Table of Contents

1. Introduction to Deep Learning *3*

Lesson 1: Welcome to the Deep Learning Nanodegree 3

Lesson 2: Knowledge, Community, and Careers 3

Lesson 3: Get Help with Your Account 3

Lesson 4: Anaconda 3

Lesson 5: Applying Deep Learning 5

Lesson 6: Jupyter Notebooks 5

Lesson 7: Matrix Math and NumPy Refresher 7

2. Neural Networks *9*

Lesson 1: Introduction to Neural Networks 9

Lesson 2: Implementing Gradient Descent 26

Lesson 3: Training Neural Networks 36

Lesson 4: GPU Workspaces Demo 41

Lesson 5: Sentiment Analysis 42

Project: Predicting Bike-Sharing Patterns 44

Lesson 7: Deep Learning with PyTorch 45

3. Convolutional Neural Networks *46*

Lesson 1: Convolutional Neural Networks 46

Lesson 2: GPU Workspaces Demo 61

Lesson 3: Cloud Computing 62

Lesson 4: Transfer Learning 63

Lesson 5: Weight Initialization 65

Lesson 6: Autoencoders 66

Lesson 7: Style Transfer 69

Project: Dog-Breed Classifier 73

Lesson 9: Deep Learning for Cancer Detection 74

Lesson 10: Jobs in Deep Learning 81

Project: Optimize Your GitHub Profile 84

4. Recurrent Neural Networks *86*

Lesson 1: Recurrent Neural Networks 86

Lesson 2: Long Short-Term Memory Networks (LSTMs) **118**
Lesson 3: Implementation of RNN & LSTM **125**
Lesson 4: Hyperparameters **127**
Lesson 5: Embeddings & Word2Vec **130**
Lesson 6: Sentiment Prediction RNN **132**
Project: Generate TV Scripts **133**
Lesson 8: Attention **134**
5. Generative Adversarial Networks ***143***
Lesson 1: Generative Adversarial Networks **143**
Lesson 2: Deep Convolutional GANs **148**
Lesson 3: Pix2Pix & CycleGAN **154**
Lesson 4: Implementing a CycleGAN **159**
Project: Generate Faces **160**
Project: Take 30 Min to Improve your LinkedIn **161**
6. Deploying a Model ***163***
Lesson 1: Introduction to Deployment **163**
Lesson 2: Building a model using SageMaker **163**
Lesson 3: Deploying and Using a Model **163**
Lesson 4: Hyperparameter Tuning **163**
Lesson 5: Updating a Model **163**
Project: Deploying a Sentiment Analysis Model **163**
Additional Lessons ***164***
Lesson 1: Evaluation Metrics **164**
Lesson 2: Regression **164**
Lesson 3: MiniFlow **164**
TensorFlow, Keras Frameworks ***165***
Lesson 1: Introduction to Keras **165**
Lesson 2: Keras CNNs **165**
Lesson 3: Introduction to TensorFlow **165**

1. Introduction to Deep Learning

Lesson 1: Welcome to the Deep Learning Nanodegree

(course logistics)

Lesson 2: Knowledge, Community, and Careers

1.1 Instructor: Cezanne & Matt & Luis

1.2 Program Structure

Lesson 3: Get Help with Your Account

1.3 Community Guidelines

1.4 Career Portal

Lesson 4: Anaconda

1.5 Anaconda

a. Create a conda environment: `conda create -n [env_name] python=[2|3]` *or* `conda -n [env_name] python=[2|3] [package names]`

b. Enter a conda environment: `conda activate [env_name`

c. List the libraries installed : `conda list`

d. Install packages (and dependencies): `conda install numpy pandas matplotlib`

e. Install Jupyter Notebook: `conda install jupyter notebook`

f. Remove a package: `conda remove [package name]`

g. Update a package version: `conda update [package name` *or* `conda update --all`

h. Search a package: `conda search [package name]` *(wildcard can be used, e.g.* `conda search '*nump*'`*)*

i. Exit from a conda environment: `conda deactivate`

j. Export a conda environment to a YAML file: `conda env export > conda_environment.yaml`

k. Create an environment with the name specified in the file from an YAML file: `conda env create -f conda_environment.yaml`

l. List all conda environments: `conda env list` *`(* is the current env)`*

m. Remove an environment: `conda env remove -n [env name]`

n. Practices :

- Create Python 2 environment: `conda create -n py2 python=2`
- Create Python 3 environment: `conda create -n py3 python=3`
- Create an conda environment for each project
- It is a good practice to share your conda & pip environment using `conda env export > conda_environment.yaml` and `pip freeze > pip_requirements.txt` while sharing projects on GitHub

1.6 Anaconda & Miniconda

a. Anaconda

Anaconda is a distribution of software that comes with conda, Python, scientific packages and their dependencies.

b. Miniconda

Miniconda is a smaller distribution that includes only conda and Python

c. Conda

Conda is a program (a package and environment manager). It is similar with `pip`, default package manager

`pip` focuses on general use, while `conda` focuses on data science

1.7 Python 2 and Python 3

a. a print function works for Python 2.6+

```
from __future__ import print_function
print("Hello World")
```

Lesson 5: Applying Deep Learning

1.8 Deep learning applications

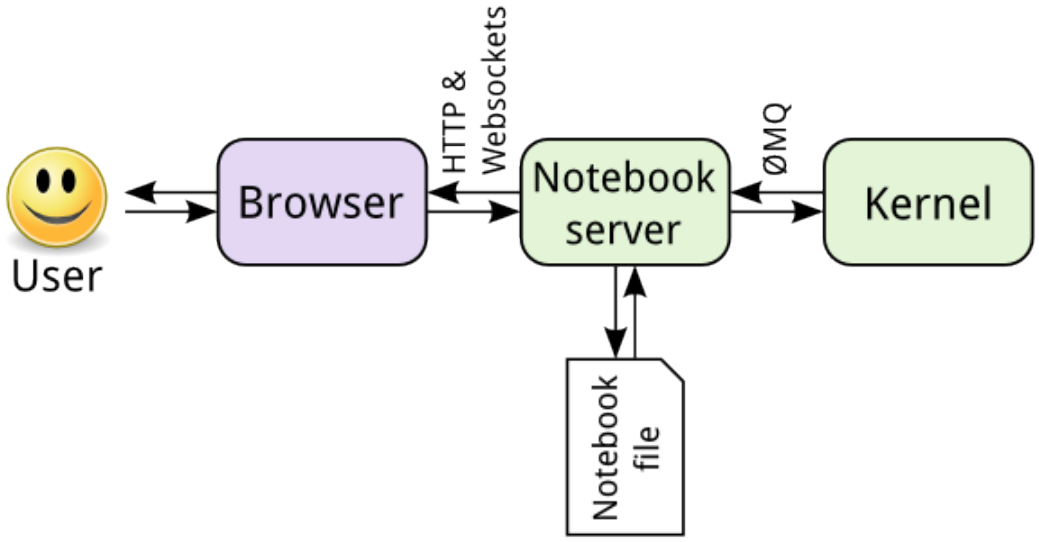
a. Style Transfer (go ./reference applications/Fast Style Transfer)

b. Deep Traffic (go ./reference applications/Deep Traffic/)

c. Flappy Bird (go ./reference applications/Flappy Bird/)

Lesson 6: Jupyter Notebooks

1.9 Jupyter Notebook



(image from Jupyter documentation)

Notebook seen by browser is a web application.

The web application connects Python/R/other kernel through notebook server.

Kernel runs Python/R code and sends back the results to server, then is rendered in the browser.

Saving the code actually saves a ".json" file on the server and a code text file ".ipynb"

Advantages:

1) Notebook and kernel are separate, so code in any language can be sent between server and kernel.

2) We can access to a remote server at anywhere where the data and notebook files are stored, e.g. Amazon EC2.

Install Jupyter Notebook:

Run `conda install jupyter notebook` or `pip install jupyter notebook`

Launch jupyter notebook:

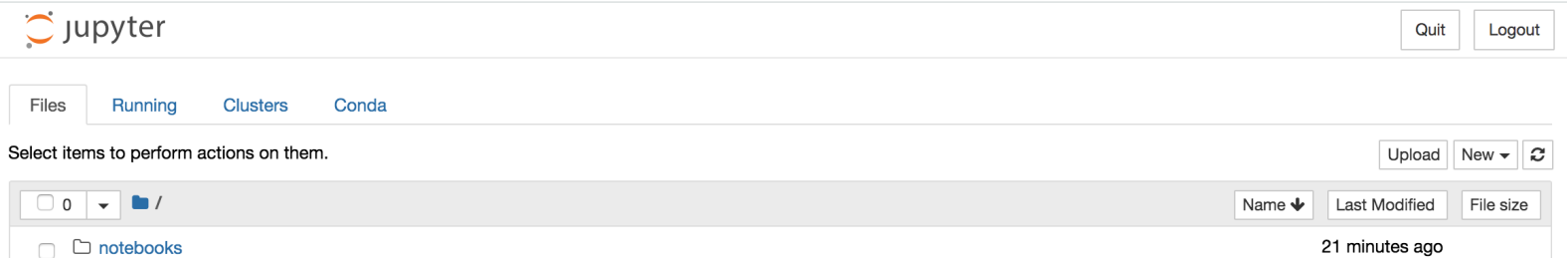
Run `jupyter notebook` on the terminal

Default URL: `http://localhost:8888`

Install kernels: https://ipython.readthedocs.io/en/latest/install/kernel_install.html

Install Notebook conda: Run `conda install nb_conda` helps to manager conda environments

Panels



1) Files: files & folders

2) Running: list all currently running notebooks

3) Clusters: is taken over by *ipyparallel*

4) Conda: help manager conda environments

Attention:

1) Before shutting down, save all the changes in notebook

Tooltips:

1) Shift + tab: show brief documentation

2) Shift + tab (press tab twice): show full documentation

1.10 Convert a Jupyter Notebook to HTML

Notebooks are big JSON files with extension ".ipynb"

Use command `jupyter nbconvert -to html notebook.ipynb` to convert the notebook to HTML. It creates a .html file with the same name as the notebook.

1.11 Slideshow

View > Cell Toolbar > Slideshow

Slides	full slides that can move left to right
Sub-slides	sub-slides that can press up or down
Fragment	hidden at first, then appear with a button press
Skip	hidden, speaker notes
Notes	hidden, speaker notes

A Slideshow example:
https://nbviewer.jupyter.org/format/slides/github/jorisvandenbossche/2015-PyDataParis/blob/master/pandas_introduction.ipynb#/

Convert to a Slideshow: run `jupyter nbconvert [notebook name] --to [Slideshow name]`

Convert and show: run `jupyter nbconvert [notebook name] -to [Slideshow name] -post serve`

Lesson 7: Matrix Math and NumPy Refresher

1.12 Scalar

Scalar is a single number, zero dimension

1.13 Vector

Vector has 1 dimension

Row vector $[1\ 2\ 3]$

Column vector $[1\ 2\ 3]^T$

1.14 Matrices

Matrices are grids, 2 dimensions

1.15 Tensors

Tensor refers to any n-dimensional collection of values

Scalar: 0-d tensor

Vector: 1-d tensor

Matrices: 2-d tensor

1.16 **More notes: see *../ notebooks (filled) / 1. Introduction to Deep Learning / 4. Numpy refresher.ipynb***

2. Neural Networks

Lesson 1: Introduction to Neural Networks

1.1 Classification problems

Terminologies:

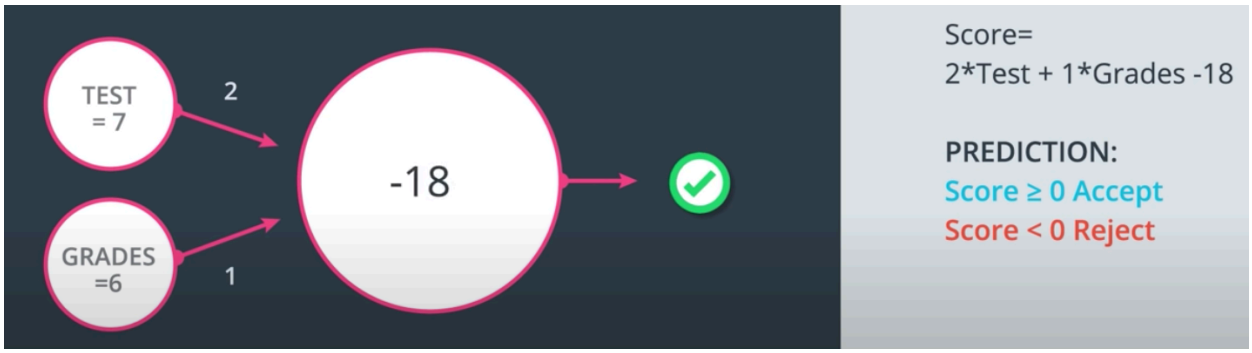
a. Data

b. Prediction $\hat{y}$

c. Boundary

1.2 Perceptron

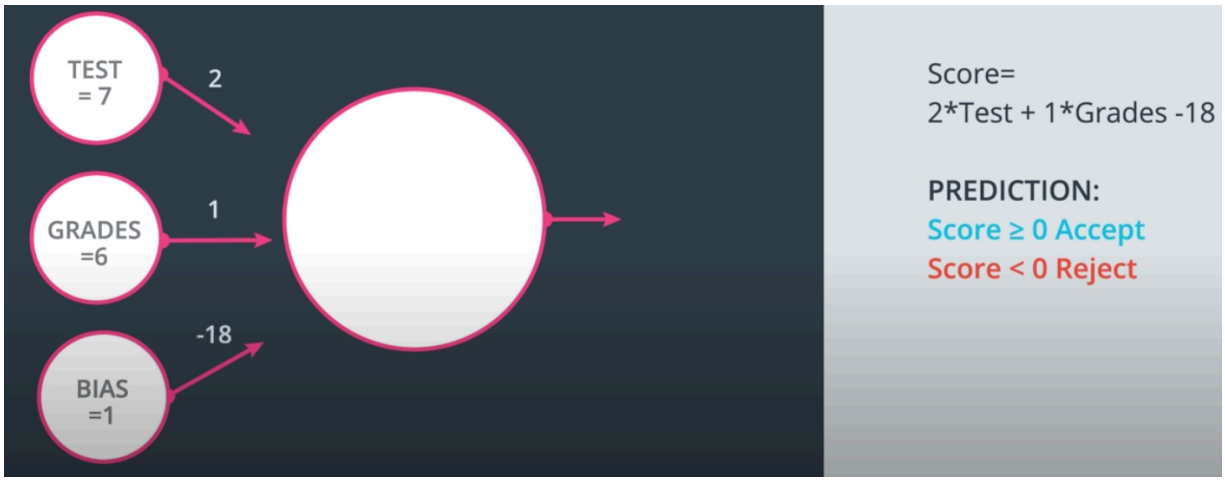
Way 1 to represent a perceptron:



(image source from Udacity)

Weights $w = [2, 1]$	Bias $b = [-18]$
Instance $x = \big[[7], [6]\big]$	Prediction $\hat{y} = w \cdot x + b$

Way 2 to represent a perceptron (treat bias as a neuron in the input layer) :



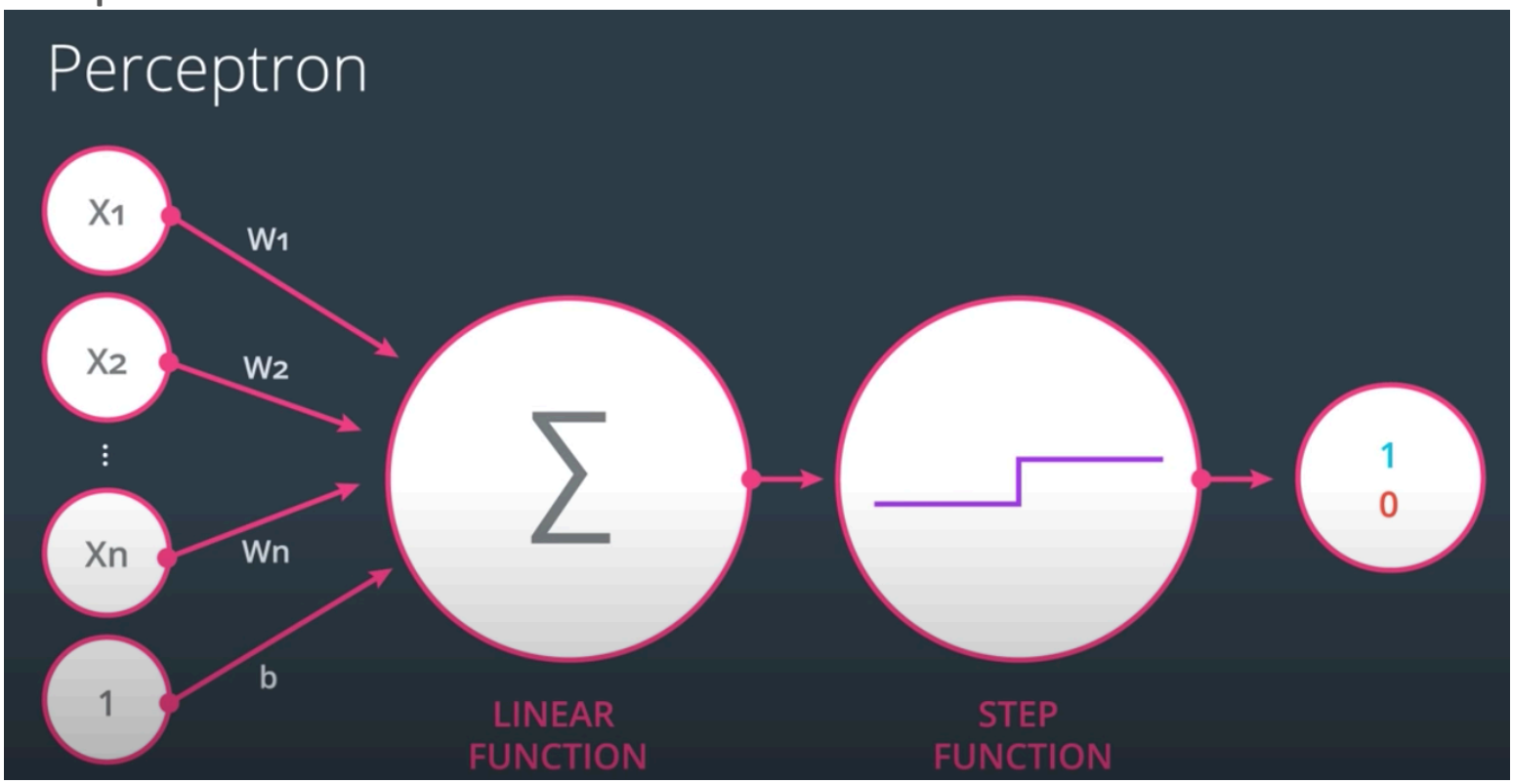
(image source from Udacity)

Weights $w = [2, 1, -18]$	Bias $b = [1]$
Instance $x = \left[[7], [6], [1]\right]$	Prediction $\hat{y} = w \cdot x + b$

Here we doesn't apply any activation function, or we can say, we apply step function:

$$\hat{y} = \begin{cases} 0, & w \cdot x + b < 0 \\ 1, & w \cdot x + b \geq 0 \end{cases}$$

Perceptron:



(image source from Udacity)

Input layer	weights	weighted sum	activation function	prediction (activation)

1.3 How to modify the boundary if there is a misclassified point

First, we need to have a learning rate α

Then,

If the true label of the misclassified point is positive but predicted as negative, we multiply the point with the α, then add the result on the weights w;

If the true label of the misclassified point is negative but predicted as positive, we multiply the point with the ε, then subtract the result from the weights w.

1.4 Perceptron Algorithm

```
Start with random weights w1, w2, ..., wn, b

For every misclassified point x (x1, x2, ..., xn)
    if true_label(x) == pos i.e. predict(x) == neg
        for i = 1, 2, ..., n
            wi += α * xi
        b += α
    else if true_label(x) == neg i.e. predict(x) == pos
        for i = 1, 2, ..., n
            wi -= α * xi
        b -= α
```

1.5 Refine the Perceptron Algorithm so that the boundary can be a curve, not just a straight line

Adjustment: We add an **Error Function** telling us how far the current boundary is from the optimal

Error Function:

a. Error function should be continuous, not discrete.
b. Error function also should be differentiable.
c. Here are some normal error functions:

Log-loss Error Function

Gradient Descent: take a small step that makes the error function descend the most

Note: We may end up with a local minimum during the process.

For using a continuous error function to optimize the boundary, we need to change predictions from discrete (0/1) to continuous (0.9/0.3).

I.e. we change our activation function. From step function $y(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$ (discrete activation function) to sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ (continuous activation function)

Prediction $\hat{y}$ changing from $y(W \cdot x + b) \rightarrow \sigma(W \cdot x + b)$

If we define the boundary as $score = W \cdot x + b$, the point x who makes score 0 has 50% probability to be positive or negative.

1.6 Multi-class classification

Softmax

Instead of using sigmoid as activation function, we use **softmax** as activation function

$$softmax(x) = \frac{e^x}{\sum_{i \in all\ classes\ C} e^i}$$

Note: When $|C| = 2$, $softmax == sigmoid$

Proof:

TODO

One-hot encoding:

When the feature is not numerical, we have 2 options:

Option 1: Change to values (worse)

Bear	0
Duck	1
Dog	2

The reason why it is worse is that it creates dependencies (Dog > Duck > Bear).

Option 2: Use one-hot encoding (better)

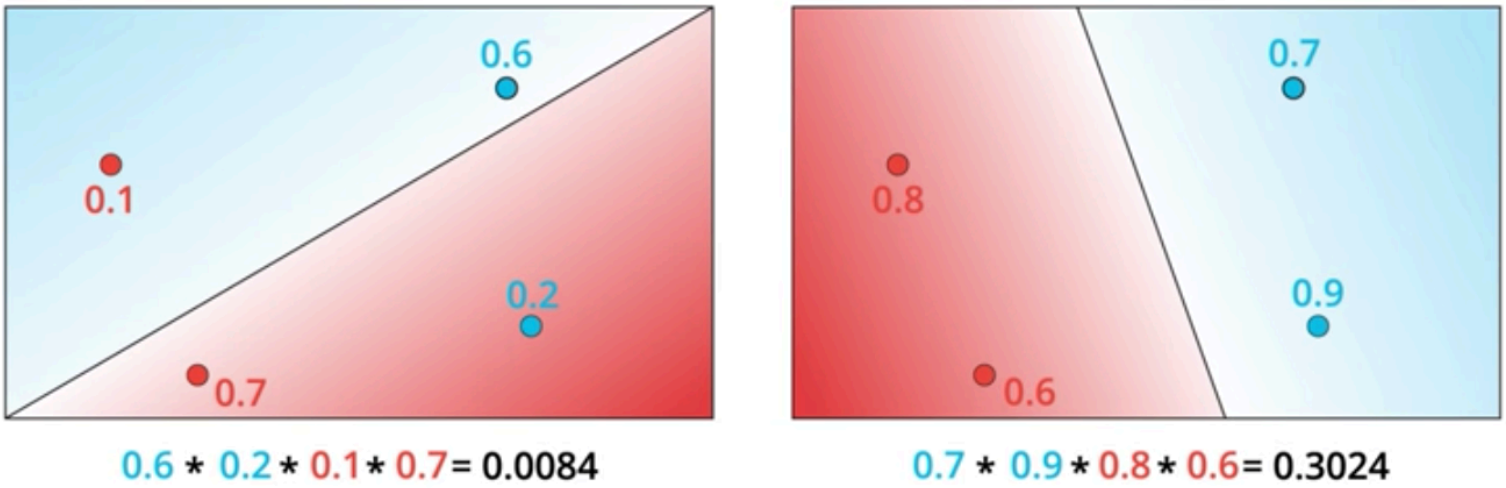
	Is bear?	Is Duck?	Is Dog?	One-hot encoding
Bear	1	0	0	$[1, 0, 0]$
Duck	0	1	0	$[0, 1, 0]$
Dog	0	0	1	$[0, 0, 1]$

1.7 Maximum Likelihood

Maximum likelihood means that the model gives the highest probability to all the existing point with respect to their true label, i.e.

$$\prod_{x \in X} p(x) \quad where\ p(x)\ is\ the\ probability\ such\ that\ \hat{y} == y$$

$\prod_{x \in X} p(x)$ indicates how accurate the model is on all the existing points.



(image source from Udacity)

maximum likelihood = 0.0084	maximum likelihood = 0.3024

Refine Maximum Likehood to avoid "0.000…000006" things

Step 1: Turn "*" to "+" **using logarithm**

$$\log\left(\prod_{x \in X} p(x)\right) = \sum_{x \in X} \log\left(p(x)\right) \quad where\ p(x)\ is\ the\ probability\ such\ that\ \hat{y} == y$$

Step 2: Because $p(x) \in [0, 1]$, so $\log\big(p(x)\big) \leq 0$. We make $\log\big(p(x)\big) \leq 0$ positive by taking negation.

$$Cross\ Entropy = \sum_{x \in X} -\log\left(p(x)\right) \quad where\ p(x)\ is\ the\ probability\ such\ that\ \hat{y} == y$$

Cross Entropy =	Cross Entropy =

$\sum_{x \in X} -\log(p(x))$ $= -\log(0.6) - \log(0.2)$ $-\log(0.1) - \log(0.7) = 4.8$	$\sum_{x \in X} -\log(p(x))$ $= -\log(0.7) - \log(0.9)$ $-\log(0.8) - \log(0.6) = 1.2$

Attribute: Note that **a good model gives a low Cross Entropy, a bad model gives a high cross entropy**. Because a good model gives great values of $p(x)$ between 0 and 1, then $-\log(p(x))$ is a small positive number; whereas a bad model gives small values of of $p(x)$ between 0 and 1, then $-\log(p(x))$ is a large positive number.

We can take advantage of the attribute and **use Cross Entropy as Error on all existing points**. So **our goal changing from "Maximizing maximum likelihood → Minimizing Cross Entropy"**

Step 3: Instead of using a sentence to describe in the forumula, we replace it by its true label

$$Cross\ Entropy = -\sum_{i=1}^{|X|} \{y_i \log(p(x_i)) + (1 - y_i) \log(1 - p_i)\} \quad (1)$$

$$where\ y_i\ is\ the\ true\ label\ of\ i^{th}\ instance$$

Cross Entropy **tells how similar the two vectors are**, gives small value if similar, gives high value if different.

The Cross Entropy works well for 2 classes. What if more classes ?

Multi-class Cross Entropy

$$Cross\ Entropy = -\sum_{i=1}^{|X|} \sum_{j=1}^{m} \{y_{ij} \log(p_{ij})\} \quad (2)$$

$$where\ there\ are\ m\ classes\ in\ total, and$$

$$p_{ij}\ is\ the\ probability\ of\ i^{th}\ instance\ belongs\ to\ class\ j, and$$

$$y_{ij}\ indicates\ whether\ i^{th}\ instance\ belongs\ to\ class\ j, if\ yes\ 1, no\ 0$$

Note that when m = 2, Cross Entropy (1) == Cross Entropy (2)

Proof:

TODO

Recap:

Cross Entropy is **the negative of the summation of the logarithmes of the probabilities of the points being their true labels for all instances**, is **inversely proportional to the total probability of an outcome**.

For binary classification problems:

$$Error\ (on\ each\ instance) = -(1-y)\,(\log(1-\hat{y})) - y\log(\hat{y})$$

$$where\ y\ is\ the\ true\ label, \hat{y}\ is\ the\ prediction$$

$$(Average) Error\ Function\ (on\ all\ instances) = -\frac{1}{m}\sum_{i=1}^{m}\{\,(1-y_i)\log(1-\hat{y}_i) + y_i\log(\hat{y}_i)\,\}$$

$$where\ m\ is\ the\ no.\ of\ instances\ so\ that\ here\ is\ average\ error\ over\ all\ instances, and$$

$$\hat{y}_i = \sigma\left(W\cdot x^{(i)} + b\right) = \frac{1}{1+e^{W\cdot x^{(i)}+b}}\quad (x^{(i)}\ is\ i^{th}\ instance, W\ is\ weights, b\ is\ bias)$$

For multi-class classification problems:

$$Error\ Function = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n}\{\,y_{ij}\log(\hat{y}_{ij})\,\}\ where\ j\ indicates\ class, i\ indicates\ instance$$

Note, **when n =2, (Average) Error Function (on all instances) == Error Function**

Proof:

TODO

1.8 Logistic Regression (on binary classification)

Minimize the Error Function E(W, b)

$$(Average) Error\ Function\ (on\ all\ instances) = -\frac{1}{m}\sum_{i=1}^{m}\{\,(1-y_i)\log(1-\hat{y}_i) + y_i\log(\hat{y}_i)\,\}$$

using gradiant descent, so that we get W' and b' which reach the minimum E(W', b') and give the boundary $\sigma\,(W'\cdot x+b)$

1.9 Gradient Descent in Logistic Regression

Gradient ∇E is given by $\left[\frac{\partial E(W,b)}{\partial w_1} \quad \frac{\partial E(W,b)}{\partial w_2} \quad \dots \quad \frac{\partial E(W,b)}{\partial w_n} \quad \frac{\partial E(W,b)}{\partial b}\right]$

At each step, Gradient Descent takes a step by magnitude of $learn\_rate\ \alpha * \nabla E$ at the direction of $-\nabla E$, such that Error Function E(W, b) decreases.

After each step,

$$w_i^{new} = w_i + \alpha * \left(-\frac{\partial E(W,b)}{\partial w_1}\right) \quad for\ i = 1, \dots, n$$

$$b^{new} = b + \alpha * \left(-\frac{\partial E(W,b)}{\partial b}\right)$$

$$\hat{y}^{new} = \sigma\,(W^{new}\cdot x + b^{new})$$

Gradient of sigmoid function:

$$\begin{aligned}\sigma'(x) &= \frac{\partial}{\partial x}\frac{1}{1+e^{-x}}\\ &= \frac{-1}{(1+e^{-x})^2} * (1+e^{-x})'\\ &= \frac{e^{-x}}{(1+e^{-x})^2}\\ &= \frac{1}{1+e^{-x}} * \frac{e^{-x}}{1+e^{-x}}\\ &= \sigma(x)\,(1-\sigma(x))\end{aligned}$$

Gradient Desent ∇E_x at each point $x = [x^{(1)}, x^{(2)}, \dots, x^{(m)}]$:

$$\nabla E_x = [\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots, \frac{\partial E}{\partial w_n}, \frac{\partial E}{\partial b}]$$

We already know the error at the point x is:

$$Error\ E_x\ (on\ instance\ x) = -y\log(\hat{y}) - (1-y)\,(\log(1-\hat{y}))$$

$$where\ \hat{y} = \sigma\,(W\cdot x + b)$$

Then, each term in ∇E_x is :

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_j}$$

This is the error with respect to each weight at instance x.

$$\begin{aligned}
\frac{\partial}{\partial w_j}\hat{y} &= \frac{\partial}{\partial w_j}\sigma(Wx+b) \\
&= \sigma(Wx+b)(1-\sigma(Wx+b)) \cdot \frac{\partial}{\partial w_j}(Wx+b) \\
&= \hat{y}(1-\hat{y}) \cdot \frac{\partial}{\partial w_j}(Wx+b) \\
&= \hat{y}(1-\hat{y}) \cdot \frac{\partial}{\partial w_j}(w_1x_1 + \cdots + w_jx_j + \cdots + w_nx_n + b) \\
&= \hat{y}(1-\hat{y}) \cdot x_j
\end{aligned}$$

(image source from Udacity)

For each weight error at x:

$$\begin{aligned}
\frac{\partial}{\partial w_j}E &= \frac{\partial}{\partial w_j}[-y\log(\hat{y}) - (1-y)\log(1-\hat{y})] \\
&= -y\frac{\partial}{\partial w_j}\log(\hat{y}) - (1-y)\frac{\partial}{\partial w_j}\log(1-\hat{y}) \\
&= -y \cdot \frac{1}{\hat{y}} \cdot \frac{\partial}{\partial w_j}\hat{y} - (1-y) \cdot \frac{1}{1-\hat{y}} \cdot \frac{\partial}{\partial w_j}(1-\hat{y}) \\
&= -y \cdot \frac{1}{\hat{y}} \cdot \hat{y}(1-\hat{y})x_j - (1-y) \cdot \frac{1}{1-\hat{y}} \cdot (-1)\hat{y}(1-\hat{y})x_j \\
&= -y(1-\hat{y}) \cdot x_j + (1-y)\hat{y} \cdot x_j \\
&= -(y-\hat{y})x_j
\end{aligned}$$

(image source from Udacity)

For bias error at x:

$$\frac{\partial}{\partial b}E = -(y-\hat{y})$$

(image source from Udacity)

Gradient Desent ∇E_x at each point $x = [x^{(1)}, x^{(2)}, \dots, x^{(m)}]$:

$$\nabla E_x = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots, \frac{\partial E}{\partial w_n}, \frac{\partial E}{\partial b}\right]$$

$$= \left[-(y - \hat{y})\, x^{(1)}, -(y - \hat{y}) x^{(2)}, \dots, -(y - \hat{y}) x^{(m)}, -(y - \hat{y})\right]$$

Gradient at x ∇E_x relates to:

a. Instance x's coordinate in feature space
b. Difference between true label y and model's prediction $\hat{y}$

Conclusion:

a. Closer the y and $\hat{y}$, smaller ∇E_x
b. Farther the y and $\hat{y}$, larger ∇E_x

Combining with Perceptron Algorithm in <1.4>, after each gradient descent step at each point, update every weight:

$$w_i' = w_i - \alpha\,[-(y - \hat{y})\, x_i] = w_i + \alpha\,(y - \hat{y})\, x_i$$

$$b' = b - \alpha\,[-(y - \hat{y})\,] = b + \alpha\,(y - \hat{y})$$

Gradient Descent Algorithm

Start with random weights W = [w_1, w_2, ..., w_n, b], it gives an initial boundary as $\sigma(Wx + b)$

For every point x = [x_1, x_2, ..., x_n]:
    // update all weights W
    For i = 1, ..., n:
        // update each weight in W
        $w_i' = w_i - \alpha \frac{\partial E}{\partial w_i} = w_i + \alpha\,(y - \hat{y})\, x_i$
        $b' = b - \alpha \frac{\partial E}{\partial b} = b + \alpha\,(y - \hat{y})$

Repeat until error (Error E_x (on instance x) $= -y \log(\hat{y}) - (1 - y)\,(\log(1 - \hat{y}))$) is small or repeat for a certain number of epoches.

Gradient Descent Algorithm vs. Perceptron Algorithm

Gradient Descent Algorithm	Perceptron Algorithm

$w_i' = w_i + \alpha\,(y - \hat{y})\,x_i$
 	
$b' = b + \alpha\,(y - \hat{y})$
	For every misclassified point x (x_1, ..., x_n)
	  if true_label(x) == pos
	  for i = 1, 2, ..., n
	  w_i += α * x_i
	  b += α
	  else if true_label(x) == neg
	  for i = 1, 2, ..., n
	  w_i -= α * x_i
	  b -= α

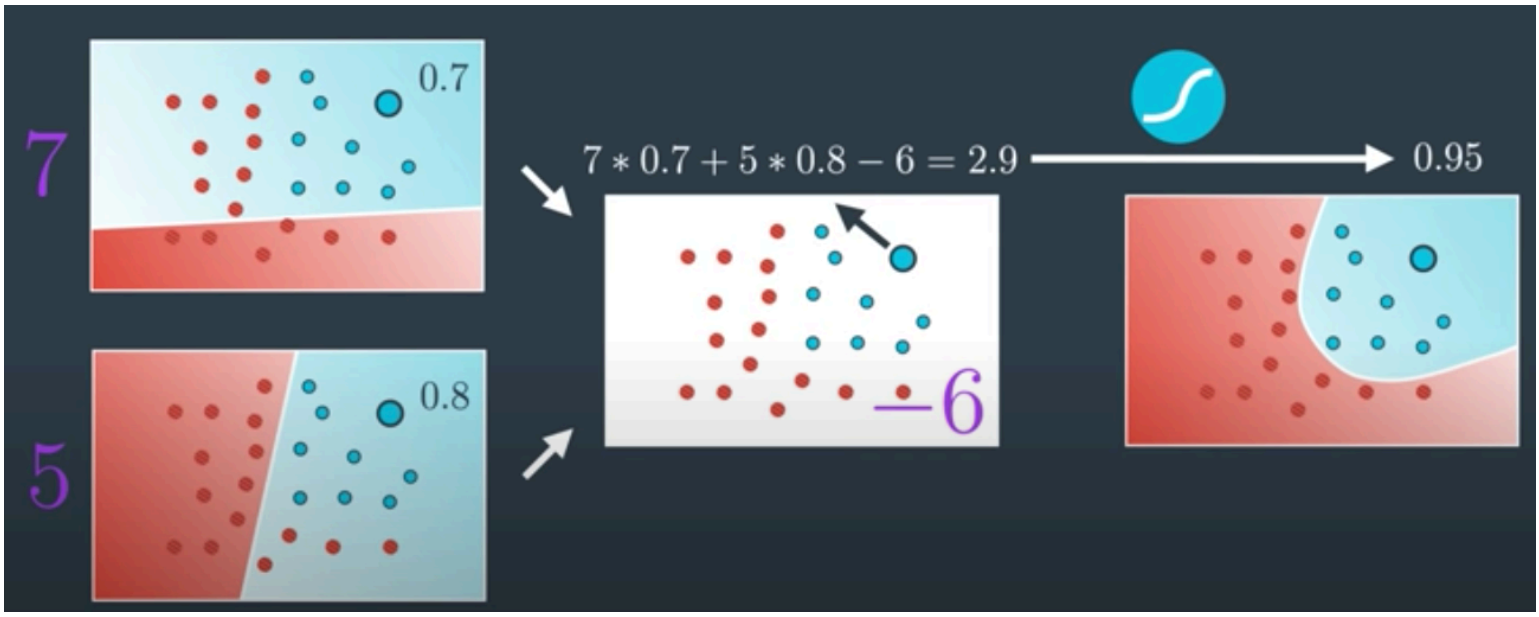
$y, \hat{y}$ cound be any number (probability). Misclassified points ask boundary to come closer; Correctly classified points ask boundary to go farther.	In perceptron algorithm, label and prediction $\in$ {0, 1}, so if x is correctly classified, $y - \hat{y} = 0$. If misclassified, $y - \hat{y} = \begin{cases} 1 - 0 = 1, & true\_label(x)\ is\ positive \\ 0 - 1 = -1, & true\_label\ (x)\ is\ negative \end{cases}$ This is same as Gradient Descent Algorithm except that $y, \hat{y} \in \{0, 1\}$ only. ONLY Misclassified points ask boundary to come closer;

1.10 Non-linear Neural Networks

When the data is not separable by a line, the data is non-linear.

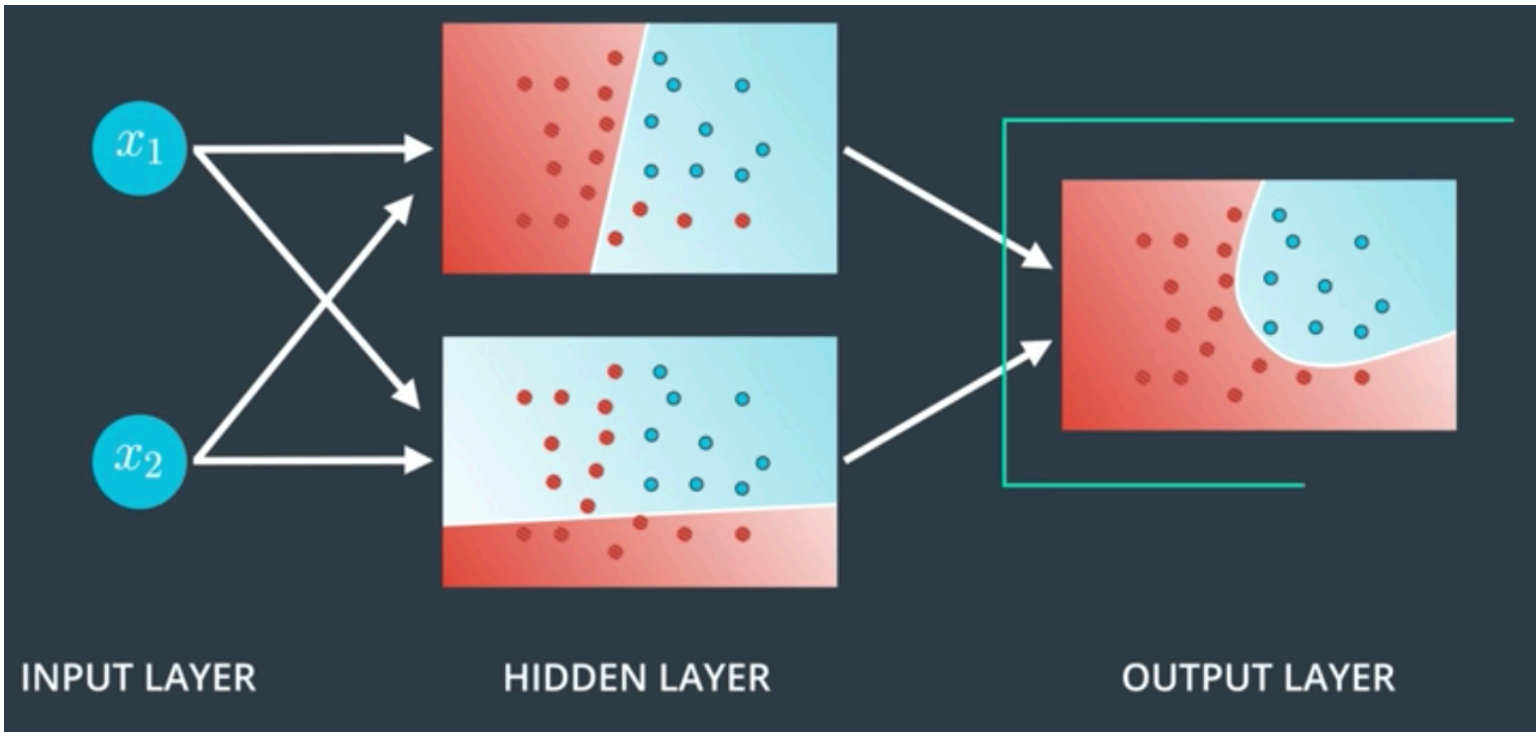
We could combine 2 linear perceptron into a non-linear perceptron:

a. Each linear perceptron has a weight
b. For each point in feature space, we take the linear combination of the 2 probabilities in 2 linear perceptron
c. Then take sigmoid(linear combination + bias) as the new probability in non-linear perceptron



(image source from Udacity)

1.11 Multi-layer Perceptron



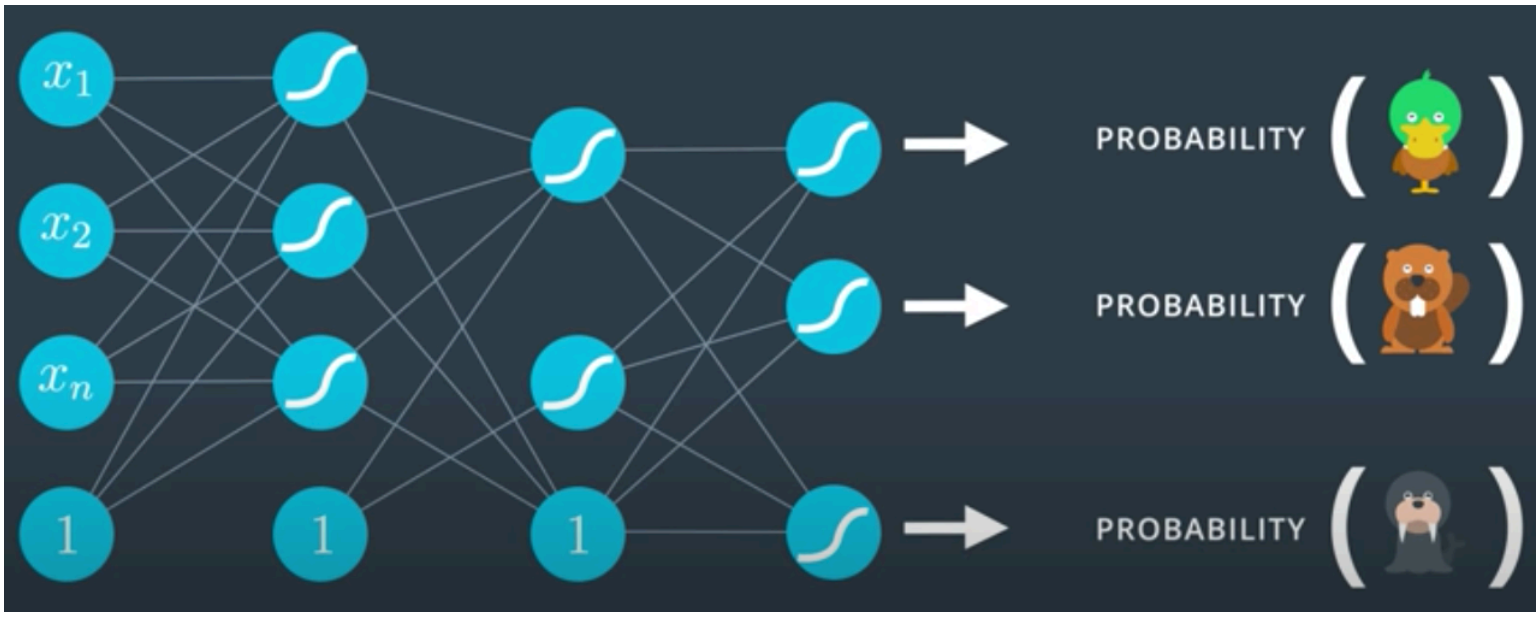
(image source from Udacity)

When there are more # of hidden layers, it is called "**deep neural network**".

The boundary will be more and more complex.

1.12 Multi-class classification

Instead of training multiple deep neural networks for each class, we train a large neural network where # of neuron in output layer = # of classes



(image source from Udacity)

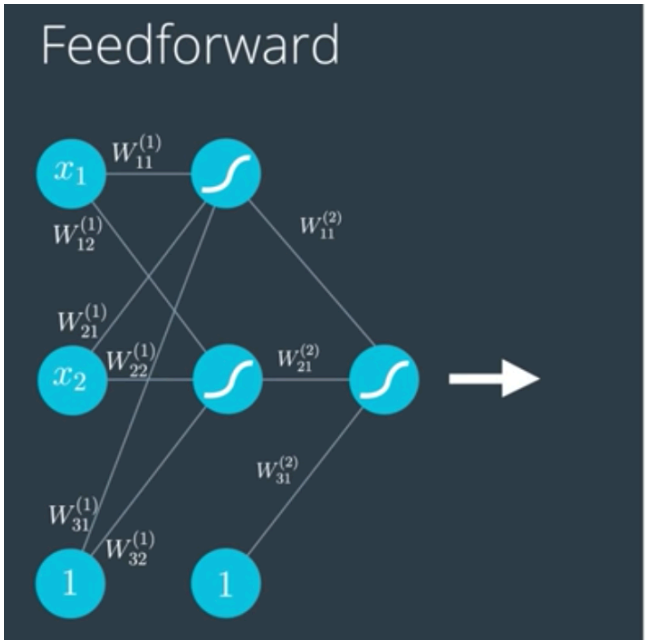
1.13 Feed-forwarding & Back-propagation

Feed-forwarding

Feed-forward is the process of turning input to prediction through all layers.

$$w_{ij}^{(l)} \text{ indicates the weight from } i^{th} \text{ neuron in layer } l \text{ to } j^{th} \text{ neuron in layer } l+1$$

$$W^{(l)} \text{ indicates weights between layer } l \text{ and layer } l+1$$



$$\hat{y} = \sigma \begin{pmatrix} W_{11}^{(2)} \\ W_{21}^{(2)} \\ W_{31}^{(2)} \end{pmatrix} \sigma \begin{pmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

$$\hat{y} = \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)}(x)$$

(image source from Udacity)

Term	***Explanation***
$W^{(1)} = \begin{pmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} \end{pmatrix}$	weights between layer 1 and layer 2
$x = \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$	input data
$W^{(1)} \cdot x$	weighted sum in layer 2
$\sigma\left(W^{(1)}\right)(x)$	output in layer 2, or input to layer 3
$W^{(2)} = \begin{pmatrix} W_{11}^{(2)} \\ W_{21}^{(2)} \\ W_{31}^{(2)} \end{pmatrix}$	weights between layer 2 and layer 3
$W^{(2)} \cdot \sigma\left(W^{(1)}\right)(x)$	weighted sum in layer 3
$\sigma\left(W^{(1)}\right) \cdot \sigma\left(W^{(1)}\right)(x)$	output in layer 3, or prediction $\hat{y}$, or output

Error Function

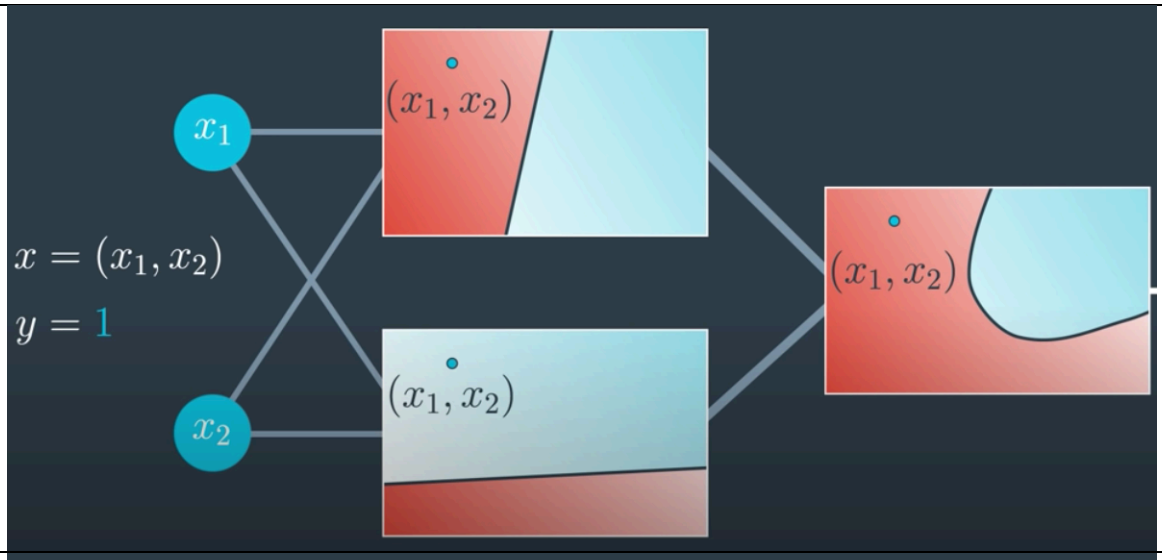
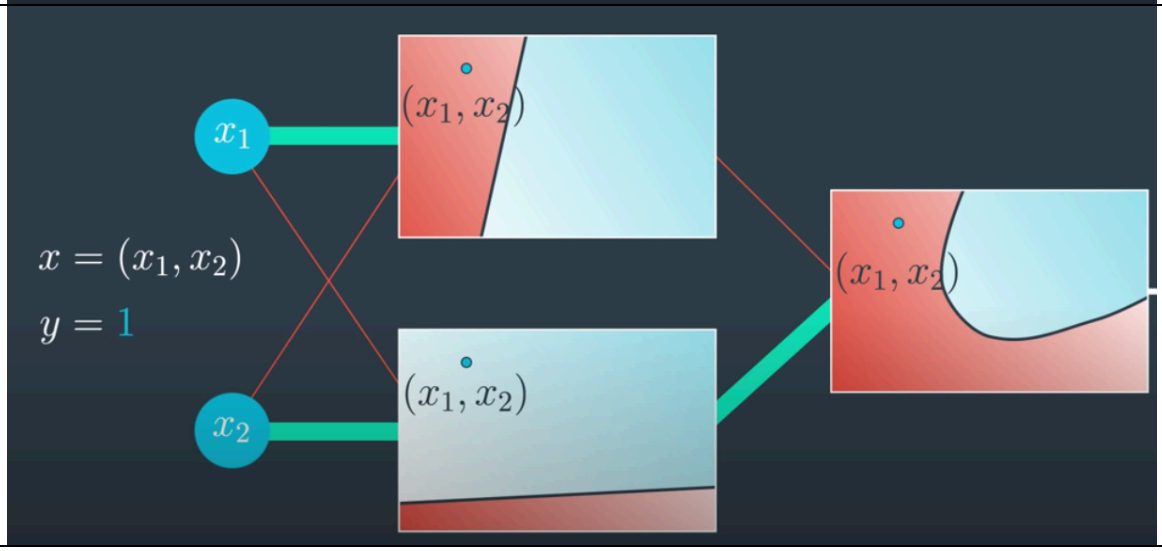
$$E(W) = -\frac{1}{m}\sum_{i=1}^{m}\{\, y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\,\}$$

$$where\ \hat{y} = \sigma \circ W^{(L-1)} \circ \sigma \circ W^{(L-2)} \circ \sigma \circ \ldots \circ \sigma \circ W^{(1)}\, x, and$$

$$W = [W^1 \quad W^2 \quad \ldots \quad W^{L-1}]$$

Back-propagation

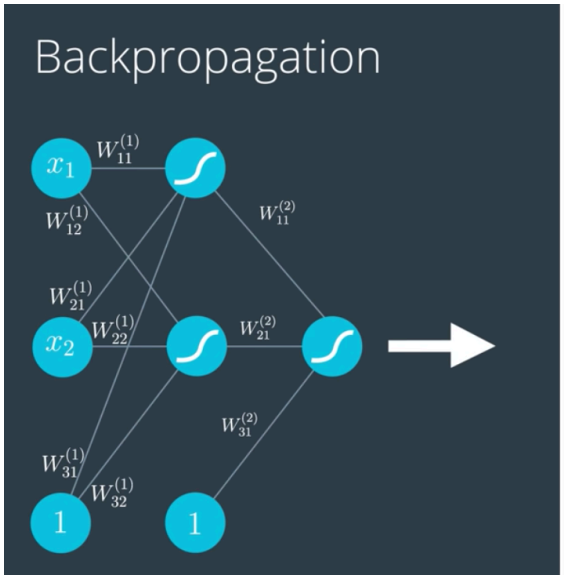
Back-propagation is the process of spreading the error ($y - \hat{y}$) in output layer backwards to each of the weights in each forward layers, and update the weights to minimize the error.

Before Back-propagation	
After Back-propagation	

(image source from Udacity)

Green edge → increase the weight **Red edge** → decrease the edge

Back-propagation in multi-layer perceptron:



$$\hat{y} = \sigma W^{(2)} \circ \sigma \circ W^{(1)}(x)$$

$$W^{(1)} = \begin{pmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} \end{pmatrix} \quad W^{(2)} = \begin{pmatrix} W_{11}^{(2)} \\ W_{21}^{(2)} \\ W_{31}^{(2)} \end{pmatrix}$$

$$\nabla E = \begin{pmatrix} \frac{\partial E}{\partial W_{11}^{(1)}} & \frac{\partial E}{\partial W_{12}^{(1)}} & \frac{\partial E}{\partial W_{11}^{(2)}} \\ \frac{\partial E}{\partial W_{21}^{(1)}} & \frac{\partial E}{\partial W_{22}^{(1)}} & \frac{\partial E}{\partial W_{21}^{(2)}} \\ \frac{\partial E}{\partial W_{31}^{(1)}} & \frac{\partial E}{\partial W_{32}^{(1)}} & \frac{\partial E}{\partial W_{31}^{(2)}} \end{pmatrix}$$

$$W_{ij}'^{(k)} \leftarrow W_{ij}^{(k)} - \alpha \frac{\partial E}{\partial W_{ij}^{(k)}}$$

(image source from Udacity)

Chain Rule

$$A = f(x)$$
$$B = g(f(x))$$

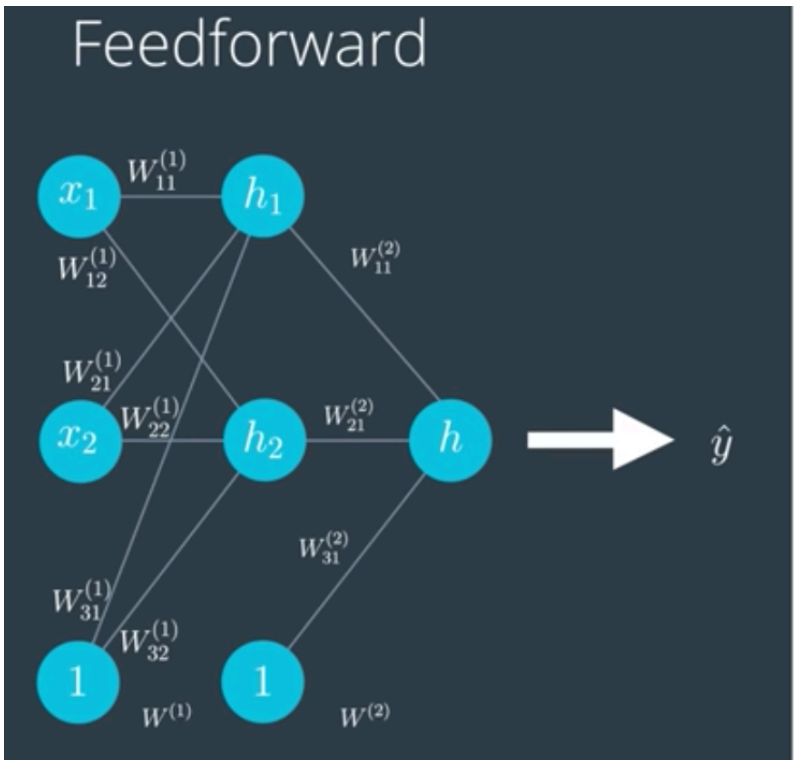
Then we have

$$\frac{\partial B}{\partial x} = \frac{\partial B}{\partial A} \cdot \frac{\partial A}{\partial x}$$

Feed-forwarding is composing functions; Back-propagation is taking the derivative w.r.t. each weight, which is same as multiplying the partial derivatives.

