Course summary

Here are the course summary as its given on the course link:

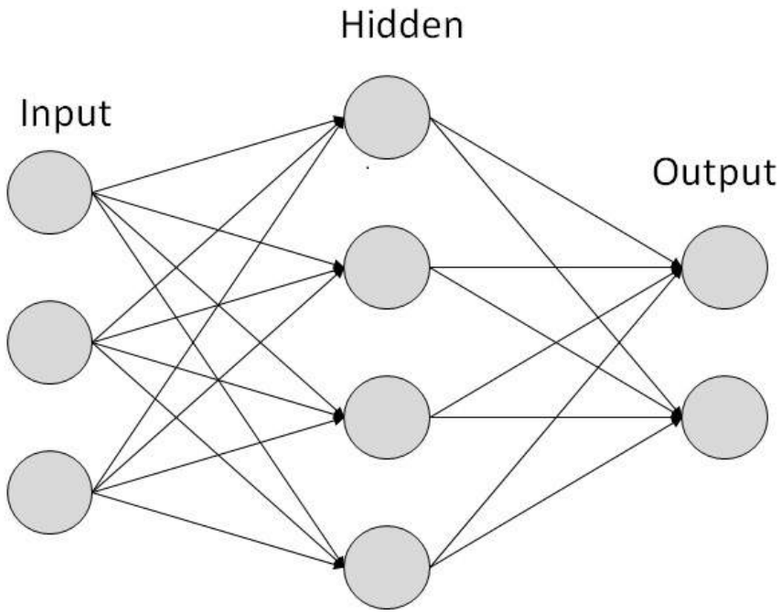
> If you want to break into cutting-edge AI, this course will help you do so. Deep learning engineers are highly sought after, and mastering deep learning will give you numerous new career opportunities. Deep learning is also a new "superpower" that will let you build AI systems that just weren't possible a few years ago.
>
> In this course, you will learn the foundations of deep learning. When you finish this class, you will:
>
> - Understand the major technology trends driving Deep Learning
> - Be able to build, train and apply fully connected deep neural networks
> - Know how to implement efficient (vectorized) neural networks
> - Understand the key parameters in a neural network's architecture
>
> This course also teaches you how Deep Learning actually works, rather than presenting only a cursory or surface-level description. So after completing it, you will be able to apply deep learning to a your own applications. If you are looking for a job in AI, after this course you will also be able to answer basic interview questions.

Introduction to deep learning

> Be able to explain the major trends driving the rise of deep learning, and understand where and how it is applied today.

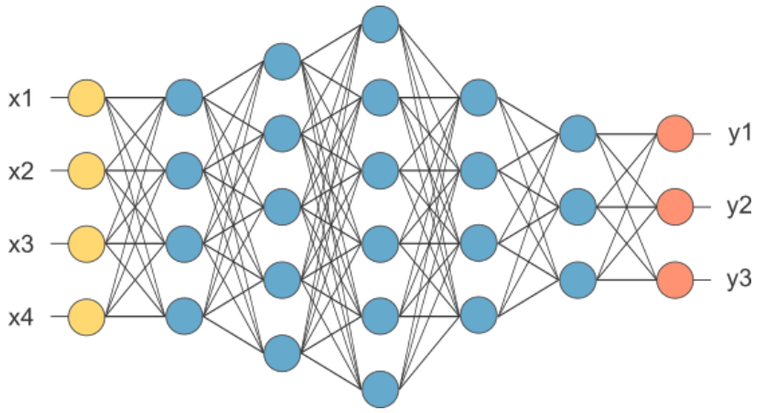
What is a (Neural Network) NN?

- Single neuron == linear regression without applying activation(perceptron)
- Basically a single neuron will calculate weighted sum of input(W.T*X) and then we can set a threshold to predict output in a perceptron. If weighted sum of input cross the threshold, perceptron fires and if not then perceptron doesn't predict.
- Perceptron can take real values input or boolean values.
- Actually, when w·x+b=0 the perceptron outputs 0.
- Disadvantage of perceptron is that it only output binary values and if we try to give small change in weight and bais then perceptron can flip the output. We need some system which can modify the output slightly according to small change in weight and bias. Here comes sigmoid function in picture.
- If we change perceptron with a sigmoid function, then we can make slight change in output.
- e.g. output in perceptron = 0, you slightly changed weight and bias, output becomes = 1 but actual output is 0.7. In case of sigmoid, output1 = 0, slight change in weight and bias, output = 0.7.
- If we apply sigmoid activation function then Single neuron will act as Logistic Regression.
- we can understand difference between perceptron and sigmoid function by looking at sigmoid function graph.
- Simple NN graph:



  - Image taken from tutorialspoint.com
- RELU stands for rectified linear unit is the most popular activation function right now that makes deep NNs train faster now.
- Hidden layers predicts connection between inputs automatically, thats what deep learning is good at.

- Deep NN consists of more hidden layers (Deeper layers)



  - Image taken from opennn.net

- Each Input will be connected to the hidden layer and the NN will decide the connections.
- Supervised learning means we have the (X,Y) and we need to get the function that maps X to Y.

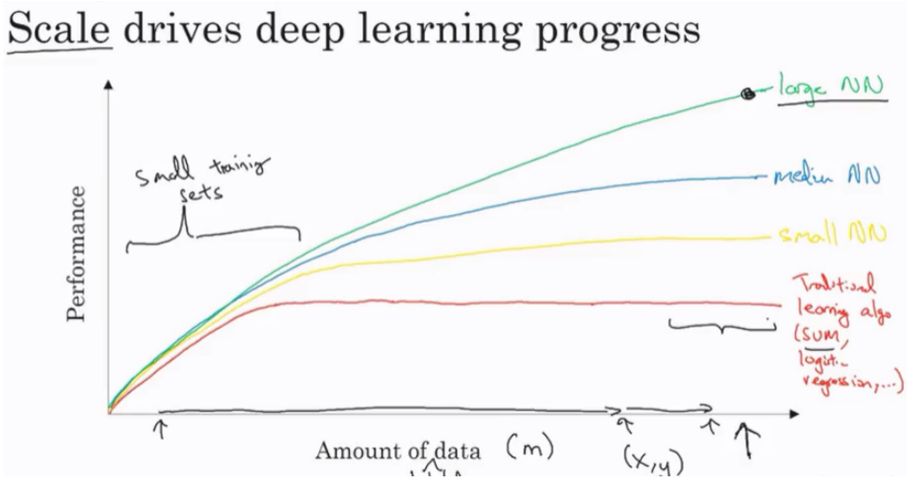
Supervised learning with neural networks

- Different types of neural networks for supervised learning which includes:
  - CNN or convolutional neural networks (Useful in computer vision)
  - RNN or Recurrent neural networks (Useful in Speech recognition or NLP)
  - Standard NN (Useful for Structured data)
  - Hybrid/custom NN or a Collection of NNs types
- Structured data is like the databases and tables.
- Unstructured data is like images, video, audio, and text.
- Structured data gives more money because companies relies on prediction on its big data.

Why is deep learning taking off?

- Deep learning is taking off for 3 reasons:

  i. Data:
  - Using this image we can conclude:



  - For small data NN can perform as Linear regression or SVM (Support vector machine)
  - For big data a small NN is better that SVM
  - For big data a big NN is better that a medium NN is better that small NN.
  - Hopefully we have a lot of data because the world is using the computer a little bit more
    - Mobiles
    - IOT (Internet of things)

  ii. Computation:
  - GPUs.
  - Powerful CPUs.
  - Distributed computing.
  - ASICs

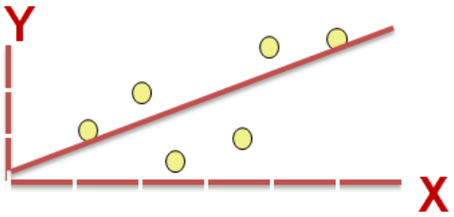
  iii. Algorithm:
  a. Creative algorithms has appeared that changed the way NN works.
    - For example using RELU function is so much better than using SIGMOID function in training a NN because it helps with the vanishing gradient problem.

Neural Networks Basics

> Learn to set up a machine learning problem with a neural network mindset. Learn to use vectorization to speed up your models.

Binary classification

- Mainly he is talking about how to do a logistic regression to make a binary classifier.



  - Image taken from 3.bp.blogspot.com
- He talked about an example of knowing if the current image contains a cat or not.
- Here are some notations:
  - `M is the number of training vectors`
  - `Nx is the size of the input vector`
  - `Ny is the size of the output vector`
  - `X(1) is the first input vector`
  - `Y(1) is the first output vector`
  - `X = [x(1) x(2).. x(M)]`
  - `Y = (y(1) y(2).. y(M))`
- We will use python in this course.
- In NumPy we can make matrices and make operations on them in a fast and reliable time.

Logistic regression

- Algorithm is used for classification algorithm of 2 classes.
- Equations:
  - Simple equation: `y = wx + b`
  - If x is a vector: `y = w(transpose)x + b`
  - If we need y to be in between 0 and 1 (probability): `y = sigmoid(w(transpose)x + b)`
  - In some notations this might be used: `y = sigmoid(w(transpose)x)`
    - While `b` is `w0` of `w` and we add `x0 = 1`. but we won't use this notation in the course (Andrew said that the first notation is better).
- In binary classification `Y` has to be between `0` and `1`.
- In the last equation `w` is a vector of `Nx` and `b` is a real number

Logistic regression cost function

- First loss function would be the square root error: `L(y',y) = 1/2 (y' - y)^2`
  - But we won't use this notation because it leads us to optimization problem which is non convex, means it contains local optimum points.
- This is the function that we will use: `L(y',y) = - (y*log(y') + (1-y)*log(1-y'))`
- To explain the last function lets see:
  - if `y = 1` ==> `L(y',1) = -log(y')` ==> we want `y'` to be the largest ==> `y'` biggest value is 1
  - if `y = 0` ==> `L(y',0) = -log(1-y')` ==> we want `1-y'` to be the largest ==> `y'` to be smaller as possible because it can only has 1 value.
- Then the Cost function will be: `J(w,b) = (1/m) * Sum(L(y'[i],y[i]))`
- The loss function computes the error for a single training example; the cost function is the average of the loss functions of the entire training set.

Gradient Descent

- We want to predict `w` and `b` that minimize the cost function.
- Our cost function is convex.
- First we initialize `w` and `b` to 0,0 or initialize them to a random value in the convex function and then try to improve the values the reach minimum value.
- In Logistic regression people always use 0,0 instead of random.
- The gradient decent algorithm repeats: `w = w - alpha * dw` where alpha is the learning rate and `dw` is the derivative of `w` (Change to `w`) The derivative is also the slope of `w`
- Looks like greedy algorithms. the derivative give us the direction to improve our parameters.

- The actual equations we will implement:
  - `w = w - alpha * d(J(w,b) / dw)` (how much the function slopes in the w direction)
  - `b = b - alpha * d(J(w,b) / db)` (how much the function slopes in the d direction)

Derivatives

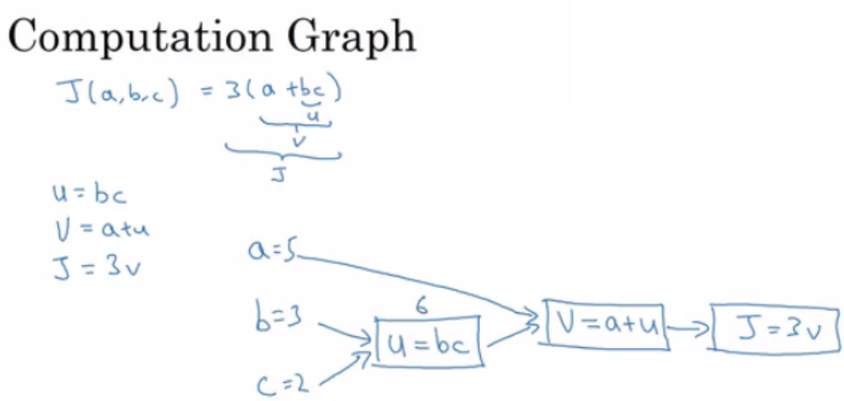
- We will talk about some of required calculus.
- You don't need to be a calculus geek to master deep learning but you'll need some skills from it.
- Derivative of a linear line is its slope.
  - ex. `f(a) = 3a` `d(f(a))/d(a) = 3`
  - if `a = 2` then `f(a) = 6`
  - if we move a a little bit `a = 2.001` then `f(a) = 6.003` means that we multiplied the derivative (Slope) to the moved area and added it to the last result.

More Derivatives examples

- `f(a) = a^2` `==>` `d(f(a))/d(a) = 2a`
  - `a = 2` `==>` `f(a) = 4`
  - `a = 2.0001` `==>` `f(a) = 4.0004` approx.
- `f(a) = a^3` `==>` `d(f(a))/d(a) = 3a^2`
- `f(a) = log(a)` `==>` `d(f(a))/d(a) = 1/a`
- To conclude, Derivative is the slope and slope is different in different points in the function thats why the derivative is a function.

Computation graph

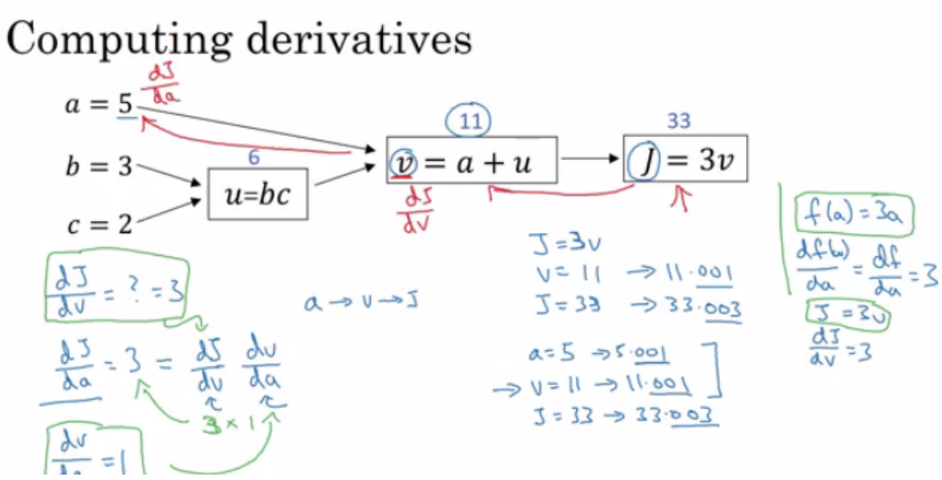
- Its a graph that organizes the computation from left to right.



  -

Derivatives with a Computation Graph

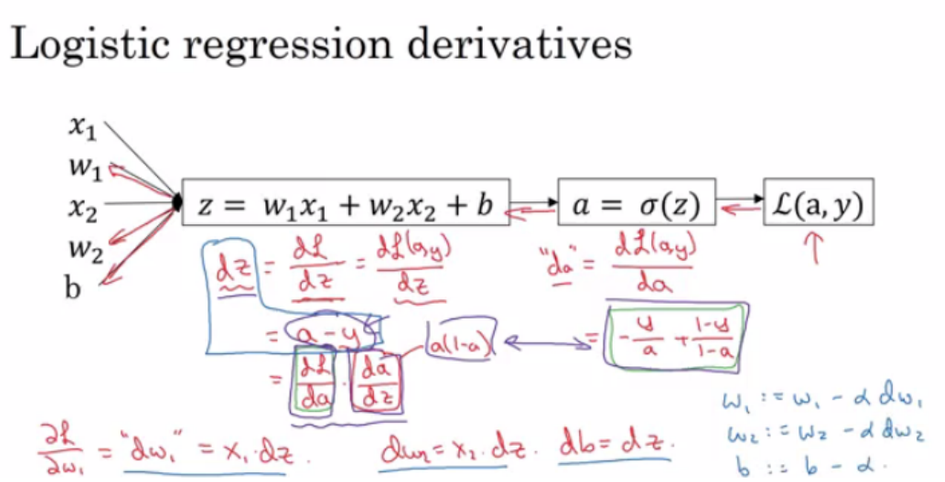
- Calculus chain rule says: If `x -> y -> z` (x effect y and y effects z) Then `d(z)/d(x) = d(z)/d(y) * d(y)/d(x)`
- The video illustrates a big example.



  -
- We compute the derivatives on a graph from right to left and it will be a lot more easier.
- `dvar` means the derivatives of a final output variable with respect to various intermediate quantities.

Logistic Regression Gradient Descent

- In the video he discussed the derivatives of gradient decent example for one sample with two features `x1` and `x2`.



Gradient Descent on m Examples

- Lets say we have these variables:

```
X1                                      Feature
X2                  Feature
W1                  Weight of the first feature.
W2                  Weight of the second feature.
B                   Logistic Regression parameter.
M                   Number of training examples
Y(i)                            Expected output of i
```

- So we have:

```
X1  \
W1   \
X2   ===>  z(i) = X1W1 + X2W2 + B ===> a(i) = Sigmoid(z(i)) ===> l(a(i),Y(i)) = - (Y(i)*log(a(i)) + (1-Y(i))*log(1-a(i)))
Y2   /
B   /
```

- Then from right to left we will calculate derivations compared to the result:

```
d(a)  = d(l)/d(a) = -(y/a) + ((1-y)/(1-a))
d(z)  = d(l)/d(z) = a - y
d(W1) = X1 * d(z)
d(W2) = X2 * d(z)
d(B) = d(z)
```

- From the above we can conclude the logistic regression pseudo code:

```
J = 0; dw1 = 0; dw2 =0; db = 0;                 # Devs.
w1 = 0; w2 = 0; b=0;                                                    # Weights
for i = 1 to m
        # Forward pass
        z(i) = W1*x1(i) + W2*x2(i) + b
        a(i) = Sigmoid(z(i))
        J += (Y(i)*log(a(i)) + (1-Y(i))*log(1-a(i)))

        # Backward pass
        dz(i) = a(i) - Y(i)
        dw1 += dz(i) * x1(i)
        dw2 += dz(i) * x2(i)
        db  += dz(i)
J /= m
dw1/= m
dw2/= m
db/= m

# Gradient descent
w1 = w1 - alpa * dw1
w2 = w2 - alpa * dw2
b = b - alpa * db
```

- The above code should run for some iterations to minimize error.
- So there will be two inner loops to implement the logistic regression.
- Vectorization is so important on deep learning to reduce loops. In the last code we can make the whole loop in one step using vectorization!

Vectorization

- Deep learning shines when the dataset are big. However for loops will make you wait a lot for a result. Thats why we need vectorization to get rid of some of our for loops.

- NumPy library (dot) function is using vectorization by default.
- The vectorization can be done on CPU or GPU thought the SIMD operation. But its faster on GPU.
- Whenever possible avoid for loops.
- Most of the NumPy library methods are vectorized version.

Vectorizing Logistic Regression

- We will implement Logistic Regression using one for loop then without any for loop.
- As an input we have a matrix `X` and its `[Nx, m]` and a matrix `Y` and its `[Ny, m]`.
- We will then compute at instance `[z1,z2...zm] = W' * X + [b,b,...b]`. This can be written in python as:

```
Z = np.dot(W.T,X) + b    # Vectorization, then broadcasting, Z shape is (1, m)
A = 1 / 1 + np.exp(-Z)   # Vectorization, A shape is (1, m)
```

- Vectorizing Logistic Regression's Gradient Output:

```
dz = A - Y                  # Vectorization, dz shape is (1, m)
dw = np.dot(X, dz.T) / m    # Vectorization, dw shape is (Nx, 1)
db = dz.sum() / m           # Vectorization, dz shape is (1, 1)
```

Notes on Python and NumPy

- In NumPy, `obj.sum(axis = 0)` sums the columns while `obj.sum(axis = 1)` sums the rows.
- In NumPy, `obj.reshape(1,4)` changes the shape of the matrix by broadcasting the values.
- Reshape is cheap in calculations so put it everywhere you're not sure about the calculations.
- Broadcasting works when you do a matrix operation with matrices that doesn't match for the operation, in this case NumPy automatically makes the shapes ready for the operation by broadcasting the values.
- In general principle of broadcasting. If you have an (m,n) matrix and you add(+) or subtract(-) or multiply(*) or divide(/) with a (1,n) matrix, then this will copy it m times into an (m,n) matrix. The same with if you use those operations with a (m , 1) matrix, then this will copy it n times into (m, n) matrix. And then apply the addition, subtraction, and multiplication of division element wise.
- Some tricks to eliminate all the strange bugs in the code:
  - If you didn't specify the shape of a vector, it will take a shape of `(m,)` and the transpose operation won't work. You have to reshape it to `(m, 1)`
  - Try to not use the rank one matrix in ANN
  - Don't hesitate to use `assert(a.shape == (5,1))` to check if your matrix shape is the required one.
  - If you've found a rank one matrix try to run reshape on it.
- Jupyter / IPython notebooks are so useful library in python that makes it easy to integrate code and document at the same time. It runs in the browser and doesn't need an IDE to run.
  - To open Jupyter Notebook, open the command line and call: `jupyter-notebook` It should be installed to work.
- To Compute the derivative of Sigmoid:

```
s = sigmoid(x)
ds = s * (1 - s)       # derivative  using calculus
```

- To make an image of `(width,height,depth)` be a vector, use this:

```
v = image.reshape(image.shape[0]*image.shape[1]*image.shape[2],1)  #reshapes the image.
```

- Gradient descent converges faster after normalization of the input matrices.

General Notes

- The main steps for building a Neural Network are:
  - Define the model structure (such as number of input features and outputs)
  - Initialize the model's parameters.
  - Loop.
    - Calculate current loss (forward propagation)
    - Calculate current gradient (backward propagation)
    - Update parameters (gradient descent)
- Preprocessing the dataset is important.
- Tuning the learning rate (which is an example of a "hyperparameter") can make a big difference to the algorithm.

- kaggle.com is a good place for datasets and competitions.
- Pieter Abbeel is one of the best in deep reinforcement learning.

Shallow neural networks

> Learn to build a neural network with one hidden layer, using forward propagation and backpropagation.

Neural Networks Overview

- In logistic regression we had:

```
X1  \
X2   ==>  z = XW + B ==> a = Sigmoid(z) ==> l(a,Y)
X3  /
```

- In neural networks with one layer we will have:

```
X1  \
X2   =>  z1 = XW1 + B1 => a1 = Sigmoid(z1) => z2 = a1W2 + B2 => a2 = Sigmoid(z2) => l(a2,Y)
X3  /
```

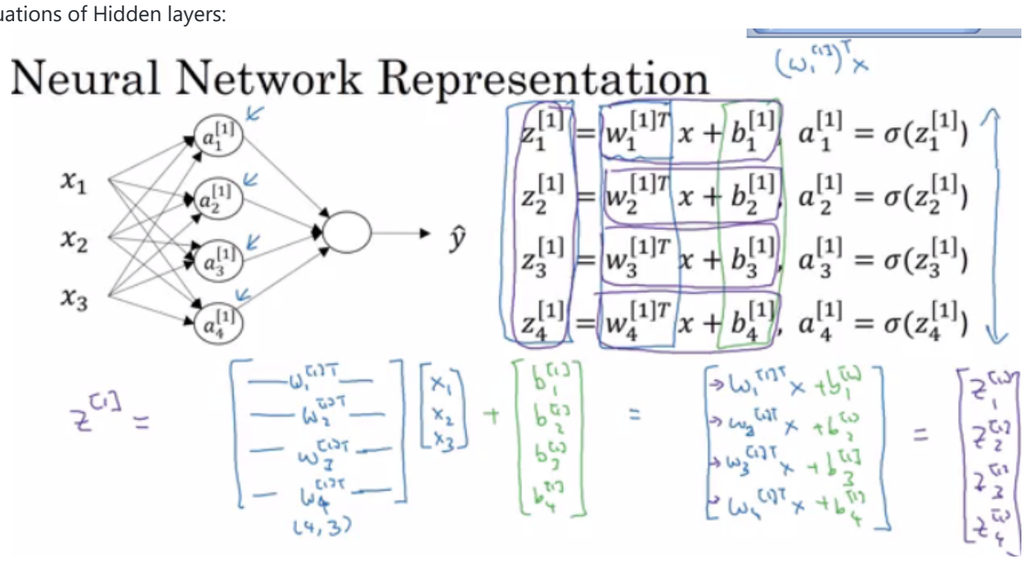
- `X` is the input vector `(X1, X2, X3)`, and `Y` is the output variable `(1x1)`
- NN is stack of logistic regression objects.

Neural Network Representation

- We will define the neural networks that has one hidden layer.
- NN contains of input layers, hidden layers, output layers.
- Hidden layer means we cant see that layers in the training set.
- `a0 = x` (the input layer)
- `a1` will represent the activation of the hidden neurons.
- `a2` will represent the output layer.
- We are talking about 2 layers NN. The input layer isn't counted.

Computing a Neural Network's Output

- Equations of Hidden layers:



- Here are some informations about the last image:
  - `noOfHiddenNeurons = 4`
  - `Nx = 3`
  - Shapes of the variables:
    - `W1` is the matrix of the first hidden layer, it has a shape of `(noOfHiddenNeurons,nx)`
    - `b1` is the matrix of the first hidden layer, it has a shape of `(noOfHiddenNeurons,1)`
    - `z1` is the result of the equation `z1 = W1*X + b`, it has a shape of `(noOfHiddenNeurons,1)`
    - `a1` is the result of the equation `a1 = sigmoid(z1)`, it has a shape of `(noOfHiddenNeurons,1)`
    - `W2` is the matrix of the second hidden layer, it has a shape of `(1,noOfHiddenNeurons)`
    - `b2` is the matrix of the second hidden layer, it has a shape of `(1,1)`
    - `z2` is the result of the equation `z2 = W2*a1 + b`, it has a shape of `(1,1)`
    - `a2` is the result of the equation `a2 = sigmoid(z2)`, it has a shape of `(1,1)`

Vectorizing across multiple examples

- Pseudo code for forward propagation for the 2 layers NN:

```
for i = 1 to m
  z[1, i] = W1*x[i] + b1      # shape of z[1, i] is (noOfHiddenNeurons,1)
  a[1, i] = sigmoid(z[1, i])  # shape of a[1, i] is (noOfHiddenNeurons,1)
  z[2, i] = W2*a[1, i] + b2   # shape of z[2, i] is (1,1)
  a[2, i] = sigmoid(z[2, i])  # shape of a[2, i] is (1,1)
```

- Lets say we have `X` on shape `(Nx,m)`. So the new pseudo code:

```
Z1 = W1X + b1     # shape of Z1 (noOfHiddenNeurons,m)
A1 = sigmoid(Z1)  # shape of A1 (noOfHiddenNeurons,m)
Z2 = W2A1 + b2    # shape of Z2 is (1,m)
A2 = sigmoid(Z2)  # shape of A2 is (1,m)
```

- If you notice always m is the number of columns.
- In the last example we can call `X` = `A0`. So the previous step can be rewritten as:

```
Z1 = W1A0 + b1    # shape of Z1 (noOfHiddenNeurons,m)
A1 = sigmoid(Z1)  # shape of A1 (noOfHiddenNeurons,m)
Z2 = W2A1 + b2    # shape of Z2 is (1,m)
A2 = sigmoid(Z2)  # shape of A2 is (1,m)
```

Activation functions

- So far we are using sigmoid, but in some cases other functions can be a lot better.
- Sigmoid can lead us to gradient decent problem where the updates are so low.
- Sigmoid activation function range is [0,1] `A = 1 / (1 + np.exp(-z)) # Where z is the input matrix`
- Tanh activation function range is [-1,1] (Shifted version of sigmoid function)
  - In NumPy we can implement Tanh using one of these methods: `A = (np.exp(z) - np.exp(-z)) / (np.exp(z) + np.exp(-z)) # Where z is the input matrix`

    Or `A = np.tanh(z) # Where z is the input matrix`
- It turns out that the tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer.
- Sigmoid or Tanh function disadvantage is that if the input is too small or too high, the slope will be near zero which will cause us the gradient decent problem.
- One of the popular activation functions that solved the slow gradient decent is the RELU function. `RELU = max(0,z) # so if z is negative the slope is 0 and if z is positive the slope remains linear.`
- So here is some basic rule for choosing activation functions, if your classification is between 0 and 1, use the output activation as sigmoid and the others as RELU.
- Leaky RELU activation function different of RELU is that if the input is negative the slope will be so small. It works as RELU but most people uses RELU. `Leaky_RELU = max(0.01z,z) #the 0.01 can be a parameter for your algorithm.`
- In NN you will decide a lot of choices like:
  - No of hidden layers.
  - No of neurons in each hidden layer.
  - Learning rate. (The most important parameter)
  - Activation functions.
  - And others..
- It turns out there are no guide lines for that. You should try all activation functions for example.

Why do you need non-linear activation functions?

- If we removed the activation function from our algorithm that can be called linear activation function.
- Linear activation function will output linear activations
  - Whatever hidden layers you add, the activation will be always linear like logistic regression (So its useless in a lot of complex problems)
- You might use linear activation function in one place - in the output layer if the output is real numbers (regression problem). But even in this case if the output value is non-negative you could use RELU instead.

Derivatives of activation functions

- Derivation of Sigmoid activation function:

```
g(z) = 1 / (1 + np.exp(-z))
g'(z) = (1 / (1 + np.exp(-z))) * (1 - (1 / (1 + np.exp(-z))))
g'(z) = g(z) * (1 - g(z))
```

- Derivation of Tanh activation function:

```
g(z)  = (e^z - e^-z) / (e^z + e^-z)
g'(z) = 1 - np.tanh(z)^2 = 1 - g(z)^2
```

- Derivation of RELU activation function:

```
g(z)  = np.maximum(0,z)
g'(z) = { 0  if z < 0
          1  if z >= 0  }
```

- Derivation of leaky RELU activation function:

```
g(z)  = np.maximum(0.01 * z, z)
g'(z) = { 0.01  if z < 0
          1     if z >= 0   }
```

Gradient descent for Neural Networks

- In this section we will have the full back propagation of the neural network (Just the equations with no explanations).
- Gradient descent algorithm:
  - NN parameters:
    - `n[0] = Nx`
    - `n[1] = NoOfHiddenNeurons`
    - `n[2] = NoOfOutputNeurons = 1`
    - `W1` shape is `(n[1],n[0])`
    - `b1` shape is `(n[1],1)`
    - `W2` shape is `(n[2],n[1])`
    - `b2` shape is `(n[2],1)`
  - Cost function `I = I(W1, b1, W2, b2) = (1/m) * Sum(L(Y,A2))`
  - Then Gradient descent:

```
Repeat:
                Compute predictions (y'[i], i = 0,...m)
                Get derivatives: dW1, db1, dW2, db2
                Update: W1 = W1 - LearningRate * dW1
                                b1 = b1 - LearningRate * db1
                                W2 = W2 - LearningRate * dW2
                                b2 = b2 - LearningRate * db2
```

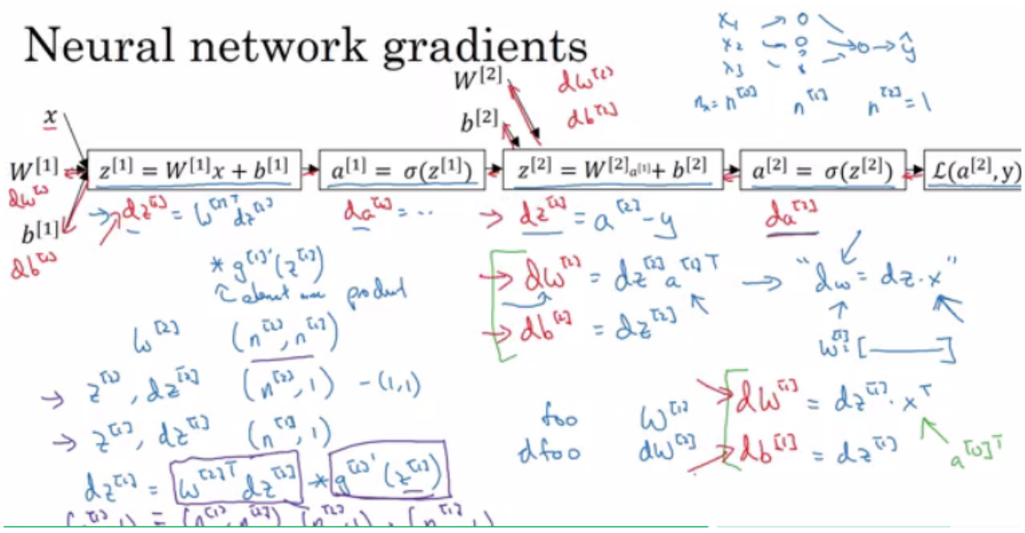
- Forward propagation:

```
Z1 = W1A0 + b1    # A0 is X
A1 = g1(Z1)
Z2 = W2A1 + b2
A2 = Sigmoid(Z2)      # Sigmoid because the output is between 0 and 1
```

- Backpropagation (derivations):

```
dZ2 = A2 - Y      # derivative of cost function we used * derivative of the sigmoid function
dW2 = (dZ2 * A1.T) / m
db2 = Sum(dZ2) / m
dZ1 = (W2.T * dZ2) * g'1(Z1)  # element wise product (*)
dW1 = (dZ1 * A0.T) / m   # A0 = X
db1 = Sum(dZ1) / m
# Hint there are transposes with multiplication because to keep dimensions correct
```

- How we derived the 6 equations of the backpropagation:



Random Initialization

- In logistic regression it wasn't important to initialize the weights randomly, while in NN we have to initialize them randomly.
- If we initialize all the weights with zeros in NN it won't work (initializing bias with zero is OK):
  - all hidden units will be completely identical (symmetric) - compute exactly the same function
  - on each gradient descent iteration all the hidden units will always update the same
- To solve this we initialize the W's with a small random numbers:

```
W1 = np.random.randn((2,2)) * 0.01    # 0.01 to make it small enough
b1 = np.zeros((2,1))                  # its ok to have b as zero, it won't get us to the symmetry breaking
problem
```

- We need small values because in sigmoid (or tanh), for example, if the weight is too large you are more likely to end up even at the very start of training with very large values of Z. Which causes your tanh or your sigmoid activation function to be saturated, thus slowing down learning. If you don't have any sigmoid or tanh activation functions throughout your neural network, this is less of an issue.
- Constant 0.01 is alright for 1 hidden layer networks, but if the NN is deep this number can be changed but it will always be a small number.

Deep Neural Networks

> Understand the key computations underlying deep learning, use them to build and train deep neural networks, and apply it to computer vision.

Deep L-layer neural network

- Shallow NN is a NN with one or two layers.
- Deep NN is a NN with three or more layers.
- We will use the notation `L` to denote the number of layers in a NN.
- `n[l]` is the number of neurons in a specific layer `l`.
- `n[0]` denotes the number of neurons input layer. `n[L]` denotes the number of neurons in output layer.
- `g[l]` is the activation function.
- `a[l] = g[l](z[l])`
- `w[l]` weights is used for `z[l]`
- `x = a[0]`, `a[l] = y'`
- These were the notation we will use for deep neural network.
- So we have:
  - A vector `n` of shape `(1, NoOfLayers+1)`
  - A vector `g` of shape `(1, NoOfLayers)`
  - A list of different shapes `w` based on the number of neurons on the previous and the current layer.
  - A list of different shapes `b` based on the number of neurons on the current layer.

Forward Propagation in a Deep Network

- Forward propagation general rule for one input:

```
z[l] = W[l]a[l-1] + b[l]
a[l] = g[l](a[l])
```

- Forward propagation general rule for `m` inputs:

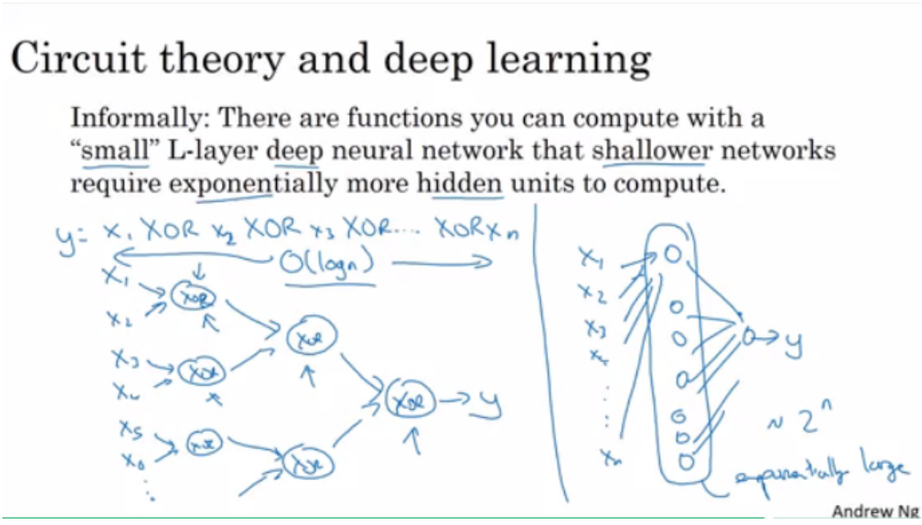
```
Z[l] = W[l]A[l-1] + B[l]
A[l] = g[l](A[l])
```

- We can't compute the whole layers forward propagation without a for loop so its OK to have a for loop here.
- The dimensions of the matrices are so important you need to figure it out.

Getting your matrix dimensions right

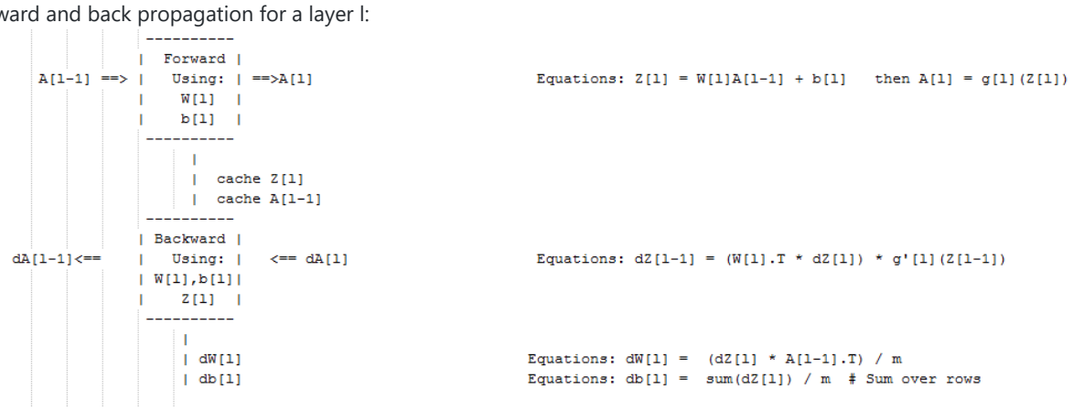
- The best way to debug your matrices dimensions is by a pencil and paper.
- Dimension of `W` is `(n[l],n[l-1])` . Can be thought by right to left.
- Dimension of `b` is `(n[l],1)`
- `dw` has the same shape as `W`, while `db` is the same shape as `b`
- Dimension of `Z[l]`, `A[l]`, `dZ[l]`, and `dA[l]` is `(n[l],m)`

Why deep representations?

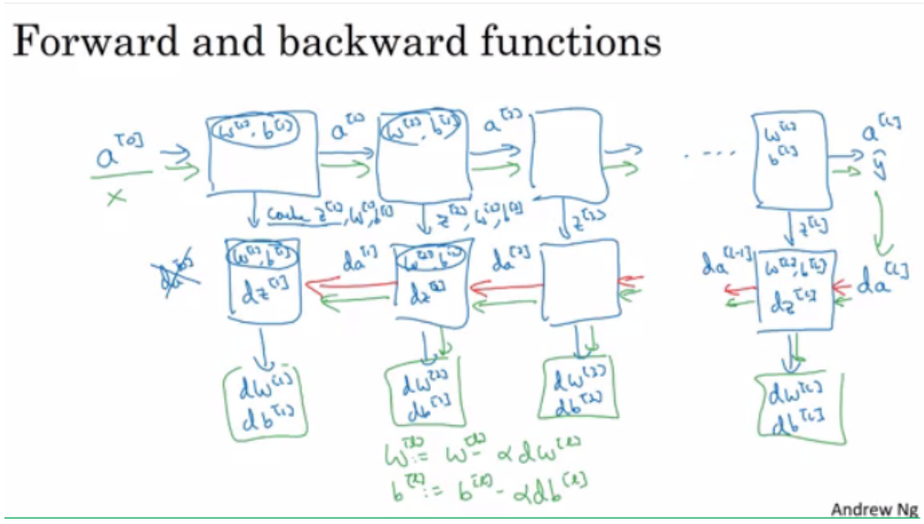
- Why deep NN works well, we will discuss this question in this section.
- Deep NN makes relations with data from simpler to complex. In each layer it tries to make a relation with the previous layer. E.g.:
  - a. Face recognition application:
    - Image ==> Edges ==> Face parts ==> Faces ==> desired face
  - b. Audio recognition application:
    - Audio ==> Low level sound features like (sss,bb) ==> Phonemes ==> Words ==> Sentences
- Neural Researchers think that deep neural networks "think" like brains (simple ==> complex)
- Circuit theory and deep learning:
  - 

- When starting on an application don't start directly by dozens of hidden layers. Try the simplest solutions (e.g. Logistic Regression), then try the shallow neural network and so on.

Building blocks of deep neural networks

- Forward and back propagation for a layer l:
  - 

- Deep NN blocks:



Forward and Backward Propagation

- Pseudo code for forward propagation for layer l:

```
Input  A[l-1]
Z[l] = W[l]A[l-1] + b[l]
A[l] = g[l](Z[l])
Output A[l], cache(Z[l])
```

- Pseudo code for back propagation for layer l:

```
Input da[l], Caches
dZ[l] = dA[l] * g'[l](Z[l])
dW[l] = (dZ[l]A[l-1].T) / m
db[l] = sum(dZ[l])/m                # Dont forget axis=1, keepdims=True
dA[l-1] = w[l].T * dZ[l]            # The multiplication here are a dot product.
Output dA[l-1], dW[l], db[l]
```

- If we have used our loss function then:

```
dA[L] = (-(y/a) + ((1-y)/(1-a)))
```

Parameters vs Hyperparameters

- Main parameters of the NN is `W` and `b`
- Hyper parameters (parameters that control the algorithm) are like:
  - Learning rate.
  - Number of iteration.
  - Number of hidden layers `L`.
  - Number of hidden units `n`.
  - Choice of activation functions.
- You have to try values yourself of hyper parameters.
- In the earlier days of DL and ML learning rate was often called a parameter, but it really is (and now everybody call it) a hyperparameter.
- On the next course we will see how to optimize hyperparameters.

What does this have to do with the brain

- The analogy that "It is like the brain" has become really an oversimplified explanation.
- There is a very simplistic analogy between a single logistic unit and a single neuron in the brain.
- No human today understand how a human brain neuron works.
- No human today know exactly how many neurons on the brain.
- Deep learning in Andrew's opinion is very good at learning very flexible, complex functions to learn X to Y mappings, to learn input-output mappings (supervised learning).
- The field of computer vision has taken a bit more inspiration from the human brains then other disciplines that also apply deep learning.
- NN is a small representation of how brain work. The most near model of human brain is in the computer vision (CNN)

Extra: Ian Goodfellow interview

- Ian is one of the world's most visible deep learning researchers.
- Ian is mainly working with generative models. He is the creator of GANs.

Course summary

Here are the course summary as its given on the course link:

> This course will teach you the "magic" of getting deep learning to work well. Rather than the deep learning process being a black box, you will understand what drives performance, and be able to more systematically get good results. You will also learn TensorFlow.
>
> After 3 weeks, you will:
>
> - Understand industry best-practices for building deep learning applications.
> - Be able to effectively use the common neural network "tricks", including initialization, L2 and dropout regularization, Batch normalization, gradient checking,
> - Be able to implement and apply a variety of optimization algorithms, such as mini-batch gradient descent, Momentum, RMSprop and Adam, and check for their convergence.
> - Understand new best-practices for the deep learning era of how to set up train/dev/test sets and analyze bias/variance
> - Be able to implement a neural network in TensorFlow.
>
> This is the second course of the Deep Learning Specialization.

