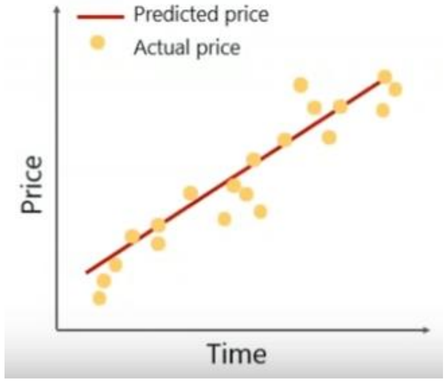
MiniFlow to TensorFlow

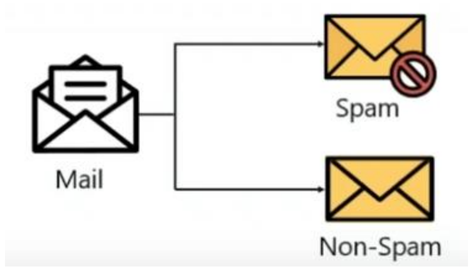
20210330

MiniFlow

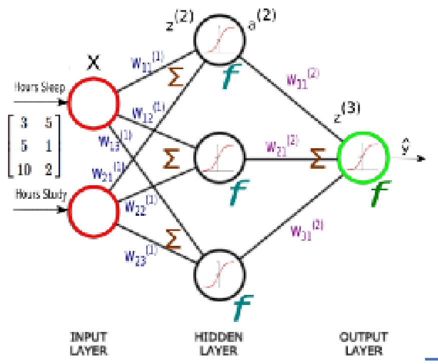
20210330



Regression



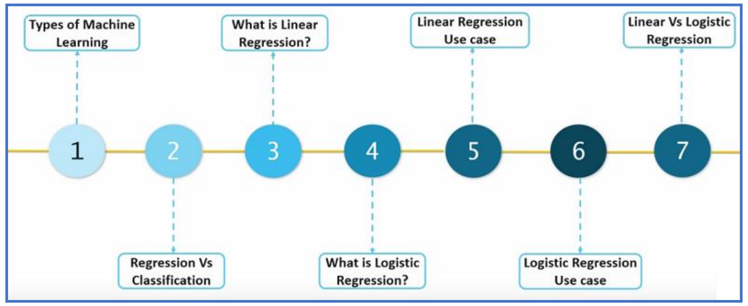
Classification



Neural network

Key Differences Between Classification and Regression

- The Classification process models a function through which the data is predicted in discrete class labels. On the other hand, regression is the process of creating a model which predict continuous quantity.
- The classification algorithms involve decision tree, logistic regression, etc. In contrast, regression tree (e.g. Random forest) and linear regression are the examples of regression algorithms.
- Classification predicts unordered data while regression predicts ordered data.
- Regression can be evaluated using root mean square error. On the contrary, classification is evaluated by measuring accuracy.


Types of Machine Learning
What is Linear Regression?
Linear Regression Use case
Linear Vs Logistic Regression
1
2
3
4
5
6
7
Regression Vs Classification
What is Logistic Regression?
Logistic Regression Use case

Reference: Linear Regression vs Logistic Regression

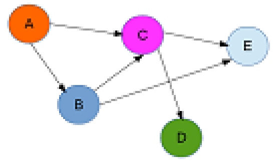
Logistic regression: The simplest form of Neural Network, that results in decision boundaries that are a straight line.

Neural Networks: A superset that includes Logistic regression and also other classifiers that can generate more complex decision boundaries. (Non-linear decision boundaries)

Move from the high level concepts down to the low level details

Build your own neural network library and implement back propagation by hand.

Introduction



MiniFlow

your own version of TensorFlow

TensorFlow

Google Brain

In this lab, you'll build a library called MiniFlow which will be your own version of TensorFlow! (link for China) TensorFlow is one of the most popular open source neural network libraries, built by the team at Google Brain over just the last few years.

Following this lab, you'll spend the remainder of this module actually working with open-source deep learning libraries like TensorFlow and Keras. So why bother building MiniFlow?
The goal of this lab is to demystify two concepts at the heart of neural networks - **backpropagation** and **differentiable graphs**.

backpropagation

Backpropagation is the process by which neural networks update the weights of the network over time.

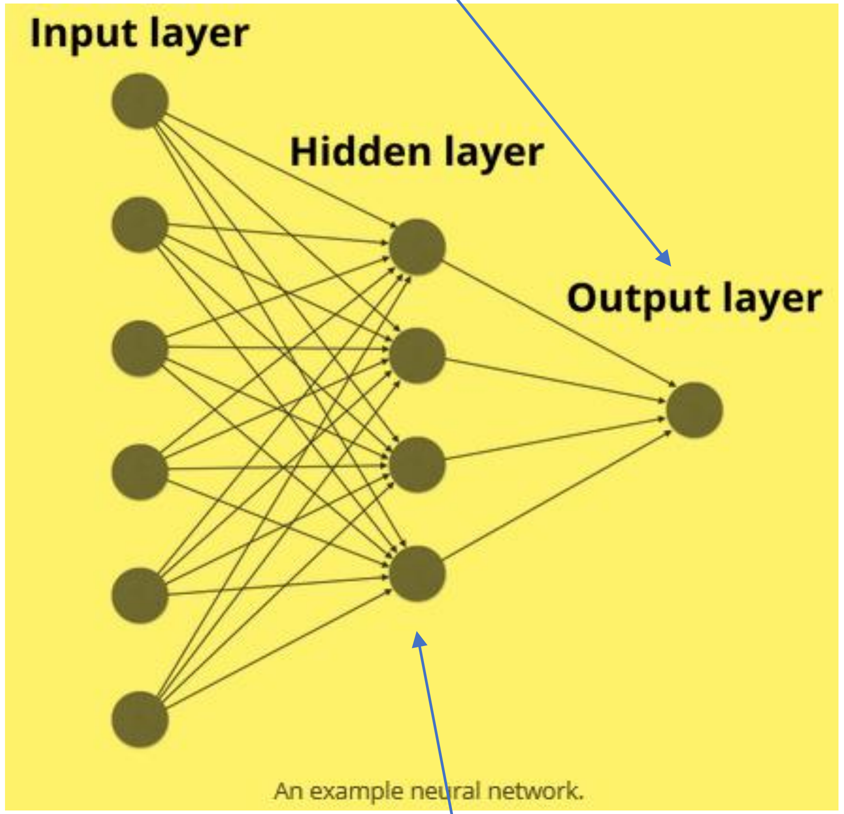
differentiable graphs

Differentiable graphs are graphs where the nodes are differentiable functions. They are also useful as *visual aids* for understanding and calculating complicated derivatives.

This is the fundamental abstraction of TensorFlow - it's a framework for creating differentiable graphs.
With graphs and backpropagation, you will be able to create your own nodes and properly compute the derivatives. Even more importantly, you will be able to think and reason in terms of these graphs. Now, let's take the first peek under the hood...

Graphs

The output value from the output layer does not get passed to a future layer (last layer!)



Layers between the input layer and the output layer are called **hidden layers**.

What is a Neural Network?

1. A neural network is a graph of mathematical functions such as linear combinations and activation functions.
2.The graph consists of **nodes**, and **edges**. Nodes in each layer (except for nodes in the input layer) perform mathematical functions using inputs from nodes in the previous layers.
3. For example, a node could represent *f(x,y)=x+y*, where *x* and *y* are input values from nodes in the previous layer.

4. Similarly, each node creates an output value which may be passed to nodes in the next layer.

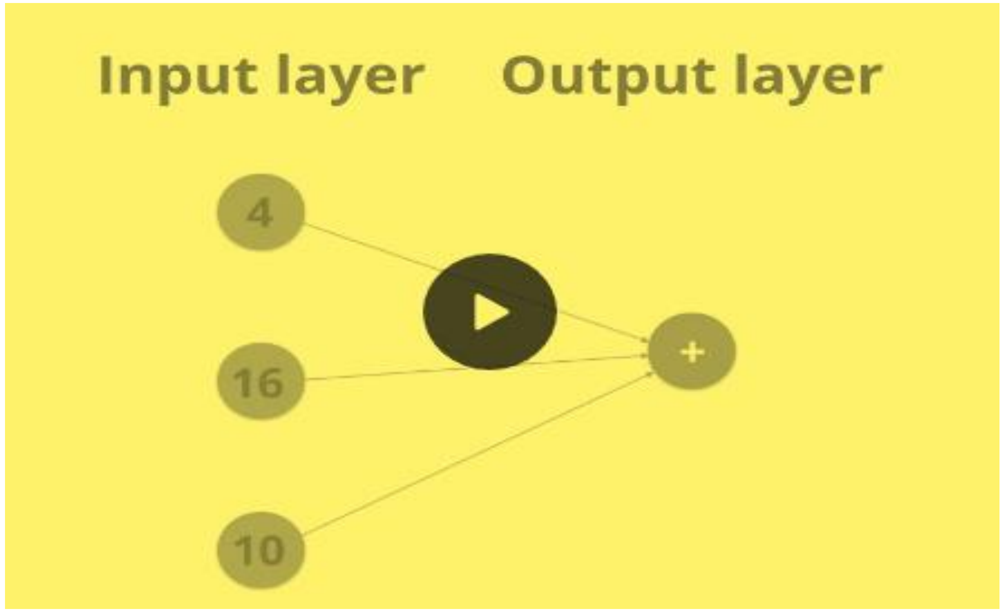
The edges in the graph describe the connections between the nodes, along which the values flow from one layer to the next. These edges can also apply operations to the values that flow along them, such as multiplying by weights, adding biases, etc..

Graphs

Forward Propagation

By propagating values from the first layer (the input layer) through all the mathematical functions represented by each node, the network outputs a value. This process is called a **forward pass**.

Here's an example of a simple forward pass.



Notice that the output layer performs a mathematical function, addition, on its inputs. There is no hidden layer.

Graphs

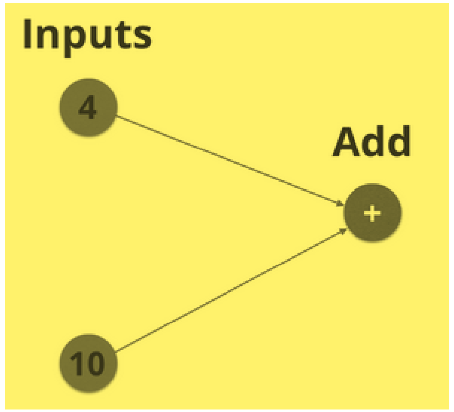
The nodes and edges create a graph structure.

There are generally two steps to create neural networks:

1. Define the graph of nodes and edges.
2. Propagate values through the graph.

MiniFlow works the same way. You'll define the nodes and edges of your network with one method and then propagate values through the graph with another method. MiniFlow comes with some starter code to help you out. Let's take a look.

Graph Quiz



Quiz Question

In the graph above, what will the value of the addition node be?

❍4 ❍10 ❍14 ❍0

MiniFlow Architecture

MiniFlow Architecture

```
class Node(object):
    def __init__(self):
        # Properties will go here!
```

Let's add two lists: one to store references to the inbound nodes, and the other to store references to the outbound nodes.

```
class Node(object):
    def __init__(self, inbound_nodes=[]):
        # Node(s) from which this Node receives values
        self.inbound_nodes = inbound_nodes
        # Node(s) to which this Node passes values
        self.outbound_nodes = []
        # For each inbound Node here, add this Node as
        # an outbound Node to _that_ Node.
        for n in self.inbound_nodes:
            n.outbound_nodes.append(self)
```

```
class Node(object):
    def __init__(self, inbound_nodes=[]):
        # Node(s) from which this Node receives values
        self.inbound_nodes = inbound_nodes
        # Node(s) to which this Node passes values
        self.outbound_nodes = []
# For each inbound Node here, add this Node as an outbound
        #Node to _that_ Node.
        for n in self.inbound_nodes:
            n.outbound_nodes.append(self)
        # A calculated value
        self.value = None
```

Each node will eventually calculate a value that represents its output. Let's initialize the value to None to indicate that it exists but hasn't been set yet.

MiniFlow Architecture

For now, let's add a placeholder method for forward propagation.

```
class Node(object):
    def __init__(self, inbound_nodes=[]):
        # Node(s) from which this Node receives values
        self.inbound_nodes = inbound_nodes
        # Node(s) to which this Node passes values
        self.outbound_nodes = []
        # For each inbound Node here, add this Node as an
          outbound Node to _that_ Node.
        for n in self.inbound_nodes:
            n.outbound_nodes.append(self)
        # A calculated value
        self.value = None

    def forward(self):
        """
        Forward propagation.

        Compute the output value based on `inbound_nodes` and
        store the result in self.value.
        """
        raise NotImplemented
```

Nodes that Calculate

While Node defines the base set of properties that every node holds, only specialized subclasses of Node will end up in the graph. For example, consider the Input subclass of Node.

```
class Input(Node):
    def __init__(self):
        # An Input node has no inbound nodes,
        # so no need to pass anything to the Node instantiator.
        Node.__init__(self)

    # NOTE: Input node is the only node where the value
    # may be passed as an argument to forward().
    #
    # All other node implementations should get the value
    # of the previous node from self.inbound_nodes
    #
    # Example:
    # val0 = self.inbound_nodes[0].value
    def forward(self, value=None):
        # Overwrite the value if one is passed in.
        if value is not None:
            self.value = value
```

MiniFlow Architecture

Unlike the other subclasses of Node, the Input subclass does not actually calculate anything. The Input subclass just holds a value, such as a data feature or a model parameter (weight/bias).

You can set value either explicitly or with the forward() method. This value is then fed through the rest of the neural network.

The Add Subclass
Add, which is another subclass of Node, actually can perform a calculation (addition).

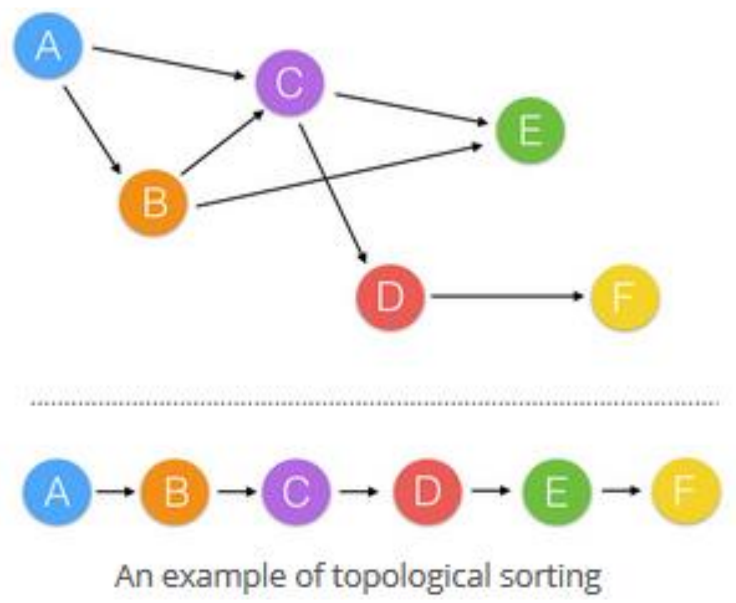
```
class Add(Node):
    def __init__(self, x, y):
        Node.__init__(self, [x, y])

    def forward(self):
        """
        You'll be writing code here in the next quiz!
        """
```

Notice the difference in the __init__ method, Add.__init__(self, [x, y]). Unlike the Input class, which has no inbound nodes, the Add class takes 2 inbound nodes, x and y, and adds the values of those nodes.

Forward Propagation

Forward propagation

MiniFlow has two methods to help you define and then run values through your graphs: topological_sort() and forward_pass().



An example of topological sorting

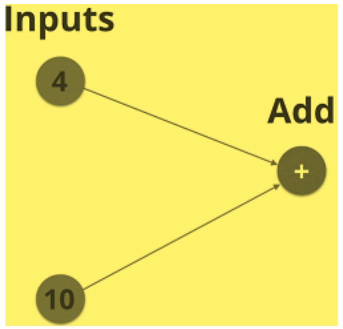
In order to define your network, you'll need to define the order of operations for your nodes. Given that the input to some node depends on the outputs of others, you need to flatten the graph in such a way where all the input dependencies for each node are resolved before trying to run its calculation. This is a technique called a topological sort. The topological_sort() function implements topological sorting using Kahn's Algorithm. The details of this method are not important, the result is; topological_sort() returns a sorted list of nodes in which all of the calculations can run in series. topological_sort() takes in a feed_dict, which is how we initially set a value for an Input node. The feed_dict is represented by the Python dictionary data structure. Here's an example use case:

```
# Define 2 `Input` nodes.
x, y = Input(), Input()
# Define an `Add` node, the two above`Input` nodes being the input.
add = Add(x, y)
# The value of `x` and `y` will be set to 10 and 20 respectively.
feed_dict = {x: 10, y: 20}
# Sort the nodes with topological sort.
sorted_nodes = topological_sort(feed_dict=feed_dict)
```

Forward Propagation

```
def forward_pass(output_node, sorted_nodes):
    """
    Performs a forward pass through a list of sorted nodes.
    Arguments:
        `output_node`: The output node of the graph (no outgoing edges).
        `sorted_nodes`: a topologically sorted list of nodes.
    Returns the output node's value
    """
    for n in sorted_nodes:
        n.forward()

    return output_node.value
```

Quiz 1 - Passing Values Forward
Create and run this graph!



The graph you'll run in this quiz. The node values may change, though!

Setup
Review nn.py and miniflow.py.
The neural network architecture is already there for you in nn.py. It's your job to finish MiniFlow to make it work.
For this quiz, I want you to:

1. Open nn.py below. You don't need to change anything. I just want you to see how MiniFlow works.
2. Open miniflow.py. Finish the forward method on the Add class. All that's required to pass this quiz is a correct implementation of forward.
3. Test your network by hitting "Test Run!" When the output looks right, hit "Submit!"

(You'll find the solution on the next page.)