Summary

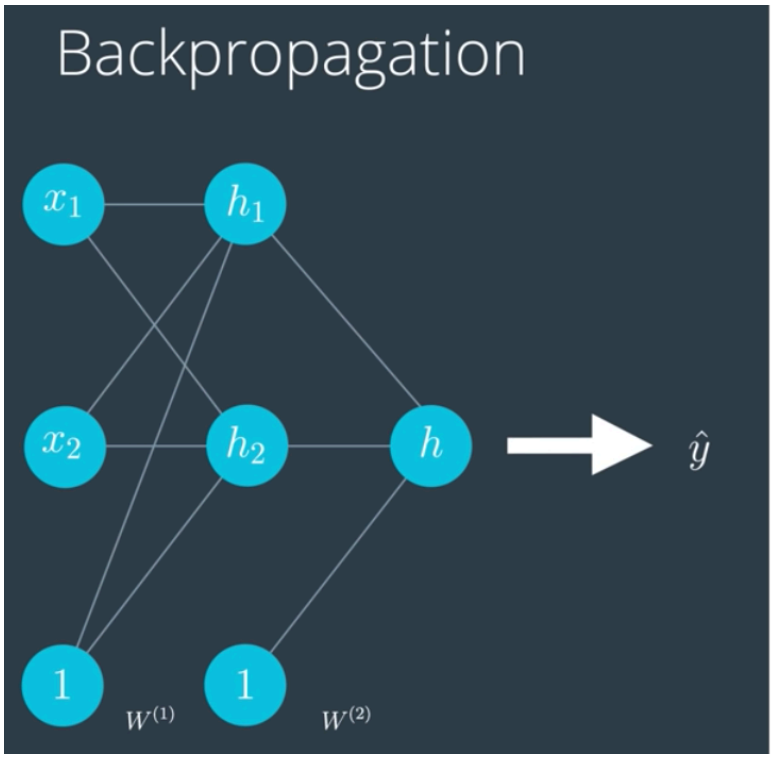
Feed-forward



$$h_1 = W_{11}^{(1)} x_1 + W_{21}^{(1)} x_2 + W_{31}^{(1)}$$
$$h_2 = W_{12}^{(1)} x_1 + W_{22}^{(1)} x_2 + W_{32}^{(1)}$$
$$h = W_{11}^{(2)} \sigma(h_1) + W_{21}^{(2)} \sigma(h_2) + W_{31}^{(2)}$$
$$\hat{y} = \sigma(h)$$
$$\hat{y} = \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)}(x)$$

(image source from Udacity)

Back-propagate



$$E(W) = -\frac{1}{m}\sum_{i=1}^{m} y_i ln(\hat{y}_i) + (1-y_i) ln(1-\hat{y}_i)$$

$$E(W) = E(W_{11}^{(1)}, W_{12}^{(1)}, ..., W_{31}^{(2)})$$

$$\bigtriangledown E = (\frac{\partial E}{\partial W_{11}^{(1)}}, ..., \frac{\partial E}{\partial W_{31}^{(2)}})$$

$$\frac{\partial E}{\partial W_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial h_1} \frac{\partial h_1}{\partial W_{11}^{(1)}}$$

(image source from Udacity)

We apply the chain rule in $\frac{\partial E}{\partial W_{11}^{(1)}}$

Since

$$h = W_{11}^{(2)}\, \sigma(h_1) + W_{21}^{(2)}\, \sigma(h_2) + W_{31}^{(2)}$$

Then, only $W_{11}^{(2)}\, \sigma(h_1)$ has relation with $\frac{\partial h}{\partial h_1}$

$$\frac{\partial h}{\partial h_1} = \frac{\partial}{\partial h_1}(W_{11}^{(2)}\, \sigma(h_1)) = \; W_{11}^{(2)} \cdot \frac{\partial}{\partial h_1}\sigma(h_1) = \; W_{11}^{(2)}\, \sigma(h_1)\, [1 - (h_1)\,]$$

1.14 **More notes: see *../ notebooks (filled) / 2. Neural Networks / 1. Introduction to Neural Networks.ipynb***

Lesson 2: Implementing Gradient Descent

2.1 Gradient Descent with Mean Squared Errors Function

In <1.8 Logistic Regression>, we introduced Log-loss Error Function $E_{log\text{-}loss}$

$$Log - loss\ Error\ Function\ (on\ all\ instances) = -\frac{1}{m}\sum_{i=1}^{m}\{\,(1-y_i)\log(1-\hat{y}_i) + y_i\log(\hat{y}_i)\,\}$$

Here, we introduce Mean Squared Error Function E_{MSE}

$$Mean\ Squared\ Error\ Function\ (on\ all\ instances) = \frac{1}{2}\sum_{\mu}\sum_{j}\left[y_j^{\mu} - \hat{y}_j^{\mu}\right]^2$$

$$where\ j\ indicates\ output\ unit\ and\ \mu\ indicates\ data\ points, and$$

$$\hat{y}_j{}^{\mu} = activation\_function\left(\sum_i w_{ij}\, x_i^{\mu}\right)$$

a. Inside sum over j : find the sum of squared differences between true label y and prediction $\hat{y}$ over all units in the output layer.
b. Outside sum over μ : sum over all the data points

During Gradient Descent, we want to update weights that minimize E_{MSE}

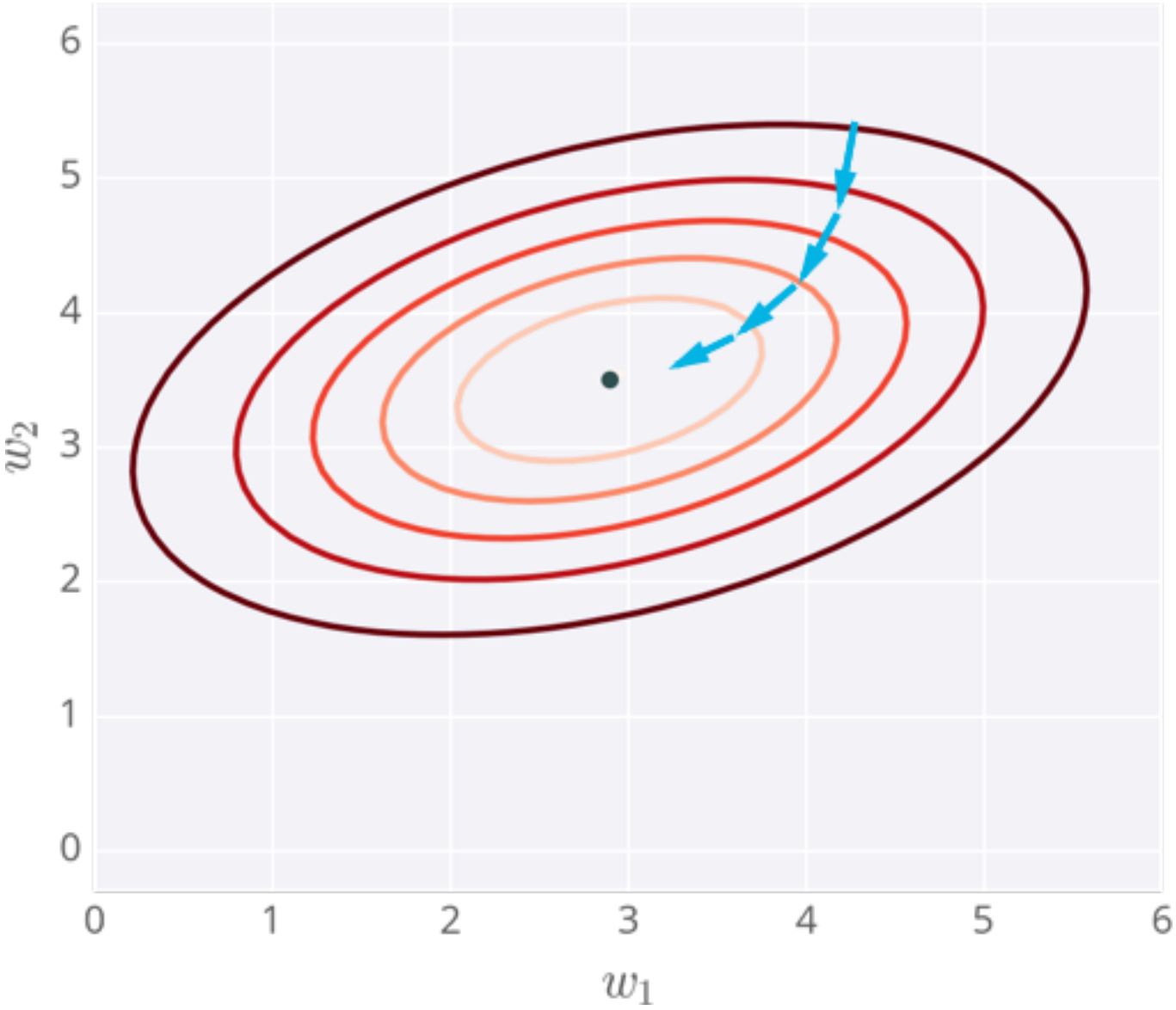
Deep understanding with "Gradient":

a. Partial derivatives
b. Rate of change / slope

(Reference: Khan Academy

https://classroom.udacity.com/nanodegrees/nd101/parts/94643112-2cab-46f8-a5be-1b6e4fa7a211/modules/89a1ec1d-4c22-4a77-b230-b0da99240c89/lessons/07f472eb-0210-446f-8ec2-d297b06c86d0/concepts/7d480208-0453-4457-97c3-56c720c23a89)

Gradient contour plot w.r.t. 2 inputs:



Darker contour lines: larger errors

Brighter coutour lines: smaller errors

Caveats: the dark point in the contour graph may represent a local minima, not global minima

Another technique: use **momentum** can avoir local minima

(see https://distill.pub/2017/momentum/)

Sum of Squared Errors (SSE)

$$SSE = \frac{1}{2}\sum_{\mu}(y^{\mu} - \hat{y}^{\mu})^2$$

$$where\ \hat{y}^{\mu} = f\left(\sum_i w_i\, x_i^{\mu}\right)\ \ f\ is\ activation\ function, and$$

$$\mu\ represents\ each\ data\ point, and\ i\ represents\ each\ feature\ in\ 1\ data\ point$$

SSE is the error if neural network only has 1 output unit

Gradient Descent with MSE (recap)

SSE on the entire dataset:

$$SSE = \frac{1}{2}\sum_{\mu}(y^{\mu} - \hat{y}^{\mu})^2 = \frac{1}{2}\sum_{\mu}\left(y^{\mu} - f(\sum_i w_i\, x_i^{\mu})\right)^2$$

MSE on the entire dataset:

$$MSE = \frac{1}{2m}\sum_{\mu}(y^{\mu} - \hat{y}^{\mu})^2 = \frac{1}{2}\sum_{\mu}\left(y^{\mu} - f(\sum_i w_i\, x_i^{\mu})\right)^2\ \ (where\ m\ is\ size\ of\ dataset)$$

Direction that minimize the MSE Error Function: opposite to the gradient, the slope

$$-graadient$$

Δw signifies the change on the weight:

$$\Delta w_i\ \propto\ -\frac{\partial E}{\partial w_i}\ (negative\ of\ gradient)$$

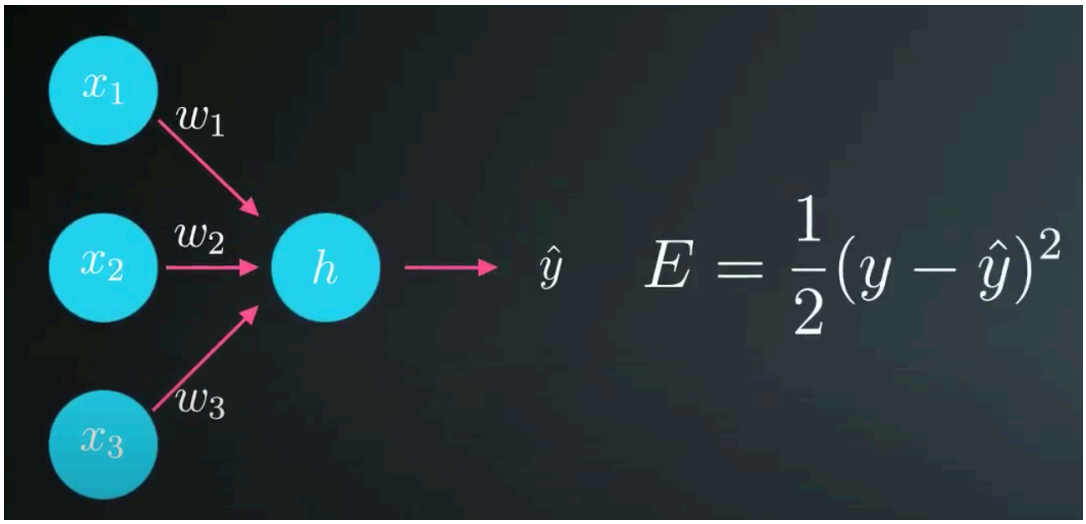
$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i}\ \ \eta\ is\ learning\ rate$$

$\frac{\partial E}{\partial w_i}$ requires knowledge of **multivaiable calculus** (refresher: Khan Academy https://www.khanacademy.org/math/multivariable-calculus)

Chain Rule refresher:

$$\frac{\partial}{\partial z}\, p(\, q(z)\,) = \frac{\partial p}{\partial q}\,\frac{\partial q}{\partial z}$$

Assuming we only have 1 output unit:



(image source from Udacity)

$$gradient = \frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\left(\frac{1}{2}\,(y - \hat{y}(w_i))^2\right)$$

Apply the Chain Rule here: $q = y - \hat{y} \quad p = \frac{1}{2}\,q(w_i)^2$

$$\frac{\partial}{\partial w_i}\left(\frac{1}{2}\,(y - \hat{y}(w_i))^2\right) = \frac{\partial p}{\partial q}\frac{\partial q}{\partial w_i}\left(\frac{1}{2}\,(y - \hat{y}(w_i))^2\right) = (y - \hat{y})\frac{\partial}{\partial w_i}(y - \hat{y}) = -(y - \hat{y})\frac{\partial \hat{y}}{\partial w_i}$$

$$where\ \hat{y} = f(h) = f(\sum_i w_i\, x_i)$$

So

$$\frac{\partial E}{\partial w_i} = -(y - \hat{y})\frac{\partial \hat{y}}{\partial w_i} = -(y - \hat{y})\, f'(h)\, \frac{\partial}{\partial w_i}\sum_i w_i\, x_i = -(y - \hat{y})\, f'(h)\, x_i$$

$$where\ (y - \hat{y})\ indicates\ output\ error, and$$

$$f'(h)\ indicates\ derivative, and$$

$$h\ indicates\ weighted\ sum\ \sum_i w_i\, x_i\,, and$$

$$x_i\ indicates\ i^{th}\ feature\ input$$

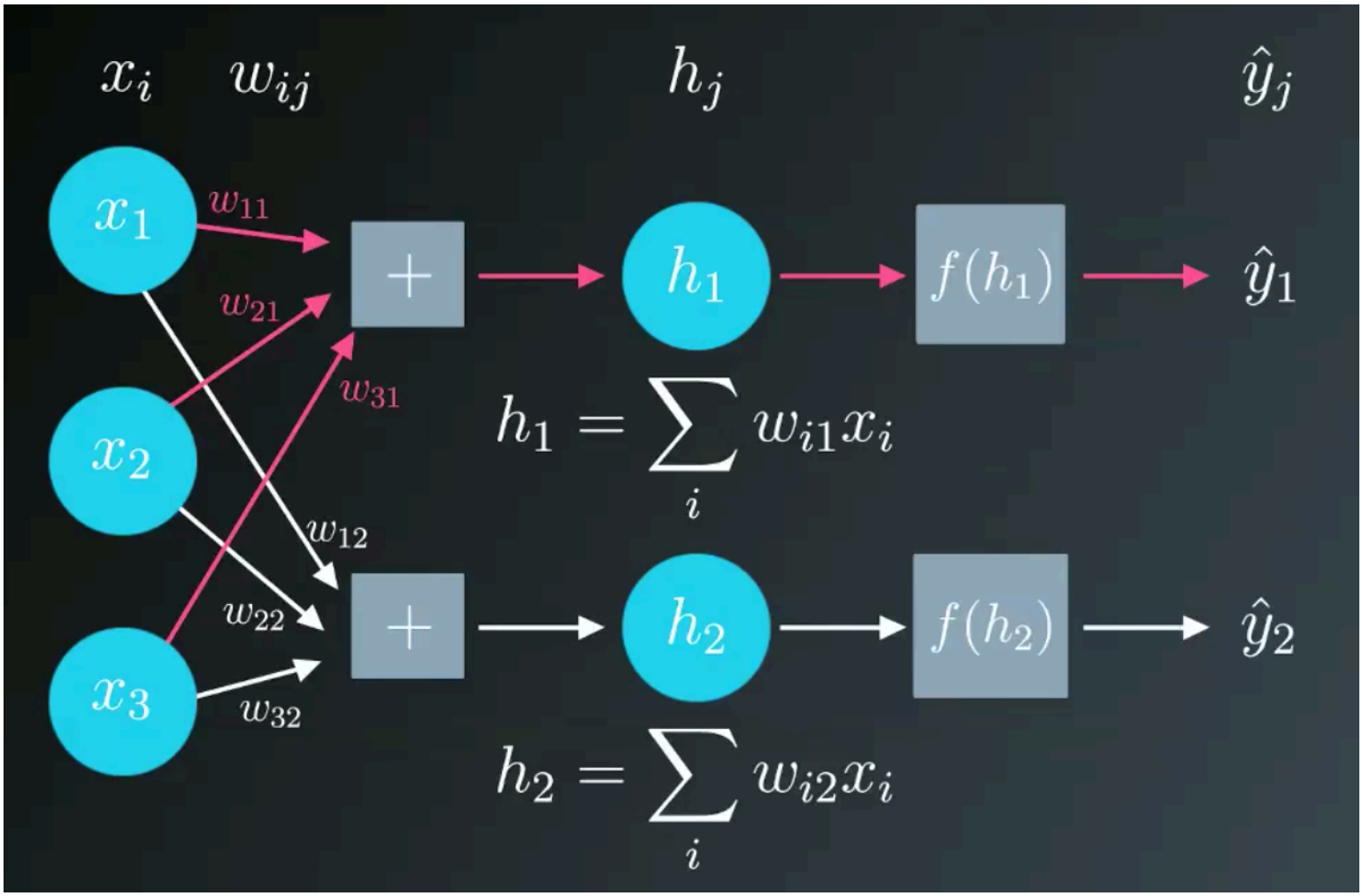
Let's introduce another **error term $\boldsymbol{\delta = (y - \hat{y})\, f'(h)}$**

$$\frac{\partial E}{\partial w_i} = -\,\delta\, x_i$$

Update the weight:

$$w_i' = w_i + \Delta w = w_i - \eta\frac{\partial E}{\partial w_i} = w_i + \eta\,\delta\, x_i$$

In multiple output units:



(image source from Udacity)

$\delta_j = (y_j - \hat{y}_j) f'(h_j)$ *where j denotes each output units*

$\Delta w_{ij} = \eta \, \delta_j \, x_i$

Pseudo-code of Gradient Descent with updating the weights:

Set the weight step to zero: $\Delta w_i = 0$
For each record in the training data:
    Make a forward pass through the network, calculating $\hat{y} = f(\sum_i w_i x_i)$
    Calculate the error term for the output unit $\delta = (y - y') * f'(\sum_i w_i x_i)$
    Update the weight step $\Delta w_i = \Delta w_i + \delta x_i$
Update the weights $w_i = w_i + \eta \frac{\Delta w_i}{m}$ $(m = \#\ of\ records, \eta = learning\ rate)$

Repeat the processes above for e epochs

Initialize the weights

1) Initial weights should be random in order to break symmetry
2) Initial weights should be small but centered at 0 so that the sigmoid(weights) is near 0
3) A good value for scale of weights is $\frac{1}{\sqrt{n}}$ $where\ n\ is\ \#\ of\ input\ neurons$
4) An implementation code:

```
weights = np.random.normal(
    loc=0.0,
    scale=1/input_nodes**.5,
    size=(input_nodes, next_layer_nodes))

np.random.normal:
https://docs.scipy.org/doc/numpy-
1.14.1/reference/generated/numpy.random.normal.html
```

2.2 Multi-layer Perceptrons

Refresher

a. Vectors: https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/vectors/v/vector-introduction-linear-algebra
b. Matrices: https://www.khanacademy.org/math/precalculus-2018/precalc-matrices

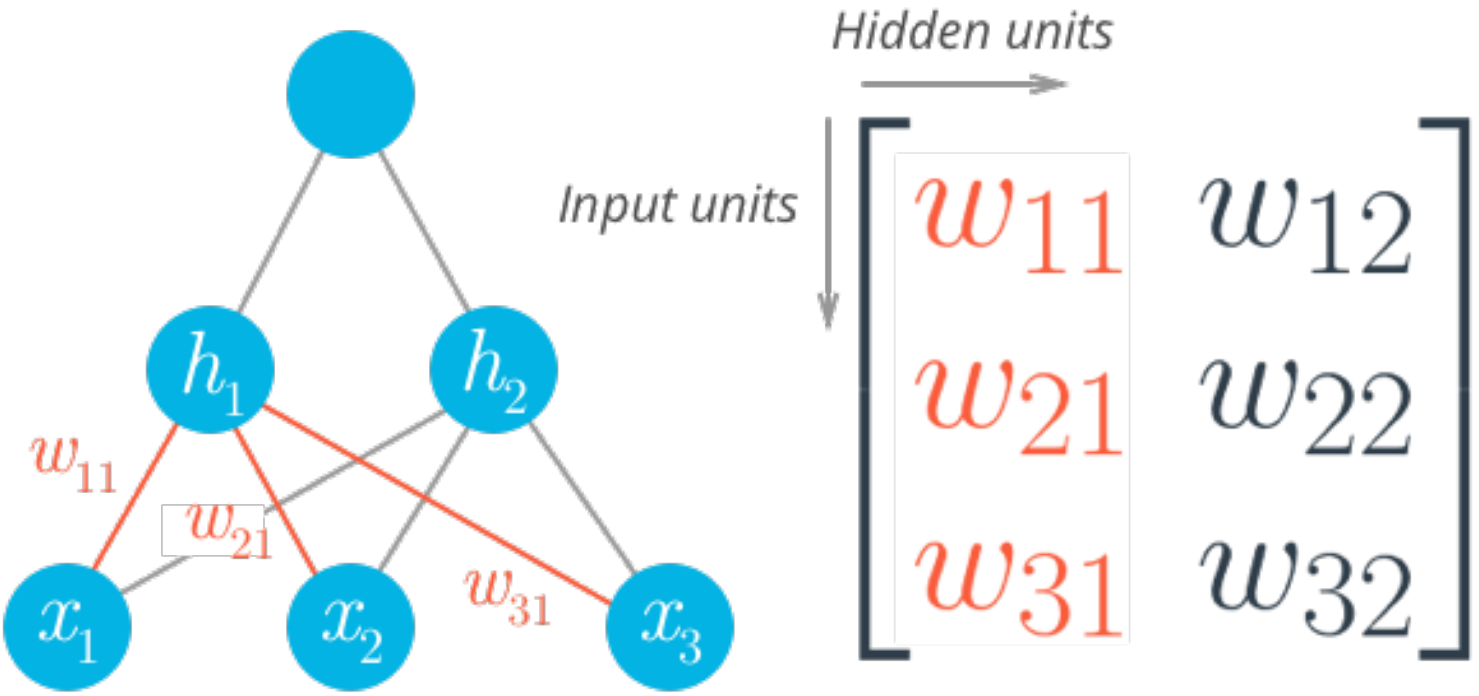
Multi-layer sketch

$$W = \left[W^{(1)}, W^{(2)}, \ldots, W^{(L-1)}\right]\ (assuming\ L\ layers\ in\ total)$$

$$W^{(l)}\ indicates\ weights\ between\ layer\ l\ and\ layer\ l+1$$

$$W^{(l)} == \begin{bmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1q}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \cdots & w_{2q}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{p1}^{(l)} & w_{p2}^{(l)} & \cdots & w_{pq}^{(l)} \end{bmatrix}\ (assuming\ layer\ l\ has\ p\ neurons, layer\ l+1\ has\ q\ neuron)$$

$$w_{ij}^{(l)}\ indicates\ the\ weight\ from\ i^{th}\ neuron\ in\ layer\ l\ to\ j^{th}\ neuron\ in\ layer\ l+1$$



(image source from Udacity)

Weighted sum at hidden layer j :

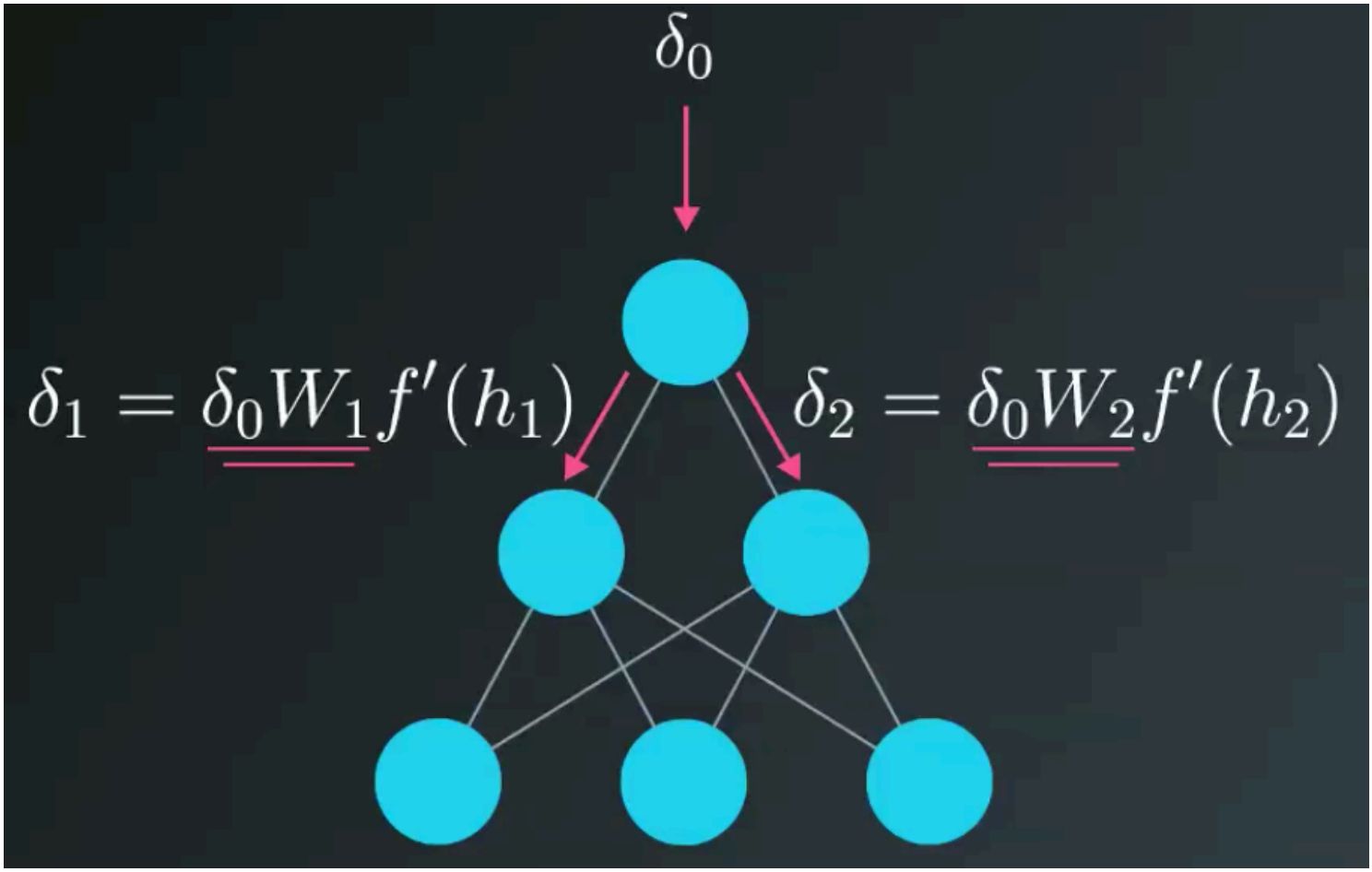
$$h_j = np.dot\left(A^{(l-1)}, W^{j-1}\right) = a_{1j}^{(l-1)} w_{1j}^{(l-1)} + a_{2j}^{(l-1)} w_{2j}^{(l-1)} + \cdots$$

$$where\ A^{(l-1)} = f\left(h_{j-1}\right)$$

Back-propagation in Multi-layer Perceptrons

Error for units is proportional to the error in the output layer * the weight between the units.

So we propagate the error backwards:



(image source from Udacity)

$\delta_0 W_1, \delta_2 W_2$ *is same as* $feedforwarding$*, but this time,* $activation$ *is replaced by the* $error$

Error at unit j in layer h:

$$\delta_j^h = \sum W_{jk}\, \delta_k^o\, f'(h_j)$$

$where\ W_{jk}\ is\ the\ weight\ between\ j^{th}\ unit\ in\ layer\ h\ and\ k^{th}\ unit\ in\ layer\ o\ (o = h + 1)$

Gradient descent weight step between the i-th input unit and j-th unit in hidden layer:

$$\Delta\, w_{ij} = \eta\, \delta_j^h\, x_i$$

$where\ \eta\ is\ learning\ rate, \delta_j^h\ is\ the\ error\ at\ j^{th}\ unit\ in\ layer\ h, x_i\ is\ the\ input\ values$

An example: an instance x = [0.1, 0.3], true label y = 1, learn_rate = 0.5

Output layer o

Hidden layer h

Input layer

(image source from Udacity)

Weighted sum at layer h:

$$h^{(h)} = \sum_i w_i x_i = 0.1 * 0.4 + 0.3 * (-0.2) = -0.02$$

Activation at layer h:

$$a^{(h)} = \sigma\left(h^{(h)}\right) = \frac{1}{1 + e^{-(-0.02)}} = 0.495 \ (this\ is\ \boldsymbol{output\ in\ layer\ h}, also\ \boldsymbol{input\ to\ layer\ o})$$

Weighted sum at layer o:

$$h^{(o)} = 0.1 * 0.495 = 0.0495$$

Activation at layer o:

$$a^{(o)} = \sigma\left(h^{(o)}\right) = \frac{1}{1 + e^{-0.0495}} = 0.512 = \hat{y}$$

Error term at output layer:

$$\delta^o = (y - \hat{y})\,\sigma'\left(h^{(o)}\right) = (y - \hat{y})\,\sigma\left(h^{(o)}\right) * \left(1 - \sigma\left(h^{(o)}\right)\right)$$
$$= (1 - 0.512) * 0.512 * (1 - 0.512) = 0.122$$

Error term at hidden layer:

$$\delta^h = W\,\delta^o \sigma'\left(h^{(h)}\right) = W\delta^o\,\sigma\left(h^{(h)}\right) * \left(1 - \ \sigma\left(h^{(h)}\right)\right)$$
$$= 0.1 * 0.122 * 0.495 * (1 - 0.495) = 0.003$$

Gradient Descent weight step:

$$\Delta\, W = \ \eta\, \delta^{\,o}\, a^h = \ 0.5 * 0.122 * 0.495 = 0.0302$$
$$\Delta\, w_i = \ \eta\, \delta^h\, x_i = 0.5 * 0.003 * (0.1, 0.3) = (0.00015, 0.00045)$$

Vanishing gradient problem: errors in the late layers vanish quickly while back-propagating

The maximum derivative of sigmoid is $0.5 * (1 - 0.5) = 0.25$. When back-propagating, at least 93.75% of errors is vanished.

Solutions: use different activation function, e.g. ReLu

Recap:

a. Error term at the output layer:
$$\sigma_k = (y_k - \hat{y}_k) f'(a_k)$$
b. Error term at hidden layer:
$$\sigma_j = \sum \left[w_{jk}\sigma_k\right] f'\left(h_j\right)$$

2.3 Further Reading

c. Back-prop: https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b#.vt3ax2kg9
d. Stanford CS231: https://www.youtube.com/watch?v=59Hbtz7XgjM

2.4 **More notes:** ***../ notebooks (filled) / 2. Neural Networks / 4. Implementing Gradient Descent***

Lesson 3: Training Neural Networks

3.1 Training set & Testing set

Train the model ONLY on the training set, and evaluate it on testing set.

ALWAYS choose the simpler model if multiple models perform similar

3.2 Overfitting * Underfitting

Overfitting ~ Killing Godzilla (哥斯拉) with a flyswatter

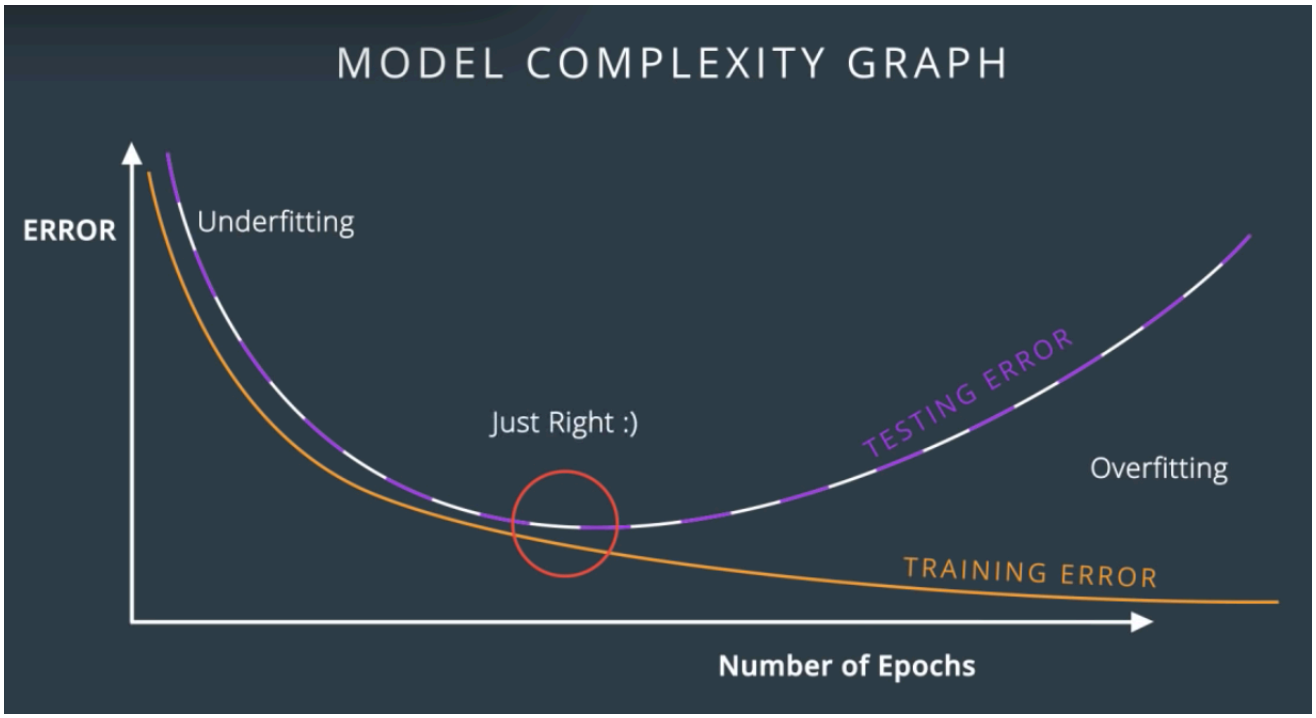
a. High variance, complicate boundary
b. Not work well for generalization
c. Low training error, high testing error
d. **We prefer overfitting models to underfitting models**, then set rules to avoid overfitting

Underfitting ~ Killing a fly with bazooka (火箭炮)

a. High bias, simple boundary
b. Not work well even for normal data
c. High training error, high testing error

3.3 Early Stopping

Model complexity graph



(image source from Udacity)

Early stopping: Do gradient descent until testing error stop decreasing and start to increase.

3.4 Regularization

The weights change the slope of activation function $w_1 * x_1 + w_2 * x_2 + \cdots + b$

BertrAIND Russell: "The whole problem with AI is that bad models are so certain of themselves, and good models so full of doubts."

Regularization: punish for large values in weights

Tune the Error Function:

L1 regularization:

$$ERROR\ FUNCTION = -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)\log(1-\hat{y}_i) + y_i\log(\hat{y}_i) + \lambda(|w_1| + \cdots + |w_n|)$$

L2 regularization:

$$ERROR\ FUNCTION = -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)\log(1-\hat{y}_i) + y_i\log(\hat{y}_i) + \lambda(w_1^2 + \cdots + w_n^2)$$

L1 regularization is good for feature selections, since weights are sparse 1 and 0;

L2 regularization is good for training models, since weights are small and homogeneous;

3.5 Dropout

a. Purpose: avoid overfitting, don't make network so dependent on certain neurons
b. Randomly turn off some neurons during different epochs
c. Each neuron has a **probability** to be turned off

3.6 Local Minima & Solutions

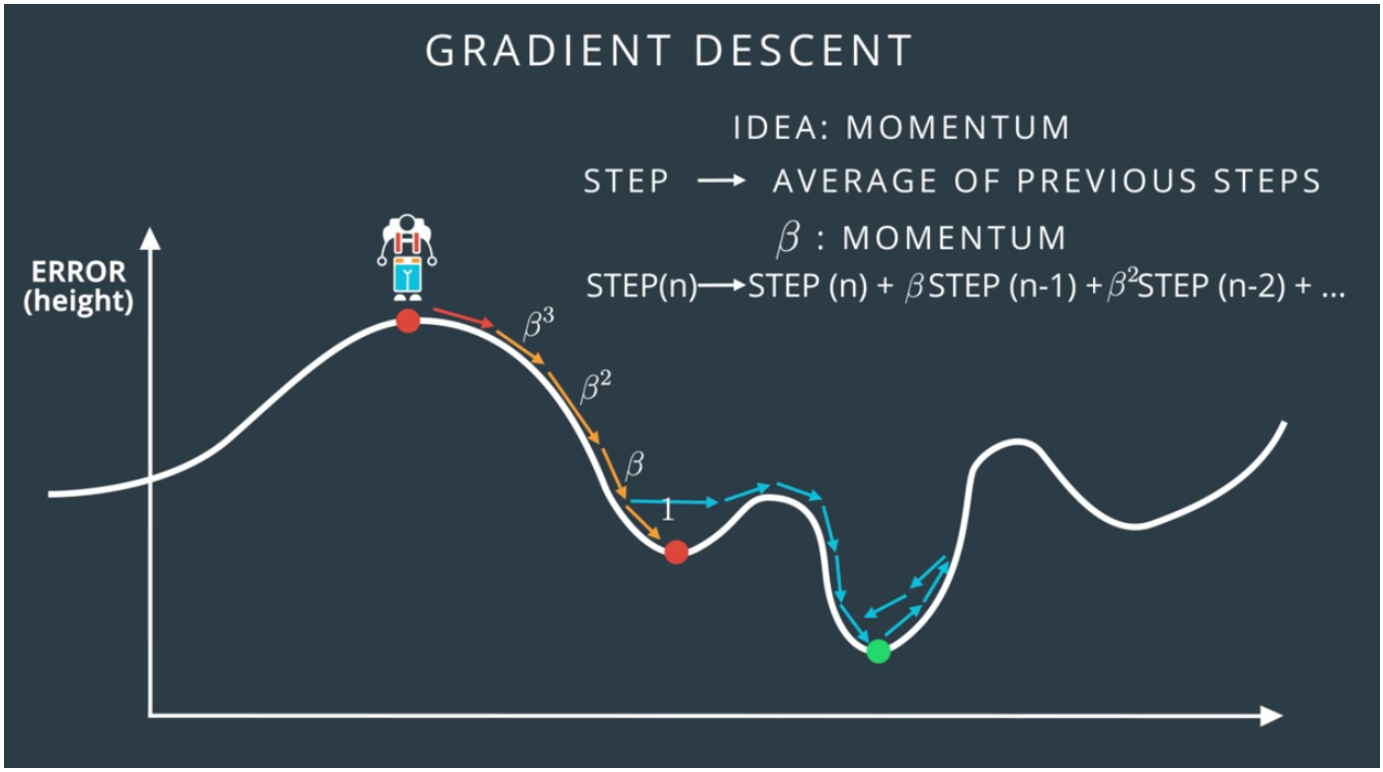
Solution 1: Random start

Start Gradient Descent at different initial states

Solution 2: Momentum $0 \leq \beta \leq 1$

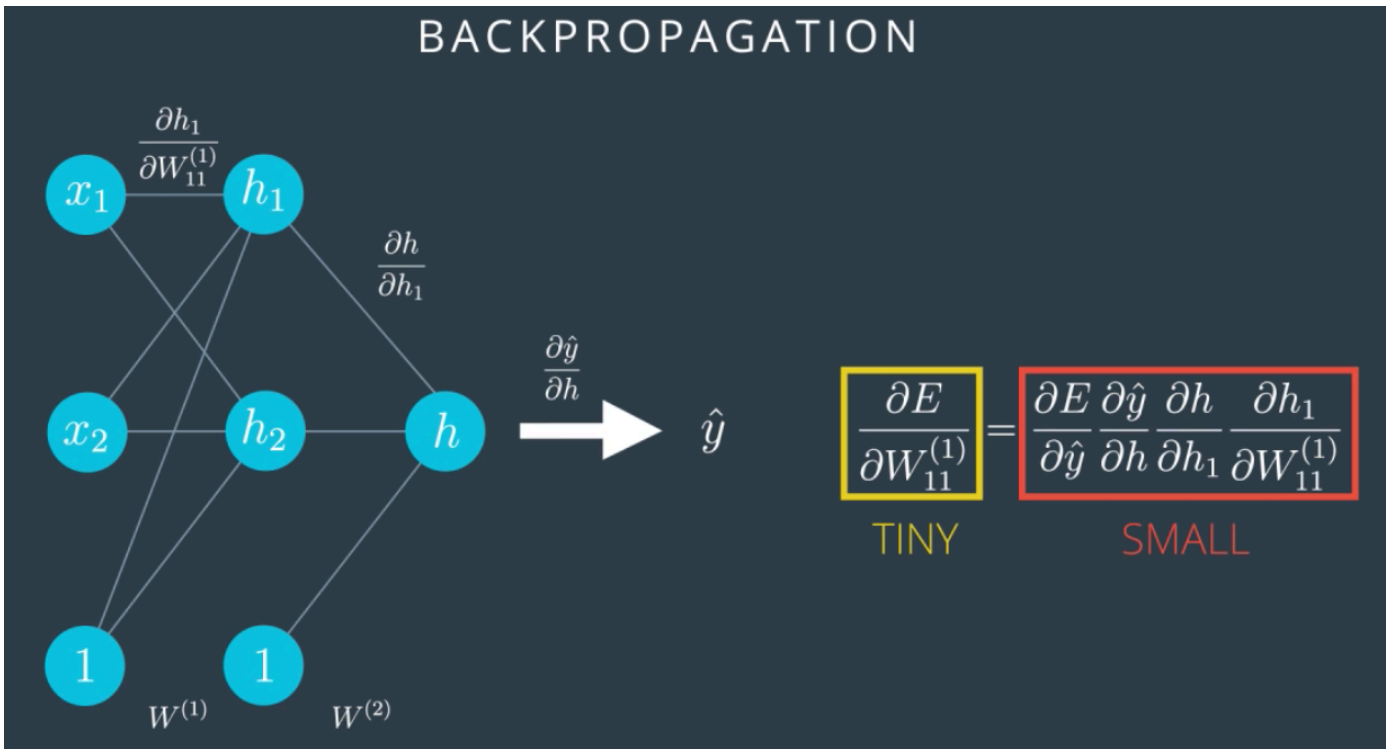
We save the average of gradients of the lastest few steps.

The momentum can push us when we reach to a local minimum



(image source from Udacity)

3.7 Vanishing Gradient

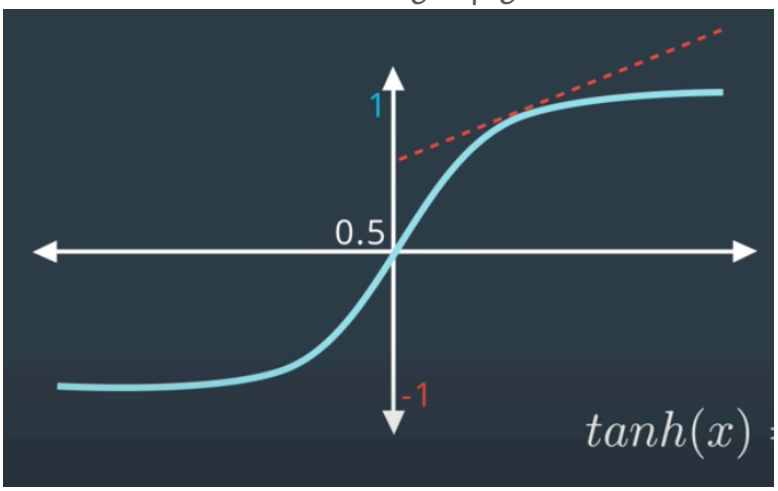


(image source from Udacity)

Solution: Change Activation Function (AF). By changing AF, fractions in red rectangle would be larger.

a. AF #1: Hyperbolic Tangent

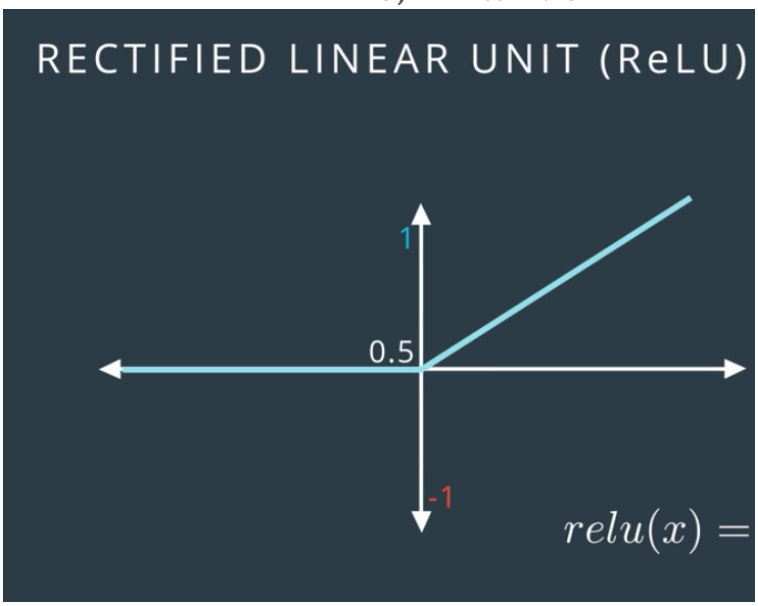
$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



(image source from Udacity)

b. AF #2: Rectified Linear Unit (ReLU)

$$relu(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$



For regression problems, AF in output layer could be ReLU

For classification problems, AF in output layer should be sigmoid

3.8 Batch Gradient Descent & Stochastic Gradient Descent

Batch Gradient Descent (BGD)

a. In each epoch, we go through **the entire dataset**, and accumulate the error term in output layer, then back-propagate the errors, then update the weights. Repeat epochs.

Mini-Batch Gradient Descent (MBGD)

a. In each epoch, we go through **a set of instances**, and accumulate the error term in output layer, then back-propagate the errors, then update the weights. Repeat epochs.

Stochastic Gradient Descent (SGD)

a. In each step, we feed **1 instance at a time** to the network, calculate the gradients and error terms based on the instance(s), then back-propagate the errors, then update the weights. Feed the second instance (or the second batch of instances).
b. Less accurate than BGD, but practical and efficient

3.9 Learning Rate Decay

Huge learning rate could result in diverging instead of converging

Rule of thumb: decrease the learning rate if the current model is not learning

3.10 Other Error Functions

a. Kilimanjerror
b. Reinerror
c. Ves-Oops-Vius
d. Eyjafvallajökull

Lesson 4: GPU Workspaces Demo

4.1 GPU Workspaces

Each student has 100 hours in GPU mode.

Before toggle GPU mode, make sure you save the notebook.

If GPU mode is active, before shutting down a notebook, disable GPU mode, otherwise your time limit will consume.

Workspace will disconnect **30 minutes** after the active action.

Some notebook need to submit manually in the classroom. Submit the notebook + all required files used in notebook.

Reset data: discard all changes and resotres a clean workspace. **All data will be lost.**

Lesson 5: Sentiment Analysis

5.1 Instructor: Andrew

5.2 Sentiment analysis

Input: Text

Output: Positive / Negative

Since neural networks only accepts numbers, we need to transform textual input data into numerical form in such a way that neural network can discover the correlation.

Goal: define a such transformation such that neural networks can discover correlation efficiently.

Tutorial steps:

1) Curating the dataset: come up with a theory for where the correlation exists in dataset
2) Validate the theory, transform it into input and output data
3) Iterate several epochs
4) Try to increase correslation of what the neural networks are able to discover
5) Check the weights to understand what's going to when back-propagation

What neural networks do is to search direct or indirect correlations in dataset.

Open the notebook and review the note below at the meantime**: *../ notebooks (filled) / 2. Neural Networks / sentiment-analysis-network / Sentiment_Classification_Projects.ipynb***

1) Curate a Dataset

Neural networks just try to learn the corrleation between **what we know** and **what we want to know**

2) Develop a Predictive Theory

How? **Try to think of I am the neural net myself**, and ask how would I figure out the correlations.

Remember: most reviews are original and have nuances (not duplicated), so for most reviews, we only solve once. Training neural net on the entire sentence doesn't generalize well.

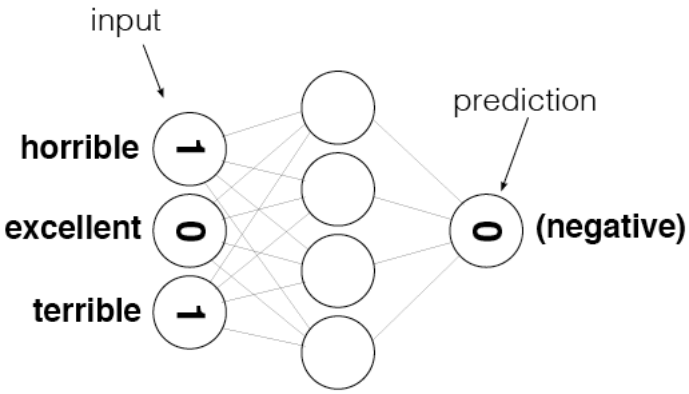
Theory ideas:

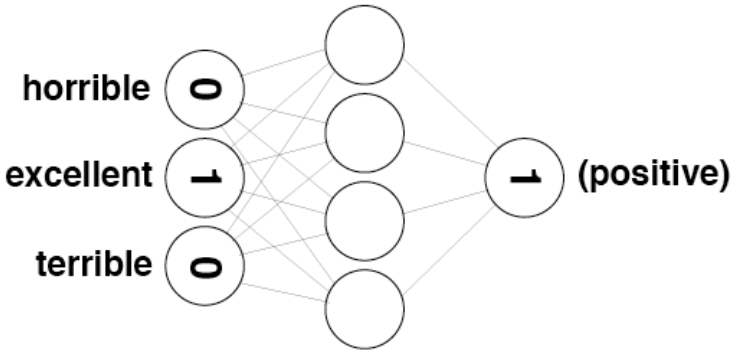
a. Find keywords: terrible, trash, great, excellent, impossible, fascinating, …

Project 1: Quick Theory Validation

Counter: a dict subclass for counting hashable objects.

Project 2: Transform Text into Numbers





Instead of drawing 2 neurons in output layer, why we there are 2 neural nets with different output neurons ?

Because if "positive" and "negative" are mutually exclusive. Instead of predict which of "positive" and "negative" is most likely, in this "2-neural-net" way, We can make "positive" and "negative" mutually exclusive, which reduces the number of ways that a neural net can make a mistake and help it learn a particular pattern.

Project 3: Build a neural net

What the input vector should be to neural net?

Idea 1: input are the word counts

Idea 2: input are 0-1 vector (1: the word apperas; 0: the word doesn't appear)

Weights initialization for better accuracy:

Rules:

1) Set initial weights to be close to 0 without being too small
2) Set initial weights in the range of [y, -y], where $y = 1 / \sqrt{(n)}$ $where\ n = \#of\ inputs\ to\ a\ given\ layer$

Project 4: Noise in Input data

Noise vs. Signal

1) Relevant signal → contribution of relevant words to hidden layer (i.e. relevant words * weights)
2) Irrelevant signal → contribution of irrelevant words ('.', 'the') to hidden layer

Our goal is to tune our neural net so that it can identify which words or a combination of words are revelent.

Project 5: Make the Neural Net more efficient

In the hidden layer, we only do the addition on the weighted sum if the input (word count) is positive

Project 6: Further reduce the noise and Increase the signal

Reduce the noise using the word frequency statistics

We frame the problem so that neural net can discover the correlations easily and reduce the noise.

Assignment: Project 1: Bike sharing

Project: Predicting Bike-Sharing Patterns

Lesson 7: Deep Learning with PyTorch

Open the notebooks and review the note below at the meantime**: *../ notebooks (filled) / 2. Neural Networks / intro-to-pytorch / Part 1, Part 2, … , Part 8.ipynb***

7.1 Tensors

A tensor is a generalization of vectors and matrices.

1) 1-dimensional tensor: vectors
2) 2-dimensional tensor: matrix
3) 3-dimensional tensor: example → an RGB image
4) …
5) N-dimensional tensor

7.2 Lots of **Lecture Note** are made inside the notebooks. Check with that.

Lesson 1: Convolutional Neural Networks

1.1 Instructors: Luis & Alexis

1.2 CNNs are used a lot in Voice User Interfaces, NLP and CV

1.3 Features in an image:

A feature is like what you firstly visually see an object when you identify the object from different objects.

Example: When you distinguish a human and a cat, you may distinguish them by the size, shape of eyes, etc.

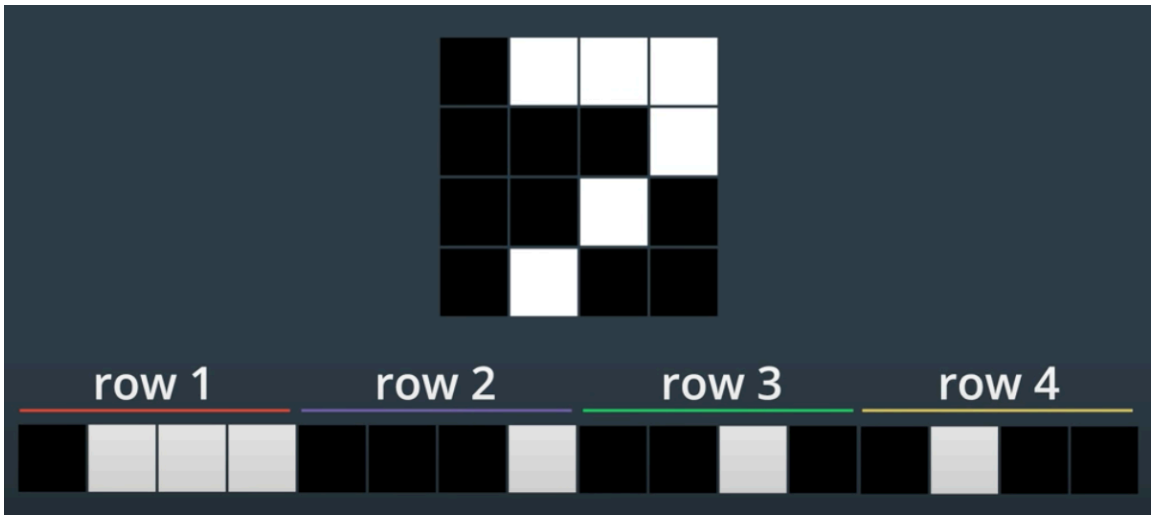
1.4 Normalization

Divide each pixel by 255 so that the range [0, 255] → [0, 1]

Normalization helps the gradient calculations stay consistent and converge faster.

$$normalized\ pixel\ value = \frac{pixel\ value - average\ of\ all\ pixel\ values}{standard\ diviation\ of\ all\ pixel\ values}$$

1.5 Flattenng



In order to feed an image to neural network which only accept a vector, we need to flatten the image from a matrix to a vector.

1.6 Class scores:

Indicates how sure the network is that a given input is of a specific class. A high score will result in a high probability after softmax.

1.7 Neural network architecture design (NNAD)

NNAD is usually to design the in-between hidden layers, such as:

1) How many hidden layers?
2) How many neurons in each hidden layers
3) …

A good way is to search online about related paper or architecture and

The more hidden layers included in the network, the more complex patterns the network will be able to detect.

1.8 Learn from mistakes

Loss function: measure how much mistake between the prediction and its true class

Example: cross-entropy loss (negative log)

$$\text{Cross} - \text{entropy loss} = \ \text{f(x)} = \begin{cases} \text{low value}, & \text{when prediction and true label agree} \\ \text{high value}, & \text{when prediction and true label disagree} \end{cases}$$

Backpropagation: quantify how bad a particular weight is in making a mistake (i.e. calculate the gradient and delta_weight)

Optimization: update the weight values

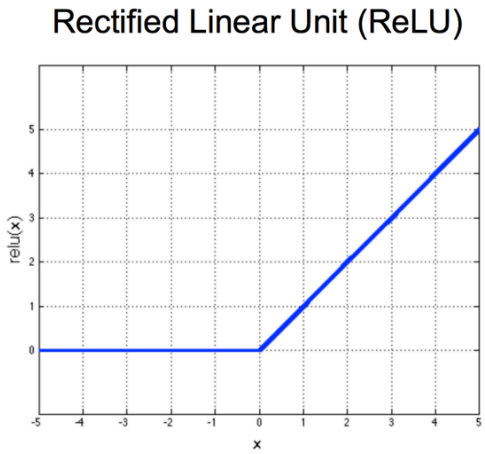
1) Gradient descent
2) Optimizer takes in

1.9 Define a PyTorch Network

The purpose of activation function: scale the outputs of a layer to a certain range so that they are consistent small values.

Example:

$$\text{relu}(x) \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x >= 0 \end{cases}$$



More Lecture Notes see *../ notebooks (filled) / 3. Convolutional Neural Networks / mnist-mlp / mnist_mlp_exercise.ipynb*

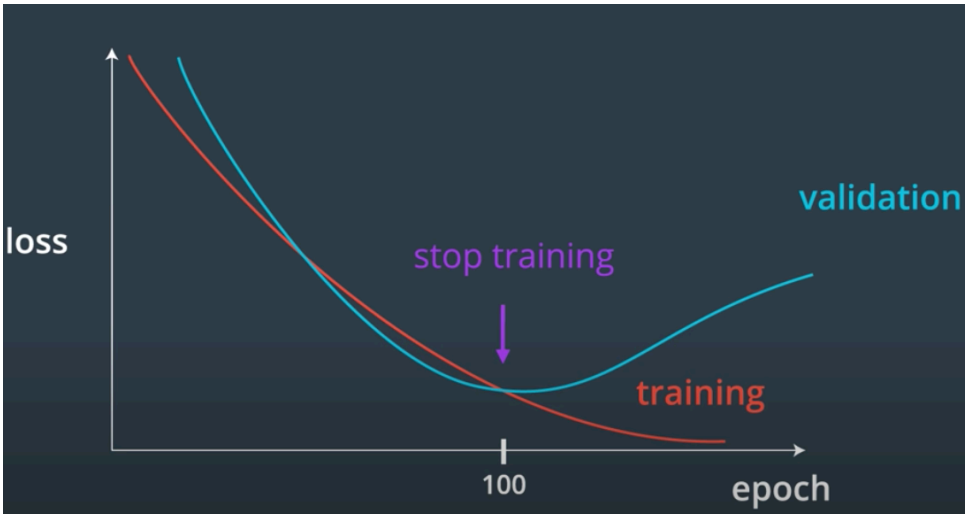
1.10 Model Validation

How many epochs should the model train ?

We need to introduce validation set.

Validation set: is not used while training and backpropagation. We update the weights on the training set. We use the validation set to validate if the model is performing well on the set. If yes, then stop training. Finally, we use test set to get the accuracy.

We can **use validation set to avoid overfitting**.



