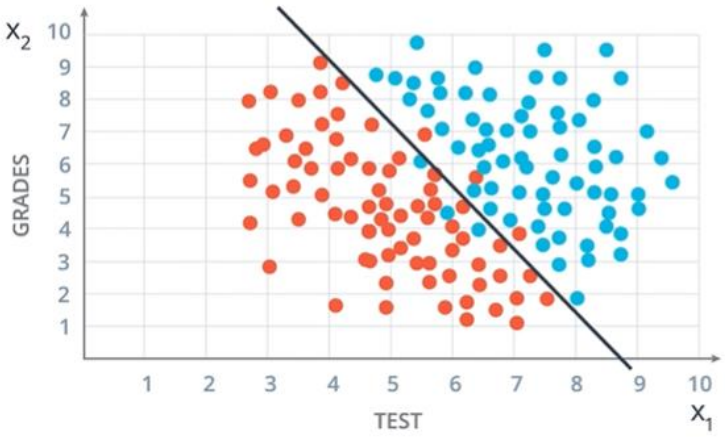
Acceptance at a University

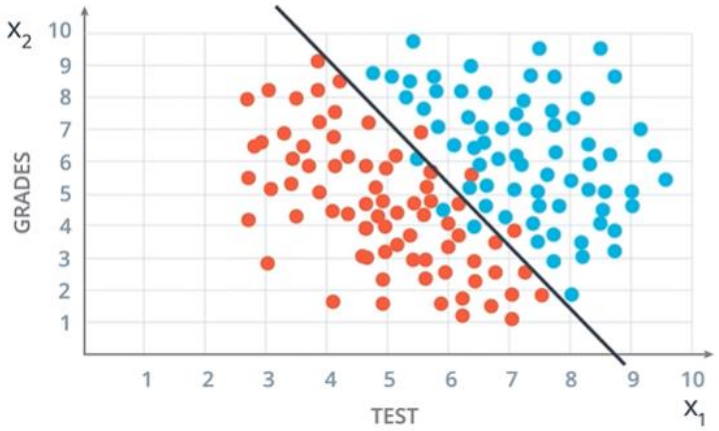


BOUNDARY:
A LINE
$2x_1 + x_2 - 18 = 0$

Score =
2*Test + Grades - 18

PREDICTION:
Score > 0: Accept
Score < 0: Reject

Acceptance at a University

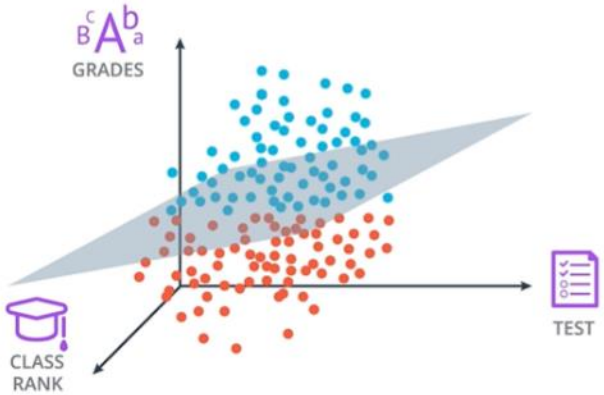


BOUNDARY:
A LINE
$w_1x_1 + w_2x_2 + b = 0$
$Wx + b = 0$
$W = (w_1, w_2)$
$x = (x_1, x_2)$
y = label: 0 or 1

PREDICTION:

$$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$$

Acceptance at a University



BOUNDARY:
A PLANE
$w_1x_1 + w_2x_2 + w_3x_3 + b = 0$
$Wx + b = 0$

PREDICTION:

$$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$$

Acceptance at a University

	x_1	x_2	x_3		x_n	y
	EXAM 1	EXAM 2	GRADES	...	ESSAY	PASS?
STUDENT 1	9	6	5	...	6	1(yes)
STUDENT 2	8	4	8	...	3	0(no)
...	...	...	...	...	...	
STUDENT n	6	7	2	...	8	1(yes)

n columns

n-dimensional space
$x_1, x_2, ..., x_n$

BOUNDARY:
n-1 dimensional hyperplane
$w_1x_1 + w_2x_2 + w_nx_n + b = 0$
$Wx + b = 0$

PREDICTION:

$$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$$

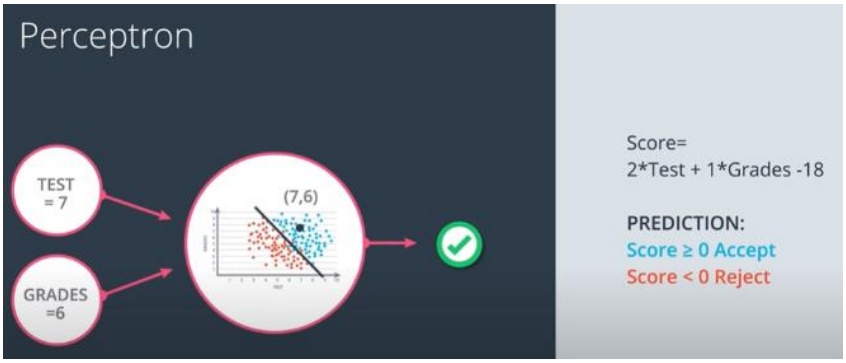
zero and y head equals zero if Wx plus b is less than zero.

QUIZ QUESTION

Given the table in the video above, what would the dimensions be for input features (x), the weights (W), and the bias (b) to satisfy (Wx + b)?

- W: (nx1), x: (1xn), b: (1x1)
- W: (1xn), x: (1xn), b: (nx1)
- ✓ W: (1xn), x: (nx1), b: (1x1)
- W: (1xn), x: (nx1), b: (1xn)

Perceptron



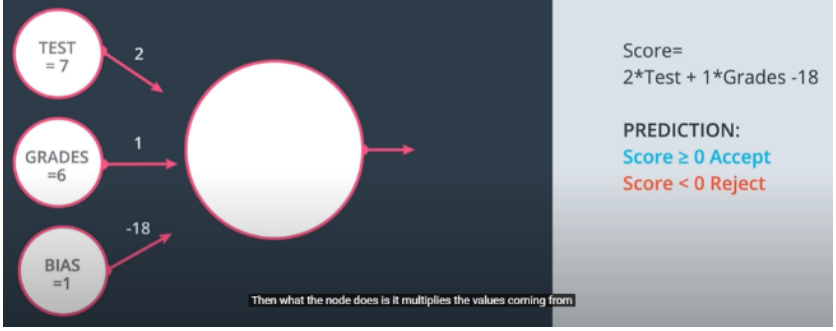
Score=
2*Test + 1*Grades -18

PREDICTION:
Score ≥ 0 Accept
Score < 0 Reject

Score=
2*Test + 1*Grades -18

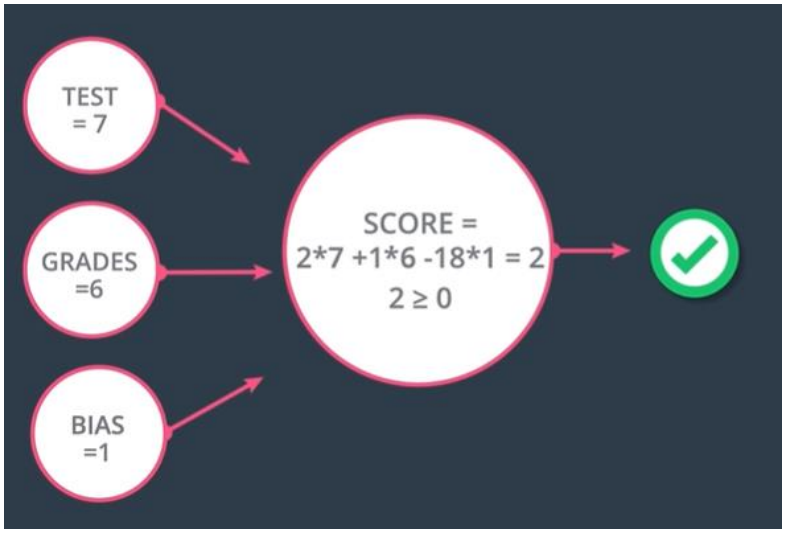
PREDICTION:
Score ≥ 0 Accept
Score < 0 Reject

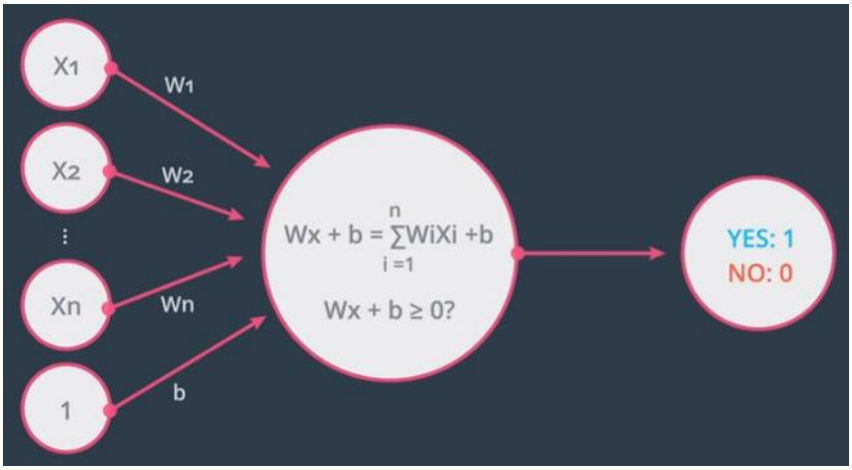
Perceptron

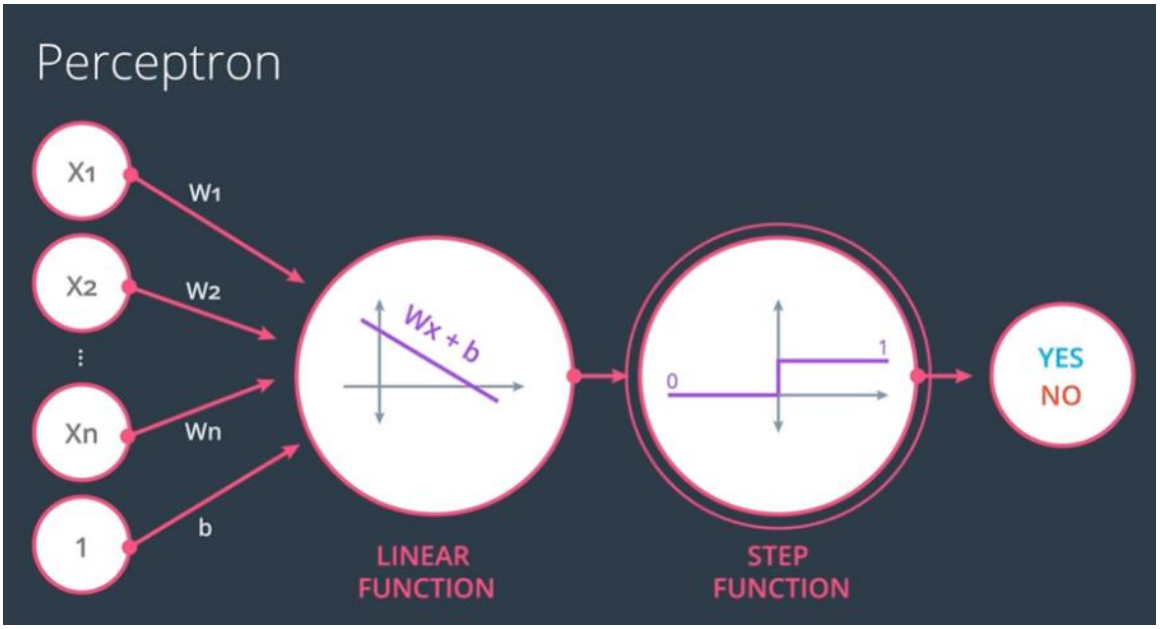


Score=
2*Test + 1*Grades -18

PREDICTION:
Score ≥ 0 Accept
Score < 0 Reject



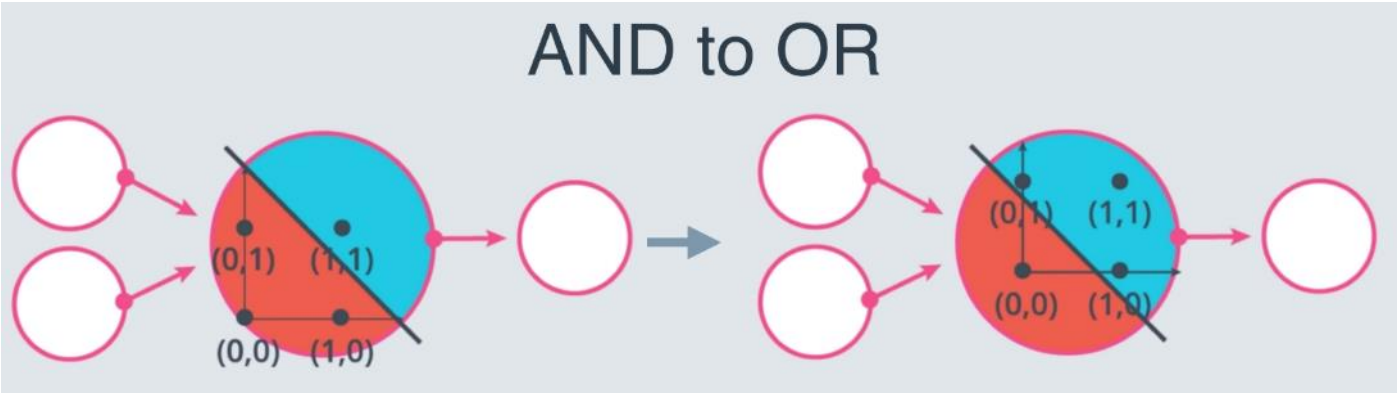




These weights start out as random values, and as the neural network network learns more about what kind of input data leads to a student being accepted into a university, the network adjusts the weights based on any errors in categorization that results from the previous weights. This is called **training** the neural network.

A higher weight means the neural network considers that input more important than other inputs, and lower weight means that the data is considered less important

For the image above, let's say that the weights are: $w_{grades} = -1, w_{test} = -0.2$. You don't have to be concerned with the actual values, but their relative values are important. w_{grades} is 5 times larger than w_{test}, which means the neural network considers `grades` input 5 times more important than `test` in determining whether a student will be accepted into a university.



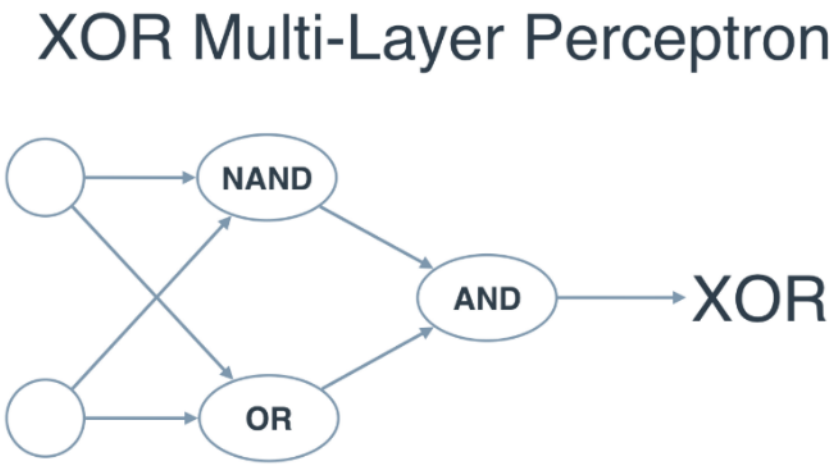
AND:

w1 = 1, w2 = 1, b > 1

OR:

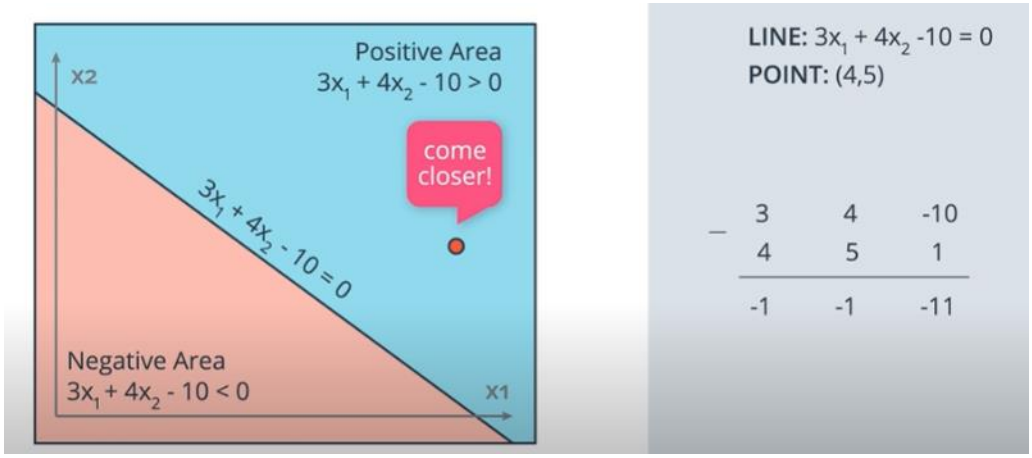
w1 = 1, w2 = 1, b < 1 or w1 > 1, w2 > 1, b = 1

And if we introduce the **NAND** operator as the combination of **AND** and **NOT**, then we get the following two-layer perceptron that will model **XOR**. That's our first neural network!

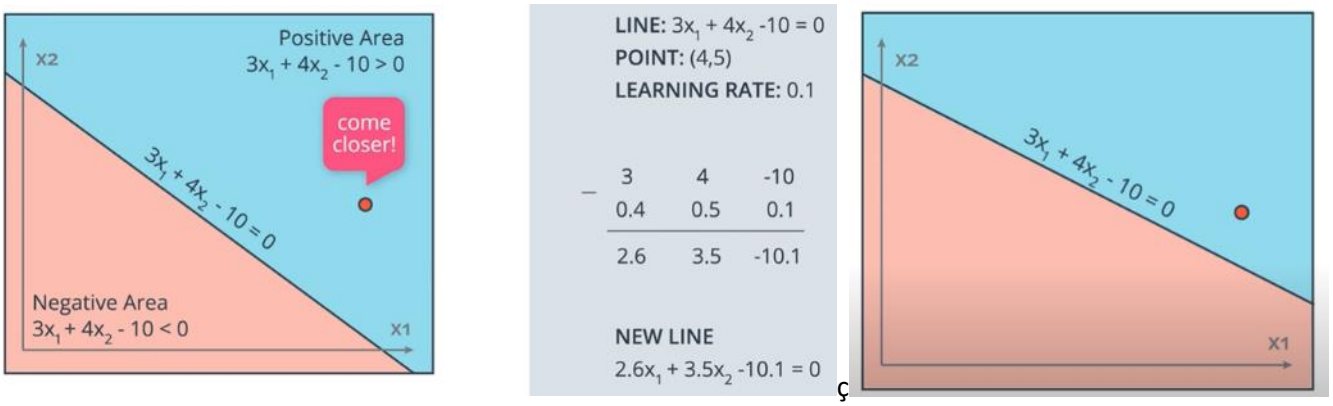


If a point is misclassified, it is interesting that the line comes closer, so it can be classified correctly.

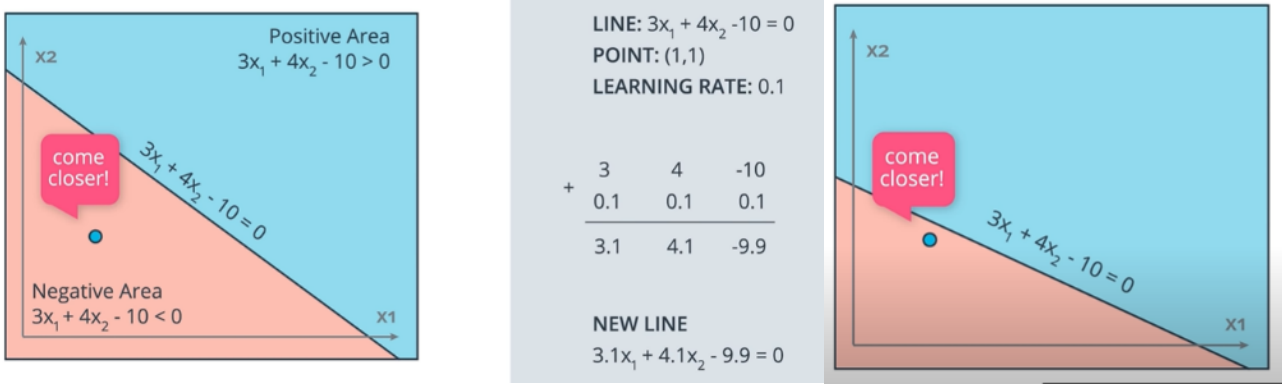
Subtracting line from the incorrect point with b = 1 because it is in the right side of the line



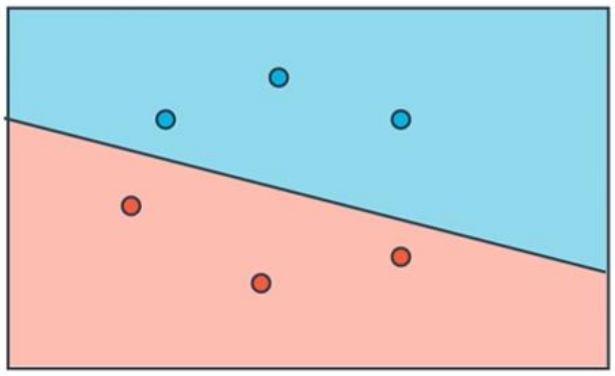
New line would possibly classify it correctly but this is not clever since it would misclassify other points with this drastic approach → so small steps (learning rate)



ç



Perceptron Algorithm



1. Start with random weights: $w_1, \ldots, w_n, b$

2. For every misclassified point $(x_1,\ldots,x_n)$:

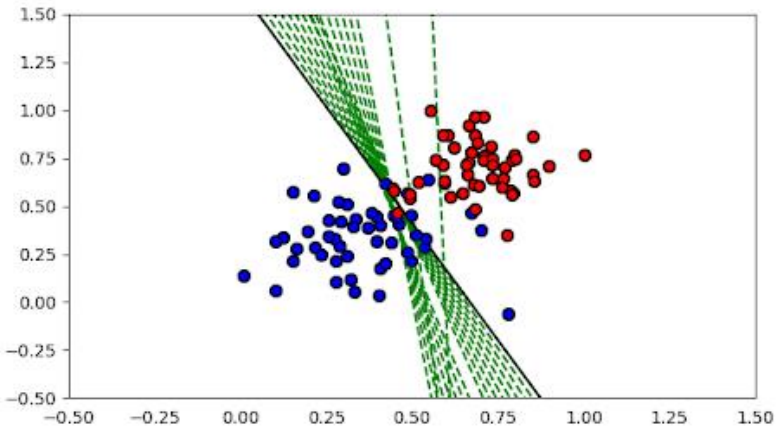
   2.1. If prediction = 0:

   - For i = 1 ...n
     - Change $w_i + \alpha x_i$
   - Change b to $b + \alpha$

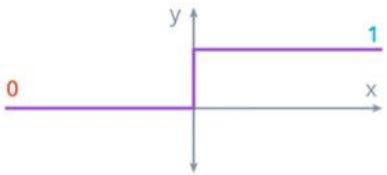
   2.2. If prediction = 1:

   - For i = 1 ...n
     - Change $w_i - \alpha x_i$
   - Change b to $b - \alpha$

```python
import numpy as np
# Setting the random seed, feel free to change it and see different solutions.
np.random.seed(42)

def stepFunction(t):
    if t >= 0:
        return 1
    return 0

def prediction(X, W, b):
    return stepFunction((np.matmul(X,W)+b)[0])

# TODO: Fill in the code below to implement the perceptron trick.
# The function should receive as inputs the data X, the labels y,
# the weights W (as an array), and the bias b,
# update the weights and bias W, b, according to the perceptron algorithm,
# and return W and b.
def perceptronStep(X, y, W, b, learn_rate = 0.01):
    for i in range(len(X)):
        y_hat = prediction(X[i], W, b)
        if y[i]-y_hat == 1:
            W[0] += X[i][0]*learn_rate
            W[1] += X[i][1]*learn_rate
            b += learn_rate
        elif y[i]-y_hat == -1:
            W[0] -= X[i][0]*learn_rate
            W[1] -= X[i][1]*learn_rate
            b -= learn_rate
    # Fill in code
    return W, b
```

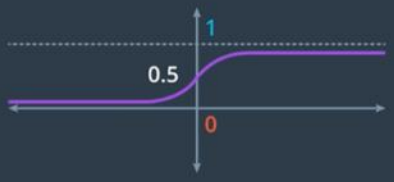


Activation Functions



DISCRETE:
Step Function

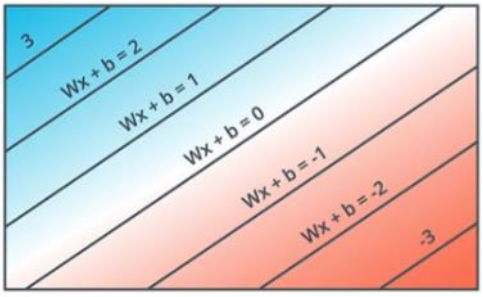
$$y = \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$$



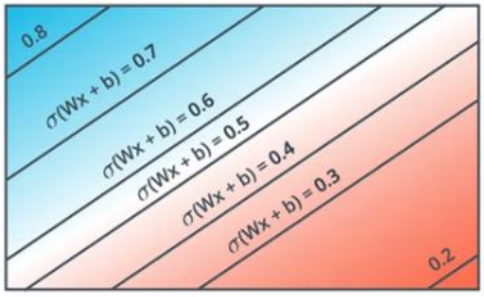
CONTINUOUS:
Sigmoid Function

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

Predictions



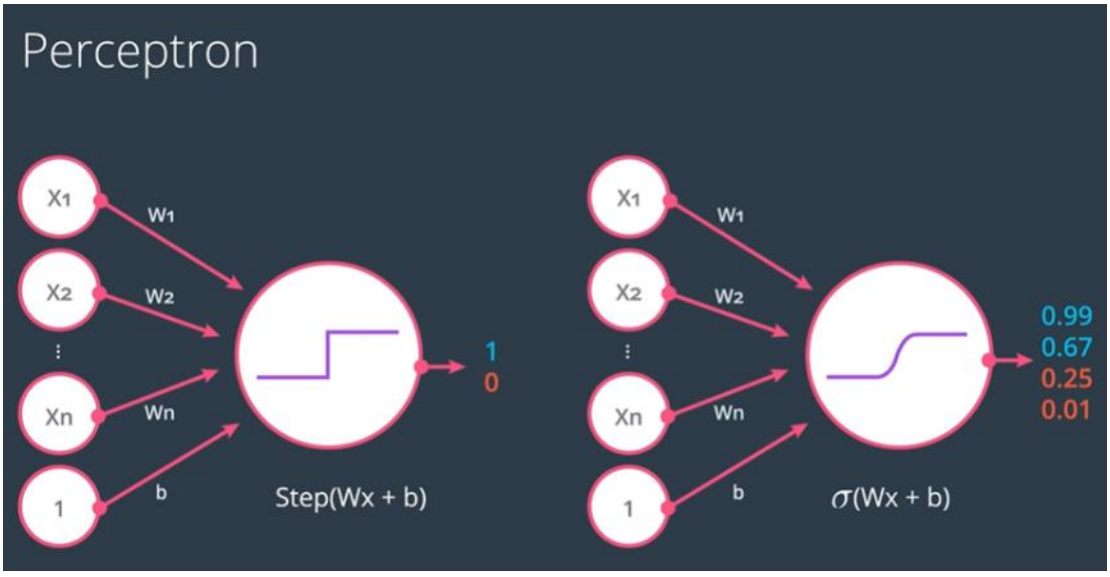
σ



$Wx + b$

$\hat{y} = \sigma(Wx + b)$

Perceptron



QUIZ QUESTION

The sigmoid function is defined as sigmoid(x) = $1/(1+e^{-x})$. If the score is defined by $4x_1 + 5x_2 - 9$ = score, then which of the following points has exactly a 50% probability of being blue or red? (Choose all that are correct.)

- [x] (1, 1)
- [ ] (2, 4)
- [ ] (5, -5)
- [x] (-4, 5)

If score = 0 → Sigmoid returns 0.5 of probability

Softmax function:

Classification Problem

2 $\quad \frac{2}{2+1+0} \quad \frac{e^2}{e^2+e^1+e^0} = 0.67$ P(duck)

1 $\quad \frac{1}{2+1+0} \quad \frac{e^1}{e^2+e^1+e^0} = 0.24$ P(beaver)

0 $\quad \frac{0}{2+1+0} \quad \frac{e^0}{e^2+e^1+e^0} = 0.09$ P(walrus)

Softmax Function

LINEAR FUNCTION SCORES:
$Z_1, \ldots, Z_n$

$$P(\text{class } i) = \frac{e^{Z_i}}{e^{Z_1} + \ldots + e^{Z_n}}$$

QUESTION

Is Softmax for n=2 values the same as the sigmoid function?

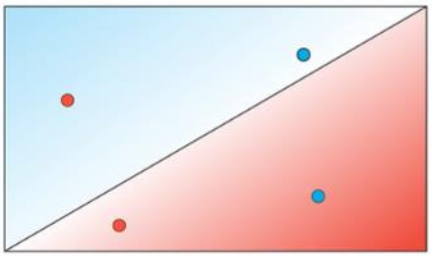
```python
import numpy as np

# Write a function that takes as input a list of numbers, and returns
# the list of values given by the softmax function.
def softmax(L):
    expL = np.exp(L)
    sumExpL = sum(expL)
    sm_values = []
    for i in expL:
        sm_values.append(i * 1.0 / sumExpL)
    return sm_values
```

One-Hot Encoding

ANIMAL	VALUE
	?
	?
	?
	?
	?

ANIMAL	DUCK?	BEAVER?	WALRUS?
	1	0	0
	0	1	0
	1	0	0
	0	0	1
	0	1	0

Probability



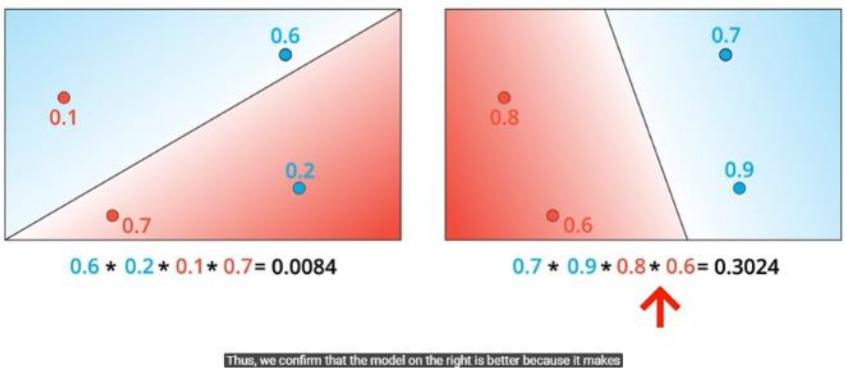
$\hat{y} = \sigma(Wx+b)$

$P(\text{blue}) = \sigma(Wx+b)$

P(red) = 0.1
P(blue) = 0.6
P(red) = 0.7
x P(blue) = 0.2

P(all) = 0.0084

Probability



Maximum likelihood maximizes the total probability

QUIZ QUESTION

Based on the above video, which of the following is true for a very high value for P(all)?

- The model classifies most blue points correctly.
- The model classifies most red points correctly.
- ✓ The model classifies most points correctly with P(all) indicating how accurate the model is.
- The model classifies all points correctly.

Products

0.6 * 0.2 * 0.1 * 0.7 = 0.0084

ln(0.6) + ln(0.2) + ln(0.1) + ln(0.7)

-0.51 -1.61 -2.3 -0.36

-ln(0.6) - ln(0.2) - ln(0.1) - ln(0.7) = 4.8

0.51 1.61 2.3 0.36

0.7 * 0.9 * 0.8 * 0.6 = 0.3024

ln(0.7) + ln(0.9) + ln(0.8) + ln(0.6)

-0.36 -0.1 -.22 -0.51

-ln(0.7) - ln(0.9) -ln(0.8) -ln(0.6) = 1.2

0.36 0.1 .22 0.51

The smaller values, the better. Minimize Cross Entropy

			Probability	-ln(Probability)
0.8	0.7	0.1	0.056	2.88
0.8	0.7	✗ 0.9	0.504	0.69
0.8	✗ 0.3	0.1	0.024	3.73
✗ 0.2	0.7	0.1	0.014	4.27
0.8	✗ 0.3	✗ 0.9	0.216	1.53
✗ 0.2	0.7	✗ 0.9	0.126	2.07
✗ 0.2	✗ 0.3	0.1	0.006	5.12
✗ 0.2	✗ 0.3	✗ 0.9	0.054	2.92

Cross-Entropy

$p_1 = 0.8$
$p_2 = 0.7$
$p_3 = 0.1$

$y_i = 1$ if present on box i

0.8	0.7	✗ 0.9
p_1	p_2	$1 - p_3$
$y_1 = 1$	$y_2 = 1$	$y_3 = 0$

Cross-Entropy

-ln(0.8) - ln(0.7) - ln(0.9)

$$\text{Cross-Entropy} = -\sum_{i=1}^{m} y_i \ln(p_i) + (1 - y_i)\ln(1 - p_i)$$

CE[(1,1,0), (0.8,0.7,0.1)] = 0.69

CE[(0,0,1), (0.8,0.7,0.1)] = 5.12

```
5  def cross_entropy(Y, P):
6      Y = np.float32(Y)
7      P = np.float32(P)
8      return -np.sum(Y * np.log(P) + (1 - Y) * np.log(1 - P))
```

Multi-Class Cross-Entropy

ANIMAL	DOOR 1	DOOR 2	DOOR 3
	0.7	0.3	0.1
	0.2	0.4	0.5
	0.1	0.3	0.4

P = 0.7 * 0.3 * 0.4 = 0.084

CE = -ln(0.7) + -ln(0.3) + -ln(0.4) = 2.48

Multi-Class Cross-Entropy

ANIMAL	DOOR 1	DOOR 2	DOOR 3
	p_{11}	p_{12}	p_{13}
	p_{21}	p_{22}	p_{23}
	p_{31}	p_{32}	p_{33}

$y_{1j} = 1$ if  behind 

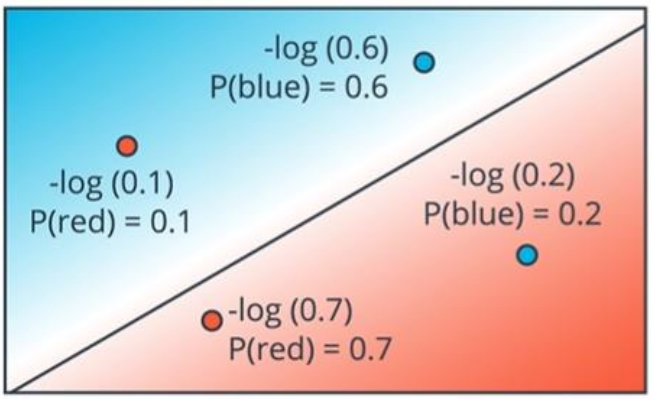
$y_{2j} = 1$ if  behind 

$y_{3j} = 1$ if  behind 

$$\text{Cross-Entropy} = -\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij} \ln(p_{ij})$$

Cross-entropy is inversely proportional to the total probability of an outcome.

