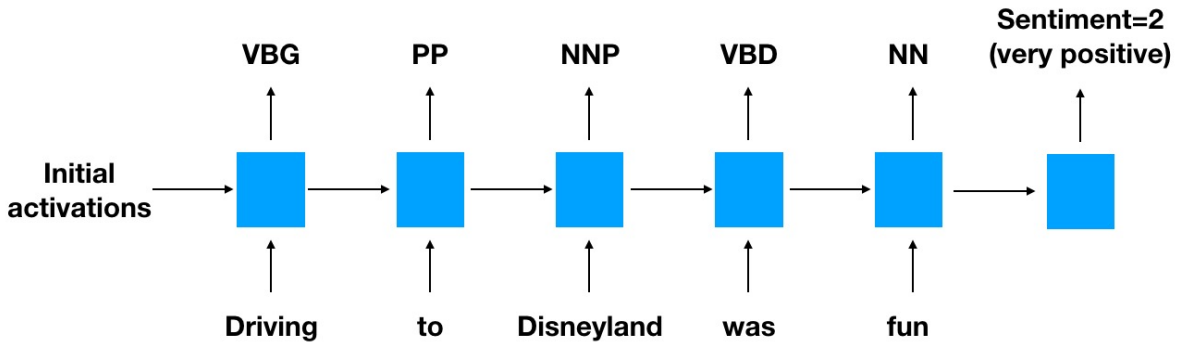


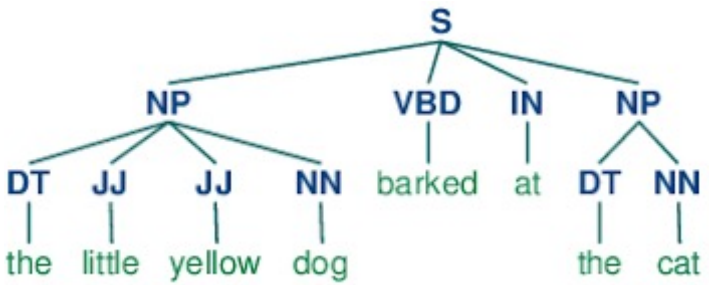


Recurrent Neural Networks: Process Sequences

- Part-of-speech (POS) tagging/parsing



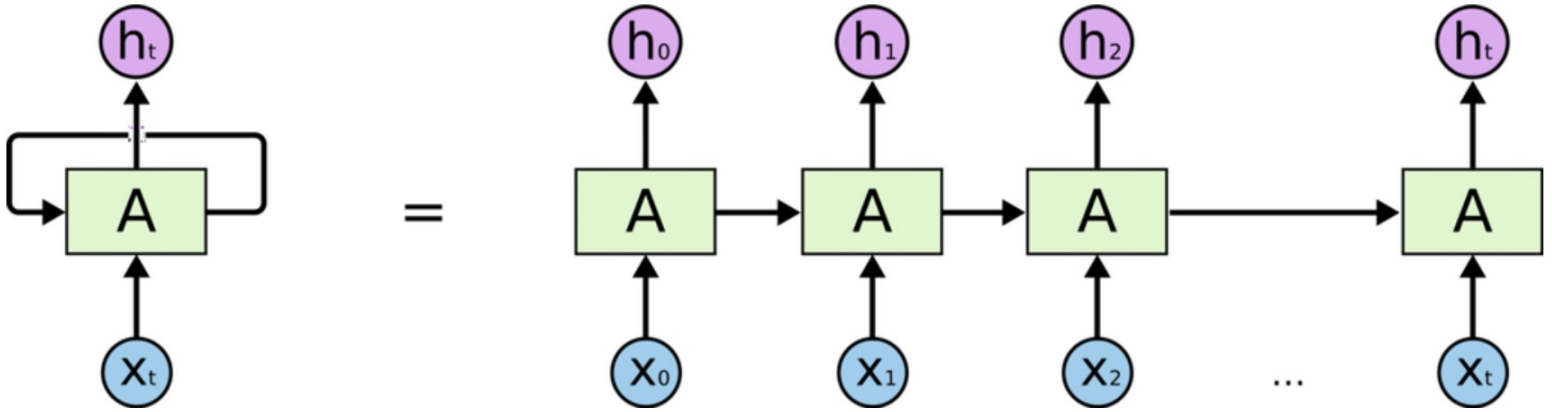




POS tagging is the process of marking up a word in a corpus to a corresponding part of a speech **tag**, based on its context and definition. This task is not straightforward, as a particular word may have a different **part of speech** based on the context in which the word is used.

