

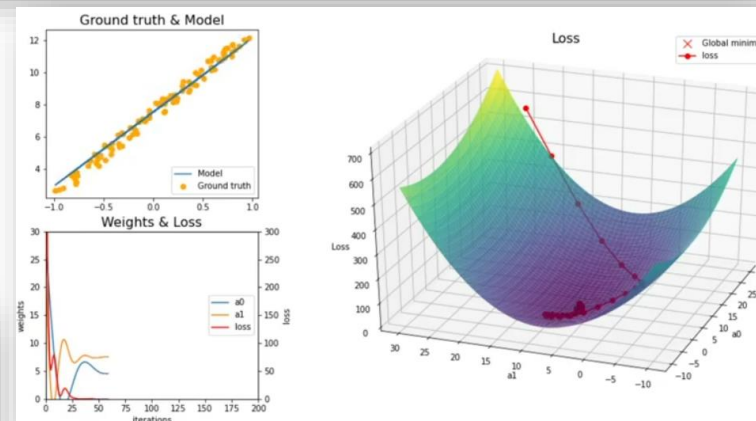
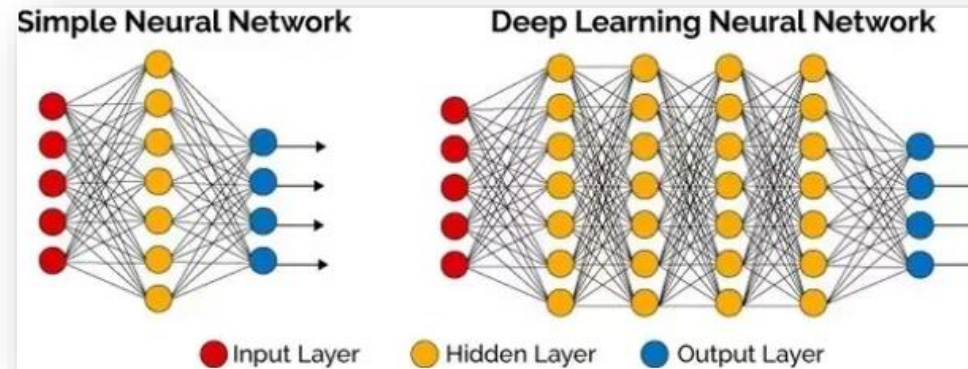
Deep Neural Networks

20210413

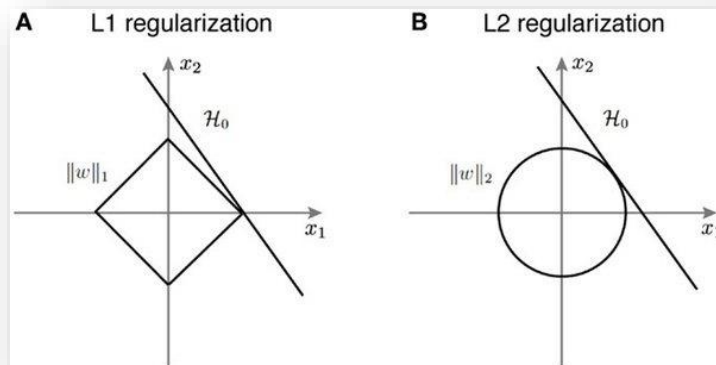
Intro to Deep Neural Networks

NEXT...

TURN YOUR CLASSIFIER
INTO DEEP NETWORK



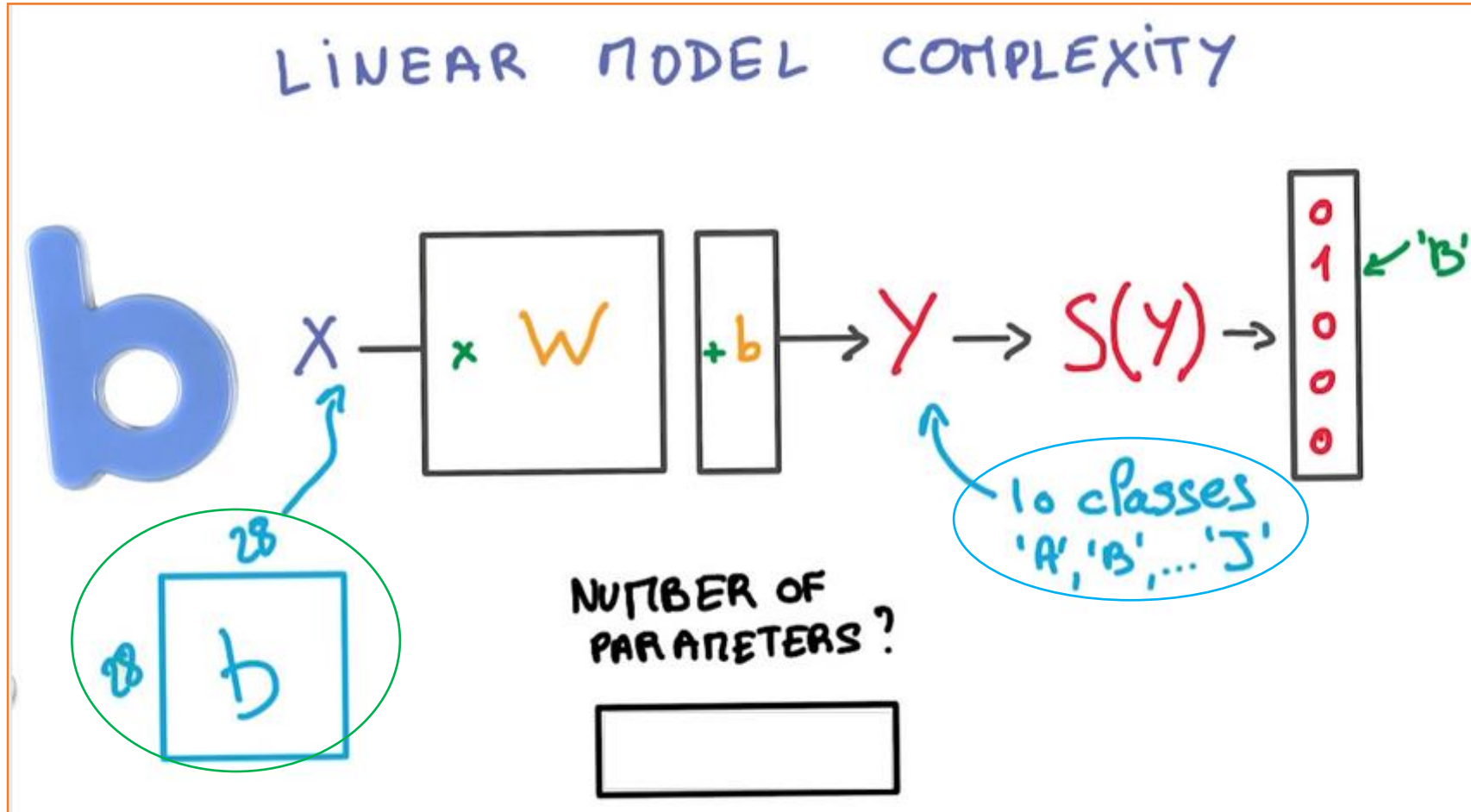
USE THE OPTIMIZER TO
COMPUTE GRADIENTS



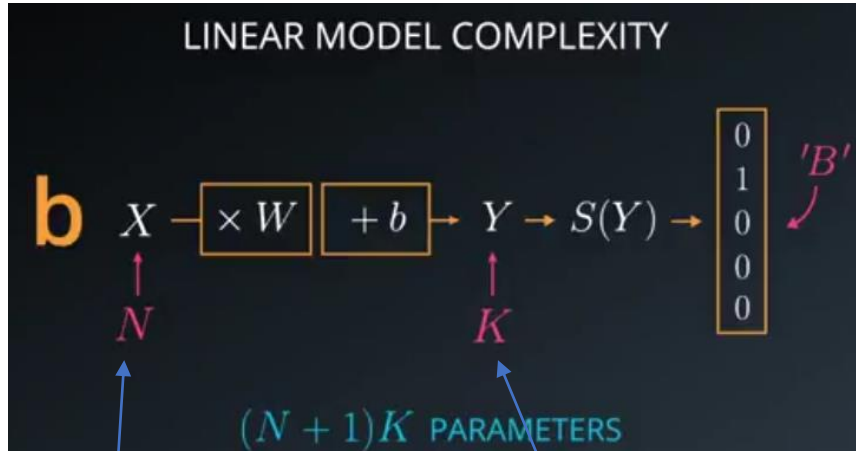
UNDERSTAND
REGULARIZATION

Quiz: Number of Parameters

Quiz: Number of Parameters



Linear Models are Limited



INPUT

OUTPUT

LINEAR MODELS ARE...
LINEAR!

$$Y = X_1 + X_2 \quad \checkmark$$

$$Y = X_1 \times X_2 \quad \times$$

LINEAR MODELS ARE...
EFFICIENT!



