MODULE 1 – INTRODUCTION TO TENSORFLOW

INTRODUCTION

What is TensorFlow?

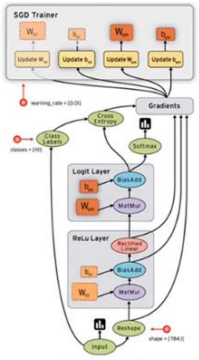
- Open Source software library by Google
- Originally created for tasks with heavy numerical computations
- Main Application: Machine Learning & Deep Neural Networks
- C/C++ backend
- Based on data flow graphs

Why TensorFlow?

- Python and C++ API
- Faster compile times
- Supports CPUs, GPUs, and distributed processing

What is a data flow graph?

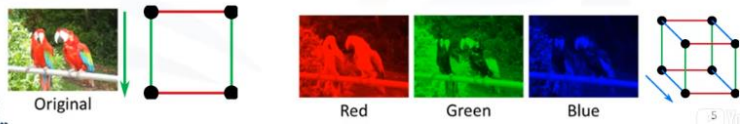
- We create a graph with the following computation units:
  - Nodes = Mathematical Operations
  - Edges = Multi-Dimensional Arrays (Tensors)



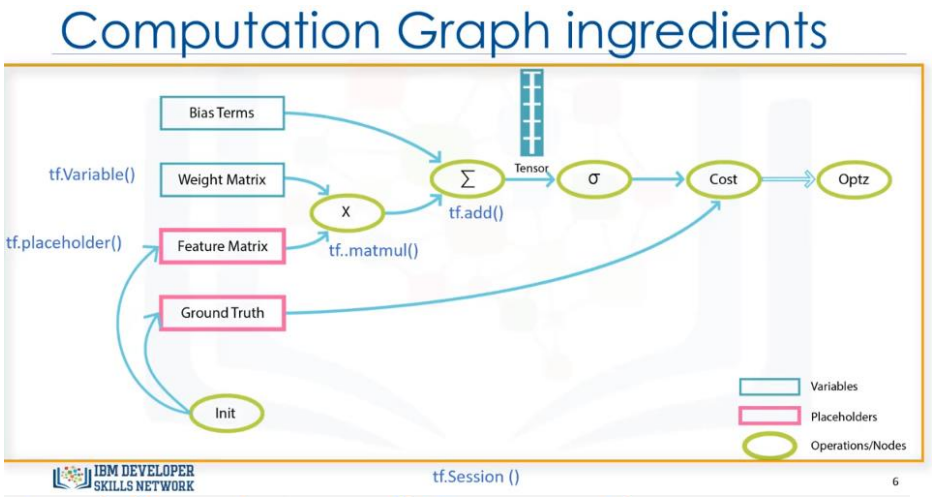
IBM DEVELOPER

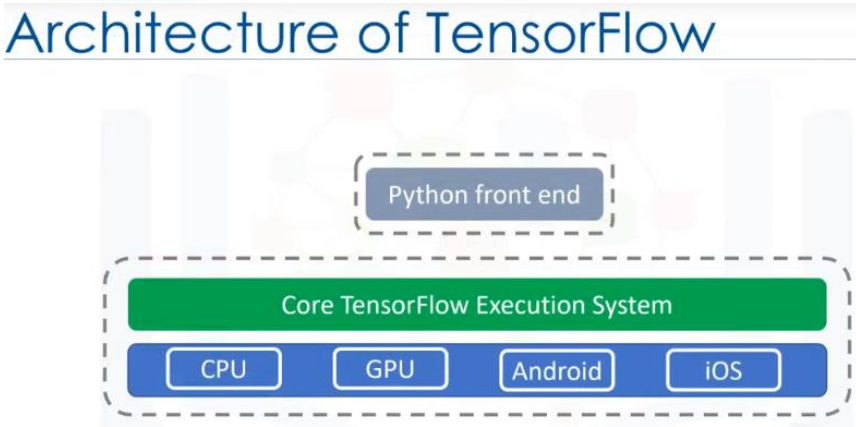
What is the meaning of Tensor?

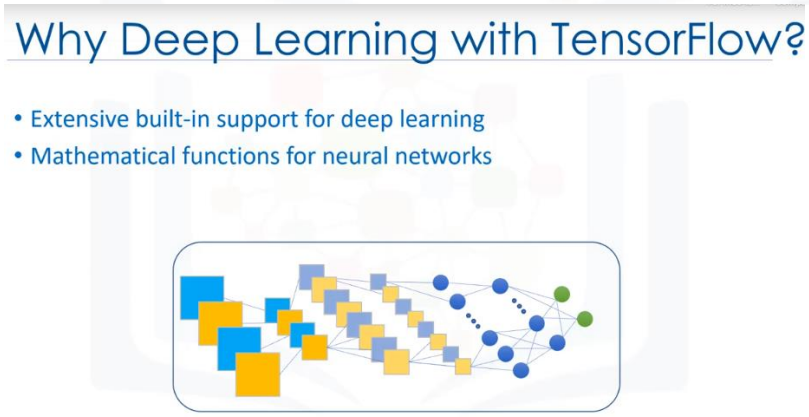
Dimension	In Code	Shape
0	1	
1	[1,2,3,4,...]	
2	[[1,2,3,4,...], [1,2,3,4,...]...]	
3	[[[1,2,3,4,...],[1,2,3,4,...]...], [[1,2,3,4,...], [1,2,3,4,...]...], ...]	
4	[[[[1,2,3,4,...],[1,2,3,4,...]...], [[1,2,3,4,...], [1,2,3,4,...]...], ...] [[[1,2,3,4,...],[1,2,3,4,...]...], [[1,2,3,4,...], [1,2,3,4,...]...], ...], ..]	



IBM DEVELOPER SKILLS NETWORK







Hello, and welcome! In this video, we'll be going through a
brief overview of the structure and capabilities of the TensorFlow library.
So let's get started.
TensorFlow is an open source library developed by the Google Brain Team.
It's an extremely versatile library, originally created for tasks that require heavy numerical computations. For this reason, TensorFlow was geared towards
the problem of machine learning, and deep neural networks.
Due to a C/C++ backend, TensorFlow is able to run faster than pure Python code.
The last thing I'd like to mention is that a TensorFlow application uses a structure known as a data flow graph, which is very useful to first build and then execute it
in a session. It is also a very common programming model
for parallel computing. We'll cover this in more detail shortly.
TensorFlow offers several advantages for an application.

For instance, it provides both a Python and a C++ API. It should be noted, though, that Python API is more complete and it's generally easier to use.
TensorFlow also has great compilation times in comparison to the alternative deep learning libraries. And it supports CPUs, GPUs, and even distributed
processing in a cluster. It is a very important feature as you can train a neural network using CPU and multiple GPUs, which makes the models very efficient on large-scale systems.
TensorFlow's structure is based on the execution of a data flow graph. So, let's take a closer look at this. A data flow graph, has two basic units.
The nodes that represent a mathematical operation, and the edges which represent the multi-dimensional
arrays, known as a tensors. So this high-level abstraction reveals how the data flows between operations. Please notice that using the data flow graph,
we can easily visualize different parts of the graph, which is not an option while using other Python libraries such as Numpy or SciKit. The standard usage is to build a graph first and then execute it in a session.
So, What is a Tensor? Well as we've already noted, the data that's passed between the operations are Tensors. In effect, a Tensor is a multidimensional
array. It can be zero dimensional, such as scalar values, one dimensional as a line or vector, or 2-dimensional, such as a Matrix, and so on.
The Tensor structure helps us by giving us the freedom to shape the dataset the way we want. It's also particularly helpful when dealing
with images, due to the nature of how the information within images is encoded.
Indeed, when you think about an image, it's easy to understand that it has height and width, so it would make sense to represent the information contained in it with a two dimensional structure, such as a matrix, for example.
But as you know, images have colors, and to add information about the colors, we need another dimension, and that's when a 3-dimensional tensor becomes particularly helpful.
Now, let's take a look at a dataflow graph and see how tensors and operations build the graph. As mentioned before, in a dataflow graph,
the nodes are called operations, which represent units of computation.
The edges are tensors which represent the data consumed or produced by an operation.
In this graph, Feature matrix is a placeholder. Placeholders can be seen as "holes" in your model -- "holes" through which you can pass the data from outside of the graph. Placeholders allow us to create our operations in the graph, without needing the data.
When we want to execute the graph, we have to feed placeholders with our input data.
This is why we need to initialize placeholders before using them.
Let's look at another operation which builds the variables for our program.
In this graph, Weight Matrix is a variable. TensorFlow variables, are used to share and persist some values, that are manipulated by the program
Please notice that when you define a place- holder or variable, TensorFlow adds an operation to
your graph. In our graph, "Weight matrix" and "Feature
matrix" should be multiplied using a multiplication operation. After that, Add operation is called, which
adds the result of the previous operation with bias term. The output of each operation is a tensor. The resulting tensors of each operation crosses the next one until the end where it's possible to get the desired result. After adding all these operations in a graph, we can create a session to run the graph, and perform the computations.
For a better understanding of TensorFlow graphs, I recommend that you run the labs from this module, which walk you through different elements of the data flow, in the TensorFlow library.
As previously mentioned, TensorFlow comes with an easy to use Python interface to build and execute your computational graphs. But what makes TensorFlow so popular today, is

its architecture. TensorFlow's flexible architecture allows
you to deploy computation on one or more CPUs, or GPUs, or on a desktop, server, or even a mobile device. This means you build your program once, and then you can run it easily on different devices.
So let's briefly review the reasons why TensorFlow is well-suited for deep learning applications. First, TensorFlow has built-in support for
deep learning and neural networks, so it's easy to assemble a net, assign parameters, and run the training process. Second, it also has a collection of simple,
trainable mathematical functions that are useful for neural networks.
Finally, deep learning as a gradient-based machine learning algorithm will benefit from TensorFlow's auto-differentiation and optimizers.
This concludes the video … Thanks for watching!
End of transcript. Skip to the start.

TENSORFOW'S HELLO WORD

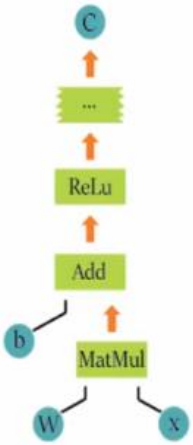
How does TensorFlow work?

TensorFlow's capability to execute the code on different devices such as CPUs and GPUs is a consequence of it's specific structure:

TensorFlow defines computations as Graphs, and these are made with operations (also know as "ops"). So, when we work with TensorFlow, it is the same as defining a series of operations in a Graph.

To execute these operations as computations, we must launch the Graph into a Session. The session translates and passes the operations represented into the graphs to the device you want to execute them on, be it a GPU or CPU.

For example, the image below represents a graph in TensorFlow. *W*, *x* and b are tensors over the edges of this graph. *MatMul* is an operation over the tensors *W* and *x*, after that *Add* is called and add the result of the previous operator with *b*. The resultant tensors of each operation cross the next one until the end where it's possible to get the wanted result.



Importing TensorFlow

To use TensorFlow, we need to import the library. We imported it and optionally gave it the name "tf", so the modules can be accessed by **tf.module-name**:

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

Building a Graph

As we said before, TensorFlow works as a graph computational model. Let's create our first graph.

To create our first graph we will utilize **source operations**, which do not need any information input. These source operations or **source ops** will pass their information to other operations which will execute computations.

To create two source operations which will output numbers we will define two constants:

```
a = tf.constant([2])
b = tf.constant([3])
```

After that, let's make an operation over these variables. The function **tf.add()** adds two elements (you could also use `c = a + b`).

```
[ ]: c = tf.add(a,b)
     #c = a + b is also a way to define the sum of the terms
```

Then TensorFlow needs to initialize a session to run our code. Sessions are, in a way, a context for creating a graph inside TensorFlow. Let's define our session:

```
session = tf.Session()
```

Let's run the session to get the result from the previous defined 'c' operation:

```
: result = session.run(c)
  print(result)
```

```
[5]
```

Close the session to release resources:

```
session.close()
```

To avoid having to close sessions every time, we can define them in a **with** block, so after running the **with** block the session will close automatically:

```
with tf.Session() as session:
    result = session.run(c)
    print(result)
```

Even this silly example of adding 2 constants to reach a simple result defines the basis of TensorFlow. Define your edge (In this case our constants), include nodes (operations, like *tf.add*), and start a session to build a graph.

Start of transcript. Skip to the end.
hello and welcome in this video we will provide an overview of the tensorflow library as well as the structure of a basic tensorflow application tensorflow is an open source library for creating large-scale machine learning applications it can model computations on a wide variety of hardware ranging from android devices to heterogeneous multi-gpu systems tensorflow uses a special structure in order to execute code on different devices like CPUs and GPUs computations are defined as a graph and each graph is made up of operations

also known as ops so whenever we work with tensorflow we define the series of operations in a graph to run these operations we need to launch the graph into a session the session translates the operations and passes them to a device for execution to use tensorflow we need to import the library we'll give it the name TF so that we can access a module by writing TF dot and then the modules name to create our first graph will start by using source operations which do not require any input these source operations or source Ops will pass their information to other operations which will actually run computations let's create two source operations that will output numbers will find them is A and B which you can see in the following piece of code we'll also define a simple computational operation TF dot add is used to sum two elements you could also use C equals A plus B since graphs need to be executed in the context of a session we need to create a session object to watch the graph let's run the session to get the result from the previously defined see operation the following command closes the session in order to release resources to avoid having to close the sessions every time we can define them in a with block after running the with block the session will close automatically
you're probably thinking that was a lot of work just to add two numbers together but it's extremely important that you understand the basic structure of tensorflow because once you do you can define any computations that you want and again tensorflow structure allows it to handle computations on different devices and even in clusters if you want to learn more about this you can run the method TF dot device feel free to experiment with the structure of tensorflow in order to get a better idea of how it works if you want a list of all the mathematical operations that tensorflow supports you can check out the documentation by now you should understand the structure of tensorflow

and how to create a basic application
thank you for watching this video to
practice and learn more go to the lab
and run the code for yourself
End of transcript. Skip to the start.

>>LAB

Importing TensorFlow

To use TensorFlow, we need to import the library. We imported it and optionally gave it the name "tf", so the modules can be accessed by **tf.module-name**:

```
import tensorflow as tf
```

Building a Graph

As we said before, TensorFlow works as a graph computational model. Let's create our first graph which we named as **graph1**.

```
graph1 = tf.Graph()
```

Now we call the TensorFlow functions that construct new **tf.Operation** and **tf.Tensor** objects and add them to the **graph1**. As mentioned, each **tf.Operation** is a **node** and each **tf.Tensor** is an edge in the graph.

Lets add 2 constants to our graph. For example, calling tf.constant([2], name = 'constant_a') adds a single **tf.Operation** to the default graph. This operation produces the value 2, and returns a **tf.Tensor** that represents the value of the constant.
Notice: tf.constant([2], name="constant_a") creates a new tf.Operation named "constant_a" and returns a tf.Tensor named "constant_a:0".

```
with graph1.as_default():
    a = tf.constant([2], name = 'constant_a')
    b = tf.constant([3], name = 'constant_b')
```

Lets look at the tensor **a**.

```
a
```

As you can see, it just show the name, shape and type of the tensor in the graph. We will see it's value when we run it in a TensorFlow session.

```
# Printing the value of a
sess = tf.Session(graph = graph1)
result = sess.run(a)
print(result)
sess.close()
```

```
[2]
```

After that, let's make an operation over these tensors. The function **tf.add()** adds two tensors (you could also use `c = a + b`).

```
with graph1.as_default():
    c = tf.add(a, b)
    #c = a + b is also a way to define the sum of the terms
```

Then TensorFlow needs to initialize a session to run our code. Sessions are, in a way, a context for creating a graph inside TensorFlow. Let's define our session:

```
sess = tf.Session(graph = graph1)
```

Let's run the session to get the result from the previous defined 'c' operation:

```
result = sess.run(c)
print(result)
```

```
[5]
```

Close the session to release resources:

```
sess.close()
```

To avoid having to close sessions every time, we can define them in a **with** block, so after running the **with** block the session will close automatically:

```
with tf.Session(graph = graph1) as sess:
    result = sess.run(c)
    print(result)
```

```
[5]
```

Even this silly example of adding 2 constants to reach a simple result defines the basis of TensorFlow. Define your operations (In this case our constants and *tf.add*), and start a session to build a graph.

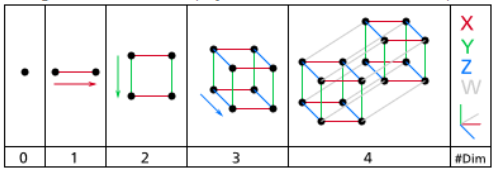
What is the meaning of Tensor?

In TensorFlow all data is passed between operations in a computation graph, and these are passed in the form of Tensors, hence the name of TensorFlow.

The word **tensor** from new latin means "that which stretches". It is a mathematical object that is named "tensor" because an early application of tensors was the study of materials stretching under tension. The contemporary meaning of tensors can be taken as multidimensional arrays.

That's great, but... what are these multidimensional arrays?

Going back a little bit to physics to understand the concept of dimensions:



The zero dimension can be seen as a point, a single object or a single item.

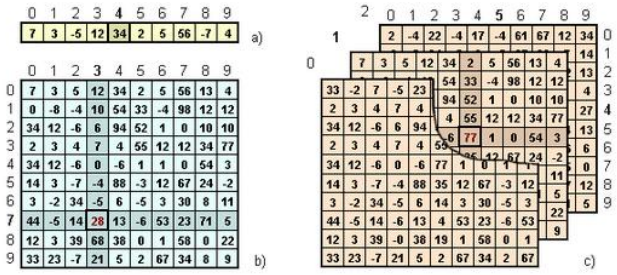
The first dimension can be seen as a line, a one-dimensional array can be seen as numbers along this line, or as points along the line. One dimension can contain infinite zero dimension/points elements.

The second dimension can be seen as a surface, a two-dimensional array can be seen as an infinite series of lines along an infinite line.

The third dimension can be seen as volume, a three-dimensional array can be seen as an infinite series of surfaces along an infinite line.

The Fourth dimension can be seen as the hyperspace or spacetime, a volume varying through time, or an infinite series of volumes along an infinite line. And so forth on...

As mathematical objects:



Summarizing:

Dimension	Physical Representation	Mathematical Object	In Code
Zero	Point	Scalar (Single Number)	[1]
One	Line	Vector (Series of Numbers)	[1,2,3,4,...]
Two	Surface	Matrix (Table of Numbers)	[[1,2,3,4,...], [1,2,3,4,...], [1,2,3,4,...],...]
Three	Volume	Tensor (Cube of Numbers)	[[[1,2,...], [1,2,...], [1,2,...],...], [[1,2,...], [1,2,...], [1,2,...],...], [[1,2,...], [1,2,...], [1,2,...] ,...]...]

Defining multidimensional arrays using TensorFlow

Now we will try to define such arrays using TensorFlow:

```
graph2 = tf.Graph()
with graph2.as_default():
    Scalar = tf.constant(2)
    Vector = tf.constant([5,6,2])
    Matrix = tf.constant([[1,2,3],[2,3,4],[3,4,5]])
    Tensor = tf.constant( [ [[1,2,3],[2,3,4],[3,4,5]] , [[4,5,6],[5,6,7],[6,7,8]] , [[7,8,9],[8,9,10],[9,10,11]] ] )
with tf.Session(graph = graph2) as sess:
    result = sess.run(Scalar)
    print ("Scalar (1 entry):\n %s \n" % result)
    result = sess.run(Vector)
    print ("Vector (3 entries) :\n %s \n" % result)
    result = sess.run(Matrix)
    print ("Matrix (3x3 entries):\n %s \n" % result)
    result = sess.run(Tensor)
    print ("Tensor (3x3x3 entries) :\n %s \n" % result)
```

```
Scalar (1 entry):
 2

Vector (3 entries) :
 [5 6 2]

Matrix (3x3 entries):
 [[1 2 3]
 [2 3 4]
 [3 4 5]]

Tensor (3x3x3 entries) :
 [[[ 1  2  3]
  [ 2  3  4]
  [ 3  4  5]]

 [[ 4  5  6]
  [ 5  6  7]
  [ 6  7  8]]
```

tf.shape returns the shape of our data structure.

```
Scalar.shape
```

```
TensorShape([])
```

```
Tensor.shape
```

```
TensorShape([Dimension(3), Dimension(3), Dimension(3)])
```

We then need to use another TensorFlow function called **tf.matmul():**

```
graph4 = tf.Graph()
with graph4.as_default():
    Matrix_one = tf.constant([[2,3],[3,4]])
    Matrix_two = tf.constant([[2,3],[3,4]])

    mul_operation = tf.matmul(Matrix_one, Matrix_two)

with tf.Session(graph = graph4) as sess:
    result = sess.run(mul_operation)
    print ("Defined using tensorflow function :")
    print(result)
```

```
Defined using tensorflow function :
[[13 18]
 [18 25]]
```

We could also define this multiplication ourselves, but there is a function that already does that, so no need to reinvent the wheel!

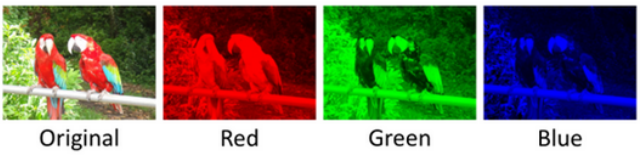
Why Tensors?

The Tensor structure helps us by giving the freedom to shape the dataset in the way we want.

And it is particularly helpful when dealing with images, due to the nature of how information in images are encoded,

Thinking about images, its easy to understand that it has a height and width, so it would make sense to represent the information contained in it with a two dimensional structure (a matrix)... until you remember that images have colors, and to add information about the colors, we need another dimension, and thats when Tensors become particularly helpful.

Images are encoded into color channels, the image data is represented into each color intensity in a color channel at a given point, the most common one being RGB, which means Red, Blue and Green. The information contained into an image is the intensity of each channel color into the width and height of the image, just like this:



Variables

Now that we are more familiar with the structure of data, we will take a look at how TensorFlow handles variables. **First of all, having tensors, why do we need variables?**
TensorFlow variables are used to share and persistent some stats that are manipulated by our program. That is, when you define a variable, TensorFlow adds a **tf.Operation** to your graph. Then, this operation will store a writable tensor value that persists between tf.Session.run calls. So, you can update the value of a variable through each run, while you cannot update tensor (e.g a tensor created by tf.constant()) through multiple runs in a session.

How to define a variable?
To define variables we use the command **tf.Variable()**. To be able to use variables in a computation graph it is necessary to initialize them before running the graph in a session. This is done by running **tf.global_variables_initializer()**.

To update the value of a variable, we simply run an assign operation that assigns a value to the variable:

```
v = tf.Variable(0)
```

Let's first create a simple counter, a variable that increases one unit at a time:

To do this we use the **tf.assign(reference_variable, value_to_update)** command. **tf.assign** takes in two arguments, the **reference_variable** to update, and assign it to the **value_to_update** it by.

```
update = tf.assign(v, v+1)
```

Variables must be initialized by running an initialization operation after having launched the graph. We first have to add the initialization operation to the graph:

```
init_op = tf.global_variables_initializer()
```

We then start a session to run the graph, first initialize the variables, then print the initial value of the **state** variable, and then run the operation of updating the **state** variable and printing the result after each update:

```
with tf.Session() as session:
    session.run(init_op)
    print(session.run(v))
    for _ in range(3):
        session.run(update)
        print(session.run(v))
```

```
0
1
2
3
```

Placeholders

Now we know how to manipulate variables inside TensorFlow graph, but what about feeding data outside of a TensorFlow graph?

If you want to feed data to a TensorFlow graph from outside a graph, you will need to use placeholders.

So **what are these placeholders and what do they do?**

Placeholders can be seen as "holes" in your model, "holes" which you will pass the data to, you can create them using **tf.placeholder(*datatype*)**, where ***datatype*** specifies the type of data (integers, floating points, strings, booleans) along with its precision (8, 16, 32, 64) bits.

The definition of each data type with the respective python syntax is defined as:

Data type	Python type	Description
DT_FLOAT	tf.float32	32 bits floating point.
DT_DOUBLE	tf.float64	64 bits floating point.
DT_INT8	tf.int8	8 bits signed integer.
DT_INT16	tf.int16	16 bits signed integer.
DT_INT32	tf.int32	32 bits signed integer.
DT_INT64	tf.int64	64 bits signed integer.
DT_UINT8	tf.uint8	8 bits unsigned integer.
DT_STRING	tf.string	Variable length byte arrays. Each element of a Tensor is a byte array.
DT_BOOL	tf.bool	Boolean.
DT_COMPLEX64	tf.complex64	Complex number made of two 32 bits floating points: real and imaginary parts.
DT_COMPLEX128	tf.complex128	Complex number made of two 64 bits floating points: real and imaginary parts.
DT_QINT8	tf.qint8	8 bits signed integer used in quantized Ops.
DT_QINT32	tf.qint32	32 bits signed integer used in quantized Ops.

So we create a placeholder:

```
a = tf.placeholder(tf.float32)
```

And define a simple multiplication operation:

```
b = a * 2
```

Now we need to define and run the session, but since we created a "hole" in the model to pass the data, when we initialize the session we are obligated to pass an argument with the data, otherwise we would get an error.

To pass the data into the model we call the session with an extra argument **feed_dict** in which we should pass a dictionary with each placeholder name followed by its respective data, just like this:

```
with tf.Session() as sess:
    result = sess.run(b,feed_dict={a:3.5})
    print (result)
```

```
7.0
```

Since data in TensorFlow is passed in form of multidimensional arrays we can pass any kind of tensor through the placeholders to get the answer to the simple multiplication operation:

```
dictionary={a: [ [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ] , [ [13,14,15],[16,17,18],[19,20,21],[22,23,24] ] ] }

with tf.Session() as sess:
    result = sess.run(b,feed_dict=dictionary)
    print (result)
```

```
[[[ 2.  4.  6.]
  [ 8. 10. 12.]
  [14. 16. 18.]
  [20. 22. 24.]]

 [[26. 28. 30.]
  [32. 34. 36.]
  [38. 40. 42.]
  [44. 46. 48.]]]
```

Operations
Operations are nodes that represent the mathematical operations over the tensors on a graph. These operations can be any kind of functions, like add and subtract tensor or maybe an activation function.

tf.constant, **tf.matmul**, **tf.add**, **tf.nn.sigmoid** are some of the operations in TensorFlow. These are like functions in python but operate directly over tensors and each one does a specific thing.

Other operations can be easily found in: https://www.tensorflow.org/versions/r0.9/api_docs/python/index.html

```
graph5 = tf.Graph()
with graph5.as_default():
    a = tf.constant([5])
    b = tf.constant([2])
    c = tf.add(a,b)
    d = tf.subtract(a,b)

with tf.Session(graph = graph5) as sess:
    result = sess.run(c)
    print ('c =: %s' % result)
    result = sess.run(d)
    print ('d =: %s' % result)
```

```
c =: [7]
d =: [3]
```

tf.nn.sigmoid is an activation function, it's a little more complicated, but this function helps learning models to evaluate what kind of information is good or not.

Lab Linear Regression

Linear Regression

Defining a linear regression in simple terms, is the approximation of a linear model used to describe the relationship between two or more variables. In a simple linear regression there are two variables, the dependent variable, which can be seen as the "state" or "final goal" that we study and try to predict, and the independent variables, also known as explanatory variables, which can be seen as the "causes" of the "states".

When more than one independent variable is present the process is called multiple linear regression.
When multiple dependent variables are predicted the process is known as multivariate linear regression.

The equation of a simple linear model is

$$Y = aX + b$$

Where Y is the dependent variable and X is the independent variable, and **a** and **b** being the parameters we adjust. **a** is known as "slope" or "gradient" and **b** is the "intercept". You can interpret this equation as Y being a function of X, or Y being dependent on X.

If you plot the model, you will see it is a line, and by adjusting the "slope" parameter you will change the angle between the line and the independent variable axis, and the "intercept parameter" will affect where it crosses the dependent variable's axis.

Let's first import the required packages:

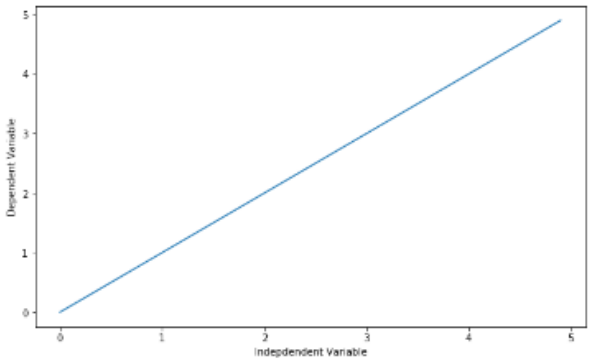
```
import matplotlib.pyplot as plt
import pandas as pd
import pylab as pl
import numpy as np
import tensorflow as tf
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['figure.figsize'] = (10, 6)
```

Let's define the independent variable:

```
X = np.arange(0.0, 5.0, 0.1)
X
```

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. , 1.1, 1.2,
       1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. , 2.1, 2.2, 2.3, 2.4, 2.5,
       2.6, 2.7, 2.8, 2.9, 3. , 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8,
       3.9, 4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9])
```

```
##You can adjust the slope and intercept to verify the changes in the graph
a = 1
b = 0

Y= a * X + b

plt.plot(X, Y)
plt.ylabel('Dependent Variable')
plt.xlabel('Indepdendent Variable')
plt.show()
```



OK... but how can we see this concept of linear relations with a more meaningful point of view?

Simple linear relations were used to try to describe and quantify many observable physical phenomena, the easiest to understand are speed and distance traveled:

$$DistanceTraveled = Speed \times Time + InitialDistance$$

$$Speed = Acceleration \times Time + InitialSpeed$$

They are also used to describe properties of different materials:

$$Force = Deformation \times Stiffness$$

$$HeatTransfered = TemperatureDifference \times ThermalConductivity$$

$$ElectricalTension(Voltage) = ElectricalCurrent \times Resistance$$

$$Mass = Volume \times Density$$

Linear Regression with TensorFlow

A simple example of a linear function can help us understand the basic mechanism behind TensorFlow.
For the first part we will use a sample dataset, and then we'll use TensorFlow to adjust and get the right parameters. We download a dataset that is related to fuel consumption and Carbon dioxide emission of cars.

```
!wget -O FuelConsumption.csv https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/ML0101ENv3/lab
```

```
--2020-05-19 14:42:33--  https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/ML0101ENv3/labs/Fu
elConsumptionCo2.csv
Resolving s3-api.us-geo.objectstorage.softlayer.net (s3-api.us-geo.objectstorage.softlayer.net)... 67.228.254.196
Connecting to s3-api.us-geo.objectstorage.softlayer.net (s3-api.us-geo.objectstorage.softlayer.net)|67.228.254.196|:443... c
onnected.
HTTP request sent, awaiting response... 200 OK
Length: 72629 (71K) [text/csv]
Saving to: 'FuelConsumption.csv'

FuelConsumption.csv 100%[===================>]  70.93K  --.-KB/s    in 0.06s

2020-05-19 14:42:33 (1.13 MB/s) - 'FuelConsumption.csv' saved [72629/72629]
```

Understanding the Data

`FuelConsumption.csv`:

We have downloaded a fuel consumption dataset, **`FuelConsumption.csv`**, which contains model-specific fuel consumption ratings and estimated carbon dioxide emissions for new light-duty vehicles for retail sale in Canada. Dataset source

- **MODELYEAR** e.g. 2014
- **MAKE** e.g. Acura
- **MODEL** e.g. ILX
- **VEHICLE CLASS** e.g. SUV
- **ENGINE SIZE** e.g. 4.7
- **CYLINDERS** e.g 6
- **TRANSMISSION** e.g. A6
- **FUEL CONSUMPTION in CITY(L/100 km)** e.g. 9.9
- **FUEL CONSUMPTION in HWY (L/100 km)** e.g. 8.9
- **FUEL CONSUMPTION COMB (L/100 km)** e.g. 9.2
- **CO2 EMISSIONS (g/km)** e.g. 182 --> low --> 0

```
df = pd.read_csv("FuelConsumption.csv")

# take a look at the dataset
df.head()
```

	MODELYEAR	MAKE	MODEL	VEHICLECLASS	ENGINESIZE	CYLINDERS	TRANSMISSION	FUELTYPE	FUELCONSUMPTION_CITY	FUELCONSUMPTION_HWY
0	2014	ACURA	ILX	COMPACT	2.0	4	AS5	Z	9.9	6.7
1	2014	ACURA	ILX	COMPACT	2.4	4	M6	Z	11.2	7.7
2	2014	ACURA	ILX HYBRID	COMPACT	1.5	4	AV7	Z	6.0	5.8
3	2014	ACURA	MDX 4WD	SUV - SMALL	3.5	6	AS6	Z	12.7	9.1
4	2014	ACURA	RDX AWD	SUV - SMALL	3.5	6	AS6	Z	12.1	8.7

Lets say we want to use linear regression to predict Co2Emission of cars based on their engine size. So, lets define X and Y value for the linear regression, that is, train_x and train_y:

```
train_x = np.asanyarray(df[['ENGINESIZE']])
train_y = np.asanyarray(df[['CO2EMISSIONS']])
```

First, we initialize the variables **a** and **b**, with any random guess, and then we define the linear function:

```
a = tf.Variable(20.0)
b = tf.Variable(30.2)
y = a * train_x + b
```

Now, we are going to define a loss function for our regression, so we can train our model to better fit our data. In a linear regression, we minimize the squared error of the difference between the predicted values(obtained from the equation) and the target values (the data that we have). In other words we want to minimize the square of the predicted values minus the target value. So we define the equation to be minimized as loss.

To find value of our loss, we use **tf.reduce_mean()**. This function finds the mean of a multidimensional tensor, and the result can have a different dimension.

```
loss = tf.reduce_mean(tf.square(y - train_y))
```

Then, we define the optimizer method. The gradient Descent optimizer takes in parameter: learning rate, which corresponds to the speed with which the optimizer should learn; there are pros and cons for increasing the learning-rate parameter, with a high learning rate the training model converges quickly, but there is a risk that a high learning rate causes instability and the model will not converge. **Please feel free to make changes to learning parameter and check its effect**. On the other hand decreasing the learning rate might reduce the convergence speed, but it would increase the chance of converging to a solution. You should note that the solution might not be a global optimal solution as there is a chance that the optimizer will get stuck in a local optimal solution. Please review other material for further information on the optimization. Here we will use a simple gradient descent with a learning rate of 0.05:

```
optimizer = tf.train.GradientDescentOptimizer(0.05)
```

Now we will define the training method of our graph, what method we will use for minimize the loss? We will use the **.minimize()** which will minimize the error function of our optimizer, resulting in a better model.