Open the notebook: *../ notebooks (filled) / 3. Convolutional Neural Networks / mnist-mlp / mnist_mlp_solution_with_validation.ipynb*

Summary:

Validation set is used to

1) Measure how well the current model generalizes while training
2) Tell us when to stop training a model (or save a model) → save when we meet the "elbow"

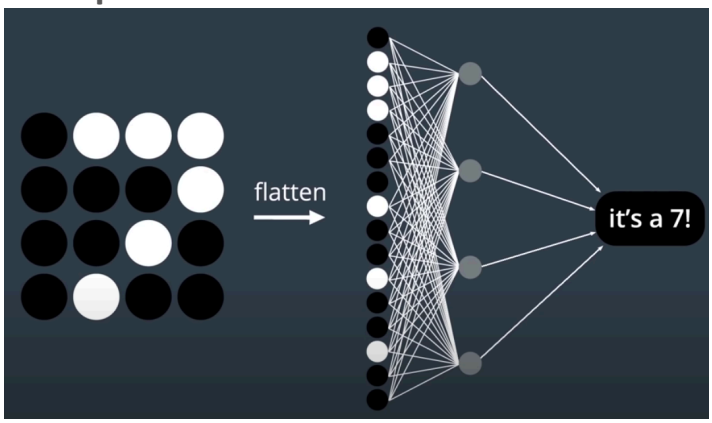
1.11 Pipeline to develop a neural network

1) Visualize data
2) Preprocess data (normalize, transformation)
3) Define a Network (after doing a research)
4) Define loss criterion and optimizer, Train the model
5) Save the best model
6) Test the best model on test set

1.12 MLPs vs CNNs

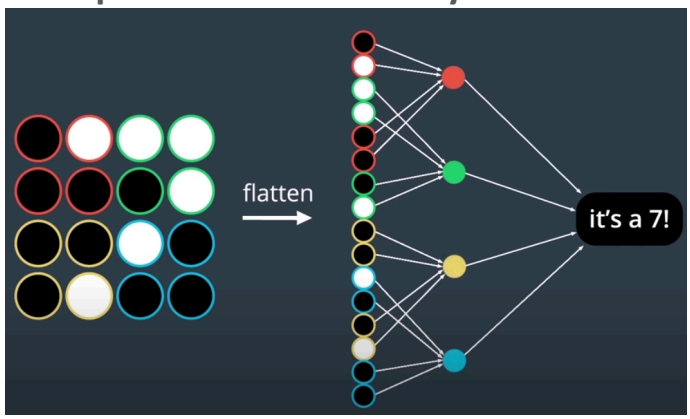
MLPs	CNNs
Issue #1: MLPs don't handle well if the input images is not centered or have different colors, size, …. Because we flatten the input image to a long vector, there is no structure and nowledge of the spatial in the image.	**Good #1:** CNNs are designed to consider the spatial knowledge in the input image.
Issue #2 Only use fully-connected layers. So if the input image is large, the weight matrix bombs.	**Good #2:** CNN will use sparely connected layers, so it can remember 2D structure in input image.
Issue #3: Only accept vectors as input	**Good #3:** CNNs accept matrices as inputs

Example:



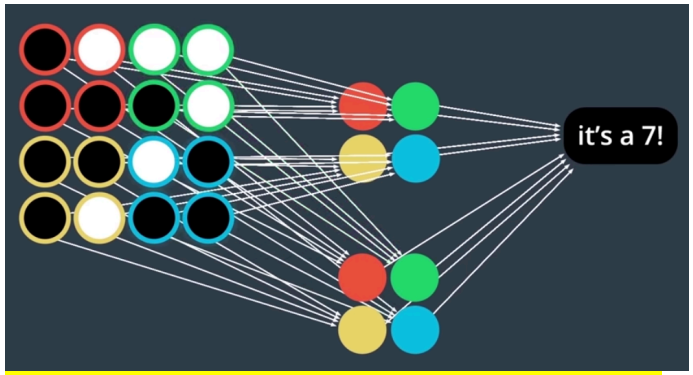
There are <u>many redundency</u> in the network, since every hidden neuron gets knowledge from all of the input neurons.

Example: Local connectivity



Here, we only use 1 kernel/filter
Each hidden neuron sees a local group of input neurons.

We can use 2 kernels/filters



(confusing: the 4 nodes in the midde are kernel or feature map??)

each hidden neuron exams a different regions of the image.

I think I could do a comparison project applying MLPs and CNNs on a same dataset to show how CNNs perform over MLPs.

TODO

1.13 Filters/Kernels:

CNNs analyze patches (groups of pixels) in the image at a time.

High-frequency components (e.g.  instead of ) corresponds to the edges of objects in images, which help the network classify the objects. Filters usually learn the patterns of those high-frequency components.

High-pass filters: Used to make image sharper and enhance high-frequency parts of an image.

The right is the result after high-pass filter.

It is really like an edge detector (remember Canny & Laplacian?)

Definitions:

A **kernel** is a matrix that modifies an image

A **kernel convolution** ('*' operation) is taking a kernel and apply it pixel-by-pixel on all pixels in the image.

When we apply kernel convolution on the corners of the image, we may use:

1) **Padding** is when we apply convolution on the corners in the image, we add a border of 0s
2) **Extend** is we repeat the pixel values on the corner to provide values for the convolution
3) **Crop** is we don't apply convolution in the image corners, so the output image will be a little smaller

Practice in Notebook: ***../ notebooks (filled) / 3. Convolutional Neural Networks / conv-visualization / custom_filters.ipynb***

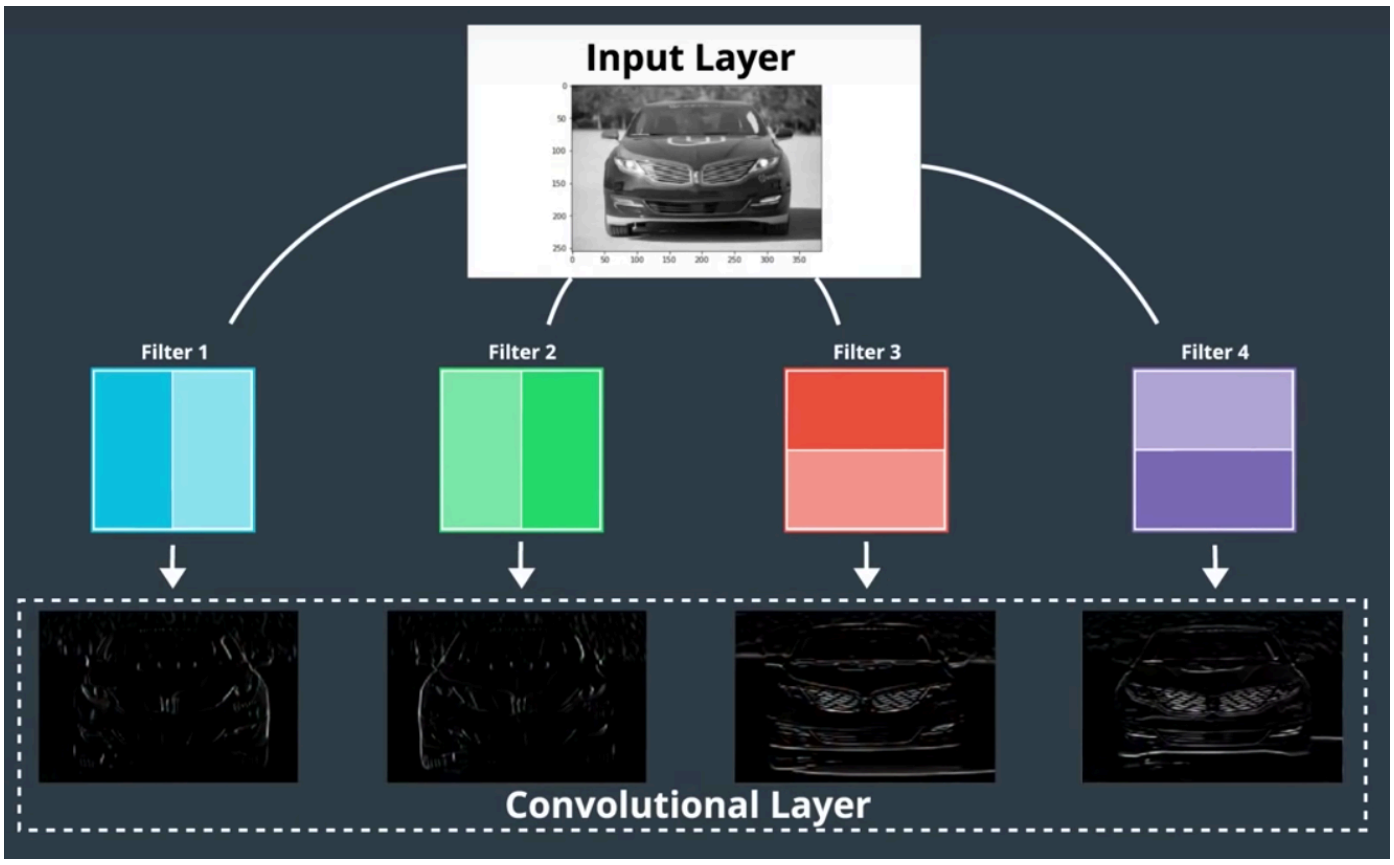
In CNNs, it will learn the weights in each kernel defined and learn the best filter weights as training.

Practice in Notebook: *../ notebooks (filled) / 3. Convolutional Neural Networks / conv-visualization / conv_visualization.ipynb*

Assume we use 4 filters to classify cars:

Filter 1	Filter 2	Filter 3	Filter 4
detect left dark, right bright	detect left bright, right dark	detect up dark, bottom bright	detect up bright, bottom dark

We get 4 **feature map (or activation map)** after applying kernel convolution.



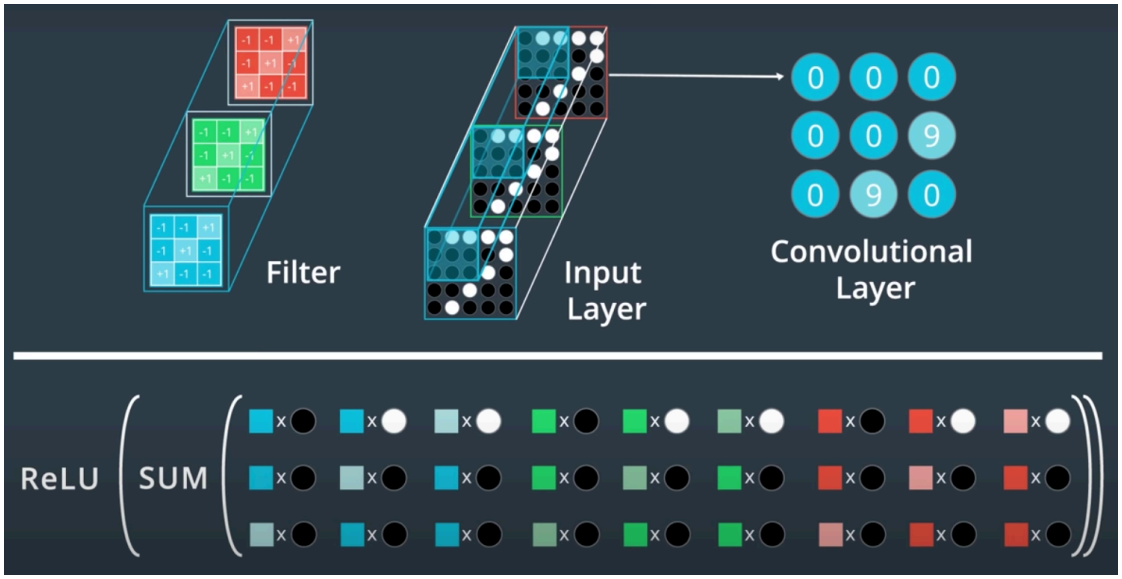
The lighter in feature map means that the pattern in the filter was detected in the image

Grayscale image: (height, width)

- Kernels/filters are also 2D matrices

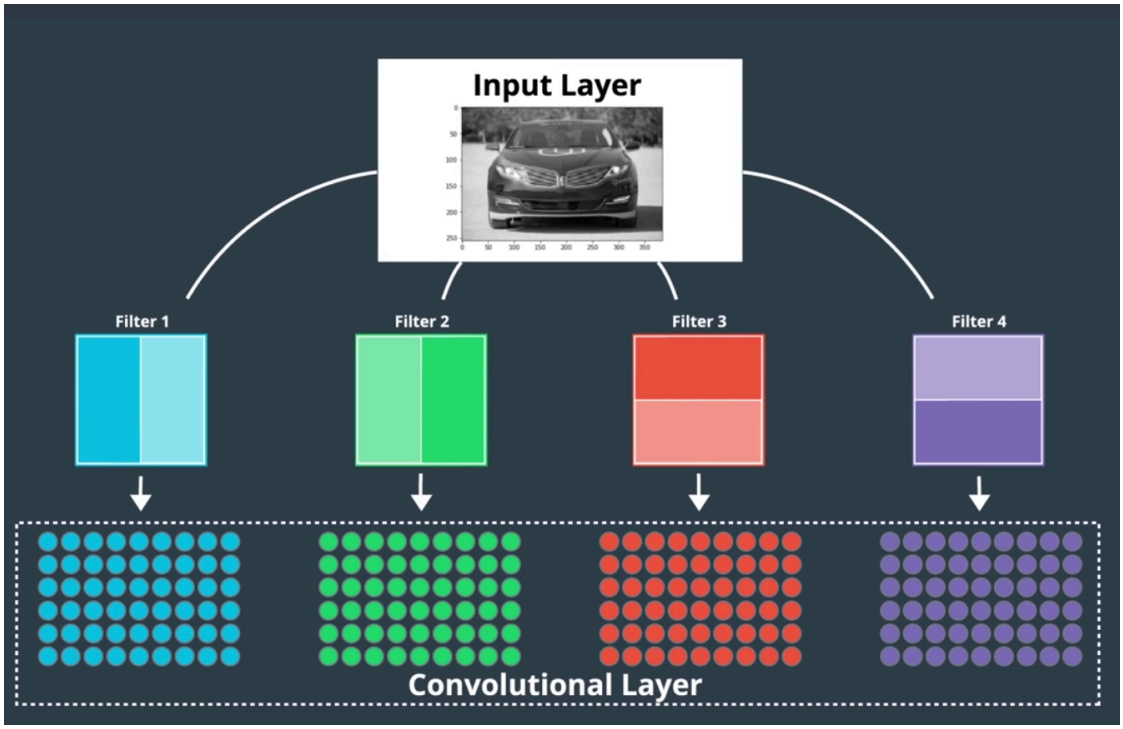
Colored image: (height, width, depth=3)

- Kernels/filters are also 3D (or say three stacked 2D matrices)

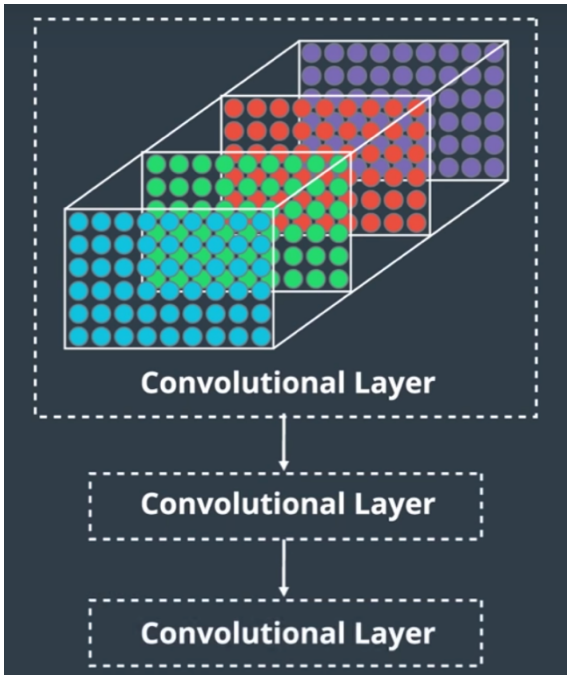


Deep CNNs:

We get 4 feature maps after the first convolutional layer A.



Then we stack the 4 feature maps and get a 3D array with shape (height, width, 4). Then we pass it to the next convolutional layer B. The B will discover patterns within the patterns which A discovered.



Stacked 4 feature maps discovered by A

Convolutional layer B: discover patterns within 4 feature maps

Convolutional layer C: discover patterns within result of B

Recap:

Dense Layers (DLs) in MLPs	**Convolutional Layers** (CLs) in CNNs
1. DLs are fully-connected, meaning each neuron in the next layer connects to **every** node in the previous layer.	1. CLs are locally connected where each neuron in the next CL only connect to a **subset** of neurons in the previous layer. 2. Parameter sharing 3. We can visually see the pattern which a kernel detects
Hyper-parameters: 1. number of neurons in hidden layer	Hyper-parameters: 1. Increase the kernel size → To detect size of the patterns 2. increase the number of filters → increase the neurons in convolutional layer

	3. Stride of convolution → filter step size
Both DLs and CLs have weights and biases that initially randomly generated Both needs to define a loss function As training, the weights are updated to minimize the loss function	

1.14 Stride and Padding

Stride

The step size when each filter moves horizontally or vertically

Padding

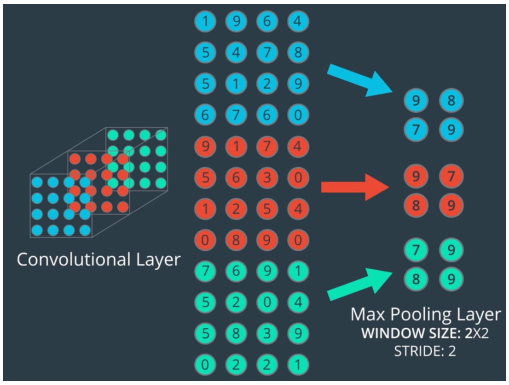
The size of 0s we append onto the input

Purpose to use padding:

1) Allow us to control the spatial size of the output size (typically same as input size)
2) Take care of the pixels in the border

1.15 Pooling layer

1) Pooling layer is usually just after a convolutional layer
2) **Purpose**: reduce the dimensionality (complexity) of the result of convolutional layers
3) For each pooling layer, we also have stride and window size
4) Types:
    a. Max Pooling



    b. Average Pooling

Instead of doing maximum, we do the average

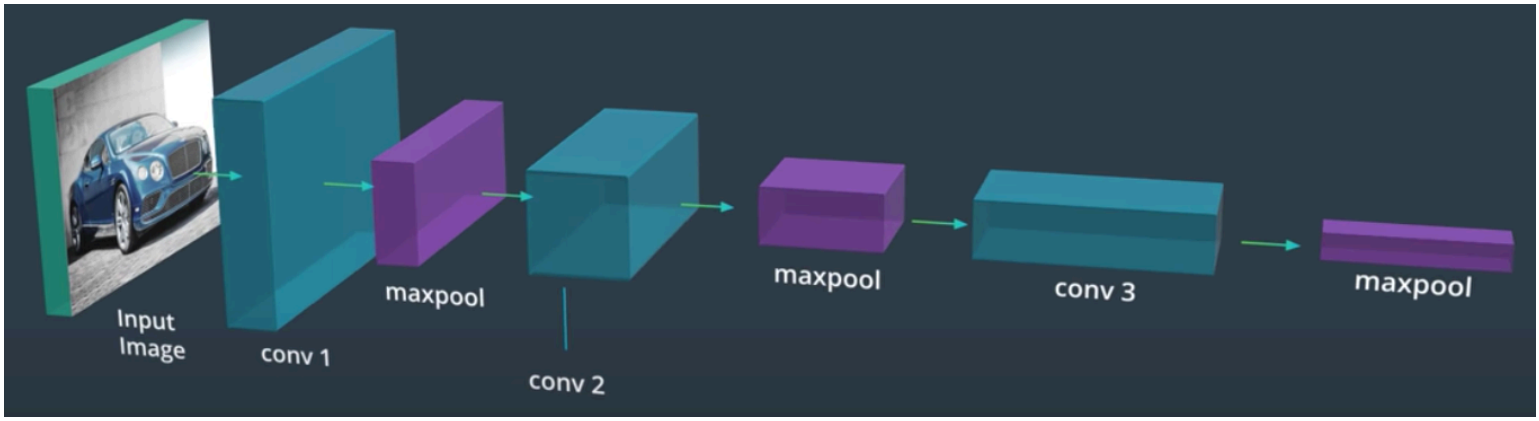
Practice in Notebook: ***../ notebooks (filled) / 3. Convolutional Neural Networks / conv-visualization / maxpooling_visualization.ipynb***

5) Drawbacks: Pooling layers throw away some information in the output of convolutional layers, which works well in classification problem, but doesn't in some problems such as face recognition (real face or fake face).
6) Alternatives to Pooling: **capsule network**
   Capsule Networks solve the problems which pooling layers have. It tries to learn spatial relationship between parts of objects, like 2 eyes,
   Capsule Networks are made of parent and child nodes that build up a complete picture of an object that is going to recognize.
   Capsules: a collection of nodes, each of which has a certain properties of a specific part, and it outputs a vector with magnitude and orientation
      Magnitude = the probability that the part exist, ranging from 0 ~ 1
      Orientation = the state of the part properties

Practice in Notebook: ***../ notebooks (filled) / 3. Convolutional Neural Networks / capsule_net_pytorch-master / Capsule_Network.ipynb***

1.16 Pipelines to design CNNs Architecture

1) Resize all input images to the same size
2) Preprocess the input data (normalization, numpy array to tensor, etc.)
3) Define a network containing convolutional layers, pooling layers, fully-connected layers, dropout layers, …



1.17 Define a convolutional layer

```
torch.nn.Conv2d(expected input depth = ?, desired output depth = ?,
kernel size = ?, stride = ?, padding = ?)
```

desired output depth: number of filters in the current convolutional layer
kernel size: usually ranging between (2, 2) for small images, and (7, 7) for large images
padding: we may get better result if we use padding such that a convolutional layer has the same height and width as its input from previous layer

Note:
Usually, as the network goes deeper, the depth increases in sequence, but the height and width decreases and discard some spatial information (by pooling layers). After each convolutional layer, we will follow ReLU activation function

1.18 Define a pooling layer

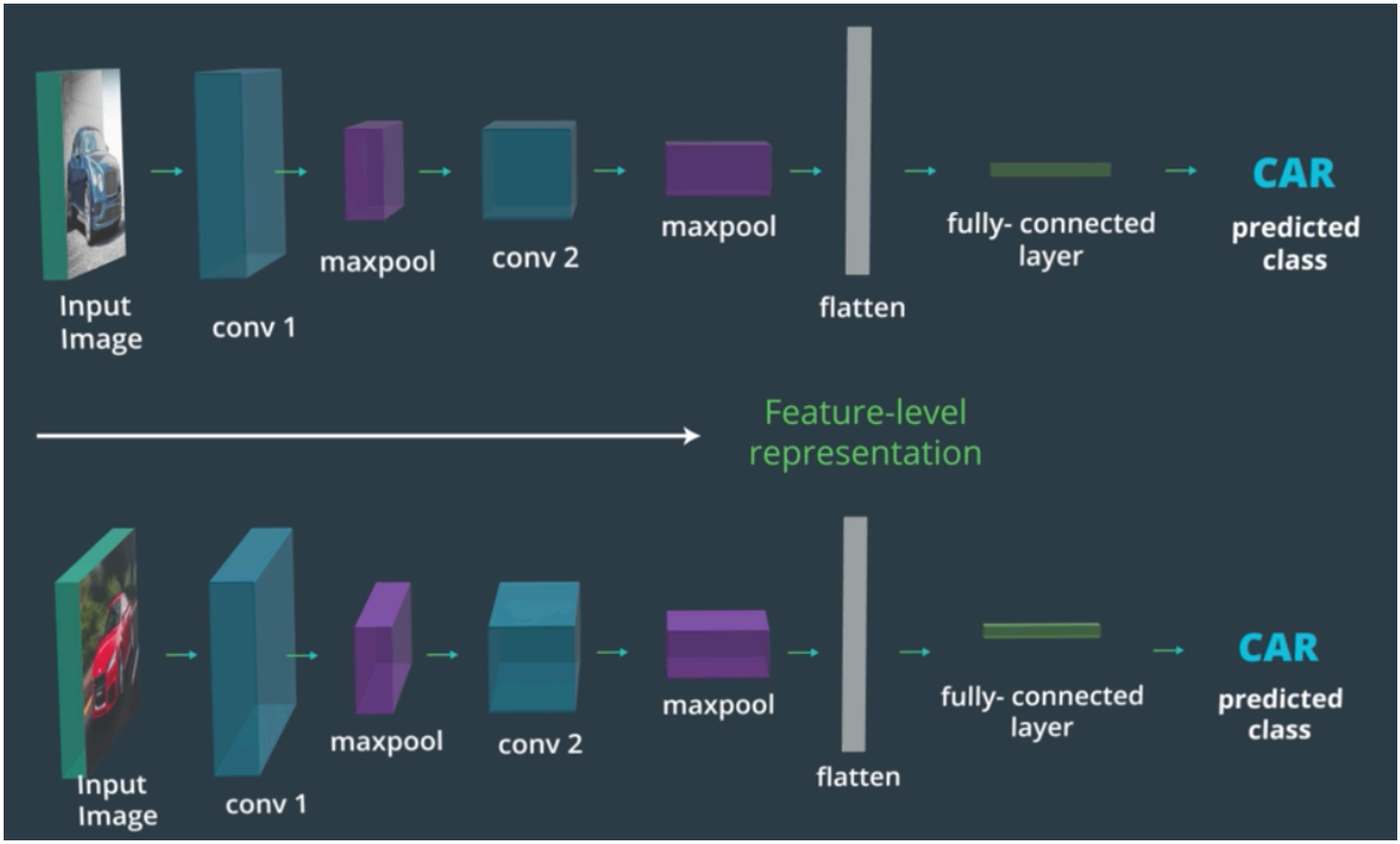
```
torch.MaxPool2d(kernel size = ?, stride = ?)
```

1.19 **More notes: *../ notebooks (filled) / 3. Convolutional Neural Networks / 1. CNNs-note.ipynb***

1.20 Feature Vector

Feature vector is so-called Feature-level representation

If we pass 2 car images to a same CNN, as we go deeper in the CNN, the detail is discarded and later output of 2 should look very similar. That is the details about what the car look like become more and more irrelevant as we go deeper.



1.21 GPU CUDA

1.22 Image Augmentation

Irrelevant information

1) Size (scale invariance)
2) Angle (rotation invariance)
3) Position (translation invariance)

We want network to learn an invariant representation of images

CNNs have 3 invariances above to some extent

Image Augmentation will make CNNs more invariant

1) Rotation existing images and add them to training set
2) Translate existing images and add them to training set

Data Augmentation also avoid overfitting

Open Notebook: *../ notebooks (filled) / 3. Convolutional Neural Networks / cifar-cnn / cifar10_cnn_augmentation.ipynb*

1.23 CNN breakthrough

AlexNet (Toronto Univ.)

- Pioneer dropout for avoid overfitting
- ReLU activation function
- Convolutional window: (11 x 11)

VGG Net (Oxford Univ.)

- VGG-16
- Conv layer + pooling layer + FC layer
- Pioneer exclusive use of (3 x 3) convolution window

ResNet (Microsoft)

- 152 layers!
- Find out vanishing gradient problem
- Pioneer “skip layer” concept, so gradients have shorter route to backpropagate

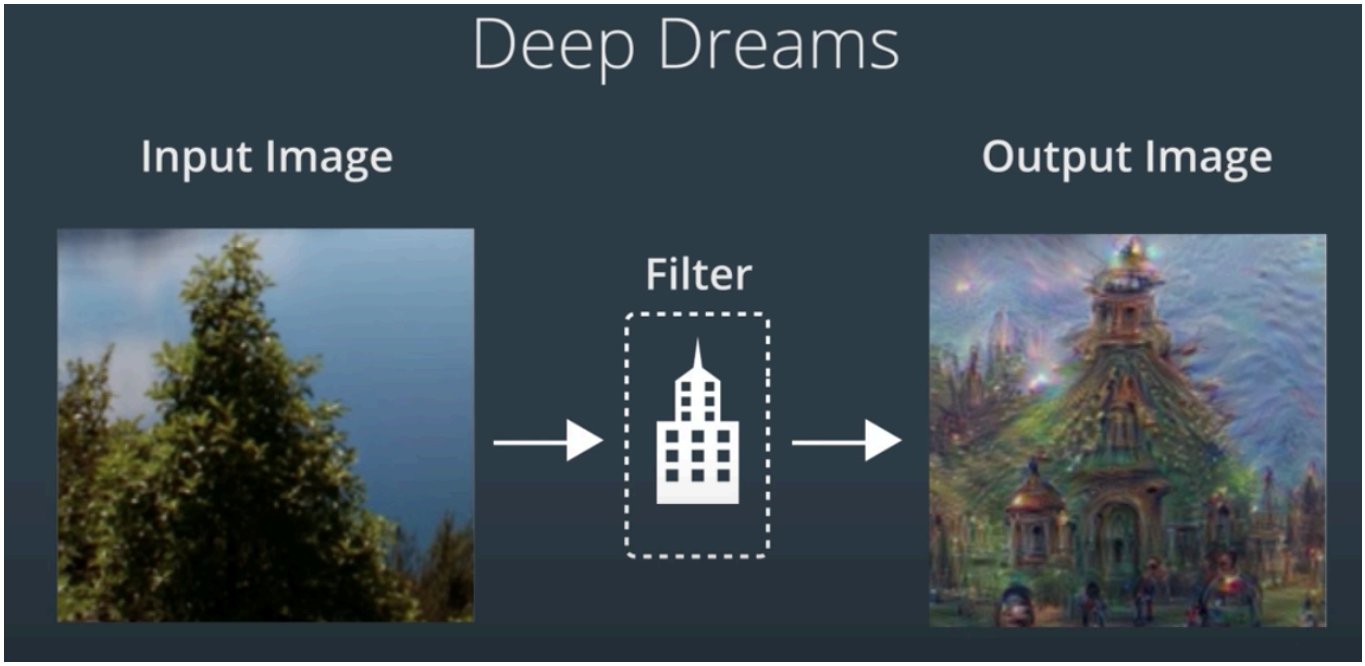
1.24 Visualize CNNs

1) Visualize activation maps

   https://experiments.withgoogle.com/what-neural-nets-see

2) Activation Maximization
   Construct input images that maximize the activations
   We can observe that
   a. First layer: detects shapes, colors (simpler features)
   b. Second layer: may detect stripes, geometry (complex features)
   c. Later layers: detect more complext patterns
   Example: Deep Dreams (Google)

We pass an image (e.g. a tree) to a CNN that classifies building, and at each step, we maintain the weights but change the input image a little to maximize the activations



Deep Dreams:

https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html

https://deepdreamgenerator.com/

Adversarial example:

https://openai.com/blog/adversarial-example-research/

Paper: Visualizing and Understanding Convolutional Networks

Recap:

CNNs can do many things, like detect human poses, gestures, etc.

Lesson 3: Cloud Computing

3.1 Amazon EC2 (Elastic Compute Clouds)

We can run instances allowd GPUs on EC2

AMI (Amazon Machine Image)

Documentation: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instances-and-amis.html

P2 instance : is scalable parallel processing GPU instance

Documentation: https://aws.amazon.com/ec2/instance-types/p2/

SHUTDOWN EC2 instance after finishing homework. Re-instantiate another instance for later work.

To Be Continued…

Lesson 4: Transfer Learning

4.1 Transfer Learning

Instead of constructing a CNN from scratch, it is better to take the knowledge (e.g. architecture, learning rate, …) from a trained CNN like ResNet, AlexNet, …

Definition:

Transfer Learning is to use a pre-trained network and apply it to a different task. There are many different approaches of transfer learning. It depends on difference on dataset of pre-trained CNN and the new task's dataset.

4.2 Approaches of transfer learning

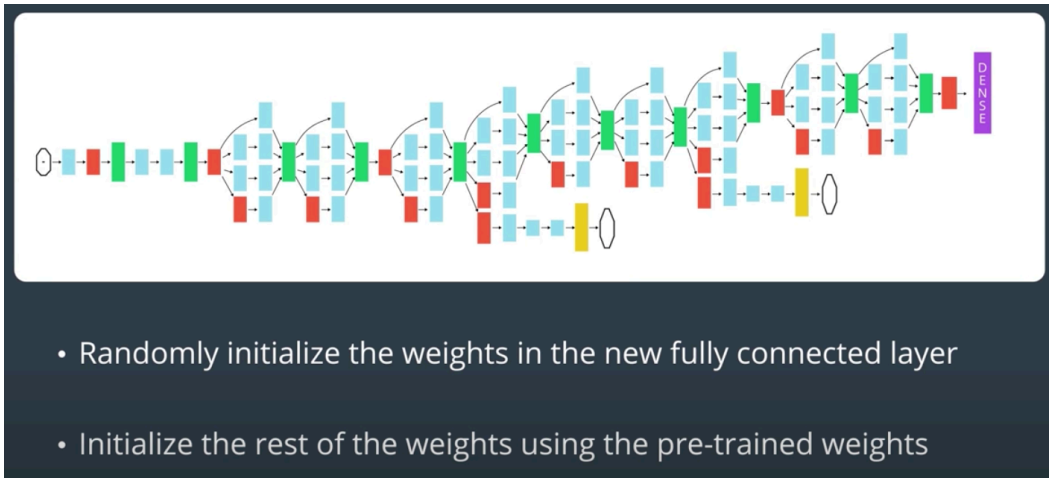
As we know that a pre-trained CNN has a "hierarchy" of feature extraction, that is, the first layer detects simple features (e.g. edges, colors, etc.), the second layer detects shapes, stripes, …, **the more deeper layer, the more complex pattern detected and the more specific to the dataset**.

The simpler features can be generalized to any dataset, so we can re-use those layers.

To transfer a pre-trained CNN to our own task, we can just replace the last few fully-connected layers, and re-train the whole network.

4.3 Fine-Tuning

If your dataset is very large and different from a dataset of a pre-trained model, we can:



An example: reuse inception network by replacing the last few layers

This is an example of fine-tuning which only has a slight changes or tuning on a pre-trained network.

Approaches of transfer learning depends on:

a. New dataset size
b. Similarity of new dataset to the original dataset of the pre-trained model

(Assume the original dataset is large and generic)

		new dataset similarity	
		Similar	Different
new dataset size	Large	• remove the last fully connected layer and replace with a layer matching the number of classes in the new data set • randomly initialize the weights in the new fully connected layer • initialize the rest of the weights using the pre-trained weights • re-train the entire neural network	• remove the last fully connected layer and replace with a layer matching the number of classes in the new data set • retrain the network from scratch with randomly initialized weights • alternatively, you could just use the same strategy as the "large and similar" data case
	Small	• slice off the end of the neural network • add a new fully connected layer that matches the number of classes in the new data set • randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network • train the network to update the weights of the new fully connected layer	• slice off all but some of the pre-trained layers near the beginning of the network • add to the remaining pre-trained layers a new fully connected layer that matches the number of classes in the new data set • randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network • train the network to update the weights of the new fully connected layer

Open Notebook: *../ notebooks (filled) / 3. Convolutional Neural Networks / transfer-learning / Transfer_Learning_Exercise.ipynb*

Use

```
for param in model.features.parameters():
  param.requires_grad = False
```

to freeze parameters

Lesson 5: Weight Initialization

When we do transfer learning, we initialize a network with a pre-trained network's weights.

For non-pre-trained models, there are some ways to initialize the weights

1) Initialize all weights to 0
2) Initialize all weights to large values
3) Initialize all weights to random values

Open Notebook: *../ notebooks (filled) / 3. Convolutional Neural Networks / weight-initialization / weight_initialization_exercise.ipynb*

We compare different weight initialization strategies on a same model.

Randomness is the key to weight initialization.
Better technique: **randomly initialize weights within a specific range**, it ensures that activations in hidden layers are different, so that backpropagation can response to the "responsibility" of weights to the error.

The notebook shows 6 strategies:

	Strategy	Performance
Constant weight	All Zeros	Bad validation error
	All Ones	Bad validation error
Uniform Distribution	Centered at 0 [-0.5, 0.5]	Better
	General Rule: Centered at y [-y, y] where $y = \frac{1}{\sqrt{n}}$ $n\ is\ number\ inputs\ to\ a\ given\ node$	Best
Normal Distribution	Normal Distribution (mean=0, stddev=y) where $y = \frac{1}{\sqrt{n}}$ $n\ is\ number\ inputs\ to\ a\ given\ node$	As good as to General Rule, perform slightly better on validation accuracy
Default	Automatic initialization	Default initialization is quite decent. Because every framework has implemented a initialization strategy its own

Lesson 6: Autoencoders

6.1 Autoencoder

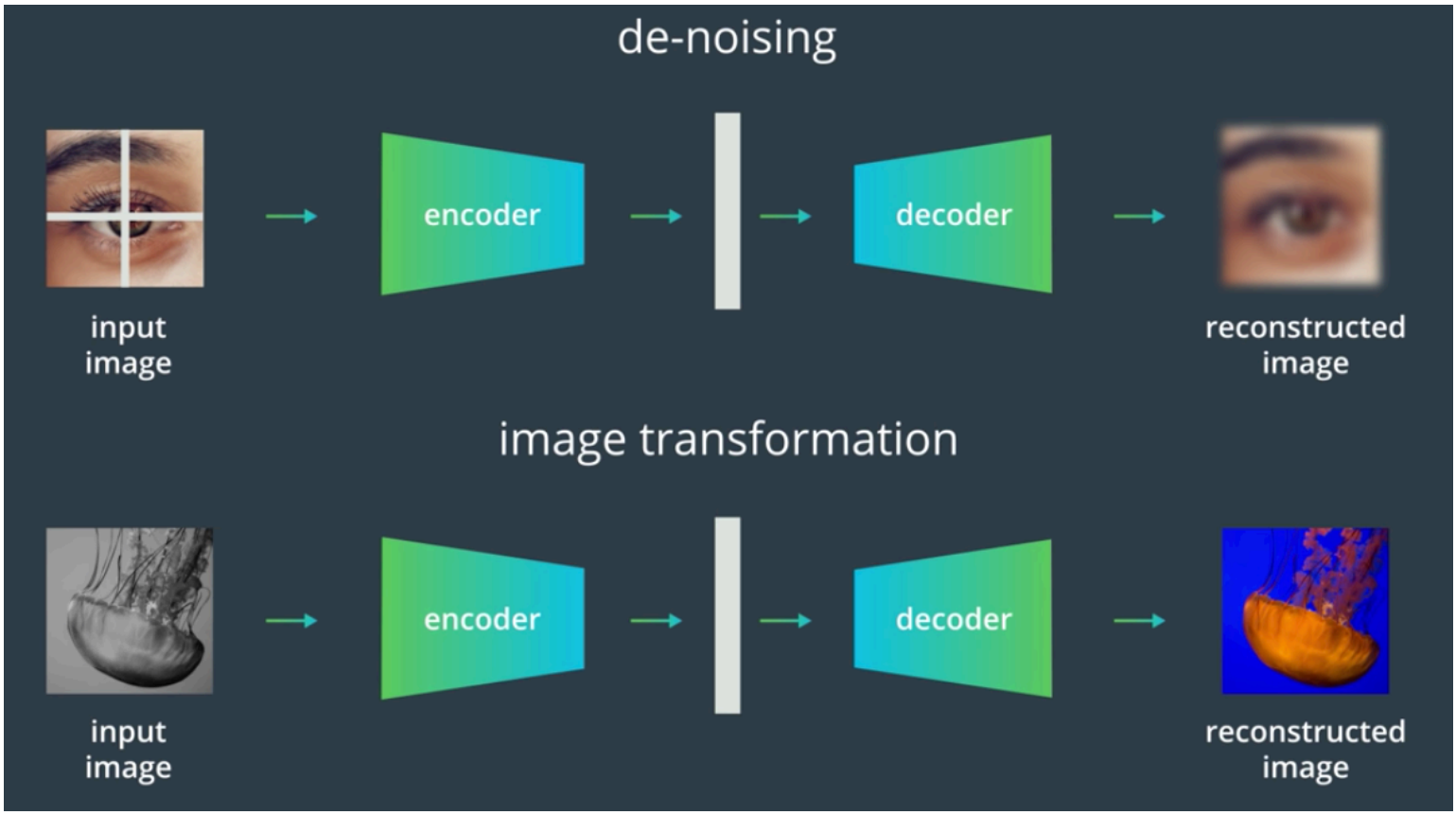
In CNN, before we go to FC layers, we flatten the output of the last covonlutional layer, we can also treat all convolutional layers kindom as a “data compression” process. We compress the original image to a feature vector.

Autoencoder consists of:

1) Encoder: compress input data
2) Decoder: reconstruct the compressed data to the original image as possible as it can

Use cases of autoencoder:

1) Compression & Reconstruction
2) Data Transformation



Our toal is to let neural network learn an encoder and a decoder to minimize the difference of the reconstructed image and input image.

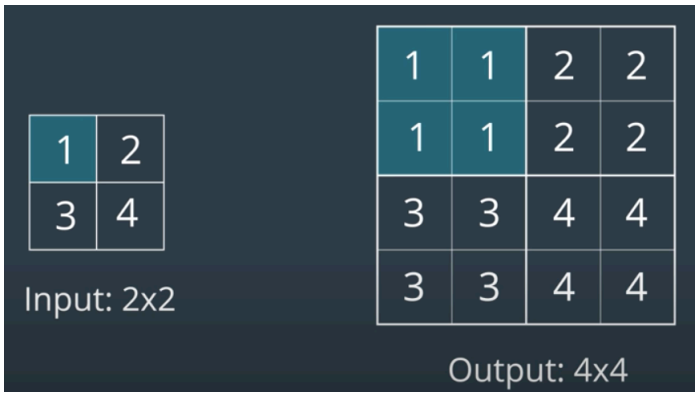
Open Notebook: ***../ notebooks (filled) / 3. Convolutional Neural Networks / autoencoder / linear-autoencoder / Simple_Autoencoder_Exercise.ipynb***

6.2 MLP autoencoder & CNN autoencoder

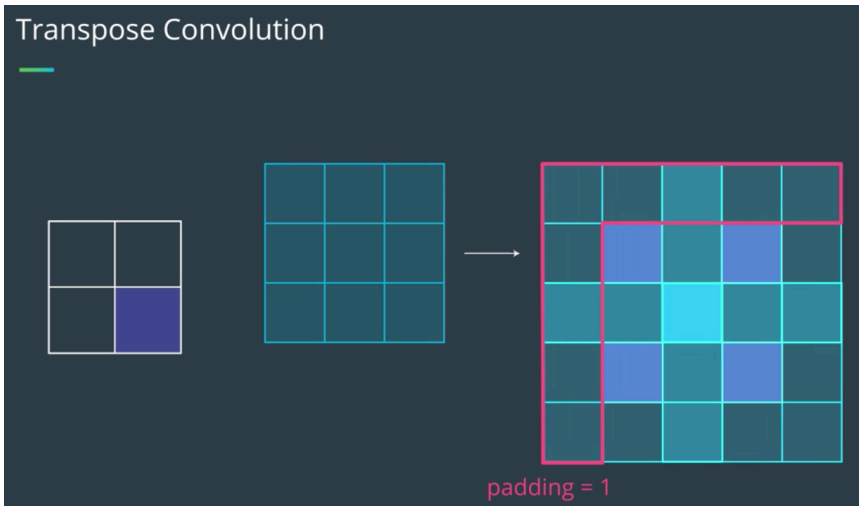
Decoder in CNN autoencoder:

We need to un-pool the activation maps so that the dimensionality increases, there are couple of ways:

1) Nearest neighbors



2) Transpose Convolutional Layer **TCL** (de-convolution layer)
   TCL uses Transpose Convolution operation



   Input: 2 x 2
   Kernel: 3 x 3 (we need to learn the weights in it)
   Output: 5 x 5
   Stride = 2
   Padding = 1

Open Notebook: *../ notebooks (filled) / 3. Convolutional Neural Networks / autoencoder / convolutional-autoencoder / Convolutional_Autoencoder_Exercise.ipynb*

Open Notebook: *../ notebooks (filled) / 3. Convolutional Neural Networks / autoencoder / convolutional-autoencoder / Upsampling_Solution.ipynb*

This notebook replaced a transpose convolutional layer with **nearest neighbor interpolation + a convolutional layer** in decoder

Recap: We tried 2 strategies in decoder

1) Transpose convolutional layer
2) Upsampling layer + convolutional layer

6.3 De-noising autoencoder

Open Notebook: *../ notebooks (filled) / 3. Convolutional Neural Networks / autoencoder / denoising-autoencoder / Denoising_Autoencoder_Exercise.ipynb*

Encoder detects important features from the noisy image, then decoder reconstructs the original image from the important features

Lesson 7: Style Transfer

7.1 What is style transfer?

Style transfer detects styles in a style image and content in a content image, then create a third image which keeps the content but renders it with the style.

The key of style transfer is to use a trained CNN to separate the content from the style of an image. Then later on we can merge the style of an image with another image to create a fancy image.

What is a style in an image?

Style of an image can be considered as its texture, colors, curvature etc.

7.2 Content Loss:

We have:

Cc: Content Representation from the content image

Tc: Content Representation from target image

Even though the style of the content image is changed, the 2 representations should be close.

$$L_{content} = \frac{1}{2} \sum (T_c - C_c)^2$$

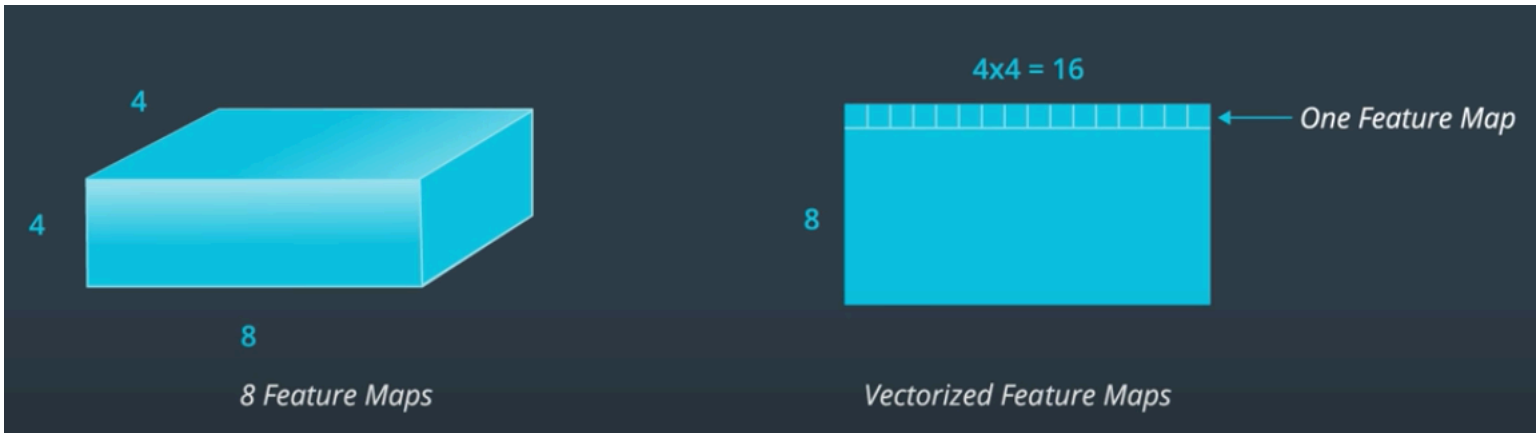
Our goal is to minimize the content loss.

7.3 Gram Matrix

The style representation of an image repleis on looking at correlations between the features in individual layers of the VGG-19 network, i.e. looking at how similar the features in a single layer are.

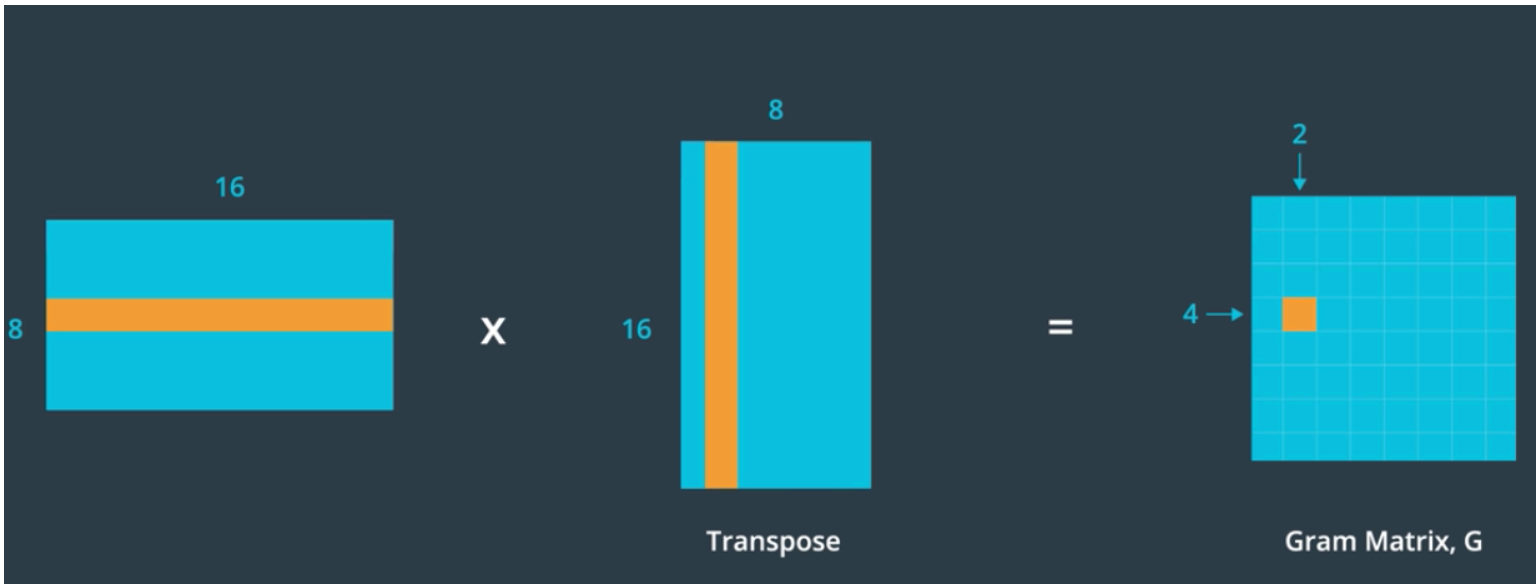
The correlations at each layer are given by a Gram matrix.

We can get the Gram matrix of a convolutional layer by vectorizing/flattening feature maps of the layer.



And then we multiply the vectorized feature maps by its transpose to get a square matrix. The square matrix is called Gram Matrix.

In Gram Matrix, it measures the similarities between the layers. Example: G[4, 2] measures how similar between the 2nd feature map and 4th feature map.



The resultant of Gram Matrix contains **non-localized information** about the layer. Non-localized information is information of an image which maintains the same even if the image is shuffled around in space. For example, we can still be able to see prominent colors and textures (the style) in the filtered image which is not identifiable.

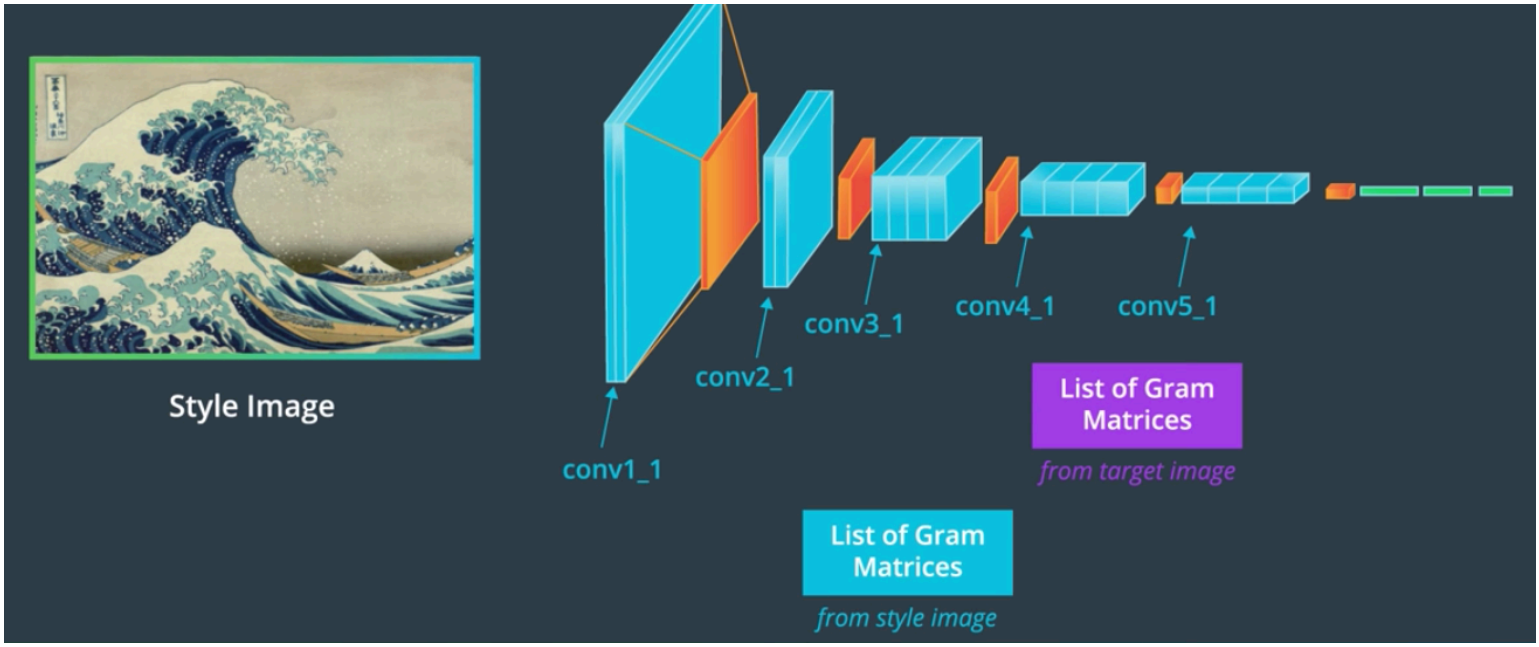
Note that, the dimension of Gram Matrix is relevant to number of feature maps in the convolutional layer, is irrelevant to dimensions or number of input image.

7.4 Style Loss

First, let's introduce

Ss: a list of (conv layer #, gram matrix) from style image

Ts: a list of (conv layer #, gram matrix) from target image



Style loss:

$$L_{style} = a \sum_{i} w_i \left(T_{s,i} - S_{s,i}\right)^2$$

$$where\ a\ is\ a\ constant\ that\ accounts\ for\ the\ number\ of\ values\ in\ each\ layer, and$$

$$w_i\ indicates\ the\ style\ weight\ of\ each\ convolutional\ layer, and$$

$$i\ indicates\ the\ conv\ layer\ number$$

Our goal is to change Ts to minimize L_{style}

So far,

Content loss $L_{content}$ tells how close the content of the target image is to the content image,

Style loss L_{style} tells how close the content of the target image is in style to the style image

Our total loss:

$$L_{total} = \alpha\, L_{content} + \beta\, L_{style}$$

$$where\ \alpha\ denotes\ the\ content\ weight\ and\ \beta\ denotes\ the\ style\ weight$$

*Typically, **beta is much larger than alpha**. The more $\frac{\alpha}{\beta}$ ratio, the more stylistic effect in the target image. Different combinations of content image and style image have different optimal alpha-beta ratio.*

Then we do back-propagation and optimize the target image to minimize L_{total}

Open Notebook: *../ notebooks (filled) / 3. Convolutional Neural Networks style-transfer / Style_Transfer_Exercise_TrainedGPU.ipynb*

Project: Dog-Breed Classifier

See Notebook: *../ projects / 2. Dog-Breed Classifier / dog_project / dog_app.ipynb*

Lesson 9: Deep Learning for Cancer Detection

Instructor: Bebastian (God of self-driving car)

For cancer detection, the earlier, the better.

Machine Learning engineers spend lots of time on data clearn up.

Pre-trained models with completely different images like cats, dogs will make cancer classification better if we use the weights to initialize our cancer detection model.

9.1 Measurement

4 Measurement:

1) Sensitivity (Recall, True Positive Rate)
2) Specificity
3) Recall (Sensitivity, True Positive Rate)
4) Precision
5) False Positive Rate

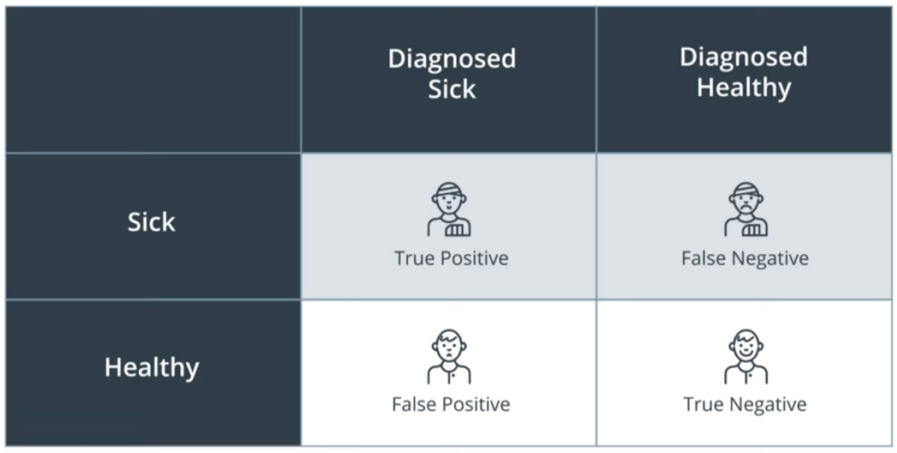
In the cancer example, sensitivity and specificity are the following:

- Sensitivity: Of all the people **with** cancer, how many were correctly diagnosed?
- Specificity: Of all the people **without** cancer, how many were correctly diagnosed?

And precision and recall are the following:

- Recall: Of all the people who **have cancer**, how many did **we diagnose** as having cancer?
- Precision: Of all the people **we diagnosed** with cancer, how many actually **had cancer**?

Confusion Matrix:

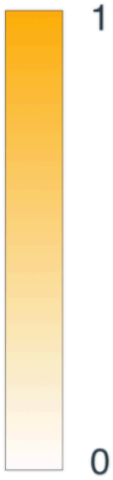


False Positive is also known as “Type 1 Error”; False Negative is also known as “Type 2 Error”

In confusion matrix, the sum of each row should add up to 1. However, the sum of each column isn’t necessary to add up to 1.

If we always predict B, the confusion matrix will look like this

	Predicted A	Predicted B	Predicted C
A	0	1	0
B	0	1	0
C	0	1	0



A very good indicator of a good classifier is the entries in the diagonal are larger than entries off of the diagonal.