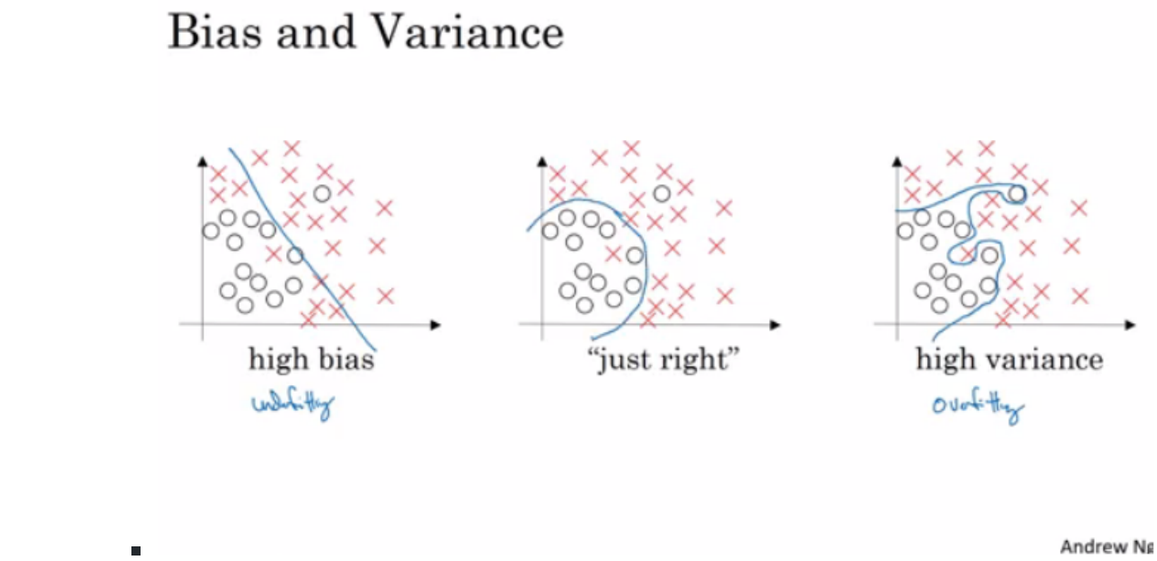
Practical aspects of Deep Learning

Train / Dev / Test sets

- Its impossible to get all your hyperparameters right on a new application from the first time.
- So the idea is you go through the loop: `Idea ==> Code ==> Experiment`.
- You have to go through the loop many times to figure out your hyperparameters.
- Your data will be split into three parts:
    - Training set. (Has to be the largest set)
    - Hold-out cross validation set / Development or "dev" set.
    - Testing set.
- You will try to build a model upon training set then try to optimize hyperparameters on dev set as much as possible. Then after your model is ready you try and evaluate the testing set.
- so the trend on the ratio of splitting the models:
    - If size of the dataset is 100 to 1000000 ==> 60/20/20
    - If size of the dataset is 1000000 to INF ==> 98/1/1 or 99.5/0.25/0.25
- The trend now gives the training data the biggest sets.
- Make sure the dev and test set are coming from the same distribution.
    - For example if cat training pictures is from the web and the dev/test pictures are from users cell phone they will mismatch. It is better to make sure that dev and test set are from the same distribution.
- The dev set rule is to try them on some of the good models you've created.
- Its OK to only have a dev set without a testing set. But a lot of people in this case call the dev set as the test set. A better terminology is to call it a dev set as its used in the development.

Bias / Variance

- Bias / Variance techniques are Easy to learn, but difficult to master.
- So here the explanation of Bias / Variance:
    - If your model is underfitting (logistic regression of non linear data) it has a "high bias"
    - If your model is overfitting then it has a "high variance"
    - Your model will be alright if you balance the Bias / Variance
    - For more:



- Another idea to get the bias / variance if you don't have a 2D plotting mechanism:
  - High variance (overfitting) for example:
    - Training error: 1%
    - Dev error: 11%
  - high Bias (underfitting) for example:
    - Training error: 15%
    - Dev error: 14%
  - high Bias (underfitting) && High variance (overfitting) for example:
    - Training error: 15%
    - Test error: 30%
  - Best:
    - Training error: 0.5%
    - Test error: 1%
  - These Assumptions came from that human has 0% error. If the problem isn't like that you'll need to use human error as baseline.

Basic Recipe for Machine Learning

- If your algorithm has a high bias:
  - Try to make your NN bigger (size of hidden units, number of layers)
  - Try a different model that is suitable for your data.
  - Try to run it longer.
  - Different (advanced) optimization algorithms.
- If your algorithm has a high variance:
  - More data.
  - Try regularization.
  - Try a different model that is suitable for your data.
- You should try the previous two points until you have a low bias and low variance.
- In the older days before deep learning, there was a "Bias/variance tradeoff". But because now you have more options/tools for solving the bias and variance problem its really helpful to use deep learning.
- Training a bigger neural network never hurts.

Regularization

- Adding regularization to NN will help it reduce variance (overfitting)
- L1 matrix norm:
  - `||W|| = Sum(|w[i,j]|) # sum of absolute values of all w`
- L2 matrix norm because of arcane technical math reasons is called Frobenius norm:
  - `||W||^2 = Sum(|w[i,j]|^2) # sum of all w squared`
  - Also can be calculated as `||W||^2 = W.T * W if W is a vector`
- Regularization for logistic regression:
  - The normal cost function that we want to minimize is: `J(w,b) = (1/m) * Sum(L(y(i),y'(i)))`
  - The L2 regularization version: `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum(|w[i]|^2)`
  - The L1 regularization version: `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum(|w[i]|)`
  - The L1 regularization version makes a lot of w values become zeros, which makes the model size smaller.
  - L2 regularization is being used much more often.
  - `lambda` here is the regularization parameter (hyperparameter)
- Regularization for NN:

- The normal cost function that we want to minimize is:
  `J(W1,b1...,WL,bL) = (1/m) * Sum(L(y(i),y'(i)))`
- The L2 regularization version:
  `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum((||W[l]||^2)`
- We stack the matrix as one vector `(mn,1)` and then we apply `sqrt(w1^2 + w2^2.....)`
- To do back propagation (old way):
  `dw[l] = (from back propagation)`
- The new way:
  `dw[l] = (from back propagation) + lambda/m * w[l]`
- So plugging it in weight update step:

```
w[l] = w[l] - learning_rate * dw[l]
     = w[l] - learning_rate * ((from back propagation) + lambda/m * w[l])
     = w[l] - (learning_rate*lambda/m) * w[l] - learning_rate * (from back propagation)
     = (1 - (learning_rate*lambda)/m) * w[l] - learning_rate * (from back propagation)
```

- In practice this penalizes large weights and effectively limits the freedom in your model.
- The new term `(1 - (learning_rate*lambda)/m) * w[l]` causes the **weight to decay** in proportion to its size.

Why regularization reduces overfitting?

Here are some intuitions:

- Intuition 1:
  - If `lambda` is too large - a lot of w's will be close to zeros which will make the NN simpler (you can think of it as it would behave closer to logistic regression).
  - If `lambda` is good enough it will just reduce some weights that makes the neural network overfit.
- Intuition 2 (with *tanh* activation function):
  - If `lambda` is too large, w's will be small (close to zero) - will use the linear part of the *tanh* activation function, so we will go from non linear activation to *roughly* linear which would make the NN a *roughly* linear classifier.
  - If `lambda` good enough it will just make some of *tanh* activations *roughly* linear which will prevent overfitting.

Implementation tip: if you implement gradient descent, one of the steps to debug gradient descent is to plot the cost function J as a function of the number of iterations of gradient descent and you want to see that the cost function J decreases **monotonically** after every elevation of gradient descent with regularization. If you plot the old definition of J (no regularization) then you might not see it decrease monotonically.

Dropout Regularization

- In most cases Andrew Ng tells that he uses the L2 regularization.
- The dropout regularization eliminates some neurons/weights on each iteration based on a probability.
- A most common technique to implement dropout is called "Inverted dropout".
- Code for Inverted dropout:

```
keep_prob = 0.8   # 0 <= keep_prob <= 1
l = 3  # this code is only for layer 3
# the generated number that are less than 0.8 will be dropped. 80% stay, 20% dropped
d3 = np.random.rand(a[l].shape[0], a[l].shape[1]) < keep_prob

a3 = np.multiply(a3,d3)   # keep only the values in d3

# increase a3 to not reduce the expected value of output
# (ensures that the expected value of a3 remains the same) - to solve the scaling problem
a3 = a3 / keep_prob
```

- Vector d[l] is used for forward and back propagation and is the same for them, but it is different for each iteration (pass) or training example.
- At test time we don't use dropout. If you implement dropout at test time - it would add noise to predictions.

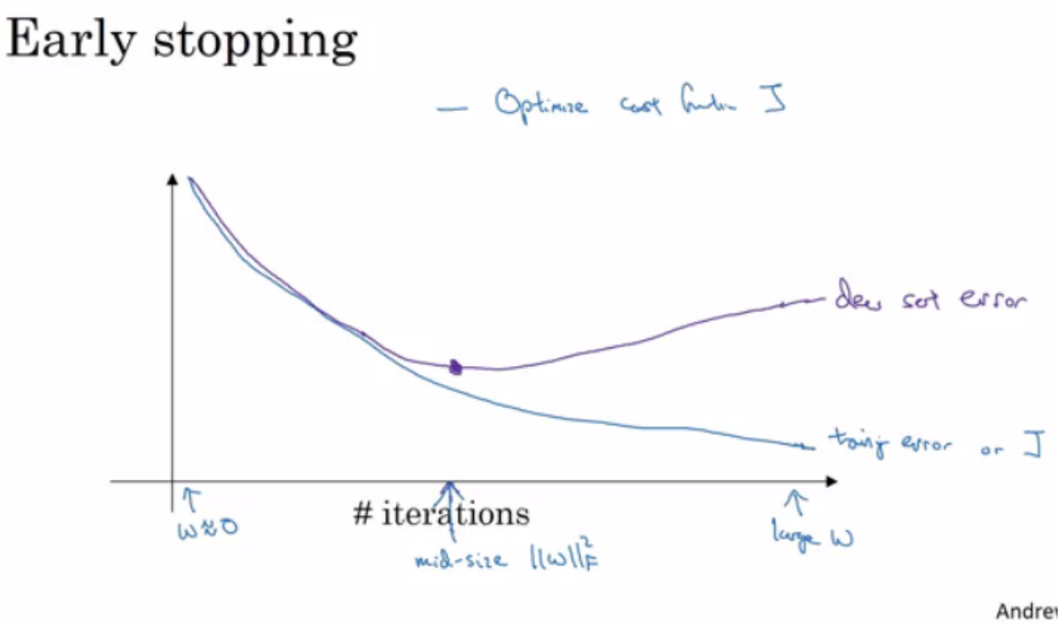
Understanding Dropout

- In the previous video, the intuition was that dropout randomly knocks out units in your network. So it's as if on every iteration you're working with a smaller NN, and so using a smaller NN seems like it should have a regularizing effect.
- Another intuition: can't rely on any one feature, so have to spread out weights.
- It's possible to show that dropout has a similar effect to L2 regularization.

- Dropout can have different `keep_prob` per layer.
- The input layer dropout has to be near 1 (or 1 - no dropout) because you don't want to eliminate a lot of features.
- If you're more worried about some layers overfitting than others, you can set a lower `keep_prob` for some layers than others. The downside is, this gives you even more hyperparameters to search for using cross-validation. One other alternative might be to have some layers where you apply dropout and some layers where you don't apply dropout and then just have one hyperparameter, which is a `keep_prob` for the layers for which you do apply dropouts.
- A lot of researchers are using dropout with Computer Vision (CV) because they have a very big input size and almost never have enough data, so overfitting is the usual problem. And dropout is a regularization technique to prevent overfitting.
- A downside of dropout is that the cost function J is not well defined and it will be hard to debug (plot J by iteration).
    - To solve that you'll need to turn off dropout, set all the `keep_prob`s to 1, and then run the code and check that it monotonically decreases J and then turn on the dropouts again.

Other regularization methods

- **Data augmentation**:
    - For example in a computer vision data:
        - You can flip all your pictures horizontally this will give you m more data instances.
        - You could also apply a random position and rotation to an image to get more data.
    - For example in OCR, you can impose random rotations and distortions to digits/letters.
    - New data obtained using this technique isn't as good as the real independent data, but still can be used as a regularization technique.
- **Early stopping**:
    - In this technique we plot the training set and the dev set cost together for each iteration. At some iteration the dev set cost will stop decreasing and will start increasing.
    - We will pick the point at which the training set error and dev set error are best (lowest training cost with lowest dev cost).
    - We will take these parameters as the best parameters.



    - Andrew prefers to use L2 regularization instead of early stopping because this technique simultaneously tries to minimize the cost function and not to overfit which contradicts the orthogonalization approach (will be discussed further).
    - But its advantage is that you don't need to search a hyperparameter like in other regularization approaches (like `lambda` in L2 regularization).
- **Model Ensembles**:
    - Algorithm:
        - Train multiple independent models.
        - At test time average their results.
    - It can get you extra 2% performance.
    - It reduces the generalization error.
    - You can use some snapshots of your NN at the training ensembles them and take the results.

Normalizing inputs

- If you normalize your inputs this will speed up the training process a lot.
- Normalization are going on these steps:
    i. Get the mean of the training set: `mean = (1/m) * sum(x(i))`
    ii. Subtract the mean from each input: `X = X - mean`
        - This makes your inputs centered around 0.

iii. Get the variance of the training set: `variance = (1/m) * sum(x(i)^2)`

iv. Normalize the variance. `X /= variance`

- These steps should be applied to training, dev, and testing sets (but using mean and variance of the train set).
- Why normalize?
  - If we don't normalize the inputs our cost function will be deep and its shape will be inconsistent (elongated) then optimizing it will take a long time.
  - But if we normalize it the opposite will occur. The shape of the cost function will be consistent (look more symmetric like circle in 2D example) and we can use a larger learning rate alpha - the optimization will be faster.

Vanishing / Exploding gradients

- The Vanishing / Exploding gradients occurs when your derivatives become very small or very big.
- To understand the problem, suppose that we have a deep neural network with number of layers L, and all the activation functions are **linear** and each `b = 0`
  - Then:

```
Y' = W[L]W[L-1].....W[2]W[1]X
```

  - Then, if we have 2 hidden units per layer and x1 = x2 = 1, we result in:

```
if W[l] = [1.5   0]
          [0   1.5] (l != L because of different dimensions in the output layer)
Y' = W[L] [1.5  0]^(L-1) X = 1.5^L      # which will be very large
          [0  1.5]
```

```
if W[l] = [0.5  0]
          [0  0.5]
Y' = W[L] [0.5  0]^(L-1) X = 0.5^L      # which will be very small
          [0  0.5]
```

- The last example explains that the activations (and similarly derivatives) will be decreased/increased exponentially as a function of number of layers.
- So If W > I (Identity matrix) the activation and gradients will explode.
- And If W < I (Identity matrix) the activation and gradients will vanish.
- Recently Microsoft trained 152 layers (ResNet)! which is a really big number. With such a deep neural network, if your activations or gradients increase or decrease exponentially as a function of L, then these values could get really big or really small. And this makes training difficult, especially if your gradients are exponentially smaller than L, then gradient descent will take tiny little steps. It will take a long time for gradient descent to learn anything.
- There is a partial solution that doesn't completely solve this problem but it helps a lot - careful choice of how you initialize the weights (next video).

Weight Initialization for Deep Networks

- A partial solution to the Vanishing / Exploding gradients in NN is better or more careful choice of the random initialization of weights
- In a single neuron (Perceptron model): `Z = w1x1 + w2x2 + ... + wnxn`
  - So if `n_x` is large we want `W`'s to be smaller to not explode the cost.
- So it turns out that we need the variance which equals `1/n_x` to be the range of `W`'s
- So lets say when we initialize `W`'s like this (better to use with `tanh` activation):

```
np.random.rand(shape) * np.sqrt(1/n[l-1])
```

  or variation of this (Bengio et al.):

```
np.random.rand(shape) * np.sqrt(2/(n[l-1] + n[l]))
```

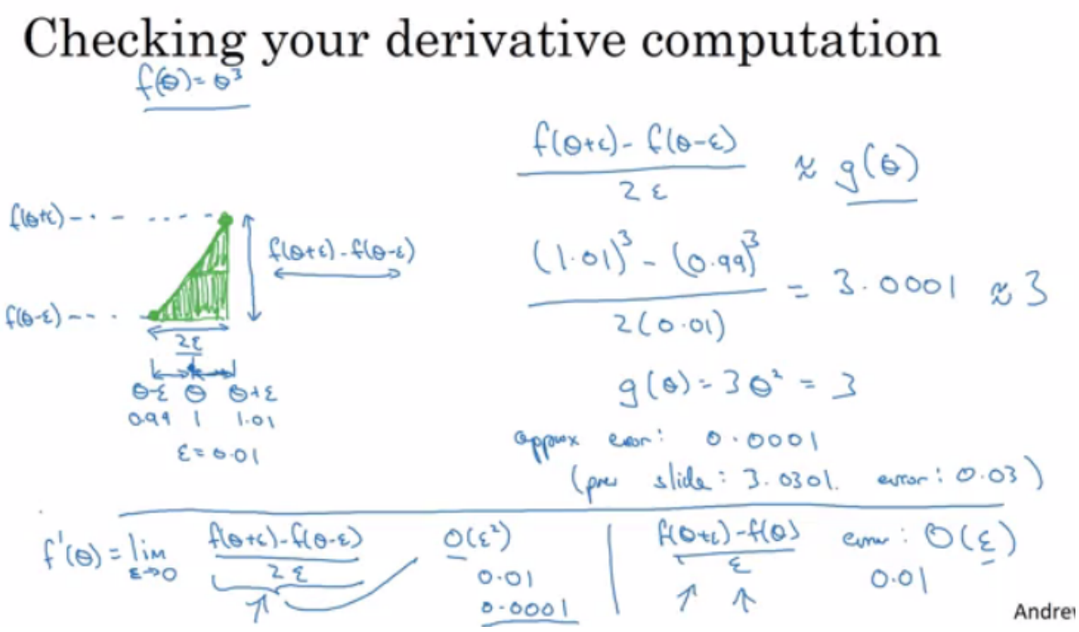
- Setting initialization part inside sqrt to `2/n[l-1]` for `ReLU` is better:

```
np.random.rand(shape) * np.sqrt(2/n[l-1])
```

- Number 1 or 2 in the nominator can also be a hyperparameter to tune (but not the first to start with)
- This is one of the best way of partially solution to Vanishing / Exploding gradients (ReLU + Weight Initialization with variance) which will help gradients not to vanish/explode too quickly
- The initialization in this video is called "He Initialization / Xavier Initialization" and has been published in 2015 paper.

Numerical approximation of gradients

- There is an technique called gradient checking which tells you if your implementation of backpropagation is correct.
- There's a numerical way to calculate the derivative:



- Gradient checking approximates the gradients and is very helpful for finding the errors in your backpropagation implementation but it's slower than gradient descent (so use only for debugging).
- Implementation of this is very simple.
- Gradient checking:
  - First take `W[1],b[1],...,W[L],b[L]` and reshape into one big vector (`theta`)
  - The cost function will be `J(theta)`
  - Then take `dW[1],db[1],...,dW[L],db[L]` into one big vector (`d_theta`)
  - **Algorithm**:

    ```
    eps = 10^-7   # small number
    for i in len(theta):
      d_theta_approx[i] = (J(theta1,...,theta[i] + eps) -  J(theta1,...,theta[i] - eps)) / 2*eps
    ```

  - Finally we evaluate this formula `(||d_theta_approx - d_theta||) / (||d_theta_approx||+||d_theta||)` (`||` - Euclidean vector norm) and check (with eps = 10^-7):
    - if it is < 10^-7 - great, very likely the backpropagation implementation is correct
    - if around 10^-5 - can be OK, but need to inspect if there are no particularly big values in `d_theta_approx - d_theta` vector
    - if it is >= 10^-3 - bad, probably there is a bug in backpropagation implementation

Gradient checking implementation notes

- Don't use the gradient checking algorithm at training time because it's very slow.
- Use gradient checking only for debugging.
- If algorithm fails grad check, look at components to try to identify the bug.
- Don't forget to add `lamda/(2m) * sum(W[l])` to `J` if you are using L1 or L2 regularization.
- Gradient checking doesn't work with dropout because J is not consistent.
  - You can first turn off dropout (set `keep_prob = 1.0`), run gradient checking and then turn on dropout again.
- Run gradient checking at random initialization and train the network for a while maybe there's a bug which can be seen when w's and b's become larger (further from 0) and can't be seen on the first iteration (when w's and b's are very small).

Initialization summary

- The weights $W^{[l]}$ should be initialized randomly to break symmetry
- It is however okay to initialize the biases $b^{[l]}$ to zeros. Symmetry is still broken so long as $W^{[l]}$ is initialized randomly
- Different initializations lead to different results
- Random initialization is used to break symmetry and make sure different hidden units can learn different things
- Don't intialize to values that are too large
- He initialization works well for networks with ReLU activations.

Regularization summary

1. L2 Regularization

Observations:

- The value of λ is a hyperparameter that you can tune using a dev set.
- L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to "oversmooth", resulting in a model with high bias.

What is L2-regularization actually doing?:

- L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.

What you should remember:
Implications of L2-regularization on:

- cost computation:
  - A regularization term is added to the cost
- backpropagation function:
  - There are extra terms in the gradients with respect to weight matrices
- weights:
  - weights end up smaller ("weight decay") - are pushed to smaller values.

2. Dropout

What you should remember about dropout:

- Dropout is a regularization technique.
- You only use dropout during training. Don't use dropout (randomly eliminate nodes) during test time.
- Apply dropout both during forward and backward propagation.
- During training time, divide each dropout layer by keep_prob to keep the same expected value for the activations. For example, if `keep_prob` is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when keep_prob is other values than 0.5.

Optimization algorithms

Mini-batch gradient descent

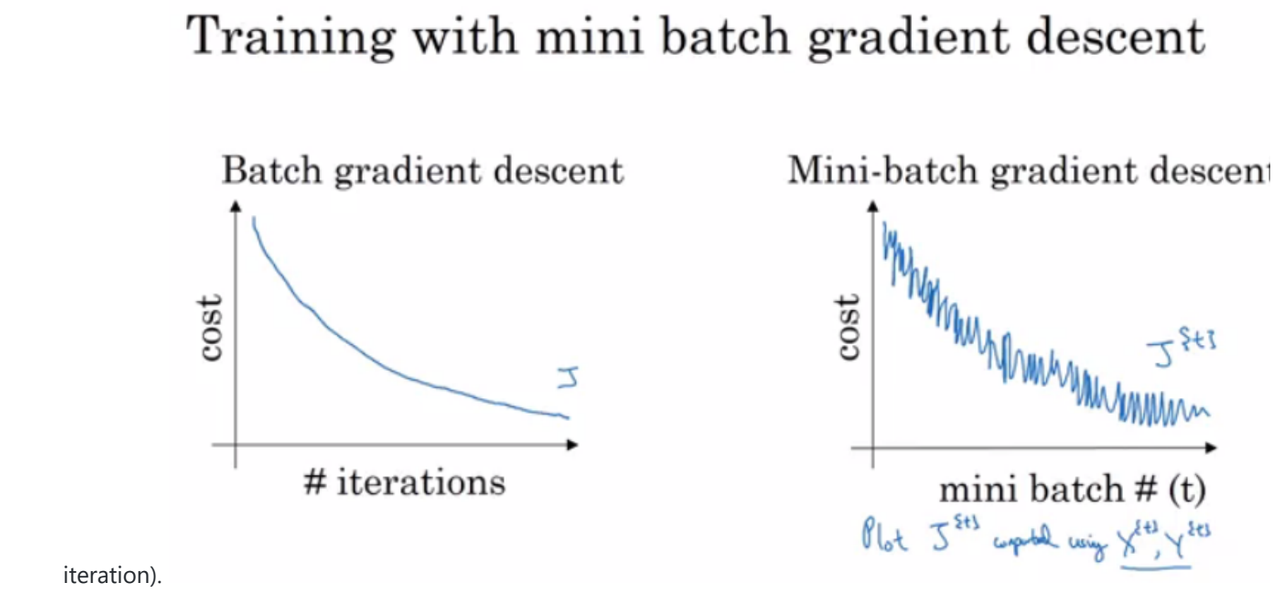
- Training NN with a large data is slow. So to find an optimization algorithm that runs faster is a good idea.
- Suppose we have `m = 50 million`. To train this data it will take a huge processing time for one step.
  - because 50 million won't fit in the memory at once we need other processing to make such a thing.
- It turns out you can make a faster algorithm to make gradient descent process some of your items even before you finish the 50 million items.
- Suppose we have split m to **mini batches** of size 1000.
  - `X{1} = 0 ... 1000`
  - `X{2} = 1001 ... 2000`
  - `...`
  - `X{bs} = ...`
- We similarly split `X` & `Y`.
- So the definition of mini batches ==> `t: X{t}, Y{t}`
- In **Batch gradient descent** we run the gradient descent on the whole dataset.
- While in **Mini-Batch gradient descent** we run the gradient descent on the mini datasets.
- Mini-Batch algorithm pseudo code:

```
for t = 1:No_of_batches                         # this is called an epoch
        AL, caches = forward_prop(X{t}, Y{t})
        cost = compute_cost(AL, Y{t})
        grads = backward_prop(AL, caches)
        update_parameters(grads)
```

- The code inside an epoch should be vectorized.
- Mini-batch gradient descent works much faster in the large datasets.

Understanding mini-batch gradient descent

- In mini-batch algorithm, the cost won't go down with each step as it does in batch algorithm. It could contain some ups and downs but generally it has to go down (unlike the batch gradient descent where cost function descreases on each



iteration).

- Mini-batch size:
  - (`mini batch size = m`) ==> Batch gradient descent
  - (`mini batch size = 1`) ==> Stochastic gradient descent (SGD)
  - (`mini batch size = between 1 and m`) ==> Mini-batch gradient descent
- Batch gradient descent:
  - too long per iteration (epoch)
- Stochastic gradient descent:
  - too noisy regarding cost minimization (can be reduced by using smaller learning rate)
  - won't ever converge (reach the minimum cost)
  - lose speedup from vectorization
- Mini-batch gradient descent:
  i. faster learning:
    - you have the vectorization advantage
    - make progress without waiting to process the entire training set

  ii. doesn't always exactly converge (oscelates in a very small region, but you can reduce learning rate)
- Guidelines for choosing mini-batch size:
  i. If small training set (< 2000 examples) - use batch gradient descent.
  ii. It has to be a power of 2 (because of the way computer memory is layed out and accessed, sometimes your code runs faster if your mini-batch size is a power of 2): `64, 128, 256, 512, 1024, ...`
  iii. Make sure that mini-batch fits in CPU/GPU memory.
- Mini-batch size is a `hyperparameter`.

Exponentially weighted averages

- There are optimization algorithms that are better than **gradient descent**, but you should first learn about Exponentially weighted averages.
- If we have data like the temperature of day through the year it could be like this:

```
t(1) = 40
t(2) = 49
t(3) = 45
...
t(180) = 60
...
```

- This data is small in winter and big in summer. If we plot this data we will find it some noisy.
- Now lets compute the Exponentially weighted averages:

```
V0 = 0
V1 = 0.9 * V0 + 0.1 * t(1) = 4          # 0.9 and 0.1 are hyperparameters
V2 = 0.9 * V1 + 0.1 * t(2) = 8.5
V3 = 0.9 * V2 + 0.1 * t(3) = 12.15
...
```

- General equation

```
V(t) = beta * v(t-1) + (1-beta) * theta(t)
```

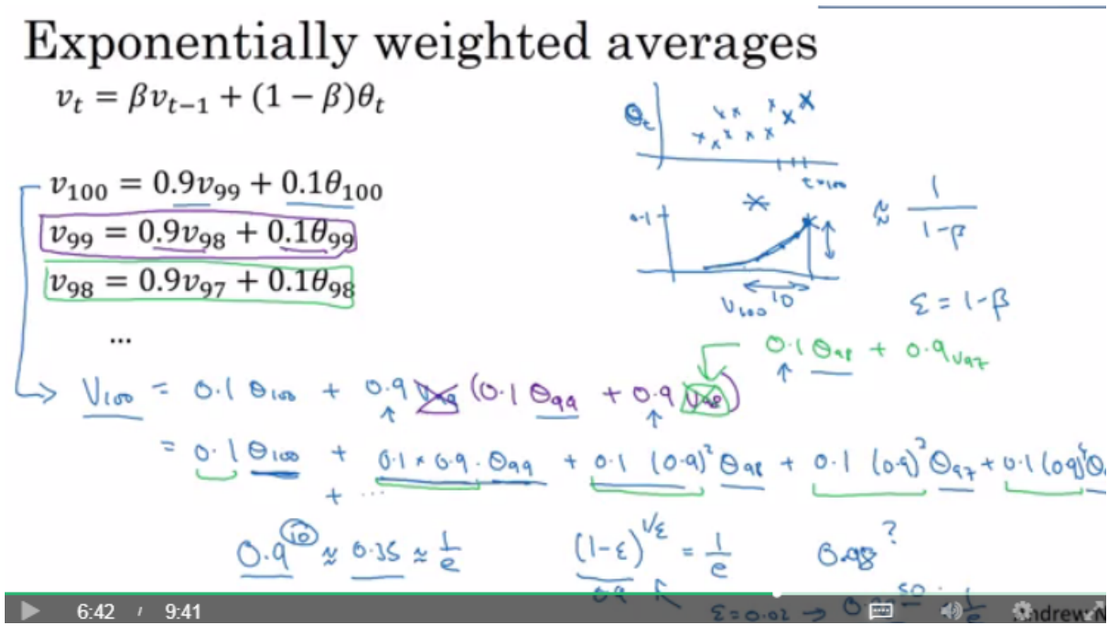
- If we plot this it will represent averages over `~ (1 / (1 - beta))` entries:
  - `beta = 0.9` will average last 10 entries
  - `beta = 0.98` will average last 50 entries
  - `beta = 0.5` will average last 2 entries

- Best beta average for our case is between 0.9 and 0.98
- Another imagery example:

(taken from investopedia.com)

Understanding exponentially weighted averages

- Intuitions:



- We can implement this algorithm with more accurate results using a moving window. But the code is more efficient and faster using the exponentially weighted averages algorithm.
- Algorithm is very simple:

```
v = 0
Repeat
{
        Get theta(t)
        v = beta * v + (1-beta) * theta(t)
}
```

Bias correction in exponentially weighted averages

- The bias correction helps make the exponentially weighted averages more accurate.
- Because `v(0) = 0`, the bias of the weighted averages is shifted and the accuracy suffers at the start.
- To solve the bias issue we have to use this equation:

```
v(t) = (beta * v(t-1) + (1-beta) * theta(t)) / (1 - beta^t)
```

- As t becomes larger the `(1 - beta^t)` becomes close to `1`

Gradient descent with momentum

- The momentum algorithm almost always works faster than standard gradient descent.
- The simple idea is to calculate the exponentially weighted averages for your gradients and then update your weights with the new values.
- Pseudo code:

```
vdW = 0, vdb = 0
on iteration t:
        # can be mini-batch or batch gradient descent
        compute dw, db on current mini-batch
```

```
        vdW = beta * vdW + (1 - beta) * dW
        vdb = beta * vdb + (1 - beta) * db
        W = W - learning_rate * vdW
        b = b - learning_rate * vdb
```

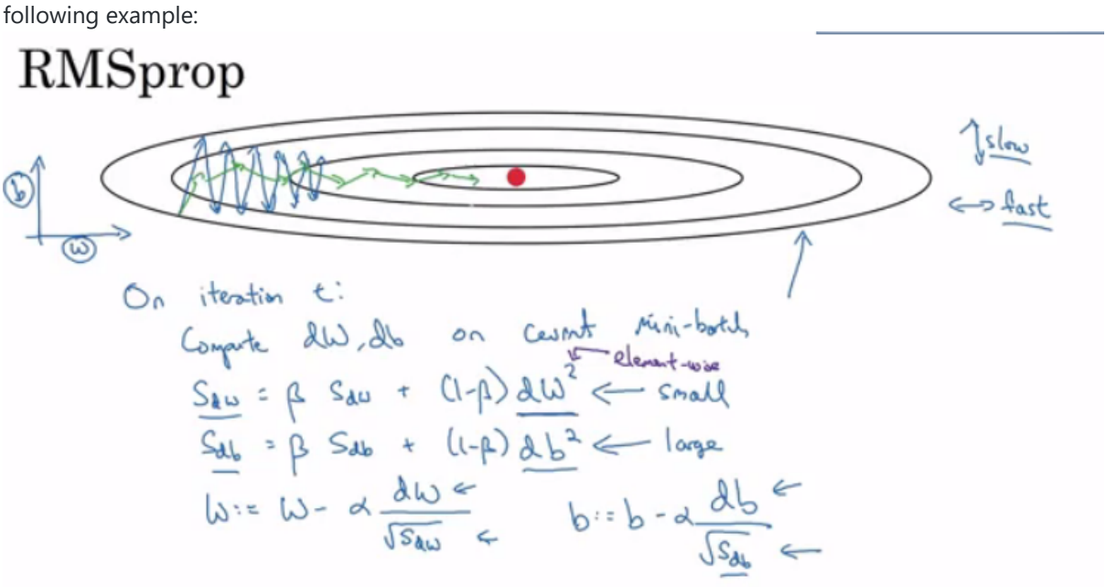
- Momentum helps the cost function to go to the minimum point in a more fast and consistent way.
- `beta` is another `hyperparameter`. `beta = 0.9` is very common and works very well in most cases.
- In practice people don't bother implementing **bias correction**.

RMSprop

- Stands for **Root mean square prop**.
- This algorithm speeds up the gradient descent.
- Pseudo code:

```
sdW = 0, sdb = 0
on iteration t:
        # can be mini-batch or batch gradient descent
        compute dw, db on current mini-batch

        sdW = (beta * sdW) + (1 - beta) * dW^2  # squaring is element-wise
        sdb = (beta * sdb) + (1 - beta) * db^2  # squaring is element-wise
        W = W - learning_rate * dW / sqrt(sdW)
        b = B - learning_rate * db / sqrt(sdb)
```

- RMSprop will make the cost function move slower on the vertical direction and faster on the horizontal direction in the following example:



- Ensure that `sdW` is not zero by adding a small value `epsilon` (e.g. `epsilon = 10^-8`) to it: `W = W - learning_rate * dW / (sqrt(sdW) + epsilon)`
- With RMSprop you can increase your learning rate.
- Developed by Geoffrey Hinton and firstly introduced on Coursera.org course.

Adam optimization algorithm

- Stands for **Adaptive Moment Estimation**.
- Adam optimization and RMSprop are among the optimization algorithms that worked very well with a lot of NN architectures.
- Adam optimization simply puts RMSprop and momentum together!
- Pseudo code:

```
vdW = 0, vdW = 0
sdW = 0, sdb = 0
on iteration t:
        # can be mini-batch or batch gradient descent
        compute dw, db on current mini-batch

        vdW = (beta1 * vdW) + (1 - beta1) * dW     # momentum
        vdb = (beta1 * vdb) + (1 - beta1) * db     # momentum

        sdW = (beta2 * sdW) + (1 - beta2) * dW^2   # RMSprop
        sdb = (beta2 * sdb) + (1 - beta2) * db^2   # RMSprop

        vdW = vdW / (1 - beta1^t)      # fixing bias
        vdb = vdb / (1 - beta1^t)      # fixing bias

        sdW = sdW / (1 - beta2^t)      # fixing bias
        sdb = sdb / (1 - beta2^t)      # fixing bias
```

```
        W = W - learning_rate * vdW / (sqrt(sdW) + epsilon)
        b = B - learning_rate * vdb / (sqrt(sdb) + epsilon)
```

- Hyperparameters for Adam:
  - Learning rate: needed to be tuned.
  - `beta1`: parameter of the momentum - `0.9` is recommended by default.
  - `beta2`: parameter of the RMSprop - `0.999` is recommended by default.
  - `epsilon`: `10^-8` is recommended by default.

Learning rate decay

- Slowly reduce learning rate.
- As mentioned before mini-batch gradient descent won't reach the optimum point (converge). But by making the learning rate decay with iterations it will be much closer to it because the steps (and possible oscillations) near the optimum are smaller.
- One technique equations is `learning_rate = (1 / (1 + decay_rate * epoch_num)) * learning_rate_0`
  - `epoch_num` is over all data (not a single mini-batch).
- Other learning rate decay methods (continuous):
  - `learning_rate = (0.95 ^ epoch_num) * learning_rate_0`
  - `learning_rate = (k / sqrt(epoch_num)) * learning_rate_0`
- Some people perform learning rate decay discretely - repeatedly decrease after some number of epochs.
- Some people are making changes to the learning rate manually.
- `decay_rate` is another `hyperparameter`.
- For Andrew Ng, learning rate decay has less priority.

The problem of local optima

- The normal local optima is not likely to appear in a deep neural network because data is usually high dimensional. For point to be a local optima it has to be a local optima for each of the dimensions which is highly unlikely.
- It's unlikely to get stuck in a bad local optima in high dimensions, it is much more likely to get to the saddle point rather to the local optima, which is not a problem.
- Plateaus can make learning slow:
  - Plateau is a region where the derivative is close to zero for a long time.
  - This is where algorithms like momentum, RMSprop or Adam can help.

Hyperparameter tuning, Batch Normalization and Programming Frameworks

Tuning process

- We need to tune our hyperparameters to get the best out of them.
- Hyperparameters importance are (as for Andrew Ng):
  i. Learning rate.
  ii. Momentum beta.
  iii. Mini-batch size.
  iv. No. of hidden units.
  v. No. of layers.
  vi. Learning rate decay.
  vii. Regularization lambda.
  viii. Activation functions.
  ix. Adam `beta1` & `beta2`.
- Its hard to decide which hyperparameter is the most important in a problem. It depends a lot on your problem.
- One of the ways to tune is to sample a grid with `N` hyperparameter settings and then try all settings combinations on your problem.
- Try random values: don't use a grid.
- You can use `Coarse to fine sampling scheme`:
  - When you find some hyperparameters values that give you a better performance - zoom into a smaller region around these values and sample more densely within this space.
- These methods can be automated.

Using an appropriate scale to pick hyperparameters

- Let's say you have a specific range for a hyperparameter from "a" to "b". It's better to search for the right ones using the logarithmic scale rather then in linear scale:
  - Calculate: `a_log = log(a) # e.g. a = 0.0001 then a_log = -4`

- Calculate: `b_log = log(b) # e.g. b = 1 then b_log = 0`
- Then:

```
r = (a_log - b_log) * np.random.rand() + b_log
# In the example the range would be from [-4, 0] because rand range [0,1)
result = 10^r
```

It uniformly samples values in log scale from [a,b].

- If we want to use the last method on exploring on the "momentum beta":
  - Beta best range is from 0.9 to 0.999.
  - You should search for `1 - beta in range 0.001 to 0.1 (1 - 0.9 and 1 - 0.999)` and the use `a = 0.001` and `b = 0.1`. Then:

```
a_log = -3
b_log = -1
r = (a_log - b_log) * np.random.rand() + b_log
beta = 1 - 10^r   # because 1 - beta = 10^r
```

Hyperparameters tuning in practice: Pandas vs. Caviar

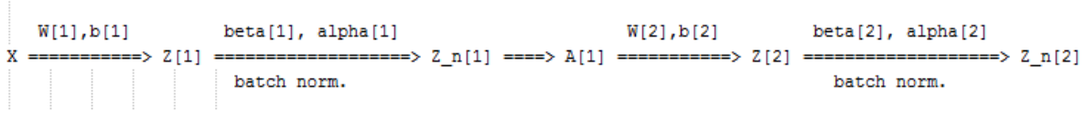
- Intuitions about hyperparameter settings from one application area may or may not transfer to a different one.
- If you don't have much computational resources you can use the "babysitting model":
  - Day 0 you might initialize your parameter as random and then start training.
  - Then you watch your learning curve gradually decrease over the day.
  - And each day you nudge your parameters a little during training.
  - Called panda approach.
- If you have enough computational resources, you can run some models in parallel and at the end of the day(s) you check the results.
  - Called Caviar approach.

Normalizing activations in a network

- In the rise of deep learning, one of the most important ideas has been an algorithm called **batch normalization**, created by two researchers, Sergey Ioffe and Christian Szegedy.
- Batch Normalization speeds up learning.
- Before we normalized input by subtracting the mean and dividing by variance. This helped a lot for the shape of the cost function and for reaching the minimum point faster.
- The question is: *for any hidden layer can we normalize* `A[l]` *to train* `W[l]`, `b[l]` *faster?* This is what batch normalization is about.
- There are some debates in the deep learning literature about whether you should normalize values before the activation function `Z[l]` or after applying the activation function `A[l]`. In practice, normalizing `Z[l]` is done much more often and that is what Andrew Ng presents.
- Algorithm:
  - Given `Z[l] = [z(1), ..., z(m)]`, i = 1 to m (for each input)
  - Compute `mean = 1/m * sum(z[i])`
  - Compute `variance = 1/m * sum((z[i] - mean)^2)`
  - Then `Z_norm[i] = (z[i] - mean) / np.sqrt(variance + epsilon)` (add `epsilon` for numerical stability if variance = 0)
    - Forcing the inputs to a distribution with zero mean and variance of 1.
  - Then `Z_tilde[i] = gamma * Z_norm[i] + beta`
    - To make inputs belong to other distribution (with other mean and variance).
    - gamma and beta are learnable parameters of the model.
    - Making the NN learn the distribution of the outputs.
    - *Note:* if `gamma = sqrt(variance + epsilon)` and `beta = mean` then `Z_tilde[i] = z[i]`

Fitting Batch Normalization into a neural network

- Using batch norm in 3 hidden layers NN:



- Our NN parameters will be:
  - `W[1]`, `b[1]`, ..., `W[L]`, `b[L]`, `beta[1]`, `gamma[1]`, ..., `beta[L]`, `gamma[L]`
  - `beta[1]`, `gamma[1]`, ..., `beta[L]`, `gamma[L]` are updated using any optimization algorithms (like GD, RMSprop, Adam)
- If you are using a deep learning framework, you won't have to implement batch norm yourself:
  - Ex. in Tensorflow you can add this line: `tf.nn.batch-normalization()`

- Batch normalization is usually applied with mini-batches.
- If we are using batch normalization parameters `b[1]`, ..., `b[L]` doesn't count because they will be eliminated after mean subtraction step, so:

```
Z[l] = W[l]A[l-1] + b[l] => Z[l] = W[l]A[l-1]
Z_norm[l] = ...
Z_tilde[l] = gamma[l] * Z_norm[l] + beta[l]
```

  - Taking the mean of a constant `b[l]` will eliminate the `b[l]`
- So if you are using batch normalization, you can remove b[l] or make it always zero.
- So the parameters will be `W[l]`, `beta[l]`, and `alpha[l]`.
- Shapes:
  - `Z[l] - (n[l], m)`
  - `beta[l] - (n[l], m)`
  - `gamma[l] - (n[l], m)`

Why does Batch normalization work?

- The first reason is the same reason as why we normalize X.
- The second reason is that batch normalization reduces the problem of input values changing (shifting).
- Batch normalization does some regularization:
  - Each mini batch is scaled by the mean/variance computed of that mini-batch.
  - This adds some noise to the values `Z[l]` within that mini batch. So similar to dropout it adds some noise to each hidden layer's activations.
  - This has a slight regularization effect.
  - Using bigger size of the mini-batch you are reducing noise and therefore regularization effect.
  - Don't rely on batch normalization as a regularization. It's intended for normalization of hidden units, activations and therefore speeding up learning. For regularization use other regularization techniques (L2 or dropout).

Batch normalization at test time

- When we train a NN with Batch normalization, we compute the mean and the variance of the mini-batch.
- In testing we might need to process examples one at a time. The mean and the variance of one example won't make sense.
- We have to compute an estimated value of mean and variance to use it in testing time.
- We can use the weighted average across the mini-batches.
- We will use the estimated values of the mean and variance to test.
- This method is also sometimes called "Running average".
- In practice most often you will use a deep learning framework and it will contain some default implementation of doing such a thing.