Code: 09nn_py.txt

Code: 09miniflow_py.txt

Forward Propagation Solution

Here's solution (forward method of the Add class):

```
def forward(self):
    x_value = self.inbound_nodes[0].value
    y_value = self.inbound_nodes[1].value
    self.value = x_value + y_value
```

While this looks simple, I want to show you why I used x_value and y_value from the inbound_nodes array. Let's take a look at the start with Node's constructor:

```
class Node(object):
    def __init__(self, inbound_nodes=[]):
          #  Node(s) from which this Node receives values.
        self.inbound_nodes = inbound_nodes
          #  Removed everything else for brevity.
```

Of course, you weren't using Node directly, rather you used Add, which is a subclass of Node. Add's constructor is responsible for passing the inbound_nodes to Node, which happens here:

```
class Add(Node):
    def __init__(self, x, y):
        Node.__init__(self, [x, y]) #calls Node 's  constructor
    ...
```

Lastly, there's the question of why node.value holds the value of the inputs. For each node of the Input() class, the nodes are set directly when you run topological_sort:

```
def topological_sort(feed_dict):
    ...
    if isinstance(n, Input):
        n.value = feed_dict[n]
    ...
```

For other classes, the value of node.value is set in the forward pass:

```
def forward_pass(output_node, sorted_nodes):
    ...
    for n in sorted_nodes:
        n.forward()
    ...
```

Keep going to make MiniFlow more capable.
These are **ungraded** challenges as they are more of a test of your Python skills than neural network skills.

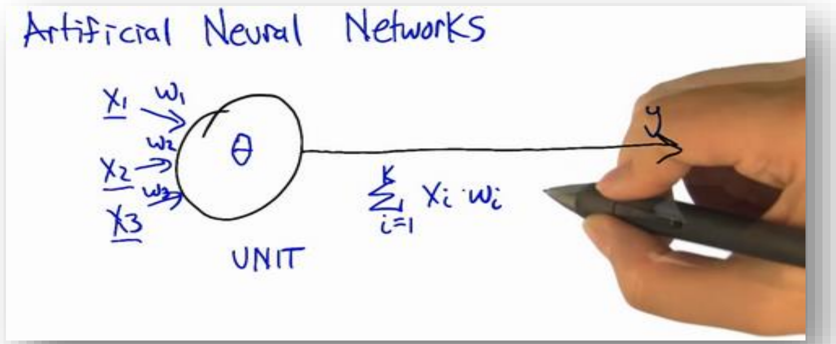
1. Can you make Add accept any number of inputs? Eg. Add(x, y, z).
2. Can you make a Mul class that multiplies *n* inputs?

Code: 10nn_py.txt **Code: 10miniflow_py.txt**

Learning and Loss

Like MiniFlow in its current state, neural networks take inputs and produce outputs. But unlike MiniFlow in its current state, neural networks can *improve* the accuracy of their outputs over time . To explore why accuracy matters, I want you to first implement a trickier (and more useful!) node than Add: the Linear node.

The Linear Function



A Neuron calculates the weighted sum of its inputs.

Think back to Neural Networks lesson. A simple artificial neuron depends on three components:

- inputs, *x* (vector)
- weights, *w* (vector)
- bias, *b* (scalar)

The output, *o*, is just the weighted sum of the inputs plus the bias:

$$o = \sum_i x_i w_i + b$$

Equation (1)

Remember, by varying the weights, you can vary the amount of influence any given input has on the output. The learning aspect of neural networks takes place during a process known as backpropagation. In backpropagation, the network modifies the weights to improve the network's output accuracy. You'll be applying all of this shortly.

In this next quiz, you'll try to build a linear neuron that generates an output by applying a simplified version of Equation (1). Linear should take an list of inbound nodes of length *n*, a list of weights of length *n*, and a bias.

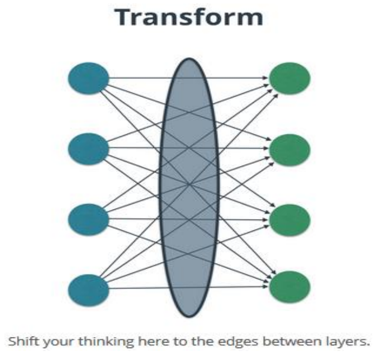
Instructions

1. Open nn.py below. Read through the neural network to see the expected output of Linear.
2. Open miniflow.py below. Modify Linear, which is a subclass of Node, to generate an output with Equation (1).

(Hint: you could use numpy to solve this quiz if you'd like, but it's possible to solve this with vanilla Python.)

Code: 11nn_py.txt Code: 11miniflow_py.txt

Linear Transform



Solution to Linear Node: Here's my solution to the last quiz:

```
class Linear(Node):
    def __init__(self, inputs, weights, bias):
        Node.__init__(self, [inputs, weights, bias])
    def forward(self):
        """
        Set self.value to the value of the linear function output.
        Your code goes here!
        """
        inputs = self.inbound_nodes[0].value
        weights = self.inbound_nodes[1].value
        bias = self.inbound_nodes[2].value
        self.value = bias
        for x, w in zip(inputs, weights):
            self.value += x * w
```

In the solution, I set self.value to the bias and then loop through the inputs and weights, adding each weighted input to self.value. Notice calling .value on self.inbound_nodes[0] or self.inbound_nodes[1] gives us a list.

Linear algebra nicely reflects the idea of transforming values between layers in a graph. In fact, the concept of a transform does exactly what a layer should do - it converts inputs to outputs in many dimensions.
Let's go back to our equation for the output.

$$o = \sum_i x_i w_i + b$$ Equation (1)

For the rest of this section we 'll denote x as X and w as W since they are now matrices, and b is now a vector instead of a scalar. Consider a Linear node with 1 input and k outputs (mapping 1 input to k outputs). In this context an input/output is synonymous with a feature.
In this case X is a 1 by 1 matrix.

$$X = [X_{11}]$$ 1 by 1 matrix, 1 element.

W becomes a 1 by k matrix (looks like a row).

$$W = [W_{11} \quad W_{12} \quad W_{13} \quad \dots \quad W_{1k}]$$

A 1 by k weights row matrix.

The result of the matrix multiplication of X and W is a 1 by k matrix. Since b is also a 1 by k row matrix (1 bias per output), b is added to the output of the X and W matrix multiplication.

Linear Transform

What if we are mapping n inputs to k outputs?
Then *X* is now a 1 by n matrix and *W* is a n by k matrix. The result of the matrix multiplication is still a 1 by k matrix so the use of the biases remain the same.

$$X = \begin{bmatrix} X_{11} & X_{12} & X_{13} & \dots & X_{1n} \end{bmatrix}$$

X is now a 1 by n matrix, n inputs/features.

$$W = \begin{bmatrix} W_{11} & W_{12} & W_{13} & \dots & W_{1k} \\ W_{21} & W_{22} & W_{23} & \dots & W_{2k} \\ W_{31} & W_{32} & W_{33} & \dots & W_{3k} \\ \vdots & & & \ddots & \vdots \\ W_{n1} & W_{n2} & W_{n3} & \dots & W_{nk} \end{bmatrix}$$

W is now a n by k matrix.

$$b = \begin{pmatrix} b_1 & b_2 & b_3 & \dots & b_k \end{pmatrix}$$

Row matrix of biases, one for each output.

Let's take a look at an example of n inputs. Consider an 28px by 28px greyscale image, as is in the case of images in the MNIST dataset. We can reshape the image such that it's a 1 by 784 matrix, n = 784. Each pixel is an input/feature. Here's an animated example emphasizing a pixel is a feature.

影片片長:00:10

In practice, it's common to feed in multiple data examples in each forward pass rather than just 1. The reasoning for this is the examples can be processed in parallel, resulting in big performance gains. The number of examples is called the *batch size*. Common numbers for the batch size are 32, 64, 128, 256, 512. Generally, it's the most we can comfortably fit in memory.
What does this mean for *X*, *W* and *b*?
X becomes a m by n matrix and *W* and *b* remain the same. The result of the matrix multiplication is now m by k, so the addition of *b* is broadcast over each row.

$$X = \begin{bmatrix} X_{11} & X_{12} & X_{13} & \dots & X_{1n} \\ X_{21} & X_{22} & X_{23} & \dots & X_{2n} \\ X_{31} & X_{32} & X_{33} & \dots & X_{3n} \\ \vdots & & & \ddots & \vdots \\ X_{m1} & X_{m2} & X_{m3} & \dots & X_{mn} \end{bmatrix}$$

X is now an m by n matrix. Each row has n inputs/features.

Linear Transform

In the context of MNIST each row of X is an image reshaped from 28 by 28 to 1 by 784.

Equation (1) turns into:

$$Z = XW + b$$

Equation (2).

Equation (2) can also be viewed as $Z = XW + B$ where B is the biases vector, b, stacked m times as a row. Due to broadcasting it's abbreviated to $Z = XW + b$.

I want you to rebuild Linear to handle matrices and vectors using the venerable Python math package numpy to make your life easier. numpy is often abbreviated as np, so we'll refer to it as np when referring to code.

I used np.array (documentation) to create the matrices and vectors. You'll want to use np.dot, which functions as matrix multiplication for 2D arrays (documentation), to multiply the input and weights matrices from Equation (2). It's also worth noting that numpy actually overloads the __add__ operator so you can use it directly with np.array (eg. np.array() + np.array()).

Instructions

1. Open nn.py. See how the neural network implements the `Linear` node.
2. Open miniflow.py. Implement Equation (2) within the forward pass for the `Linear` node.
3. Test your work!

Code: 12nn_py.txt　　Code: 12miniflow_py.txt

Sigmoid Function

Here's my solution to the last quiz:

```
class Linear(Node):
    def __init__(self, X, W, b):
        # Notice the ordering of the inputs passed to the
        # Node constructor.
        Node.__init__(self, [X, W, b])

    def forward(self):
        X = self.inbound_nodes[0].value
        W = self.inbound_nodes[1].value
        b = self.inbound_nodes[2].value
        self.value = np.dot(X, W) + b
```

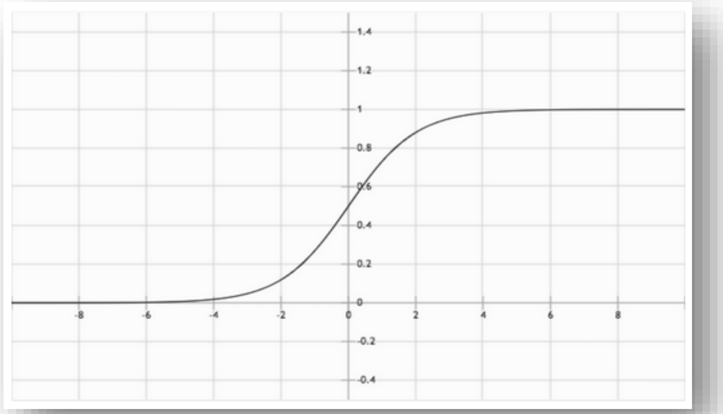
Nothing fancy in my solution. I pulled the value of the X, W and b from their respective inputs. I used np.dot to handle the matrix multiplication.

Neural networks take advantage of alternating transforms and activation functions to better categorize outputs. The sigmoid function is among the most common activation functions.

Sigmoid Function

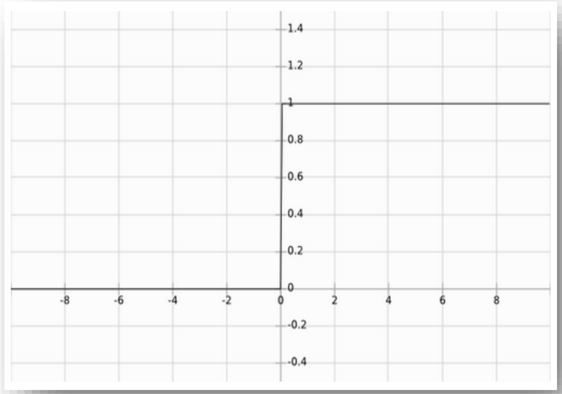
$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

Equation (3)



Graph of the sigmoid function. Notice the "S" shape.

Linear transforms are great for simply *shifting* values, but neural networks often require a more nuanced transform. For instance, one of the original designs for an artificial neuron, the perceptron, exhibit binary output behavior. Perceptrons compare a weighted input to a threshold. When the weighted input exceeds the threshold, the perceptron is **activated** and outputs 1, otherwise it outputs 0.
You could model a perceptron's behavior as a step function.



Example of a step function (The jump between y = 0 and y = 1 should be instantaneous).

Sigmoid Function

Activation, the idea of binary output behavior, generally makes sense for classification problems. For example, if you ask the network to hypothesize if a handwritten image is a '9', you're effectively asking for a binary output - *yes*, this is a '9', or *no*, this is not a '9'. A step function is the starkest form of a binary output, which is great, but step functions are not continuous and not differentiable, which is *very bad*. Differentiation is what makes gradient descent possible. The sigmoid function, Equation (3) above, replaces thresholding with a beautiful S-shaped curve (also shown above) that mimics the activation behavior of a perceptron while being differentiable. As a bonus, the sigmoid function has a very simple derivative that that can be calculated from the sigmoid function itself.

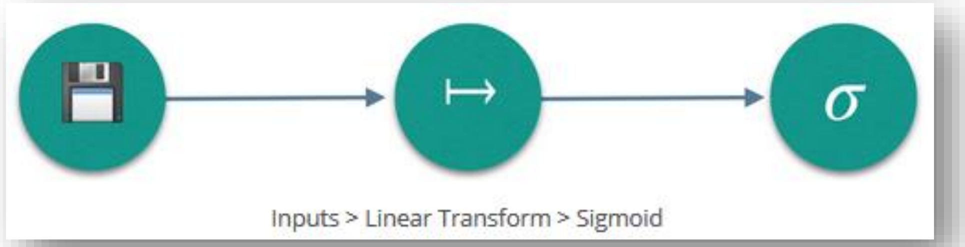
$$\boxed{\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))}$$

Equation (4). σ represents Equation (3)

Notice that the sigmoid function only has one parameter. Remember that sigmoid is an *activation* function (*non-linearity*), meaning it takes a single input and performs a mathematical operation on it.

Conceptually, the sigmoid function makes decisions. When given weighted features from some data, it indicates whether or not the features contribute to a classification. In that way, a sigmoid activation works well following a linear transformation. As it stands right now with random weights and bias, the sigmoid node's output is also random. The process of learning through backpropagation and gradient descent, which you will implement soon, modifies the weights and bias such that activation of the sigmoid node begins to match expected outputs.

Now that I've given you the equation for the sigmoid function, I want you to add it to the MiniFlow library. To do so, you'll want to use np.exp (documentation) to make your life much easier.
You'll be using Sigmoid in conjunction with Linear. Here's how it should look:



Inputs > Linear Transform > Sigmoid

Instructions

1.Open nn.py to see how the network will use Sigmoid.
2.Open miniflow.py. Modify the forward method of the Sigmoid class to reflect the sigmoid function's behavior.
3.Test your work! Hit "Submit" when your Sigmoid works as expected.

Code: 13nn_py.txt   Code: 13miniflow_py.txt

Cost

Here's how I implemented the sigmoid function.

```
class Sigmoid(Node):
    def __init__(self, node):
        Node.__init__(self, [node])
    def _sigmoid(self, x):
        """
        This method is separate from `forward` because it
        will be used with `backward` as well.
        `x`: A numpy array-like object.
        """
        return 1. / (1. + np.exp(-x))  # the `.` ensures that `1` is a float
    def forward(self):
        input_value = self.inbound_nodes[0].value
        self.value = self._sigmoid(input_value)
```

You have used weights and biases to compute outputs. And you've used an activation function to categorize the output. As you may recall, neural networks improve the **accuracy** of their outputs by modifying weights and biases in response to training against labeled datasets. There are many techniques for defining the accuracy of a neural network, all of which center on the network's ability to produce values that come as close as possible to known correct values. People use different names for this accuracy measurement, often terming it **loss** or **cost**. I'll use the term *cost* most often.

For this lab, you will calculate the cost using the mean squared error (MSE). It looks like so:

$$C(w,b) = \frac{1}{m}\sum_x \left\|y(x) - a\right\|^2 \qquad \text{Equation (5)}$$

Here *w* denotes the collection of all weights in the network, *b* all the biases, *m* is the total number of training examples, *a* is the approximation of *y(x)* by the network, both *a* and *y(x)* are vectors of the same length.
The collection of weights is all the weight matrices flattened into vectors and concatenated to one big vector. The same goes for the collection of biases except they're already vectors so there's no need to flatten them prior to the concatenation.
Here's an example of creating *w* in code:

```
# 2 by 2 matrices
w1  = np.array([[1, 2], [3, 4]])
w2  = np.array([[5, 6], [7, 8]])
# flatten
w1_flat = np.reshape(w1, -1)
w2_flat = np.reshape(w2, -1)
w = np.concatenate((w1_flat, w2_flat))
# array([1, 2, 3, 4, 5, 6, 7, 8])
```

Cost

It's a nice way to abstract all the weights and biases used in the neural network and makes some things easier to write as we'll see soon in the upcoming gradient descent sections.

NOTE: It's not required you do this in your code! It's just easier to do this talk about the weights and biases as a collective than consider them invidually.

The cost, *C*, depends on the difference between the correct output, *y(x)*, and the network's output, *a*. It's easy to see that no difference between *y(x)* and *a* (for all values of *x*) leads to a cost of 0.

This is the ideal situation, and in fact the learning process revolves around minimizing the cost as much as possible.

Instructions
For this quiz, you will run the forward pass against the network in nn.py. I want you to finish implementing the MSE method so that it calculates the cost from the equation above.
I recommend using the np.square (documentation) method to make your life easier.

1. Check out nn.py to see how MSE will calculate the cost.
2. Open miniflow.py. Finish building MSE.
3. Test your network! See if the cost makes sense given the inputs by playing with nn.py.