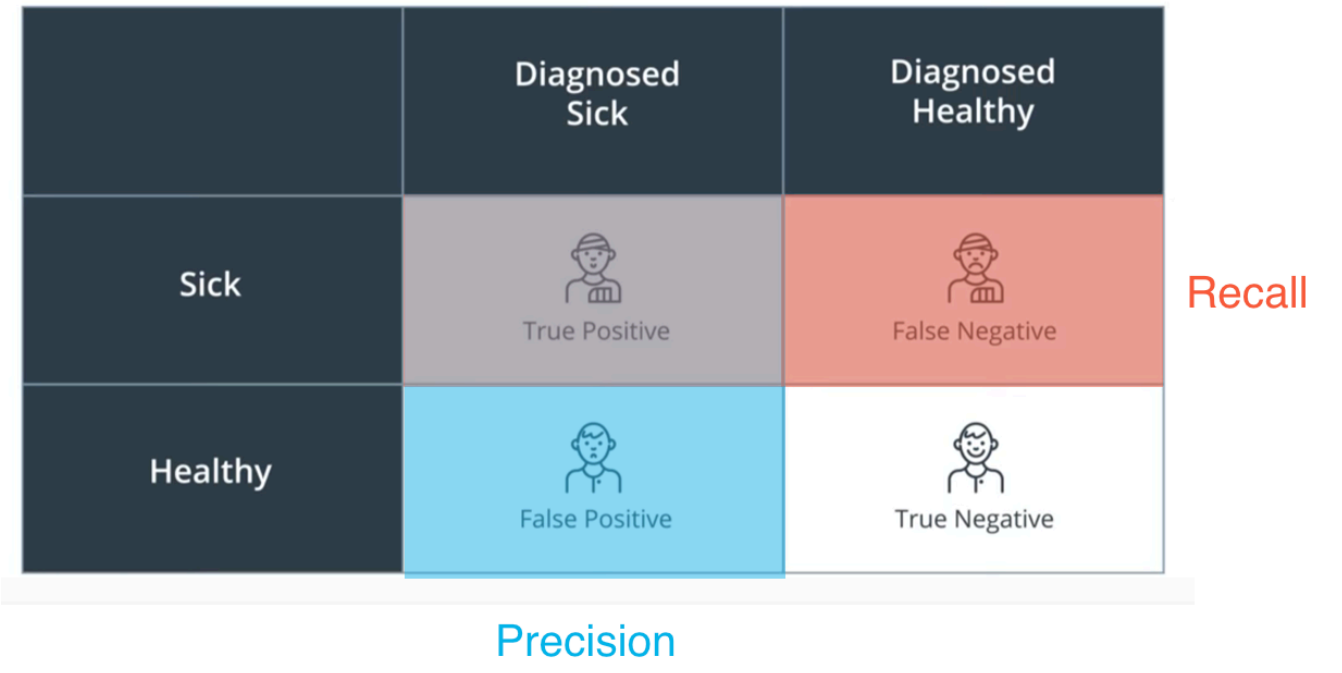
Formulas:
Horizontal : prediction; Vertical: ground-truth
(Sick – positive; Healthy – Negative)

	Diagnosed Sick	Diagnosed Healthy	
Sick	True Positive	False Negative	Sensitivity
Healthy	False Positive	True Negative	Specificity

$$Sensitivity = \frac{TP}{TP + FN}$$

$$False\ Positive\ Rate = \frac{FP}{TN + FP}$$

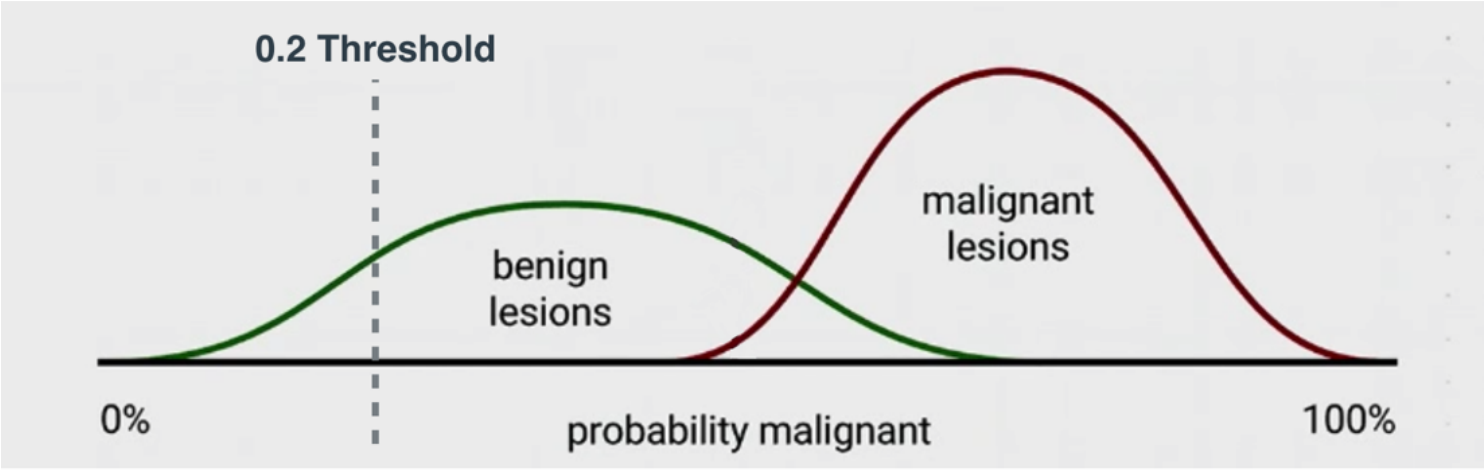
$$Specificity = \frac{TN}{TN + FP}$$

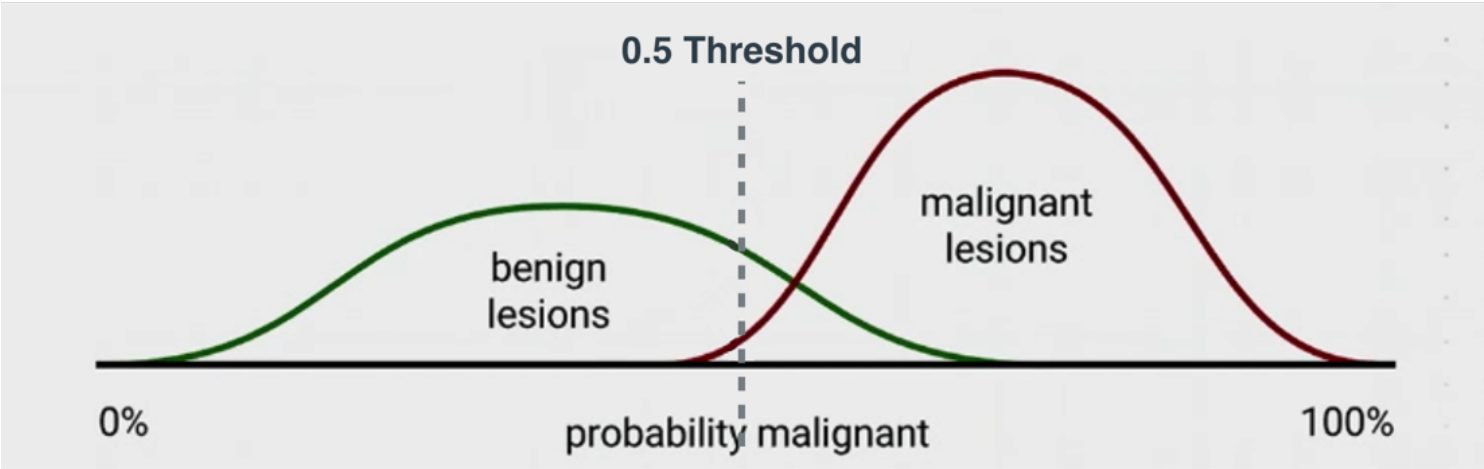


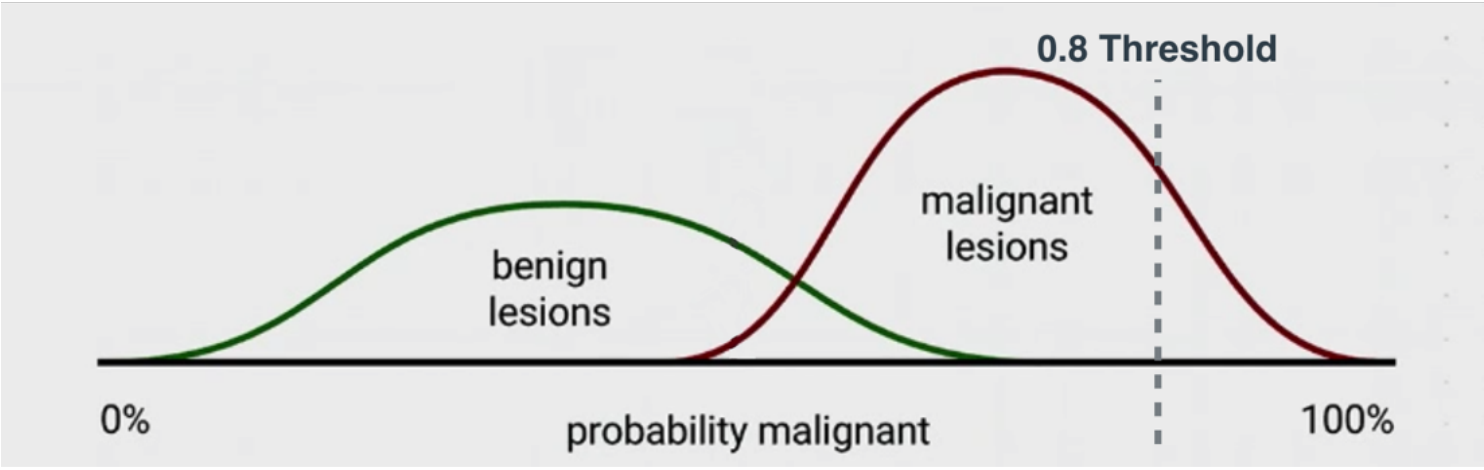
$$Recall = \frac{TP}{TP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

In Cancer Detection, we have a bias towards having **a low threshold** so that we won't mis-diagnoise any potential cases







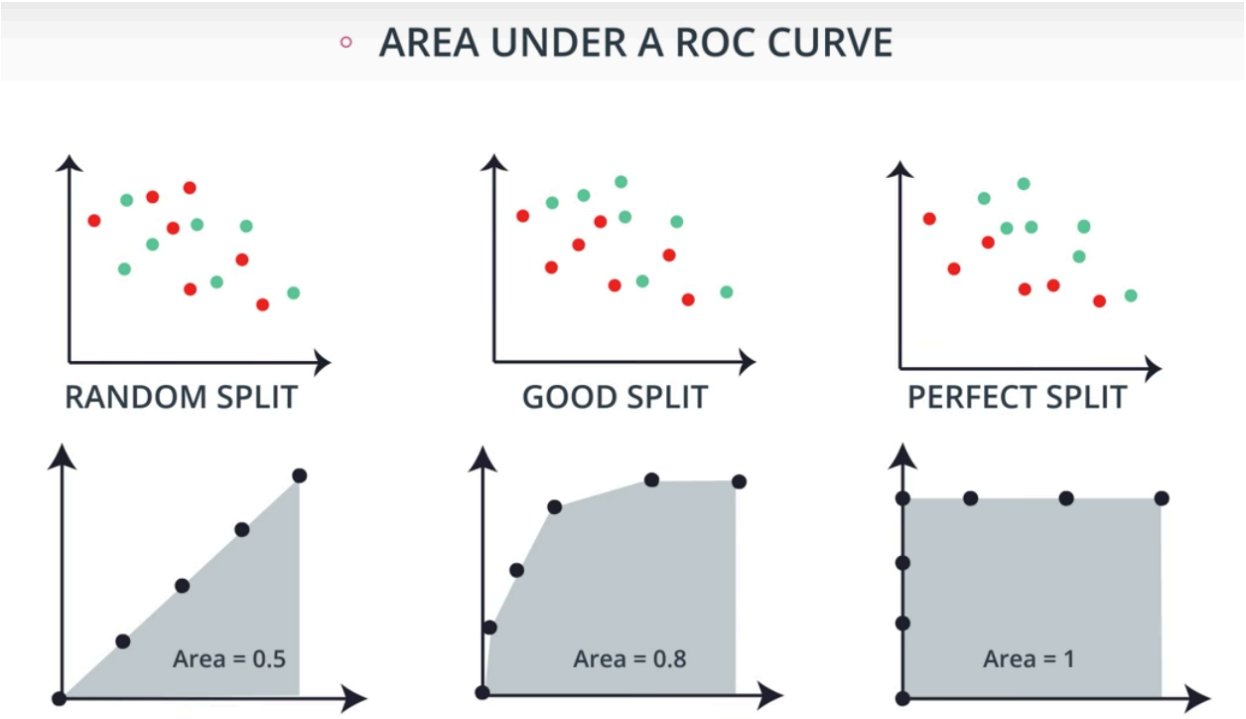
ROC Curve (Receiver Operating Characteristic Curve)

Motivation: we want a metric that:

- High value if perfect split
- Medium value if good split
- Low value if random/bad split

How do we calculate the ROC curve?

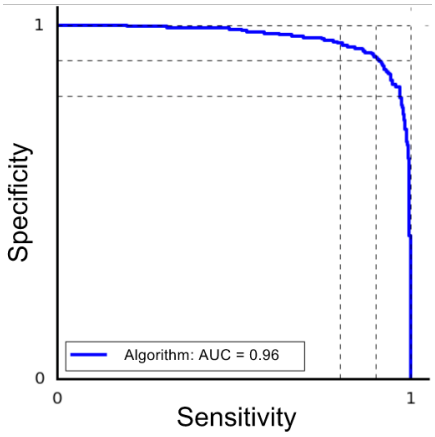
1. Step #1: calcualte the TPR and FPR along we move the boundary
2. Step #2: calculate the area under the ROC curve in TPR (horizontal axis) – FPR (vertical axis) graph



Summary: The closer the area under the ROC curve is to 1, the more perfect the split is

Note that: the area under the ROC curve can be less than 0.5

If we change the 2 axes to Sensitivity (horizontal axis) – Specificity (vertical axis), a good split will look like this

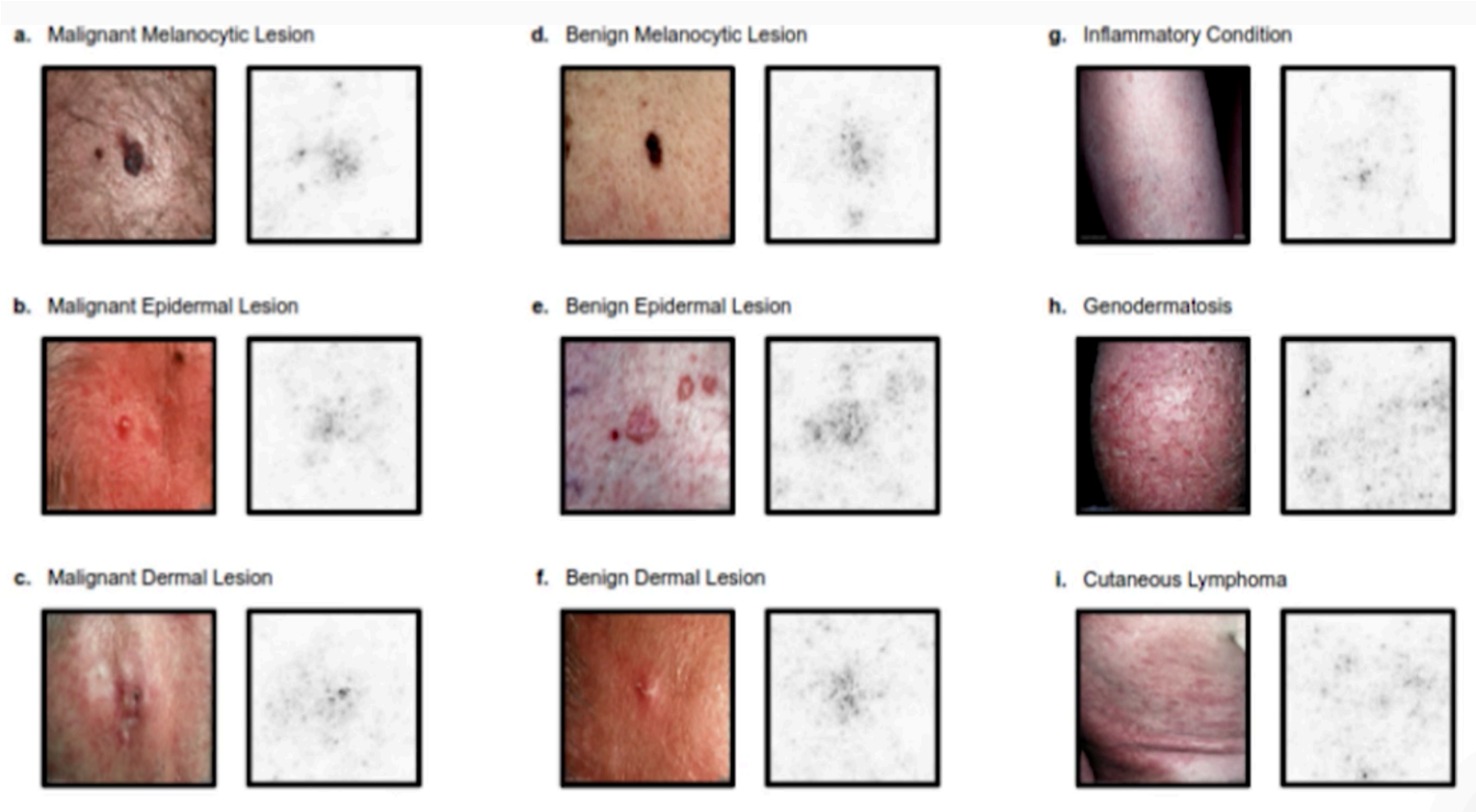


This is so-called **Sensitivity-Specificity Curve**.

Visualization techniques in Machine Learning

Technically, to visualize, we need to cluster a high-dimensional output into 2-dimensional space.

One of the technique is t-SNE



The black area in the right sub-plots shows importance to prediction of neural network

Conclusion:

Deep Learning can really lease people from doing so many repetitive work and give everyone the best service and make our life efficient.

Nature article written by Sebastian et al.:

Dermatologist-level classification of skin cancer with deep neural networks

Mini-project: Train a skin cancer detector using Deep Learning

TODO

Lesson 10: Jobs in Deep Learning

1. Suggestions in Deep Learning:

Finding opportunities in Deep Learning:

1) Keep reading learding blog post on Medium of famous post
2) Do job researches based on skills (e.g. neural networks) and keywords rather than job title
3) Create models and share it
    a. AWS Deeplens: a platform to run the model
    b. Google; AIR Vision Kit
4) Read recent papers, then demonstrate out understanding and implementation

2. Knowledges included in the deep learning degree:

- Building and training neural networks
- Model evaluation and validation
- Convolutional neural networks
- Autoencoders and feature extraction
- Transfer learning
- Recurrent neural networks
- Natural language processing
- Data augmentation
- Generative adversarial networks
- Hyperparameter tuning
- Model deployment and serving

More additional knowledge to explore:

- Explore related domains such as CV, NLP, RL
- Programming compentency in C++
- Build networks in PyTorch and TensorFlow
- Learn SQZ and apply it on data analysis

3. Jobs in Deep Learning Domain

There is a lack of people who can both build DL models and **deploy the models to production**.

4. Job titles in DL

- ML Engineer
- DL Engineer
- ML researcher
- Applied Research Scientist
- Software Enginner
- Data Scientist

5. Duties as a DL engineer:

- Design and build machine intelligence features
- Develop machine learning algorithms related to deep learning, such as object detection, language translation, and image retrieval in search algorithms
- Deploy analytics models in production and evaluate their scalability
- Code in C++ and Python
- Use ML frameworks such as PyTorch and Tensorflow to implement and prototype deep learning models
- Monitor and update a model after it has been deployed to production
- Employ data augmentation to work with small datasets
- Collaborate with other data and engineering teams on hardware, software architecture hardware and quality assurance

Udacity Job Boards:

https://career-resource-center.udacity.com/job-boards

Interview for a DL role:

https://career-resource-center.udacity.com/networking/informational-interviews

Real-world Applications:

- Language Translation

- Self-driving cars
- Intelligent traffic signals
- Predicting consumer behavior
- …

Project: Optimize Your GitHub Profile

1. Prove your skills with GitHub – Career Review Requirement

TODO (before apply for graduation)

2. Free course with Git:

https://www.udacity.com/course/version-control-with-git--ud123

3. When hiring managers looks at your profile:

1) Focus on the heat map for contributions
2) Meatiest work/repo
3) Format of commit messages
    a. Guideline to commit messages
       http://udacity.github.io/git-styleguide/
4) Githut “Issues” section (indicator of good collaborator
5) Break the ice, start with reading readme and fixing bugs in open source project

4. How to write README

https://classroom.udacity.com/courses/ud777/lessons/5338568539/concepts/53317786070923

5. How to contribute a open-source project

- Start from improve the documentation
- Activate “Watch” when you want to get any notification of a project

6. Good GitHub practices

- Be active on commits
- Get 2-4 weeks streaks

4. Recurrent Neural Networks

Lesson 1: Recurrent Neural Networks

1.1 CNN vs. RNN

CNN	RNN
1. Good on finding spatial and visible pattern in training data	1. Good in analyzing sequential data
Applications: 1. Image Classification 2. Voice User Interface (WaveNet) 3. NLP (may underperform than RNNs) 4. Computer Vision (e.g. text classifiation) 5. Teach game agent to play game 6. Guess you draw (e.g. Quick Draw) 7. Train a model to play game Go 8. Self-driving cars …	**Applications:** 1. Text generation 2. TV script generator 3. Generate drowing (Sketch RNN) 4. Speech Recognition (e.g. Alexa) 5. Time series Prediction (e.g. traffic patern, stock prediction) 6. NLP (e.g. translation, Q&A, chatbots) 7. Gesture Recognition …

1.2 RNN Introduction

Instructor: Ortal

RNNs reply on Temporary Dependendies in input

RNNs perform the same task for each element in the input sequence

Comparing with ANNs, RNNs add memory/time delay (TDNN, Time Delay Neural Network)

RNNs example:

- ELMAN Network (1990)
- Jordan Networks
- LSTM (Long Short-Term Memory)
- GRUs (Gated Recurrent Units)

Vanishing Gradient Issue

As we train our network using backpropagation to update the weights, we update those weights with gradients, and while backpropagating, those gradients are continuously multiplying with multivariate derivatives and becoming smaller and smaller, making the earlier layers have extremely small gradients, so they rarely learn as we train.

Solutions:

1) LSTMs
2) GRUs

1.3 **Reminder:** Vanilla FFNN (Feed-Forward Neural Network)

1.3.1 Non-linear function approximation

1.3.2 Main type of applications

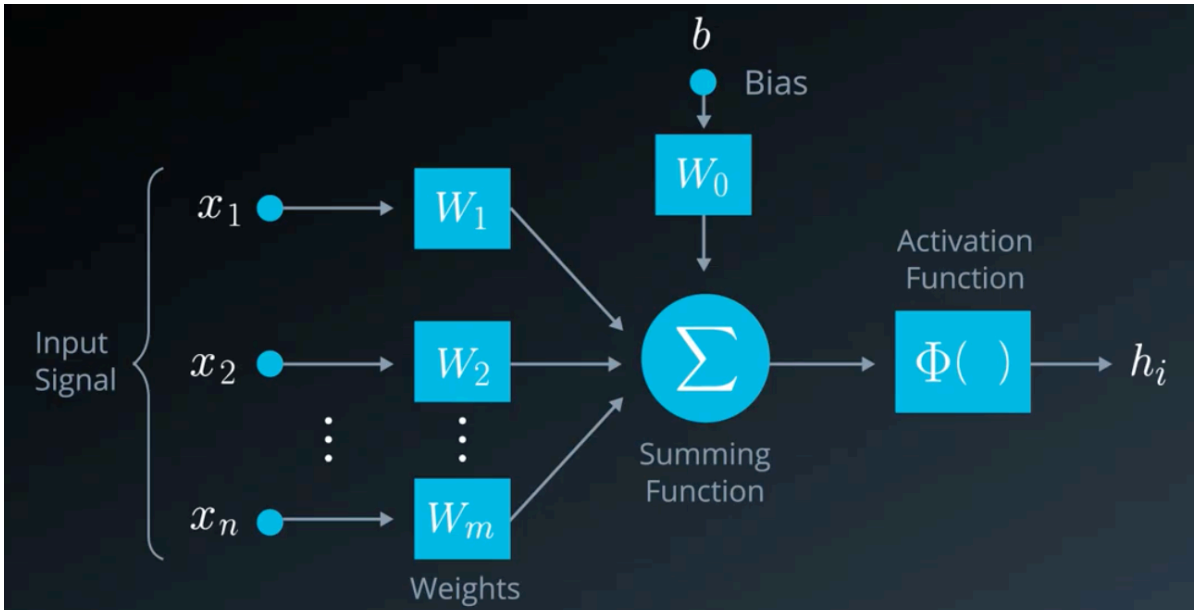
1) Classification
2) Regression

1.3.3 Vanilla FFNN

Task: find the best weights of all layers that best fit the relations between input and output. This is called static mapping, since the network has no memory, the current output only relies on the current input

Goal of training phase: To yield a network that generalizes beyond the train set

For a single neuron, this is how we calculate its output:



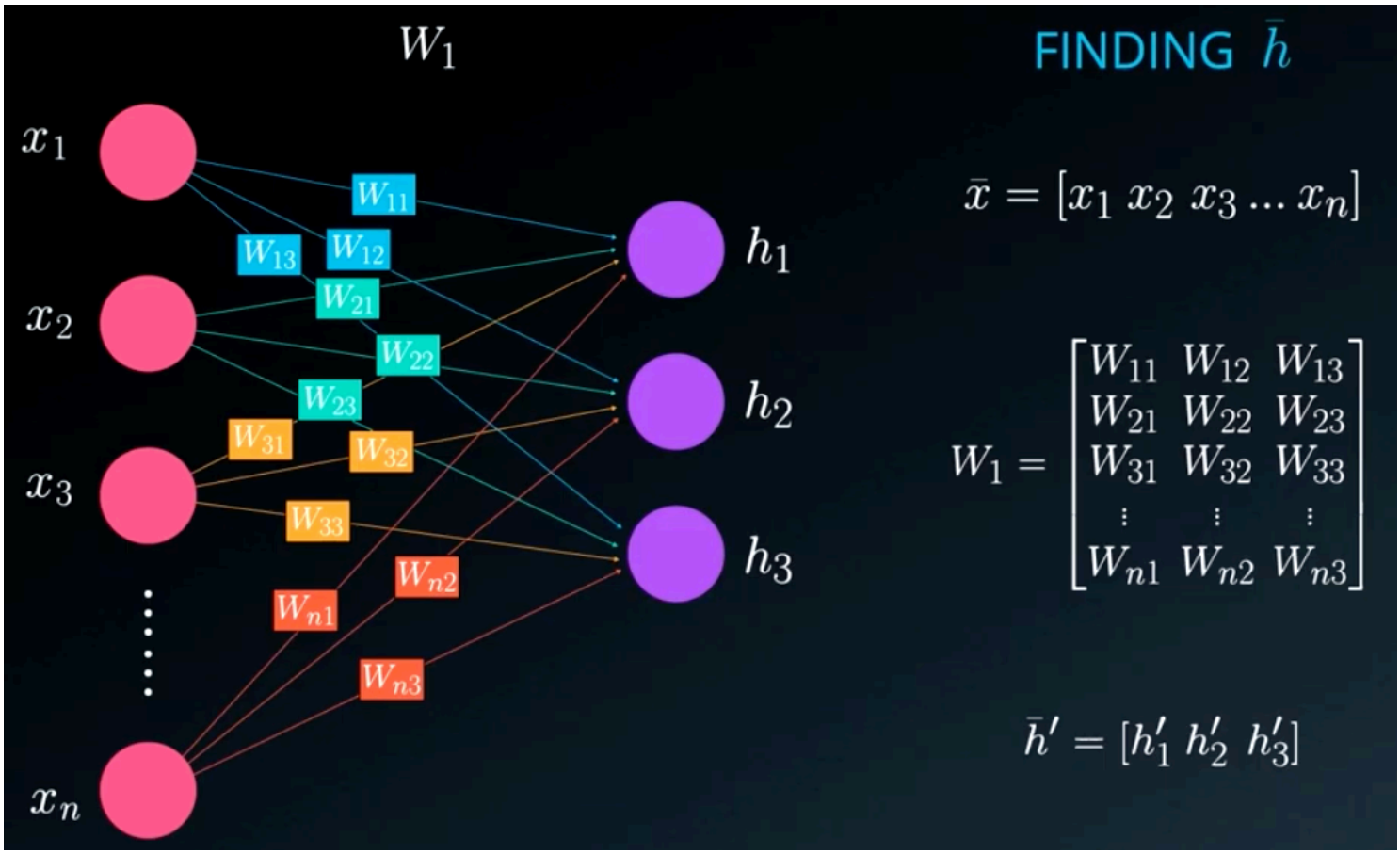
1.3.4 Two steps in Training Phase

1) Feed-forward
    - Compute the output throughout the network
    - Compare it with the target
    - Calculate the error
2) Back-propagate
    - Change the weight to minimize the error

1.3.5 Feed-forward

Notations:

- W_k is weight matrix k, between layer k and layer k+1
- W_{ij}^k is the weight between i-th node in layer k and j-th node in layer k+1



h' denotes the weighted sum

$$\begin{bmatrix} h'_1 & h'_2 & h'_3 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & \ldots\ldots & x_n \end{bmatrix} \cdot \begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ \vdots & & \\ W_{n1} & W_{n2} & W_{n3} \end{bmatrix}$$

We need to pass the weighted sum to activation function to get the activation (output) of that neuron.

Non-linear activation functions:

1) Sigmoid
2) ReLU
3) Tanh
4) Hyperbolic Tangent
5) …

For the activation at hidden layer p

$$h^p = \phi\left(h^{p\,\prime}\right) = \phi\left(h^{p-1} \cdot W^p\right)$$

For a specific hidden node i:

$$h_i = \phi\left(x_1 W_{1i} + x_2 W_{2i} + \cdots + x_n W_{ni}\right)$$

Non-linear activation function **allows the network to represent non-linear relationships between the input and the output**. It is valuable because most real-world data has non-linear relationship.

Error Calculation

Error: $\bar{e} = y - \hat{y}$

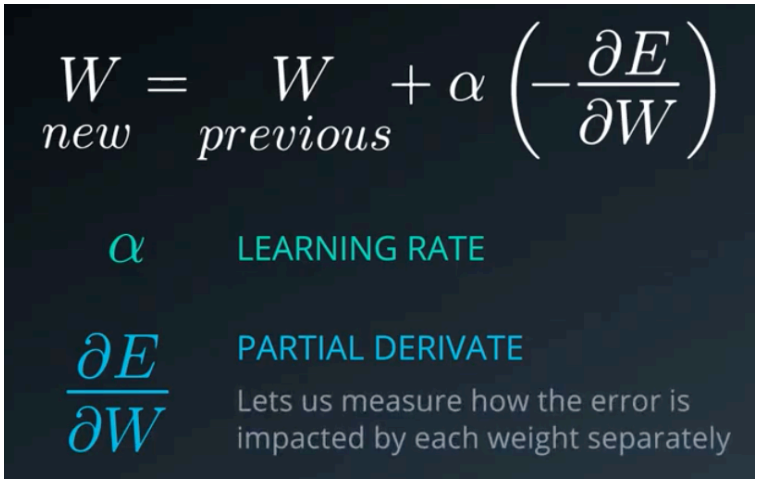
We use squared error, the loss function, $E = \frac{(y-\hat{y})^2}{2}$ to do backpropagation

In general, **Mean Squared Error (MSE)** is used in regression problems, whereas **Cross Entropy** is used in classification problems.

1.3.6 Back-propagation

Stochastic Gradient Descent with the Chain Rules

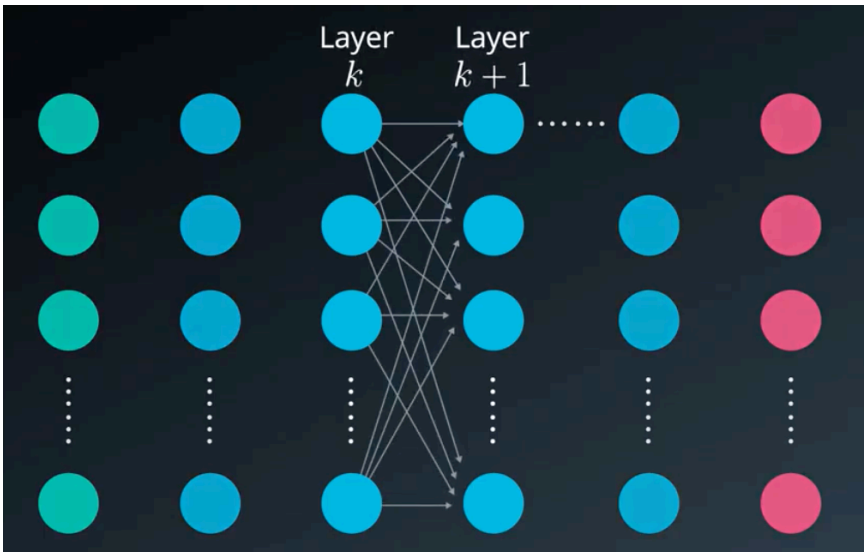
How to update the weights



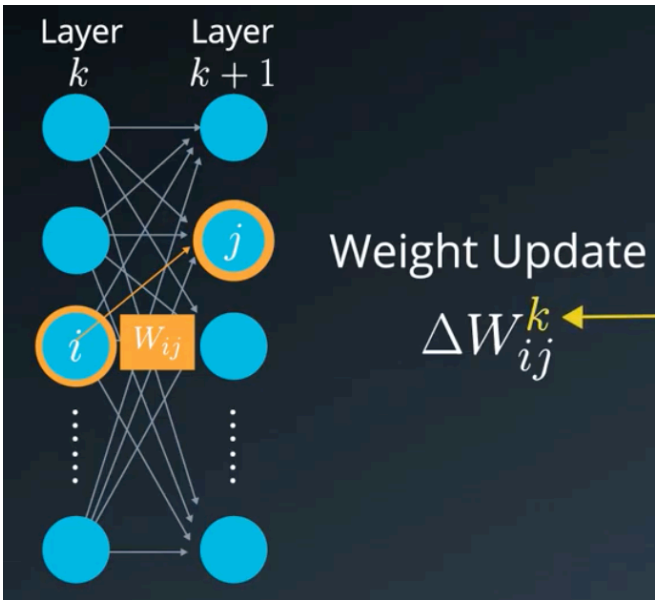
In another form:

The **inverted triangle symbol $\nabla_W\ (-E)$** indicates the vector of partial derivatives of the error E with respect to each of the weights.

Let's focus on the weight update between 2 certain layers:

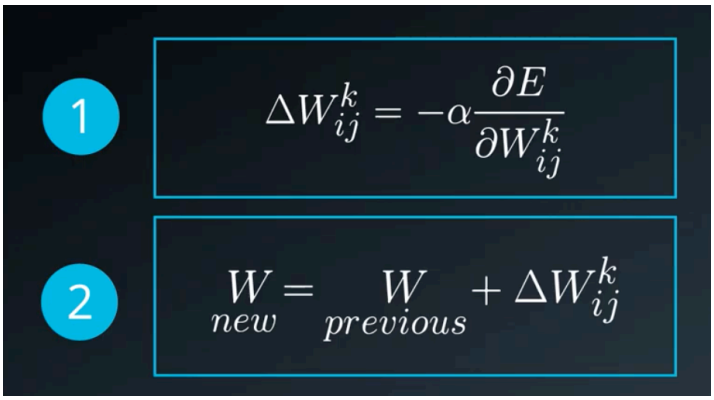


Let's see how doe we change the weight W_{ij}^k with ΔW_{ij}^k



The superscript k means that the weight is originated from layer k

How to calculate the ΔW_{ij}^k and update the weight ?



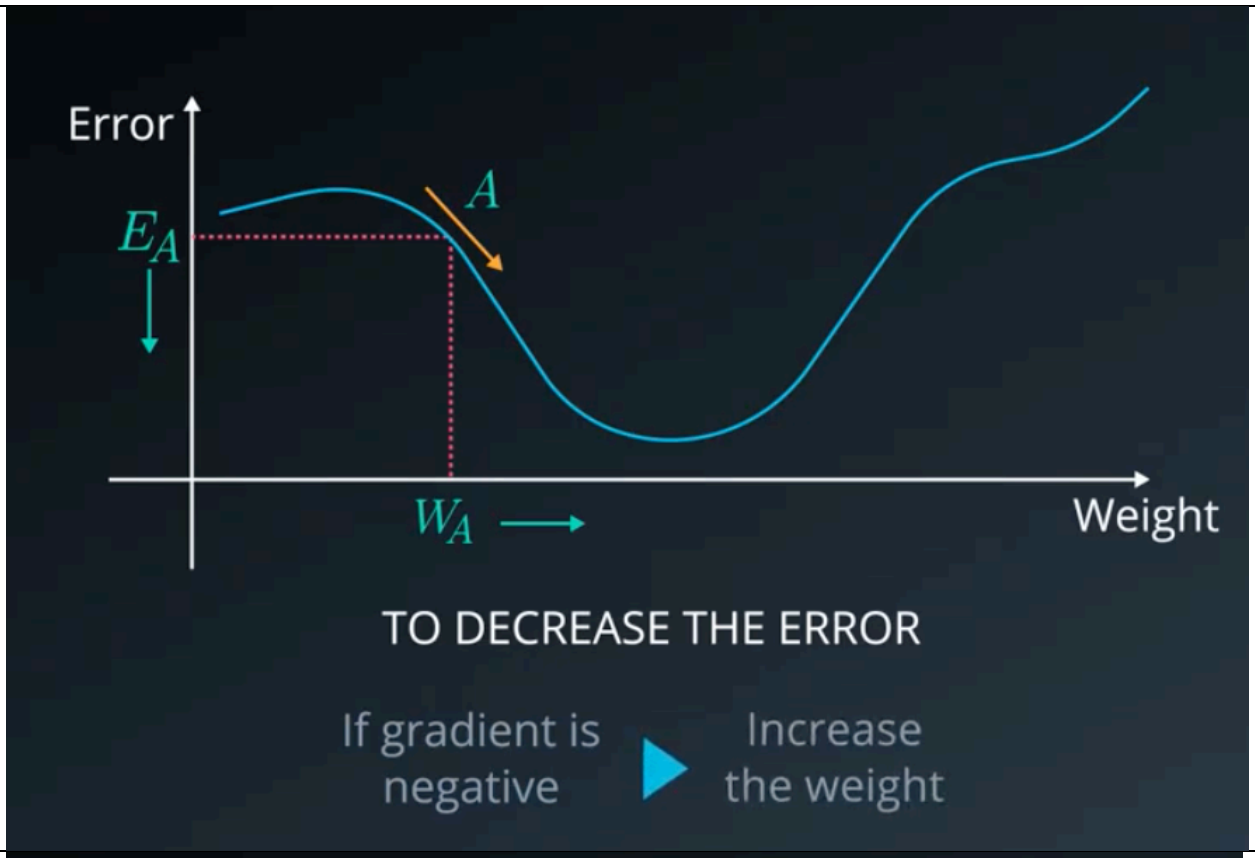
Why ΔW_{ij}^k is calculated through **the negative of its partial derivative** $\frac{\partial E}{\partial W_{ij}^k}$?

Because the partial derivative direction is the opposite direction towards the minimum in the space of loss function.

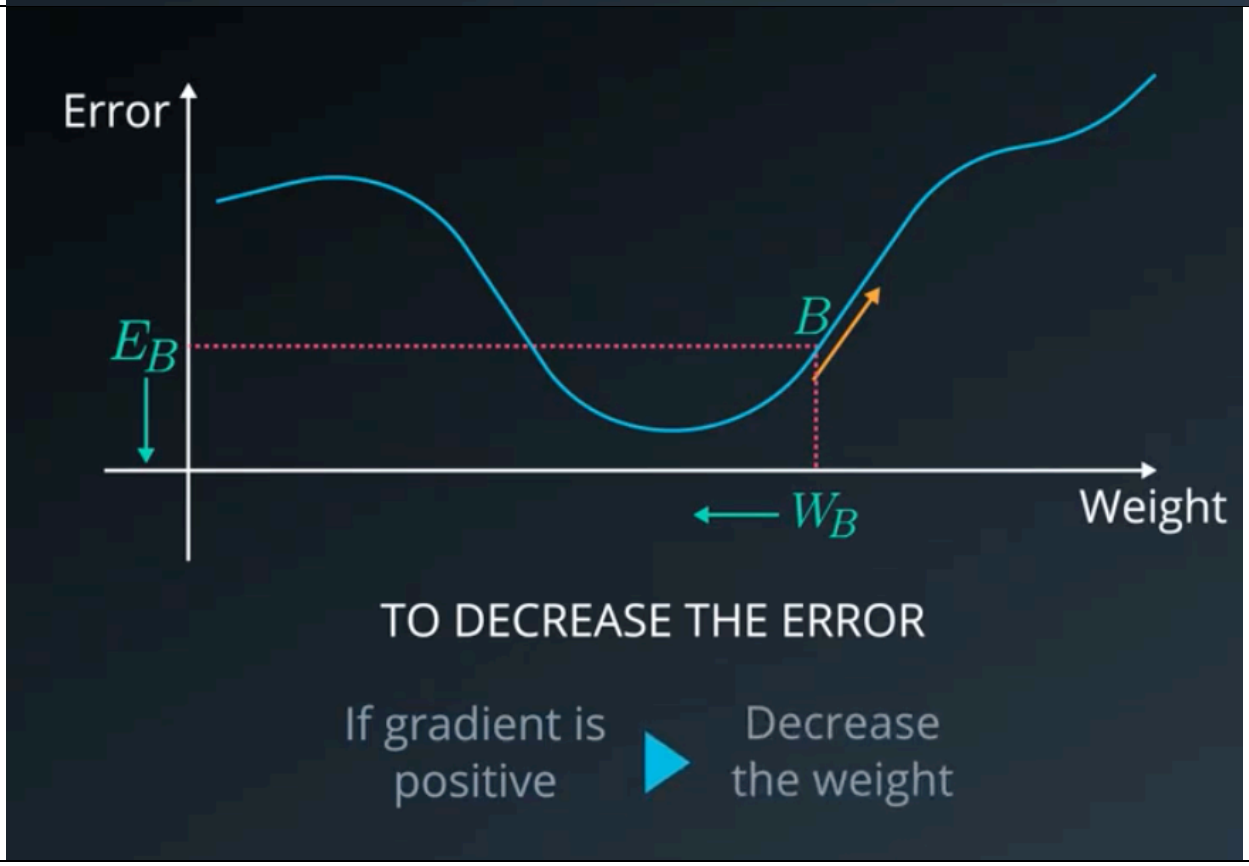
Then we can apply the Chain Rule to calculate $\frac{\partial E}{\partial W_{ij}^k}$

Let’s see in this example with a single weight and see why:

negative gradient (↘) on weight, but we need to **increase the weight** to **decrease the error**



positive gradient (↗) on weight, but we need to **decrase the weight** to **decrease the error**



How to tune the learning rate?

Resource 1: http://blog.datumbox.com/tuning-the-learning-rate-in-gradient-descent/

Resource 2: https://cs231n.github.io/neural-networks-3/#loss

1.3.7 Over-fitting

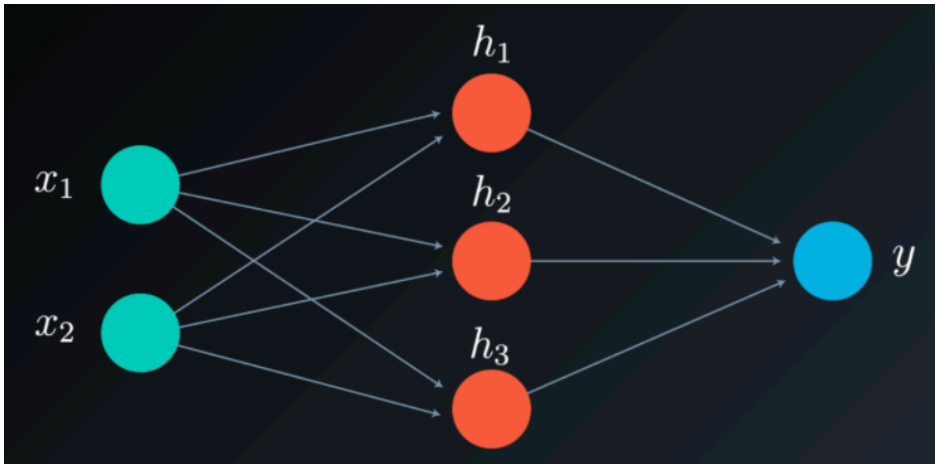
Overfitting is caused by the fact that we unintentionally model the noise or random elements in the training set.

Two main solutions to resolve it:

1) Early Stopping
   Stop training once the validation error stops decreasing
   Drawback of this approach: we end up with fewer samples to really train the model since we take some samples out to the validation set.

2) Regularization
   We impose a constraint on the training of the network such that it better generalizes
   Two ways to achieve regularization:
   a. Dropout
   b. Modify the loss function in the output layer
   c. Batch Norm
   d. …

1.3.8 An example to wrap up the Reminder

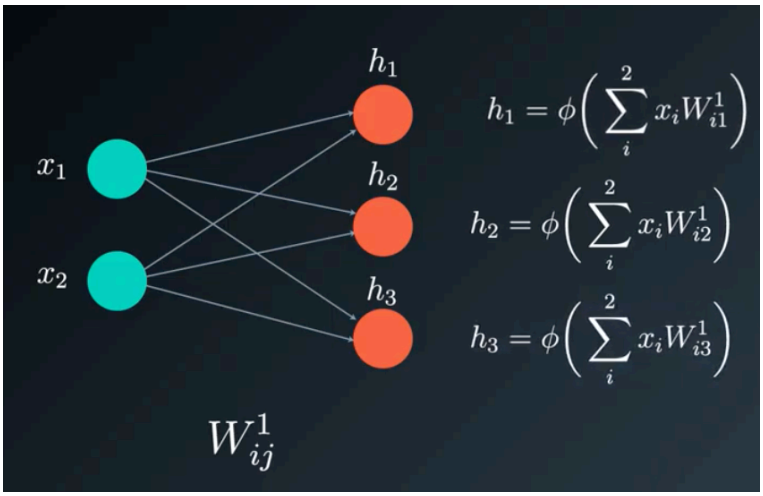


This is our network.

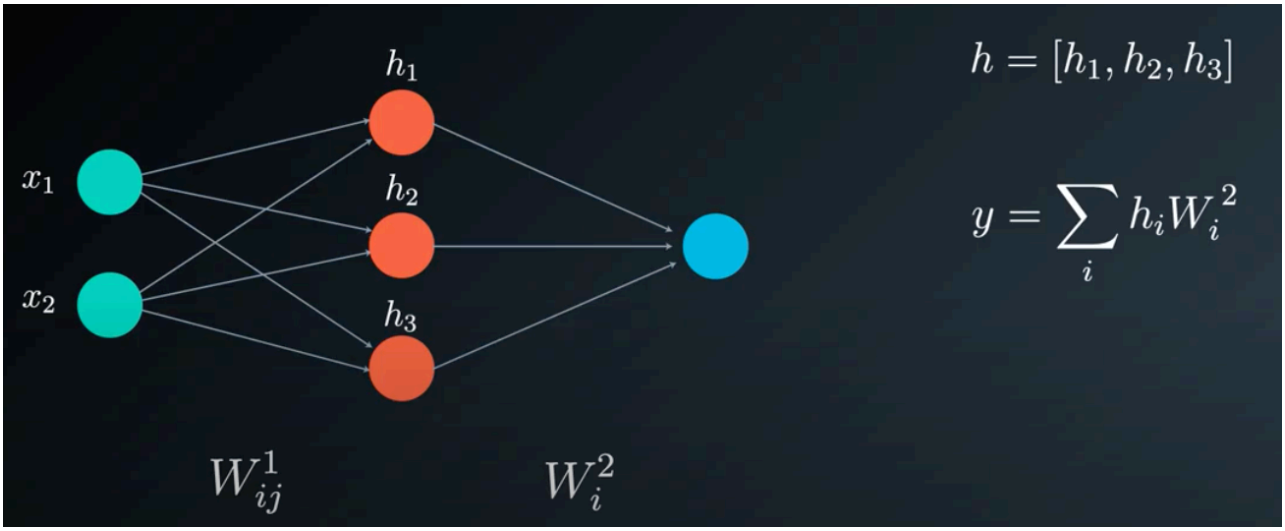
Notations:

- Input: $x = [x_1,\ x_2]$
- Weight:
  - Layer 1: $W^1 = \begin{bmatrix} W_{11}^1 & W_{12}^1 & W_{13}^1 \\ W_{21}^1 & W_{22}^1 & W_{23}^1 \end{bmatrix}$
  - Layer 2: $W^2 = \begin{bmatrix} W_{11}^2 \\ W_{21}^2 \\ W_{31}^2 \end{bmatrix}$

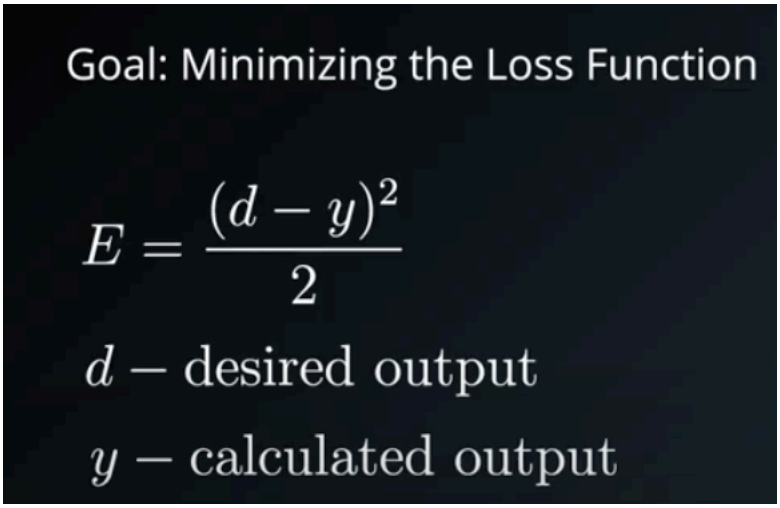
1) Feed-forward
   a. Calculate the activations in the hidden layer:



   b. Calculate the prediction



c. Calculate the error (loss function). Here we use MSE



2) Back-propagate

We need to minimize the error E by finding the updated values of ΔW_{ij}^k for every single layer k

$$\Delta W_{ij}^k = -\alpha \frac{\partial E}{\partial W_{ij}^k}$$

$$= -\alpha \frac{\partial \left(\frac{(d-y)^2}{2}\right)}{\partial W_{ij}^k}$$

$$= -\alpha (d-y) \frac{\partial (d-y)}{\partial W_{ij}^k}$$

$$= -\alpha (d-y) \left(\frac{\partial d}{\partial W_{ij}^k} - \frac{\partial y}{\partial W_{ij}^k}\right)$$

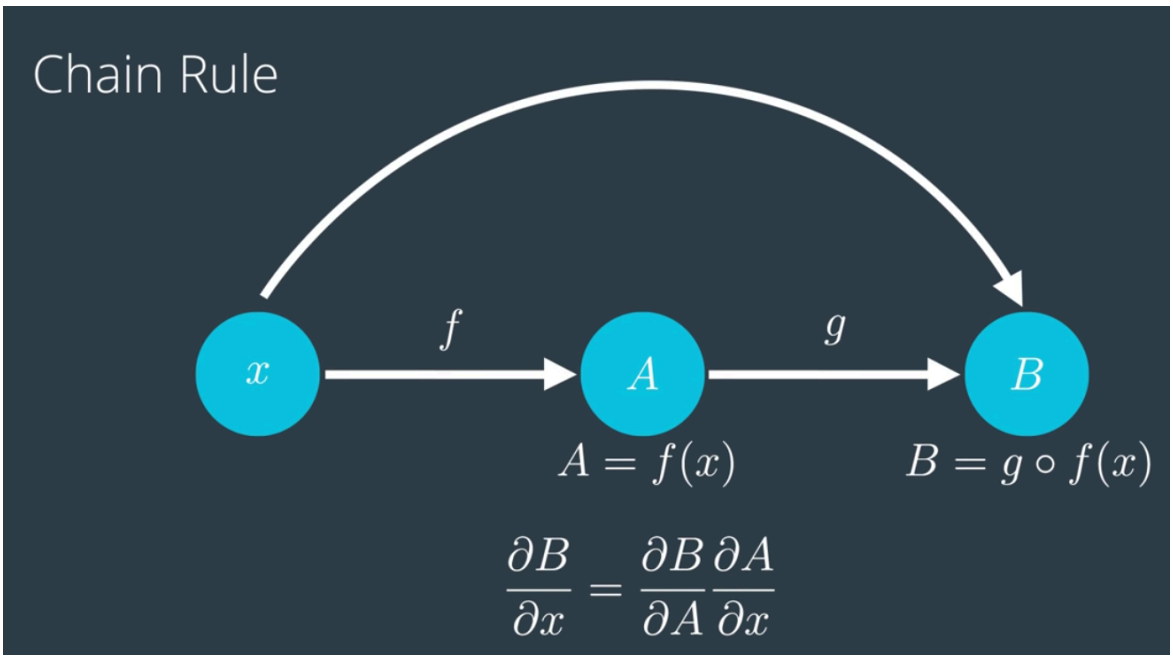
$$= -\alpha (d-y) \left(0 - \frac{\partial y}{\partial W_{ij}^k}\right) \quad since\ d\ is\ a\ constant, and\ only\ y\ relates\ with\ W_{ij}^k$$

$$= \alpha (d-y) \frac{\partial y}{\partial W_{ij}^k}$$

$$= \alpha (d-y)\, \delta_{ij}^k$$

We will calculate δ_{ij}^{k} using the **Chain Rule**

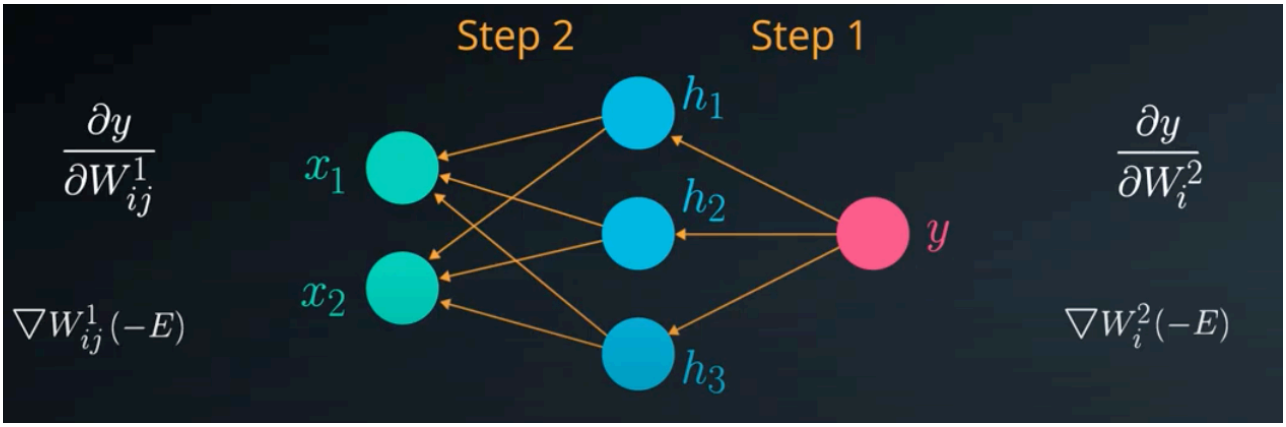
Recap: Chain Rule



There are 2 steps in backpropagation:

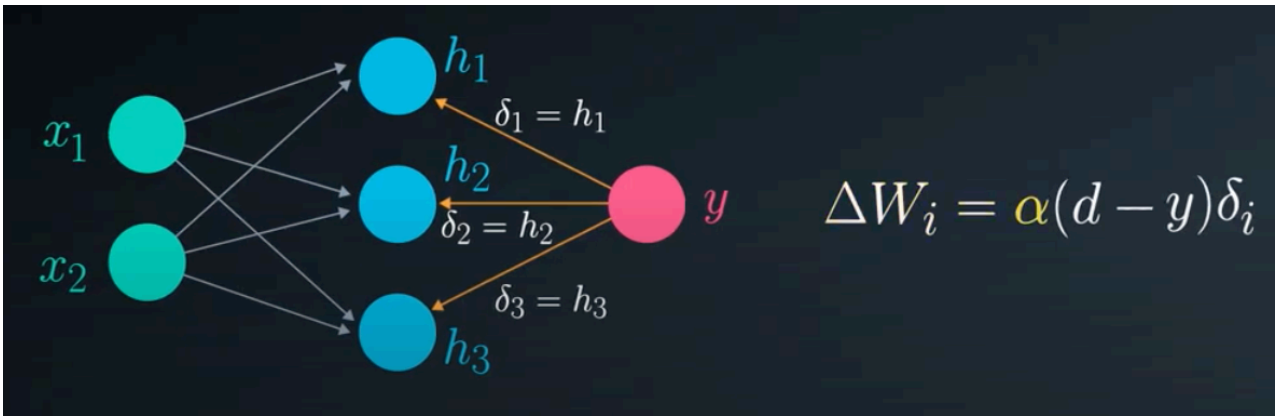
Step 1: calculate the gradient w.r.t. to W^2 from the output layer to hidden layer

Step 2: calculate the gradient w.r.t. to W^1 from the hidden layer to input layer

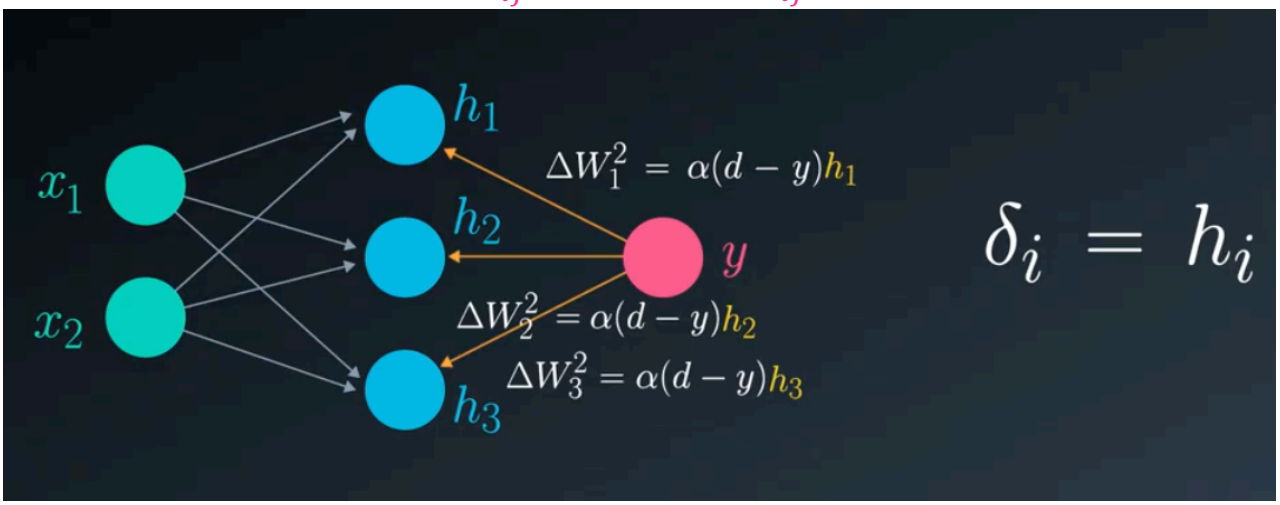


a. Step 1:

$$\frac{\partial y}{\partial W_{i1}^2} = \frac{\partial \left(\sum_i^3 h_i W_{i1}^2\right)}{\partial W_{i1}^2} = h_i = \delta_i$$