Softmax Regression

- In every example we have used so far we were talking about binary classification.
- There are a generalization of logistic regression called Softmax regression that is used for multiclass classification/regression.
- For example if we are classifying by classes `dog`, `cat`, `baby chick` and `none of that`
  - Dog `class = 1`
  - Cat `class = 2`
  - Baby chick `class = 3`
  - None `class = 0`
  - To represent a dog vector `y = [0 1 0 0]`
  - To represent a cat vector `y = [0 0 1 0]`
  - To represent a baby chick vector `y = [0 0 0 1]`
  - To represent a none vector `y = [1 0 0 0]`
- Notations:
  - `C = no. of classes`
  - Range of classes is `(0, ..., C-1)`
  - In output layer `Ny = C`
- Each of C values in the output layer will contain a probability of the example to belong to each of the classes.
- In the last layer we will have to activate the Softmax activation function instead of the sigmoid activation.
- Softmax activation equations:

```
t = e^(Z[L])                    # shape(C, m)
A[L] = e^(Z[L]) / sum(t)        # shape(C, m), sum(t) - sum of t's for each example (shape (1, m))
```

Training a Softmax classifier

- There's an activation which is called hard max, which gets 1 for the maximum value and zeros for the others.
  - If you are using NumPy, its `np.max` over the vertical axis.
- The Softmax name came from softening the values and not harding them like hard max.
- Softmax is a generalization of logistic activation function to `C` classes. If `C = 2` softmax reduces to logistic regression.
- The loss function used with softmax:

```
L(y, y_hat) = - sum(y[j] * log(y_hat[j])) # j = 0 to C-1
```

- The cost function used with softmax:

```
J(w[1], b[1], ...) = - 1 / m * (sum(L(y[i], y_hat[i]))) # i = 0 to m
```

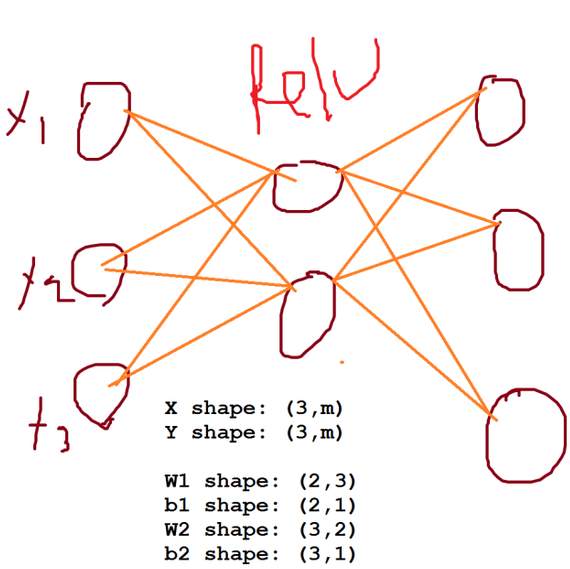
- Back propagation with softmax:

```
dZ[L] = Y_hat - Y
```

- The derivative of softmax is:

```
Y_hat * (1 - Y_hat)
```

- Example:




```
Z1 = W1* X + b1      #shape (2,m)
A1 = RELU(Z1)        #shape (2,m)

Z2 = W2 * A1 + b2    #shape (3,m)

Then we need to get A of softmax
We compute:
t = e^Z2    #shape (3,m)
sumAll = sum(t,C)      #shape (1,m)

Finally
A2 = t / sumAll
```

$$\begin{bmatrix} 1.2 \\ 0.9 \\ 0.4 \end{bmatrix} \rightarrow \boxed{\text{Softmax}} \rightarrow \begin{bmatrix} 0.46 \\ 0.34 \\ 0.20 \end{bmatrix}$$

Deep learning frameworks

- It's not practical to implement everything from scratch. Our numpy implementations were to know how NN works.
- There are many good deep learning frameworks.
- Deep learning is now in the phase of doing something with the frameworks and not from scratch to keep on going.
- Here are some of the leading deep learning frameworks:
  - Caffe/ Caffe2
  - CNTK
  - DL4j
  - Keras
  - Lasagne
  - mxnet
  - PaddlePaddle
  - TensorFlow
  - Theano
  - Torch/Pytorch
- These frameworks are getting better month by month. Comparison between them can be found here.
- How to choose deep learning framework:
  - Ease of programming (development and deployment)
  - Running speed
  - Truly open (open source with good governance)
- Programming frameworks can not only shorten your coding time but sometimes also perform optimizations that speed up your code.

TensorFlow

- In this section we will learn the basic structure of TensorFlow programs.
- Lets see how to implement a minimization function:
  - Example function: `J(w) = w^2 - 10w + 25`
  - The result should be `w = 5` as the function is `(w-5)^2 = 0`
  - Code v.1:

```
import numpy as np
import tensorflow as tf


w = tf.Variable(0, dtype=tf.float32)                # creating a variable w
cost = tf.add(tf.add(w**2, tf.multiply(-10.0, w)), 25.0)        # can be written as this - cost = w**2 - 10
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
session.run(w)    # Runs the definition of w, if you print this it will print zero
session.run(train)

print("W after one iteration:", session.run(w))

for i in range(1000):
        session.run(train)

print("W after 1000 iterations:", session.run(w))
```

  - Code v.2 (we feed the inputs to the algorithm through coefficients):

```
import numpy as np
import tensorflow as tf


coefficients = np.array([[1.], [-10.], [25.]])

x = tf.placeholder(tf.float32, [3, 1])
w = tf.Variable(0, dtype=tf.float32)                # Creating a variable w
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]

train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
session.run(w)    # Runs the definition of w, if you print this it will print zero
session.run(train, feed_dict={x: coefficients})

print("W after one iteration:", session.run(w))

for i in range(1000):
        session.run(train, feed_dict={x: coefficients})

print("W after 1000 iterations:", session.run(w))
```

- In TensorFlow you implement only the forward propagation and TensorFlow will do the backpropagation by itself.
- In TensorFlow a placeholder is a variable you can assign a value to later.
- If you are using a mini-batch training you should change the `feed_dict={x: coefficients}` to the current mini-batch data.
- Almost all TensorFlow programs use this:

```
with tf.Session() as session:       # better for cleaning up in case of error/exception
        session.run(init)
        session.run(w)
```

- In deep learning frameworks there are a lot of things that you can do with one line of code like changing the optimizer.

Side notes:

- Writing and running programs in TensorFlow has the following steps:
  i. Create Tensors (variables) that are not yet executed/evaluated.
  ii. Write operations between those Tensors.
  iii. Initialize your Tensors.
  iv. Create a Session.
  v. Run the Session. This will run the operations you'd written above.

- Instead of needing to write code to compute the cost function we know, we can use this line in TensorFlow : `tf.nn.sigmoid_cross_entropy_with_logits(logits = ..., labels = ...)`
- To initialize weights in NN using TensorFlow use:

```
W1 = tf.get_variable("W1", [25,12288], initializer = tf.contrib.layers.xavier_initializer(seed = 1))

b1 = tf.get_variable("b1", [25,1], initializer = tf.zeros_initializer())
```

- For 3-layer NN, it is important to note that the forward propagation stops at `Z3`. The reason is that in TensorFlow the last linear layer output is given as input to the function computing the loss. Therefore, you don't need `A3`!
- To reset the graph use `tf.reset_default_graph()`

Extra Notes

- If you want a good papers in deep learning look at the ICLR proceedings (Or NIPS proceedings) and that will give you a really good view of the field.
- Who is Yuanqing Lin?
  - Head of Baidu research.
  - First one to win ImageNet
  - Works in PaddlePaddle deep learning platform.

These Notes were made by Mahmoud Badry @2017

- Be able to prioritize the most promising directions for reducing error
- Understand complex ML settings, such as mismatched training/test sets, and comparing to and/or surpassing human-level performance
- Know how to apply end-to-end learning, transfer learning, and multi-task learning

I've seen teams waste months or years through not understanding the principles taught in this course. I hope this two week course will save you months of time.

This is a standalone course, and you can take this so long as you have basic machine learning knowledge. This is the third course in the Deep Learning Specialization.

ML Strategy 1

Why ML Strategy

- You have a lot of ideas for how to improve the accuracy of your deep learning system:
  - Collect more data.
  - Collect more diverse training set.
  - Train algorithm longer with gradient descent.
  - Try different optimization algorithm (e.g. Adam).
  - Try bigger network.
  - Try smaller network.
  - Try dropout.
  - Add L2 regularization.
  - Change network architecture (activation functions, # of hidden units, etc.)
- This course will give you some strategies to help analyze your problem to go in a direction that will help you get better results.

Orthogonalization

- Some deep learning developers know exactly what hyperparameter to tune in order to try to achieve one effect. This is a process we call orthogonalization.
- In orthogonalization, you have some controls, but each control does a specific task and doesn't affect other controls.
- For a supervised learning system to do well, you usually need to tune the knobs of your system to make sure that four things hold true - chain of assumptions in machine learning:
  i. You'll have to fit training set well on cost function (near human level performance if possible).
     - If it's not achieved you could try bigger network, another optimization algorithm (like Adam)...
  ii. Fit dev set well on cost function.
     - If its not achieved you could try regularization, bigger training set...
  iii. Fit test set well on cost function.
     - If its not achieved you could try bigger dev. set...
  iv. Performs well in real world.
     - If its not achieved you could try change dev. set, change cost function...

Single number evaluation metric

- Its better and faster to set a single number evaluation metric for your project before you start it.
- Difference between precision and recall (in cat classification example):
  - Suppose we run the classifier on 10 images which are 5 cats and 5 non-cats. The classifier identifies that there are 4 cats, but it identified 1 wrong cat.
  - Confusion matrix:

	Predicted cat	**Predicted non-cat**
Actual cat	3	2
Actual non-cat	1	4

  - **Precision**: percentage of true cats in the recognized result: P = 3/(3 + 1)
  - **Recall**: percentage of true recognition cat of the all cat predictions: R = 3/(3 + 2)
  - **Accuracy**: (3+4)/10
- Using a precision/recall for evaluation is good in a lot of cases, but separately they don't tell you which algothims is better. Ex:

Classifier	Precision	Recall
A	95%	90%
B	98%	85%

- A better thing is to combine precision and recall in one single (real) number evaluation metric. There a metric called `F1` score, which combines them
  - You can think of `F1` score as an average of precision and recall `F1 = 2 / ((1/P) + (1/R))`

Satisfying and Optimizing metric

- Its hard sometimes to get a single number evaluation metric. Ex:

Classifier	F1	Running time
A	90%	80 ms
B	92%	95 ms
C	92%	1,500 ms

- So we can solve that by choosing a single optimizing metric and decide that other metrics are satisfying. Ex:

```
Maximize F1                    # optimizing metric
subject to running time < 100ms # satisficing metric
```

- So as a general rule:

```
Maximize 1     # optimizing metric (one optimizing metric)
subject to N-1 # satisficing metric (N-1 satisficing metrics)
```

Train/dev/test distributions

- Dev and test sets have to come from the same distribution.
- Choose dev set and test set to reflect data you expect to get in the future and consider important to do well on.
- Setting up the dev set, as well as the validation metric is really defining what target you want to aim at.

Size of the dev and test sets

- An old way of splitting the data was 70% training, 30% test or 60% training, 20% dev, 20% test.
- The old way was valid for a number of examples ~ <100000
- In the modern deep learning if you have a million or more examples a reasonable split would be 98% training, 1% dev, 1% test.

When to change dev/test sets and metrics

- Let's take an example. In a cat classification example we have these metric results:

Metric	Classification error
Algorithm A	3% error (But a lot of porn images are treated as cat images here)
Algorithm B	5% error

  - In the last example if we choose the best algorithm by metric it would be "A", but if the users decide it will be "B"
  - Thus in this case, we want and need to change our metric.
  - `OldMetric = (1/m) * sum(y_pred[i] != y[i] ,m)`
    - Where m is the number of Dev set items.
  - `NewMetric = (1/sum(w[i])) * sum(w[i] * (y_pred[i] != y[i]) ,m)`
    - where:
      - `w[i] = 1 if x[i] is not porn`
      - `w[i] = 10 if x[i] is porn`

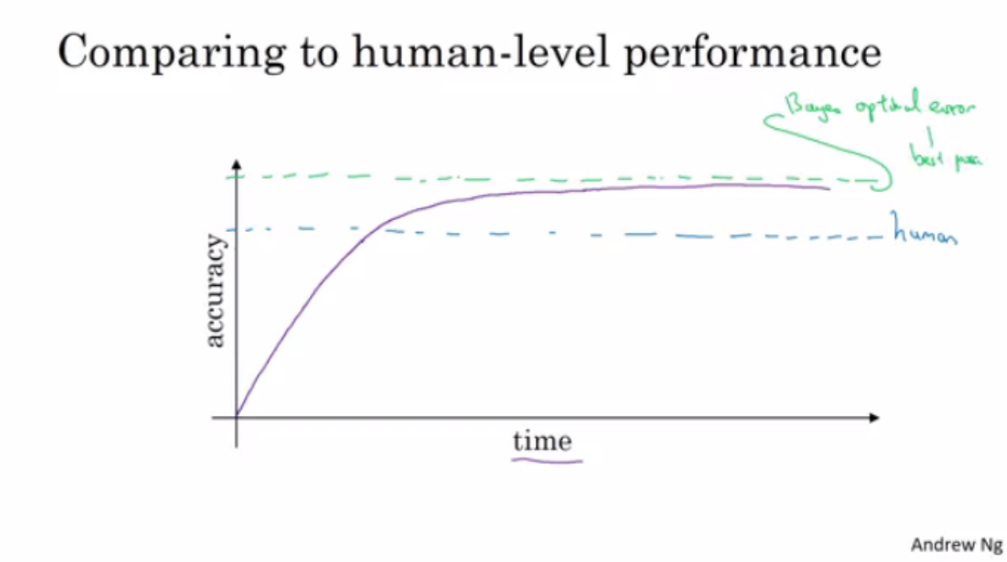
- This is actually an example of an orthogonalization where you should take a machine learning problem and break it into distinct steps:

  i. Figure out how to define a metric that captures what you want to do - place the target.
  ii. Worry about how to actually do well on this metric - how to aim/shoot accurately at the target.

- Conclusion: if doing well on your metric + dev/test set doesn't correspond to doing well in your application, change your metric and/or dev/test set.

Why human-level performance?

- We compare to human-level performance because of two main reasons:
  i. Because of advances in deep learning, machine learning algorithms are suddenly working much better and so it has become much more feasible in a lot of application areas for machine learning algorithms to actually become competitive with human-level performance.
  ii. It turns out that the workflow of designing and building a machine learning system is much more efficient when you're trying to do something that humans can also do.
- After an algorithm reaches the human level performance the progress and accuracy slow down.



- You won't surpass an error that's called "Bayes optimal error".
- There isn't much error range between human-level error and Bayes optimal error.
- Humans are quite good at a lot of tasks. So as long as Machine learning is worse than humans, you can:
  - Get labeled data from humans.
  - Gain insight from manual error analysis: why did a person get it right?
  - Better analysis of bias/variance.

Avoidable bias

- Suppose that the cat classification algorithm gives these results:

Humans	**1%**	**7.5%**
Training error	8%	8%
Dev Error	10%	10%

  - In the left example, because the human level error is 1% then we have to focus on the **bias**.
  - In the right example, because the human level error is 7.5% then we have to focus on the **variance**.
  - The human-level error as a proxy (estimate) for Bayes optimal error. Bayes optimal error is always less (better), but human-level in most cases is not far from it.
  - You can't do better than Bayes error unless you are overfitting.
  - `Avoidable bias = Training error - Human (Bayes) error`
  - `Variance = Dev error - Training error`

Understanding human-level performance

- When choosing human-level performance, it has to be chosen in the terms of what you want to achieve with the system.
- You might have multiple human-level performances based on the human experience. Then you choose the human-level performance (proxy for Bayes error) that is more suitable for the system you're trying to build.
- Improving deep learning algorithms is harder once you reach a human-level performance.
- Summary of bias/variance with human-level performance:
  i. human-level error (proxy for Bayes error)
    - Calculate `avoidable bias = training error - human-level error`
    - If **avoidable bias** difference is the bigger, then it's *bias* problem and you should use a strategy for **bias** resolving.
  ii. training error
    - Calculate `variance = dev error - training error`
    - If **variance** difference is bigger, then you should use a strategy for **variance** resolving.

iii. Dev error

- So having an estimate of human-level performance gives you an estimate of Bayes error. And this allows you to more quickly make decisions as to whether you should focus on trying to reduce a bias or trying to reduce the variance of your algorithm.
- These techniques will tend to work well until you surpass human-level performance, whereupon you might no longer have a good estimate of Bayes error that still helps you make this decision really clearly.

Surpassing human-level performance

- In some problems, deep learning has surpassed human-level performance. Like:
  - Online advertising.
  - Product recommendation.
  - Loan approval.
- The last examples are not natural perception task, rather learning on structural data. Humans are far better in natural perception tasks like computer vision and speech recognition.
- It's harder for machines to surpass human-level performance in natural perception task. But there are already some systems that achieved it.

Improving your model performance

- The two fundamental asssumptions of supervised learning:
  i. You can fit the training set pretty well. This is roughly saying that you can achieve low **avoidable bias**.
  ii. The training set performance generalizes pretty well to the dev/test set. This is roughly saying that **variance** is not too bad.
- To improve your deep learning supervised system follow these guidelines:
  i. Look at the difference between human level error and the training error - **avoidable bias**.
  ii. Look at the difference between the dev/test set and training set error - **Variance**.
  iii. If **avoidable bias** is large you have these options:
    - Train bigger model.
    - Train longer/better optimization algorithm (like Momentum, RMSprop, Adam).
    - Find better NN architecture/hyperparameters search.
  iv. If **variance** is large you have these options:
    - Get more training data.
    - Regularization (L2, Dropout, data augmentation).
    - Find better NN architecture/hyperparameters search.

ML Strategy 2

Carrying out error analysis

- Error analysis - process of manually examining mistakes that your algorithm is making. It can give you insights into what to do next. E.g.:
  - In the cat classification example, if you have 10% error on your dev set and you want to decrease the error.
  - You discovered that some of the mislabeled data are dog pictures that look like cats. Should you try to make your cat classifier do better on dogs (this could take some weeks)?
  - Error analysis approach:
    - Get 100 mislabeled dev set examples at random.
    - Count up how many are dogs.
    - if 5 of 100 are dogs then training your classifier to do better on dogs will decrease your error up to 9.5% (called ceiling), which can be too little.
    - if 50 of 100 are dogs then you could decrease your error up to 5%, which is reasonable and you should work on that.
- Based on the last example, error analysis helps you to analyze the error before taking an action that could take lot of time with no need.
- Sometimes, you can evaluate multiple error analysis ideas in parallel and choose the best idea. Create a spreadsheet to do that and decide, e.g.:

Image	Dog	Great Cats	blurry	Instagram filters	Comments
1	✓			✓	Pitbull
2	✓		✓	✓	
3					Rainy day at zoo

Image	Dog	Great Cats	blurry	Instagram filters	Comments
4		✓			
....					
% totals	8%	43%	61%	12%	

- In the last example you will decide to work on great cats or blurry images to improve your performance.
- This quick counting procedure, which you can often do in, at most, small numbers of hours can really help you make much better prioritization decisions, and understand how promising different approaches are to work on.

Cleaning up incorrectly labeled data

- DL algorithms are quite robust to random errors in the training set but less robust to systematic errors. But it's OK to go and fix these labels if you can.
- If you want to check for mislabeled data in dev/test set, you should also try error analysis with the mislabeled column. Ex:

Image	Dog	Great Cats	blurry	Mislabeled	Comments
1	✓				
2	✓		✓		
3					
4		✓			
....					
% totals	8%	43%	61%	6%	

  - Then:
    - If overall dev set error: 10%
      - Then errors due to incorrect data: 0.6%
      - Then errors due to other causes: 9.4%
    - Then you should focus on the 9.4% error rather than the incorrect data.
- Consider these guidelines while correcting the dev/test mislabeled examples:
  - Apply the same process to your dev and test sets to make sure they continue to come from the same distribution.
  - Consider examining examples your algorithm got right as well as ones it got wrong. (Not always done if you reached a good accuracy)
  - Train and (dev/test) data may now come from a slightly different distributions.
  - It's very important to have dev and test sets to come from the same distribution. But it could be OK for a train set to come from slightly other distribution.

Build your first system quickly, then iterate

- The steps you take to make your deep learning project:
  - Setup dev/test set and metric
  - Build initial system quickly
  - Use Bias/Variance analysis & Error analysis to prioritize next steps.

Training and testing on different distributions

- A lot of teams are working with deep learning applications that have training sets that are different from the dev/test sets due to the hunger of deep learning to data.
- There are some strategies to follow up when training set distribution differs from dev/test sets distribution.
  - Option one (not recommended): shuffle all the data together and extract randomly training and dev/test sets.
    - Advantages: all the sets now come from the same distribution.
    - Disadvantages: the other (real world) distribution that was in the dev/test sets will occur less in the new dev/test sets and that might be not what you want to achieve.
  - Option two: take some of the dev/test set examples and add them to the training set.
    - Advantages: the distribution you care about is your target now.
    - Disadvantage: the distributions in training and dev/test sets are now different. But you will get a better performance over a long time.

Bias and Variance with mismatched data distributions

- Bias and Variance analysis changes when training and Dev/test set is from the different distribution.

- Example: the cat classification example. Suppose you've worked in the example and reached this
  - Human error: 0%
  - Train error: 1%
  - Dev error: 10%
  - In this example, you'll think that this is a variance problem, but because the distributions aren't the same you can't tell for sure. Because it could be that train set was easy to train on, but the dev set was more difficult.
- To solve this issue we create a new set called train-dev set as a random subset of the training set (so it has the same distribution) and we get:
  - Human error: 0%
  - Train error: 1%
  - Train-dev error: 9%
  - Dev error: 10%
  - Now we are sure that this is a high variance problem.
- Suppose we have a different situation:
  - Human error: 0%
  - Train error: 1%
  - Train-dev error: 1.5%
  - Dev error: 10%
  - In this case we have something called *Data mismatch* problem.
- Conclusions:
  i. Human-level error (proxy for Bayes error)
  ii. Train error
    - Calculate `avoidable bias = training error - human level error`
    - If the difference is big then its **Avoidable bias** problem then you should use a strategy for high **bias**.
  iii. Train-dev error
    - Calculate `variance = training-dev error - training error`
    - If the difference is big then its high **variance** problem then you should use a strategy for solving it.
  iv. Dev error
    - Calculate `data mismatch = dev error - train-dev error`
    - If difference is much bigger then train-dev error its **Data mismatch** problem.
  v. Test error
    - Calculate `degree of overfitting to dev set = test error - dev error`
    - Is the difference is big (positive) then maybe you need to find a bigger dev set (dev set and test set come from the same distribution, so the only way for there to be a huge gap here, for it to do much better on the dev set than the test set, is if you somehow managed to overfit the dev set).
- Unfortunately, there aren't many systematic ways to deal with data mismatch. There are some things to try about this in the next section.

Addressing data mismatch

- There aren't completely systematic solutions to this, but there some things you could try.

1. Carry out manual error analysis to try to understand the difference between training and dev/test sets.
2. Make training data more similar, or collect more data similar to dev/test sets.

- If your goal is to make the training data more similar to your dev set one of the techniques you can use **Artificial data synthesis** that can help you make more training data.
  - Combine some of your training data with something that can convert it to the dev/test set distribution.
    - Examples:
      a. Combine normal audio with car noise to get audio with car noise example.
      b. Generate cars using 3D graphics in a car classification example.
  - Be cautious and bear in mind whether or not you might be accidentally simulating data only from a tiny subset of the space of all possible examples because your NN might overfit these generated data (like particular car noise or a particular design of 3D graphics cars).

Transfer learning

- Apply the knowledge you took in a task A and apply it in another task B.
- For example, you have trained a cat classifier with a lot of data, you can use the part of the trained NN it to solve x-ray classification problem.
- To do transfer learning, delete the last layer of NN and it's weights and:
  i. Option 1: if you have a small data set - keep all the other weights as a fixed weights. Add a new last layer(-s) and initialize the new layer weights and feed the new data to the NN and learn the new weights.
  ii. Option 2: if you have enough data you can retrain all the weights.
- Option 1 and 2 are called **fine-tuning** and training on task A called **pretraining**.

- When transfer learning make sense:
  - Task A and B have the same input X (e.g. image, audio).
  - You have a lot of data for the task A you are transferring from and relatively less data for the task B your transferring to.
  - Low level features from task A could be helpful for learning task B.

Multi-task learning

- Whereas in transfer learning, you have a sequential process where you learn from task A and then transfer that to task B. In multi-task learning, you start off simultaneously, trying to have one neural network do several things at the same time. And then each of these tasks helps hopefully all of the other tasks.
- Example:
  - You want to build an object recognition system that detects pedestrians, cars, stop signs, and traffic lights (image has multiple labels).
  - Then Y shape will be `(4,m)` because we have 4 classes and each one is a binary one.
  - Then
    `Cost = (1/m) * sum(sum(L(y_hat(i)_j, y(i)_j))), i = 1..m, j = 1..4`, where
    `L = - y(i)_j * log(y_hat(i)_j) - (1 - y(i)_j) * log(1 - y_hat(i)_j)`
- In the last example you could have trained 4 neural networks separately but if some of the earlier features in neural network can be shared between these different types of objects, then you find that training one neural network to do four things results in better performance than training 4 completely separate neural networks to do the four tasks separately.
- Multi-task learning will also work if y isn't complete for some labels. For example:

```
Y = [1 ? 1 ...]
    [0 0 1 ...]
    [? 1 ? ...]
```

  - And in this case it will do good with the missing data, just the loss function will be different:
    `Loss = (1/m) * sum(sum(L(y_hat(i)_j, y(i)_j) for all j which y(i)_j != ?))`
- Multi-task learning makes sense:
  i. Training on a set of tasks that could benefit from having shared lower-level features.
  ii. Usually, amount of data you have for each task is quite similar.
  iii. Can train a big enough network to do well on all the tasks.
- If you can train a big enough NN, the performance of the multi-task learning compared to splitting the tasks is better.
- Today transfer learning is used more often than multi-task learning.

What is end-to-end deep learning?

- Some systems have multiple stages to implement. An end-to-end deep learning system implements all these stages with a single NN.
- Example 1:
  - Speech recognition system:

```
Audio ---> Features --> Phonemes --> Words --> Transcript    # non-end-to-end system
Audio ---------------------------------------> Transcript    # end-to-end deep learning system
```

  - End-to-end deep learning gives data more freedom, it might not use phonemes when training!
- To build the end-to-end deep learning system that works well, we need a big dataset (more data then in non end-to-end system). If we have a small dataset the ordinary implementation could work just fine.
- Example 2:
  - Face recognition system:

```
Image ---------------------> Face recognition    # end-to-end deep learning system
Image --> Face detection --> Face recognition    # deep learning system - best approach for now
```

  - In practice, the best approach is the second one for now.
  - In the second implementation, it's a two steps approach where both parts are implemented using deep learning.
  - Its working well because it's harder to get a lot of pictures with people in front of the camera than getting faces of people and compare them.
  - In the second implementation at the last step, the NN takes two faces as an input and outputs if the two faces are the same person or not.
- Example 3:
  - Machine translation system:

```
English --> Text analysis --> ... --> French    # non-end-to-end system
English ----------------------------> French    # end-to-end deep learning system - best approach
```

    - Here end-to-end deep leaning system works better because we have enough data to build it.
- Example 4:
    - Estimating child's age from the x-ray picture of a hand:

```
Image --> Bones --> Age    # non-end-to-end system - best approach for now
Image ------------> Age    # end-to-end system
```

    - In this example non-end-to-end system works better because we don't have enough data to train end-to-end system.

Whether to use end-to-end deep learning

- Pros of end-to-end deep learning:
    - Let the data speak. By having a pure machine learning approach, your NN learning input from X to Y may be more able to capture whatever statistics are in the data, rather than being forced to reflect human preconceptions.
    - Less hand-designing of components needed.
- Cons of end-to-end deep learning:
    - May need a large amount of data.
    - Excludes potentially useful hand-design components (it helps more on the smaller dataset).
- Applying end-to-end deep learning:
    - Key question: Do you have sufficient data to learn a function of the **complexity** needed to map x to y?
    - Use ML/DL to learn some individual components.
    - When applying supervised learning you should carefully choose what types of X to Y mappings you want to learn depending on what task you can get data for.

These Notes were made by Mahmoud Badry @2017

Course summary

Here is the course summary as given on the course link:

> This course will teach you how to build convolutional neural networks and apply it to image data. Thanks to deep learning, computer vision is working far better than just two years ago, and this is enabling numerous exciting applications ranging from safe autonomous driving, to accurate face recognition, to automatic reading of radiology images.
>
> You will:
>
> - Understand how to build a convolutional neural network, including recent variations such as residual networks.
> - Know how to apply convolutional networks to visual detection and recognition tasks.
> - Know to use neural style transfer to generate art.
> - Be able to apply these algorithms to a variety of image, video, and other 2D or 3D data.
>
> This is the fourth course of the Deep Learning Specialization.

Foundations of CNNs

> Learn to implement the foundational layers of CNNs (pooling, convolutions) and to stack them properly in a deep network to solve multi-class image classification problems.

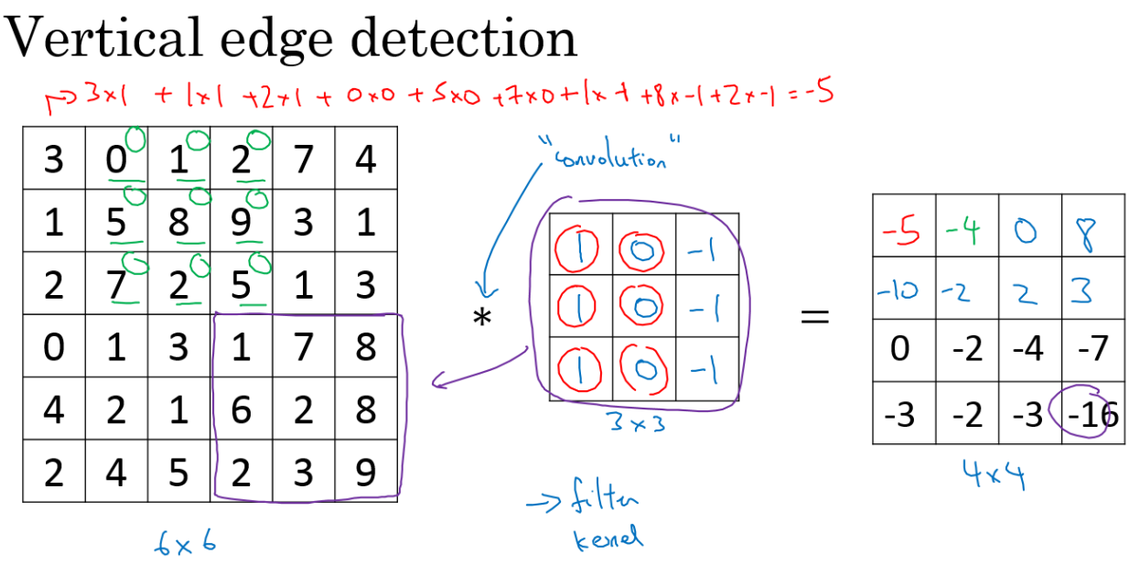
Computer vision

- Computer vision is one of the applications that are rapidly active thanks to deep learning.
- Some of the applications of computer vision that are using deep learning includes:
    - Self driving cars.
    - Face recognition.
- Deep learning is also enabling new types of art to be created.
- Rapid changes to computer vision are making new applications that weren't possible a few years ago.
- Computer vision deep leaning techniques are always evolving making a new architectures which can help us in other areas other than computer vision.
    - For example, Andrew Ng took some ideas of computer vision and applied it in speech recognition.
- Examples of a computer vision problems includes:
    - Image classification.
    - Object detection.
        - Detect object and localize them.
    - Neural style transfer
        - Changes the style of an image using another image.
- One of the challenges of computer vision problem that images can be so large and we want a fast and accurate algorithm to work with that.
    - For example, a `1000x1000` image will represent 3 million feature/input to the full connected neural network. If the following hidden layer contains 1000, then we will want to learn weights of the shape `[1000, 3 million]` which is 3 billion parameter only in the first layer and thats so computationally expensive!
- One of the solutions is to build this using **convolution layers** instead of the **fully connected layers**.

Edge detection example

- The convolution operation is one of the fundamentals blocks of a CNN. One of the examples about convolution is the image edge detection operation.

- Early layers of CNN might detect edges then the middle layers will detect parts of objects and the later layers will put the these parts together to produce an output.
- In an image we can detect vertical edges, horizontal edges, or full edge detector.
- Vertical edge detection:
  - An example of convolution operation to detect vertical edges:



  - In the last example a `6x6` matrix convolved with `3x3` filter/kernel gives us a `4x4` matrix.
  - If you make the convolution operation in TensorFlow you will find the function `tf.nn.conv2d`. In keras you will find `Conv2d` function.
  - The vertical edge detection filter will find a `3x3` place in an image where there are a bright region followed by a dark region.
  - If we applied this filter to a white region followed by a dark region, it should find the edges in between the two colors as a positive value. But if we applied the same filter to a dark region followed by a white region it will give us negative values. To solve this we can use the abs function to make it positive.
- Horizontal edge detection
  - Filter would be like this

```
1       1       1
0       0       0
-1      -1      -1
```

- There are a lot of ways we can put number inside the horizontal or vertical edge detections. For example here are the vertical **Sobel** filter (The idea is taking care of the middle row):

```
1       0       -1
2       0       -2
1       0       -1
```

- Also something called **Scharr** filter (The idea is taking great care of the middle row):

```
3       0       -3
10      0       -10
3       0       -3
```

- What we learned in the deep learning is that we don't need to hand craft these numbers, we can treat them as weights and then learn them. It can learn horizontal, vertical, angled, or any edge type automatically rather than getting them by hand.

Padding

- In order to to use deep neural networks we really need to use **paddings**.
- In the last section we saw that a `6x6` matrix convolved with `3x3` filter/kernel gives us a `4x4` matrix.
- To give it a general rule, if a matrix `nxn` is convolved with `fxf` filter/kernel give us `n-f+1,n-f+1` matrix.
- The convolution operation shrinks the matrix if f>1.
- We want to apply convolution operation multiple times, but if the image shrinks we will lose a lot of data on this process. Also the edges pixels are used less than other pixels in an image.

- So the problems with convolutions are:
  - Shrinks output.
  - throwing away a lot of information that are in the edges.
- To solve these problems we can pad the input image before convolution by adding some rows and columns to it. We will call the padding amount `P` the number of row/columns that we will insert in top, bottom, left and right of the image.
- In almost all the cases the padding values are zeros.
- The general rule now, if a matrix `nxn` is convolved with `fxf` filter/kernel and padding `p` give us `n+2p-f+1,n+2p-f+1` matrix.
- If n = 6, f = 3, and p = 1 Then the output image will have `n+2p-f+1 = 6+2-3+1 = 6`. We maintain the size of the image.
- Same convolutions is a convolution with a pad so that output size is the same as the input size. Its given by the equation:

```
P = (f-1) / 2
```

- In computer vision f is usually odd. Some of the reasons is that its have a center value.

Strided convolution

- Strided convolution is another piece that are used in CNNs.
- We will call stride `S`.
- When we are making the convolution operation we used `S` to tell us the number of pixels we will jump when we are convolving filter/kernel. The last examples we described S was 1.
- Now the general rule are:
  - if a matrix `nxn` is convolved with `fxf` filter/kernel and padding `p` and stride `s` it give us `(n+2p-f)/s + 1,(n+2p-f)/s + 1` matrix.
- In case `(n+2p-f)/s + 1` is fraction we can take **floor** of this value.
- In math textbooks the conv operation is filpping the filter before using it. What we were doing is called cross-correlation operation but the state of art of deep learning is using this as conv operation.
- Same convolutions is a convolution with a padding so that output size is the same as the input size. Its given by the equation:

```
p = (n*s - n + f - s) / 2
When s = 1 ==> P = (f-1) / 2
```

Convolutions over volumes

- We see how convolution works with 2D images, now lets see if we want to convolve 3D images (RGB image)
- We will convolve an image of height, width, # of channels with a filter of a height, width, same # of channels. Hint that the image number channels and the filter number of channels are the same.
- We can call this as stacked filters for each channel!
- Example:
  - Input image: `6x6x3`
  - Filter: `3x3x3`
  - Result image: `4x4x1`
  - In the last result p=0, s=1
- Hint the output here is only 2D.
- We can use multiple filters to detect multiple features or edges. Example.
  - Input image: `6x6x3`
  - 10 Filters: `3x3x3`
  - Result image: `4x4x10`
  - In the last result p=0, s=1

One Layer of a Convolutional Network

- First we convolve some filters to a given input and then add a bias to each filter output and then get RELU of the result. Example:

    - Input image: `6x6x3` `# a0`
    - 10 Filters: `3x3x3` `#W1`
    - Result image: `4x4x10` `#W1a0`
    - Add b (bias) with `10x1` will get us : `4x4x10` image `#W1a0 + b`
    - Apply RELU will get us: `4x4x10` image `#A1 = RELU(W1a0 + b)`
    - In the last result p=0, s=1
    - Hint number of parameters here are: `(3x3x3x10) + 10 = 280`
- The last example forms a layer in the CNN.
- Hint: no matter the size of the input, the number of the parameters is same if filter size is same. That makes it less prone to overfitting.
- Here are some notations we will use. If layer l is a conv layer:

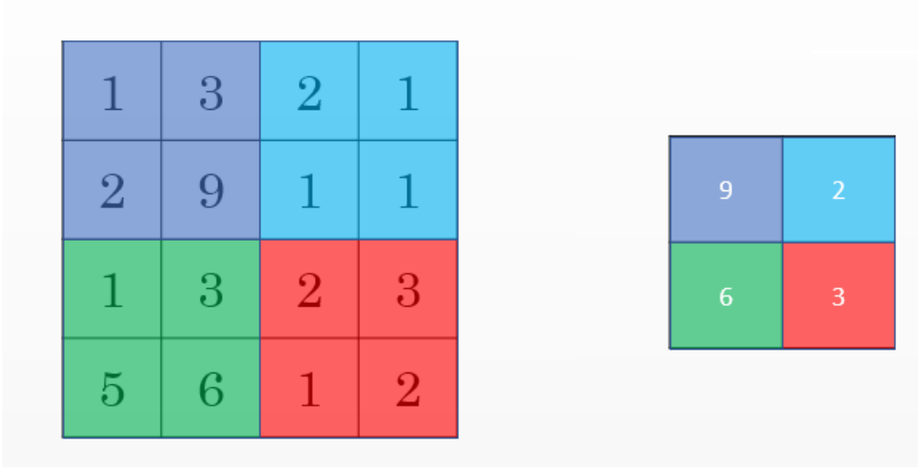
```
Hyperparameters
f[l] = filter size
p[l] = padding  # Default is zero
s[l] = stride
nc[l] = number of filters

Input:  n[l-1] x n[l-1] x nc[l-1]       Or       nH[l-1] x nW[l-1] x nc[l-1]
Output: n[l] x n[l] x nc[l]     Or       nH[l] x nW[l] x nc[l]
Where n[l] = (n[l-1] + 2p[l] - f[l] / s[l]) + 1

Each filter is: f[l] x f[l] x nc[l-1]

Activations: a[l] is nH[l] x nW[l] x nc[l]
                    A[l] is m x nH[l] x nW[l] x nc[l]   # In batch or minbatch training

Weights: f[l] * f[l] * nc[l-1] * nc[l]
bias:  (1, 1, 1, nc[l])
```

A simple convolution network example

- Lets build a big example.
    - Input Image are: `a0 = 39x39x3`
        - `n0 = 39` and `nc0 = 3`
    - First layer (Conv layer):
        - `f1 = 3`, `s1 = 1`, and `p1 = 0`
        - `number of filters = 10`
        - Then output are `a1 = 37x37x10`
            - `n1 = 37` and `nc1 = 10`
    - Second layer (Conv layer):
        - `f2 = 5`, `s2 = 2`, `p2 = 0`
        - `number of filters = 20`
        - The output are `a2 = 17x17x20`
            - `n2 = 17`, `nc2 = 20`
        - Hint shrinking goes much faster because the stride is 2
    - Third layer (Conv layer):
        - `f3 = 5`, `s3 = 2`, `p3 = 0`
        - `number of filters = 40`
        - The output are `a3 = 7x7x40`
            - `n3 = 7`, `nc3 = 40`
    - Forth layer (Fully connected Softmax)
        - `a3 = 7x7x40 = 1960` as a vector..
- In the last example you seen that the image are getting smaller after each layer and thats the trend now.
- Types of layer in a convolutional network:
    - Convolution. `#Conv`
    - Pooling `#Pool`
    - Fully connected `#FC`

Pooling layers

- Other than the conv layers, CNNs often uses pooling layers to reduce the size of the inputs, speed up computation, and to make some of the features it detects more robust.
- Max pooling example:



-
  - This example has `f = 2`, `s = 2`, and `p = 0` hyperparameters
- The max pooling is saying, if the feature is detected anywhere in this filter then keep a high number. But the main reason why people are using pooling because its works well in practice and reduce computations.
- Max pooling has no parameters to learn.
- Example of Max pooling on 3D input:
  - Input: `4x4x10`
  - `Max pooling size = 2` and `stride = 2`
  - Output: `2x2x10`
- Average pooling is taking the averages of the values instead of taking the max values.
- Max pooling is used more often than average pooling in practice.
- If stride of pooling equals the size, it will then apply the effect of shrinking.
- Hyperparameters summary
  - f : filter size.
  - s : stride.
  - Padding are rarely uses here.
  - Max or average pooling.

Convolutional neural network example

- Now we will deal with a full CNN example. This example is something like the ***LeNet-5*** that was invented by Yann Lecun.
  - Input Image are: `a0 = 32x32x3`
    - `n0 = 32` and `nc0 = 3`
  - First layer (Conv layer): `#Conv1`
    - `f1 = 5`, `s1 = 1`, and `p1 = 0`
    - `number of filters = 6`
    - Then output are `a1 = 28x28x6`
      - `n1 = 28` and `nc1 = 6`
    - Then apply (Max pooling): `#Pool1`
      - `f1p = 2`, and `s1p = 2`
      - The output are `a1 = 14x14x6`
  - Second layer (Conv layer): `#Conv2`
    - `f2 = 5`, `s2 = 1`, `p2 = 0`
    - `number of filters = 16`
    - The output are `a2 = 10x10x16`
      - `n2 = 10`, `nc2 = 16`
    - Then apply (Max pooling): `#Pool2`
      - `f2p = 2`, and `s2p = 2`
      - The output are `a2 = 5x5x16`
  - Third layer (Fully connected) `#FC3`
    - Number of neurons are 120
    - The output `a3 = 120 x 1` . 400 came from `5x5x16`
  - Forth layer (Fully connected) `#FC4`
    - Number of neurons are 84
    - The output `a4 = 84 x 1` .
  - Fifth layer (Softmax)
    - Number of neurons is 10 if we need to identify for example the 10 digits.
- Hint a Conv1 and Pool1 is treated as one layer.
- Some statistics about the last example:

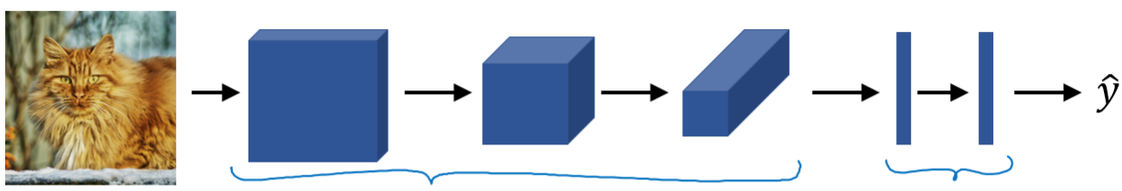
	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	208
POOL1	(14,14,8)	1,568	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
Softmax	(10,1)	10	841

- Hyperparameters are a lot. For choosing the value of each you should follow the guideline that we will discuss later or check the literature and takes some ideas and numbers from it.
- Usually the input size decreases over layers while the number of filters increases.
- A CNN usually consists of one or more convolution (Not just one as the shown examples) followed by a pooling.
- Fully connected layers has the most parameters in the network.
- To consider using these blocks together you should look at other working examples firsts to get some intuitions.

Why convolutions?

- Two main advantages of Convs are:
  - Parameter sharing.
    - A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.
  - sparsity of connections.
    - In each layer, each output value depends only on a small number of inputs which makes it translation invariance.
- Putting it all together:

$$\text{Training set } (x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)}).$$



$$\text{Cost } J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

$$\text{Use gradient descent to optimize parameters to reduce } J$$

Deep convolutional models: case studies

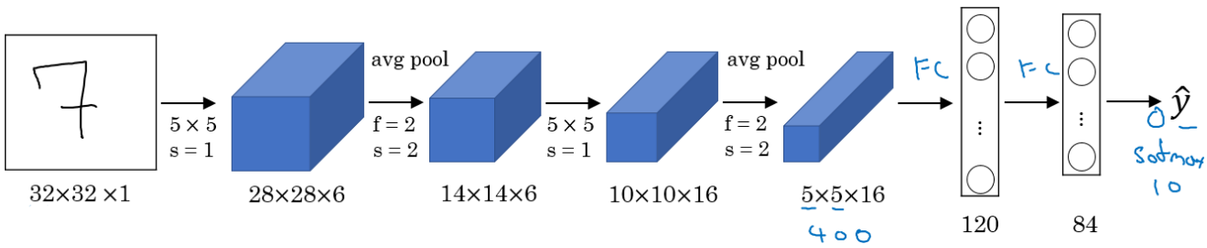
> Learn about the practical tricks and methods used in deep CNNs straight from the research papers.