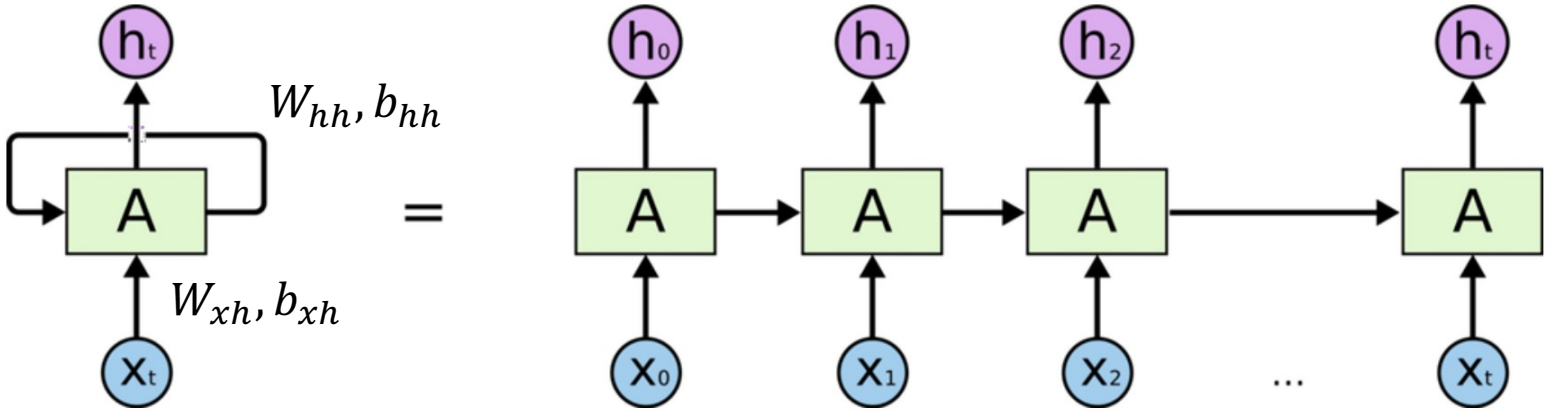
Recurrent Neural Network

$x_t \in \mathbb{R}^{|V|}, h_t \in \mathbb{R}^d$ where |V| ≫ d



Recurrent Neural Network

$x_t \in \mathbb{R}^{|V|}, h_t \in \mathbb{R}^d$ where |V| ≫ d

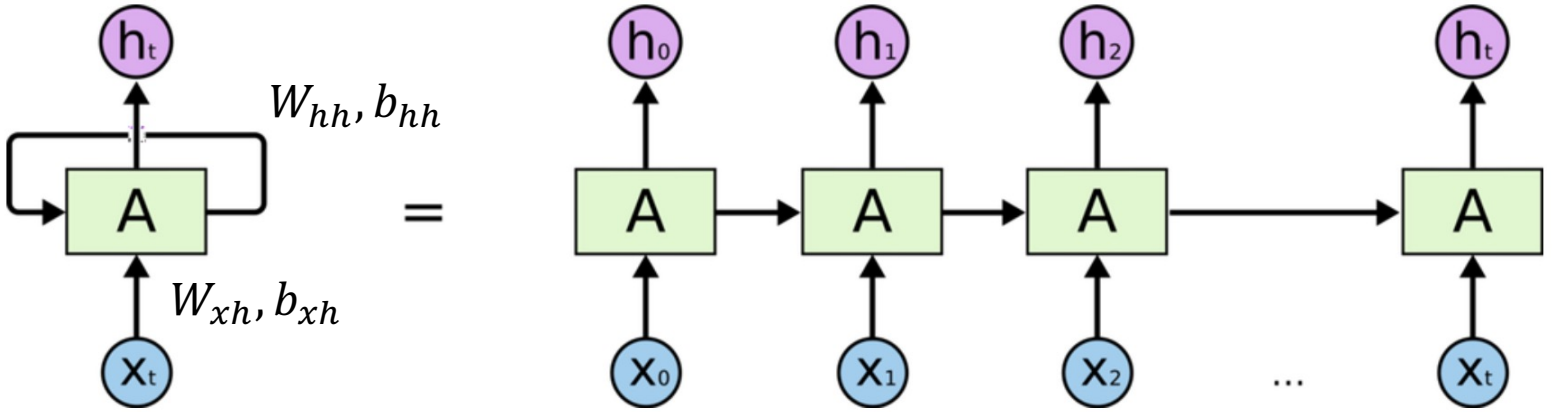


Trainable parameters $W_{xh} \in \mathbb{R}^{d \times |V|}, b_{xh} \in \mathbb{R}^d$

Trainable parameters $W_{hh} \in \mathbb{R}^{d \times d}, b_{hh} \in \mathbb{R}^d$

Recurrent Neural Network

$$x_t \in \mathbb{R}^{|V|}, h_t \in \mathbb{R}^d \text{ where } |\mathrm{V}| \gg \mathrm{d}$$



$$h_t = f(x_t, h_{t-1}; \theta) = \tanh(W_{xh}x_t + b_{xh} + W_{hh}h_{t-1} + b_{hh})$$
$$where\ \theta = \{W_{xh}, b_{xh}, W_{hh}, b_{hh}\}$$

Recurrent Neural Network: Sentiment Classification

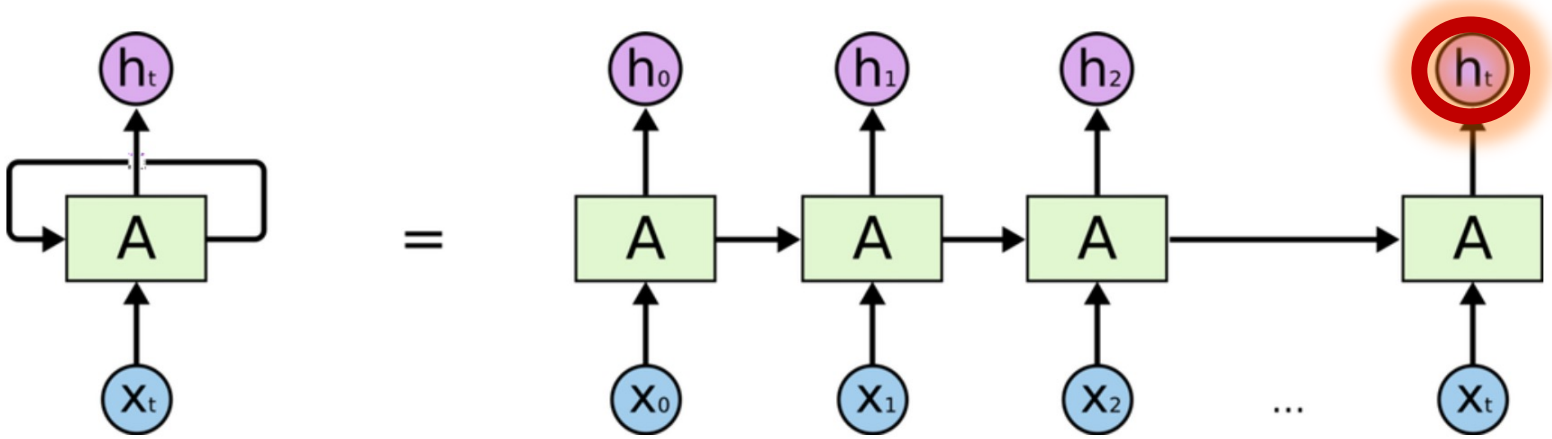
Class $\in \{\boldsymbol{positive}, \boldsymbol{negative}\}$

This movie is fun…
[one-hot representation] → [hidden representation] → [class probabilities]