Given by δ_i, we can calculate $\Delta W_{ij}^k = \alpha\,(d-y)\,\delta_{ij}^k$



b. Step 2:

Here is where the chain rule comes in

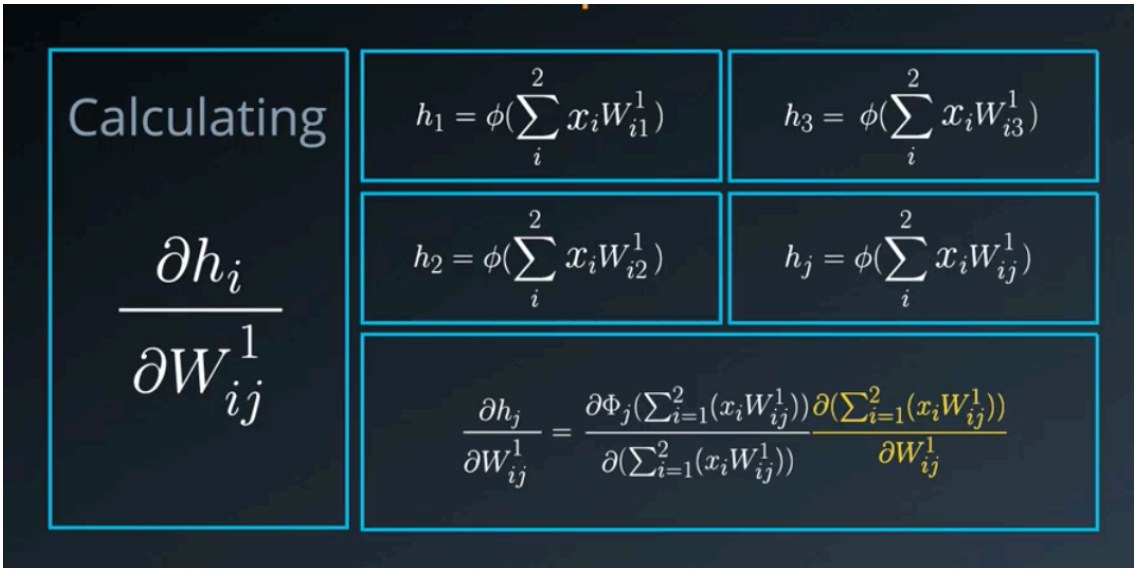
$$\delta_{ij} = \frac{\partial y}{\partial W_{ij}^1} = \sum_{p=1}^{N}\left(\frac{\partial y}{\partial h_p}\frac{\partial h_p}{\partial W_{ij}^1}\right) = \sum_{p=1}^{N}(Part\ 1 * Part\ 2)$$

Let's calculate each derivative separately:

a) Part 1

$$\frac{\partial y}{\partial h_j} = \frac{\partial\left(\sum_{i=1}^{3}(h_i\,W_i^2)\right)}{\partial h_j} = W_j^2$$

b) Part 2

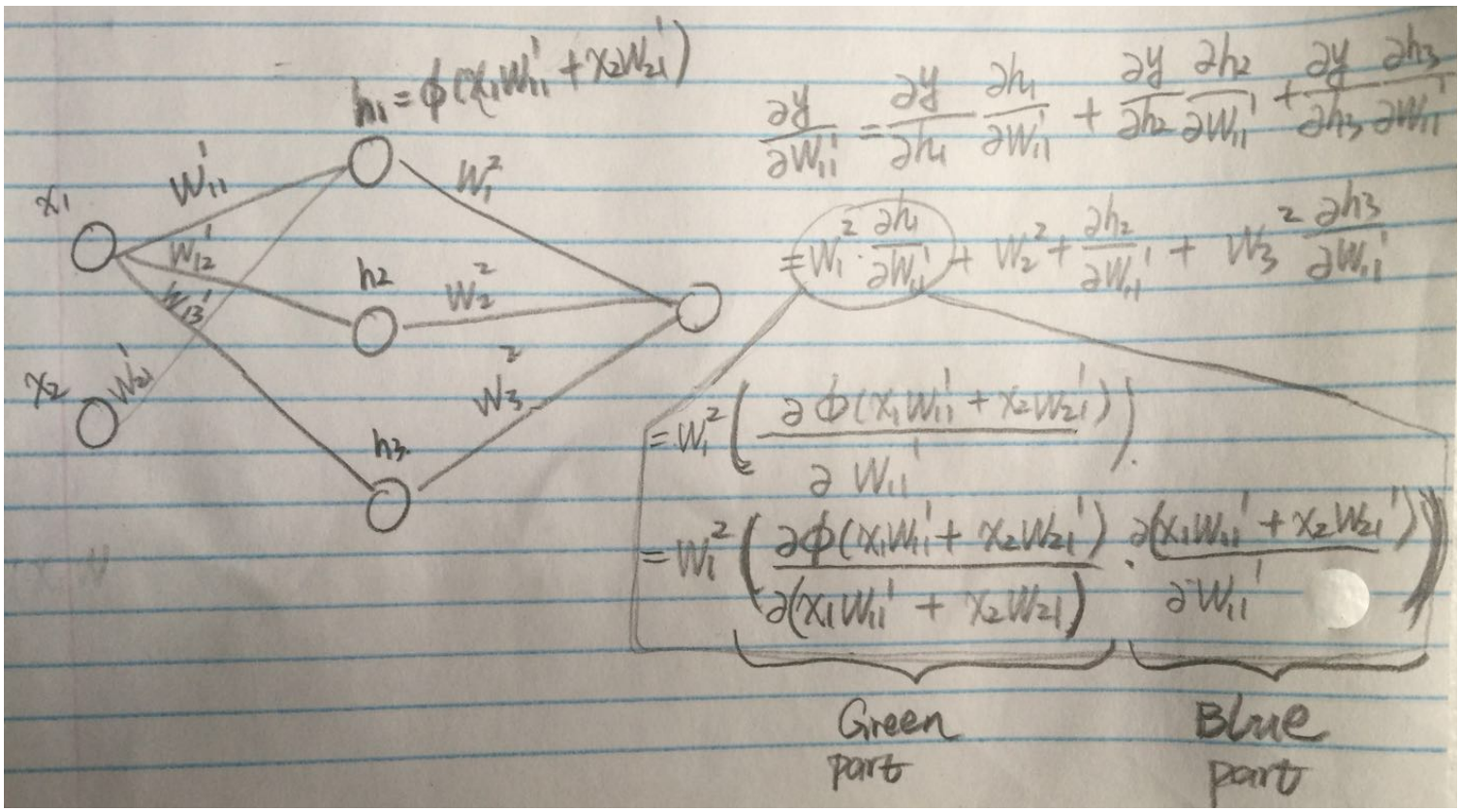


$$\frac{\partial h_j}{\partial W_{ij}^1} = \frac{\partial \Phi_j\left(\sum_{i=1}^2 (x_i W_{ij}^1)\right)}{\partial\left(\sum_{i=1}^2 x_i W_{ij}^1\right)} \frac{\partial\left(\sum_{i=1}^2 x_i W_{ij}^1\right)}{\partial W_{ij}^1}$$

$\frac{\partial \Phi_j\left(\sum_{i=1}^2 (x_i W_{ij}^1)\right)}{\partial\left(\sum_{i=1}^2 x_i W_{ij}^1\right)}$ *is the partial derivative of the activation function* Φ

$\frac{\partial\left(\sum_{i=1}^2 x_i W_{ij}^1\right)}{\partial W_{ij}^1}$ *is the partial deriavtive of the linear combination* $w.r.t. W_{ij}^1$

Here is my derivation note for calculating $\frac{\partial h_1}{\partial W_{11}^1}$



Let's introduce a few shortcut in the formula

$$\frac{\partial h_j}{\partial W_{ij}^1} = \frac{\partial \Phi_j\left(\sum_{i=1}^2 (x_i W_{ij}^1)\right)}{\partial\left(\sum_{i=1}^2 x_i W_{ij}^1\right)} \frac{\partial\left(\sum_{i=1}^2 x_i W_{ij}^1\right)}{\partial W_{ij}^1}$$

Let's

$$\phi_j' = \frac{\partial \Phi_j\left(\sum_{i=1}^2 (x_i W_{ij}^1)\right)}{\partial\left(\sum_{i=1}^2 x_i W_{ij}^1\right)}$$

$$x_i = \frac{\partial\left(\sum_{i=1}^2 x_i W_{ij}^1\right)}{\partial W_{ij}^1}$$

$$\frac{\partial h_j}{\partial W_{ij}^1} = \frac{\partial \Phi_j(\sum_{i=1}^2 (x_i W_{ij}^1))}{\partial(\sum_{i=1}^2 (x_i W_{ij}^1))} \frac{\partial(\sum_{i=1}^2 (x_i W_{ij}^1))}{\partial W_{ij}^1} \qquad \frac{\partial h_j}{\partial W_{ij}^1} = \Phi_j' x_i$$

Φ_j' x_i

Then

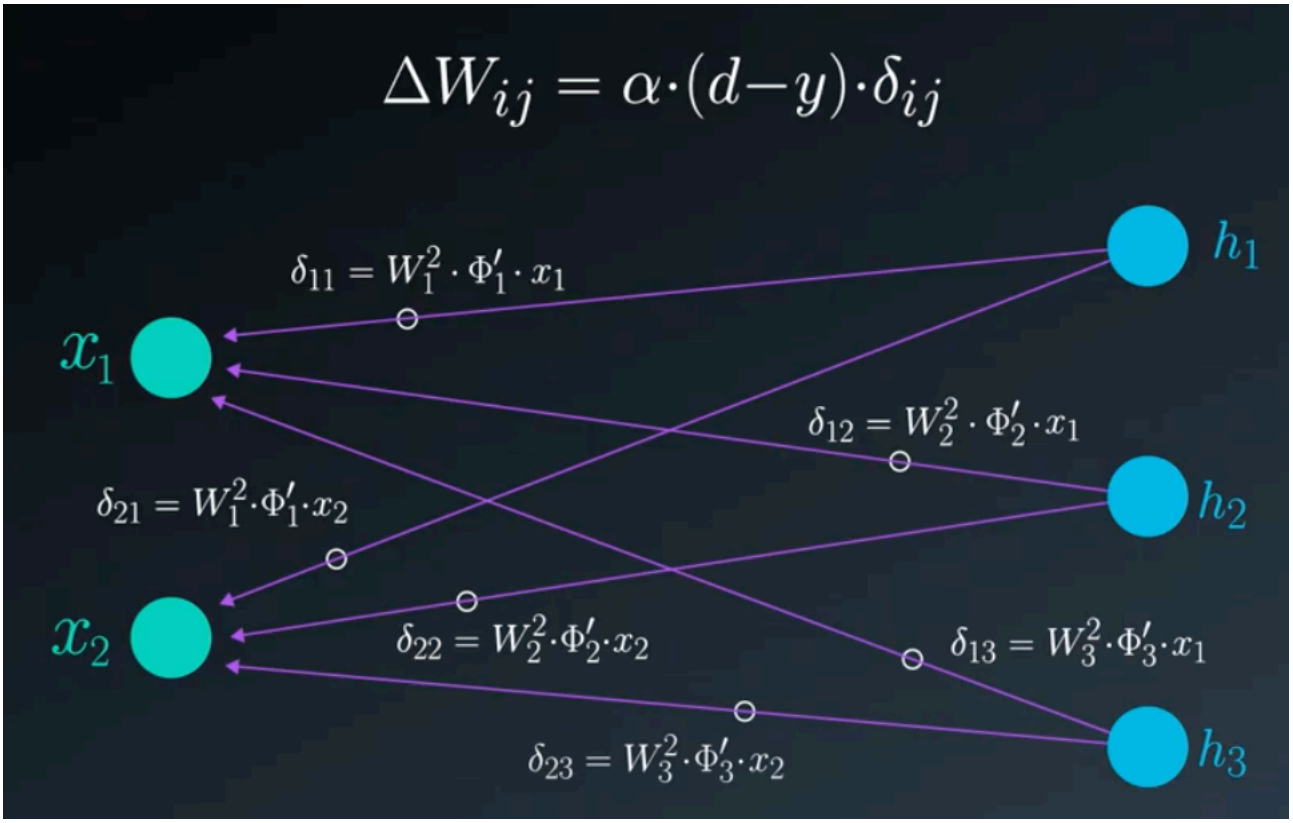
$$\boldsymbol{\frac{\partial h_j}{\partial W_{ij}^1} = \phi_j' x_i}$$

So far, we have gotten the Part 1 and Part 2 in the formula:

$$\frac{\partial y}{\partial W_{ij}^1} = \sum_{p=1}^{N} \left(\frac{\partial y}{\partial h_p} \frac{\partial h_p}{\partial W_{ij}^1}\right) = \sum_{p=1}^{N} (Part\ 1 * Part\ 2)$$

Let's integrate them together:

$$\delta_{ij} = \frac{\partial y}{\partial W_{ij}^1} = \sum_{p=1}^{N} \left(\frac{\partial y}{\partial h_p} \frac{\partial h_p}{\partial W_{ij}^1}\right) = W_j^2 \cdot \boldsymbol{\phi_j'} \cdot \boldsymbol{x_i}$$



Then, we can calculate the weight update values for W^1:

$$\Delta W_{ij}^1 = \alpha\ (d - y)\ \delta_{ij}^1$$

Finally, we can update the weights W^1 using ΔW_{ij}^1

$$W_{new}^1 = W_{previous}^1 + \Delta W_{ij}^1$$

In vanilla network, we still need to **consider the bias term at eacy layer**, but that won't change much to our process above.

1.3.9 Different training strategies

1) Mini-batch training with Gradient Descent: update the weight once every N steps
   Only change is

$$\delta = \frac{1}{N} \sum_{k}^{N} \delta_{ij_k}$$

   Advantages:

- Reduce the complexity of training process (fewer computation required)
- Reduce noise, and learning process converges faster and more accurate

1.4 Recurrent Neural Network (RNN)

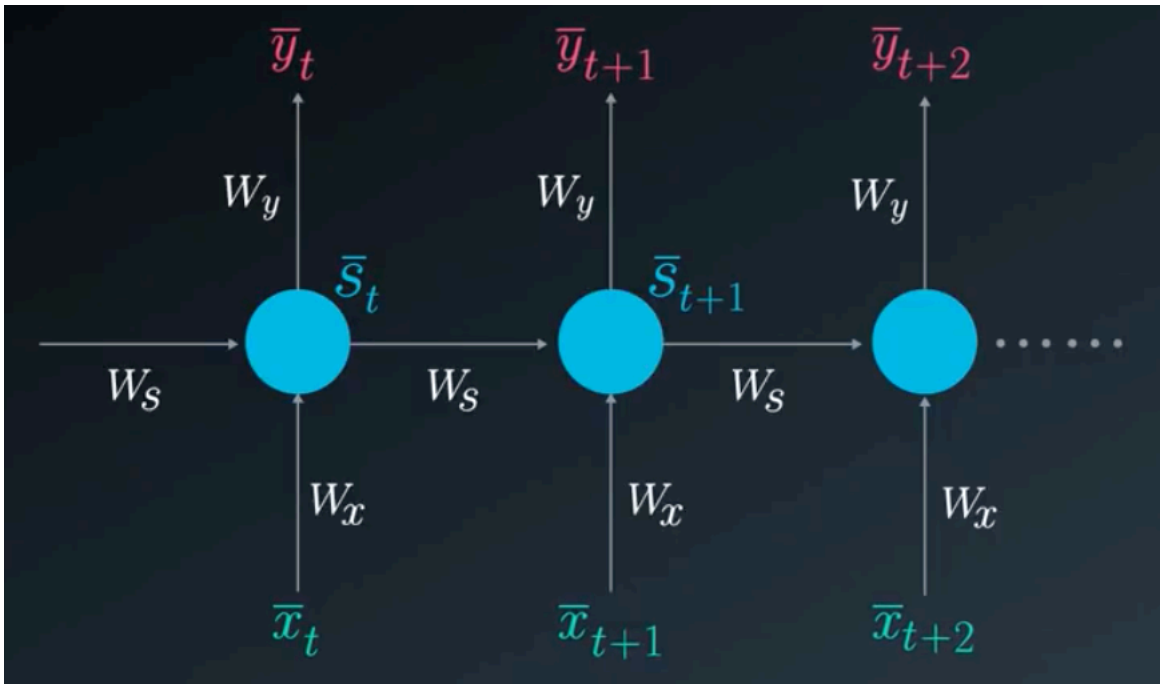
RNNs capture information of previous inputs by maintaining internal memory elements, knowns as “states”

Temproral Dependencies

Mean: current output depends not only on the current input, but also on a memory elements which takes into account past inputs

Difference between RNNs and vanilla FFNNs

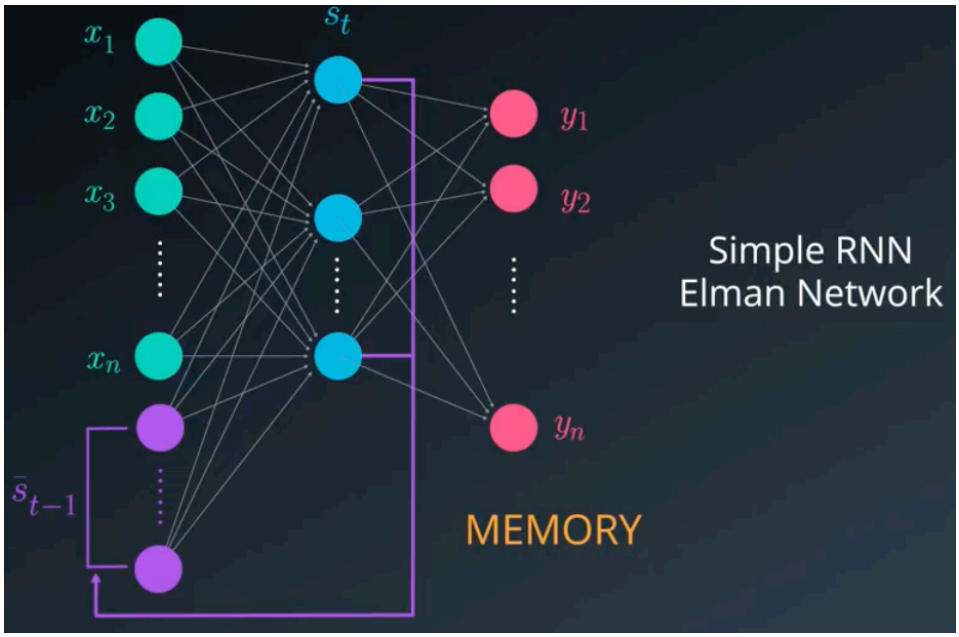
1) We train the RNN using **sequences of inputs** with different timesteps



2) Current input and activations of neurons, those memory elements, serve as inputs to the next time step

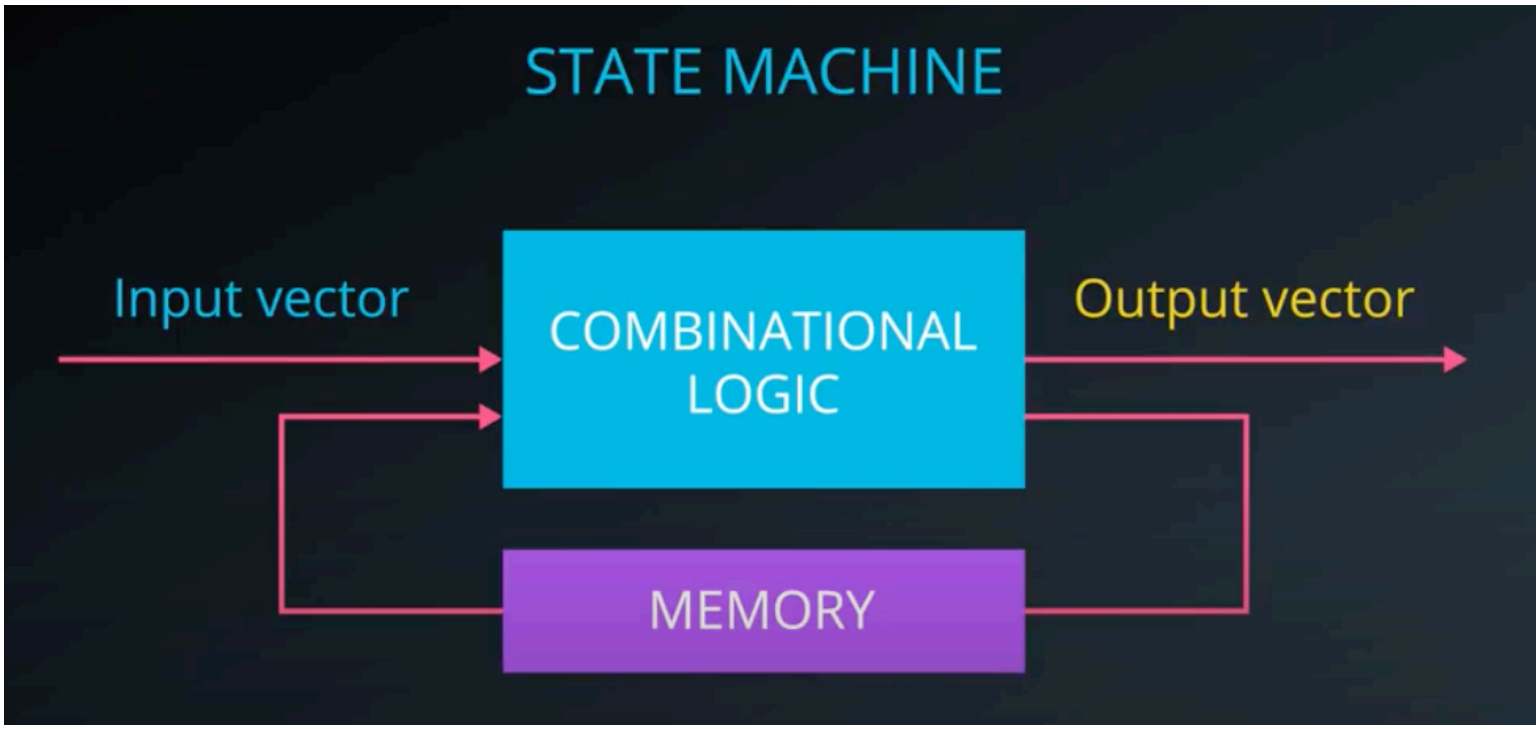
In RNNs, hidden units are called states s.

Simple RNN / Elman Network

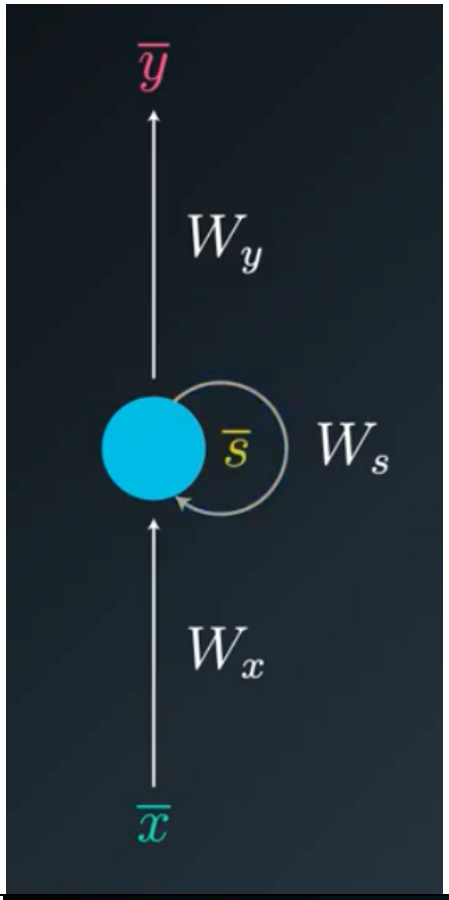


State Machine with logic and memory:

The output depends on not only the external inputs, but also on previous inputs which come from memory cells.



Folded RNN



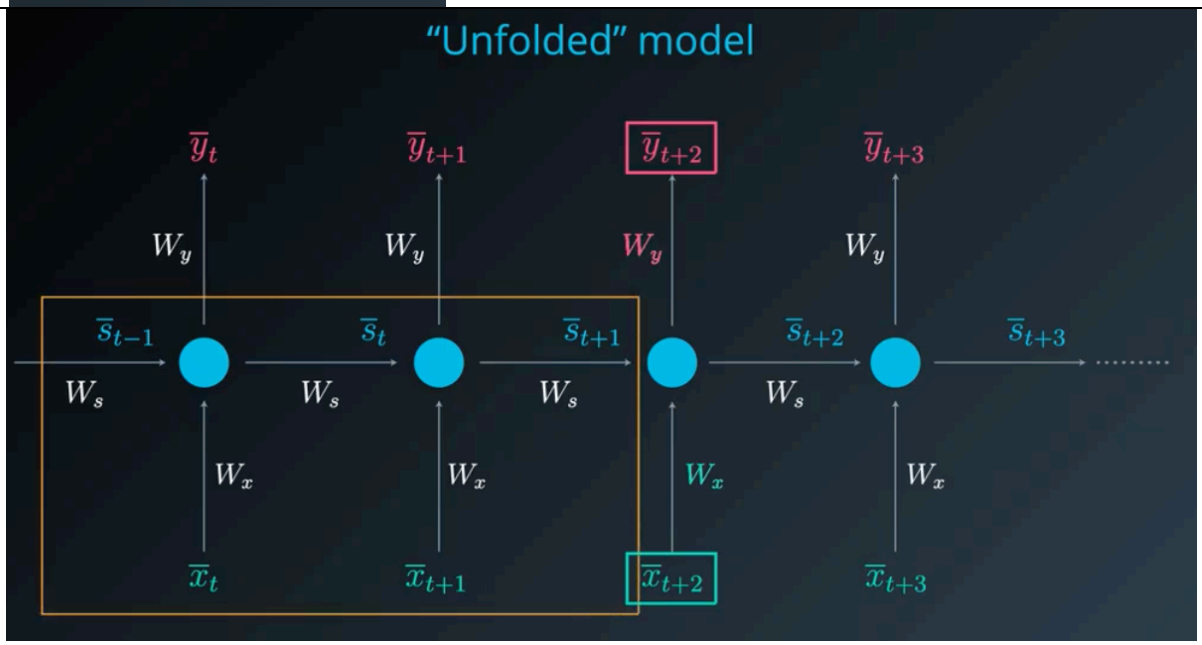
W_x: weights between input layer and state layer

W_y: weights between state layer and output

W_s: weights connecting the state from the previous timestep to the state in the next timestep

$\bar{s}$: state

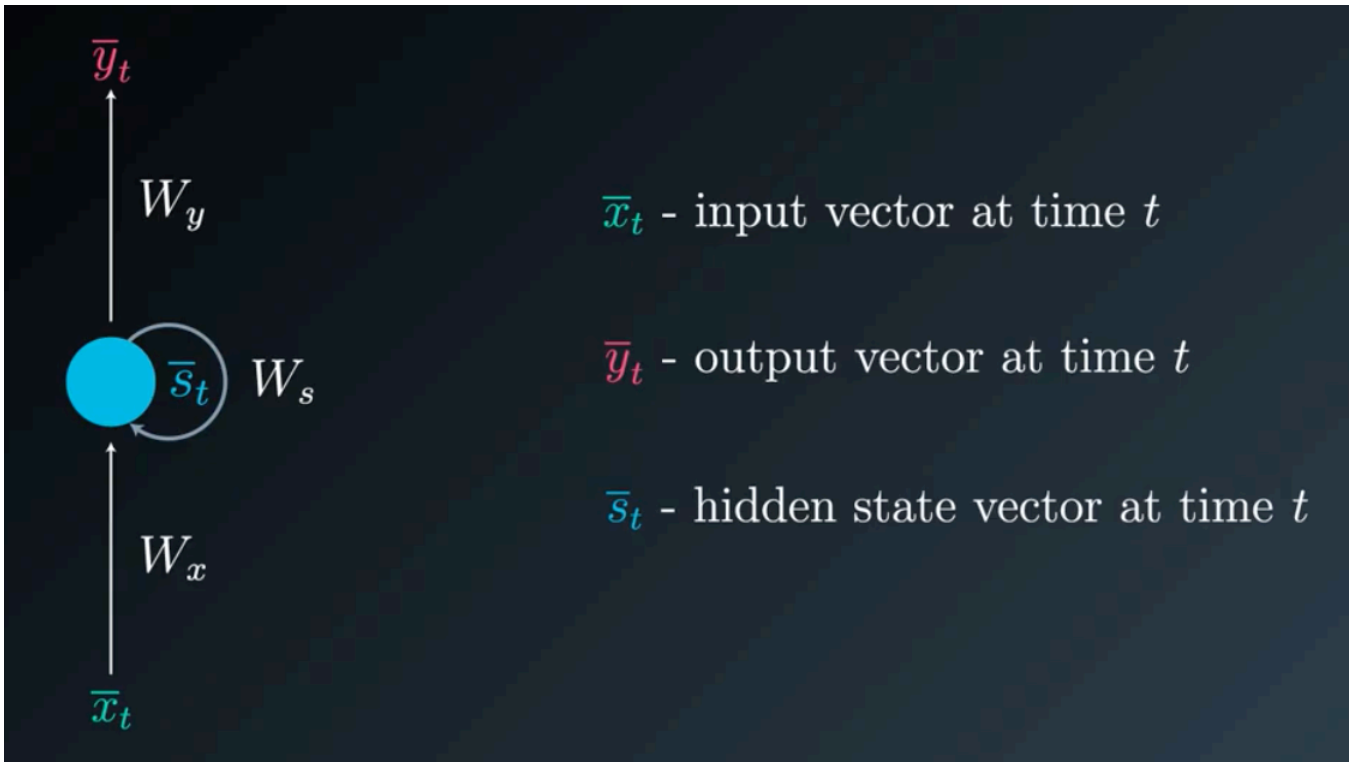
Un-folded RNN



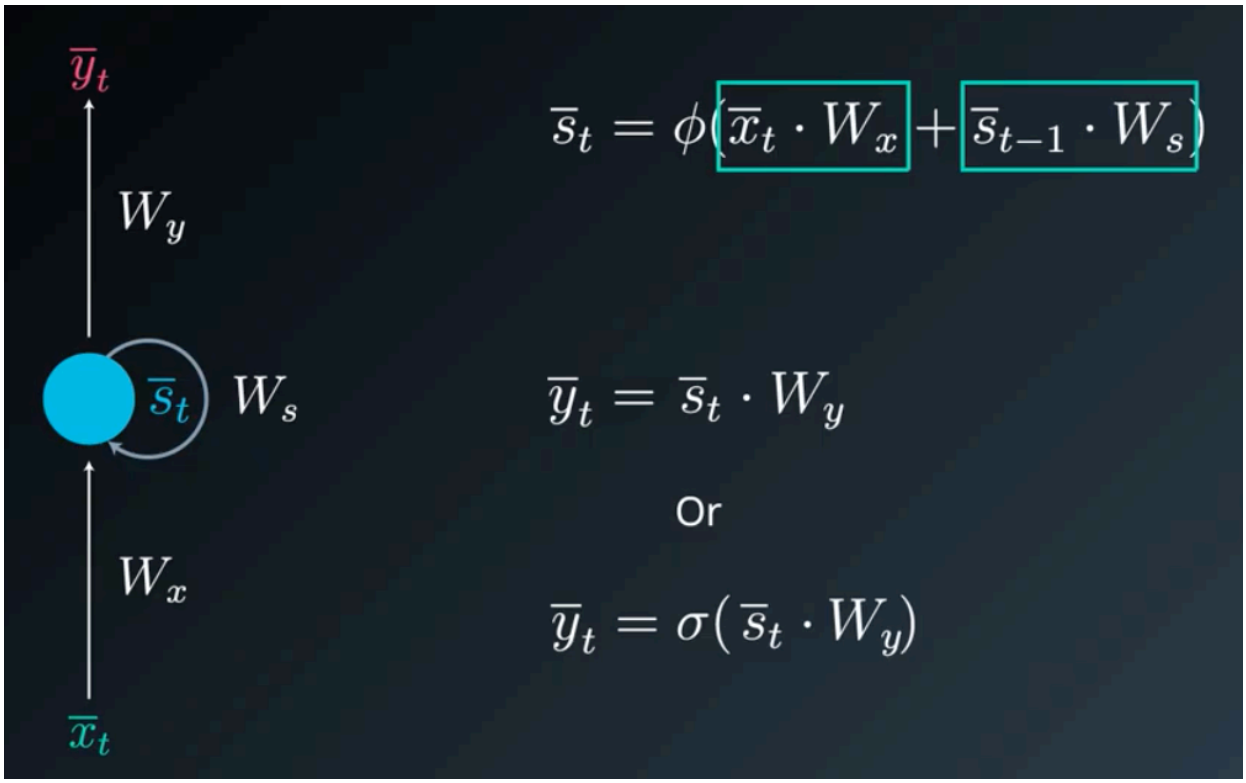
output $\bar{y}_{t+2}$ at timestep t+2 depends on input $\bar{x}_{t+2}$ at timestep t+2 and previous inputs/states

$$\bar{y}_{t+2} = \phi\left(\bar{x}_{t+2}, \bar{x}_{t+1}, \bar{x}_t, \bar{x}_{t-1}, \ldots, \bar{x}_{t-t_0}, W_x, W_y\right) \; where \; t_0 + 2 \; is \; memory \; size$$

Notations:

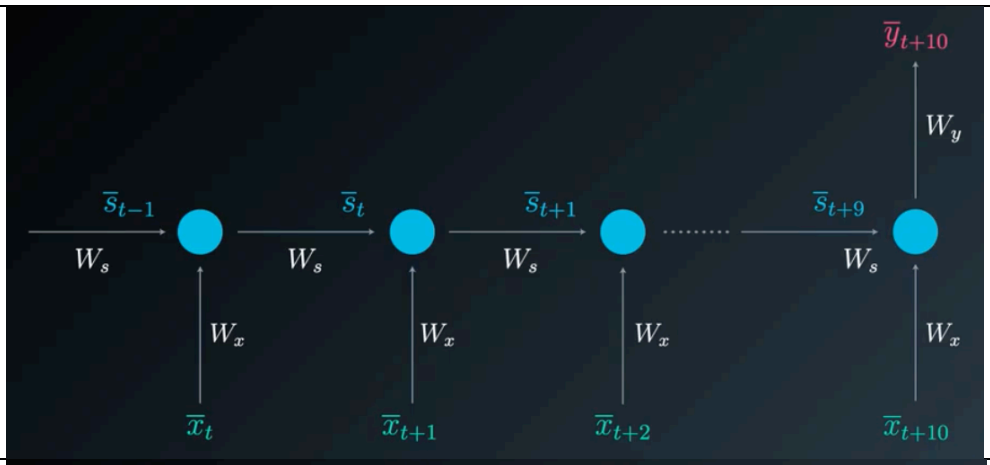
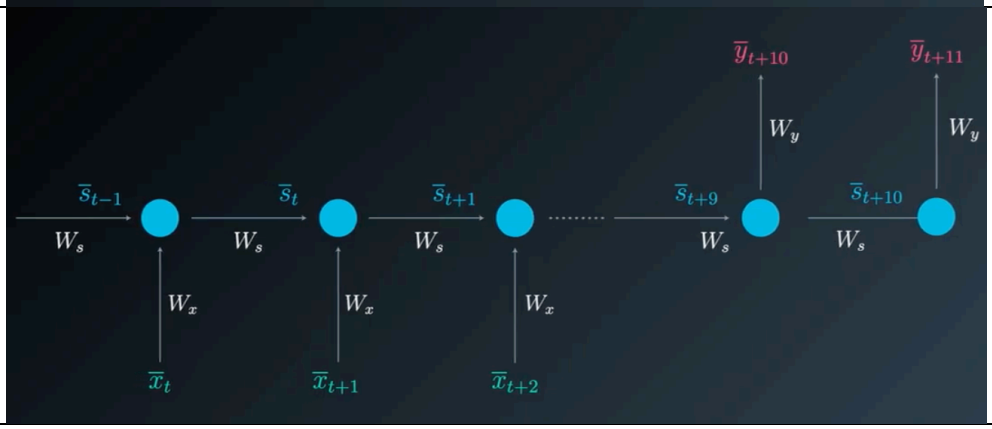


Feed-forward in RNN:



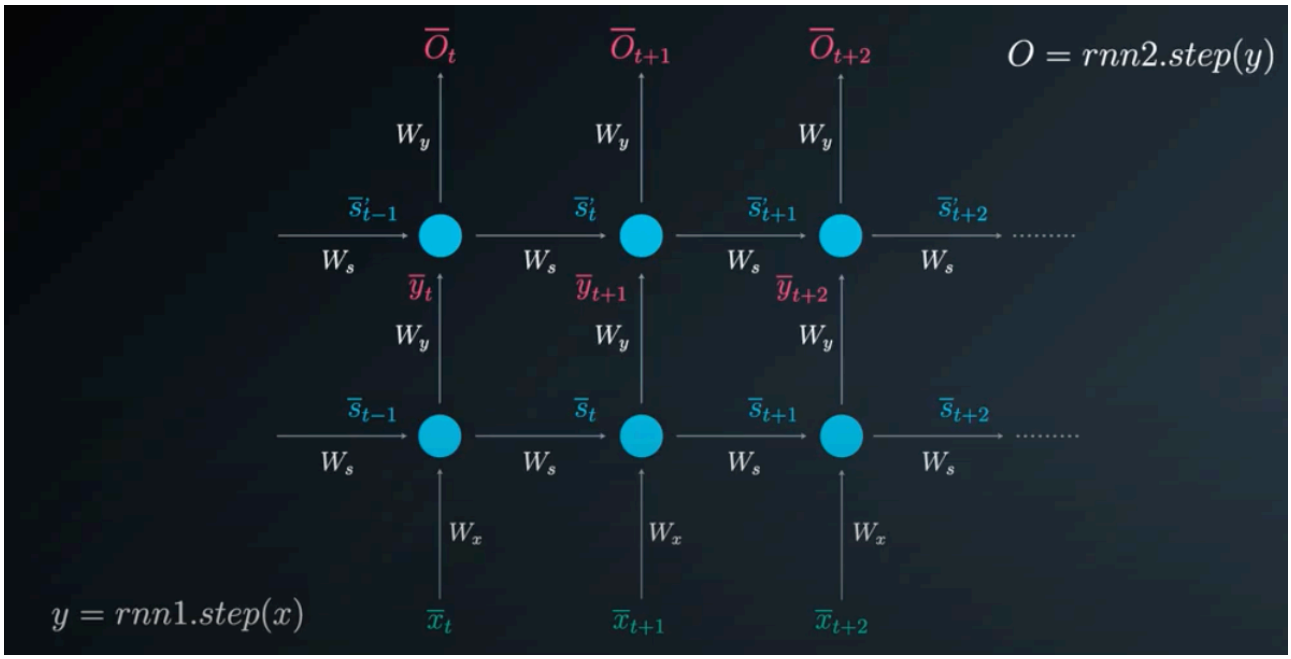
The state layer s_t depends on the current input $\bar{x}_t$, the weight matrices (W_x, W_s), activation function, and the previous state $\bar{s}_{t-1}$

Different architectures in RNNs:

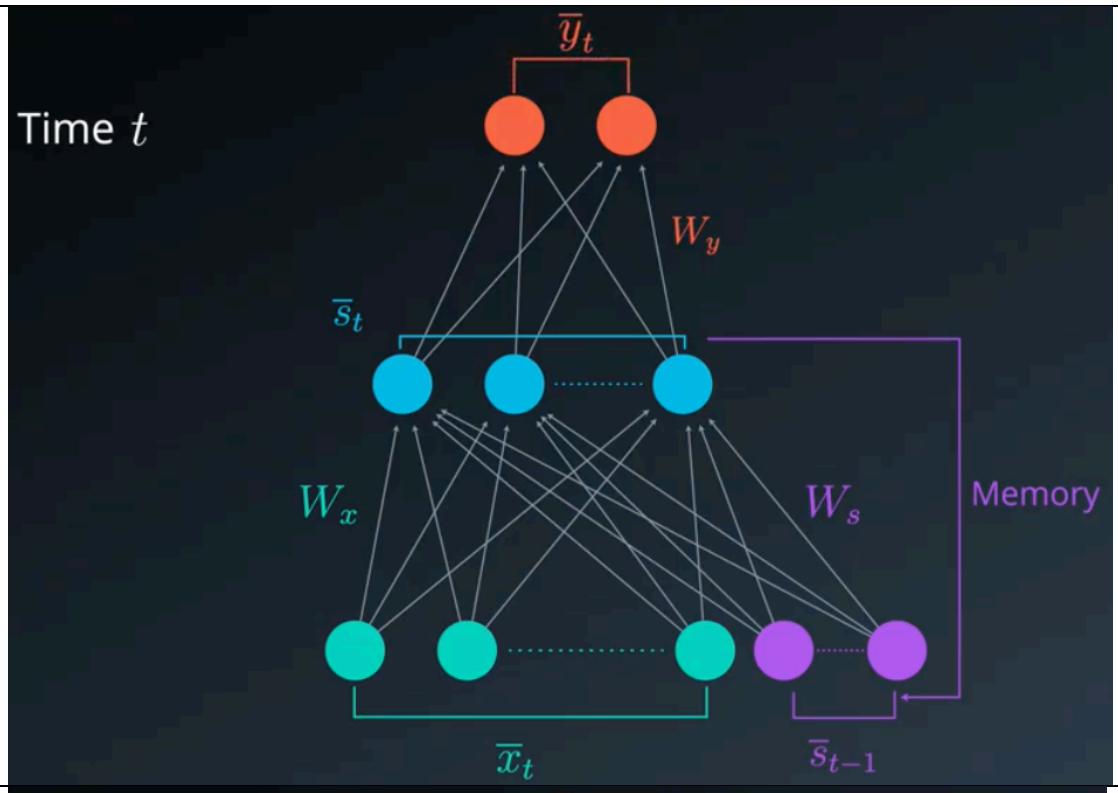
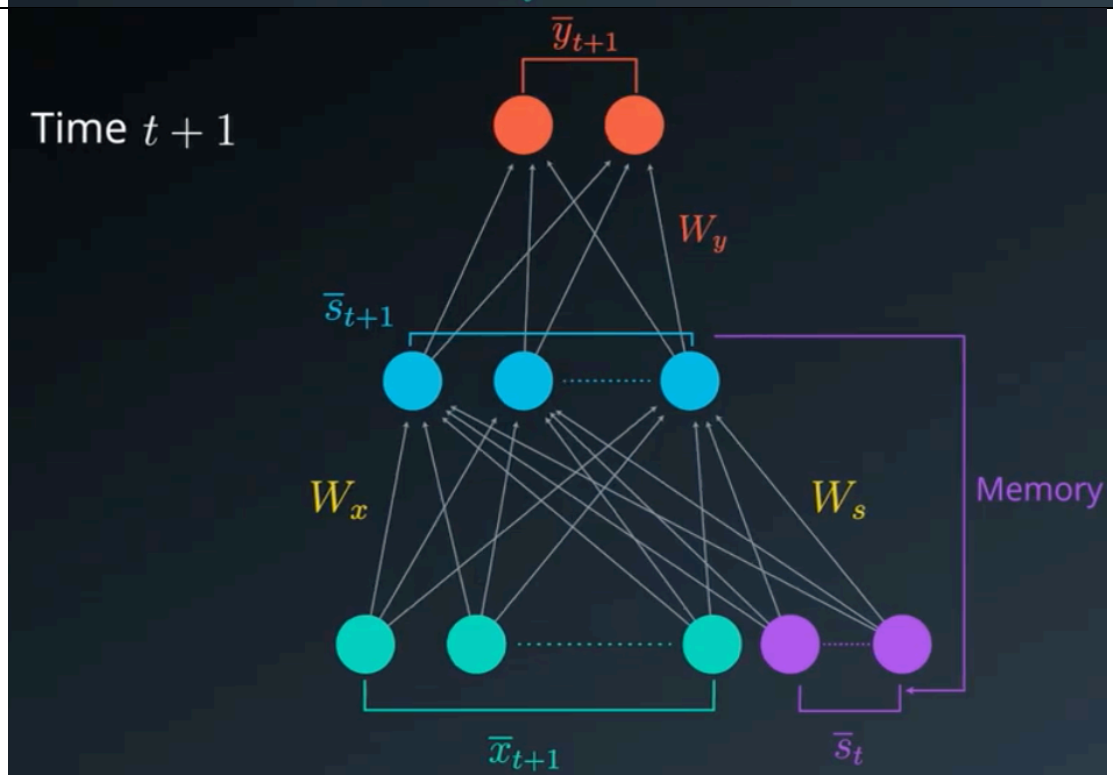
Many-to-One		
Many-to-Many		

Stacked RNN

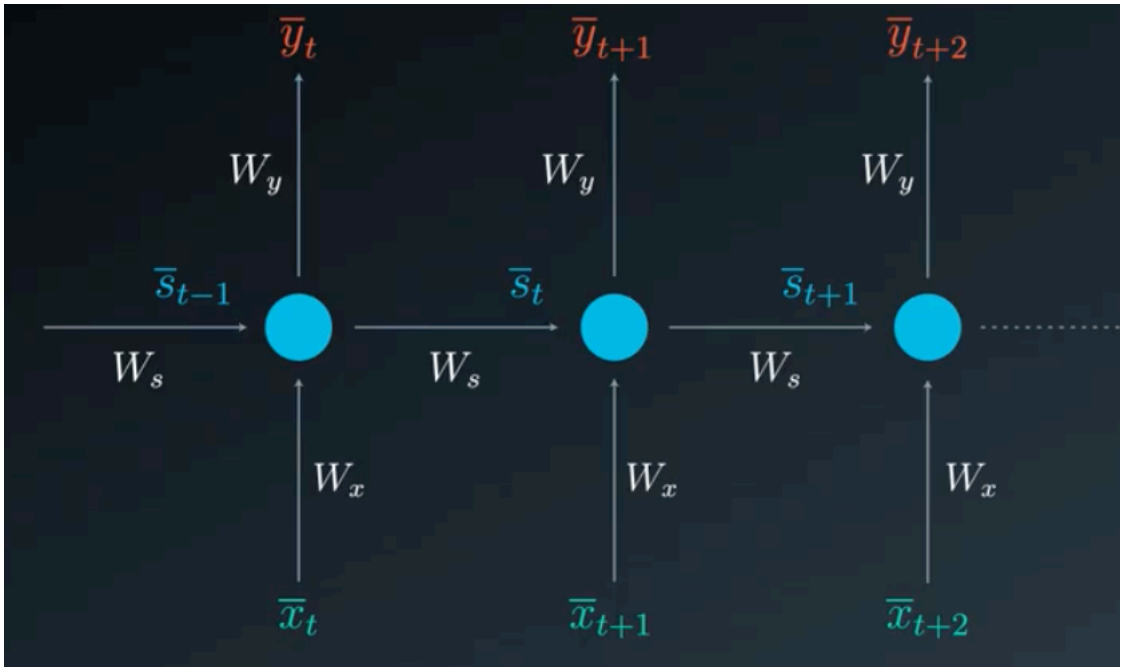
Vector $y = [\bar{y}_t, \bar{y}_{t+1}, \bar{y}_{t+2}, \dots]$ feeds to the next layer which output $\bar{O}$



Recap: Folded model

At time t	
At time t+1	 Changes: • Input: $\bar{x}_t \rightarrow \bar{x}_{t+1}$ • Memory: $\bar{s}_{t-1} \rightarrow \bar{s}_t$ • State: $\bar{s}_t \rightarrow \bar{s}_{t+1}$ • Prediction: $\bar{y}_t \rightarrow \bar{y}_{t+1}$

Recap: Un-folded model

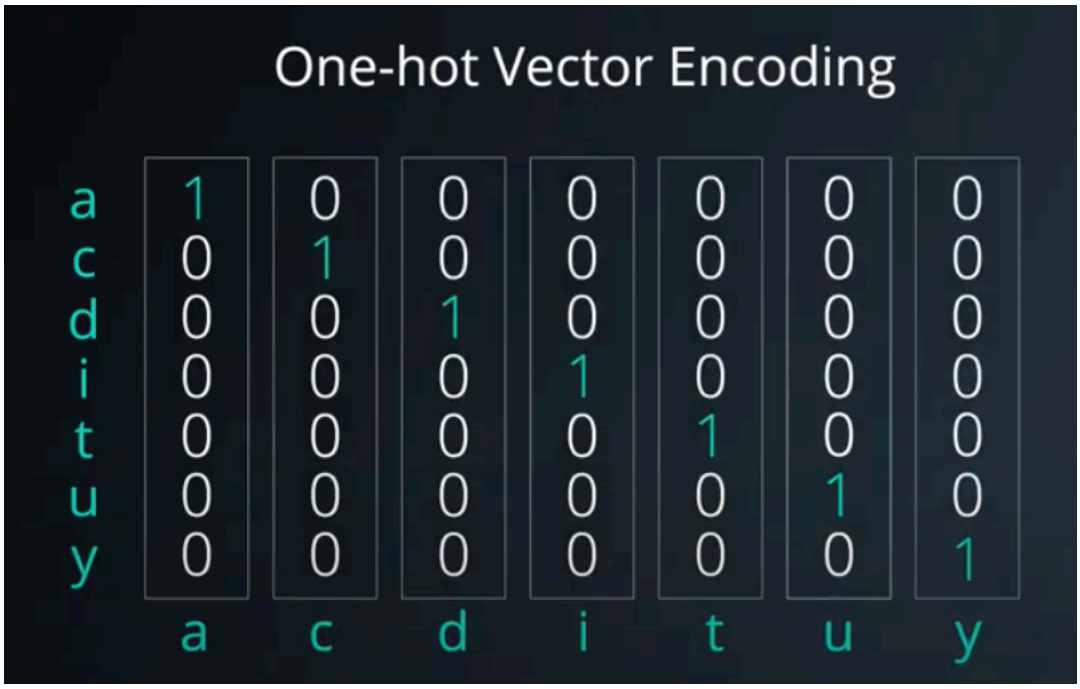


Sequence Detecction in RNN

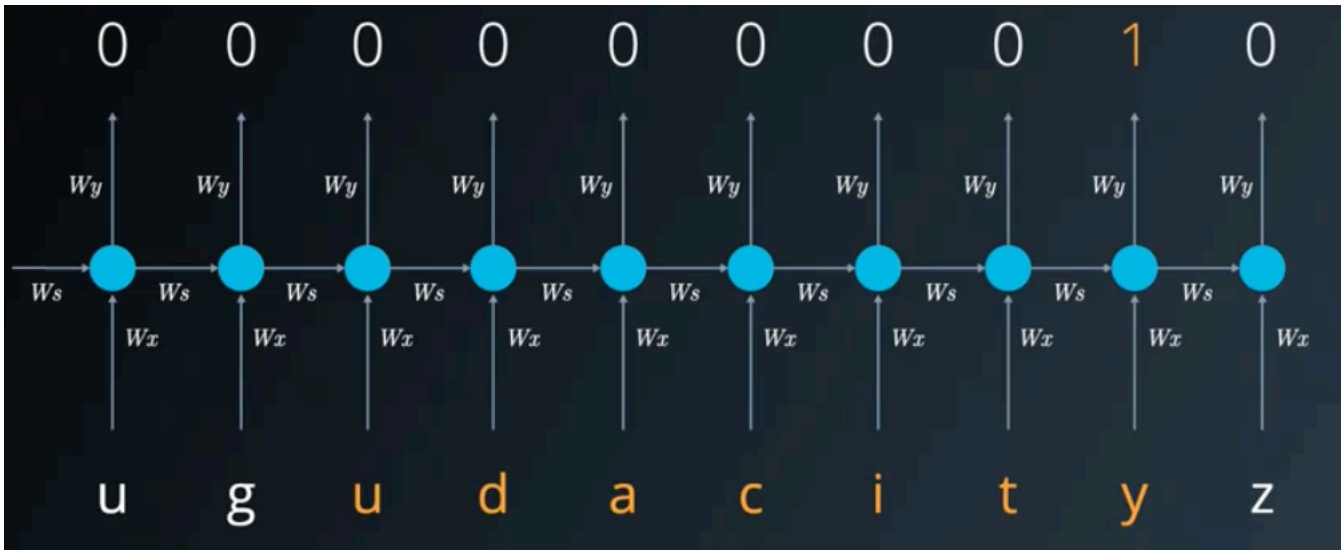
Idea: detect a specific pattern of inputs to RNN

Goal: detect the word “Udacity” in inputs

Step 1: Convert letter to One-hot vector encoding

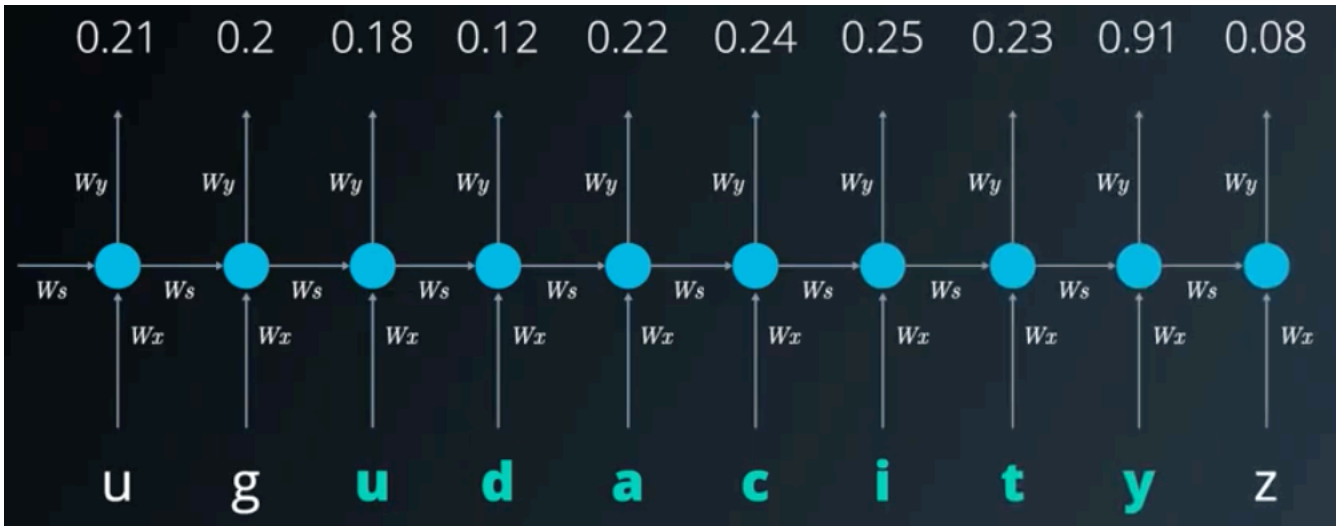


Step 2: Feed random sequence of letters to network, output is 1 if and only if the last letter “y” detected



The first state vector is set to 0. As training, we set the target to 0 if “Udacity” is not detected and set to 1 if “udacity” is detected.