Code: 14nn_py.txt

Code: 14miniflow_py.txt

Gradient Descent Part 1

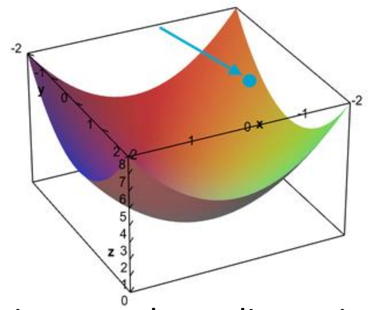
Here's how I implemented MSE:

```
class MSE(Node):
    def __init__(self, y, a):
        """
        The mean squared error cost function.Should be used as the last
         node for a network.
        """
        # Call the base class' constructor.
        Node.__init__(self, [y, a])
    def forward(self):
        """
        Calculates the mean squared error.
        """
        # NOTE: We reshape these to avoid possible matrix/vector broadcast errors.
        # For example, if we subtract an array of shape (3,) from an array of shape
        # (3,1) we get an array of shape(3,3) as the result when we want
        # an array of shape (3,1) instead.
        # Making both arrays (3,1) insures the result is (3,1) and does
        # an elementwise subtraction as expected.
        y = self.inbound_nodes[0].value.reshape(-1, 1)
        a = self.inbound_nodes[1].value.reshape(-1, 1)
        m = self.inbound_nodes[0].value.shape[0]
        diff = y - a
        self.value = np.mean(diff**2)
```

The math behind MSE reflects Equation (5), where *y* is target output and *a* is output computed by the neural network. We then square the difference diff**2, alternatively, this could be np.square(diff). Lastly we need to sum the squared differences and divide by the total number of examples *m*. This can be achieved in with np.mean or (1 /m) * np.sum(diff**2).
Note the order of y and a doesn't actually matter, we could switch them around (a - y) and get the same value.

Backpropagation
Next we need to start a backwards pass, which starts with backpropagation. Backpropagation is the process by which the network runs error values backwards. During this process, the network calculates the way in which the weights need to change (also called the gradient) to reduce the overall error of the network. Changing the weights usually occurs through a technique called gradient descent. Making sense of the purpose of backpropagation comes more easily after you work through the intended outcome. I'll come back to backpropagation in a bit, but first, I want to dive deeper into gradient descent.

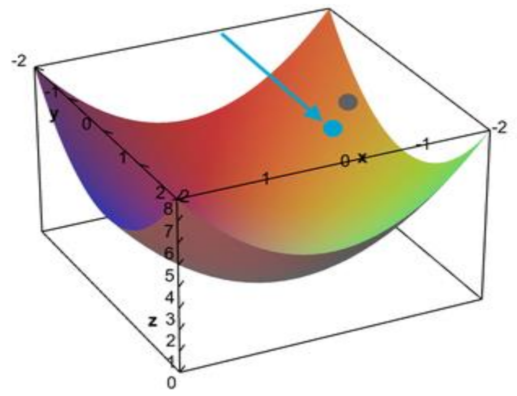
Gradient Descent


A point on a three dimension surface.

Gradient Descent Part 1

Gradient descent works by first calculating the slope of the plane at the current point, which includes calculating the partial derivatives of the loss with respect to all of the parameters. This set of partial derivatives is called the **gradient**. Then it uses the gradient to modify the weights such that the next forward pass through the network moves the output lower in the hypersurface. Physically, this would be the same as measuring the slope of the valley at the location of the ball, and then moving the ball a small amount in the direction of the slope. Over time, it's possible to find the bottom of the valley with many small movements.

While gradient descent works remarkably well, the technique isn't **guaranteed** to find the absolute minimum difference between the network's output and the known output. Why?



It moved a little bit!

Quiz Question

How might gradient descent fail to find the absolute minimum difference between the neural network's output and the known output? Select all that apply.

☐A minimum may not exist at all.
☐ There may be more than one absolute minima.
☐ It may get stuck in a local minimum while the absolute minimum may be "over the next hill" (in the ball analogy).

☐ The learning rate (determines how much the point moves) will cause the point to overshoot the absolute minimum.

Gradient Descent Part 2

Journey to the Bottom of the Valley

We know we'd like to move the ball to the bottom of the valley, but how do we accomplish this?

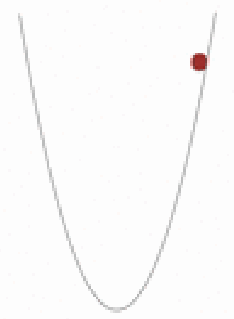
Intuitively, we want to push the ball downhill. And that makes sense, but when we're talking about our cost function, how do we know which way is downhill?

Luckily, the gradient provides this exact information.

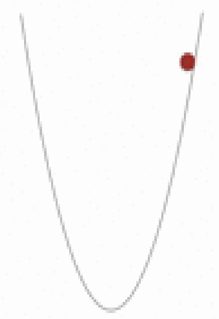
Technically, the gradient actually points uphill, in the direction of **steepest ascent**. But if we put a - sign at the front this value, we get the direction of **steepest descent**, which is what we want.

You'll learn more about the gradient in a moment, but, for now, just think of it as a vector of numbers. Each number represents the amount by which we should adjust a corresponding weight or bias in the neural network. Adjusting all of the weights and biases by the gradient values reduces the cost (or error) of the network.

Now we know where to push the ball. The next thing to consider is how much force should be applied to the push. This is known as the learning rate, which is an apt name since this value determines how quickly or slowly the neural network learns. You might be tempted to set a really big learning rate, so the network learns really fast, right? Be careful! If the value is too large you could overshoot the target and eventually diverge.



Convergence. This is the ideal behaviour.



Divergence. This can happen when the learning rate is too large.

Gradient Descent Part 2

So what is a good learning rate, then?

This is more of a guessing game than anything else but empirically values in the range 0.1 to 0.0001 work well. The range 0.001 to 0.0001 is popular, as 0.1 and 0.01 are sometimes too large.

Here's the formula for gradient descent (pseudocode):

```
x = x - learning_rate * gradient_of_x
```

x is a parameter used by the neural network (i.e. a single weight or bias).
We multiply gradient_of_x (the uphill direction) by learning_rate (the force of the push) and then subtract that from x to make the push go downhill.

Setup
For this quiz you'll complete TODOs in both the f.py and gd.py files.
Tasks:
- Set the learning_rate in f.py.
- Complete the gradient descent implementation in gradient_descent_update function in gd.py.

Notes:
- Setting the learning_rate to 0.1 should result in x -> 0 and f(x) -> 5 if you've implemented gradient descent correctly.
- Play around with different values for the learning rate. Try very small values, values close to 1, above 1, etc. What happens?

Code: 15f_py.txt Code: 15gd_py.txt

Backpropagation

Gradient Descent Solution

```
def gradient_descent_update(x, gradx, learning_rate):
"""
Performs a gradient descent update.
"""
x = x - learning_rate * gradx
# Return the new value for x
return x
```

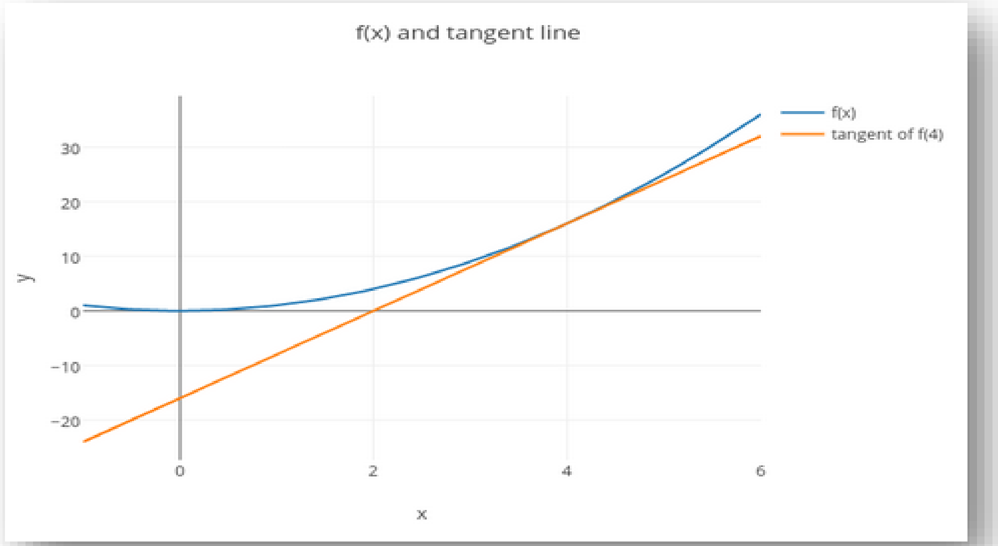
The Gradient & Backpropagation

Specifically we'll focus on the following insight: *In order to figure out how we should alter a parameter to minimize the cost, we must first find out what effect that parameter has on the cost.* That makes sense. After all, we can't just blindly change parameter values and hope to get meaningful results. The gradient takes into account the effect each parameter has on the cost, so that's how we find the direction of steepest ascent. How do we determine the effect a parameter has on the cost? This technique is famously known as **backpropagation** or **reverse-mode differentiation**. Those names might sound intimidating, but behind it all, it's just a clever application of the **chain rule**. Before we get into the chain rule let's revisit plain old derivatives.

Derivatives

In calculus, the derivative tells us how something changes with respect to something else. Or, put differently, how *sensitive* something is to something else.

Let's take the function f(x)=x^2 as an example. In this case, the derivative of f(x) is 2x. Another way to state this is, "the derivative of f(x) with respect to x is 2x". Using the derivative, we can say *how much* a change in x effects f(x). For example, when x is 4, the derivative is 8 (2x=2∗4=8). This means that if x is increased or decreased by 1 unit, then f(x) will increase or decrease by 8.



f(x) and the tangent line of f(x) when x = 4.

Backpropagation

Chain Rule

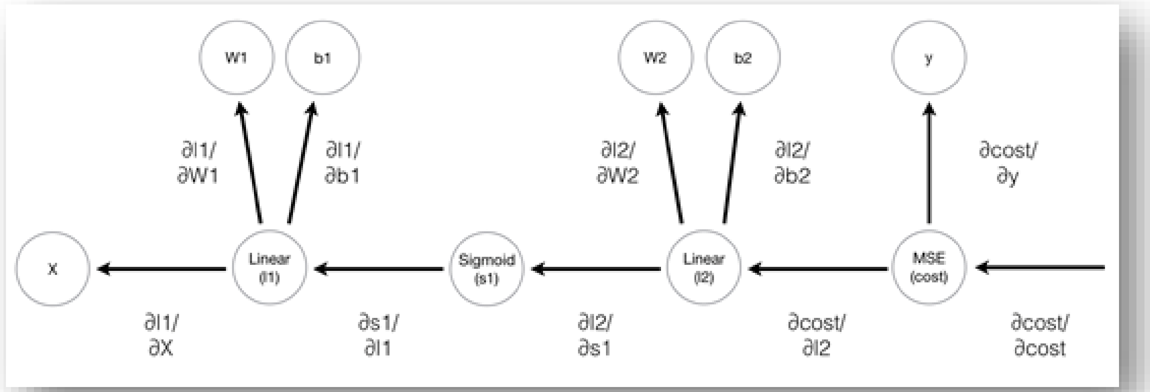
The gradient is a vector of all these derivatives. In reality, neural networks are a composition of functions, so computing the derivative of the cost w.r.t a parameter isn't quite as straightforward as calculating the derivative of a polynomial function like f(x)=x^2. This is where the chain rule comes into play.

Say we have a new function $f \circ g(x) = f(g(x))$. We can calculate the derivative of $f \circ g$ w.r.t x, denoted $\frac{\partial f \circ g}{\partial x}$, by applying the chain rule.

$\frac{\partial f \circ g}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$ The way to think about this is: In order to know the effect x has on *f*, we first need to know the effect x has on g, and then the effect g has on *f*. Let's now look at a more complex example. Consider the following neural network in MiniFlow:

```
X, y = Input(), Input()
W1, b1 = Input(), Input()
W2, b2 = Input(), Input()

l1 = Linear(X, W1, b1)
s1 = Sigmoid(l1)
l2 = Linear(s1, W2, b2)
cost = MSE(l2, y)
```

This also can be written as a composition of functions MSE(Linear(Sigmoid(Linear(X, W1, b1)), W2, b2), y). Our goal is to adjust the weights and biases represented by the Input nodes W1, b1, W2, b2, such that the cost is minimized.



Graph of the above neural network. The backward pass and gradients flowing through are illustrated.

Backpropagation

First, we unwrap the derivative of **cost** w.r.t. ***l*1** (the input to the Sigmoid node). Once again, apply the chain rule:

$$\frac{\partial cost}{\partial l1} = \frac{\partial cost}{\partial s1}\frac{\partial s1}{\partial l1}$$

We can unwrap $\frac{\partial cost}{\partial s1}$ further:

$$\frac{\partial cost}{\partial s1} = \frac{\partial cost}{\partial l2}\frac{\partial l2}{\partial s1}$$

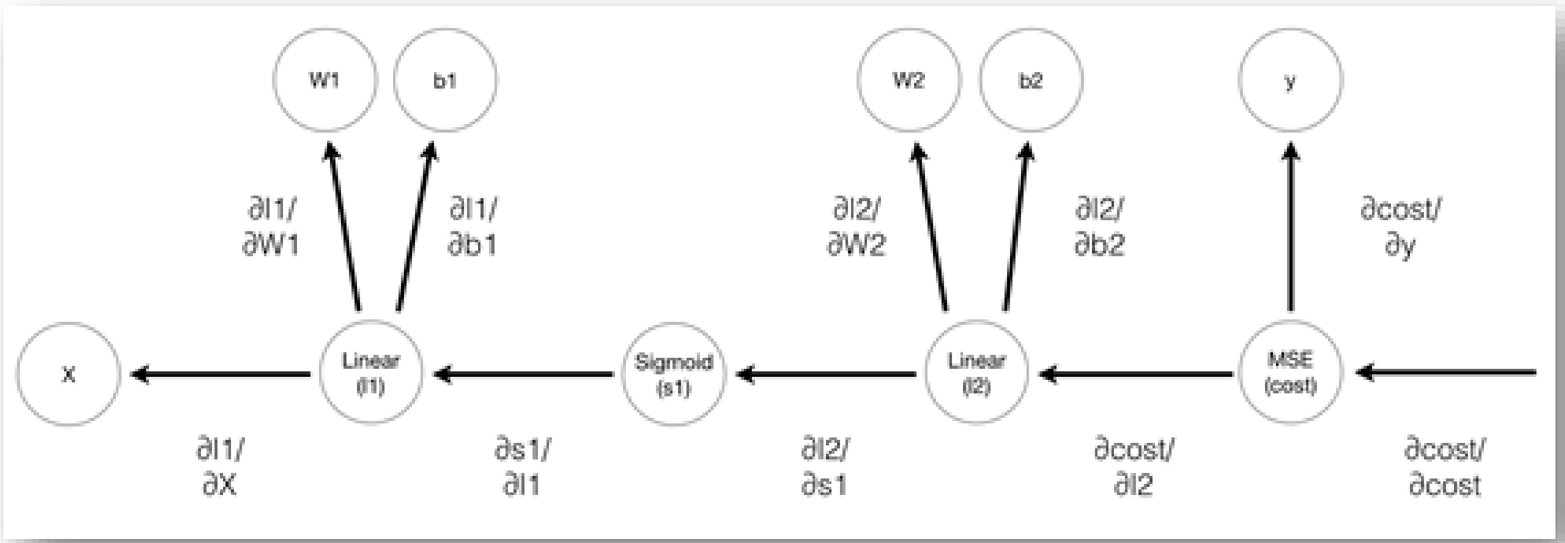
Finally:

$$\frac{\partial cost}{\partial l1} = \frac{\partial cost}{\partial l2}\frac{\partial l2}{\partial s1}\frac{\partial s1}{\partial l1}$$

In order to calculate the derivative of cost w.r.t ***l*1** we need to figure out these 3 values:

- $\frac{\partial s1}{\partial l1}$,
- $\frac{\partial l2}{\partial s1}$
- $\frac{\partial cost}{\partial l2}$

Backpropagation makes computing these values convenient.



During backpropagation, the derivatives of nodes in the graph are computed back to front. In the case of the 3 above values, $\frac{\partial cost}{\partial l2}$ would be computed first, followed by $\frac{\partial l2}{\partial s1}$ and $\frac{\partial s1}{\partial l1}$. Thus, if we compute $\frac{\partial s1}{\partial l1}$, then we can also *assume* the 2 other values have already been computed! This insight makes backpropagation much easier to implement. When computing the backward pass for a Node we only need to concern ourselves with the computation of that node w.r.t its inputs.

Backpropagation

```
class Node(object):
    """
    Base class for nodes in the network.
    Arguments:
        `inbound_nodes`: A list of nodes with edges into this node.
    """
    def __init__(self, inbound_nodes=[]):
        """
        Node's constructor (runs when the object is instantiated). Sets
        properties that all nodes need.
        """
        # A list of nodes with edges into this node.
        self.inbound_nodes = inbound_nodes
        # The eventual value of this node. Set by running
        # the forward() method.
        self.value = None
        # A list of nodes that this node outputs to.
        self.outbound_nodes = []
        # New property! Keys are the inputs to this node and
        # their values are the partials of this node with
        # respect to that input.
        self.gradients = {}
        # Sets this node as an outbound node for all of
        # this node's inputs.
        for node in inbound_nodes:
            node.outbound_nodes.append(self)
```

```
    def forward(self):
        """
        Every node that uses this class as a base class will
        need to define its own `forward` method.
        """
        raise NotImplementedError

    def backward(self):
        """
        Every node that uses this class as a base class will
        need to define its own `backward` method.
        """
        raise NotImplementedError
```

The second change is to the helper function forward_pass(). That function has been replaced with forward_and_backward().

```
def forward_and_backward(graph):
    """
    Performs a forward pass and a backward pass through a list of sorted nodes.
    Arguments:
        `graph`: The result of calling `topological_sort`.
    """
    # Forward pass
    for n in graph:
        n.forward()

    # Backward pass
    # see: https://docs.python.org/2.3/whatsnew/section-slices.html
    for n in graph[::-1]:
        n.backward()
```

Backpropagation

Setup

Here's the derivative of the sigmoid function w.r.t x:

$sigmoid(x)=1/(1+exp(-x))$

$$\frac{\partial sigmoid}{\partial x} = sigmoid(x)*(1- sigmoid(x))$$

- Complete the implementation of backpropagation for the Sigmoid node by finishing the backward method in miniflow.py.
- The backward methods for all other nodes have already been implemented. Taking a look at them might be helpful.