Error Function



-log(0.6) - log(0.2) - log(0.1) - log(0.7) = 4.8

0.51 1.61 2.3 0.36

If $y = 1$

$P(\text{blue}) = \hat{y}$

$\text{Error} = -\ln(\hat{y})$

If $y = 0$

$P(\text{red}) = 1 - P(\text{blue}) = 1 - \hat{y}$

$\text{Error} = -\ln(1 - \hat{y})$

$\text{Error} = -(1-y)(\ln(1-\hat{y})) - y\ln(\hat{y})$

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

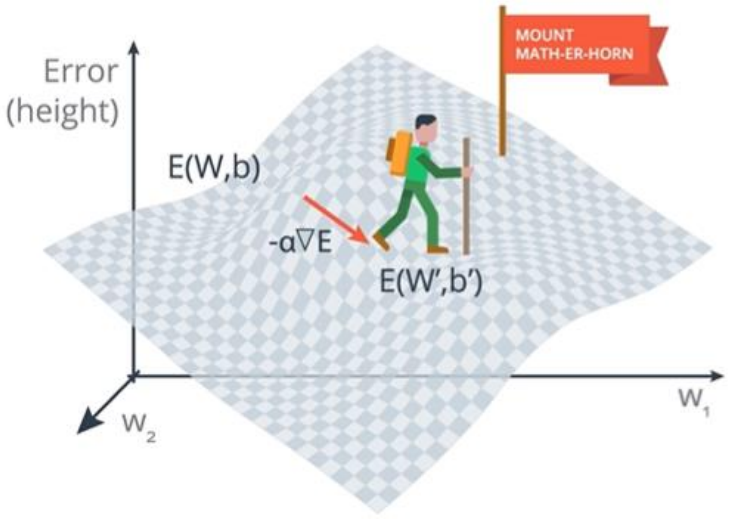
$$\text{Error Function} = -\frac{1}{m}\sum_{i=1}^{m} (1-y_i)(\ln(1-\hat{y}_i)) + y_i \ln(\hat{y}_i)$$

$$E(W,b) = -\frac{1}{m}\sum_{i=1}^{m} (1-y_i)(\ln(1-\sigma(Wx^{(i)}+b)) + y_i \ln(\sigma(Wx^{(i)}+b)$$

GOAL

Minimize Error Function

Gradient Descent



$\hat{y} = \sigma(Wx+b)$ ← Bad

$\hat{y} = \sigma(w_1x_1 + \ldots + w_nx_n + b)$

$\nabla E = (\partial E/\partial w_1, \ldots, \partial E/\partial w_n, \partial E/\partial b)$

$\alpha = 0.1$ (learning rate)

$w_i' \leftarrow w_i - \alpha\, \partial E/\partial w_i$

$b' \leftarrow b - \alpha\, \partial E/\partial b$

$\hat{y} = \sigma(W'x+b')$

Gradient Calculation

In the last few videos, we learned that in order to minimize the error function, we need to take some derivatives. So let's get our hands dirty and actually compute the derivative of the error function. The first thing to notice is that the sigmoid function has a really nice derivative. Namely,

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

The reason for this is the following, we can calculate it using the quotient formula:

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

And now, let's recall that if we have m points labelled $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$, the error formula is:

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

where the prediction is given by $\hat{y}_i = \sigma(Wx^{(i)} + b)$.

Our goal is to calculate the gradient of E, at a point $x = (x_1, \ldots, x_n)$, given by the partial derivatives

$$\nabla E = \left(\frac{\partial}{\partial w_1}E, \cdots, \frac{\partial}{\partial w_n}E, \frac{\partial}{\partial b}E\right)$$

To simplify our calculations, we'll actually think of the error that each point produces, and calculate the derivative of this error. The total error, then, is the average of the errors at all the points. The error produced by each point is, simply,

$$E = -y\ln(\hat{y}) - (1-y)\ln(1-\hat{y})$$

In order to calculate the derivative of this error with respect to the weights, we'll first calculate $\frac{\partial}{\partial w_j}\hat{y}$. Recall that $\hat{y} = \sigma(Wx+b)$, so:

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

The last equality is because the only term in the sum which is not a constant with respect to w_j is precisely w_jx_j, which clearly has derivative x_j.

Now, we can go ahead and calculate the derivative of the error E at a point x, with respect to the weight w_j.

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

A similar calculation will show us that

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

This actually tells us something very important. For a point with coordinates $(x_1, \ldots, x_n)$, label y, and prediction $\hat{y}$, the gradient of the error function at that point is $(-(y-\hat{y})x_1, \cdots, -(y-\hat{y})x_n, -(y-\hat{y}))$. In summary, the gradient is

$$\nabla E = -(y-\hat{y})(x_1, \ldots, x_n, 1).$$

If you think about it, this is fascinating. The gradient is actually a scalar times the coordinates of the point! And what is the scalar? Nothing less than a multiple of the difference between the label and the prediction. What significance does this have?

QUIZ QUESTION

What does the scalar we obtained above signify? (Check all that are true.)

- [ ] Closer the label to the prediction, larger the gradient.
- [x] Closer the label to the prediction, smaller the gradient.
- [x] Farther the label from the prediction, larger the gradient.
- [ ] Farther the label to the prediction, smaller the gradient.

SUBMIT

So, a small gradient means we'll change our coordinates by a little bit, and a large gradient means we'll change our coordinates by a lot.

If this sounds anything like the perceptron algorithm, this is no coincidence! We'll see it in a bit.

Gradient Descent Step

Therefore, since the gradient descent step simply consists in subtracting a multiple of the gradient of the error function at every point, then this updates the weights in the following way:

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

which is equivalent to

$$w_i' \leftarrow w_i + \alpha(y - \hat{y})x_i.$$

Similarly, it updates the bias in the following way:

$$b' \leftarrow b + \alpha(y - \hat{y}),$$

Note: Since we've taken the average of the errors, the term we are adding should be $\frac{1}{m} \cdot \alpha$ instead of α, but as α is a constant, then in order to simplify calculations, we'll just take $\frac{1}{m} \cdot \alpha$ to be our learning rate, and abuse the notation by just calling it α.

Gradient Descent: The Code

From before we saw that one weight update can be calculated as:

$$\Delta w_i = \alpha * \delta * x_i$$

where α is the learning rate and δ is the error term.

Previously, we utilized the loss (error) function for logistic regression, which was because we were performing a binary classification task. This time we'll try to get the function to learn a value instead of a class. Therefore, we'll use a simpler loss function, as defined below in the error term δ.

$$\delta = (y - \hat{y})f'(h) = (y - \hat{y})f'(\sum w_i x_i)$$

Note that $f'(h)$ is the derivative of the activation function $f(h)$, and h is defined as the output, which in the case of a neural network is a sum of the weights times the inputs.

Now I'll write this out in code for the case of only one output unit. We'll also be using the sigmoid as the activation function $f(h)$.

```
2
3  def sigmoid(x):
4      """
5      Calculate sigmoid
6      """
7      return 1/(1+np.exp(-x))
8
9  learnrate = 0.5
10 x = np.array([1, 2])
11 y = np.array(0.5)
12
13 # Initial weights
14 w = np.array([0.5, -0.5])
15
16 # Calculate one gradient descent step for each weight
17 # TODO: Calculate output of neural network
18 nn_output = sigmoid(np.dot(w, x))
19
20 # TODO: Calculate error of neural network
21 error = y - nn_output
22 # TODO: Calculate change in weights
23 del_w = learnrate * error * nn_output * (1 - nn_output) * x
24
25 print('Neural Network output:')
26 print(nn_output)
27 print('Amount of Error:')
28 print(error)
29 print('Change in Weights:')
30 print(del_w)
```

Perceptron vs Gradient Descent

GRADIENT DESCENT ALGORITHM:

Change
w_i to $w_i + \alpha(y-\hat{y})x_i$

PERCEPTRON ALGORITHM:

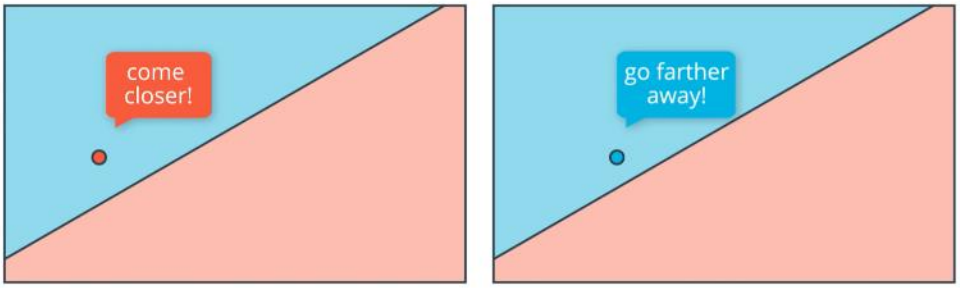
If x is missclassified:

Change w_i to $\begin{cases} w_i + \alpha x_i & \text{if positive} \\ w_i - \alpha x_i & \text{if negative} \end{cases}$

If correctly classified: $y-\hat{y}=0$

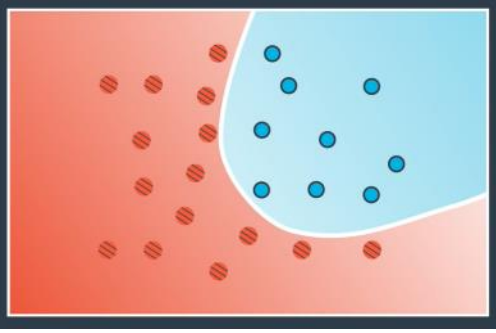
If missclassified: $\begin{cases} y-\hat{y} = 1 & \text{if positive} \\ y-\hat{y} = -1 & \text{if negative} \end{cases}$

Gradient Descent Algorithm

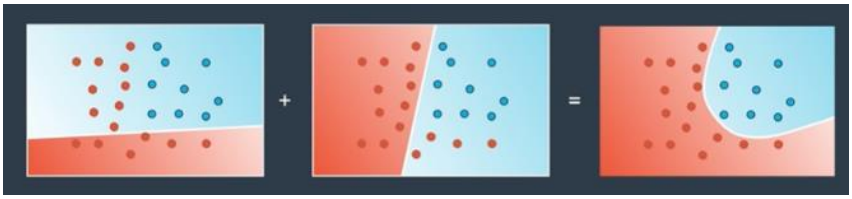


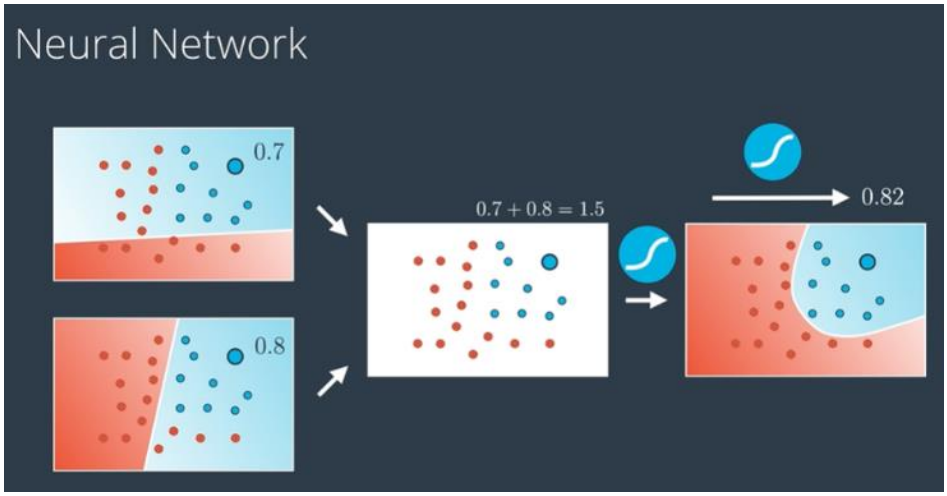
INCORRECTLY CLASSIFIED **CORRECTLY CLASSIFIED**

Non-Linear Regions



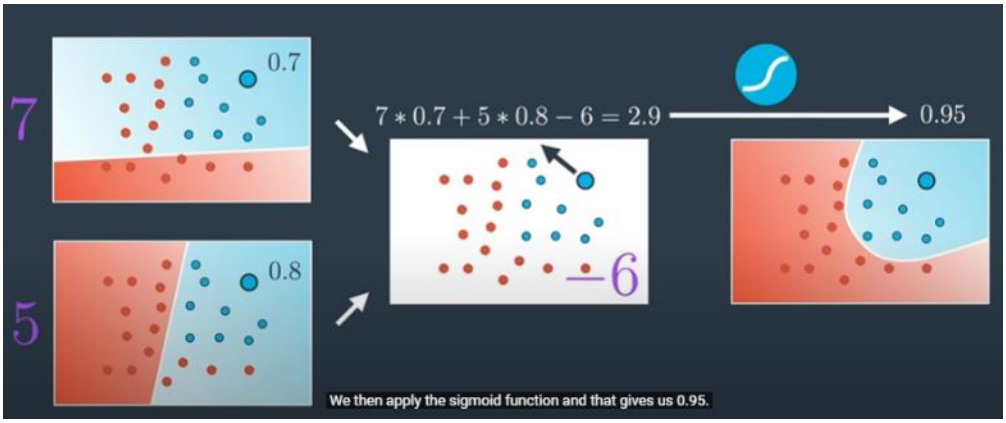
Combining Regions



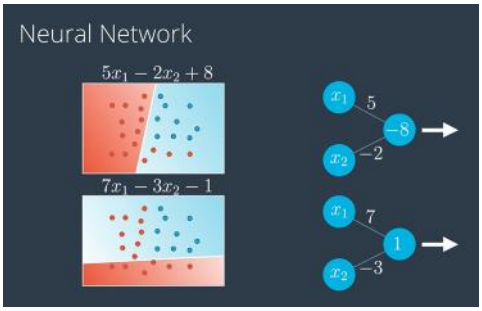


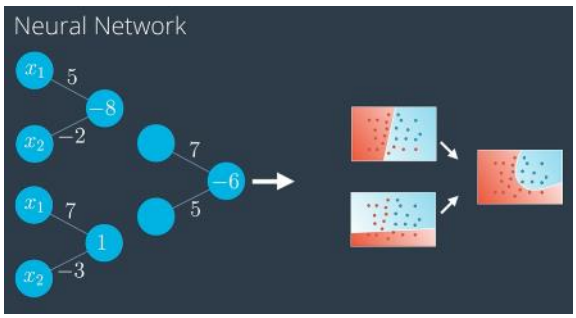
It is possible to ponderate the summs: (increasing weight to the 1st term)

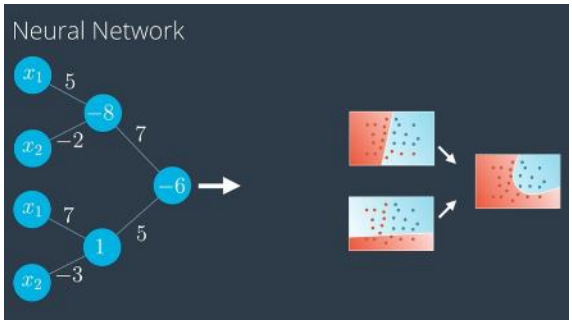
Linear combination of two linear models: With a 1st weight of 7 and 2nd weight of 5 and bias of -6:

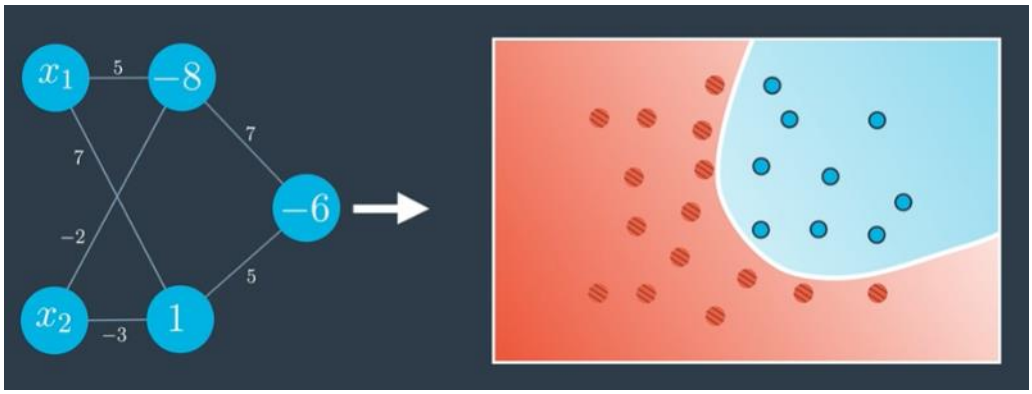


Domains and their perceptrons:

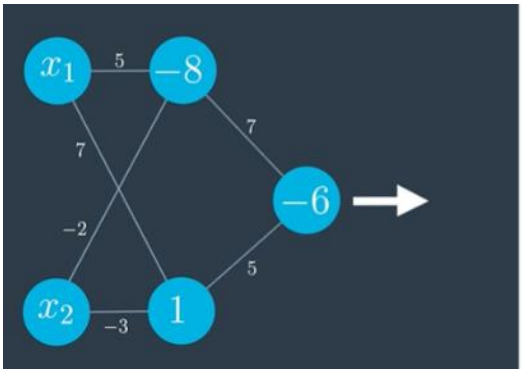


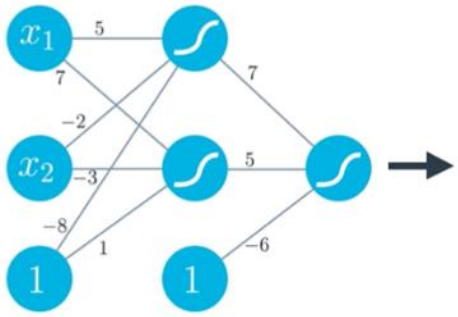


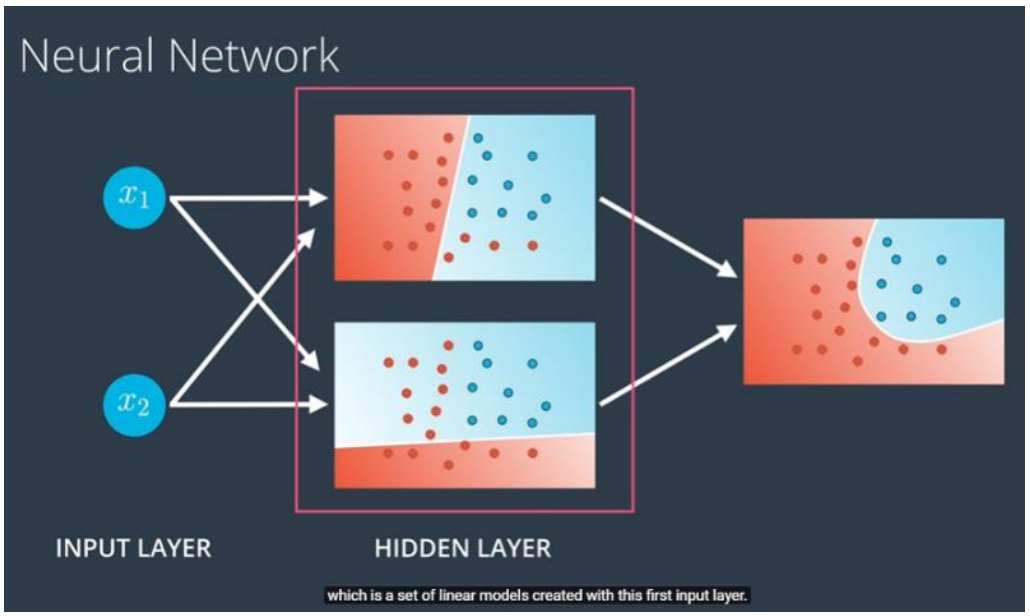


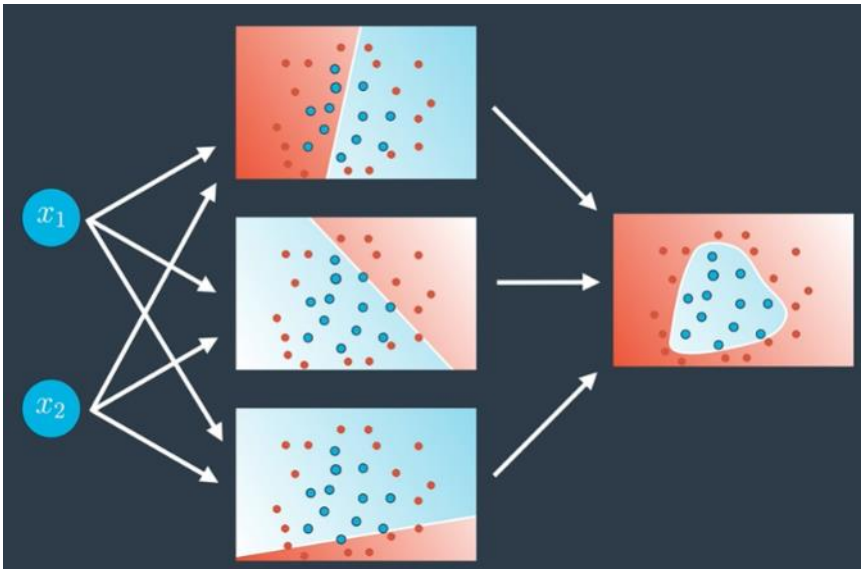


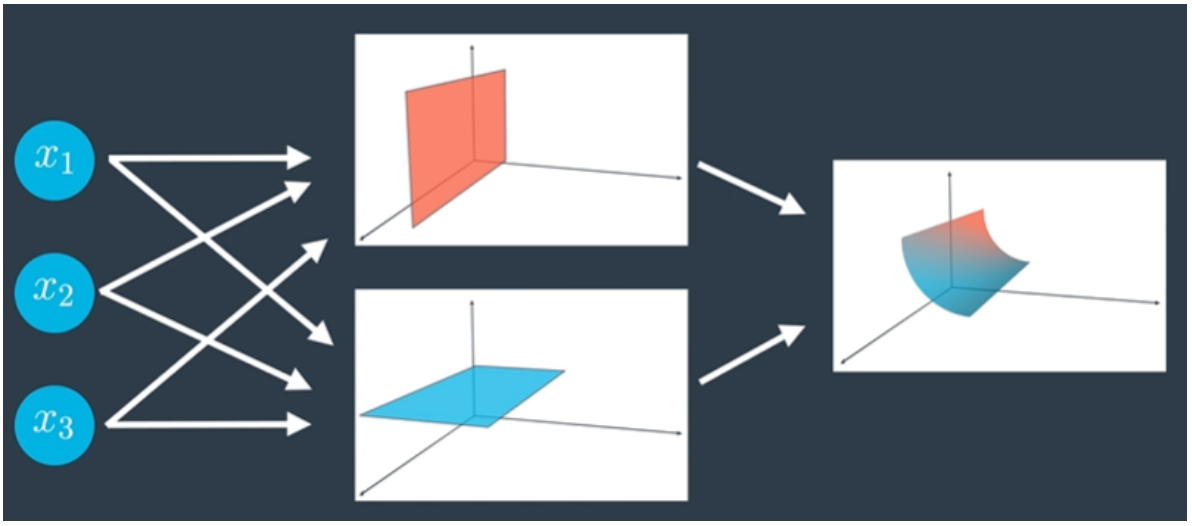
With another notation:

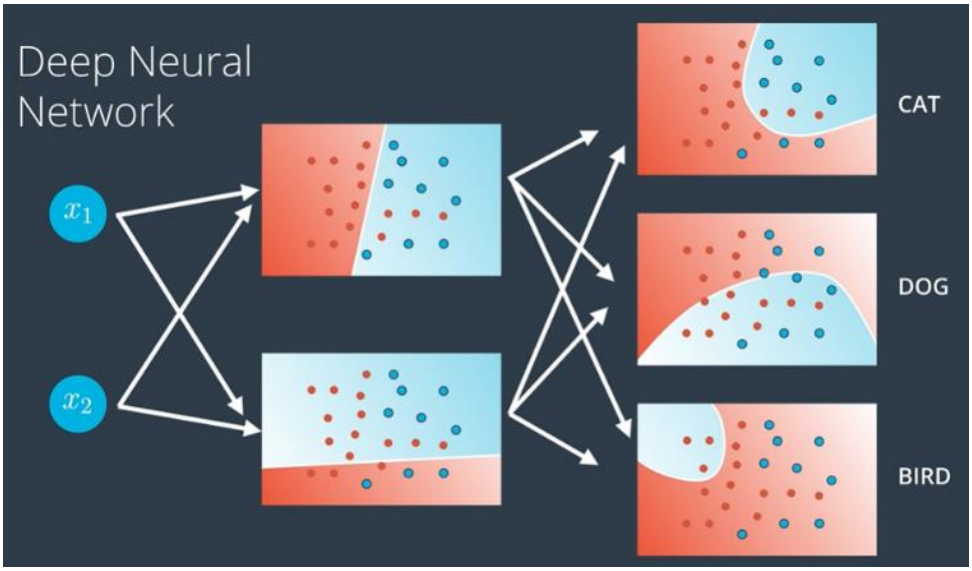


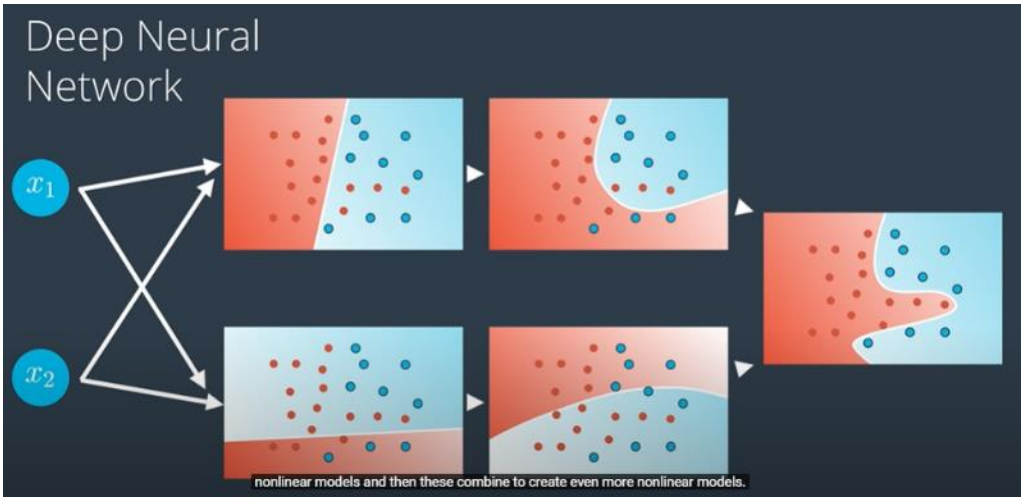


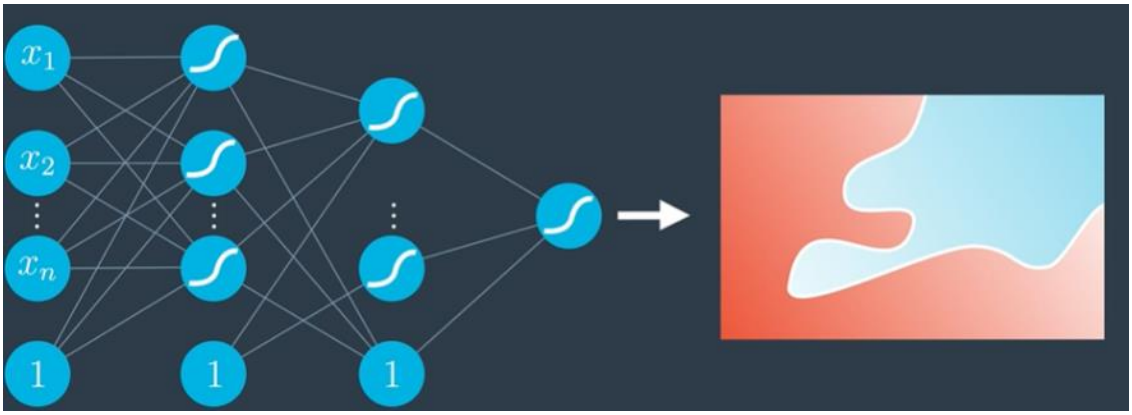



x_1
x_2

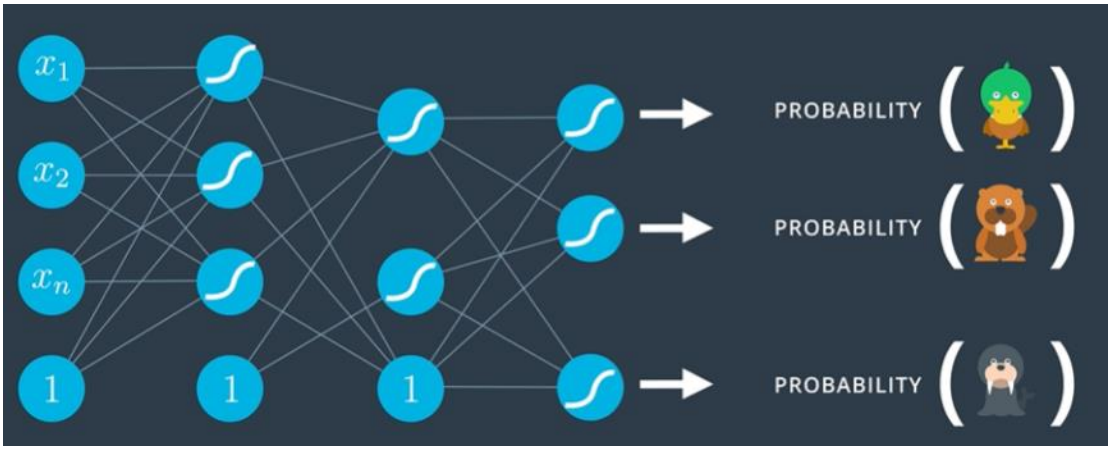

x_1
x_2
x_3


Deep Neural Network
x_1
x_2
CAT
DOG
BIRD



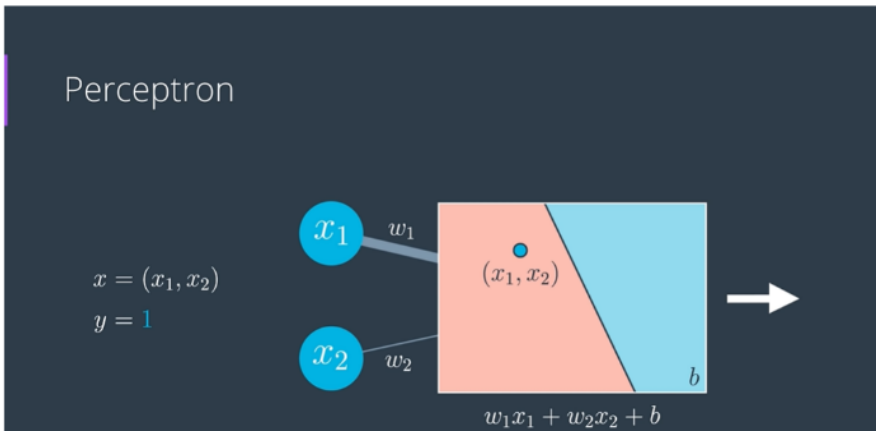


Multi-class classification:

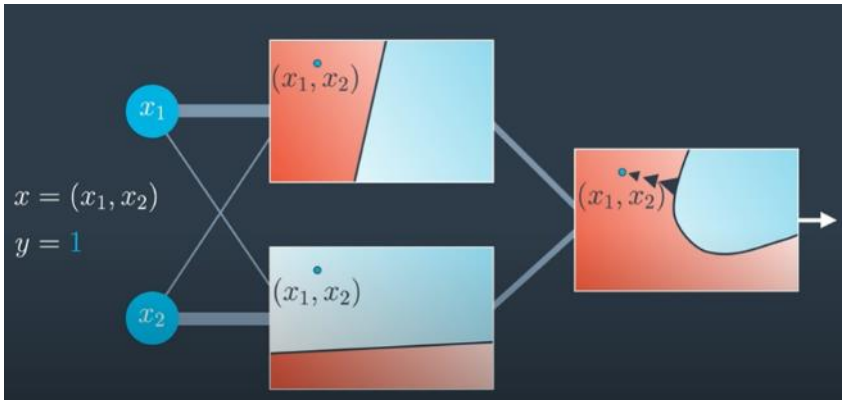


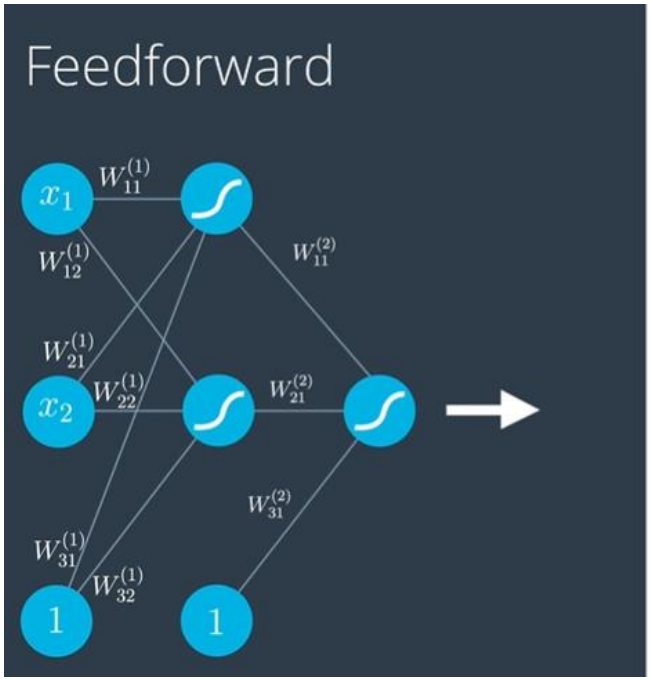
Feedforward

Feedforward is the process neural networks use to turn the input into an output. Let's study it more carefully, before we dive into how to train the networks.