Here’s a possible output



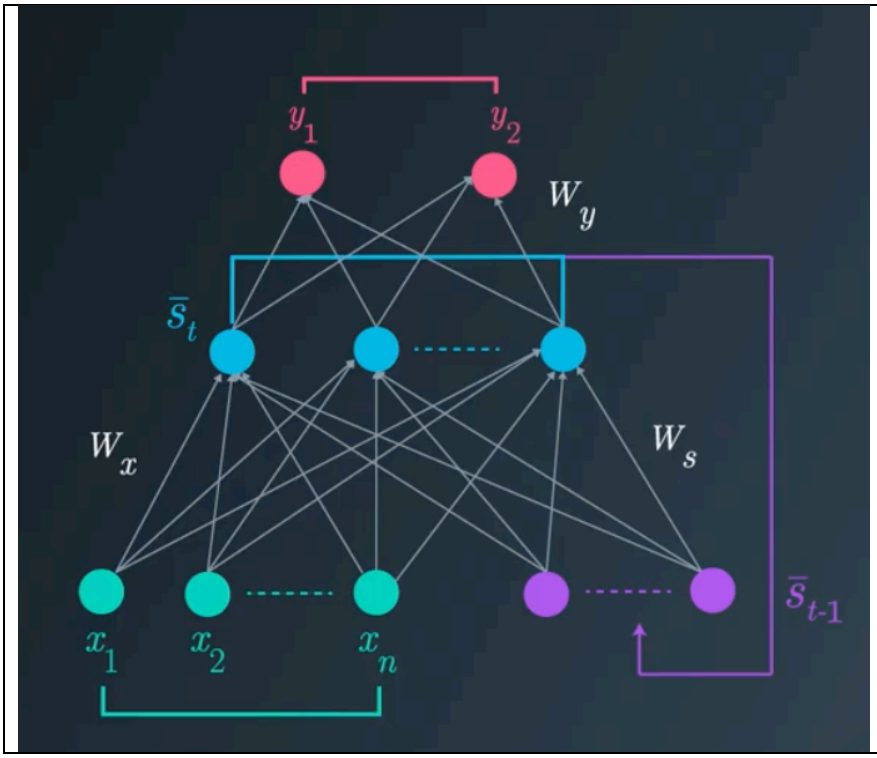
Step 3: we set a threshold, when the output exceeds the threshold, we can treat that as “the pattern is detected”

1.5 Train RNNs using BPTT (Back-Propagation Through Time)

Loss Function = The square of the difference between the desired and the calculated ouputs

In classification problems, cross entropy is usually used as loss function;

In regression problems, MSE is usually used as loss function;



Notations:

state vector:

$$\bar{s}_t = \phi\left(\bar{x}_t \cdot W_x + \bar{s}_{t-1} \cdot W_s\right)$$

$$\bar{} \ is\ vector\ symbol$$

output at timestep t:

$$\bar{y}_t = \bar{s}_t \cdot W_y$$

or

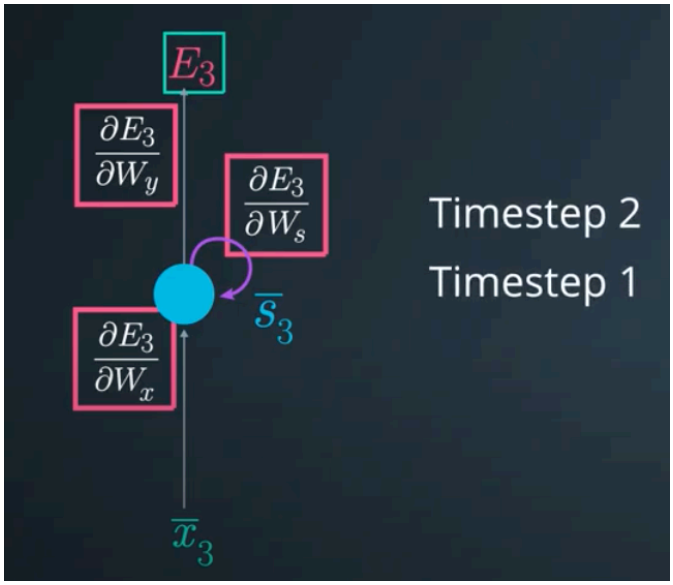
$$\bar{y}_t = \sigma\left(\bar{s}_t \cdot W_y\right)\ where\ \sigma\ is\ softmax$$

MSE loss function:

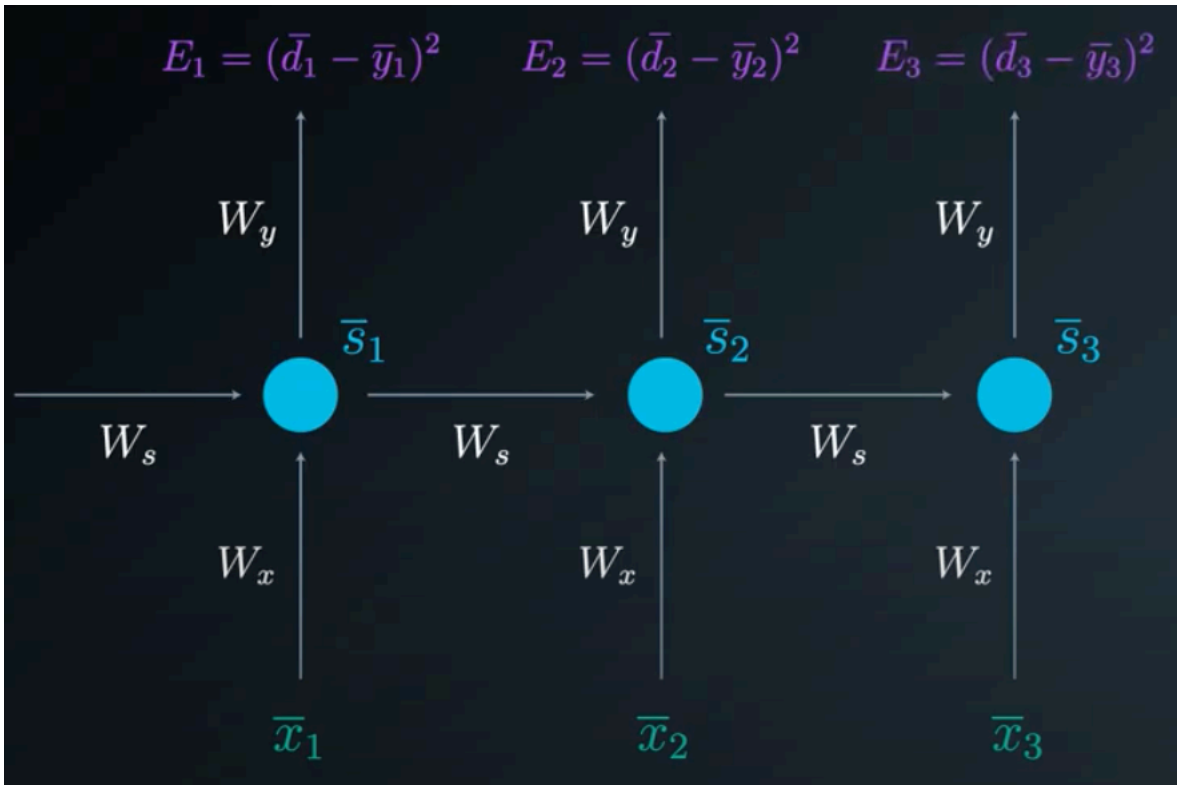
$$E_t = \left(\bar{d}_t - \bar{y}_t\right)^2$$

During BPTT:

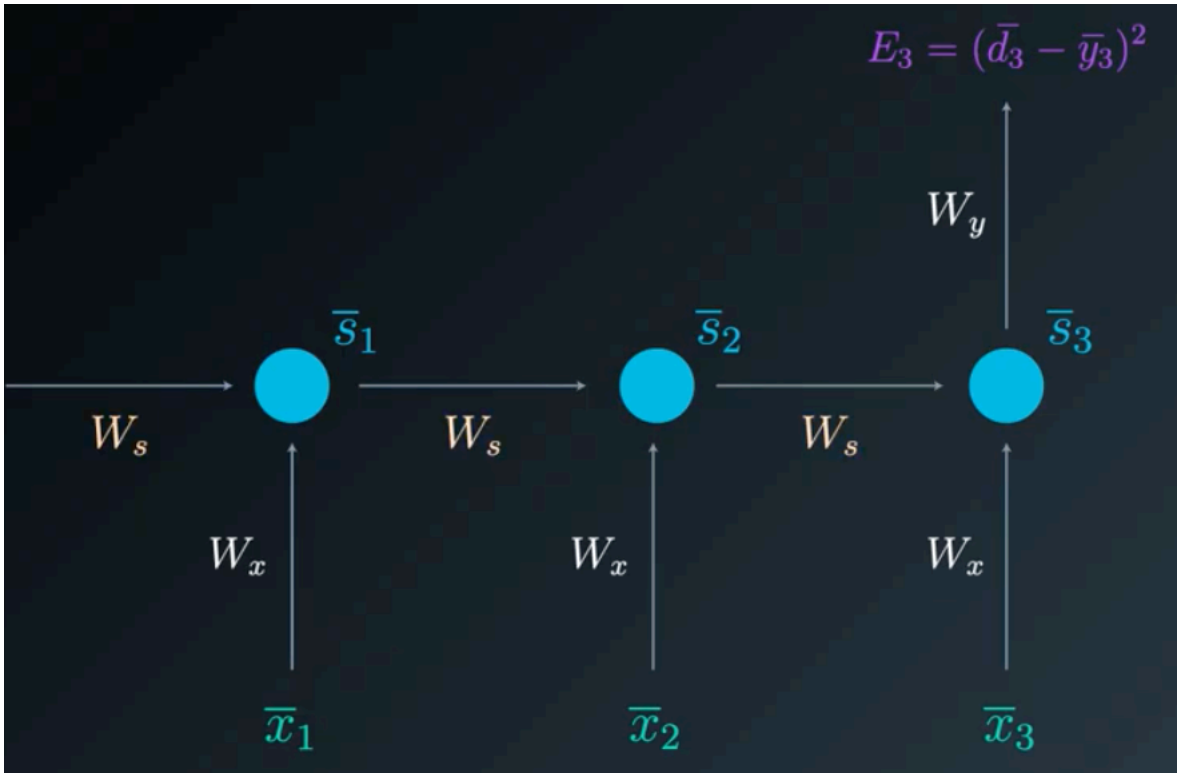
In order to compute the partial derivatives w.r.t. weights and adjust the weights in the current timestep 3, we need to consider the current timestep and previous timestep 2 and timestep 1, since previous states also “contributes” to the error at the current timestep.



Let’s unfold the RNNs to visualize different timesteps easily.



Let’s ONLY focus on the error at 3rd timestep’s error E_3 :



There are **3 weight matrices** we need to adjust:

1) W_y

$$\frac{\partial E_3}{\partial W_y} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial W_y}$$

2) W_s

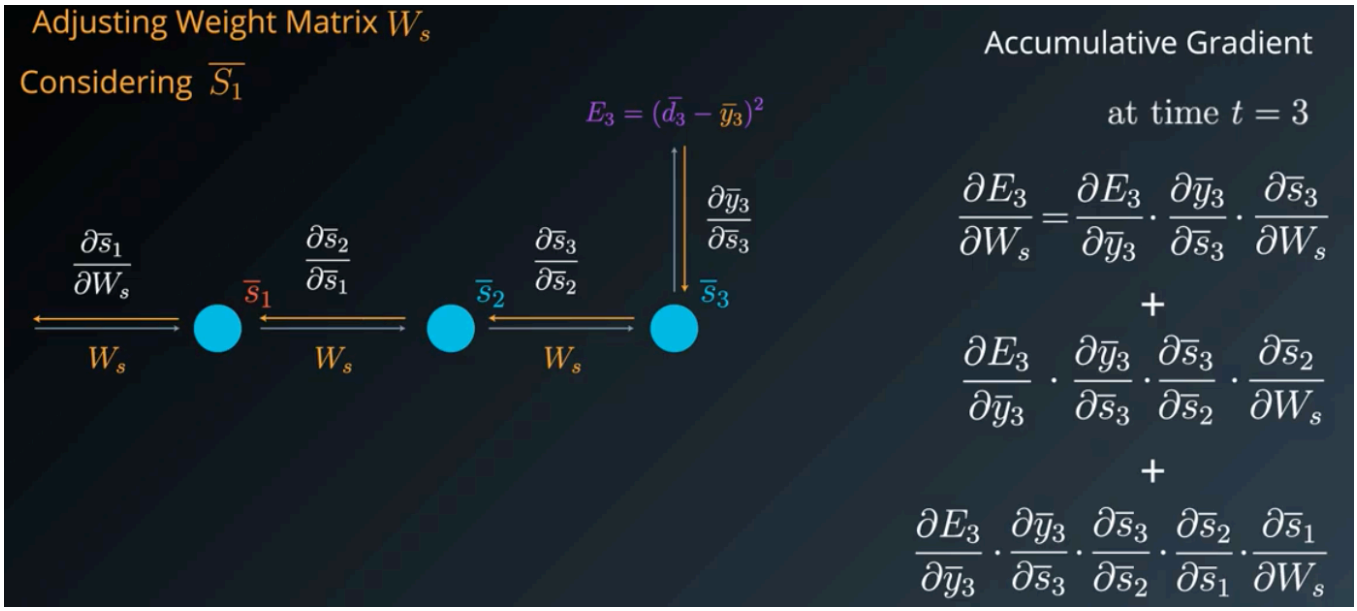
We need to take into account every gradient stemming from each previous state

$$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial W_s} + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial W_s} + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial \bar{s}_1} \cdot \frac{\partial \bar{s}_1}{\partial W_s}$$

Considering $\bar{s}_3$: $\frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial W_s}$

Considering $\bar{s}_2$: $\frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial W_s}$

Considering $\bar{s}_1$: $\frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial \bar{s}_1} \cdot \frac{\partial \bar{s}_1}{\partial W_s}$



General formula:

$$\frac{\partial E_N}{\partial W_s} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \cdot \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \cdot \frac{\partial \bar{s}_i}{\partial W_s} \quad \textit{in which we need to consider the previous N timesteps}$$

Due to vanishing problem, N cannot be larger than 8~10, but **LSTM** can solve this problem.

3) W_x

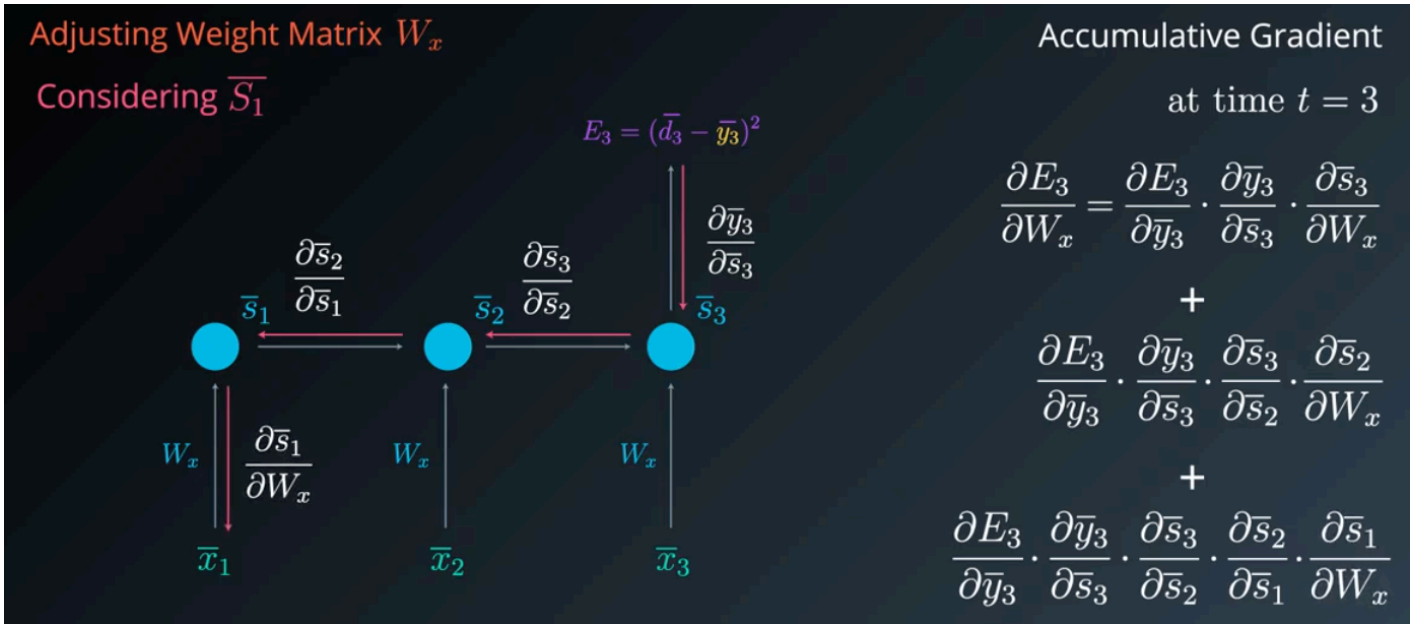
We need to take into account every gradient stemming from each previous state

$$\frac{\partial E_3}{\partial W_x} = \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial W_x} + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial W_x} + \frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial \bar{s}_1} \cdot \frac{\partial \bar{s}_1}{\partial W_x}$$

Considering $\bar{s}_3$: $\frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial W_x}$

Considering $\bar{s}_2$: $\frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial W_x}$

Considering $\bar{s}_1$: $\frac{\partial E_3}{\partial \bar{y}_3} \cdot \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \cdot \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \cdot \frac{\partial \bar{s}_2}{\partial \bar{s}_1} \cdot \frac{\partial \bar{s}_1}{\partial W_x}$

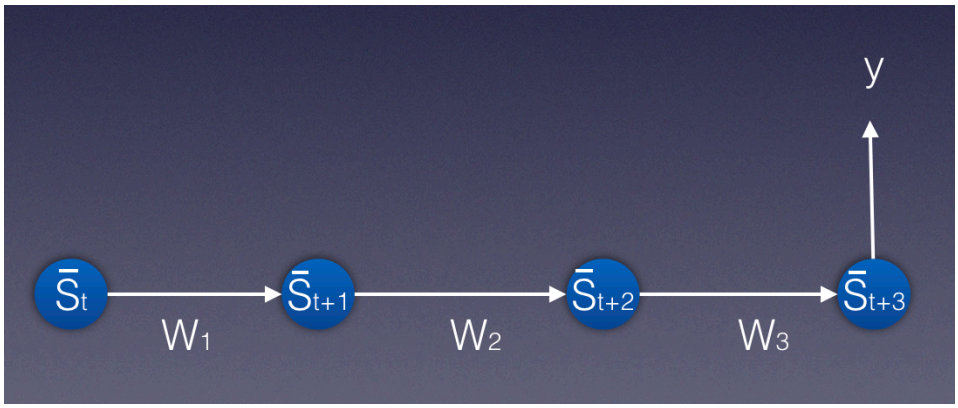


General formula:

$$\frac{\partial E_N}{\partial W_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \cdot \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \cdot \frac{\partial \bar{s}_i}{\partial W_x} \quad \textit{in which we need to consider the previous N timesteps}$$

More math during BPTT

Considering the network like this



The current timestep is t+3

$$\frac{\partial y}{\partial W_3} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial W_3}$$

$$\frac{\partial y}{\partial W_2} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial W_2}$$

$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W_1}$$

Since W1 = W2 = W3 = W

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial W_1} + \frac{\partial y}{\partial W_2} + \frac{\partial y}{\partial W_3}$$

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial W} + \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial W} + \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W}$$

This formula can be simplified as:

$$\frac{\partial y}{\partial W} = \sum_{i=t+1}^{t+3} \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W}$$

This formula can be generalized and displayed the following way:

$$\frac{\partial y}{\partial W} = \sum_{i=t+1}^{t+N} \frac{\partial y}{\partial \bar{s}_{t+N}} \frac{\partial \bar{s}_{t+N}}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W}$$

Where N is the current timestep, and we take the previous N states into account

1.6 Summary of RNN

1) The current state depends on the input of current timestep and the inputs on the previous states, with an activation function (e.g. Hyperbolic Tangent, Sigmoid, ReLU, etc.)
2) Current output is a linear combination of the current state and weight matrix. Softmax can be applied optionally.
3) "Folded" model
4) "Un-folded" model
5) Three weights, Wy, Ws, Wx
6) Gradient Calculations w.r.t. weight matrices (should consider the previous timestep)
7) BPTT
8) Training in **mini-batch using Gradient Descent**: update the weights once every N steps

$$\sigma_{ij} = \frac{1}{N}\sum_{k}^{N} \delta_{ij\,k}$$

   Advantages (same as in :
   - Reduce the complexity
   - Reduce the noise and converge faster
9) Stacked RNNs (more than 1 hidden/state layer)
10) Vanishing Gradient Problem resulting in short memory
    Solution: using LSTMs
11) Exploding Gradient Problem
    Solution: Gradient Clipping
    If the gradient $\delta = \frac{\partial y}{\partial w_{ij}} > threshold$, then normalize the successive gradients

    Paper: https://arxiv.org/abs/1211.5063

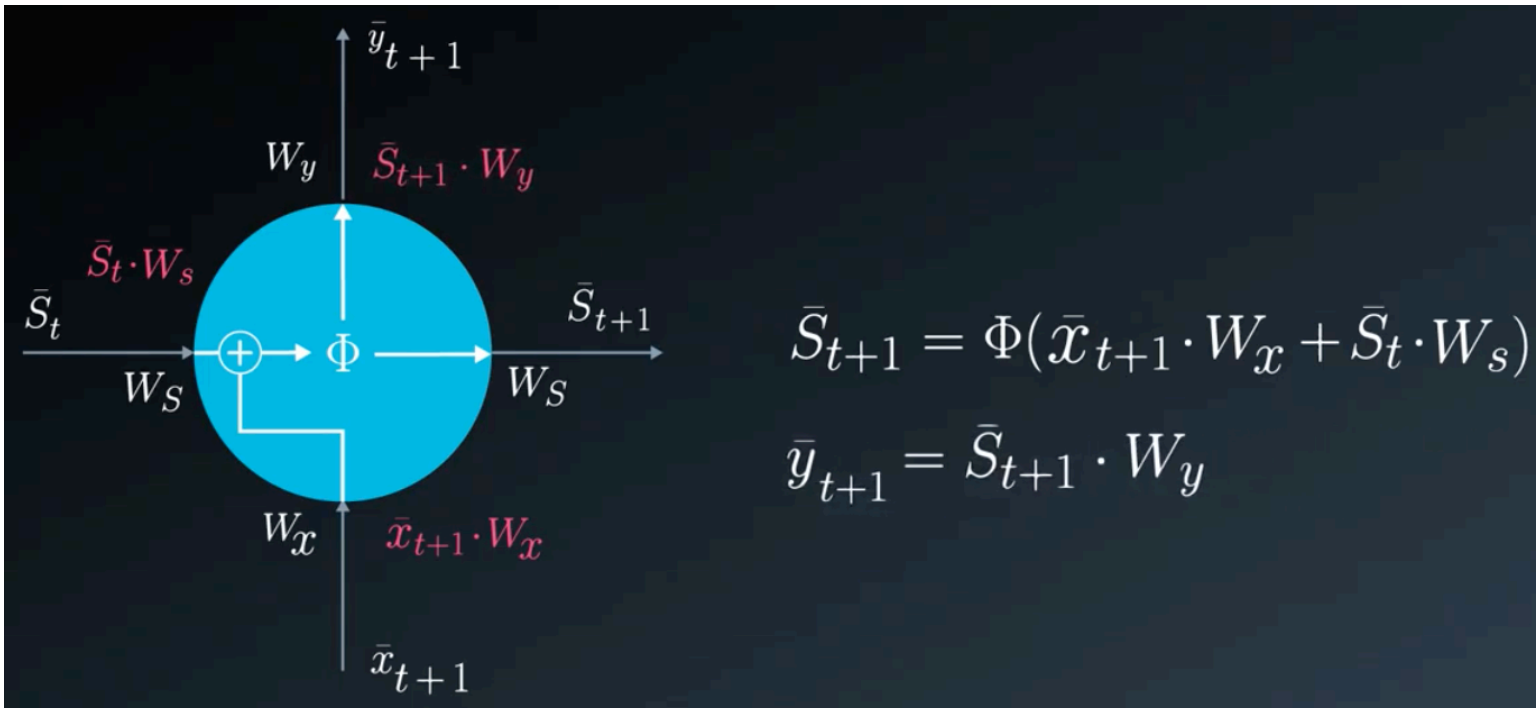
1.7 LSTM preview

Long Short-Term Memory Cell

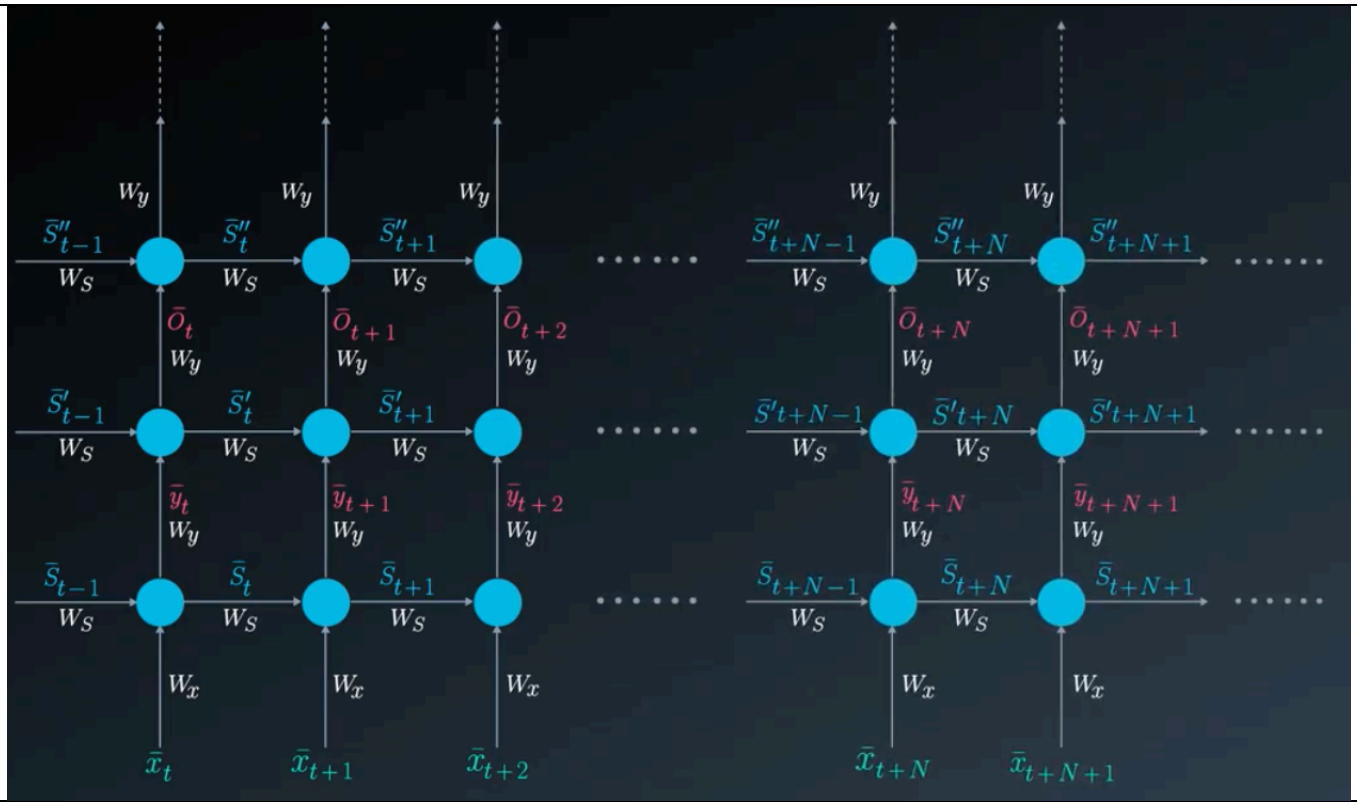
The goal of the cell is to overcome vanishing gradient problem. Input will be stored without forgetting them.

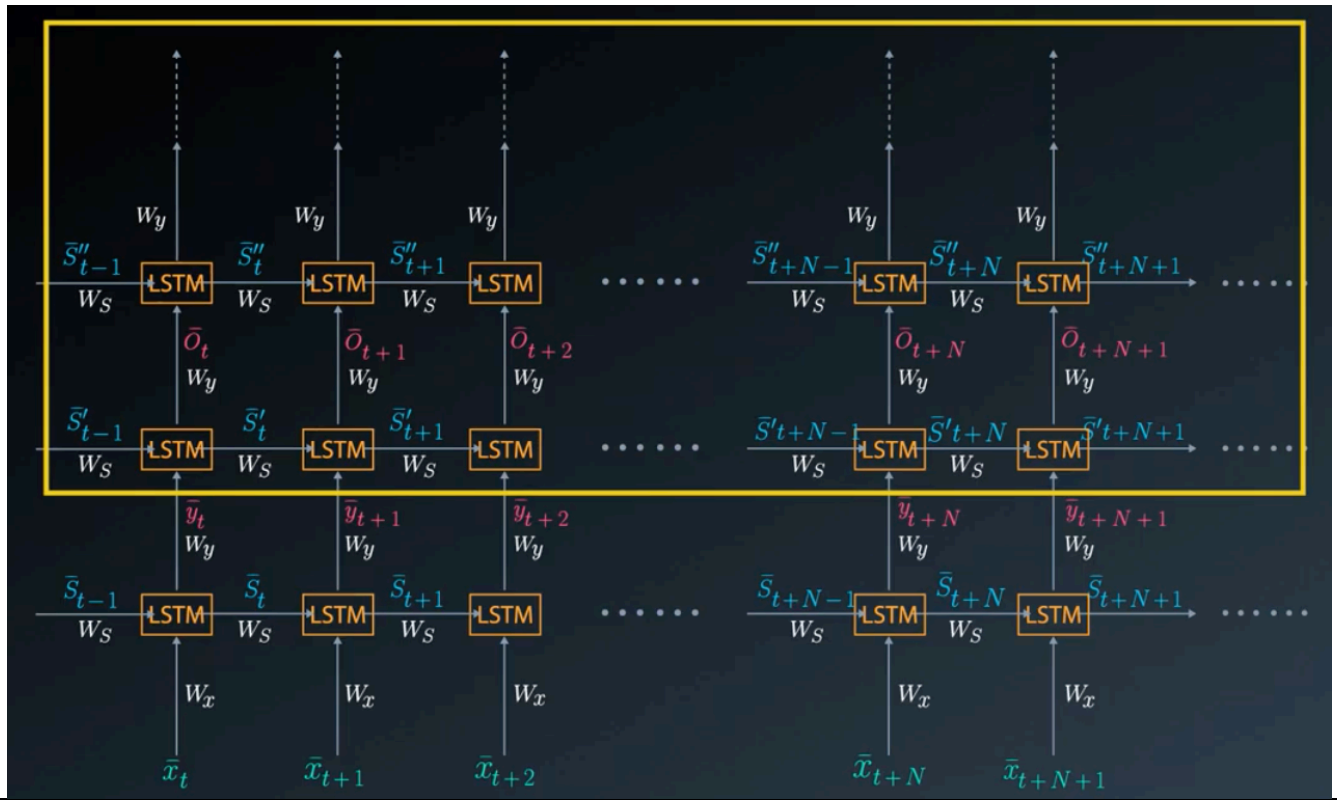
The cell latches on to information over many timesteps

In each RNN cell, we calculate the next state $\bar{s}_{t+1}$ and output $\bar{y}_{t+1}$ like this

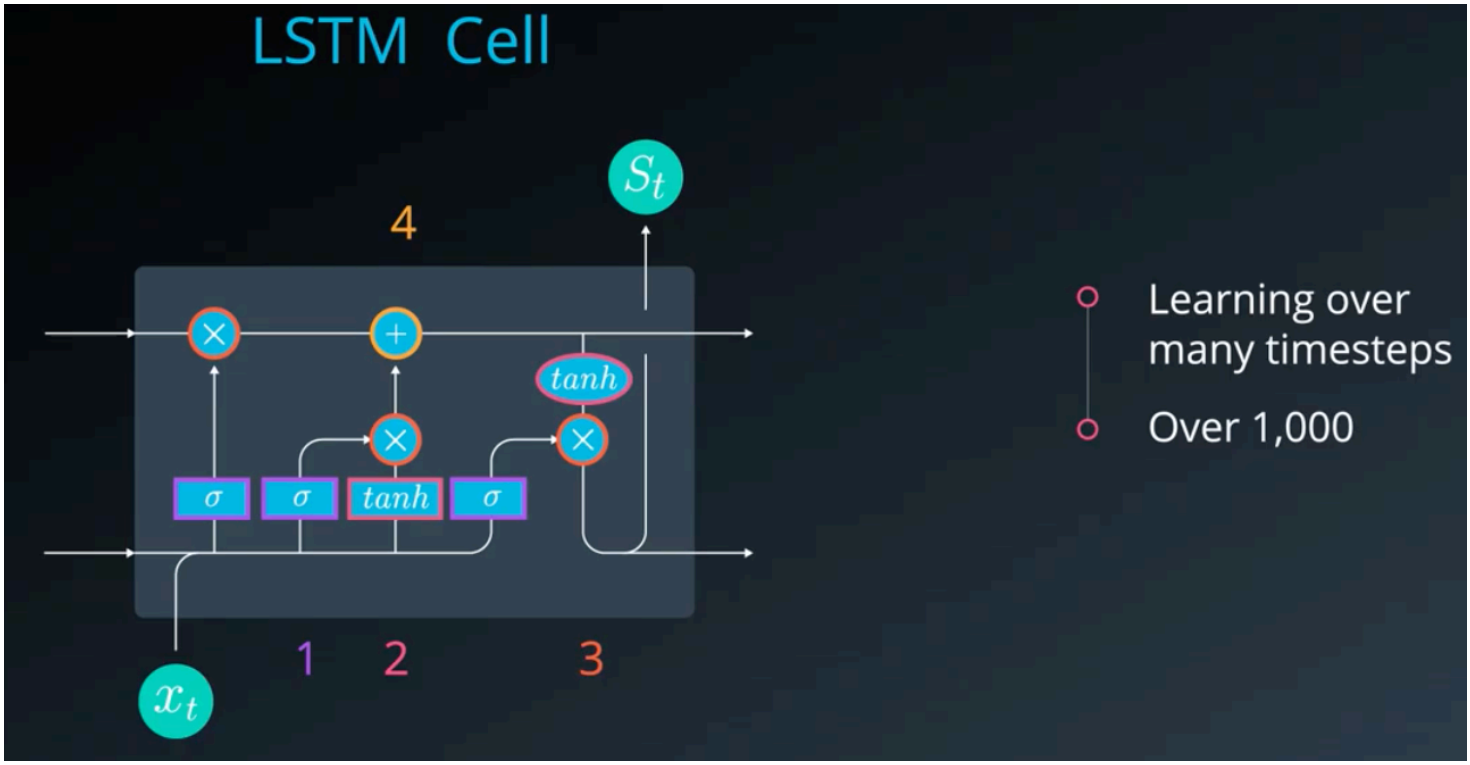


Compare RNNs and LSTMs

RNNs	

LSTMs	Replace each state neuron by a LSTM cell 

In each LSTM cell, we have 4 separate calculations



All of the functions (Sigmoid, Hyperbolic tangent, Multiplication, Addition) in a LSTM Cell have their gradient and derivative. We use back-propagation and stochastic gradient descent to update their weights.

LSTM cells can

1) Choose which information to forget
2) Choose which information to store
3) When to use the information
4) When to move the information to the next LSTM cell

using gating functions. Pretty powerful hah!

Lesson 2: Long Short-Term Memory Networks (LSTMs)

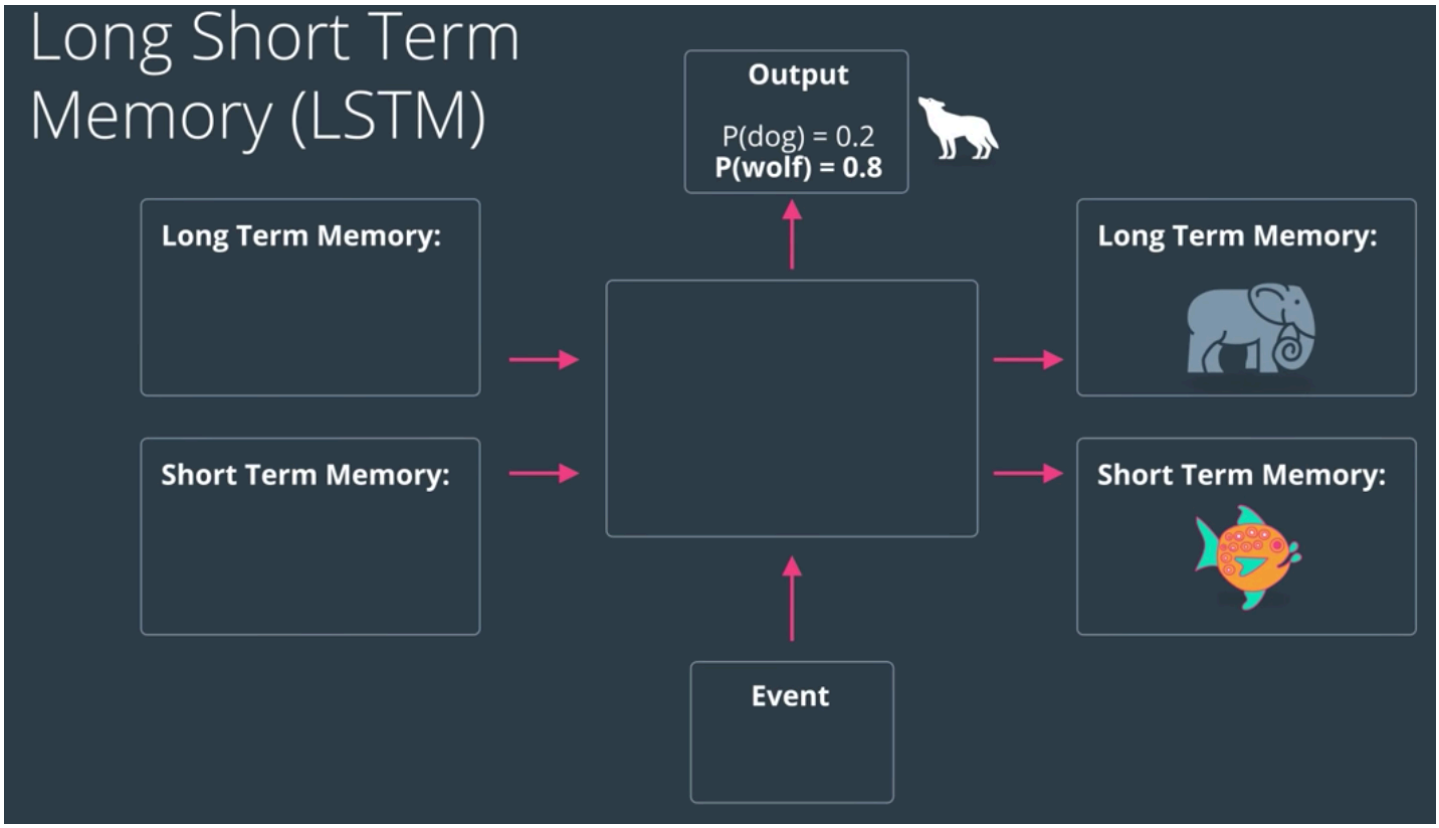
2.1 Basics of LSTMs

In each LSTM Cell, there are 3 inputs:

1) Long Term Memory
2) Short Term Memory
3) Current Event at the current timestep

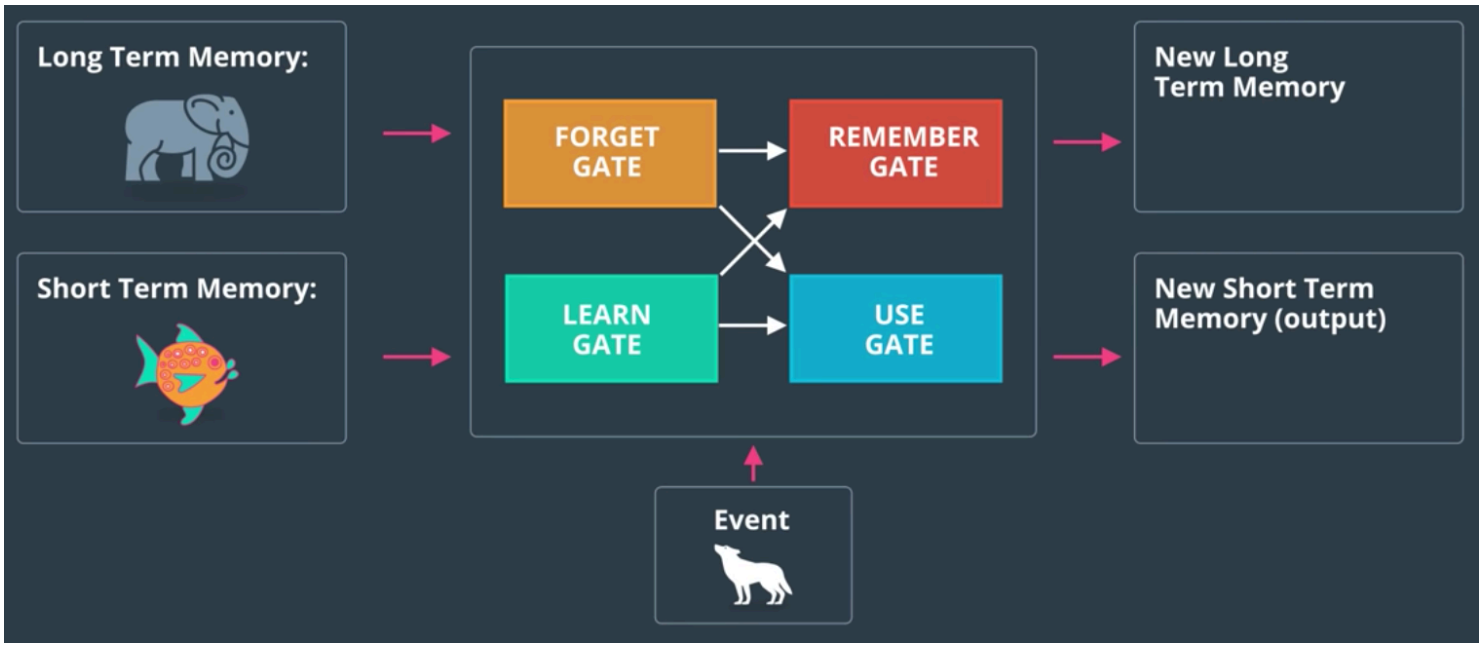
What we need to do in an LSTM Cell:

1) Generate an output of the cell based on 3 inputs
2) Update the long term memory based on 3 inputs
3) Update the short term memory based on 3 inputs

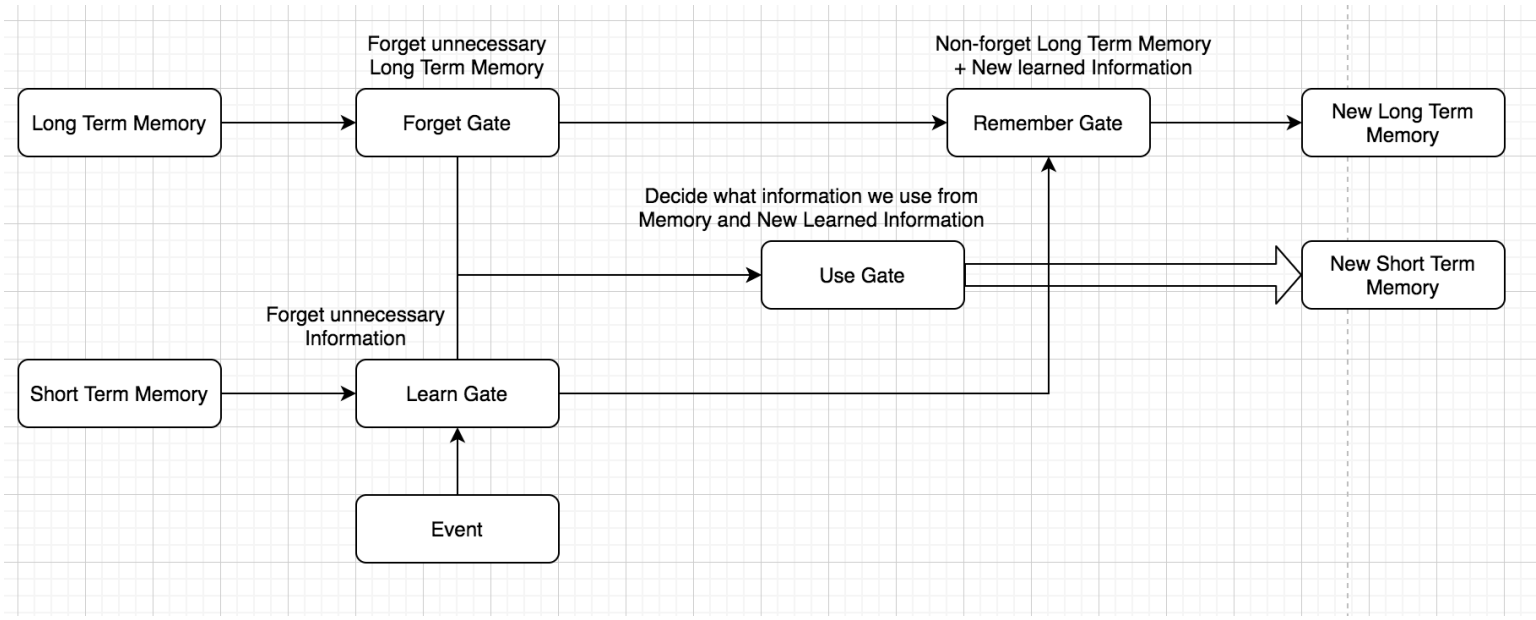


There are 4 gates in an LSTM Cell

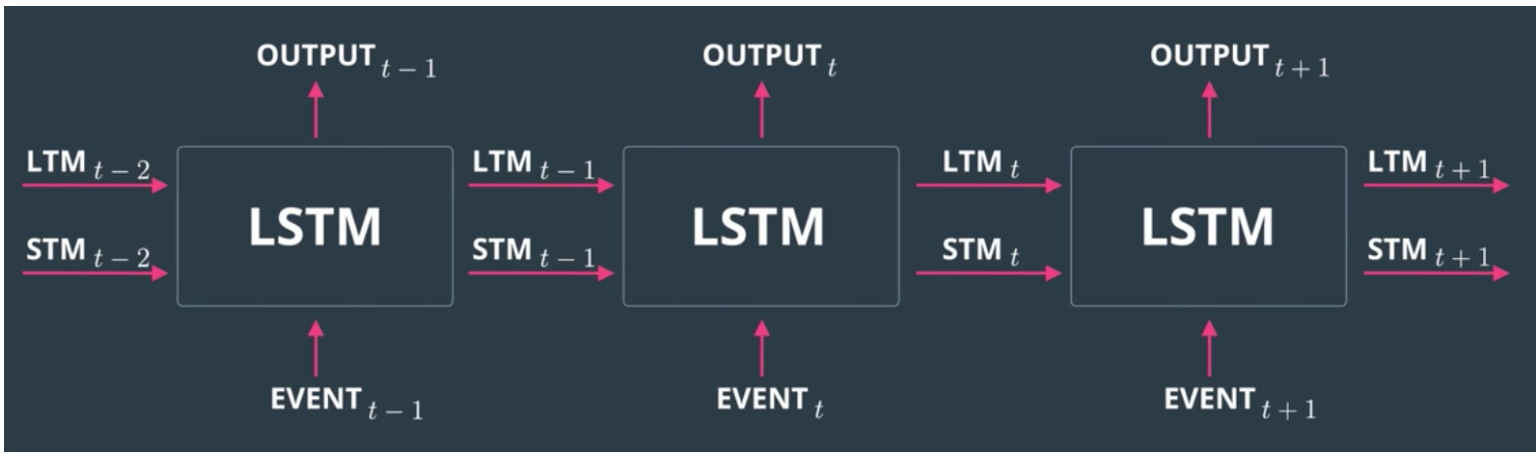
1) Forget Gate
2) Remember Gate
3) Learn Gate
4) Use Gate



This is how memory and event move:



This is how we end up with multiple LSTM Cells

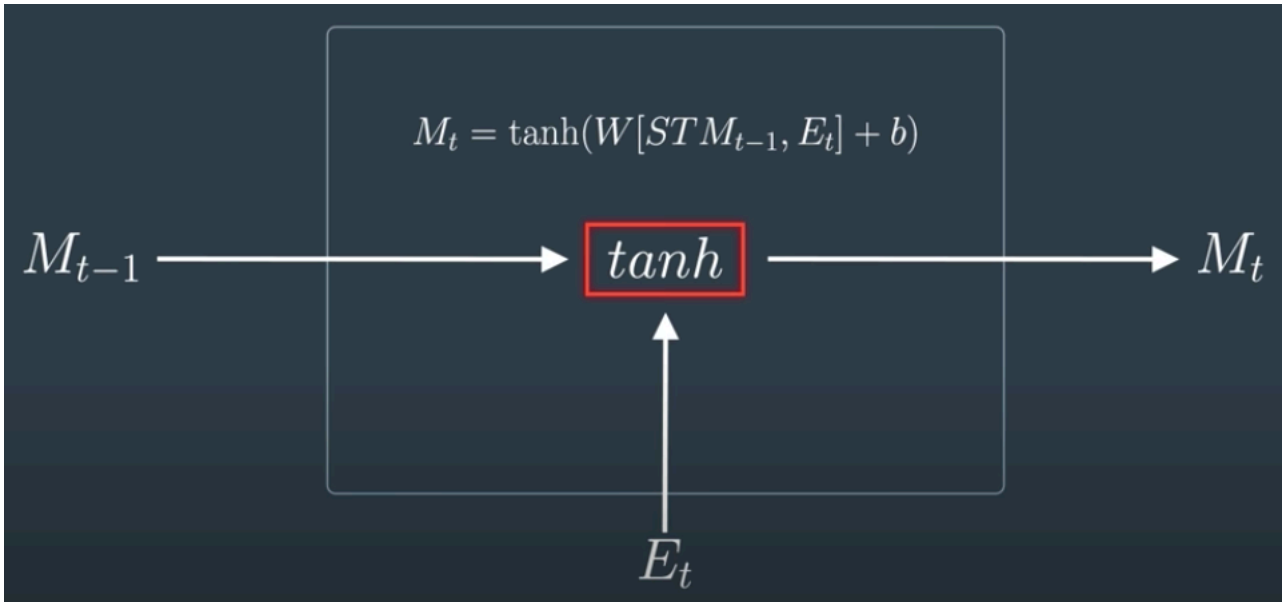


2.2 Architecture of LSTMs

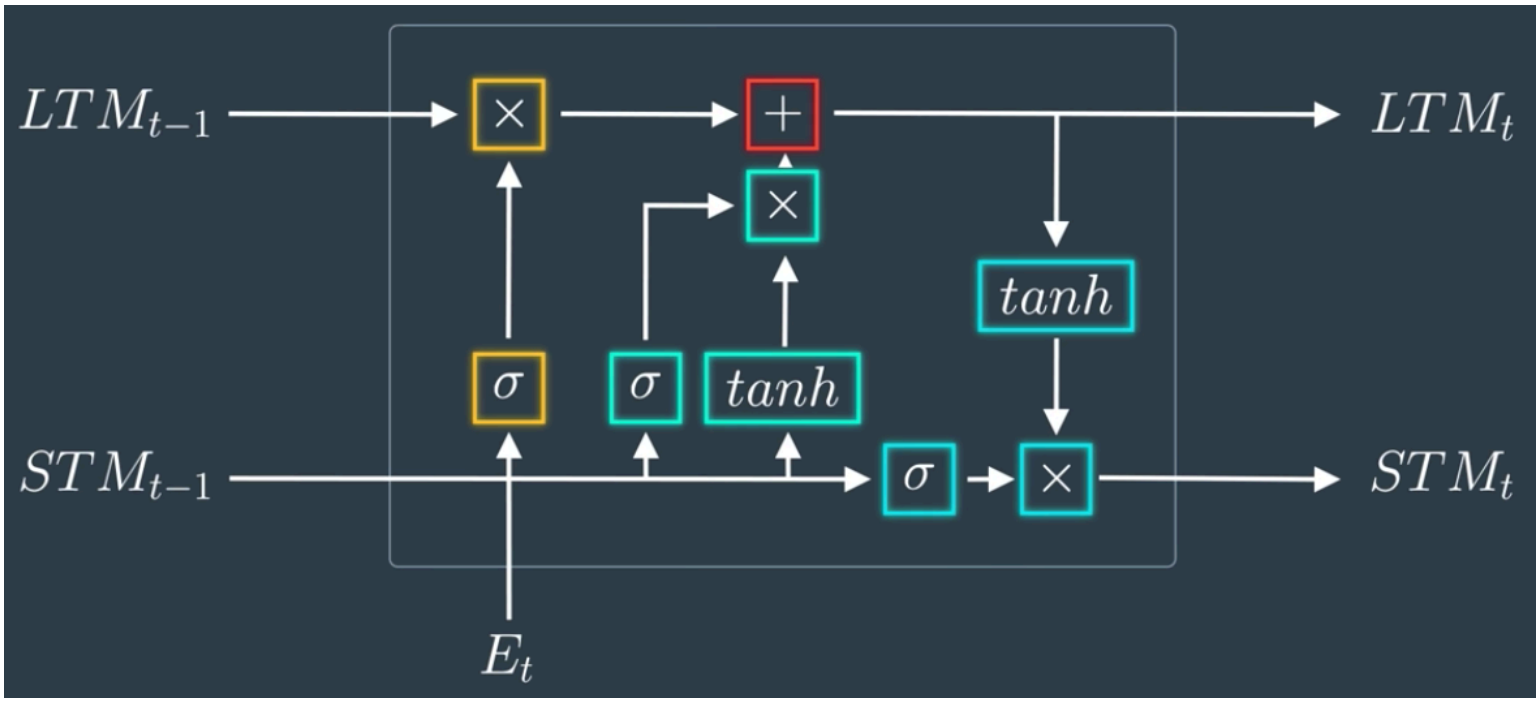
Notations:

- M_{t-1} : Memory at previous timestep
- E_t : Event at current timestep
- M_{t-1} : Memory at current timestep which will be passing to the next timestep

On higher level, its architecture is like this:



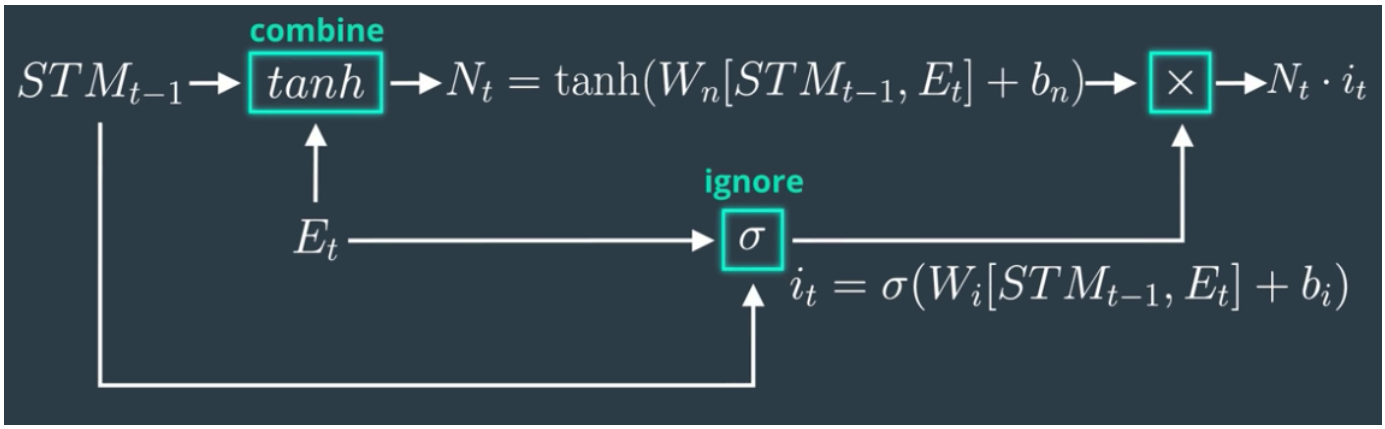
Specifically, this is the architecture of LSTM Cell:



-  : matrix multiplication
-  : sigmoid activation function
-  : matrix addition
-  : tanh activation function

Learn Gate

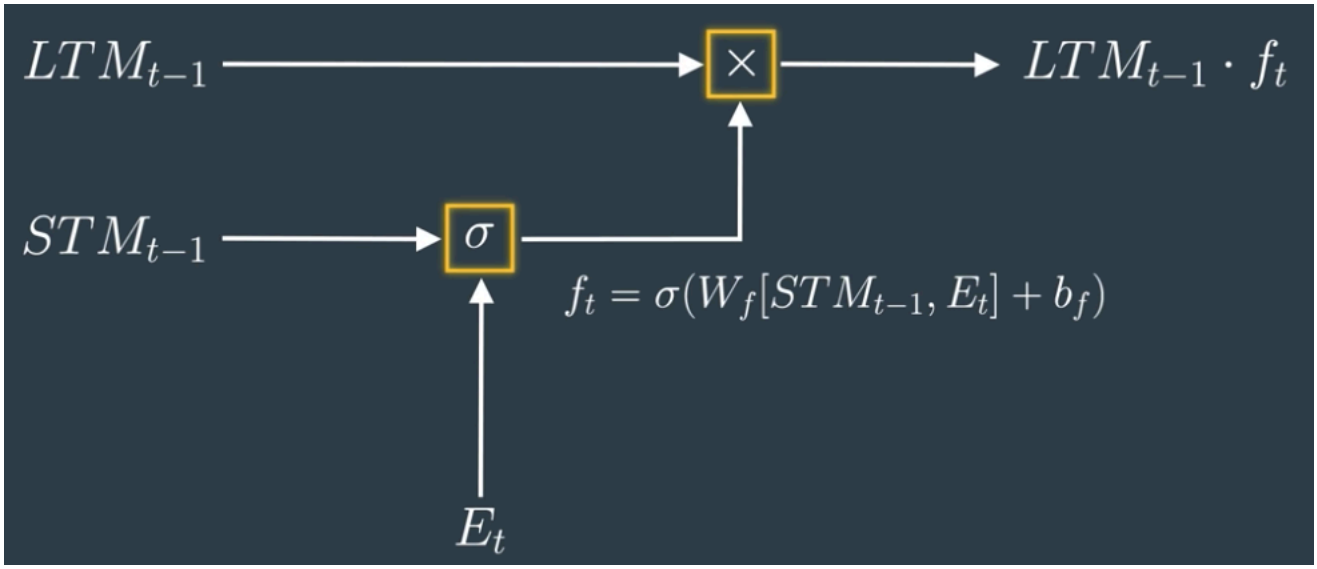
1) Takes a short-term memory STM_{t-1} and event E_t, and comines them as N_t.
2) Then ignore unnecessary information by an ignore factor i_t and keep the important memory
3) Mathematically,



- i_t : ignore factor matrix
  - i_t is calculated by the current event E_t and previous short-term memory STM_{t-1}
- $N_t \cdot i_t$: multiply **element-wise**
- σ : sigmoid function

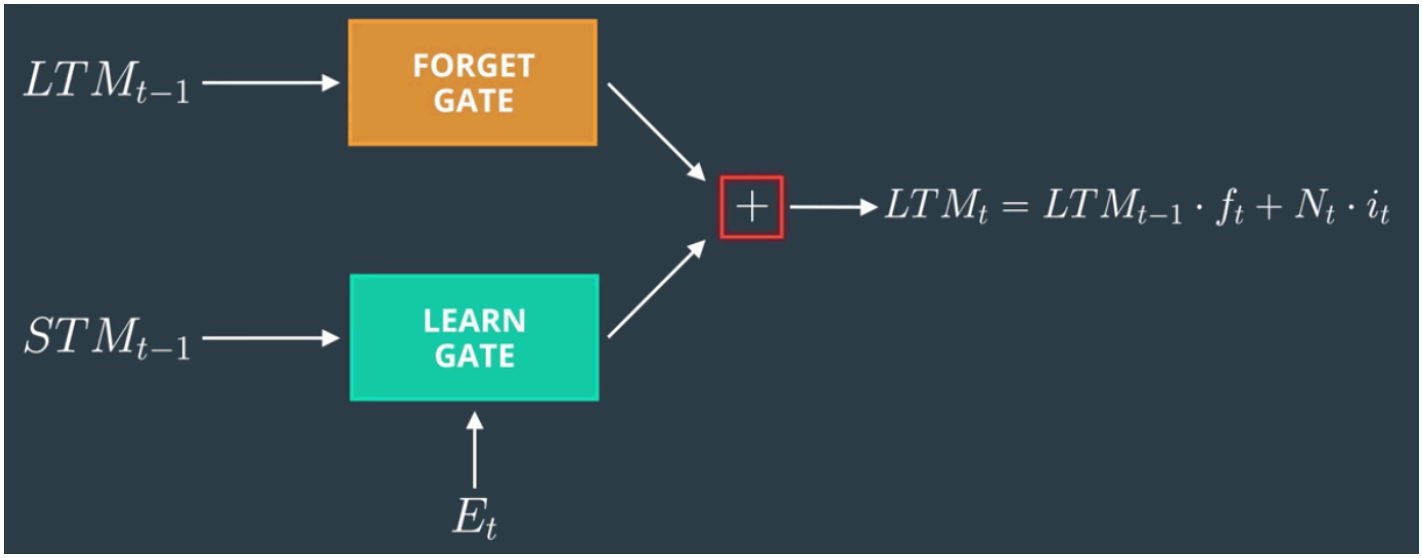
Forget Gate

1) Takes a long-term memory and decide which parts to keep and which parts to forget by a forget factor f_t which is calculated by the previous short-term memory STM_{t-1} and current event E_t
2) Mathematically,