$x_t \in \mathbb{R}^{|V|}$

$h_t \in \mathbb{R}^d$

$\hat{y} = softmax(W_{hy} h_t)$
$where\ W_{hy} \in \mathbb{R}^{2 \times d}$



Recurrent Neural Network: Sentiment Classification

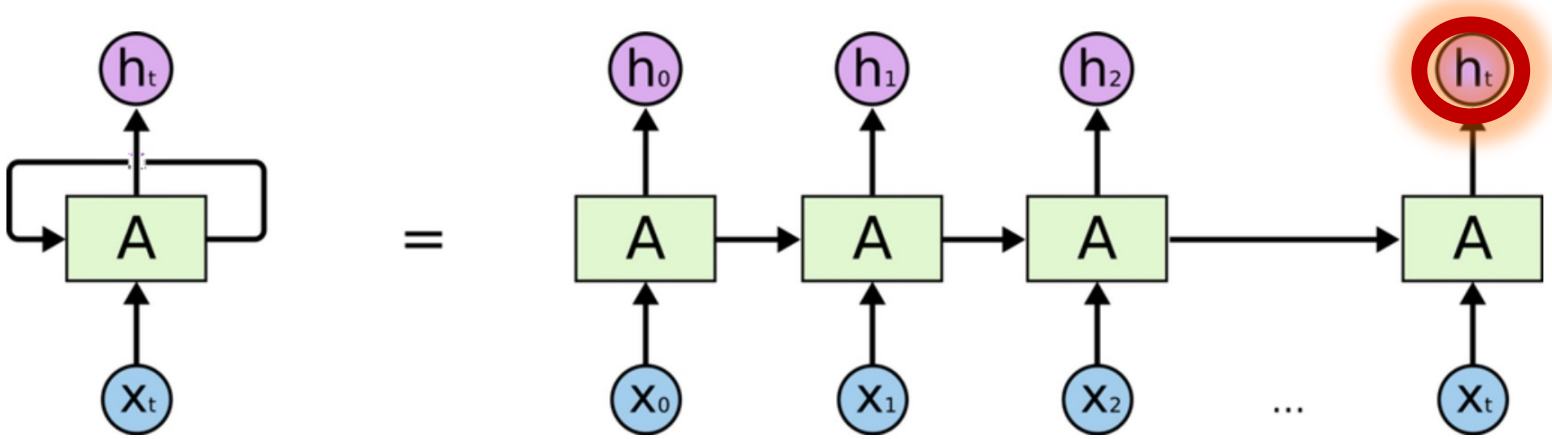
Class $\in \{\boldsymbol{positive}, \boldsymbol{not\ positive}\}$

This movie is fun…
[one-hot representation] → [hidden representation] → [class probabilities]

$x_t \in \mathbb{R}^{|V|}$

$h_t \in \mathbb{R}^d$

$\hat{y} = sigmoid(W_{hy} h_t)$
$where\ W_{hy} \in \mathbb{R}^{1 \times d}$



Recurrent Neural Network: POS Tagging

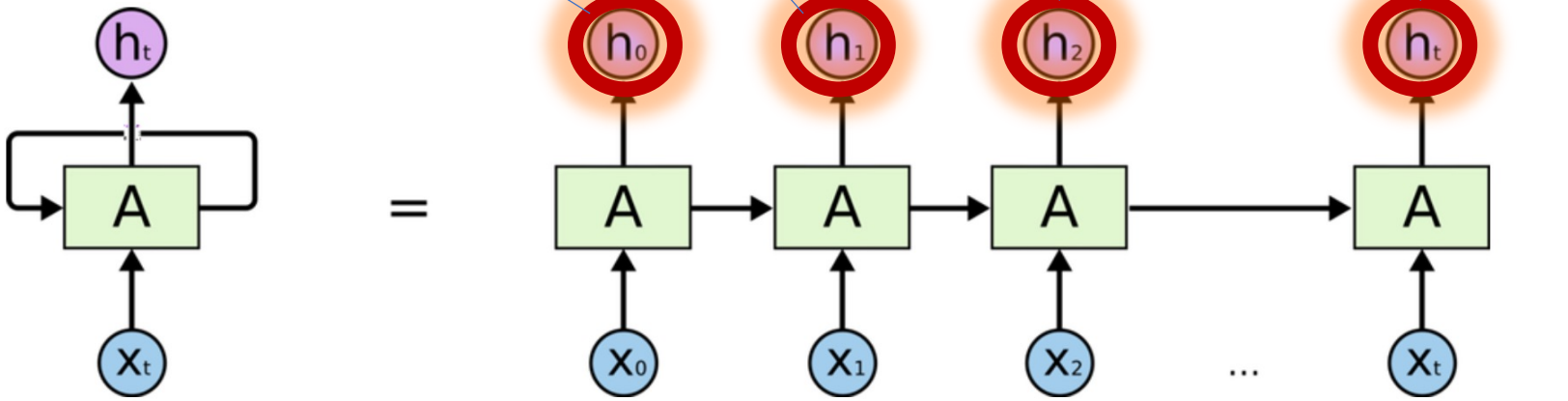
Class $\in \{\mathbf{1}.\boldsymbol{NN},\ \mathbf{2}.\boldsymbol{VB}, \ldots,\ \mathbf{36}.\boldsymbol{JJ}\}$

$\hat{y}_0 = softmax(W_{hy} h_0)$
$where\ W_{hy} \in \mathbb{R}^{36 \times d}$

$\hat{y}_1 = softmax(W_{hy} h_1)$
$where\ W_{hy} \in \mathbb{R}^{36 \times d}$

$\hat{y}_2 = softmax(W_{hy} h_2)$
$where\ W_{hy} \in \mathbb{R}^{36 \times d}$

$\hat{y}_t = \cdots$



Recurrent Neural Network (Equation)

- In each neuron of RNN, the output of previous time step is fed as input of the next time step.

We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

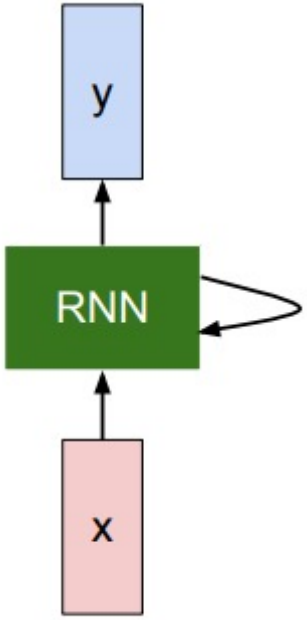
$$h_t = f_W(h_{t-1}, x_t)$$

new state

old state

input vector at some time step

some function with parameters W



Stanford University
https://www.youtube.com/watch?v=6niqTuYFZLQ
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

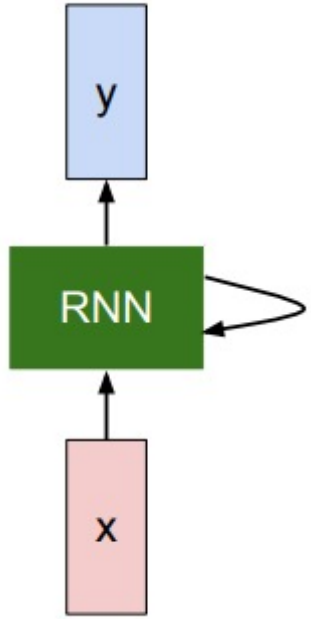
Recurrent Neural Network (Equation)

- This makes RNN be aware of temporal dependency while the Feed-forward NN has none.

We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

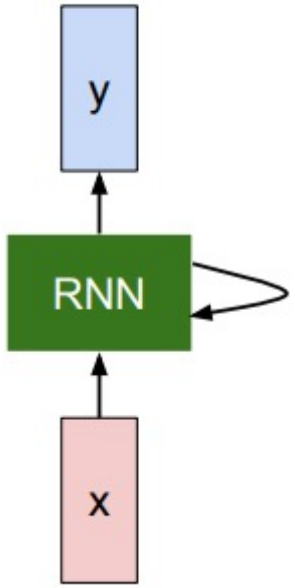
Notice: the same function and the same set of parameters are used at every time step.