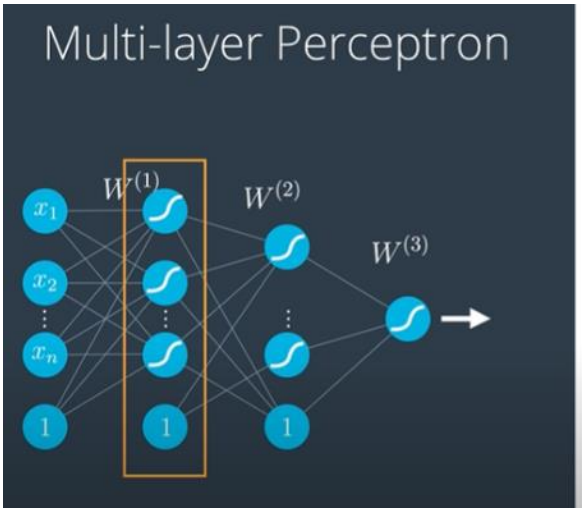
Thicker lines, higher weights





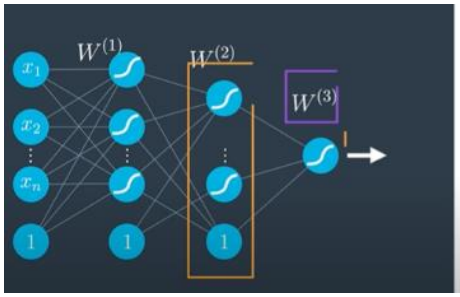
$$\hat{y} = \sigma \begin{pmatrix} W_{11}^{(2)} \\ W_{21}^{(2)} \\ W_{31}^{(2)} \end{pmatrix} \sigma \begin{pmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

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



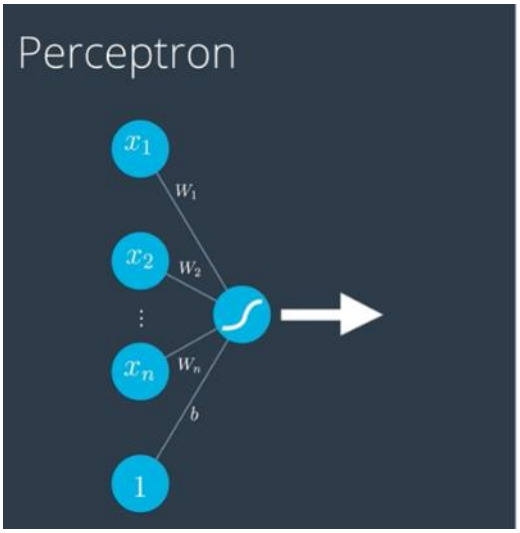
PREDICTION

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



PREDICTION

$\hat{y} = \sigma \circ W^{(3)} \circ \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)}(x)$

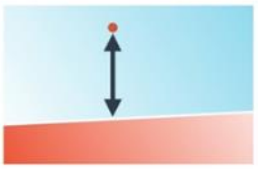


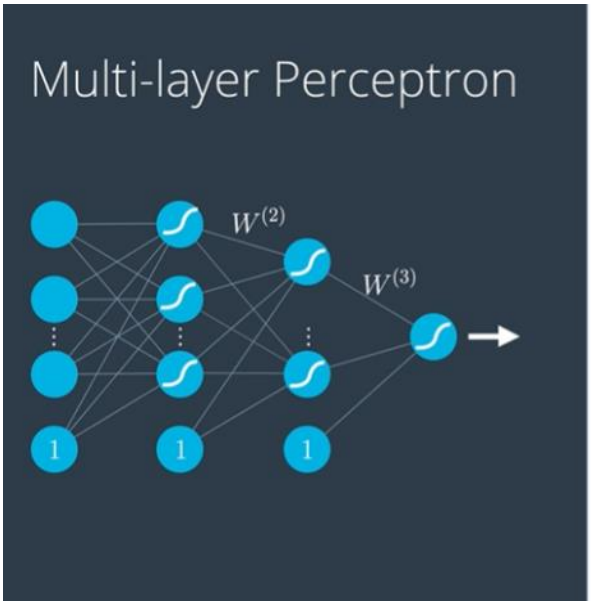
PREDICTION

$$\hat{y} = \sigma(Wx + b)$$

ERROR FUNCTION

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



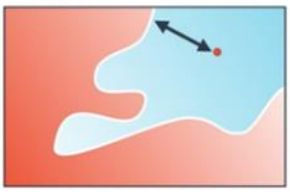


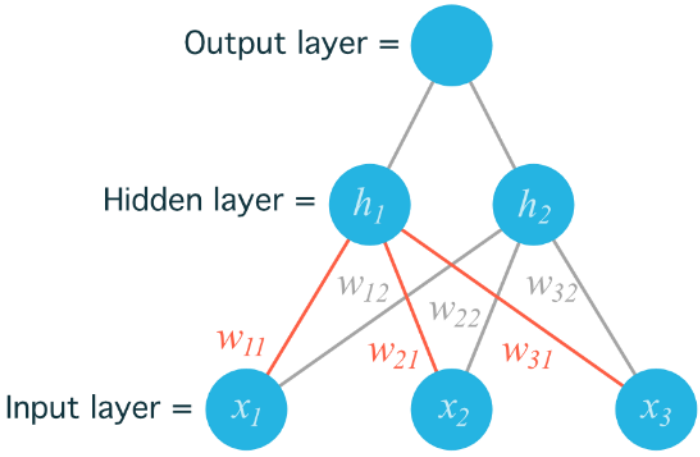
PREDICTION

$$\hat{y} = \sigma \circ W^{(3)} \circ \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)}(x)$$

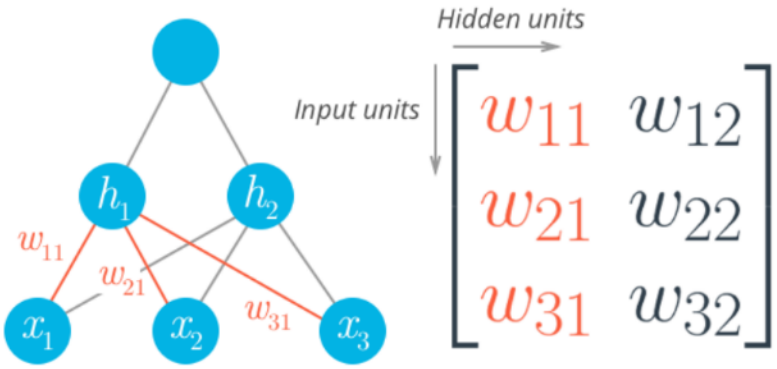
ERROR FUNCTION

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





But now, the weights need to be stored in a **matrix**, indexed as w_{ij}. Each **row** in the matrix will correspond to the weights **leading out** of a **single input unit**, and each **column** will correspond to the weights **leading in** to a **single hidden unit**. For our three input units and two hidden units, the weights matrix looks like this:



Making a column vector

You see above that sometimes you'll want a column vector, even though by default Numpy arrays work like row vectors. It's possible to get the transpose of an array like so `arr.T`, but for a 1D array, the transpose will return a row vector. Instead, use `arr[:,None]` to create a column vector:

```
print(features)
> array([ 0.49671415, -0.1382643 ,  0.64768854])

print(features.T)
> array([ 0.49671415, -0.1382643 ,  0.64768854])

print(features[:, None])
> array([[ 0.49671415],
       [-0.1382643 ],
       [ 0.64768854]])
```

Alternatively, you can create arrays with two dimensions. Then, you can use `arr.T` to get the column vector.

```
np.array(features, ndmin=2)
> array([[ 0.49671415, -0.1382643 ,  0.64768854]])

np.array(features, ndmin=2).T
> array([[ 0.49671415],
       [-0.1382643 ],
       [ 0.64768854]])
```

I personally prefer keeping all vectors as 1D arrays, it just works better in my head.

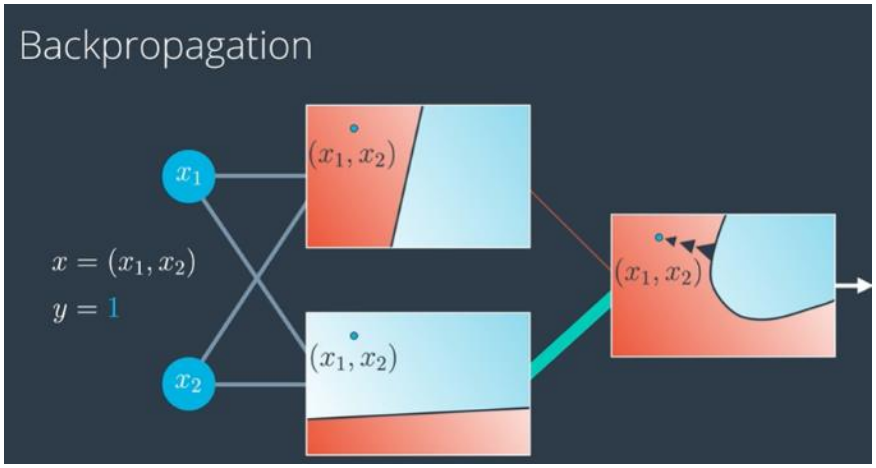
```
def sigmoid(x):
    """
    Calculate sigmoid
    """
    return 1/(1+np.exp(-x))

# Network size
N_input = 4
N_hidden = 3
N_output = 2

np.random.seed(42)
# Make some fake data
X = np.random.randn(4)
print(X)

weights_input_to_hidden = np.random.normal(0, scale=0.1, size=(N_input, N_hidden))
weights_hidden_to_output = np.random.normal(0, scale=0.1, size=(N_hidden, N_output))
print(weights_input_to_hidden.shape)
print(weights_hidden_to_output.shape)

# TODO: Make a forward pass through the network

hidden_layer_in = np.dot(weights_input_to_hidden.T, X)
hidden_layer_out = sigmoid(hidden_layer_in)

print('Hidden-layer Output:')
print(hidden_layer_out)

output_layer_in = np.dot(weights_hidden_to_output.T, hidden_layer_out)
output_layer_out = sigmoid(output_layer_in)
```

Backpropagation

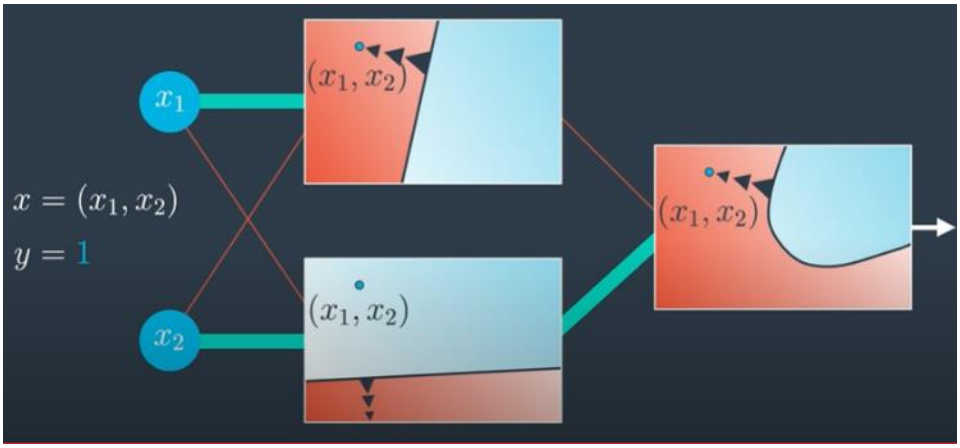
Now, we're ready to get our hands into training a neural network. For this, we'll use the method known as **backpropagation**. In a nutshell, backpropagation will consist of:

- Doing a feedforward operation.
- Comparing the output of the model with the desired output.
- Calculating the error.
- Running the feedforward operation backwards (backpropagation) to spread the error to each of the weights.
- Use this to update the weights, and get a better model.
- Continue this until we have a model that is good.

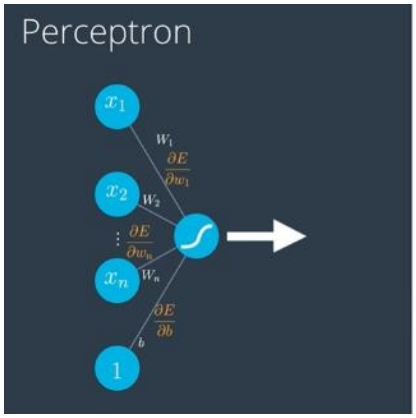
The point says that the bottom model is better, so it increments its weight in detriment of the other one:



And also:



Bias unit is also updated



PREDICTION

$$\hat{y} = \sigma(Wx + b)$$

ERROR FUNCTION

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

GRADIENT OF THE ERROR FUNCTION

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}, \frac{\partial E}{\partial b}\right)$$

Multi-layer Perceptron

PREDICTION

$$\hat{y} = \sigma W^{(3)} \circ \sigma W^{(2)} \circ \sigma \circ W^{(1)}(x)$$

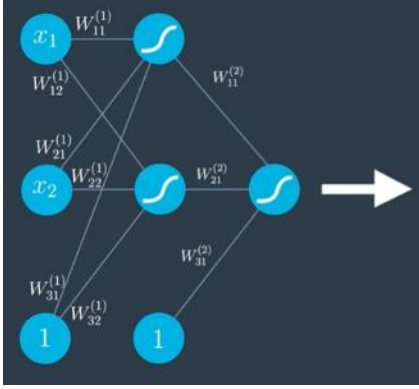
ERROR FUNCTION

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

GRADIENT OF THE ERROR FUNCTION

$$\nabla E = (..., \frac{\partial E}{\partial w_j^{(i)}}, ...)$$

Backpropagation



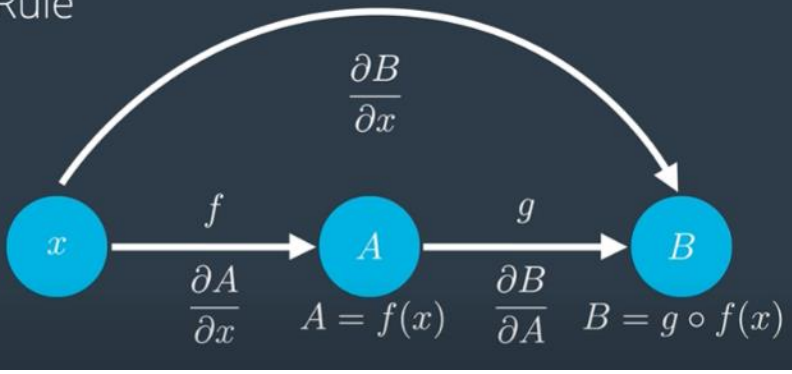
$$\hat{y} = \sigma W^{(2)} \circ \sigma \circ W^{(1)}(x)$$

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

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

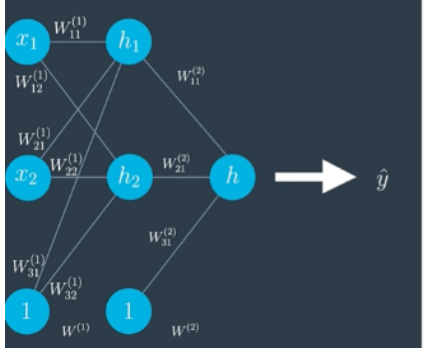
$$W_{ij}^{\prime(k)} \leftarrow W_{ij}^{(k)} - \alpha \frac{\partial E}{\partial W_{ij}^{(k)}}$$

Chain Rule



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

Feedforward



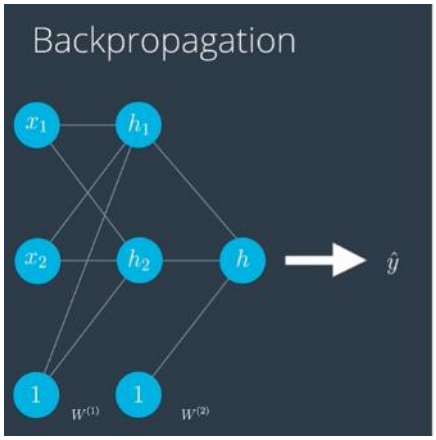
$$h_1 = W_{11}^{(1)}x_1 + W_{21}^{(1)}x_2 + W_{31}^{(1)}$$

$$h_2 = W_{12}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{32}^{(1)}$$

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

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

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



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

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

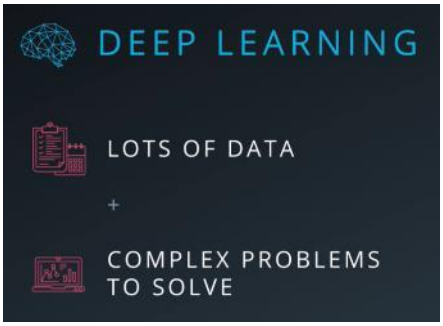
$$\nabla E = (\frac{\partial E}{\partial W_{11}^{(1)}}, ..., \frac{\partial E}{\partial W_{31}^{(2)}})$$

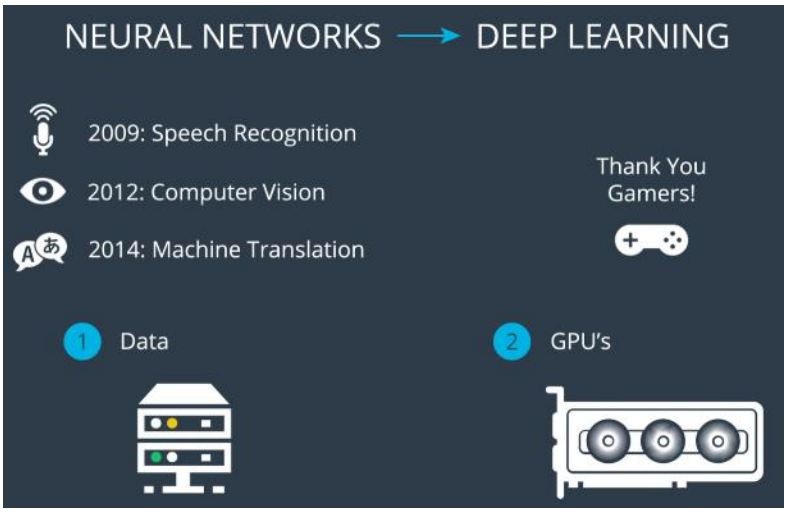
$$\frac{\partial E}{\partial W_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial h_1} \frac{\partial h_1}{\partial W_{11}^{(1)}}$$

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

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

Deep Learning is useful for:





Hello, Tensor World!

Let's analyze the Hello World script you ran. For reference, I've added the code below.

```
import tensorflow as tf

# Create TensorFlow object called hello_constant
hello_constant = tf.constant('Hello World!')

with tf.Session() as sess:
    # Run the tf.constant operation in the session
    output = sess.run(hello_constant)
    print(output)
```

Tensor

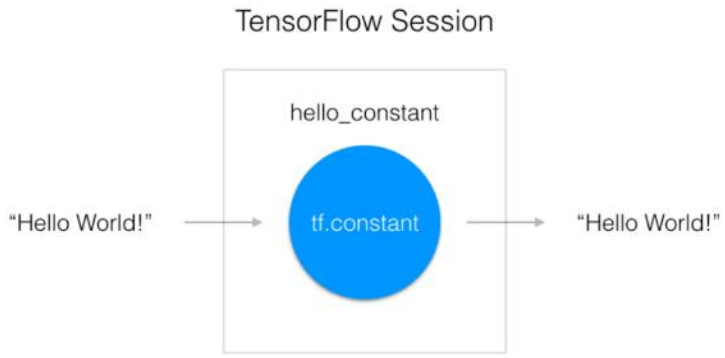
In TensorFlow, data isn't stored as integers, floats, or strings. These values are encapsulated in an object called a tensor. In the case of `hello_constant = tf.constant('Hello World!')`, `hello_constant` is a 0-dimensional string tensor, but tensors come in a variety of sizes as shown below:

```
# A is a 0-dimensional int32 tensor
A = tf.constant(1234)
# B is a 1-dimensional int32 tensor
B = tf.constant([123,456,789])
# C is a 2-dimensional int32 tensor
C = tf.constant([ [123,456,789], [222,333,444] ])
```

In newer versions of TF instead of:
Import tensorflow as tf -→
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
since the udacity course is in TF older sintax

Session

TensorFlow's api is built around the idea of a **computational graph**, a way of visualizing a mathematical process. Let's take the TensorFlow code you ran and turn that into a graph:



A "TensorFlow Session", as shown above, is an environment for running a graph. The session is in charge of allocating the operations to GPU(s) and/or CPU(s), including remote machines. Let's see how you use it.

```
with tf.Session() as sess:
    output = sess.run(hello_constant)
    print(output)
```

The code has already created the tensor, `hello_constant`, from the previous lines. The next step is to evaluate the tensor in a session.

The code creates a session instance, `sess`, using `tf.Session`. The `sess.run()` function then evaluates the tensor and returns the results.

Input

In the last section, you passed a tensor into a session and it returned the result. What if you want to use a non-constant? This is where `tf.placeholder()` and `feed_dict` come into place. In this section, you'll go over the basics of feeding data into TensorFlow.

tf.placeholder()

Sadly you can't just set `x` to your dataset and put it in TensorFlow, because over time you'll want your TensorFlow model to take in different datasets with different parameters. You need `tf.placeholder()`!

`tf.placeholder()` returns a tensor that gets its value from data passed to the `tf.session.run()` function, allowing you to set the input right before the session runs.

Session's feed_dict

```
x = tf.placeholder(tf.string)

with tf.Session() as sess:
    output = sess.run(x, feed_dict={x: 'Hello World'})
```

Use the `feed_dict` parameter in `tf.session.run()` to set the placeholder tensor. The above example shows the tensor `x` being set to the string `"Hello, world"`. It's also possible to set more than one tensor using `feed_dict` as shown below.

```
x = tf.placeholder(tf.string)
y = tf.placeholder(tf.int32)
z = tf.placeholder(tf.float32)

with tf.Session() as sess:
    output = sess.run(x, feed_dict={x: 'Test String', y: 123, z: 45.67})
```

Note: If the data passed to the `feed_dict` doesn't match the tensor type and can't be cast into the tensor type, you'll get the error "`ValueError: invalid literal for`...".

Addition

```
x = tf.add(5, 2)  # 7
```

You'll start with the add function. The `tf.add()` function does exactly what you expect it to do. It takes in two numbers, two tensors, or one of each, and returns their sum as a tensor.

Subtraction and Multiplication

Here's an example with subtraction and multiplication.

```
x = tf.subtract(10, 4) # 6
y = tf.multiply(2, 5)  # 10
```

Converting types

It may be necessary to convert between types to make certain operators work together. For example, if you tried the following, it would fail with an exception:

```
tf.subtract(tf.constant(2.0),tf.constant(1))  # Fails with ValueError: Tensor conversion requested dtype
```

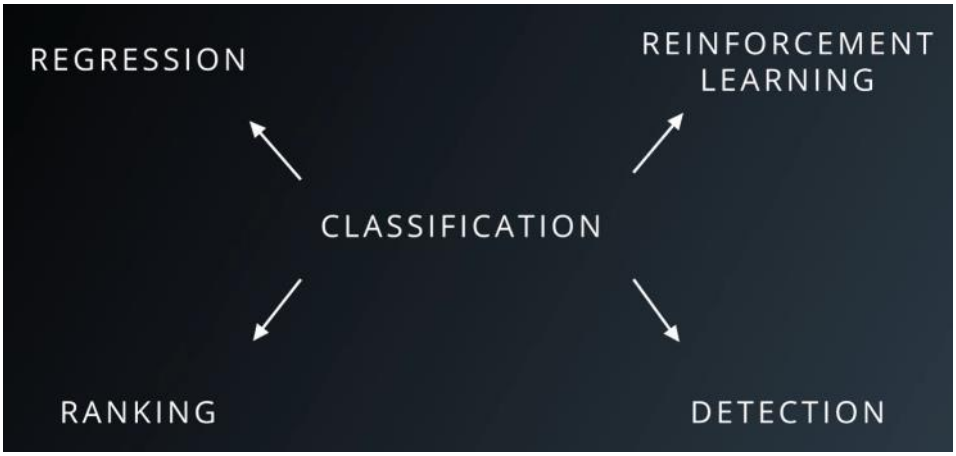
That's because the constant `1` is an integer but the constant `2.0` is a floating point value and `subtract` expects them to match.

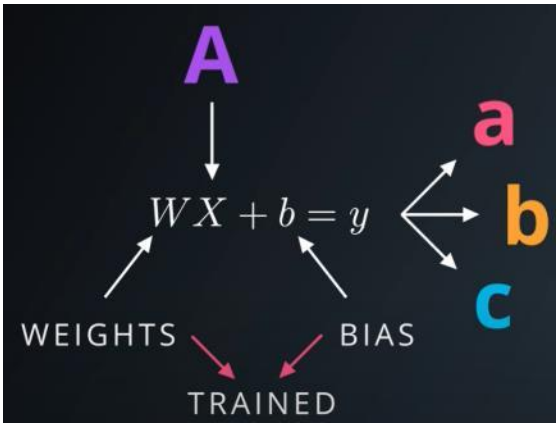
In cases like these, you can either make sure your data is all of the same type, or you can cast a value to another type. In this case, converting the `2.0` to an integer before subtracting, like so, will give the correct result:

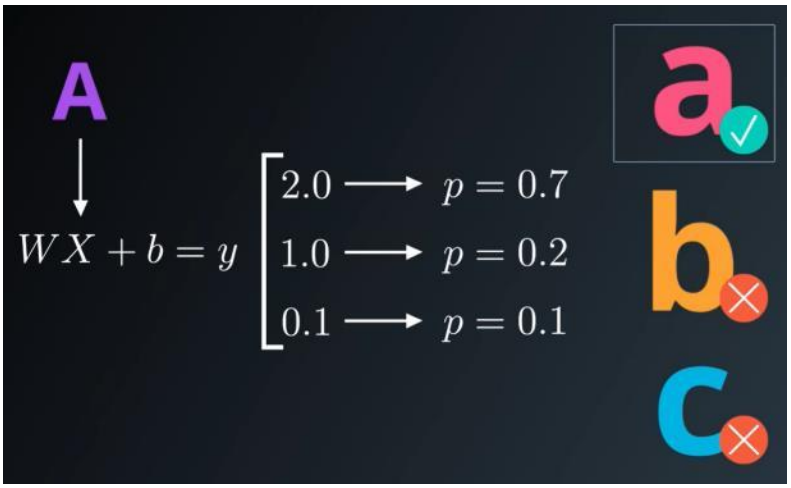
```
tf.subtract(tf.cast(tf.constant(2.0), tf.int32), tf.constant(1))   # 1
```

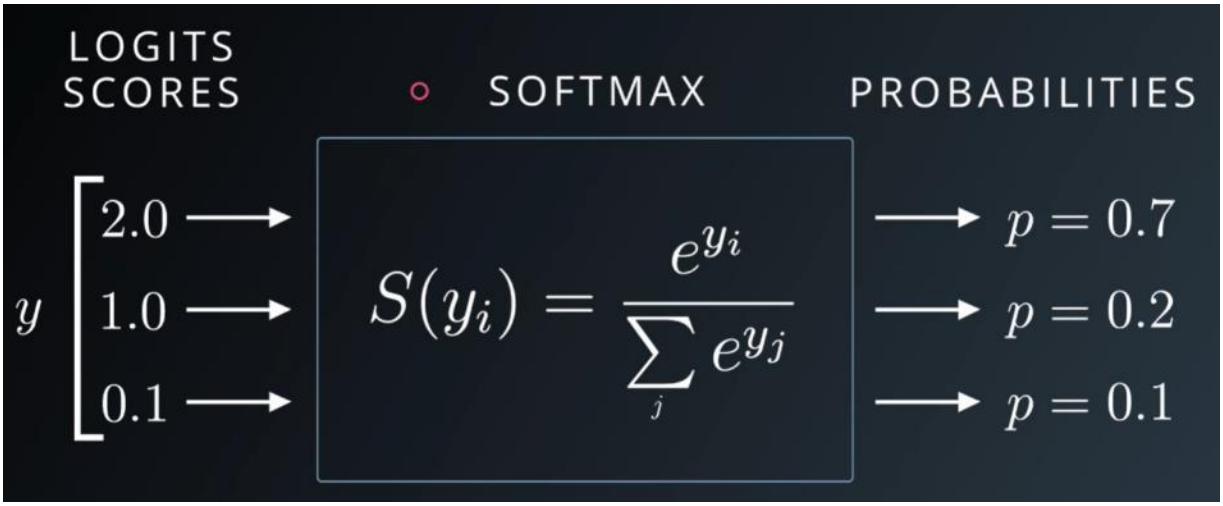
```
# Solution is available in the "solution.ipynb"
import tensorflow as tf

# TODO: Convert the following to TensorFlow:
# x = 10
x = tf.constant(10)
# y = 2
y = tf.constant(2)
# z = x/y - 1
z = tf.subtract(tf.divide(x, y), tf.cast(tf.constant(1), tf.float64))

# TODO: Print z from a session as the variable output
with tf.Session() as sess:
    output = sess.run(z)
    print(output)
```

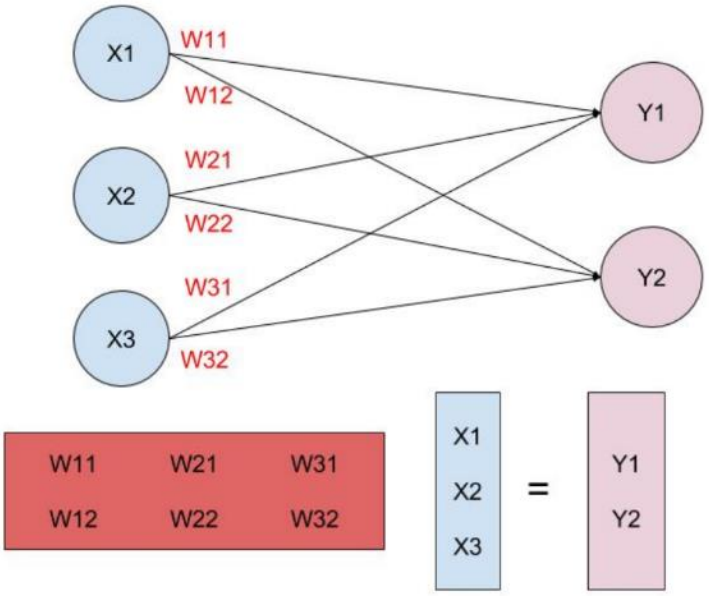
```
4.0
```


REGRESSION
REINFORCEMENT LEARNING
CLASSIFICATION
RANKING
DETECTION


A
$WX + b = y$
a
b
c
WEIGHTS
BIAS
TRAINED


A
$WX + b = y$
$2.0 \longrightarrow p = 0.7$
$1.0 \longrightarrow p = 0.2$
$0.1 \longrightarrow p = 0.1$

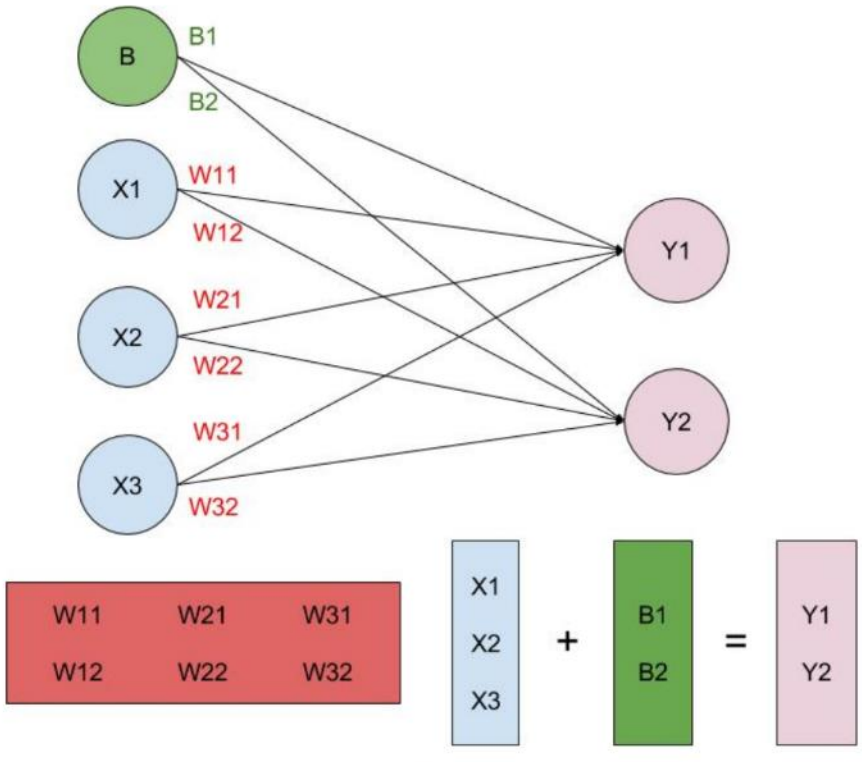

LOGITS
SCORES
SOFTMAX
PROBABILITIES
y
$2.0 \longrightarrow$
$1.0 \longrightarrow$
$0.1 \longrightarrow$
$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$
$\longrightarrow p = 0.7$
$\longrightarrow p = 0.2$
$\longrightarrow p = 0.1$



`y = Wx` allows us to segment the data into their respective labels using a line.

However, this line has to pass through the origin, because whenever `x` equals 0, then `y` is also going to equal 0.

We want the ability to shift the line away from the origin to fit more complex data. The simplest solution is to add a number to the function, which we call "bias".



Function y = Wx + b

Our new function becomes `Wx + b`, allowing us to create predictions on linearly separable data. Let's use a concrete example and calculate the logits.

Transposition

We've been using the `y = Wx + b` function for our linear function.

But there's another function that does the same thing, `y = xW + b`. These functions do the same thing and are interchangeable, except for the dimensions of the matrices involved.

To shift from one function to the other, you simply have to swap the row and column dimensions of each matrix. This is called transposition.

For rest of this lesson, we actually use `xW + b`, because this is what TensorFlow uses.

$$y = \begin{pmatrix} 0.2 & 0.5 & 0.6 \end{pmatrix} \begin{pmatrix} -0.5 & 0.7 \\ 0.2 & -0.8 \\ 0.1 & 0.2 \end{pmatrix} + \begin{pmatrix} 0.1 & 0.2 \end{pmatrix}$$

y = xW + b

The above example is identical to the quiz you just completed, except that the matrices are transposed.

`x` now has the dimensions 1x3, `W` now has the dimensions 3x2, and `b` now has the dimensions 1x2. Calculating this will produce a matrix with the dimension of 1x2.

Weights and Bias in TensorFlow

The goal of training a neural network is to modify weights and biases to best predict the labels. In order to use weights and bias, you'll need a Tensor that can be modified. This leaves out **`tf.placeholder()`** and **`tf.constant()`**, since those Tensors can't be modified. This is where **`tf.Variable`** class comes in.

tf.Variable()

```
x = tf.Variable(5)
```

The **`tf.Variable`** class creates a tensor with an initial value that can be modified, much like a normal Python variable. This tensor stores its state in the session, so you must initialize the state of the tensor manually. You'll use the **`tf.global_variables_initializer()`** function to initialize the state of all the Variable tensors.

Initialization

```
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
```

The **`tf.global_variables_initializer()`** call returns an operation that will initialize all TensorFlow variables from the graph. You call the operation using a session to initialize all the variables as shown above. Using the **`tf.Variable`** class allows us to change the weights and bias, but an initial value needs to be chosen.

Initializing the weights with random numbers from a normal distribution is good practice. Randomizing the weights helps the model from becoming stuck in the same place every time you train it. You'll learn more about this in the next lesson, when you study gradient descent.

Similarly, choosing weights from a normal distribution prevents any one weight from overwhelming other weights. You'll use the **`tf.truncated_normal()`** function to generate random numbers from a normal distribution.

tf.truncated_normal()

```
n_features = 120
n_labels = 5
weights = tf.Variable(tf.truncated_normal((n_features, n_labels)))
```

The `tf.truncated_normal()` function returns a tensor with random values from a normal distribution whose magnitude is no more than 2 standard deviations from the mean.

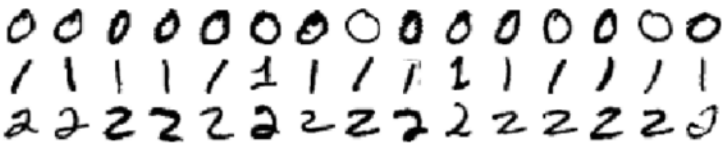
Since the weights are already helping prevent the model from getting stuck, you don't need to randomize the bias. Let's use the simplest solution, setting the bias to 0.

tf.zeros()

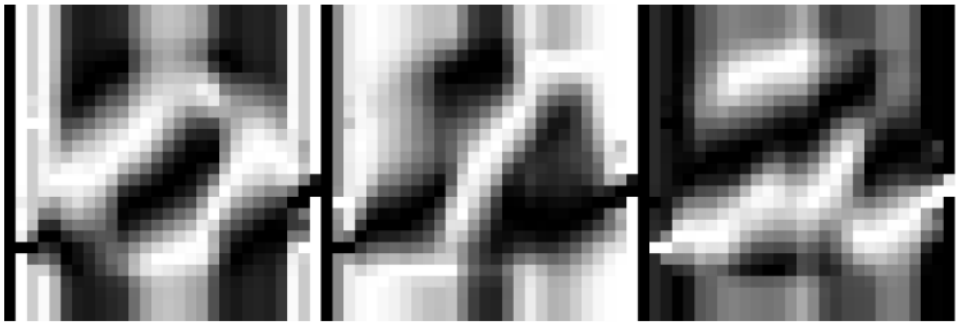
```
n_labels = 5
bias = tf.Variable(tf.zeros(n_labels))
```

The `tf.zeros()` function returns a tensor with all zeros.

Linear Classifier Quiz



You'll be classifying the handwritten numbers `0`, `1`, and `2` from the MNIST dataset using TensorFlow. The above is a small sample of the data you'll be training on. Notice how some of the `1`s are written with a **serif** at the top and at different angles. The similarities and differences will play a part in shaping the weights of the model.

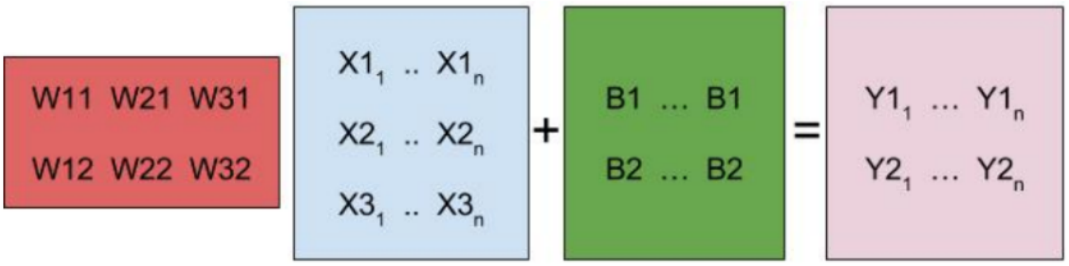


Left: Weights for labeling 0. Middle: Weights for labeling 1. Right: Weights for labeling 2.

The images above are trained weights for each label (`0`, `1`, and `2`). The weights display the unique properties of each digit they have found. Complete this quiz to train your own weights using the MNIST dataset.

Linear Update

You can't train a neural network on a single sample. Let's apply n samples of `x` to the function `y = Wx + b`, which becomes `Y = WX + B`.