```
train = optimizer.minimize(loss)
```

Don't forget to initialize the variables before executing a graph:

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

Now we are ready to start the optimization and run the graph:

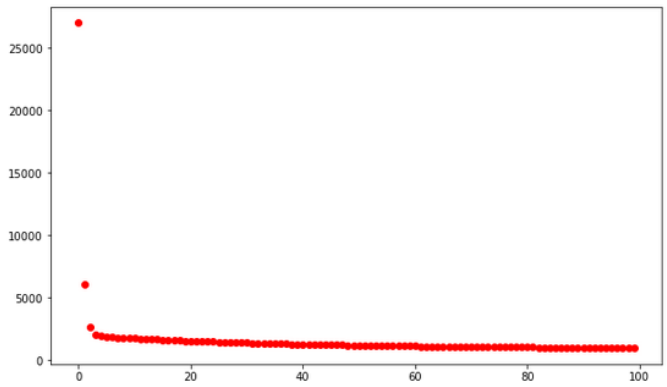
```
loss_values = []
train_data = []
for step in range(100):
    _, loss_val, a_val, b_val = sess.run([train, loss, a, b])
    loss_values.append(loss_val)
    if step % 5 == 0:
        print(step, loss_val, a_val, b_val)
        train_data.append([a_val, b_val])
```

```
0 26992.594 77.07106 46.110275
5 1891.7205 58.84462 47.59573
10 1762.7241 57.65104 53.019833
15 1653.5897 56.36652 58.023922
20 1559.0441 55.172844 62.68204
25 1477.1372 54.061794 67.01765
30 1406.179 53.027664 71.05309
35 1344.7057 52.065136 74.809135
40 1291.4506 51.169243 78.30512
45 1245.3145 50.33538 81.559074
50 1205.3451 49.55925 84.58775
55 1170.7189 48.83685 87.40674
60 1140.7214 48.164467 90.03055
65 1114.734 47.53864 92.472694
70 1092.2203 46.956135 94.74576
75 1072.7163 46.413967 96.86146
80 1055.8193 45.909332 98.83067
```

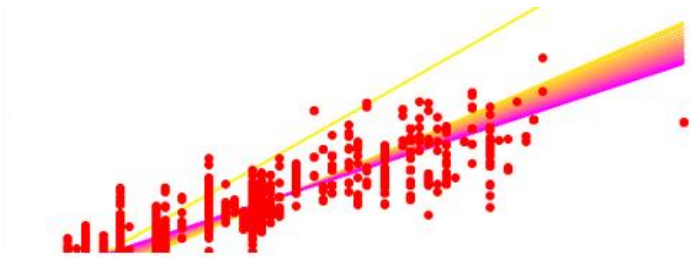
Lets plot the loss values to see how it has changed during the training:

```
plt.plot(loss_values, 'ro')
```

```
[<matplotlib.lines.Line2D at 0x7f32d335cd30>]
```



Lets visualize how the coefficient and intercept of line has changed to fit the data:



Lets visualize how the coefficient and intercept of line has changed to fit the data:

```
cr, cg, cb = (1.0, 1.0, 0.0)
for f in train_data:
    cb += 1.0 / len(train_data)
    cg -= 1.0 / len(train_data)
    if cb > 1.0: cb = 1.0
    if cg < 0.0: cg = 0.0
    [a, b] = f
    f_y = np.vectorize(lambda x: a*x + b)(train_x)
    line = plt.plot(train_x, f_y)
    plt.setp(line, color=(cr,cg,cb))

plt.plot(train_x, train_y, 'ro')


green_line = mpatches.Patch(color='red', label='Data Points')

plt.legend(handles=[green_line])

plt.show()
```

Lab Logistic Regression

Logistic Regression is a variation of Linear Regression, useful when the observed dependent variable, *y*, is categorical. It produces a formula that predicts the probability of the class label as a function of the independent variables.

Despite the name logistic *regression*, it is actually a **probabilistic classification** model. Logistic regression fits a special s-shaped curve by taking the linear regression and transforming the numeric estimate into a probability with the following function:

$$ProbabilityOfaClass = \theta(y) = \frac{e^y}{1+e^y} = exp(y)/(1+exp(y)) = p$$

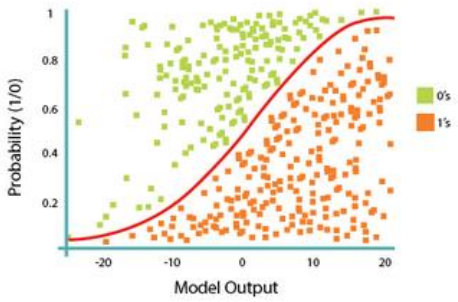
which produces p-values between 0 (as y approaches minus infinity $-\infty$) and 1 (as y approaches plus infinity $+\infty$). This now becomes a special kind of non-linear regression.

In this equation, *y* is the regression result (the sum of the variables weighted by the coefficients), `exp` is the exponential function and $\theta(y)$ is the logistic function, also called logistic curve. It is a common "S" shape (sigmoid curve), and was first developed for modeling population growth.

You might also have seen this function before, in another configuration:

$$ProbabilityOfaClass = \theta(y) = \frac{1}{1+e^{-y}}$$

So, briefly, Logistic Regression passes the input through the logistic/sigmoid function but then treats the result as a probability:



Utilizing Logistic Regression in TensorFlow

For us to utilize Logistic Regression in TensorFlow, we first need to import the required libraries. To do so, you can run the code cell below.

```
import tensorflow as tf
import pandas as pd
import numpy as np
import time
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

Next, we will load the dataset we are going to use. In this case, we are utilizing the `iris` dataset, which is inbuilt -- so there's no need to do any preprocessing and we can jump right into manipulating it. We separate the dataset into *xs* and *ys*, and then into training *xs* and *ys* and testing *xs* and *ys*, (pseudo)randomly.

Understanding the Data

`Iris Dataset`:

This dataset was introduced by British Statistician and Biologist Ronald Fisher, it consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). In total it has 150 records under five attributes - petal length, petal width, sepal length, sepal width and species. Dataset source
Attributes Independent Variable

- petal length
- petal width
- sepal length
- sepal width

Dependent Variable

- Species
  - Iris setosa
  - Iris virginica
  - Iris versicolor

```
iris = load_iris()
iris_X, iris_y = iris.data[:-1,:], iris.target[:-1]
iris_y= pd.get_dummies(iris_y).values
trainX, testX, trainY, testY = train_test_split(iris_X, iris_y, test_size=0.33, random_state=42)
```

Now we define x and y. These placeholders will hold our iris data (both the features and label matrices), and help pass them along to different parts of the algorithm. You can consider placeholders as empty shells into which we insert our data. We also need to give them shapes which correspond to the shape of our data. Later, we will insert data into these placeholders by "feeding" the placeholders the data via a "feed_dict" (Feed Dictionary).

Why use Placeholders?

1. This feature of TensorFlow allows us to create an algorithm which accepts data and knows something about the shape of the data without knowing the amount of data going in.
2. When we insert "batches" of data in training, we can easily adjust how many examples we train on in a single step without changing the entire algorithm.

```
# numFeatures is the number of features in our input data.
# In the iris dataset, this number is '4'.
numFeatures = trainX.shape[1]

# numLabels is the number of classes our data points can be in.
# In the iris dataset, this number is '3'.
numLabels = trainY.shape[1]


# Placeholders
# 'None' means TensorFlow shouldn't expect a fixed number in that dimension
X = tf.placeholder(tf.float32, [None, numFeatures]) # Iris has 4 features, so X is a tensor to hold our data.
yGold = tf.placeholder(tf.float32, [None, numLabels]) # This will be our correct answers matrix for 3 classes.
```

Set model weights and bias

Much like Linear Regression, we need a shared variable weight matrix for Logistic Regression. We initialize both `W` and `b` as tensors full of zeros. Since we are going to learn `W` and `b`, their initial value does not matter too much. These variables are the objects which define the structure of our regression model, and we can save them after they have been trained so we can reuse them later.

We define two TensorFlow variables as our parameters. These variables will hold the weights and biases of our logistic regression and they will be continually updated during training.

Notice that `W` has a shape of [4, 3] because we want to multiply the 4-dimensional input vectors by it to produce 3-dimensional vectors of evidence for the difference classes. `b` has a shape of [3] so we can add it to the output. Moreover, unlike our placeholders above which are essentially empty shells waiting to be fed data, TensorFlow variables need to be initialized with values, e.g. with zeros.

```
W = tf.Variable(tf.zeros([4, 3]))  # 4-dimensional input and  3 classes
b = tf.Variable(tf.zeros([3])) # 3-dimensional output [0,0,1],[0,1,0],[1,0,0]
```

```
#Randomly sample from a normal distribution with standard deviation .01

weights = tf.Variable(tf.random_normal([numFeatures,numLabels],
                                       mean=0,
                                       stddev=0.01,
                                       name="weights"))

bias = tf.Variable(tf.random_normal([1,numLabels],
                                    mean=0,
                                    stddev=0.01,
                                    name="bias"))
```

Logistic Regression model

We now define our operations in order to properly run the Logistic Regression. Logistic regression is typically thought of as a single equation:

$$\hat{y} = sigmoid(WX + b)$$

However, for the sake of clarity, we can have it broken into its three main components:

- a weight times features matrix multiplication operation,
- a summation of the weighted features and a bias term,
- and finally the application of a sigmoid function.

As such, you will find these components defined as three separate operations below.

```
# Three-component breakdown of the Logistic Regression equation.
# Note that these feed into each other.
apply_weights_OP = tf.matmul(X, weights, name="apply_weights")
add_bias_OP = tf.add(apply_weights_OP, bias, name="add_bias")
activation_OP = tf.nn.sigmoid(add_bias_OP, name="activation")
```

As we have seen before, the function we are going to use is the *logistic function* ($\frac{1}{1+e^{-Wx}}$), which is fed the input data after applying weights and bias. In TensorFlow, this function is implemented as the `nn.sigmoid` function. Effectively, this fits the weighted input with bias into a 0-100 percent curve, which is the probability function we want.

Training

The learning algorithm is how we search for the best weight vector ($\mathbf{w}$). This search is an optimization problem looking for the hypothesis that optimizes an error/cost measure.

What tell us our model is bad?
The Cost or Loss of the model, so what we want is to minimize that.

What is the cost function in our model?
The cost function we are going to utilize is the Squared Mean Error loss function.

How to minimize the cost function?
We can't use **least-squares linear regression** here, so we will use gradient descent instead. Specifically, we will use batch gradient descent which calculates the gradient from all data points in the data set.

Cost function

Before defining our cost function, we need to define how long we are going to train and how should we define the learning rate.

```
# Number of Epochs in our training
numEpochs = 700

# Defining our learning rate iterations (decay)
learningRate = tf.train.exponential_decay(learning_rate=0.0008,
                                          global_step= 1,
                                          decay_steps=trainX.shape[0],
                                          decay_rate= 0.95,
                                          staircase=True)
```

```
#Defining our cost function - Squared Mean Error
cost_OP = tf.nn.l2_loss(activation_OP-yGold, name="squared_error_cost")

#Defining our Gradient Descent
training_OP = tf.train.GradientDescentOptimizer(learningRate).minimize(cost_OP)
```

Now we move on to actually running our operations. We will start with the operations involved in the prediction phase (i.e. the logistic regression itself).

First, we need to initialize our weights and biases with zeros or random values via the inbuilt Initialization Op, **tf.initialize_all_variables()**. This Initialization Op will become a node in our computational graph, and when we put the graph into a session, then the Op will run and create the variables.

First, we need to initialize our weights and biases with zeros or random values via the inbuilt Initialization Op, **tf.initialize_all_variables()**. This Initialization Op will become a node in our computational graph, and when we put the graph into a session, then the Op will run and create the variables.

```
# Create a tensorflow session
sess = tf.Session()

# Initialize our weights and biases variables.
init_OP = tf.global_variables_initializer()

# Initialize all tensorflow variables
sess.run(init_OP)
```

We also want some additional operations to keep track of our model's efficiency over time. We can do this like so:

```
# argmax(activation_OP, 1) returns the label with the most probability
# argmax(yGold, 1) is the correct label
correct_predictions_OP = tf.equal(tf.argmax(activation_OP,1),tf.argmax(yGold,1))

# If every false prediction is 0 and every true prediction is 1, the average returns us the accuracy
accuracy_OP = tf.reduce_mean(tf.cast(correct_predictions_OP, "float"))

# Summary op for regression output
activation_summary_OP = tf.summary.histogram("output", activation_OP)

# Summary op for accuracy
accuracy_summary_OP = tf.summary.scalar("accuracy", accuracy_OP)

# Summary op for cost
cost_summary_OP = tf.summary.scalar("cost", cost_OP)
```

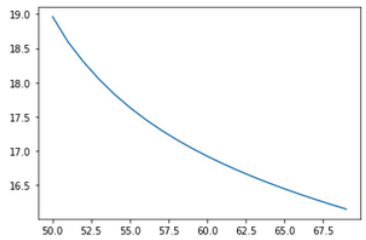
```
# Summary ops to check how variables (W, b) are updating after each iteration
weightSummary = tf.summary.histogram("weights", weights.eval(session=sess))
biasSummary = tf.summary.histogram("biases", bias.eval(session=sess))

# Merge all summaries
merged = tf.summary.merge([activation_summary_OP, accuracy_summary_OP, cost_summary_OP, weightSummary, biasSummary])

# Summary writer
writer = tf.summary.FileWriter("summary_logs", sess.graph)
```

Now we can define and run the actual training loop, like this:

```
# Initialize reporting variables
cost = 0
diff = 1
epoch_values = []
accuracy_values = []
cost_values = []

# Training epochs
for i in range(numEpochs):
    if i > 1 and diff < .0001:
        print("change in cost %g; convergence."%diff)
        break
    else:
        # Run training step
        step = sess.run(training_OP, feed_dict={X: trainX, yGold: trainY})
        # Report occasional stats
        if i % 10 == 0:
            # Add epoch to epoch_values
            epoch_values.append(i)
            # Generate accuracy stats on test data
            train_accuracy, newCost = sess.run([accuracy_OP, cost_OP], feed_dict={X: trainX, yGold: trainY})
            # Add accuracy to live graphing variable
            accuracy_values.append(train_accuracy)
            # Add cost to live graphing variable
            cost_values.append(newCost)
            # Re-assign values for variables
            diff = abs(newCost - cost)
            cost = newCost

            #generate print statements
            print("step %d, training accuracy %g, cost %g, change in cost %g"%(i, train_accuracy, newCost, diff))


# How well do we perform on held-out test data?
print("final accuracy on test set: %s" %str(sess.run(accuracy_OP,
                                                     feed_dict={X: testX,
                                                                yGold: testY})))
```

Why don't we plot the cost to see how it behaves?

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.plot([np.mean(cost_values[i-50:i]) for i in range(len(cost_values))])
plt.show()
```



Assuming no parameters were changed, you should reach a peak accuracy of 90% at the end of training, which is commendable. Try changing the parameters such as the length of training, and maybe some operations to see how the model behaves. Does it take much longer? How is the performance?

Lab Activation Functions

Importing Dependencies

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import Axes3D

%matplotlib inline
```

The next cell implements a basic function that plots a surface for an arbitrary activation function. The plot is done for all possible values of weight and bias between -0.5 and 0.5 with a step of 0.05. The input, the weight, and the bias are one-dimensional. Additionally, the input can be passed as an argument.

The next cell implements a basic function that plots a surface for an arbitrary activation function. The plot is done for all possible values of weight and bias between -0.5 and 0.5 with a step of 0.05. The input, the weight, and the bias are one-dimensional. Additionally, the input can be passed as an argument.

```
def plot_act(i=1.0, actfunc=lambda x: x):
    ws = np.arange(-0.5, 0.5, 0.05)
    bs = np.arange(-0.5, 0.5, 0.05)

    X, Y = np.meshgrid(ws, bs)

    os = np.array([actfunc(tf.constant(w*i + b)).eval(session=sess) \
                   for w,b in zip(np.ravel(X), np.ravel(Y))])

    Z = os.reshape(X.shape)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(X, Y, Z, rstride=1, cstride=1)
```

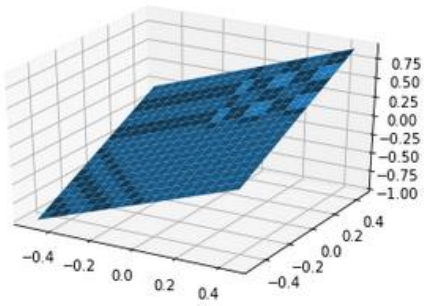
Basic Structure

In this example we illustrate how, in Tensorflow, to compute the weighted sum that goes into the neuron and direct it to the activation function. For further details, read the code comments below.

```
#start a session
sess = tf.Session();
#create a simple input of 3 real values
i = tf.constant([1.0, 2.0, 3.0], shape=[1, 3])
#create a matrix of weights
w = tf.random_normal(shape=[3, 3])
#create a vector of biases
b = tf.random_normal(shape=[1, 3])
#dummy activation function
def func(x): return x
#tf.matmul will multiply the input(i) tensor and the weight(w) tensor then sum the result with the bias(b) tensor.
act = func(tf.matmul(i, w) + b)
#Evaluate the tensor to a numpy array
act.eval(session=sess)
```

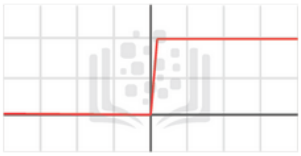
```
array([[0.6827395, 3.0230877, 3.241613 ]], dtype=float32)
```

```
plot_act(1.0, func)
```



The Step Functions ¶

The Step function was the first one designed for Machine Learning algorithms. It consists of a simple threshold function that varies the Y value from 0 to 1. This function has been historically utilized for classification problems, like Logistic Regression with two classes.



The Step Function simply functions as a limiter. Every input that goes through this function will be applied to gets either assigned a value of 0 or 1. As such, it is easy to see how it can be handy in classification problems.

There are other variations of the Step Function such as the Rectangle Step and others, but those are seldom used.

Tensorflow dosen't have a Step Function.

The Sigmoid Functions

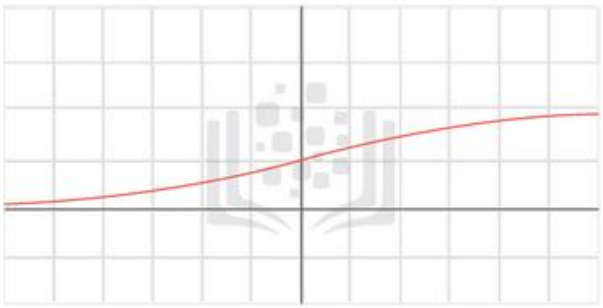
The next in line for Machine Learning problems is the family of the ever-present Sigmoid functions. Sigmoid functions are called that due to their shape in the Cartesian plane, which resembles an "S" shape.

Sigmoid functions are very useful in the sense that they "squash" their given inputs into a bounded interval. This is exceptionally handy when combining these functions with others such as the Step function.

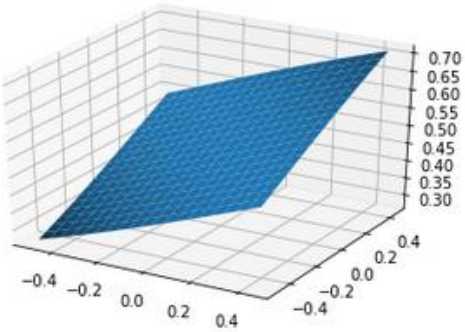
Most of the Sigmoid functions you should find in applications will be the Logistic, Arctangent, and Hyperbolic Tangent functions.

Logistic Regression (sigmoid)

The Logistic function, as its name implies, is widely used in Logistic Regression. It is defined as $f(x) = \frac{1}{1+e^{-x}}$. Effectively, this makes it so you have a Sigmoid over the $(0, 1)$ interval, like so:



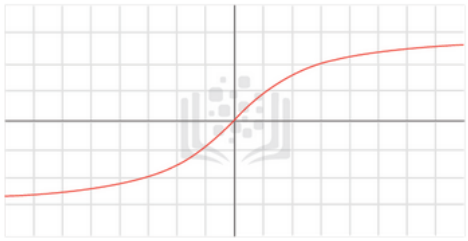
```
]: plot_act(1, tf.sigmoid)
```



Using sigmoid in a neural net layer

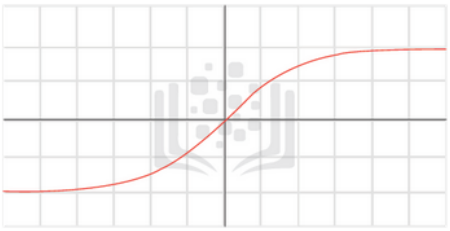
```
: act = tf.sigmoid(tf.matmul(i, w) + b)
act.eval(session=sess)
```

The Arctangent and Hyperbolic Tangent functions on the other hand, as the name implies, are based on the Tangent function. Arctangent is defined by $f(x) = tan^{-1}x$, and produces a sigmoid over the $(\frac{-\pi}{2}, \frac{\pi}{2})$ interval.

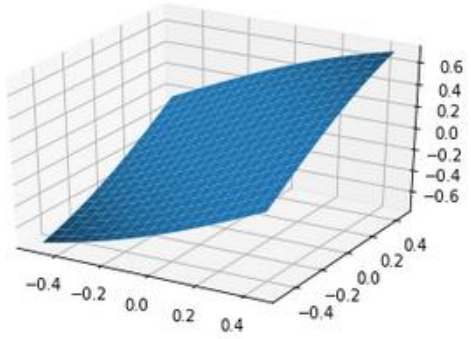


It has no implementation in Tensorflow

The Hyperbolic Tangent, or TanH as it's usually called, is defined as $f(x) = \frac{2}{1+e^{-2x}} - 1$. It produces a sigmoid over the $(-1, 1)$ interval. TanH is widely used in a wide range of applications, and is probably the most used function of the Sigmoid family.



```
plot_act(1, tf.tanh)
```



3D tanh plot. The x-axis is the weight, the y-axis is the bias.

Using tanh in a neural net layer

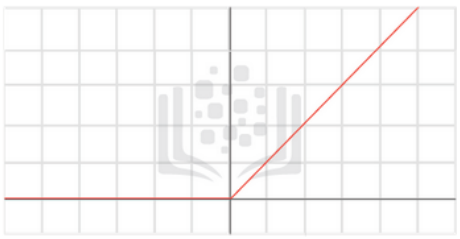
```
act = tf.tanh(tf.matmul(i, w) + b)
act.eval(session=sess)
```

```
array([[-0.8726205 , -0.99384177, -0.95158195]], dtype=float32)
```

The Linear Unit functions

Linear Units are the next step in activation functions. They take concepts from both Step and Sigmoid functions and behave within the best of the two types of functions. Linear Units in general tend to be variation of what is called the Rectified Linear Unit, or ReLU for short.

The ReLU is a simple function which operates within the $[0, \infty)$ interval. For the entirety of the negative value domain, it returns a value of 0, while on the positive value domain, it returns x for any $f(x)$.



While it may seem counterintuitive to utilize a pseudo-linear function instead of something like Sigmoids, ReLUs provide some benefits which might not be understood at first glance. For example, during the initialization process of a Neural Network model, in which weights are distributed at random for each unit, ReLUs will only activate approximately only in 50% of the times -- which saves some processing power. Additionally, the ReLU structure takes care of what is called the **Vanishing and Exploding Gradient** problem by itself. Another benefit -- if not only marginally relevant to us -- is that this kind of activation function is directly relatable to the nervous system analogy of Neural Networks (this is called *Biological Plausibility*).

The ReLU structure has also has many variations optimized for certain applications, but those are implemented on a case-by-case basis and therefore aren't in the scope of this notebook. If you want to know more, search for *Parametric Rectified Linear Units* or maybe *Exponential Linear Units*.

```
plot_act(1, tf.nn.relu)
```

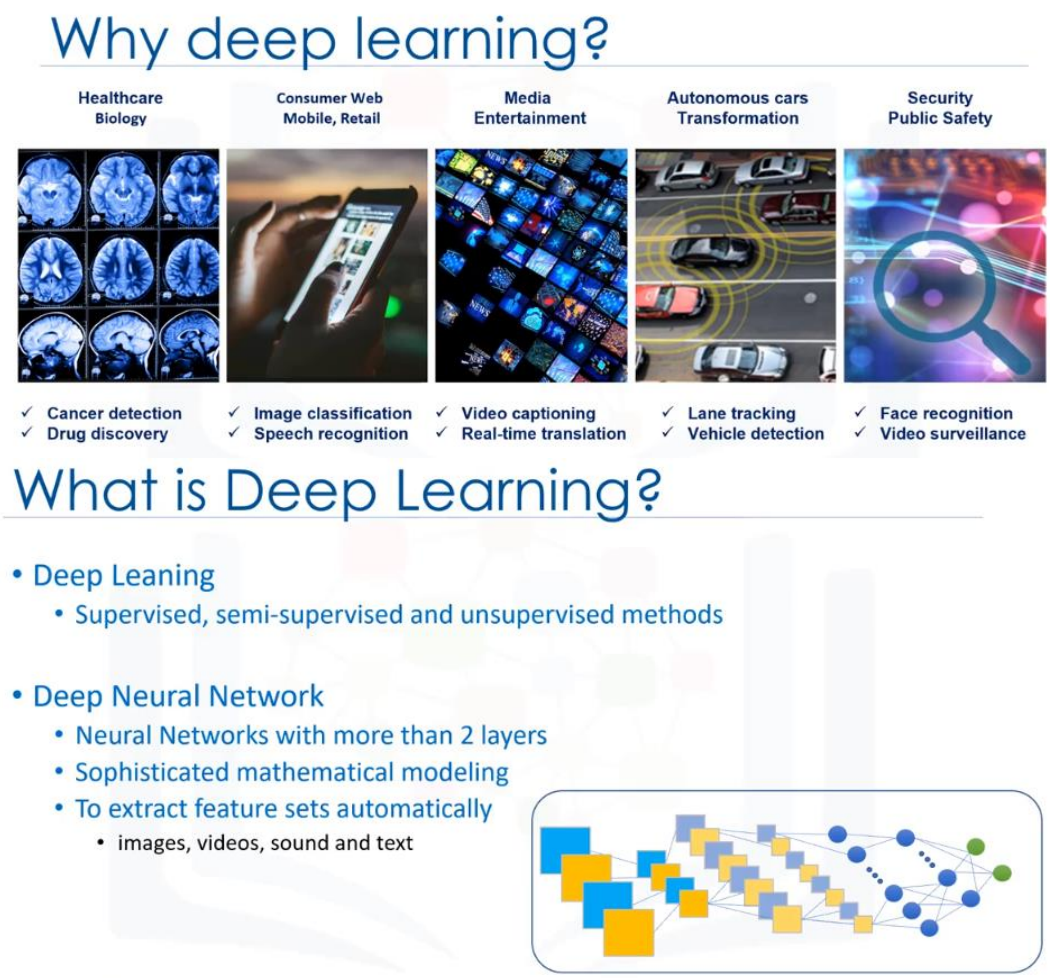
3D relu plot. The x-axis is the weight, the y-axis is the bias.

Using relu in a neural net layer

TensorFlow has ReLU and some other variants of this function. Take a look:

```
act = tf.nn.relu(tf.matmul(i, w) + b)
act.eval(session=sess)
```

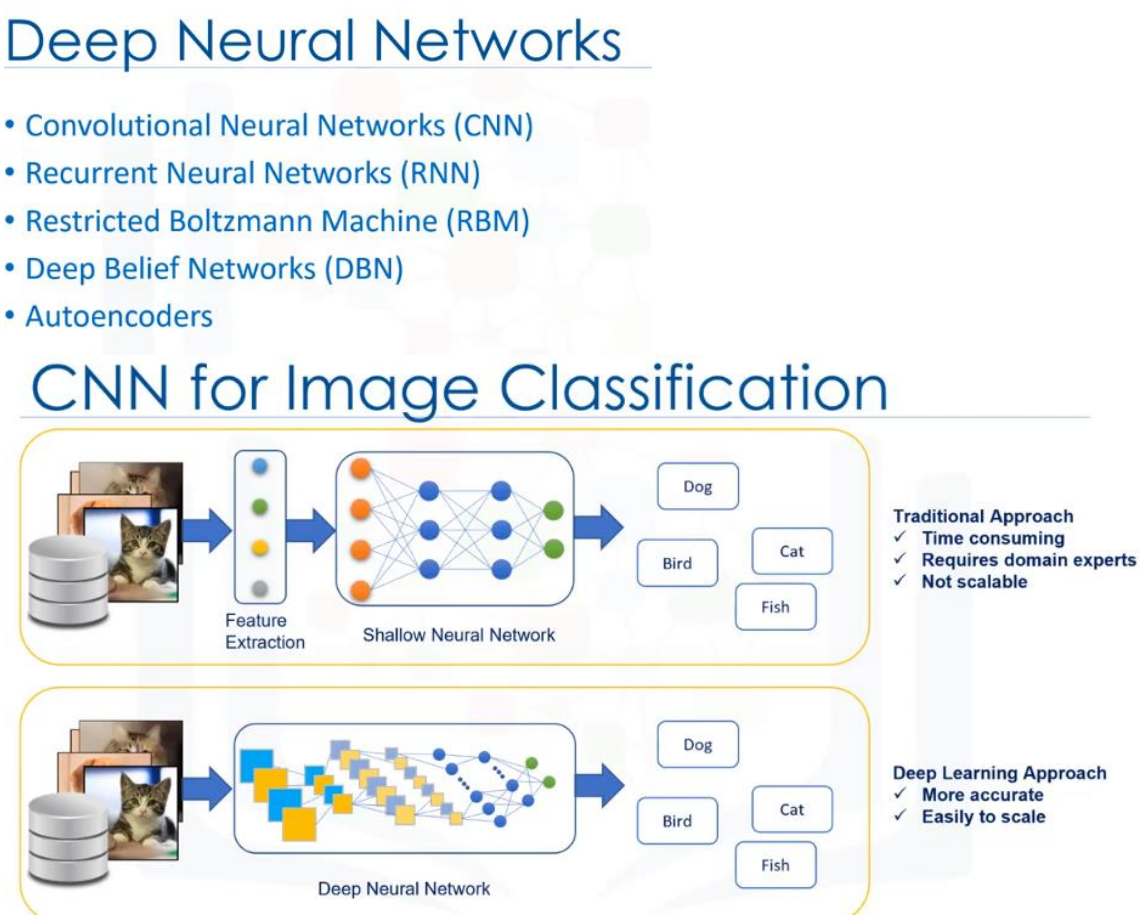
INTRO DEEP LEARNING



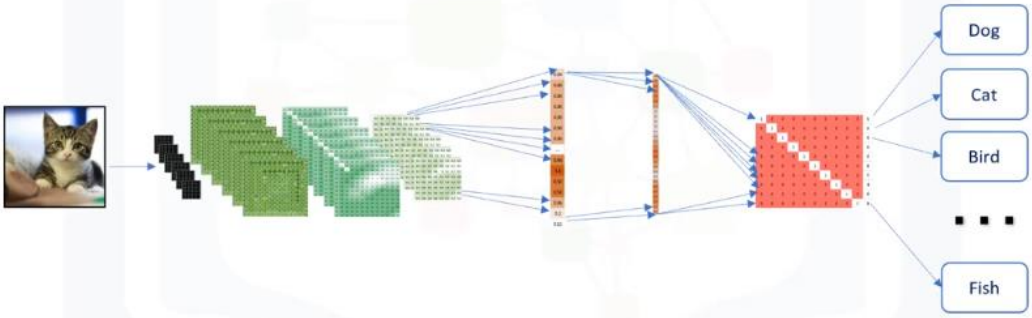
Hello, and welcome! In this video, we will provide an introduction to Deep Learning, and see why it is such a hot topic today.
It is not an overstatement to say that Deep learning can be found all around us, as it is used in countless ways across all industries. For example, Deep Learning is trying to help the health care industry on tasks such as cancer detection and drug discovery.
In the internet service and mobile phone industries, we can see various apps which are using Deep
Learning for image/video classification and speech recognition, such as, Google Voice, Apple Siri, Microsoft Skype, and so on. In media, entertainment, and news, we can see applications such as video captioning, real-time translation and personalization, or recommendation systems such as Netflix. In the development of self-driving cars, Deep Learning is helping researchers to overcome significant research problems, such as sign and passenger detection or lane tracking. In the Security field, Deep Learning is used for face recognition and video surveillance. These are just a few examples of the industries

in which Deep Learning is being applied; and it is being used in many other fields and domains as well.
The increasing popularity of Deep Learning today comes from three recent advances in our field: First, in the dramatic increases in computer processing capabilities; second, in the availability of massive amounts of data for training computer systems; and third, in the advances in machine learning algorithms and research.
So, what is deep learning? Deep Learning is a series of supervised, semi-supervised and unsupervised methods that try to solve some machine learning problems using deep neural networks. A deep neural network is a neural network which often has more than two layers, and uses specific mathematical modeling in each layer to process data. Generally speaking, these networks are trying to automatically extract feature sets from the data, and this is why, they are mostly used in data types where the feature selection process is difficult, such as when analyzing unstructured datasets, such as image data, videos, sound and text.
Thanks for watching.
End of transcript. Skip to the start.

DEEP NEURAL NETWORKS

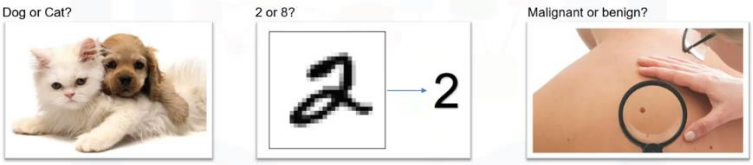


CONVOLUTIONAL NEURAL NETWORKS



CNN Applications

Image Recognition/Classification



CNN Applications

Object Detection in images

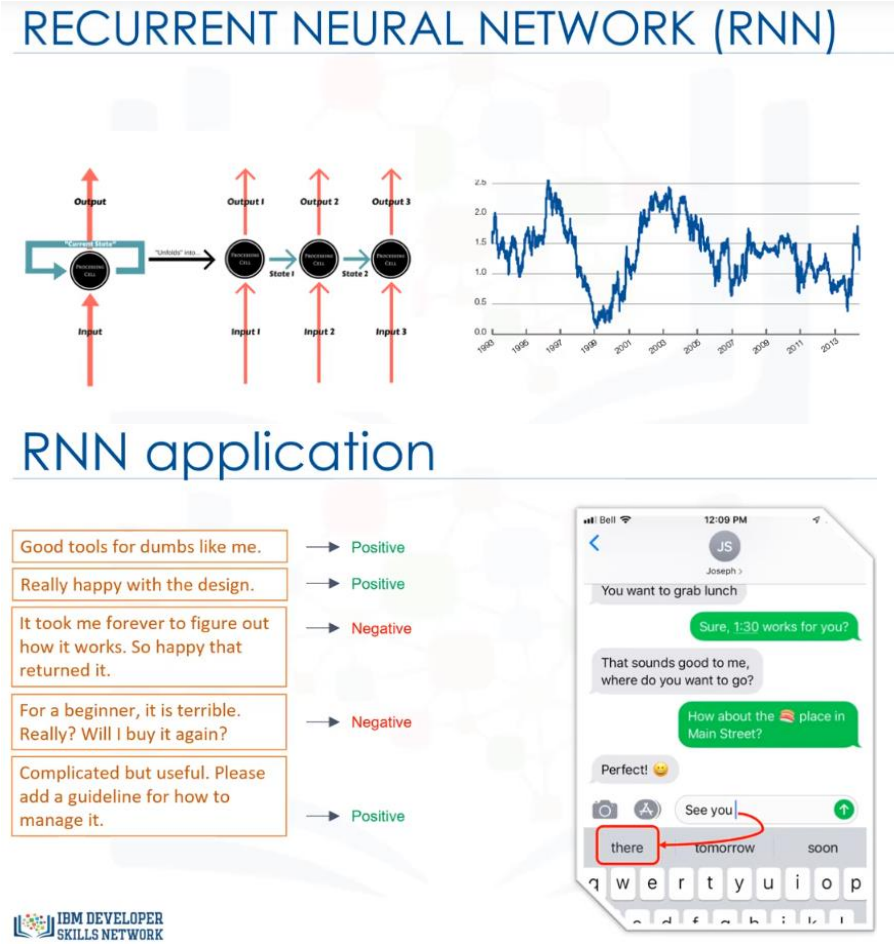
self-driving vehicles

IBM DEVELOPER

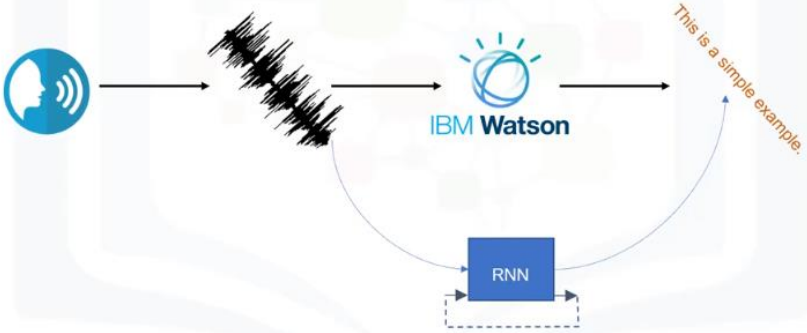
CNN Applications

Coloring black and white images

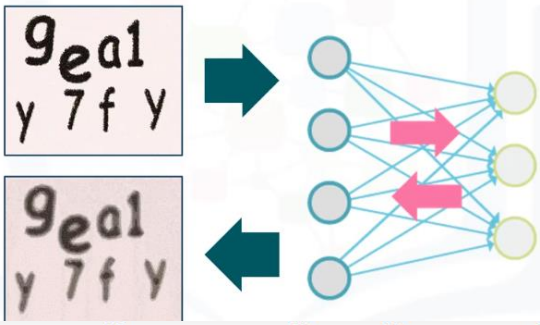
Creating art images


RECURRENT NEURAL NETWORK (RNN)
Output
Output 1
Output 2
Output 3
State 1
State 2
Input
Input 1
Input 2
Input 3
RNN application
Good tools for dumbs like me.
Positive
Really happy with the design.
Positive
It took me forever to figure out how it works. So happy that returned it.
Negative
For a beginner, it is terrible. Really? Will I buy it again?
Negative
Complicated but useful. Please add a guideline for how to manage it.
Positive
12:09 PM
JS
Joseph
You want to grab lunch
Sure, 1:30 works for you?
That sounds good to me, where do you want to go?
How about the place in Main Street?
Perfect!
See you
there
tomorrow
soon
IBM DEVELOPER SKILLS NETWORK
RNN application
Google
Translate
Turn off instant translation
English French Persian Detect language
French Persian English
Translate
This is a very simple example.
30/5000
Ceci est un exemple très simple.
RNN

RNN application



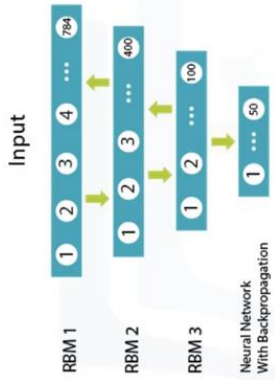
Restricted Boltzmann Machines (RBMs)



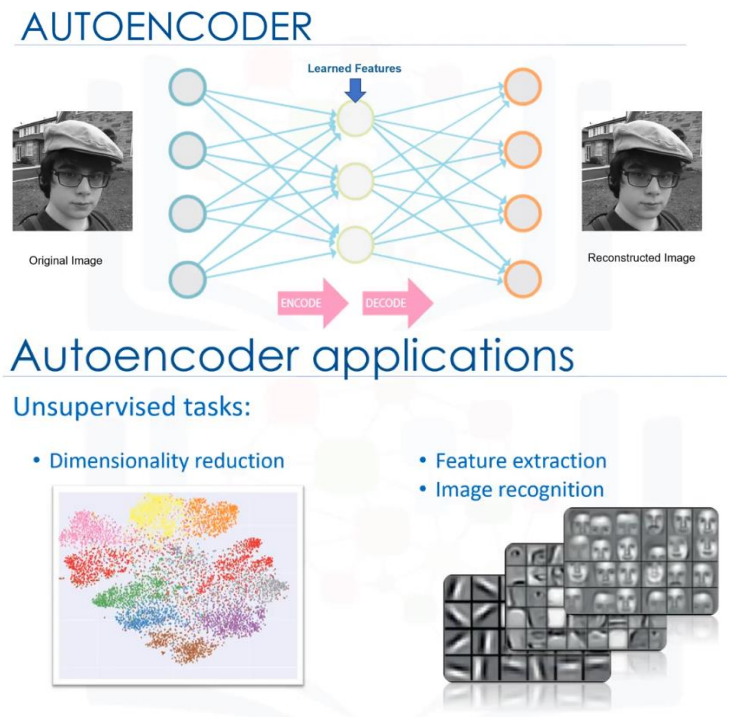
What are the applications of RBM?