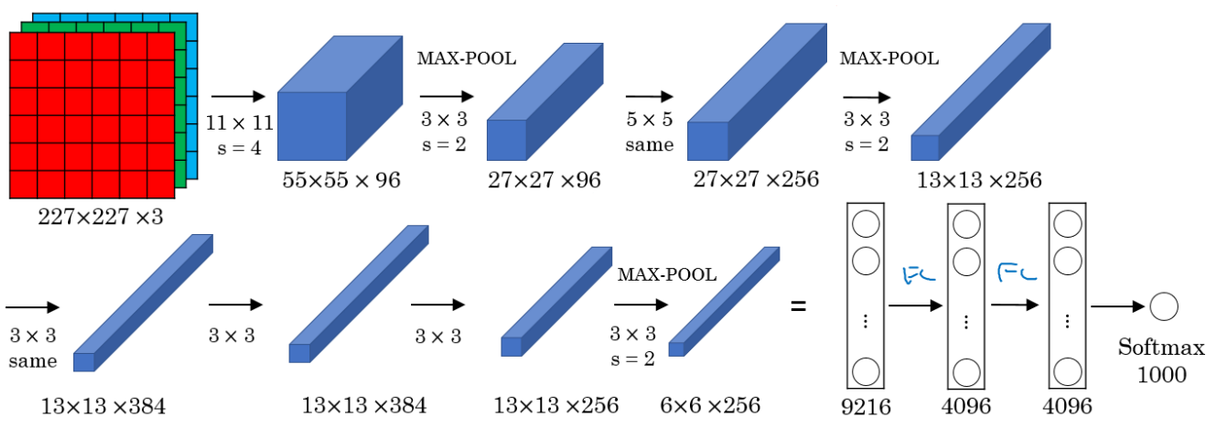
Why look at case studies?

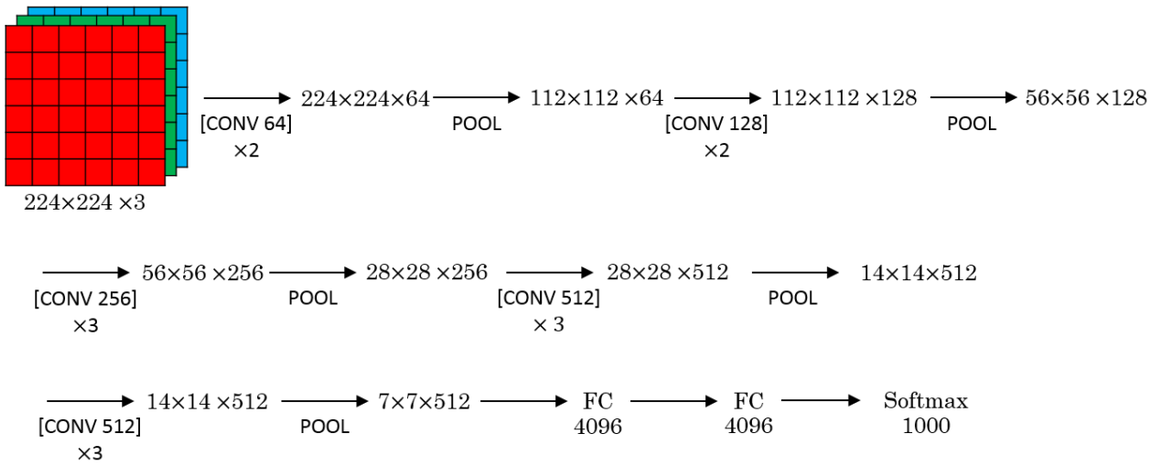
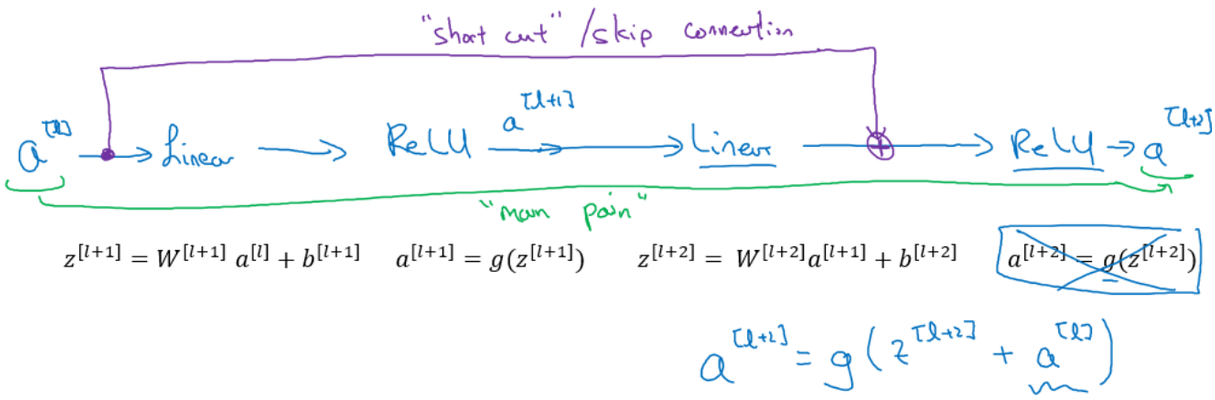
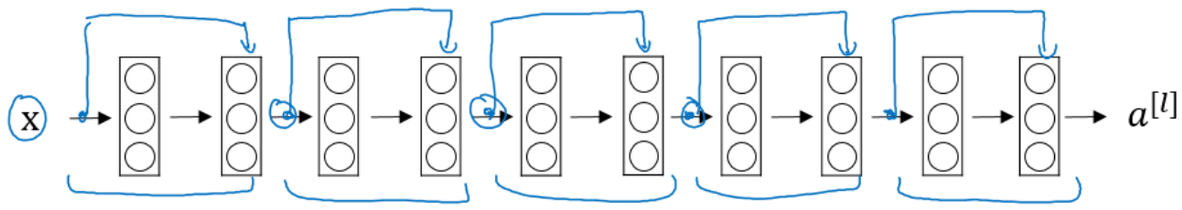
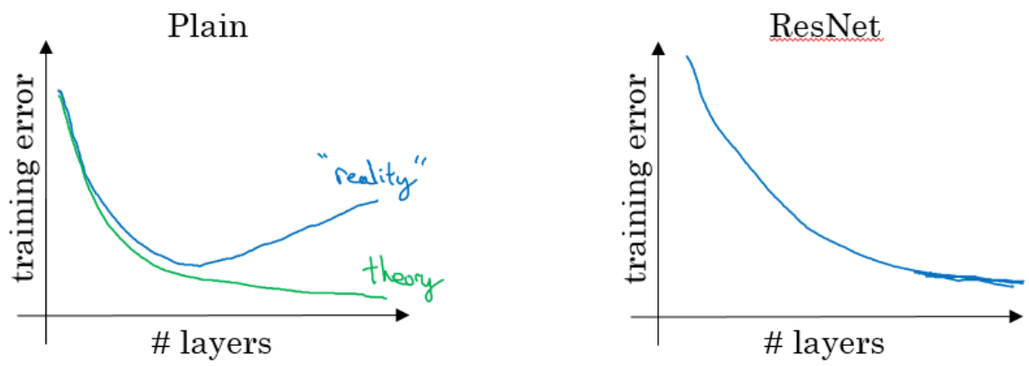
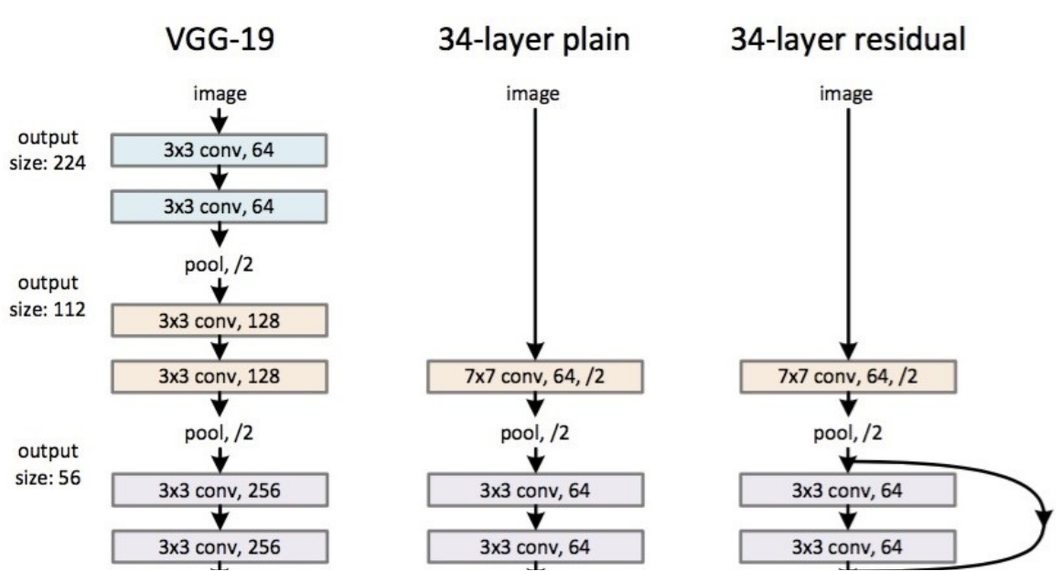
- We learned about Conv layer, pooling layer, and fully connected layers. It turns out that computer vision researchers spent the past few years on how to put these layers together.
- To get some intuitions you have to see the examples that has been made.
- Some neural networks architecture that works well in some tasks can also work well in other tasks.
- Here are some classical CNN networks:
  - **LeNet-5**
  - **AlexNet**
  - **VGG**
- The best CNN architecture that won the last ImageNet competition is called **ResNet** and it has 152 layers!
- There are also an architecture called **Inception** that was made by Google that are very useful to learn and apply to your tasks.

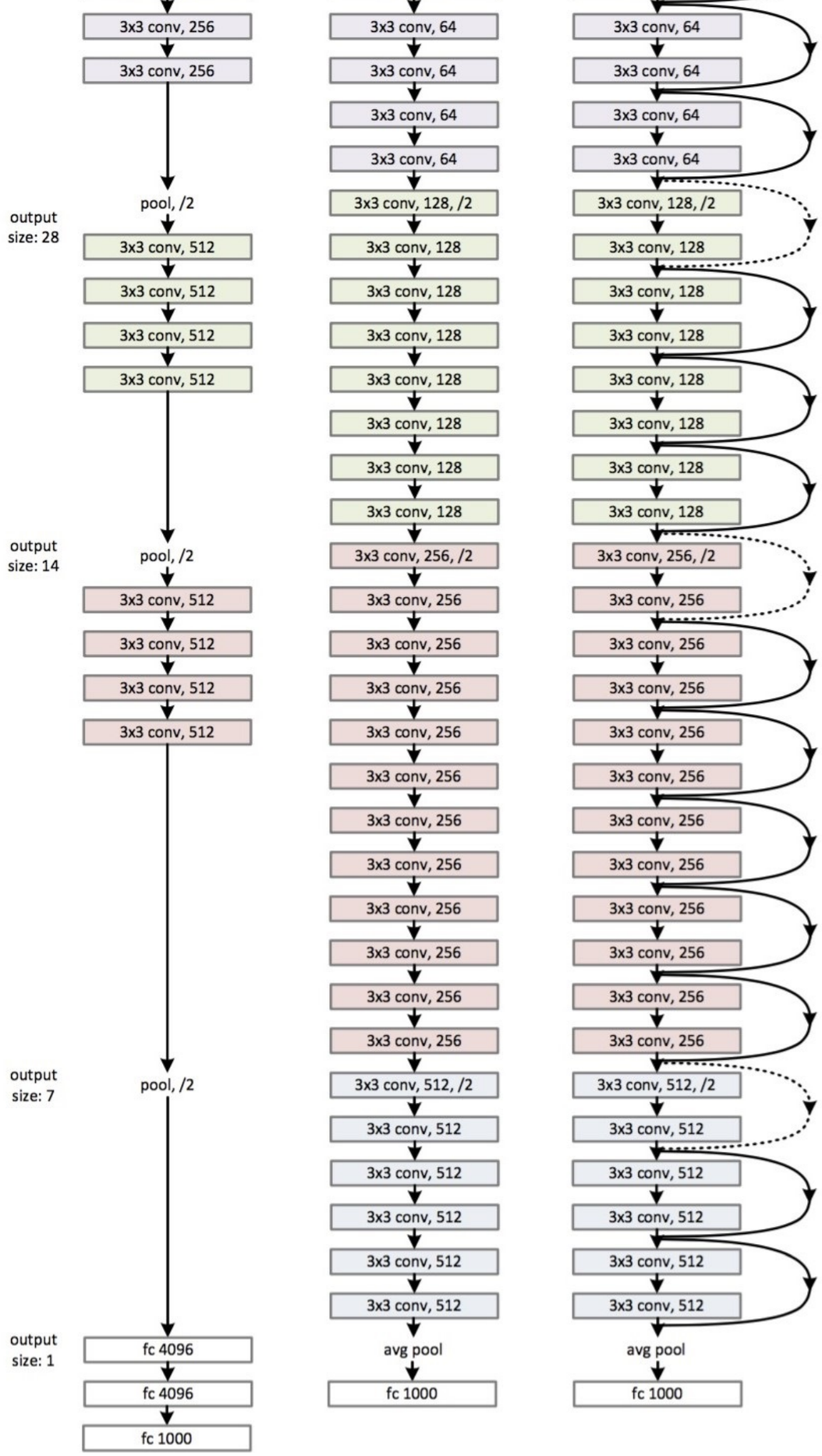
- Reading and trying the mentioned models can boost you and give you a lot of ideas to solve your task.

Classic networks

- In this section we will talk about classic networks which are **LeNet-5**, **AlexNet**, and **VGG**.
- **LeNet-5**
  - The goal for this model was to identify handwritten digits in a `32x32x1` gray image. Here are the drawing of it:
  - 

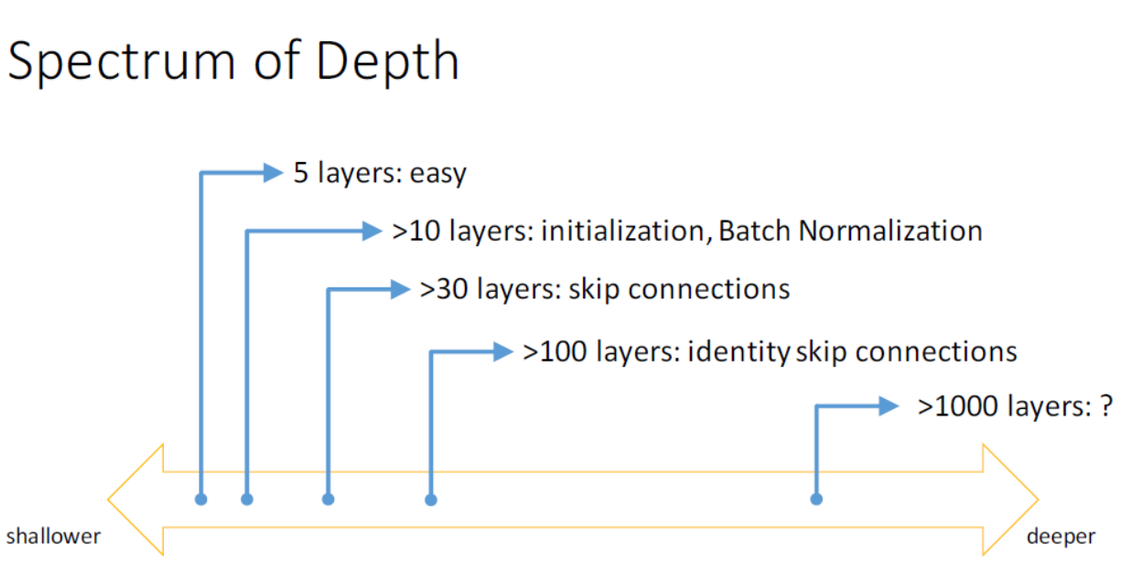
  - This model was published in 1998. The last layer wasn't using softmax back then.
  - It has 60k parameters.
  - The dimensions of the image decreases as the number of channels increases.
  - `Conv ==> Pool ==> Conv ==> Pool ==> FC ==> FC ==> softmax` this type of arrangement is quite common.
  - The activation function used in the paper was Sigmoid and Tanh. Modern implementation uses RELU in most of the cases.
  - [LeCun et al., 1998. Gradient-based learning applied to document recognition]
- **AlexNet**
  - Named after Alex Krizhevsky who was the first author of this paper. The other authors includes Geoffrey Hinton.
  - The goal for the model was the ImageNet challenge which classifies images into 1000 classes. Here are the drawing of the model:
  - 

  - Summary:
    - ```
      Conv => Max-pool => Conv => Max-pool => Conv => Conv => Conv => Max-pool ==> Flatten ==> FC ==> FC
      ==> Softmax
      ```
  - Similar to LeNet-5 but bigger.
  - Has 60 Million parameter compared to 60k parameter of LeNet-5.
  - It used the RELU activation function.
  - The original paper contains Multiple GPUs and Local Response normalization (RN).
    - Multiple GPUs were used because the GPUs were not so fast back then.
    - Researchers proved that Local Response normalization doesn't help much so for now don't bother yourself for understanding or implementing it.
  - This paper convinced the computer vision researchers that deep learning is so important.
  - [Krizhevsky et al., 2012. ImageNet classification with deep convolutional neural networks]
- **VGG-16**
  - A modification for AlexNet.
  - Instead of having a lot of hyperparameters lets have some simpler network.
  - Focus on having only these blocks:
    - CONV = 3 X 3 filter, s = 1, same
    - MAX-POOL = 2 X 2 , s = 2
      ```

- Here are the architecture:



  -
- This network is large even by modern standards. It has around 138 million parameters.
  - Most of the parameters are in the fully connected layers.
- It has a total memory of 96MB per image for only forward propagation!
  - Most memory are in the earlier layers.
- Number of filters increases from 64 to 128 to 256 to 512. 512 was made twice.
- Pooling was the only one who is responsible for shrinking the dimensions.
- There are another version called **VGG-19** which is a bigger version. But most people uses the VGG-16 instead of the VGG-19 because it does the same.
- VGG paper is attractive it tries to make some rules regarding using CNNs.
- [Simonyan & Zisserman 2015. Very deep convolutional networks for large-scale image recognition]

Residual Networks (ResNets)

- Very, very deep NNs are difficult to train because of vanishing and exploding gradients problems.
- In this section we will learn about skip connection which makes you take the activation from one layer and suddenly feed it to another layer even much deeper in NN which allows you to train large NNs even with layers greater than 100.
- **Residual block**
  - ResNets are built out of some Residual blocks.



  -
  - They add a shortcut/skip connection before the second activation.
  - The authors of this block find that you can train a deeper NNs using stacking this block.
  - [He et al., 2015. Deep residual networks for image recognition]
- **Residual Network**
  - Are a NN that consists of some Residual blocks.



  -
  - These networks can go deeper without hurting the performance. In the normal NN - Plain networks - the theory tell us that if we go deeper we will get a better solution to our problem, but because of the vanishing and exploding gradients problems the performance of the network suffers as it goes deeper. Thanks to Residual Network we can go deeper as we want now.



- On the left is the normal NN and on the right are the ResNet. As you can see the performance of ResNet increases as the network goes deeper.
- In some cases going deeper won't effect the performance and that depends on the problem on your hand.
- Some people are trying to train 1000 layer now which isn't used in practice.
- [He et al., 2015. Deep residual networks for image recognition]

Why ResNets work

- Lets see some example that illustrates why resNet work.
  - We have a big NN as the following:
    - `X --> Big NN --> a[l]`
  - Lets add two layers to this network as a residual block:
    - `X --> Big NN --> a[l] --> Layer1 --> Layer2 --> a[l+2]`
    - And a `[l]` has a direct connection to `a[l+2]`
  - Suppose we are using RELU activations.
  - Then:
    - ```
      a[l+2] = g( z[l+2] + a[l] )
                = g( W[l+2] a[l+1] + b[l+2] + a[l] )
      ```
  - Then if we are using L2 regularization for example, `W[l+2]` will be zero. Lets say that `b[l+2]` will be zero too.
  - Then `a[l+2] = g( a[l] ) = a[l]` with no negative values.
  - This show that identity function is easy for a residual block to learn. And that why it can train deeper NNs.
  - Also that the two layers we added doesn't hurt the performance of big NN we made.
  - Hint: dimensions of z[l+2] and a[l] have to be the same in resNets. In case they have different dimensions what we put a matrix parameters (Which can be learned or fixed)
    - `a[l+2] = g( z[l+2] + ws * a[l] ) # The added Ws should make the dimensions equal`
    - ws also can be a zero padding.
- Using a skip-connection helps the gradient to backpropagate and thus helps you to train deeper networks
- Lets take a look at ResNet on images.
  - Here are the architecture of **ResNet-34**:


      ```



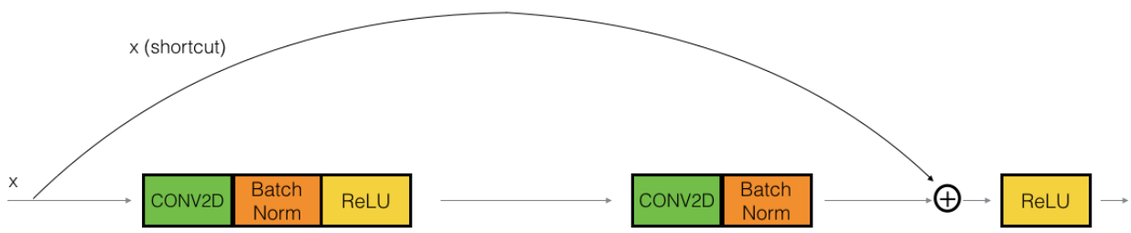
-
  - All the 3x3 Conv are same Convs.
  - Keep it simple in design of the network.
  - spatial size /2 => # filters x2
  - No FC layers, No dropout is used.
  - Two main types of blocks are used in a ResNet, depending mainly on whether the input/output dimensions are same or different. You are going to implement both of them.
  - The dotted lines is the case when the dimensions are different. To solve then they down-sample the input by 2 and then pad zeros to match the two dimensions. There's another trick which is called bottleneck which we will explore later.

- Useful concept (**Spectrum of Depth**):

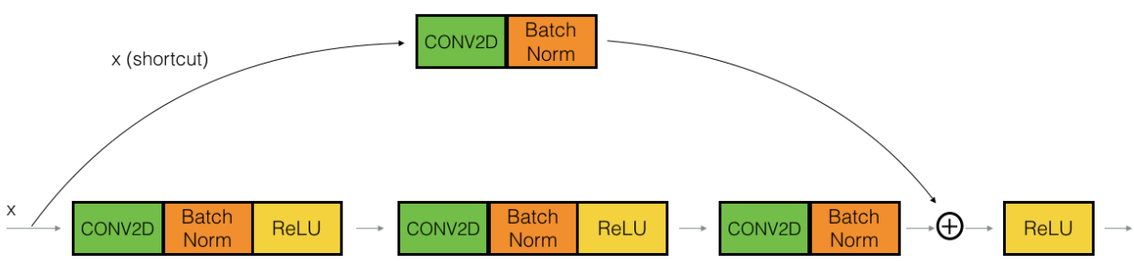


-
- Taken from icml.cc/2016/tutorials/icml2016_tutorial_deep_residual_networks_kaiminghe.pdf

- Residual blocks types:
  - Identity block:



    -
    - Hint the conv is followed by a batch norm `BN` before `RELU`. Dimensions here are same.
    - This skip is over 2 layers. The skip connection can jump n connections where n>2
    - This drawing represents Keras layers.
  - The convolutional block:



    -
    - The conv can be bottleneck 1 x 1 conv

Network in Network and 1 X 1 convolutions

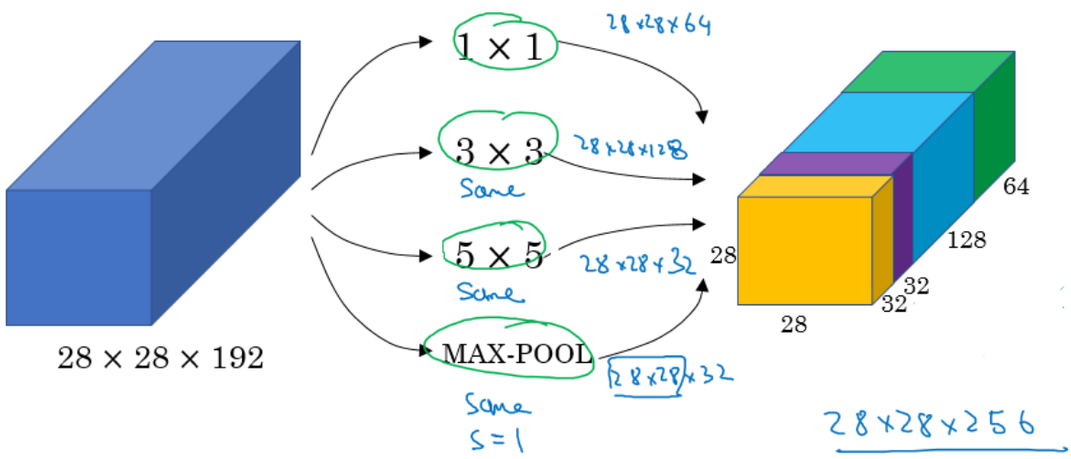
- A 1 x 1 convolution - We also call it Network in Network- is so useful in many CNN models.
- What does a 1 X 1 convolution do? Isn't it just multiplying by a number?
  - Lets first consider an example:
    - Input: `6x6x1`
    - Conv: `1x1x1` one filter. `# The 1 x 1 Conv`
    - Output: `6x6x1`
  - Another example:
    - Input: `6x6x32`
    - Conv: `1x1x32` 5 filters. `# The 1 x 1 Conv`
    - Output: `6x6x5`
- The Network in Network is proposed in [Lin et al., 2013. Network in network]
- It has been used in a lot of modern CNN implementations like ResNet and Inception models.
- A 1 x 1 convolution is useful when:
  - We want to shrink the number of channels. We also call this feature transformation.
    - In the second discussed example above we have shrinked the input from 32 to 5 channels.
  - We will later see that by shrinking it we can save a lot of computations.

- If we have specified the number of 1 x 1 Conv filters to be the same as the input number of channels then the output will contain the same number of channels. Then the 1 x 1 Conv will act like a non linearity and will learn non linearity operator.

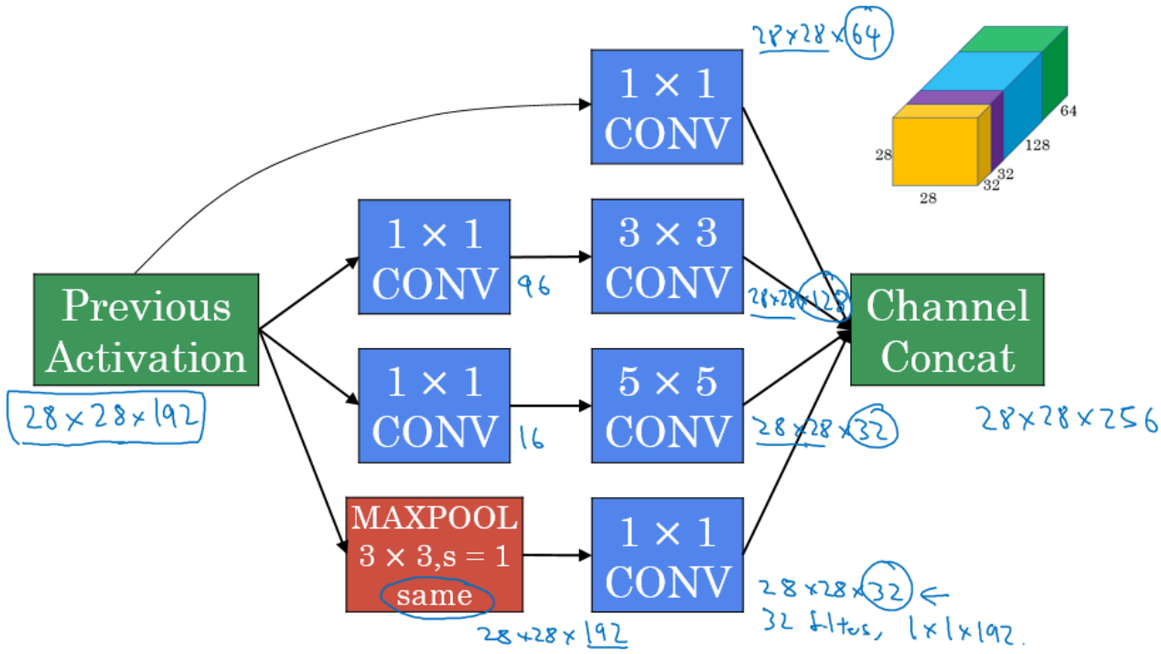
- Replace fully connected layers with 1 x 1 convolutions as Yann LeCun believes they are the same.
  - > In Convolutional Nets, there is no such thing as "fully-connected layers". There are only convolution layers with 1x1 convolution kernels and a full connection table. Yann LeCun
- [Lin et al., 2013. Network in network]

Inception network motivation

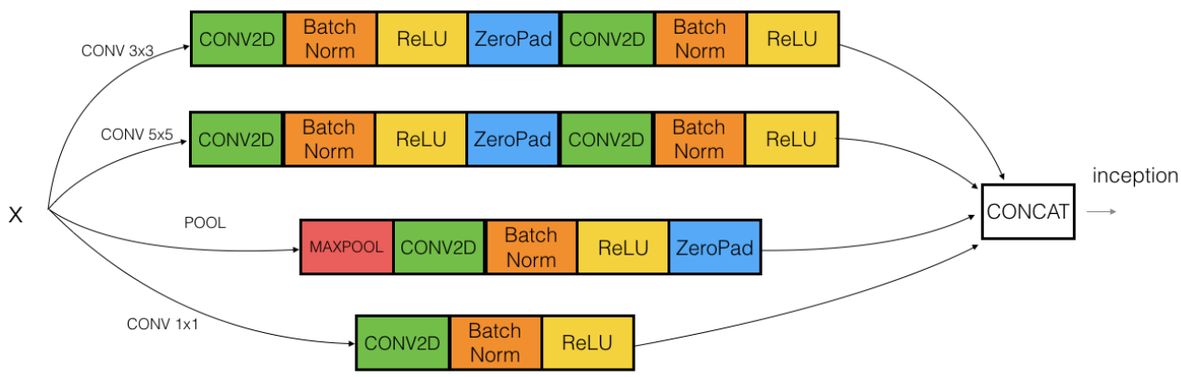
- When you design a CNN you have to decide all the layers yourself. Will you pick a 3 x 3 Conv or 5 x 5 Conv or maybe a max pooling layer. You have so many choices.
- What **inception** tells us is, Why not use all of them at once?
- **Inception module**, naive version:



  - Hint that max-pool are same here.
  - Input to the inception module are 28 x 28 x 192 and the output are 28 x 28 x 256
  - We have done all the Convs and pools we might want and will let the NN learn and decide which it want to use most.
  - [Szegedy et al. 2014. Going deeper with convolutions]
- The problem of computational cost in Inception model:
  - If we have just focused on a 5 x 5 Conv that we have done in the last example.
  - There are 32 same filters of 5 x 5, and the input are 28 x 28 x 192.
  - Output should be 28 x 28 x 32
  - The total number of multiplications needed here are:
    - Number of outputs * Filter size * Filter size * Input dimensions
    - Which equals: `28 * 28 * 32 * 5 * 5 * 192 = 120 Mil`
    - 120 Mil multiply operation still a problem in the modern day computers.
  - Using a 1 x 1 convolution we can reduce 120 mil to just 12 mil. Lets see how.
- Using 1 X 1 convolution to reduce computational cost:
  - The new architecture are:
    - X0 shape is (28, 28, 192)
    - We then apply 16 (1 x 1 Convolution)
    - That produces X1 of shape (28, 28, 16)
      - Hint, we have reduced the dimensions here.
    - Then apply 32 (5 x 5 Convolution)
    - That produces X2 of shape (28, 28, 32)
  - Now lets calculate the number of multiplications:
    - For the first Conv: `28 * 28 * 16 * 1 * 1 * 192 = 2.5 Mil`
    - For the second Conv: `28 * 28 * 32 * 5 * 5 * 16 = 10 Mil`
    - So the total number are 12.5 Mil approx. which is so good compared to 120 Mil
- A 1 x 1 Conv here is called Bottleneck `BN`.
- It turns out that the 1 x 1 Conv won't hurt the performance.
- **Inception module**, dimensions reduction version:

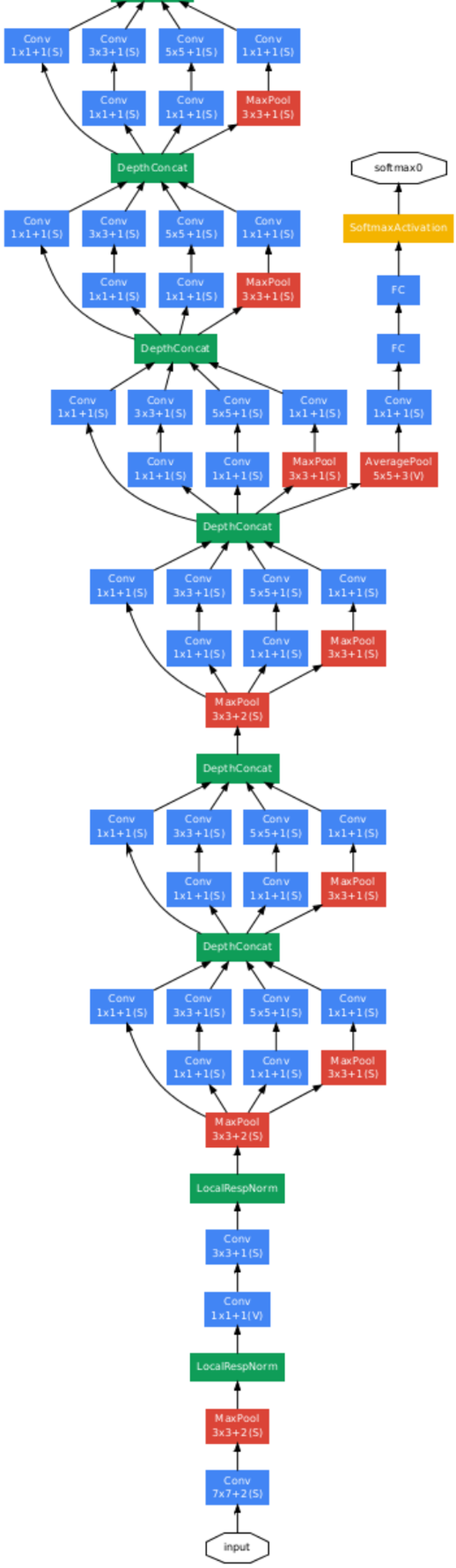


- Example of inception model in Keras:



Inception network (GoogleNet)

- The inception network consist of concatenated blocks of the Inception module.
- The name inception was taken from a *meme* image which was taken from **Inception movie**
- Here are the full model:



- Some times a Max-Pool block is used before the inception module to reduce the dimensions of the inputs.
- There are a 3 Sofmax branches at different positions to push the network toward its goal. and helps to ensure that the intermediate features are good enough to the network to learn and it turns out that softmax0 and sofmax1 gives regularization effect.
- Since the development of the Inception module, the authors and the others have built another versions of this network. Like inception v2, v3, and v4. Also there is a network that has used the inception module and the ResNet together.
- [Szegedy et al., 2014, Going Deeper with Convolutions]

Using Open-Source Implementation

- We have learned a lot of NNs and ConvNets architectures.
- It turns out that a lot of these NN are difficult to replicated. because there are some details that may not presented on its papers. There are some other reasons like:
  - Learning decay.
  - Parameter tuning.
- A lot of deep learning researchers are opening sourcing their code into Internet on sites like Github.

- If you see a research paper and you want to build over it, the first thing you should do is to look for an open source implementation for this paper.
- Some advantage of doing this is that you might download the network implementation along with its parameters/weights. The author might have used multiple GPUs and spent some weeks to reach this result and its right in front of you after you download it.

Transfer Learning

- If you are using a specific NN architecture that has been trained before, you can use this pretrained parameters/weights instead of random initialization to solve your problem.
- It can help you boost the performance of the NN.
- The pretrained models might have trained on a large datasets like ImageNet, Ms COCO, or pascal and took a lot of time to learn those parameters/weights with optimized hyperparameters. This can save you a lot of time.
- Lets see an example:
  - Lets say you have a cat classification problem which contains 3 classes Tigger, Misty and neither.
  - You don't have much a lot of data to train a NN on these images.
  - Andrew recommends to go online and download a good NN with its weights, remove the softmax activation layer and put your own one and make the network learn only the new layer while other layer weights are fixed/frozen.
  - Frameworks have options to make the parameters frozen in some layers using `trainable = 0` or `freeze = 0`
  - One of the tricks that can speed up your training, is to run the pretrained NN without final softmax layer and get an intermediate representation of your images and save them to disk. And then use these representation to a shallow NN network. This can save you the time needed to run an image through all the layers.
    - Its like converting your images into vectors.
- Another example:
  - What if in the last example you have a lot of pictures for your cats.
  - One thing you can do is to freeze few layers from the beginning of the pretrained network and learn the other weights in the network.
  - Some other idea is to throw away the layers that aren't frozen and put your own layers there.
- Another example:
  - If you have enough data, you can fine tune all the layers in your pretrained network but don't random initialize the parameters, leave the learned parameters as it is and learn from there.

Data Augmentation

- If data is increased, your deep NN will perform better. Data augmentation is one of the techniques that deep learning uses to increase the performance of deep NN.
- The majority of computer vision applications needs more data right now.
- Some data augmentation methods that are used for computer vision tasks includes:
  - Mirroring.
  - Random cropping.
    - The issue with this technique is that you might take a wrong crop.
    - The solution is to make your crops big enough.
  - Rotation.
  - Shearing.
  - Local warping.
  - Color shifting.
    - For example, we add to R, G, and B some distortions that will make the image identified as the same for the human but is different for the computer.
    - In practice the added value are pulled from some probability distribution and these shifts are some small.
    - Makes your algorithm more robust in changing colors in images.
    - There are an algorithm which is called ***PCA color augmentation*** that decides the shifts needed automatically.
- Implementing distortions during training:
  - You can use a different CPU thread to make you a distorted mini batches while you are training your NN.
- Data Augmentation has also some hyperparameters. A good place to start is to find an open source data augmentation implementation and then use it or fine tune these hyperparameters.

State of Computer Vision

- For a specific problem we may have a little data for it or a lots of data.
- Speech recognition problems for example has a big amount of data, while image recognition has a medium amount of data and the object detection has a small amount of data nowadays.
- If your problem has a large amount of data, researchers are tend to use:
  - Simpler algorithms.
  - Less hand engineering.

- If you don't have that much data people tend to try more hand engineering for the problem "Hacks". Like choosing a more complex NN architecture.
- Because we haven't got that much data in a lot of computer vision problems, it relies a lot on hand engineering.
- We will see in the next chapter that because the object detection has less data, a more complex NN architectures will be presented.
- Tips for doing well on benchmarks/winning competitions:
  - Ensembling.
    - Train several networks independently and average their outputs. Merging down some classifiers.
    - After you decide the best architecture for your problem, initialize some of that randomly and train them independently.
    - This can give you a push by 2%
    - But this will slow down your production by the number of the ensembles. Also it takes more memory as it saves all the models in the memory.
    - People use this in competitions but few uses this in a real production.
  - Multi-crop at test time.
    - Run classifier on multiple versions of test versions and average results.
    - There is a technique called 10 crops that uses this.
    - This can give you a better result in the production.
- Use open source code
  - Use architectures of networks published in the literature.
  - Use open source implementations if possible.
  - Use pretrained models and fine-tune on your dataset.

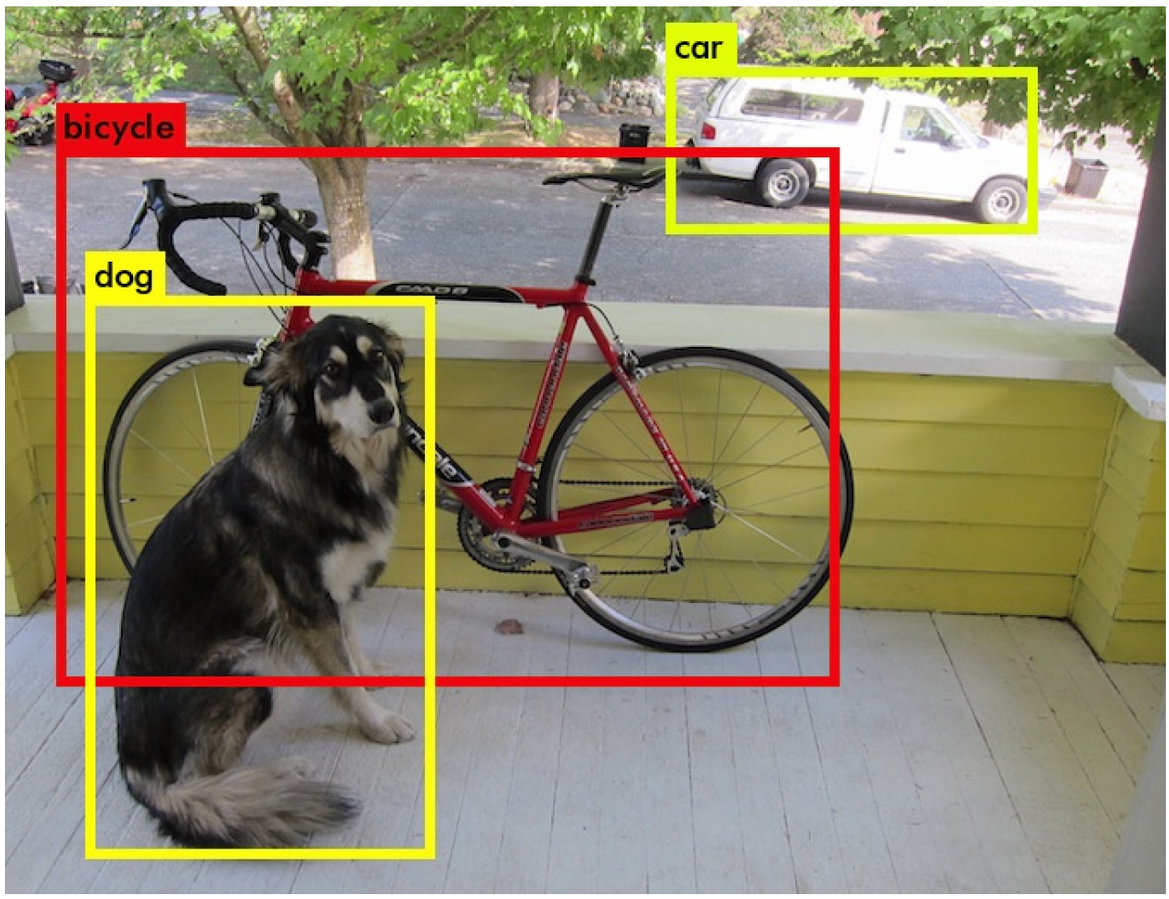
Object detection

> Learn how to apply your knowledge of CNNs to one of the toughest but hottest field of computer vision: Object detection.

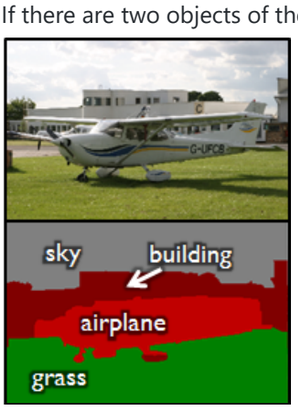
Object Localization

- Object detection is one of the areas in which deep learning is doing great in the past two years.
- What are localization and detection?
  - **Image Classification**:
    - Classify an image to a specific class. The whole image represents one class. We don't want to know exactly where are the object. Usually only one object is presented.
    - 
  - **Classification with localization**:
    - Given an image we want to learn the class of the image and where are the class location in the image. We need to detect a class and a rectangle of where that object is. Usually only one object is presented.

- **Object detection**:
  - Given an image we want to detect all the object in the image that belong to a specific classes and give their location. An image can contain more than one object with different classes.



- **Semantic Segmentation**:
  - We want to Label each pixel in the image with a category label. Semantic Segmentation Don't differentiate instances, only care about pixels. It detects no objects just pixels.
  - If there are two objects of the same class is intersected, we won't be able to separate them.



- **Instance Segmentation**
  - This is like the full problem. Rather than we want to predict the bounding box, we want to know which pixel label but also distinguish them.