Y = WX + B

For every sample of `X` (`X1`, `X2`, `X3`), we get logits for label 1 (`Y1`) and label 2 (`Y2`).

In order to add the bias to the product of `WX`, we had to turn `b` into a matrix of the same shape. This is a bit unnecessary, since the bias is only two numbers. It should really be a vector.

We can take advantage of an operation called broadcasting used in TensorFlow and Numpy. This operation allows arrays of different dimension to be added or multiplied with each other. For example:

```
import numpy as np
t = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
u = np.array([1, 2, 3])
print(t + u)
```

The code above will print...

```
[[ 2  4  6]
 [ 5  7  9]
 [ 8 10 12]
 [11 13 15]]
```

This is because `u` is the same dimension as the last dimension in `t`.

```
2  import numpy as np
3
4
5  def softmax(x):
6      """Compute softmax values for each sets of scores in x."""
7      # TODO: Compute and return softmax(x)
8      return np.exp(x) / np.sum(np.exp(x), axis = 0)
9
10 logits = [3.0, 1.0, 0.2]
11 print(softmax(logits))
```

TensorFlow Softmax

Now that you've built a softmax function from scratch, let's see how softmax is done in TensorFlow.

```
x = tf.nn.softmax([2.0, 1.0, 0.2])
```

Easy as that! `tf.nn.softmax()` implements the softmax function for you. It takes in logits and returns softmax activations.

```
# Solution is available in the other "solution.ipynb"
import tensorflow as tf


def run():
    output = None
    logit_data = [2.0, 1.0, 0.1]
    logits = tf.placeholder(tf.float32)

    # TODO: Calculate the softmax of the logits
    softmax = tf.nn.softmax(logit_data)

    with tf.Session() as sess:
        pass
        # TODO: Feed in the logit data
        output = sess.run(softmax, feed_dict = {logits: logit_data})

    return output
```

Quiz

Answer the following 2 questions about softmax.

QUESTION 1 OF 2

What happens to the softmax probabilities when you multiply the logits by 10?

- ✓ Probabilities get close to 0.0 or 1.0
- Probabilities get close to the uniform distribution

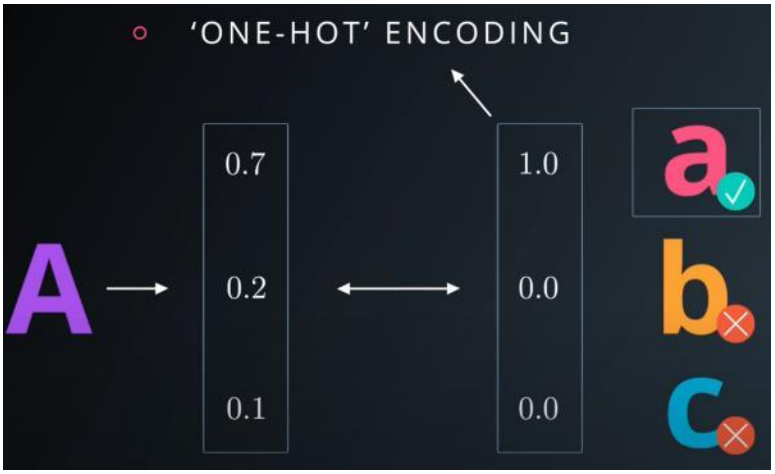
SUBMIT

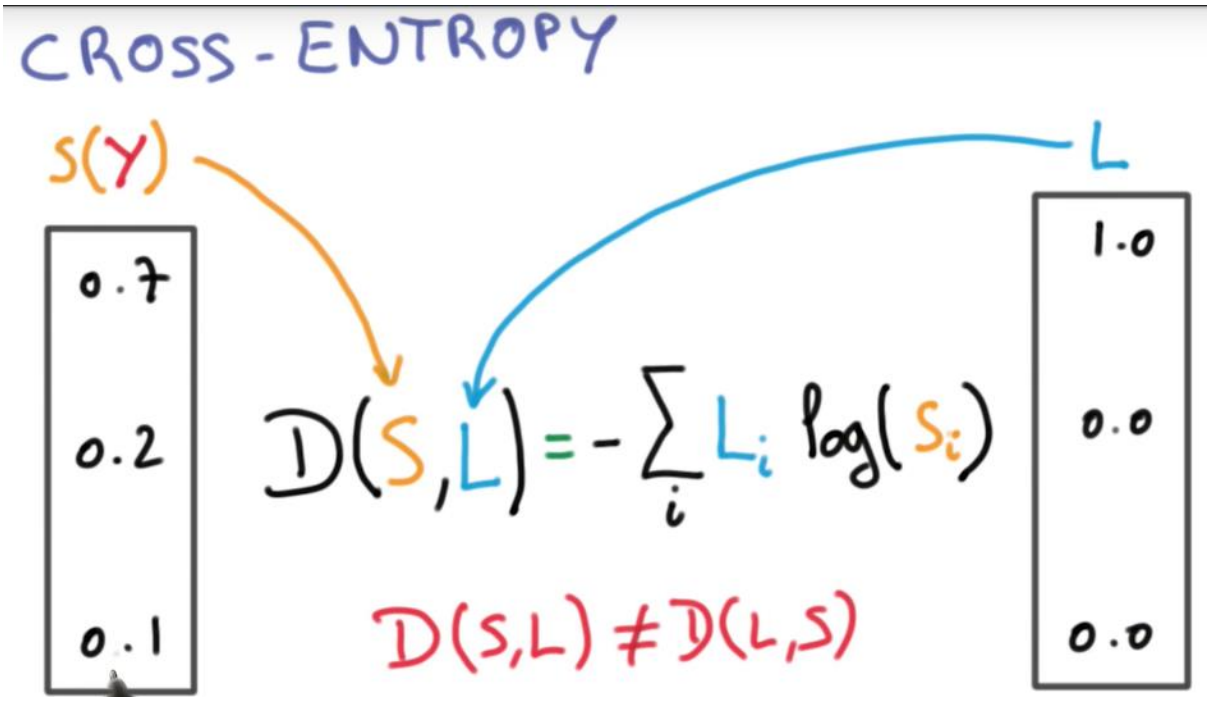
QUESTION 2 OF 2

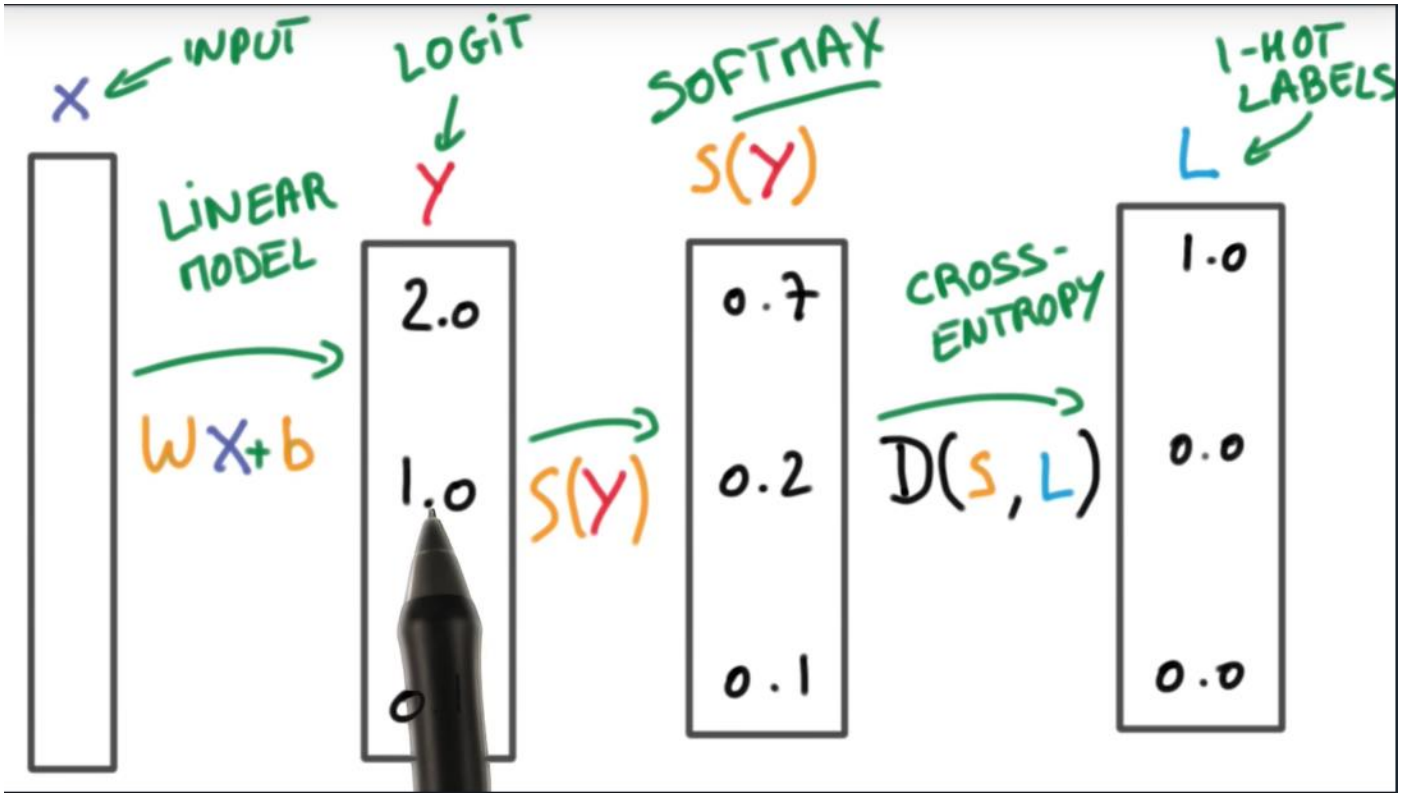
What happens to the softmax probabilities when you divide the logits by 10?

- The probabilities get close to 0.0 or 1.0
- ✓ The probabilities get close to the uniform distribution

SUBMIT


'ONE-HOT' ENCODING
A
0.7
0.2
0.1
1.0
0.0
0.0
a
b
c


CROSS-ENTROPY
S(Y)
L
0.7
0.2
0.1
1.0
0.0
0.0
D(S,L) = -∑ L_i log(S_i)
D(S,L) ≠ D(L,S)

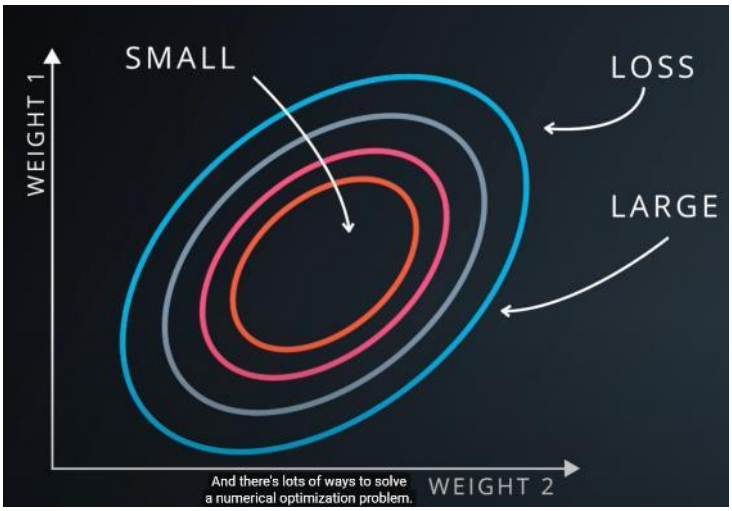

X
INPUT
LINEAR MODEL
WX+b
LOGIT
Y
2.0
1.0
0
S(Y)
SOFTMAX
S(Y)
0.7
0.2
0.1
CROSS-ENTROPY
D(S,L)
1-HOT LABELS
L
1.0
0.0
0.0

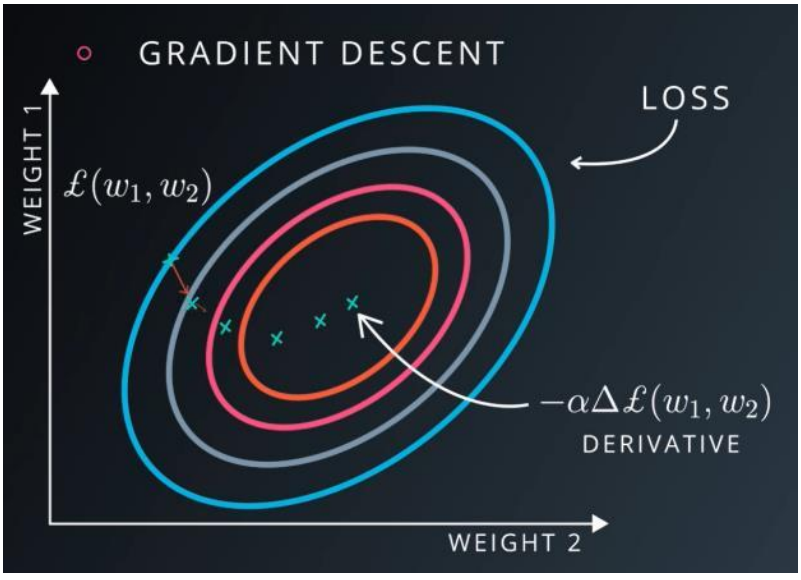
MULTINOMIAL LOGISTIC CLASSIFICATION

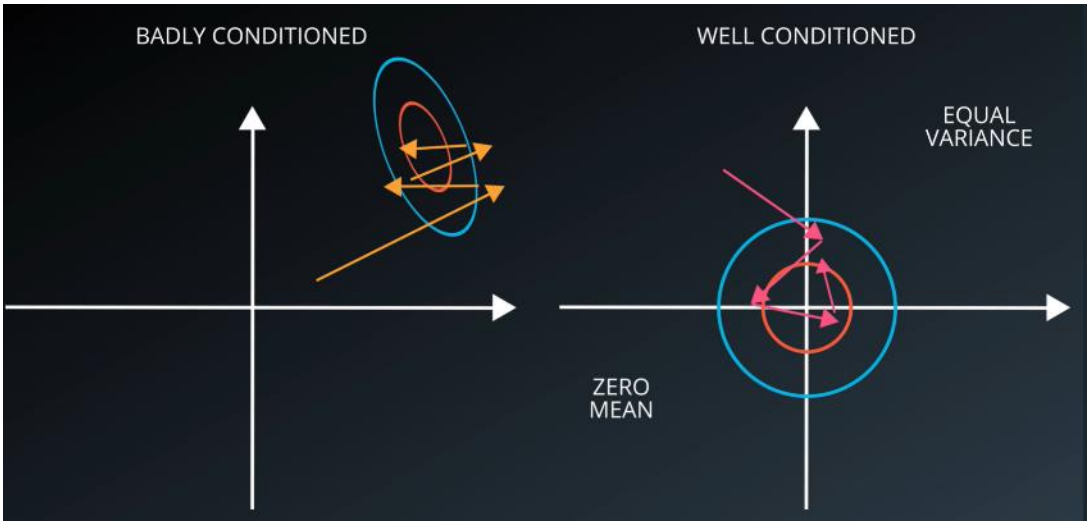
$$D(S(wx+b), L)$$

LOSS = AVERAGE CROSS-ENTROPY

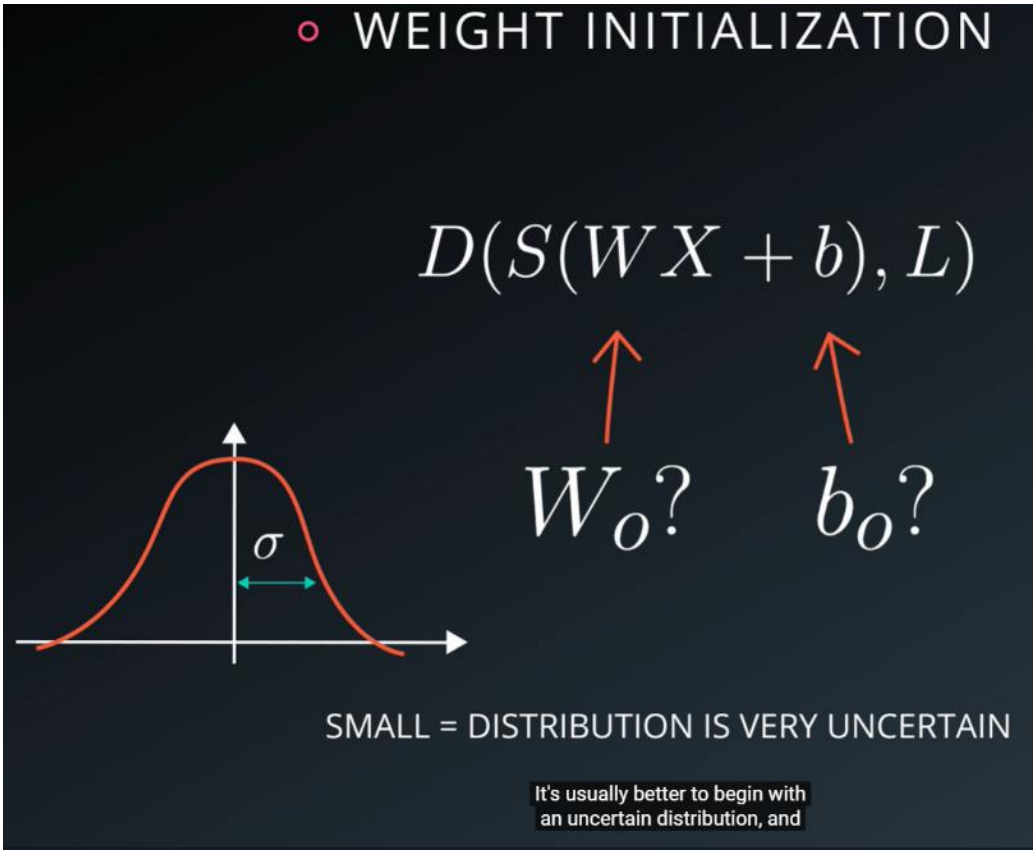
$$\mathcal{L} = \frac{1}{N} \sum_i D(S(wx_i + b), L_i)$$







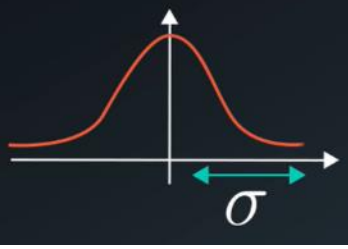
Much easier to the optimization to proceed numerically (normalization):



INITIALIZATION OF THE LOGIC CLASSIFIER

PIXELS - 128 / 128

$$\mathcal{L} = \frac{1}{N} \sum_i D(S(Wx_i + b), L_i)$$



OPTIMIZATION

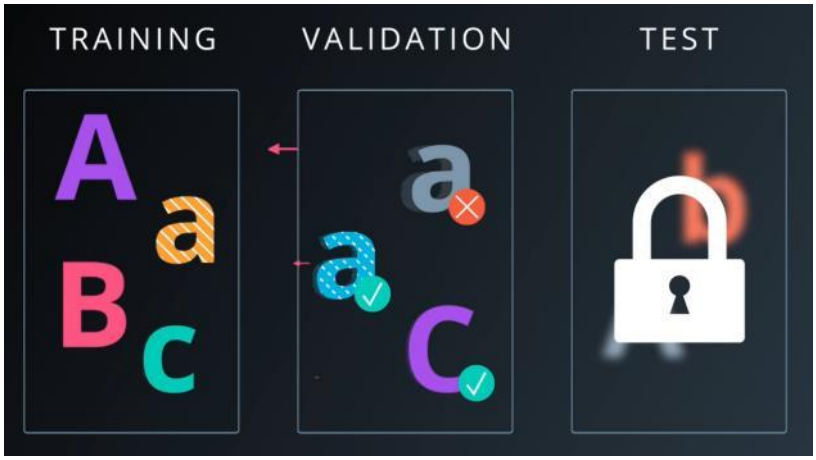
$$w \leftarrow w - \alpha \Delta_w \mathcal{L}$$

$$b \leftarrow b - \alpha \Delta_b \mathcal{L}$$

LOOP!

- TRAINING
- VALIDATION
- TESTING

Every classifier that is built, tends to memorize the training set which is negative for the real performance. When it sees a new sample, it does not know what to do. The point is to generalize the classifier to new data



The bigger the test set, the less noisy the result will be:

RULE OF '30'

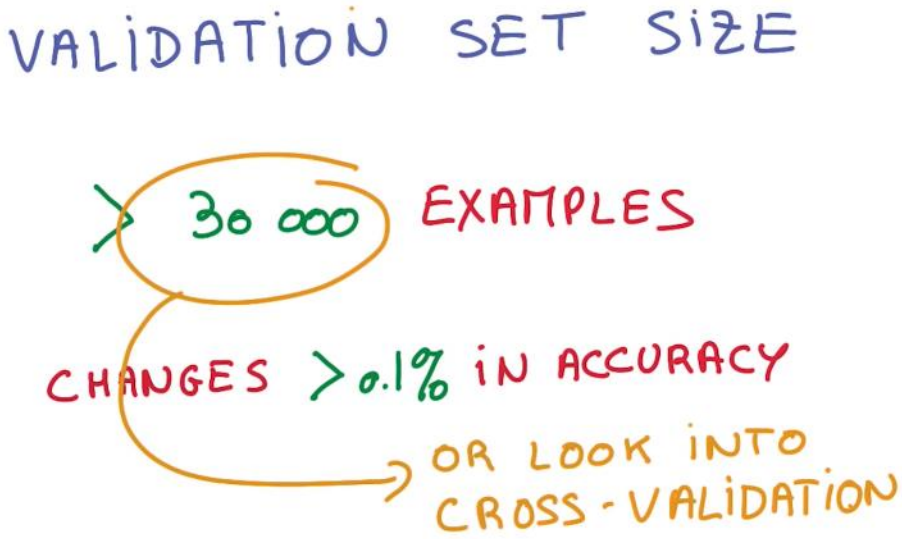
3000 EXAMPLES

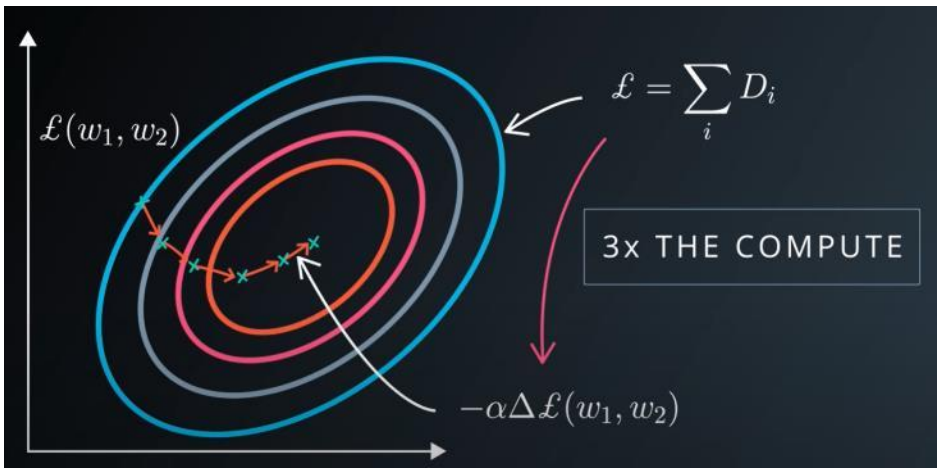
YES NO

$\frac{1 \times 3000}{100} = 30$ ✓ ○ 80% → 81%

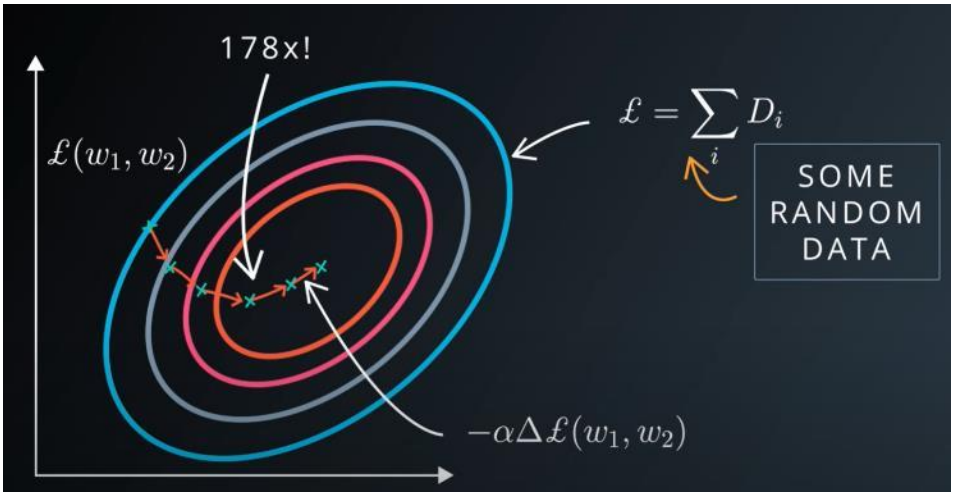
$\frac{0.5 \times 3000}{100} = 15$ ○ ✓ 80% → 80.5%

$\frac{0.1 \times 3000}{100} = 3$ ○ ✓ 80% → 80.1%

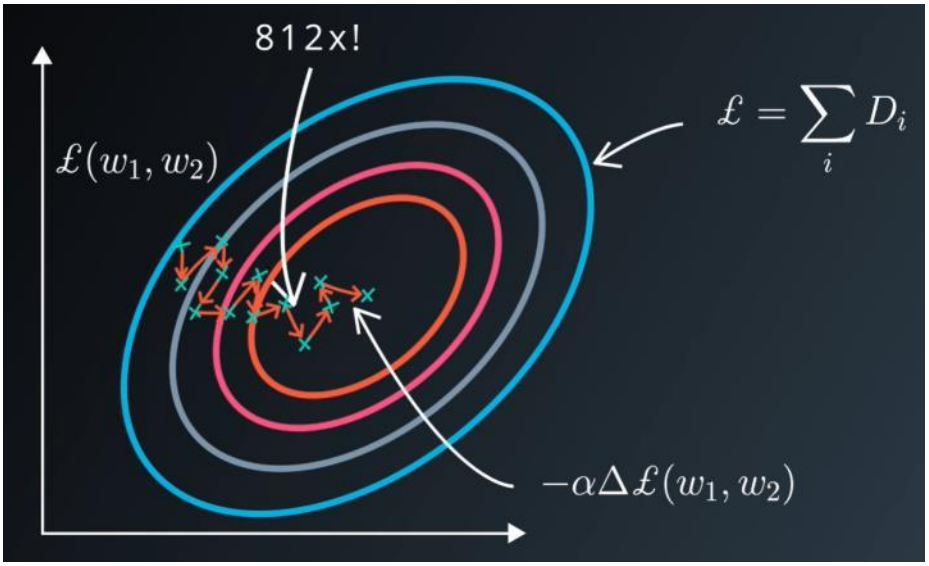


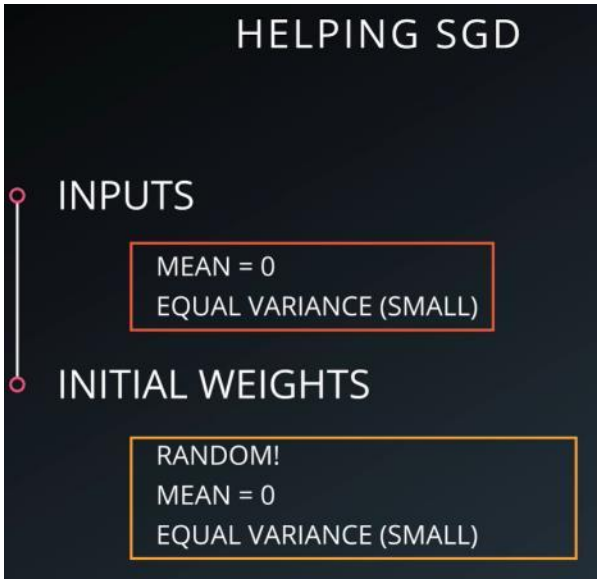


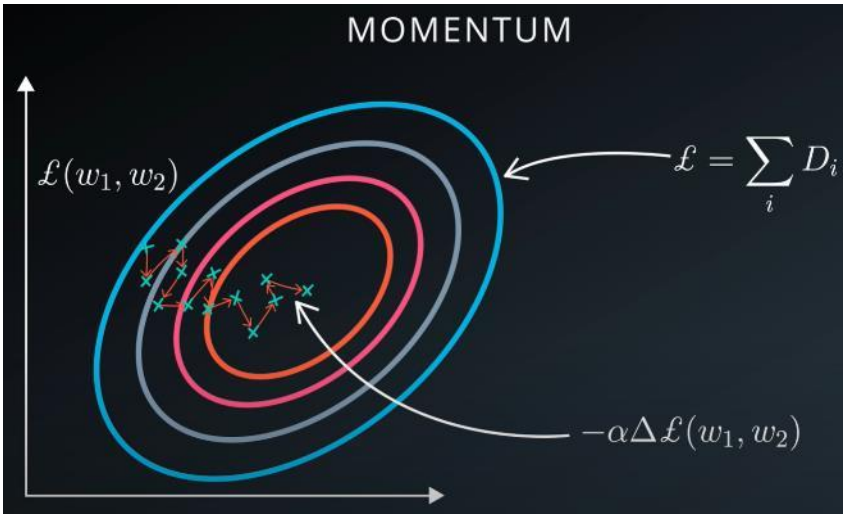
It is needed to feed all the data into L function. If this is iterated throughout all the required approximations, this is quite a heavy approach. Instead it is possible to compute an approximation per iteration

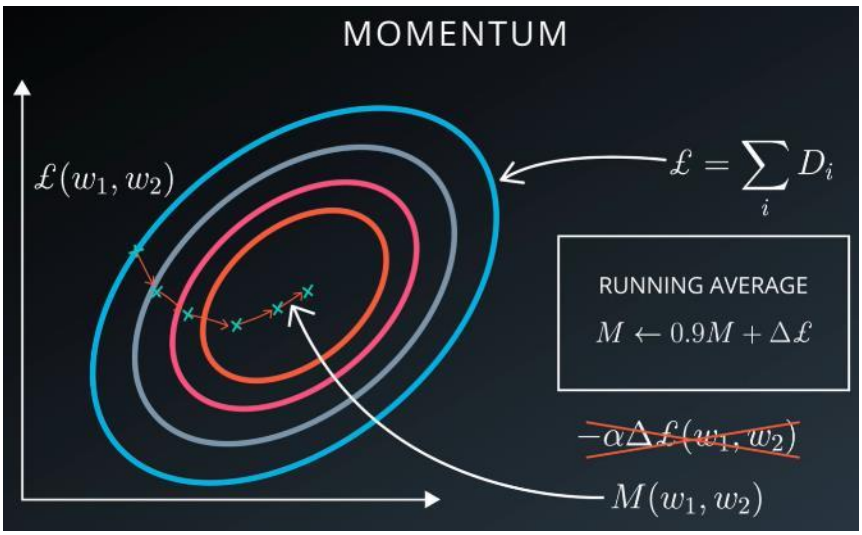


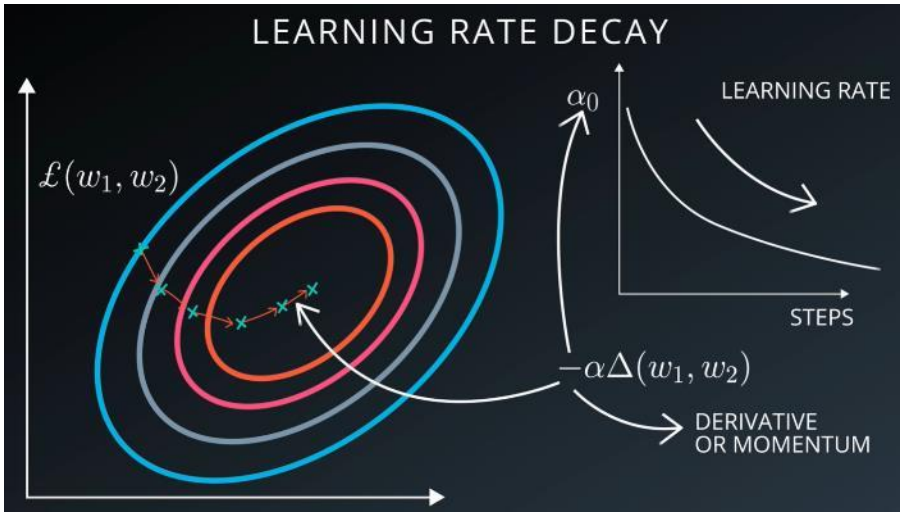
But this requires smaller steps → so more amount of epochs

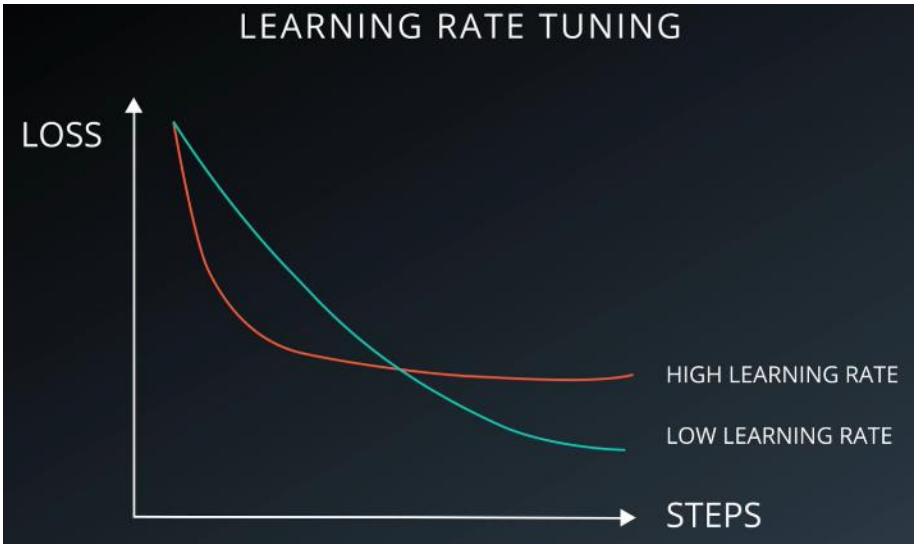


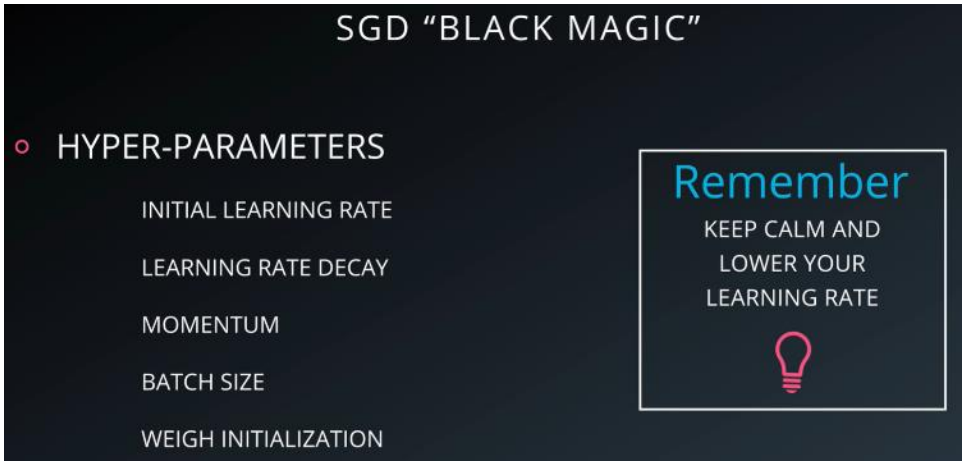

HELPING SGD
INPUTS
MEAN = 0
EQUAL VARIANCE (SMALL)
INITIAL WEIGHTS
RANDOM!
MEAN = 0
EQUAL VARIANCE (SMALL)


MOMENTUM
$\mathcal{L}(w_1, w_2)$
$\mathcal{L} = \sum_i D_i$
$-\alpha\Delta\mathcal{L}(w_1, w_2)$

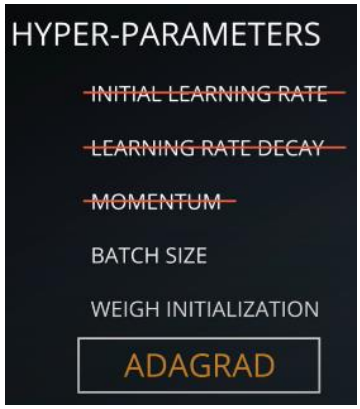

MOMENTUM
$\mathcal{L}(w_1, w_2)$
$\mathcal{L} = \sum_i D_i$
RUNNING AVERAGE
$M \leftarrow 0.9M + \Delta\mathcal{L}$
$-\alpha\Delta\mathcal{L}(w_1, w_2)$
$M(w_1, w_2)$







Modification of SGD: ADAGRAD



Mini-batching

In this section, you'll go over what mini-batching is and how to apply it in TensorFlow.