THANK YOU GAMERS!

LINEAR MODELS ARE...
STABLE!

BOUNDED

$$Y = WX \rightarrow \Delta Y \sim |W| \Delta X$$

small

small

LINEAR MODELS ARE...
STABLE!

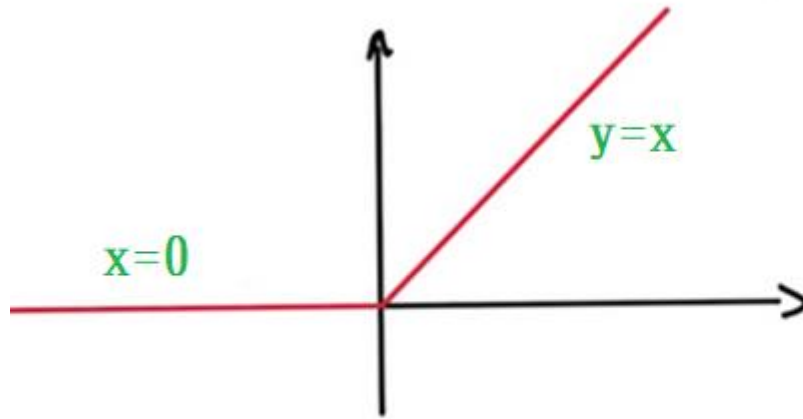
$$Y = WX \rightarrow \begin{aligned} \frac{dY}{dX} &= W^T \\ \frac{dY}{dW} &= X^T \end{aligned} \rightarrow \text{CONSTANTS}$$

LINEAR MODELS ARE...
HERE TO STAY!

$$Y = W_1 W_2 W_3 X = \cancel{W} X$$

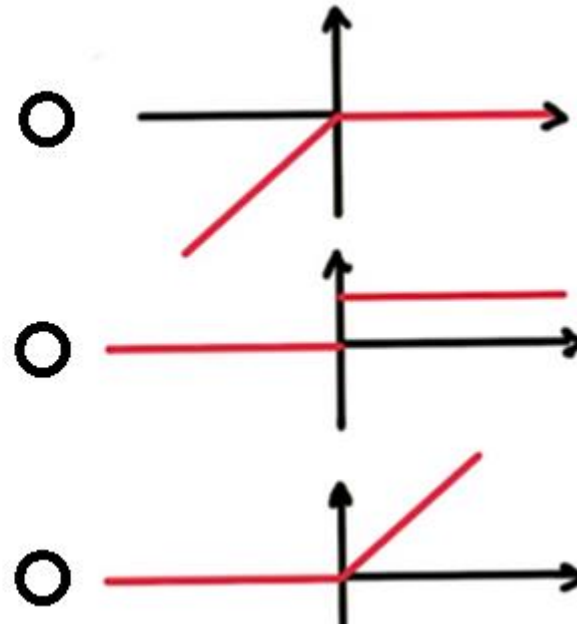
↑ ↑
NON-LINEARITIES

Quiz: Rectified Linear Units

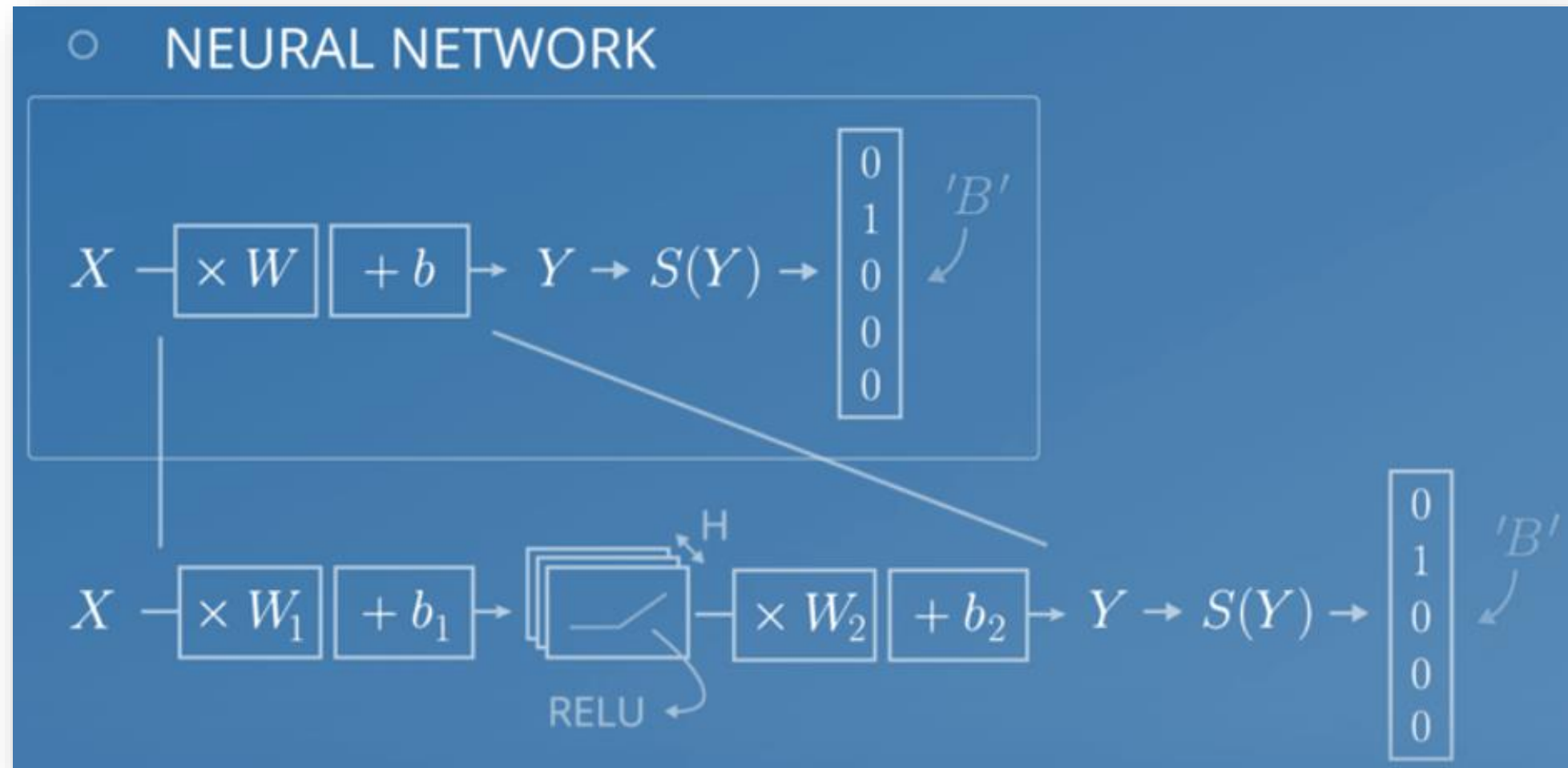


Rectified Linear Units(RELU)

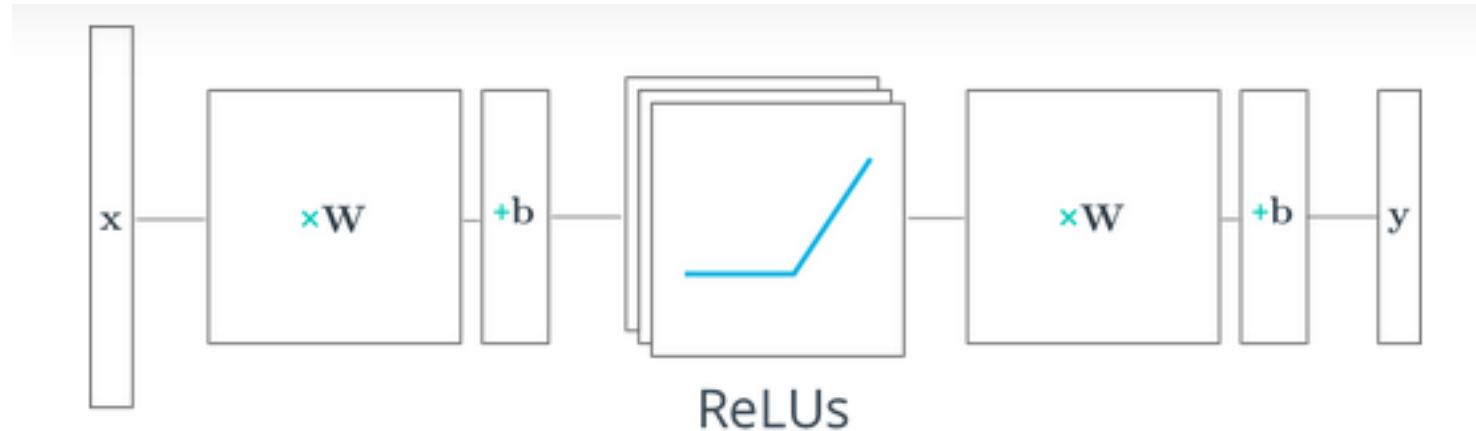
DERIVATIVE?