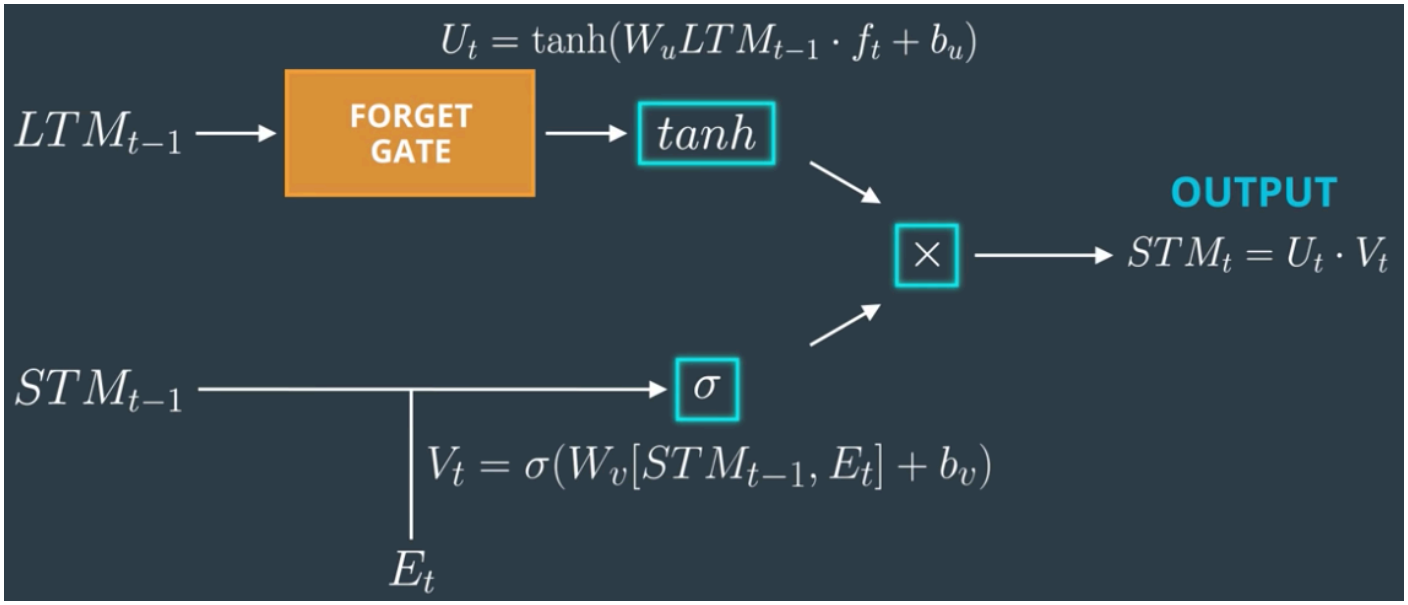
Remember Gate

1) Takes long-term memory $LTM_{t-1} \cdot f_t$ from the Forget Gate and short-term memory $N_t \cdot i_t$ from the Learn Gate, and combines them
2) Outputs the new long-term momory
3) Mathematically,

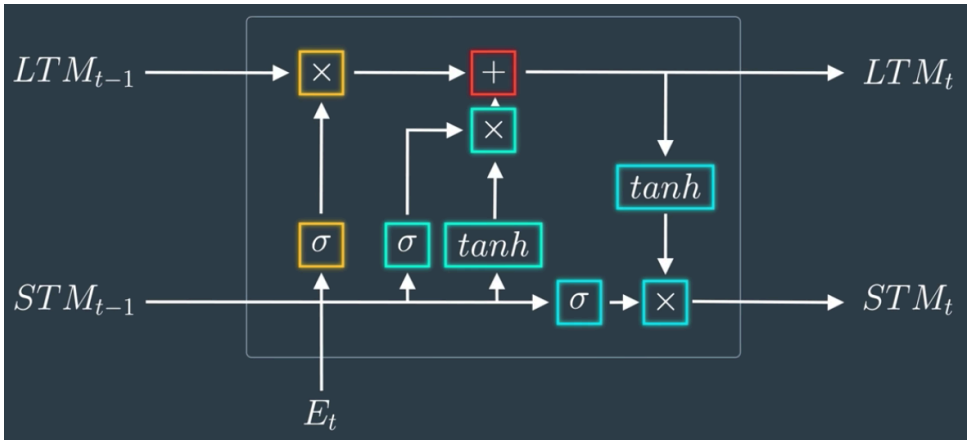


Use Gate / Output Gate

1) Takes long-term memory $LTM_{t-1} \cdot f_t$ from the Forget Gate and short-term memory $N_t \cdot i_t$ from the Learn Gate
2) Outputs a new short-term memory and the output of the cell
3) Mathematically,

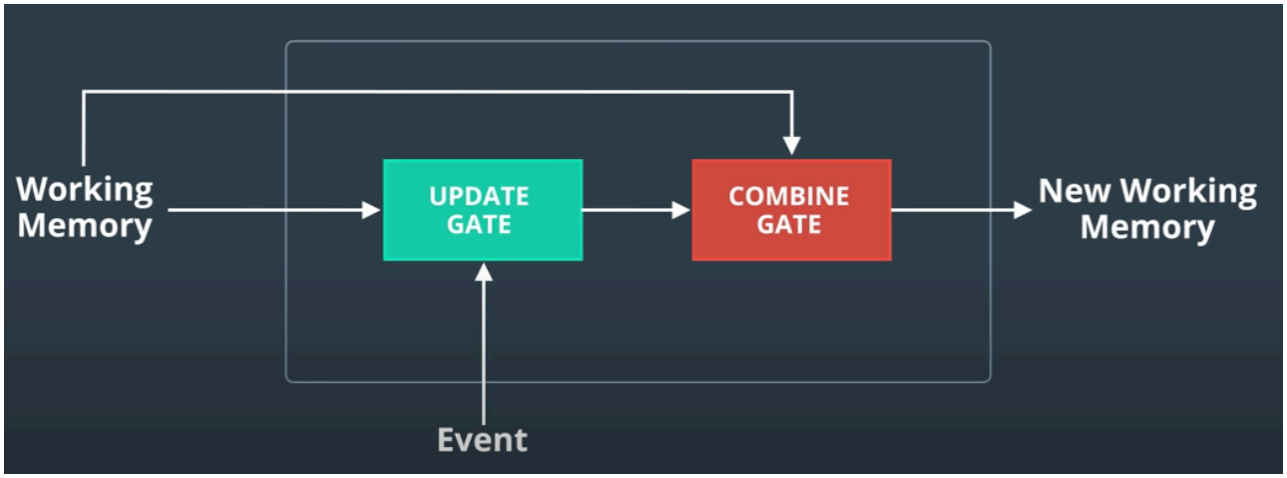


Note that, there are more architectures other than this architecture



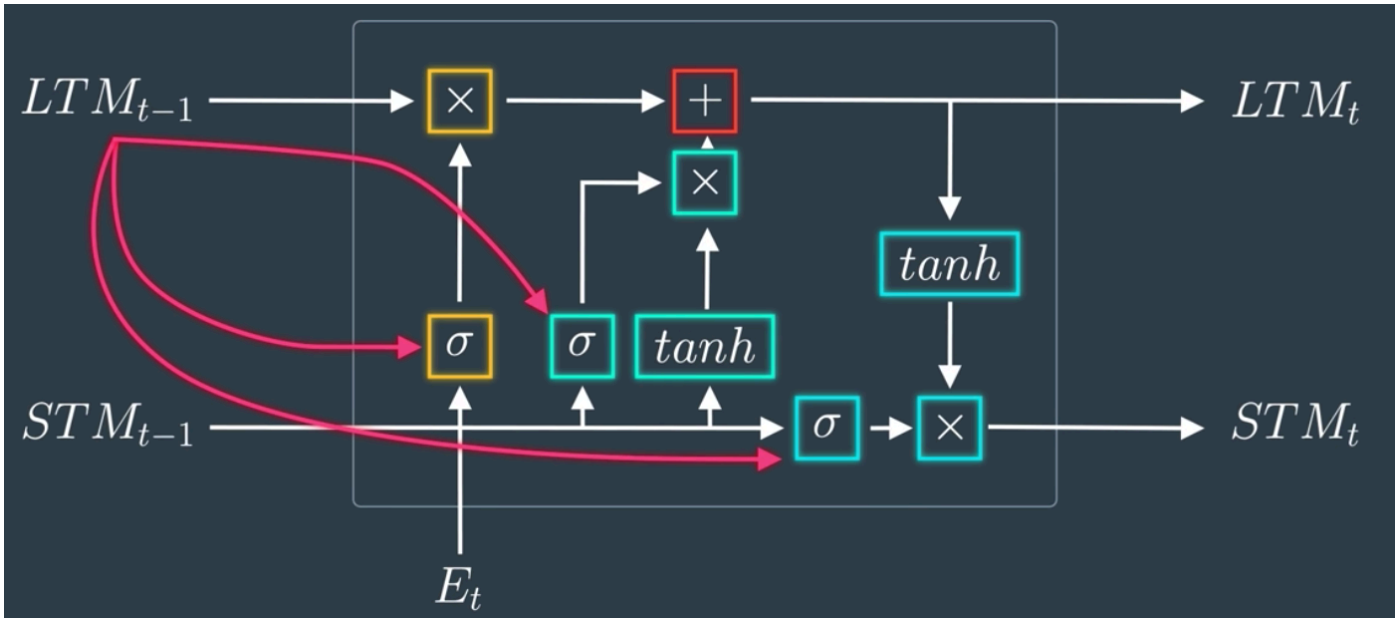
Examples of various LSTM Cell:

1) GRU (Gated Recurrent Unit)



Long Term Memory and Short Term Memory are replaced by Working Memory

2) LSTM with Peephole Connections



Add the feature that Long Term Memory also takes part in the calculation of the forget factor f_t which decides which parts to keep and which parts to forget

Lesson 3: Implementation of RNN & LSTM

Pipeline of implementation of RNN & LSTM

1) Pre-process sequential data, convert the data to the form that can be understood by RNN & LSTM
2) Represent memory

An RNN example:

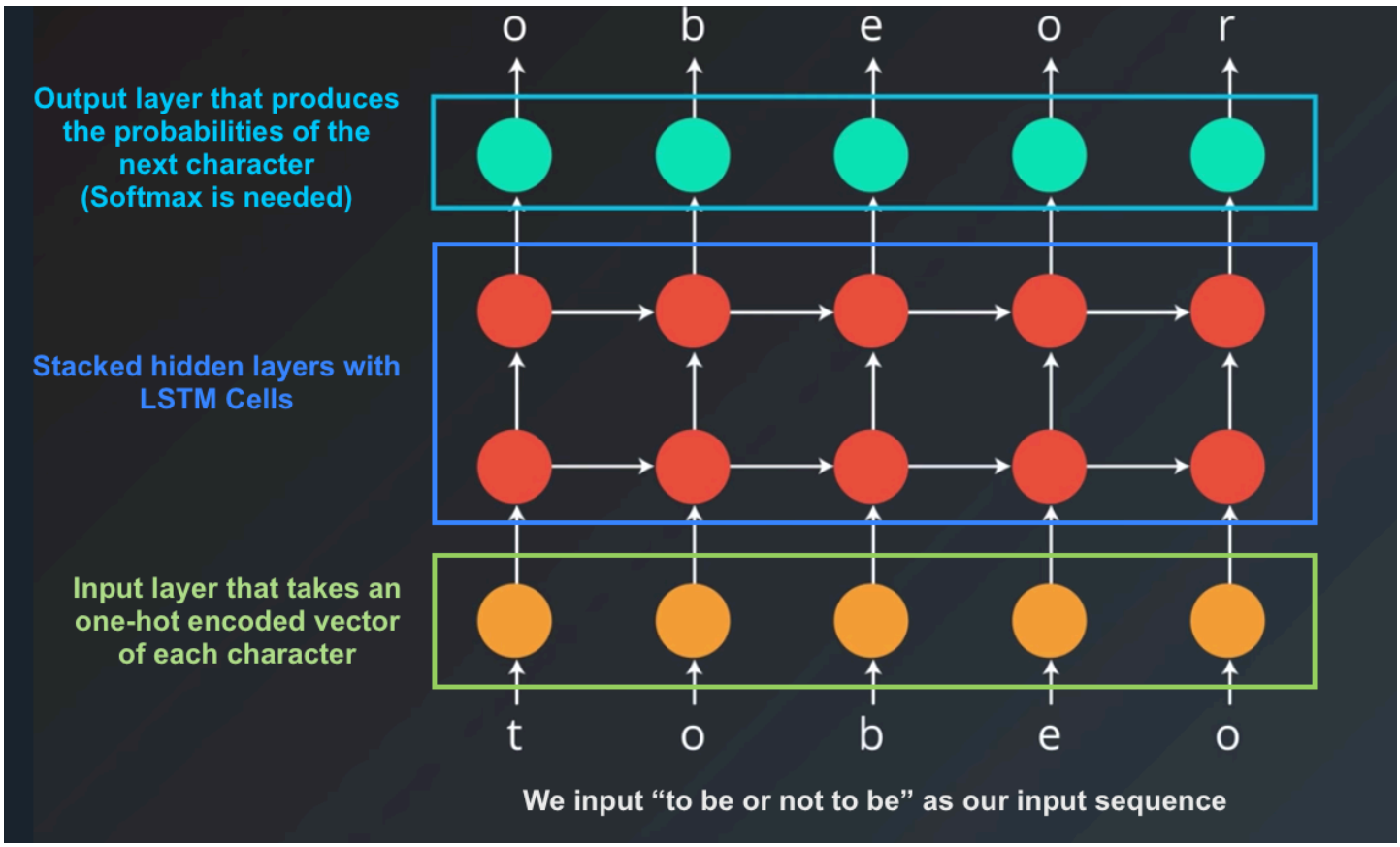
Open Notebook: *../ notebooks (filled) / 4. Recurrent Neural Networks / time-series / Simple_RNN.ipynb*

Character-wise RNNs

Definition: the RNN will learn 1 character from the text at a time, and generate new text 1 character at a time.

- Pre-process sequential data
  Convert the characters as one-hot encoded vectors

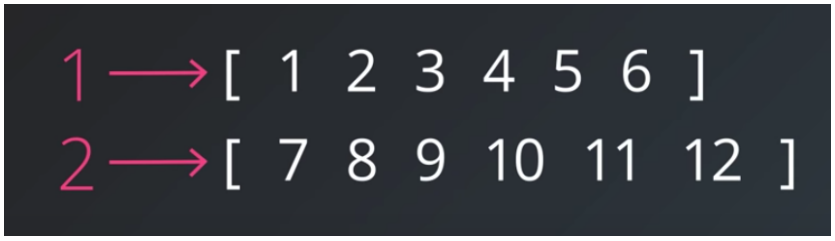
- Here's the architecture of LSTM Network



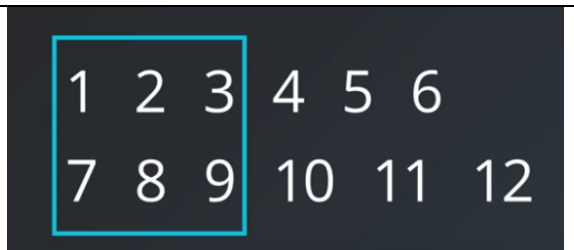
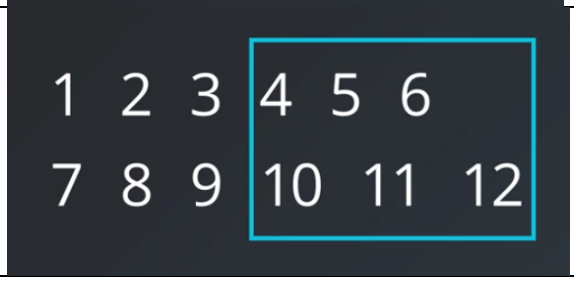
Choose the batch size, length of each sequence

Batch size corresponds to the number of sequences feeded to our RNN

An example of batch_size = 2:



An example of batch_size = 2, length_of_sequence = 3:

First batch	
Second batch	

Open Notebook: ***../ notebooks (filled) / 4. Recurrent Neural Networks / recurrent-neural-networks / char-rnn / Character_Level_RNN_Exercise.ipynb***

LSTMs creates memory by the fact that each cell pass along its hidden state to the next cell, so that the next cell can have a kind of “memory” of the output of the previous memory.

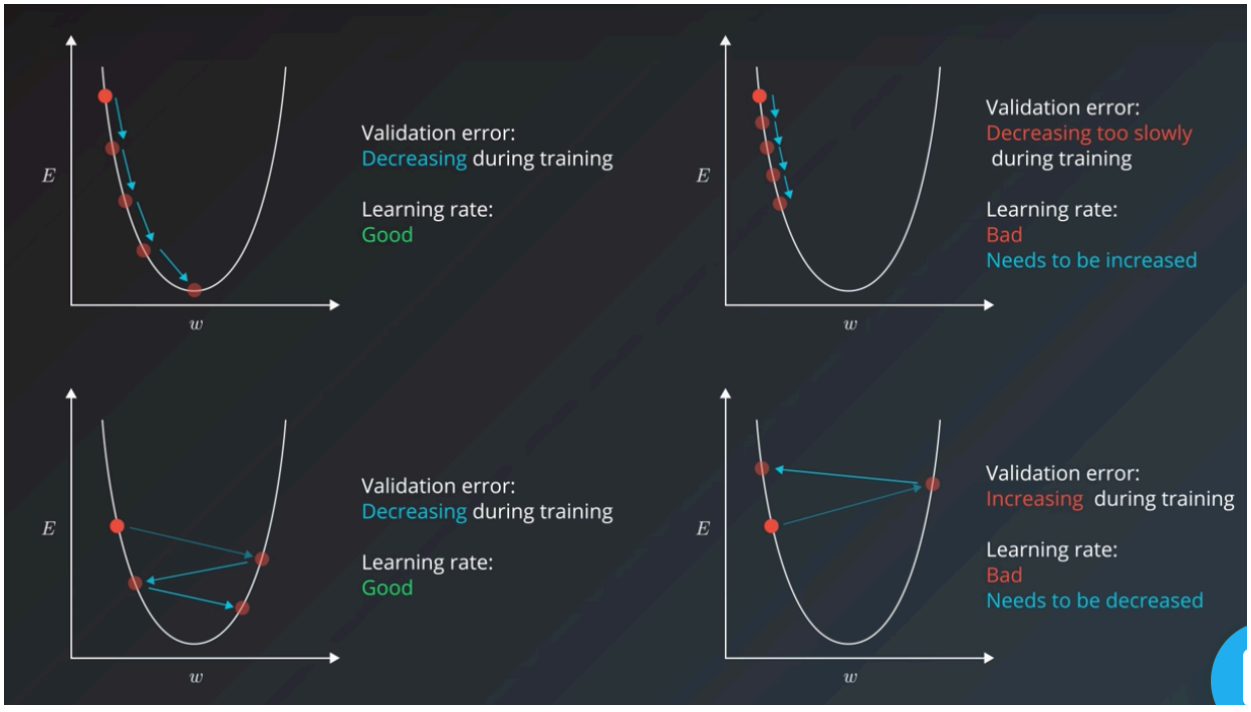
Category	Explanation	Examples
Optimizer Hyper-parameters	relates to optimization and training process than to the model itself	• Learning rate • Mini batch size • Training iteration (epochs)
Model Hyper-parameters	involves in the model structure	• Number of hidden layers • Number of hidden units • Model specific parameter (e.g. max_depth in decision tree etc.)

4.1 Learning Rate

Good start = 0.01

In Error-Weight Space, learning rate is the multiplier we use to push the weight towards to right direction (minimize the error)

Large learning rate problem → diverge



Other challenges with learning rate:

- Error curve can by any shape, even vary in higher dimensions

Other techniques:

1) Learning Rate Decay: decrease the learning rate as training the model
    a. Decrease lr linearly
    b. Decrease lr exponentially
       Tensorflow doc:
       https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/exponential_decay
       Tensorflow: AdamOptimizer, AdagradOptimizer
2) Adaptive Learning rate: adjust the lr based on what the learning algorithm knowns and the data the model has seen so far.

4.2 Mini-barch Size

In **Stochastic Gradient Descent**, we do forward + backward passes and adjust the weight after a single sample.

In Batch Gradient Descent, we do forward + backward passes and adjust the weight after 1 epoch (entire of dataset)

In Mini-Batch Gradient Descent, we we do forward + backward passes and adjust the weight after 1 batch of samples.

Good start = 32, 64, 128, 256

Small Mini-batch	More noise causing stop training in local minima
Large Mini-batch	Cause more memory and computations

4.3 Number of iterations:

Early stopping helps us choose number of iterations (stop training once validation error stop decreasing)

4.4 Number of Hidden Units, Hidden Layers

4.4.1 Number of hidden units

This relates to the complexity of the model. Too complex models are prone to memorize rather than learning.

In this step, **regularization and dropout can help**.

Strategy: Keep adding hidden units until the model starts getting worse.

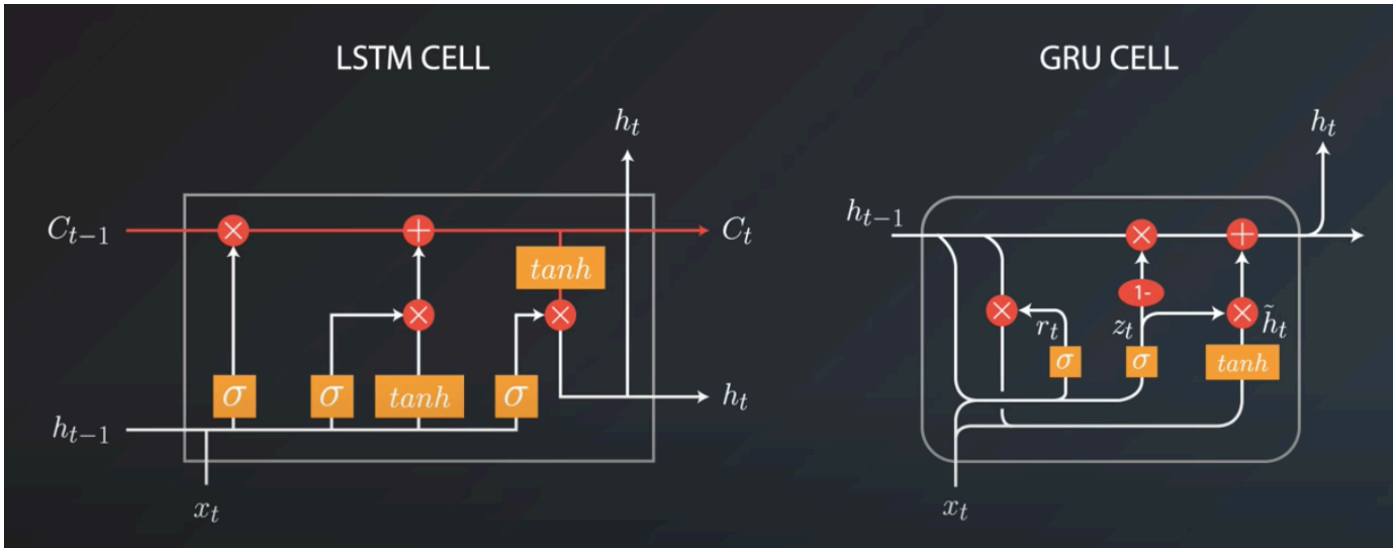
4.4.2 Number of hidden layers

Conclusion: 3-layer NN outperforms 2-layer NN, going deeper rarely helps much. Exception: CNN (the deeper, the better)

4.5 RNN Hyperparameters:

4.5.1 Cell Type

Conclusion: LSTM Cell and GRU Cell ourperform than vanilla RNN Cell



Which one of the two is even better ? Depends on the task and dataset

4.5.2 How deep the RNN

Good Embedding size : 50-200

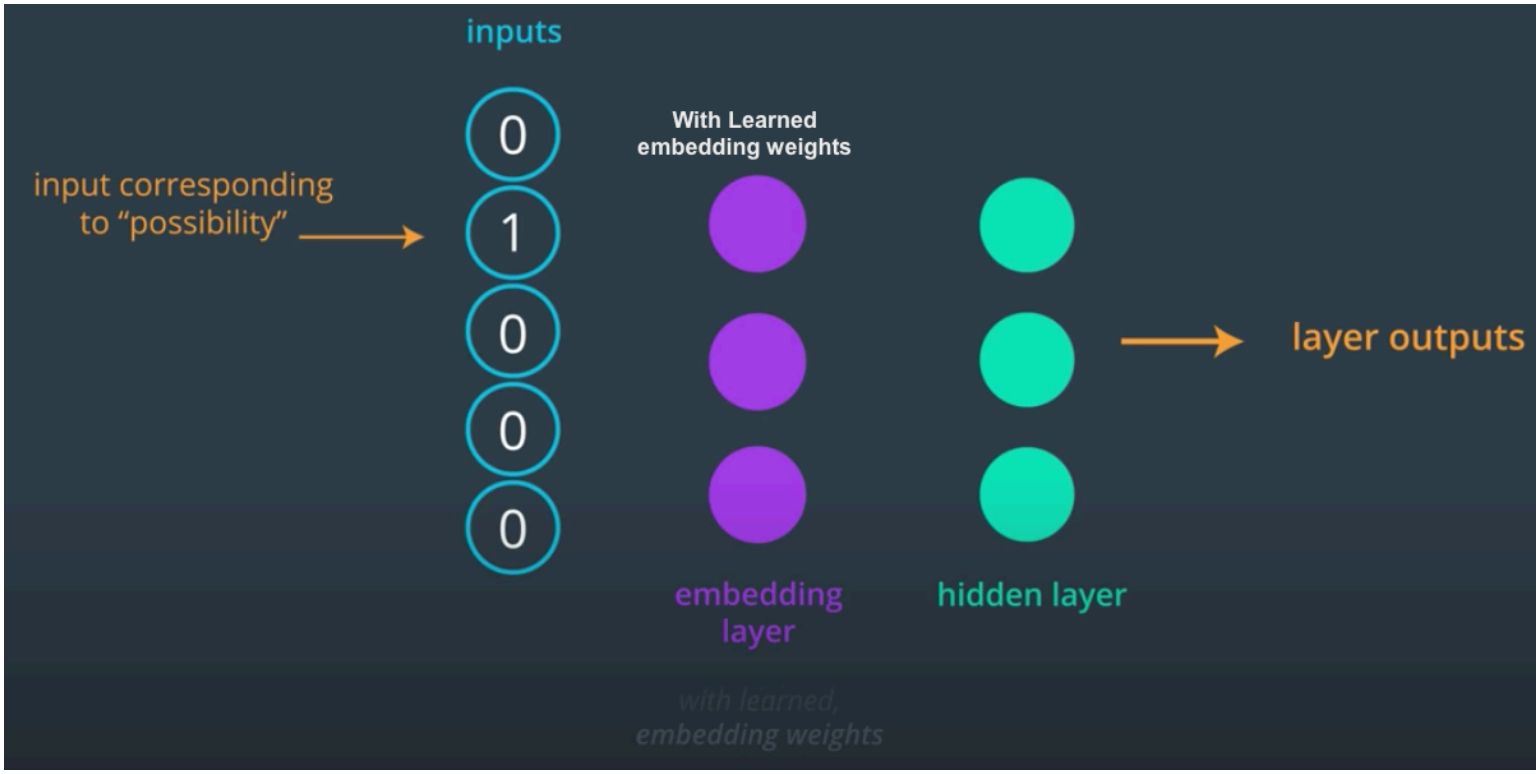
Lesson 5: Embeddings & Word2Vec

We can use NN to learn word embedding for NLP tasks

Word2Vec model: map words to embeddings that contain semantic meaning

Debiasing word embedding: if the original training text has biased associations, the biases will be replicated in word embeddings

To learn the embedding, we add an embedding layer just between input layer and the first hidden layer



Embedding lookup: use embedding weight matrix as a loopup table. A different one-hot vector input results in different rows in the weight matrix fed to the first hidden layer.

Note that: The computation between one-hot vector and weight matrix is NOT dot matrix multiplication. The row number of weight matrix == vocabulary size

Open Notebook: *../ notebooks (filled) / 4. Recurrent Neural Networks / char-rnn / Character_Level_RNN_Exercise.ipynb*

Word2Vec is implemented in one of the 2 ways:

1) CBOW (Continuous Bag-Of-Word): give the model the context (surrounding words) and let the model predict the missing word
2) Skip-gram: give the model the word, let the model predict the context (surrounding words)

Context size → surrounding word length

Training Word2Vec model:

Open Notebook: *../ notebooks (filled) / 4. Recurrent Neural Networks / word2vec-embeddings / Skip_Grams_Exercise.ipynb*

To solve slow training process using Negative Sampling

Open Notebook: *../ notebooks (filled) / 4. Recurrent Neural Networks / word2vec-embeddings / Negative_Sampling_Exercise.ipynb*

We use 2 embedding layers instead of 1, one for input word and one for output word, and modify the loss function

Open Notebook: *../ notebooks (filled) / 4. Recurrent Neural Networks / sentiment-rnn / Sentiment_RNN_Exercise.ipynb*

Use “TensorDataset wrapper” and “DataLoaders”: create a batches of dataset out of standard data structures

Two way to test on new data:

1. Use inference
2. Use test data

More Lecture Note and results, see in notebook ***Sentiment_RNN_Exercise.ipynb***

Open Notebook: ***../ projects / 4. Generate TV Scripts / project-tv-script-generation / dlnd_tv_script_generation.ipynb***

Lesson 8: Attention

8.1 What is Attention ?

Example applications with Attention:

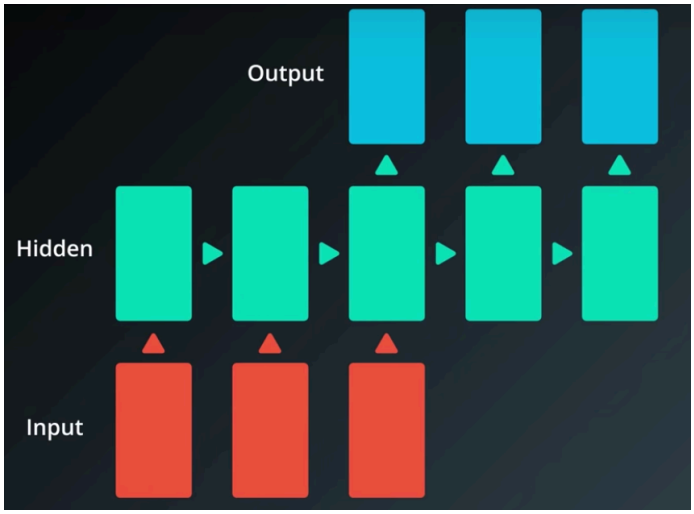
1. Neural Machine Translation (Google Translate, 2016)
2. Speech Recognition
3. …

8.2 Sequence-to-Sequence models (S2S model)

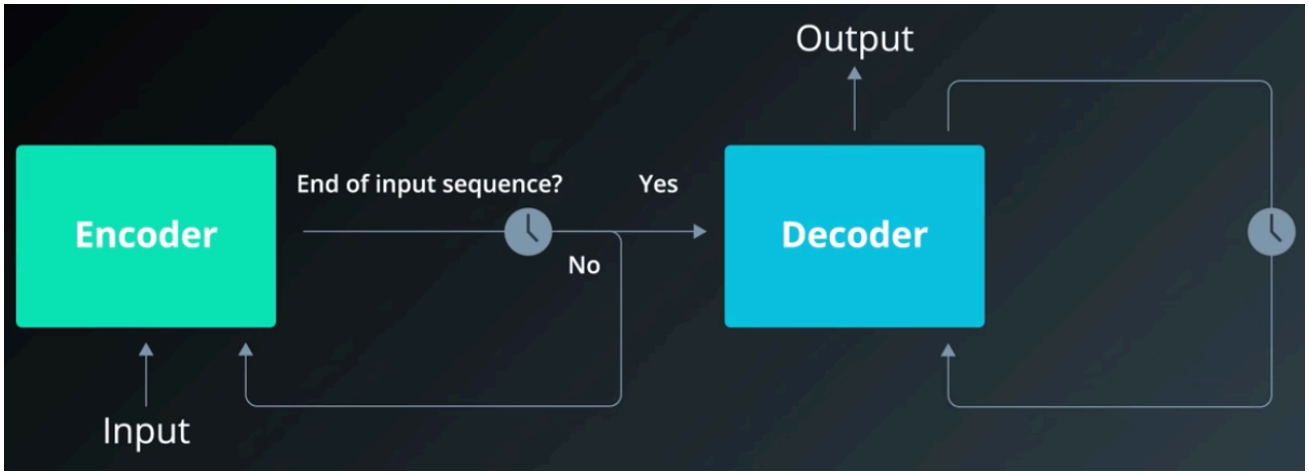
A S2S model takes a sequence of input, and produce a sequence of outputs

Examples:

1. RNN

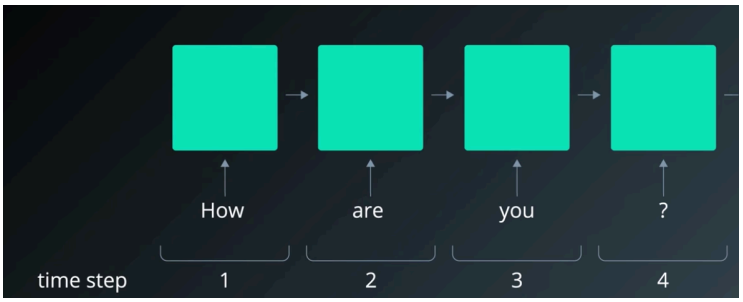


2. Encoder and Decoder wth LSTM cells (CNN can also be used for Encoder and Decoder)

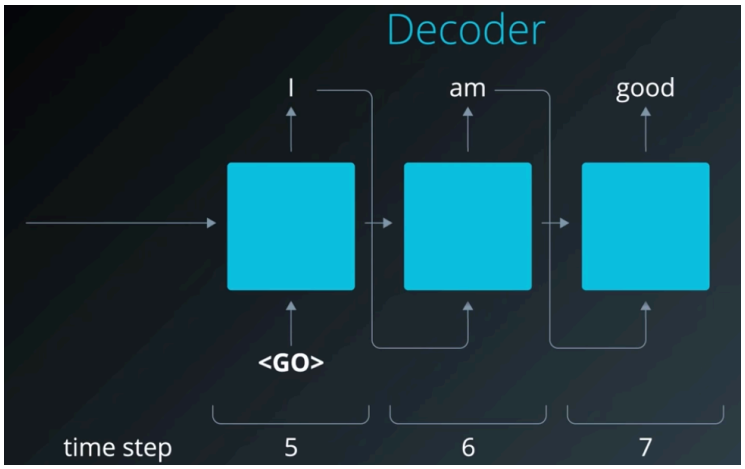


Encoder passes Context to Decoder (Context = State)

Unroll the Encoder:



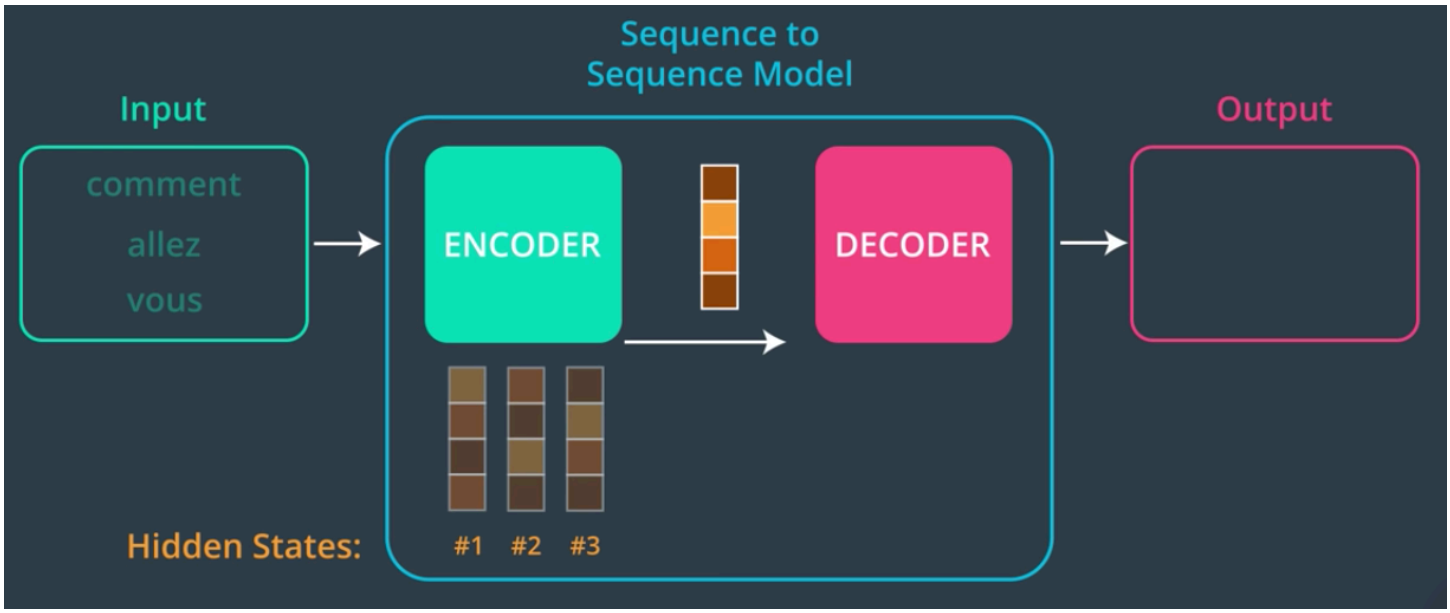
Unroll the Decoder:



Applications:

1. Machine Translation
2. Text summarization

A S2S model consists of an Encoder and a Decoder



An input sequence → Encoder → a single vector (Context/State) → Decoder → An output sequence

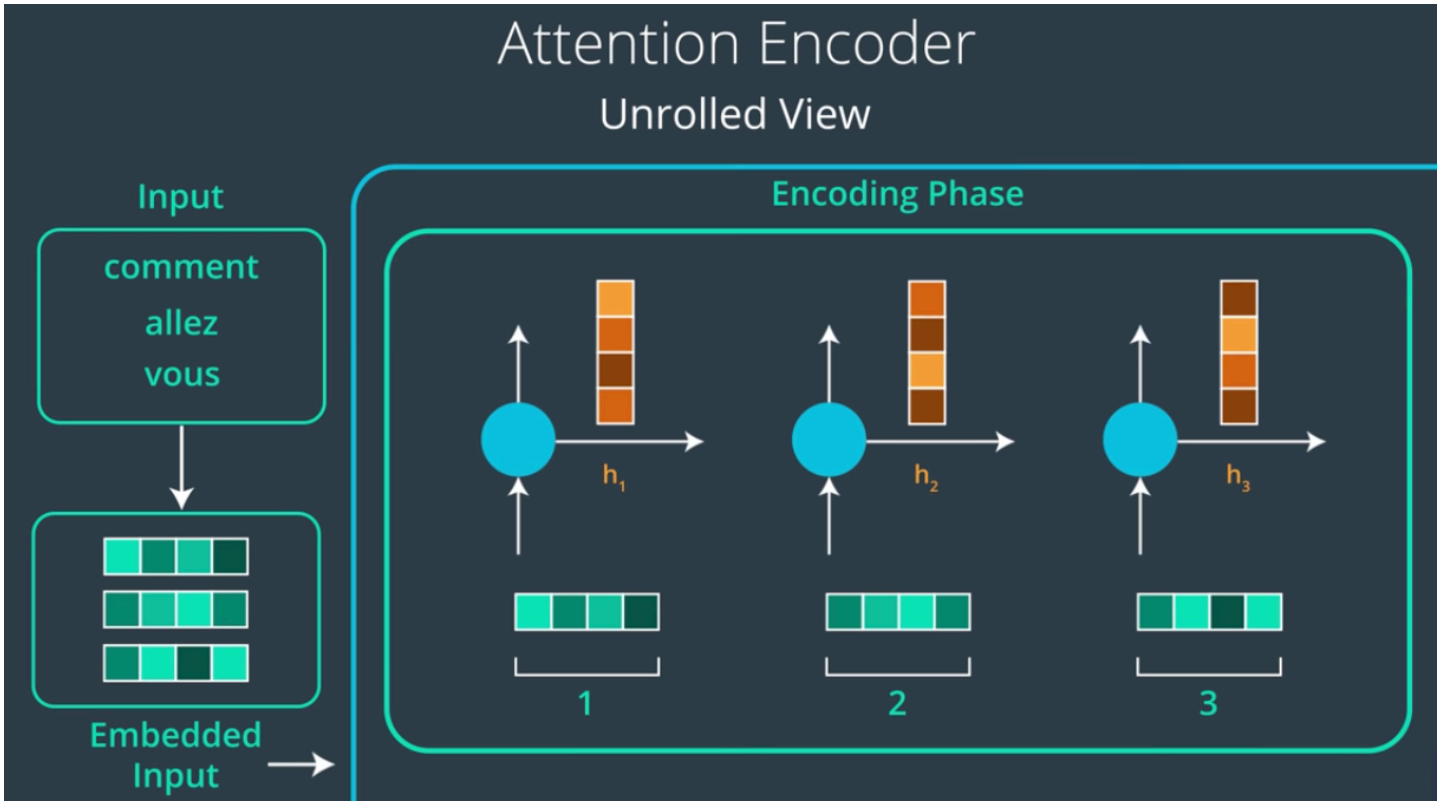
3 words → Encoder →

Challenges:

1. Size of Context vector
    a. Large: overfit to short text
    b. Small: cannot store much information

A S2S model works by feeding 1 element of the input sequence at a time to the encoder

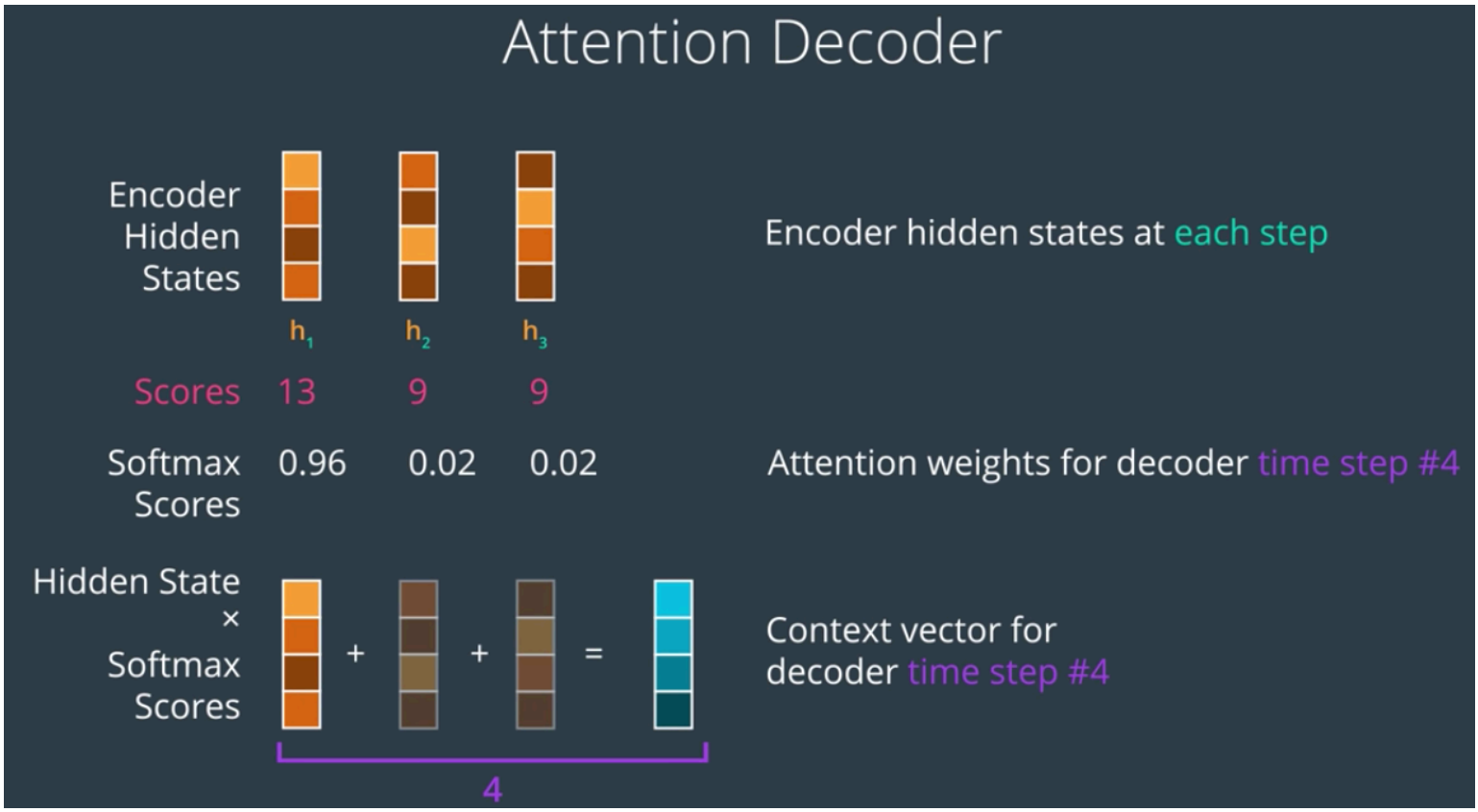
Attention Encoder (“unrolled” view)



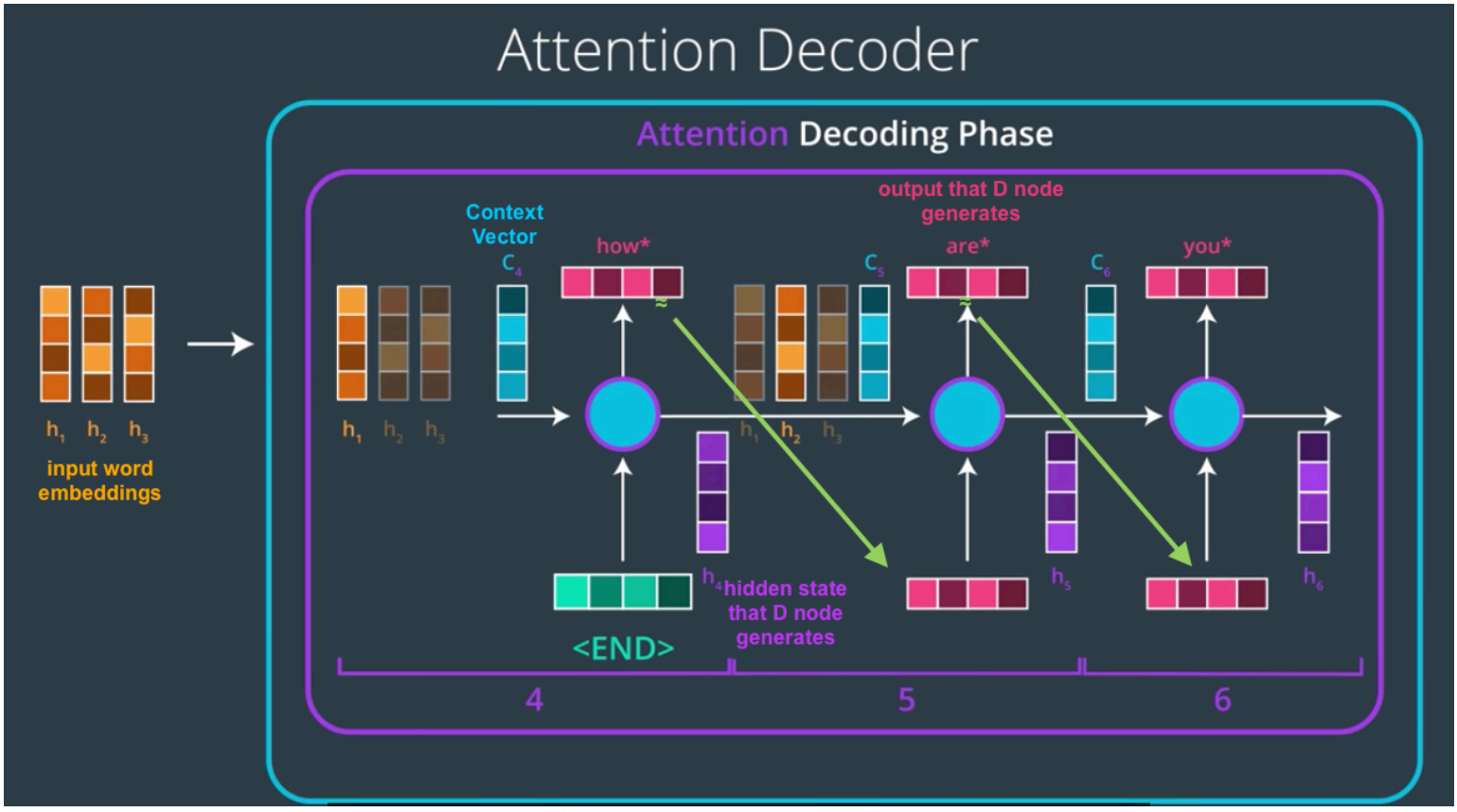
Attention Decoder (“unrolled view)

Vanilla Decoder only see the last hidden state, however, Attention Decoder sees all of the hidden states, it uses a scoring function to score each hidden state in the Context Matrix.

Scores will be fed to a softmax function, ending a set of softmax scores which are positive and sum up to 1

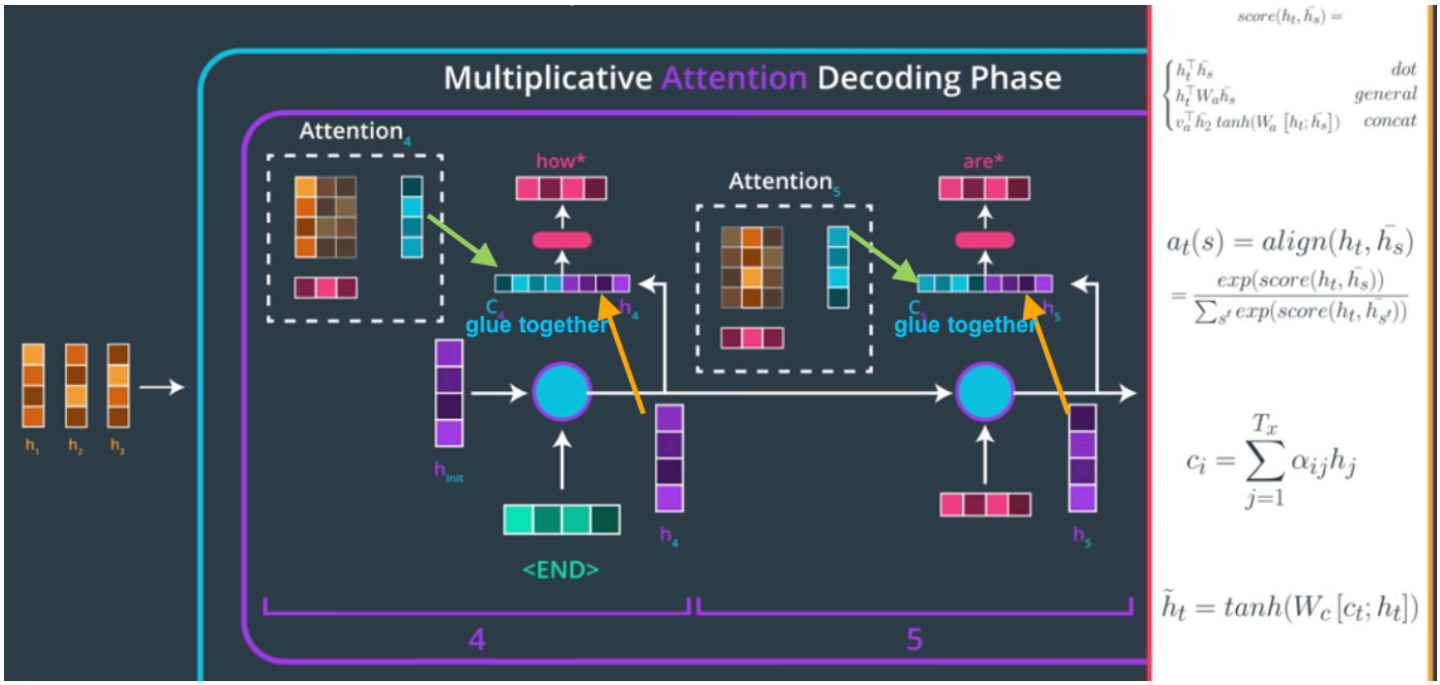


Attention Decoder will look at input embeddings and context vector



(Input word embeddings are typicall 200~300)

Notice there is a * on the “how” word, because this is a just a big picture, let’s look at how does “how” word output:



(This uses multiplicative attention, see below)

Types of Attention:

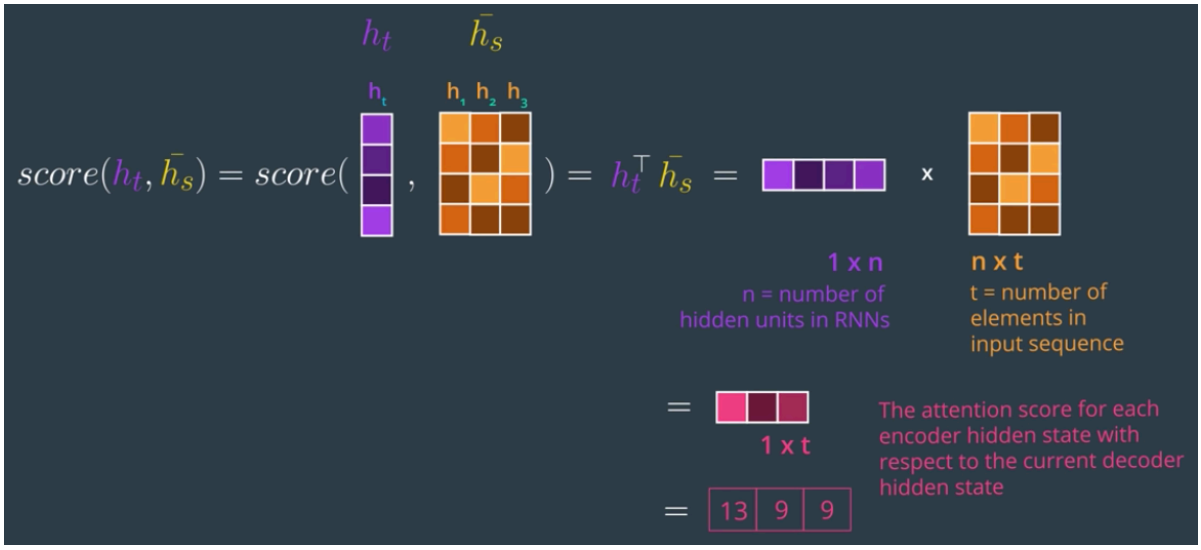
1) Additive Attention (Bahdanau Attention)
   Score function at each timestep: $e_{ij} = v_a^T \tanh\left(W_a\, s_{i-1} + U_a h_j\right)$ *where*
   h_j: *hidden state from the Encoder*,
   s_{i-1}: *hidden state of the decoder in the previous time step*,
   v_a, W_a, U_a: *weight matrices to learn during training*

2) Multiplicative Attention (Luong Attention)
   Score function at each timestep:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W_a} \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_a}[\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

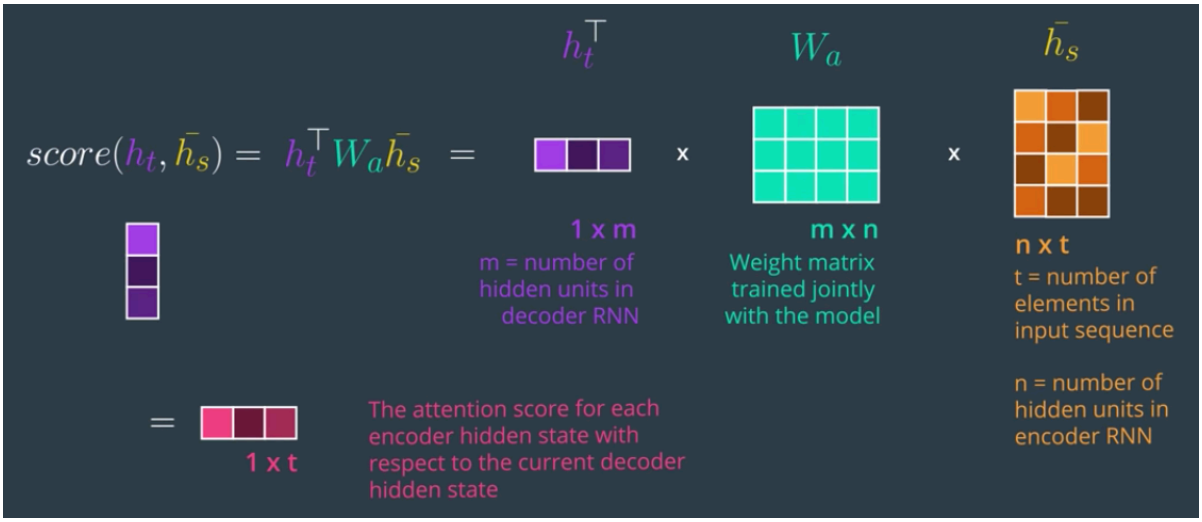
   Where h_t : *hidden state of encoder*; $\bar{h}_s$: *hidden state of decoder*

   a. First scoring method – dot
      Intuition: Dot product of 2 vectors in word-embedding space is a measure of **similarity** between the 2 vectors, since dot product → |a||b|**cos θ**

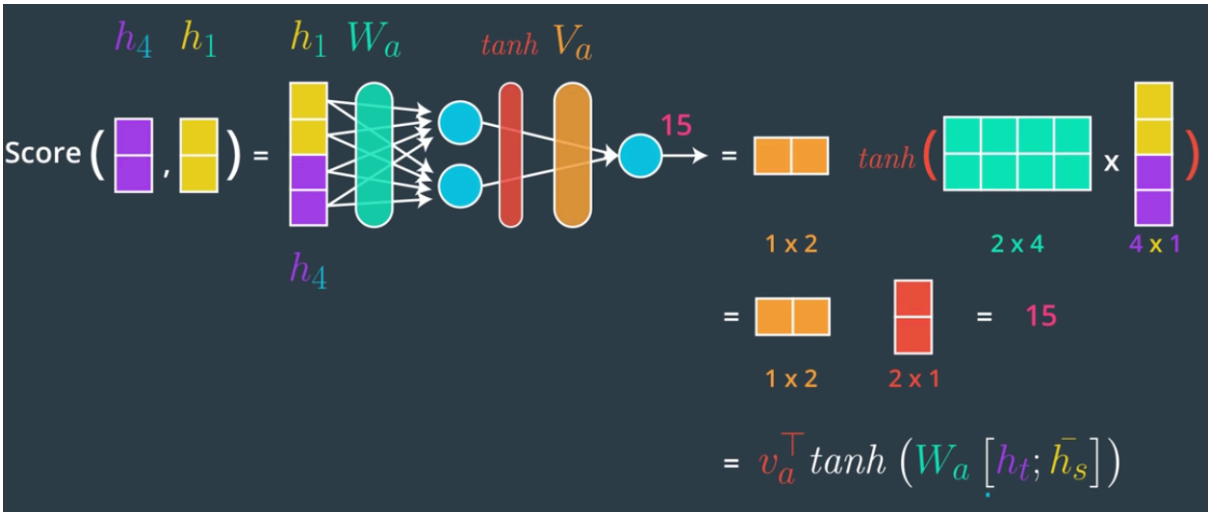


The score function **assumes Encoder and Decoder uses the same embedding space**. **(Drawback)**

b. Second scoring method – general
   Introduce a weight matrix W_a



c. Third scoring method – concat



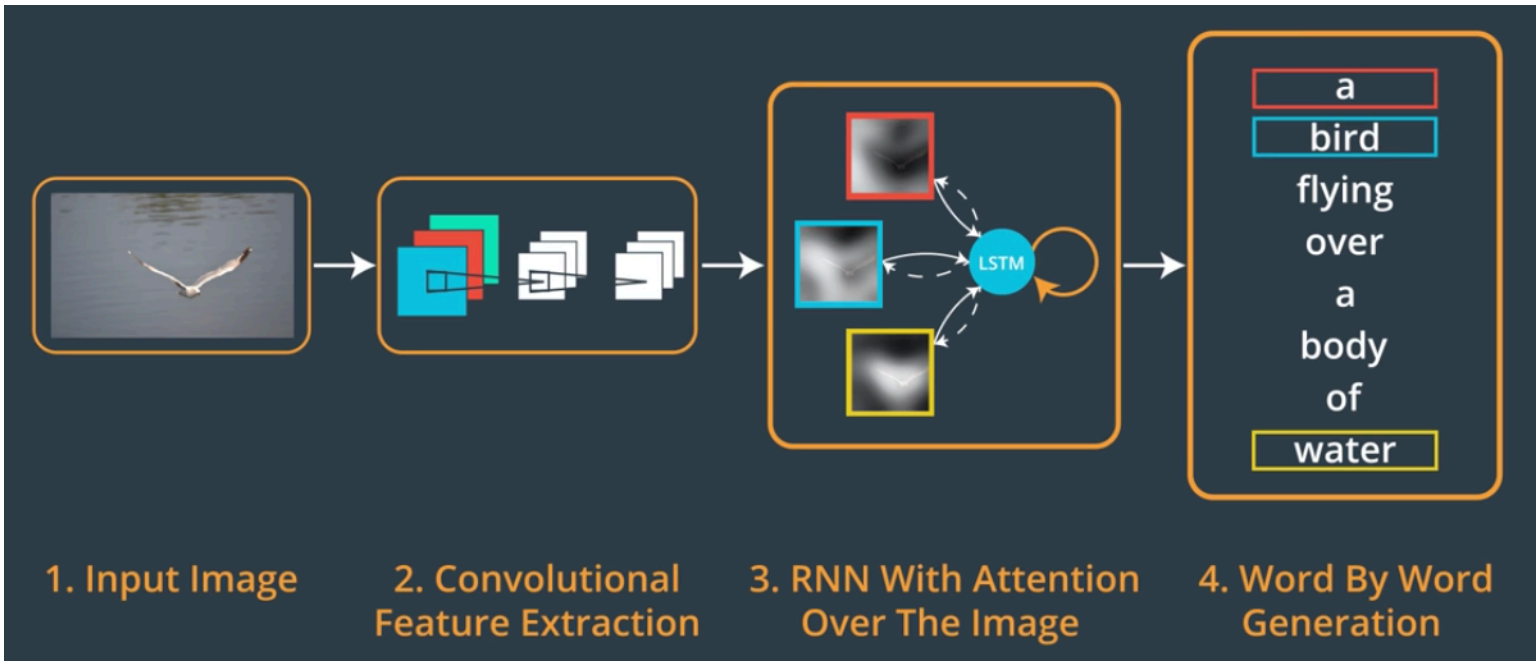
Concat scoring method is similar to Additive Attention, instead, there are only 2 weight matrices;
Concat scoring method uses the current hidden state rather than previous hidden state

These types of attention give the encoder the ability to look at parts of the input sequence

8.3 Paper about Image Captioning: *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*

Dataset source: COCO Common Objects in Context

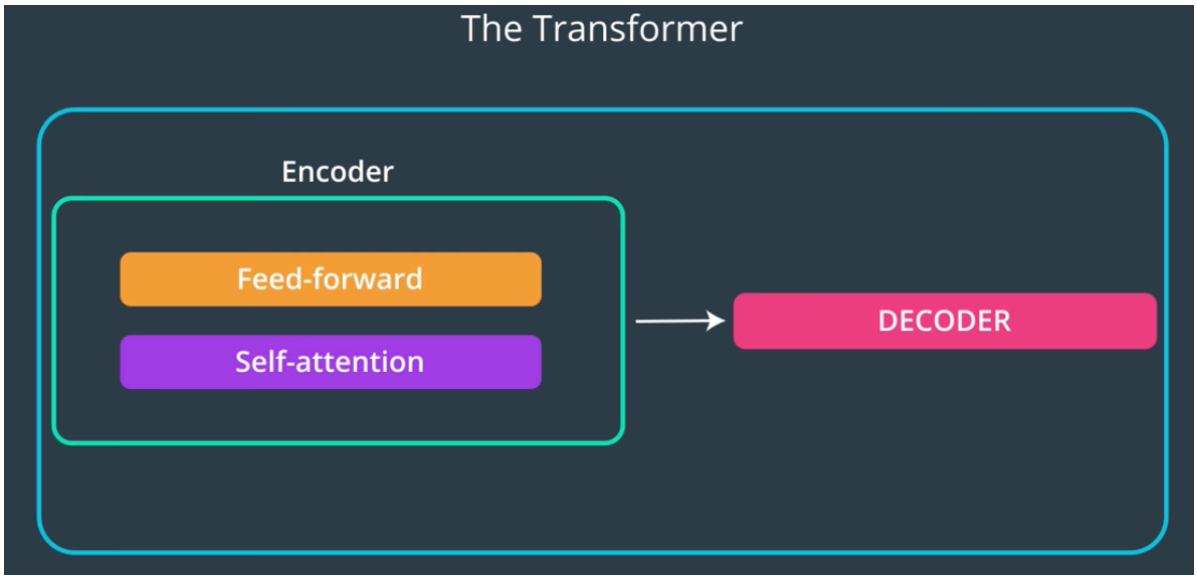
The model used is similar to S2S model, instead, we feed an image rather an a sequence of words to Encoder.