Stanford University
https://www.youtube.com/watch?v=6niqTuYFZLQ
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

Recurrent Neural Network (Equation)

The state consists of a single *"hidden"* vector **h**:



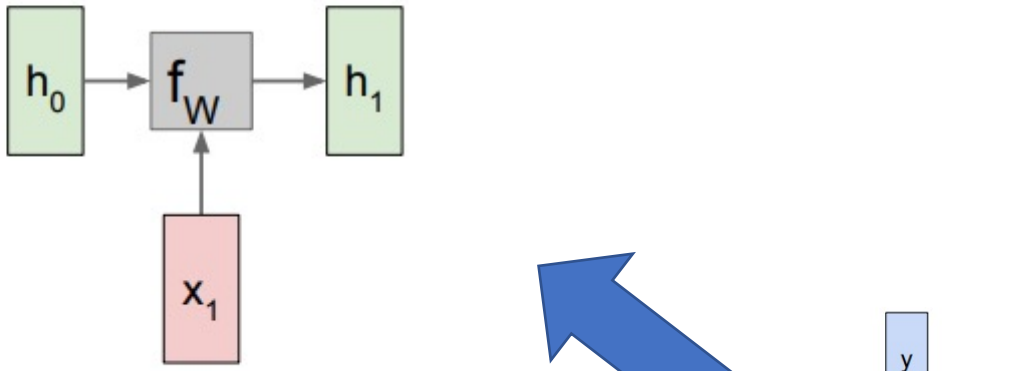
$$h_t = f_W(h_{t-1}, x_t)$$

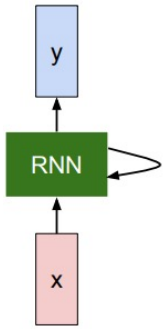
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Stanford University
https://www.youtube.com/watch?v=6niqTuYFZLQ
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

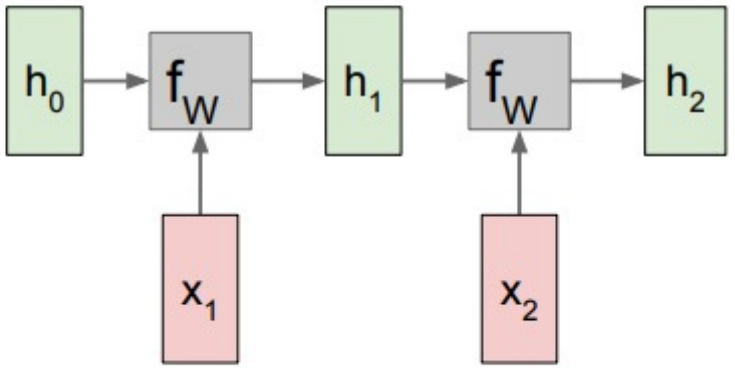
RNN: Computational Graph





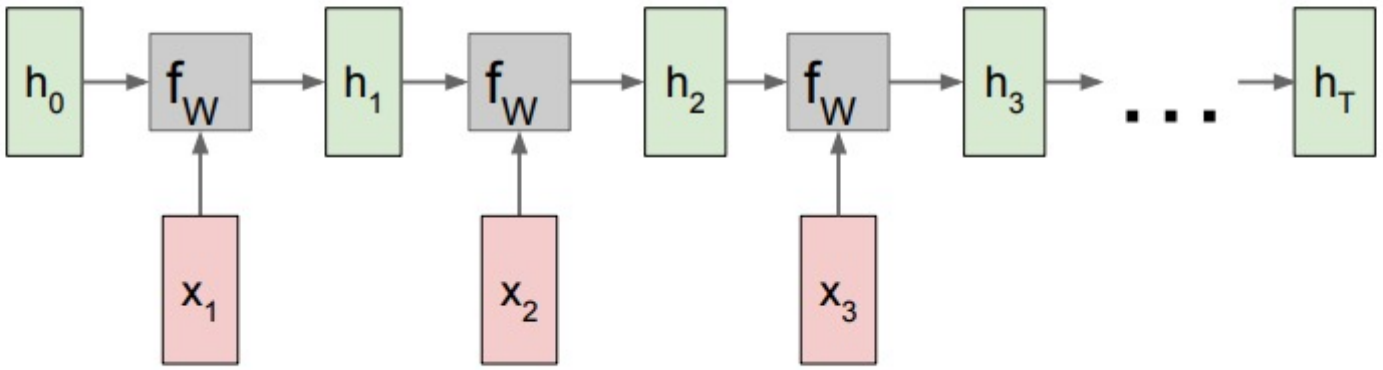
Stanford University
https://www.youtube.com/watch?v=6niqTuYFZLQ
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

RNN: Computational Graph



Stanford University
https://www.youtube.com/watch?v=6niqTuYFZLQ
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

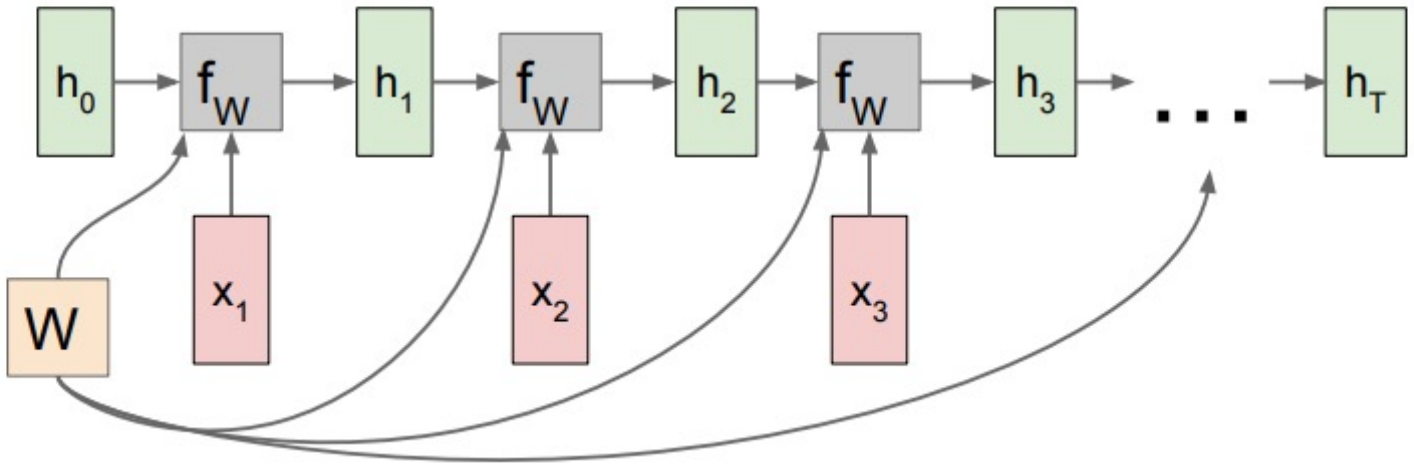
RNN: Computational Graph



Stanford University
https://www.youtube.com/watch?v=6niqTuYFZLQ
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