Network Of ReLUs



2-Layer Neural Network



Multilayer Neural Networks

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

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

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

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

Quiz: TensorFlow ReLUs

TensorFlow ReLUs

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

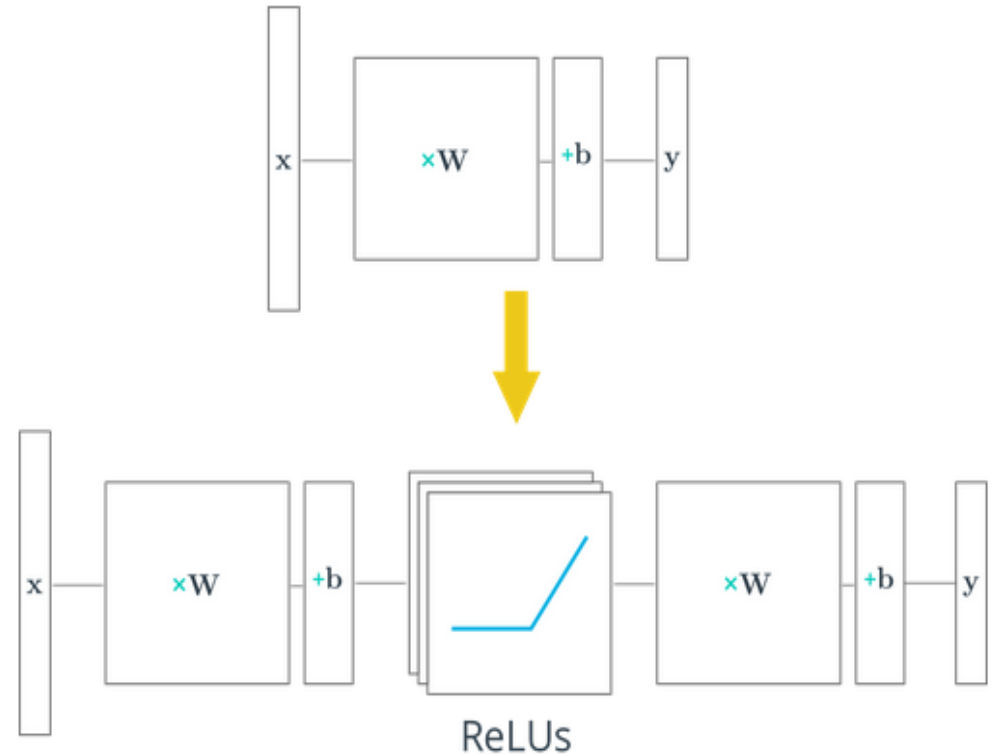
```
# Hidden Layer with ReLU activation function
```

```
hidden_layer = tf.add(tf.matmul(features, hidden_weights), hidden_biases)
hidden_layer = tf.nn.relu(hidden_layer)
```

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

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

Quiz



In this quiz, you'll use TensorFlow's ReLU function to turn the linear model below into a nonlinear model.

Code: [01quiz_py.txt](#)

Code: [01solution_py.txt](#)

Quiz: TensorFlow ReLUs

In this quiz, you'll use TensorFlow's ReLU function to turn the linear model below into a nonlinear model.

Hidden Layer with ReLU activation function

```
hidden_layer = tf.add(tf.matmul(features, hidden_weights), hidden_biases)
hidden_layer = tf.nn.relu(hidden_layer)
```

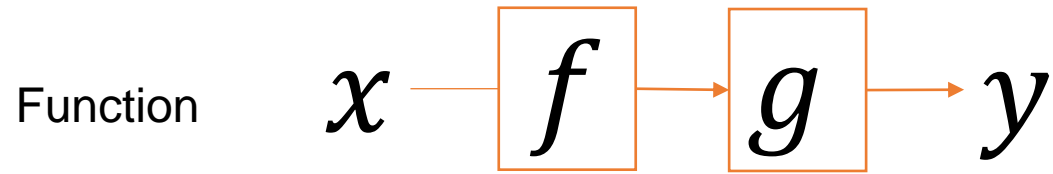
```
output = tf.add(tf.matmul(hidden_layer, output_weights), output_biases)
```

```
[[ 5.6000004  4.9  5.1000004]
 [-5.6000004 -4.9 -5.1000004]
 [23.599998  23.9  24.1   ]]
```

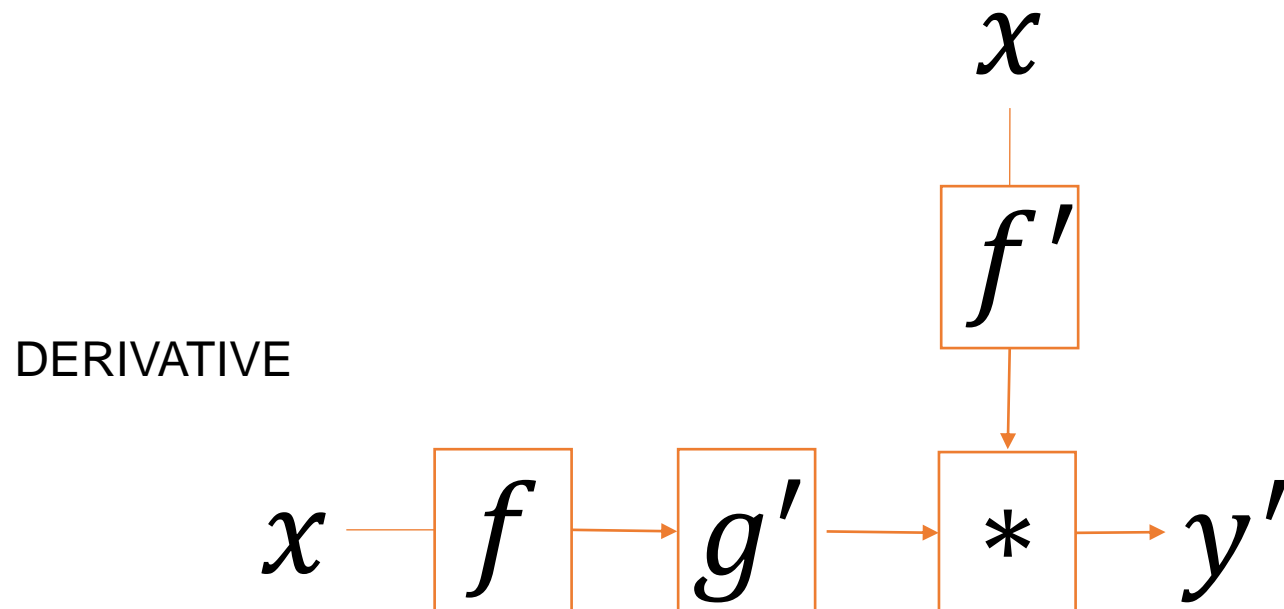
$$\text{tf.nn.relu} \left(\begin{array}{l} \text{features} = \text{tf.Variable}([[1.0, 2.0, 3.0, 4.0], \\ [-1.0, -2.0, -3.0, -4.0], \\ [11.0, 12.0, 13.0, 14.0]]) \\ \text{hidden_layer_weights} = [[0.1, 0.2, 0.4], \\ [0.4, 0.6, 0.6], \\ [0.5, 0.9, 0.1], \\ [0.8, 0.2, 0.8]] \end{array} \right) = \begin{array}{l} \text{biases} = [\text{tf.Variable(tf.zeros(3))}, \\ \text{tf.Variable(tf.zeros(2))}] \end{array} \begin{array}{l} [[5.6000004 \ 4.9 \ 5.1000004] \\ [0. \ 0. \ 0.] \\ [23.599998 \ 23.9 \ 24.1 \]] \end{array}$$

$$\text{output} = \left(\begin{array}{l} [[5.6000004 \ 4.9 \ 5.1000004] \\ [0. \ 0. \ 0.] \\ [23.599998 \ 23.9 \ 24.1 \]] \end{array} \text{out_weights} = [[0.1, 0.6], \\ [0.2, 0.1], \\ [0.7, 0.9]] \right) = \begin{array}{l} \text{biases} = [\text{tf.Variable(tf.zeros(3))}, \\ \text{tf.Variable(tf.zeros(2))}] \end{array} \begin{array}{l} [[5.11 \ 8.440001] \\ [0. \ 0.] \\ [24.010002 \ 38.239998]] \end{array}$$