- **RBM as building blocks of other networks (e.g. DBN)**
- Feature extraction/learning
- Dimensionality reduction
- Pattern recognition
- Recommender systems (Collaborative Filtering)
- Handling missing values

Deep Belief Network (DBN)



- Classification
  - Image recognition
- Very accurate discriminative classifier
  - Using a stack of RBMs



.

Hello, and welcome! In this video, we will provide an overview of several deep neural network models, and their applications.

To better understand Deep Learning, let's first take a look at different deep neural networks and their applications, namely: • Convolutional Neural Networks (or CNNs) • Recurrent Neural Networks (or RNNs) • Restricted Boltzmann Machines (or RBMs) • Deep Belief Networks (or DBNs), and finally • Autoencoders.

Let's start with the Convolutional Neural Network, and see how it helps us to do a task, such as image classification.

Assume that you have a dataset made up of a great many photos of cats and dogs, and you want to build a model that can recognize and differentiate them.

This model is supposed to look at this particular sample set of images and learn from them, toward becoming trained. Later, given an unseen image of either a cat or a dog, we should be able to use this trained model to recognize the image as being one or the other.

Traditionally, your first step in building such a model would be "feature extraction." That is, to choose the best features from your images, and then use those features in a classification algorithm, such as a shallow Neural Network.

Ideally, the result would be a model that, upon analyzing a new image, could accurately distinguish the animal in that photo as being either a "cat" or a "dog."

There are countless features that could be extracted from the image, such as color, object edges, pixel location, and so on. Of course, the better that you can define the feature sets, the more accurate and efficient your image-classification will be.

In fact, in the last two decades, there has been a lot of scientific research in the field of image processing, that is devoted to helping data scientists find the best feature sets from images, for the purposes of classification. However, as you can imagine, the process of selecting and using the best features is tremendously time-consuming and is often ineffective.

Furthermore, extending the features to other types of images becomes an even greater challenge.

Indeed, it's easy to see how difficult it would be to apply the features we've used

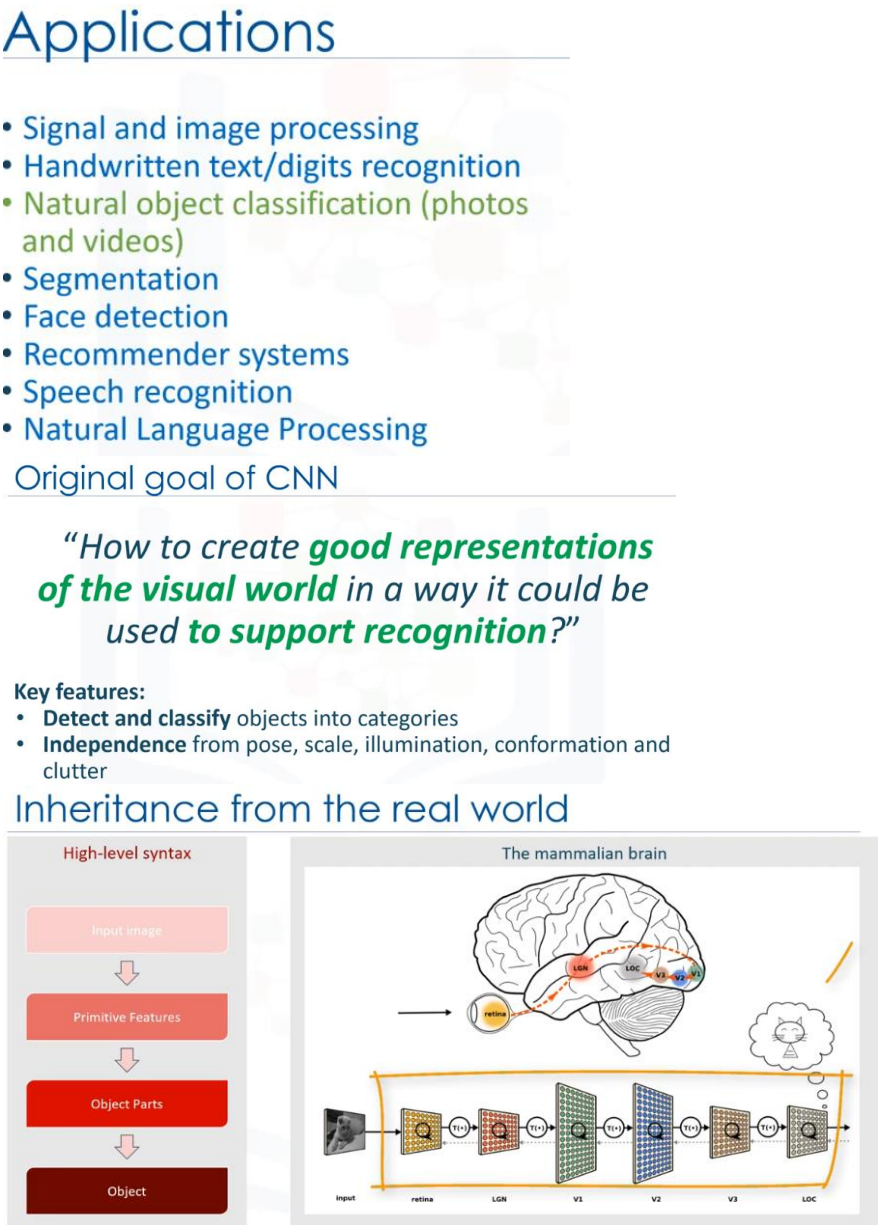
to discriminate cats and dogs, to other discrimination tasks, such as recognizing hand-written digits,
for example. Therefore, the importance of feature selection
can't be overstated. Enter "convolutional neural networks".
Suddenly, without having to find or select features, this network automatically and effectively
finds the best features for you. So instead of you choosing what image features
to use in classifying dogs vs. cats, Convolutional Neural Networks can automatically find those
features and classify the images for you. So, we can say that a Convolutional Neural Network - or CNN for short - is a deep learning approach that learns directly from samples in a way that is much more effective than traditional Neural networks.
CNNs achieve this type of automatic feature selection and classification through multiple specific layers of sophisticated mathematical operations.
Through multiple layers, a CNN learns multiple levels of feature sets at different levels of abstraction. And this leads to very effective classification.
CNNs have gained a lot of attention in the machine learning community over the last few years. This is due to the wide range of applications
where CNNs excel, especially machine vision projects, including image recognition or classification,
such as distinguishing animal photos or digit recognition, to skin cancer classification.
CNNs are also used in object detection, for example real-time recognition of passengers in images captured by self-driving cars.
Or coloring black and white images, and creating art images.
Now, let's look at Recurrent Neural networks and the types of situations in which they can be used, to solve a problem.
A Recurrent Neural Network, or RNN for short, is a type of deep learning approach, that tries to solve the problem of modeling sequential data.
Whenever the points in a dataset are dependent on the previous points, the data is said to be sequential. For example, a stock market price is a sequential
type of data because the price of any given stock in tomorrow's market, to a great extent, depends on its price today. As such, predicting the stock price tomorrow,
is something that RNNs can be used for. We simply need to feed the network with the sequential data, it then maintains the context of the data and thus, learns the patterns within the data.
We can also use RNNs for sentiment analysis. Let's say for example, you're scrolling
through your product catalogue on a social network site and you see many comments related
to a particular product of yours. Rather than reading through dozens and dozens of comments yourself and having to manually calculate if they were mostly positive, you can let an RNN do that for you. Indeed, an RNN can examine the sentiment of keywords in those reviews. Please remember, though, that the sequence
of the words or sentences, as well as the context in which they are used, is very important as well. By feeding a sentence into an RNN, it takes
all of this into account and determines if it the sentiment within it those product reviews are positive or negative. RNNs can also be used to predict the next
word in a sentence. I'm sure we've all seen how our mobile
phone suggests words when we're typing an email or a text.
This is a type of language modeling within RNN, where the model has learned from a big textual corpus, and now can predict the next word in the sentence.
As you can see, thinking in a sequential way, the word being suggested is very dependent on the previously typed words and the context of that message.

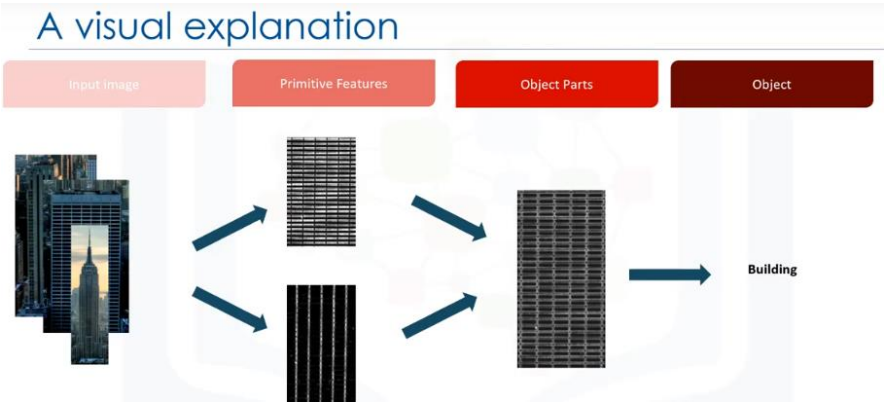
When needing quick translation of certain words into another language, a great many people today, use the translation service of Google translator.
We enter a sequence of words in English and it outputs a sequence of the words in French, as seen here. This type of text translation is another example
of how RNNs can be used. This task is not based on a word-by-word translation
and applying grammar rules. Instead, it is a probability model that has
been trained on lots of data where the exact same text is translated into another language.
Speech-to-text is yet another useful and increasingly common application of RNNs.
In this case, the recognized voice is not only based on the word sound; RNNs also use the context around that sound to accurately recognize of the words being spoken into the device's microphone.
Now, let's look at another type of neural network called a Restricted Boltzman Machine.
Restricted Boltzman Machines, or RBMs, are used to find the patterns in data in an unsupervised
manner. They are shallow neural nets that learn to
reconstruct data by themselves. They are very important models, because they
can automatically extract meaningful features from a given input, without the need to label them.
RBMs might not be outstanding if you look at them as independent networks, but they are significant as building blocks of other networks, such as Deep Belief Networks.
Essentially, RBMs are useful for unsupervised tasks such as:
• feature extraction, • dimensionality reduction,
• pattern recognition, • recommender systems,
• handling missing values, and • topic modeling.
Now let's look at Deep Belief Networks and see how they are built on top of RBMs.
A Deep Belief Network is a network that was invented to solve an old problem in traditional artificial neural networks. Which problem?
The back-propagation problem, that can often cause "local minima" or "vanishing gradients" issues in the learning process. A DBN is built to solve this by the stacking
of multiple RBMs. So, what are the applications of DBNs?
DBNs are generally used for classification -- same as traditional MLPs.
So, one of the most important applications of DBNs is image recognition.
The important part to remember, here, is that a DBN is a very accurate discriminative classifier.
As such, we don't need a big set of labeled data to train a Deep Belief Network; in fact, a small set works fine because feature extraction is unsupervised by a stack of RBMs.
Now, let's look at Autoencoders.
Much like RBMs, Autoencoders were invented to address the issue of extracting desirable features. And again, much like RBMs, Autoencoders try
to recreate a given input, but do so with a slightly different network architecture
and learning method. Autoencoders take a set of unlabeled inputs,
encodes them into short codes, and then uses those to reconstruct the original image, while extracting the most valuable information from the data.
What are the applications of Autoencoders? Well, Autoencoders are employed in some of the largest deep learning applications, especially for unsupervised tasks.
As the encoder part of the network, Autoencoders compress data from the input layer into a short code -- a method that can be used for "dimensionality reduction" tasks.
Also, in stacking multiple Autoencoder layers, the network learns multiple levels of representation
at different levels of abstraction. For example, to detect a face in an image,
the network encodes the primitive features, like the edges of a face.
Then, the first layer's output goes to the second Autoencoder, to encode the less local

features, like the nose, and so on. Therefore, it can be used for Feature Extraction and image recognition. By now, you should have a good sense of deep neural networks.
This concludes this video … Thanks for watching.
End of transcript. Skip to the start.

MODULE 2 - CONVOLUTIONAL NETWORKS

INTRODUCTION

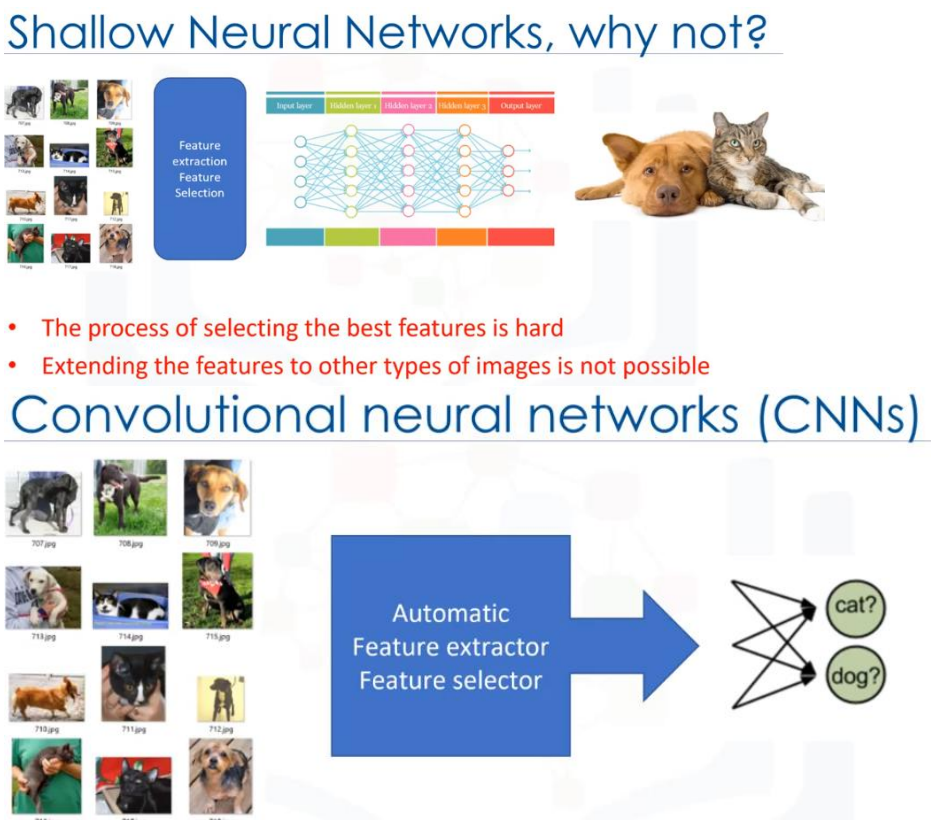




tart of transcript. Skip to the end.
Hello, and welcome! In this video, we'll be reviewing a brief introduction to the convolutional neural network model.
Convolutional neural networks, or CNNs, have gained a lot of attention in the machine learning community over the last few years. This is due to the wide range of applications where CNNs excel, such as object detection and speech recognition.
In a later part of this video, we'll examine the object recognition problem, which is a key consideration when using CNNs. You'll also get an idea of how a CNN's structure allows it to extract the elements from an image.
For example, you'll see how a CNN learns to pick out a person, or a dog, or even chairs from an image, even if the chairs are only partially visible. Indeed, this is not an easy task for computers, and it took years of dedicated research to achieve this.
The original goal of the CNN was to form the best possible representation of our visual world, in order to support recognition tasks. The CNN solution needed to have two key features
to make it viable: First, it needed to be able to detect the objects in the image and place them into the appropriate category.
And second, it also needed to be robust against differences in pose, scale, illumination, conformation, and clutter. The importance of this second feature simply cannot be overstated, since this was historically a huge limitation of hand-coded algorithms.
Interestingly enough, the solution to the object recognition issue was inspired by examining the way our own visual cortex operates. In short, the CNN starts with an input image, it then extracts a few primitive features, and combines those features to form certain parts of the object; and finally, it pulls together all of the various parts to form the object itself. In essence, it is a hierarchical way of seeing objects. That is, in the first layer, very simple features
are detected. Then, these are combined to shape more complicated features in the second layer, and so on to detect things like a cat or a dog or any other object we're needing to detect. The Convolutional Neural Network was developed from this sequence of steps.
Let's consider the building in the lower right hand corner of this image. How can a trained CNN learn and recognize that it's an image of building, rather than some of the other elements that are situated in the photograph, such as a person, or even those clouds in the sky?
Well, in the training phase, CNNs would receive many building images as input.
Then, CNNs will automatically find that the best primitive features for a building would be things like horizontal and vertical lines. Much like the neurons firing in a person's visual cortex, the first layer in a CNN reacts, so to speak, when it receives an image that has horizontal or vertical lines. Once these simpler features, such as lines,

are combined, CNNs learn to form higher abstract components, like the windows of the building
and the building's external shape. It can then use these basic parts to form
the complete object, and learn how the entire building essentially looks like.
Later, if it sees an image of a building that it hasn't seen before, CNNs can make a decision if the new input image is of a building, based on the persistence of the various features it has stored. It is the same process for learning and detecting
other objects, such as faces, animals, cars, and so on.
So, as you can see, the CNN, is a set of layers, with each of them being responsible to detect a set of feature sets. And these features are going to be more abstract as it goes further into examining the next layer. So at this point, you should have a basic
understanding of the principles behind a convolutional neural network, and how CNNs might be used
in an application.
Thanks for watching.
End of transcript. Skip to the start.

CNN FOR CLASSIFICATION

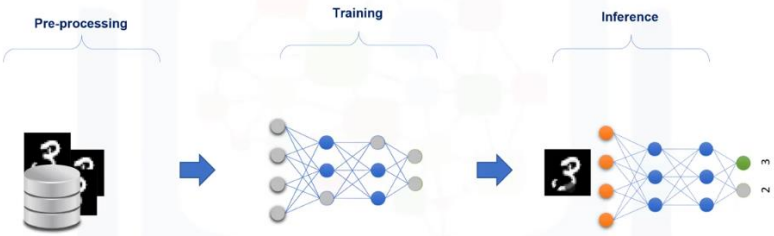


Datasets

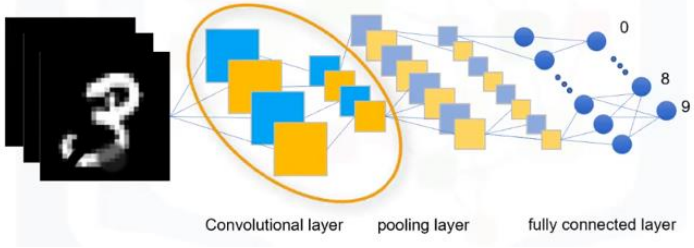
- MNIST Dataset: database of handwritten digits

Digit Recognition

Training and Inference



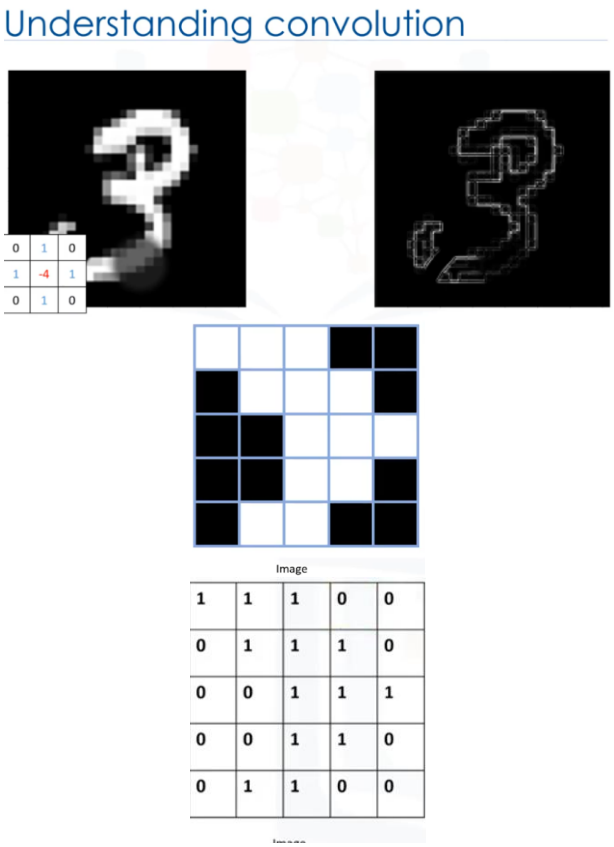
Digit Recognition

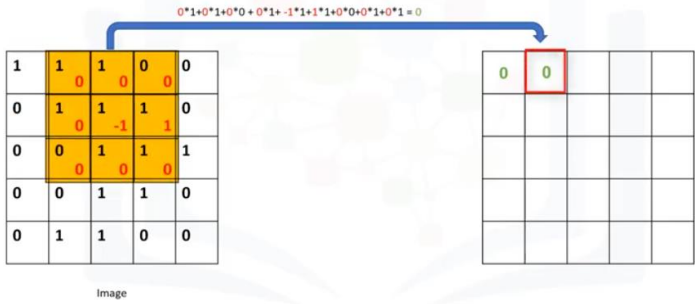


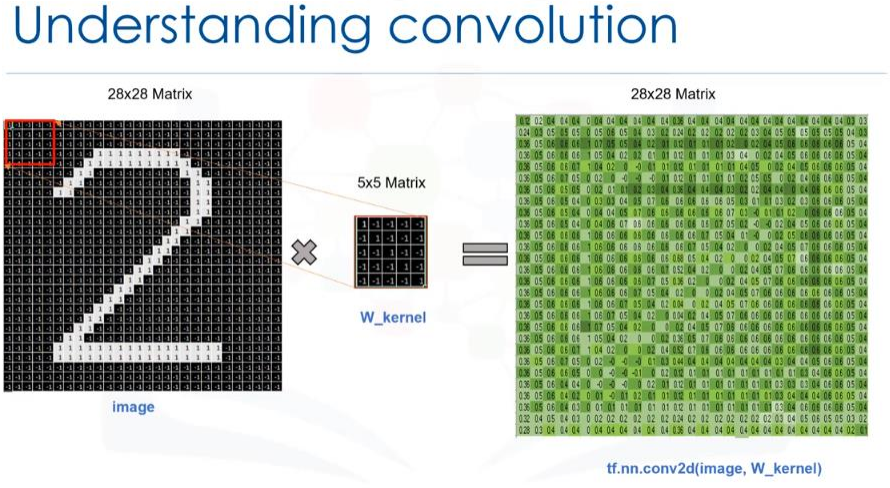
Skip to main content
Cognitive Class Home Page
CognitiveClass: ML0120ENv2 Deep Learning with TensorFlow
Courses
Help
Dashboard for: rafasolis1984

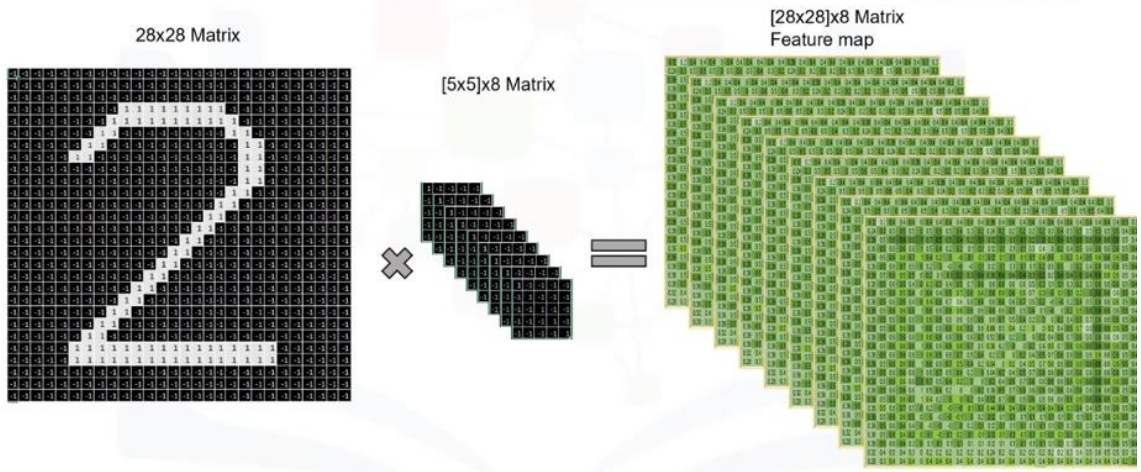
Start of transcript. Skip to the end.
Hello, and welcome! In this video, we'll explain how CNN can
solve an image classification problem such as digit recognition.
First, let's see why traditional shallow neural networks do not work as well as CNN.
As mentioned before, one of the important steps in the modeling for classification with
Shallow Neural Networks, is the feature extraction step.
These chosen features could simply be the color, object edges, pixel location, or countless
other features that could be extracted from the images. Of course, the better and more
effective the feature sets you find, the more accurate and efficient the image classification
you can obtain. However, as you can imagine, the process of
selecting the best features is tremendously time-consuming, and is often ineffective.
Also, extending the features to other types of images becomes an even greater problem.
Convolutional neural networks (or CNNs) try to solve this problem by using more hidden
layers, and also with more specific layers. So, when using CNN, instead of you choosing
image features, to classify dogs vs. cats, for instance, CNNs can automatically find
those features and classify the images for you.
Instead of using a dogs and cats dataset, though, let's use a more practical example
here. The MNIST dataset is a "database of handwritten
digits that has a training set of 60,000 examples. The great thing about the MNIST dataset is
that the digits have been size-normalized and centered in a fixed-size image".
Now, consider the digit recognition problem. We would like to classify images of
handwritten
numbers, where the observations are the intensity of the pixels. And, the target will be a
digit,
from 0 to 9. So, our objective here is to build a digit recognition system using CNN.
Now let's look at it from a higher level. Basically, if we look at the pipeline of our
deep learning process, we can see the following phases:
First, pre-processing of input data. Second, training the deep learning model.
And third, inference and deployment of the model.
In the first part, we have to convert the images to a readable and proper format for
our network. Then, an untrained network is fed with a big
dataset of the images in the Training phase, so as to allow the network to learn. That is,
we build a CNN, and train it with many hand-written images in the training set.
Finally, we use the trained model in the Inference phase, which classifies a new image by
inferring
its similarity to the trained model. So, this model can be deployed and used as
a digit recognition model for unseen cases.
Now let's focus on the training process. As mentioned before, a deep neural network
not only has multiple hidden layers, the type of layers and their connectivity also is different
from a shallow neural network, in that it usually has multiple Convolutional layers,
pooling layers, as well as fully connected layers.
The convolutional layer applies a convolution operation to the input, and passes the result
to the next layer. The pooling layer combines the outputs of
a cluster of neurons in the previous layer into a single neuron in the next layer.
And then, the fully connected layers connect every neuron in the previous layer to every
neuron in the next layer. In the next video, you will learn more about
these layers and their specifications, in detail.
Thanks for watching.
End of transcript. Skip to the start.

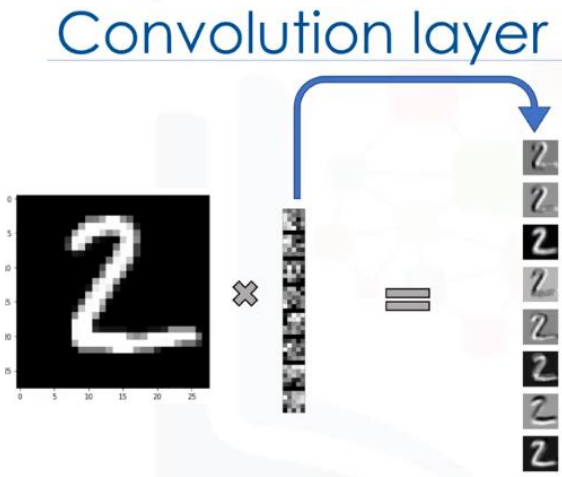
CNN ARCHITECTURE

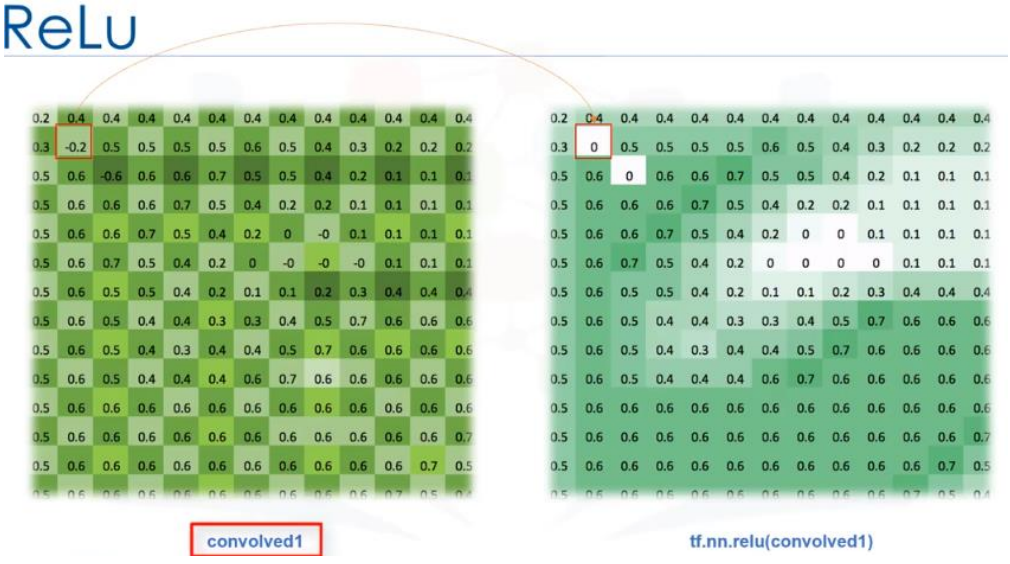


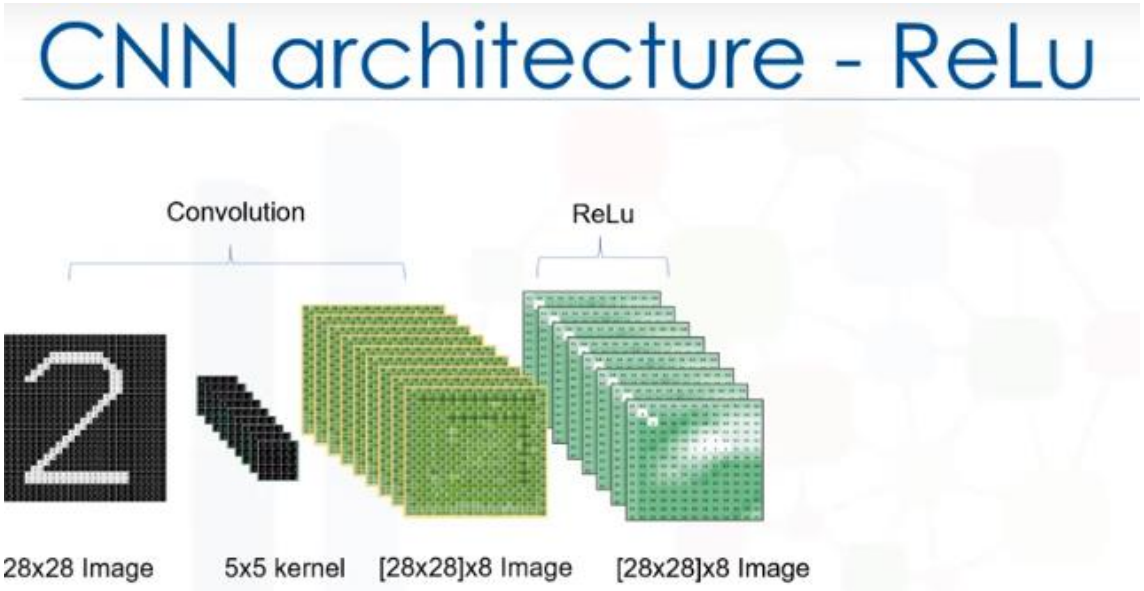


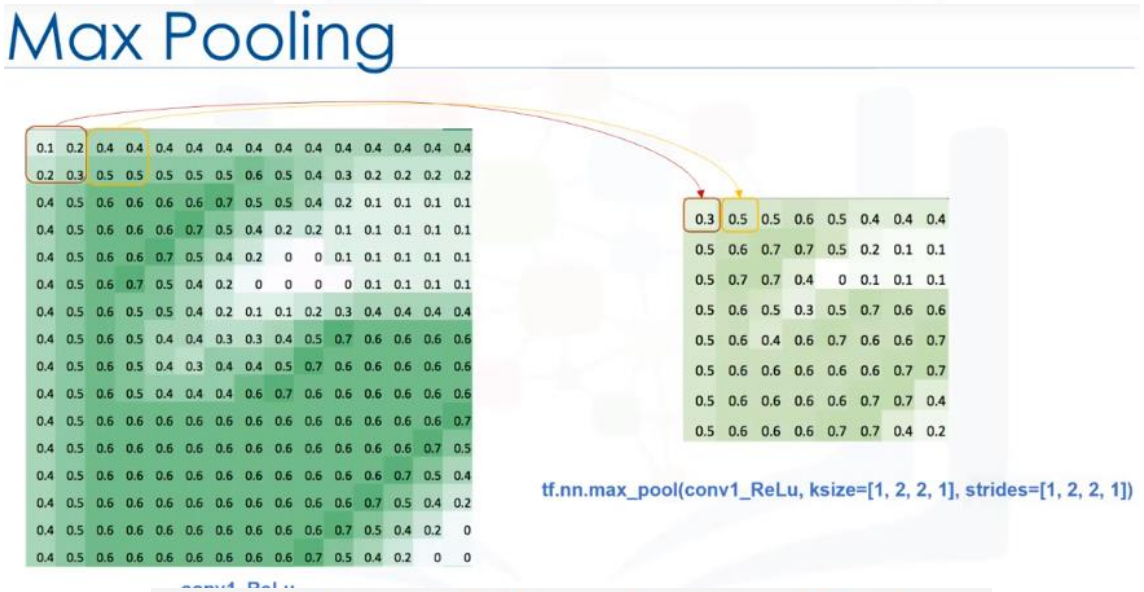


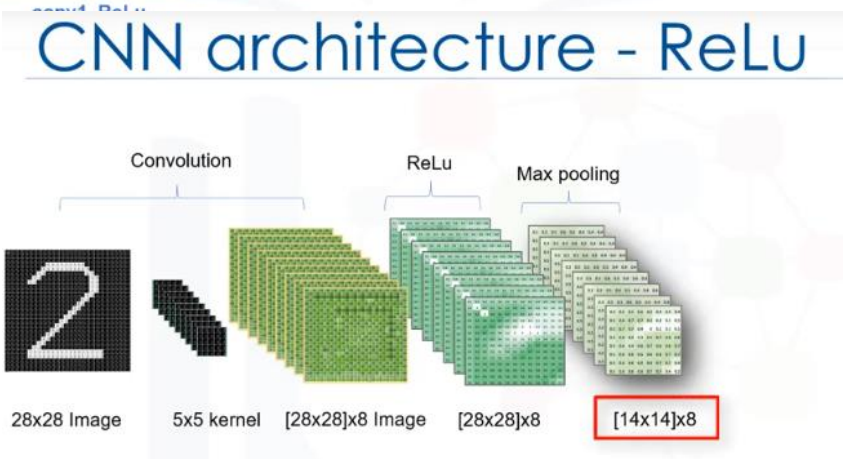

28x28 Matrix
[5x5]x8 Matrix
[28x28]x8 Matrix
Feature map

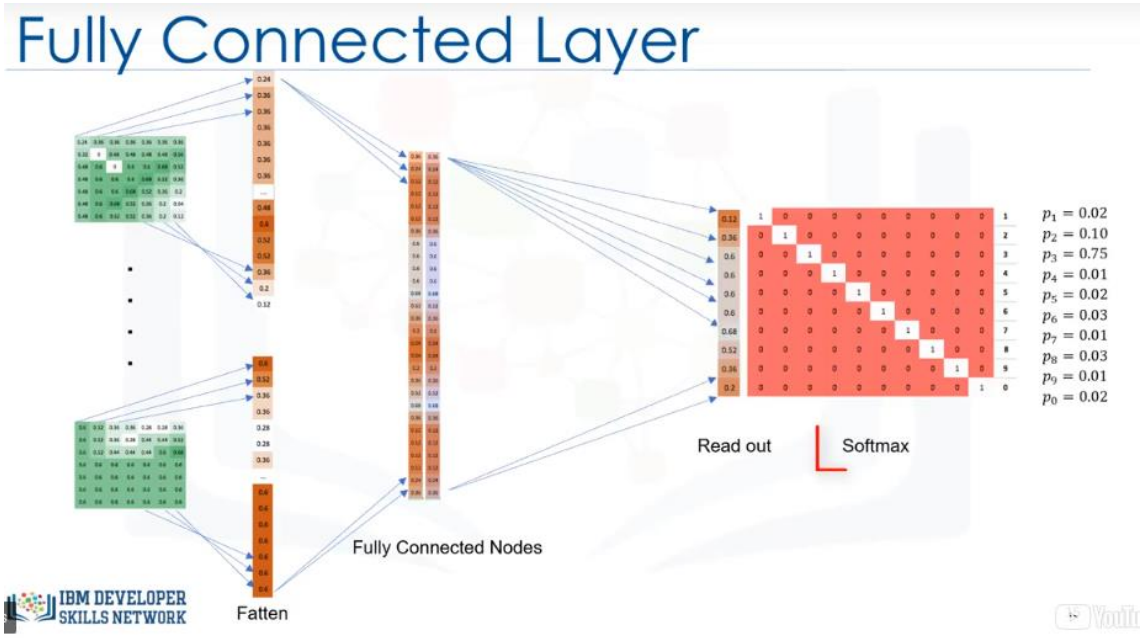

Convolution layer

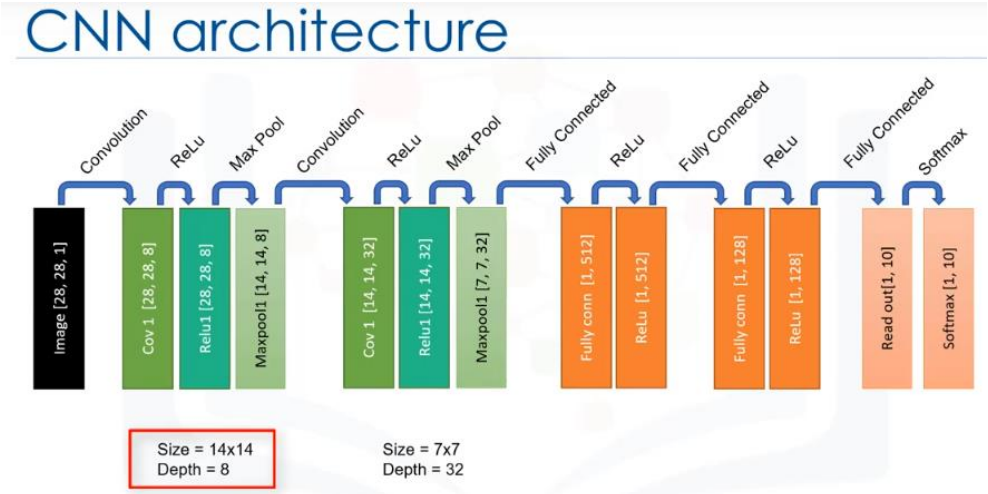

ReLu
convolved1
tf.nn.relu(convolved1)