Mini-batching is a technique for training on subsets of the dataset instead of all the data at one time. This provides the ability to train a model, even if a computer lacks the memory to store the entire dataset.

Mini-batching is computationally inefficient, since you can't calculate the loss simultaneously across all samples. However, this is a small price to pay in order to be able to run the model at all.

It's also quite useful combined with SGD. The idea is to randomly shuffle the data at the start of each epoch, then create the mini-batches. For each mini-batch, you train the network weights with gradient descent. Since these batches are random, you're performing SGD with each batch.

train_features Shape: (55000, 784) Type: float32
train_labels Shape: (55000, 10) Type: float32
weights Shape: (784, 10) Type: float32
bias Shape: (10,) Type: float32

The total memory space required for the inputs, weights and bias is around 174 megabytes, which isn't that much memory. You could train this whole dataset on most CPUs and GPUs.

But larger datasets that you'll use in the future measured in gigabytes or more. It's possible to purchase more memory, but it's expensive. A Titan X GPU with 12 GB of memory costs over $1,000.

Instead, in order to run large models on your machine, you'll learn how to use mini-batching.

Let's look at how you implement mini-batching in TensorFlow.

TensorFlow Mini-batching

In order to use mini-batching, you must first divide your data into batches.

Unfortunately, it's sometimes impossible to divide the data into batches of exactly equal size. For example, imagine you'd like to create batches of 128 samples each from a dataset of 1000 samples. Since 128 does not evenly divide into 1000, you'd wind up with 7 batches of 128 samples, and 1 batch of 104 samples. (7*128 + 1*104 = 1000)

In that case, the size of the batches would vary, so you need to take advantage of TensorFlow's `tf.placeholder()` function to receive the varying batch sizes.

Continuing the example, if each sample had `n_input = 784` features and `n_classes = 10` possible labels, the dimensions for `features` would be `[None, n_input]` and `labels` would be `[None, n_classes]`.

```
# Features and Labels
features = tf.placeholder(tf.float32, [None, n_input])
labels = tf.placeholder(tf.float32, [None, n_classes])
```

What does `None` do here?

The `None` dimension is a placeholder for the batch size. At runtime, TensorFlow will accept any batch size greater than 0.

Going back to our earlier example, this setup allows you to feed `features` and `labels` into the model as either the batches of 128 samples or the single batch of 104 samples.

Question 2

Use the parameters below, how many batches are there, and what is the last batch size?

features is (50000, 400)

labels is (50000, 10)

batch_size is 128

How many batches are there?

391 RESET

What is the last batch size?

80 RESET

```
# 4 Samples of features
example_features = [
    ['F11','F12','F13','F14'],
    ['F21','F22','F23','F24'],
    ['F31','F32','F33','F34'],
    ['F41','F42','F43','F44']]
# 4 Samples of labels
example_labels = [
    ['L11','L12'],
    ['L21','L22'],
    ['L31','L32'],
    ['L41','L42']]

example_batches = batches(3, example_features, example_labels)
```

he `example_batches` variable would be the following:

```
[
    # 2 batches:
    #   First is a batch of size 3.
    #   Second is a batch of size 1
    [
        # First Batch is size 3
        [
            # 3 samples of features.
            # There are 4 features per sample.
            ['F11', 'F12', 'F13', 'F14'],
            ['F21', 'F22', 'F23', 'F24'],
            ['F31', 'F32', 'F33', 'F34']
        ], [
            # 3 samples of labels.
            # There are 2 labels per sample.
            ['L11', 'L12'],
            ['L21', 'L22'],
            ['L31', 'L32']
        ]
    ], [
        # Second Batch is size 1.
        # Since batch size is 3, there is only one sample left from the 4 samples.
        [
            # 1 sample of features.
            ['F41', 'F42', 'F43', 'F44']
        ], [
            # 1 sample of labels.
            ['L41', 'L42']
        ]
    ]
]
```

sandbox.py | quiz.py | quiz_solution.py

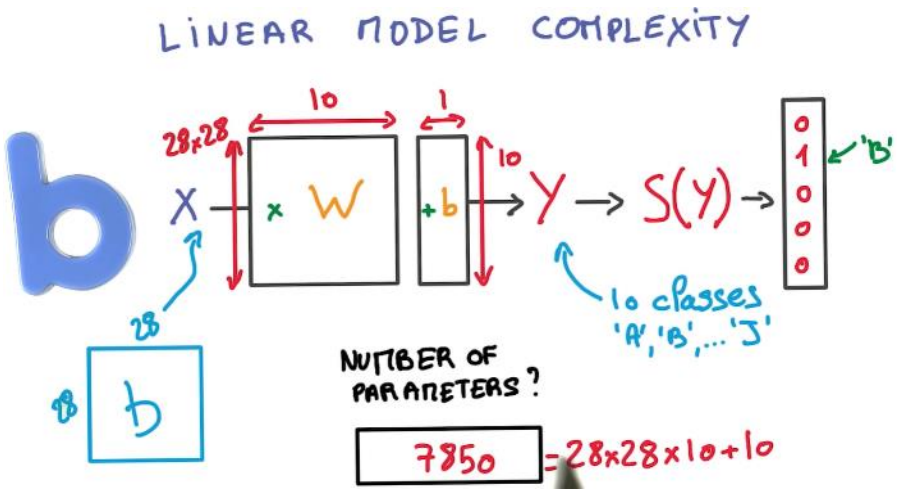
```python
import math
def batches(batch_size, features, labels):
    """
    Create batches of features and labels
    :param batch_size: The batch size
    :param features: List of features
    :param labels: List of labels
    :return: Batches of (Features, Labels)
    """
    assert len(features) == len(labels)
    # TODO: Implement batching

    output_batches = []
    sample_size = len(features)
    for start_i in range(0, sample_size, batch_size):
        end_i = start_i + batch_size
        batch = [features[start_i : end_i], labels[start_i : end_i]]
        output_batches.append(batch)
    return output_batches

```

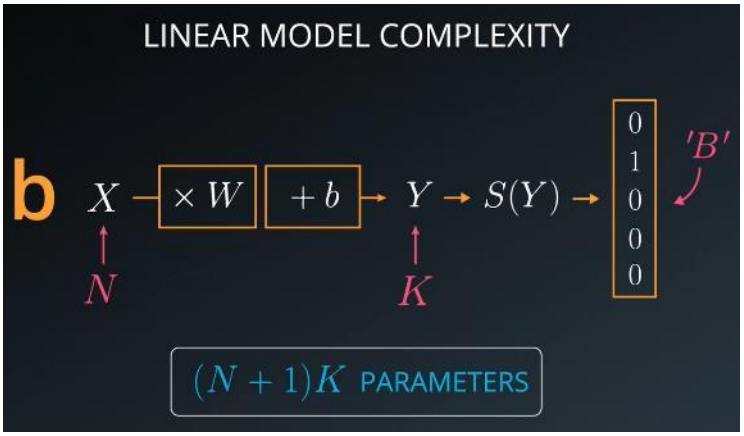
```
[[[['F11', 'F12', 'F13', 'F14'],
   ['F21', 'F22', 'F23', 'F24'],
   ['F31', 'F32', 'F33', 'F34']],
  [['L11', 'L12'], ['L21', 'L22'], ['L31', 'L32']]],
 [[['F41', 'F42', 'F43', 'F44']], [['L41', 'L42']]]]
```

An epoch is a single forward and backward pass of the whole dataset. This is used to increase the accuracy of the model without requiring more data.

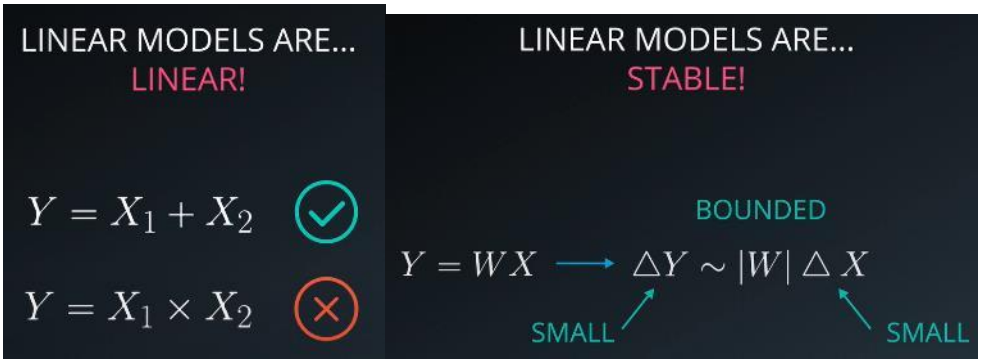
```
Epoch: 65   - Cost: 0.122   Valid Accuracy: 0.868
Epoch: 66   - Cost: 0.121   Valid Accuracy: 0.868
Epoch: 67   - Cost: 0.12    Valid Accuracy: 0.868
Epoch: 68   - Cost: 0.119   Valid Accuracy: 0.868
Epoch: 69   - Cost: 0.118   Valid Accuracy: 0.868
Epoch: 70   - Cost: 0.118   Valid Accuracy: 0.868
Epoch: 71   - Cost: 0.117   Valid Accuracy: 0.868
Epoch: 72   - Cost: 0.116   Valid Accuracy: 0.868
Epoch: 73   - Cost: 0.115   Valid Accuracy: 0.868
Epoch: 74   - Cost: 0.115   Valid Accuracy: 0.868
Epoch: 75   - Cost: 0.114   Valid Accuracy: 0.868
Epoch: 76   - Cost: 0.113   Valid Accuracy: 0.868
Epoch: 77   - Cost: 0.113   Valid Accuracy: 0.868
Epoch: 78   - Cost: 0.112   Valid Accuracy: 0.868
Epoch: 79   - Cost: 0.111   Valid Accuracy: 0.868
Epoch: 80   - Cost: 0.111   Valid Accuracy: 0.869
Test Accuracy: 0.8690999418258667
```

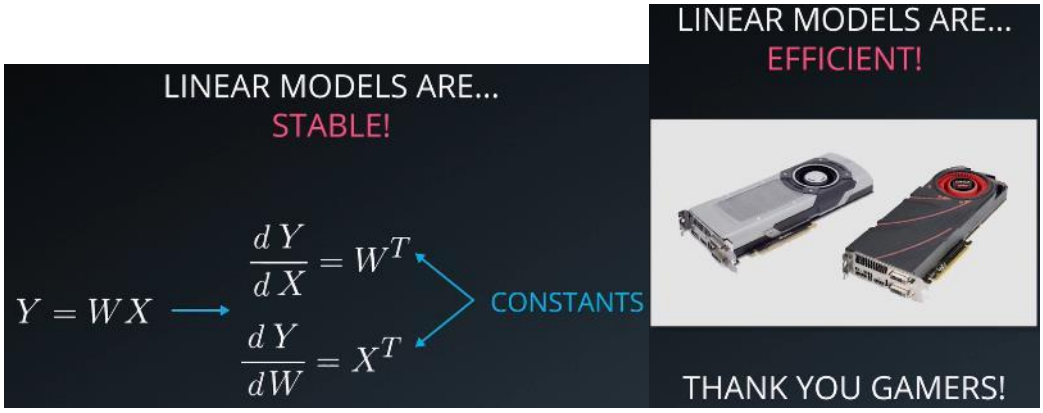
The accuracy only reached 0.86, but that could be because the learning rate was too high. Lowering the learning rate would require more epochs, but could ultimately achieve better accuracy.

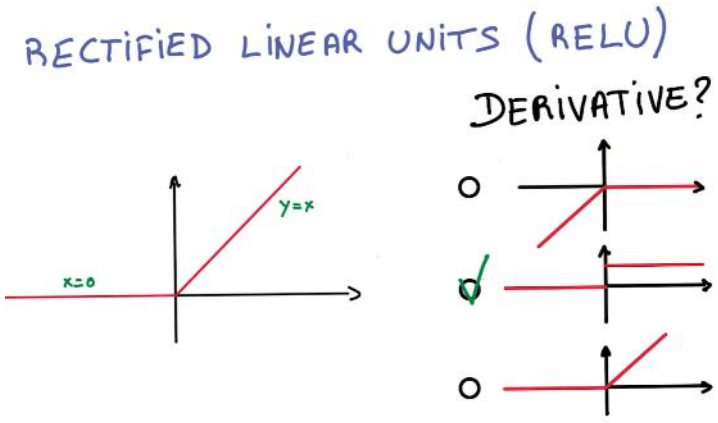




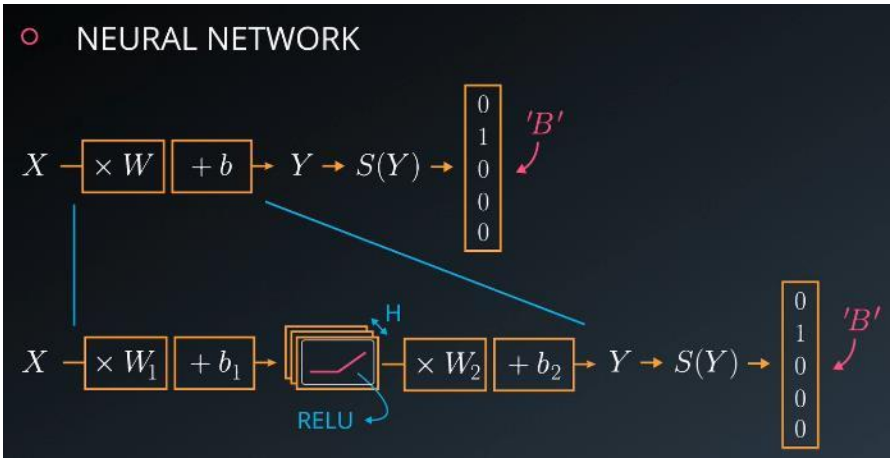
N inputs, K outputs

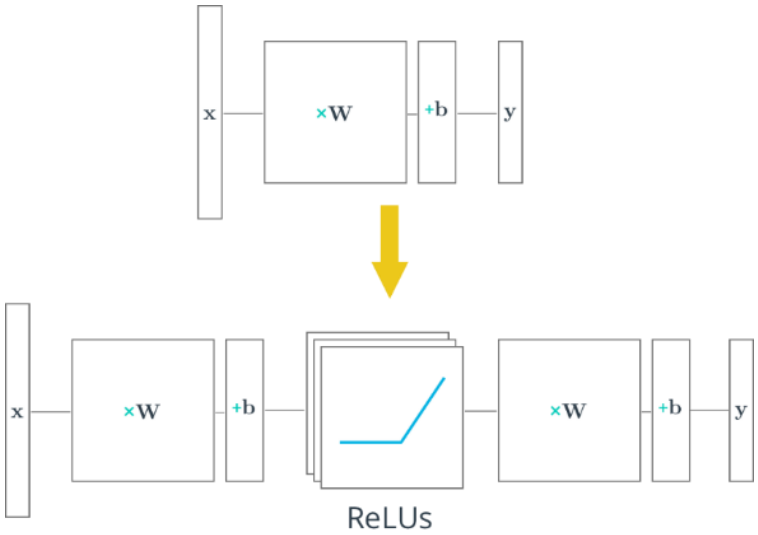






This is how you get a deep NN





Multilayer Neural Networks

In this lesson, you'll learn how to build multilayer neural networks with TensorFlow. Adding a hidden layer to a network allows it to model more complex functions. Also, using a non-linear activation function on the hidden layer lets it model non-linear functions.

Next, you'll see how a ReLU hidden layer is implemented in TensorFlow.

Note: Depicted above is a "2-layer" neural network:

1. The first layer effectively consists of the set of weights and biases applied to X and passed through ReLUs. The output of this layer is fed to the next one, but is not observable outside the network, hence it is known as a *hidden layer*.
2. The second layer consists of the weights and biases applied to these intermediate outputs, followed by the softmax function to generate probabilities.

A Rectified linear unit (ReLU) is type of **activation function** that is defined as `f(x) = max(0, x)`. The function returns 0 if `x` is negative, otherwise it returns `x`. TensorFlow provides the ReLU function as `tf.nn.relu()`, as shown below.

```
# Hidden Layer with ReLU activation function
hidden_layer = tf.add(tf.matmul(features, hidden_weights), hidden_biases)
hidden_layer = tf.nn.relu(hidden_layer)

output = tf.add(tf.matmul(hidden_layer, output_weights), output_biases)
```

The above code applies the `tf.nn.relu()` function to the `hidden_layer`, effectively turning off any negative weights and acting like an on/off switch. Adding additional layers, like the `output` layer, after an activation function turns the model into a nonlinear function. This nonlinearity allows the network to solve more complex problems.

```
import tensorflow as tf

output = None
hidden_layer_weights = [
    [0.1, 0.2, 0.4],
    [0.4, 0.6, 0.6],
    [0.5, 0.9, 0.1],
    [0.8, 0.2, 0.8]]
out_weights = [
    [0.1, 0.6],
    [0.2, 0.1],
    [0.7, 0.9]]
```

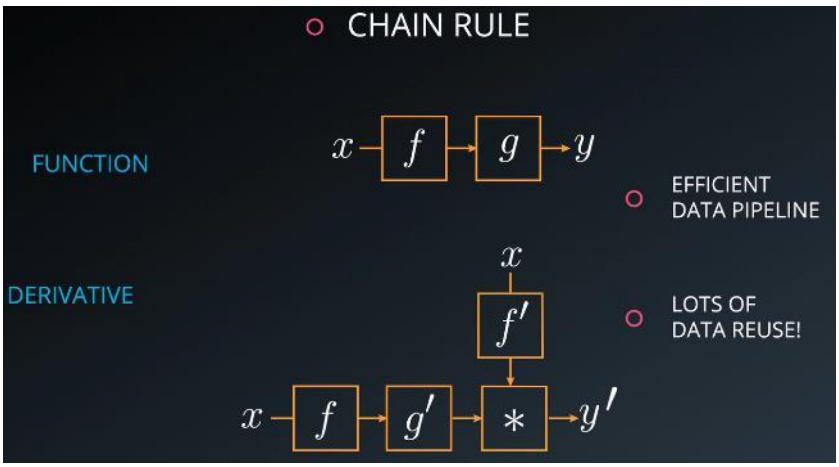
```
# Weights and biases
weights = [
    tf.Variable(hidden_layer_weights),
    tf.Variable(out_weights)]
biases = [
    tf.Variable(tf.zeros(3)),
    tf.Variable(tf.zeros(2))]
```

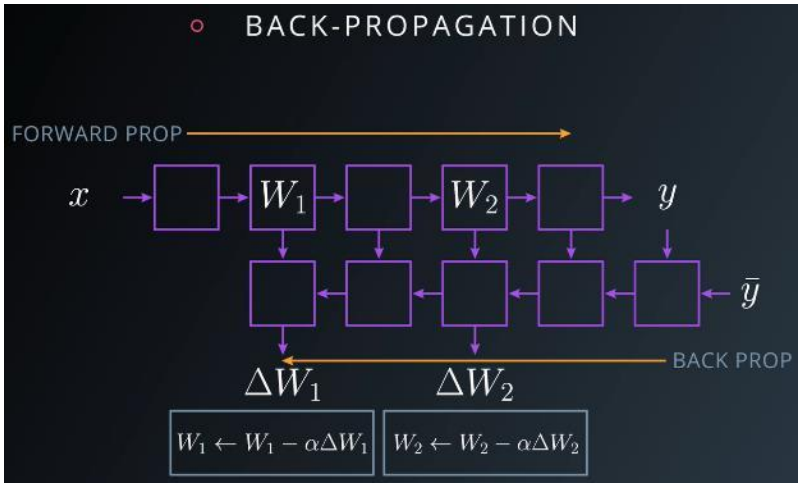
```
# Input
features = tf.Variable([[1.0, 2.0, 3.0, 4.0], [-1.0, -2.0, -3.0, -4.0], [11.0, 1
2.0, 13.0, 14.0]])
```

```
# TODO: Create Model
hidden_layer = tf.add(tf.matmul(features, weights[0]), biases[0])
hidden_layer = tf.nn.relu(hidden_layer)

logits = tf.add((tf.matmul(hidden_layer, weights[1])), biases[1])
```

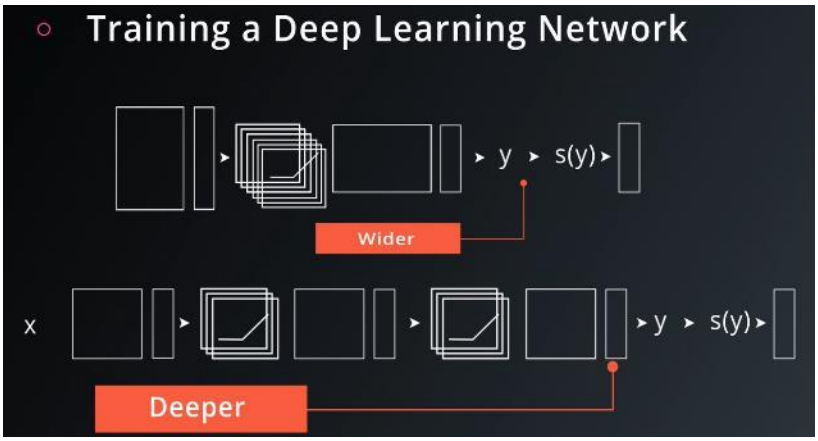
```
# TODO: save and print session results on a variable named "output"
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    output = sess.run(logits)
    print(output)
```

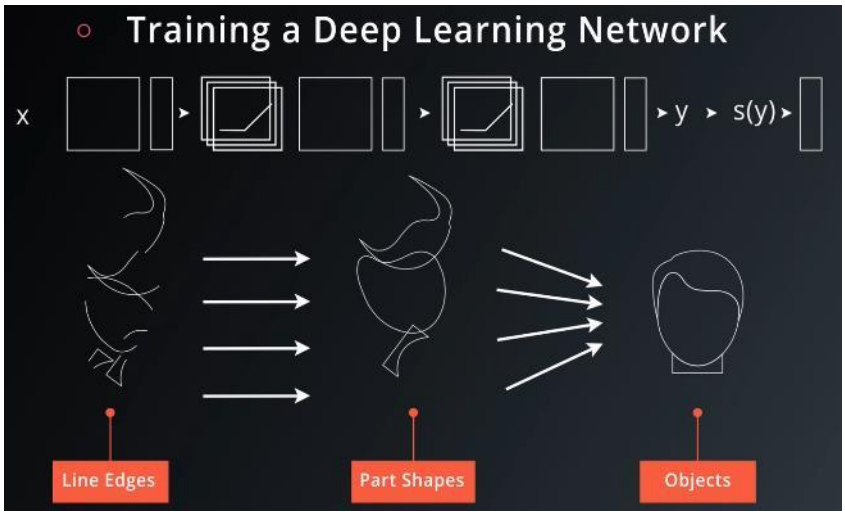
```
[[  5.11000013   8.44000053]
 [  0.           0.        ]
 [ 24.01000214  38.23999786]]
```





Fewer parameters by going deeper rather than wider





Save and Restore TensorFlow Models

Training a model can take hours. But once you close your TensorFlow session, you lose all the trained weights and biases. If you were to reuse the model in the future, you would have to train it all over again!

Fortunately, TensorFlow gives you the ability to save your progress using a class called `tf.train.Saver`. This class provides the functionality to save any `tf.Variable` to your file system.

Saving Variables

Let's start with a simple example of saving `weights` and `bias` Tensors. For the first example you'll just save two variables. Later examples will save all the weights in a practical model.

```
import tensorflow as tf

# The file path to save the data
save_file = './model.ckpt'

# Two Tensor Variables: weights and bias
weights = tf.Variable(tf.truncated_normal([2, 3]))
bias = tf.Variable(tf.truncated_normal([3]))

# Class used to save and/or restore Tensor Variables
saver = tf.train.Saver()

with tf.Session() as sess:
    # Initialize all the Variables
    sess.run(tf.global_variables_initializer())

    # Show the values of weights and bias
    print('Weights:')
    print(sess.run(weights))
    print('Bias:')
    print(sess.run(bias))

    # Save the model
    saver.save(sess, save_file)
```

> Weights:
>
> [[-0.97990924 1.03016174 0.74119264]
>
> [-0.82581609 -0.07361362 -0.86653847]]
>
> Bias:
>
> [1.62978125 -0.37812829 0.64723819]

The Tensors `weights` and `bias` are set to random values using the `tf.truncated_normal()` function. The values are then saved to the `save_file` location, "model.ckpt", using the `tf.train.Saver.save()` function. (The ".ckpt" extension stands for "checkpoint".)

If you're using TensorFlow 0.11.0RC1 or newer, a file called "model.ckpt.meta" will also be created. This file contains the TensorFlow graph.

Loading Variables

Now that the Tensor Variables are saved, let's load them back into a new model.

```
# Remove the previous weights and bias
tf.reset_default_graph()

# Two Variables: weights and bias
weights = tf.Variable(tf.truncated_normal([2, 3]))
bias = tf.Variable(tf.truncated_normal([3]))

# Class used to save and/or restore Tensor Variables
saver = tf.train.Saver()

with tf.Session() as sess:
    # Load the weights and bias
    saver.restore(sess, save_file)

    # Show the values of weights and bias
    print('Weight:')
    print(sess.run(weights))
    print('Bias:')
    print(sess.run(bias))
```

> Weights:
>
> [[-0.97990924 1.03016174 0.74119264]
>
> [-0.82581609 -0.07361362 -0.86653847]]
>
> Bias:
>
> [1.62978125 -0.37812829 0.64723819]

Save a Trained Model

Let's see how to train a model and save its weights.

First start with a model:

```
# Remove previous Tensors and Operations
tf.reset_default_graph()

from tensorflow.examples.tutorials.mnist import input_data
import numpy as np

learning_rate = 0.001
n_input = 784  # MNIST data input (img shape: 28*28)
n_classes = 10  # MNIST total classes (0-9 digits)

# Import MNIST data
mnist = input_data.read_data_sets('.', one_hot=True)

# Features and Labels
features = tf.placeholder(tf.float32, [None, n_input])
labels = tf.placeholder(tf.float32, [None, n_classes])

# Weights & bias
weights = tf.Variable(tf.random_normal([n_input, n_classes]))
bias = tf.Variable(tf.random_normal([n_classes]))

# Logits - xW + b
logits = tf.add(tf.matmul(features, weights), bias)

# Define loss and optimizer
cost = tf.reduce_mean(\
    tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=labels))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)\
    .minimize(cost)

# Calculate accuracy
correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(labels, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Let's train that model, then save the weights:

```
import math

save_file = './train_model.ckpt'
batch_size = 128
n_epochs = 100

saver = tf.train.Saver()

# Launch the graph
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    # Training cycle
    for epoch in range(n_epochs):
        total_batch = math.ceil(mnist.train.num_examples / batch_size)

        # Loop over all batches
        for i in range(total_batch):
            batch_features, batch_labels = mnist.train.next_batch(batch_size)
            sess.run(
                optimizer,
                feed_dict={features: batch_features, labels: batch_labels})

        # Print status for every 10 epochs
        if epoch % 10 == 0:
            valid_accuracy = sess.run(
                accuracy,
                feed_dict={
                    features: mnist.validation.images,
                    labels: mnist.validation.labels})
            print('Epoch {:<3} - Validation Accuracy: {}'.format(
                epoch,
                valid_accuracy))

    # Save the model
    saver.save(sess, save_file)
    print('Trained Model Saved.')
```

Load a Trained Model

Let's load the weights and bias from memory, then check the test accuracy.

```
saver = tf.train.Saver()

# Launch the graph
with tf.Session() as sess:
    saver.restore(sess, save_file)

    test_accuracy = sess.run(
        accuracy,
        feed_dict={features: mnist.test.images, labels: mnist.test.labels})

print('Test Accuracy: {}'.format(test_accuracy))
```

> Test Accuracy: 0.7229999899864197

Loading the Weights and Biases into a New Model

Sometimes you might want to adjust, or "finetune" a model that you have already trained and saved.

However, loading saved Variables directly into a modified model can generate errors. Let's go over how to avoid these problems.

Naming Error

TensorFlow uses a string identifier for Tensors and Operations called `name`. If a name is not given, TensorFlow will create one automatically. TensorFlow will give the first node the name `<Type>`, and then give the name `<Type>_<number>` for the subsequent nodes. Let's see how this can affect loading a model with a different order of `weights` and `bias`:

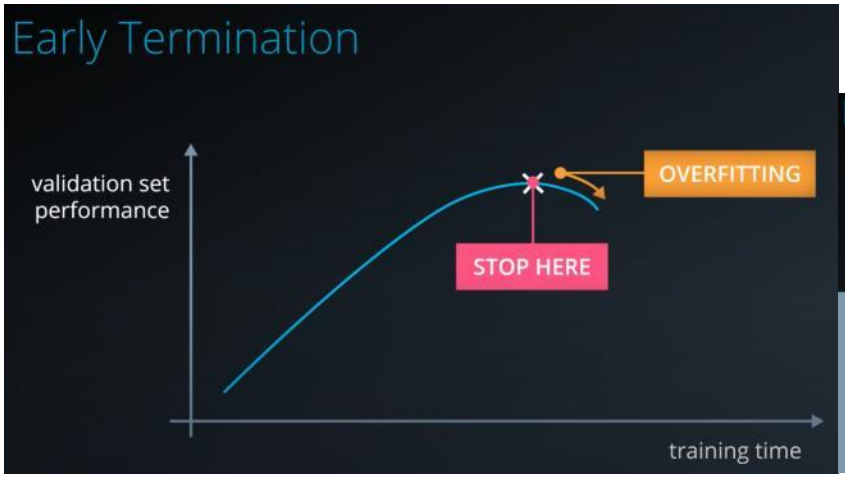
```
import tensorflow as tf

# Remove the previous weights and bias
tf.reset_default_graph()

save_file = 'model.ckpt'

# Two Tensor Variables: weights and bias
weights = tf.Variable(tf.truncated_normal([2, 3]))
bias = tf.Variable(tf.truncated_normal([3]))

saver = tf.train.Saver()

# Print the name of Weights and Bias
print('Save Weights: {}'.format(weights.name))
print('Save Bias: {}'.format(bias.name))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver.save(sess, save_file)

# Remove the previous weights and bias
tf.reset_default_graph()

# Two Variables: weights and bias
bias = tf.Variable(tf.truncated_normal([3]))
weights = tf.Variable(tf.truncated_normal([2, 3]))

saver = tf.train.Saver()

# Print the name of Weights and Bias
print('Load Weights: {}'.format(weights.name))
print('Load Bias: {}'.format(bias.name))

with tf.Session() as sess:
    # Load the weights and bias - ERROR
    saver.restore(sess, save_file)
```

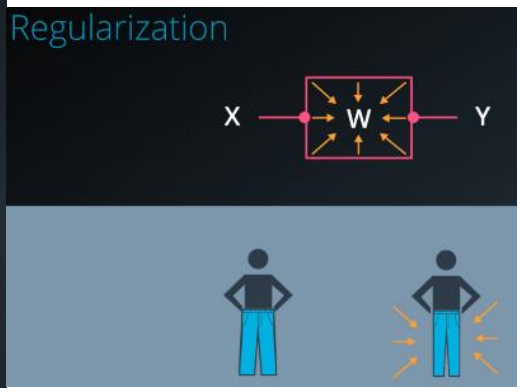
The code above prints out the following:

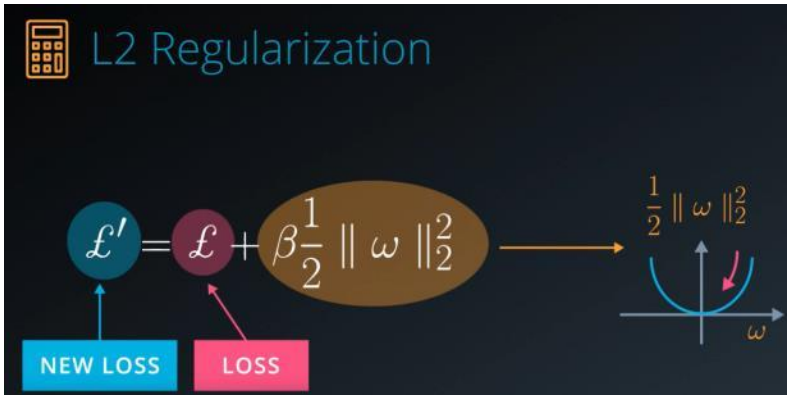
You'll notice that the `name` properties for `weights` and `bias` are different than when you saved the model. This is why the code produces the "Assign requires shapes of both tensors to match" error. The code `saver.restore(sess, save_file)` is trying to load weight data into `bias` and bias data into `weights`.

Instead of letting TensorFlow set the `name` property, let's set it manually:

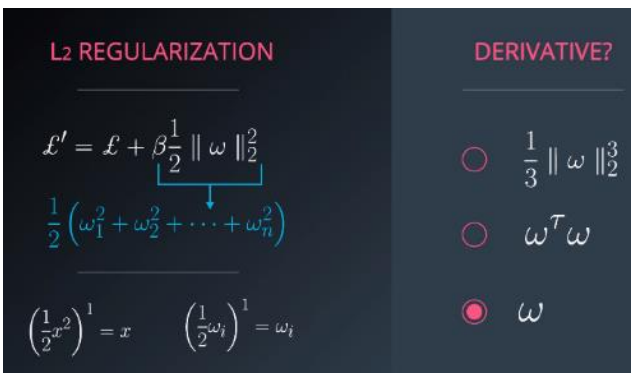
```
import tensorflow as tf

tf.reset_default_graph()

save_file = 'model.ckpt'

# Two Tensor Variables: weights and bias
weights = tf.Variable(tf.truncated_normal([2, 3]), name='weights_0')
bias = tf.Variable(tf.truncated_normal([3]), name='bias_0')

saver = tf.train.Saver()

# Print the name of Weights and Bias
print('Save Weights: {}'.format(weights.name))
print('Save Bias: {}'.format(bias.name))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver.save(sess, save_file)

# Remove the previous weights and bias
tf.reset_default_graph()

# Two Variables: weights and bias
bias = tf.Variable(tf.truncated_normal([3]), name='bias_0')
weights = tf.Variable(tf.truncated_normal([2, 3]) ,name='weights_0')

saver = tf.train.Saver()

# Print the name of Weights and Bias
print('Load Weights: {}'.format(weights.name))
print('Load Bias: {}'.format(bias.name))

with tf.Session() as sess:
    # Load the weights and bias - No Error
    saver.restore(sess, save_file)

print('Loaded Weights and Bias successfully.')
```

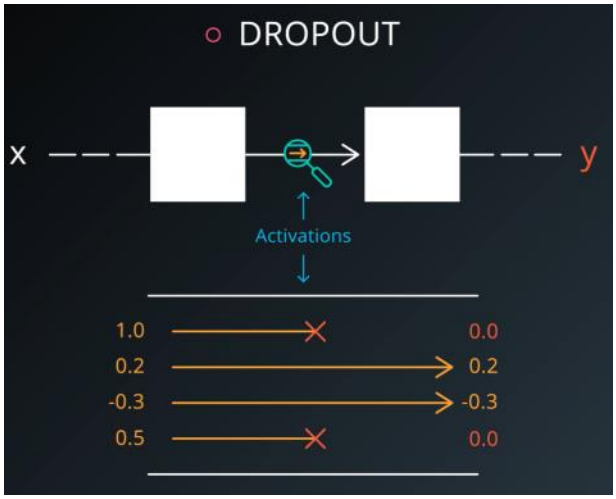


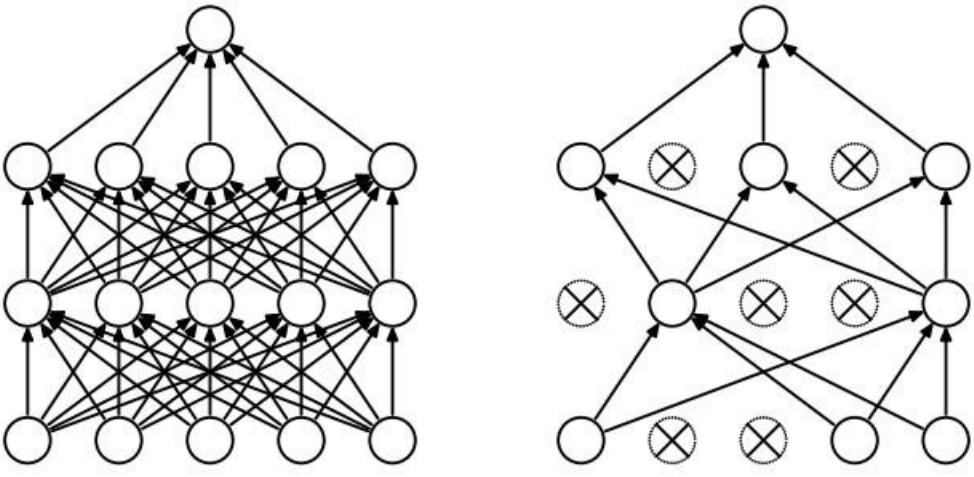




It penalizes large weights







(a) Standard Neural Net

(b) After applying dropout.

Dropout is a regularization technique for reducing overfitting. The technique temporarily drops units (artificial neurons) from the network, along with all of those units' incoming and outgoing connections. Figure 1 illustrates how dropout works.

TensorFlow provides the `tf.nn.dropout()` function, which you can use to implement dropout.

Let's look at an example of how to use `tf.nn.dropout()`.