The Chain Rule & Backprop



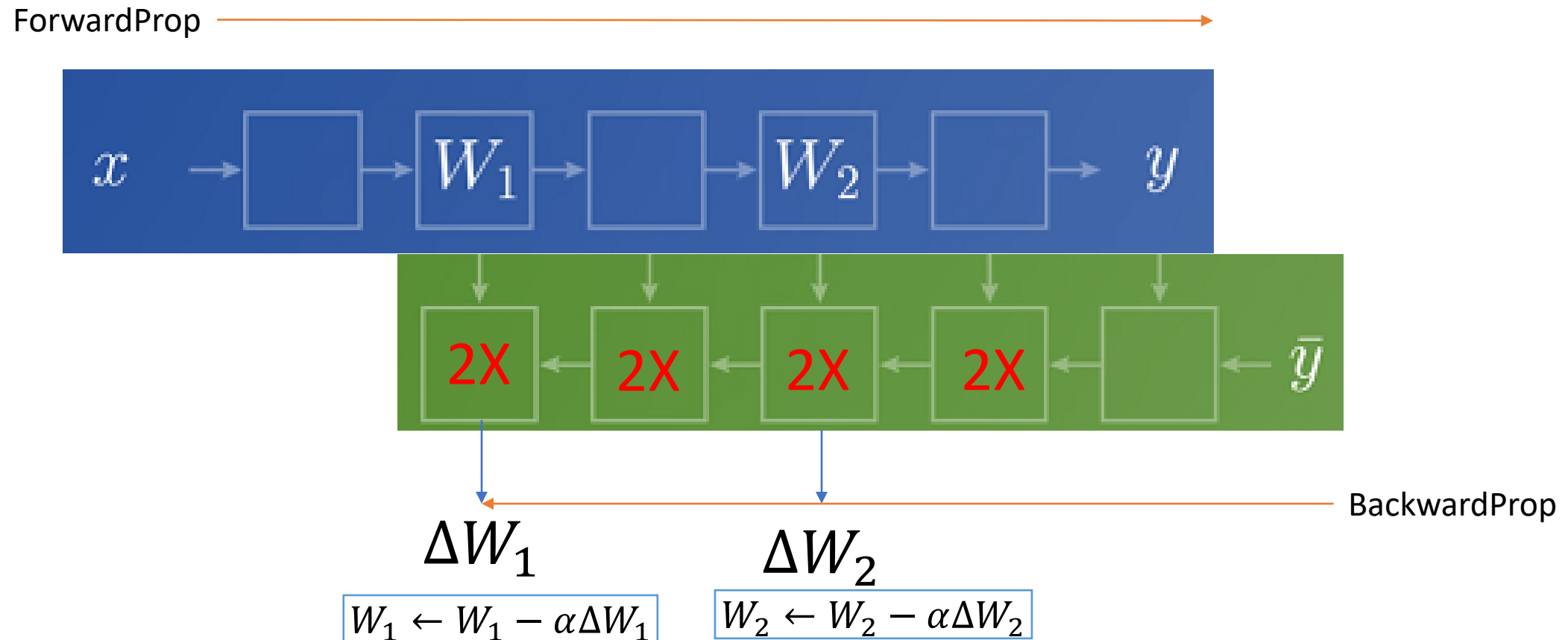
 Chain Rule



- ◆ EFFICIENT DATA PIPELINE
- ◆ LOTS OF DATA REUSE!

The Chain Rule & Backprop

BACK-PROPAGATION



Deep Neural Network in TensorFlow

Deep Neural Network in TensorFlow

You've seen how to build a logistic classifier using TensorFlow. Now you're going to see how to use the logistic classifier to build a deep neural network.

Step by Step

In the following walkthrough, we'll step through TensorFlow code written to classify the letters in the MNIST database. If you would like to run the network on your computer, the file is provided [here](#). You can find this and many more examples of TensorFlow at [Aymeric Damien's GitHub repository](#).

Code --TensorFlow MNIST

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets(".", one_hot=True, reshape=False)
```

You'll use the MNIST dataset provided by TensorFlow, which batches and One-Hot encodes the data for you.

Learning Parameters

```
import tensorflow as tf
# Parameters
learning_rate = 0.001
training_epochs = 20
batch_size = 128 # Decrease batch size if you don't have enough memory
display_step = 1
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)
```

The focus here is on the architecture of multilayer neural networks, not parameter tuning, so here we'll just give you the learning parameters.

Deep Neural Network in TensorFlow

Hidden Layer Parameters

```
n_hidden_layer = 256 # layer number of features
```

The variable `n_hidden_layer` determines the size of the hidden layer in the neural network. This is also known as the width of a layer.

Weights and Biases

```
# Store layers weight & bias
weights = {
    'hidden_layer': tf.Variable(tf.random_normal([n_input,
n_hidden_layer])),
    'out': tf.Variable(tf.random_normal([n_hidden_layer, n_classes]))
}
biases = {
    'hidden_layer': tf.Variable(tf.random_normal([n_hidden_layer])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

Deep neural networks use multiple layers with each layer requiring it's own weight and bias. The 'hidden_layer' weight and bias is for the hidden layer. The 'out' weight and bias is for the output layer. If the neural network were deeper, there would be weights and biases for each additional layer.

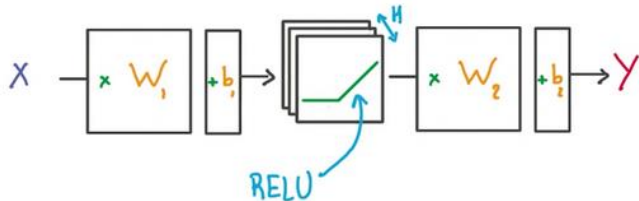
Input

```
# tf Graph input
x = tf.placeholder("float", [None, 28, 28, 1])
y = tf.placeholder("float", [None, n_classes])
x_flat = tf.reshape(x, [-1, n_input])
```

The MNIST data is made up of 28px by 28px images with a single [channel](#). The [tf.reshape\(\)](#) function above reshapes the 28px by 28px matrices in **x** into row vectors of 784px.

Deep Neural Network in TensorFlow

Multilayer Perceptron



Hidden layer with RELU activation

```
layer_1 = tf.add(tf.matmul(x_flat, weights['hidden_layer']),\
    biases['hidden_layer'])
layer_1 = tf.nn.relu(layer_1)
```

Output layer with linear activation

```
logits = tf.add(tf.matmul(layer_1, weights['out']), biases['out'])
```

You've seen the linear function `tf.add(tf.matmul(x_flat, weights['hidden_layer']), biases['hidden_layer'])` before, also known as $xw + b$. Combining linear functions together using a ReLU will give you a two layer network.

Optimizer

Define loss and optimizer

```
cost = tf.reduce_mean(\
    tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)\
    .minimize(cost)
```

This is the same optimization technique used in the Intro to TensorFlow lab.