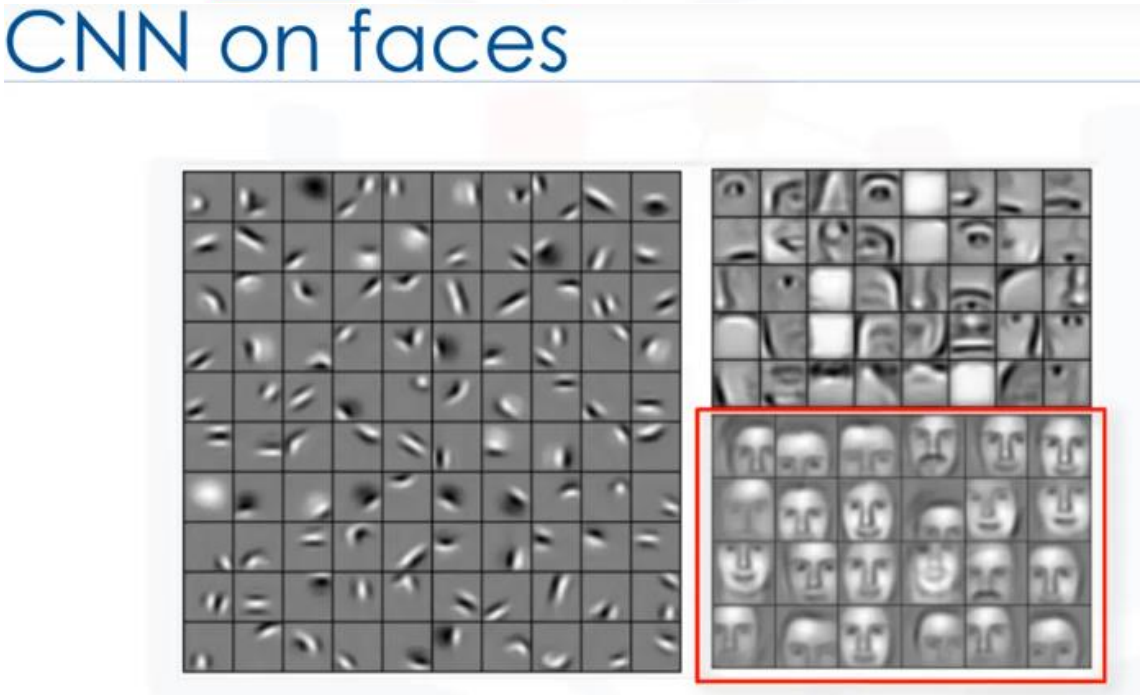

CNN architecture - ReLu
Convolution
ReLu
28x28 Image
5x5 kernel
[28x28]x8 Image
[28x28]x8 Image


Max Pooling
tf.nn.max_pool(conv1_ReLu, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1])


CNN architecture - ReLu
Convolution
ReLu
Max pooling
28x28 Image
5x5 kernel
[28x28]x8 Image
[28x28]x8
[14x14]x8


Fully Connected Layer
$p_1 = 0.02$
$p_2 = 0.10$
$p_3 = 0.75$
$p_4 = 0.01$
$p_5 = 0.02$
$p_6 = 0.03$
$p_7 = 0.01$
$p_8 = 0.03$
$p_9 = 0.01$
$p_0 = 0.02$
Read out
Softmax
Fully Connected Nodes
Fatten
IBM DEVELOPER SKILLS NETWORK


CNN architecture
Convolution
ReLu
Max Pool
Convolution
ReLu
Max Pool
Fully Connected
ReLu
Fully Connected
ReLu
Fully Connected
Softmax
Image [28, 28, 1]
Cov 1 [28, 28, 8]
Relu1 [28, 28, 8]
Maxpool1 [14, 14, 8]
Cov 1 [14, 14, 32]
Relu1 [14, 14, 32]
Maxpool1 [7, 7, 32]
Fully conn [1, 512]
ReLu [1, 512]
Fully conn [1, 128]
ReLu [1, 128]
Read out[1, 10]
Softmax [1, 10]
Size = 14x14
Depth = 8
Size = 7x7
Depth = 32


CNN on faces

Hello, and welcome! In this video, we'll be examining the architecture of the convolutional neural network model.

As previously mentioned, CNN is a type of neural network, empowered with some specific hidden layers, including the Convolutional layer, the Pooling layer, and the Fully-Connected layer. Let's start with the first layer, the Convolutional layer.

The main purpose of the Convolutional layer is to detect different patterns or features from an input image, for example, its edges. Assume that you have an image, and you want to find the edges from the photo ... so that you can define a filter, which is also called a kernel. Now, if you slide the filter over the image, and apply the dot product of the filter to the image pixels, the result would be a new image with all the edges. That is, applying the filter to the left image will result in the right features that are typically useful for the initial layers of a CNN. In fact, the output is one of the very first primitive feature sets in the hierarchy of features.

It is one of the key concepts behind CNNs. Now, the question is: what really happens mathematically when we apply a filter or kernel on an image?

This brings us to the Convolution process. Convolution is a function that can detect edges, orientations, and small patterns in an image.

The convolution operation can be seen as a mathematical function. Imagine that we have a simple black and white image as you see here.

We can convert the image to a matrix of pixels with binary values, where 1 means a white pixel and 0 represents a black pixel. Please keep in mind, however, that the most common usage is a value between 0 and 255 for a grayscale image, or a 3-channel image for a color image. But for now and for the sake of simplicity, let's use 0 and 1 as our values. In this step, we define a filter.

It can be any type of filter, even a random filter, but let's use the edge detector filter for now. Ok, now let's slide it over the image.

And then calculate the convolved value for the first slide.

Taking the values −1 and 1 on two adjacent pixels and zero everywhere else for the filter, results in the following image.

Ok, now let's slide the kernel again. It will calculate the second element of our convolved image. We repeat this process, to complete the convolved matrix. So, in essence, the Convolution Function is a simple matrix multiplication, and in this instance, the result shows us the edges of

the image.
Now, let's consider a hand-written digit that we want to recognize.
We can apply a kernel to it. It is, in fact, the sum of "element-wise multiplication" of the kernel matrix and the image matrix, as shown on the previous slide.
So, we can show the result as convolving a kernel on the image.
Applying multiple kernels on one image will result in multiple convolved images.
Now you can understand that leveraging different kernels allows us to find different patterns in an image, such as edges, curves, and so on.
The output of the convolution process is called a 'feature map'.
At this point, you might be asking yourself: "How do I choose or initialize the proper kernels?" Well, typically, you initialize the kernels with random values, and during the training phase, the values will be updated with optimum values, in such a way that the digit is recognized.
For example, here is the result of applying 8 different kernels on the digit 2. As you can see, each kernel will recognize a particular pattern in the digit.
This is the first layer of Convolutional Deep Learning.
We can interpret this layer to traditional neural network terminology.
The input image can be considered as a matrix that we feed into the neural network through input nodes. For example, in this case, the input layer of the network has 28-by-28 nodes. The output of the co-evolution process, are 8 of [28-by-28] neurons. We can assume those as hidden nodes.
So, what are the weights between the input and hidden nodes?
Yes, the 8 kernels, which are 5-by-5. Now we need to make a decision as to whether each hidden neuron should be "fired" or not.
So, we have to add "activation functions" to the neurons.
We use ReLu (which is short for rectified linear unit) as the activation function on top of nodes in the Convolution layer. ReLu is a nonlinear activation function, which is used to increase the nonlinear properties of the decision function.
So, how can we apply it on Convolution layer? Well, we just go through all outputs of the Convolution layer, covolved1, and wherever a negative number occurs, we swap it out for a 0. This is called the ReLu activation Function.
Now, we have gone one step further. In this step, we have to down-sample the output images from the ReLu function, to reduce the dimensionality of the activate neurons.
So, we use another layer, which is called Max-Pooling.
"Max-Pooling" is an operation that finds the maximum values and simplifies the inputs.
In other words, it reduces the number of parameters within the model.
In a sense, it turns the low level data into higher level information.
For example, we can select a window of size of 2-by-2, and then select the maximum value for this matrix. That is, if the image is a 2-by-2 matrix, it would result in one output pixel. Also, in this step, we can define Strides.
A Stride dictates the sliding behavior of the Max-Pooling.
For example, if we select stride=2, the window will move 2 pixels every time, thus not overlapping.
Ok, now we should have our output matrices, but with lower dimensions, for example 14-by-14.
The next layer is the Fully-Connected layer. Fully-Connected layers take the high-level filtered images from the previous layer, that is, all 8 matrices in our case, and convert them into a vector. First, each previous layer matrix will be converted to a one-dimensional vector. Then, it will be Fully-Connected to the next hidden layer, which usually has a low number of hidden units.
We call it Fully-Connected because each node from the previous layer is connected to all the nodes of this layer. It is connected through a weight matrix.

We can use ReLu activation again here. And finally, we use Softmax to find the class of each digit. Softmax is an activation function that is normally used in classification problems. It generates the probabilities for the output. For example, our model will not be 100% sure that one digit is the number 3; instead, the answer will be a distribution of probabilities where, if the model is right, the 3 will be assigned the larger probability. That is, Softmax can output a multiclass categorical probability distribution.

Now, let's put all these layers together. This chart shows the main layers of a convolutional neural network. As you can see, the whole network generally is doing two tasks. The first part of this network is all about feature learning and extraction ... and the second part revolves around classification.

So, if we look at each operation as a building block, we can see that a typical convolutional neural network might look like this diagram. As you can imagine, though, a typical CNN architecture for an image classification can be much more complicated.

It can be a chain of repeating Conv, ReLu, Max-Pool operations.

For example, here we see more than one convolutional layer, followed by a few Fully-Connected layers.

Also, note the effect of each single Conv, ReLu and Max-Pool operations pass through the image: it reduces height and width of the individual image, and then it increases the depth of the images.

For example, the output of the first convolutional layer is an image of depth 8, and the output of the second layer is an image of depth 32. This is very important, because using multiple layers, CNN will be able to break down the complex patterns into a series of simpler patterns.

If we pass one of the digits through the network, you can see the outputs.

In the first convolutional layer, we apply 8 kernels of size 5-by-5.

Then we apply ReLu and Max-Pool. In the second convolutional layer, we use 32 kernels. Again, we apply ReLu and Max-Pool.

We connect the outputs to a Fully-Connected layer.

Also, we use a second Fully-Connected layer with a lower number of nodes.

And, finally, we use Softmax in the last layer.

And now you can imagine that if we use a CNN trained on human faces, the first layers will represent mostly primitive features, for example the details of each part of the faces; and the next layers represent more abstract patterns such as the nose and eyes; with the last layers representing more face-like patterns.

Now, let's take a closer look at this architecture and see what happens in the training phase of Convolutional Neural Networks . As mentioned, a CNN is a type of feed-forward neural network, consisting of multiple layers of neurons.

Each neuron in a layer receives some input, processes it, and optionally follows it with a non-linear output using an activation function. So, what is the network going to learn through the training process? Well, it learns the connections between the layers. These are the learnable weights and biases matrices. In fact, the whole network is about a series of dot products between the weight matrices and the input matrix.

This means we must first initialize the weights of the network randomly.

Then, we will keep feeding the network with a big dataset of images.

We then check the output of the network and depending on how far the output is from the expected one, we change/update the weights. We keep repeating this process until it reaches a high accuracy of prediction. Of course, this is a very high-level picture of the whole training process. In any case, I hope you've absorbed the

main ideas behind CNNs. For a better understanding of Convolutional Neural Networks, I recommend that you run the labs of this module, which will walk you through different layers of CNN.

Thanks for watching!

End of transcript. Skip to the start.

>>LAB

UNDERSTANDING CONVOLUTIONS

Introduction

Lesson

In this lesson, we will learn more about the key concepts behind the CNNs (Convolutional Neural Networks from now on). This lesson is not intended to be a reference for **machine learning**, **deep learning**, **convolutions** or **TensorFlow**. The intention is to give notions to the user about these fields.

Audience

- Data scientists. General public related to computer science and machine learning.
- Readers interested on TensorFlow and in need of a cloud platform like Workbench Data Scientist

Analogies

There are several ways to understand Convolutional Layers without using a mathematical approach. We are going to explore some of the ideas proposed by the Machine Learning community.

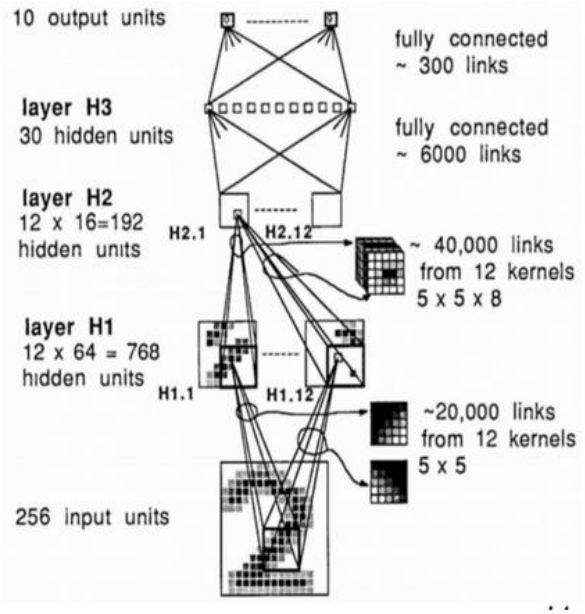
Instances of Neurons

When you start to learn a programming language, one of the first phases of your development is the learning and application of functions. Instead of rewriting pieces of code everytime that you would, a good student is encouraged to code using functional programming, keeping the code organized, clear and concise. CNNs can be thought of as a simplification of what is really going on, a special kind of neural network which uses identical copies of the same neuron. These copies include the same parameters (shared weights and biases) and activation functions.

Location and type of connections

In a fully connected layer NN, each neuron in the current layer is connected to every neuron in the previous layer, and each connection has it's own weight. This is a general purpose connection pattern and makes no assumptions about the features in the input data thus not taking any advantage that the knowledge of the data being used can bring. These types of layers are also very expensive in terms of memory and computation.

In contrast, in a convolutional layer each neuron is only connected to a few nearby local neurons in the previous layer, and the same set of weights is used to connect to them. For example, in the following image, the neurons in the h1 layer are connected only to some input units (pixels).



Feature Learning

Feature engineering is the process of extracting useful patterns from input data that will help the prediction model to understand better the real nature of the problem. A good feature learning will present patterns in a way that significantly increase the accuracy and performance of the applied machine learning algorithms in a way that would otherwise be impossible or too expensive by just machine learning itself.

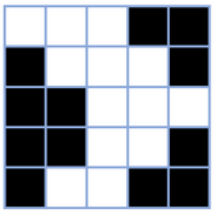
Feature learning algorithms finds the common patterns that are important to distinguish between the wanted classes and extract them automatically. After this process, they are ready to be used in a classification or regression problem.

The great advantage of CNNs is that they are uncommonly good at finding features in images that grow after each level, resulting in high-level features in the end. The final layers (can be one or more) use all these generated features for classification or regression.

Basically, Convolutional Neural Networks is your best friend to **automatically do Feature Engineering** (Feature Learning) without wasting too much time creating your own codes and with no prior need of expertise in the field of Feature Engineering.

Image Filter

How to create a convolved freature from an image ?
The image below is a 8x8 matrix of an image's pixels, converted to binary values in the next image(left), where 1 means a white pixel and 0 a black pixel. Later we will find out that typically this is a normalization, these values can actually have different scales. The most common usage is values between 0 and 255 for 8-bit grayscale images.



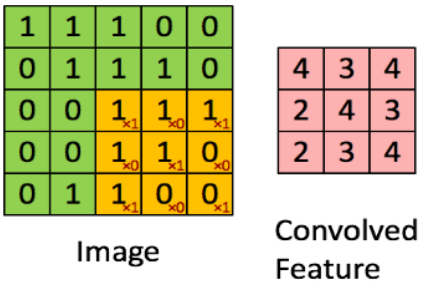
An example of a low resolution image to be recognized.

In the below image, with an animation, you can see how the two-dimensional convolution operation would operate on the images. This operation is performed in most of the Deep Learning frameworks in their first phase. We need a sliding windows to create the convolved matrix:

$$kernel = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

The sliding window (a.k.a kernel, filter or feature detector) with a preset calculation ([[x1, x0,x1], [x0,x1,x0], [x1,x0,x1]]) goes through the image and creates a new matrix (feature map).

The sliding window (a.k.a kernel, filter or feature detector) with a preset calculation ([[x1, x0,x1], [x0,x1,x0], [x1,x0,x1]]) goes through the image and creates a new matrix (feature map).



Animations showing how a kernel interact with a matrix representing an image. ref

In the example above we used a 3×3 filter (5x5 could also be used, but would be too complex). The values from the filter were multiplied element-wise with the original matrix (input image), then summed up. To get the full convolved matrix, the algorithm keep repeating this small procedure for each element by sliding the filter over the whole original matrix.

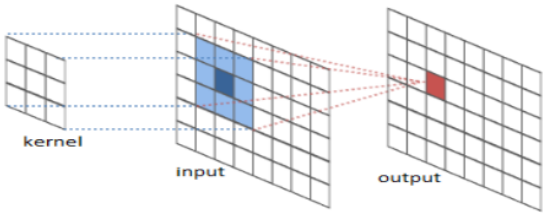


Illustration of the operation for one position of the kernel. ref

Just like the referenced example, we can think of a one-dimensional convolution as sliding function (1x1 or 1x2 filter) multiplying and adding on top of an array (1 dimensional array, instead of the original matrix).

What is the output of applying a kernel on an image?

The famous GIMP (Open Source Image Editor) has an explanation about the convolution operation applied to images that can help us understand hov Neural Networks will interact with this tool.

0	0	0	0	0
0	1/9	1/9	1/9	0
0	1/9	1/9	1/9	0
0	1/9	1/9	1/9	0
0	0	0	0	0

Applying the left kernel to the image will result into a blur effect. ref

Applying the left kernel to the image will result into a blur effect. ref

Well, this is very good if you want nice effects for your social media photos, but in the field of computer vision you need detailed patterns (remember feature learning) that are almost erased using a kernel like that. A more suitable example would be the Kernel/filter that shows edges from photos (the first recognizable feature of an image).

Lets try another kernel:

Taking the values −1 and 1 on two adjacent pixels and zero everywhere else for the kernel, results in the following image. That is, we subtract two adjacent pixels. When side by side pixels are similar, this gives us approximately zero. On edges, however, adjacent pixels are very different in the direction perpendicular to the edge. Knowing that results differs from zero will result in brighter pixels, you can already guess the result of this type of kernel.

0	0	0	0	0
0	0	0	0	0
0	-1	1	0	0
0	0	0	0	0
0	0	0	0	0

Understanding and coding with Python

Convolution: 1D operation with Python (Numpy/Scipy)

Mathematical notation

In this first example, we will use the pure mathematical notation. Here we have a one dimensional convolution operation. Lets say h is our image and x is our kernel:
x[i] = { 3, 4, 5 }
h[i] = { 2, 1, 0 }

where i = index

To use the convolution operation between the two arrays try the code below to see how easy it is to do in Python.

```
import numpy as np

h = [2, 1, 0]
x = [3, 4, 5]


y = np.convolve(x, h)
y
```

```
[1]: array([ 6, 11, 14,  5,  0])
```

sliding x window over h:

- 6 = 2 * 3 : $\begin{bmatrix}3 & 4 & 5\end{bmatrix}$
  $\begin{bmatrix}2 & 0 & 0\end{bmatrix}$
- 11 = 1 * 3 + 2 * 4 : $\begin{bmatrix}3 & 4 & 5\end{bmatrix}$
  $\begin{bmatrix}1 & 2 & 0\end{bmatrix}$
- 14 = 0 * 3 + 1 * 4 + 2 * 5 : $\begin{bmatrix}3 & 4 & 5\end{bmatrix}$
  $\begin{bmatrix}0 & 1 & 2\end{bmatrix}$
- 5 = 0 * 4 + 1 * 5 : $\begin{bmatrix}3 & 4 & 5\end{bmatrix}$
  $\begin{bmatrix}0 & 0 & 1\end{bmatrix}$
- 0 = 0 * 5 : $\begin{bmatrix}3 & 4 & 5\end{bmatrix}$
  $\begin{bmatrix}0 & 0 & 0\end{bmatrix}$

Now we are going to verify what Python did, because we don't trust computer outputs while we are learning. Using the equation of convolution for y[n]:

$$y[n] = \sum_{k \to -\infty}^{\infty} x[k] \cdot h[n-k]$$

And then, manually executing computation:

$$y[0] = \sum_{k \to -\infty}^{\infty} x[k] \cdot h[0-k] = x[0] \cdot h[0] = 3 \cdot 2 = 6$$

$$y[1] = \sum_{k \to -\infty}^{\infty} x[k] \cdot h[1-k] = x[0] \cdot h[1-0] + x[1] \cdot h[1-1] + \ldots$$
$$= x[0] \cdot h[1] + x[1] \cdot h[0] = 3 \cdot 1 + 4 \cdot 2 = 11$$

$$y[2] = \sum_{k \to -\infty}^{\infty} x[k] \cdot h[2-k] = x[0] \cdot h[2-0] + x[1] \cdot h[2-1] + x[2] \cdot h[2-2] + \ldots$$
$$= x[0] \cdot h[2] + x[1] \cdot h[1] + x[2] \cdot h[0] = 3 \cdot 0 + 4 \cdot 1 + 5 \cdot 2 = 14$$

$$y[3] = \sum_{k \to -\infty}^{\infty} x[k] \cdot h[3-k] = x[0] \cdot h[3-0] + x[1] \cdot h[3-1] + x[2] \cdot h[3-2] + x[3] \cdot h[3-3] + \ldots$$
$$= x[0] \cdot h[3] + x[1] \cdot h[2] + x[2] \cdot h[1] + x[3] \cdot h[0] = 0 + 0 + 5 \cdot 1 + 0 = 5$$

$$y[4] = \sum_{k \to -\infty}^{\infty} x[k] \cdot h[4-k] = x[0] \cdot h[4-0] + x[1] \cdot h[4-1] + x[2] \cdot h[4-2] + \ldots = 0$$

```
print("Compare with the following values from Python: y[0] = {0} ; y[1] = {1}; y[2] = {2}; y[3] = {3}; y[4] = {4}".format(y[0], y[
```

```
Compare with the following values from Python: y[0] = 6 ; y[1] = 11; y[2] = 14; y[3] = 5; y[4] = 0
```

There are three methods to apply kernel on the matrix, **with padding (full)**, **with padding(same)** and **without padding(valid)**:

1) Visually understanding the operation with padding (full)
Lets think of the kernel as a sliding window. We have to come with the solution of padding zeros on the input array. This is a very famous implementation and will be easier to show how it works with a simple example, consider this case:

x[i] = [6,2]
h[i] = [1,2,5,4]

Using the zero padding, we can calculate the convolution.

You have to invert the filter x, otherwise the operation would be cross-correlation. First step, (now with zero padding):

1) Visually understanding the operation with padding (full)

Lets think of the kernel as a sliding window. We have to come with the solution of padding zeros on the input array. This is a very famous implementation and will be easier to show how it works with a simple example, consider this case:

x[] = [6,2]
h[] = [1,2,5,4]

Using the zero padding, we can calculate the convolution.

You have to invert the filter x, otherwise the operation would be cross-correlation. First step, (now with zero padding):

```
[2  6]
 |  |
 V  V
 0 [1 2 5 4]
```

= 2 * 0 + 6 * 1 = 6

Second step:

```
  [2  6]
   |  |
   V  V
0 [1  2  5  4]
```

= 2 * 1 + 6 * 2 = 14 (the arrows represent the connection between the kernel and the input)

Third step:

```
    [2  6]
     |  |
     V  V
0 [1  2  5  4]
```

= 2 * 2 + 6 * 5 = 34

Fourth step:

```
      [2  6]
       |  |
       V  V
0 [1  2  5  4]
```

= 2 * 5 + 6 * 4 = 34

Fifth step:

```
            [2  6]
             |  |
             V  V
0 [1  2  5  4] 0
```

= 2 * 4 + 6 * 0 = 8

The result of the convolution for this case, listing all the steps, would then be: Y = [6 14 34 34 8]

Below we verify with numpy:

```
import numpy as np

x = [6, 2]
h = [1, 2, 5, 4]

y = np.convolve(x, h, "full")  #now, because of the zero padding, the final dimension of the array is bigger
y
```

```
array([ 6, 14, 34, 34,  8])
```

2) Visually understanding the operation with "same"

In this approach, we just add the zero to left (and top of the matrix in 2D). That is, only the first 4 steps of "full" method:

```
import numpy as np

x = [6, 2]
h = [1, 2, 5, 4]

y = np.convolve(x, h, "same")  # it is same as zero padding, but with returns an ouput with the same length as max of x or h
y
```

```
array([ 6, 14, 34, 34])
```

3) Visually understanding the operation with no padding (valid)

In the last case we only applied the kernel when we had a compatible position on the h array, in some cases you want a dimensionality reduction. For this purpose, we simple ignore the steps that would need padding:

x[i] = [6 2]

h[i] = [1 2 5 4]

You have to invert the filter x, otherwise the operation would be cross-correlation. First step, (now without zero padding):

You have to invert the filter x, otherwise the operation would be cross-correlation. First step, (now without zero padding):

```
[2  6]
 |  |
 V  V
[1  2  5  4]
```

= 2 * 1 + 6 * 2 = 14 (the arrows represent the connection between the kernel and the input)

Second step:

```
   [2  6]
    |  |
    V  V
[1  2  5  4]
```

= 2 * 2 + 6 * 5 = 34

Third step:

```
      [2  6]
       |  |
       V  V
[1  2  5  4]
```

= 2 * 5 + 6 * 4 = 34

The result of the convolution for this mode would then be Y= [14 34 34] = [First, second, third step]

Let's verify with numpy

Let's verify with numpy

```
import numpy as np

x = [6, 2]
h = [1, 2, 5, 4]

y = np.convolve(x, h, "valid")  # valid returns output of length max(x, h) - min(x, h) + 1, this is to ensure that values outside of the boundary of
                                # h will not be used in the calculation of the convolution
                                # in the next example we will understand why we used the argument valid
y
```

```
array([14, 34, 34])
```

Convolution: 2D operation with Python (Numpy/Scipy)

The 2D convolution operation is defined as:

$$I' = \sum_{u,v} I(x-u, y-v)g(u,v)$$

Below we will apply the equation to an image represented by a 3x3 matrix according to the function g = (-1 1). Please note that when we apply the kernel we always use its inversic

$$I = \begin{bmatrix} 255 & 7 & 3 \\ 212 & 240 & 4 \\ 218 & 216 & 230 \end{bmatrix}$$

$$g = \begin{bmatrix} -1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{1 \cdot 0} & \mathbf{-1 * 255} & 7 & 3 \\ 0 & 212 & 240 & 4 \\ 0 & 218 & 216 & 230 \end{bmatrix} \rightarrow \begin{bmatrix} \mathbf{-255} & 7 & 3 \\ 212 & 240 & 4 \\ 218 & 216 & 230 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{1 * 255} & \mathbf{-1 * 7} & 3 \\ 212 & 240 & 4 \\ 218 & 216 & 230 \end{bmatrix} \rightarrow \begin{bmatrix} -255 & \mathbf{248} & 3 \\ 212 & 240 & 4 \\ 218 & 216 & 230 \end{bmatrix}$$

$$\begin{bmatrix} 255 & \mathbf{1 * 7} & \mathbf{-1 * 3} \\ 212 & 240 & 4 \\ 218 & 216 & 230 \end{bmatrix} \rightarrow \begin{bmatrix} -255 & 248 & \mathbf{4} \\ 212 & 240 & 4 \\ 218 & 216 & 230 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 255 & 7 & 3 \\ \mathbf{1 * 0} & \mathbf{-1 * 212} & 240 & 4 \\ 0 & 218 & 216 & 230 \end{bmatrix} \rightarrow \begin{bmatrix} \mathbf{-255} & 248 & 4 \\ -212 & 240 & 4 \\ 218 & 216 & 230 \end{bmatrix}$$

```
from scipy import signal as sg

I= [[255,   7,  3],
    [212, 240,  4],
    [218, 216, 230],]

g= [[-1, 1]]

print('Without zero padding \n')
print('{0} \n'.format(sg.convolve( I, g, 'valid')))
# The 'valid' argument states that the output consists only of those elements
# that do not rely on the zero-padding.

print('With zero padding \n')
print(sg.convolve( I, g))
```

```
Without zero padding

[[248   4]
 [-28 236]
 [  2 -14]]

With zero padding

[[-255  248    4    3]
 [-212  -28  236    4]
 [-218    2  -14  230]]
```

For a more difficult case where h= [[-1 1] , [2 3]]

For a more difficult case where h= [[-1 1] , [2 3]]

$$\begin{bmatrix} \mathbf{3*0} & \mathbf{2*0} & 0 & 0 \\ \mathbf{1*0} & \mathbf{-1*255} & 7 & 3 \\ 0 & 212 & 240 & 4 \\ 0 & 218 & 216 & 230 \end{bmatrix} \rightarrow \begin{bmatrix} \mathbf{-255} & 7 & 3 \\ 212 & 240 & 4 \\ 218 & 216 & 230 \end{bmatrix}$$

```
from scipy import signal as sg

I= [[255,   7,  3],
    [212, 240,  4],
    [218, 216, 230],]

g= [[-1,  1],
    [ 2,  3],]

print ('With zero padding \n')
print ('{0} \n'.format(sg.convolve( I, g, 'full')))
# The output is the full discrete linear convolution of the inputs.
# It will use zero to complete the input matrix

print ('With zero padding_same_ \n')
print ('{0} \n'.format(sg.convolve( I, g, 'same')))
# The output is the full discrete linear convolution of the inputs.
# It will use zero to complete the input matrix


print ('Without zero padding \n')
print (sg.convolve( I, g, 'valid'))
# The 'valid' argument states that the output consists only of those elements
#that do not rely on the zero-padding.
```

```
With zero padding

[[-255  248    4    3]
 [ 298  751  263   13]
 [ 206 1118  714  242]
 [ 436 1086 1108  690]]

With zero padding_same_

[[-255  248    4]
 [ 298  751  263]
 [ 206 1118  714]]
```

Coding with TensorFlow

Numpy is great because it has high optimized matrix operations implemented in a backend using C/C++. However, if our goal is to work with DeepLearning, we need much more. TensorFlow does the same work, but instead of returning to Python everytime, it creates all the operations in the form of graphs and execute them once with the highly optimized backend.

Suppose that you have two tensors:

- 3x3 filter (4D tensor = [3,3,1,1] = [width, height, channels, number of filters])
- 10x10 image (4D tensor = [1,10,10,1] = [batch size, width, height, number of channels]

The output size for zero padding 'SAME' mode will be:

- the same as input = 10x10

The output size without zero padding 'VALID' mode:

- input size - kernel dimension + 1 = 10 -3 + 1 = 8 = 8x8

```
import tensorflow as tf

#Building graph

input = tf.Variable(tf.random_normal([1, 10, 10, 1]))
filter = tf.Variable(tf.random_normal([3, 3, 1, 1]))
op = tf.nn.conv2d(input, filter, strides=[1, 1, 1, 1], padding='VALID')
op2 = tf.nn.conv2d(input, filter, strides=[1, 1, 1, 1], padding='SAME')

#Initialization and session
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)

    print("Input \n")
    print('{0} \n'.format(input.eval()))
    print("Filter/Kernel \n")
    print('{0} \n'.format(filter.eval()))
    print("Result/Feature Map with valid positions \n")
    result = sess.run(op)
    print(result)
    print('\n')
    print("Result/Feature Map with padding \n")
    result2 = sess.run(op2)
    print(result2)
```

```
Input

[[[[ 0.22506092]
   [-2.1198196 ]
   [ 0.2441127 ]
   [ 0.54004675]
   [ 0.1768044 ]
   [-0.35375097]
   [-1.4412675 ]
   [-1.315734  ]
   [-0.40511343]
   [ 1.5017577 ]]

  [[-1.3078064 ]
   [ 0.26152012]
   [-0.09683216]
   [ 0.36832494]
   [-0.23609185]
   [ 0.19725367]
   [-0.7985585 ]
   [-1.1443577 ]
   [-1.0497836 ]
   [-0.12730429]]]


Filter/Kernel

[[[[ 0.30544773]]

  [[ 0.807559  ]]

  [[ 0.28313854]]]


 [[[ 1.417794  ]]

  [[ 1.3607131 ]]

  [[ 1.3013833 ]]]


 [[[ 1.275018  ]]

  [[-1.3338234 ]]

  [[-0.62375325]]]]


Result/Feature Map with valid positions
```

```
Result/Feature Map with padding

[[[[-8.7119567e-01]
   [-4.2035909e+00]
   [-1.7376366e+00]
   [ 8.4356034e-01]
   [ 1.2073777e+00]
   [-2.1723423e+00]
   [-2.1445370e+00]
   [-3.1979644e+00]
   [-4.4176450e-01]
   [ 3.0040261e-01]]

  [[-3.4385810e+00]
   [-3.4621747e+00]
   [ 1.6440883e+00]
   [ 6.4053118e-02]
   [ 1.8813981e+00]
   [-2.7286854e+00]
   [-5.0936313e+00]
   [-6.5295434e+00]
   [-3.3482094e+00]
   [ 2.7136097e+00]]

  [[-7.4019915e-01]
```

Convolution applied on images

Upload your own image (drag and drop to this window) and type its name on the input field on the next cell (press **shift** + **enter**). The result of this pre-processing will be an image with only a grayscale channel.