RNN: Computational Graph

Re-use the same weight matrix at every time-step

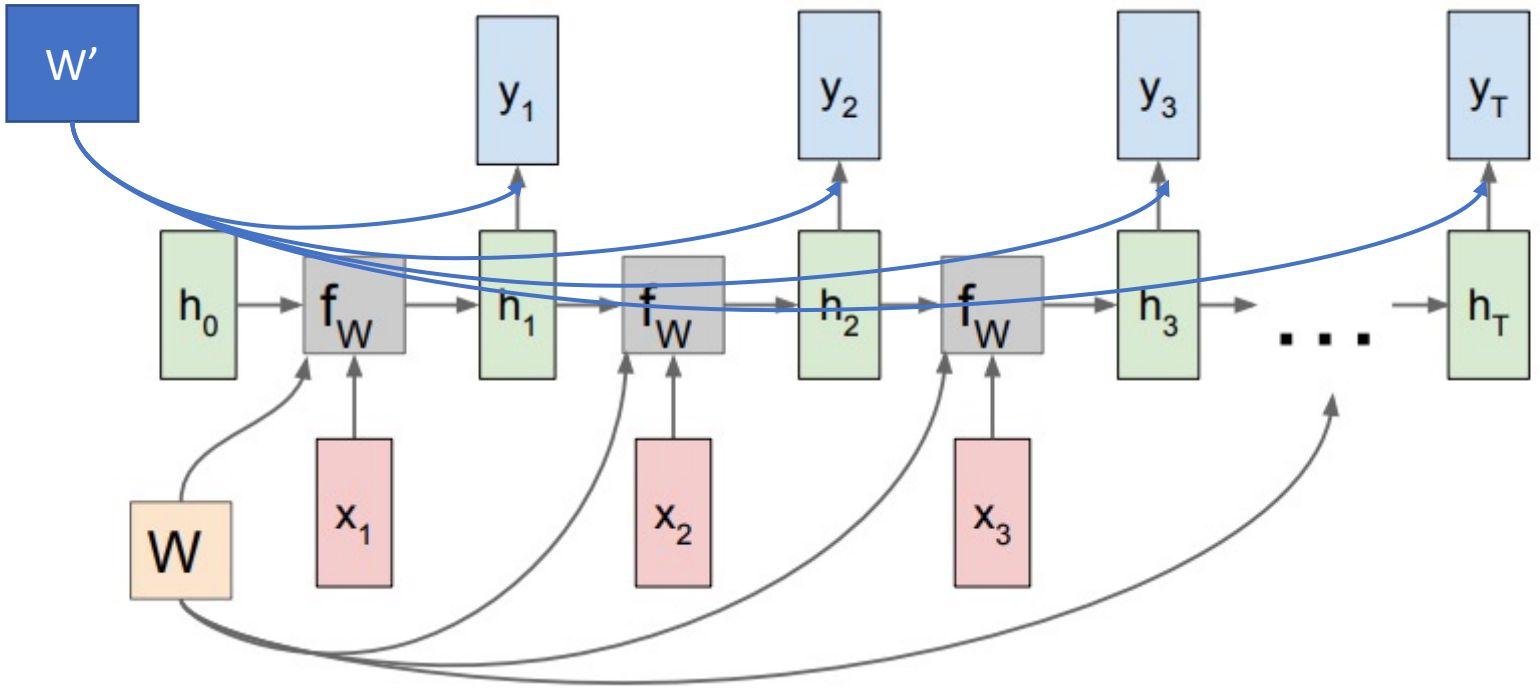


$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

Stanford University
https://www.youtube.com/watch?v=6niqTuYFZLQ
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