Session

Initializing the variables

```
init = tf.global_variables_initializer()
```

Launch the graph

```
with tf.Session() as sess:
```

```
    sess.run(init)
```

Training cycle

```
for epoch in range(training_epochs):
```

```
    total_batch = int(mnist.train.num_examples/batch_size)
```

Loop over all batches

```
    for i in range(total_batch):
```

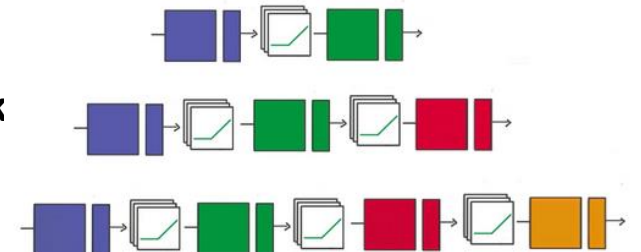
```
        batch_x, batch_y = mnist.train.next_batch(batch_size)
```

Run optimization op (backprop) and cost op (to get loss value)

```
        sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
```

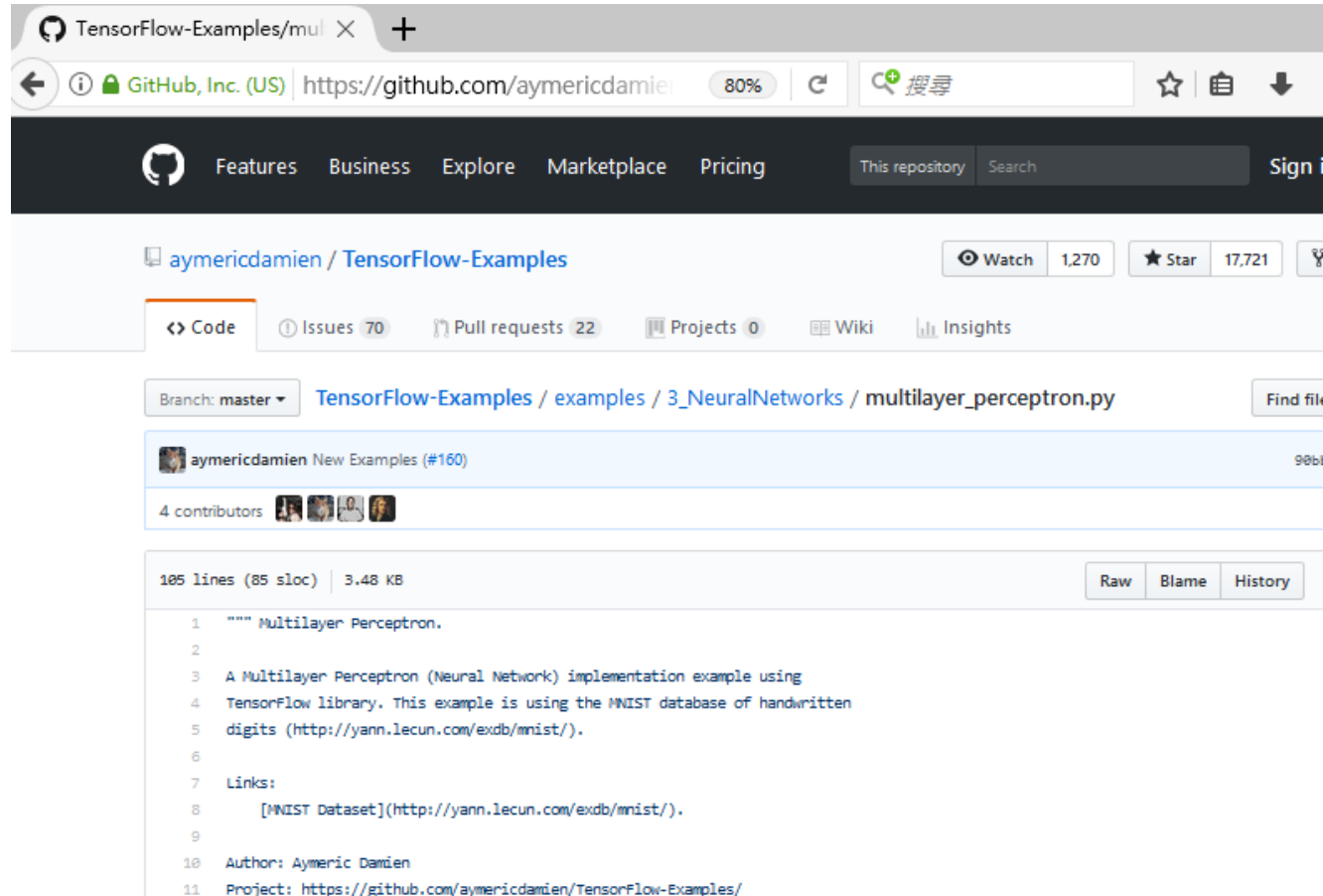
The MNIST library in TensorFlow provides the ability to receive the dataset in batches. Calling the `mnist.train.next_batch()` function returns a subset of the training data.

Deeper Neural Network



Going from one layer to two is easy. Adding more layers to the network allows you to solve more complicated problems. In the next session, you'll see how changing the number of layers can affect your network.

Tensorflow.examples.tutorials (MNIST)



The screenshot shows a web browser displaying the GitHub repository page for 'aymericdamien / TensorFlow-Examples'. The browser's address bar shows the URL 'https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/multilayer_perceptron.py'. The repository page includes a header with the repository name, a 'Watch' button (1,270), a 'Star' button (17,721), and a 'Find file' button. Below the header, there are tabs for 'Code', 'Issues' (70), 'Pull requests' (22), 'Projects' (0), 'Wiki', and 'Insights'. The 'Code' tab is selected, showing the file 'multilayer_perceptron.py' with 105 lines (85 sloc) and 3.48 KB. The file content is displayed in a code editor, showing a Python script for a Multilayer Perceptron (Neural Network) implementation using the TensorFlow library. The script includes a docstring, a description of the example, and links to the MNIST dataset and the author's project.

```
1 """ Multilayer Perceptron.  
2  
3 A Multilayer Perceptron (Neural Network) implementation example using  
4 TensorFlow library. This example is using the MNIST database of handwritten  
5 digits (http://yann.lecun.com/exdb/mnist/).  
6  
7 Links:  
8 [MNIST Dataset](http://yann.lecun.com/exdb/mnist/).  
9  
10 Author: Aymeric Damien  
11 Project: https://github.com/aymericdamien/TensorFlow-Examples/
```

https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/multilayer_perceptron.py

ModuleNotFoundError: No module named 'tensorflow.examples.tutorials'

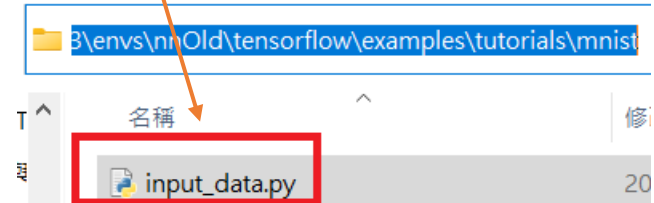
```
from tensorflow.examples.tutorials.mnist import input_data
```