Code: 16nn_py.txt Code: 16miniflow_py.txt

Stochastic Gradient Descent

Here's my solution to the last quiz.

```
class Sigmoid(Node):
    """
    Represents a node that performs the sigmoid
    activation function.
    """

    def __init__(self, node):
        # The base class constructor.
        Node.__init__(self, [node])

    def _sigmoid(self, x):
        """
        This method is separate from `forward` because it
        will be used with `backward` as well.
        `x`: A numpy array-like object.
        """
        return 1. / (1. + np.exp(-x))

   def forward(self):
        """
        Perform the sigmoid function and set the value.
        """
        input_value = self.inbound_nodes[0].value
        self.value = self._sigmoid(input_value)
```

```
def backward(self):
    #Calculates the gradient using the derivative of  the sigmoid function.
    # Initialize the gradients to 0.
    self.gradients = {n: np.zeros_like(n.value) for n in self.inbound_nodes}
    # Sum the derivative with respect to the input over all the outputs.
    for n in self.outbound_nodes:
      grad_cost = n.gradients[self]
      sigmoid = self.value
      self.gradients[self.inbound_nodes[0]] += sigmoid * (1 - sigmoid) * grad_cost
```

The last line implements the derivative, $\frac{\partial sigmoid}{\partial x} \frac{\partial cost}{\partial sigmoid}$.

Replacing the math expression with code:

$\frac{\partial sigmoid}{\partial x}$ is sigmoid * (1- sigmoid) and $\frac{\partial cost}{\partial sigmoid}$ is grad_cost

Now that you have the gradient of the cost with respect to each input (the return value from forward_and_backward()) your network can start learning! To do so, you will implement a technique called **Stochastic Gradient Descent**.

Stochastic Gradient Descent

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a version of Gradient Descent where on each forward pass a batch of data is randomly sampled from total dataset. SGD is an approximation of Gradient Descent, the more batches processed by the neural network, the better the approximation.

A naïve implementation of SGD involves:

1. Randomly sample a batch of data from the total dataset.
2. Running the network forward and backward to calculate the gradient (with data from (1)).
3. Apply the gradient descent update.
4. Repeat steps 1-3 until convergence or the loop is stopped by another mechanism (i.e. the number of epochs).

If all goes well, the network's loss should generally trend downwards, indicating more useful weights and biases over time.

So far, steps 1 and 4 are already implemented. It will be your job to implement MiniFlow can already do step 2. In the following quiz, step 3.

As a reminder, here's the gradient descent update equation, where α represents the learning rate:

$$x = x - \alpha * \frac{\partial cost}{\partial x}$$

We're also going to use an actual dataset for this quiz, the Boston Housing dataset. After training the network will be able to predict prices of Boston housing!

Boston's Back Bay

By Robbie Shade (Flickr: Boston's Back Bay) [CC BY 2.0 (http://creativecommons.org/licenses/by/2.0)], via Wikimedia Commons

Each example in the dataset is a description of a house in the Boston suburbs, the description consists of 13 numerical values (features). Each example also has an associated price. With SGD, we're going to minimize the MSE between the actual price and the price predicted by the neural network based on the features.

Stochastic Gradient Descent

If all goes well the output should look something like this:

When the batch size is 11:

```
Total number of examples = 506
Epoch: 1, Loss: 140.256
Epoch: 2, Loss: 34.570
Epoch: 3, Loss: 27.501
Epoch: 4, Loss: 25.343
Epoch: 5, Loss: 20.421
Epoch: 6, Loss: 17.600
Epoch: 7, Loss: 18.176
Epoch: 8, Loss: 16.812
Epoch: 9, Loss: 15.531
Epoch: 10, Loss: 16.429
```

When the batch size is the same as the total number of examples (batch is the whole dataset):

```
Total number of examples = 506
Epoch: 1, Loss: 646.134
Epoch: 2, Loss: 587.867
Epoch: 3, Loss: 510.707
Epoch: 4, Loss: 446.558
Epoch: 5, Loss: 407.695
Epoch: 6, Loss: 324.440
Epoch: 7, Loss: 295.542
Epoch: 8, Loss: 251.599
Epoch: 9, Loss: 219.888
Epoch: 10, Loss: 216.155
```

Notice the *cost* or *loss* trending towards 0.

Instructions

1. Open nn.py. See how the network runs with this new architecture.
2. Find the sgd_update method in miniflow.py and implement SGD.
3. Test your network! Does your loss decrease with more epochs?

Note! The virtual machines on which we run your code have time limits. If your network takes more than 10 seconds to run, you will get a timeout error. Keep this in mind as you play with the number of epochs.

Code: 17nn_py.txt Code: 17miniflow_py.txt

SGD Solution

Here's my solution to the last quiz.

```
def sgd_update(trainables, learning_rate=1e-2):
    """
    Updates the value of each trainable with SGD.
    Arguments:
   `trainables`: A list of `Input` nodes representing weights/biases.
   `learning_rate`: The learning rate.
    """
    # Performs SGD
    # Loop over the trainables
    for t in trainables:
        # Change the trainable's value by subtracting the learning rate
        # multiplied by the partial of the cost with respect to this
        # trainable.
        partial = t.gradients[t]
        t.value -= learning_rate * partial
```

Take a look at the last few lines:

```
# Performs SGD
# Loop over the trainables
for t in trainables:
    # Change the trainable's value by subtracting the learning rate
    # multiplied by the partial of the cost with respect to this
    # trainable.
    partial = t.gradients[t]
    t.value -= learning_rate * partial
```

There are two keys steps. First, the partial of the cost (*C*) with respect to the trainable t is accessed.

```
partial = t.gradients[t]
```

Second, the value of the trainable is updated according to Equation (12).

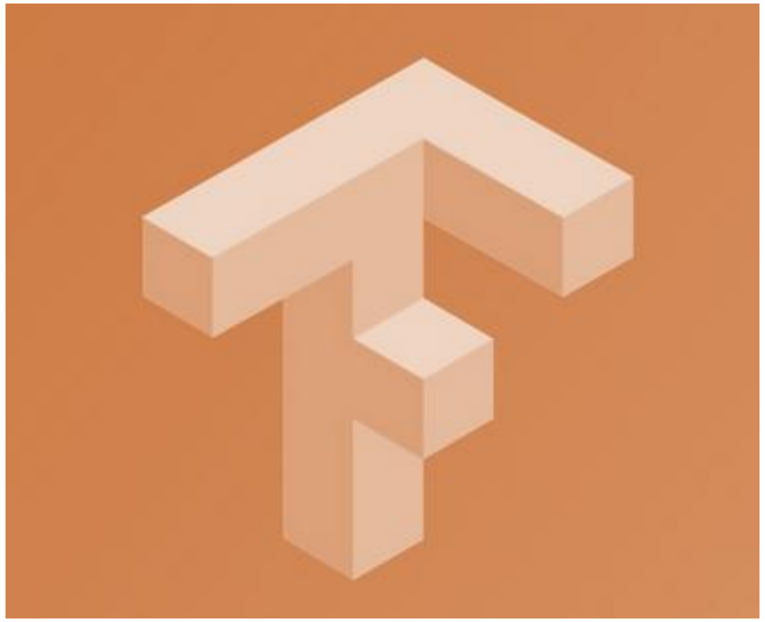
```
t.value -= learning_rate * partial
```

This is done for all trainables.

$$\boxed{x' = x - \eta \nabla C} \qquad \text{Equation (12)}$$

With that, the loss decreases on the next pass through the network. I'm putting the same quiz below again. If you haven't already, set the number of epochs to something like 1000 and watch as the loss decreases!

Code: 18nn_py.txt   Code: 18miniflow_py.txt


You just built a mini version of TensorFlow!

MiniFlow has the makings of becoming a powerful deep learning tool. It is entirely possible to classify something like the MNIST database with MiniFlow.

I'll leave it as an exercise for you to finish MiniFlow from here.

In the next lessons, you'll work with TensorFlow and then Keras, a high level framework on top of TensorFlow. Now that you've built a neural network from scratch, you should be in great shape!

Introduction to TensorFlow

20210330

Open Source Software : GitHub Project Statistics

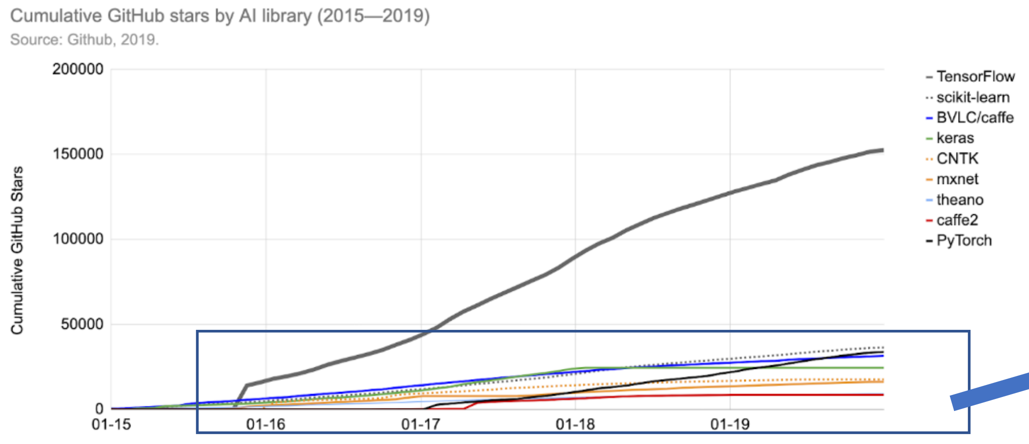


Fig. 1.14a.

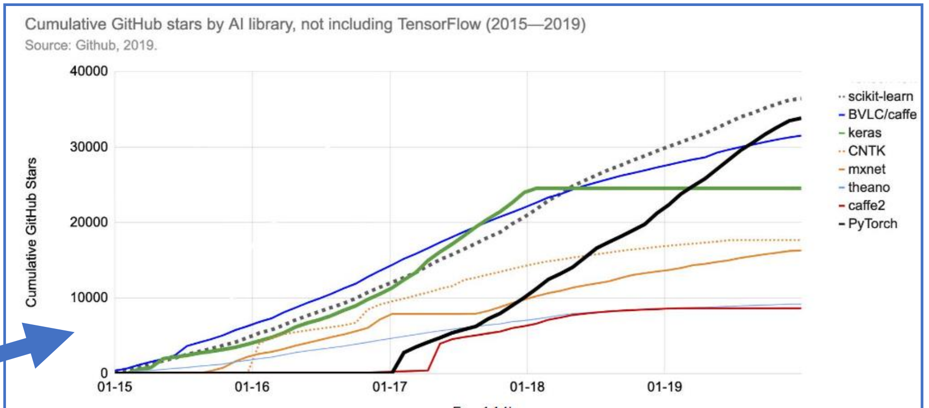
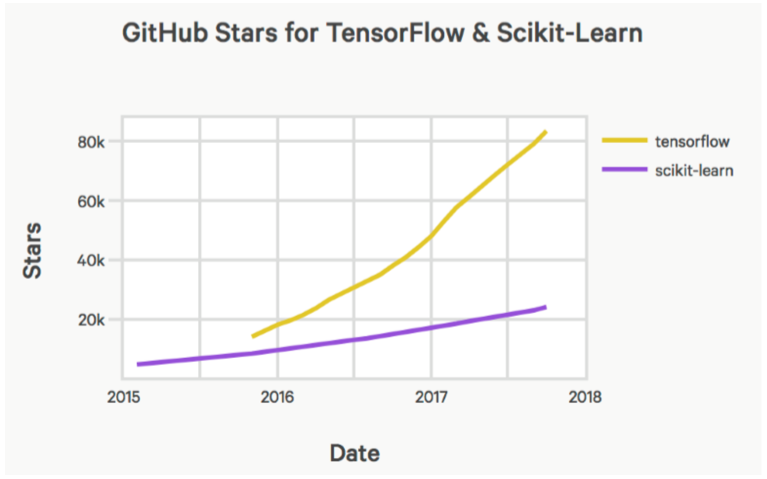
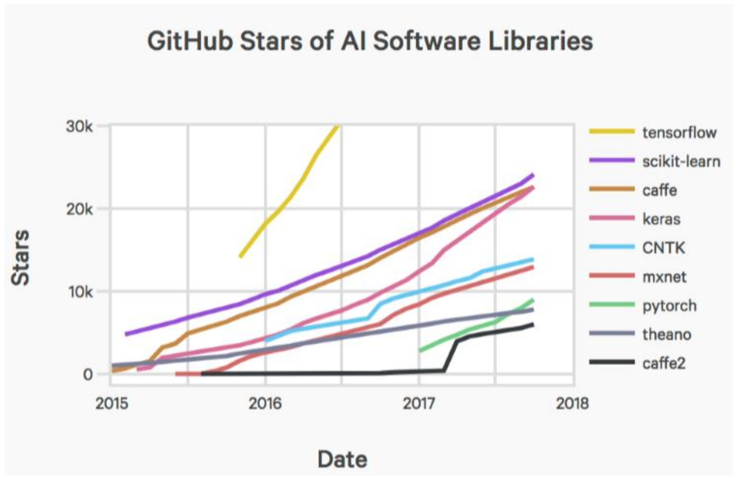


Fig. 1.14b.

https://hai.stanford.edu/sites/g/files/sbiybj10986/f/ai_index_2019_report.pdf



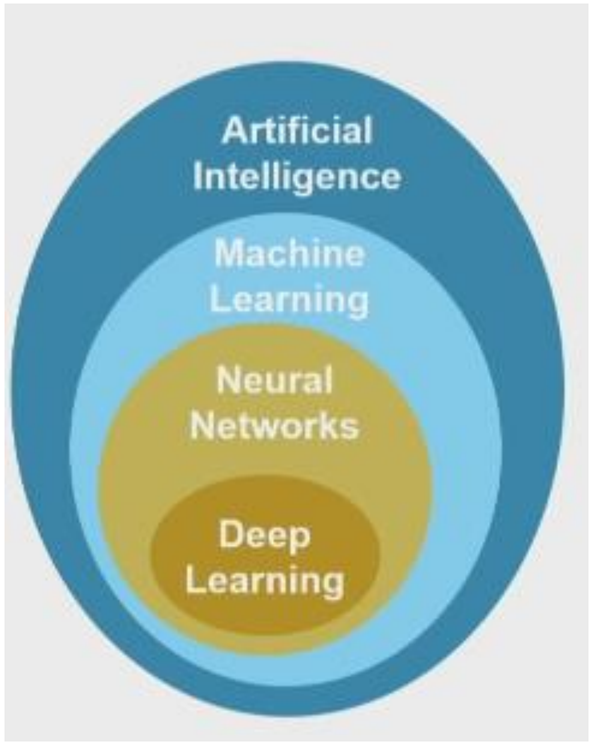


Reference: http://cdn.aiindex.org/2017-report.pdf http://cdn.aiindex.org/2018/AI_Index_2018_Chinese.pdf

Introduction to Deep Neural Networks

- Machine Learning to Deep Learning
- The architecture of multi-layer deep neural network
- Convolutional neural network (Important for self-driving car)
- Using the Tensorflow deep learning library to build CNN

What is Deep Learning?



PERCEPTION:

RECOGNIZING IMAGES

WHAT PEOPLE ARE SAYING

HELPING ROBOTS INTERACT WITH THE WORLD

DEEP LEARNING

- DISCOVERING NEW MEDICINES
- UNDERSTANDING NATURAL LANGUAGE
- UNDERSTANDING DOCUMENTS

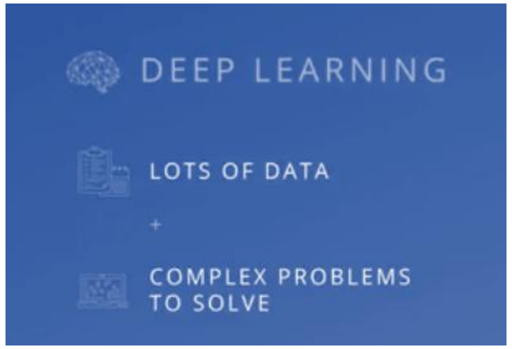
COMPUTER VISION

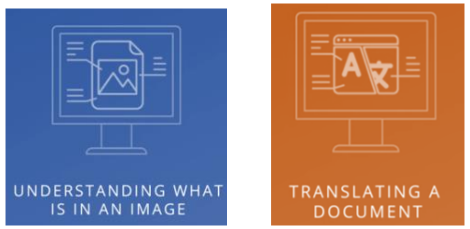
SPEECH RECOGNITION

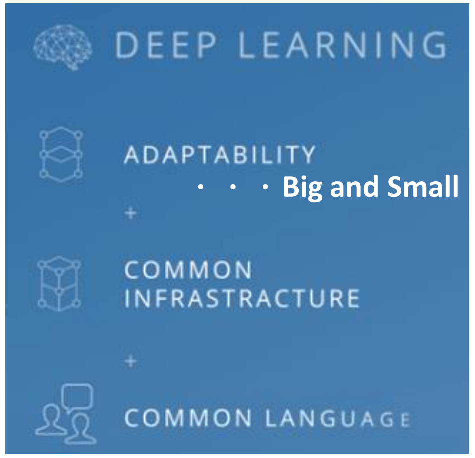
DP has emerged as a central tool to solve perception problems in recent years.

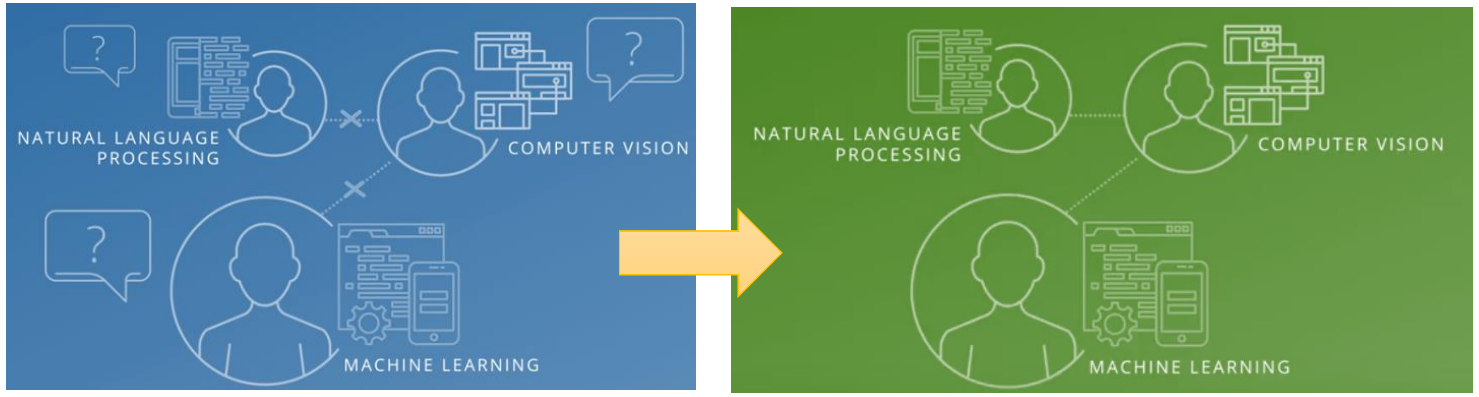
Solving Problems - Big and Small

Using DP in their products and pushing the research forward.