```
keep_prob = tf.placeholder(tf.float32) # probability to keep units

hidden_layer = tf.add(tf.matmul(features, weights[0]), biases[0])
hidden_layer = tf.nn.relu(hidden_layer)
hidden_layer = tf.nn.dropout(hidden_layer, keep_prob)

logits = tf.add(tf.matmul(hidden_layer, weights[1]), biases[1])
```

The code above illustrates how to apply dropout to a neural network.

The `tf.nn.dropout()` function takes in two parameters:

1. `hidden_layer`: the tensor to which you would like to apply dropout
2. `keep_prob`: the probability of keeping (i.e. *not* dropping) any given unit

`keep_prob` allows you to adjust the number of units to drop. In order to compensate for dropped units, `tf.nn.dropout()` multiplies all units that are kept (i.e. *not* dropped) by `1/keep_prob`.

During training, a good starting value for `keep_prob` is `0.5`.

During testing, use a `keep_prob` value of `1.0` to keep all units and maximize the power of the model.

You should only drop units while training the model. During validation or testing, you should keep all of the units to maximize accuracy

```
import tensorflow as tf
from test import *
tf.set_random_seed(123456)


hidden_layer_weights = [
    [0.1, 0.2, 0.4],
    [0.4, 0.6, 0.6],
    [0.5, 0.9, 0.1],
    [0.8, 0.2, 0.8]]
out_weights = [
    [0.1, 0.6],
    [0.2, 0.1],
    [0.7, 0.9]]
```

In [21]:
```
# set random seed
tf.set_random_seed(123456)
```

In [22]:
```
# Weights and biases
weights = [
    tf.Variable(hidden_layer_weights),
    tf.Variable(out_weights)]
biases = [
    tf.Variable(tf.zeros(3)),
    tf.Variable(tf.zeros(2))]
```

In [23]:
```
# Input
features = tf.Variable([[0.0, 2.0, 3.0, 4.0], [0.1, 0.2, 0.3, 0.4]
```

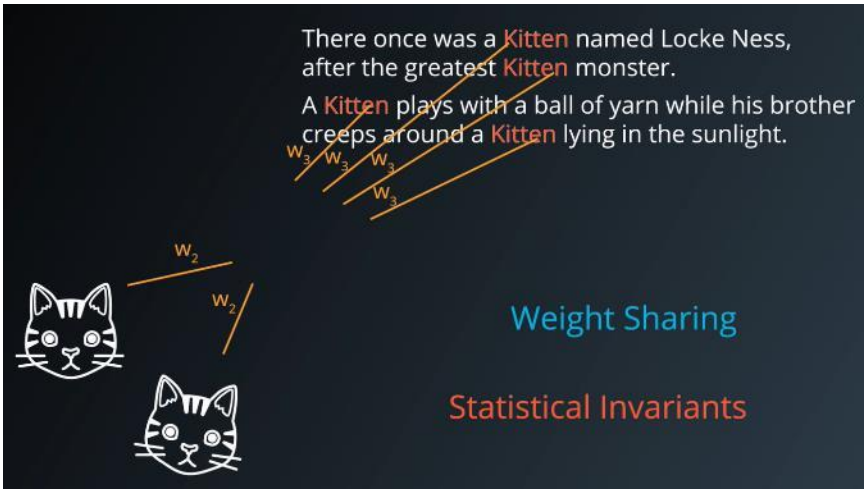
In [24]:
```
# TODO: Create Model with Dropout
keep_prob = tf.placeholder(tf.float32)
hidden_layer = tf.add(tf.matmul(features, weights[0]), biases[0])
hidden_layer = tf.nn.relu(hidden_layer)
hidden_layer = tf.nn.dropout(hidden_layer, keep_prob)

logits = tf.add(tf.matmul(hidden_layer, weights[1]), biases[1])
```

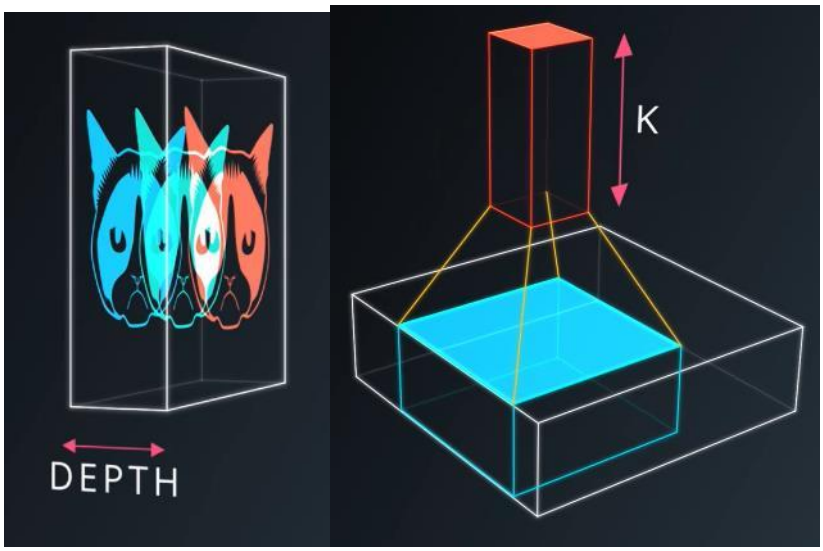

In [25]:
```
# TODO: save and print session results as variable named "output"

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    output = sess.run(logits, feed_dict = {keep_prob : 0.5})
    print(output)
```

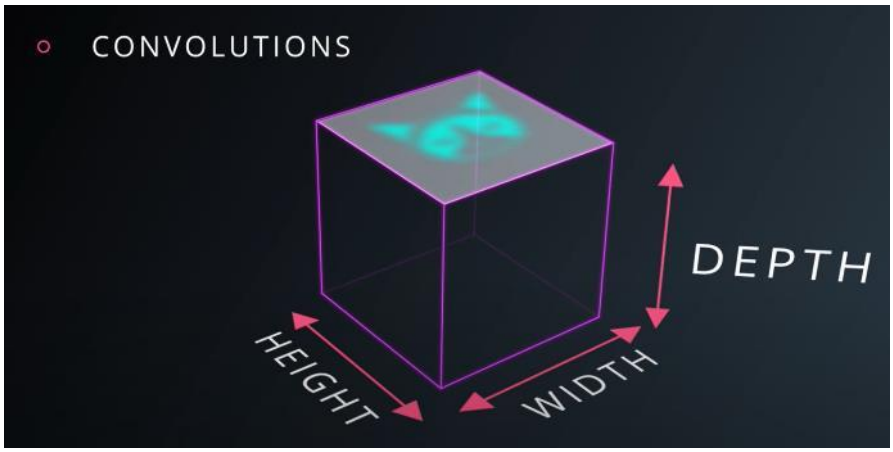
```
[[  9.55999947  16.        ]
 [  0.91000009   1.01600003]
 [  0.           0.        ]]
```

If your data has some structure, and your network does not have to learn such structure from scratch… it is going to perform better

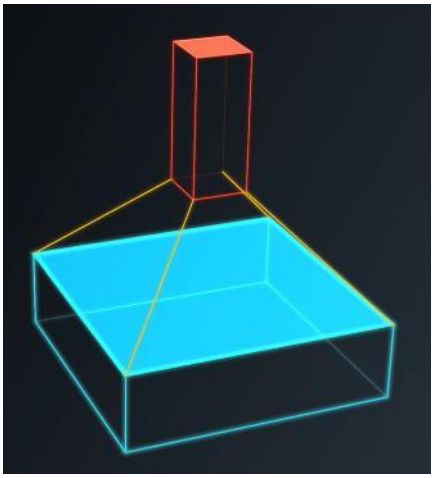
If there is a feature of the data that is irrelevant, it should be removed so the learning is easier

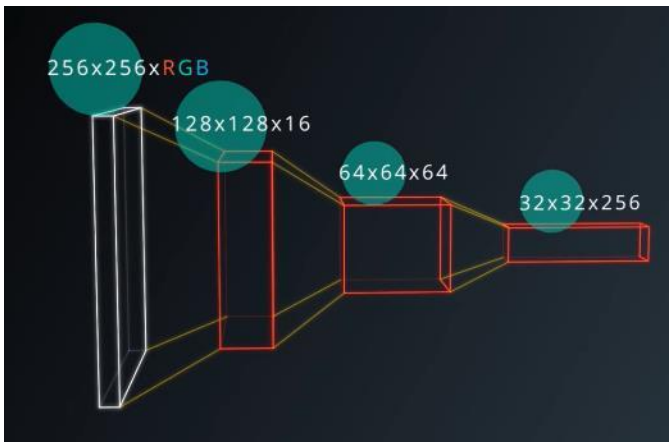


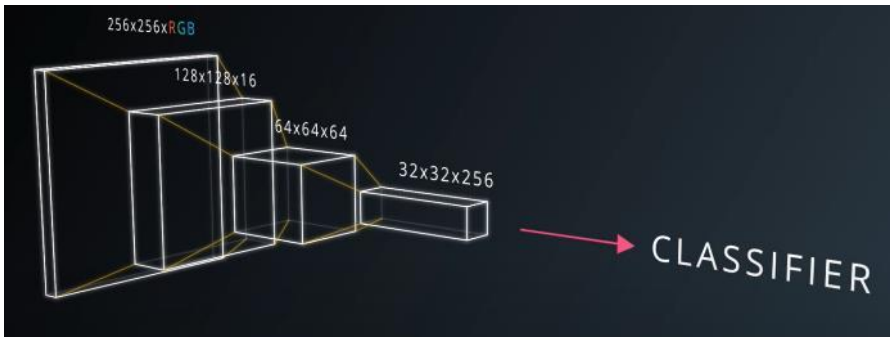




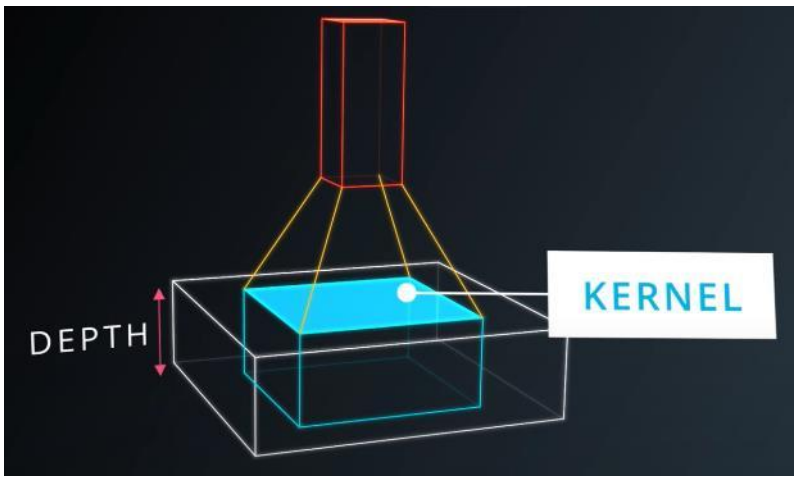
If the patch size == image size, it would be a regular network



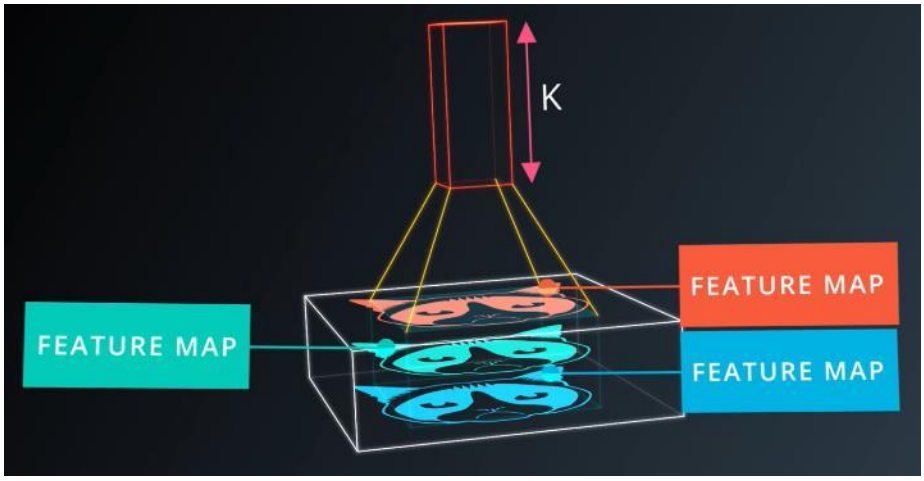


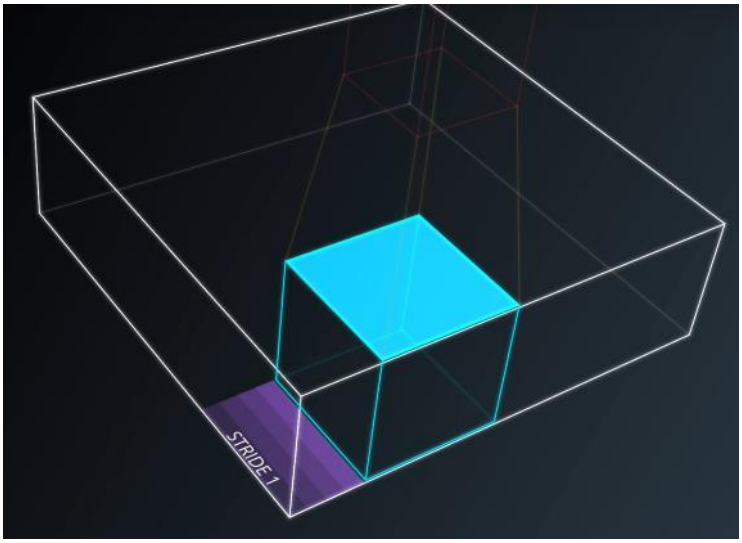


Patch = Kernel

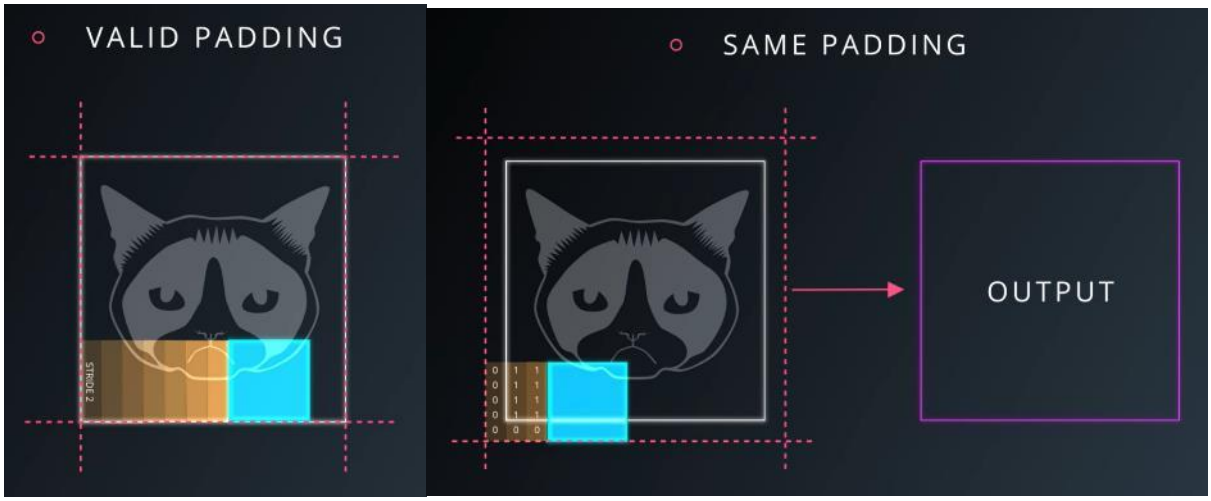


Mapping 3 feature maps into K feature maps





Stride 1: makes output size roughly the original size
Stride 2: makes output size double the original size



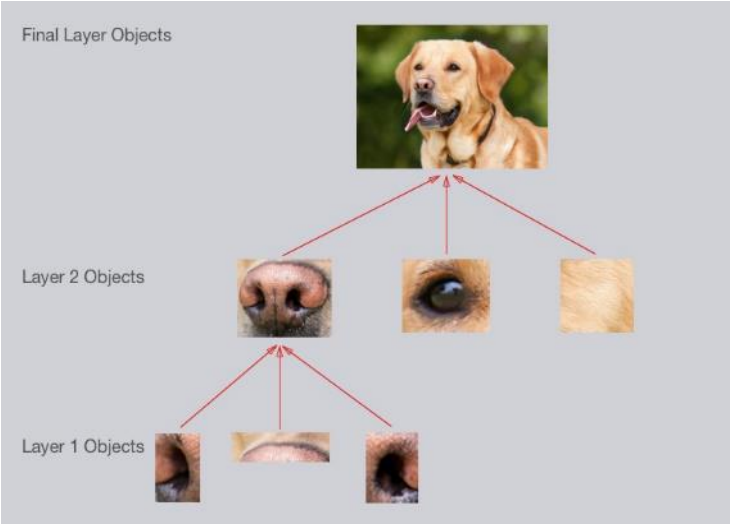
Broadly speaking, this is what a CNN learns to do. It learns to recognize basic lines and curves, then shapes and blobs, and then increasingly complex objects within the image. Finally, the CNN classifies the image by combining the larger, more complex objects.

In our case, the levels in the hierarchy are:

- Simple shapes, like ovals and dark circles
- Complex objects (combinations of simple shapes), like eyes, nose, and fur
- The dog as a whole (a combination of complex objects)

With deep learning, we don't actually program the CNN to recognize these specific features. Rather, the CNN learns on its own to recognize such objects through forward propagation and backpropagation!

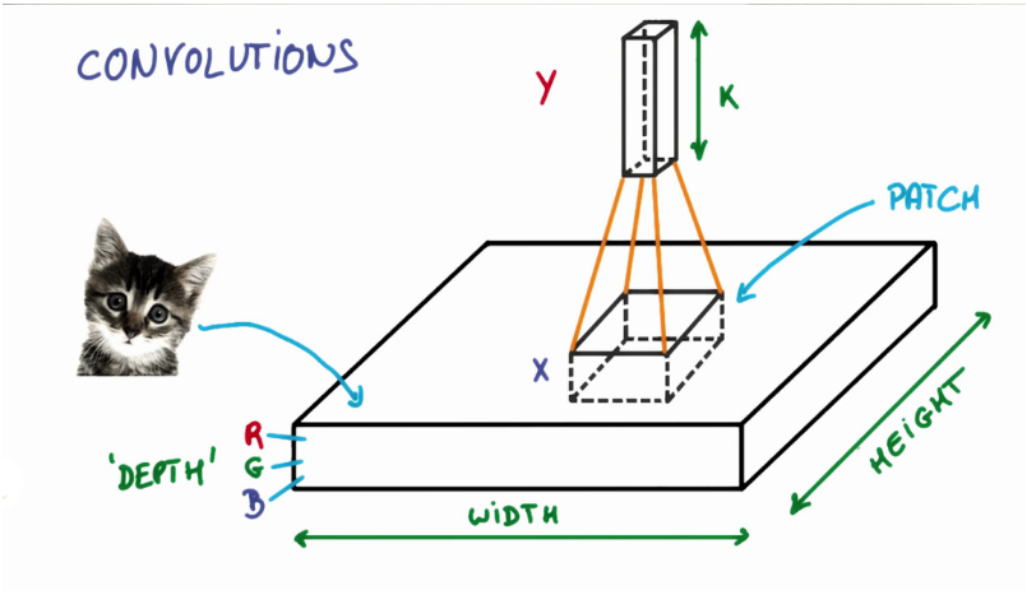
It's amazing how well a CNN can learn to classify images, even though we never program the CNN with information about specific features to look for.



Breaking up an Image

The first step for a CNN is to break up the image into smaller pieces. We do this by selecting a width and height that defines a filter.

The filter looks at small pieces, or patches, of the image. These patches are the same size as the filter.



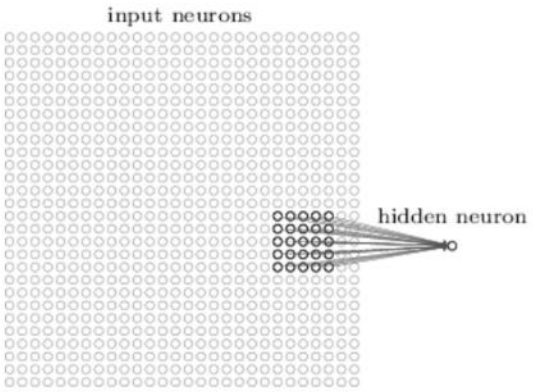
As shown in the previous video, a CNN uses filters to split an image into smaller patches. The size of these patches matches the filter size.

We then simply slide this filter horizontally or vertically to focus on a different piece of the image.

The amount by which the filter slides is referred to as the 'stride'. The stride is a hyperparameter which you, the engineer, can tune. Increasing the stride reduces the size of your model by reducing the number of total patches each layer observes. However, this usually comes with a reduction in accuracy.

Filter Depth

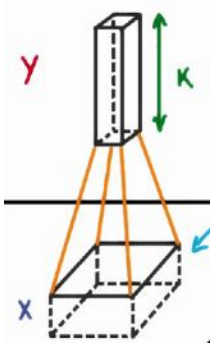
It's common to have more than one filter. Different filters pick up different qualities of a patch. For example, one filter might look for a particular color, while another might look for a kind of object of a specific shape. The amount of filters in a convolutional layer is called the *filter depth*.



In the above example, a patch is connected to a neuron in the next layer. Source: MIchael Nielsen.

How many neurons does each patch connect to?

That's dependent on our filter depth. If we have a depth of k, we connect each patch of pixels to k neurons in the next layer. This gives us the height of k in the next layer, as shown below. In practice, k is a hyperparameter we tune, and most CNNs tend to pick the same starting values.



But why connect a single patch to multiple neurons in the next layer? Isn't one neuron good enough?

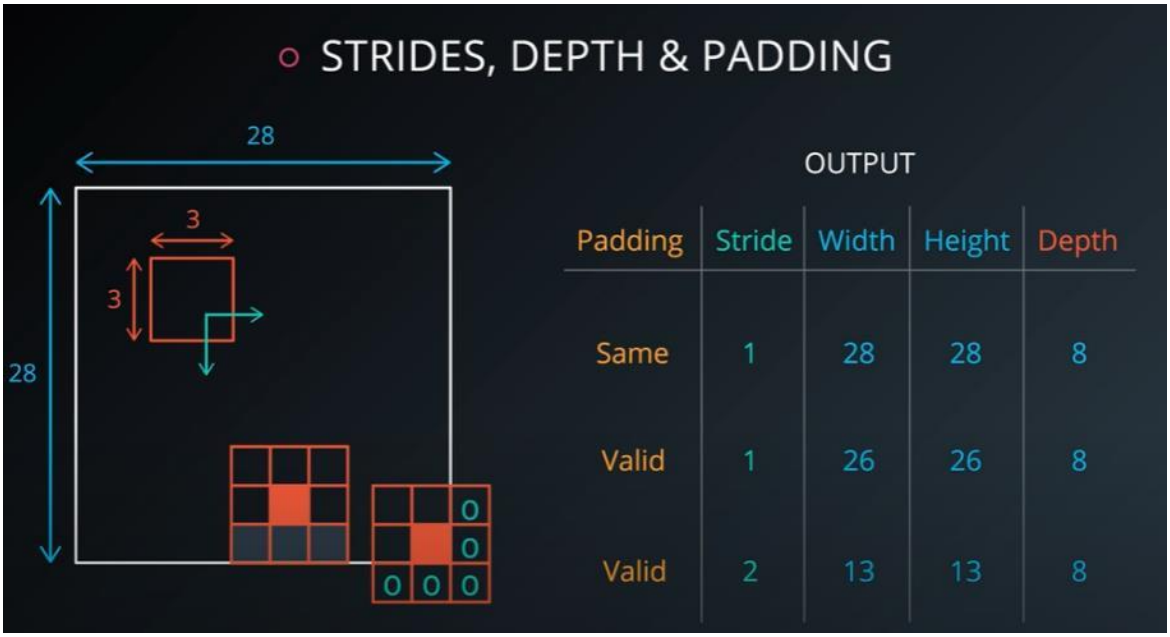
Multiple neurons can be useful because a patch can have multiple interesting characteristics that we want to capture.

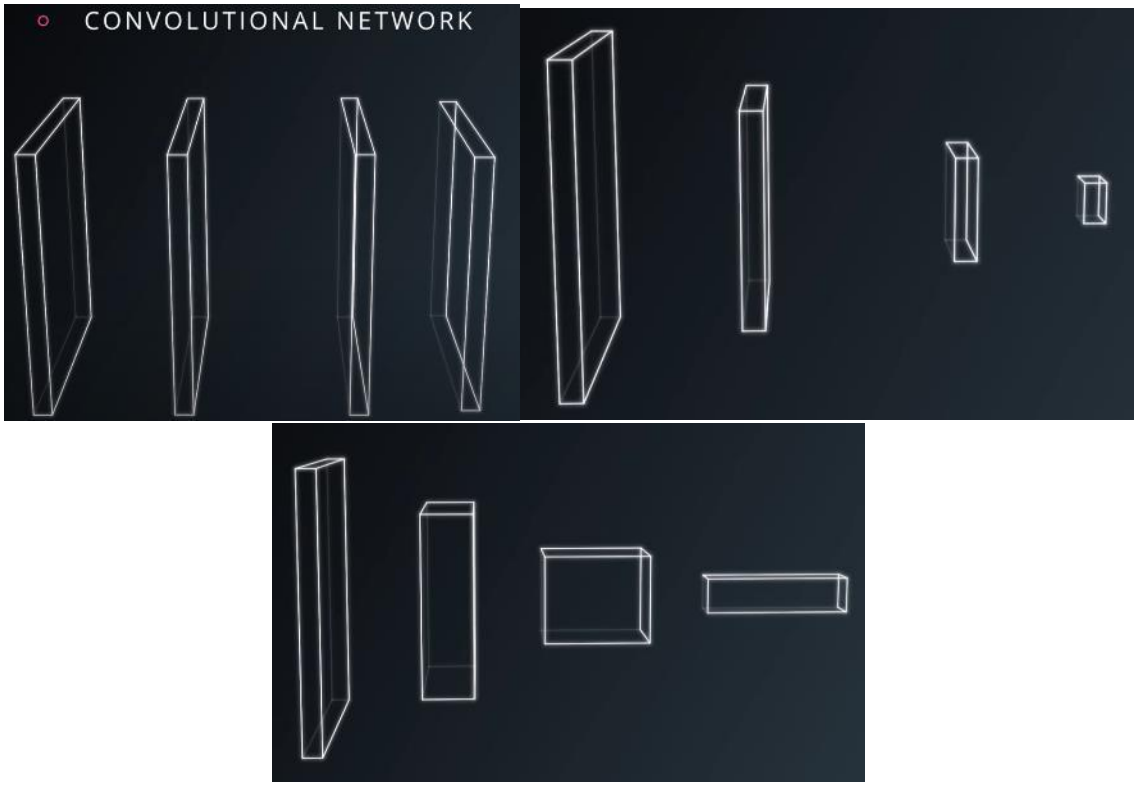
For example, one patch might include some white teeth, some blonde whiskers, and part of a red tongue. In that case, we might want a filter depth of at least three - one for each of teeth, whiskers, and tongue.

This patch of the dog has many interesting features we may want to capture. These include the presence of teeth, the presence of whiskers, and the pink color of the tongue.

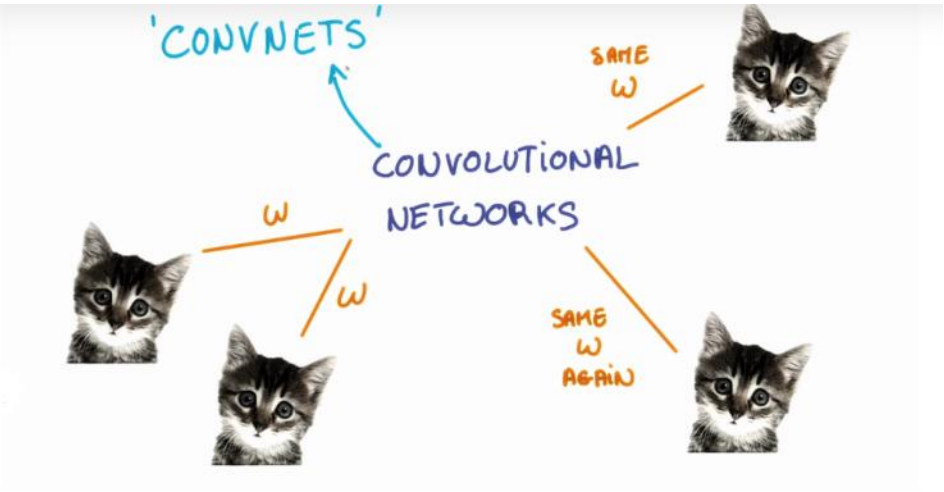
Having multiple neurons for a given patch ensures that our CNN can learn to capture whatever characteristics the CNN learns are important.

Remember that the CNN isn't "programmed" to look for certain characteristics. Rather, it learns **on its own** which characteristics to notice.





Note, a "Fully Connected" layer is a standard, non convolutional layer, where all inputs are connected to all output neurons. This is also referred to as a "dense" layer, and is what we used in the previous two lessons.



The weights, `w`, are shared across patches for a given layer in a CNN to detect the cat above regardless of where in the image it is located.

When we are trying to classify a picture of a cat, we don't care where in the image a cat is. If it's in the top left or the bottom right, it's still a cat in our eyes. We would like our CNNs to also possess this ability known as translation invariance. How can we achieve this?

As we saw earlier, the classification of a given patch in an image is determined by the weights and biases corresponding to that patch.

If we want a cat that's in the top left patch to be classified in the same way as a cat in the bottom right patch, we need the weights and biases corresponding to those patches to be the same, so that they are classified the same way.

This is exactly what we do in CNNs. The weights and biases we learn for a given output layer are shared across all patches in a given input layer. Note that as we increase the depth of our filter, the number of weights and biases we have to learn still increases, as the weights aren't shared across the output channels.

There's an additional benefit to sharing our parameters. If we did not reuse the same weights across all patches, we would have to learn new parameters for every single patch and hidden layer neuron pair. This does not scale well, especially for higher fidelity images. Thus, sharing parameters not only helps us with translation invariance, but also gives us a smaller, more scalable model.

Dimensionality

From what we've learned so far, how can we calculate the number of neurons of each layer in our CNN?

Given:

- our input layer has a width of `W` and a height of `H`
- our convolutional layer has a filter size `F`
- we have a stride of `S`
- a padding of `P`
- and the number of filters `K`,

the following formula gives us the width of the next layer: `W_out =[ (W-F+2P)/S] + 1`.

The output height would be `H_out = [(H-F+2P)/S] + 1`.

And the output depth would be equal to the number of filters `D_out = K`.

The output volume would be `W_out * H_out * D_out`.

This would correspond to the following code:

```
input = tf.placeholder(tf.float32, (None, 32, 32, 3))
filter_weights = tf.Variable(tf.truncated_normal((8, 8, 3, 20))) # (height, width, input_depth, output_dept
filter_bias = tf.Variable(tf.zeros(20))
strides = [1, 2, 2, 1] # (batch, height, width, depth)
padding = 'SAME'
conv = tf.nn.conv2d(input, filter_weights, strides, padding) + filter_bias
```

Note the output shape of `conv` will be [1, 16, 16, 20]. It's 4D to account for batch size, but more importantly, it's not [1, 14, 14, 20]. This is because the padding algorithm TensorFlow uses is not exactly the same as the one above. An alternative algorithm is to switch `padding` from `'SAME'` to `'VALID'` which would result in an output shape of [1, 13, 13, 20]. If you're curious how padding works in TensorFlow, read **this document**.

In summary TensorFlow uses the following equation for 'SAME' vs 'VALID'

SAME Padding, the output height and width are computed as:

`out_height` = ceil(float(in_height) / float(strides[1]))

`out_width` = ceil(float(in_width) / float(strides[2]))

VALID Padding, the output height and width are computed as:

`out_height` = ceil(float(in_height - filter_height + 1) / float(strides[1]))

`out_width` = ceil(float(in_width - filter_width + 1) / float(strides[2]))

Setup

H = height, W = width, D = depth

- We have an input of shape 32x32x3 (HxWxD)
- 20 filters of shape 8x8x3 (HxWxD)
- A stride of 2 for both the height and width (S)
- Zero padding of size 1 (P)

Output Layer

- 14x14x20 (HxWxD)

That's right, there are `756560` total parameters. That's a HUGE amount! Here's how we calculate it:

`(8 * 8 * 3 + 1) * (14 * 14 * 20) = 756560`

`8 * 8 * 3` is the number of weights, we add `1` for the bias. Remember, each weight is assigned to every single part of the output (`14 * 14 * 20`). So we multiply these two numbers together and we get the final answer.

Setup

H = height, W = width, D = depth

- We have an input of shape 32x32x3 (HxWxD)
- 20 filters of shape 8x8x3 (HxWxD)
- A stride of 2 for both the height and width (S)
- Zero padding of size 1 (P)

Output Layer

- 14x14x20 (HxWxD)

Hint

With parameter sharing, each neuron in an output channel shares its weights with every other neuron in that channel. So the number of parameters is equal to the number of neurons in the filter, plus a bias neuron, all multiplied by the number of channels in the output layer.

That's right, there are `3860` total parameters. That's 196 times fewer parameters! Here's how the answer is calculated:

`(8 * 8 * 3 + 1) * 20 = 3840 + 20 = 3860`

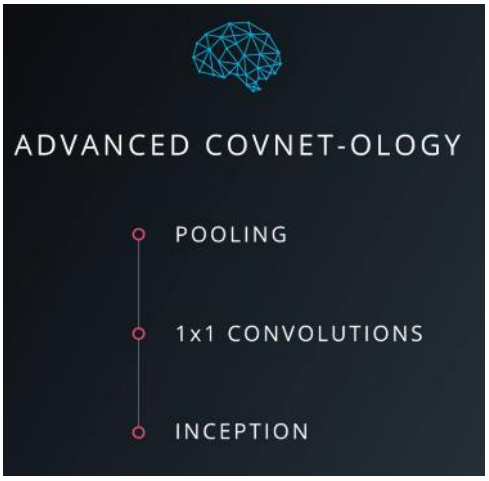
That's `3840` weights and `20` biases. This should look similar to the answer from the previous quiz. The difference being it's just `20` instead of (`14 * 14 * 20`). Remember, with weight sharing we use the same filter for an entire depth slice. Because of this we can get rid of `14 * 14` and be left with only `20`.

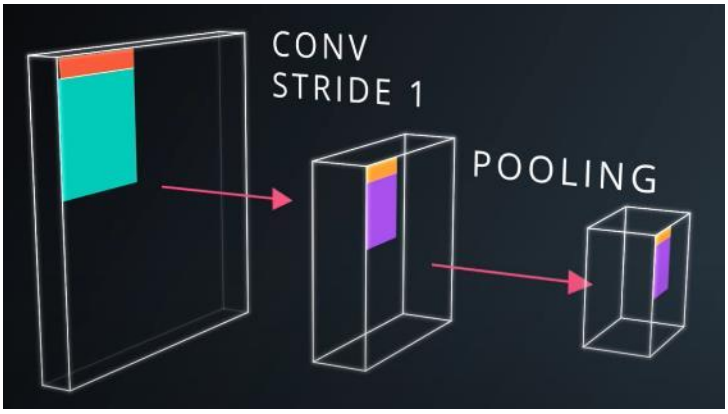
```
# Output depth
k_output = 64