C:\Users\XXX\anaconda3\envs\mnOld\tensorflow\examples\tutorials\mnist

User account

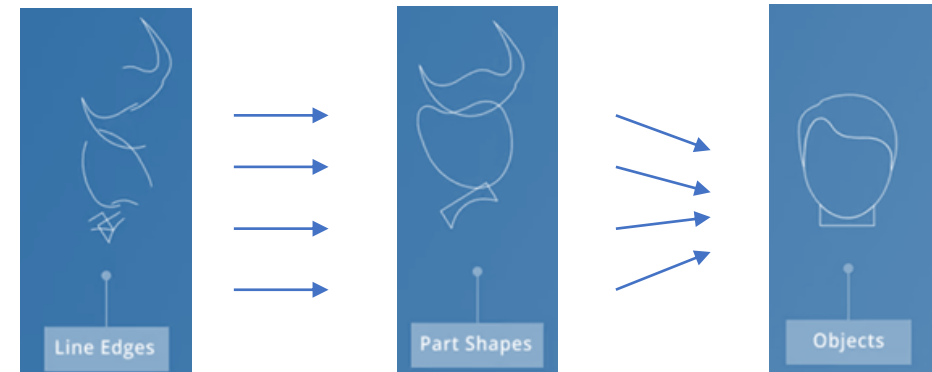
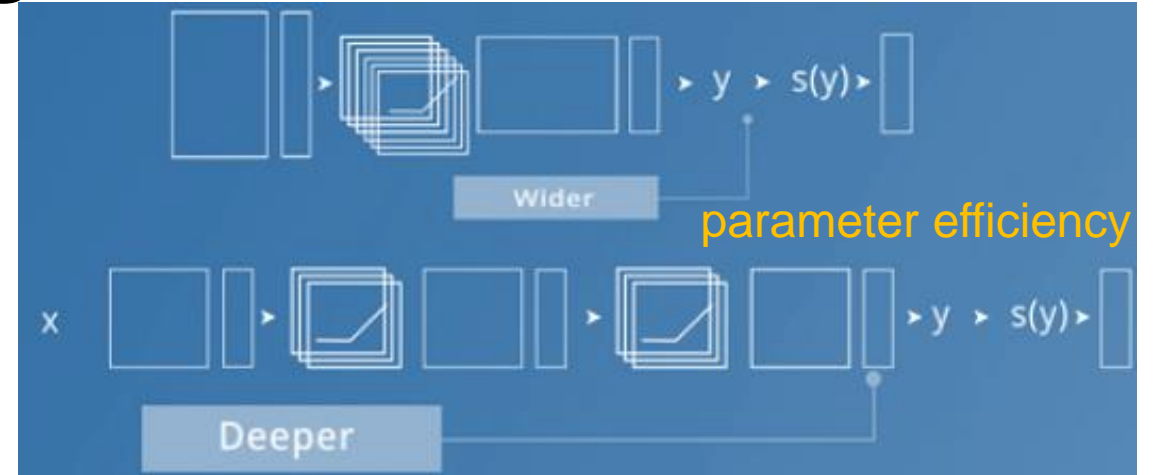
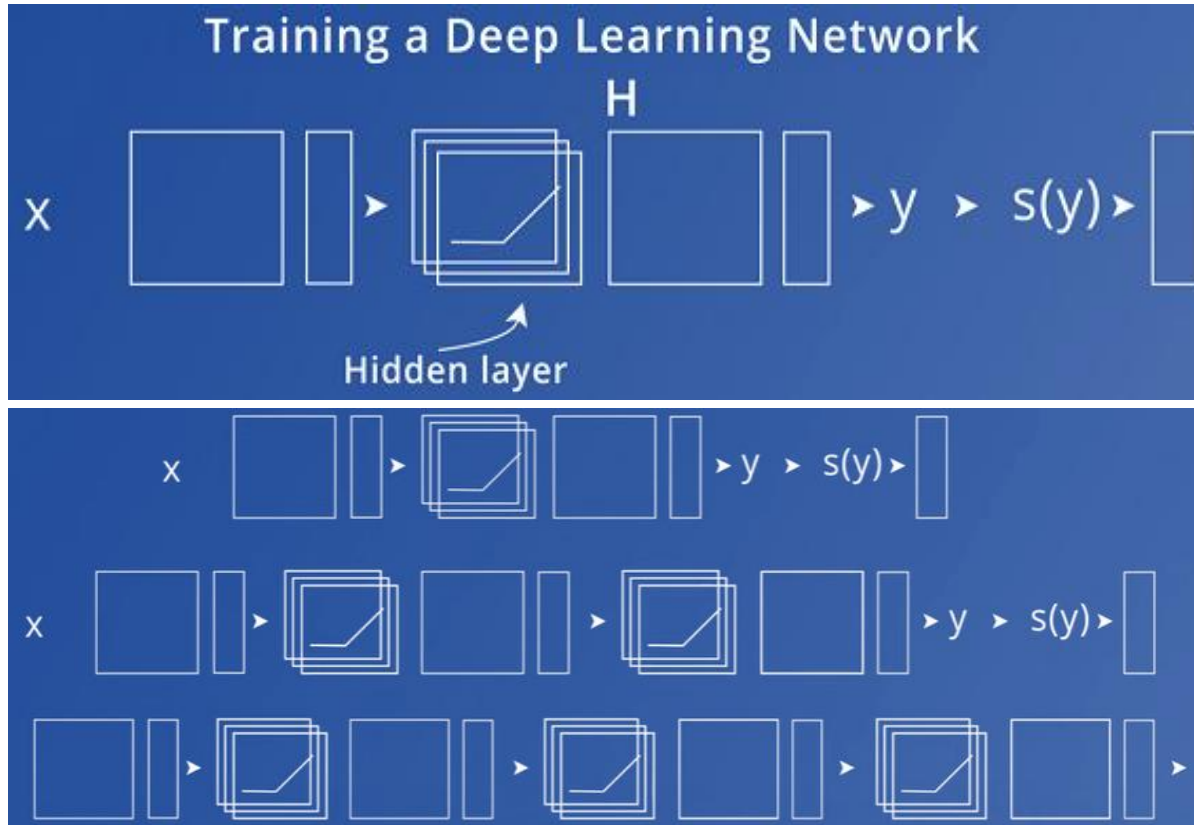
environment name

- A Multilayer Perceptron (Neural Network) implementation example using TensorFlow library. This example is using the MNIST database of handwritten digits (<http://yann.lecun.com/exdb/mnist/>).
- Links:
[MNIST Dataset](<http://yann.lecun.com/exdb/mnist/>).



```
"""Functions for downloading and reading MNIST data."""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import gzip
import os
import numpy
from six.moves import urllib
from six.moves import xrange  # pylint: disable=redefined-builtin
SOURCE_URL = 'http://yann.lecun.com/exdb/mnist/'
```


Training a Deep Learning Network



Save and Restore TensorFlow Models

Save and Restore TensorFlow Models

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

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

Saving Variables

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

```
import tensorflow as tf
# The file path to save the data
save_file = './model.ckpt'
# Two Tensor Variables: weights and bias
weights = tf.Variable(tf.truncated_normal([2, 3]))
bias = tf.Variable(tf.truncated_normal([3]))
# Class used to save and/or restore Tensor Variables
saver = tf.train.Saver()
```

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

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

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

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

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

Loading Variables

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

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

# Two Variables: weights and bias
weights = tf.Variable(tf.truncated_normal([2, 3]))
bias = tf.Variable(tf.truncated_normal([3]))
# Class used to save and/or restore Tensor Variables
saver = tf.train.Saver()

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

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

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

You'll notice you still need to create the weights and bias Tensors in Python. The [tf.train.Saver.restore\(\)](#) function loads the saved data into weights and bias. Since [tf.train.Saver.restore\(\)](#) sets all the TensorFlow Variables, **you don't need to call** [tf.global_variables_initializer\(\)](#).

Save and Restore TensorFlow Models

Save a Trained Model

Let's see how to train a model and save its weights. First start with a model:

```
# Remove previous Tensors and Operations
tf.reset_default_graph()
from tensorflow.examples.tutorials.mnist import input_data
import numpy as np
learning_rate = 0.001
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)
# Import MNIST data
mnist = input_data.read_data_sets('.', one_hot=True)
# Features and Labels
features = tf.placeholder(tf.float32, [None, n_input])
labels = tf.placeholder(tf.float32, [None, n_classes])
# Weights & bias
weights = tf.Variable(tf.random_normal([n_input, n_classes]))
bias = tf.Variable(tf.random_normal([n_classes]))
# Logits - xW + b
logits = tf.add(tf.matmul(features, weights), bias)
# Define loss and optimizer
cost = tf.reduce_mean(\
    tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=labels))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)\
    .minimize(cost)
# Calculate accuracy
correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(labels, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

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

```
import math
save_file = './train_model.ckpt'
batch_size = 128
n_epochs = 100
saver = tf.train.Saver()
# Launch the graph
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # Training cycle
    for epoch in range(n_epochs):
        total_batch = math.ceil(mnist.train.num_examples / batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_features, batch_labels = mnist.train.next_batch(batch_size)
            sess.run(
                optimizer,
                feed_dict={features: batch_features, labels: batch_labels})
        # Print status for every 10 epochs
        if epoch % 10 == 0:
            valid_accuracy = sess.run(
                accuracy,
                feed_dict={
                    features: mnist.validation.images,
                    labels: mnist.validation.labels})
            print('Epoch {:<3} - Validation Accuracy: {}'.format(
                epoch,
                valid_accuracy))
    # Save the model
    saver.save(sess, save_file)
    print('Trained Model Saved.')
```

Save and Restore TensorFlow Models