You can type **bird.jpg** to use a default image

```
# download standard image
!wget --quiet https://ibm.box.com/shared/static/cn7yt7z10j8rx6um1v9seagpgmzzxnlz.jpg --output-document bird.jpg
```

```
#Importing
import numpy as np
from scipy import signal
from scipy import misc
import matplotlib.pyplot as plt
from PIL import Image

im = Image.open('bird.jpg')  # type here your image's name

image_gr = im.convert("L")    # convert("L") translate color images into black and white
                              # uses the ITU-R 601-2 luma transform (there are several
                              # ways to convert an image to grey scale)
print("\n Original type: %r \n\n" % image_gr)

# convert image to a matrix with values from 0 to 255 (uint8)
arr = np.asarray(image_gr)
print("After conversion to numerical representation: \n\n %r" % arr)
### Activating matplotlib for Ipython
%matplotlib inline

### Plot image

imgplot = plt.imshow(arr)
imgplot.set_cmap('gray')  #you can experiment different colormaps (Greys,winter,autumn)
print("\n Input image converted to gray scale: \n")
plt.show(imgplot)
```

```
Original type: <PIL.Image.Image image mode=L size=1920x1440 at 0x7F6F4C3ABE10>
```

```
Original type: <PIL.Image.Image image mode=L size=1920x1440 at 0x7F6F4C3ABE10>
```

```
After conversion to numerical representation:
```

```
array([[ 64,  71,  65, ...,  49,  47,  48],
       [ 68,  71,  64, ...,  54,  52,  51],
       [ 65,  69,  66, ...,  54,  50,  55],
       ...,
       [ 21,  24,  23, ..., 184, 170, 155],
       [ 18,  21,  26, ..., 179, 166, 153],
       [ 27,  22,  21, ..., 170, 159, 149]], dtype=uint8)
```

```
Input image converted to gray scale:
```

Now, we will experiment using an edge detector kernel.

```
kernel = np.array([[ 0, 1, 0],
                   [ 1,-4, 1],
                   [ 0, 1, 0],])

grad = signal.convolve2d(arr, kernel, mode='same', boundary='symm')
```

```
%matplotlib inline

print('GRADIENT MAGNITUDE - Feature map')

fig, aux = plt.subplots(figsize=(10, 10))
aux.imshow(np.absolute(grad), cmap='gray')
```

```
GRADIENT MAGNITUDE - Feature map
<matplotlib.image.AxesImage at 0x7f6fc8a0e0b8>
```

GRADIENT MAGNITUDE - Feature map
<matplotlib.image.AxesImage at 0x7f6fc8a0e0b8>

If we change the kernel and start to analyze the outputs we would be acting as a CNN. The difference is that a Neural Network do all this work automatically (the kernel adjustment using different weights). In addition, we can understand how biases affect the behaviour of feature maps

Please note that when you are dealing with most of the real applications of CNNs, you usually convert the pixels values to a range from 0 to 1. This process is called normalization.

```
type(grad)

grad_biases = np.absolute(grad) + 100

grad_biases[grad_biases > 255] = 255
```

Please note that when you are dealing with most of the real applications of CNNs, you usually convert the pixels values to a range from 0 to 1. This process is called normalization.

```
type(grad)

grad_biases = np.absolute(grad) + 100

grad_biases[grad_biases > 255] = 255
```

```
%matplotlib inline

print('GRADIENT MAGNITUDE - Feature map')

fig, aux = plt.subplots(figsize=(10, 10))
aux.imshow(np.absolute(grad_biases), cmap='gray')
```

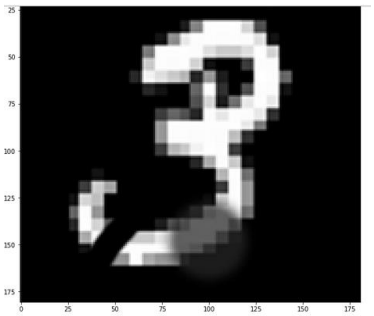
```
GRADIENT MAGNITUDE - Feature map
```

Lets see how it works for a digit:

```
# download standard image
!wget --quiet https://ibm.box.com/shared/static/vvm1b63uvuxq88vbw9znpwu5ol380mco.jpg --output-document num3.jpg
```

```
#Importing
import numpy as np
from scipy import signal
from scipy import misc
import matplotlib.pyplot as plt
from PIL import Image

im = Image.open('num3.jpg')  # type here your image's name

image_gr = im.convert("L")    # convert("L") translate color images into black and white
                              # uses the ITU-R 601-2 luma transform (there are several
                              # ways to convert an image to grey scale)
print("\n Original type: %r \n\n" % image_gr)

# convert image to a matrix with values from 0 to 255 (uint8)
arr = np.asarray(image_gr)
print("After conversion to numerical representation: \n\n %r" % arr)
### Activating matplotlib for Ipython
%matplotlib inline

### Plot image
fig, aux = plt.subplots(figsize=(10, 10))
imgplot = plt.imshow(arr)
imgplot.set_cmap('gray')  #you can experiment different colormaps (Greys,winter,autumn)
print("\n Input image converted to gray scale: \n")
plt.show(imgplot)
```

```
 Original type: <PIL.Image.Image image mode=L size=181x181 at 0x7F6F8F244470>


After conversion to numerical representation:

 array([[26, 14, 12, ..., 11, 11, 11],
       [ 0,  0,  1, ...,  0,  0,  0],
       [12,  1,  0, ...,  0,  0,  0],
       ...,
       [12,  0,  0, ...,  0,  0,  0],
       [12,  0,  0, ...,  0,  0,  0],
       [12,  0,  0, ...,  0,  0,  0]], dtype=uint8)

 Input image converted to gray scale:
```



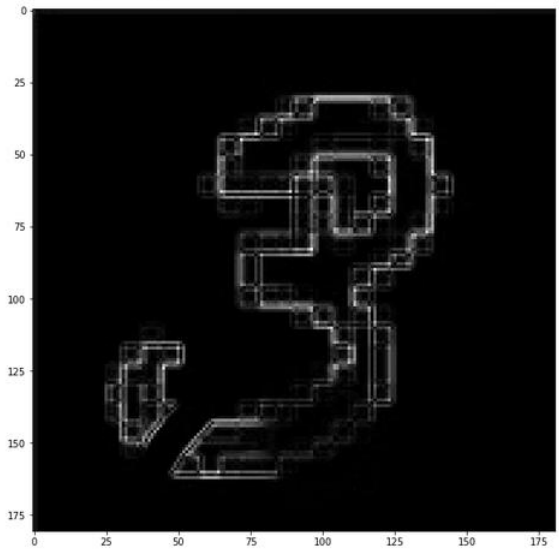
Now, we will experiment using an edge detector kernel.

```
kernel = np.array([
                        [ 0, 1, 0],
                        [ 1,-4, 1],
                        [ 0, 1, 0],
                                  ])

grad = signal.convolve2d(arr, kernel, mode='same', boundary='symm')
%matplotlib inline

print('GRADIENT MAGNITUDE - Feature map')

fig, aux = plt.subplots(figsize=(10, 10))
aux.imshow(np.absolute(grad), cmap='gray')
```

```
GRADIENT MAGNITUDE - Feature map
<matplotlib.image.AxesImage at 0x7f6f8a8611d0>
```

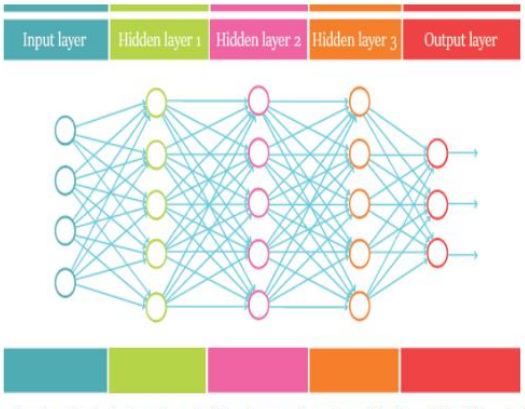
GRADIENT MAGNITUDE - Feature map

```
<matplotlib.image.AxesImage at 0x7f6f8a8611d0>
```



What is Deep Learning?

Brief Theory: Deep learning (also known as deep structured learning, hierarchical learning or deep machine learning) is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data by using multiple processing layers, with complex structures or otherwise, composed of multiple non-linear transformations.



It's time for deep learning. Our brain doesn't work with only one or three layers. Why it would be different with machines?

In Practice, defining the term "Deep": in this context, deep means that we are studying a Neural Network which has several hidden layers (more than one), no matter what type (convolutional, pooling, normalization, fully-connected etc). The most interesting part is that some papers noticed that Deep Neural Networks with the right architectures/hyper-parameters achieve better results than shallow Neural Networks with the same computational power (e.g. number of neurons or connections).

In Practice, defining "Learning": In the context of supervised learning, digits recognition in our case, the learning part consists of a target/feature which is to be predicted using a given set of observations with the already known final prediction (label). In our case, the target will be the digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and the observations are the intensity and relative position of the pixels. After some training, it is possible to generate a "function" that map inputs (digit image) to desired outputs(type of digit). The only problem is how well this map operation occurs. While trying to generate this "function", the training process continues until the model achieves a desired level of accuracy on the training data.

1st part: classify MNIST using a simple model.

We are going to create a simple Multi-layer perceptron, a simple type of Neural Network, to perform classification tasks on the MNIST digits dataset. If you are not familiar with the MNIST dataset, please consider to read more about it: click here

What is MNIST?

According to LeCun's website, the MNIST is a: "database of handwritten digits that has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image".

Import the MNIST dataset using TensorFlow built-in feature

It's very important to notice that MNIST is a high optimized data-set and it does not contain images. You will need to build your own code if you want to see the real digits. Another important side note is the effort that the authors invested on this data-set with normalization and centering operations.

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

```
WARNING:tensorflow:From <ipython-input-2-8bf8ae5a5303>:2: read_data_sets (from tensorflow.contrib.learn.python.learn.datasets.mnist) is deprecated and will be removed in a future version.
```

The one-hot = True argument only means that, in contrast to Binary representation, the labels will be presented in a way that to represent a number N, the N^{th} bit is 1 while the the other bits are 0. For example, five and zero in a binary code would be:

```
Number representation:    0
Binary encoding:        [2^5]  [2^4]   [2^3]   [2^2]   [2^1]   [2^0]
Array/vector:             0      0       0       0       0       0

Number representation:    5
Binary encoding:        [2^5]  [2^4]   [2^3]   [2^2]   [2^1]   [2^0]
Array/vector:             0      0       0       1       0       1
```

Using a different notation, the same digits using one-hot vector representation can be show as:

```
Number representation:    0
One-hot encoding:       [5]   [4]    [3]    [2]    [1]   [0]
Array/vector:             0     0      0      0      0     1

Number representation:    5
One-hot encoding:       [5]   [4]    [3]    [2]    [1]    [0]
Array/vector:             1     0      0      0      0      0
```

Understanding the imported data

The imported data can be divided as follow:

- Training (mnist.train) >> Use the given dataset with inputs and related outputs for training of NN. In our case, if you give an image that you know that represents a "nine", this set will tell the neural network that we expect a "nine" as the output.
  - 55,000 data points
  - mnist.train.images for inputs
  - mnist.train.labels for outputs
- Validation (mnist.validation) >> The same as training, but now the data is used to generate model properties (classification error, for example) and from this, tune parameters like the optimal number of hidden units or determine a stopping point for the back-propagation algorithm
  - 5,000 data points
  - mnist.validation.images for inputs
  - mnist.validation.labels for outputs
- Test (mnist.test) >> the model does not have access to this informations prior to the testing phase. It is used to evaluate the performance and accuracy of the model against "real life situations". No further optimization beyond this point.
  - 10,000 data points
  - mnist.test.images for inputs
  - mnist.test.labels for outputs

Creating an interactive section

You have two basic options when using TensorFlow to run your code:

- [Build graphs and run session] Do all the set-up and THEN execute a session to evaluate tensors and run operations (ops)
- [Interactive session] create your coding and run on the fly.

For this first part, we will use the interactive session that is more suitable for environments like Jupyter notebooks.

```
]: sess = tf.InteractiveSession()
```

Creating placeholders

It iss a best practice to create placeholders before variable assignments when using TensorFlow. Here we'll create placeholders for inputs ("Xs") and outputs ("Ys").

Placeholder 'X': represents the "space" allocated input or the images.

- Each input has 784 pixels distributed by a 28 width x 28 height matrix
- The 'shape' argument defines the tensor size by its dimensions.
- 1st dimension = None. Indicates that the batch size, can be of any size.
- 2nd dimension = 784. Indicates the number of pixels on a single flattened MNIST image.

Placeholder 'Y': represents the final output or the labels.

- 10 possible classes (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- The 'shape' argument defines the tensor size by its dimensions.
- 1st dimension = None. Indicates that the batch size, can be of any size.
- 2nd dimension = 10. Indicates the number of targets/outcomes

dtype for both placeholders: if you not sure, use tf.float32. The limitation here is that the later presented softmax function only accepts float32 or float64 dtypes. For more dtypes, check TensorFlow's documentation here

```
x  = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])
```

Assigning bias and weights to null tensors

Now we are going to create the weights and biases, for this purpose they will be used as arrays filled with zeros. The values that we choose here can be critical, but we'll cover a better way on the second part, instead of this type of initialization.

```
# Weight tensor
W = tf.Variable(tf.zeros([784, 10],tf.float32))
# Bias tensor
b = tf.Variable(tf.zeros([10],tf.float32))
```

Execute the assignment operation

Before, we assigned the weights and biases but we did not initialize them with null values. For this reason, TensorFlow need to initialize the variables that you assign.
Please notice that we're using this notation "sess.run" because we previously started an interactive session.

```
# run the op initialize_all_variables using an interactive session
sess.run(tf.global_variables_initializer())
```

Adding Weights and Biases to input

The only difference for our next operation to the picture below is that we are using the mathematical convention for what is being executed in the illustration. The tf.matmul operation performs a matrix multiplication between x (inputs) and W (weights) and after the code add biases.

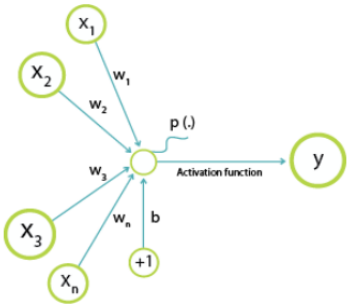


Illustration showing how weights and biases are added to neurons/nodes.

```
# mathematical operation to add weights and biases to the inputs
tf.matmul(x,W) + b
```

```
<tf.Tensor 'add:0' shape=(?, 10) dtype=float32>
```

Softmax Regression

Softmax is an activation function that is normally used in classification problems. It generate the probabilities for the output. For example, our model will not be 100% sure that one digit is the number nine, instead, the answer will be a distribution of probabilities where, if the model is right, the nine number will have a larger probability than the other other digits.

For comparison, below is the one-hot vector for a nine digit label:

```
0 --> 0
1 --> 0
2 --> 0
3 --> 0
4 --> 0
5 --> 0
6 --> 0
7 --> 0
8 --> 0
9 --> 1
```

A machine does not have all this certainty, so we want to know what is the best guess, but we also want to understand how sure it was and what was the second better option. Below is an example of a hypothetical distribution for a nine digit:

```
0 -->0.01
1 -->0.02
2 -->0.03
3 -->0.02
4 -->0.12
5 -->0.01
6 -->0.03
7 -->0.06
8 -->0.1
9 -->0.6
```

```
y = tf.nn.softmax(tf.matmul(x,W) + b)
```

Logistic function output is used for the classification between two target classes 0/1. Softmax function is generalized type of logistic function. That is, Softmax can output a multiclass categorical probability distribution.

Cost function

It is a function that is used to minimize the difference between the right answers (labels) and estimated outputs by our Network.

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

Type of optimization: Gradient Descent

This is the part where you configure the optimizer for your Neural Network. There are several optimizers available, in our case we will use Gradient Descent because it is a well established optimizer.

```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

Training batches

Train using minibatch Gradient Descent.

In practice, Batch Gradient Descent is not often used because is too computationally expensive. The good part about this method is that you have the true gradient, but with the expensive computing task of using the whole dataset in one time. Due to this problem, Neural Networks usually use minibatch to train.

```
#Load 50 training examples for each training iteration
for i in range(1000):
    batch = mnist.train.next_batch(50)
    train_step.run(feed_dict={x: batch[0], y_: batch[1]})
```

Test

```
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
acc = accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels}) * 100
print("The final accuracy for the simple ANN model is: {} % ".format(acc) )
```

```
The final accuracy for the simple ANN model is: 90.93999862670898 %
```

```
sess.close() #finish the session
```

Evaluating the final result

Is the final result good?

Let's check the best algorithm available out there (10th June 2016):

Result: 0.21% error (99.79% accuracy)
Reference here

How to improve our model?

Several options as follow:

- Regularization of Neural Networks using DropConnect
- Multi-column Deep Neural Networks for Image Classification
- APAC: Augmented Pattern Classification with Neural Networks
- Simple Deep Neural Network with Dropout

In the next part we are going to explore the option:

- Simple Deep Neural Network with Dropout (more than 1 hidden layer)

2nd part: Deep Learning applied on MNIST

In the first part, we learned how to use a simple ANN to classify MNIST. Now we are going to expand our knowledge using a Deep Neural Network.

Architecture of our network is:

- (Input) -> [batch_size, 28, 28, 1] >> Apply 32 filter of [5x5]
- (Convolutional layer 1) -> [batch_size, 28, 28, 32]
- (ReLU 1) -> [?, 28, 28, 32]
- (Max pooling 1) -> [?, 14, 14, 32]
- (Convolutional layer 2) -> [?, 14, 14, 64]
- (ReLU 2) -> [?, 14, 14, 64]
- (Max pooling 2) -> [?, 7, 7, 64]
- [fully connected layer 3] -> [1x1024]
- [ReLU 3] -> [1x1024]
- [Drop out] -> [1x1024]
- [fully connected layer 4] -> [1x10]

The next cells will explore this new architecture.

Starting the code

```
import tensorflow as tf

# finish possible remaining session
sess.close()

#Start interactive session
sess = tf.InteractiveSession()
```

The MNIST data

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

Initial parameters

Create general parameters for the model

```
width = 28 # width of the image in pixels
height = 28 # height of the image in pixels
flat = width * height # number of pixels in one image
class_output = 10 # number of possible classifications for the problem
```

Input and output

Create place holders for inputs and outputs

```
x  = tf.placeholder(tf.float32, shape=[None, flat])
y_ = tf.placeholder(tf.float32, shape=[None, class_output])
```

Converting images of the data set to tensors

The input image is 28 pixels by 28 pixels, 1 channel (grayscale). In this case, the first dimension is the **batch number** of the image, and can be of any size (so we set it to -1). The second and third dimensions are width and height, and the last one is the image channels.

```
x_image = tf.reshape(x, [-1,28,28,1])
x_image
```

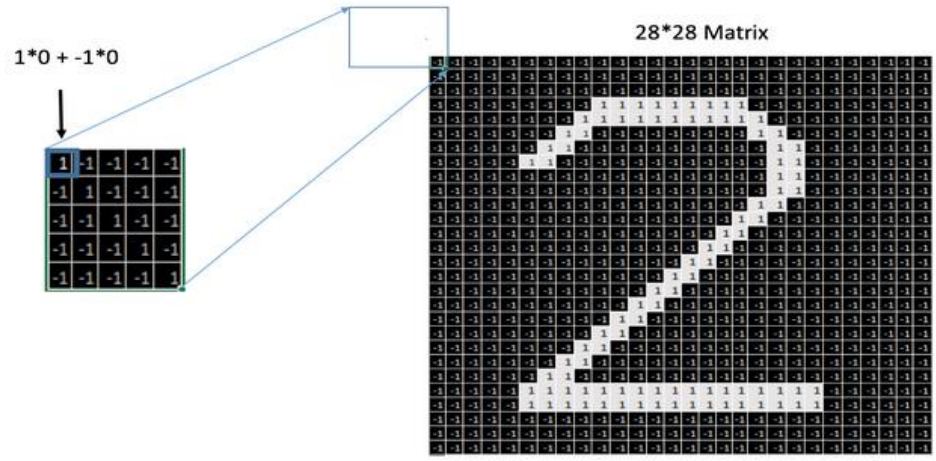
```
<tf.Tensor 'Reshape:0' shape=(?, 28, 28, 1) dtype=float32>
```

Convolutional Layer 1

Defining kernel weight and bias

We define a kernel here. The Size of the filter/kernel is 5x5; Input channels is 1 (grayscale); and we need 32 different feature maps (here, 32 feature maps means 32 different filters are applied on each image. So, the output of convolution layer would be 28x28x32). In this step, we create a filter / kernel tensor of shape `[filter_height, filter_width, in_channels, out_channels]`

```
W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
b_conv1 = tf.Variable(tf.constant(0.1, shape=[32])) # need 32 biases for 32 outputs
```



Convolve with weight tensor and add biases.
To create convolutional layer, we use **tf.nn.conv2d**. It computes a 2-D convolution given 4-D input and filter tensors.

Inputs:

- tensor of shape [batch, in_height, in_width, in_channels]. x of shape [batch_size,28 ,28, 1]
- a filter / kernel tensor of shape [filter_height, filter_width, in_channels, out_channels]. W is of size [5, 5, 1, 32]
- stride which is [1, 1, 1, 1]. The convolutional layer, slides the "kernel window" across the input tensor. As the input tensor has 4 dimensions: [batch, height, width, channels], then the convolution operates on a 2D window on the height and width dimensions. **strides** determines how much the window shifts by in each of the dimensions. As the first and last dimensions are related to batch and channels, we set the stride to 1. But for second and third dimension, we could set other values, e.g. [1, 2, 2, 1]

Process:

- Change the filter to a 2-D matrix with shape [5*5*1,32]
- Extracts image patches from the input tensor to form a *virtual* tensor of shape `[batch, 28, 28, 5*5*1]`.
- For each batch, right-multiplies the filter matrix and the image vector.

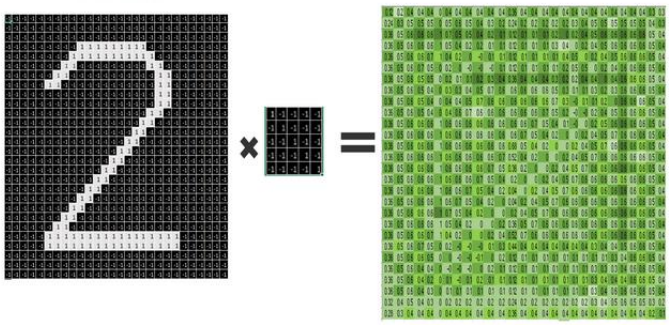
Output:

- A `Tensor` (a 2-D convolution) of size tf.Tensor 'add_7:0' shape=(?, 28, 28, 32)- Notice: the output of the first convolution layer is 32 [28x28] images. Here 32 is considered as volume/depth of the output image.

Output:

- A `Tensor` (a 2-D convolution) of size tf.Tensor 'add_7:0' shape=(?, 28, 28, 32)- Notice: the output of the first convolution layer is 32 [28x28] images. Here 32 is considered as volume/depth of the output image.

```
convolve1= tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_conv1
```



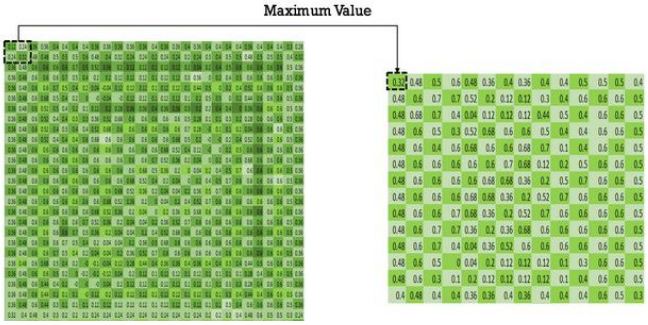
In this step, we just go through all outputs convolution layer, **convolve1**, and wherever a negative number occurs, we swap it out for a 0. It is called ReLU activation Function.
Let f(x) is a ReLU activation function $f(x) = max(0, x)$.

```
h_conv1 = tf.nn.relu(convolve1)
```

Apply the max pooling

max pooling is a form of non-linear down-sampling. It partitions the input image into a set of rectangles and, and then find the maximum value for that region.

Lets use **tf.nn.max_pool** function to perform max pooling. **Kernel size:** 2x2 (if the window is a 2x2 matrix, it would result in one output pixel)
Strides: dictates the sliding behaviour of the kernel. In this case it will move 2 pixels everytime, thus not overlapping. The input is a matrix of size 28x28x32, and the output would be a matrix of size 14x14x32.



```
conv1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME') #max_pool_2x2
conv1
```

```
<tf.Tensor 'MaxPool:0' shape=(?, 14, 14, 32) dtype=float32>
```

First layer completed

Convolutional Layer 2

Weights and Biases of kernels

We apply the convolution again in this layer. Lets look at the second layer kernel:

- Filter/kernel: 5x5 (25 pixels)
- Input channels: 32 (from the 1st Conv layer, we had 32 feature maps)
- 64 output feature maps

Notice: here, the input image is [14x14x32], the filter is [5x5x32], we use 64 filters of size [5x5x32], and the output of the convolutional layer would be 64 convolved image, [14x14x64].

Notice: the convolution result of applying a filter of size [5x5x32] on image of size [14x14x32] is an image of size [14x14x1], that is, the convolution is functioning on volume.

```
W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))
b_conv2 = tf.Variable(tf.constant(0.1, shape=[64])) #need 64 biases for 64 outputs
```

Convolve image with weight tensor and add biases.

```
convolve2= tf.nn.conv2d(conv1, W_conv2, strides=[1, 1, 1, 1], padding='SAME') + b_conv2
```

Apply the ReLU activation Function

```
h_conv2 = tf.nn.relu(convolve2)
```

Apply the max pooling

```
conv2 = tf.nn.max_pool(h_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME') #max_pool_2x2
conv2
```

```
<tf.Tensor 'MaxPool_1:0' shape=(?, 7, 7, 64) dtype=float32>
```

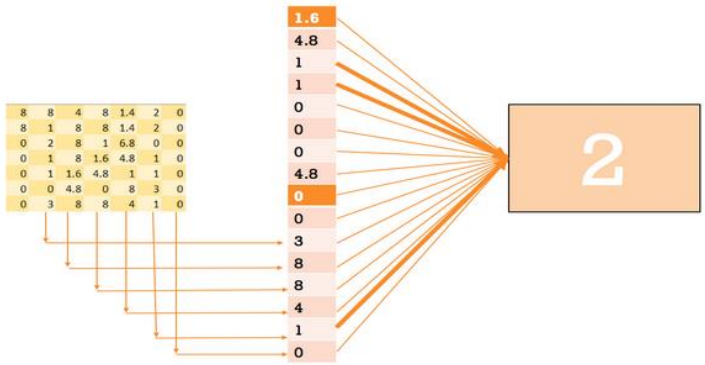
Second layer completed. So, what is the output of the second layer, layer2?

- it is 64 matrix of [7x7]

Fully Connected Layer

You need a fully connected layer to use the Softmax and create the probabilities in the end. Fully connected layers take the high-level filtered images from previous layer, that is all 64 matrices, and convert them to a flat array.

So, each matrix [7x7] will be converted to a matrix of [49x1], and then all of the 64 matrix will be connected, which make an array of size [3136x1]. We will connect it into another layer of size [1024x1]. So, the weight between these 2 layers will be [3136x1024]



Flattening Second Layer

```
layer2_matrix = tf.reshape(conv2, [-1, 7 * 7 * 64])
```

Weights and Biases between layer 2 and 3

Composition of the feature map from the last layer (7x7) multiplied by the number of feature maps (64); 1027 outputs to Softmax layer

```
W_fc1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1024], stddev=0.1))
b_fc1 = tf.Variable(tf.constant(0.1, shape=[1024])) # need 1024 biases for 1024 outputs
```

Matrix Multiplication (applying weights and biases)

```
fc1 = tf.matmul(layer2_matrix, W_fc1) + b_fc1
```

Apply the ReLU activation Function

```
h_fc1 = tf.nn.relu(fc1)
h_fc1
```

```
<tf.Tensor 'Relu_2:0' shape=(?, 1024) dtype=float32>
```

Dropout Layer, Optional phase for reducing overfitting

It is a phase where the network "forget" some features. At each training step in a mini-batch, some units get switched off randomly so that it will not interact with the network. That is, it weights cannot b updated, nor affect the learning of the other network nodes. This can be very useful for very large neural networks to prevent overfitting.

```
keep_prob = tf.placeholder(tf.float32)
layer_drop = tf.nn.dropout(h_fc1, keep_prob)
layer_drop
```

```
<tf.Tensor 'dropout/mul:0' shape=(?, 1024) dtype=float32>
```

Readout Layer (Softmax Layer)

Type: Softmax, Fully Connected Layer.

Weights and Biases

In last layer, CNN takes the high-level filtered images and translate them into votes using softmax. Input channels: 1024 (neurons from the 3rd Layer); 10 output features

```
W_fc2 = tf.Variable(tf.truncated_normal([1024, 10], stddev=0.1)) #1024 neurons
b_fc2 = tf.Variable(tf.constant(0.1, shape=[10])) # 10 possibilities for digits [0,1,2,3,4,5,6,7,8,9]
```

Matrix Multiplication (applying weights and biases)

```
fc=tf.matmul(layer_drop, W_fc2) + b_fc2
```

Apply the Softmax activation Function

softmax allows us to interpret the outputs of **fcl4** as probabilities. So, **y_conv** is a tensor of probabilities.

```
y_CNN= tf.nn.softmax(fc)
y_CNN
```

```
<tf.Tensor 'Softmax_1:0' shape=(?, 10) dtype=float32>
```

Summary of the Deep Convolutional Neural Network

Now is time to remember the structure of our network

0) Input - MNIST dataset

1) Convolutional and Max-Pooling

2) Convolutional and Max-Pooling

3) Fully Connected Layer

4) Processing - Dropout

5) Readout layer - Fully Connected

6) Outputs - Classified digits

Define functions and train the model

Define the loss function

We need to compare our output, layer4 tensor, with ground truth for all mini_batch. we can use **cross entropy to see how bad our CNN is working - to measure the error at a softmax layer.**

The following code shows an toy sample of cross-entropy for a mini-batch of size 2 which its items have been classified. You can run it (first change the cell type to **code** in the toolbar) to see how cross entropy changes.

```
import numpy as np
layer4_test =[[0.9, 0.1, 0.1],[0.9, 0.1, 0.1]]
y_test=[[1.0, 0.0, 0.0],[1.0, 0.0, 0.0]]
np.mean( -np.sum(y_test * np.log(layer4_test),1))
```

reduce_sum computes the sum of elements of **(y_ * tf.log(layer4)** across second dimension of the tensor, and **reduce_mean** computes the mean of all elements in the tensor..

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_CNN), reduction_indices=[1]))
```

Define the optimizer

It is obvious that we want minimize the error of our network which is calculated by cross_entropy metric. To solve the problem, we have to compute gradients for the loss (which is minimizing the cross-entropy) and apply gradients to variables. It will be done by an optimizer: GradientDescent or Adagrad.

```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

Define prediction

Do you want to know how many of the cases in a mini-batch has been classified correctly? lets count them.

```
correct_prediction = tf.equal(tf.argmax(y_CNN, 1), tf.argmax(y_, 1))
```

Define accuracy

It makes more sense to report accuracy using average of correct cases.

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Run session, train

```
sess.run(tf.global_variables_initializer())
```

*If you want a fast result (**it might take sometime to train it**)*

```
for i in range(1100):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1], keep_prob: 1.0})
        print("step %d, training accuracy %g"%(i, float(train_accuracy)))
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
```

```
step 0, training accuracy 0.12
step 100, training accuracy 0.82
step 200, training accuracy 0.94
step 300, training accuracy 0.9
step 400, training accuracy 0.92
step 500, training accuracy 0.96
step 600, training accuracy 0.94
step 700, training accuracy 0.96
step 800, training accuracy 0.98
step 900, training accuracy 0.96
```

You can run this cell if you REALLY have time to wait, or you are running it using PowerAI (change the type of the cell to code)

```
for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={
            x:batch[0], y_: batch[1], keep_prob: 1.0})
        print("step %d, training accuracy %g"%(i, train_accuracy))
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
```

PS. If you have problems running this notebook, please shutdown all your Jupyter runnning notebooks, clear all cells outputs and run each cell only after the completion of the previous cell.

Evaluate the model

Print the evaluation to the user

```
# evaluate in batches to avoid out-of-memory issues
n_batches = mnist.test.images.shape[0] // 50
cumulative_accuracy = 0.0
for index in range(n_batches):
    batch = mnist.test.next_batch(50)
    cumulative_accuracy += accuracy.eval(feed_dict={x: batch[0], y_: batch[1], keep_prob: 1.0})
print("test accuracy {}".format(cumulative_accuracy / n_batches))
```

Visualization

Do you want to look at all the filters?

```
kernels = sess.run(tf.reshape(tf.transpose(W_conv1, perm=[2, 3, 0,1]),[32, -1]))
```

```
!wget --output-document utils1.py http://deeplearning.net/tutorial/code/utils.py
import utils1
from utils1 import tile_raster_images
import matplotlib.pyplot as plt
from PIL import Image
%matplotlib inline
image = Image.fromarray(tile_raster_images(kernels, img_shape=(5, 5) ,tile_shape=(4, 8), tile_spacing=(1, 1)))
### Plot image
plt.rcParams['figure.figsize'] = (18.0, 18.0)
imgplot = plt.imshow(image)
imgplot.set_cmap('gray')
```

Do you want to see the output of an image passing through first convolution layer?