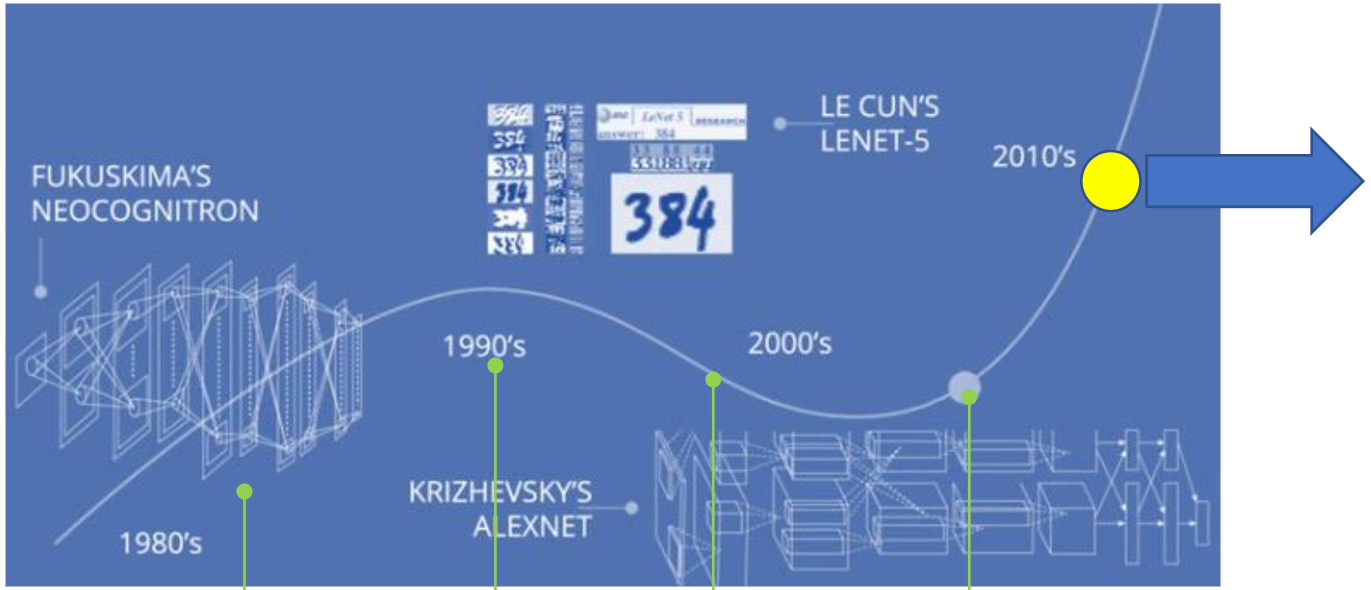
When to Use Deep Learning?

Deep learning is a promising approach when…

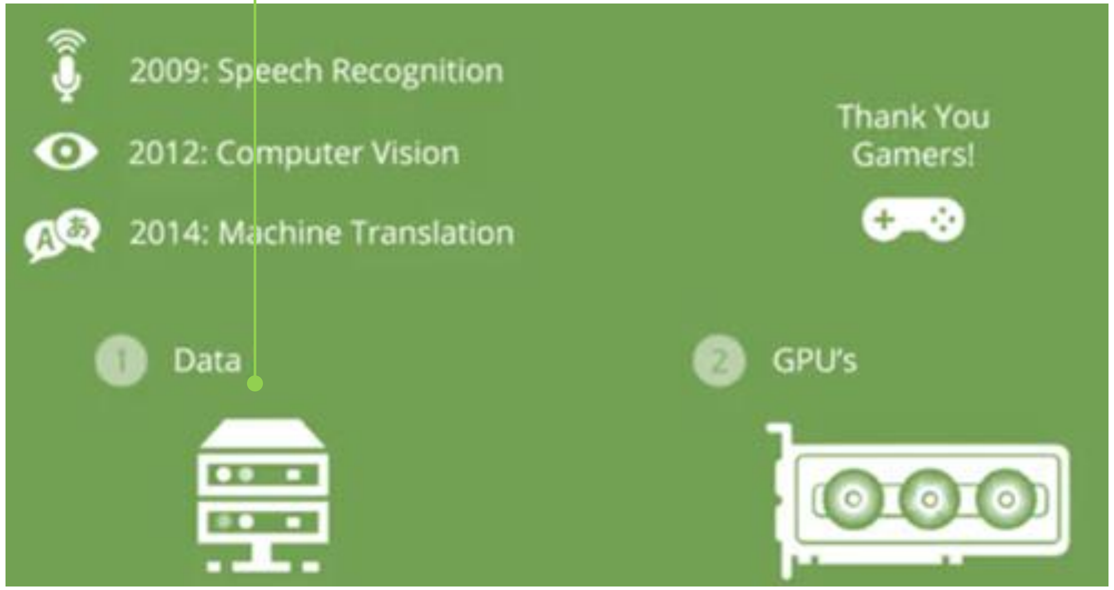
- You have a large amount of training data
- Your are solving an image/audio/natural language problem
- The raw input data has little structure and you need the model to learn meaningful representations(e.g., pixels in an image)

Let's Get Started!

NEURAL NETWORK → DEEP LEARNING



2012 NY Time – The Age of Big Data



A lot of important work on NN happen in the 80s.

Computers were slow and data sets very tiny

The research didn't really find many application in the real world.

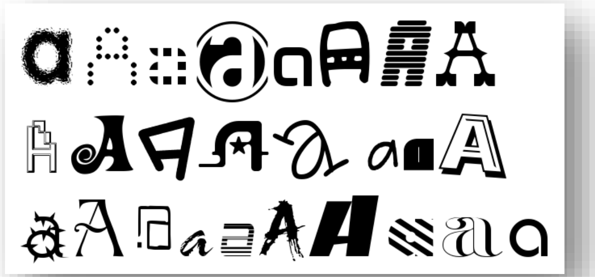
In the first decade of the 21st century, NN have completely disappeared from the world of machine learning.

The NN made a big comeback.

Installing TensorFlow

- Open-source library for deep learning
- Define model structures, library takes care of efficient execution
- Define once, run anywhere: can run on CPUs and GPUs, many devices,
- Can be used in Python and many other languages
- Built for large-scale machine learning development and operations
- Development led by Google

You'll use TensorFlow to classify images from the MNIST dataset - a dataset of images of English letters from A to J. You can see a few example images below.



Your goal is to automatically detect the letter based on the image in the dataset. You'll be working on your own computer for this lab, so, first things first, install TensorFlow!

Install (OS X, Linux, Windows)

Prerequisites

Intro to TensorFlow requires Python 3.4 or higher and Anaconda. If you don't meet all of these requirements, please install the appropriate package(s).

Install TensorFlow

You're going to use an Anaconda environment for this class. If you're unfamiliar with Anaconda environments, check out the official documentation. More information, tips, and troubleshooting for installing tensorflow on Windows can be found here.

Run the following commands to setup your environment:

```
conda create --name=IntroToTensorFlow python=3.5
source activate IntroToTensorFlow
conda install -c conda-forge tensorflow
```

Docker on Windows

Docker instructions were offered prior to the availability of a stable Windows installation via pip or Anaconda. Please try Anaconda first, Docker instructions have been retained as an alternative to an installation via Anaconda.

Install Docker

Download and install Docker from the official Docker website.

Run the Docker Container

Run the command below to start a jupyter notebook server with TensorFlow:

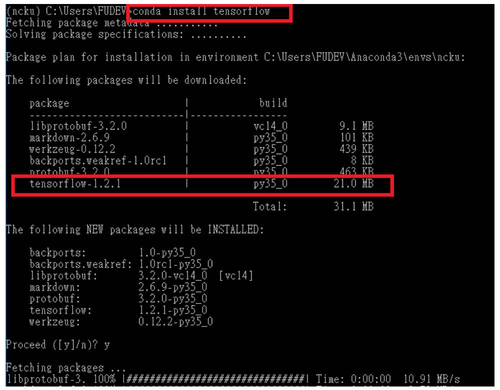
```
docker run -it -p 8888:8888 gcr.io/tensorflow/tensorflow
```

Users in China should use the b.gcr.io/tensorflow/tensorflow instead of gcr.io/tensorflow/tensorflow

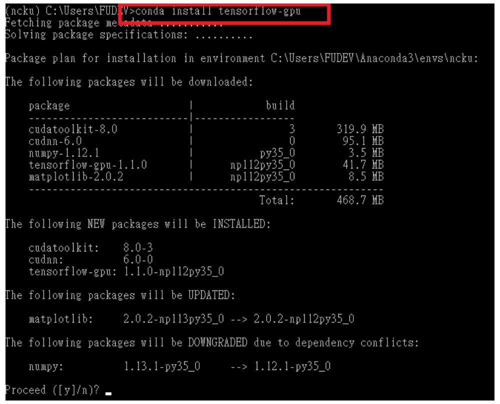
You can access the jupyter notebook at localhost:8888. The server includes 3 examples of TensorFlow notebooks, but you can create a new notebook to test all your code.

Install TensorFlow on Windows

https://www.tensorflow.org/install/install_windows



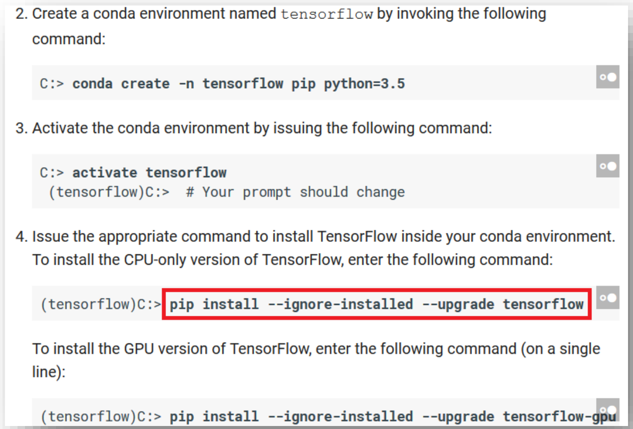
```
(ncku) C:\Users\FUDEV>conda install tensorflow
Fetching package metadata ...........
Solving package specifications: ..........

Package plan for installation in environment C:\Users\FUDEV\Anaconda3\envs\ncku:

The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    libprotobuf-3.2.0          |           vc14_0         9.1 MB
    markdown-2.6.9             |           py35_0         101 KB
    werkzeug-0.12.2            |           py35_0         439 KB
    backports.weakref-1.0rc1   |           py35_0           8 KB
    protobuf-3.2.0             |           py35_0         463 KB
    tensorflow-1.2.1           |           py35_0        21.0 MB

                                           Total:        31.1 MB

The following NEW packages will be INSTALLED:

    backports:         1.0-py35_0
    backports.weakref: 1.0rc1-py35_0
    libprotobuf:       3.2.0-vc14_0  [vc14]
    markdown:          2.6.9-py35_0
    protobuf:          3.2.0-py35_0
    tensorflow:        1.2.1-py35_0
    werkzeug:          0.12.2-py35_0

Proceed ([y]/n)? y

Fetching packages ...
libprotobuf-3. 100% |###############################| Time: 0:00:00  10.91 MB/s
```



```
(ncku) C:\Users\FUDEV>conda install tensorflow-gpu
Fetching package metadata ...........
Solving package specifications: ..........

Package plan for installation in environment C:\Users\FUDEV\Anaconda3\envs\ncku:

The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    cudatoolkit-8.0            |                3       319.9 MB
    cudnn-6.0                  |                0        95.1 MB
    numpy-1.12.1               |           py35_0         3.5 MB
    tensorflow-gpu-1.1.0       |      np112py35_0        41.7 MB
    matplotlib-2.0.2           |      np112py35_0         8.5 MB
    ------------------------------------------------------------
                                           Total:       468.7 MB

The following NEW packages will be INSTALLED:

    cudatoolkit:    8.0-3
    cudnn:          6.0-0
    tensorflow-gpu: 1.1.0-np112py35_0

The following packages will be UPDATED:

    matplotlib:     2.0.2-np113py35_0 --> 2.0.2-np112py35_0

The following packages will be DOWNGRADED due to dependency conflicts:

    numpy:          1.13.1-py35_0     --> 1.12.1-py35_0

Proceed ([y]/n)?
```

Test:

- "native" pip
- Anaconda



2. Create a conda environment named `tensorflow` by invoking the following command:

```
C:> conda create -n tensorflow pip python=3.5
```

3. Activate the conda environment by issuing the following command:

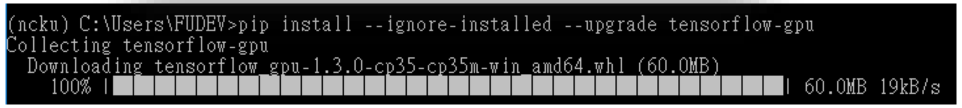
```
C:> activate tensorflow
 (tensorflow)C:>  # Your prompt should change
```

4. Issue the appropriate command to install TensorFlow inside your conda environment. To install the CPU-only version of TensorFlow, enter the following command:

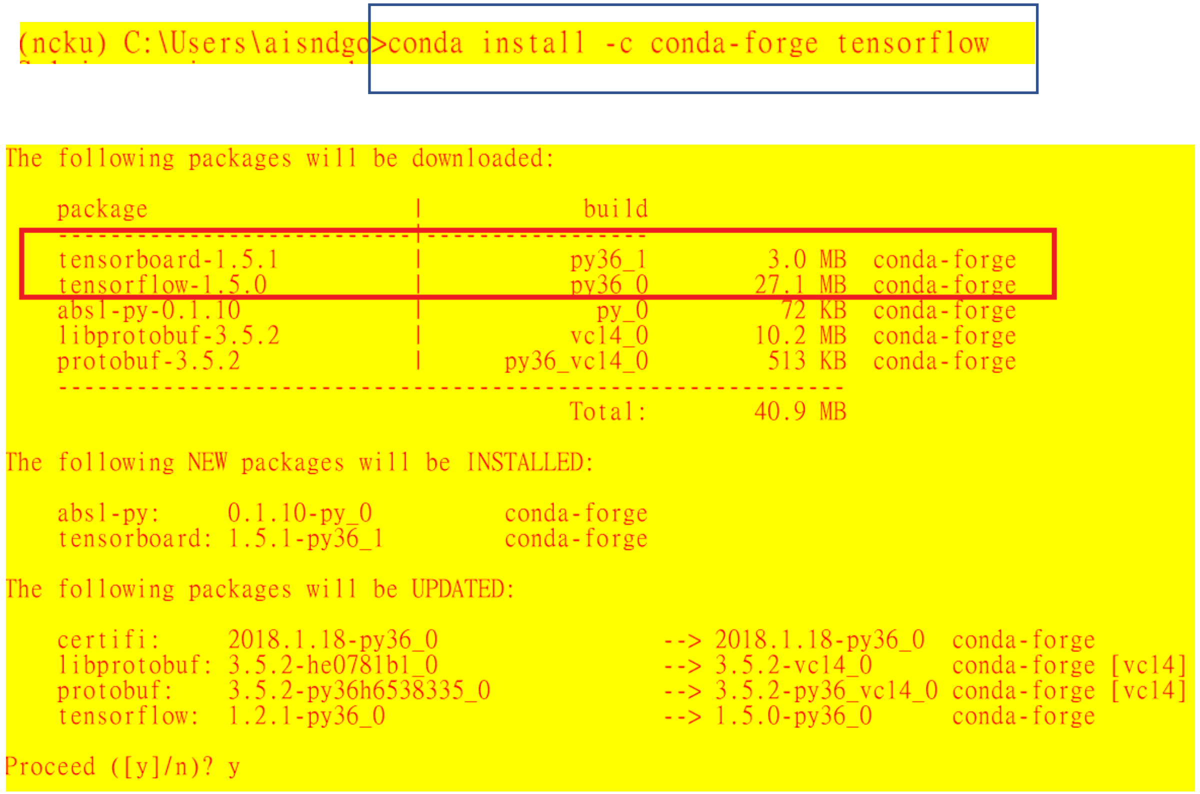
```
(tensorflow)C:> pip install --ignore-installed --upgrade tensorflow
```

To install the GPU version of TensorFlow, enter the following command (on a single line):

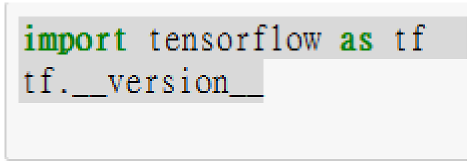
```
(tensorflow)C:> pip install --ignore-installed --upgrade tensorflow-gpu
```



```
(ncku) C:\Users\FUDEV>pip install --ignore-installed --upgrade tensorflow-gpu
Collecting tensorflow-gpu
  Downloading tensorflow_gpu-1.3.0-cp35-cp35m-win_amd64.whl (60.0MB)
    100% |████████████████████████████████| 60.0MB 19kB/s
```