- 

- To make image classification we use a Conv Net with a Softmax attached to the end of it.
- To make classification with localization we use a Conv Net with a softmax attached to the end of it and a four numbers `bx`, `by`, `bh`, and `bw` to tell you the location of the class in the image. The dataset should contain this four numbers with the class too.
- Defining the target label Y in classification with localization problem:

```
Y = [
                Pc                              # Probability of an object is presented
                bx                              # Bounding box
                by                              # Bounding box
                bh                              # Bounding box
                bw                              # Bounding box
                c1                              # The classes
                c2
                ...
]
```

  - Example (Object is present):

```
Y = [
                1               # Object is present
                0
                0
                100
                100
                0
                1
                0
]
```

  - Example (When object isn't presented):

```
Y = [
                0               # Object isn't presented
                ?               # ? means we dont care with other values
                ?
                ?
                ?
                ?
                ?
                ?
]
```

- The loss function for the Y we have created (Example of the square error):

```
L(y',y) = {
                    (y1'-y1)^2 + (y2'-y2)^2 + ...           if y1 = 1
                    (y1'-y1)^2                                          if y1 = 0
                }
```

  - In practice we use logistic regression for `pc`, log likely hood loss for classes, and squared error for the bounding box.

Landmark Detection

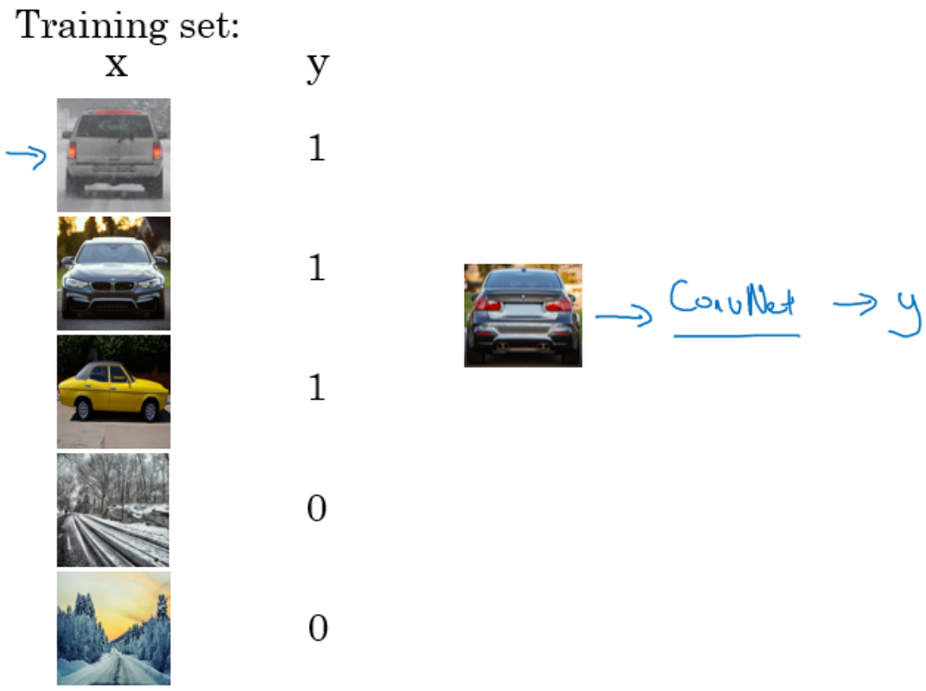
- In some of the computer vision problems you will need to output some points. That is called **landmark detection**.
- For example, if you are working in a face recognition problem you might want some points on the face like corners of the eyes, corners of the mouth, and corners of the nose and so on. This can help in a lot of application like detecting the pose of the face.
- Y shape for the face recognition problem that needs to output 64 landmarks:

```
Y = [
                THereIsAface                        # Probability of face is presented 0 or 1
                l1x,
                l1y,
                ....,
                l64x,
                l64y
]
```

- Another application is when you need to get the skeleton of the person using different landmarks/points in the person which helps in some applications.
- Hint, in your labeled data, if `l1x,l1y` is the left corner of left eye, all other `l1x,l1y` of the other examples has to be the same.

Object Detection

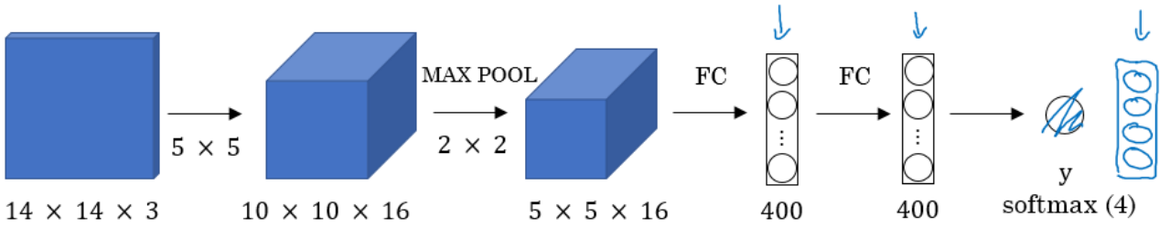
- We will use a Conv net to solve the object detection problem using a technique called the sliding windows detection algorithm.
- For example lets say we are working on Car object detection.
- The first thing, we will train a Conv net on cropped car images and non car images.

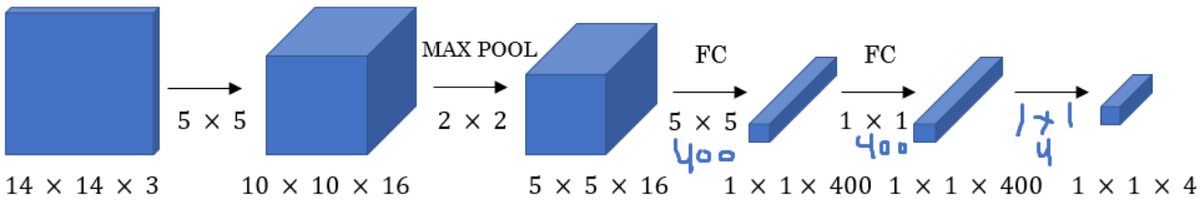


- After we finish training of this Conv net we will then use it with the sliding windows technique.
- Sliding windows detection algorithm:
  i. Decide a rectangle size.
  ii. Split your image into rectangles of the size you picked. Each region should be covered. You can use some strides.
  iii. For each rectangle feed the image into the Conv net and decide if its a car or not.
  iv. Pick larger/smaller rectangles and repeat the process from 2 to 3.
  v. Store the rectangles that contains the cars.
  vi. If two or more rectangles intersects choose the rectangle with the best accuracy.
- Disadvantage of sliding window is the computation time.
- In the era of machine learning before deep learning, people used a hand crafted linear classifiers that classifies the object and then use the sliding window technique. The linear classier make it a cheap computation. But in the deep learning era that is so computational expensive due to the complexity of the deep learning model.
- To solve this problem, we can implement the sliding windows with a ***Convolutional approach***.
- One other idea is to compress your deep learning model.

Convolutional Implementation of Sliding Windows

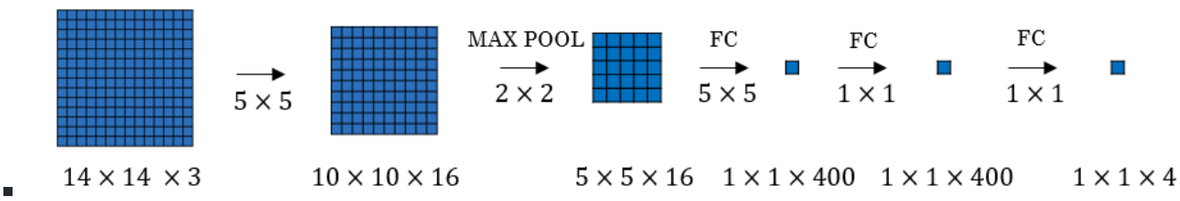
- Turning FC layer into convolutional layers (predict image class from four classes):



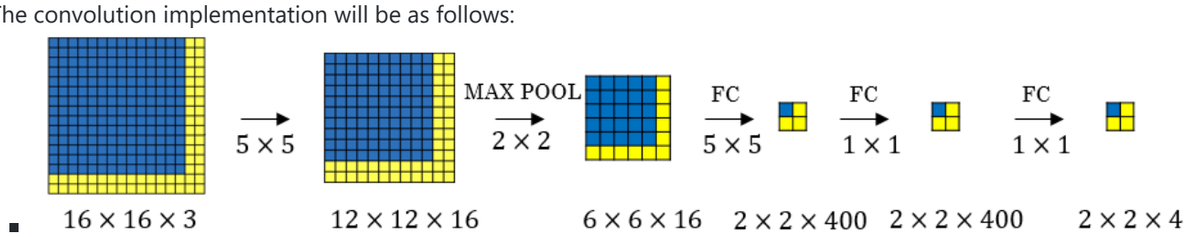


- As you can see in the above image, we turned the FC layer into a Conv layer using a convolution with the width and height of the filter is the same as the width and height of the input.

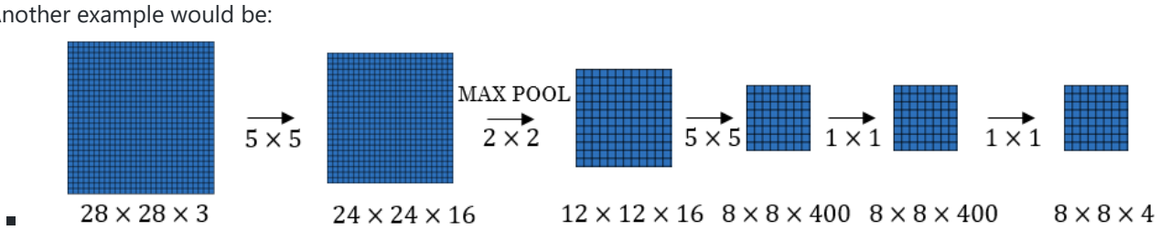
- **Convolution implementation of sliding windows**:
  - First lets consider that the Conv net you trained is like this (No FC all is conv layers):



  - Say now we have a 16 x 16 x 3 image that we need to apply the sliding windows in. By the normal implementation that have been mentioned in the section before this, we would run this Conv net four times each rectangle size will be 16 x 16.
  - The convolution implementation will be as follows:

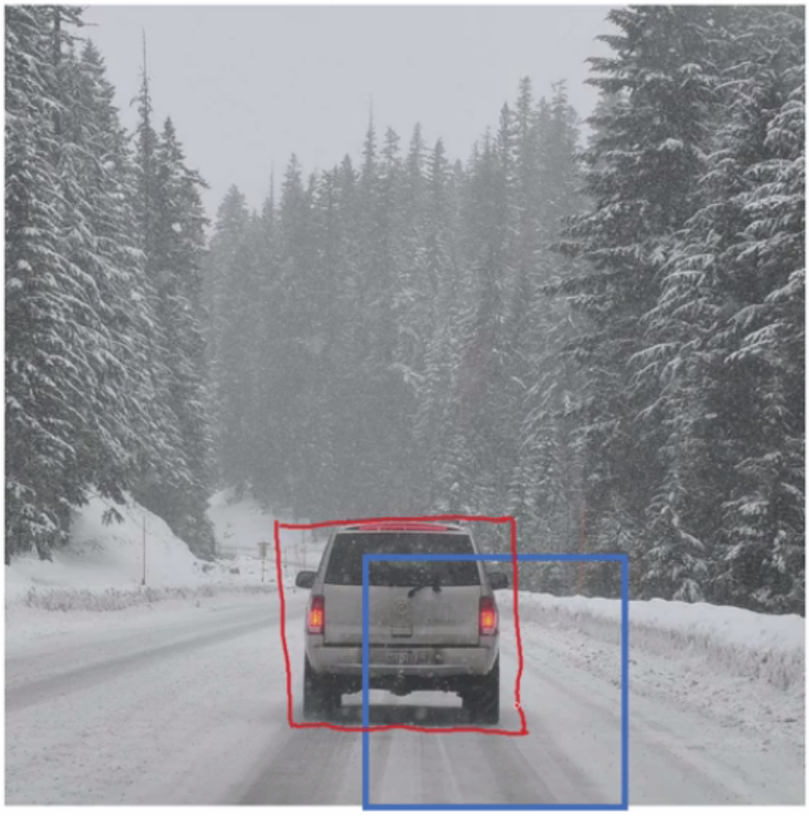


  - Simply we have feed the image into the same Conv net we have trained.
  - The left cell of the result "The blue one" will represent the the first sliding window of the normal implementation. The other cells will represent the others.
  - Its more efficient because it now shares the computations of the four times needed.
  - Another example would be:



  - This example has a total of 16 sliding windows that shares the computation together.
  - [Sermanet et al., 2014, OverFeat: Integrated recognition, localization and detection using convolutional networks]

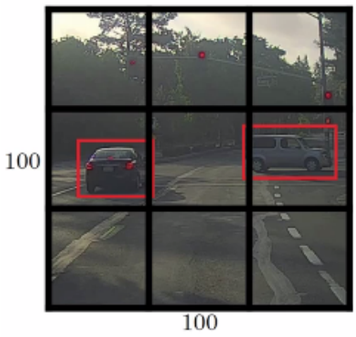
- The weakness of the algorithm is that the position of the rectangle wont be so accurate. Maybe none of the rectangles is exactly on the object you want to recognize.



- In red, the rectangle we want and in blue is the required car rectangle.

Bounding Box Predictions

- A better algorithm than the one described in the last section is the YOLO algorithm.
- YOLO stands for *you only look once* and was developed back in 2015.
- Yolo Algorithm:

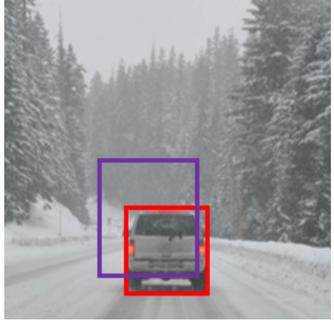


  i. Lets say we have an image of 100 X 100
  ii. Place a 3 x 3 grid on the image. For more smother results you should use 19 x 19 for the 100 x 100
  iii. Apply the classification and localization algorithm we discussed in a previous section to each section of the grid. `bx` and `by` will represent the center point of the object in each grid and will be relative to the box so the range is between 0 and 1 while `bh` and `bw` will represent the height and width of the object which can be greater than 1.0 but still a floating point value.
  iv. Do everything at once with the convolution sliding window. If Y shape is 1 x 8 as we discussed before then the output of the 100 x 100 image should be 3 x 3 x 8 which corresponds to 9 cell results.
  v. Merging the results using predicted localization mid point.

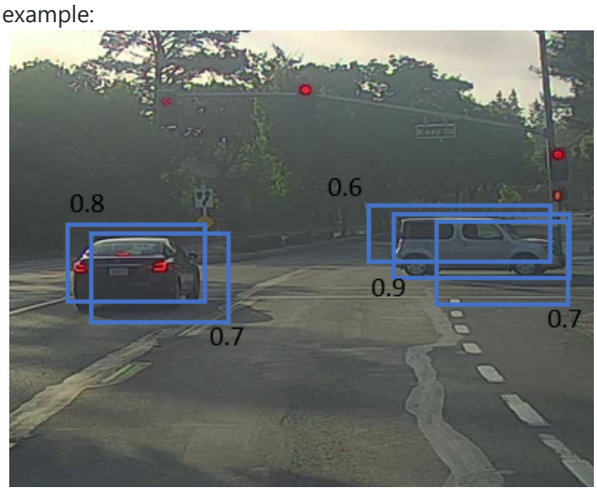
- We have a problem if we have found more than one object in one grid box.
- One of the best advantages that makes the YOLO algorithm popular is that it has a great speed and a Conv net implementation.
- How is YOLO different from other Object detectors? YOLO uses a single CNN network for both classification and localizing the object using bounding boxes.
- In the next sections we will see some ideas that can make the YOLO algorithm better.

Intersection Over Union

- Intersection Over Union is a function used to evaluate the object detection algorithm.
- It computes size of intersection and divide it by the union. More generally, *IoU is a measure of the overlap between two bounding boxes*.
- For example:

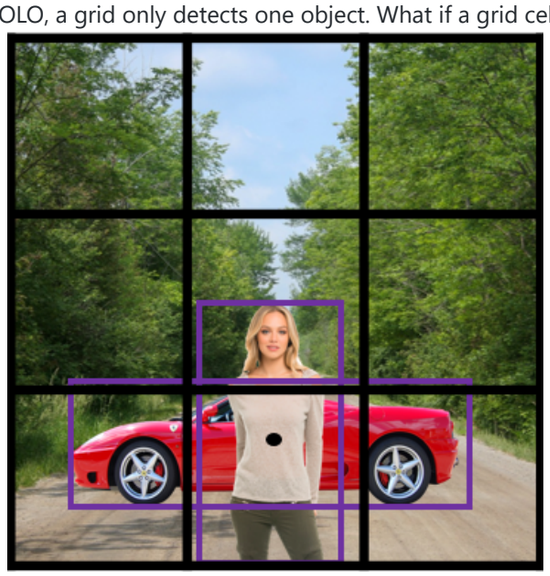
- 
  - The red is the labeled output and the purple is the predicted output.
  - To compute Intersection Over Union we first compute the union area of the two rectangles which is "the first rectangle + second rectangle" Then compute the intersection area between these two rectangles.
  - Finally `IOU = intersection area / Union area`
- If `IOU >=0.5` then its good. The best answer will be 1.
- The higher the IOU the better is the accuracy.

Non-max Suppression

- One of the problems we have addressed in YOLO is that it can detect an object multiple times.
- Non-max Suppression is a way to make sure that YOLO detects the object just once.
- For example:
  - 

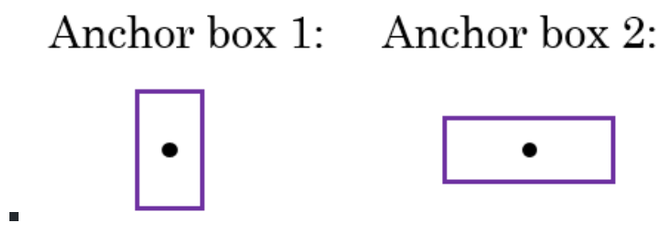
  - Each car has two or more detections with different probabilities. This came from some of the grids that thinks that this is the center point of the object.
- Non-max suppression algorithm:
  i. Lets assume that we are targeting one class as an output class.
  ii. Y shape should be `[Pc, bx, by, bh, hw]` Where Pc is the probability if that object occurs.
  iii. Discard all boxes with `Pc < 0.6`
  iv. While there are any remaining boxes:
    a. Pick the box with the largest Pc Output that as a prediction.
    b. Discard any remaining box with `IoU > 0.5` with that box output in the previous step i.e any box with high overlap(greater than overlap threshold of 0.5).
- If there are multiple classes/object types `c` you want to detect, you should run the Non-max suppression `c` times, once for every output class.

Anchor Boxes

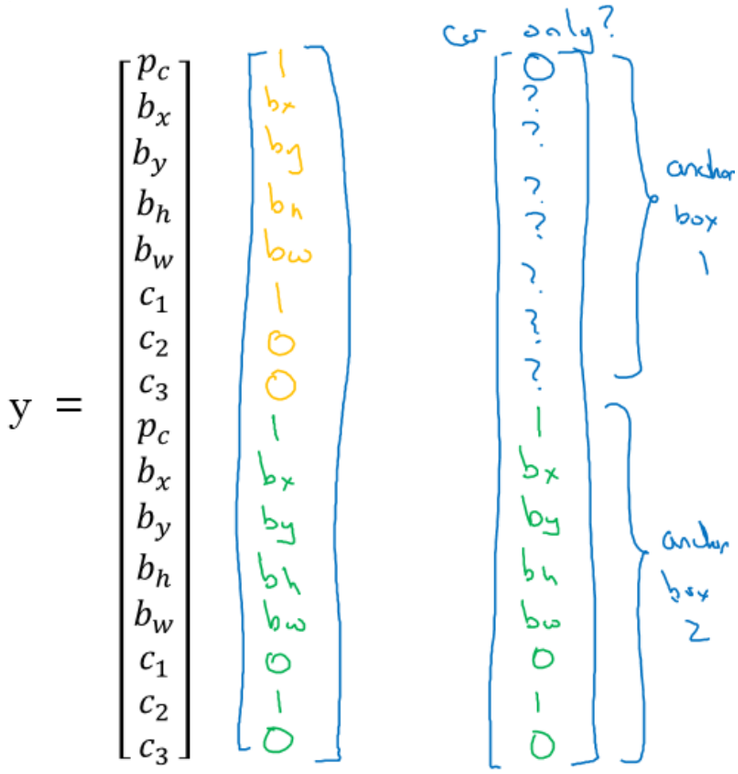
- In YOLO, a grid only detects one object. What if a grid cell wants to detect multiple object?
  - 
  - Car and person grid is same here.

- In practice this happens rarely.

- The idea of Anchor boxes helps us solving this issue.
- If Y = `[Pc, bx, by, bh, bw, c1, c2, c3]` Then to use two anchor boxes like this:
  - Y = `[Pc, bx, by, bh, bw, c1, c2, c3, Pc, bx, by, bh, bw, c1, c2, c3]` We simply have repeated the one anchor Y.
  - The two anchor boxes you choose should be known as a shape:



- So Previously, each object in training image is assigned to grid cell that contains that object's midpoint.
- With two anchor boxes, Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU. You have to check where your object should be based on its rectangle closest to which anchor box.
- Example of data:

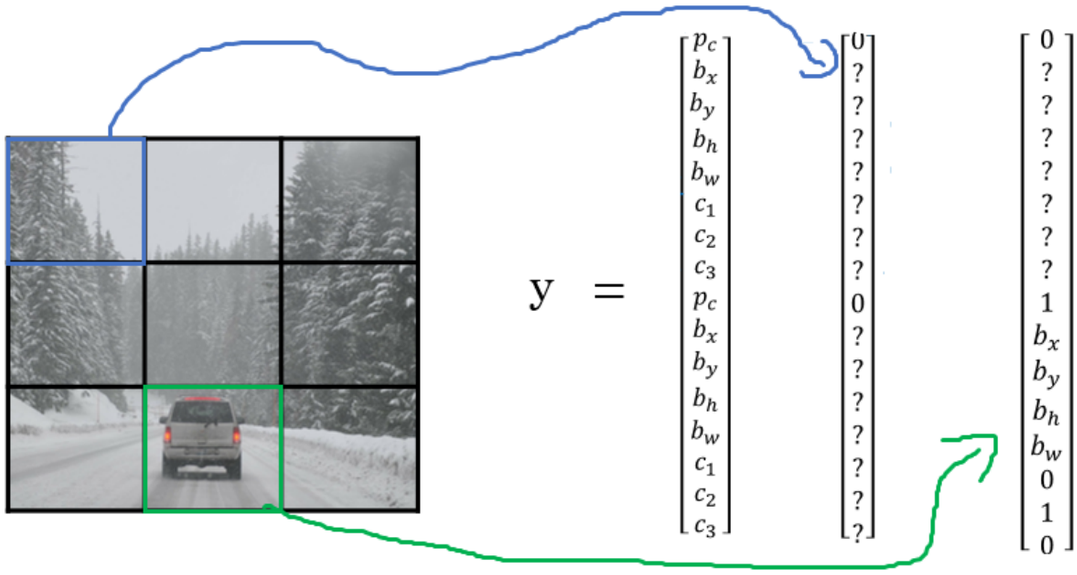


  - Where the car was near the anchor 2 than anchor 1.
- You may have two or more anchor boxes but you should know their shapes.
  - how do you choose the anchor boxes and people used to just choose them by hand. Maybe five or ten anchor box shapes that spans a variety of shapes that cover the types of objects you seem to detect frequently.
  - You may also use a k-means algorithm on your dataset to specify that.
- Anchor boxes allows your algorithm to specialize, means in our case to easily detect wider images or taller ones.

YOLO Algorithm

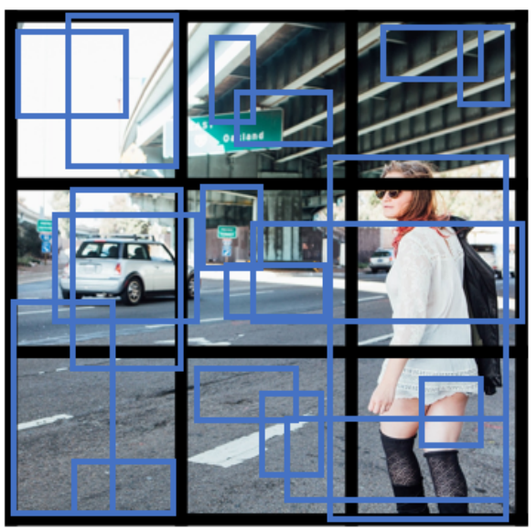
- YOLO is a state-of-the-art object detection model that is fast and accurate
- Lets sum up and introduce the whole YOLO algorithm given an example.
- Suppose we need to do object detection for our autonomous driver system.It needs to identify three classes:
  i. Pedestrian (Walks on ground).
  ii. Car.
  iii. Motorcycle.
- We decided to choose two anchor boxes, a taller one and a wide one.
  - Like we said in practice they use five or more anchor boxes hand made or generated using k-means.
- Our labeled Y shape will be `[Ny, HeightOfGrid, WidthOfGrid, 16]`, where Ny is number of instances and each row (of size 16) is as follows:
  - `[Pc, bx, by, bh, bw, c1, c2, c3, Pc, bx, by, bh, bw, c1, c2, c3]`
- Your dataset could be an image with a multiple labels and a rectangle for each label, we should go to your dataset and make the shape and values of Y like we agreed.

- An example:



  - We first initialize all of them to zeros and ?, then for each label and rectangle choose its closest grid point then the shape to fill it and then the best anchor point based on the IOU. so that the shape of Y for one image should be `[HeightOfGrid, WidthOfGrid,16]`

- Train the labeled images on a Conv net. you should receive an output of `[HeightOfGrid, WidthOfGrid,16]` for our case.
- To make predictions, run the Conv net on an image and run Non-max suppression algorithm for each class you have in our case there are 3 classes.
  - You could get something like that:



    - Total number of generated boxes are grid_width * grid_height * no_of_anchors = 3 x 3 x 2
  - By removing the low probability predictions you should have:

  - Then get the best probability followed by the IOU filtering:

- YOLO are not good at detecting smaller object.
- YOLO9000 Better, faster, stronger
  - Summary:

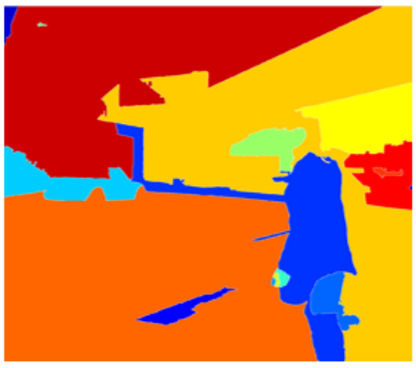
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 608, 608, 3)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 608, 608, 32)      864
_________________________________________________________________
batch_normalization_1 (BatchNorm (None, 608, 608, 32)      128
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 608, 608, 32)      0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2D)   (None, 304, 304, 32)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 304, 304, 64)      18432
_________________________________________________________________
batch_normalization_2 (BatchNorm (None, 304, 304, 64)      256
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 304, 304, 64)      0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2D)   (None, 152, 152, 64)      0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 152, 152, 128)     73728
_________________________________________________________________
batch_normalization_3 (BatchNorm (None, 152, 152, 128)     512
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 152, 152, 128)     0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 152, 152, 64)      8192
_________________________________________________________________
batch_normalization_4 (BatchNorm (None, 152, 152, 64)      256
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 152, 152, 64)      0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 152, 152, 128)     73728
_________________________________________________________________
batch_normalization_5 (BatchNorm (None, 152, 152, 128)     512
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 152, 152, 128)     0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2D)   (None, 76, 76, 128)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 76, 76, 256)       294912
_________________________________________________________________
batch_normalization_6 (BatchNorm (None, 76, 76, 256)       1024
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 76, 76, 256)       0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 76, 76, 128)       32768
_________________________________________________________________
batch_normalization_7 (BatchNorm (None, 76, 76, 128)       512
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 76, 76, 128)       0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 76, 76, 256)       294912
_________________________________________________________________
batch_normalization_8 (BatchNorm (None, 76, 76, 256)       1024
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, 76, 76, 256)       0
_________________________________________________________________
```

```
max_pooling2d_4 (MaxPooling2D)   (None, 38, 38, 256)   0           leaky_re_lu_8[0][0]
__________________________________________________________________________________________________
conv2d_9 (Conv2D)                (None, 38, 38, 512)   1179648     max_pooling2d_4[0][0]
__________________________________________________________________________________________________
batch_normalization_9 (BatchNorm (None, 38, 38, 512)   2048        conv2d_9[0][0]
__________________________________________________________________________________________________
leaky_re_lu_9 (LeakyReLU)        (None, 38, 38, 512)   0     batch_normalization_9[0][0]
__________________________________________________________________________________________________
conv2d_10 (Conv2D)               (None, 38, 38, 256)   131072      leaky_re_lu_9[0][0]
__________________________________________________________________________________________________
batch_normalization_10 (BatchNor (None, 38, 38, 256)   1024        conv2d_10[0][0]
__________________________________________________________________________________________________
leaky_re_lu_10 (LeakyReLU)       (None, 38, 38, 256)   0    batch_normalization_10[0][0]
__________________________________________________________________________________________________
conv2d_11 (Conv2D)               (None, 38, 38, 512)   1179648    leaky_re_lu_10[0][0]
__________________________________________________________________________________________________
batch_normalization_11 (BatchNor (None, 38, 38, 512)   2048        conv2d_11[0][0]
__________________________________________________________________________________________________
leaky_re_lu_11 (LeakyReLU)       (None, 38, 38, 512)   0    batch_normalization_11[0][0]
__________________________________________________________________________________________________
conv2d_12 (Conv2D)               (None, 38, 38, 256)   131072      leaky_re_lu_11[0][0]
__________________________________________________________________________________________________
batch_normalization_12 (BatchNor (None, 38, 38, 256)   1024        conv2d_12[0][0]
__________________________________________________________________________________________________
leaky_re_lu_12 (LeakyReLU)       (None, 38, 38, 256)   0   batch_normalization_12[0][0]
__________________________________________________________________________________________________
conv2d_13 (Conv2D)               (None, 38, 38, 512)   1179648     leaky_re_lu_12[0][0]
__________________________________________________________________________________________________
batch_normalization_13 (BatchNor (None, 38, 38, 512)   2048        conv2d_13[0][0]
__________________________________________________________________________________________________
leaky_re_lu_13 (LeakyReLU)       (None, 38, 38, 512)   0    batch_normalization_13[0][0]
__________________________________________________________________________________________________
max_pooling2d_5 (MaxPooling2D)   (None, 19, 19, 512)   0           leaky_re_lu_13[0][0]
__________________________________________________________________________________________________
conv2d_14 (Conv2D)               (None, 19, 19, 1024)  4718592     max_pooling2d_5[0][0]
__________________________________________________________________________________________________
batch_normalization_14 (BatchNor (None, 19, 19, 1024)  4096        conv2d_14[0][0]
__________________________________________________________________________________________________
leaky_re_lu_14 (LeakyReLU)       (None, 19, 19, 1024)  0    batch_normalization_14[0][0]
__________________________________________________________________________________________________
conv2d_15 (Conv2D)               (None, 19, 19, 512)   524288      leaky_re_lu_14[0][0]
__________________________________________________________________________________________________
batch_normalization_15 (BatchNor (None, 19, 19, 512)   2048        conv2d_15[0][0]
__________________________________________________________________________________________________
leaky_re_lu_15 (LeakyReLU)       (None, 19, 19, 512)   0    batch_normalization_15[0][0]
__________________________________________________________________________________________________
conv2d_16 (Conv2D)               (None, 19, 19, 1024)  4718592     leaky_re_lu_15[0][0]
__________________________________________________________________________________________________
batch_normalization_16 (BatchNor (None, 19, 19, 1024)  4096        conv2d_16[0][0]
__________________________________________________________________________________________________
leaky_re_lu_16 (LeakyReLU)       (None, 19, 19, 1024)  0    batch_normalization_16[0][0]
__________________________________________________________________________________________________
conv2d_17 (Conv2D)               (None, 19, 19, 512)   524288      leaky_re_lu_16[0][0]
__________________________________________________________________________________________________
batch_normalization_17 (BatchNor (None, 19, 19, 512)   2048        conv2d_17[0][0]
__________________________________________________________________________________________________
leaky_re_lu_17 (LeakyReLU)       (None, 19, 19, 512)   0    batch_normalization_17[0][0]
__________________________________________________________________________________________________
conv2d_18 (Conv2D)               (None, 19, 19, 1024)  4718592     leaky_re_lu_17[0][0]
__________________________________________________________________________________________________
batch_normalization_18 (BatchNor (None, 19, 19, 1024)  4096        conv2d_18[0][0]
__________________________________________________________________________________________________
leaky_re_lu_18 (LeakyReLU)       (None, 19, 19, 1024)  0    batch_normalization_18[0][0]
__________________________________________________________________________________________________
conv2d_19 (Conv2D)               (None, 19, 19, 1024)  9437184     leaky_re_lu_18[0][0]
__________________________________________________________________________________________________
batch_normalization_19 (BatchNor (None, 19, 19, 1024)  4096        conv2d_19[0][0]
__________________________________________________________________________________________________
conv2d_21 (Conv2D)               (None, 38, 38, 64)    32768       leaky_re_lu_13[0][0]
__________________________________________________________________________________________________
leaky_re_lu_19 (LeakyReLU)       (None, 19, 19, 1024)  0    batch_normalization_19[0][0]
__________________________________________________________________________________________________
batch_normalization_21 (BatchNor (None, 38, 38, 64)    256         conv2d_21[0][0]
__________________________________________________________________________________________________
conv2d_20 (Conv2D)               (None, 19, 19, 1024)  9437184     leaky_re_lu_19[0][0]
__________________________________________________________________________________________________
leaky_re_lu_21 (LeakyReLU)       (None, 38, 38, 64)    0    batch_normalization_21[0][0]
__________________________________________________________________________________________________
batch_normalization_20 (BatchNor (None, 19, 19, 1024)  4096        conv2d_20[0][0]
__________________________________________________________________________________________________
space_to_depth_x2 (Lambda)       (None, 19, 19, 256)   0           leaky_re_lu_21[0][0]
__________________________________________________________________________________________________
leaky_re_lu_20 (LeakyReLU)       (None, 19, 19, 1024)  0    batch_normalization_20[0][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)      (None, 19, 19, 1280)  0         space_to_depth_x2[0][0]
                                                                  leaky_re_lu_20[0][0]
__________________________________________________________________________________________________
```