```
import numpy as np
plt.rcParams['figure.figsize'] = (5.0, 5.0)
sampleimage = mnist.test.images[1]
plt.imshow(np.reshape(sampleimage,[28,28]), cmap="gray")
```

```
ActivatedUnits = sess.run(convolve1,feed_dict={x:np.reshape(sampleimage,[1,784],order='F'),keep_prob:1.0})
filters = ActivatedUnits.shape[3]
plt.figure(1, figsize=(20,20))
n_columns = 6
n_rows = np.math.ceil(filters / n_columns) + 1
for i in range(filters):
    plt.subplot(n_rows, n_columns, i+1)
    plt.title('Filter ' + str(i))
    plt.imshow(ActivatedUnits[0,:,:,i], interpolation="nearest", cmap="gray")
```

```
sess.close() #finish the session
```

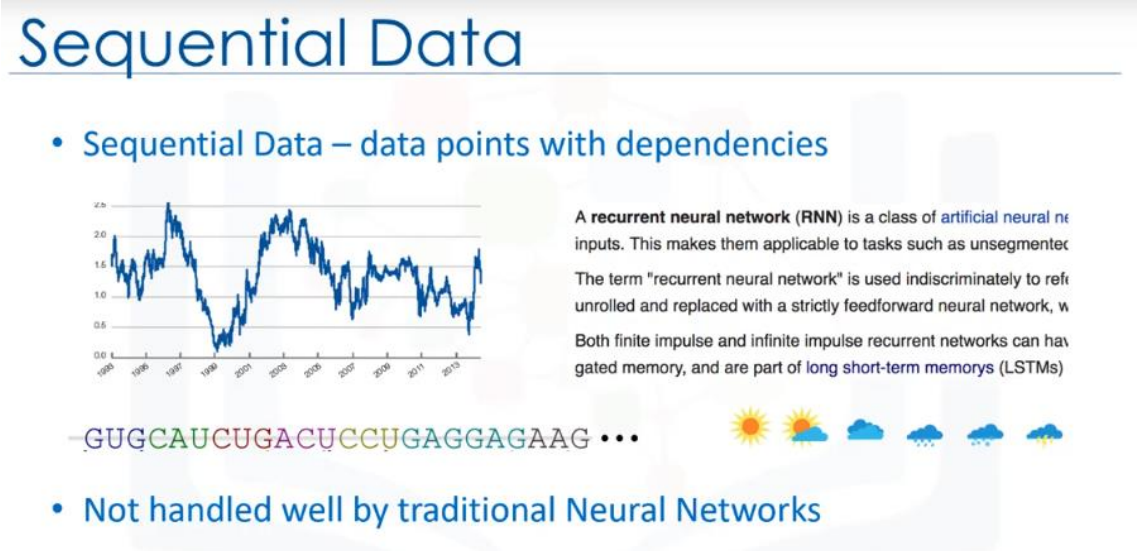
Want to learn more?

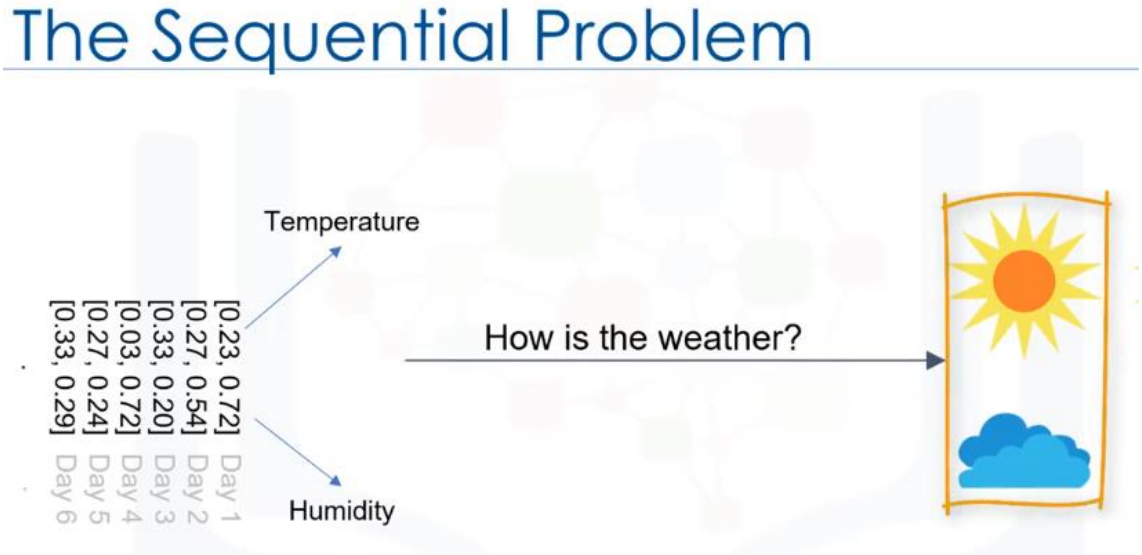
Running deep learning programs usually needs a high performance platform. **PowerAI** speeds up deep learning and AI. Built on IBM's Power Systems, **PowerAI** is a scalable software platform that accelerates deep learning and AI with blazing performance for individual users or enterprises. The **PowerAI** platform supports popular machine learning libraries and dependencies including TensorFlow, Caffe, Torch, and Theano. You can use PowerAI on IMB Cloud.

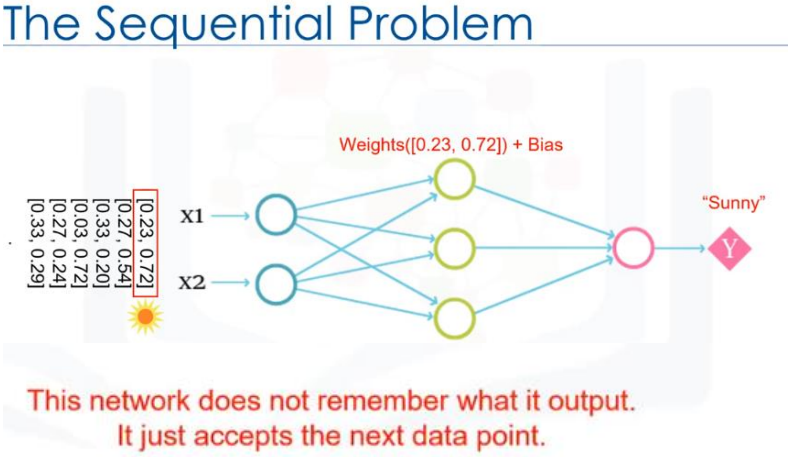
Also, you can use **Watson Studio** to run these notebooks faster with bigger datasets.**Watson Studio** is IBM's leading cloud solution for data scientists, built by data scientists. With Jupyter notebooks, RStudio, Apache Spark and popular libraries pre-packaged in the cloud, **Watson Studio** enables data scientists to collaborate on their projects without having to install anything. Join the fast-growing community of **Watson Studio** users today with a free account at Watson Studio.This is the end of this lesson. Thank you for reading this notebook, and good luck on your studies.

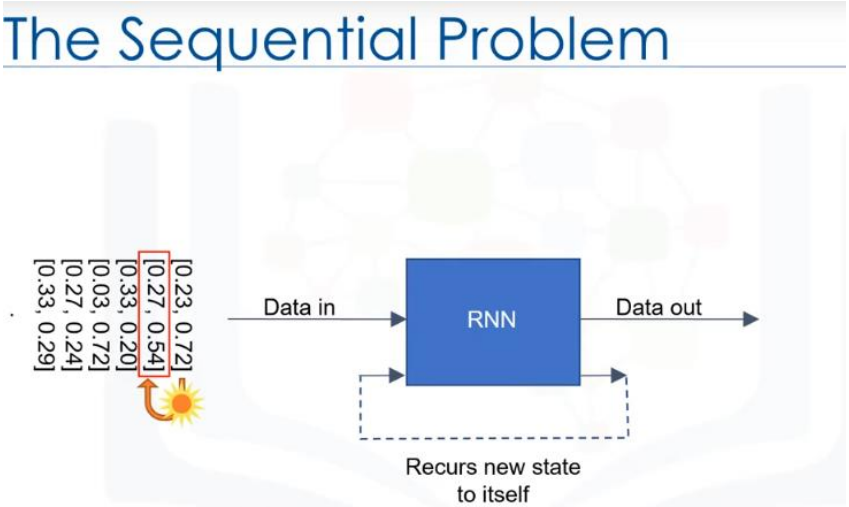
MODULE 3 – RECURRENT NEURAL NETWORK

THE SECUENCIAL PROBLEM





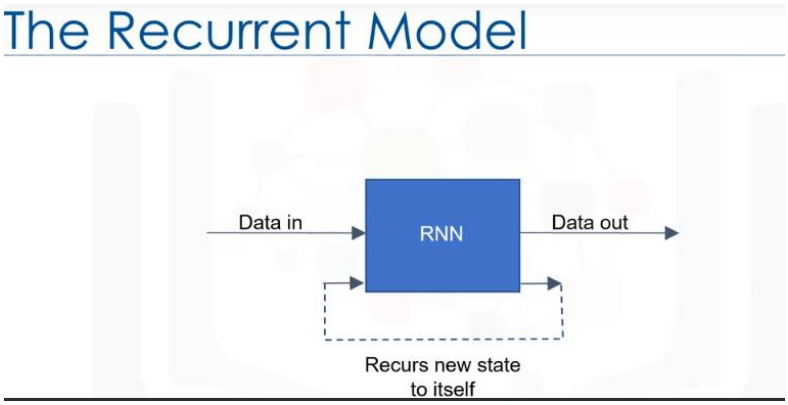


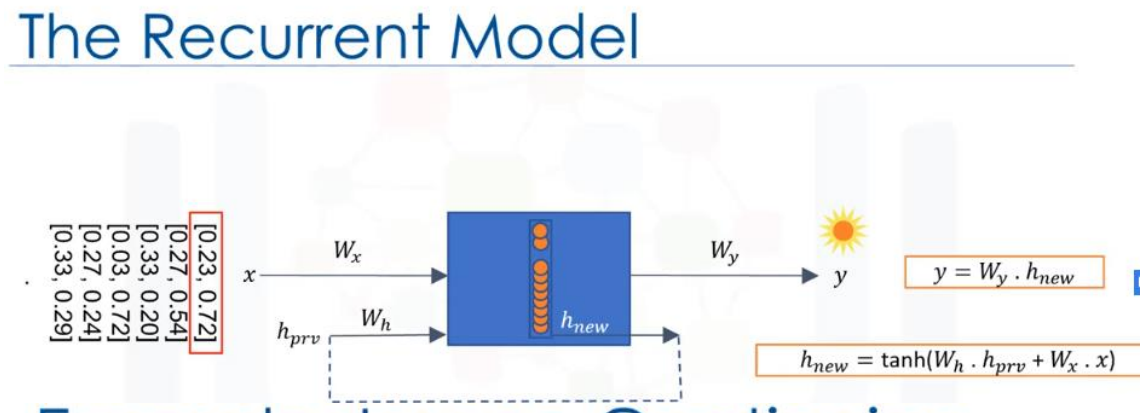


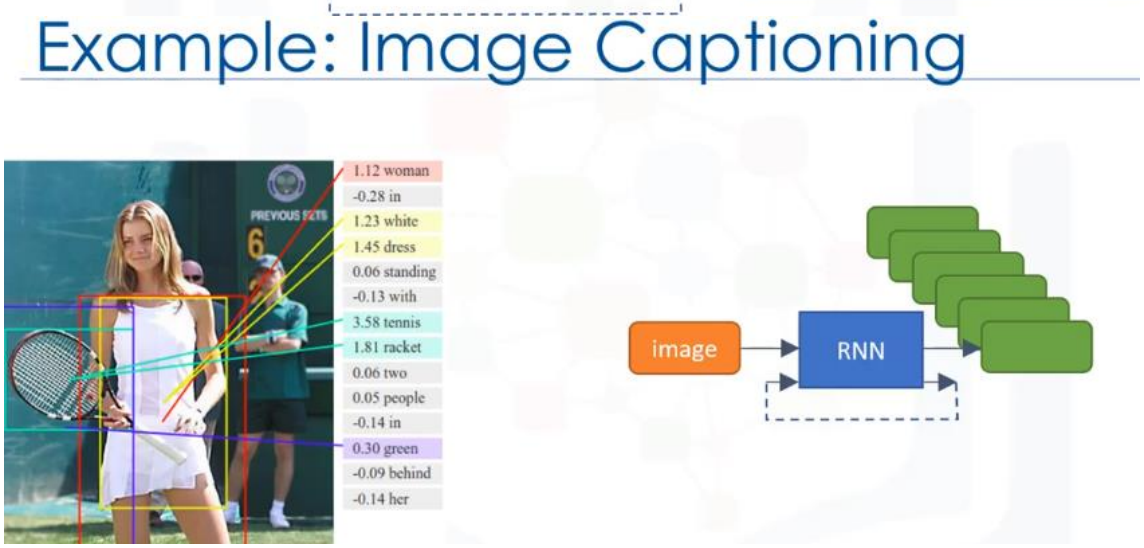
Hello, and welcome! In this video, we'll provide an overview of
sequential data, and explain why it poses a problem for traditional neural networks.
Whenever the points in a dataset are dependent on the other points, the data is said to be sequential. A common example of this is a time series,
such as a stock price or sensor data, where each data point represents an observation
at a certain point in time. There are other examples of sequential data,
like sentences, gene sequences, and weather data.
But traditional neural networks typically can't handle this type of data.
So, let's see why we can't use feedforward neural networks to analyze sequential data.
Let's consider a sequential problem to see
how well-suited a basic neural network might be.
Suppose we have a sequence of data that contains temperature and humidity values for every day. Our goal is to build a neural network that
imports the temperature and humidity values of a given day as input and output. For instance, to predict if the weather for that day is sunny or rainy.
This is a straightforward task for traditional feedforward neural networks.
Using our dataset, we first feed a data point into the input layer.
The data then flows to the hidden layer or layers, where the weights and biases are applied.
Then, the output layer classifies the results from the hidden layer, which ultimately produces the output of sunny or cloudy.
Of course, we can repeat this for the second day, and get the result.
However, it's important to note that the model does not remember the data that it just analyzed. All it does is accept input after input, and
produces individual classifications for every day.
In fact, a traditional neural network assumes that the data is non-sequential, and that each data point is independent of the others. As a result, the inputs are analyzed in isolation, which can cause problems if there are dependencies in the data.
To see how this can be a limitation, let's go back to the weather example again.
As you can imagine when examining weather, there is often a strong correlation of the weather on one day having some influence on the weather in subsequent days.
That is, if it was sunny on one day in the middle of summer, it's not unreasonable
to presume that it'll also be sunny on the following day.
A traditional neural network model does not use this information, however, so we'd have to turn to a different type of model, like a recurrent neural networks model.
A Recurrent Neural Network has a mechanism that can handle a sequential dataset.
By now, you should have a good understanding of the problem that the recurrent neural network

model is trying to address.
This concludes the video … thanks for watching.
End of transcript. Skip to the start.

THE RECURRENT MODEL







Start of transcript. Skip to the end.
Hello, and welcome! In this video, we'll provide an overview
of Recurrent Neural Networks, and explain their architecture.
A Recurrent neural network, or RNN for short, is a great tool for modeling sequential data.
The RNN is able to remember the analysis that was done up to a given point by maintaining
a state, or a context, so to speak. You can think of the "state" as the "memory"
of RNN, which captures information about what's been previously calculated.
This "state" recurs back into the net with each new input, which is where the network

gets its name.
Let's take a closer look at how this works. Imagine that the RNN we're using has only one hidden layer. The first data point flows into the network as input data, denoted as x. As we mentioned before, the hidden units, also receive the previous state or the context, denoted as h_prv, along with the input.
Then, in the hidden layer, two values will be calculated:
First, the new or updated state, denoted as h_new, is to be used for the next data point in the sequence. And second, the output of the network will be computed, which is denoted as y. The new state is a function of the previous state and the input data, as shown here. If this is the first data point, then some form of "initial state" is used, which will differ, depending on the type of data being analyzed. But, typically it is initialized to all zeroes.
Please notice that W_x, in this equation, is the weight matrix between the input and the hidden unit. And W_h are the weights that are multiplied by the previously hidden state in the equation. The output of the hidden unit is simply calculated by multiplication of the new hidden state and the output weight matrix.
So, after processing the first data point, in addition to the output, a new context is generated that represents the most recent point.
Then, this context is fed back into the net with the next data point, and we repeat these steps until all the data is processed.
Recurrent neural networks are extremely versatile and are used in a wide range of applications that deal with sequential data. One of these applications is speech recognition.
As you can see, it is a type of a many-to-many network.
That is, the goal is to consume a sequence of data and then produce another sequence.
Another application of RNN is image captioning. Although it's not purely recurrent, you can create a model that's capable of understanding the elements in an image.
Then, using the RNN, you can string the elements as words together to form a caption that describes the scene. Typically, RNN has outputs at each time step, but it depends on the problem that RNN is addressing.
For example, in this type of RNN for captioning, there is one input as image, and the output is a sequence of words. So, it is sometimes called one-to-many.
RNN can also be Many-to-one, that is, it consumes a sequence of data and produces just one output.
For example, to predict the stock market price, where we might only be interested in the price of a particular stock in tomorrow's market. Or, for sentiment analysis, where we may only care about the final output, not the sentiment after each word.
We've only covered a few applications, but variants of the recurrent models are continuing to solve increasingly complex problems, which are beyond the scope of this video.
Despite all of its strengths, the recurrent neural network is not a perfect model.
One issue is that the network needs to keep track of the states at any given time.
There could be many units of data, or many time steps, so this becomes computationally expensive. One compromise is to only store a portion of the recent states in a time window. Another issue is that Recurrent Neural Networks are extremely sensitive to changes in their parameters.
As a result, gradient descent optimizers may struggle to train the net.
Also, the net may suffer from the "Vanishing Gradient" problem, where the gradient drops to nearly zero and training slows to a halt. Finally, it may also suffer from the "Exploding Gradient", where the gradient grows exponentially off to infinity.

In either case, the model's capacity to learn will be diminished.

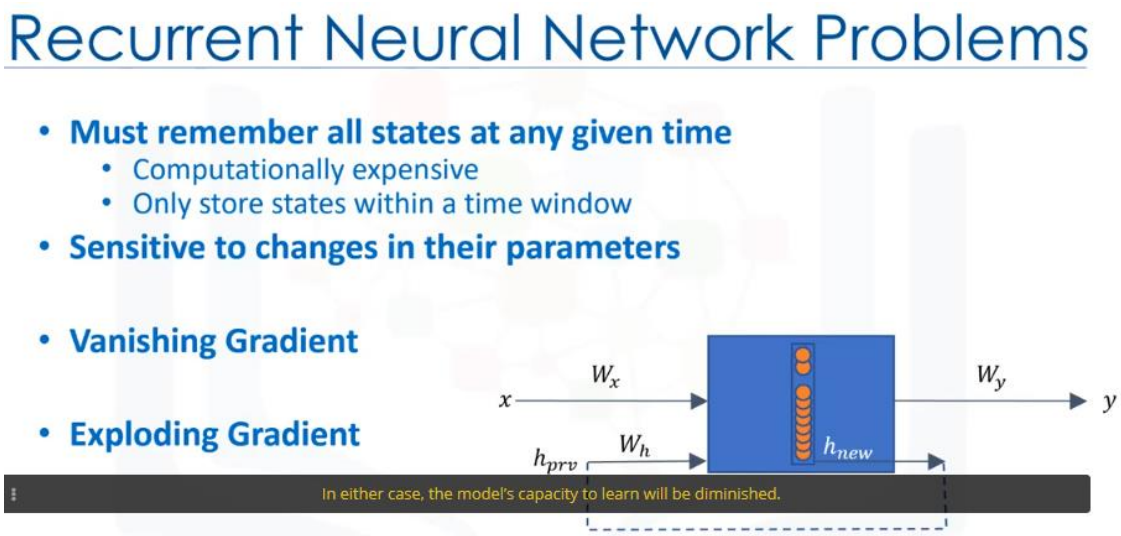
By now, you should have a good understanding of the main ideas behind the recurrent neural

network model.

Thanks for watching this video.

End of transcript. Skip to the start.

THE L



THE LSTM MODEL

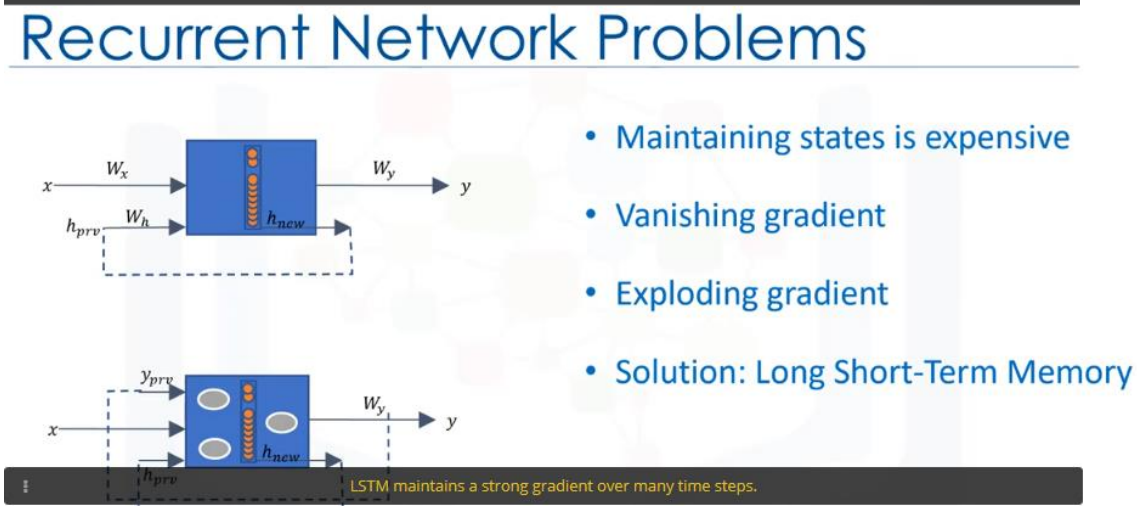
The recurrent neural network is a great tool for modeling sequential data, but there are

For example, recurrent nets are needed to keep track of states, which is computationally

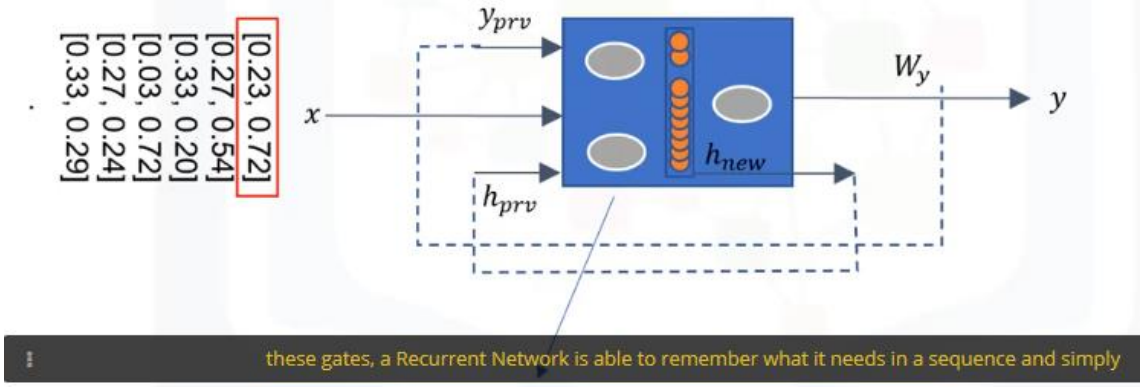
expensive. Also, there are issues with training, like

the vanishing gradient, and the exploding gradient.

As a result, the RNN, or to be precise, the "Vanilla RNN" cannot learn long sequences



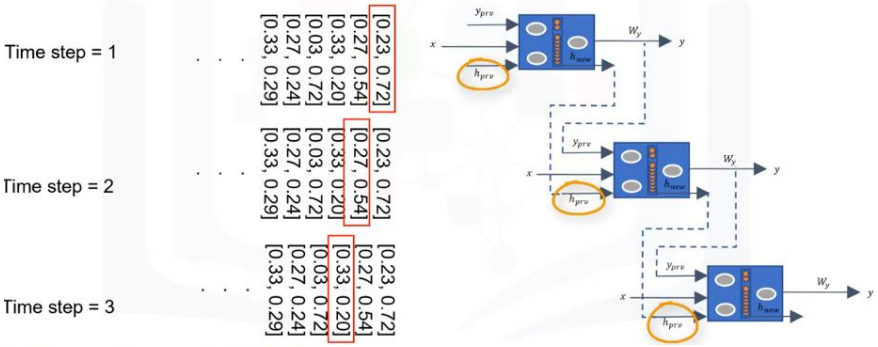
The LSTM Unit



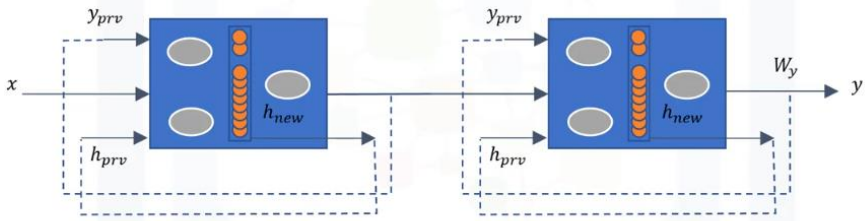
these gates, a Recurrent Network is able to remember what it needs in a sequence and simply

```
lstm_cell = tf.contrib.rnn.BasicLSTMCell(hidden_size)
```

Unfolded LSTM Network



Stacked LSTM

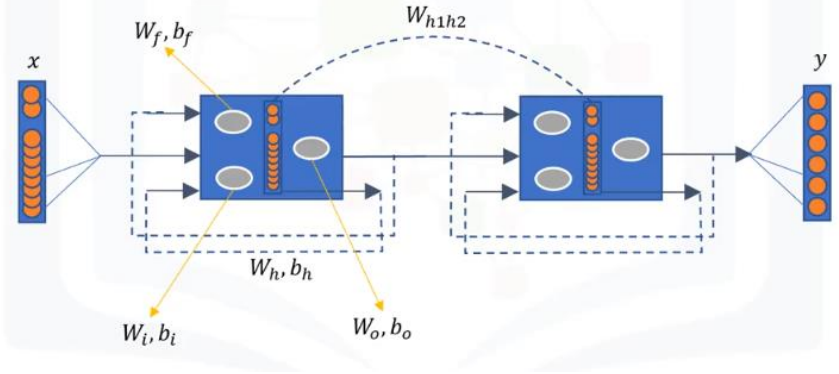



```
lstm_cell = tf.contrib.rnn.BasicLSTMCell(hidden_size)
```

```
stacked_lstm = tf.contrib.rnn.MultiRNNCell([lstm_cell] * 2)