# Image Properties
image_width = 10
image_height = 10
color_channels = 3

# Convolution filter
filter_size_width = 5
filter_size_height = 5

# Input/Image
input = tf.placeholder(
    tf.float32,
    shape=[None, image_height, image_width, color_channels])

# Weight and bias
weight = tf.Variable(tf.truncated_normal(
    [filter_size_height, filter_size_width, color_channels, k_output]))
bias = tf.Variable(tf.zeros(k_output))

# Apply Convolution
conv_layer = tf.nn.conv2d(input, weight, strides=[1, 2, 2, 1], padding='SAME')
# Add bias
conv_layer = tf.nn.bias_add(conv_layer, bias)
# Apply activation function
conv_layer = tf.nn.relu(conv_layer)
```

he code above uses the `tf.nn.conv2d()` function to compute the convolution with `weight` as the lter and `[1, 2, 2, 1]` for the strides. TensorFlow uses a stride for each `input` dimension, `[batch, input_height, input_width, input_channels]`. We are generally always going to set the stride or `batch` and `input_channels` (i.e. the first and fourth element in the `strides` array) to be `1`.

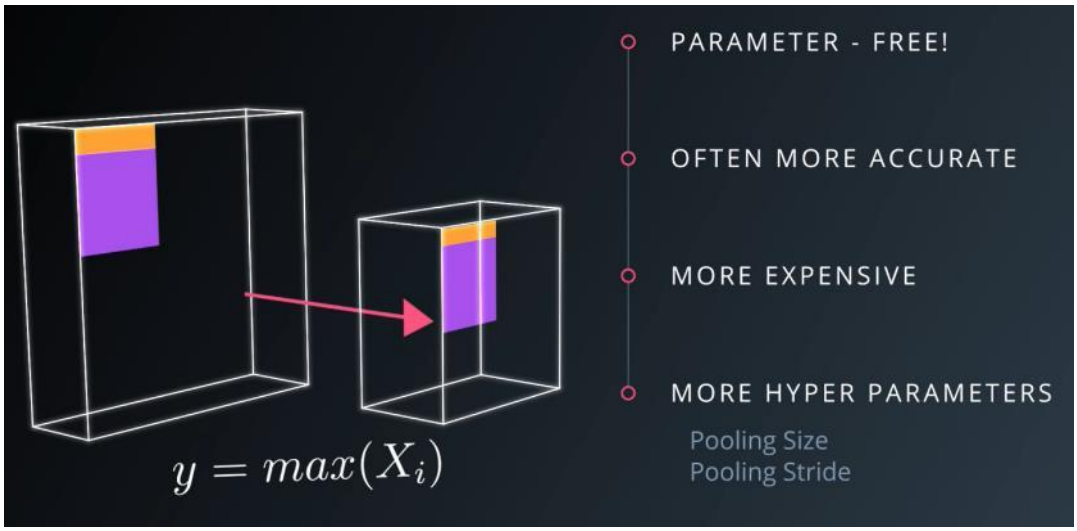
ou'll focus on changing `input_height` and `input_width` while setting `batch` and `input_channels` to 1. he `input_height` and `input_width` strides are for striding the filter over `input`. This example code ses a stride of 2 with 5x5 filter over `input`.

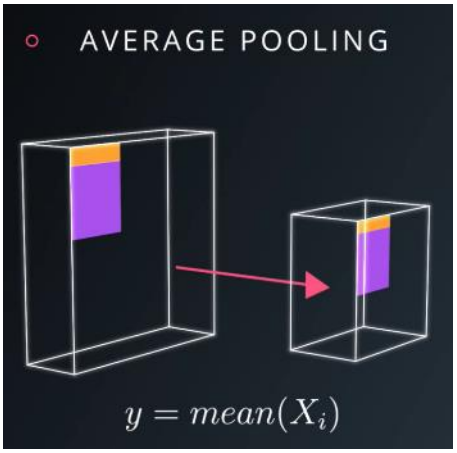
he `tf.nn.bias_add()` function adds a 1-d bias to the last dimension in a matrix.

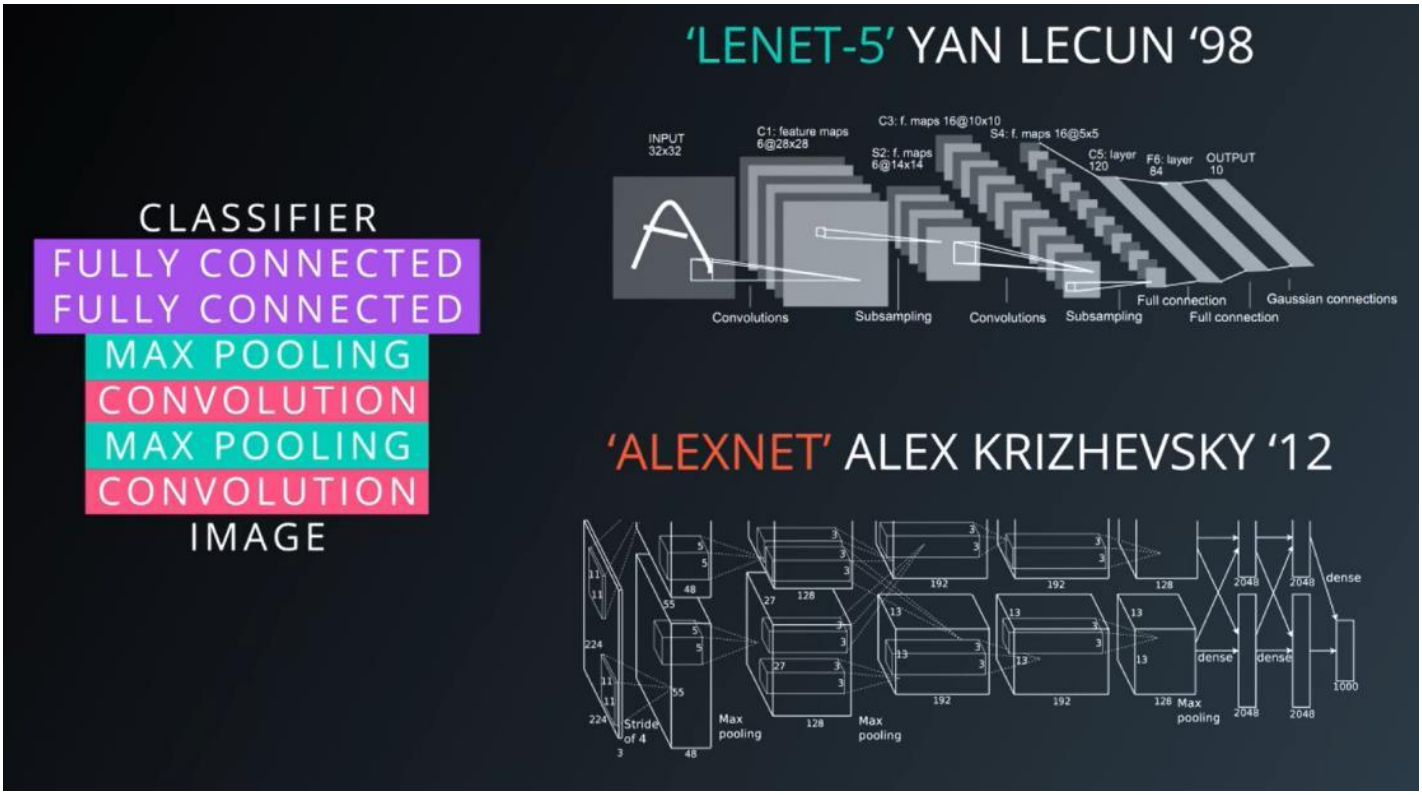

ADVANCED COVNET-OLOGY
POOLING
1x1 CONVOLUTIONS
INCEPTION


CONV
STRIDE 1
POOLING

Max Pooling


PARAMETER - FREE!
OFTEN MORE ACCURATE
MORE EXPENSIVE
MORE HYPER PARAMETERS
Pooling Size
Pooling Stride
$y = max(X_i)$


AVERAGE POOLING
$y = mean(X_i)$



Conceptually, the benefit of the max pooling operation is to reduce the size of the input, and allow the neural network to focus on only the most important elements. Max pooling does this by only retaining the maximum value for each filtered area, and removing the remaining values.

TensorFlow provides the `tf.nn.max_pool()` function to apply **max pooling** to your convolutional layers.

```
...
conv_layer = tf.nn.conv2d(input, weight, strides=[1, 2, 2, 1], padding='SAME')
conv_layer = tf.nn.bias_add(conv_layer, bias)
conv_layer = tf.nn.relu(conv_layer)
# Apply Max Pooling
conv_layer = tf.nn.max_pool(
    conv_layer,
    ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1],
    padding='SAME')
```

The `tf.nn.max_pool()` function performs max pooling with the `ksize` parameter as the size of the filter and the `strides` parameter as the length of the stride. 2x2 filters with a stride of 2x2 are common in practice.

The `ksize` and `strides` parameters are structured as 4-element lists, with each element corresponding to a dimension of the input tensor (`[batch, height, width, channels]`). For both `ksize` and `strides`, the batch and channel dimensions are typically set to `1`.

QUIZ QUESTION

A pooling layer is generally used to ...

- [ ] Increase the size of the output
- [x] Decrease the size of the output
- [x] Prevent overfitting
- [ ] Gain information

Solution

The correct answer is **decrease the size of the output** and **prevent overfitting**. Preventing overfitting is a consequence of reducing the output size, which in turn, reduces the number of parameters in future layers.

Recently, pooling layers have fallen out of favor. Some reasons are:

- Recent datasets are so big and complex we're more concerned about underfitting.
- Dropout is a much better regularizer.
- Pooling results in a loss of information. Think about the max pooling operation as an example. We only keep the largest of *n* numbers, thereby disregarding *n-1* numbers completely.

Setup

H = height, W = width, D = depth

- We have an input of shape 4x4x5 (HxWxD)
- Filter of shape 2x2 (HxW)
- A stride of 2 for both the height and width (S)

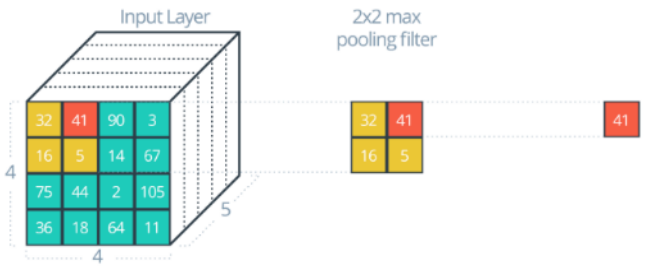
Recall the formula for calculating the new height or width:

```
new_height = (input_height - filter_height)/S + 1
new_width = (input_width - filter_width)/S + 1
```

NOTE: For a pooling layer the output depth is the same as the input depth. Additionally, the pooling operation is applied individually for each depth slice.

The image below gives an example of how a max pooling layer works. In this case, the max pooling filter has a shape of 2x2. As the max pooling filter slides across the input layer, the filter will output the maximum value of the 2x2 square.

Pooling Mechanics Quiz



Pooling Layer Output Shape

What's the shape of the output? Format is **HxWxD**.

2x2x5

RESET

Solution

The answer is **2x2x5**. Here's how it's calculated using the formula:

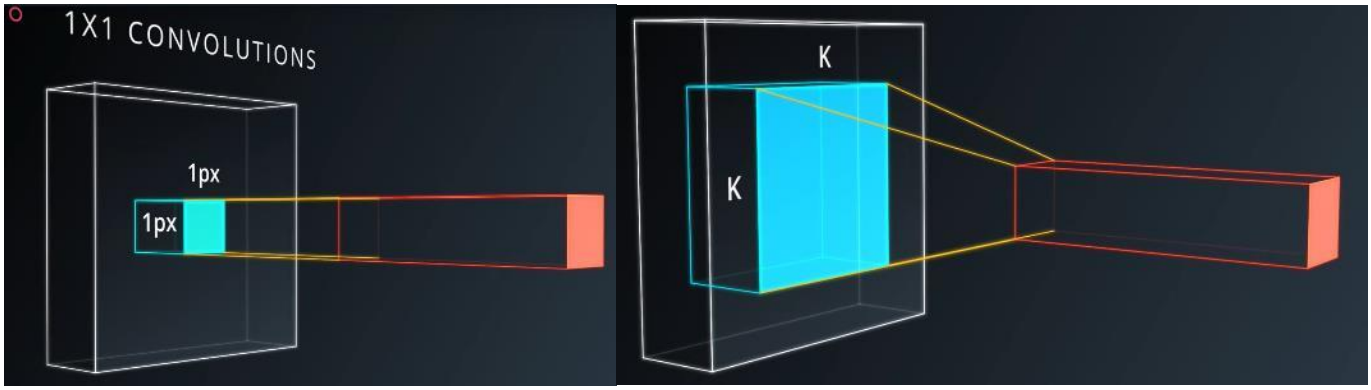
```
(4 - 2)/2 + 1 = 2
(4 - 2)/2 + 1 = 2
```

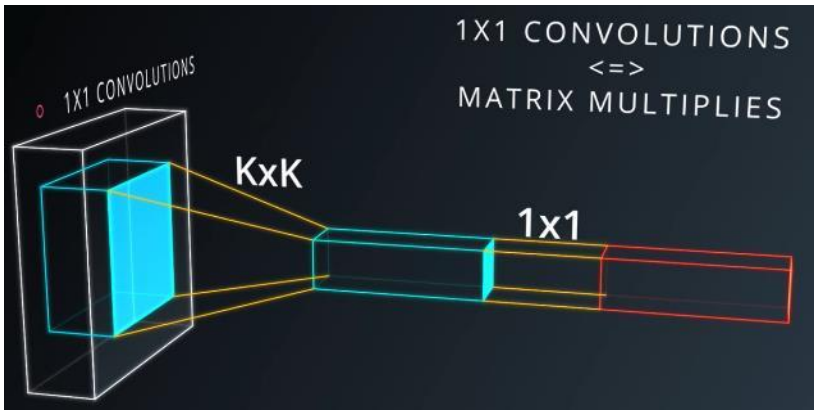
The depth stays the same.

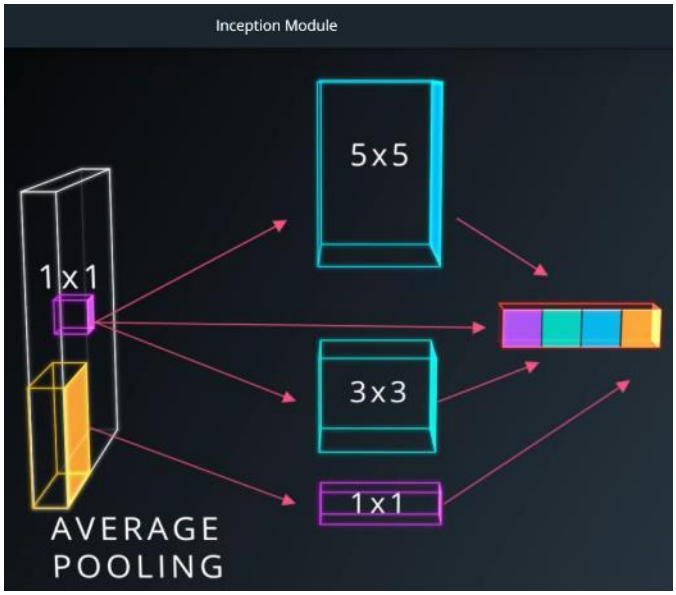
Here's the corresponding code:

```
input = tf.placeholder(tf.float32, (None, 4, 4, 5))
filter_shape = [1, 2, 2, 1]
strides = [1, 2, 2, 1]
padding = 'VALID'
pool = tf.nn.max_pool(input, filter_shape, strides, padding)
```

The output shape of `pool` will be [1, 2, 2, 5], even if `padding` is changed to `'SAME'`.








```
import tensorflow as tf
import numpy as np
```

```
"""
Setup the strides, padding and filter weight/bias such that
the output shape is (1, 2, 2, 3).
"""
# `tf.nn.conv2d` requires the input be 4D (batch_size, height, width, depth)
# (1, 4, 4, 1)
x = np.array([
    [0, 1, 0.5, 10],
    [2, 2.5, 1, -8],
    [4, 0, 5, 6],
    [15, 1, 2, 3]], dtype=np.float32).reshape((1, 4, 4, 1))
X = tf.constant(x)

def conv2d(input_array):
    # Filter (weights and bias)
    # The shape of the filter weight is (height, width, input_depth, output_
    # The shape of the filter bias is (output_depth,)
    # TODO: Define the filter weights `F_W` and filter bias `F_b`.
    # NOTE: Remember to wrap them in `tf.Variable`, they are trainable param
    F_W = tf.Variable(tf.truncated_normal((2, 2, 1, 3)))
    F_b = tf.Variable(tf.zeros(3))
    # TODO: Set the stride for each dimension (batch_size, height, width, de
    strides = [1, 2, 2, 1]
    # TODO: set the padding, either 'VALID' or 'SAME'.
    padding = 'VALID'
    # https://www.tensorflow.org/versions/r0.11/api_docs/python/nn.html#conv
    # `tf.nn.conv2d` does not include the bias computation so we have to add
    return tf.nn.conv2d(input_array, F_W, strides, padding) + F_b

output = conv2d(X)
```

Solution

Here's how I did it. **NOTE**: there's more than 1 way to get the correct output shape. Your answer might differ from mine.

```
def conv2d(input):
    # Filter (weights and bias)
    F_W = tf.Variable(tf.truncated_normal((2, 2, 1, 3)))
    F_b = tf.Variable(tf.zeros(3))
    strides = [1, 2, 2, 1]
    padding = 'VALID'
    return tf.nn.conv2d(input, F_W, strides, padding) + F_b
```

I want to transform the input shape `(1, 4, 4, 1)` to `(1, 2, 2, 3)`. I choose `'VALID'` for the padding algorithm. I find it simpler to understand and it achieves the result I'm looking for.

```
out_height = ceil(float(in_height - filter_height + 1) / float(strides[1]))
out_width  = ceil(float(in_width - filter_width + 1) / float(strides[2]))
```

Plugging in the values:

```
out_height = ceil(float(4 - 2 + 1) / float(2)) = ceil(1.5) = 2
out_width  = ceil(float(4 - 2 + 1) / float(2)) = ceil(1.5) = 2
```

In order to change the depth from 1 to 3, I have to set the output depth of my filter appropriately:

```
F_W = tf.Variable(tf.truncated_normal((2, 2, 1, 3))) # (height, width, input_depth, output_depth)
F_b = tf.Variable(tf.zeros(3)) # (output_depth)
```

The input has a depth of 1, so I set that as the `input_depth` of the filter.

```
"""
Set the values to `strides` and `ksize` such that
the output shape after pooling is (1, 2, 2, 1).
"""
import tensorflow as tf
import numpy as np

# `tf.nn.max_pool` requires the input be 4D (batch_size, height, width, dep
# (1, 4, 4, 1)
x = np.array([
    [0, 1, 0.5, 10],
    [2, 2.5, 1, -8],
    [4, 0, 5, 6],
    [15, 1, 2, 3]], dtype=np.float32).reshape((1, 4, 4, 1))
X = tf.constant(x)

def maxpool(input):
    # TODO: Set the ksize (filter size) for each dimension (batch_size, hei
    ksize = [1, 2, 2, 1]
    # TODO: Set the stride for each dimension (batch_size, height, width, d
    strides = [1, 2, 2, 1]
    # TODO: set the padding, either 'VALID' or 'SAME'.
    padding = 'VALID'
    # https://www.tensorflow.org/versions/r0.11/api_docs/python/nn.html#max_
    return tf.nn.max_pool(input, ksize, strides, padding)

out = maxpool(X)
```

Solution

Here's how I did it. **NOTE**: there's more than 1 way to get the correct output shape. Your answer might differ from mine.

```
def maxpool(input):
    ksize = [1, 2, 2, 1]
    strides = [1, 2, 2, 1]
    padding = 'VALID'
    return tf.nn.max_pool(input, ksize, strides, padding)
```

I want to transform the input shape `(1, 4, 4, 1)` to `(1, 2, 2, 1)`. I choose `'VALID'` for the padding algorithm. I find it simpler to understand and it achieves the result I'm looking for.

```
out_height = ceil(float(in_height - filter_height + 1) / float(strides[1]))
out_width  = ceil(float(in_width - filter_width + 1) / float(strides[2]))
```

Plugging in the values:

```
out_height = ceil(float(4 - 2 + 1) / float(2)) = ceil(1.5) = 2
out_width  = ceil(float(4 - 2 + 1) / float(2)) = ceil(1.5) = 2
```

The depth doesn't change during a pooling operation so I don't have to worry about that.

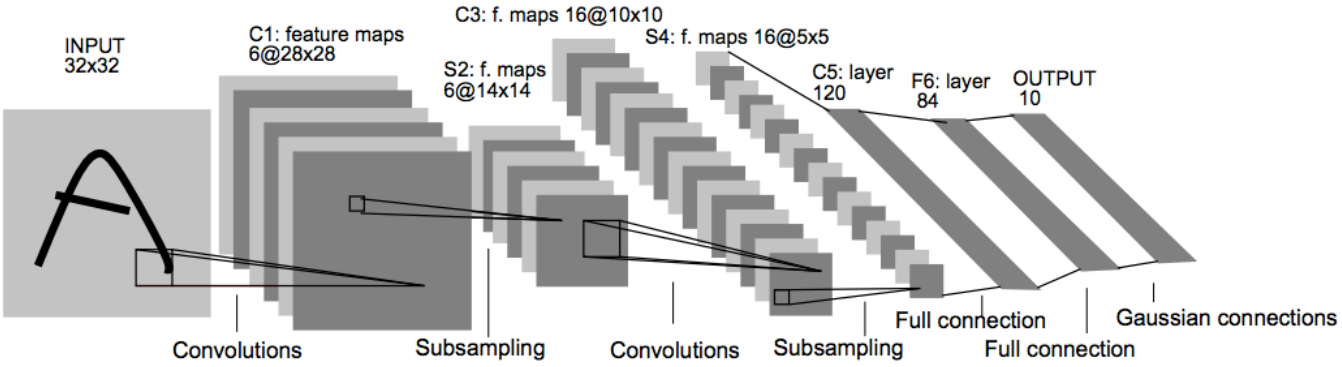


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Strategies to Improve Your Classification Model

Looking at where to go next what could you do to further improve this model? There are several broad categories, to try.

- experiment with different network architectures, or just change the dimensions of the LeNet layers
- add regularization features like drop out or L2 regularization to make sure the network doesn't overfit the training data
- tune the hyperparameters
- improve the data pre-processing with steps like normalization and setting a zero mean
- augment the training data by rotating or shifting images or by changing colors

Neural Networks in Keras

Here are some core concepts you need to know for working with Keras. All Keras exercises will be performed in JUPYTER workspaces, using python 3.5, Tensorflow 1.3, and Keras 2.09. More information on using JUPYTER in workspaces, can be found earlier in the term in the Workspaces lesson.

Sequential Model

```
from keras.models import Sequential

# Create the Sequential model
model = Sequential()
```

The keras.models.Sequential class is a wrapper for the neural network model. It provides common functions like `fit()`, `evaluate()`, and `compile()`. We'll cover these functions as we get to them. Let's

Layers

A Keras layer is just like a neural network layer. There are fully connected layers, max pool layers, and activation layers. You can add a layer to the model using the model's `add()` function. For example, a simple model would look like this:

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten

# Create the Sequential model
model = Sequential()

#1st Layer - Add a flatten layer
model.add(Flatten(input_shape=(32, 32, 3)))

#2nd Layer - Add a fully connected layer
model.add(Dense(100))

#3rd Layer - Add a ReLU activation layer
model.add(Activation('relu'))

#4th Layer - Add a fully connected layer
model.add(Dense(60))

#5th Layer - Add a ReLU activation layer
model.add(Activation('relu'))
```

Keras will automatically infer the shape of all layers after the first layer. This means you only have to set the input dimensions for the first layer.

The first layer from above, `model.add(Flatten(input_shape=(32, 32, 3)))`, sets the input dimension to (32, 32, 3) and output dimension to (3072=32 x 32 x 3). The second layer takes in the output of the first layer and sets the output dimensions to (100). This chain of passing output to the next layer continues until the last layer, which is the output of the model.

```
import pickle
import numpy as np
import tensorflow as tf

# Load pickled data
with open('small_train_traffic.p', mode='rb') as f:
    data = pickle.load(f)
```

```
# split data
X_train, y_train = data['features'], data['labels']
```

```
# Setup Keras
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten
```

```
# TODO: Build the Fully Connected Neural Network in Keras Here
model = Sequential()
model.add(Flatten(input_shape = (32, 32, 3)))
model.add(Dense(128, activation = 'relu'))
model.add(Dense(5, activation = 'softmax'))
'''
model = Sequential()
model.add(Flatten(input_shape = (32, 32, 3)))
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dense(5))
model.add(Activation('softmax'))
'''
```

```
"\nmodel = Sequential()\nmodel.add(Flatten(input_shape = (32, 32, 3)))\nmodel.add(Dense(128))\nmodel.add(Activation('relu'))
\nmodel.add(Dense(5))\nmodel.add(Activation('softmax'))\n"
```

```
# preprocess data
X_normalized = np.array(X_train / 255.0 - 0.5 )

from sklearn.preprocessing import LabelBinarizer
label_binarizer = LabelBinarizer()
y_one_hot = label_binarizer.fit_transform(y_train)

# TODO: change the number of training epochs to 3
model.compile('adam', 'categorical_crossentropy', ['accuracy'])
history = model.fit(X_normalized, y_one_hot, epochs=3, validation_split=0.2)
```

```
Train on 80 samples, validate on 20 samples
Epoch 1/3
80/80 [==============================] - 0s 2ms/step - loss: 1.1708 - acc: 0.4375 - val_loss: 0.7039 - val_acc: 0.7000
Epoch 2/3
80/80 [==============================] - 0s 491us/step - loss: 0.8265 - acc: 0.5750 - val_loss: 0.5217 - val_acc: 0.8500
Epoch 3/3
80/80 [==============================] - 0s 550us/step - loss: 0.6076 - acc: 0.7000 - val_loss: 0.4023 - val_acc: 1.0000
```

```
import pickle
import numpy as np
import tensorflow as tf

# Load pickled data
with open('small_train_traffic.p', mode='rb') as f:
    data = pickle.load(f)
```

```
# split data
X_train, y_train= data['features'], data['labels']
```

```
# Setup Keras
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten
from keras.layers.convolutional import Conv2D
```

```
# TODO: Build Convolutional Neural Network in Keras Here
model = Sequential()
model.add(Conv2D(32, kernel_size = (3, 3), activation = 'relu', input_shape = (32, 32, 3)))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dense(5, activation = 'relu'))
```

```
# Preprocess data
X_normalized = np.array(X_train / 255.0 - 0.5 )

from sklearn.preprocessing import LabelBinarizer
label_binarizer = LabelBinarizer()
y_one_hot = label_binarizer.fit_transform(y_train)
```

```
# compile and train model
# Training for 3 epochs should result in > 50% accuracy
model.compile('adam', 'categorical_crossentropy', ['accuracy'])
history = model.fit(X_normalized, y_one_hot, epochs=3, validation_split=0.2)
```

```
Train on 80 samples, validate on 20 samples
Epoch 1/3
80/80 [==============================] - 1s 9ms/step - loss: 6.1404 - acc: 0.4000 - val_loss: 2.9334 - val_acc: 0.7000
Epoch 2/3
80/80 [==============================] - 0s 5ms/step - loss: 3.7229 - acc: 0.5750 - val_loss: 2.8525 - val_acc: 0.7000
Epoch 3/3
80/80 [==============================] - 0s 5ms/step - loss: 3.6614 - acc: 0.6375 - val_loss: 2.7917 - val_acc: 0.7000
```

```
import pickle
import numpy as np
import tensorflow as tf

# Load pickled data
with open('small_train_traffic.p', mode='rb') as f:
    data = pickle.load(f)
```

```
# split the data
X_train, y_train = data['features'], data['labels']
```

```
# Setup Keras
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.pooling import MaxPooling2D
```

```
# TODO: Build Convolutional Neural Network in Keras Here
model = Sequential()
model.add(Conv2D(32, kernel_size = (3, 3), activation = 'relu', input_shape = (32, 32, 3)))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dense(5, activation = 'softmax'))
```

```
# Preprocess data
X_normalized = np.array(X_train / 255.0 - 0.5 )

from sklearn.preprocessing import LabelBinarizer
label_binarizer = LabelBinarizer()
y_one_hot = label_binarizer.fit_transform(y_train)
```

```
# compile and fit model
model.compile('adam', 'categorical_crossentropy', ['accuracy'])
history = model.fit(X_normalized, y_one_hot, epochs=3, validation_split=0.2)
```

```
import pickle
import numpy as np
import tensorflow as tf

# Load pickled data
with open('small_train_traffic.p', mode='rb') as f:
    data = pickle.load(f)
```

```
# split the data
X_train, y_train = data['features'], data['labels']
```

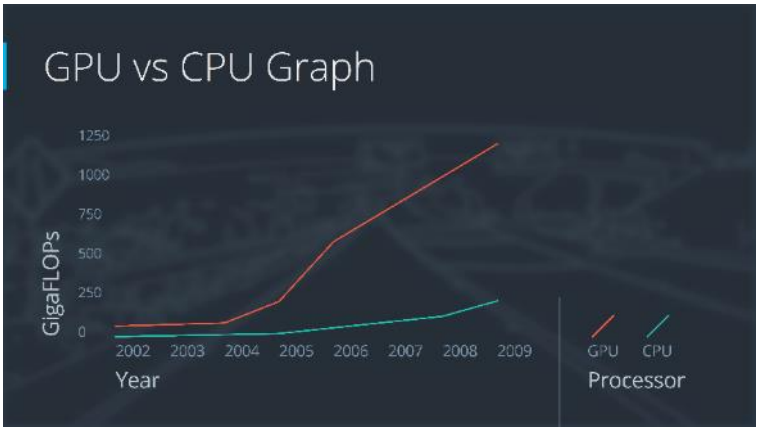
```
# Setup Keras
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten, Dropout
from keras.layers.convolutional import Conv2D
from keras.layers.pooling import MaxPooling2D
```

```
# TODO: Build Convolutional Pooling Neural Network with Dropout in Keras Here
model  = Sequential()
model.add(Conv2D(32, kernel_size = (3, 3), activation = 'relu', input_shape = (32, 32, 3)))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout(rate = 0.5))
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dense(5, activation = 'softmax'))
```

```
# preprocess data
X_normalized = np.array(X_train / 255.0 - 0.5 )

from sklearn.preprocessing import LabelBinarizer
label_binarizer = LabelBinarizer()
y_one_hot = label_binarizer.fit_transform(y_train)
```

```
# compile and fit model
model.compile('adam', 'categorical_crossentropy', ['accuracy'])
history = model.fit(X_normalized, y_one_hot, epochs=3, validation_split=0.2)
```

```
Train on 80 samples, validate on 20 samples
Epoch 1/3
80/80 [==============================] - 1s 8ms/step - loss: 1.3653 - acc: 0.3000 - val_loss: 0.9025 - val_acc: 0.4500
Epoch 2/3
80/80 [==============================] - 0s 3ms/step - loss: 0.8874 - acc: 0.4875 - val_loss: 0.6581 - val_acc: 0.6500
Epoch 3/3
80/80 [==============================] - 0s 3ms/step - loss: 0.6564 - acc: 0.7750 - val_loss: 0.4588 - val_acc: 1.0000
```



The Four Main Cases When Using Transfer Learning

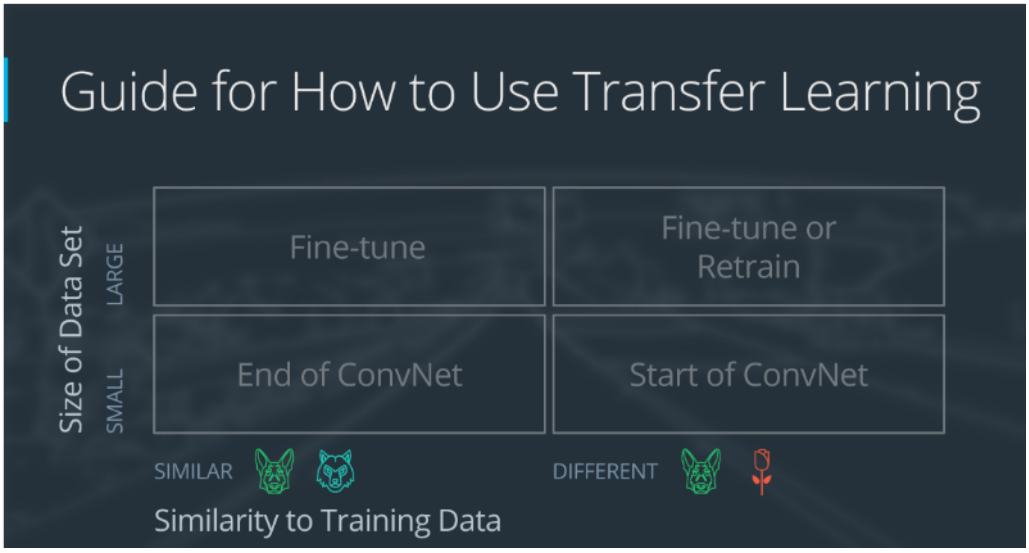
Transfer learning involves taking a pre-trained neural network and adapting the neural network to a new, different data set.