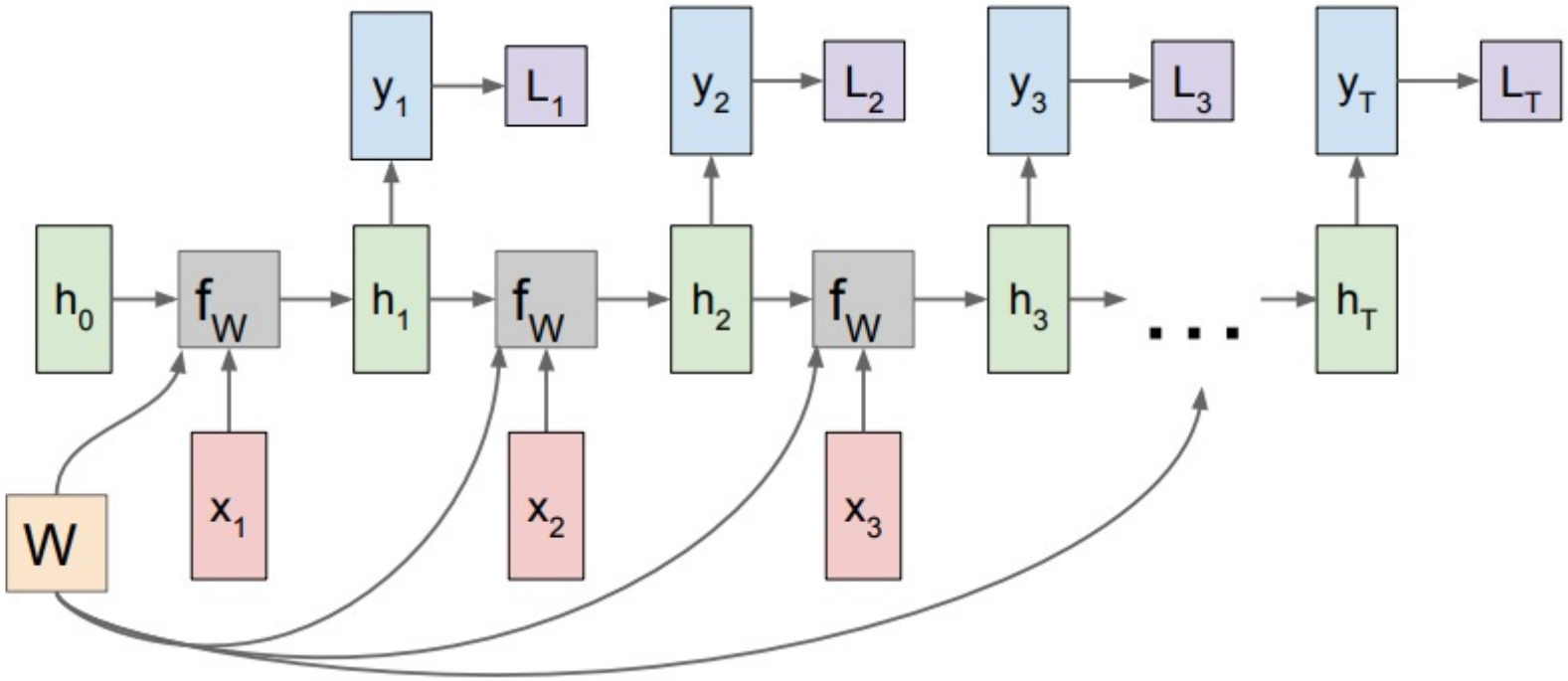
RNN: Computational Graph: Many to Many

$$y_t = W_{hy} h_t$$



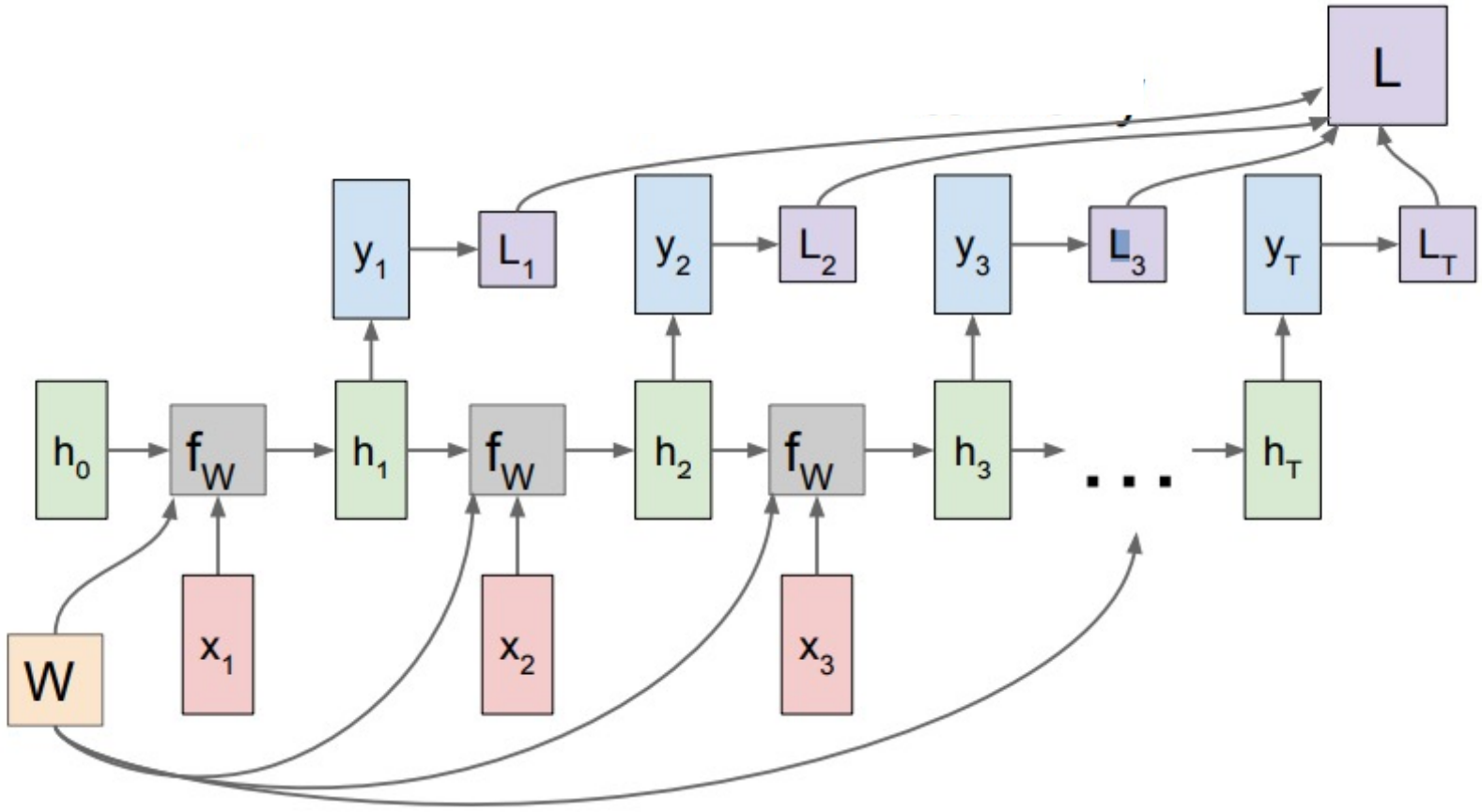
Stanford University
https://www.youtube.com/watch?v=6niqTuYFZLQ
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

RNN: Computational Graph: Many to Many



Stanford University
https://www.youtube.com/watch?v=6niqTuYFZLQ
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