```
conv2d_22 (Conv2D)               (None, 19, 19, 1024)  11796480    concatenate_1[0][0]
__________________________________________________________________________________________________
batch_normalization_22 (BatchNor (None, 19, 19, 1024)  4096        conv2d_22[0][0]
__________________________________________________________________________________________________
leaky_re_lu_22 (LeakyReLU)       (None, 19, 19, 1024)  0    batch_normalization_22[0][0]
__________________________________________________________________________________________________
conv2d_23 (Conv2D)               (None, 19, 19, 425)   435625      leaky_re_lu_22[0][0]
==================================================================================================
Total params: 50,983,561
Trainable params: 50,962,889
Non-trainable params: 20,672
__________________________________________________________________________________________________
```

- You can find implementations for YOLO here:
  - https://github.com/allanzelener/YAD2K
  - https://github.com/thtrieu/darkflow
  - https://pjreddie.com/darknet/yolo/

Region Proposals (R-CNN)

- R-CNN is an algorithm that also makes an object detection.
- Yolo tells that its faster:
  - > Our model has several advantages over classifier-based systems. It looks at the whole image at test time so its predictions are informed by global context in the image. It also makes predictions with a single network evaluation unlike systems like R-CNN which require thousands for a single image. This makes it extremely fast, more than 1000x faster than R-CNN and 100x faster than Fast R-CNN. See our paper for more details on the full system.
- But one of the downsides of YOLO that it process a lot of areas where no objects are present.
- **R-CNN** stands for regions with Conv Nets.
- R-CNN tries to pick a few windows and run a Conv net (your confident classifier) on top of them.
- The algorithm R-CNN uses to pick windows is called a segmentation algorithm. Outputs something like this:
  - 
- If for example the segmentation algorithm produces 2000 blob then we should run our classifier/CNN on top of these blobs.
- There has been a lot of work regarding R-CNN tries to make it faster:
  - R-CNN:
    - Propose regions. Classify proposed regions one at a time. Output label + bounding box.
    - Downside is that its slow.
    - [Girshik et. al, 2013. Rich feature hierarchies for accurate object detection and semantic segmentation]
  - Fast R-CNN:
    - Propose regions. Use convolution implementation of sliding windows to classify all the proposed regions.
    - [Girshik, 2015. Fast R-CNN]
  - Faster R-CNN:
    - Use convolutional network to propose regions.
    - [Ren et. al, 2016. Faster R-CNN: Towards real-time object detection with region proposal networks]
  - Mask R-CNN:
    - https://arxiv.org/abs/1703.06870
- Most of the implementation of faster R-CNN are still slower than YOLO.
- Andrew Ng thinks that the idea behind YOLO is better than R-CNN because you are able to do all the things in just one time instead of two times.
- Other algorithms that uses one shot to get the output includes **SSD** and **MultiBox**.

- [Wei Liu, et. al 2015 SSD: Single Shot MultiBox Detector]

- **R-FCN** is similar to Faster R-CNN but more efficient.
  - [Jifeng Dai, et. al 2016 R-FCN: Object Detection via Region-based Fully Convolutional Networks]

Special applications: Face recognition & Neural style transfer

> Discover how CNNs can be applied to multiple fields, including art generation and face recognition. Implement your own algorithm to generate art and recognize faces!

Face Recognition

What is face recognition?

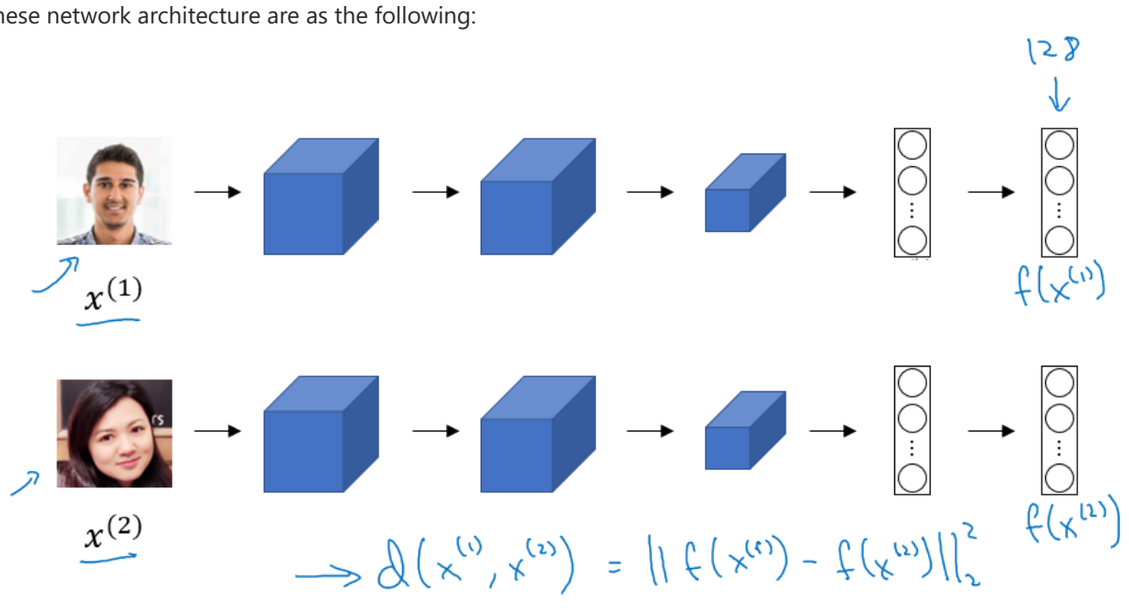
- Face recognition system identifies a person's face. It can work on both images or videos.
- **Liveness detection** within a video face recognition system prevents the network from identifying a face in an image. It can be learned by supervised deep learning using a dataset for live human and in-live human and sequence learning.
- Face verification vs. face recognition:
  - Verification:
    - Input: image, name/ID. (1 : 1)
    - Output: whether the input image is that of the claimed person.
    - "is this the claimed person?"
  - Recognition:
    - Has a database of K persons
    - Get an input image
    - Output ID if the image is any of the K persons (or not recognized)
    - "who is this person?"
- We can use a face verification system to make a face recognition system. The accuracy of the verification system has to be high (around 99.9% or more) to be use accurately within a recognition system because the recognition system accuracy will be less than the verification system given K persons.

One Shot Learning

- One of the face recognition challenges is to solve one shot learning problem.
- One Shot Learning: A recognition system is able to recognize a person, learning from one image.
- Historically deep learning doesn't work well with a small number of data.
- Instead to make this work, we will learn a **similarity function**:
  - d(**img1**, **img2**) = degree of difference between images.
  - We want d result to be low in case of the same faces.
  - We use tau T as a threshold for d:
    - If d(**img1**, **img2**) <= T Then the faces are the same.
- Similarity function helps us solving the one shot learning. Also its robust to new inputs.

Siamese Network

- We will implement the similarity function using a type of NNs called Siamease Network in which we can pass multiple inputs to the two or more networks with the same architecture and parameters.
- Siamese network architecture are as the following:



  - We make 2 identical conv nets which encodes an input image into a vector. In the above image the vector shape is (128,)

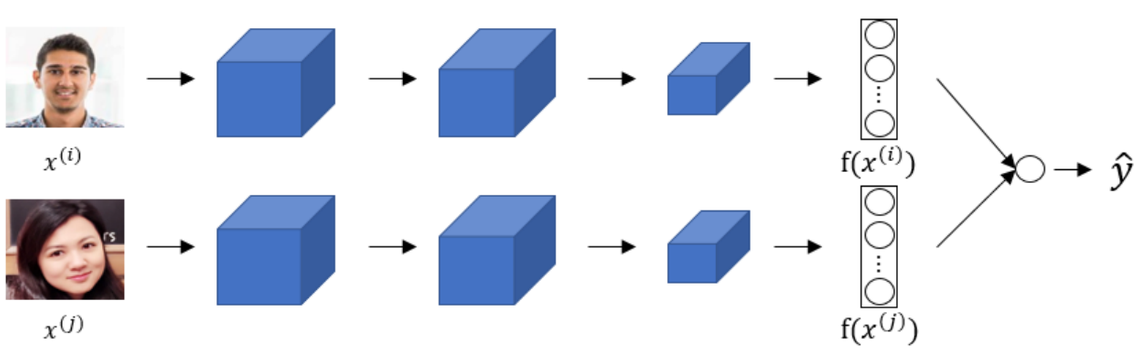
- The loss function will be `d(x1, x2) = || f(x1) - f(x2) ||^2`
- If `X1` , `X2` are the same person, we want d to be low. If they are different persons, we want d to be high.
- [Taigman et. al., 2014. DeepFace closing the gap to human level performance]

Triplet Loss

- Triplet Loss is one of the loss functions we can use to solve the similarity distance in a Siamese network.
- Our learning objective in the triplet loss function is to get the distance between an **Anchor** image and a **positive** or a **negative** image.
  - Positive means same person, while negative means different person.
- The triplet name came from that we are comparing an anchor A with a positive P and a negative N image.
- Formally we want:
  - Positive distance to be less than negative distance
  - `||f(A) - f(P)||^2 <= ||f(A) - f(N)||^2`
  - Then
  - `||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 <= 0`
  - To make sure the NN won't get an output of zeros easily:
  - `||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 <= -alpha`
    - Alpha is a small number. Sometimes its called the margin.
  - Then
  - `||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + alpha <= 0`
- Final Loss function:
  - Given 3 images (A, P, N)
  - `L(A, P, N) = max (||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + alpha , 0)`
  - `J = Sum(L(A[i], P[i], N[i]) , i)` for all triplets of images.
- You need multiple images of the same person in your dataset. Then get some triplets out of your dataset. Dataset should be big enough.
- Choosing the triplets A, P, N:
  - During training if A, P, N are chosen randomly (Subjet to A and P are the same and A and N aren't the same) then one of the problems this constrain is easily satisfied
    - `d(A, P) + alpha <= d (A, N)`
    - So the NN wont learn much
  - What we want to do is choose triplets that are **hard** to train on.
    - So for all the triplets we want this to be satisfied:
    - `d(A, P) + alpha <= d (A, N)`
    - This can be achieved by for example same poses!
    - Find more at the paper.
- Details are in this paper [Schroff et al.,2015, FaceNet: A unified embedding for face recognition and clustering]
- Commercial recognition systems are trained on a large datasets like 10/100 million images.
- There are a lot of pretrained models and parameters online for face recognition.

Face Verification and Binary Classification

- Triplet loss is one way to learn the parameters of a conv net for face recognition there's another way to learn these parameters as a straight binary classification problem.
- Learning the similarity function another way:



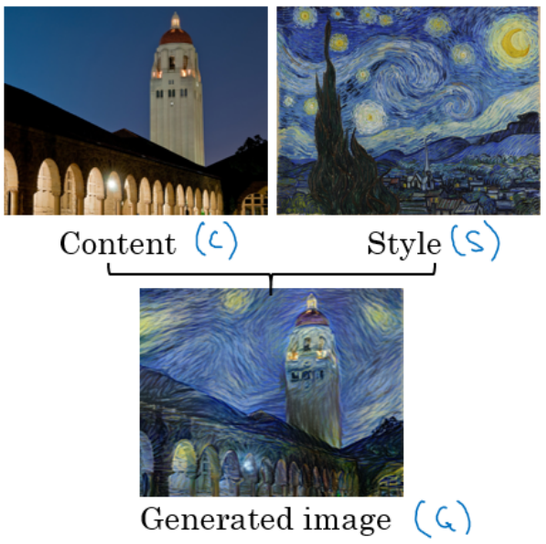
  - The final layer is a sigmoid layer.
  - `Y' = wi * Sigmoid ( f(x(i)) - f(x(j)) ) + b` where the subtraction is the Manhattan distance between f(x(i)) and f(x(j))
  - Some other similarities can be Euclidean and Ki square similarity.
  - The NN here is Siamese means the top and bottom convs has the same parameters.
- The paper for this work: [Taigman et. al., 2014. DeepFace closing the gap to human level performance]
- A good performance/deployment trick:
  - Pre-compute all the images that you are using as a comparison to the vector f(x(j))

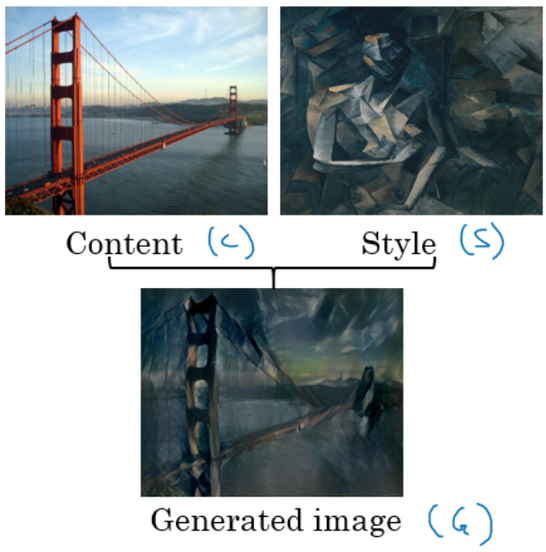
    - When a new image that needs to be compared, get its vector f(x(i)) then put it with all the pre computed vectors and pass it to the sigmoid function.
- This version works quite as well as the triplet loss function.
- Available implementations for face recognition using deep learning includes:
    - Openface
    - FaceNet
    - DeepFace

Neural Style Transfer

What is neural style transfer?

- Neural style transfer is one of the application of Conv nets.
- Neural style transfer takes a content image `C` and a style image `S` and generates the content image `G` with the style of style image.

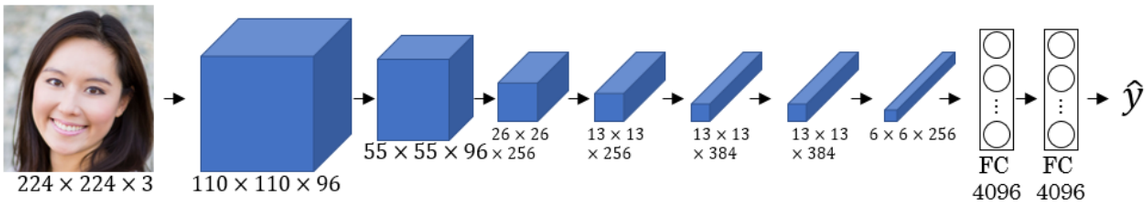


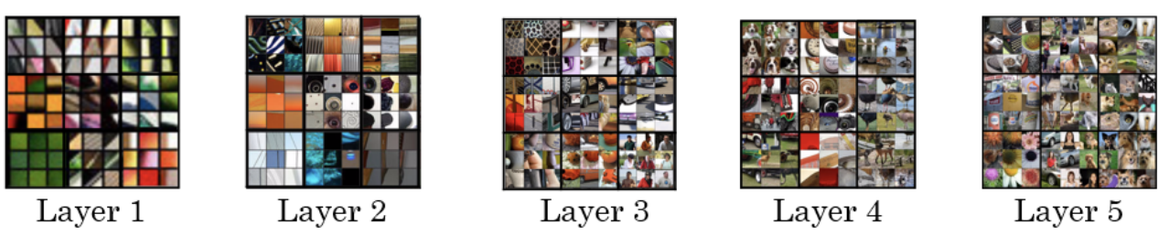


- In order to implement this you need to look at the features extracted by the Conv net at the shallower and deeper layers.
- It uses a previously trained convolutional network like VGG, and builds on top of that. The idea of using a network trained on a different task and applying it to a new task is called transfer learning.

What are deep ConvNets learning?

- Visualizing what a deep network is learning:
    - Given this AlexNet like Conv net:



    - Pick a unit in layer l. Find the nine image patches that maximize the unit's activation.
        - Notice that a hidden unit in layer one will see relatively small portion of NN, so if you plotted it it will match a small image in the shallower layers while it will get larger image in deeper layers.
    - Repeat for other units and layers.
    - It turns out that layer 1 are learning the low level representations like colors and edges.
- You will find out that each layer are learning more complex representations.
    - 

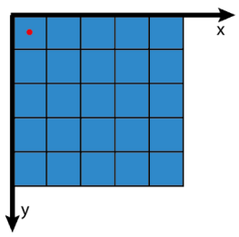
- The first layer was created using the weights of the first layer. Other images are generated using the receptive field in the image that triggered the neuron to be max.
- [Zeiler and Fergus., 2013, Visualizing and understanding convolutional networks]
- A good explanation on how to get **receptive field** given a layer:

n : number of features
r : receptive field size
j : jump (distance between two consecutive features)
$start$: center coordinate of the first feature

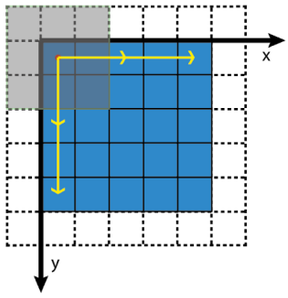
k : convolution kernel size
p : convolution padding size
s : convolution stride size

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$
$$j_{out} = j_{in} * s$$
$$r_{out} = r_{in} + (k-1) * j_{in}$$
$$start_{out} = start_{in} + \left(\frac{k-1}{2} - p\right) * j_{in}$$

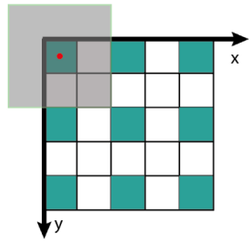
Layer 0: $n_0 = 5;\ r_0 = 1;\ j_0 = 1;\ start_0 = 0.5$



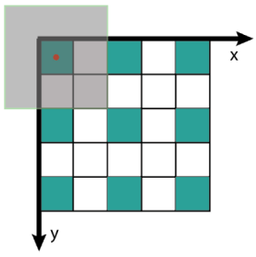
Conv1: $k_1 = 3;\ p_1 = 1;\ s_1 = 2$



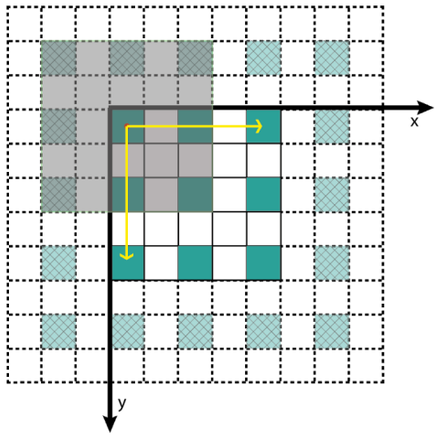
Layer 1: $n_1 = 3;\ r_1 = 3;\ j_1 = 2;\ start_1 = 0.5$



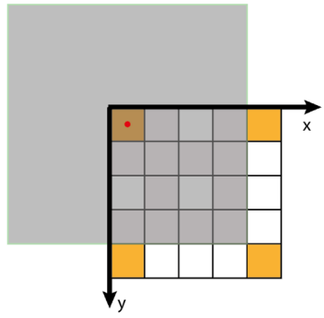
Layer 1: $n_1 = 3;\ r_1 = 3;\ j_1 = 2;\ start_1 = 0.5$



Conv2: $k_2 = 3;\ p_2 = 1;\ s_2 = 2$



Layer 2: $n_3 = 2;\ r_3 = 7;\ j_3 = 4;\ start_1 = 0.5$



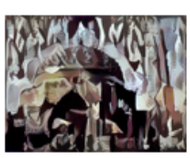
- From A guide to receptive field arithmetic for Convolutional Neural Networks

Cost Function

- We will define a cost function for the generated image that measures how good it is.
- Give a content image C, a style image S, and a generated image G:
  - `J(G) = alpha * J(C,G) + beta * J(S,G)`
  - `J(C, G)` measures how similar is the generated image to the Content image.
  - `J(S, G)` measures how similar is the generated image to the Style image.
  - alpha and beta are relative weighting to the similarity and these are hyperparameters.
- Find the generated image G:
  i. Initiate G randomly
     - For example G: 100 X 100 X 3
  ii. Use gradient descent to minimize `J(G)`
     - `G = G - dG` We compute the gradient image and use gradient decent to minimize the cost function.
- The iterations might be as following image:
  - To Generate this:


  - You will go through this:



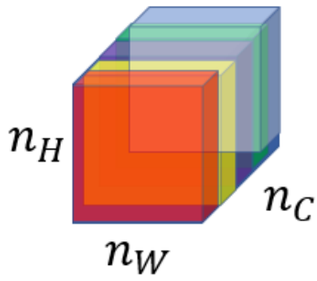
Content Cost Function

- In the previous section we showed that we need a cost function for the content image and the style image to measure how similar is them to each other.
- Say you use hidden layer `l` to compute content cost.
  - If we choose `l` to be small (like layer 1), we will force the network to get similar output to the original content image.
  - In practice `l` is not too shallow and not too deep but in the middle.
- Use pre-trained ConvNet. (E.g., VGG network)

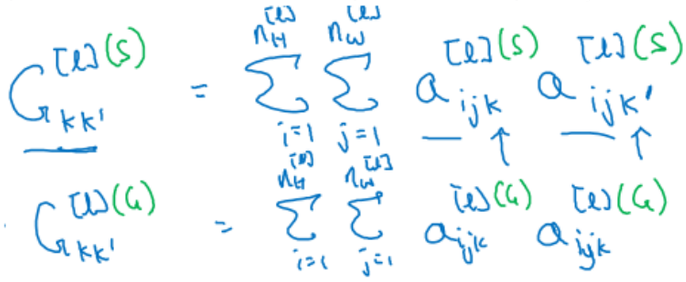
- Let `a(c)[l]` and `a(G)[l]` be the activation of layer `l` on the images.
- If `a(c)[l]` and `a(G)[l]` are similar then they will have the same content
  - `J(C, G) at a layer l = 1/2 || a(c)[l] - a(G)[l] ||^2`

Style Cost Function

- Meaning of the ***style*** of an image:
  - Say you are using layer l's activation to measure ***style***.
  - Define style as correlation between **activations** across **channels**.
    - That means given an activation like this:

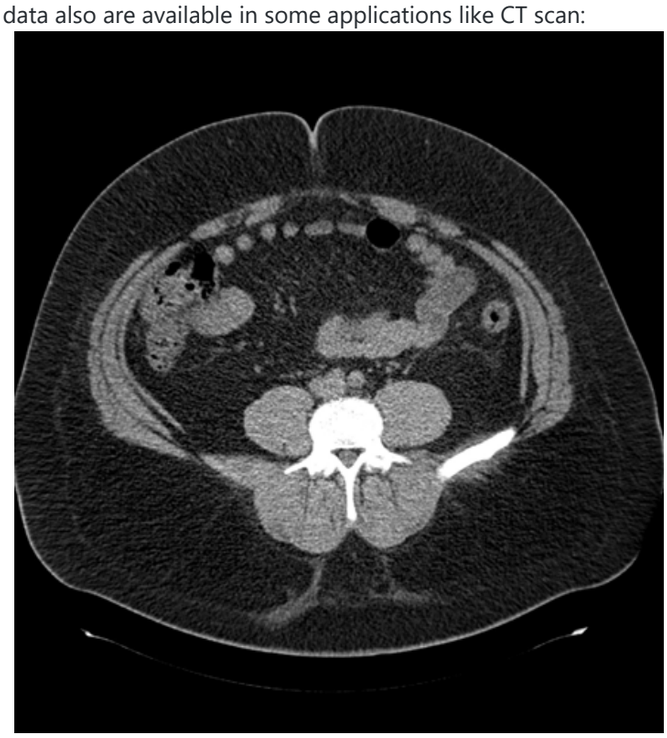


    - How correlate is the orange channel with the yellow channel?
    - Correlated means if a value appeared in a specific channel a specific value will appear too (Depends on each other).
    - Uncorrelated means if a value appeared in a specific channel doesn't mean that another value will appear (Not depend on each other)
  - The correlation tells you how a components might occur or not occur together in the same image.
- The correlation of style image channels should appear in the generated image channels.
- Style matrix (Gram matrix):
  - Let `a(l)[i, j, k]` be the activation at l with `(i=H, j=W, k=C)`
  - Also `G(l)(s)` is matrix of shape `nc(l) x nc(l)`
    - We call this matrix style matrix or Gram matrix.
    - In this matrix each cell will tell us how correlated is a channel to another channel.
  - To populate the matrix we use these equations to compute style matrix of the style image and the generated image.



    - As it appears its the sum of the multiplication of each member in the matrix.
- To compute gram matrix efficiently:
  - Reshape activation from H X W X C to HW X C
  - Name the reshaped activation F.
  - `G[l] = F * F.T`
- Finally the cost function will be as following:
  - `J(S, G) at layer l = (1/ 2 * H * W * C) || G(l)(s) - G(l)(G) ||`
- And if you have used it from some layers
  - `J(S, G) = Sum (lamda[l]*J(S, G)[l], for all layers)`
- Steps to be made if you want to create a tensorflow model for neural style transfer:
  i. Create an Interactive Session.
  ii. Load the content image.
  iii. Load the style image
  iv. Randomly initialize the image to be generated
  v. Load the VGG16 model
  vi. Build the TensorFlow graph:
    - Run the content image through the VGG16 model and compute the content cost
    - Run the style image through the VGG16 model and compute the style cost
    - Compute the total cost
    - Define the optimizer and the learning rate
  vii. Initialize the TensorFlow graph and run it for a large number of iterations, updating the generated image at every step.

1D and 3D Generalizations

- So far we have used the Conv nets for images which are 2D.
- Conv nets can work with 1D and 3D data as well.
- An example of 1D convolution:
  - Input shape (14, 1)
  - Applying 16 filters with F = 5 , S = 1
  - Output shape will be 10 X 16
  - Applying 32 filters with F = 5, S = 1
  - Output shape will be 6 X 32
- The general equation `(N - F)/S + 1` can be applied here but here it gives a vector rather than a 2D matrix.
- 1D data comes from a lot of resources such as waves, sounds, heartbeat signals.
- In most of the applications that uses 1D data we use Recurrent Neural Network RNN.
- 3D data also are available in some applications like CT scan:
  - 
- Example of 3D convolution:
  - Input shape (14, 14,14, 1)
  - Applying 16 filters with F = 5 , S = 1
  - Output shape (10, 10, 10, 16)
  - Applying 32 filters with F = 5, S = 1
  - Output shape will be (6, 6, 6, 32)

Extras

Keras

- Keras is a high-level neural networks API (programming framework), written in Python and capable of running on top of several lower-level frameworks including TensorFlow, Theano, and CNTK.
- Keras was developed to enable deep learning engineers to build and experiment with different models very quickly.
- Just as TensorFlow is a higher-level framework than Python, Keras is an even higher-level framework and provides additional abstractions.
- Keras will work fine for many common models.
- Layers in Keras:
  - Dense (Fully connected layers).
    - A linear function followed by a non linear function.
  - Convolutional layer.
  - Pooling layer.
  - Normalisation layer.
    - A batch normalization layer.
  - Flatten layer
    - Flatten a matrix into vector.
  - Activation layer
    - Different activations include: relu, tanh, sigmoid, and softmax.
- To train and test a model in Keras there are four steps:
  i. Create the model.
  ii. Compile the model by calling `model.compile(optimizer = "...", loss = "...", metrics = ["accuracy"])`
  iii. Train the model on train data by calling `model.fit(x = ..., y = ..., epochs = ..., batch_size = ...)`

        - You can add a validation set while training too.
    iv. Test the model on test data by calling `model.evaluate(x = ..., y = ...)`
- Summarize of step in Keras: Create->Compile->Fit/Train->Evaluate/Test
- `Model.summary()` gives a lot of useful informations regarding your model including each layers inputs, outputs, and number of parameters at each layer.
- To choose the Keras backend you should go to `$HOME/.keras/keras.json` and change the file to the desired backend like Theano or Tensorflow or whatever backend you want.
- After you create the model you can run it in a tensorflow session without compiling, training, and testing capabilities.
- You can save your model with `model_save` and load your model using `model_load` This will save your whole trained model to disk with the trained weights.

These Notes were made by Mahmoud Badry @2017

        - Speech recognition - Audio data
            - Speech recognition
            - Trigger Word Detection
    - Extras
        - Machine translation attention model (From notebooks)

Course summary

Here are the course summary as its given on the course link:

> This course will teach you how to build models for natural language, audio, and other sequence data. Thanks to deep learning, sequence algorithms are working far better than just two years ago, and this is enabling numerous exciting applications in speech recognition, music synthesis, chatbots, machine translation, natural language understanding, and many others.
>
> You will:
>
> - Understand how to build and train Recurrent Neural Networks (RNNs), and commonly-used variants such as GRUs and LSTMs.
> - Be able to apply sequence models to natural language problems, including text synthesis.
> - Be able to apply sequence models to audio applications, including speech recognition and music synthesis.
>
> This is the fifth and final course of the Deep Learning Specialization.

Recurrent Neural Networks

> Learn about recurrent neural networks. This type of model has been proven to perform extremely well on temporal data. It has several variants including LSTMs, GRUs and Bidirectional RNNs, which you are going to learn about in this section.

Why sequence models

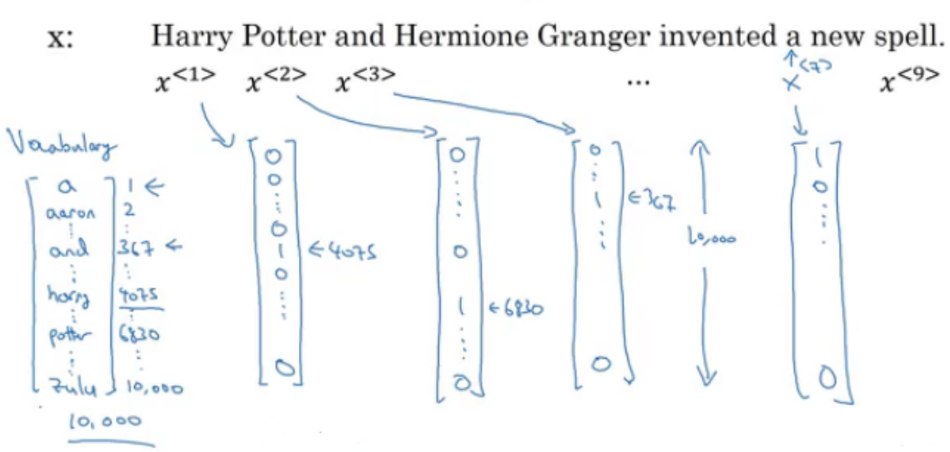
- Sequence Models like RNN and LSTMs have greatly transformed learning on sequences in the past few years.
- Examples of sequence data in applications:
    - Speech recognition (**sequence to sequence**):
        - X: wave sequence
        - Y: text sequence
    - Music generation (**one to sequence**):
        - X: nothing or an integer
        - Y: wave sequence
    - Sentiment classification (**sequence to one**):
        - X: text sequence
        - Y: integer rating from one to five
    - DNA sequence analysis (**sequence to sequence**):
        - X: DNA sequence
        - Y: DNA Labels
    - Machine translation (**sequence to sequence**):
        - X: text sequence (in one language)
        - Y: text sequence (in other language)
    - Video activity recognition (**sequence to one**):
        - X: video frames
        - Y: label (activity)
    - Name entity recognition (**sequence to sequence**):
        - X: text sequence
        - Y: label sequence
        - Can be used by seach engines to index different type of words inside a text.
- All of these problems with different input and output (sequence or not) can be addressed as supervised learning with label data X, Y as the training set.

Notation

- In this section we will discuss the notations that we will use through the course.
- **Motivating example**:
    - Named entity recognition example:
        - X: "Harry Potter and Hermoine Granger invented a new spell."

    - Y: 1 1 0 1 1 0 0 0 0
    - Both elements has a shape of 9. 1 means its a name, while 0 means its not a name.

- We will index the first element of x by $x^{<1>}$, the second $x^{<2>}$ and so on.
  - $x^{<1>}$ = Harry
  - $x^{<2>}$ = Potter
- Similarly, we will index the first element of y by $y^{<1>}$, the second $y^{<2>}$ and so on.
  - $y^{<1>}$ = 1
  - $y^{<2>}$ = 1
- T_x is the size of the input sequence and T_y is the size of the output sequence.
  - $T_x = T_y = 9$ in the last example although they can be different in other problems.
- $x^{(i)<t>}$ is the element t of the sequence of input vector i. Similarly $y^{(i)<t>}$ means the t-th element in the output sequence of the i training example.
- $T_x^{(i)}$ the input sequence length for training example i. It can be different across the examples. Similarly for $T_y^{(i)}$ will be the length of the output sequence in the i-th training example.
- **Representing words**:
  - We will now work in this course with **NLP** which stands for natural language processing. One of the challenges of NLP is how can we represent a word?
  i. We need a **vocabulary** list that contains all the words in our target sets.
    - Example:
      - [a ... And ... Harry ... Potter ... Zulu]
      - Each word will have a unique index that it can be represented with.
      - The sorting here is in alphabetical order.
    - Vocabulary sizes in modern applications are from 30,000 to 50,000. 100,000 is not uncommon. Some of the bigger companies use even a million.
    - To build vocabulary list, you can read all the texts you have and get m words with the most occurrence, or search online for m most occurrent words.
  ii. Create a **one-hot encoding** sequence for each word in your dataset given the vocabulary you have created.
    - While converting, what if we meet a word thats not in your dictionary?
    - We can add a token in the vocabulary with name `<UNK>` which stands for unknown text and use its index for your one-hot vector.
  - Full example:

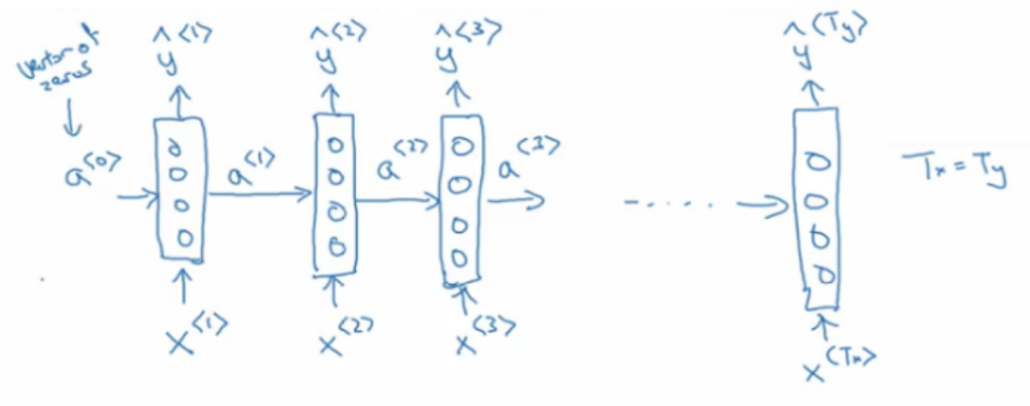


- The goal is given this representation for x to learn a mapping using a sequence model to then target output y as a supervised learning problem.

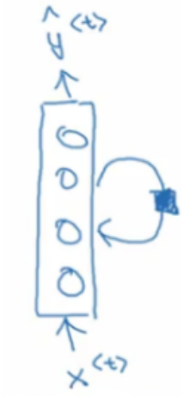
Recurrent Neural Network Model

- Why not to use a standard network for sequence tasks? There are two problems:
  - Inputs, outputs can be different lengths in different examples.
    - This can be solved for normal NNs by paddings with the maximum lengths but it's not a good solution.
  - Doesn't share features learned across different positions of text/sequence.
    - Using a feature sharing like in CNNs can significantly reduce the number of parameters in your model. That's what we will do in RNNs.
- Recurrent neural network doesn't have either of the two mentioned problems.

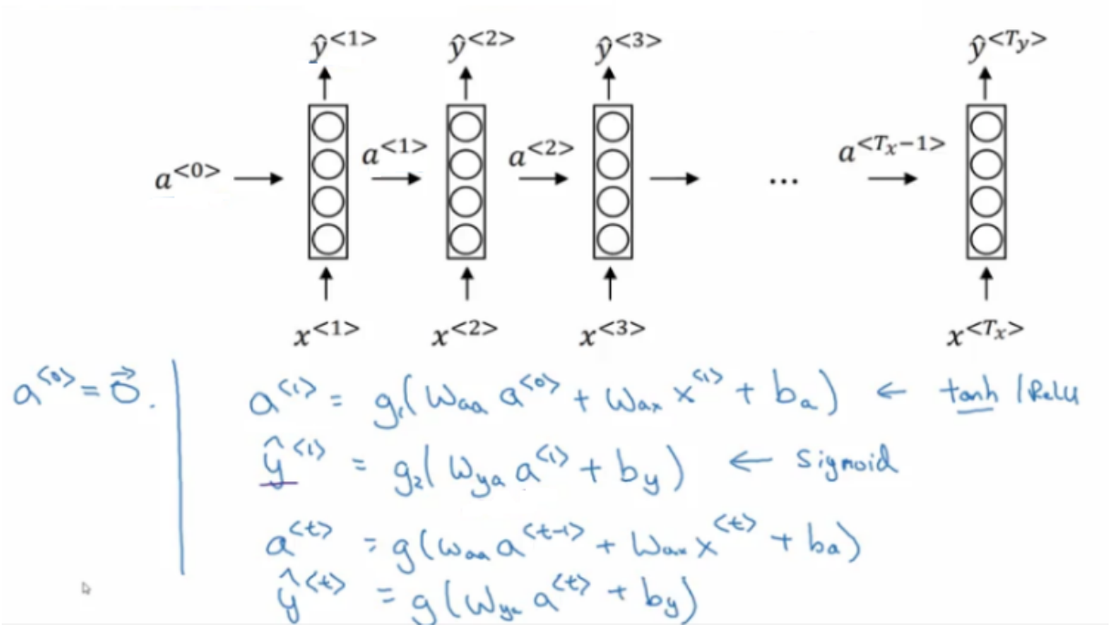
- Lets build a RNN that solves **name entity recognition** task:



  - In this problem $T_x = T_y$. In other problems where they aren't equal, the RNN architecture may be different.
  - $a^{<0>}$ is usually initialized with zeros, but some others may initialize it randomly in some cases.
  - There are three weight matrices here: W_{ax}, W_{aa}, and W_{ya} with shapes:
    - W_{ax}: (NoOfHiddenNeurons, n_x)
    - W_{aa}: (NoOfHiddenNeurons, NoOfHiddenNeurons)
    - W_{ya}: (n_y, NoOfHiddenNeurons)
- The weight matrix W_{aa} is the memory the RNN is trying to maintain from the previous layers.
- A lot of papers and books write the same architecture this way:



  - It's harder to interpreter. It's easier to roll this drawings to the unrolled version.
- In the discussed RNN architecture, the current output $\hat{y}^{<t>}$ depends on the previous inputs and activations.
- Let's have this example 'He Said, "Teddy Roosevelt was a great president"'. In this example Teddy is a person name but we know that from the word **president** that came after Teddy not from **He** and **said** that were before it.
- So limitation of the discussed architecture is that it can not learn from elements later in the sequence. To address this problem we will later discuss **Bidirectional RNN** (BRNN).
- Now let's discuss the forward propagation equations on the discussed architecture:



  - The activation function of a is usually tanh or ReLU and for y depends on your task choosing some activation functions like sigmoid and softmax. In name entity recognition task we will use sigmoid because we only have two classes.
- In order to help us develop complex RNN architectures, the last equations needs to be simplified a bit.

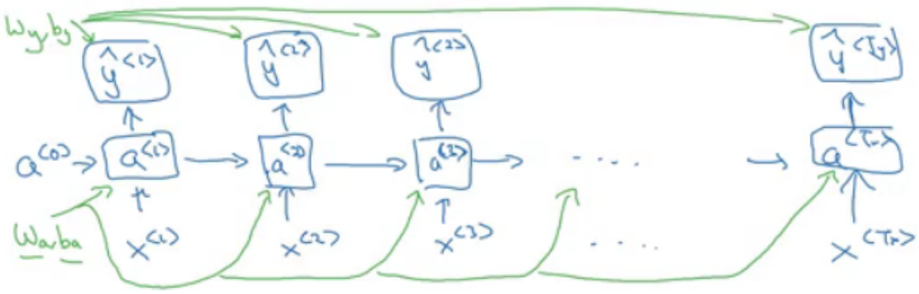
- **Simplified RNN notation**:

$a^{<t>} = g(W_{aa}a^{<t>} + W_{ax}x^{<t>} + b_a)$	$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$ $W_a = [W_{aa}; W_{ax}]$ $[a^{<t-1>}, x^{<t>}] = \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix}$
$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$	$\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$

  - w_a is w_{aa} and w_{ax} stacked horizontally.
  - $[a^{<t-1>}, x^{<t>}]$ is $a^{<t-1>}$ and $x^{<t>}$ stacked vertically.
  - w_a shape: (NoOfHiddenNeurons, NoOfHiddenNeurons + n_x)
  - $[a^{<t-1>}, x^{<t>}]$ shape: (NoOfHiddenNeurons + n_x, 1)

Backpropagation through time

- Let's see how backpropagation works with the RNN architecture.
- Usually deep learning frameworks do backpropagation automatically for you. But it's useful to know how it works in RNNs.
- Here is the graph:

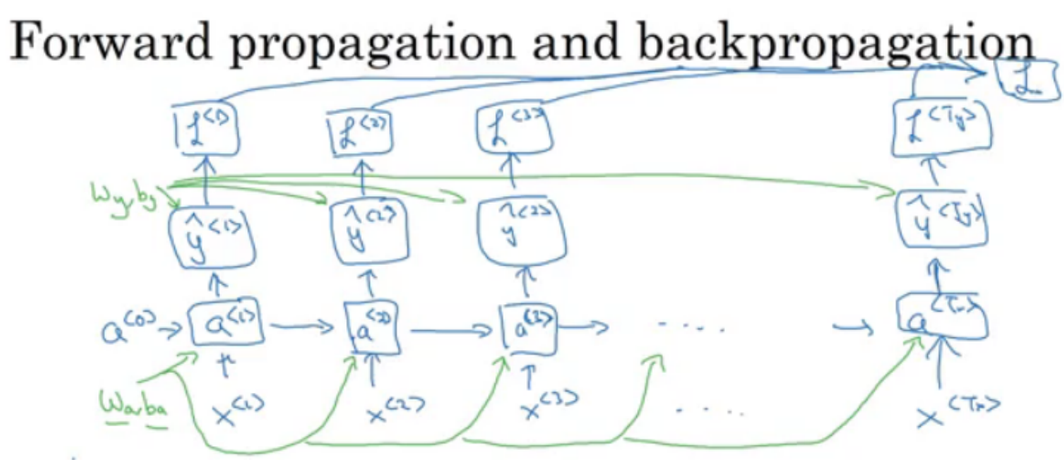


  - Where w_a, b_a, w_y, and b_y are shared across each element in a sequence.
- We will use the cross-entropy loss function:

$$\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>}\log\hat{y}^{<t>} - (1-y^{<t>})\log(1-\hat{y}^{<t>})$$

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y}\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

  - Where the first equation is the loss for one example and the loss for the whole sequence is given by the summation over all the calculated single example losses.
- Graph with losses:

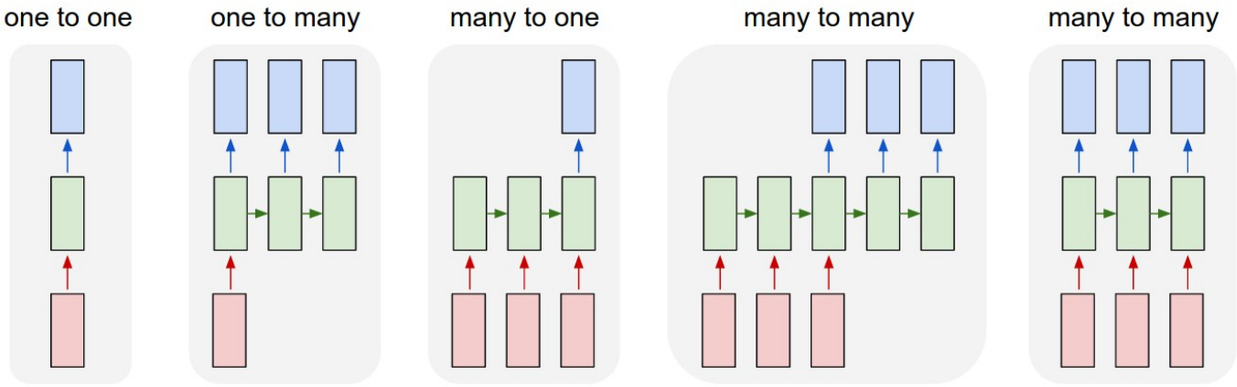


- The backpropagation here is called **backpropagation through time** because we pass activation `a` from one sequence element to another like backwards in time.

Different types of RNNs

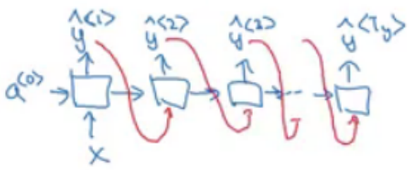
- So far we have seen only one RNN architecture in which T_x equals T_Y. In some other problems, they may not equal so we need different architectures.

- The ideas in this section was inspired by Andrej Karpathy blog. Mainly this image has all types:

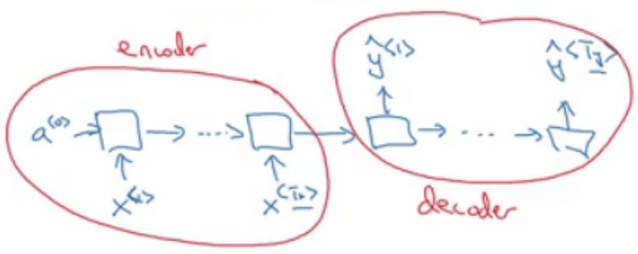


- The architecture we have descried before is called **Many to Many**.
- In sentiment analysis problem, X is a text while Y is an integer that rangers from 1 to 5. The RNN architecture for that is **Many to One** as in Andrej Karpathy image.

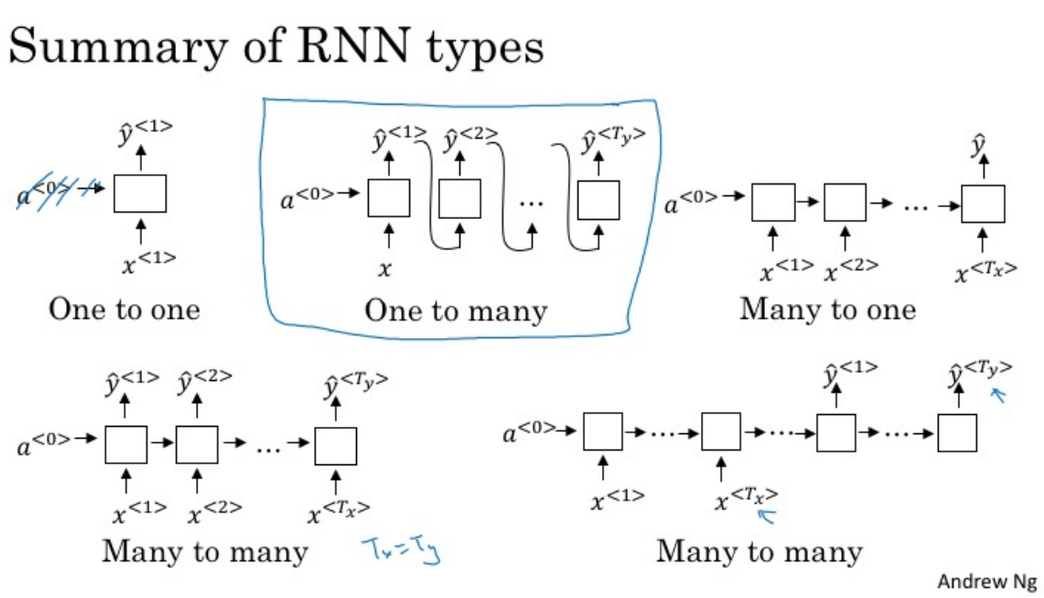
- A **One to Many** architecture application would be music generation.



  - Note that starting the second layer we are feeding the generated output back to the network.