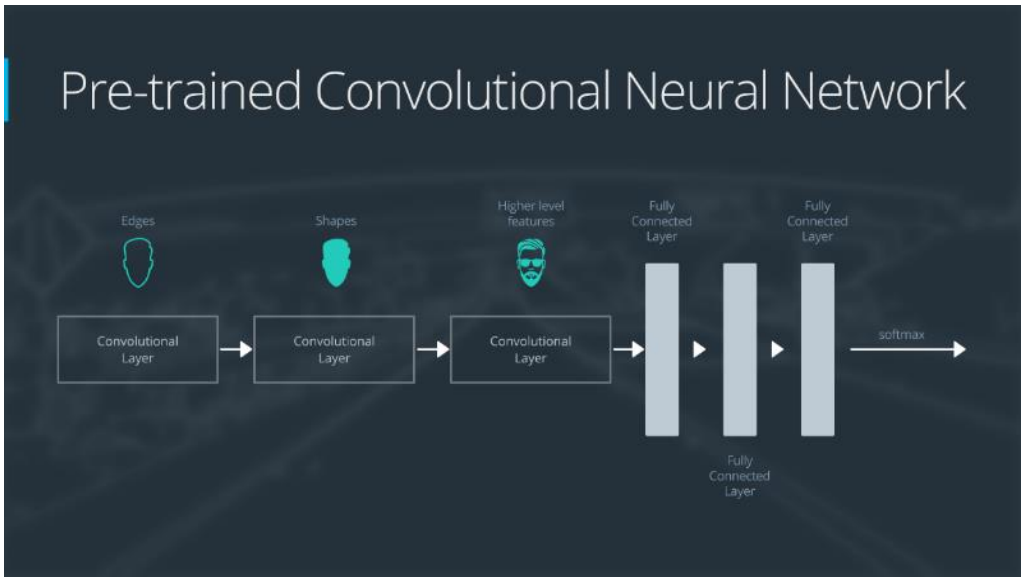
Depending on both:

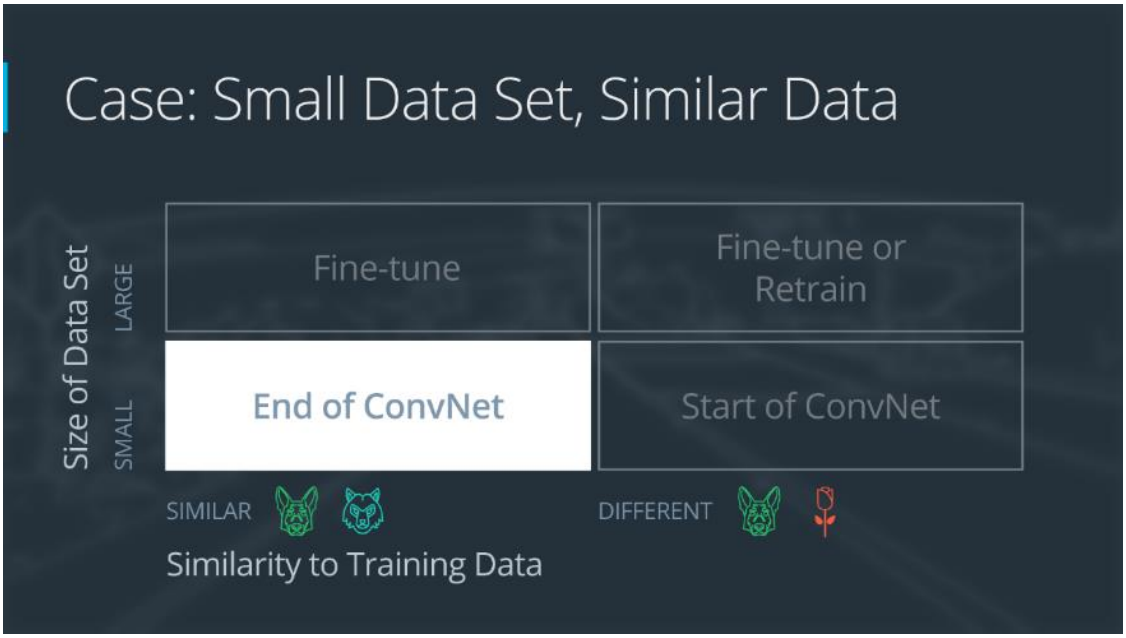
- the size of the new data set, and
- the similarity of the new data set to the original data set

the approach for using transfer learning will be different. There are four main cases:

1. new data set is small, new data is similar to original training data
2. new data set is small, new data is different from original training data
3. new data set is large, new data is similar to original training data
4. new data set is large, new data is different from original training data







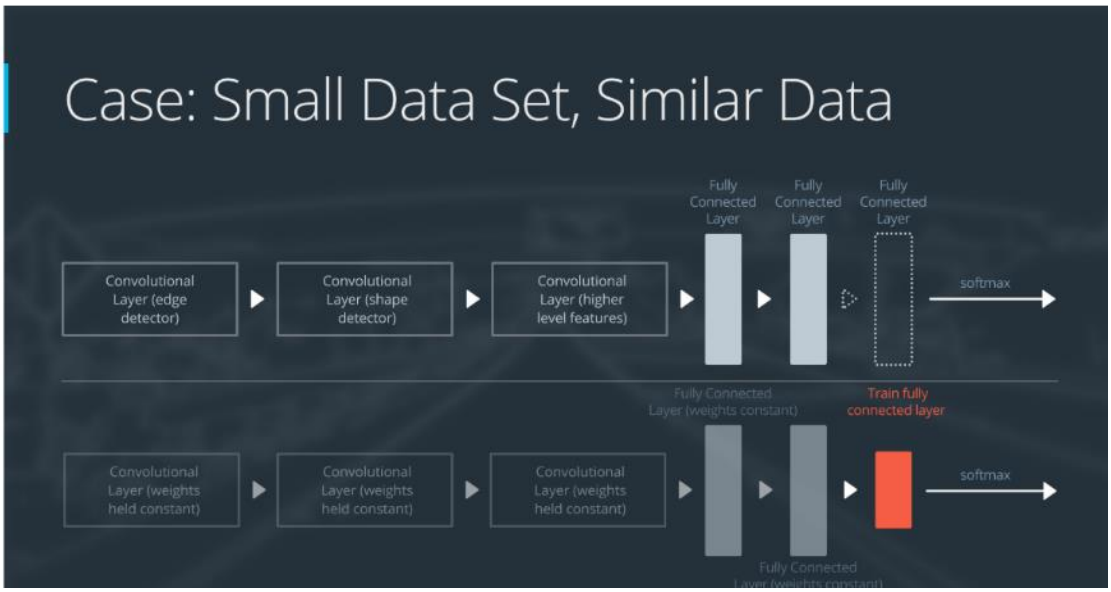
If the new data set is small and similar to the original training data:

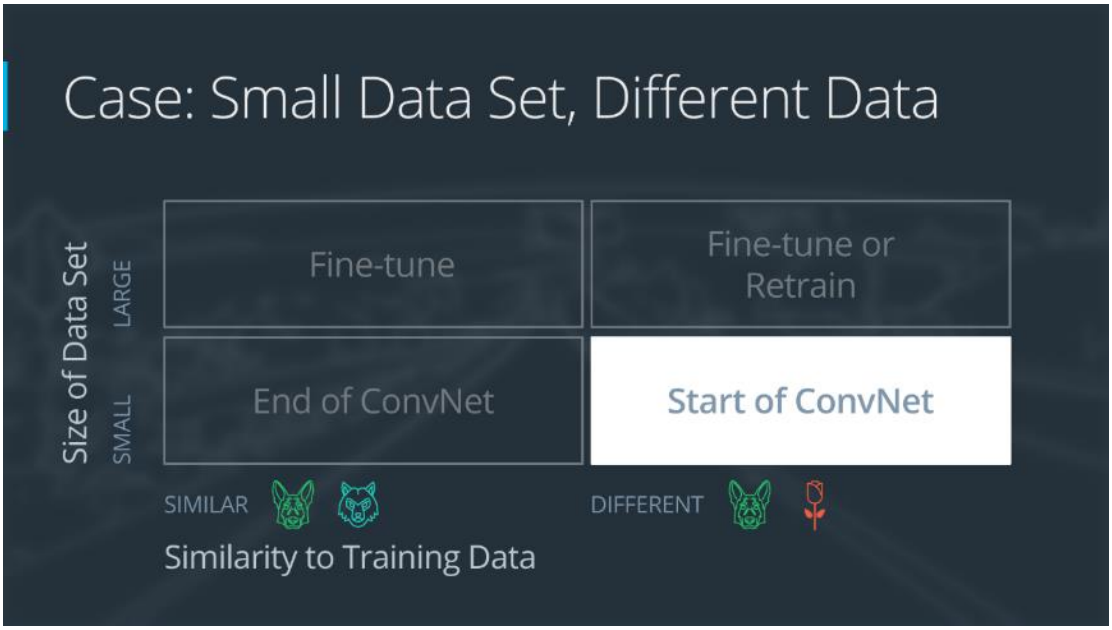
- slice off the end of the neural network
- add a new fully connected layer that matches the number of classes in the new data set
- randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network
- train the network to update the weights of the new fully connected layer

To avoid overfitting on the small data set, the weights of the original network will be held constant rather than re-training the weights.

Since the data sets are similar, images from each data set will have similar higher level features. Therefore most or all of the pre-trained neural network layers already contain relevant information about the new data set and should be kept.

Here's how to visualize this approach:





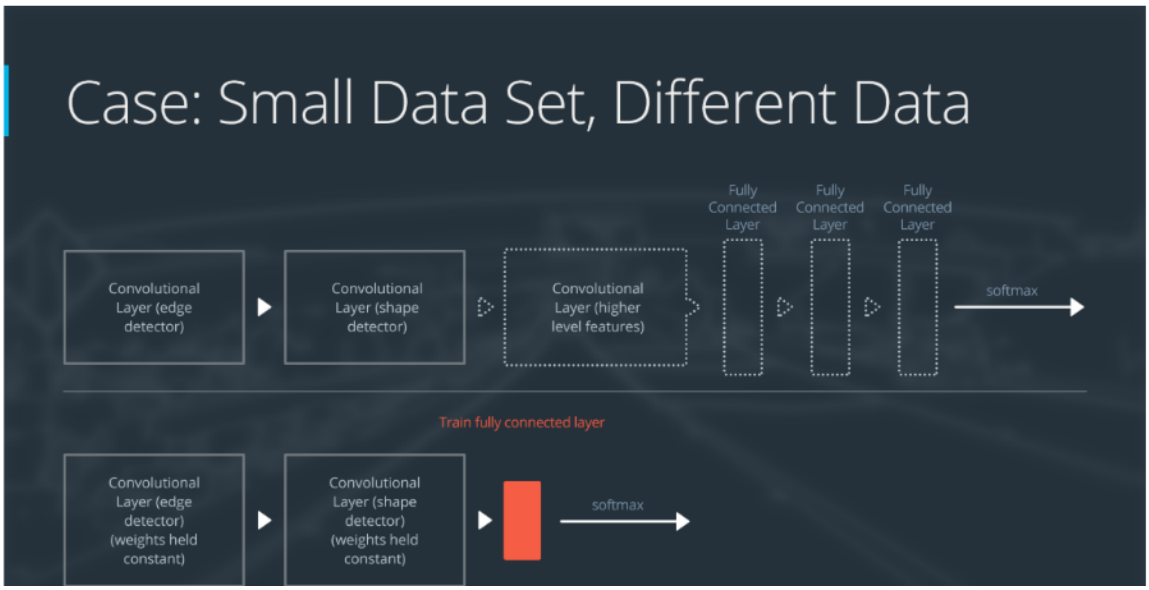
If the new data set is small and different from the original training data:

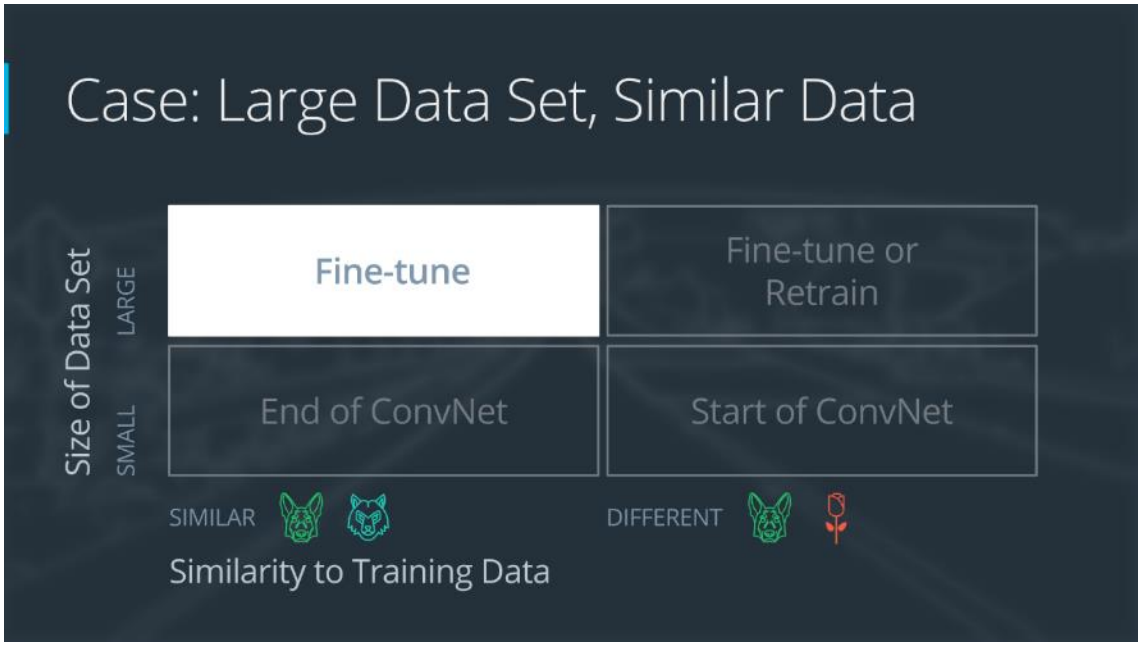
- slice off most of the pre-trained layers near the beginning of the network
- add to the remaining pre-trained layers a new fully connected layer that matches the number of classes in the new data set
- randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network
- train the network to update the weights of the new fully connected layer

Because the data set is small, overfitting is still a concern. To combat overfitting, the weights of the original neural network will be held constant, like in the first case.

But the original training set and the new data set do not share higher level features. In this case, the new network will only use the layers containing lower level features.

Here is how to visualize this approach:





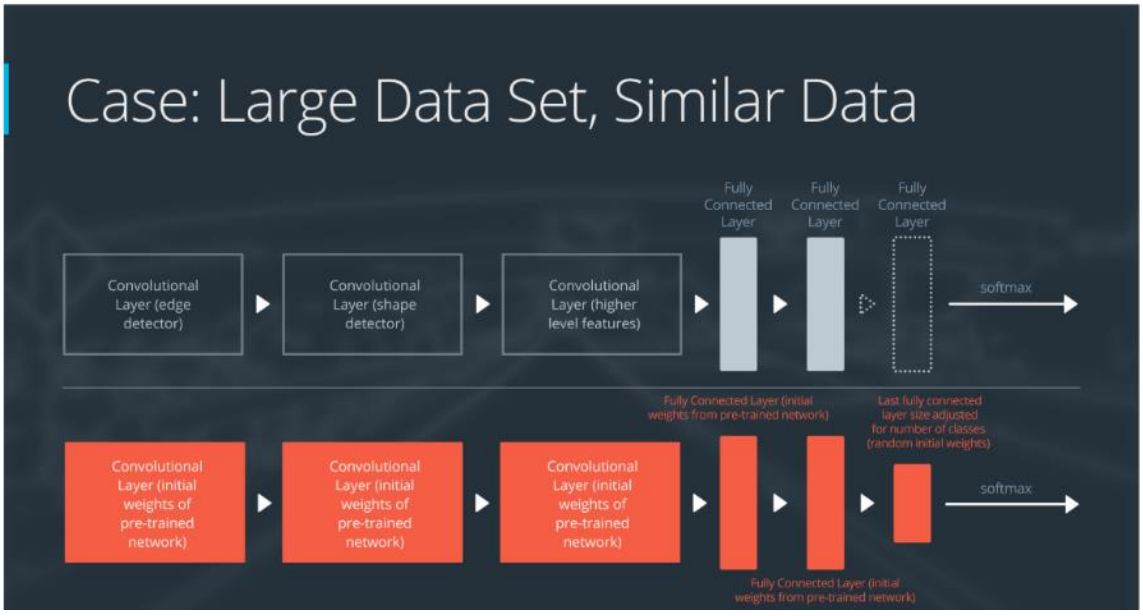
If the new data set is large and similar to the original training data:

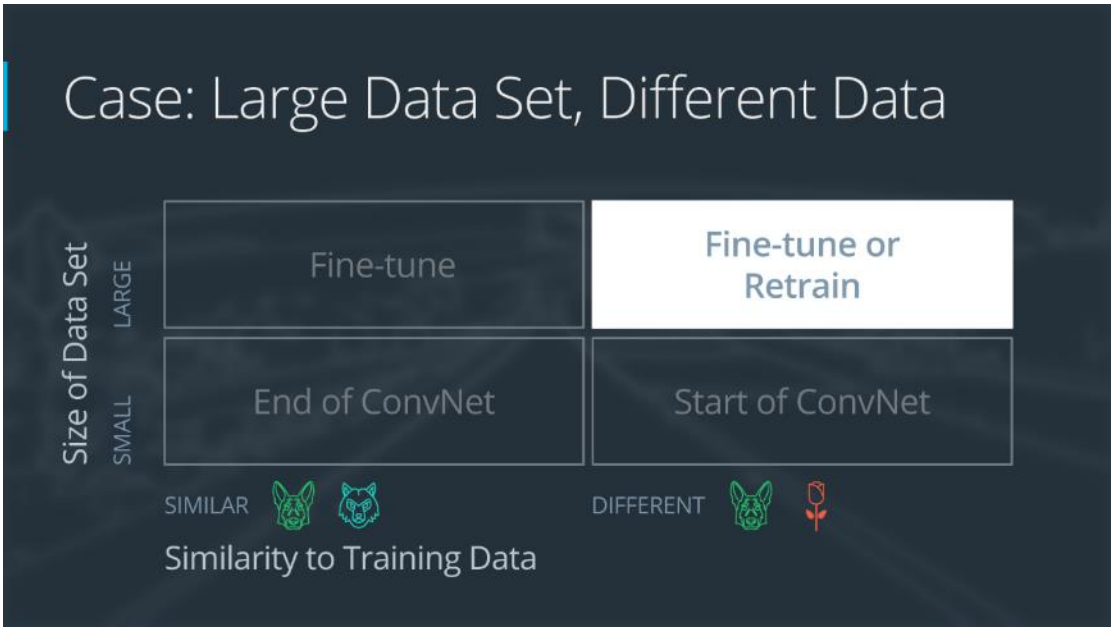
- remove the last fully connected layer and replace with a layer matching the number of classes in the new data set
- randomly initialize the weights in the new fully connected layer
- initialize the rest of the weights using the pre-trained weights
- re-train the entire neural network

Overfitting is not as much of a concern when training on a large data set; therefore, you can re-train all of the weights.

Because the original training set and the new data set share higher level features, the entire neural network is used as well.

Here is how to visualize this approach:





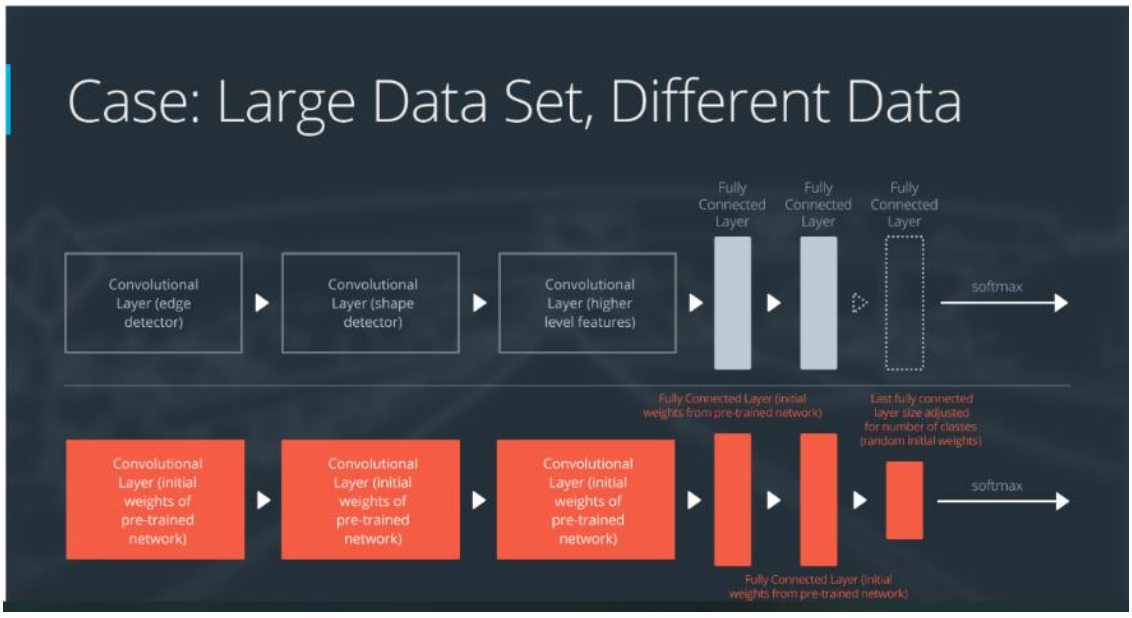
If the new data set is large and different from the original training data:

- remove the last fully connected layer and replace with a layer matching the number of classes in the new data set
- retrain the network from scratch with randomly initialized weights
- alternatively, you could just use the same strategy as the "large and similar" data case

Even though the data set is different from the training data, initializing the weights from the pre-trained network might make training faster. So this case is exactly the same as the case with a large, similar data set.

If using the pre-trained network as a starting point does not produce a successful model, another option is to randomly initialize the convolutional neural network weights and train the network from scratch.

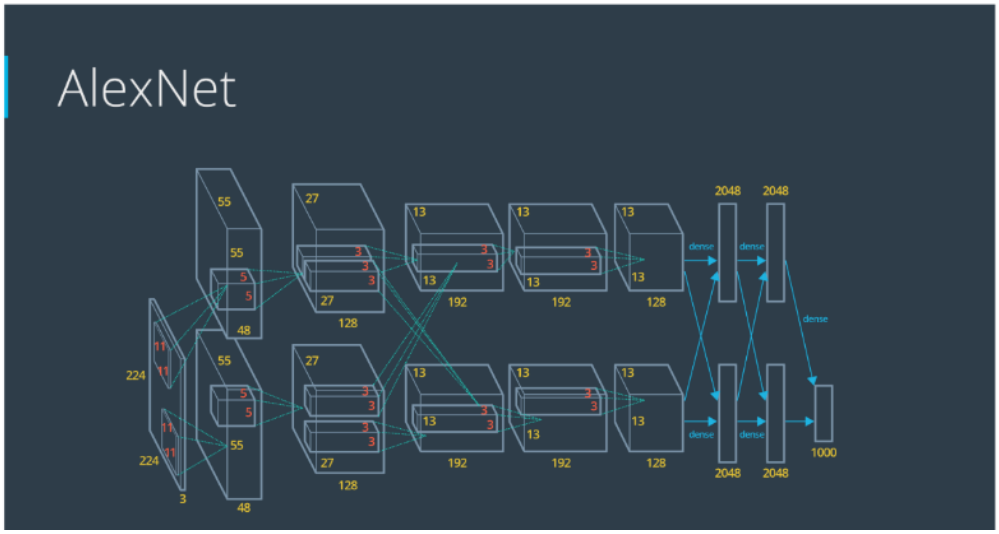
Here is how to visualize this approach:



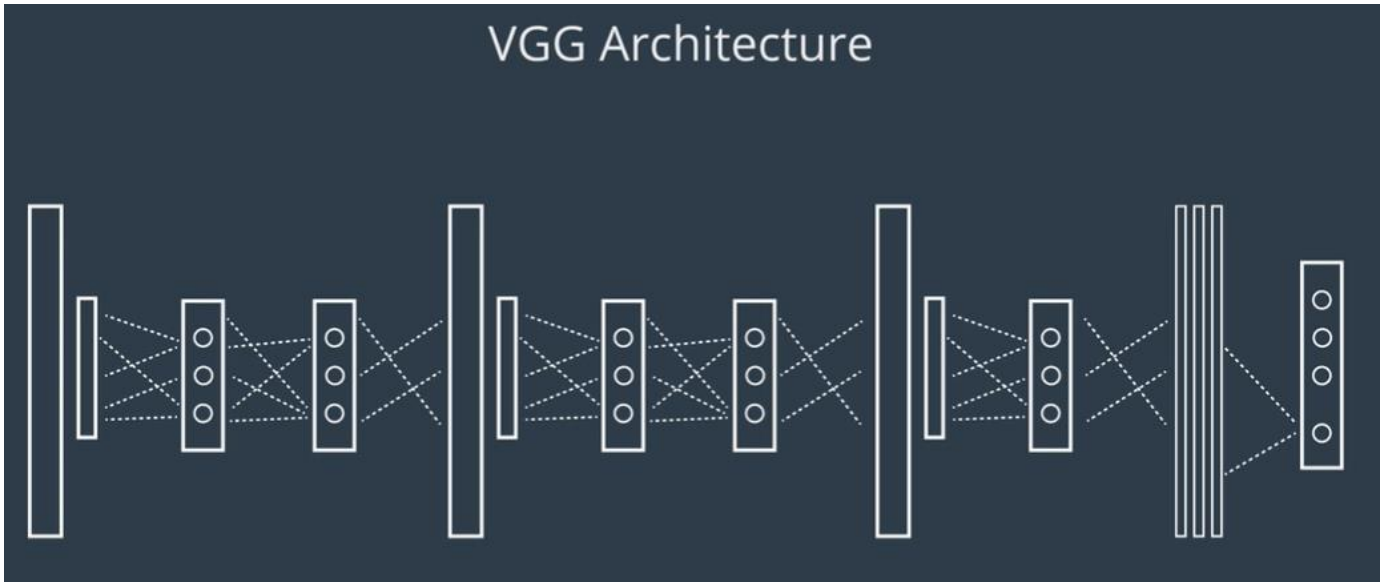
Pre-training a network with the ImageNet dataset is a very common way to get a strong neural network that can be used for transfer learning. With recent versions of Keras, you can easily import a pre-trained network by using the **Keras Applications** models

AlexNet Architecture

AlexNet puts the network on two GPUs, which allows for building a larger network. Although most of the calculations are done in parallel, the GPUs communicate with each other in certain layers. The **original research paper** on AlexNet said that parallelizing the network decreased the classification error rate by 1.7% when compared to a neural network that used half as many neurons on one GPU.



Nowadays, it has been discovered that some features are not necessary, and current implementations omit them



Error = 7%

VGG in Keras

As we mentioned earlier, you can fairly quickly utilize a pre-trained model with Keras Applications. VGG16 is one of the built-in models supported. There are actually two versions of VGG, VGG16 and VGG19 (where the numbers denote the number of layers included in each respective model), and you can utilize either with Keras, but we'll work with VGG16 here.

```
from keras.applications.vgg16 import VGG16

model = VGG16(weights='imagenet', include_top=False)
```

There are two arguments to `VGG16` in this example, although there could be more or less (check out the linked documentation to see other possible arguments). The first, `weights='imagenet'`, loads the pre-trained ImageNet weights. This is actually the default argument per the documentation, so if you don't include it, you should still be loading the ImageNet weights. However, you can also specify `None` here to get random weights if you just want the architecture of VGG16; this is not suggested here since you won't get the benefit of transfer learning.

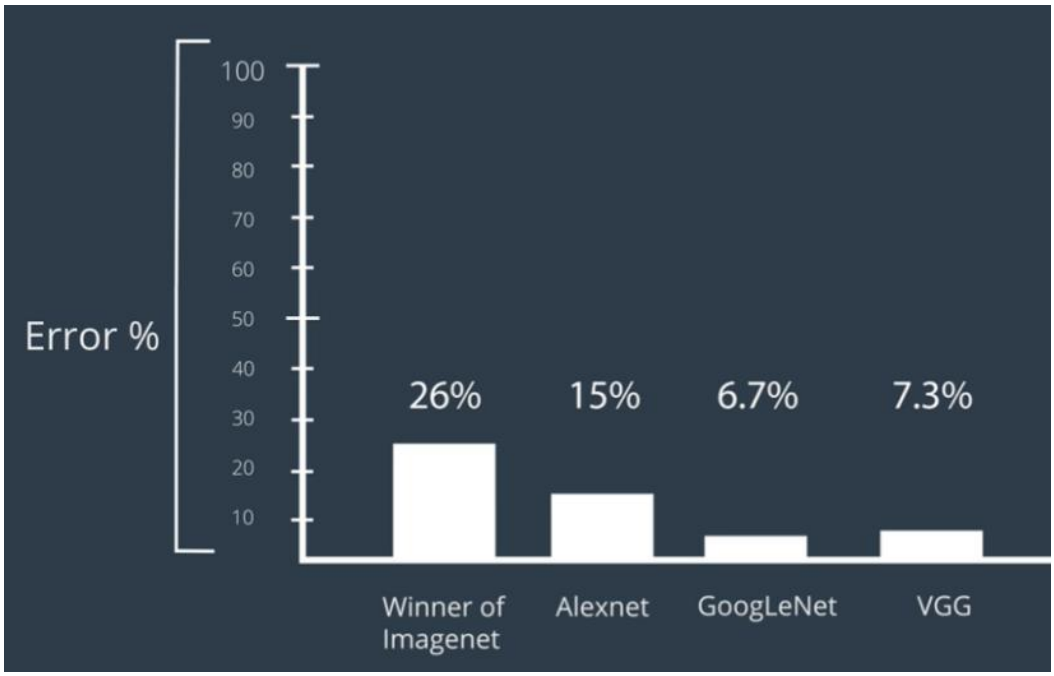
The argument `include_top` is for whether you want to include the fully-connected layer at the top of the network; unless you are actually trying to classify ImageNet's 1,000 classes, you likely want to set this to `False` and add your own additional layer for the output you desire.

Pre-processing for ImageNet weights

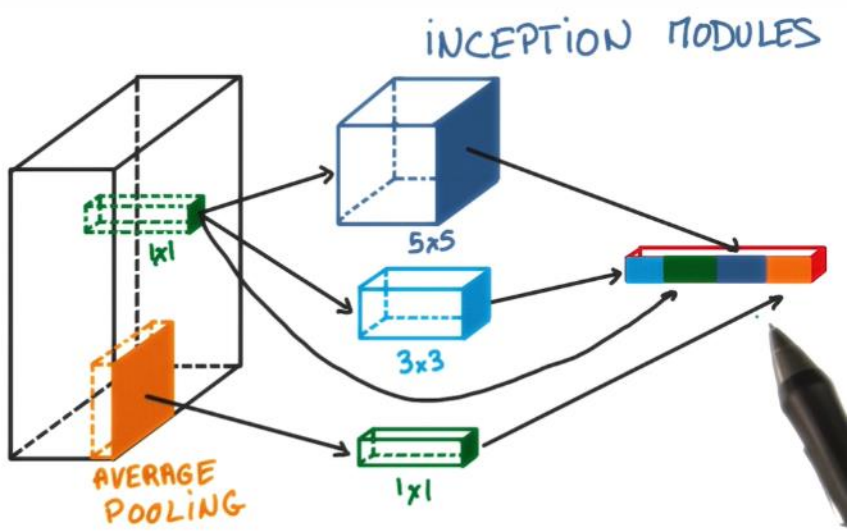
There is another item to consider before jumping into using an ImageNet pre-trained model. These networks are typically pre-trained with a specific type of pre-processing, so you need to make sure to use the same pre-processing steps, or your network's outputs will likely be erratic.

VGG uses 224x224 images as input, so that's another thing to consider.

```
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np

img_path = 'your_image.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
```



GoogLeNet is based on:



GoogLeNet/Inception in Keras

Inception is also one of the models included in Keras Applications. Utilizing this model follows pretty much the same steps as using VGG, although this time you'll use the `InceptionV3` architecture.

```
from keras.applications.inception_v3 import InceptionV3

model = InceptionV3(weights='imagenet', include_top=False)
```

Don't forget to perform the necessary pre-processing steps to any inputs you include! While the original Inception model used a 224x224 input like VGG, InceptionV3 actually uses a 299x299 input.

ResNet:

Add connections to the network to skip layers (152 layers)

ResNet in Keras

As you may have guessed, ResNet is also a model included in Keras Applications, under `ResNet50`.

```
from keras.applications.resnet50 import ResNet50

model = ResNet50(weights='imagenet', include_top=False)
```

Again, you'll need to do ImageNet-related pre-processing if you want to use the pre-trained weights for it. ResNet50 has a 224x224 input size.

Lambda Layers

In Keras, lambda layers can be used to create arbitrary functions that operate on each image as it passes through the layer.

In this project, a lambda layer is a convenient way to parallelize image normalization. The lambda layer will also ensure that the model will normalize input images when making predictions in `drive.py`.

That lambda layer could take each pixel in an image and run it through the formulas:

`pixel_normalized = pixel / 255`

`pixel_mean_centered = pixel_normalized - 0.5`

A lambda layer will look something like:

`Lambda(lambda x: (x / 255.0) - 0.5)`

Below is some example code for how a lambda layer can be used.

```
from keras.models import Sequential, Model
from keras.layers import Lambda

# set up Lambda Layer
model = Sequential()
model.add(Lambda(lambda x: (x / 255.0) - 0.5, input_shape=(160,320,3)))
...
```

Cropping Images in Keras

Original image taken from the simulator

Cropped image after passing through a Cropping2D layer

Cropping Layer Code Example

```
from keras.models import Sequential, Model
from keras.layers import Cropping2D
import cv2

# set up cropping2D layer
model = Sequential()
model.add(Cropping2D(cropping=((50,20), (0,0)), input_shape=(160,320,3)))
...
```

The example above crops:

- 50 rows pixels from the top of the image
- 20 rows pixels from the bottom of the image
- 0 columns of pixels from the left of the image
- 0 columns of pixels from the right of the image

Outputting Training and Validation Loss Metrics

In Keras, the `model.fit()` and `model.fit_generator()` methods have a `verbose` parameter that tells Keras to output loss metrics as the model trains. The `verbose` parameter can optionally be set to `verbose = 1` or `verbose = 2`.

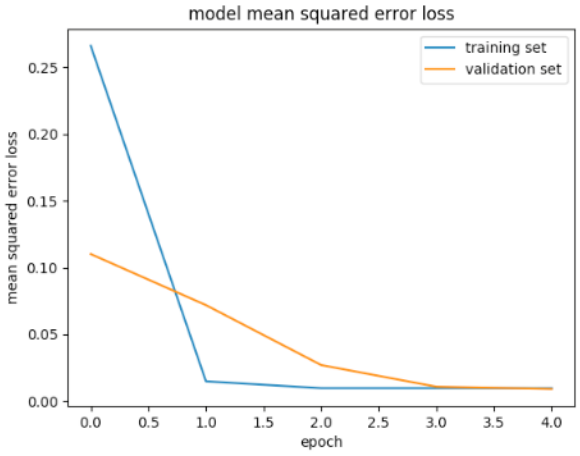
Setting `model.fit(verbose = 1)` will

- output a progress bar in the terminal as the model trains.
- output the loss metric on the training set as the model trains.
- output the loss on the training and validation sets after each epoch.

With `model.fit(verbose = 2)`, Keras will only output the loss on the training set and validation set after each epoch.

Model History Object

When calling `model.fit()` or `model.fit_generator()`, Keras outputs a history object that contains the training and validation loss for each epoch. Here is an example of how you can use the history object to visualize the loss:



The following code shows how to use the `model.fit()` history object to produce the visualization.

```
from keras.models import Model
import matplotlib.pyplot as plt

history_object = model.fit_generator(train_generator, samples_per_epoch =
    len(train_samples), validation_data =
    validation_generator,
    nb_val_samples = len(validation_samples),
    nb_epoch=5, verbose=1)

### print the keys contained in the history object
print(history_object.history.keys())

### plot the training and validation loss for each epoch
plt.plot(history_object.history['loss'])
plt.plot(history_object.history['val_loss'])
plt.title('model mean squared error loss')
plt.ylabel('mean squared error loss')
plt.xlabel('epoch')
plt.legend(['training set', 'validation set'], loc='upper right')
plt.show()
```

How to Use Generators

The images captured in the car simulator are much larger than the images encountered in the Traffic Sign Classifier Project, a size of 160 x 320 x 3 compared to 32 x 32 x 3. Storing 10,000 traffic sign images would take about 30 MB but storing 10,000 simulator images would take over 1.5 GB. That's a lot of memory! Not to mention that preprocessing data can change data types from an `int` to a `float`, which can increase the size of the data by a factor of 4.

Generators can be a great way to work with large amounts of data. Instead of storing the preprocessed data in memory all at once, using a generator you can pull pieces of the data and process them on the fly only when you need them, which is much more memory-efficient.

A generator is like a coroutine, a process that can run separately from another main routine, which makes it a useful Python function. Instead of using `return`, the generator uses `yield`, which still returns the desired output values but saves the current values of all the generator's variables. When the generator is called a second time it re-starts right after the `yield` statement, with all its variables set to the same values as before.

Below is a short quiz using a generator. This generator appends a new Fibonacci number to its list every time it is called. To pass, simply modify the generator's `yield` so it returns a list instead of `1`. The result will be we can get the first 10 Fibonacci numbers simply by calling our generator 10 times. If we need to go do something else besides generate Fibonacci numbers for a while we can do that and then always just call the generator again whenever we need more Fibonacci numbers.

```
def fibonacci():
    numbers_list = []
    while 1:
        if(len(numbers_list) < 2):
            numbers_list.append(1)
        else:
            numbers_list.append(numbers_list[-1] + numbers_list[-2])
        yield numbers_list # change this line so it yields its list instead of 1
our_generator = fibonacci()
my_output = []
for i in range(10):
    my_output = (next(our_generator))
print(my_output)
```