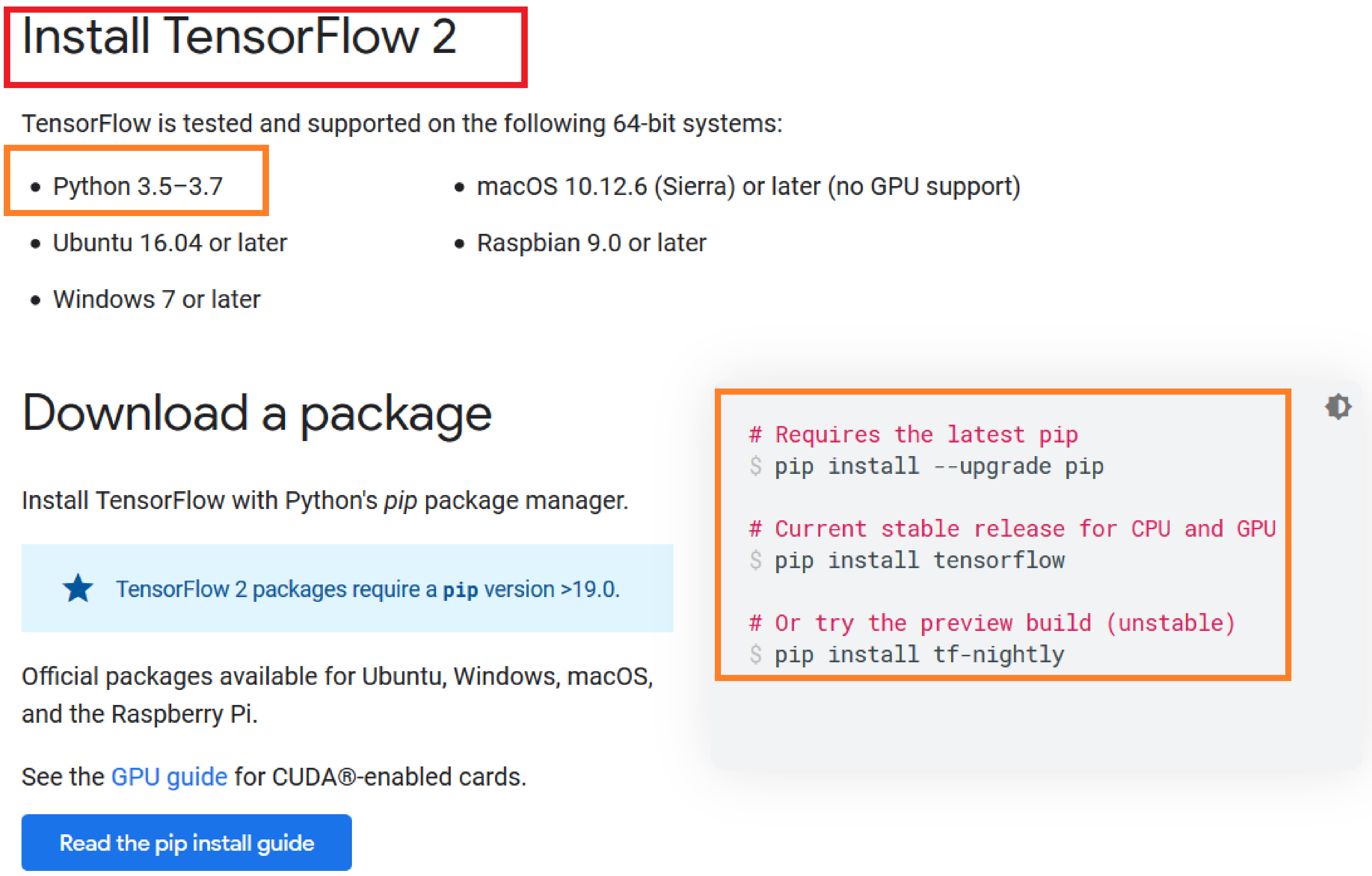


```
(ncku) C:\Users\aisndgo>conda install -c conda-forge tensorflow

The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    tensorboard-1.5.1          |           py36_1         3.0 MB  conda-forge
    tensorflow-1.5.0           |           py36_0        27.1 MB  conda-forge
    absl-py-0.1.10             |             py_0          72 KB  conda-forge
    libprotobuf-3.5.2          |           vc14_0        10.2 MB  conda-forge
    protobuf-3.5.2             |      py36_vc14_0         513 KB  conda-forge
    ------------------------------------------------------------
                                           Total:        40.9 MB

The following NEW packages will be INSTALLED:

    absl-py:     0.1.10-py_0          conda-forge
    tensorboard: 1.5.1-py36_1         conda-forge

The following packages will be UPDATED:

    certifi:     2018.1.18-py36_0                  --> 2018.1.18-py36_0   conda-forge
    libprotobuf: 3.5.2-he0781b1_0                  --> 3.5.2-vc14_0       conda-forge [vc14]
    protobuf:    3.5.2-py36h6538335_0              --> 3.5.2-py36_vc14_0  conda-forge [vc14]
    tensorflow:  1.2.1-py36_0                      --> 1.5.0-py36_0       conda-forge

Proceed ([y]/n)? y
```



```python
import tensorflow as tf
tf.__version__
```

```
'1.5.0'
```



Install TensorFlow 2

TensorFlow is tested and supported on the following 64-bit systems:

- Python 3.5–3.7
- Ubuntu 16.04 or later
- Windows 7 or later
- macOS 10.12.6 (Sierra) or later (no GPU support)
- Raspbian 9.0 or later

Download a package

Install TensorFlow with Python's *pip* package manager.

★ TensorFlow 2 packages require a **pip** version >19.0.

Official packages available for Ubuntu, Windows, macOS, and the Raspberry Pi.

See the GPU guide for CUDA®-enabled cards.

Read the pip install guide

```
# Requires the latest pip
$ pip install --upgrade pip

# Current stable release for CPU and GPU
$ pip install tensorflow

# Or try the preview build (unstable)
$ pip install tf-nightly
```

Installing TensorFlow

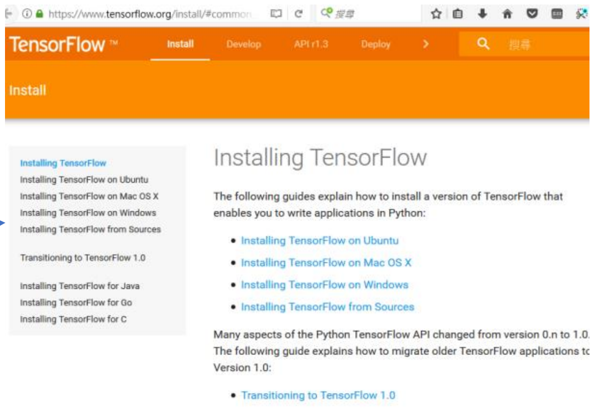
Hello, world!

Try running the following code in your Python console to make sure you have TensorFlow properly installed. The console will print "Hello, world!" if TensorFlow is installed. Don’t worry about understanding what it does. You’ll learn about it in the next section.

```
import tensorflow as tf

# Create TensorFlow object called tensor
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)
```

Errors

If you ‘re getting the error tensorflow.python.framework.errors.InvalidArgumentError: Placeholder:0 is both fed and fetched, you’ re running an older version of TensorFlow. Uninstall TensorFlow, and reinstall it using the instructions above. For more solutions, check out the Common Problems section. (←修正)



Installing TensorFlow
Installing TensorFlow on Ubuntu
Installing TensorFlow on Mac OS X
Installing TensorFlow on Windows
Installing TensorFlow from Sources

Transitioning to TensorFlow 1.0

Installing TensorFlow for Java
Installing TensorFlow for Go
Installing TensorFlow for C

Installing TensorFlow

The following guides explain how to install a version of TensorFlow that enables you to write applications in Python:

- Installing TensorFlow on Ubuntu
- Installing TensorFlow on Mac OS X
- Installing TensorFlow on Windows
- Installing TensorFlow from Sources

Many aspects of the Python TensorFlow API changed from version 0.n to 1.0. The following guide explains how to migrate older TensorFlow applications tc Version 1.0:

- Transitioning to TensorFlow 1.0

Hello, Tensor World!

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!’ )
# method 1
with tf.Session() as sess:
    # Run the tf.constant operation in the session
    output = sess.run(hello_constant)
    print(output)
#method 2
Sess2=tf.Session()
output2 = Sess2.run(hello_constant)
print(output2)
Sess2.close()  #需要關閉,method 1執行完畢會關閉Session
```

Tensor

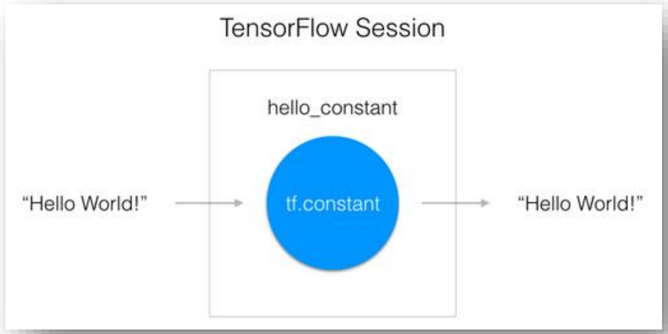
In TensorFlow, data isn’ t stored as integers, floats, or strings. These values are encapsulated in an object called a tensor. In the case of hello_constant = tf.constant('Hello World!'), hello_constant is a 0-dimensional string tensor, but tensors come in a variety of sizes as shown below:

```
# 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] ])
```

tf.constant() is one of many TensorFlow operations you will use in this lesson. The tensor returned by tf.constant() is called a constant tensor, because the value of the tensor never changes.

Session

TensorFlow’s api is built around the idea of a computational graph, a way of visualizing a mathematical process which you learned about in the MiniFlow lesson. Let’s take the TensorFlow code you ran and turn that into a graph:



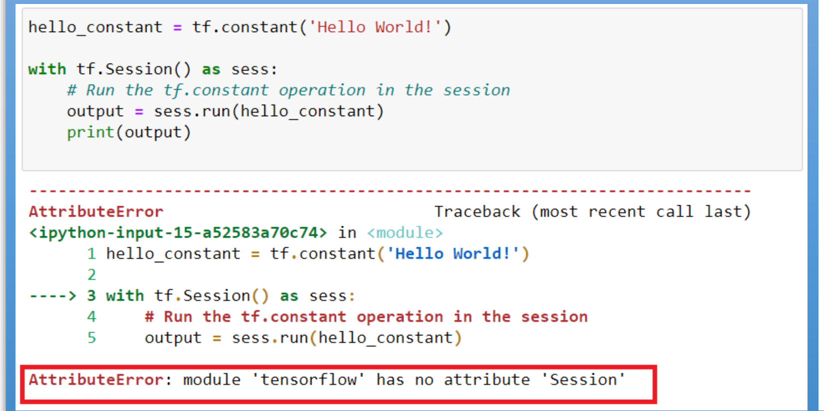
A "TensorFlow Session", as shown left, is an environment for running a graph. The session is in charge of allocating the operations to GPU(s) and/or CPU(s), including remote machines. Let’s see how you use it.

```
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. After you run the above, you will see the following printed out:

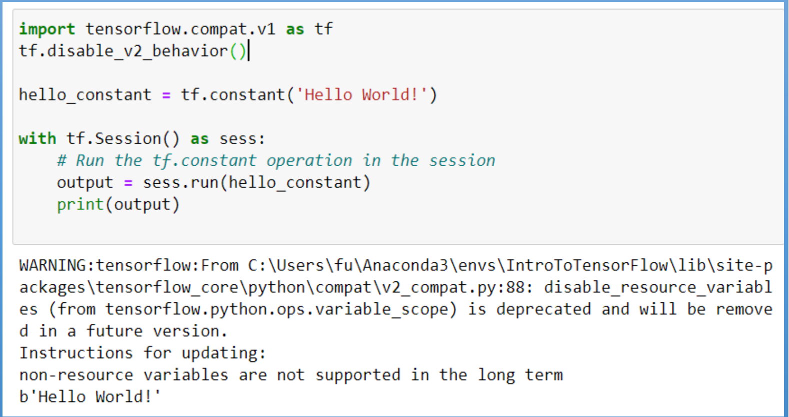
```
'Hello World!'
```

TF1.x→TF2.x



```
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)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-a52583a70c74> in <module>
      1 hello_constant = tf.constant('Hello World!')
      2 
----> 3 with tf.Session() as sess:
      4     # Run the tf.constant operation in the session
      5     output = sess.run(hello_constant)

AttributeError: module 'tensorflow' has no attribute 'Session'
```



```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

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)
```

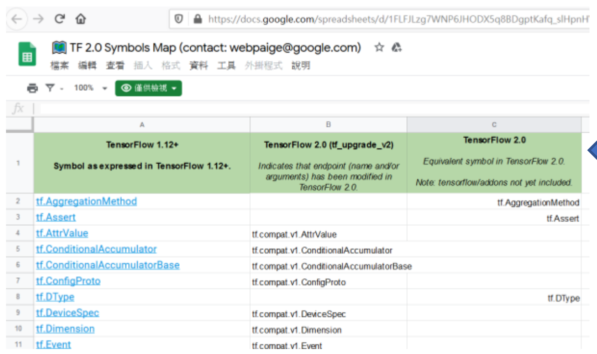
```
WARNING:tensorflow:From C:\Users\fu\Anaconda3\envs\IntroToTensorFlow\lib\site-p
ackages\tensorflow_core\python\compat\v2_compat.py:88: disable_resource_variabl
es (from tensorflow.python.ops.variable_scope) is deprecated and will be remove
d in a future version.
Instructions for updating:
non-resource variables are not supported in the long term
b'Hello World!'
```

TF1.x hello world:

```
import tensorflow as tf
msg = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(msg))
```

TF2.x hello world:

```
import tensorflow as tf
msg = tf.constant('Hello, TensorFlow!')
tf.print(msg)
```



https://docs.google.com/spreadsheets/d/1FLFJLzg7WNP6JHODX5q8BDgptKafq_slHpnH

TF 2.0 Symbols Map (contact: webpaige@google.com)

檔案 編輯 查看 插入 格式 資料 工具 外掛程式 說明

100% 僅供檢視

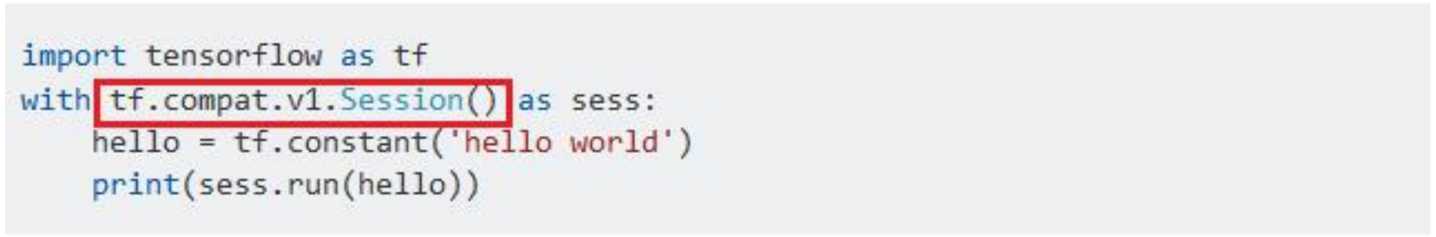
	A	B	C
1	TensorFlow 1.12+ Symbol as expressed in TensorFlow 1.12+.	TensorFlow 2.0 (tf_upgrade_v2) *Indicates that endpoint (name and/or arguments) has been modified in TensorFlow 2.0.*	TensorFlow 2.0 *Equivalent symbol in TensorFlow 2.0.* *Note: tensorflow/addons not yet included.*
2	tf.AggregationMethod		tf.AggregationMethod
3	tf.Assert		tf.Assert
4	tf.AttrValue	tf.compat.v1.AttrValue	
5	tf.ConditionalAccumulator	tf.compat.v1.ConditionalAccumulator	
6	tf.ConditionalAccumulatorBase	tf.compat.v1.ConditionalAccumulatorBase	
7	tf.ConfigProto	tf.compat.v1.ConfigProto	
8	tf.DType		tf.DType
9	tf.DeviceSpec	tf.compat.v1.DeviceSpec	
10	tf.Dimension	tf.compat.v1.Dimension	
11	tf.Event	tf.compat.v1.Event	

According to TF 1:1 Symbols Map, in TF 2.0 you should use tf.compat.v1.Session() instead of tf.Session()
https://docs.google.com/spreadsheets/d/1FLFJLzg7WNP6JHODX5q8BDgptKafq_slHpnHVbJlteQ/edit#gid=0
To get TF 1.x like behaviour in TF 2.0 one can run

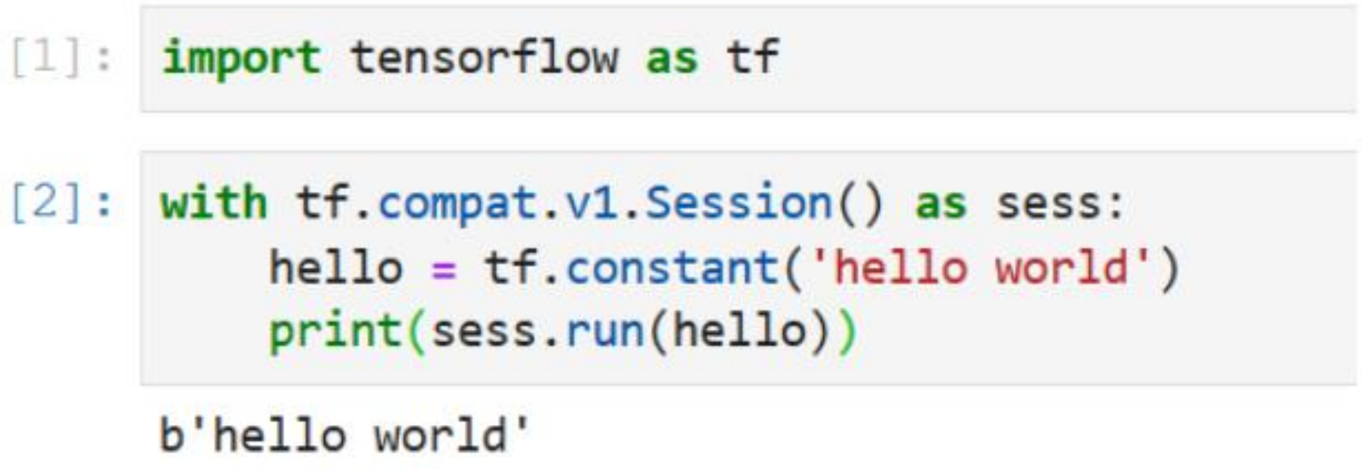
```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

TF2 runs Eager Execution by default, thus removing the need for Sessions.
If you want to run static graphs, the more proper way is to use tf.function() in TF2.
While Session can still be accessed via tf.compat.v1.Session() in TF2, I would discourage using it.

TF1.x→TF2.x



```
import tensorflow as tf
with tf.compat.v1.Session() as sess:
    hello = tf.constant('hello world')
    print(sess.run(hello))
```



```
[1]: import tensorflow as tf
```

```
[2]: with tf.compat.v1.Session() as sess:
         hello = tf.constant('hello world')
         print(sess.run(hello))
```

```
b'hello world'
```

TF2.x→TF1.x

```
pip3 install --upgrade --force-reinstall tensorflow-gpu==1.15.0
```

Effective TensorFlow 2

- There are multiple changes in TensorFlow 2.0 to make TensorFlow users more productive. TensorFlow 2.0 removes redundant APIs, makes APIs more consistent (Unified RNNs, Unified Optimizers), and better integrates with the Python runtime with Eager execution. Many RFCs have explained the changes that have gone into making TensorFlow 2.0.
- A brief summary of major changes
  - API Cleanup
  - Eager execution
  - No more globals
  - Functions, not sessions

https://www.tensorflow.org/guide/effective_tf2

Quiz: TensorFlow Input

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…".

Quiz

Let's see how well you understand tf.placeholder() and feed_dict. The code below throws an error, but I want you to make it return the number 123. Change line 11, so that the code returns the number 123.

Code: 01quiz_py.txt Code: 01solution_py.txt

Quiz: TensorFlow Math

TensorFlow Math

You're going to use basic math functions that everyone knows and loves - add, subtract, multiply, and divide - with tensors. (There's many more math functions you can check out in the documentation.)

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
```

The x tensor will evaluate to 6, because 10 - 4 = 6. The y tensor will evaluate to 10, because 2 * 5 = 10. That was easy!