output, state = tf.nn.dynamic_rnn(stacked_lstm, input_data)
```

IBM DEVELOPER SKILLS NETWORK

Training LSTM



Hello, and welcome. In this video, we'll be reviewing the Long Short-Term Memory Model.

The recurrent neural network is a great tool for modeling sequential data, but there are a few issues that need to be addressed in order to use the model on a large scale.

For example, recurrent nets are needed to keep track of states, which is computationally expensive. Also, there are issues with training, like the vanishing gradient, and the exploding gradient.

As a result, the RNN, or to be precise, the "Vanilla RNN" cannot learn long sequences very well. A popular method to solve these problems is a specific type of RNN, which is called the Long Short-Term memory, or LSTM for short.

LSTM maintains a strong gradient over many time steps.

This means you can train an LSTM with relatively long sequences.

An LSTM unit in Recurrent Neural Networks is composed of four main elements.

The "memory cell", and three logistic "gates".

The "memory cell" is responsible for holding data.

The write, read, and forget gates, define the flow of data inside the LSTM.

The Write Gate is responsible for writing data into the memory cell.

The Read Gate reads data from the memory cell and sends that data back to the recurrent network. And the Forget Gate maintains or deletes data from the information cell, or in other words, determines how much old information to forget.

In fact, these gates are the operations in the LSTM that execute some function on a linear combination of the inputs to the network, the network's previous hidden state and previous output. I don't want you to dive into the details of how these gates interact with each other, but what is important here, is that, by manipulating these gates, a Recurrent Network is able to remember what it needs in a sequence and simply forget what is no longer useful. Now, let's look at the data flow in LSTM Recurrent Networks.

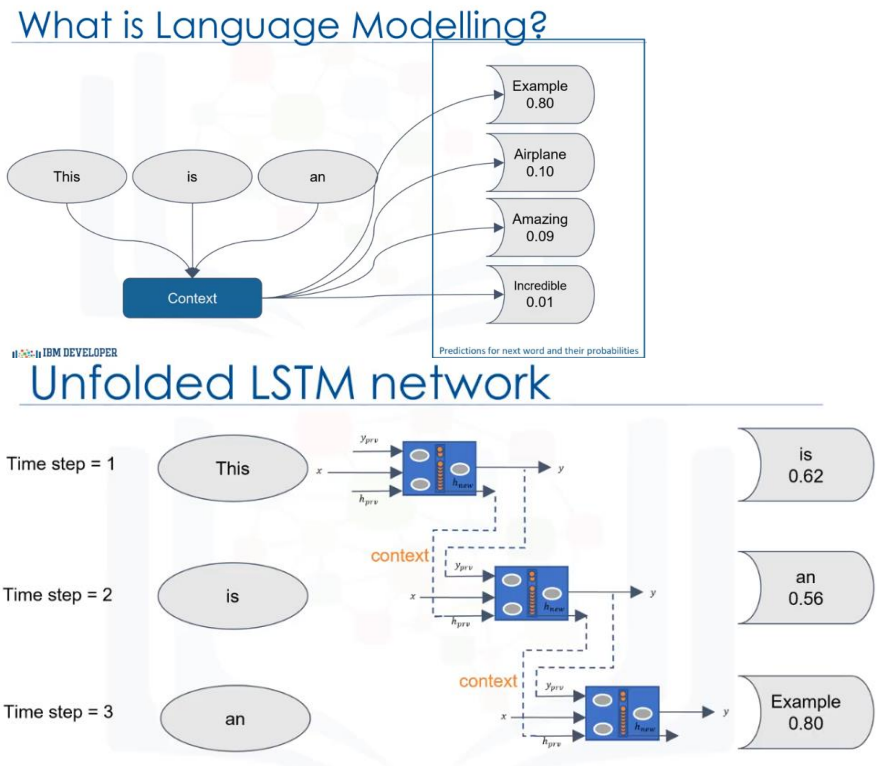
So, let's see how data is passed through the network.

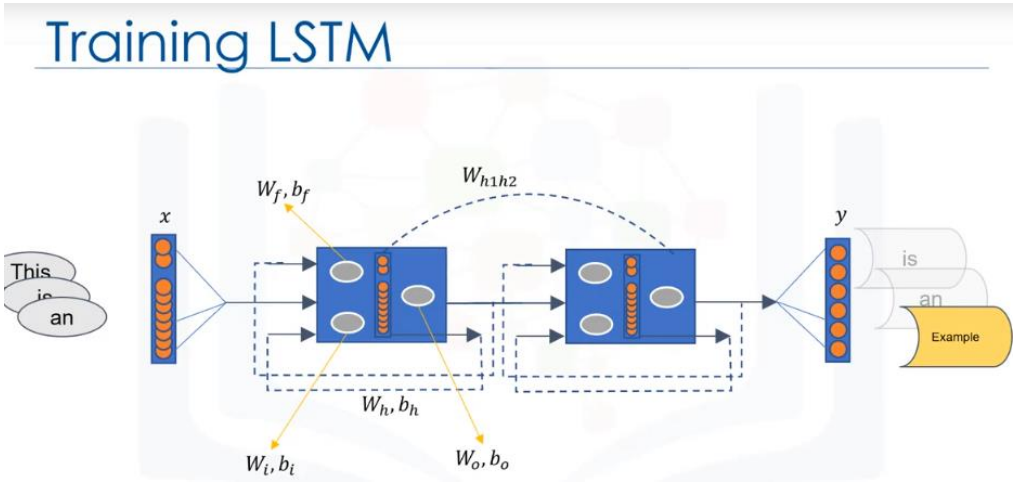
In the first time step, the first element of the sequence is passed to the network.

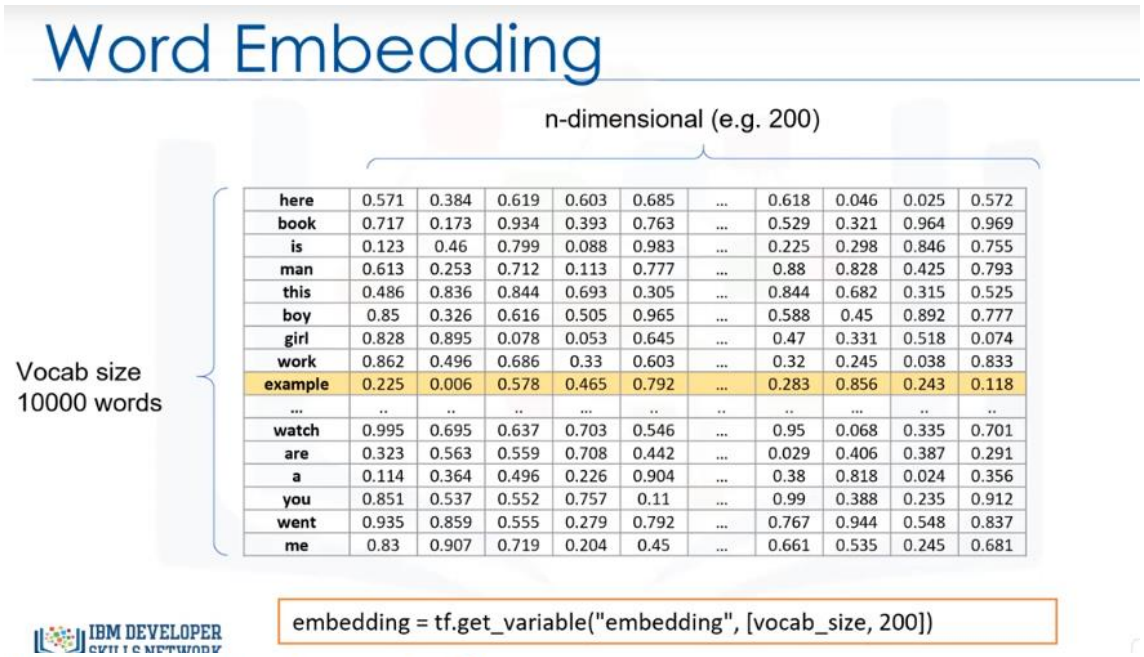
The LSTM unit uses the random initialized hidden state and output to produce the new hidden state and also the first step output. The LSTM then sends its output and hidden state to the net in the next time step. And the process continues for the next time

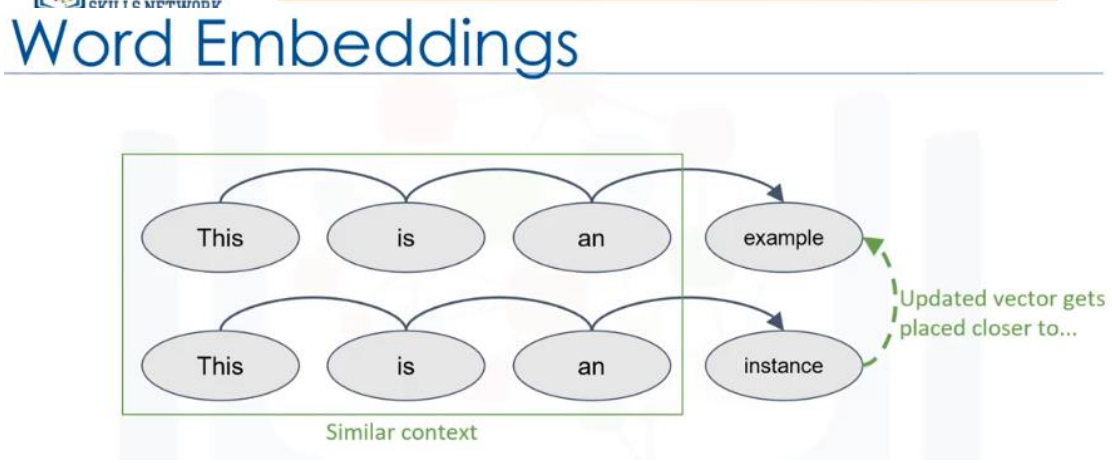
steps. So, as you can see, the LSTM unit keeps two
pieces of information as it propagates through time.
First, a hidden state, which is in fact, the memory the LSTM accumulates using its gates through time, and second, the previous time-step output.
The original LSTM model has only one single hidden LSTM layer.
But, as you know, in cases of simple feedforward neural network, we usually stack layers to create hierarchical feature representation of the input data.
So, does this also apply for LSTM's? What about if we want to have a RNN with stacked LSTM, for example, a 2-layer LSTM? In this case, the output of the first layer
will feed as the input into the second layer. Then, the second LSTM blends it with its own internal state to produce an output. Stacking an LSTM allows for greater model complexity. So, the second LSTM can create a more complex
feature representation of the current input. That is, stacking LSTM hidden layers makes the model deeper, and most probably leads to more accurate results.
Now, let's see what happens during the training process.
The network learns to determine how much old information to forget through the forget gate.
So, the weights, denoted as Wf, and biases denoted as bf will be learned through the training procedure. We also determine how much new information,
called x, to incorporate through the input gate and its weights.
We also calculate the new cell state based on the current and the previous internal state, so the network has to learn its corresponding weights and biases.
Finally, we determine how much of our cell state we want to output an output gate.
Basically, the network is learning the weights and biases used in each gate, for each layer.
By now, you should have a good understanding of the main ideas behind the LSTM.
Thanks for watching.
End of transcript. Skip to the start.

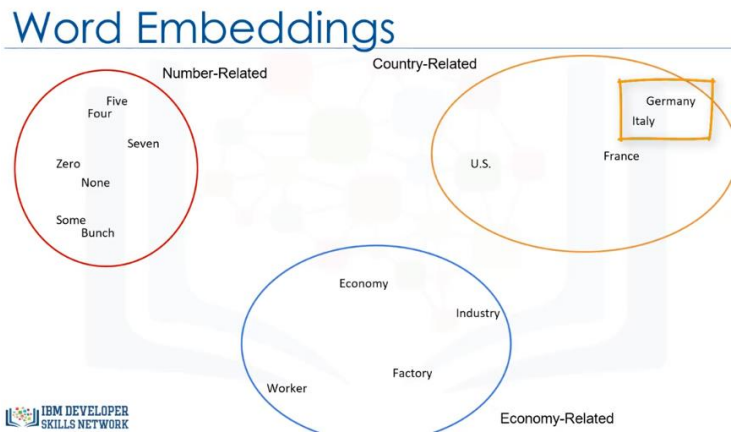
APPLYING RNN TO LANGUAGE MODELING

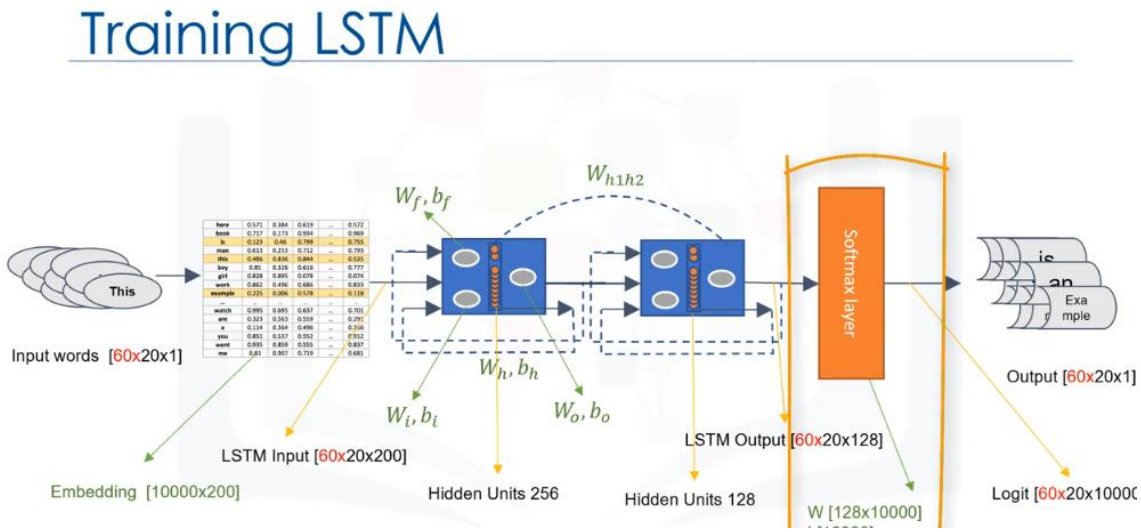



Training LSTM
W_f, b_f
W_{h1h2}
x
y
This
is
an
Example
W_h, b_h
W_i, b_i
W_o, b_o


Word Embedding
n-dimensional (e.g. 200)
Vocab size
10000 words
here 0.571 0.384 0.619 0.603 0.685 ... 0.618 0.046 0.025 0.572
book 0.717 0.173 0.934 0.393 0.763 ... 0.529 0.321 0.964 0.969
is 0.123 0.46 0.799 0.088 0.983 ... 0.225 0.298 0.846 0.755
man 0.613 0.253 0.712 0.113 0.777 ... 0.88 0.828 0.425 0.793
this 0.486 0.836 0.844 0.693 0.305 ... 0.844 0.682 0.315 0.525
boy 0.85 0.326 0.616 0.505 0.965 ... 0.588 0.45 0.892 0.777
girl 0.828 0.895 0.078 0.053 0.645 ... 0.47 0.331 0.518 0.074
work 0.862 0.496 0.686 0.33 0.603 ... 0.32 0.245 0.038 0.833
example 0.225 0.006 0.578 0.465 0.792 ... 0.283 0.856 0.243 0.118
watch 0.995 0.695 0.637 0.703 0.546 ... 0.95 0.068 0.335 0.701
are 0.323 0.563 0.559 0.708 0.442 ... 0.029 0.406 0.387 0.291
a 0.114 0.364 0.496 0.226 0.904 ... 0.38 0.818 0.024 0.356
you 0.851 0.537 0.552 0.757 0.11 ... 0.99 0.388 0.235 0.912
went 0.935 0.859 0.555 0.279 0.792 ... 0.767 0.944 0.548 0.837
me 0.83 0.907 0.719 0.204 0.45 ... 0.661 0.535 0.245 0.681
IBM DEVELOPER SKILLS NETWORK
embedding = tf.get_variable("embedding", [vocab_size, 200])


Word Embeddings
This
is
an
example
This
is
an
instance
Updated vector gets placed closer to...
Similar context





Start of transcript. Skip to the end.
Hello, and welcome. In this video, we'll be reviewing how to
apply recurrent neural networks to language modelling.
Language modelling is a gateway into many exciting deep learning applications like speech recognition, machine translation, and image captioning.
At its simplest, language modelling is the process of assigning probabilities to sequences of words. So for example, a language model could analyze
a sequence of words, and predict which word is most likely to follow.
So with the sequence, "This is an", which you see here, a language model might predict what the next word might be. Clearly, there are many options for what word
could be used as the next one in the string. But a trained model might predict, with an 80 percent probability of being correct, that the word "example" is most likely to follow.
This boils down to a sequential data analysis problem.
The sequence of words forms the context, and the most recent word is the input data.
Using these two pieces of information, you need to output both a predicted word, and a new context that contains the input word.
Recurrent neural networks are a great fit for this type of problem.
At the first time step, a recurrent net can receive a word as input along the initial context. It generates an output.
The output word with the current sequence of words as the context will then be re-fed into the network in the second time step. A new word would be predicted.
And, these steps are repeated until the sentence is complete.

Now, let's take closer look at an LSTM network for modeling the language.
In this network, we will use an RNN network with two stacked LSTM units.
For training such a network, we have to pass each word of the sentence to the network, and let the network generate an output. For example, after passing the words "this" and "is", if we pass the word "an" in the third time step, we expect the network to generate the word, "example," as output. But notice that we cannot easily pass a word to the network. We have to convert it into a vector of numbers somehow. We can use "word embedding" for this purpose.
Let's quickly examine what happens in word embedding.
An interesting way to process words is through a structure known as a Word Embedding.
A word embedding is an n-dimensional vector of real numbers for each word.
The vector is typically large, for example, 200 length.
You can see what that might look like with the word "example" here.
You think of word embedding as a type of encoding for text-to-numbers.
Now the question is, how do we find the proper values for these vectors?
In our RNN model, the vectors (also known as the matrix for the vocabulary) are initialized randomly for all the words that we are going to use for training.
Then, during the recurrent network's training, the vector values are updated based on the context into which the word is being inserted. So, words that are used in similar contexts end up with similar positions in the vector space.
This can be visualized by utilizing a dimensionality-reduction algorithm.
Take a look at the example shown here. After training the RNN, if we visualize the words based on their embedding vectors, the words are grouped together either because they're synonyms, or they're used in similar places within a sentence.
For example, the words "zero" and "none" are close semantically, so it's natural for them to be placed close together. And while "Italy" and "Germany" aren't synonyms, they can be interchanged in several sentences without distorting the grammar.
Ok, now let's look back at the RNN that we've been using.
Imagine that the input data is a batch with only one sequence of words.
Think of it as a batch that includes one sentence only - one that includes 20 words.
Assume that the vocabulary size of the words is 10,000 words, and the length of each embedding vector is 200. We have to look up those 20 words in the randomly initialized embedding matrix, and then feed them into the first LSTM unit.
Please notice that only one word in each time step is fed into the network, and one word would be the output. But, during 20 time steps, the output would be 20 words. In our network, we have 2 LSTM units, with arbitrary hidden sizes of 256 and 128. So, the output of the second LSTM unit would be a matrix of size 20-by-128. Now, we need a softmax layer to calculate the probability of the output words. It "squashes" the 128-dimensional vector of real values to a 10,000-dimensional vector, which is our vocabulary size.
This means that the output of the network at each time step is a probability vector of length 10,000. So, the output word is the one with maximum probability value in the vector. Now, we can compare the sequence of 20 output words with the ground truth words. And finally, calculate the discrepancy as a quantitative value, so called loss value, and back-propagate the errors into the network.
And of course, we will not train the model using only one sequence.
We will use a batch of sequences to train it and calculate the error.
So, instead of feeding one sequence, we can feed the network in many iterations - perhaps even a batch of 60 sentences, for example.
Now, the key question to be asked is: "What does the network learn when the error is propagated

back, in each iteration?" Well, as previously noted, the weights keep
updating based on the error of the network-in-training. First, the embedding matrix will be updated
in each iteration. Second, there are a bunch of weight matrices
related to the gates in the LSTM units which will be changed.
And finally, the weights related to the Softmax layer, which somehow plays the decoding role
for encoded words in the embedding layer. By now, you should have a good understanding
of how to use LSTM for language modeling.
Thanks for watching this video.
End of transcript. Skip to the start.

>>LAB

Lab LTSM BASICS

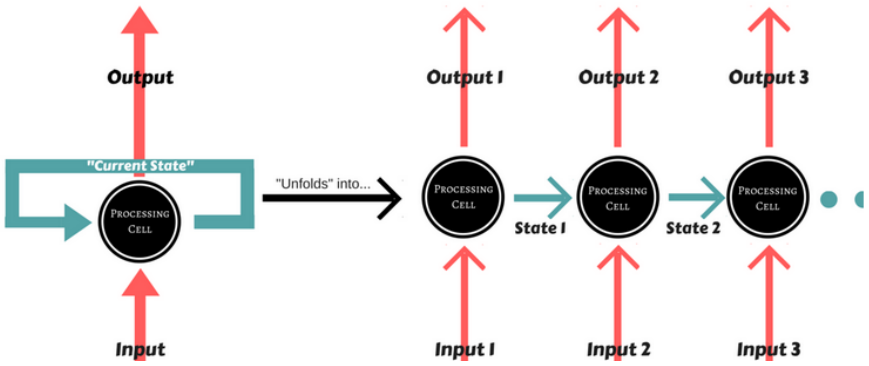
RECURRENT NETWORKS IN DEEP LEARNING

Hello and welcome to this notebook. In this notebook, we will go over concepts of the Long Short-Term Memory (LSTM) model, a refinement of the original Recurrent Neural Network model. By the end of this notebook, you should be able to understand the Long Short-Term Memory model, the benefits and problems it solves, and its inner workings and calculations.

Introduction

Recurrent Neural Networks are Deep Learning models with simple structures and a feedback mechanism built-in, or in different words, the output of a layer is added to the next input and fed back to the same layer.

The Recurrent Neural Network is a specialized type of Neural Network that solves the issue of **maintaining context for Sequential data** -- such as Weather data, Stocks, Genes, etc. At each iterative step, the processing unit takes in an input and the current state of the network, and produces an output and a new state that is **re-fed into the network**.



However, **this model has some problems**. It's very computationally expensive to maintain the state for a large amount of units, even more so over a long amount of time. Additionally, Recurrent Networks are very sensitive to changes in their parameters. As such, they are prone to different problems with their Gradient Descent optimizer -- they either grow exponentially (Exploding Gradient) or drop down to near zero and stabilize (Vanishing Gradient), both problems that greatly harm a model's learning capability.
To solve these problems, Hochreiter and Schmidhuber published a paper in 1997 describing a way to keep information over long periods of time and additionally solve the oversensitivity to parameter changes, i.e., make backpropagating through the Recurrent Networks more viable. This proposed method is called Long Short-Term Memory (LSTM).

(In this notebook, we will cover only LSTM and its implementation using TensorFlow)

Long Short-Term Memory Model

The Long Short-Term Memory, as it was called, was an abstraction of how computer memory works. It is "bundled" with whatever processing unit is implemented in the Recurrent Network, although outside of its flow, and is responsible for keeping, reading, and outputting information for the model. The way it works is simple: you have a linear unit, which is the information cell itself, surrounded by three logistic gates responsible for maintaining the data. One gate is for inputting data into the information cell, one is for outputting data from the input cell, and the last one is to keep or forget data depending on the needs of the network.

Thanks to that, it not only solves the problem of keeping states, because the network can choose to forget data whenever information is not needed, it also solves the gradient problems, since the Logistic Gates have a very nice derivative.

Long Short-Term Memory Architecture

The Long Short-Term Memory is composed of a linear unit surrounded by three logistic gates. The name for these gates vary from place to place, but the most usual names for them are:

- the "Input" or "Write" Gate, which handles the writing of data into the information cell
- the "Output" or "Read" Gate, which handles the sending of data back onto the Recurrent Network
- the "Keep" or "Forget" Gate, which handles the maintaining and modification of the data stored in the information cell

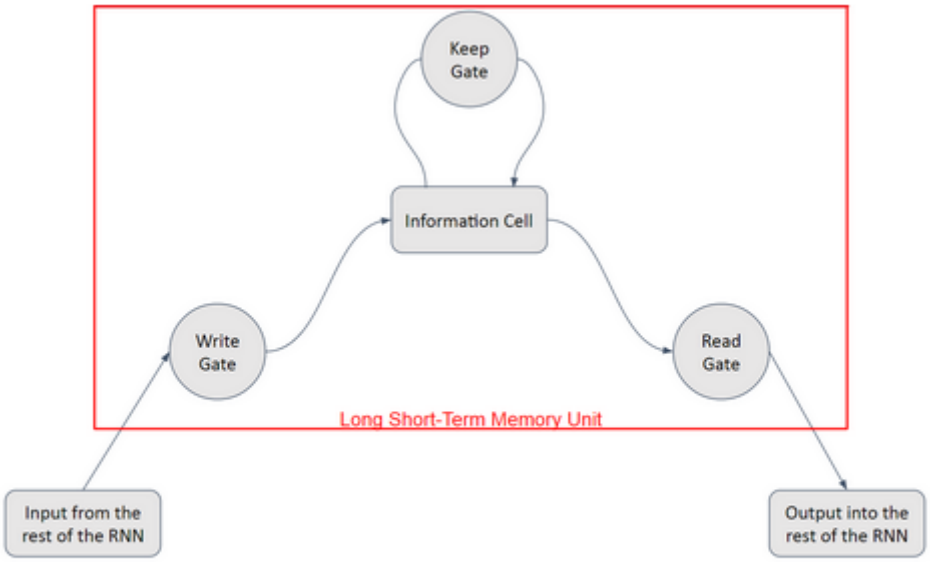


Diagram of the Long Short-Term Memory Unit

The three gates are the centerpiece of the LSTM unit. The gates, when activated by the network, perform their respective functions. For example, the Input Gate will write whatever data it is passed into the information cell, the Output Gate will return whatever data is in the information cell, and the Keep Gate will maintain the data in the information cell. These gates are analog and multiplicative, and as such, can modify the data based on the signal they are sent.

For example, an usual flow of operations for the LSTM unit is as such: First off, the Keep Gate has to decide whether to keep or forget the data currently stored in memory. It receives both the input and the state of the Recurrent Network, and passes it through its Sigmoid activation. If K_t has value of 1 means that the LSTM unit should keep the data stored perfectly and if K_t a value of 0 means that it should forget it entirely. Consider S_{t-1} as the incoming (previous) state, x_t as the incoming input, and W_k, B_k as the weight and bias for the Keep Gate. Additionally, consider Old_{t-1} as the data previously in memory. What happens can be summarized by this equation:

$$K_t = \sigma(W_k \times [S_{t-1}, x_t] + B_k)$$

$$Old_t = K_t \times Old_{t-1}$$

As you can see, Old_{t-1} was multiplied by value was returned by the Keep Gate(K_t) -- this value is written in the memory cell.

Then, the input and state are passed on to the Input Gate, in which there is another Sigmoid activation applied. Concurrently, the input is processed as normal by whatever processing unit is implemented in the network, and then multiplied by the Sigmoid activation's result I_t, much like the Keep Gate. Consider W_i and B_i as the weight and bias for the Input Gate, and C_t the result of the processing of the inputs by the Recurrent Network.

$$I_t = \sigma(W_i \times [S_{t-1}, x_t] + B_i)$$

$$New_t = I_t \times C_t$$

New_t is the new data to be input into the memory cell. This is then **added** to whatever value is still stored in memory.

$$Cell_t = Old_t + New_t$$

We now have the *candidate data* which is to be kept in the memory cell. The conjunction of the Keep and Input gates work in an analog manner, making it so that it is possible to keep part of the old data and add only part of the new data. Consider however, what would happen if the Forget Gate was set to 0 and the Input Gate was set to 1:

$$Old_t = 0 \times Old_{t-1}$$

$$New_t = 1 \times C_t$$

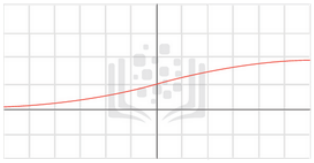
$$Cell_t = C_t$$

The old data would be totally forgotten and the new data would overwrite it completely.
The Output Gate functions in a similar manner. To decide what we should output, we take the input data and state and pass it through a Sigmoid function as usual. The contents of our memory cell, however, are pushed onto a *Tanh* function to bind them between a value of -1 to 1. Consider W_o and B_o as the weight and bias for the Output Gate.

$$O_t = \sigma(W_o \times [S_{t-1}, x_t] + B_o)$$

$$Output_t = O_t \times tanh(Cell_t)$$

And that $Output_t$ is what is output into the Recurrent Network.



The Logistic Function plotted

As mentioned many times, all three gates are logistic. The reason for this is because it is very easy to backpropagate through them, and as such, it is possible for the model to learn exactly _how_ it is supposed to use this structure. This is one of the reasons for which LSTM is a very strong structure. Additionally, this solves the gradient problems by being able to manipulate values through the gates themselves -- by passing the inputs and outputs through the gates, we have now a easily derivable function modifying our inputs.
In regards to the problem of storing many states over a long period of time, LSTM handles this perfectly by only keeping whatever information is necessary and forgetting it whenever it is not needed anymore. Therefore, LSTMs are a very elegant solution to both problems.

LSTM

Lets first create a tiny LSTM network sample to understand the architecture of LSTM networks.

We need to import the necessary modules for our code. We need `numpy` and `tensorflow`, obviously. Additionally, we can import directly the `tensorflow.contrib.rnn` model, which includes the function for building RNNs.

```
import numpy as np
import tensorflow as tf
sess = tf.Session()
```

We want to create a network that has only one LSTM cell. We have to pass 2 elements to LSTM, the **prv_output** and **prv_state**, so called, **h** and **c**. Therefore, we initialize a state vector, **state**. Here, **state** is a tuple with 2 elements, each one is of size [1 x 4], one for passing prv_output to next time step, and another for passing the prv_state to next time stamp.

```
LSTM_CELL_SIZE = 4  # output size (dimension), which is same as hidden size in the cell

lstm_cell = tf.contrib.rnn.BasicLSTMCell(LSTM_CELL_SIZE, state_is_tuple=True)
state = (tf.zeros([1,LSTM_CELL_SIZE]),)*2
state
```

```
(<tf.Tensor 'zeros:0' shape=(1, 4) dtype=float32>,
 <tf.Tensor 'zeros:0' shape=(1, 4) dtype=float32>)
```

Let define a sample input. In this example, batch_size = 1, and seq_len = 6:

```
sample_input = tf.constant([[3,2,2,2,2,2]],dtype=tf.float32)
print (sess.run(sample_input))
```

```
[[3. 2. 2. 2. 2. 2.]]
```

Now, we can pass the input to lstm_cell, and check the new state:

```
with tf.variable_scope("LSTM_sample1"):
    output, state_new = lstm_cell(sample_input, state)
sess.run(tf.global_variables_initializer())
print (sess.run(state_new))
```

```
LSTMStateTuple(c=array([[0.10281548, 0.23427807, 0.59578705, 0.26496097]], dtype=float32), h=array([[0.01619115, 0.19702487, 0.21722277, 0.09960523]], dtype=float32))
```

As we can see, the states has 2 parts, the new state c, and also the output h. Lets check the output again:

```
print (sess.run(output))
```

```
[[0.01619115 0.19702487 0.21722277 0.09960523]]
```

Stacked LSTM

What about if we want to have a RNN with stacked LSTM? For example, a 2-layer LSTM. In this case, the output of the first layer will become the input of the second.

Lets start with a new session:

```
sess = tf.Session()
```

```
input_dim = 6
```

Lets create the stacked LSTM cell:

```
cells = []
```

Creating the first layer LTSM cell.

```
LSTM_CELL_SIZE_1 = 4 #4 hidden nodes
cell1 = tf.contrib.rnn.LSTMCell(LSTM_CELL_SIZE_1)
cells.append(cell1)
```

Creating the second layer LTSM cell.

```
LSTM_CELL_SIZE_2 = 5 #5 hidden nodes
cell2 = tf.contrib.rnn.LSTMCell(LSTM_CELL_SIZE_2)
cells.append(cell2)
```

To create a multi-layer LTSM we use the **tf.contrib.rnnMultiRNNCell** function, it takes in multiple single layer LTSM cells to create a multilayer stacked LTSM model.

```
stacked_lstm = tf.contrib.rnn.MultiRNNCell(cells)
```

Now we can create the RNN from **stacked_lstm**:

```
# Batch size x time steps x features.
data = tf.placeholder(tf.float32, [None, None, input_dim])
output, state = tf.nn.dynamic_rnn(stacked_lstm, data, dtype=tf.float32)
```

Lets say the input sequence length is 3, and the dimensionality of the inputs is 6. The input should be a Tensor of shape: [batch_size, max_time, dimension], in our case it would be (2, 3, 6)

```
#Batch size x time steps x features.
sample_input = [[[1,2,3,4,3,2], [1,2,1,1,1,2],[1,2,2,2,2,2]],[[1,2,3,4,3,2],[3,2,2,1,1,2],[0,0,0,0,3,2]]]
sample_input
```

```
[[[1, 2, 3, 4, 3, 2], [1, 2, 1, 1, 1, 2], [1, 2, 2, 2, 2, 2]],
 [[1, 2, 3, 4, 3, 2], [3, 2, 2, 1, 1, 2], [0, 0, 0, 0, 3, 2]]]
```

we can now send our input to network, and check the output:

```
output
```

```
<tf.Tensor 'rnn/transpose_1:0' shape=(?, ?, 5) dtype=float32>
```

```
sess.run(tf.global_variables_initializer())
sess.run(output, feed_dict={data: sample_input})
```

```
array([[[ 0.00273579, -0.01136682, -0.01207429, -0.01471112,
         -0.0114937 ],
        [-0.00680167, -0.01081942, -0.01874172, -0.03130335,
         -0.0127462 ],
        [-0.01480113, -0.01198967, -0.02381499, -0.04589394,
         -0.01595225]],

       [[ 0.00273579, -0.01136682, -0.01207429, -0.01471112,
         -0.0114937 ],
        [-0.00697979, -0.00402291, -0.01300304, -0.01851586,
         -0.00319283],
        [-0.02692087, -0.02481141, -0.03956862, -0.06764094,
         -0.05375632]]], dtype=float32)
```

As you see, the output is of shape (2, 3, 5), which corresponds to our 2 batches, 3 elements in our sequence, and the dimensionality of the output which is 5.

Lab MNIST DATA CLASSIFICATION WITH RNN/LSTM

Building a LSTM with TensorFlow

LSTM for Classification

Although RNN is mostly used to model sequences and predict sequential data, we can still classify images using a LSTM network. If we consider every image row as a sequence of pixels, we can feed a LSTM network for classification. Lets use the famous MNIST dataset here. Because MNIST image shape is 28*28px, we will then handle 28 sequences of 28 steps for every sample.

MNIST Dataset

Tensor flow already provides **helper functions** to download and process the MNIST dataset.

```
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf
```

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

The function `input_data.read_data_sets(...)` loads the entire dataset and returns an object `tensorflow.contrib.learn.python.learn.datasets.mnist.DataSets`

The argument `( one_hot=False )` creates the label arrays as 10-dimensional binary vectors (only zeros and ones), in which the index cell for the number one, is the class label.

The argument (`one_hot=False`) creates the label arrays as 10-dimensional binary vectors (only zeros and ones), in which the index cell for the number one, is the class label.

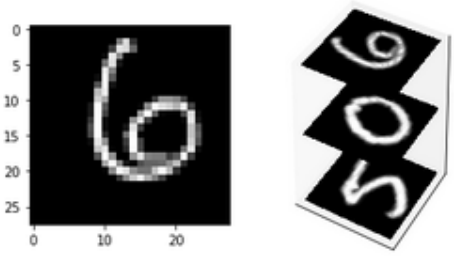
```
trainimgs = mnist.train.images
trainlabels = mnist.train.labels
testimgs = mnist.test.images
testlabels = mnist.test.labels

ntrain = trainimgs.shape[0]
ntest = testimgs.shape[0]
dim = trainimgs.shape[1]
nclasses = trainlabels.shape[1]
print ("Train Images: ", trainimgs.shape)
print ("Train Labels  ", trainlabels.shape)
print ("Test Images:  " , testimgs.shape)
print ("Test Labels:  ", testlabels.shape)
```

```
Train Images:  (55000, 784)
Train Labels   (55000, 10)
Test Images:   (10000, 784)
Test Labels:   (10000, 10)
```

Let's get one sample, just to understand the structure of MNIST dataset

The next code snippet prints the **label vector** (one_hot format), **the class** and actual sample formatted as **image**:

```
samplesIdx = [100, 101, 102]  #<-- You can change these numbers here to see other samples

from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()

ax1 = fig.add_subplot(121)
ax1.imshow(testimgs[samplesIdx[0]].reshape([28,28]), cmap='gray')


xx, yy = np.meshgrid(np.linspace(0,28,28), np.linspace(0,28,28))
X =  xx ; Y =  yy
Z =  100*np.ones(X.shape)

img = testimgs[77].reshape([28,28])
ax = fig.add_subplot(122, projection='3d')
ax.set_zlim((0,200))

offset=200
for i in samplesIdx:
    img = testimgs[i].reshape([28,28]).transpose()
    ax.contourf(X, Y, img, 200, zdir='z', offset=offset, cmap="gray")
    offset -= 100

    ax.set_xticks([])
ax.set_yticks([])
ax.set_zticks([])

plt.show()


for i in samplesIdx:
    print ("Sample: {0} - Class: {1} - Label Vector: {2} ".format(i, np.nonzero(testlabels[i])[0], testlabels[i]))
```




```
Sample: 100 - Class: [6] - Label Vector: [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
Sample: 101 - Class: [0] - Label Vector: [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Sample: 102 - Class: [5] - Label Vector: [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

Let's Understand the parameters, inputs and outputs

We will treat the MNIST image $\in \mathcal{R}^{28x28}$ as 28 sequences of a vector $\mathbf{x} \in \mathcal{R}^{28}$.

Our simple RNN consists of

1. One input layer which converts a $28 * 28$ dimensional input to an 128 dimensional hidden layer,
2. One intermediate recurrent neural network (LSTM)
3. One output layer which converts an 128 dimensional output of the LSTM to 10 dimensional output indicating a class label.

```
n_input = 28 # MNIST data input (img shape: 28*28)
n_steps = 28 # timesteps
n_hidden = 128 # hidden layer num of features
n_classes = 10 # MNIST total classes (0-9 digits)


learning_rate = 0.001
training_iters = 100000
batch_size = 100
display_step = 10
```

Construct a Recurrent Neural Network

The input should be a Tensor of shape: [batch_size, time_steps, input_dimension], but in our case it would be (?, 28, 28)

```
x = tf.placeholder(dtype="float", shape=[None, n_steps, n_input], name="x") # Current data input shape: (batch_size, n_steps, n_input) [100x28x28]
y = tf.placeholder(dtype="float", shape=[None, n_classes], name="y")
```

Lets create the weight and biases for the read out layer

```
weights = {
    'out': tf.Variable(tf.random_normal([n_hidden, n_classes]))
}
biases = {
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

Lets define a lstm cell with tensorflow

```
lstm_cell = tf.contrib.rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)
```

dynamic_rnn creates a recurrent neural network specified from **lstm_cell**:

```
outputs, states = tf.nn.dynamic_rnn(lstm_cell, inputs=x, dtype=tf.float32)
```

The output of the rnn would be a [100x28x128] matrix. we use the linear activation to map it to a [?x10 matrix]

```
output = tf.reshape(tf.split(outputs, 28, axis=1, num=None, name='split')[-1],[-1,128])
pred = tf.matmul(output, weights['out']) + biases['out']
```

labels and **logits** should be tensors of shape [100x10], lets check it out:

```
pred
```

```
<tf.Tensor 'add:0' shape=(?, 10) dtype=float32>
```

Now, we define the cost function and optimizer:

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

```
WARNING:tensorflow:From <ipython-input-12-901d12311984>:1: softmax_cross_entropy_with_logits (from tensorflow.python.ops.nn_ops) is deprecated and will be removed in a future version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.
```

Here we define the accuracy and evaluation methods to be used in the learning process:

```
correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

Just recall that we will treat the MNIST image $\in R^{28x28}$ as 28 sequences of a vector $\mathbf{x} \in R^{28}$.

```
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    step = 1
    # Keep training until reach max iterations
    while step * batch_size < training_iters:

        # We will read a batch of 100 images [100 x 784] as batch_x
        # batch_y is a matrix of [100x10]
        batch_x, batch_y = mnist.train.next_batch(batch_size)

        # We consider each row of the image as one sequence
        # Reshape data to get 28 seq of 28 elements, so that, batxh_x is [100x28x28]
        batch_x = batch_x.reshape((batch_size, n_steps, n_input))


        # Run optimization op (backprop)
        sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})


        if step % display_step == 0:
            # Calculate batch accuracy
            acc = sess.run(accuracy, feed_dict={x: batch_x, y: batch_y})
            # Calculate batch loss
            loss = sess.run(cost, feed_dict={x: batch_x, y: batch_y})
            print("Iter " + str(step*batch_size) + ", Minibatch Loss= " + \
                  "{:.6f}".format(loss) + ", Training Accuracy= " + \
                  "{:.5f}".format(acc))
        step += 1
    print("Optimization Finished!")
```

```
    # Calculate accuracy for 128 mnist test images
    test_len = 128
    test_data = mnist.test.images[:test_len].reshape((-1, n_steps, n_input))
    test_label = mnist.test.labels[:test_len]
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={x: test_data, y: test_label}))
```

```
Iter 1000, Minibatch Loss= 1.743172, Training Accuracy= 0.46000
Iter 2000, Minibatch Loss= 1.698113, Training Accuracy= 0.41000
Iter 3000, Minibatch Loss= 1.365828, Training Accuracy= 0.45000
Iter 4000, Minibatch Loss= 0.945430, Training Accuracy= 0.68000
Iter 5000, Minibatch Loss= 0.869455, Training Accuracy= 0.69000
Iter 6000, Minibatch Loss= 0.774082, Training Accuracy= 0.73000
Iter 7000, Minibatch Loss= 0.607809, Training Accuracy= 0.82000
Iter 8000, Minibatch Loss= 0.605622, Training Accuracy= 0.77000
Iter 9000, Minibatch Loss= 0.579154, Training Accuracy= 0.82000
Iter 10000, Minibatch Loss= 0.526041, Training Accuracy= 0.82000
Iter 11000, Minibatch Loss= 0.307974, Training Accuracy= 0.90000
Iter 12000, Minibatch Loss= 0.361602, Training Accuracy= 0.88000
Iter 13000, Minibatch Loss= 0.285182, Training Accuracy= 0.94000
Iter 14000, Minibatch Loss= 0.553690, Training Accuracy= 0.85000
Iter 15000, Minibatch Loss= 0.495894, Training Accuracy= 0.83000
Iter 16000, Minibatch Loss= 0.250558, Training Accuracy= 0.93000
Iter 17000, Minibatch Loss= 0.383072, Training Accuracy= 0.85000
Iter 18000, Minibatch Loss= 0.218994, Training Accuracy= 0.92000
Iter 19000, Minibatch Loss= 0.478271, Training Accuracy= 0.85000
Iter 20000, Minibatch Loss= 0.288323, Training Accuracy= 0.89000
Iter 21000, Minibatch Loss= 0.219856, Training Accuracy= 0.93000
Iter 22000, Minibatch Loss= 0.375026, Training Accuracy= 0.89000
Iter 23000, Minibatch Loss= 0.435833, Training Accuracy= 0.85000
Iter 24000, Minibatch Loss= 0.390350, Training Accuracy= 0.88000
```

```
sess.close()
```

Lab Applying RNN/LSTM TO LANGUAGE MODELLING

RECURRENT NETWORKS and LSTM IN DEEP LEARNING

Applying Recurrent Neural Networks/LSTM for Language Modeling

Hello and welcome to this part. In this notebook, we will go over the topic of Language Modelling, and create a Recurrent Neural Network model based on the Long Short-Term Memory unit to train and benchmark on the Penn Treebank dataset. By the end of this notebook, you should be able to understand how TensorFlow builds and executes a RNN model for Language Modelling.

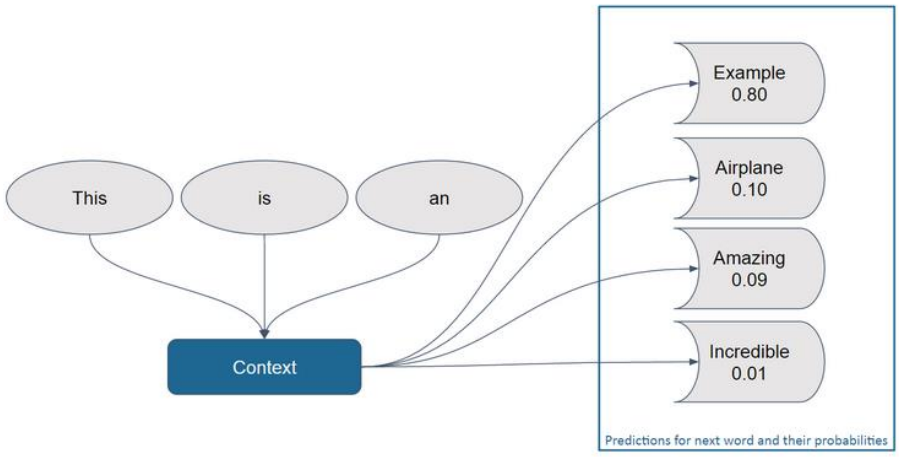
The Objective

By now, you should have an understanding of how Recurrent Networks work -- a specialized model to process sequential data by keeping track of the "state" or context. In this notebook, we go over a TensorFlow code snippet for creating a model focused on **Language Modelling** -- a very relevant task that is the cornerstone of many different linguistic problems such as **Speech Recognition, Machine Translation and Image Captioning**. For this, we will be using the Penn Treebank dataset, which is an often-used dataset for benchmarking Language Modelling models.

Table of Contents

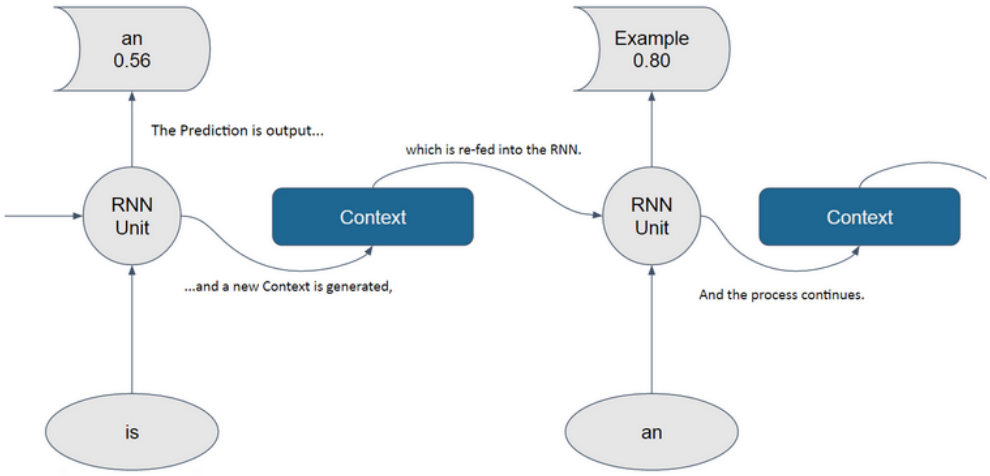
What exactly is Language Modelling?

Language Modelling, to put it simply, **is the task of assigning probabilities to sequences of words**. This means that, given a context of one or a sequence of words in the language the model was trained on, the model should provide the next most probable words or sequence of words that follows from the given sequence of words the sentence. Language Modelling is one of the most important tasks in Natural Language Processing.



Example of a sentence being predicted

In this example, one can see the predictions for the next word of a sentence, given the context "This is an". As you can see, this boils down to a sequential data analysis task -- you are given a word or sequence of words (the input data), and, given the context (the state), you need to find out what is the next word (the prediction). This kind of analysis is very important for language-related tasks such **Speech Recognition, Machine Translation, Image Captioning, Text Correction** and many other very relevant problems.



The above example is a schema of an RNN in execution

As the above image shows, Recurrent Network models fit this problem like a glove. Alongside LSTM and its capacity to maintain the model's state for over one thousand time steps, we have all the tools we need to undertake this problem. The goal for this notebook is to create a model that can reach **low levels of perplexity** on our desired dataset.
For Language Modelling problems, **perplexity** is the way to gauge efficiency. Perplexity is simply a measure of how well a probabilistic model is able to predict its sample. A higher-level way to explain this would be saying that **low perplexity means a higher degree of trust in the predictions the model makes**. Therefore, the lower perplexity is, the better.

The Penn Treebank dataset

Historically, datasets big enough for Natural Language Processing are hard to come by. This is in part due to the necessity of the sentences to be broken down and tagged with a certain degree of correctness -- or else the models trained on it won't be able to be correct at all. This means that we need a **large amount of data, annotated by or at least corrected by humans**. This is, of course, not an easy task at all.
The Penn Treebank, or PTB for short, is a dataset maintained by the University of Pennsylvania. It is *huge* -- there are over **four million and eight hundred thousand** annotated words in it, all corrected by humans. It is composed of many different sources, from abstracts of Department of Energy papers to texts from the Library of America. Since it is verifiably correct and of such a huge size, the Penn Treebank is commonly used as a benchmark dataset for Language Modelling.

The dataset is divided in different kinds of annotations, such as Piece-of-Speech, Syntactic and Semantic skeletons. For this example, we will simply use a sample of clean, non-annotated words (with the exception of one tag -- `<unk>` , which is used for rare words such as uncommon proper nouns) for our model. This means that we just want to predict what the next words would be, not what they mean in context or their classes on a given sentence.