```
Epoch 0 - Validation Accuracy: 0.06859999895095825
Epoch 10 - Validation Accuracy: 0.20239999890327454
Epoch 20 - Validation Accuracy: 0.36980000138282776
Epoch 30 - Validation Accuracy: 0.48820000886917114
Epoch 40 - Validation Accuracy: 0.5601999759674072
Epoch 50 - Validation Accuracy: 0.6097999811172485
Epoch 60 - Validation Accuracy: 0.6425999999046326
Epoch 70 - Validation Accuracy: 0.6733999848365784
Epoch 80 - Validation Accuracy: 0.6916000247001648
Epoch 90 - Validation Accuracy: 0.7113999724388123
Trained Model Saved.
```

Load a Trained Model

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

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

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

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

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

```
Test Accuracy: 0.7229999899864197
```

That's it! You now know how to save and load a trained model in TensorFlow. Let's look at loading weights and biases into modified models in the next section.

Finetuning

Loading the Weights and Biases into a New Model

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

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

Naming Error

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

```
import tensorflow as tf
# Remove the previous weights and bias
tf.reset_default_graph()
save_file = 'model.ckpt'
# Two Tensor Variables: weights and bias
weights = tf.Variable(tf.truncated_normal([2, 3]))
bias = tf.Variable(tf.truncated_normal([3]))
saver = tf.train.Saver()
# Print the name of Weights and Bias
print('Save Weights: {}'.format(weights.name))
print('Save Bias: {}'.format(bias.name))

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

# Remove the previous weights and bias
tf.reset_default_graph()
# Two Variables: weights and bias
bias = tf.Variable(tf.truncated_normal([3]))
weights = tf.Variable(tf.truncated_normal([2, 3]))
saver = tf.train.Saver()
# Print the name of Weights and Bias
print('Load Weights: {}'.format(weights.name))
print('Load Bias: {}'.format(bias.name))

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

The code above prints out the following:

```
Save Weights: Variable:0
Save Bias: Variable_1:0
Load Weights: Variable_1:0
Load Bias: Variable:0
...
InvalidArgumentError (see above for
traceback): Assign requires shapes of both
tensors to match.
...
```

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

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

Finetuning

```
import tensorflow as tf
tf.reset_default_graph()
save_file = 'model.ckpt'
# Two Tensor Variables: weights and bias
weights = tf.Variable(tf.truncated_normal([2, 3]),
name='weights_0')
bias = tf.Variable(tf.truncated_normal([3]), name='bias_0')
saver = tf.train.Saver()

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

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

```
# Remove the previous weights and bias
tf.reset_default_graph()
# Two Variables: weights and bias
bias = tf.Variable(tf.truncated_normal([3]), name='bias_0')
weights = tf.Variable(tf.truncated_normal([2,
3]), name='weights_0')
saver = tf.train.Saver()
# Print the name of Weights and Bias
print('Load Weights: {}'.format(weights.name))
print('Load Bias: {}'.format(bias.name))

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

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

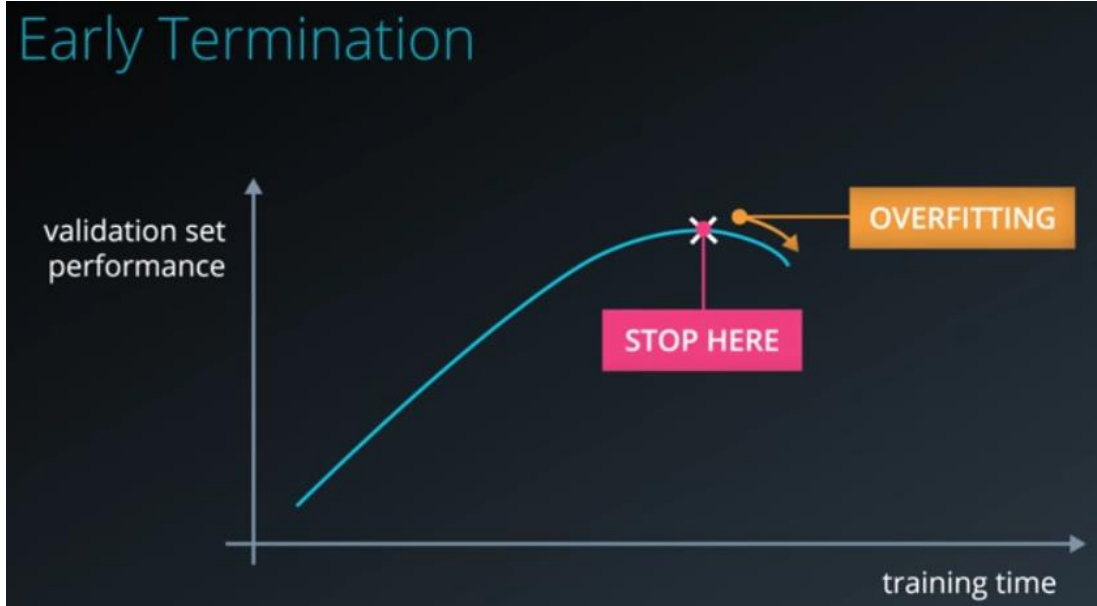
```
Save Weights: weights_0:0
Save Bias: bias_0:0
Load Weights: weights_0:0
Load Bias: bias_0:0
Loaded Weights and Bias successfully.
```

That worked! The Tensor names match and the data loaded correctly.

Reference: https://www.tensorflow.org/guide/saved_model

Regularization

Early Termination



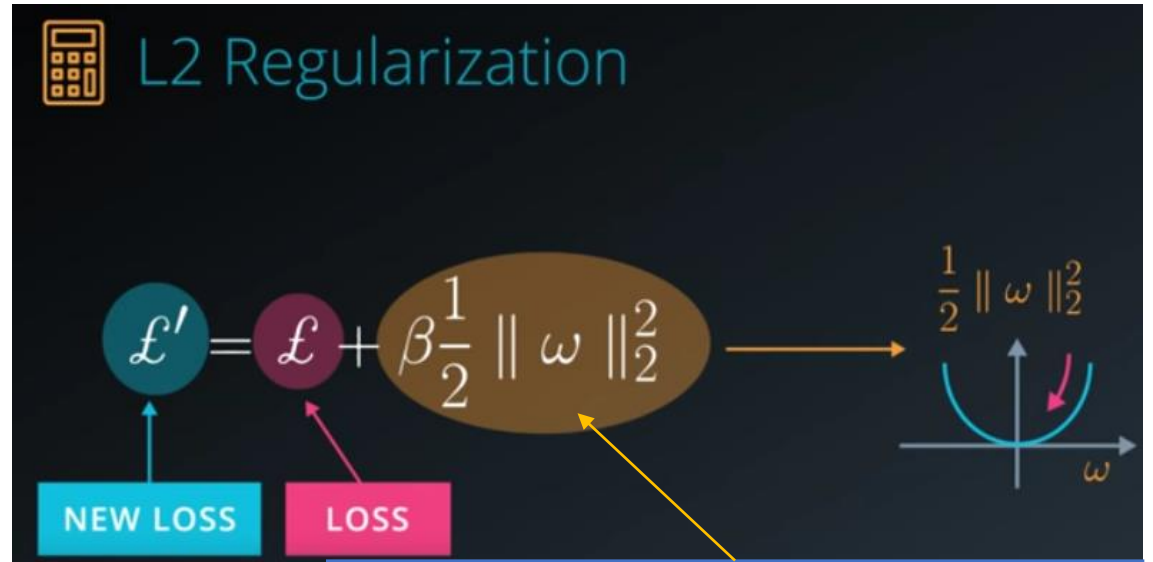
Regularization



Reduce the number of free parameters.



L2 Regularization



The idea is to add another term to the loss, which penalizes large weights.

Regularization Quiz

L₂ REGULARIZATION

$$\mathcal{L}' = \mathcal{L} + \beta \underbrace{\frac{1}{2} \|\omega\|_2^2}_{\frac{1}{2} (\omega_1^2 + \omega_2^2 + \dots + \omega_n^2)}$$

DERIVATIVE?

- a. ☐ $\frac{1}{3} \|\omega\|_2^3$
- b. ☐ $\omega^T \omega$
- c. ☐ ω

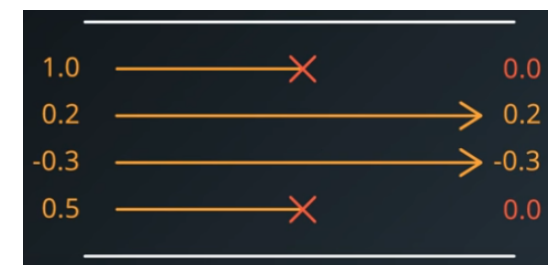
Quiz Question

Which of the options gives you the derivative of the L₂ norm of the vector?