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 float32 for Tensor with dtype int32:
```

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:

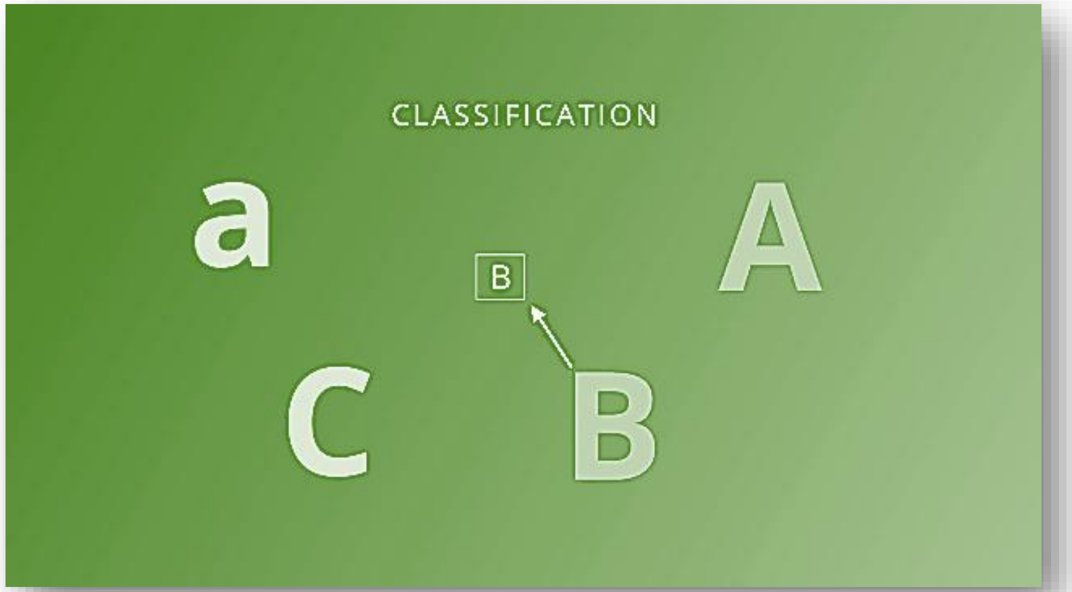
```
tf.subtract(tf.cast(tf.constant(2.0), tf.int32), tf.constant(1))   # 1
```

Quiz

Let's apply what you learned to convert an algorithm to TensorFlow. The code below is a simple algorithm using division and subtraction. Convert the following algorithm in regular Python to TensorFlow and print the results of the session. You can use tf.constant() for the values 10, 2, and 1.

Code: 02quiz_py.txt Code: 02solution_py.txt

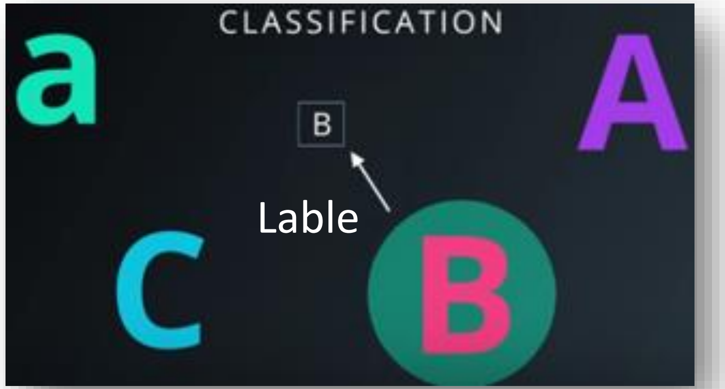
Transition to Classification



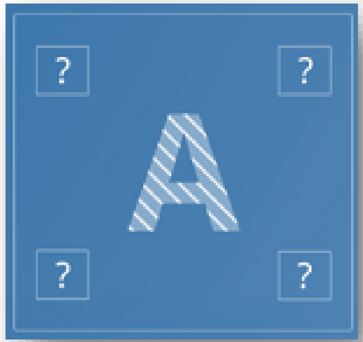
You've accomplished a lot. In particular, you did the following:
•Ran operations in tf.session.
•Created a constant tensorwith tf.constant().
•Used tf.placeholder() and feed_dict to get input.
•Applied the tf.add(), tf.subtract(), tf.multiply(), and tf.divide() functions using numeric data.

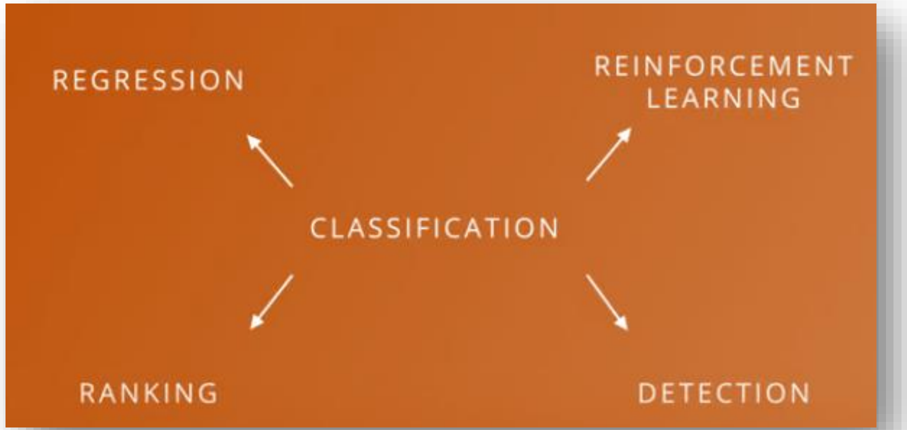
You know the basics of TensorFlow, so let's take a break and get back to the theory of neural networks. In the next sessions, you're going to learn about one of the most popular applications of neural networks - classification.

Supervised Classification

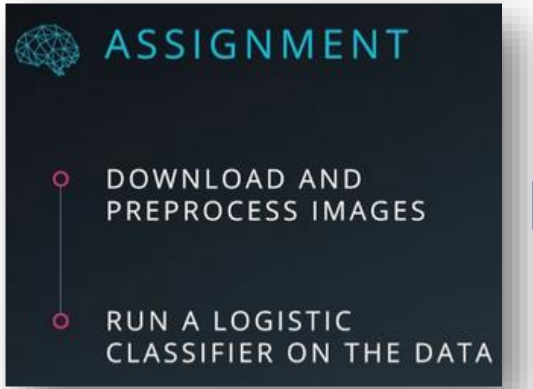


TRAINING SET
a
a
A
A
B
b
B
C
C
D
D
d





The central building block of machine learning



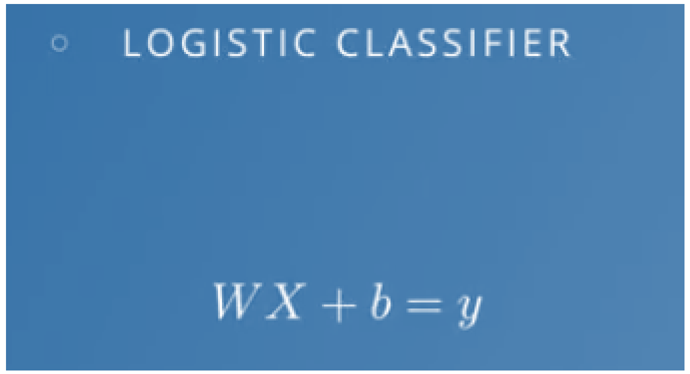
ASSIGNMENT

- DOWNLOAD AND PREPROCESS IMAGES
- RUN A LOGISTIC CLASSIFIER ON THE DATA

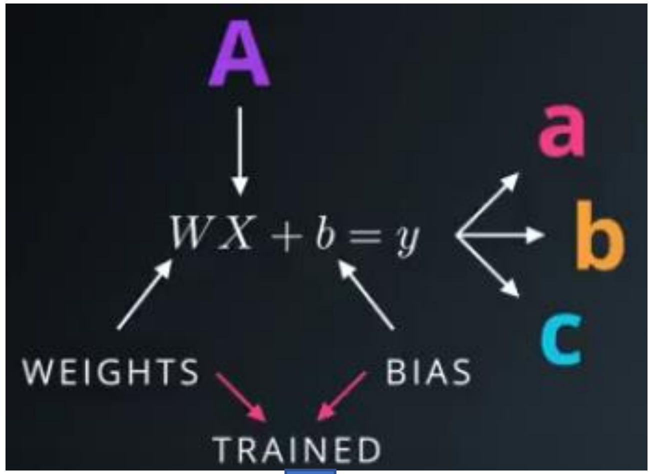
Logistic Classifier

Training Your Logistic Classifier

Logistic Classifier



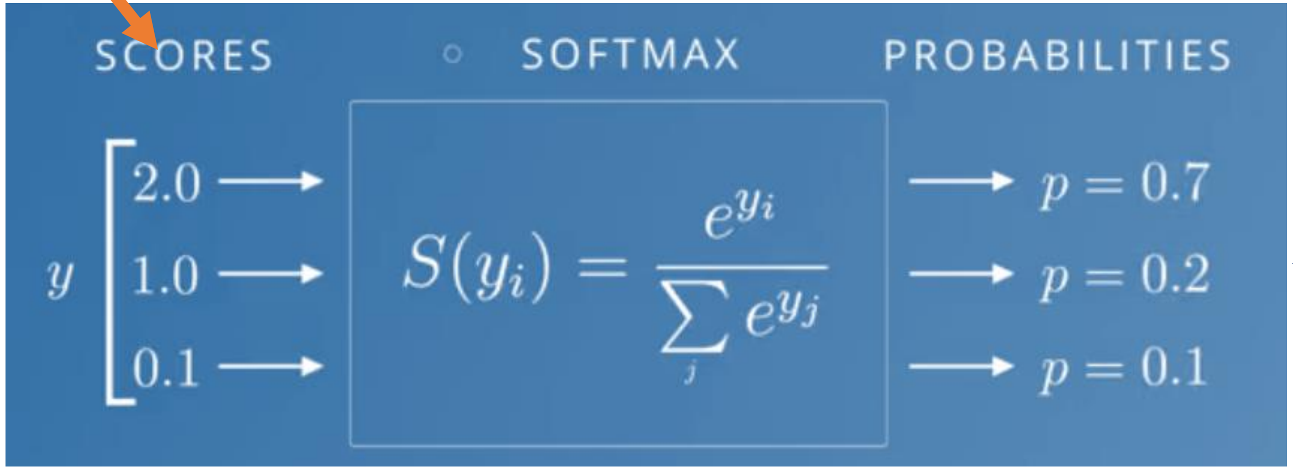
Linear classifier



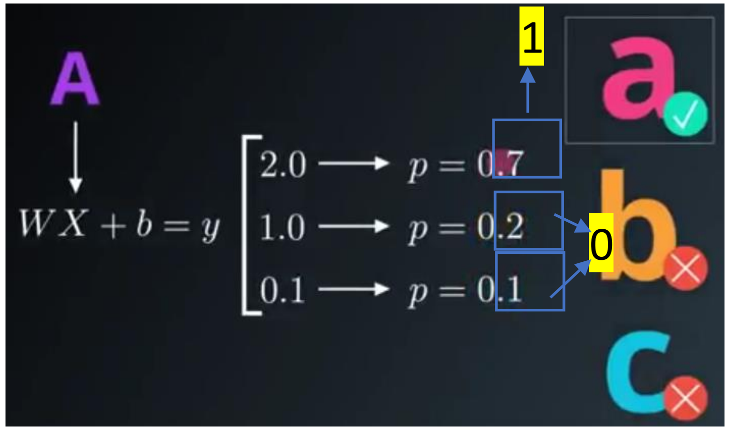
How are we going to use those scores to perform the classification?

Turn scores into the probability of the correct class.

LOGITS



$$y \begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix} \rightarrow \quad S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}} \quad \rightarrow \begin{matrix} p = 0.7 \\ p = 0.2 \\ p = 0.1 \end{matrix}$$



Turn scores into probabilities is to use a SOFTMAX function

Quiz: TensorFlow Linear Function

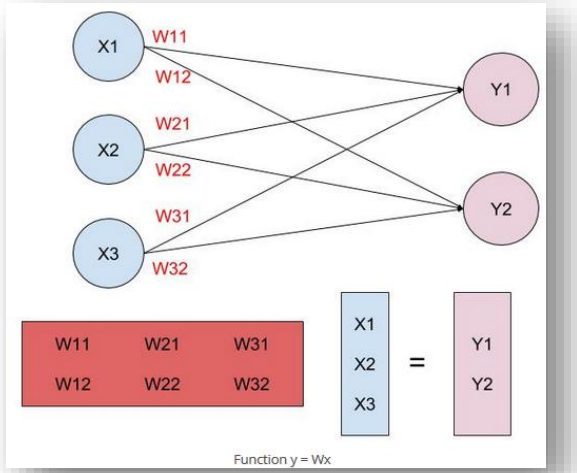
TensorFlow Linear Function

Let's derive the function *y = Wx + b*. We want to translate our input, x, to labels, y. For example, imagine we want to classify images as digits.

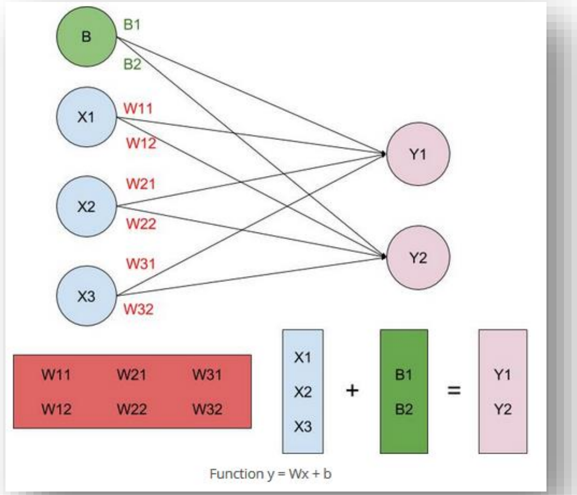
x would be our list of pixel values, and y would be the logits, one for each digit. Let's take a look at y = Wx, where the weights, W, determine the influence of x at predicting each y.

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



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.

Matrix Multiplication Quiz
Calculate the logits a and b for the following formula.

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

y = Wx + b

Question 1 of 2
What is a?
○ 0.86
○ 0.06
○ 0.16
○ 0.36

Question 2 of 2
What is b?
○ 0.86
○ 0.06
○ 0.16
○ 0.36

Quiz: TensorFlow Linear Function

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. You'll notice that the elements in this 1x2 matrix are the same as the elements in the 2x1 matrix from the quiz. Again, these matrices are simply transposed.

$$y = \begin{pmatrix} 0.16 & 0.06 \end{pmatrix}$$

We now have our logits! The columns represent the logits for our two labels.
Now you can learn how to train this function in TensorFlow.

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.

Quiz: TensorFlow Linear Function

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.

Tensorflow Variable

```
import tensorflow as tf

state = tf.Variable(0, name='counter')
one =tf.constant(1)
new_value=tf.add(state,one)
update=tf.assign(state,new_value)
init=tf.global_variables_initializer() #must have if define variable

with tf.Session() as sess:
    sess.run(init)
    for _ in range(3):
        sess.run(update)
        print(sess.run(state))
```

Quiz: TensorFlow Linear Function

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)))
```

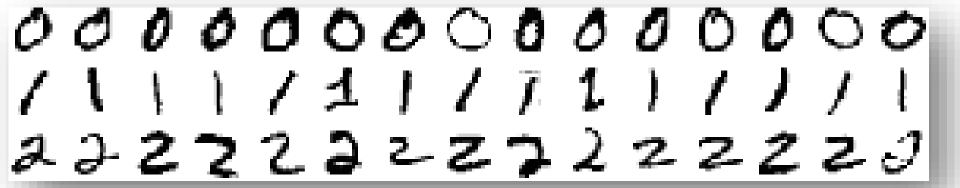
The tf.truncated_normal() function returns a tensor with random values from a normal distribution whose magnitude is no more than 2 standard deviations from the mean.
Since the weights are already helping prevent the model from getting stuck, you don't need to randomize the bias. Let's use the simplest solution, setting the bias to 0.

tf.zeros()

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

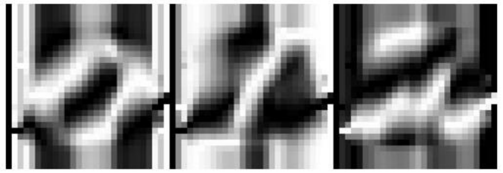
The tf.zeros() function returns a tensor with all zeros.

Linear Classifier Quiz


Subset of MNIST dataset.

Quiz: TensorFlow Linear Function

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 1s 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.

Instructions

(A.) Open quiz.py.

1. Implement get_weights to return a tf.Variable of weights
2. Implement get_biases to return a tf.Variable of biases
3. Implement xW + b in the linear function

(B.) Open sandbox.py

1. Initialize all weights

Since xW in xW + b is matrix multiplication, you have to use the tf.matmul() function instead of tf.multiply(). Don't forget that order matters in matrix multiplication, so tf.matmul(a,b) is not the same as tf.matmul(b,a).