- There are another interesting architecture in **Many To Many**. Applications like machine translation inputs and outputs sequences have different lengths in most of the cases. So an alternative *Many To Many* architecture that fits the translation would be as follows:



  - There are an encoder and a decoder parts in this architecture. The encoder encodes the input sequence into one matrix and feed it to the decoder to generate the outputs. Encoder and decoder have different weight matrices.
- Summary of RNN types:



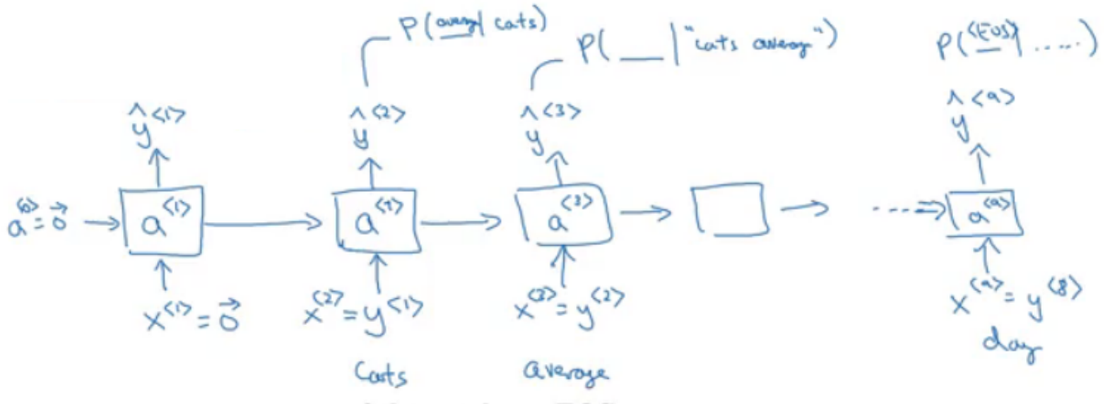
- There is another architecture which is the **attention** architecture which we will talk about in chapter 3.

Language model and sequence generation

- RNNs do very well in language model problems. In this section, we will build a language model using RNNs.
- **What is a language model**

- Let's say we are solving a speech recognition problem and someone says a sentence that can be interpreted into to two sentences:
  - The apple and **pair** salad
  - The apple and **pear** salad
- **Pair** and **pear** sounds exactly the same, so how would a speech recognition application choose from the two.
- That's where the language model comes in. It gives a probability for the two sentences and the application decides the best based on this probability.

- The job of a language model is to give a probability of any given sequence of words.
- **How to build language models with RNNs?**
  - The first thing is to get a **training set**: a large corpus of target language text.
  - Then tokenize this training set by getting the vocabulary and then one-hot each word.
  - Put an end of sentence token `<EOS>` with the vocabulary and include it with each converted sentence. Also, use the token `<UNK>` for the unknown words.
- Given the sentence "Cats average 15 hours of sleep a day. `<EOS>`"
  - In training time we will use this:



  - The loss function is defined by cross-entropy loss:

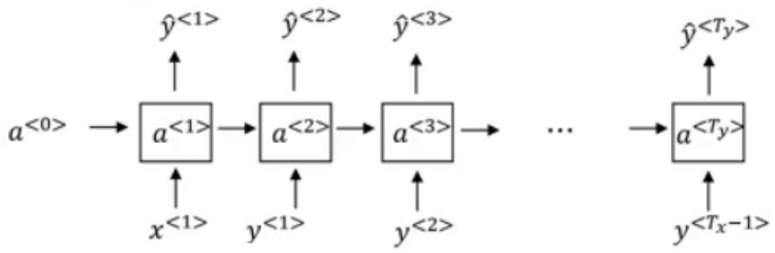
$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$$

$$\mathcal{L} = \sum \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

    - `i` is for all elements in the corpus, `t` - for all timesteps.
- To use this model:
  i. For predicting the chance of **next word**, we feed the sentence to the RNN and then get the final $y^{\wedge<t>}$ hot vector and sort it by maximum probability.
  ii. For taking the **probability of a sentence**, we compute this:
    - $p(y^{<1>}, y^{<2>}, y^{<3>}) = p(y^{<1>}) * p(y^{<2>} \mid y^{<1>}) * p(y^{<3>} \mid y^{<1>}, y^{<2>})$
    - This is simply feeding the sentence into the RNN and multiplying the probabilities (outputs).

Sampling novel sequences

- After a sequence model is trained on a language model, to check what the model has learned you can apply it to sample novel sequence.
- Lets see the steps of how we can sample a novel sequence from a trained sequence language model:
  i. Given this model:



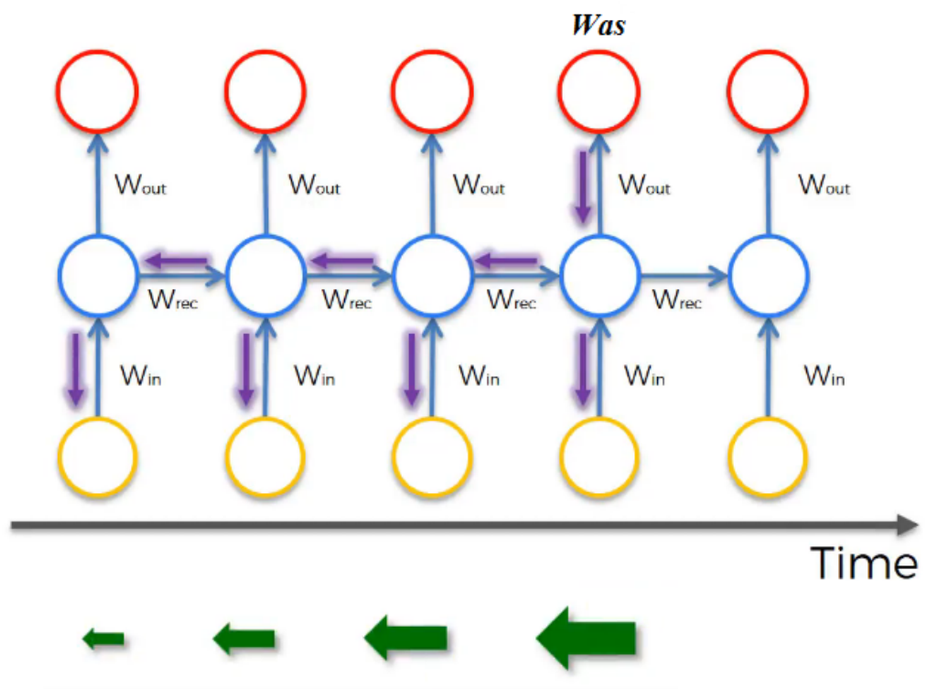
  ii. We first pass $a^{<0>}$ = zeros vector, and $x^{<1>}$ = zeros vector.
  iii. Then we choose a prediction randomly from distribution obtained by $\hat{y}^{<1>}$. For example it could be "The".
    - In numpy this can be implemented using: `numpy.random.choice(...)`
    - This is the line where you get a random beginning of the sentence each time you sample run a novel sequence.
  iv. We pass the last predicted word with the calculated $a^{<1>}$
  v. We keep doing 3 & 4 steps for a fixed length or until we get the `<EOS>` token.
  vi. You can reject any `<UNK>` token if you mind finding it in your output.
- So far we have to build a word-level language model. It's also possible to implement a **character-level** language model.
- In the character-level language model, the vocabulary will contain `[a-zA-Z0-9]`, punctuation, special characters and possibly token.
- Character-level language model has some pros and cons compared to the word-level language model
  - Pros:
    a. There will be no `<UNK>` token - it can create any word.

- Cons:
  a. The main disadvantage is that you end up with much longer sequences.
  b. Character-level language models are not as good as word-level language models at capturing long range dependencies between how the the earlier parts of the sentence also affect the later part of the sentence.
  c. Also more computationally expensive and harder to train.

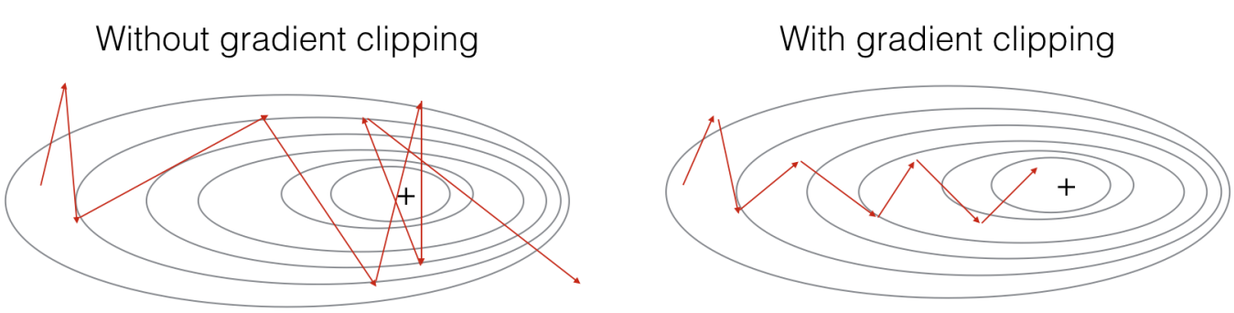
- The trend Andrew has seen in NLP is that for the most part, a word-level language model is still used, but as computers get faster there are more and more applications where people are, at least in some special cases, starting to look at more character-level models. Also, they are used in specialized applications where you might need to deal with unknown words or other vocabulary words a lot. Or they are also used in more specialized applications where you have a more specialized vocabulary.

Vanishing gradients with RNNs

- One of the problems with naive RNNs that they run into **vanishing gradient** problem.
- An RNN that process a sequence data with the size of 10,000 time steps, has 10,000 deep layers which is very hard to optimize.
- Let's take an example. Suppose we are working with language modeling problem and there are two sequences that model tries to learn:
  - "The **cat**, which already ate ..., **was** full"
  - "The **cats**, which already ate ..., **were** full"
  - Dots represent many words in between.
- What we need to learn here that "was" came with "cat" and that "were" came with "cats". The naive RNN is not very good at capturing very long-term dependencies like this.
- As we have discussed in Deep neural networks, deeper networks are getting into the vanishing gradient problem. That also happens with RNNs with a long sequence size.



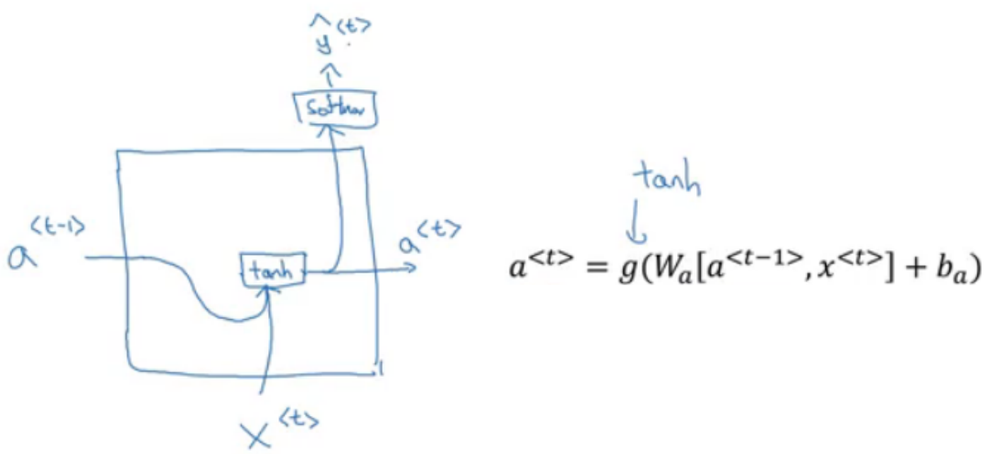
- For computing the word "was", we need to compute the gradient for everything behind. Multiplying fractions tends to vanish the gradient, while multiplication of large number tends to explode it.
  - Therefore some of your weights may not be updated properly.
- In the problem we descried it means that its hard for the network to memorize "was" word all over back to "cat". So in this case, the network won't identify the singular/plural words so that it gives it the right grammar form of verb was/were.
- The conclusion is that RNNs aren't good in **long-term dependencies**.
- > In theory, RNNs are absolutely capable of handling such “long-term dependencies.” A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don’t seem to be able to learn them. http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- *Vanishing gradients* problem tends to be the bigger problem with RNNs than the *exploding gradients* problem. We will discuss how to solve it in next sections.
- Exploding gradients can be easily seen when your weight values become `NaN`. So one of the ways solve exploding gradient is to apply **gradient clipping** means if your gradient is more than some threshold - re-scale some of your gradient vector so that is not too big. So there are cliped according to some maximum value.



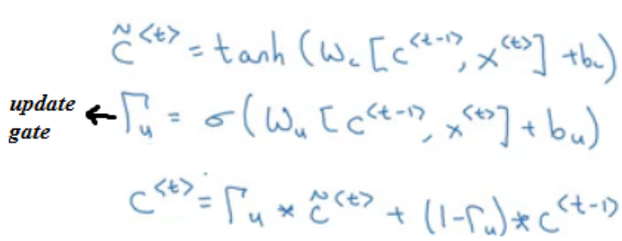
- **Extra**:
  - Solutions for the Exploding gradient problem:
    - Truncated backpropagation.
      - Not to update all the weights in the way back.
      - Not optimal. You won't update all the weights.
    - Gradient clipping.
  - Solution for the Vanishing gradient problem:
    - Weight initialization.
      - Like He initialization.
    - Echo state networks.
    - Use LSTM/GRU networks.
      - Most popular.
      - We will discuss it next.

Gated Recurrent Unit (GRU)

- GRU is an RNN type that can help solve the vanishing gradient problem and can remember the long-term dependencies.
- The basic RNN unit can be visualized to be like this:



- We will represent the GRU with a similar drawings.
- Each layer in **GRUs** has a new variable `c` which is the memory cell. It can tell to whether memorize something or not.
- In GRUs, $C^{<t>} = a^{<t>}$
- Equations of the GRUs:

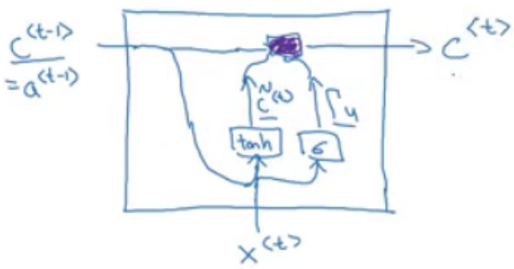


  - The update gate is between 0 and 1
    - To understand GRUs imagine that the update gate is either 0 or 1 most of the time.
  - So we update the memory cell based on the update cell and the previous cell.
- Lets take the cat sentence example and apply it to understand this equations:
  - Sentence: "The **cat**, which already ate ....., **was** full"
  - We will suppose that U is 0 or 1 and is a bit that tells us if a singular word needs to be memorized.

- Splitting the words and get values of C and U at each place:

Word	Update gate(U)	Cell memory (C)
The	0	val
cat	1	new_val
which	0	new_val
already	0	new_val
...	0	new_val
was	1 (I don't need it anymore)	newer_val
full	..	..

- Drawing for the GRUs



  - Drawings like in http://colah.github.io/posts/2015-08-Understanding-LSTMs/ is so popular and makes it easier to understand GRUs and LSTMs. But Andrew Ng finds it's better to look at the equations.

- Because the update gate U is usually a small number like 0.00001, GRUs doesn't suffer the vanishing gradient problem.
  - In the equation this makes $C^{<t>} = C^{<t-1>}$ in a lot of cases.
- Shapes:
  - $a^{<t>}$ shape is (NoOfHiddenNeurons, 1)
  - $c^{<t>}$ is the same as $a^{<t>}$
  - $c^{\sim <t>}$ is the same as $a^{<t>}$
  - $u^{<t>}$ is also the same dimensions of $a^{<t>}$
- The multiplication in the equations are element wise multiplication.
- What has been descried so far is the Simplified GRU unit. Let's now describe the full one:
  - The full GRU contains a new gate that is used with to calculate the candidate C. The gate tells you how relevant is $C^{<t-1>}$ to $C^{<t>}$
  - Equations:

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) + c^{<t-1>}$$

  - Shapes are the same
- So why we use these architectures, why don't we change them, how we know they will work, why not add another gate, why not use the simpler GRU instead of the full GRU; well researchers has experimented over years all the various types of these architectures with many many different versions and also addressing the vanishing gradient problem. They have found that full GRUs are one of the best RNN architectures to be used for many different problems. You can make your design but put in mind that GRUs and LSTMs are standards.

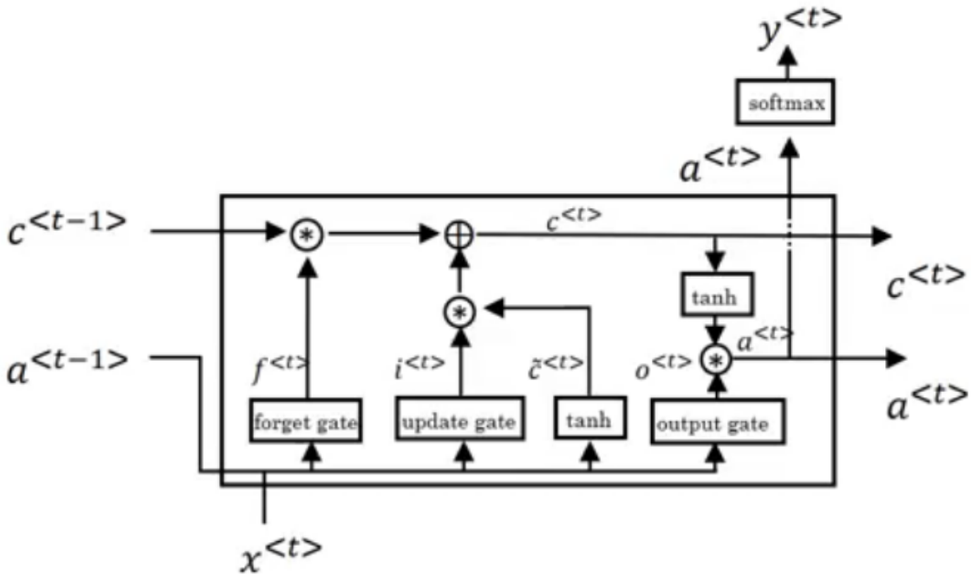
Long Short Term Memory (LSTM)

- LSTM - the other type of RNN that can enable you to account for long-term dependencies. It's more powerful and general than GRU.
- In LSTM , $C^{<t>}$!= $a^{<t>}$

- Here are the equations of an LSTM unit:

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$
$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u) \quad \text{(update)}$$
$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f) \quad \text{(forget)}$$
$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o) \quad \text{(output)}$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$
$$a^{<t>} = \Gamma_o * \tanh c^{<t>}$$

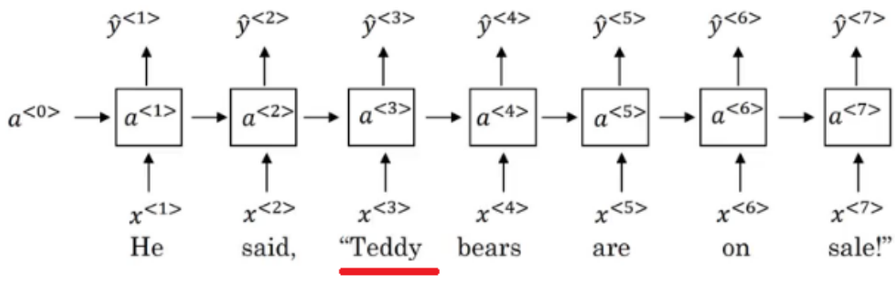
- In GRU we have an update gate `U`, a relevance gate `r`, and a candidate cell variables $C\tilde{}^{<t>}$ while in LSTM we have an update gate `U` (sometimes it's called input gate I), a forget gate `F`, an output gate `O`, and a candidate cell variables $C\tilde{}^{<t>}$
- Drawings (inspired by http://colah.github.io/posts/2015-08-Understanding-LSTMs/):



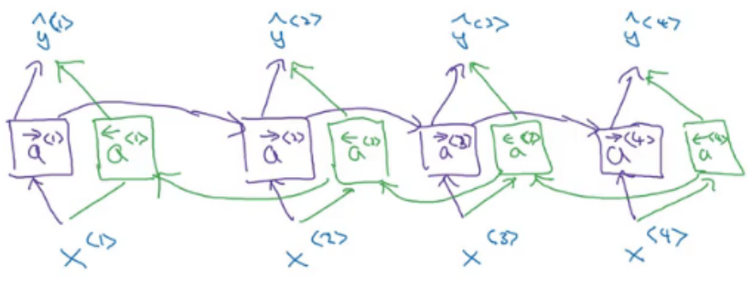
- Some variants on LSTM includes:
  - LSTM with **peephole connections**.
    - The normal LSTM with $C^{<t-1>}$ included with every gate.
- There isn't a universal superior between LSTM and it's variants. One of the advantages of GRU is that it's simpler and can be used to build much bigger network but the LSTM is more powerful and general.

Bidirectional RNN

- There are still some ideas to let you build much more powerful sequence models. One of them is bidirectional RNNs and another is Deep RNNs.
- As we saw before, here is an example of the Name entity recognition task:



- The name **Teddy** cannot be learned from **He** and **said**, but can be learned from **bears**.
- BiRNNs fixes this issue.
- Here is BRNNs architecture:

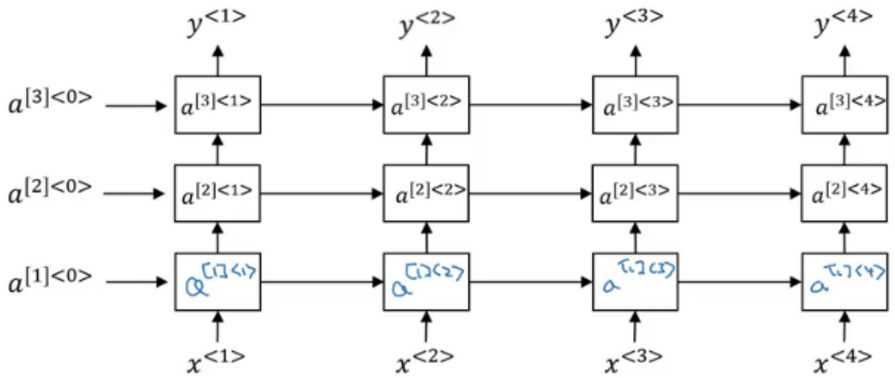


- Note, that BiRNN is an **acyclic graph**.
- Part of the forward propagation goes from left to right, and part - from right to left. It learns from both sides.
- To make predictions we use $\hat{y}^{<t>}$ by using the two activations that come from left and right.
- The blocks here can be any RNN block including the basic RNNs, LSTMs, or GRUs.
- For a lot of NLP or text processing problems, a BiRNN with LSTM appears to be commonly used.

- The disadvantage of BiRNNs that you need the entire sequence before you can process it. For example, in live speech recognition if you use BiRNNs you will need to wait for the person who speaks to stop to take the entire sequence and then make your predictions.

Deep RNNs

- In a lot of cases the standard one layer RNNs will solve your problem. But in some problems its useful to stack some RNN layers to make a deeper network.
- For example, a deep RNN with 3 layers would look like this:



- In feed-forward deep nets, there could be 100 or even 200 layers. In deep RNNs stacking 3 layers is already considered deep and expensive to train.
- In some cases you might see some feed-forward network layers connected after recurrent cell.

Back propagation with RNNs

- > In modern deep learning frameworks, you only have to implement the forward pass, and the framework takes care of the backward pass, so most deep learning engineers do not need to bother with the details of the backward pass. If however you are an expert in calculus and want to see the details of backprop in RNNs, you can work through this optional portion of the notebook.

- The quote is taken from this notebook. If you want the details of the back propagation with programming notes look at the linked notebook.

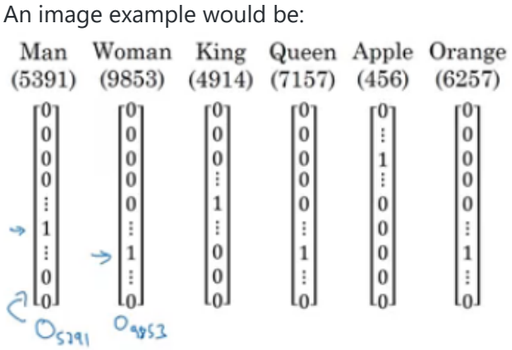
Natural Language Processing & Word Embeddings

> Natural language processing with deep learning is an important combination. Using word vector representations and embedding layers you can train recurrent neural networks with outstanding performances in a wide variety of industries. Examples of applications are sentiment analysis, named entity recognition and machine translation.

Introduction to Word Embeddings

Word Representation

- NLP has been revolutionized by deep learning and especially by RNNs and deep RNNs.
- Word embeddings is a way of representing words. It lets your algorithm automatically understand the analogies between words like "king" and "queen".
- So far we have defined our language by a vocabulary. Then represented our words with a one-hot vector that represents the word in the vocabulary.
  - An image example would be:



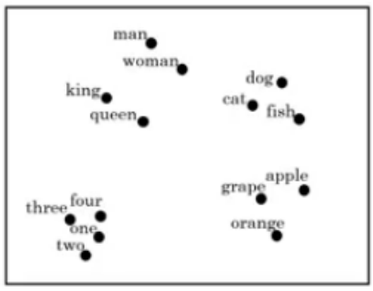
  - We will use the annotation **O** $_{idx}$ for any word that is represented with one-hot like in the image.
  - One of the weaknesses of this representation is that it treats a word as a thing that itself and it doesn't allow an algorithm to generalize across words.
    - For example: "I want a glass of **orange** _____", a model should predict the next word as **juice**.
    - A similar example "I want a glass of **apple** _____", a model won't easily predict **juice** here if it wasn't trained on it. And if so the two examples aren't related although orange and apple are similar.
  - Inner product between any one-hot encoding vector is zero. Also, the distances between them are the same.

- So, instead of a one-hot presentation, won't it be nice if we can learn a featurized representation with each of these words: man, woman, king, queen, apple, and orange?

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97
...						
size						
cost						
alive						
verb						

300

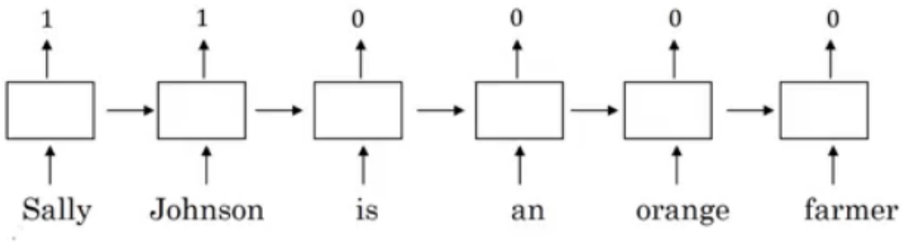
  - Each word will have a, for example, 300 features with a type of float point number.
    - Each word column will be a 300-dimensional vector which will be the representation.
    - We will use the notation $\mathbf{e}_{5391}$ to describe **man** word features vector.
    - Now, if we return to the examples we described again:
      - "I want a glass of **orange** _____"
      - I want a glass of **apple** _____
    - Orange and apple now share a lot of similar features which makes it easier for an algorithm to generalize between them.
    - We call this representation **Word embeddings**.
- To visualize word embeddings we use a t-SNE algorithm to reduce the features to 2 dimensions which makes it easy to visualize:



    - You will get a sense that more related words are closer to each other.
- The **word embeddings** came from that we need to embed a unique vector inside a n-dimensional space.

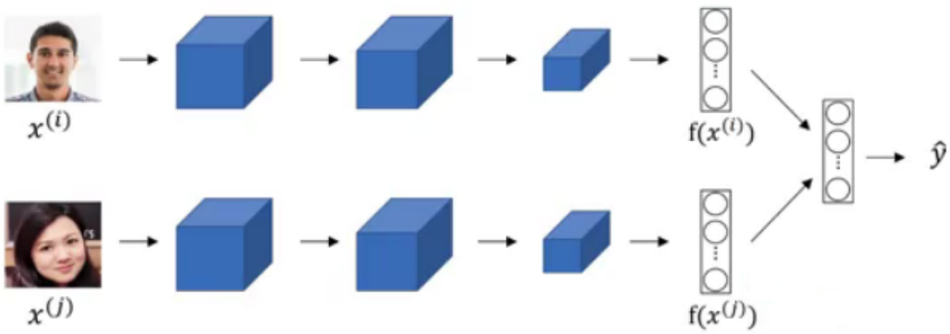
Using word embeddings

- Let's see how we can take the feature representation we have extracted from each word and apply it in the Named entity recognition problem.
- Given this example (from named entity recognition):



- **Sally Johnson** is a person's name.
- After training on this sentence the model should find out that the sentence "**Robert Lin** is an *apple* farmer" contains Robert Lin as a name, as apple and orange have near representations.
- Now if you have tested your model with this sentence "**Mahmoud Badry** is a *durian* cultivator" the network should learn the name even if it hasn't seen the word *durian* before (during training). That's the power of word representations.
- The algorithms that are used to learn **word embeddings** can examine billions of words of unlabeled text - for example, 100 billion words and learn the representation from them.
- Transfer learning and word embeddings:
  i. Learn word embeddings from large text corpus (1-100 billion of words).
    - Or download pre-trained embedding online.
  ii. Transfer embedding to new task with the smaller training set (say, 100k words).

iii. Optional: continue to finetune the word embeddings with new data.
    - You bother doing this if your smaller training set (from step 2) is big enough.
- Word embeddings tend to make the biggest difference when the task you're trying to carry out has a relatively smaller training set.
- Also, one of the advantages of using word embeddings is that it reduces the size of the input!
    - 10,000 one hot compared to 300 features vector.
- Word embeddings have an interesting relationship to the face recognition task:



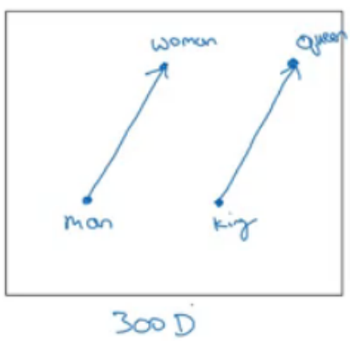
    - In this problem, we encode each face into a vector and then check how similar are these vectors.
    - Words **encoding** and **embeddings** have a similar meaning here.
- In the word embeddings task, we are learning a representation for each word in our vocabulary (unlike in image encoding where we have to map each new image to some n-dimensional vector). We will discuss the algorithm in next sections.

Properties of word embeddings

- One of the most fascinating properties of word embeddings is that they can also help with analogy reasoning. While analogy reasoning may not be by itself the most important NLP application, but it might help convey a sense of what these word embeddings can do.
- Analogies example:
    - Given this word embeddings table:

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	−1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.70	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97
	e_{Man}	e_{Woman}	e_{King}	e_{Queen}		

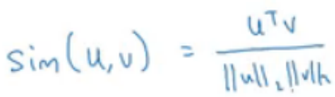
    - Can we conclude this relation:
        - Man ==> Woman
        - King ==> ??
    - Lets subtract e_{Man} from e_{Woman}. This will equal the vector `[-2 0 0 0]`
    - Similar e_{King} - e_{Queen} = `[-2 0 0 0]`
    - So the difference is about the gender in both.



        - This vector represents the gender.
        - This drawing is a 2D visualization of the 4D vector that has been extracted by a t-SNE algorithm. It's a drawing just for visualization. Don't rely on the t-SNE algorithm for finding parallels.
    - So we can reformulate the problem to find:
        - e_{Man} - e_{Woman} ≈ e_{King} - $e_{??}$
    - It can also be represented mathematically by:

- It turns out that e_{Queen} is the best solution here that gets the the similar vector.

- Cosine similarity - the most commonly used similarity function:
  - Equation:



    - `CosineSimilarity(u, v)` = `u . v` / `||u|| ||v||` = cos(θ)
    - The top part represents the inner product of `u` and `v` vectors. It will be large if the vectors are very similar.
- You can also use Euclidean distance as a similarity function (but it rather measures a dissimilarity, so you should take it with negative sign).
- We can use this equation to calculate the similarities between word embeddings and on the analogy problem where `u` = e_w and `v` = e_{king} - e_{man} + e_{woman}

Embedding matrix

- When you implement an algorithm to learn a word embedding, what you end up learning is a **embedding matrix**.
- Let's take an example:
  - Suppose we are using 10,000 words as our vocabulary (plus token).
  - The algorithm should create a matrix `E` of the shape (300, 10000) in case we are extracting 300 features.

  - If O_{6257} is the one hot encoding of the word **orange** of shape (10000, 1), then *np.dot(`E`,O_{6257}) = e_{6257}* which shape is (300, 1).
  - Generally *np.dot(`E`, O_j) = e_j*
- In the next sections, you will see that we first initialize `E` randomly and then try to learn all the parameters of this matrix.
- In practice it's not efficient to use a dot multiplication when you are trying to extract the embeddings of a specific word, instead, we will use slicing to slice a specific column. In Keras there is an embedding layer that extracts this column with no multiplication.

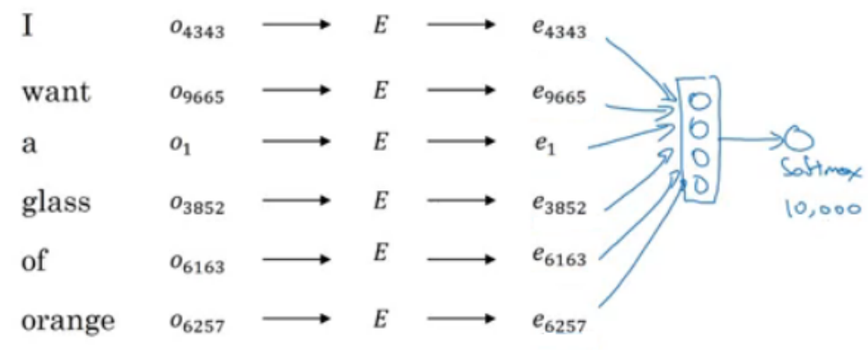
Learning Word Embeddings: Word2vec & GloVe

Learning word embeddings

- Let's start learning some algorithms that can learn word embeddings.
- At the start, word embeddings algorithms were complex but then they got simpler and simpler.
- We will start by learning the complex examples to make more intuition.
- **Neural language model**:
  - Let's start with an example:

I	want	a	glass	of	orange	_____.
4343	9665	1	3852	6163	6257	

  - We want to build a language model so that we can predict the next word.
  - So we use this neural network to learn the language model



    - We get e_j by `np.dot(` E `,o<sub>j</sub>)`
    - NN layer has parameters `W1` and `b1` while softmax layer has parameters `W2` and `b2`
    - Input dimension is (300*6, 1) if the window size is 6 (six previous words).
    - Here we are optimizing `E` matrix and layers parameters. We need to maximize the likelihood to predict the next word given the context (previous words).

    - This model was build in 2003 and tends to work pretty decent for learning word embeddings.
- In the last example we took a window of 6 words that fall behind the word that we want to predict. There are other choices when we are trying to learn word embeddings.
    - Suppose we have an example: "I want a glass of orange **juice** to go along with my cereal"
    - To learn **juice**, choices of **context** are:
        a. Last 4 words.
            - We use a window of last 4 words (4 is a hyperparameter), "a glass of orange" and try to predict the next word from it.
        b. 4 words on the left and on the right.
            - "a glass of orange" and "to go along with"
        c. Last 1 word.
            - "orange"
        d. Nearby 1 word.
            - "glass" word is near juice.
            - This is the idea of **skip grams** model.
            - The idea is much simpler and works remarkably well.
            - We will talk about this in the next section.
- Researchers found that if you really want to build a *language model*, it's natural to use the last few words as a context. But if your main goal is really to learn a *word embedding*, then you can use all of these other contexts and they will result in very meaningful work embeddings as well.
- To summarize, the language modeling problem poses a machines learning problem where you input the context (like the last four words) and predict some target words. And posing that problem allows you to learn good word embeddings.

Word2Vec

- Before presenting Word2Vec, lets talk about **skip-grams**:
    - For example, we have the sentence: "I want a glass of orange juice to go along with my cereal"
    - We will choose **context** and **target**.
    - The target is chosen randomly based on a window with a specific size.

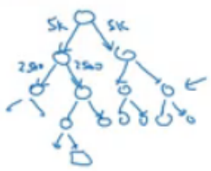
Context	Target	How far
orange	juice	+1
orange	glass	-2
orange	my	+6

      We have converted the problem into a supervised problem.
    - This is not an easy learning problem because learning within -10/+10 words (10 - an example) is hard.
    - We want to learn this to get our word embeddings model.
- Word2Vec model:
    - Vocabulary size = 10,000 words
    - Let's say that the context word are `c` and the target word is `t`
    - We want to learn `c` to `t`
    - We get e_c by `E` . o_c
    - We then use a softmax layer to get `P(t|c)` which is $\hat{y}$
    - Also we will use the cross-entropy loss function.
    - This model is called skip-grams model.
- The last model has a problem with the softmax layer:

$$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$$

    - Here we are summing 10,000 numbers which corresponds to the number of words in our vocabulary.
    - If this number is larger say 1 million, the computation will become very slow.

- One of the solutions for the last problem is to use "**Hierarchical softmax classifier**" which works as a tree classifier.

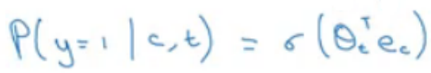


- In practice, the hierarchical softmax classifier doesn't use a balanced tree like the drawn one. Common words are at the top and less common are at the bottom.
- How to sample the context **c**?
  - One way is to choose the context by random from your corpus.
  - If you have done it that way, there will be frequent words like "the, of, a, and, to, .." that can dominate other words like "orange, apple, durian,..."
  - In practice, we don't take the context uniformly random, instead there are some heuristics to balance the common words and the non-common words.
- word2vec paper includes 2 ideas of learning word embeddings. One is skip-gram model and another is CBoW (continuous bag-of-words).

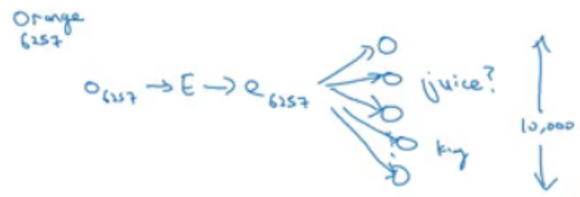
Negative Sampling

- Negative sampling allows you to do something similar to the skip-gram model, but with a much more efficient learning algorithm. We will create a different learning problem.
- Given this example:
  - "I want a glass of orange juice to go along with my cereal"
- The sampling will look like this:
Context	Word	target
orange	juice	1
orange	king	0
orange	book	0
orange	the	0
orange	of	0

  We get positive example by using the same skip-grams technique, with a fixed window that goes around.
- To generate a negative example, we pick a word randomly from the vocabulary.
- Notice, that we got word "of" as a negative example although it appeared in the same sentence.
- So the steps to generate the samples are:
  i. Pick a positive context
  ii. Pick a k negative contexts from the dictionary.
- k is recommended to be from 5 to 20 in small datasets. For larger ones - 2 to 5.
- We will have a ratio of k negative examples to 1 positive ones in the data we are collecting.
- Now let's define the model that will learn this supervised learning problem:
  - Lets say that the context word are `c` and the word are `t` and `y` is the target.
  - We will apply the simple logistic regression model.



  - The logistic regression model can be drawn like this:



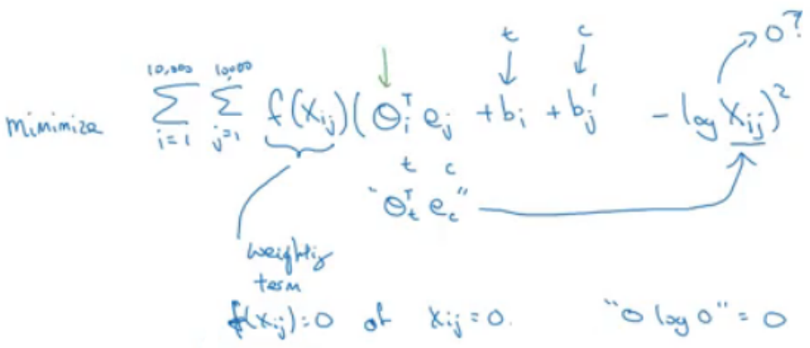
  - So we are like having 10,000 binary classification problems, and we only train k+1 classifier of them in each iteration.
- How to select negative samples:

- We can sample according to empirical frequencies in words corpus which means according to how often different words appears. But the problem with that is that we will have more frequent words like *the, of, and*...
- The best is to sample with this equation (according to authors):

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10{,}000} f(w_j)^{3/4}}$$

GloVe word vectors

- GloVe is another algorithm for learning the word embedding. It's the simplest of them.
- This is not used as much as word2vec or skip-gram models, but it has some enthusiasts because of its simplicity.
- GloVe stands for Global vectors for word representation.
- Let's use our previous example: "I want a glass of orange juice to go along with my cereal".
- We will choose a context and a target from the choices we have mentioned in the previous sections.
- Then we will calculate this for every pair: X_{ct} = # times `t` appears in context of `c`
- $X_{ct} = X_{tc}$ if we choose a window pair, but they will not equal if we choose the previous words for example. In GloVe they use a window which means they are equal