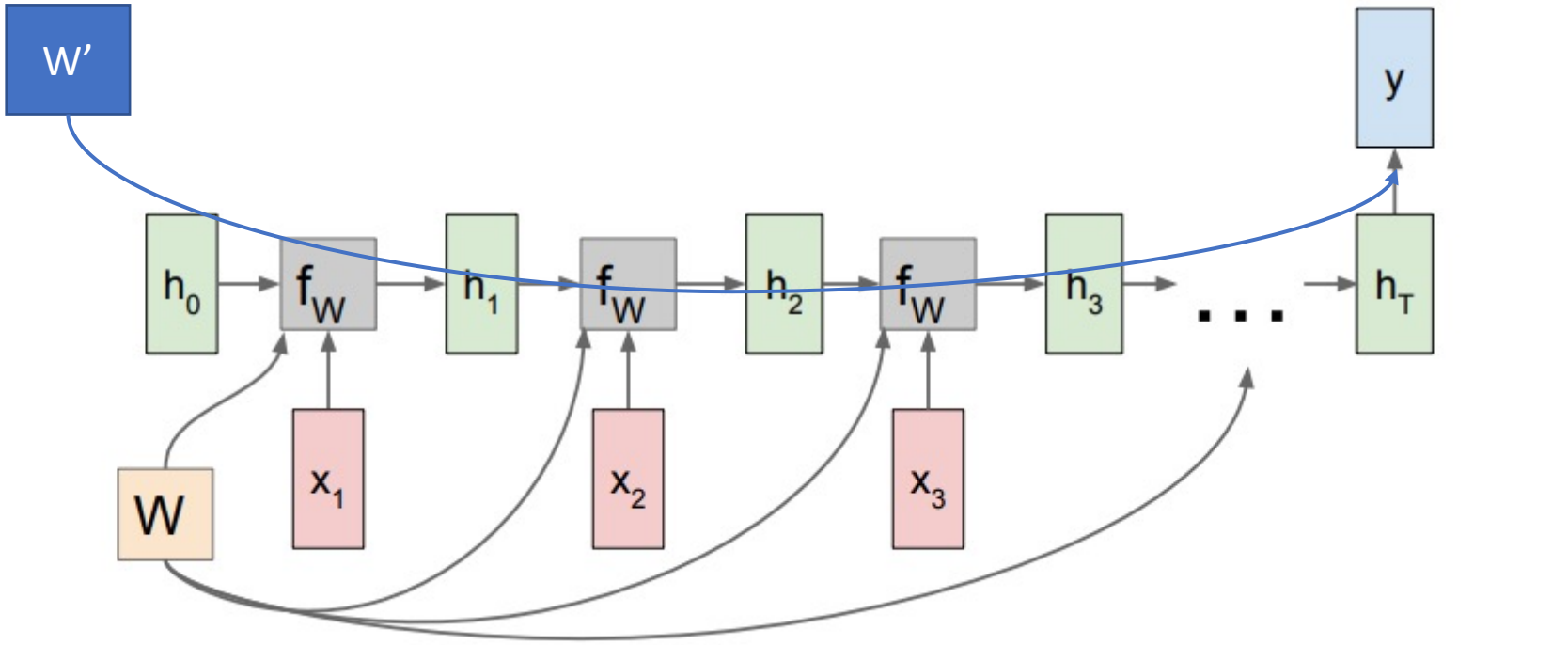
RNN: Computational Graph: Many to Many



Stanford University
https://www.youtube.com/watch?v=6niqTuYFZLQ
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

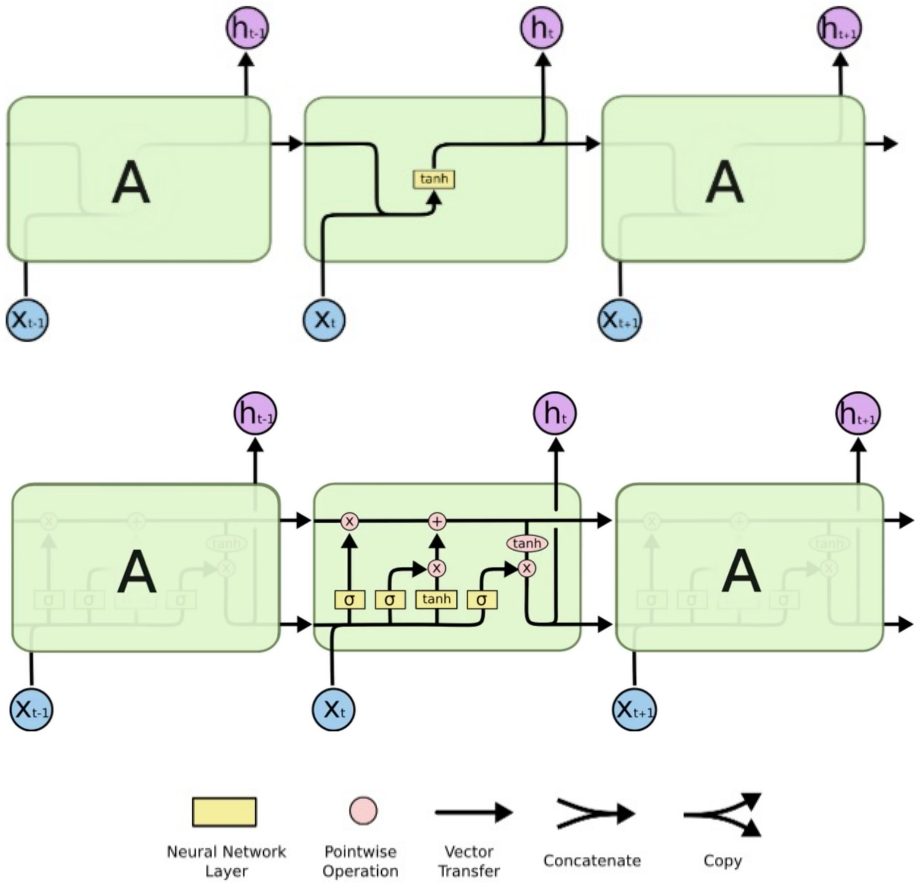
RNN: Computational Graph: Many to One

$$y_t = W_{hy} h_t$$

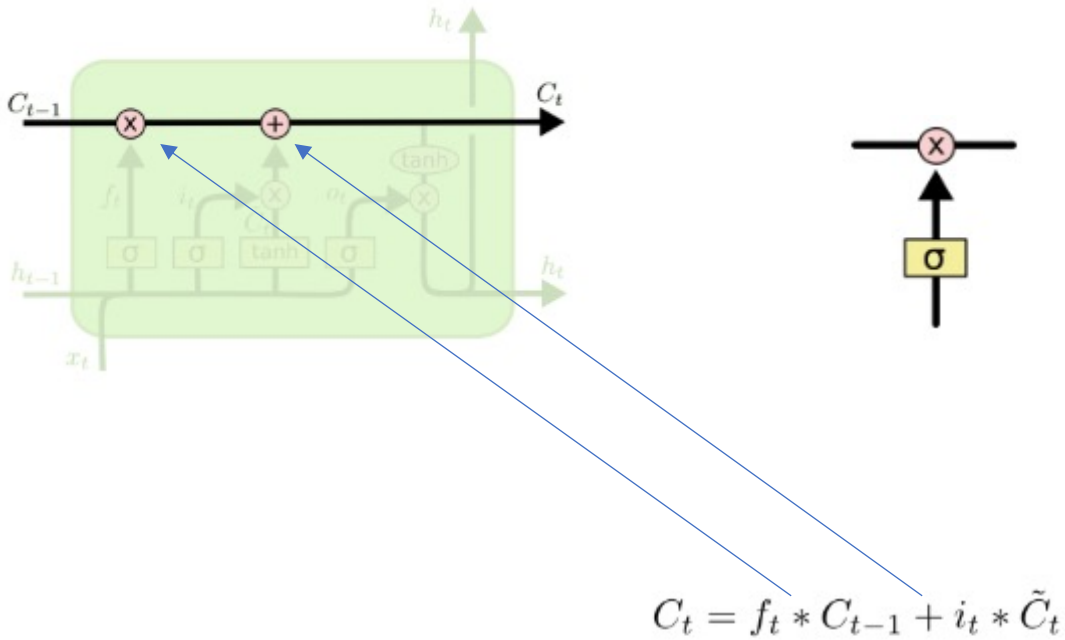


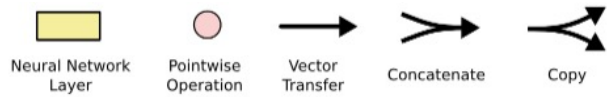
Stanford University
https://www.youtube.com/watch?v=6niqTuYFZLQ
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf

Long Short Term Memory (LSTM)