Code: 03quiz_py.txt Code: 03sandbox_py.txt Code: 03quiz_solution_py.txt Code: 03sandbox_solution_py.txt

Quiz: Softmax

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Softmax Function

Softmax

Congratulations on successfully implementing a linear function that outputs logits. You're one step closer to a working classifier.

The next step is to assign a probability to each label, which you can then use to classify the data. Use the softmax function to turn your logits into probabilities.

We can do this by using the formula above, which uses the input of y values and the mathematical constant "e" which is approximately equal to 2.718. By taking "e" to the power of any real value we always get back a positive value, this then helps us scale when having negative y values. The summation symbol on the bottom of the divisor indicates that we add together all the e^(input y value) elements in order to get our calculated probability outputs.

Quiz

For the next quiz, you'll implement a softmax(x) function that takes in x, a one or two dimensional array of logits. In the one dimensional case, the array is just a single set of logits. In the two dimensional case, each column in the array is a set of logits. The softmax(x) function should return a NumPy array of the same shape as x.

For example, given a 1-dimensional array:

```
# logits is a one-dimensional array with 3 elements
logits = [1.0, 2.0, 3.0]
# softmax will return a 1-dimensional array with 3 elements
print softmax(logits)
```

Quiz: Softmax

```
$ [ 0.09003057  0.24472847  0.66524096]
```

Given a two-dimensional array where each column represents a set of logits:

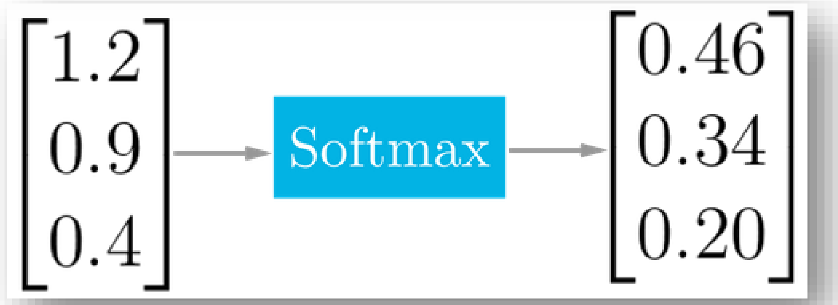
```
# logits is a two-dimensional array
logits = np.array([
    [1, 2, 3, 6],
    [2, 4, 5, 6],
    [3, 8, 7, 6]])
# softmax will return a two-dimensional array with the same shape
print softmax(logits)
```

```
$ [
    [ 0.09003057  0.00242826  0.01587624  0.33333333]
    [ 0.24472847  0.01794253  0.11731043  0.33333333]
    [ 0.66524096  0.97962921  0.86681333  0.33333333]
  ]
```

Implement the softmax function, which is specified by the formula at the top of the page. The probabilities for each column must sum to 1. Feel free to test your function with the inputs above.

Code: 04main_code_py.txt   Code: 04solution.txt

Quiz: TensorFlow Softmax



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

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.

Quiz

Use the softmax function in the quiz below to return the softmax of the logits.

Code: 05quiz_py.txt Code: 05solution.txt

Quiz

Answer the following 2 questions about softmax.

Question 2 of 3

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

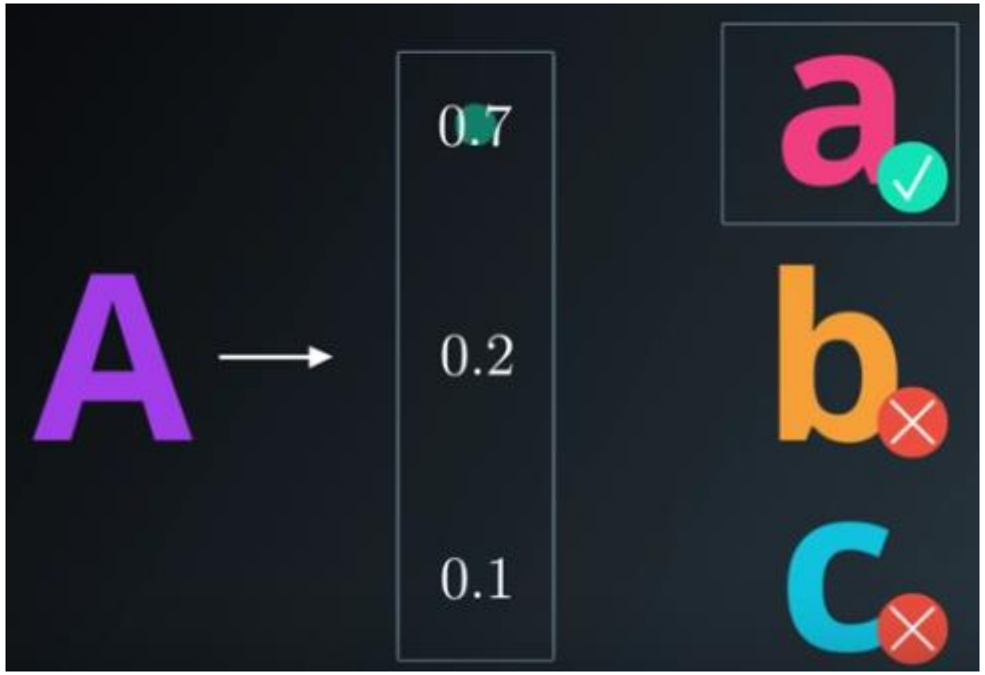
Question 3 of 3

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

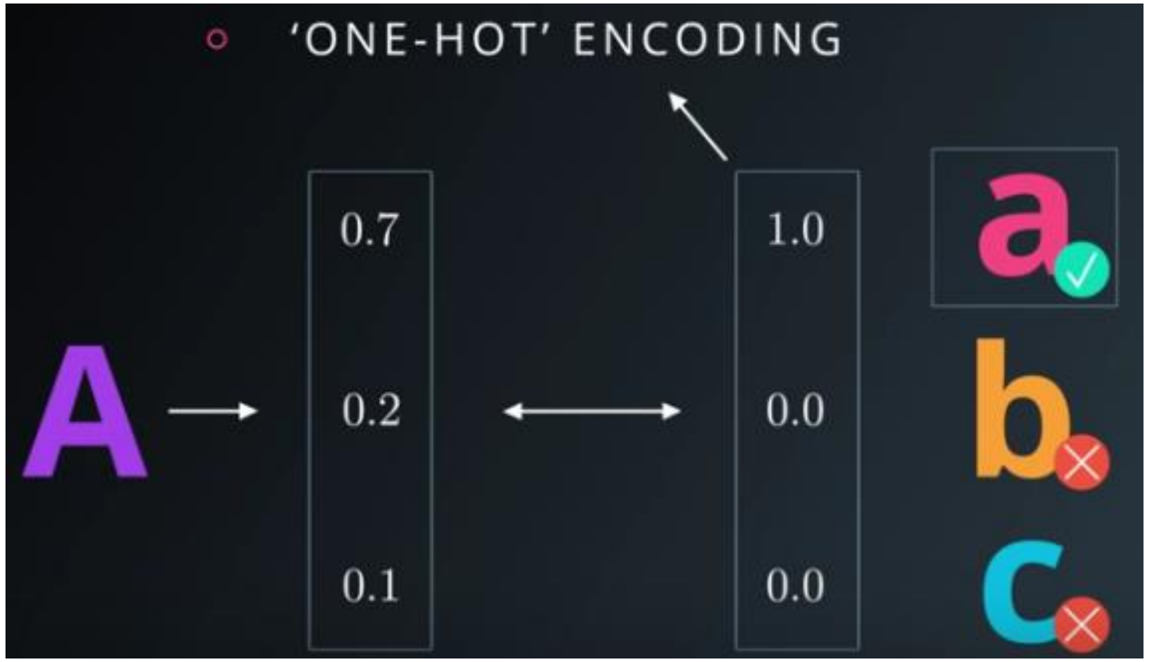
❍ The probabilities get close to 0.0 or 1.0

❍ The probabilities get close to the uniform distribution

One-Hot Encoding

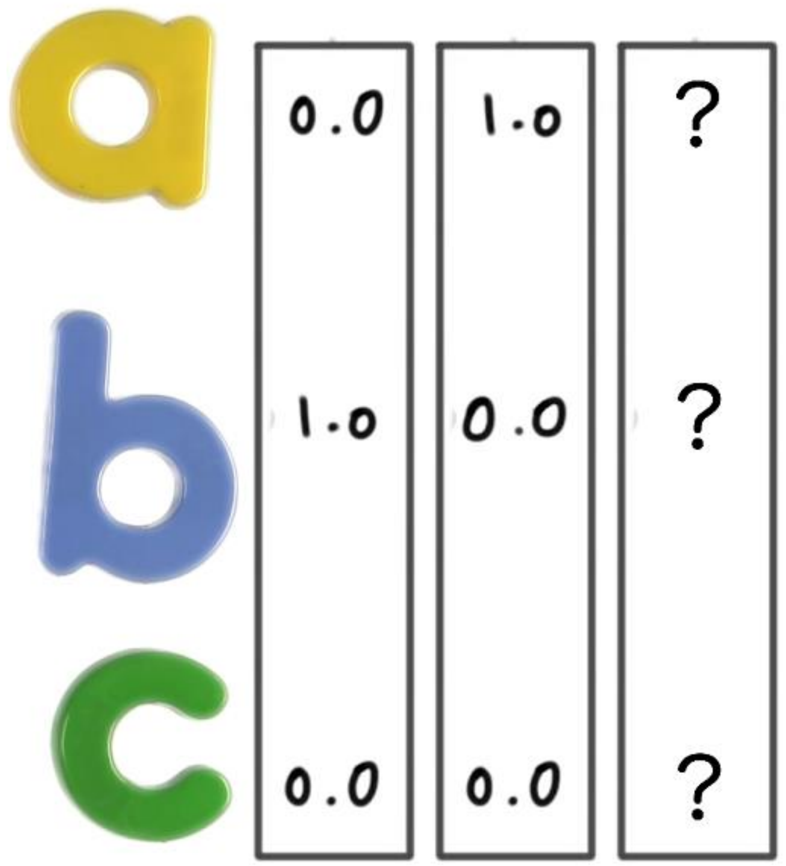


Probabilities for the correct class



[Why One-Hot Encode Data in Machine Learning?]

Quiz: One-Hot Encoding

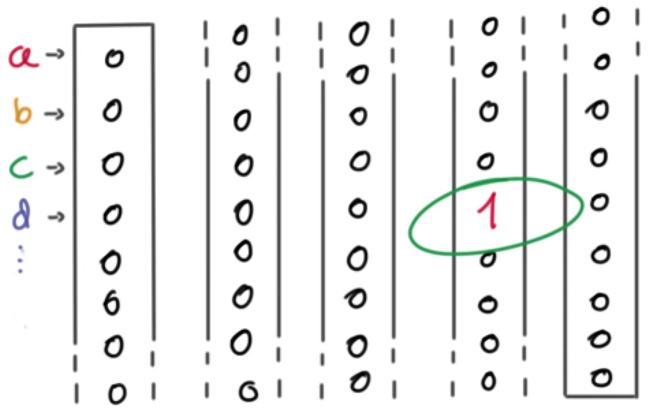


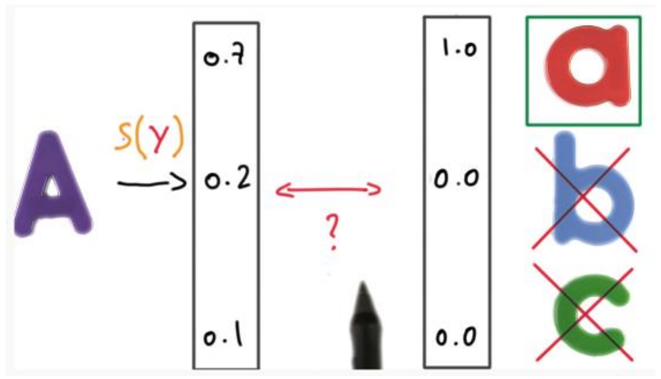
Quiz Question

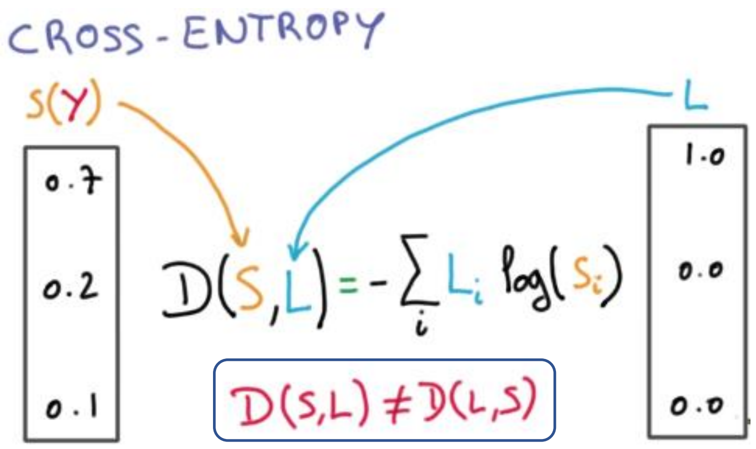
With the labels ["A", "B", "C"] and one-hot encodings from above, select the missing one-hot encoding:

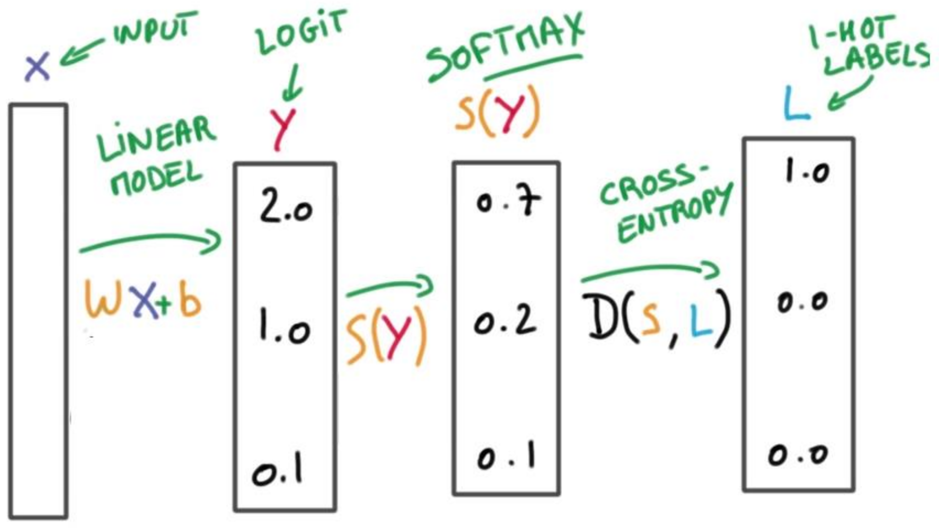
- ❍ [1.0,0.0,1.0]
- ❍ [0.0,0.0,0.0]
- ❍ [0.0,1.0,0.0]
- ❍ [0.0,0.0,1.0]

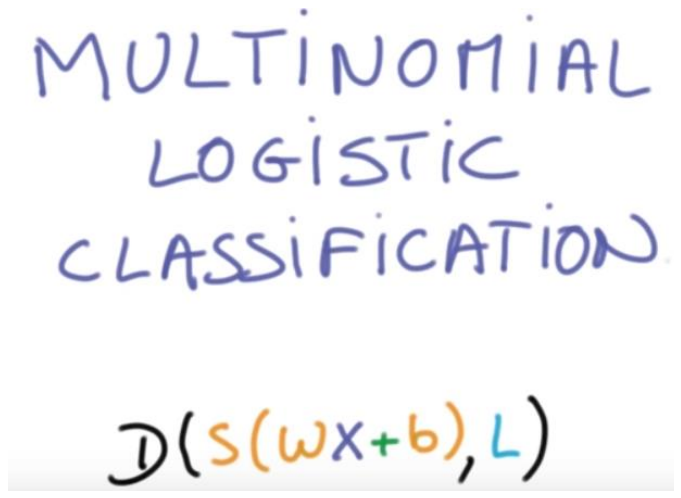
Cross Entropy



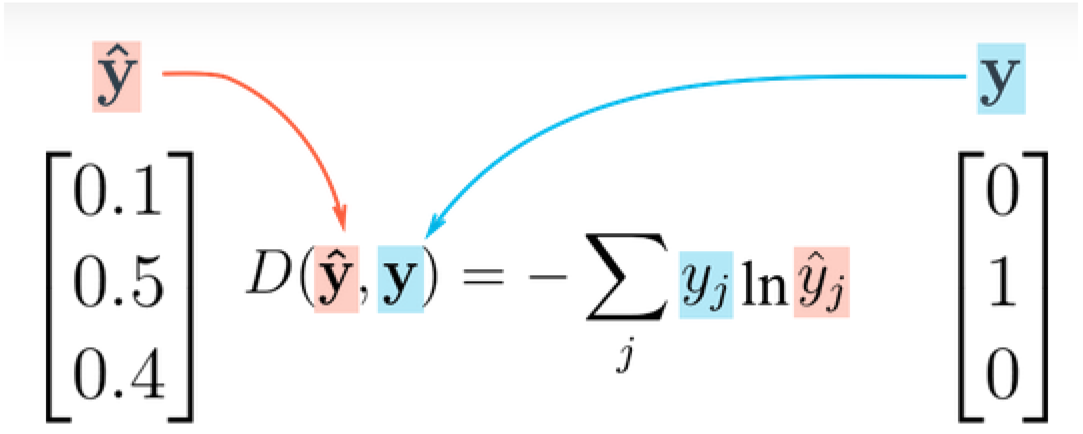








Quiz: TensorFlow Cross Entropy


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

To create a cross entropy function in TensorFlow, you'll need to use two new functions:

- tf.reduce_sum()
- tf.log()

Reduce Sum

```
x = tf.reduce_sum([1, 2, 3, 4, 5])  # 15
```

The tf.reduce_sum() function takes an array of numbers and sums them together.

Natural Log

```
x = tf.log(100)  # 4.60517
```

This function does exactly what you would expect it to do. tf.log() takes the natural log of a number.

Quiz

Print the cross entropy using softmax_data and one_hot_data.

Code: 06quiz_py.txt

Code: 06solution.txt