- The model is defined like this:

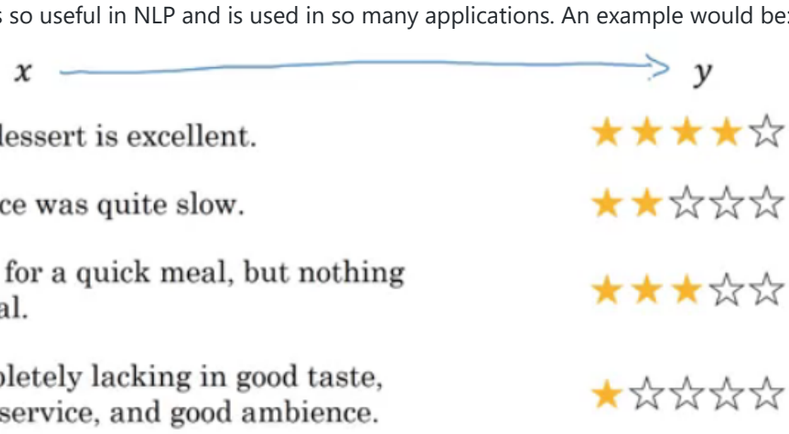


- f(x) - the weighting term, used for many reasons which include:
  - The `log(0)` problem, which might occur if there are no pairs for the given target and context values.
  - Giving not too much weight for stop words like "is", "the", and "this" which occur many times.
  - Giving not too little weight for infrequent words.
- **Theta** and **e** are symmetric which helps getting the final word embedding.
- *Conclusions on word embeddings:*
  - If this is your first try, you should try to download a pre-trained model that has been made and actually works best.
  - If you have enough data, you can try to implement one of the available algorithms.
  - Because word embeddings are very computationally expensive to train, most ML practitioners will load a pre-trained set of embeddings.
  - A final note that you can't guarantee that the axis used to represent the features will be well-aligned with what might be easily humanly interpretable axis like gender, royal, age.

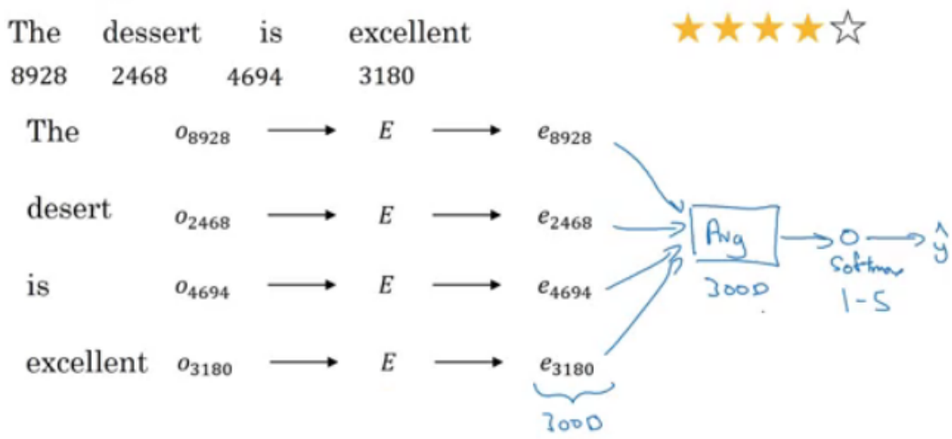
Applications using Word Embeddings

Sentiment Classification

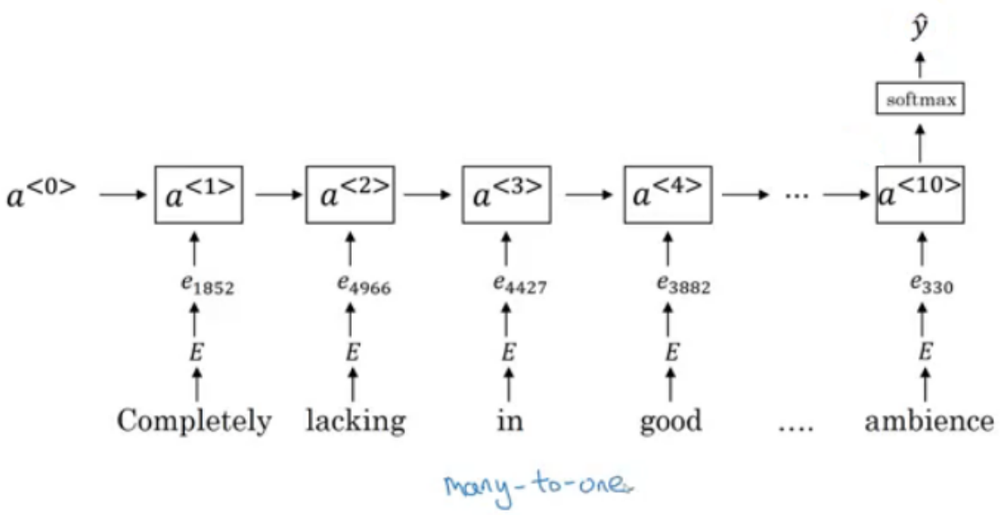
- As we have discussed before, Sentiment classification is the process of finding if a text has a positive or a negative review. Its so useful in NLP and is used in so many applications. An example would be:



- One of the challenges with it, is that you might not have a huge labeled training data for it, but using word embeddings can help getting rid of this.
- The common dataset sizes varies from 10,000 to 100,000 words.
- A simple sentiment classification model would be like this:



  - The embedding matrix may have been trained on say 100 billion words.
  - Number of features in word embedding is 300.
  - We can use **sum** or **average** given all the words then pass it to a softmax classifier. That makes this classifier works for short or long sentences.
- One of the problems with this simple model is that it ignores words order. For example "Completely lacking in **good** taste, **good** service, and **good** ambience" has the word *good* 3 times but its a negative review.
- A better model uses an RNN for solving this problem:

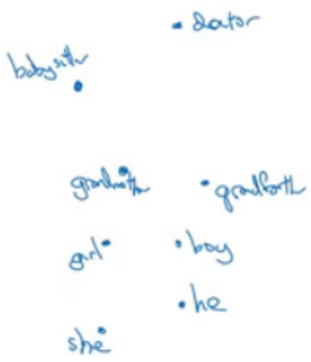


  - And so if you train this algorithm, you end up with a pretty decent sentiment classification algorithm.
  - Also, it will generalize better even if words weren't in your dataset. For example you have the sentence "Completely **absent** of good taste, good service, and good ambience", then even if the word "absent" is not in your label training set, if it was in your 1 billion or 100 billion word corpus used to train the word embeddings, it might still get this right and generalize much better even to words that were in the training set used to train the word embeddings but not necessarily in the label training set that you had for specifically the sentiment classification problem.

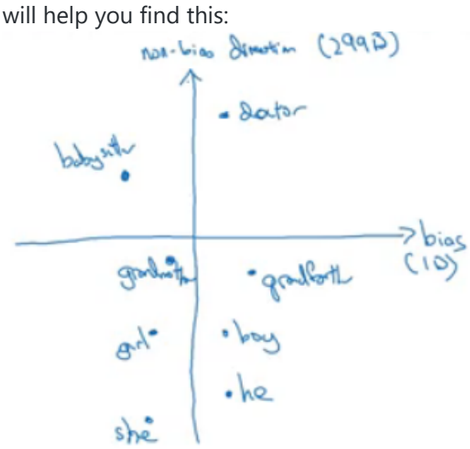
Debiasing word embeddings

- We want to make sure that our word embeddings are free from undesirable forms of bias, such as gender bias, ethnicity bias and so on.
- Horrifying results on the trained word embeddings in the context of Analogies:
  - Man : Computer_programmer as Woman : **Homemaker**
  - Father : Doctor as Mother : **Nurse**
- Word embeddings can reflect gender, ethnicity, age, sexual orientation, and other biases of text used to train the model.
- Learning algorithms by general are making important decisions and it mustn't be biased.
- Andrew thinks we actually have better ideas for quickly reducing the bias in AI than for quickly reducing the bias in the human race, although it still needs a lot of work to be done.
- Addressing bias in word embeddings steps:
  - Idea from the paper: https://arxiv.org/abs/1607.06520

- Given these learned embeddings:

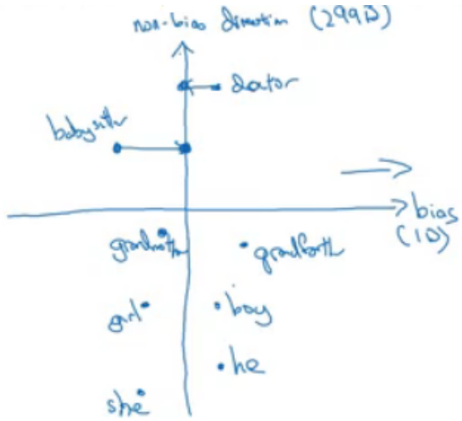


- We need to solve the **gender bias** here. The steps we will discuss can help solve any bias problem but we are focusing here on gender bias.
- Here are the steps:
  a. Identify the direction:
    - Calculate the difference between:
      - $e_{he} - e_{she}$
      - $e_{male} - e_{female}$
      - ....
    - Choose some k differences and average them.
    - This will help you find this:



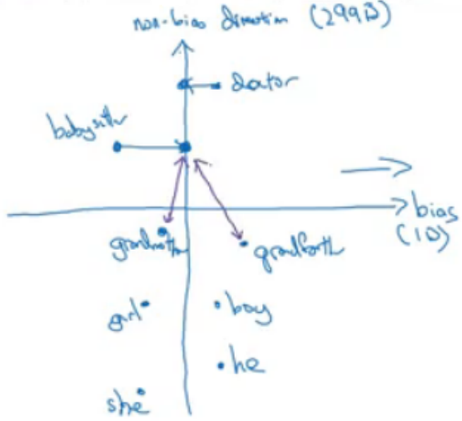
    - By that we have found the bias direction which is 1D vector and the non-bias vector which is 299D vector.

  b. Neutralize: For every word that is not definitional, project to get rid of bias.
    - Babysitter and doctor need to be neutral so we project them on non-bias axis with the direction of the bias:



      - After that they will be equal in the term of gender. - To do this the authors of the paper trained a classifier to tell the words that need to be neutralized or not.

  c. Equalize pairs
    - We want each pair to have difference only in gender. Like:
      - Grandfather - Grandmother - He - She - Boy - Girl
    - We want to do this because the distance between grandfather and babysitter is bigger than babysitter and grandmother:



- To do that, we move grandfather and grandmother to a point where they will be in the middle of the non-bias axis.
- There are some words you need to do this for in your steps. Number of these words is relatively small.

Sequence models & Attention mechanism

> Sequence models can be augmented using an attention mechanism. This algorithm will help your model understand where it should focus its attention given a sequence of inputs. This week, you will also learn about speech recognition and how to deal with audio data.

Various sequence to sequence architectures

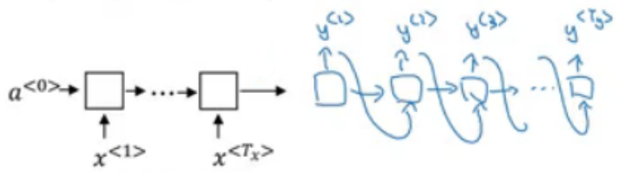
Basic Models

- In this section we will learn about sequence to sequence - *Many to Many* - models which are useful in various applications including machine translation and speech recognition.
- Let's start with the basic model:
  - Given this machine translation problem in which X is a French sequence and Y is an English sequence.

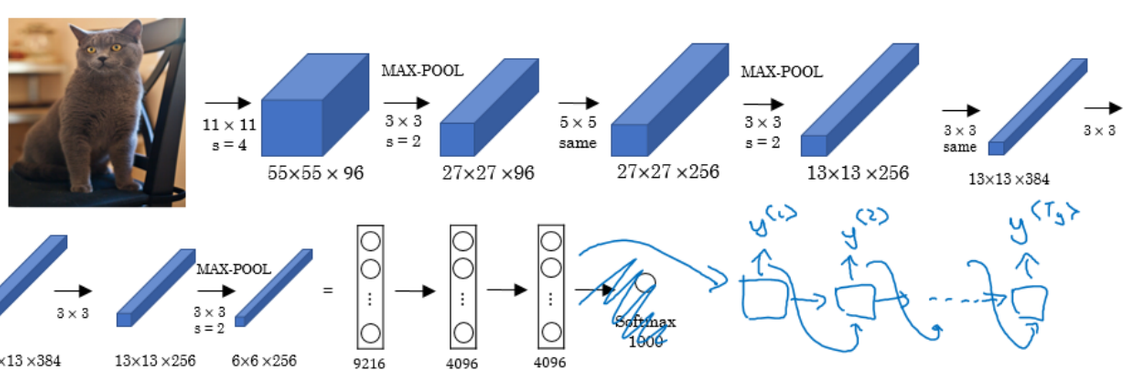
$$\underset{x^{<1>}}{\text{Jane}}\ \underset{x^{<2>}}{\text{visite}}\ \underset{x^{<3>}}{\text{l'Afrique}}\ \underset{x^{<4>}}{\text{en}}\ \underset{x^{<5>}}{\text{septembre}}$$

$$\longrightarrow\ \underset{y^{<1>}}{\text{Jane}}\ \underset{y^{<2>}}{\text{is}}\ \underset{y^{<3>}}{\text{visiting}}\ \underset{y^{<4>}}{\text{Africa}}\ \underset{y^{<5>}}{\text{in}}\ \underset{y^{<6>}}{\text{September.}}$$

  - Our architecture will include **encoder** and **decoder**.
  - The encoder is RNN - LSTM or GRU are included - and takes the input sequence and then outputs a vector that should represent the whole input.
  - After that the decoder network, also RNN, takes the sequence built by the encoder and outputs the new sequence.



  - These ideas are from the following papers:
    - Sutskever et al., 2014. Sequence to sequence learning with neural networks
    - Cho et al., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation
- An architecture similar to the mentioned above works for image captioning problem:
  - In this problem X is an image, while Y is a sentence (caption).
  - The model architecture image:

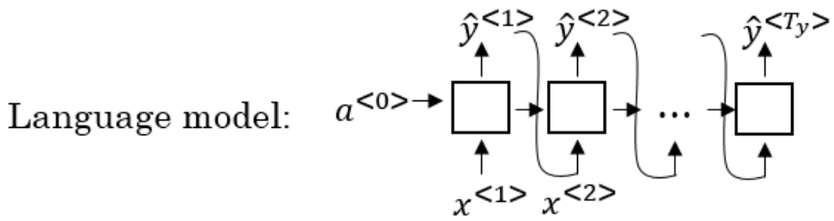


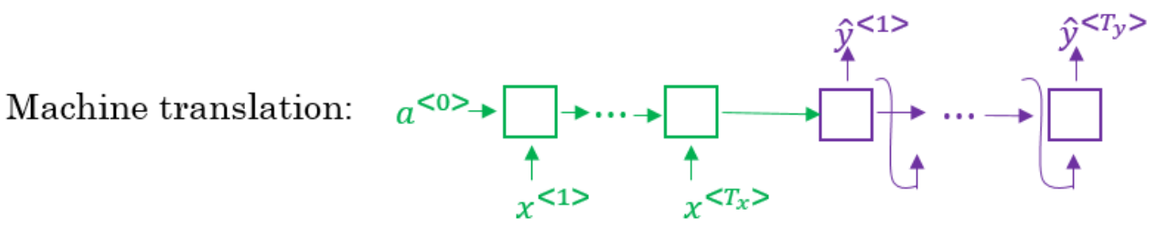
  - The architecture uses a pretrained CNN (like AlexNet) as an encoder for the image, and the decoder is an RNN.

- Ideas are from the following papers (they share similar ideas):
  - Maoet et. al., 2014. Deep captioning with multimodal recurrent neural networks
  - Vinyals et. al., 2014. Show and tell: Neural image caption generator
  - Karpathy and Li, 2015. Deep visual-semantic alignments for generating image descriptions

Picking the most likely sentence

- There are some similarities between the language model we have learned previously, and the machine translation model we have just discussed, but there are some differences as well.
- The language model we have learned is very similar to the decoder part of the machine translation model, except for $a^{<0>}$





- Problems formulations also are different:
  - In language model: $P(y^{<1>}, ..., y^{<Ty>})$
  - In machine translation: $P(y^{<1>}, ..., y^{<Ty>} | x^{<1>}, ..., x^{<Tx>})$
- What we don't want in machine translation model, is not to sample the output at random. This may provide some choices as an output. Sometimes you may sample a bad output.
  - Example:
    - X = "Jane visite l'Afrique en septembre."
    - Y may be:
      - Jane is visiting Africa in September.
      - Jane is going to be visiting Africa in September.
      - In September, Jane will visit Africa.
- So we need to get the best output it can be:

$$\arg\max_{y^{<1>},...,y^{<T_y>}} P(y^{<1>}, ..., y^{<T_y>} | x)$$

- The most common algorithm is the beam search, which we will explain in the next section.
- Why not use greedy search? Why not get the best choices each time?
  - It turns out that this approach doesn't really work!
  - Lets explain it with an example:
    - The best output for the example we talked about is "Jane is visiting Africa in September."
    - Suppose that when you are choosing with greedy approach, the first two words were "Jane is", the word that may come after that will be "going" as "going" is the most common word that comes after " is" so the result may look like this: "Jane is going to be visiting Africa in September.". And that isn't the best/optimal solution.
- So what is better than greedy approach, is to get an approximate solution, that will try to maximize the output (the last equation above).

Beam Search

- Beam search is the most widely used algorithm to get the best output sequence. It's a heuristic search algorithm.
- To illustrate the algorithm we will stick with the example from the previous section. We need Y = "Jane is visiting Africa in September."
- The algorithm has a parameter `B` which is the beam width. Lets take `B = 3` which means the algorithm will get 3 outputs at a time.
- For the first step you will get ["in", "jane", "september"] words that are the best candidates.
- Then for each word in the first output, get B next (second) words and select top best B combinations where the best are those what give the highest value of multiplying both probabilities - $P(y^{<1>}|x) * P(y^{<2>}|x,y^{<1>})$. Se we will have then ["in september", "jane is", "jane visit"]. Notice, that we automatically discard *september* as a first word.
- Repeat the same process and get the best B words for ["september", "is", "visit"] and so on.
- In this algorithm, keep only B instances of your network.
- If `B = 1` this will become the greedy search.

Refinements to Beam Search

- In the previous section, we have discussed the basic beam search. In this section, we will try to do some refinements to it.
- The first thing is **Length optimization**
  - In beam search we are trying to optimize:

$$\arg\max_{y^{<1>},\dots,y^{<T_y>}} P(y^{<1>},\dots,y^{<T_y>}|\,x)$$

  - And to do that we multiply:
    $P(y^{<1>} | x) * P(y^{<2>} | x, y^{<1>}) * ... * P(y^{<t>} | x, y^{<y(t-1)>})$
  - Each probability is a fraction, most of the time a small fraction.
  - Multiplying small fractions will cause a **numerical overflow**. Meaning that it's too small for the floating part representation in your computer to store accurately.
  - So in practice we use **summing logs of probabilities** instead of multiplying directly.

$$\arg\max_{y} \sum_{y=1}^{T_y} \log P(y^{<t>}|x, y^{<1>}, \dots, y^{<t-1>})$$

  - But there's another problem. The two optimization functions we have mentioned are preferring small sequences rather than long ones. Because multiplying more fractions gives a smaller value, so fewer fractions - bigger result.
  - So there's another step - dividing by the number of elements in the sequence.

$$\frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P(y^{<t>}|x, y^{<1>}, \dots, y^{<t-1>})$$

    - alpha is a hyperparameter to tune.
    - If alpha = 0 - no sequence length normalization.
    - If alpha = 1 - full sequence length normalization.
    - In practice alpha = 0.7 is a good thing (somewhere in between two extremes).
- The second thing is how can we choose best `B` ?
  - The larger B - the larger possibilities, the better are the results. But it will be more computationally expensive.
  - In practice, you might see in the production setting `B=10`
  - `B=100` , `B=1000` are uncommon (sometimes used in research settings)
  - Unlike exact search algorithms like BFS (Breadth First Search) or DFS (Depth First Search), Beam Search runs faster but is not guaranteed to find the exact solution.

Error analysis in beam search

- We have talked before on **Error analysis** in "*Structuring Machine Learning Projects*" course. We will apply these concepts to improve our beam search algorithm.
- We will use error analysis to figure out if the `B` hyperparameter of the beam search is the problem (it doesn't get an optimal solution) or in our RNN part.
- Let's take an example:
  - Initial info:
    - x = "Jane visite l'Afrique en septembre."
    - y^* = "Jane visits Africa in September." - right answer
    - $\hat{y}$ = "Jane visited Africa last September." - answer produced by model
  - Our model that has produced not a good result.
  - We now want to know who to blame - the RNN or the beam search.
  - To do that, we calculate $P(y^* | X)$ and $P(\hat{y} | X)$. There are two cases:
    - Case 1 ($P(y^* | X) > P(\hat{y} | X)$):
      - Conclusion: Beam search is at fault.
    - Case 2 ($P(y^* | X) <= P(\hat{y} | X)$):
      - Conclusion: RNN model is at fault.
- The error analysis process is as following:

- You choose N error examples and make the following table:

Human	Algorithm	$P(y^*\|x)$	$P(\hat{y}\|x)$	At fault?
Jane visits Africa in September.	Jane visited Africa last September.	2×10^{-10}	1×10^{-10}	B R B R R ...

  - `B` for beam search, `R` is for the RNN.
  - Get counts and decide what to work on next.

BLEU Score

- One of the challenges of machine translation, is that given a sentence in a language there are one or more possible good translation in another language. So how do we evaluate our results?
- The way we do this is by using **BLEU score**. BLEU stands for *bilingual evaluation understudy*.
- The intuition is: as long as the machine-generated translation is pretty close to any of the references provided by humans, then it will get a high BLEU score.
- Let's take an example:
  - X = "Le chat est sur le tapis."
  - Y1 = "The cat is on the mat." (human reference 1)
  - Y2 = "There is a cat on the mat." (human reference 2)
  - Suppose that the machine outputs: "the the the the the the the."
  - One way to evaluate the machine output is to look at each word in the output and check if it is in the references. This is called *precision*:
    - precision = 7/7 because "the" appeared in Y1 or Y2
  - This is not a useful measure!
  - We can use a modified precision in which we are looking for the reference with the maximum number of a particular word and set the maximum appearing of this word to this number. So:
    - modified precision = 2/7 because the max is 2 in Y1
    - We clipped the 7 times by the max which is 2.
  - Here we are looking at one word at a time - unigrams, we may look at n-grams too
- BLEU score on bigrams
  - The **n-grams** typically are collected from a text or speech corpus. When the items are words, **n-grams** may also be called shingles. An **n-gram** of size 1 is referred to as a "unigram"; size 2 is a "bigram" (or, less commonly, a "digram"); size 3 is a "trigram".
  - X = "Le chat est sur le tapis."
  - Y1 = "The cat is on the mat."
  - Y2 = "There is a cat on the mat."
  - Suppose that the machine outputs: "the cat the cat on the mat."
  - The bigrams in the machine output:

Pairs	Count	Count clip
the cat	2	1 (Y1)
cat the	1	0
cat on	1	1 (Y2)
on the	1	1 (Y1)
the mat	1	1 (Y1)
Totals	6	4

  Modified precision = sum(Count clip) / sum(Count) = 4/6

- So here are the equations for modified precision for the n-grams case:

$$p_1 = \frac{\sum_{unigram \in \hat{y}} count_{clip}\,(unigram)}{\sum_{unigram \in \hat{y}} count\,(unigram)} \qquad p_n = \frac{\sum_{ngram \in \hat{y}} count_{clip}\,(ngram)}{\sum_{ngram \in \hat{y}} count\,(ngram)}$$

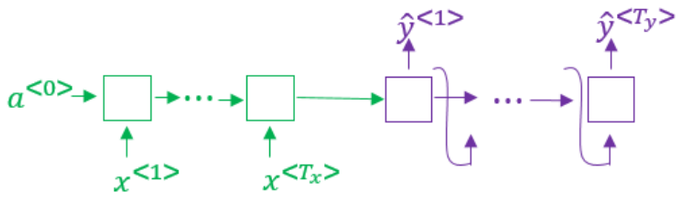
- Let's put this together to formalize the BLEU score:
  - P_n = Bleu score on one type of n-gram
  - **Combined BLEU score** = BP * exp(1/n * sum(P_n))
    - For example if we want BLEU for 4, we compute P_1, P_2, P_3, P_4 and then average them and take the exp.
  - **BP** is called **BP penalty** which stands for brevity penalty. It turns out that if a machine outputs a small number of words it will get a better score so we need to handle that.

$$BP = \begin{cases} 1 & \text{if MT\_output\_length} > \text{reference\_output\_length} \\ \exp(1 - \text{MT\_output\_length}/\text{reference\_output\_length}) & \text{otherwise} \end{cases}$$

- BLEU score has several open source implementations.
- It is used in a variety of systems like machine translation and image captioning.

Attention Model Intuition

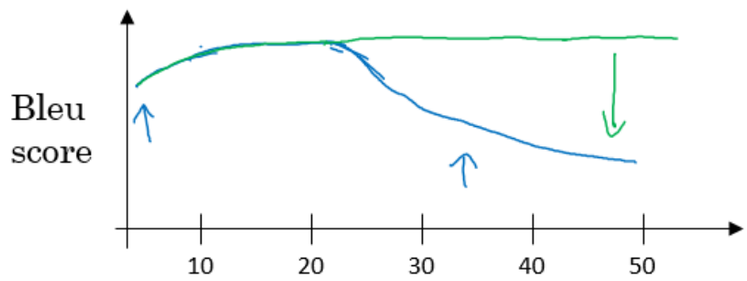
- So far we were using sequence to sequence models with an encoder and decoders. There is a technique called *attention* which makes these models even better.
- The attention idea has been one of the most influential ideas in deep learning.
- The problem of long sequences:
  - Given this model, inputs, and outputs.



Jane s'est rendue en Afrique en septembre dernier, a apprécié la culture et a rencontré beaucoup de gens merveilleux; elle est revenue en parlant comment son voyage était merveilleux, et elle me tente d'y aller aussi.

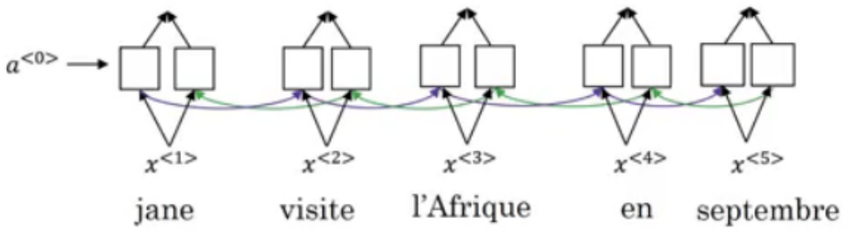
Jane went to Africa last September, and enjoyed the culture and met many wonderful people; she came back raving about how wonderful her trip was, and is tempting me to go too.

  - The encoder should memorize this long sequence into one vector, and the decoder has to process this vector to generate the translation.
  - If a human would translate this sentence, he/she wouldn't read the whole sentence and memorize it then try to translate it. He/she translates a part at a time.
  - The performance of this model decreases if a sentence is long.
  - We will discuss the attention model that works like a human that looks at parts at a time. That will significantly increase the accuracy even with longer sequence:

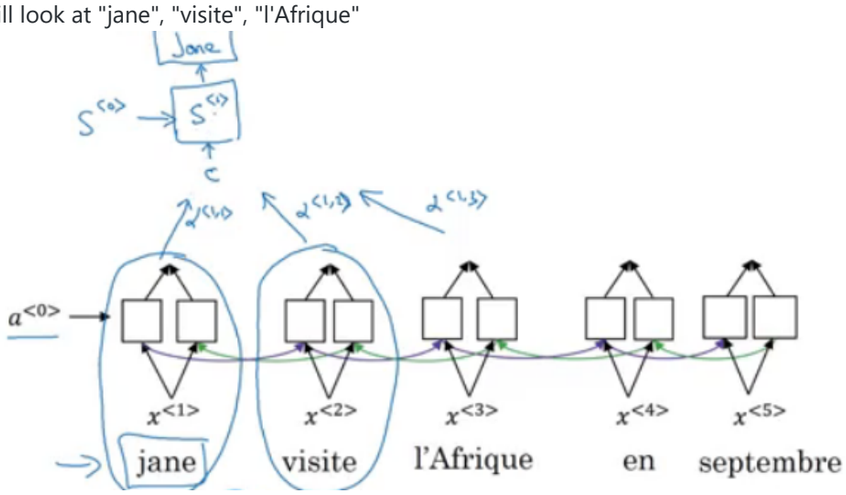


    - Blue is the normal model, while green is the model with attention mechanism.
- In this section we will give just some intuitions about the attention model and in the next section we will discuss it's details.
- At first the attention model was developed for machine translation but then other applications used it like computer vision and new architectures like Neural Turing machine.
- The attention model was descried in this paper:
  - Bahdanau et. al., 2014. Neural machine translation by jointly learning to align and translate
- Now for the intuition:

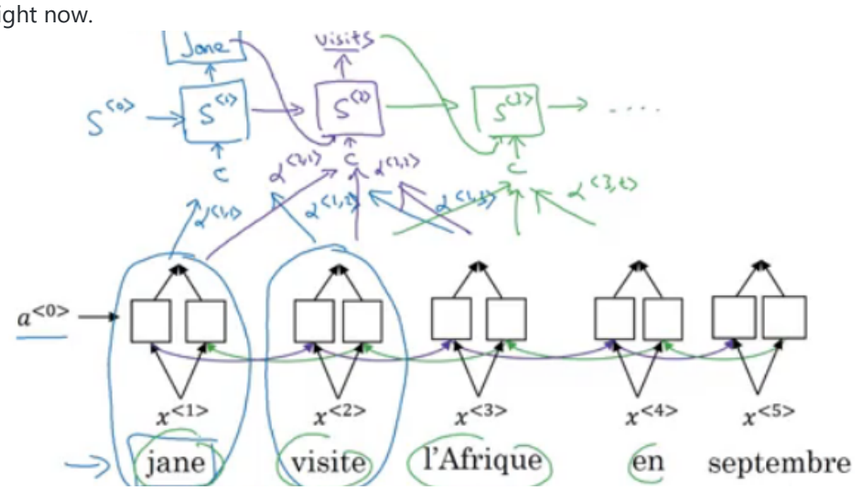
- Suppose that our encoder is a bidirectional RNN:



- We give the French sentence to the encoder and it should generate a vector that represents the inputs.
- Now to generate the first word in English which is "Jane" we will make another RNN which is the decoder.
- Attention weights are used to specify which words are needed when to generate a word. So to generate "jane" we will look at "jane", "visite", "l'Afrique"

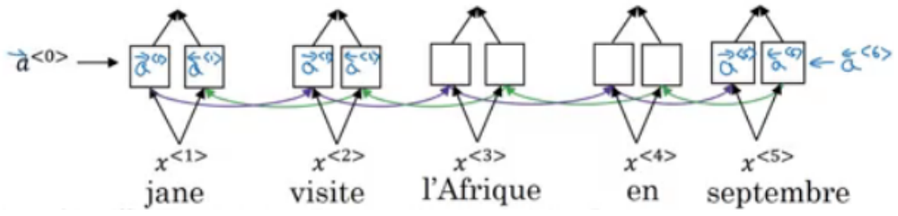


- alpha$^{<1,1>}$, alpha$^{<1,2>}$, and alpha$^{<1,3>}$ are the attention weights being used.
- And so to generate any word there will be a set of attention weights that controls which words we are looking at right now.

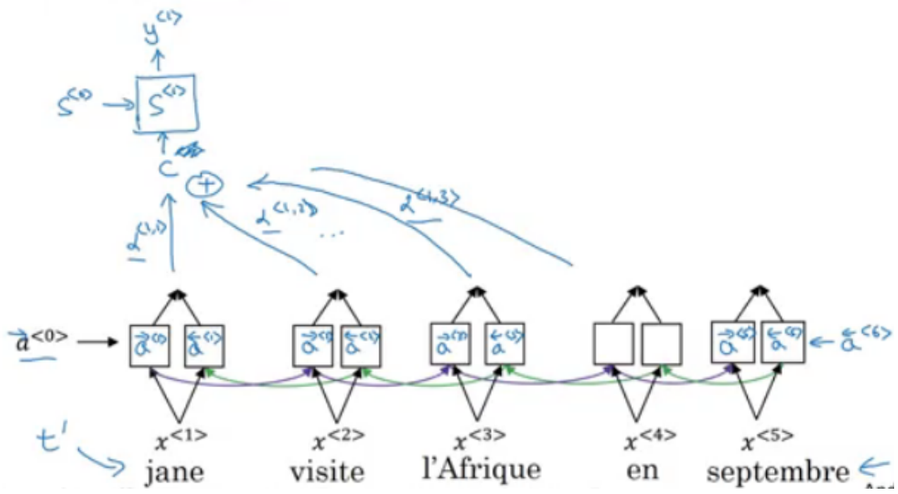


Attention Model

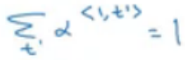
- Lets formalize the intuition from the last section into the exact details on how this can be implemented.
- First we will have an bidirectional RNN (most common is LSTMs) that encodes French language:



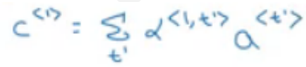
- For learning purposes, lets assume that a$^{<t'>}$ will include the both directions activations at time step t'.
- We will have a unidirectional RNN to produce the output using a context `c` which is computed using the attention weights, which denote how much information does the output needs to look in a$^{<t'>}$



- Sum of the attention weights for each element in the sequence should be 1:



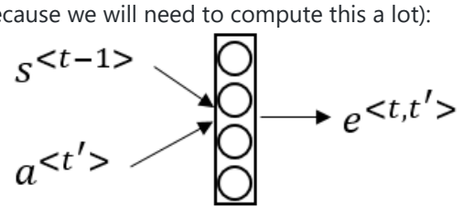
- The context `c` is calculated using this equation:



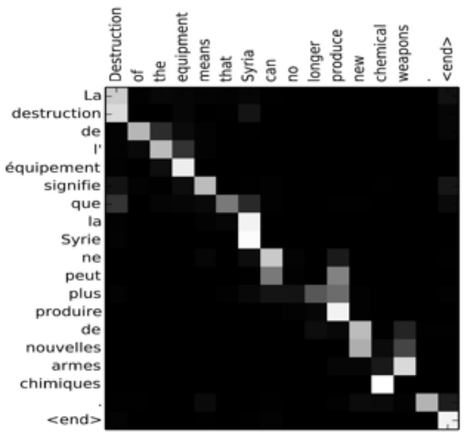
- Lets see how can we compute the attention weights:
  - So alpha$^{<t, t'>}$ = amount of attention y$^{<t>}$ should pay to a$^{<t'>}$
    - Like for example we payed attention to the first three words through alpha$^{<1,1>}$, alpha$^{<1,2>}$, alpha$^{<1,3>}$
  - We are going to softmax the attention weights so that their sum is 1:

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$$

  - Now we need to know how to calculate e$^{<t, t'>}$. We will compute e using a small neural network (usually 1-layer, because we will need to compute this a lot):



    - s$^{<t-1>}$ is the hidden state of the RNN s, and a$^{<t'>}$ is the activation of the other bidirectional RNN.
- One of the disadvantages of this algorithm is that it takes quadratic time or quadratic cost to run.
- One fun way to see how attention works is by visualizing the attention weights:

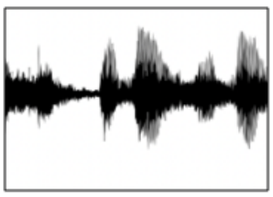


Speech recognition - Audio data

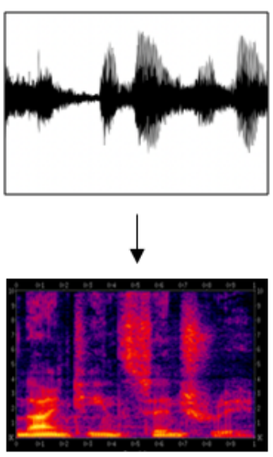
Speech recognition

- One of the most exciting developments using sequence-to-sequence models has been the rise of very accurate speech recognition.
- Let's define the speech recognition problem:
  - X: audio clip
  - Y: transcript

- If you plot an audio clip it will look like this:

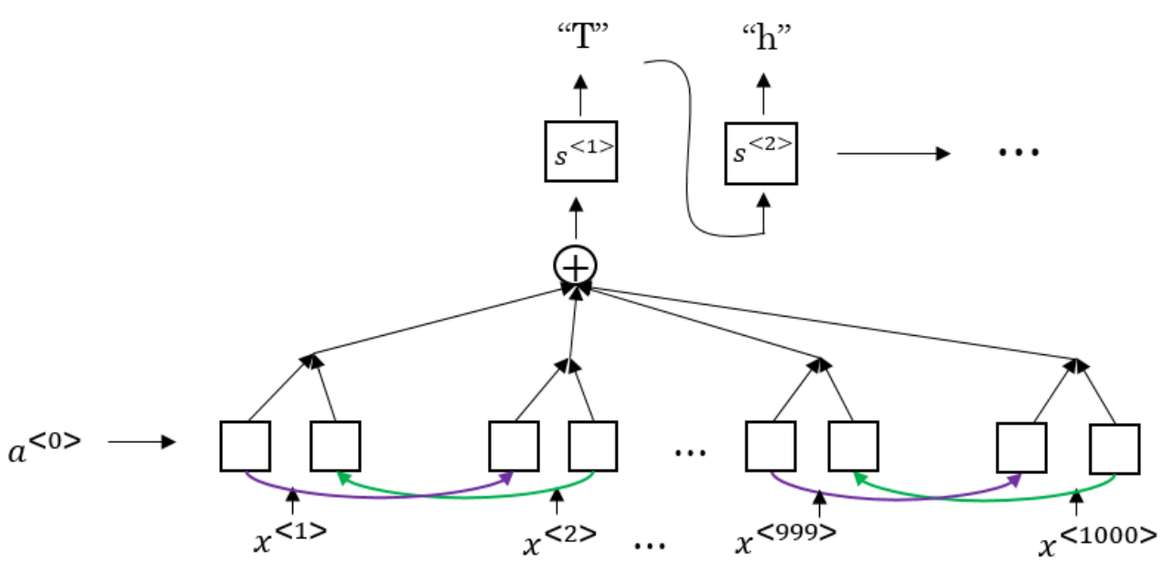


  - The horizontal axis is time while the vertical is changes in air pressure.
- What really is an audio recording? A microphone records little variations in air pressure over time, and it is these little variations in air pressure that your ear perceives as sound. You can think of an audio recording is a long list of numbers measuring the little air pressure changes detected by the microphone. We will use audio sampled at 44100 Hz (or 44100 Hertz). This means the microphone gives us 44100 numbers per second. Thus, a 10 second audio clip is represented by 441000 numbers (= 10 * 44100).
- It is quite difficult to work with "raw" representation of audio.
- Because even human ear doesn't process raw wave forms, the human ear can process different frequencies.
- There's a common preprocessing step for an audio - generate a spectrogram which works similarly to human ears.



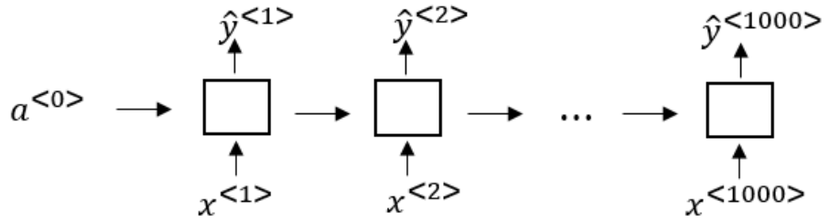
  - The horizontal axis is time while the vertical is frequencies. Intensity of different colors shows the amount of energy - how loud is the sound for different frequencies (a human ear does a very similar preprocessing step).
- A spectrogram is computed by sliding a window over the raw audio signal, and calculates the most active frequencies in each window using a Fourier transformation.
- In the past days, speech recognition systems were built using *phonemes* that are a hand engineered basic units of sound. Linguists used to hypothesize that writing down audio in terms of these basic units of sound called *phonemes* would be the best way to do speech recognition.
- End-to-end deep learning found that phonemes was no longer needed. One of the things that made this possible is the large audio datasets.
- Research papers have around 300 - 3000 hours of training data while the best commercial systems are now trained on over 100,000 hours of audio.

- You can build an accurate speech recognition system using the attention model that we have descried in the previous section:



- One of the methods that seem to work well is *CTC cost* which stands for "Connectionist temporal classification"
  - To explain this let's say that Y = "the quick brown fox"

- We are going to use an RNN with input, output structure:



  - Note: this is a unidirectional RNN, but in practice a bidirectional RNN is used.
  - Notice, that the number of inputs and number of outputs are the same here, but in speech recognition problem input X tends to be a lot larger than output Y.
    - 10 seconds of audio at 100Hz gives us X with shape (1000,). These 10 seconds don't contain 1000 character outputs.
  - The CTC cost function allows the RNN to output something like this:
    - `ttt_h_eee<SPC>___<SPC>qqq___` - this covers "the q".
    - The _ is a special character called "blank" and `<SPC>` is for the "space" character.
    - Basic rule for CTC: collapse repeated characters not separated by "blank"
  - So the 19 character in our Y can be generated into 1000 character output using CTC and it's special blanks.
  - The ideas were taken from this paper:
    - Graves et al., 2006. Connectionist Temporal Classification: Labeling unsegmented sequence data with recurrent neural networks
    - This paper's ideas were also used by Baidu's DeepSpeech.
- Using both attention model and CTC cost can help you to build an accurate speech recognition system.

Trigger Word Detection

- With the rise of deep learning speech recognition, there are a lot of devices that can be waked up by saying some words with your voice. These systems are called trigger word detection systems.
- For example, Alexa - a smart device made by Amazon - can answer your call "Alexa, what time is it?" and then Alexa will respond to you.
- Trigger word detection systems include:

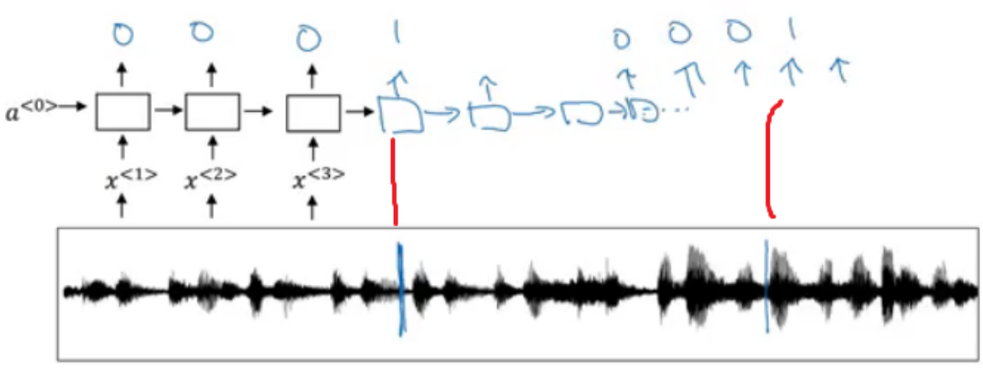
Amazon Echo (Alexa)

Baidu DuerOS (xiaodunihao)

Apple Siri (Hey Siri)

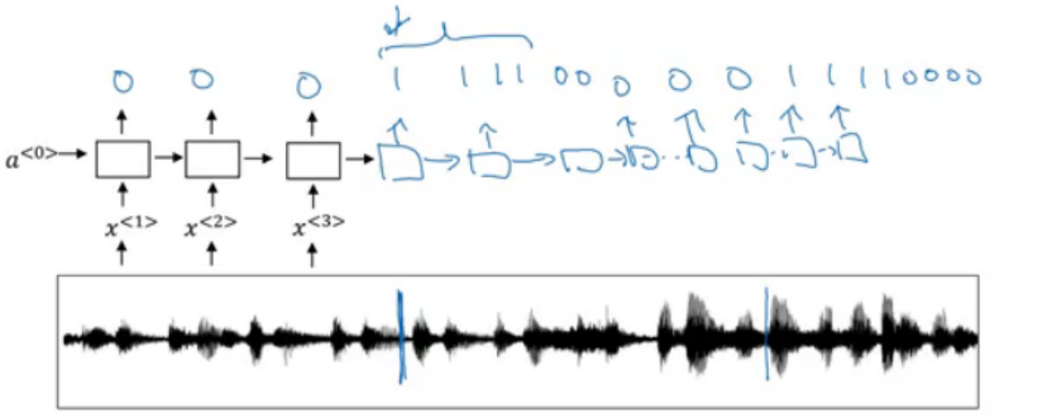
Google Home (Okay Google)

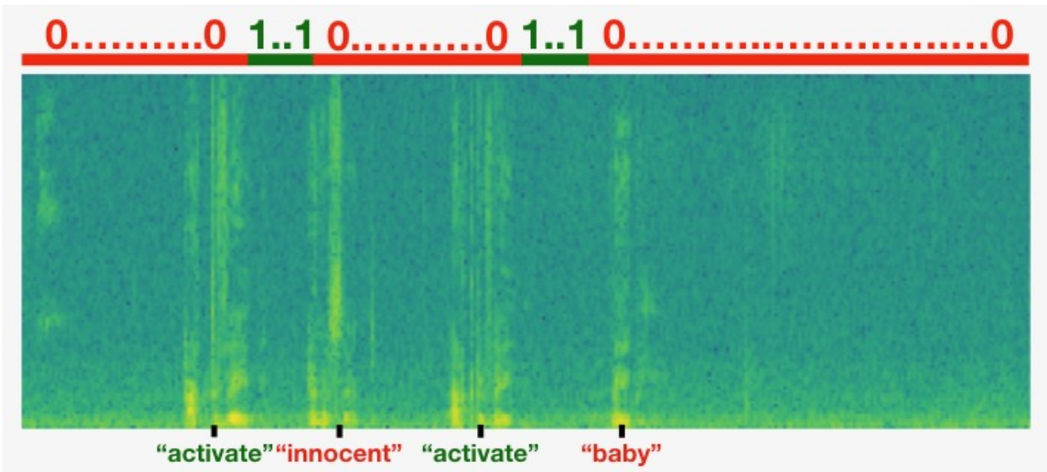
- For now, the trigger word detection literature is still evolving so there actually isn't a single universally agreed on the algorithm for trigger word detection yet. But let's discuss an algorithm that can be used.
- Let's now build a model that can solve this problem:
  - X: audio clip
  - X has been preprocessed and spectrogram features have been returned of X
    - $X^{<1>}, X^{<2>}, \dots, X^{<t>}$
  - Y will be labels 0 or 1. 0 represents the non-trigger word, while 1 is that trigger word that we need to detect.
  - The model architecture can be like this:



    - The vertical lines in the audio clip represent moment just after the trigger word. The corresponding to this will be 1.

- One disadvantage of this creates a very imbalanced training set. There will be a lot of zeros and few ones.
- A hack to solve this is to make an output a few ones for several times or for a fixed period of time before reverting back to zero.

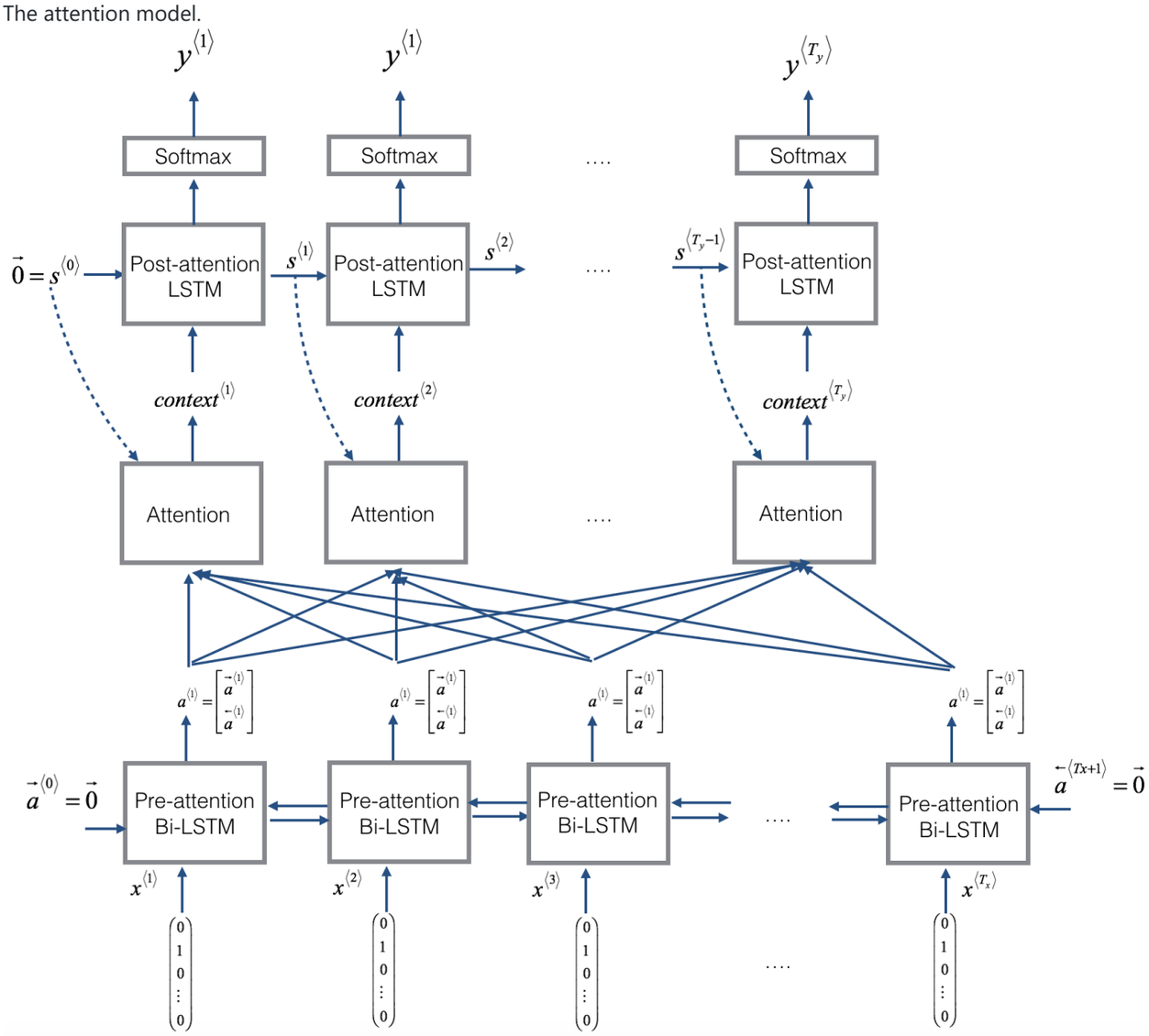




Extras

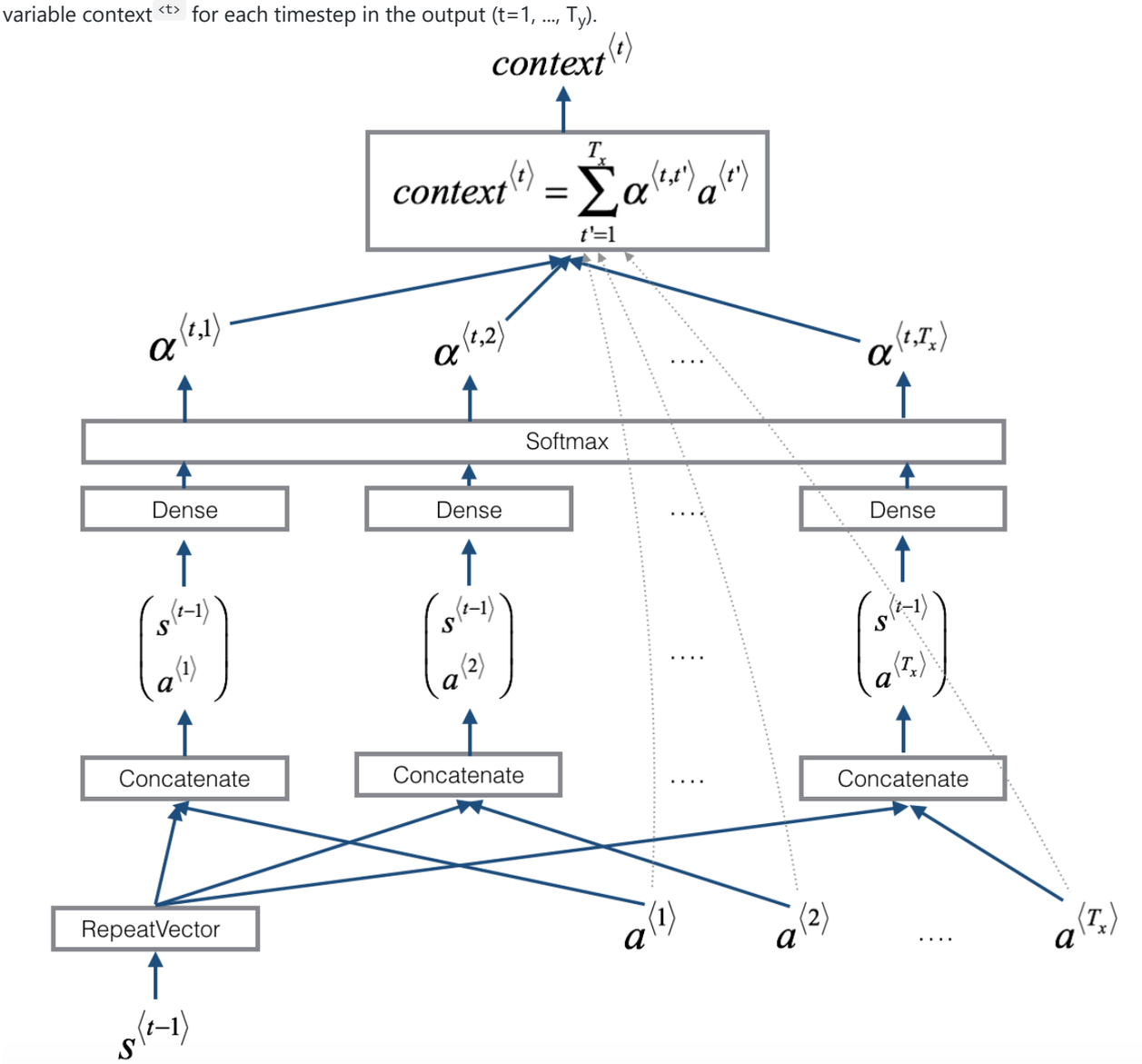
Machine translation attention model (from notebooks)

- The model is built with keras layers.
- The attention model.



- There are two separate LSTMs in this model. Because the one at the bottom of the picture is a Bi-directional LSTM and comes *before* the attention mechanism, we will call it *pre-attention* Bi-LSTM. The LSTM at the top of the diagram comes *after* the attention mechanism, so we will call it the *post-attention* LSTM. The pre-attention Bi-LSTM goes through T_x time steps; the post-attention LSTM goes through T_y time steps.
- The post-attention LSTM passes $s^{<t>}$, $c^{<t>}$ from one time step to the next. In the lecture videos, we were using only a basic RNN for the post-activation sequence model, so the state captured by the RNN output activations $s^{<t>}$. But since we are using an LSTM here, the LSTM has both the output activation $s^{<t>}$ and the hidden cell state $c^{<t>}$. However, unlike previous text generation examples (such as Dinosaurus in week 1), in this model the post-activation LSTM at time `t` does will not take the specific generated $y^{<t-1>}$ as input; it only takes $s^{<t>}$ and $c^{<t>}$ as input. We have designed the model this way, because (unlike language generation where adjacent characters are highly correlated) there isn't as strong a dependency between the previous character and the next character in a YYYY-MM-DD date.

- What one "Attention" step does to calculate the attention variables $\alpha^{<t,\ t>}$, which are used to compute the context variable context $^{<t>}$ for each timestep in the output (t=1, ..., T_y).



- The diagram uses a `RepeatVector` node to copy $s^{<t-1>}$'s value T_x times, and then `Concatenation` to concatenate $s^{<t-1>}$ and $a^{<t>}$ to compute $e^{<t,\ t>}$, which is then passed through a softmax to compute $\alpha^{<t,\ t>}$.

These Notes were made by Mahmoud Badry @2018

Follow me on LinkedIn for more:
Steve Nouri
https://www.linkedin.com/in/stevenouri/