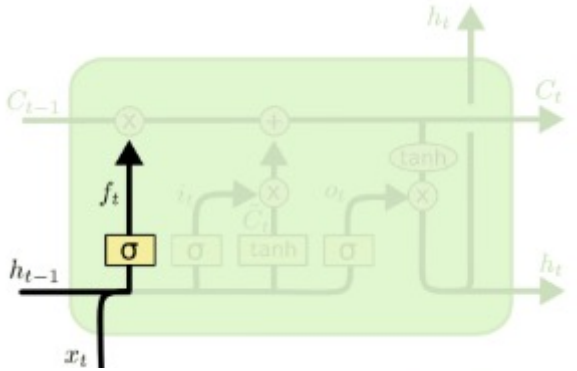


Long Short Term Memory (LSTM)

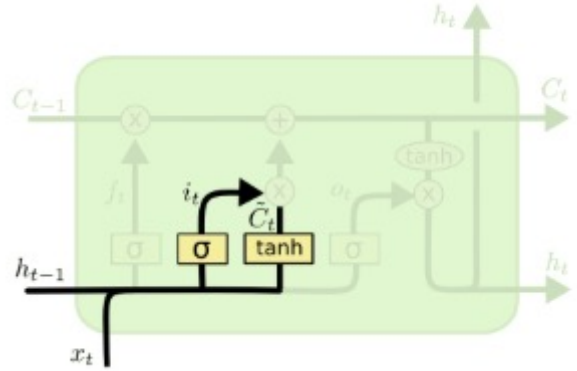




Long Short Term Memory (LSTM)

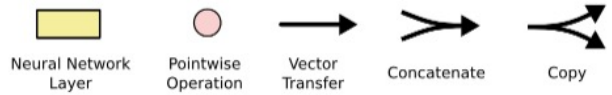


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

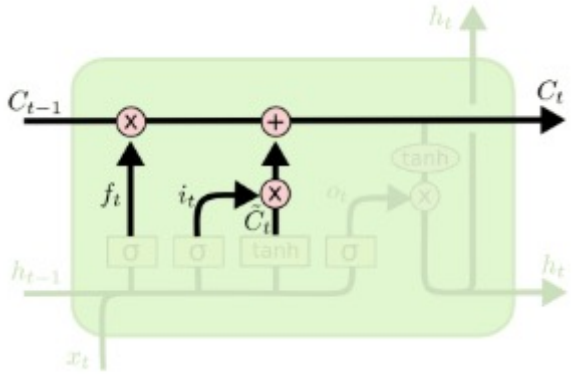


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

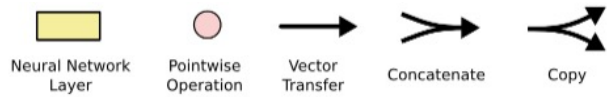
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$



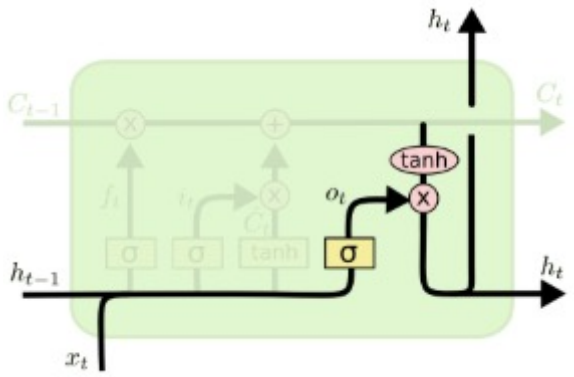
Long Short Term Memory (LSTM)



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

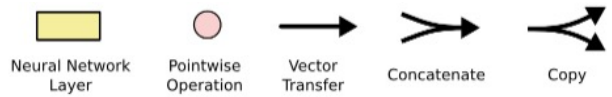


Long Short Term Memory (LSTM)

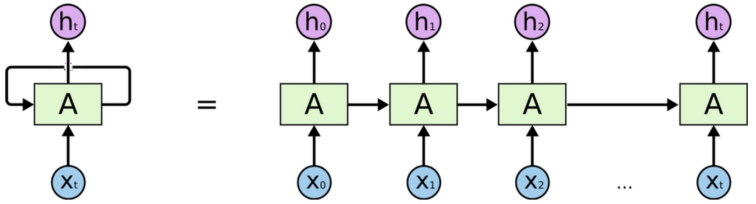


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

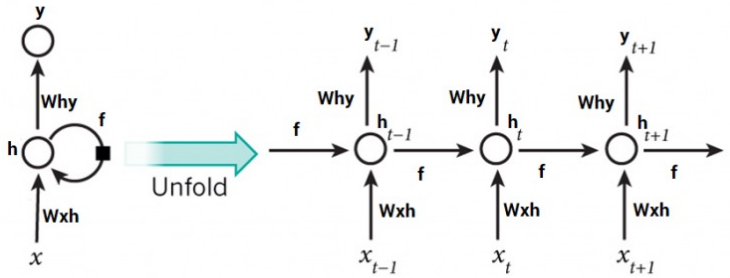
$$h_t = o_t * \tanh\left(C_t\right)$$



References

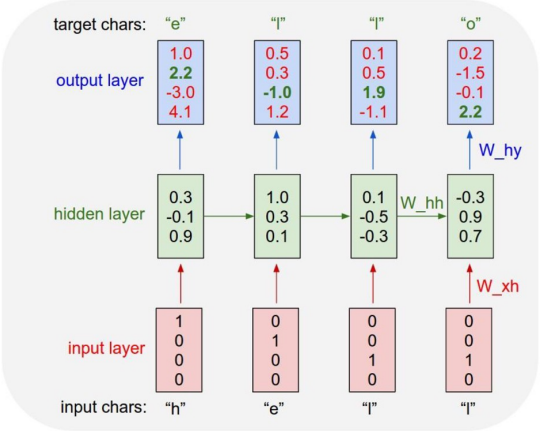


https://colah.github.io/posts/2015-08-Understanding-LSTMs/

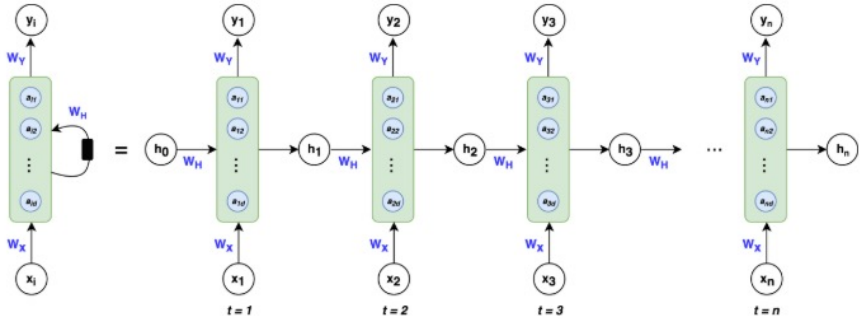


http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

https://aikorea.org/blog/rnn-tutorial-1/



http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture10.pdf



https://medium.com/towards-artificial-intelligence/whirlwind-tour-of-rnns-a11effb7808f

Thanks, any question?