8.4 Paper: *Attention is All you need*

No RNN included in the model, instead, they use a term called "Transformer" allowing parallelization (Stacked Encoder-Decoder inside one Transformer)

Each units in Transformer:

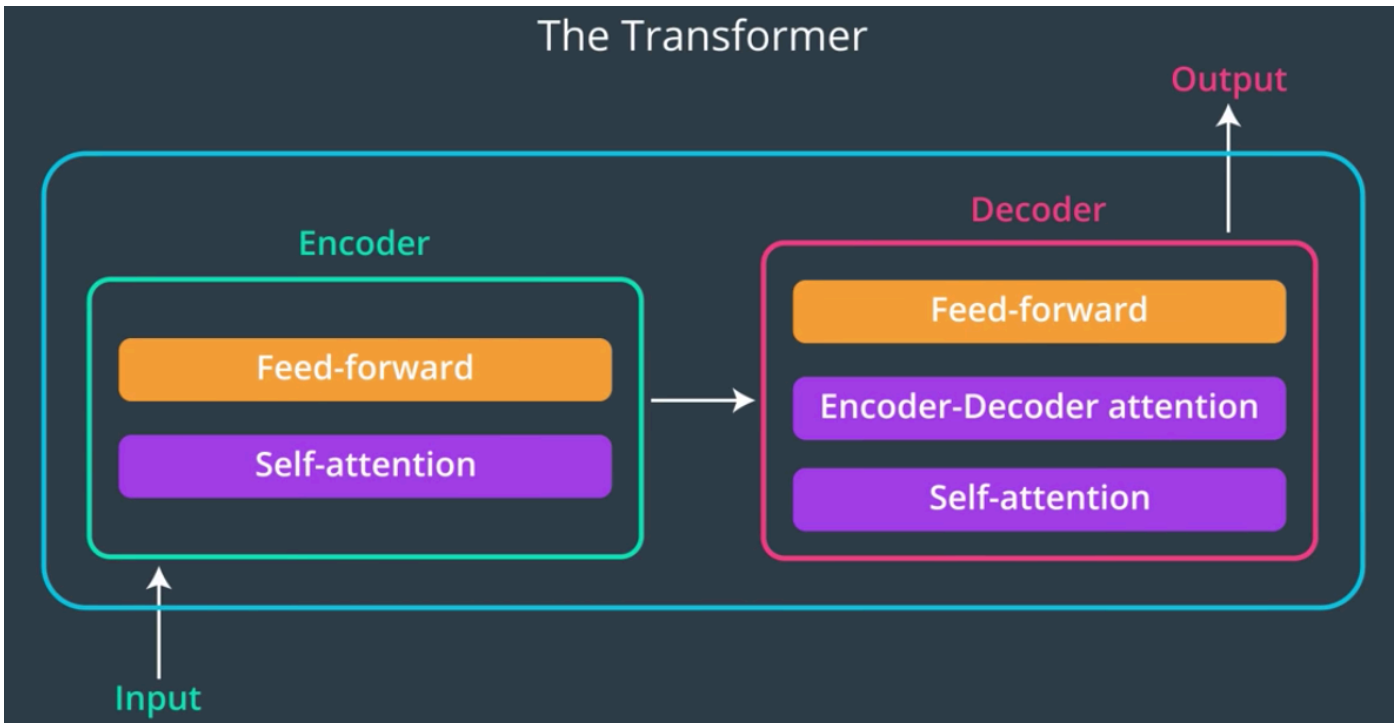


This Attention component helps the Encoder comprehend its input by focusing on other parts of the input sequence that are relevant to each input element it processes.

Idea of “Self-attention” is introduced in Encoder-Decoder each unit.

However, Decoder also contains 2 Attention components:

- Encoder-Decoder attention
- Self-attention



The three **purple components** are Attention Components, they uses Multiplicative Attention and relative scoring methods.

141

5. Generative Adversarial Networks

Lesson 1: Generative Adversarial Networks

Applications of GANs:

1. StackGAN: general bird images matching to the input description
2. iGAN: generate art images based on simple sketches
3. Pix2Pix: image-to-image translation
4. #edges2cats: cat sketch → real cats
5. Day scene photo → Night scene photo
6. CycleGAN: image-to-image translation
7. Create realistic simulating set
8. Part of image → full image
9. Imitation Learning
10. Reinforcement learning

Auto-Regressive Models

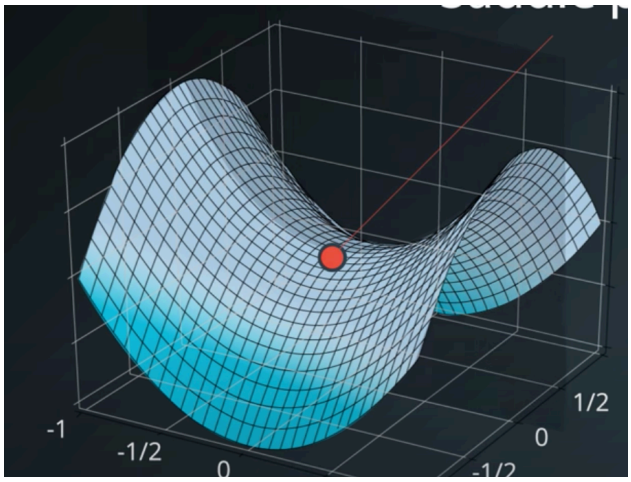
How GANs work:

1. Generator:
    a. Take noises as input
    b. Transform the noise to recognizable structure (e.g. image)
    c. Learn by update the weight based on the gradients from discriminator
2. Discriminator
    a. Trained on with-target data
    b. Output the probability of real or fake label

Game Theory and Equilibrium in GANs

1. GANs is not solving an optimization problem as in NN
2. The cost for the discriminator is the negative of the cost for the generator
3. If we have a score for the GAN, the generator wants to minimize the score, the descriminator wants to maximize the score

Saddle point:



We want to **find an equilibrium where the generator model density = real data density**

$$ratio = \frac{real\ data\ density\ at\ the\ input}{real\ data\ density + generator\ model\ density}$$

This ratio measures how much probability an area comes from the real data rather than generator.

However, we usually train GANs by running 2 optimization algorithms simultaneously, because finding equilibrium involving in high dimensional space is hard.

Activation function:

In discriminator, Leaky ReLU activation function is popular used in hidden layer, because Leaky ReLU makes sure the gradient can flow through the entire architecture. And sigmoid activation at the output makes sure it outputs probabilities.

In generator, popular activation function at output layer is hyperbolic tangent (tanh).

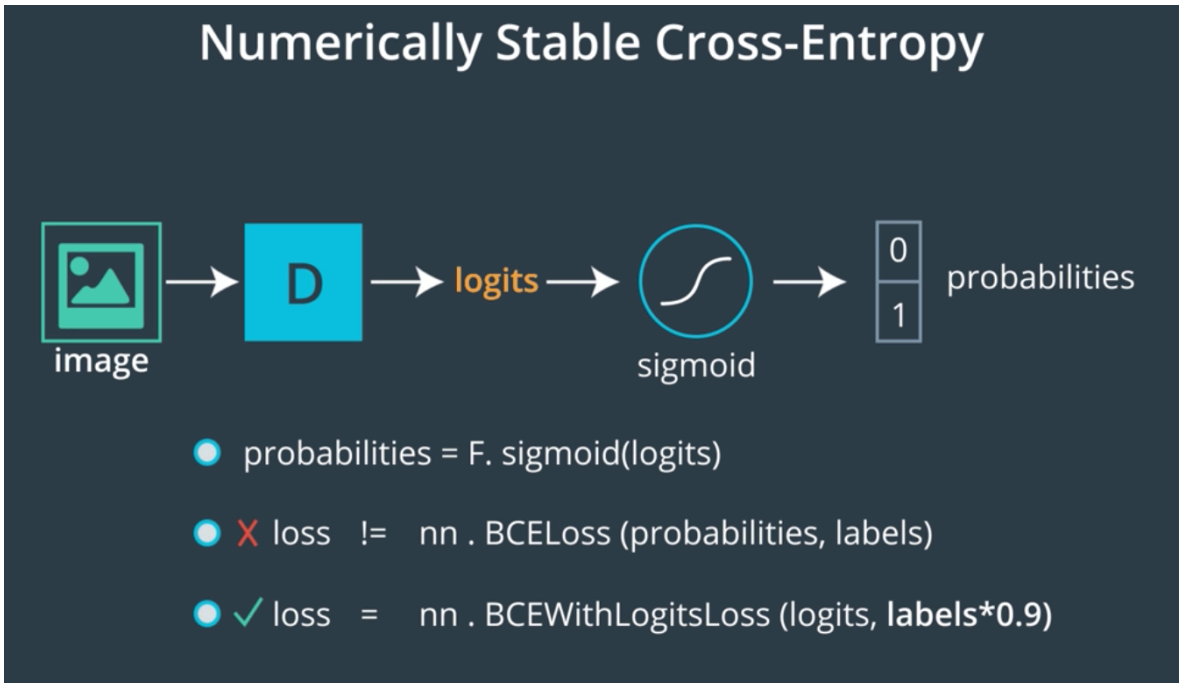
Adam optimizer is good for both discriminator and generator.

Loss function

In discriminator, loss should be computed using the logits (numerically stable cross-entropy).

Logits: values produced by the discriminator right before applying the sigmoid

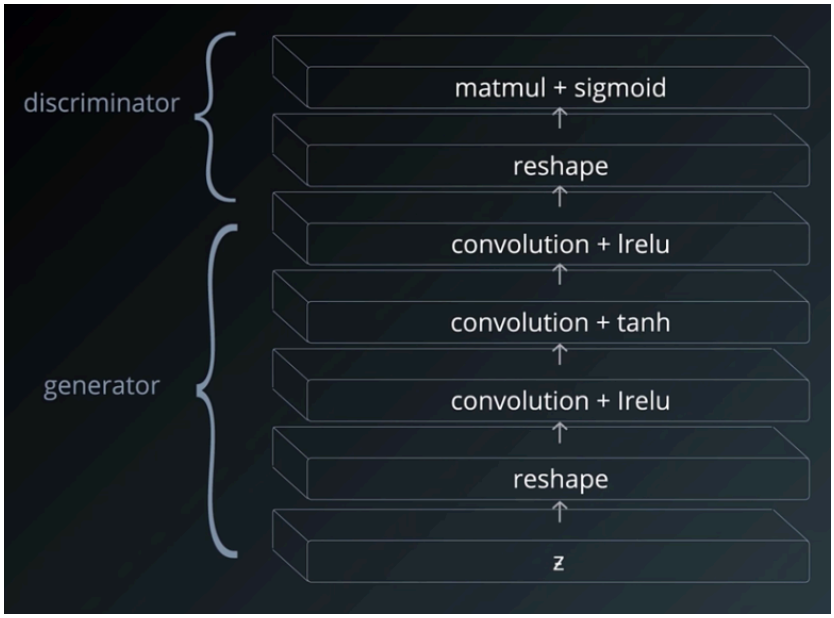
If we use sigmoid(logits) as output, there may be rounding errors when sigmoid(logits) is $\rightarrow$ 0 or 1.



In generator, we need to setup a loss function with flipped labels as in discriminator, because generator wants to maximize the probability of wrong decisions made by discriminator.

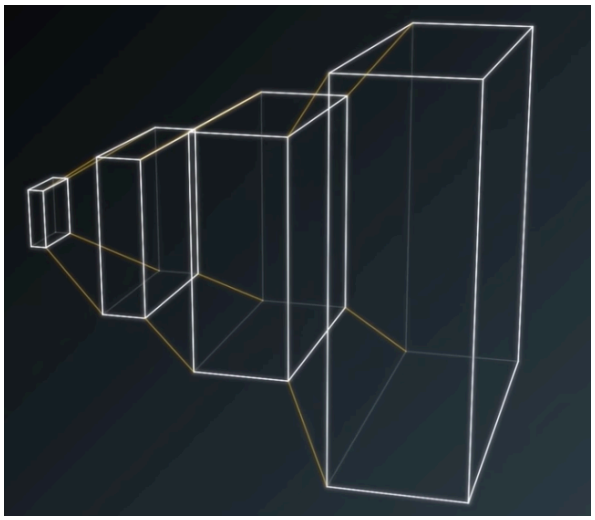
Architecture:

Convolutional networks are used a lot in both D (discriminator) and G (generator)



Both G and D have to have at least 1 hidden layer.

In generator, height, width, depth usually increases as training (opposite in classical CNN in image classifier)



Remember: Transpose Convolutional Layer when we discussed autoencoder ?

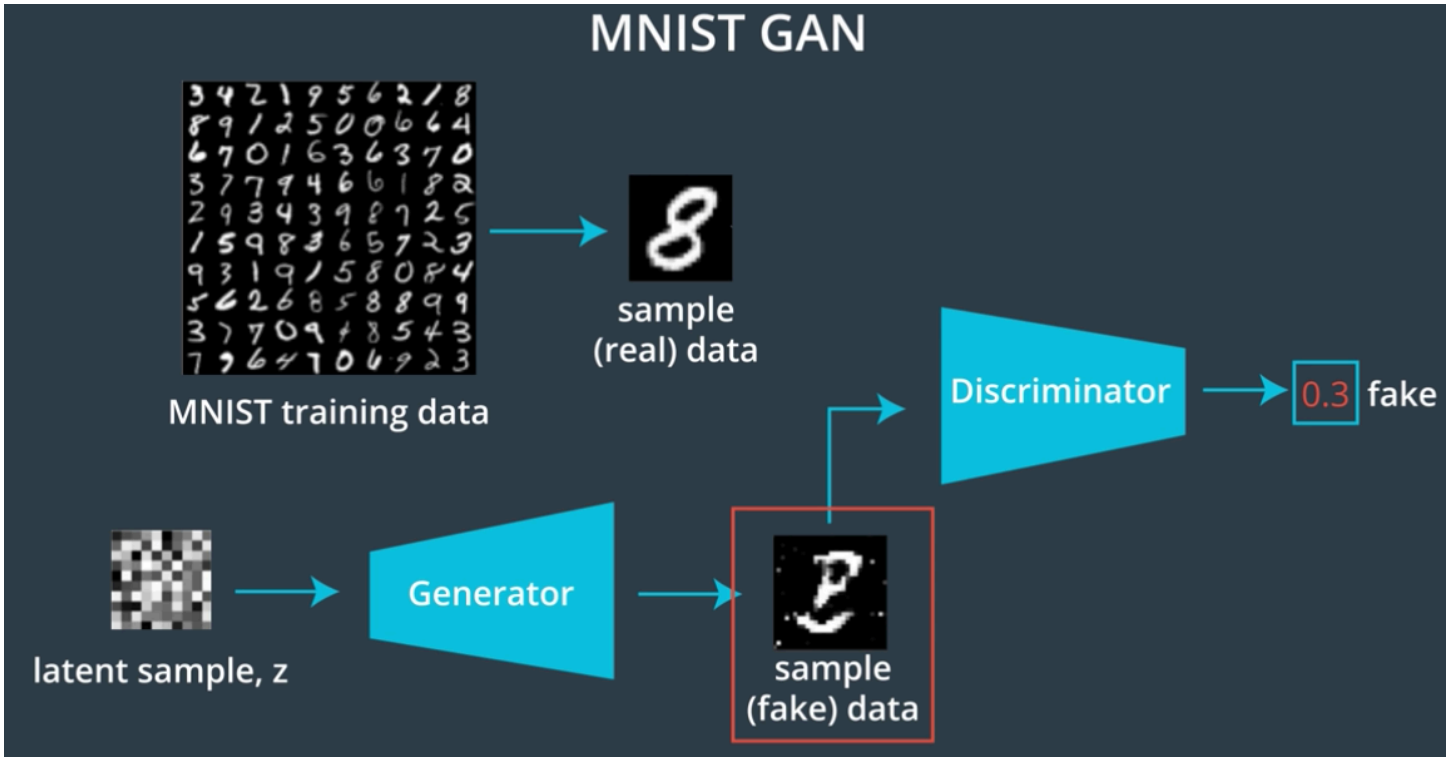
https://pytorch.org/docs/stable/nn.html#convtranspose2d

Batch Normalization is usually used in most of layers

Example: DCGAN

Paper: improved-training-techniques.pdf

Open Notebook: *../ notebooks (filled) / 5. Generative Adversarial Networks / gan-mnist / MNIST_GAN_Exercise.ipynb*



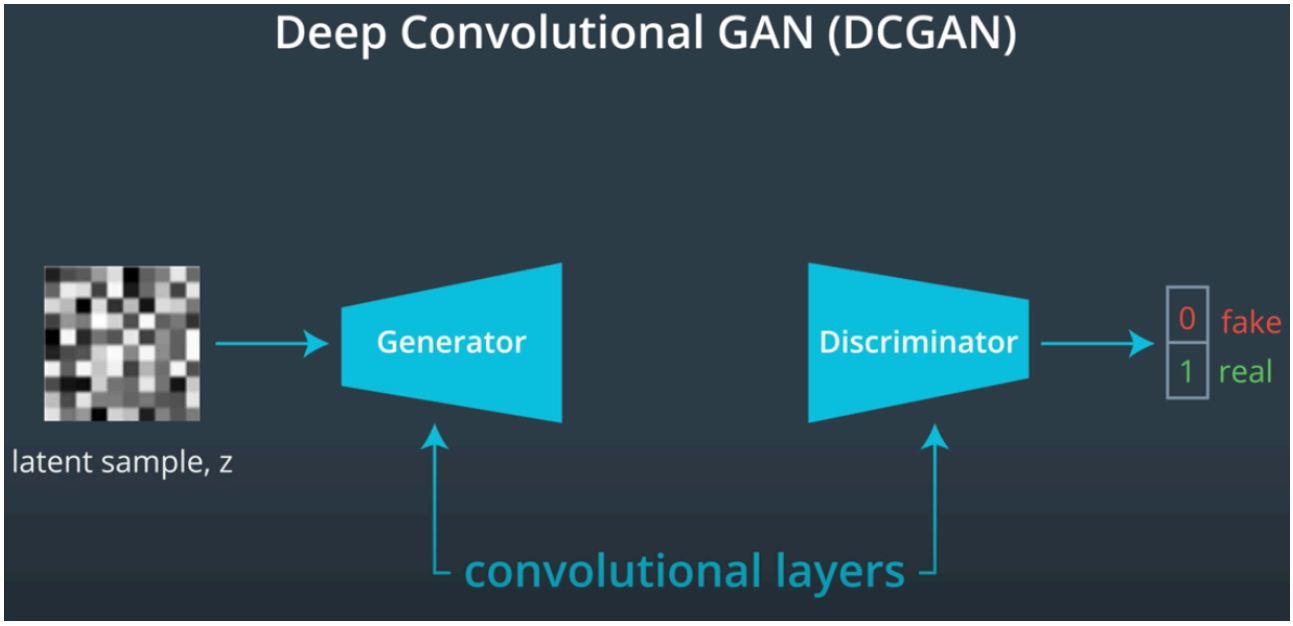
Training process:

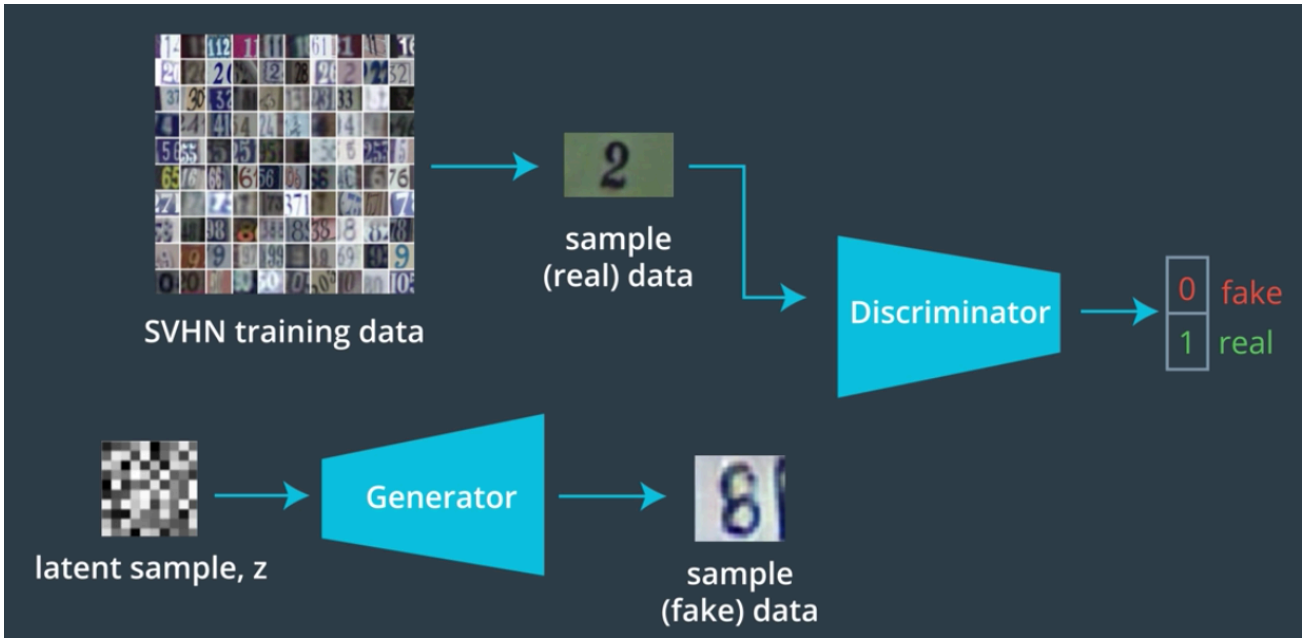
- Discriminator
    1. Losses:
        a. Loss for recognizing real images
        b. Loss for recognizing real “fake” images from generator
    2. Labels:
        a. 0 → fake image (from G)
        b. 1 → real image
- Generator
    1. Loss: loss that aims to trick the Discriminator
    2. Labels
        a. 0 → Discriminator recognizes the fake images
        b. 1 → Discriminator outputs 1 when it sees “fake” images

Lesson 2: Deep Convolutional GANs

Dataset we will use: SVHN (Street View House Numbers Dataset

2.1 DCGAN Architecture



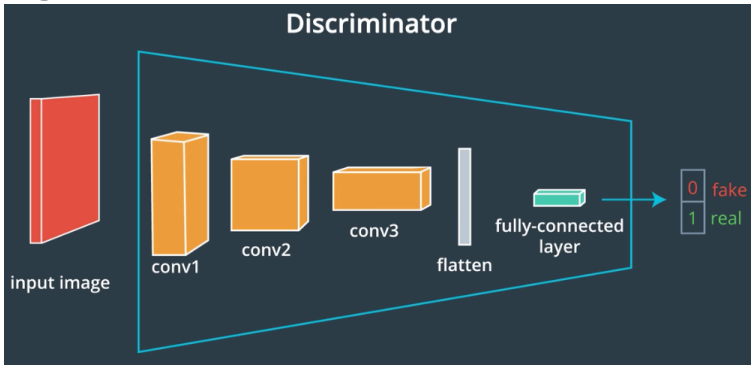


Generator: consists of **transpose convolutional network** that upsamples from latent vector z

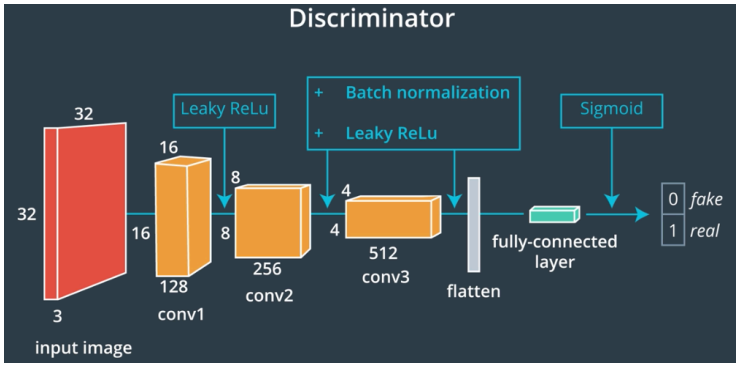
2.2 D and G Network Architecture

Discriminator (D)

Big view:



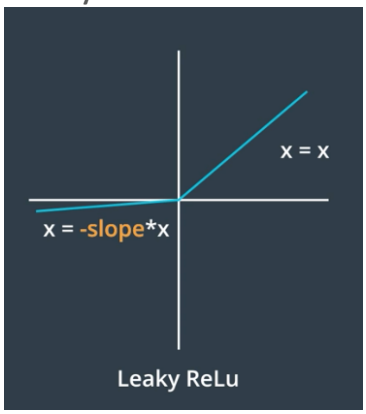
Detailed:



Attention: there are no maxpooling layer after each convolutional layer

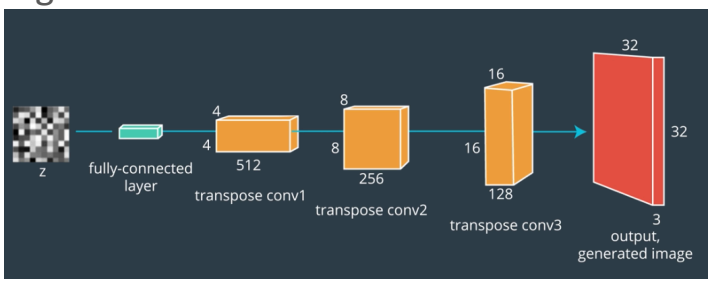
Batch normalization and Leaky ReLU are used in D

- Batch norm layer: convert the output to have mean of 0 and variance of 1, helping the network training faster
- Leaky ReLU activation function: for negative axis, x = -slope * x

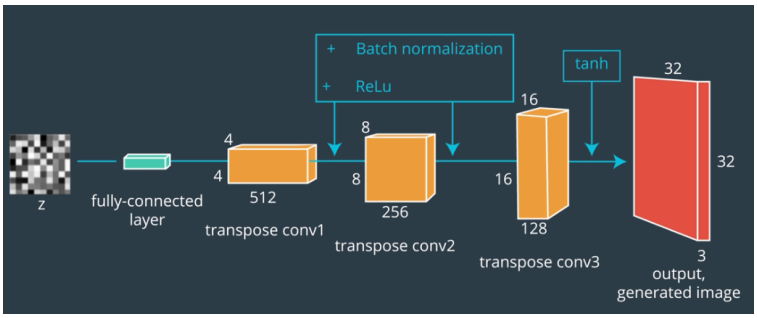


Geneartor (G)

Big view:



Detailed:

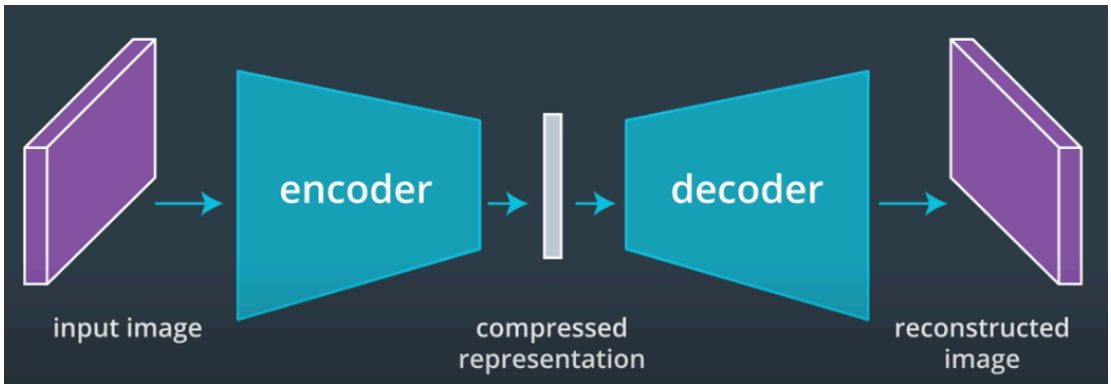


Review/Reminder:

1. There are 2 ways to upsampling:

a. Nearest neighbor interpolation
b. Transpose convoltional layers
    1) (narrow, deep) input → (wide, flat) output

2. Review the Auto-encoder architecture



2.3 Batch Normalization

"Batch" indicates we only use the mean and variance (standard deviation) in the current batch.

Batch Norm compute the batch mean and variance in the output of the previous layer, then each value

$$each\ value = \frac{each\ value - batch\ mean}{batch\ variance}$$

so that each value is between [0, 1]

Advantages:

- Help network converge faster
- Combat **internal convariate shift** discussed in paper
- Allow higher initial learning rates (we can still apply lr decay)
- Make weights easier to intialize
- Make activation function viable
- Simplify the creation of deeper networks
- Provide a bit of regularization
- Reduce oscillations in gradient descent calculations
- Make input more consistent
- Give better results overall

> *In this case, internal covariate shift refers to the change in the distribution of the inputs to different layers. It turns out that training a network is most efficient when the distribution of inputs to each layer is similar!*

Math

Compute the mean μ_B. μ_B is the average value coming out of the previous layer's weighted sum before feeding to activation function.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \quad where\ m\ is\ the\ activation\ length,\ x_i\ is\ each\ entry\ in\ activation$$

Deviation is the result of each entry in activation $x_i - \mu_B$

Compute the variance (mean squared deviation) σ_B^2

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

After batch norm, normalized each will be:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad where\ \epsilon\ is\ a\ tiny\ contant\ to\ avoid\ division\ by\ 0$$

Then we need to post-process the normalized values by two **learnable** parameters γ, β. They can be adjusted like weights. After post-processing, the final each entry that will be passed to activation function of this layer will be

$$\hat{a}_i = \gamma * \hat{x}_i + \beta$$

The value that will be fed to next layer will be

$$\sigma(\hat{a}_i) \quad where \quad \sigma \quad non - linear\ is\ activation\ function$$

Open Notebook: *../ notebooks (filled) / 5. Generative Adversarial Networks / batch-norm / Batch_Normalization.ipynb*

Attentions with Batch Norm:

1. The previous layer (fc, conv, …) should be set $bias = False$ if there is a batch norm layer after it. Because we don't want to add an offset (bias) that will deviate the mean from 0.
2. Batch norm should be applied right before passing to activation function, so always go in forward():

$$x = self.layer(x)$$
$$x = self.batchNorm(x)$$
$$x = F.activation\_function(x)$$

3. While testing the model, need to set the model to .eval() mode, which ensures that the batch norm layers use the population

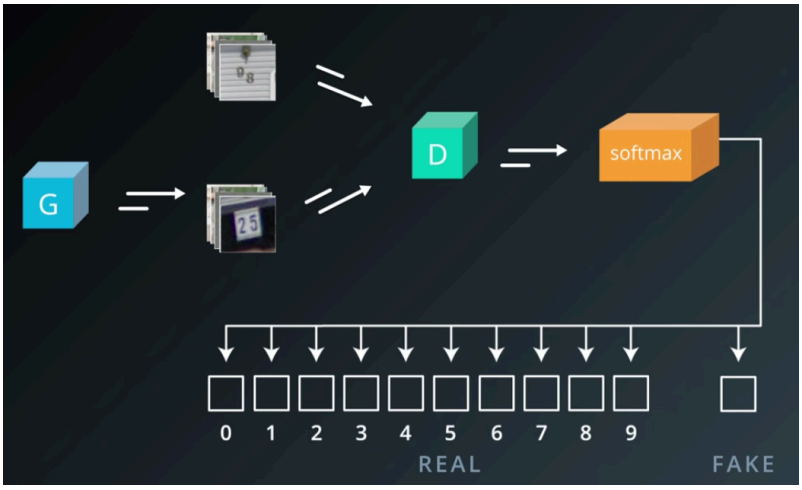
Open Notebook: *../ notebooks (filled) / 5. Generative Adversarial Networks / dcgan-svhn / DCGAN_Exercise.ipynb*

Illumination Model Weaknesses in GANs: small perturbations in image illumination can cause huge accuracy changes

Remember, GANs can also strengthen existing models, or do something worse like attacking existing models and finding security issues

Applications of GANs:

1. Image geneartion and transformation
2. Semi-supervised Learning: learn from a large dataset with a few labeled data
    a. Don't throw away the discriminator after training the GAN
    b. We use D as classifier to classify unlabeled data after training. (In classical GANs, we throw away the D at the end.)

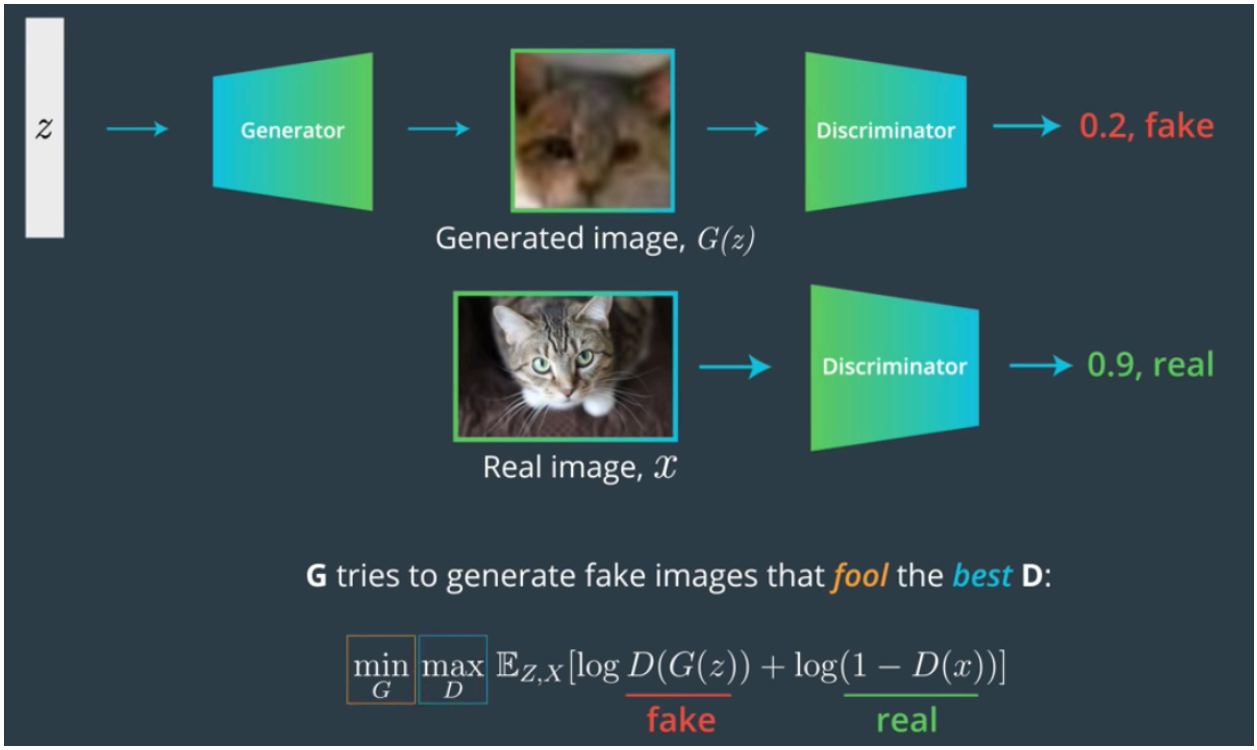


c. Semi-supervised GAN can learn from labeled data, unlabeled data, and fake data from generator G, so there will be 3 errors for D
d. Feature Matching: add a term to the cost function of generator that penalizes mean absolute error between the average value of some set of features on training data (labeled/unlabeled) and average value of some set of features on the generated data from G
e. Feature Matching should be used in Semi-Supervised GANs

Lesson 3: Pix2Pix & CycleGAN

3.1 GAN recap

D loss function = fake loss + real loss



G loss function : treat the D as G's loss function

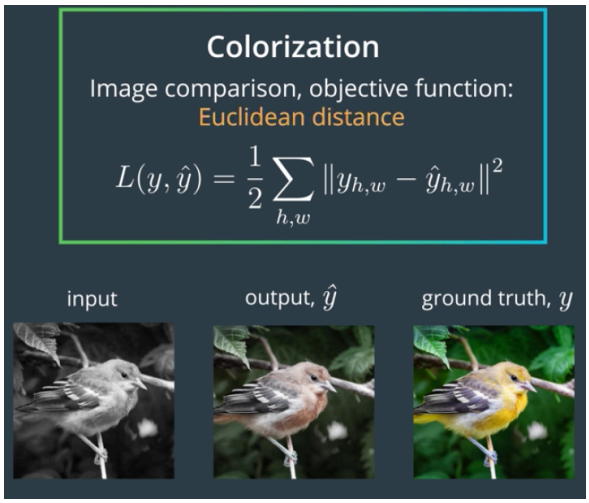
We create a loss function that is learned rather than explicitly defined

3.1 CycleGAN

Image-to-Image Translation: mapping an image from 1 domain to another domain

In classification, cross-entropy loss is an objective function $L(y, \hat{y})$, we want to minimize it as training

In colorization problem (给 input 图片着色问题)

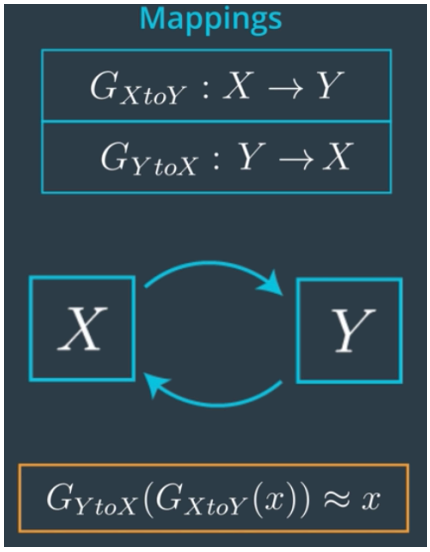


Measuring error pixel-by-pixel by Euclidean distance has weaknesses. Since in RGB space, the middle point between “Red” and “Blue” is another color.

In CycleGANs, we prefer to gather paired data which is hard to get.

Trained on unpaired data

Cycle Consistency



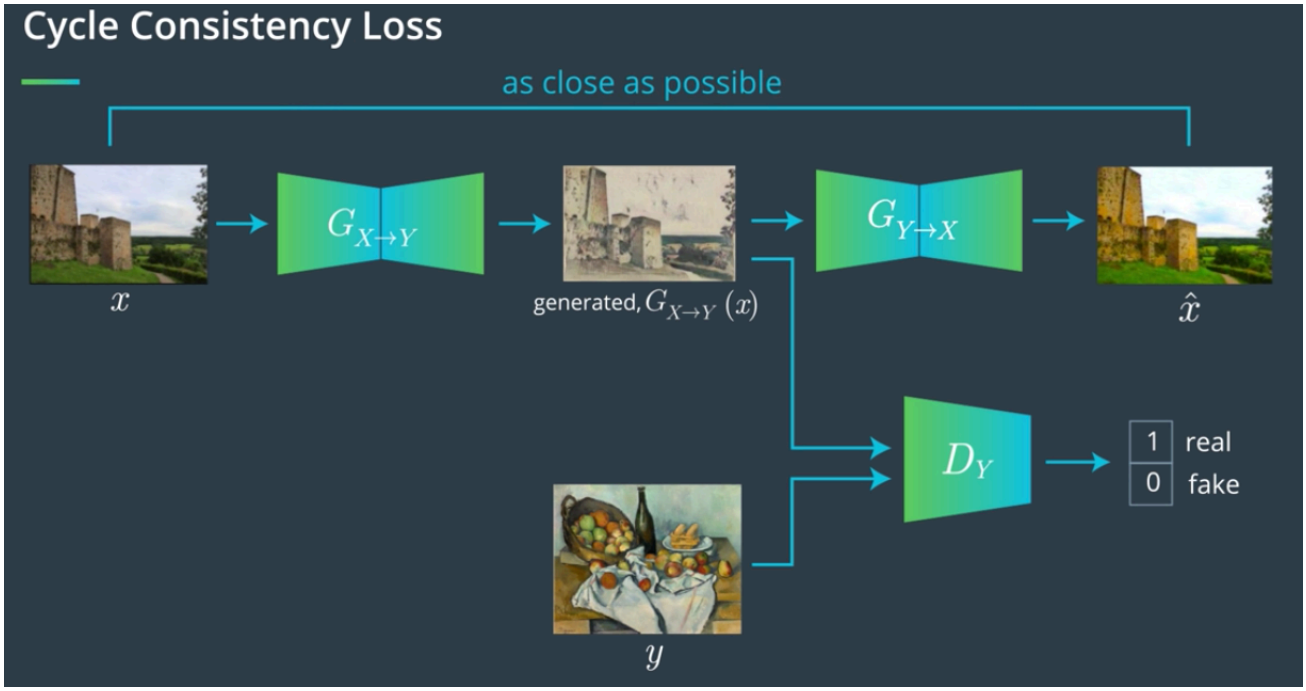
Style Transfer

CycleGAN includes 2 GANs, dx and dy.

1. dy: classify training images from real data and fake data G(x)
2. dx: generate a real image from the x-domain

Cycle consistency loss

1. **Forward cycle consistency**
   Transform image x into domain y (generated, $G_{X\to Y}\ (x)$), then transform y to domain x ($\hat{x}$)
   Its goal is to be x and reconstructed $\hat{x}$ as close as possible



Cycle consistency loss (reconstruction error) = difference between x and reconstructed $\hat{x}$

2. **Backward cycle consistency**

CycleGAN loss = Adversarial loss + forward cycle loss + backward cycle loss

As we go deeper in encoder, input sketch image is transformed into the content of the image rather than details

Applications of CycleGAN:

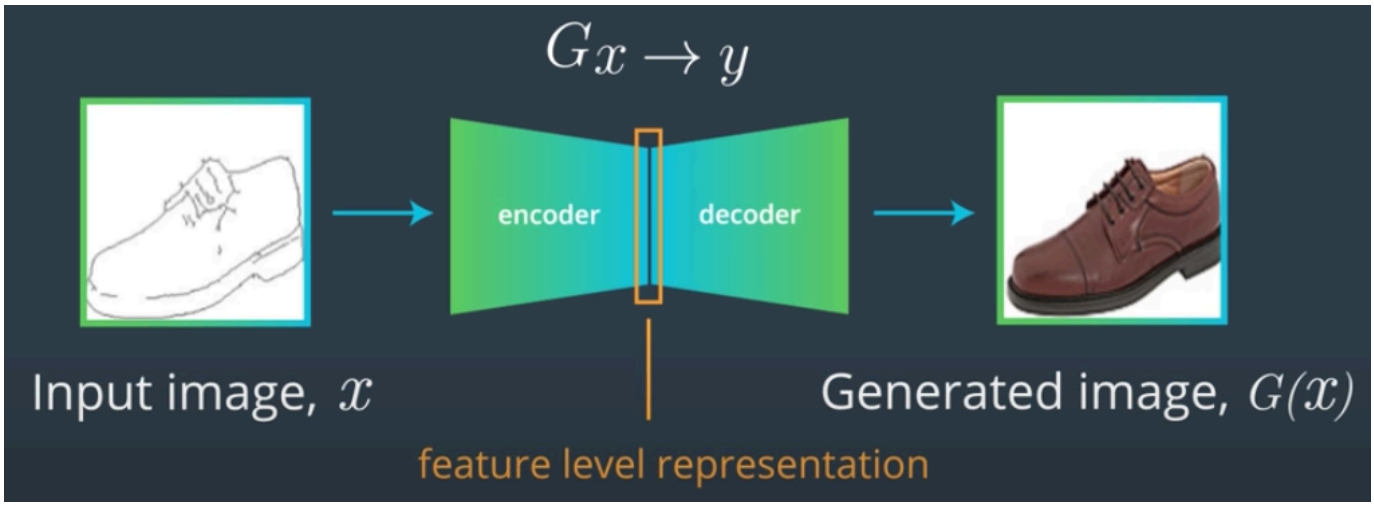
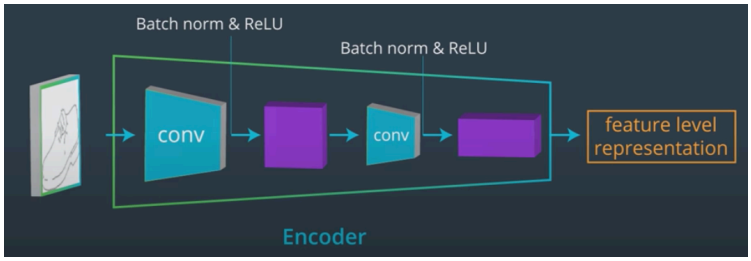
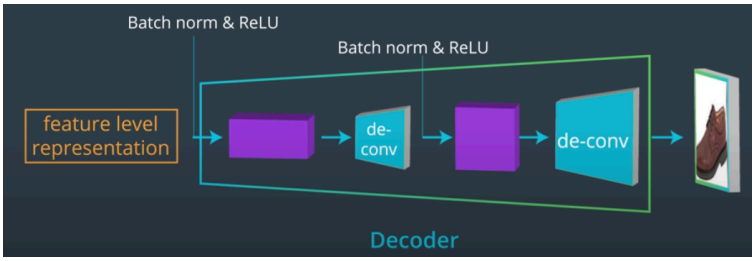
1. Image-to-Image translation: winter → summer, apple → orange, A's voice → B's voice

3.3 Pix2Pix

Pix2Pix architecture uses a conditional GAN to learn a mapping from input image to an output image

Differences to a typical DCGAN

In G, Pix2Pix adds additional layers (encoder) that change an input sketch into a smaller feature level representation

<table>
<tr><td colspan="2"> 
</td></tr>
<tr><td>Encoder
 
</td><td>Decoder:
 
</td></tr>
<tr><td>The encoder distills information about the input image x, and encode it into feature level representation y</td><td>The decoder uses feature level representation of a sketch image to generate a realistic output image.</td></tr>
</table>

	Then it passes its output G(x) to Discriminator

Open Notebook: ***../ notebooks (filled) / 5. Generative Adversarial Networks / cycle-gan / CycleGAN_Exercise.ipynb***

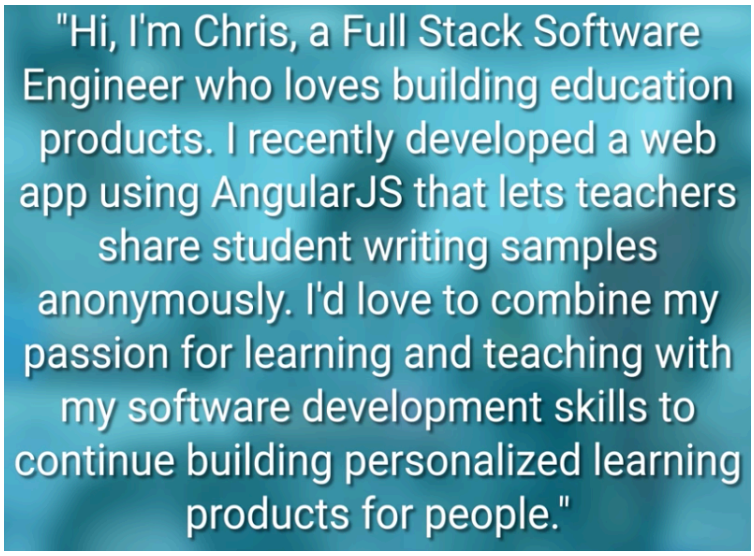
Project: Take 30 Min to Improve your LinkedIn

Elevator Pitch

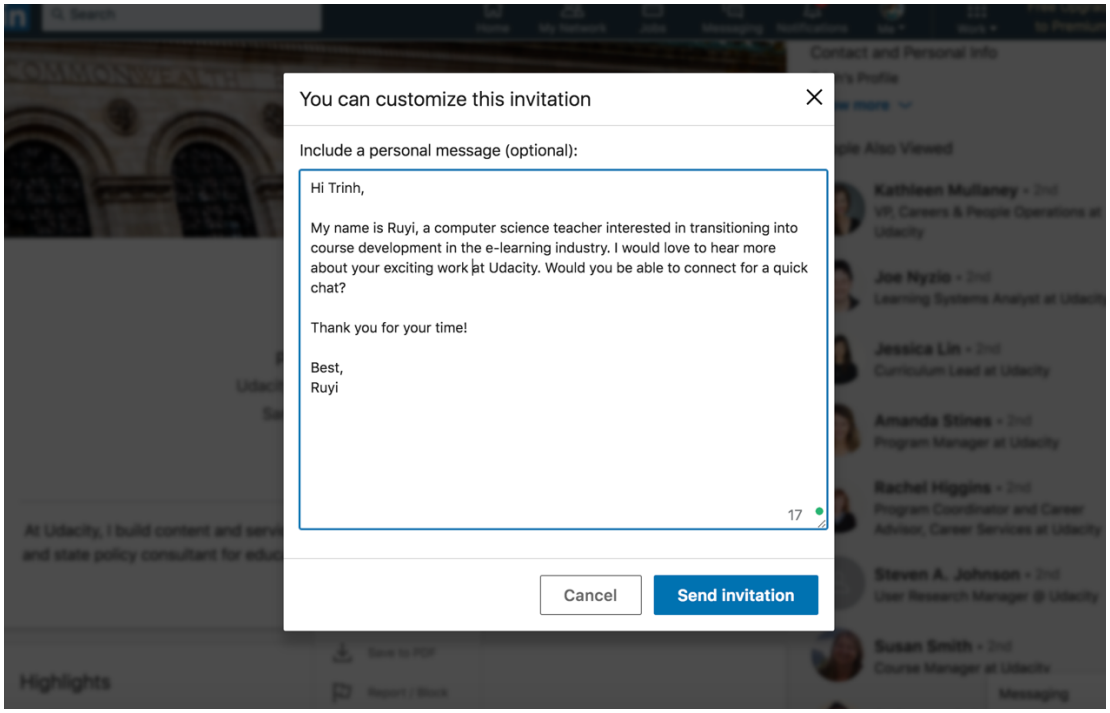
Think about:

1. What did I want employers to know about me?
2. What was I most proud of?

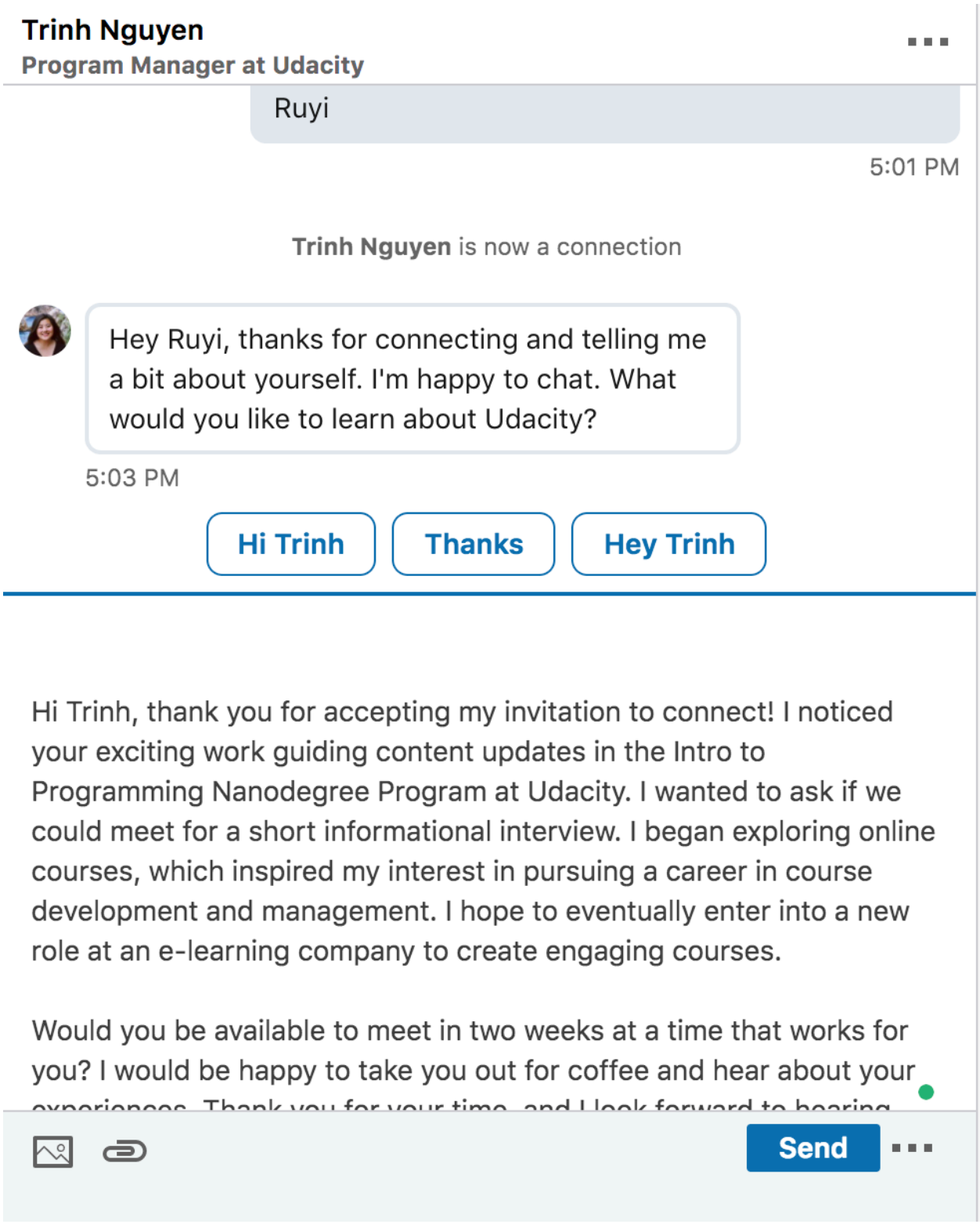
Example:



How to write connection invitation messages:



After connection is established, then maybe ask for a quick information interview



6. Deploying a Model

Lesson 1: Introduction to Deployment

Lesson 2: Building a model using SageMaker

Lesson 3: Deploying and Using a Model

Lesson 4: Hyperparameter Tuning

Lesson 5: Updating a Model

Project: Deploying a Sentiment Analysis Model

Additional Lessons

Lesson 1: Evaluation Metrics

Lesson 2: Regression

Lesson 3: MiniFlow

Lesson 1: Introduction to Keras

Lesson 2: Keras CNNs

Lesson 3: Introduction to TensorFlow