- ☐ A
- ☐ B
- ☐ C

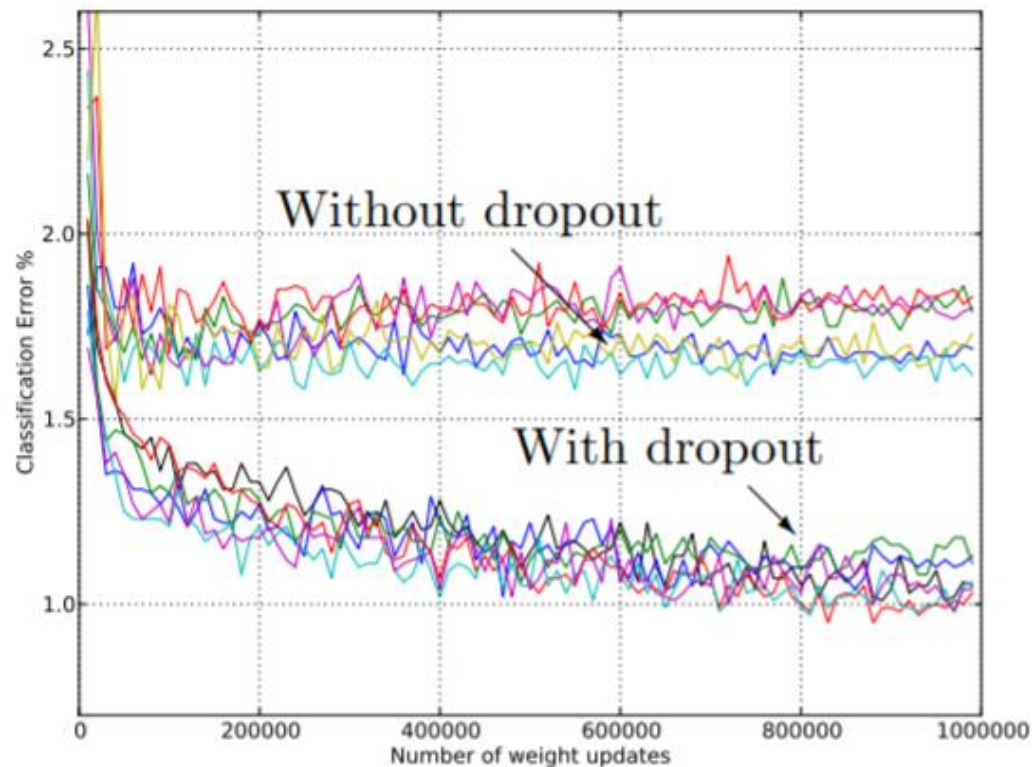
Dropout

prevent overfitting



Dropout

2014 Hinton, Dropout: A Simple Way to Prevent Neural Networks from Overfitting
(original paper: <http://jmlr.org/papers/v15/srivastava14a.html>)



Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Quiz: TensorFlow Dropout

TensorFlow Dropout

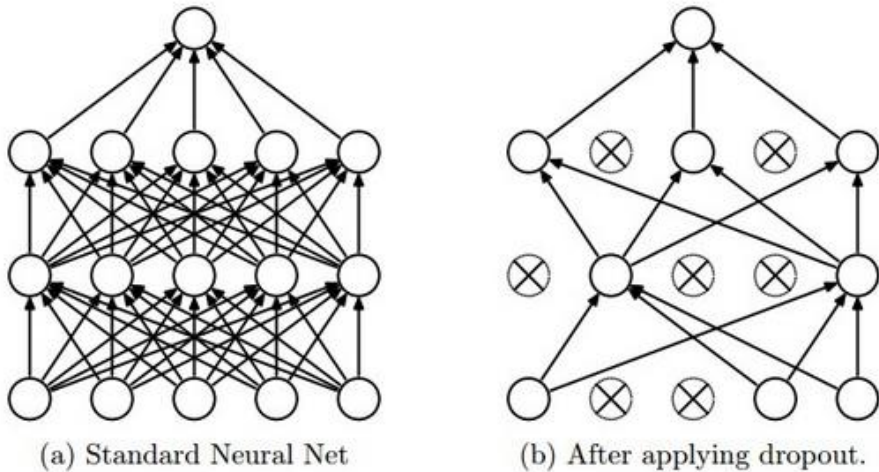


Figure 1: Taken from the paper "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"

(<https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf>)

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

TensorFlow provides the [tf.nn.dropout\(\)](#) function, which you can use to implement dropout.

Let's look at an example of how to use [tf.nn.dropout\(\)](#).

```
keep_prob = tf.placeholder(tf.float32) # probability to keep units
hidden_layer = tf.add(tf.matmul(features, weights[0]), biases[0])
hidden_layer = tf.nn.relu(hidden_layer)
hidden_layer = tf.nn.dropout(hidden_layer, keep_prob)
logits = tf.add(tf.matmul(hidden_layer, weights[1]), biases[1])
```

The code above illustrates how to apply dropout to a neural network. The [tf.nn.dropout\(\)](#) function takes in two parameters:

- 1.**hidden_layer**: the tensor to which you would like to apply dropout
 - 2.**keep_prob**: the probability of keeping (i.e. *not* dropping) any given unit
- `keep_prob` allows you to adjust the number of units to drop. In order to compensate for dropped units, [tf.nn.dropout\(\)](#) multiplies all units that are kept (i.e. *not* dropped) by $1/\text{keep_prob}$.

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

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

Quiz 1

Take a look at the code snippet below. Do you see what's wrong? There's nothing wrong with the syntax, however the test accuracy is extremely low.

Quiz: TensorFlow Dropout

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

hidden_layer = tf.add(tf.matmul(features, weights[0]), biases[0])
hidden_layer = tf.nn.relu(hidden_layer)
hidden_layer = tf.nn.dropout(hidden_layer, keep_prob)
logits = tf.add(tf.matmul(hidden_layer, weights[1]), biases[1])
...

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for epoch_i in range(epochs):
        for batch_i in range(batches):
            ....
            sess.run(optimizer, feed_dict={
                features: batch_features,
                labels: batch_labels,
                keep_prob: 0.5})

    validation_accuracy = sess.run(accuracy, feed_dict={
        features: test_features,
        labels: test_labels,
        keep_prob: 0.5})
```

Question 1 of 2

What's wrong with the above code?

- ☐ Dropout doesn't work with batching.
- ☐ The keep_prob value of 0.5 is too low.
- ☐ There shouldn't be a value passed to keep_prob when testing for accuracy.
- ☐ keep_prob should be set to 1.0 when evaluating validation accuracy.

Quiz 2

This quiz will be starting with the code from the ReLU Quiz and applying a dropout layer. Build a model with a ReLU layer and dropout layer using the **keep_prob** placeholder to pass in a probability of **0.5**. Print the logits from the model.

Note: Output will be different every time the code is run. This is caused by dropout randomizing the units it drops.

Code: [02quiz_py.txt](#)

Quiz: TensorFlow Dropout

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

hidden_layer = tf.add(tf.matmul(features, weights[0]), biases[0])
hidden_layer = tf.nn.relu(hidden_layer)
hidden_layer = tf.nn.dropout(hidden_layer, keep_prob)
logits = tf.add(tf.matmul(hidden_layer, weights[1]), biases[1])
...

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for epoch_i in range(epochs):
        for batch_i in range(batches):
            ....
            sess.run(optimizer, feed_dict={
                features: batch_features,
                labels: batch_labels,
                keep_prob: 0.5})

validation_accuracy = sess.run(accuracy, feed_dict={
    features: test_features,
    labels: test_labels,
    keep_prob: 0.5})
```

Question 1 of 2

What's wrong with the above code?

- ☐ Dropout doesn't work with batching.
- ☐ The keep_prob value of 0.5 is too low.
- ☐ There shouldn't be a value passed to keep_prob when testing for accuracy.
- ☒ keep_prob should be set to 1.0 when evaluating validation accuracy.

Quiz 2

This quiz will be starting with the code from the ReLU Quiz and applying a dropout layer. Build a model with a ReLU layer and dropout layer using the **keep_prob** placeholder to pass in a probability of **0.5**. Print the logits from the model.

Note: Output will be different every time the code is run. This is caused by dropout randomizing the units it drops.

Code: [02quiz_py.txt](#)

Code: [02solution_py.txt](#)