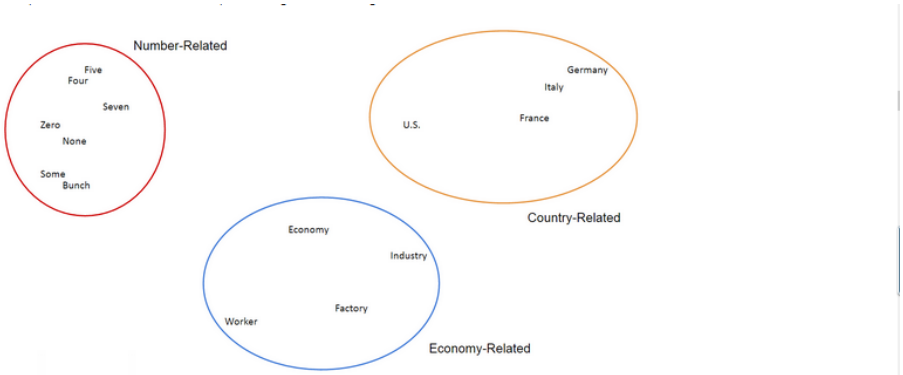
Example of text from the dataset we are going to use, **ptb.train**

Word Embeddings

For better processing, in this example, we will make use of **word embeddings**, which is **a way of representing sentence structures or words as n-dimensional vectors (where n is a reasonably high number, such as 200 or 500) of real numbers**. Basically, we will assign each word a randomly-initialized vector, and input those into the network to be processed. After a number of iterations, these vectors are expected to assume values that help the network to correctly predict what it needs to -- in our case, the probable next word in the sentence. This is shown to be a very effective task in Natural Language Processing, and is a commonplace practice.

$$Vec(\text{" Example "}) = [0.02, 0.00, 0.00, 0.92, 0.30, \ldots]$$

Word Embedding tends to group up similarly used words *reasonably* close together in the vectorial space. For example, if we use T-SNE (a dimensional reduction visualization algorithm) to flatten the dimensions of our vectors into a 2-dimensional space and plot these words in a 2-dimensional space, we might see something like this:



T-SNE Mockup with clusters marked for easier visualization

As you can see, words that are frequently used together, in place of each other, or in the same places as them tend to be grouped together -- being closer together the higher they are correlated. For example, "None" is pretty semantically close to "Zero", while a phrase that uses "Italy", you could probably also fit "Germany" in it, with little damage to the sentence structure. The vectorial "closeness" for similar words like this is a great indicator of a well-built model.

We need to import the necessary modules for our code. We need `numpy` and `tensorflow`, obviously. Additionally, we can import directly the `tensorflow.models.rnn` model, which includes the function for building RNNs, and `tensorflow.models.rnn.ptb.reader` which is the helper module for getting the input data from the dataset we just downloaded.

If you want to learn more take a look at https://github.com/tensorflow/models/blob/master/tutorials/rnn/ptb/reader.py

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

```
!mkdir data
!wget -q -O data/ptb.zip https://ibm.box.com/shared/static/z2yvmhbskc45xd2a9a4kkn6hg4g4kj5r.zip
!unzip -o data/ptb.zip -d data
!cp data/ptb/reader.py .

import reader
```

```
Archive:  data/ptb.zip
   creating: data/ptb/
  inflating: data/ptb/reader.py
   creating: data/__MACOSX/
   creating: data/__MACOSX/ptb/
  inflating: data/__MACOSX/ptb/._reader.py
  inflating: data/__MACOSX/._ptb
```

Building the LSTM model for Language Modeling

Now that we know exactly what we are doing, we can start building our model using TensorFlow. The very first thing we need to do is download and extract the `simple-examples` dataset, which can be done by executing the code cell below.

```
!wget http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz
!tar xzf simple-examples.tgz -C data/
```

```
--2020-05-20 13:21:22--  http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz
Resolving www.fit.vutbr.cz (www.fit.vutbr.cz)... 147.229.9.23, 2001:67c:1220:809::93e5:917
Connecting to www.fit.vutbr.cz (www.fit.vutbr.cz)|147.229.9.23|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 34869662 (33M) [application/x-gtar]
Saving to: 'simple-examples.tgz'

simple-examples.tgz 100%[===================>]  33.25M  3.47MB/s    in 11s

2020-05-20 13:21:33 (2.95 MB/s) - 'simple-examples.tgz' saved [34869662/34869662]
```

Additionally, for the sake of making it easy to play around with the model's hyperparameters, we can declare them beforehand. Feel free to change these -- you will see a difference in performance each time you change those!

Additionally, for the sake of making it easy to play around with the model's hyperparameters, we can declare them beforehand. Feel free to change these -- you will see a difference in performance each time you change those!

```
#Initial weight scale
init_scale = 0.1
#Initial Learning rate
learning_rate = 1.0
#Maximum permissible norm for the gradient (For gradient clipping -- another measure against Exploding Gradients)
max_grad_norm = 5
#The number of Layers in our model
num_layers = 2
#The total number of recurrence steps, also known as the number of layers when our RNN is "unfolded"
num_steps = 20
#The number of processing units (neurons) in the hidden layers
hidden_size_l1 = 256
hidden_size_l2 = 128
#The maximum number of epochs trained with the initial learning rate
max_epoch_decay_lr = 4
#The total number of epochs in training
max_epoch = 15
#The probability for keeping data in the Dropout Layer (This is an optimization, but is outside our scope for this notebook!)
#At 1, we ignore the Dropout Layer wrapping.
keep_prob = 1
#The decay for the learning rate
decay = 0.5
#The size for each batch of data
batch_size = 60
#The size of our vocabulary
vocab_size = 10000
embeding_vector_size = 200
#Training flag to separate training from testing
is_training = 1
#Data directory for our dataset
data_dir = "data/simple-examples/data/"
```

Some clarifications for LSTM architecture based on the arguments:

Network structure:

- In this network, the number of LSTM cells are 2. To give the model more expressive power, we can add multiple layers of LSTMs to process the data. The output of the first layer will become the input of the second and so on.
- The recurrence steps is 20, that is, when our RNN is "Unfolded", the recurrence step is 20.
- the structure is like:
  - 200 input units -> [200x200] Weight -> 200 Hidden units (first layer) -> [200x200] Weight matrix -> 200 Hidden units (second layer) -> [200] weight Matrix -> 200 unit output

Input layer:

- The network has 200 input units.
- Suppose each word is represented by an embedding vector of dimensionality e=200. The input layer of each cell will have 200 linear units. These e=200 linear units are connected to each of the h=200 LSTM units in the hidden layer (assuming there is only one hidden layer, though our case has 2 layers).
- The input shape is [batch_size, num_steps], that is [30x20]. It will turn into [30x20x200] after embedding, and then 20x[30x200]

Hidden layer:

- Each LSTM has 200 hidden units which is equivalent to the dimensionality of the embedding words and output.

There is a lot to be done and a ton of information to process at the same time, so go over this code slowly. It may seem complex at first, but if you try to apply what you just learned about language modelling to the code you see, you should be able to understand it.

This code is adapted from the PTBModel example bundled with the TensorFlow source code.

Train data

Train data

The story starts from data:

- Train data is a list of words, of size 929589, represented by numbers, e.g. [9971, 9972, 9974, 9975,...]
- We read data as mini-batch of size b=30. Assume the size of each sentence is 20 words (num_steps = 20). Then it will take

$$floor(\frac{N}{b \times h}) + 1 = 1548$$

  iterations for the learner to go through all sentences once. Where N is the size of the list of words, b is batch size, andh is size of each sentence. So, the number of iterators is 1548
- Each batch data is read from train dataset of size 600, and shape of [30x20]

First we start an interactive session:

```
session = tf.InteractiveSession()
```

```
# Reads the data and separates it into training data, validation data and testing data
raw_data = reader.ptb_raw_data(data_dir)
train_data, valid_data, test_data, vocab, word_to_id = raw_data
```

```
len(train_data)
```

```
929589
```

```
def id_to_word(id_list):
    line = []
    for w in id_list:
        for word, wid in word_to_id.items():
            if wid == w:
                line.append(word)
    return line


print(id_to_word(train_data[0:100]))
```

```
['aer', 'banknote', 'berlitz', 'calloway', 'centrust', 'cluett', 'fromstein', 'gitano', 'guterman', 'hydro-quebec', 'ipo', 'kia', 'memotec', 'mlx', 'nahb', 'punts', 'rake', 'regatta', 'ru
bens', 'sim', 'snack-food', 'ssangyong', 'swapo', 'wachter', '<eos>', 'pierre', '<unk>', 'N', 'years', 'old', 'will', 'join', 'the', 'board', 'as', 'a', 'nonexecutive', 'director', 'no
v.', 'N', '<eos>', 'mr.', '<unk>', 'is', 'chairman', 'of', '<unk>', 'n.v.', 'the', 'dutch', 'publishing', 'group', '<eos>', 'rudolph', '<unk>', 'N', 'years', 'old', 'and', 'former', 'chai
rman', 'of', 'consolidated', 'gold', 'fields', 'plc', 'was', 'named', 'a', 'nonexecutive', 'director', 'of', 'this', 'british', 'industrial', 'conglomerate', '<eos>', 'a', 'form', 'of', '
asbestos', 'once', 'used', 'to', 'make', 'kent', 'cigarette', 'filters', 'has', 'caused', 'a', 'high', 'percentage', 'of', 'cancer', 'deaths', 'among', 'a', 'group', 'of']
```

Lets just read one mini-batch now and feed our network:

```
itera = reader.ptb_iterator(train_data, batch_size, num_steps)
first_touple = itera.__next__()
x = first_touple[0]
y = first_touple[1]
```

```
x.shape
```

```
(60, 20)
```

Lets look at 3 sentences of our input x:

```
x[0:3]
```

```
array([[9970, 9971, 9972, 9974, 9975, 9976, 9980, 9981, 9982, 9983, 9984,
        9986, 9987, 9988, 9989, 9991, 9992, 9993, 9994, 9995],
       [ 901,   33, 3361,    8, 1279,  437,  597,    6,  261, 4276, 1089,
            8, 2836,    2,  269,    4, 5526,  241,   13, 2420],
       [2654,    6,  334, 2886,    4,    1,  233,  711,  834,   11,  130,
          123,    7,  514,    2,   63,   10,  514,    8,  605]],
      dtype=int32)
```

we define 2 place holders to feed them with mini-batchs, that is x and y:

```
_input_data = tf.placeholder(tf.int32, [batch_size, num_steps]) #[30#20]
_targets = tf.placeholder(tf.int32, [batch_size, num_steps]) #[30#20]
```

Lets define a dictionary, and use it later to feed the placeholders with our first mini-batch:

```
feed_dict = {_input_data:x, _targets:y}
```

For example, we can use it to feed `_input_data`:

```
session.run(_input_data, feed_dict)
```

```
array([[9970, 9971, 9972, ..., 9993, 9994, 9995],
       [ 901,   33, 3361, ...,  241,   13, 2420],
       [2654,    6,  334, ...,  514,    8,  605],
       ...,
       [7831,   36, 1678, ...,    4, 4558,  157],
       [  59, 2070, 2433, ...,  400,    1, 1173],
       [2097,    3,    2, ..., 2043,   23,    1]], dtype=int32)
```

In this step, we create the stacked LSTM, which is a 2 layer LSTM network:

Embeddings

We have to convert the words in our dataset to vectors of numbers. The traditional approach is to use one-hot encoding method that is usually used for converting categorical values to numerical values. However, One-hot encoded vectors are high-dimensional, sparse and in a big dataset, computationally inefficient. So, we use word2vec approach. It is, in fact, a layer in our LSTM network, where the word IDs will be represented as a dense representation before feeding to the LSTM.
The embedded vectors also get updated during the training process of the deep neural network. We create the embeddings for our input data. **embedding_vocab** is matrix of [10000x200] for all 10000 unique words.

```
embedding_vocab = tf.get_variable("embedding_vocab", [vocab_size, embeding_vector_size])  #[10000x200]
```

Lets initialize the `embedding_words` variable with random values.

```
session.run(tf.global_variables_initializer())
session.run(embedding_vocab)
```

```
array([[ 0.01853731, -0.01791934,  0.00593955, ...,  0.00209342,
         0.00758219,  0.00134525],
       [ 0.00353362, -0.00462071, -0.02147955, ...,  0.01936992,
        -0.02421559,  0.01475692],
       [ 0.02034229, -0.00634807,  0.01169099, ..., -0.00526239,
         0.01504254,  0.02064854],
       ...,
       [ 0.00452643, -0.01923032,  0.00048812, ...,  0.0016771 ,
        -0.01578872, -0.00563391],
       [ 0.003683  , -0.0212964 ,  0.00465553, ...,  0.01079313,
        -0.01647567,  0.01402071],
       [-0.0086471 ,  0.00100506,  0.00970191, ..., -0.01175196,
         0.02256839,  0.01586104]], dtype=float32)
```

embedding_lookup() finds the embedded values for our batch of 30x20 words. It goes to each row of `input_data`, and for each word in the row/sentence, finds the correspond vector in `embedding_dic`.

`It creates a [30x20x200] tensor, so, the first element of inputs (the first sentence), is a matrix of 20x200, which each row of it, is vector representing a word in the sentence.`

```
lstm_cell_l1 = tf.contrib.rnn.BasicLSTMCell(hidden_size_l1, forget_bias=0.0)
lstm_cell_l2 = tf.contrib.rnn.BasicLSTMCell(hidden_size_l2, forget_bias=0.0)
stacked_lstm = tf.contrib.rnn.MultiRNNCell([lstm_cell_l1, lstm_cell_l2])
```

Also, we initialize the states of the nework:

_initial_state

For each LCTM, there are 2 state matrices, c_state and m_state. c_state and m_state represent "Memory State" and "Cell State". Each hidden layer, has a vector of size 30, which keeps the states. so, for 200 hidden units in each LSTM, we have a matrix of size [30x200]

```
_initial_state = stacked_lstm.zero_state(batch_size, tf.float32)
_initial_state
```

```
(LSTMStateTuple(c=<tf.Tensor 'MultiRNNCellZeroState/BasicLSTMCellZeroState/zeros:0' shape=(60, 256) dtype=float32>, h=<tf.Tensor 'MultiRNNCellZeroState/BasicLSTMCellZeroState/zeros_1:0' shape=(60, 256) dtype=float32>),
 LSTMStateTuple(c=<tf.Tensor 'MultiRNNCellZeroState/BasicLSTMCellZeroState_1/zeros:0' shape=(60, 128) dtype=float32>, h=<tf.Tensor 'MultiRNNCellZeroState/BasicLSTMCellZeroState_1/zeros_1:0' shape=(60, 128) dtype=float32>))
```

Lets look at the states, though they are all zero for now:

```
session.run(_initial_state, feed_dict)
```

```
(LSTMStateTuple(c=array([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]], dtype=float32), h=array([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)),
```

Embeddings

We have to convert the words in our dataset to vectors of numbers. The traditional approach is to use one-hot encoding method that is usually used for converting categorical values to numerical values. However, One-hot encoded vectors are high-dimensional, sparse and in a big dataset, computationally inefficient. So, we use word2vec approach. It is, in fact, a layer in our LSTM network, where the word IDs will be represented as a dense representation before feeding to the LSTM.
The embedded vectors also get updated during the training process of the deep neural network. We create the embeddings for our input data. **embedding_vocab** is matrix of [10000x200] for all 10000 unique words.

```
embedding_vocab = tf.get_variable("embedding_vocab", [vocab_size, embeding_vector_size])  #[10000x200]
```

Lets initialize the `embedding_words` variable with random values.

```
session.run(tf.global_variables_initializer())
session.run(embedding_vocab)
```

```
array([[ 0.01853731, -0.01791934,  0.00593955, ...,  0.00209342,
         0.00758219,  0.00134525],
       [ 0.00353362, -0.00462071, -0.02147955, ...,  0.01936992,
        -0.02421559,  0.01475692],
       [ 0.02034229, -0.00634807,  0.01169099, ..., -0.00526239,
         0.01504254,  0.02064854],
       ...,
       [ 0.00452643, -0.01923032,  0.00048812, ...,  0.0016771 ,
        -0.01578872, -0.00563391],
       [ 0.003683  , -0.0212964 ,  0.00465553, ...,  0.01079313,
        -0.01647567,  0.01402071],
       [-0.0086471 ,  0.00100506,  0.00970191, ..., -0.01175196,
         0.02256839,  0.01586104]], dtype=float32)
```

embedding_lookup() finds the embedded values for our batch of 30x20 words. It goes to each row of `input_data`, and for each word in the row/sentence, finds the correspond vector in `embedding_dic`.

`It creates a [30x20x200] tensor, so, the first element of inputs (the first sentence), is a matrix of 20x200, which each row of it, is vector representing a word in the sentence.`

```
# Define where to get the data for our embeddings from
inputs = tf.nn.embedding_lookup(embedding_vocab, _input_data)  #shape=(30, 20, 200)
inputs
```

```
<tf.Tensor 'embedding_lookup:0' shape=(60, 20, 200) dtype=float32>
```

```
session.run(inputs[0], feed_dict)
```

```
array([[ 0.01730327, -0.01464074, -0.00882768, ..., -0.01785388,
        -0.0196295 , -0.01420598],
       [ 0.0060868 ,  0.01694117, -0.02250929, ...,  0.00941567,
         0.01426133, -0.01811414],
       [-0.01030405, -0.00116321, -0.00996175, ..., -0.0186278 ,
        -0.00537685,  0.01080235],
       ...,
       [ 0.01450446, -0.01894186, -0.02038161, ..., -0.01959989,
        -0.015589  , -0.01384776],
       [ 0.01366502, -0.00648529,  0.00643963, ...,  0.01662356,
        -0.01656075, -0.01014891],
       [ 0.00080465,  0.02178012,  0.01100771, ...,  0.00521918,
         0.01012067,  0.01193724]], dtype=float32)
```

Constructing Recurrent Neural Networks

tf.nn.dynamic_rnn() creates a recurrent neural network using **stacked_lstm**.
The input should be a Tensor of shape: [batch_size, max_time, embedding_vector_size], in our case it would be (30, 20, 200)

This method, returns a pair (outputs, new_state) where:

- **outputs**: is a length T list of outputs (one for each input), or a nested tuple of such elements.
- **new_state**: is the final state.

```
outputs, new_state =  tf.nn.dynamic_rnn(stacked_lstm, inputs, initial_state=_initial_state)
```

Constructing Recurrent Neural Networks

tf.nn.dynamic_rnn() creates a recurrent neural network using **stacked_lstm**.
The input should be a Tensor of shape: [batch_size, max_time, embedding_vector_size], in our case it would be (30, 20, 200)

This method, returns a pair (outputs, new_state) where:

- **outputs**: is a length T list of outputs (one for each input), or a nested tuple of such elements.
- **new_state**: is the final state.

```
outputs, new_state =  tf.nn.dynamic_rnn(stacked_lstm, inputs, initial_state=_initial_state)
```

so, lets look at the outputs. The output of the stackedLSTM comes from 200 hidden_layer, and in each time step(=20), one of them get activated. we use the linear activation to map the 200 hidden layer to a [?x10 matrix]

```
outputs
```

```
<tf.Tensor 'rnn/transpose_1:0' shape=(60, 20, 128) dtype=float32>
```

```
session.run(tf.global_variables_initializer())
session.run(outputs[0], feed_dict)
```

```
array([[-1.7422944e-04, -1.4524098e-04, -2.1496631e-04, ...,
         5.2803120e-04, -2.2556087e-04,  1.1466655e-04],
       [-6.3873571e-04,  1.2216427e-04, -5.9103733e-04, ...,
         9.0040572e-05,  2.6808682e-04, -2.6267480e-05],
       [-2.5008371e-04,  5.4898456e-04, -7.2405214e-04, ...,
        -3.0281467e-04,  1.0181291e-04,  4.7835882e-04],
       ...,
       [-4.1936274e-04,  1.8934035e-04,  6.5007808e-04,
```

Now, the flatten would convert this 3-dim tensor to:
[[sen1word1], [sen1word2], [sen1word3], ..., [sen1word20],[sen2word1], [sen2word2], [sen2word3], ..., [sen2word20], ..., [sen30word20]]

```
output = tf.reshape(outputs, [-1, hidden_size_l2])
output
```

```
<tf.Tensor 'Reshape:0' shape=(1200, 128) dtype=float32>
```

logistic unit

Now, we create a logistic unit to return the probability of the output word in our vocabulary with 1000 words.

$$Softmax = [600 \times 200] * [200 \times 1000] + [1 \times 1000] \Longrightarrow [600 \times 1000]$$

```
softmax_w = tf.get_variable("softmax_w", [hidden_size_l2, vocab_size]) #[200x1000]
softmax_b = tf.get_variable("softmax_b", [vocab_size]) #[1x1000]
logits = tf.matmul(output, softmax_w) + softmax_b
prob = tf.nn.softmax(logits)
```

Lets look at the probability of observing words for t=0 to t=20:

```
session.run(tf.global_variables_initializer())
output_words_prob = session.run(prob, feed_dict)
print("shape of the output: ", output_words_prob.shape)
print("The probability of observing words in t=0 to t=20", output_words_prob[0:20])
```

```
shape of the output:  (1200, 10000)
The probability of observing words in t=0 to t=20 [[1.01240556e-04 9.83963619e-05 1.01706646e-04 ... 9.82989513e-05
  9.98155883e-05 1.00487327e-04]
 [1.01240497e-04 9.83956561e-05 1.01700723e-04 ... 9.82935162e-05
  9.98188480e-05 1.00481848e-04]
 [1.01244077e-04 9.83909777e-05 1.01691905e-04 ... 9.82964848e-05
  9.98243195e-05 1.00477584e-04]
 ...
 [1.01229467e-04 9.83914215e-05 1.01702783e-04 ... 9.83006976e-05
  9.98158139e-05 1.00488091e-04]
 [1.01228390e-04 9.83888895e-05 1.01706552e-04 ... 9.82996207e-05
  9.98235919e-05 1.00482444e-04]
 [1.01223362e-04 9.83932405e-05 1.01711812e-04 ... 9.82973579e-05
  9.98231844e-05 1.00482532e-04]]
```

Prediction

What is the word correspond to the probability output? Lets use the maximum probability:

```
np.argmax(output_words_prob[0:20], axis=1)
```

```
array([2006, 2006, 2006, 2006, 2006, 2006, 2006, 1321, 1321, 4438, 4838,
       4438, 4438, 4438, 1461, 9979, 4438,   46,   46,   46])
```

So, what is the ground truth for the first word of first sentence?

```
y[0]
```

```
array([9971, 9972, 9974, 9975, 9976, 9980, 9981, 9982, 9983, 9984, 9986,
       9987, 9988, 9989, 9991, 9992, 9993, 9994, 9995, 9996], dtype=int32)
```

Also, you can get it from target tensor, if you want to find the embedding vector:

```
targ = session.run(_targets, feed_dict)
targ[0]
```

```
array([9971, 9972, 9974, 9975, 9976, 9980, 9981, 9982, 9983, 9984, 9986,
       9987, 9988, 9989, 9991, 9992, 9993, 9994, 9995, 9996], dtype=int32)
```

How similar the predicted words are to the target words?

Objective function

Now we have to define our objective function, to calculate the similarity of predicted values to ground truth, and then, penalize the model with the error. Our objective is to minimize loss function, that is, to minimize the average negative log probability of the target words:

$$\text{loss} = -\frac{1}{N}\sum_{i=1}^{N} \ln p_{\text{target}_i}$$

This function is already implemented and available in TensorFlow through **sequence_loss_by_example**. It calculates the weighted cross-entropy loss for **logits** and the **target** sequence.

The arguments of this function are:

- logits: List of 2D Tensors of shape [batch_size x num_decoder_symbols].
- targets: List of 1D batch-sized int32 Tensors of the same length as logits.
- weights: List of 1D batch-sized float-Tensors of the same length as logits.

```
loss = tf.contrib.legacy_seq2seq.sequence_loss_by_example([logits], [tf.reshape(_targets, [-1])],[tf.ones([batch_size * num_steps])])
```

loss is a 1D batch-sized float Tensor [600x1]: The log-perplexity for each sequence. Lets look at the first 10 values of loss:

```
session.run(loss, feed_dict)[:10]
```

```
array([9.198724, 9.217994, 9.213734, 9.20228 , 9.206766, 9.208064,
       9.202383, 9.223923, 9.212815, 9.204196], dtype=float32)
```

Now, we define loss as average of the losses:

Now, we define loss as average of the losses:

```
cost = tf.reduce_sum(loss) / batch_size
session.run(tf.global_variables_initializer())
session.run(cost, feed_dict)
```

```
184.21759
```

Training

To do training for our network, we have to take the following steps:

1. Define the optimizer.
2. Extract variables that are trainable.
3. Calculate the gradients based on the loss function.
4. Apply the optimizer to the variables/gradients tuple.

1. Define Optimizer

GradientDescentOptimizer constructs a new gradient descent optimizer. Later, we use constructed **optimizer** to compute gradients for a loss and apply gradients to variables.

```
# Create a variable for the learning rate
lr = tf.Variable(0.0, trainable=False)
# Create the gradient descent optimizer with our learning rate
optimizer = tf.train.GradientDescentOptimizer(lr)
```

2. Trainable Variables

Defining a variable, if you passed *trainable=True*, the variable constructor automatically adds new variables to the graph collection **GraphKeys.TRAINABLE_VARIABLES**. Now, using *tf.trainable_variables()* you can get all variables created with **trainable=True**.

```
# Get all TensorFlow variables marked as "trainable" (i.e. all of them except _lr, which we just created)
tvars = tf.trainable_variables()
tvars
```

```
[<tf.Variable 'embedding_vocab:0' shape=(10000, 200) dtype=float32_ref>,
 <tf.Variable 'rnn/multi_rnn_cell/cell_0/basic_lstm_cell/kernel:0' shape=(456, 1024) dtype=float32_ref>,
 <tf.Variable 'rnn/multi_rnn_cell/cell_0/basic_lstm_cell/bias:0' shape=(1024,) dtype=float32_ref>,
 <tf.Variable 'rnn/multi_rnn_cell/cell_1/basic_lstm_cell/kernel:0' shape=(384, 512) dtype=float32_ref>,
 <tf.Variable 'rnn/multi_rnn_cell/cell_1/basic_lstm_cell/bias:0' shape=(512,) dtype=float32_ref>,
 <tf.Variable 'softmax_w:0' shape=(128, 10000) dtype=float32_ref>,
 <tf.Variable 'softmax_b:0' shape=(10000,) dtype=float32_ref>]
```

Note: we can find the name and scope of all variables:

```
[v.name for v in tvars]
```

```
['embedding_vocab:0',
 'rnn/multi_rnn_cell/cell_0/basic_lstm_cell/kernel:0',
 'rnn/multi_rnn_cell/cell_0/basic_lstm_cell/bias:0',
 'rnn/multi_rnn_cell/cell_1/basic_lstm_cell/kernel:0',
 'rnn/multi_rnn_cell/cell_1/basic_lstm_cell/bias:0',
 'softmax_w:0',
 'softmax_b:0']
```

3. Calculate the gradients based on the loss function

Gradient

: The gradient of a function is the slope of its derivative (line), or in other words, the rate of change of a function. It's a vector (a direction to move) that points in the direction of greatest increase of the function, and calculated by the **derivative** operation.

First lets recall the gradient function using an toy example:

$$z = (2x^2 + 3xy)$$

```
var_x = tf.placeholder(tf.float32)
var_y = tf.placeholder(tf.float32)
func_test = 2.0 * var_x * var_x + 3.0 * var_x * var_y
session.run(tf.global_variables_initializer())
session.run(func_test, {var_x:1.0,var_y:2.0})
```

```
8.0
```

The **tf.gradients()** function allows you to compute the symbolic gradient of one tensor with respect to one or more other tensors—including variables. **tf.gradients(func, xs)** constructs symbolic partial derivatives of sum of **func** w.r.t. *x* in **xs**.

Now, lets look at the derivitive w.r.t. **var_x**:

$$\frac{\partial}{\partial x}(2x^2 + 3xy) = 4x + 3y$$

```
var_grad = tf.gradients(func_test, [var_x])
session.run(var_grad, {var_x:1.0,var_y:2.0})
```

```
[10.0]
```

the derivative w.r.t. **var_y**:

$$\frac{\partial}{\partial x}(2x^2 + 3xy) = 3x$$

```
var_grad = tf.gradients(func_test, [var_y])
session.run(var_grad, {var_x:1.0, var_y:2.0})
```

```
[3.0]
```

Now, we can look at gradients w.r.t all variables:

Now, we can look at gradients w.r.t all variables:

```
tf.gradients(cost, tvars)
```

```
[<tensorflow.python.framework.ops.IndexedSlices at 0x7f6e7daf1d30>,
 <tf.Tensor 'gradients_2/rnn/while/rnn/multi_rnn_cell/cell_0/basic_lstm_cell/MatMul/Enter_grad/b_acc_3:0' shape=(456, 1024) dtype=float32>,
 <tf.Tensor 'gradients_2/rnn/while/rnn/multi_rnn_cell/cell_0/basic_lstm_cell/BiasAdd/Enter_grad/b_acc_3:0' shape=(1024,) dtype=float32>,
 <tf.Tensor 'gradients_2/rnn/while/rnn/multi_rnn_cell/cell_1/basic_lstm_cell/MatMul/Enter_grad/b_acc_3:0' shape=(384, 512) dtype=float32>,
 <tf.Tensor 'gradients_2/rnn/while/rnn/multi_rnn_cell/cell_1/basic_lstm_cell/BiasAdd/Enter_grad/b_acc_3:0' shape=(512,) dtype=float32>,
 <tf.Tensor 'gradients_2/MatMul_grad/MatMul_1:0' shape=(128, 10000) dtype=float32>,
 <tf.Tensor 'gradients_2/add_grad/Reshape_1:0' shape=(10000,) dtype=float32>]
```

```
grad_t_list = tf.gradients(cost, tvars)
#sess.run(grad_t_list,feed_dict)
```

now, we have a list of tensors, t-list. We can use it to find clipped tensors. **clip_by_global_norm** clips values of multiple tensors by the ratio of the sum of their norms.

clip_by_global_norm get *t-list* as input and returns 2 things:

- a list of clipped tensors, so called *list_clipped*
- the global norm (global_norm) of all tensors in t_list

```
# Define the gradient clipping threshold
grads, _ = tf.clip_by_global_norm(grad_t_list, max_grad_norm)
grads
```

```
[<tensorflow.python.framework.ops.IndexedSlices at 0x7f6e7da97710>,
 <tf.Tensor 'clip_by_global_norm/clip_by_global_norm/_1:0' shape=(456, 1024) dtype=float32>,
 <tf.Tensor 'clip_by_global_norm/clip_by_global_norm/_2:0' shape=(1024,) dtype=float32>,
 <tf.Tensor 'clip_by_global_norm/clip_by_global_norm/_3:0' shape=(384, 512) dtype=float32>,
 <tf.Tensor 'clip_by_global_norm/clip_by_global_norm/_4:0' shape=(512,) dtype=float32>,
 <tf.Tensor 'clip_by_global_norm/clip_by_global_norm/_5:0' shape=(128, 10000) dtype=float32>,
 <tf.Tensor 'clip_by_global_norm/clip_by_global_norm/_6:0' shape=(10000,) dtype=float32>]
```

```
session.run(grads, feed_dict)
```

4. Apply the optimizer to the variables / gradients tuple.

```
# Create the training TensorFlow Operation through our optimizer
train_op = optimizer.apply_gradients(zip(grads, tvars))
```

```
session.run(tf.global_variables_initializer())
session.run(train_op, feed_dict)
```

LSTM

We learned how the model is build step by step. Noe, let's then create a Class that represents our model. This class needs a few things:

- We have to create the model in accordance with our defined hyperparameters
- We have to create the placeholders for our input data and expected outputs (the real data)
- We have to create the LSTM cell structure and connect them with our RNN structure
- We have to create the word embeddings and point them to the input data
- We have to create the input structure for our RNN
- We have to instantiate our RNN model and retrieve the variable in which we should expect our outputs to appear
- We need to create a logistic structure to return the probability of our words
- We need to create the loss and cost functions for our optimizer to work, and then create the optimizer
- And finally, we need to create a training operation that can be run to actually train our model

```
hidden_size_11
```

```
256
```

```
class PTBModel(object):

    def __init__(self, action_type):
        ###################################
        # Setting parameters for ease of use #
        ###################################
        self.batch_size = batch_size
        self.num_steps = num_steps
        self.hidden_size_l1 = hidden_size_l1
        self.hidden_size_l2 = hidden_size_l2
        self.vocab_size = vocab_size
        self.embeding_vector_size = embeding_vector_size
        ###############################################################################
        # Creating placeholders for our input data and expected outputs (target data) #
        ###############################################################################
        self._input_data = tf.placeholder(tf.int32, [batch_size, num_steps]) #[30#20]
        self._targets = tf.placeholder(tf.int32, [batch_size, num_steps]) #[30#20]

        ###########################################################################
        # Creating the LSTM cell structure and connect it with the RNN structure #
        ###########################################################################
        # Create the LSTM unit.
        # This creates only the structure for the LSTM and has to be associated with a RNN unit still.
        # The argument n_hidden(size=200) of BasicLSTMCell is size of hidden layer, that is, the number of hidden units of the LSTM (inside A).
        # Size is the same as the size of our hidden layer, and no bias is added to the Forget Gate.
        # LSTM cell processes one word at a time and computes probabilities of the possible continuations of the sentence.
        lstm_cell_l1 = tf.contrib.rnn.BasicLSTMCell(self.hidden_size_l1, forget_bias=0.0)
        lstm_cell_l2 = tf.contrib.rnn.BasicLSTMCell(self.hidden_size_l2, forget_bias=0.0)

        # Unless you changed keep_prob, this won't actually execute -- this is a dropout wrapper for our LSTM unit
        # This is an optimization of the LSTM output, but is not needed at all
        if action_type == "is_training" and keep_prob < 1:
            lstm_cell_l1 = tf.contrib.rnn.DropoutWrapper(lstm_cell_l1, output_keep_prob=keep_prob)
            lstm_cell_l2 = tf.contrib.rnn.DropoutWrapper(lstm_cell_l2, output_keep_prob=keep_prob)

        # By taking in the LSTM cells as parameters, the MultiRNNCell function junctions the LSTM units to the RNN units.
        # RNN cell composed sequentially of multiple simple cells.
        stacked_lstm = tf.contrib.rnn.MultiRNNCell([lstm_cell_l1, lstm_cell_l2])

        # Define the initial state, i.e., the model state for the very first data point
        # It initialize the state of the LSTM memory. The memory state of the network is initialized with a vector of zeros and gets updated after reading each word.
        self._initial_state = stacked_lstm.zero_state(batch_size, tf.float32)
```

```
###########################################################################
# Creating the word embeddings and pointing them to the input data #
###########################################################################
with tf.device("/cpu:0"):
    # Create the embeddings for our input data. Size is hidden size.
    embedding = tf.get_variable("embedding", [vocab_size, self.embeding_vector_size])  #[10000x200]
    # Define where to get the data for our embeddings from
    inputs = tf.nn.embedding_lookup(embedding, self._input_data)

# Unless you changed keep_prob, this won't actually execute -- this is a dropout addition for our inputs
# This is an optimization of the input processing and is not needed at all
if action_type == "is_training" and keep_prob < 1:
    inputs = tf.nn.dropout(inputs, keep_prob)

##########################################
# Creating the input structure for our RNN #
##########################################
# Input structure is 20x[30x200]
# Considering each word is represended by a 200 dimentional vector, and we have 30 batchs, we create 30 word-vectors of size [30xx2000]
# inputs = [tf.squeeze(input_, [1]) for input_ in tf.split(1, num_steps, inputs)]
# The input structure is fed from the embeddings, which are filled in by the input data
# Feeding a batch of b sentences to a RNN:
# In step 1,  first word of each of the b sentences (in a batch) is input in parallel.
# In step 2,  second word of each of the b sentences is input in parallel.
# The parallelism is only for efficiency.
# Each sentence in a batch is handled in parallel, but the network sees one word of a sentence at a time and does the computations accordingly.
# All the computations involving the words of all sentences in a batch at a given time step are done in parallel.

#################################################################################################
# Instantiating our RNN model and retrieving the structure for returning the outputs and the state #
#################################################################################################

outputs, state = tf.nn.dynamic_rnn(stacked_lstm, inputs, initial_state=self._initial_state)
#########################################################################
```

```
##########################################################################
# Creating a logistic unit to return the probability of the output word #
##########################################################################
output = tf.reshape(outputs, [-1, self.hidden_size_l2])
softmax_w = tf.get_variable("softmax_w", [self.hidden_size_l2, vocab_size]) #[200x1000]
softmax_b = tf.get_variable("softmax_b", [vocab_size]) #[1x1000]
logits = tf.matmul(output, softmax_w) + softmax_b
logits = tf.reshape(logits, [self.batch_size, self.num_steps, vocab_size])
prob = tf.nn.softmax(logits)
out_words = tf.argmax(prob, axis=2)
self._output_words = out_words
##########################################################################
# Defining the loss and cost functions for the model's learning to work #
##########################################################################


# Use the contrib sequence loss and average over the batches
loss = tf.contrib.seq2seq.sequence_loss(
    logits,
    self.targets,
    tf.ones([batch_size, num_steps], dtype=tf.float32),
    average_across_timesteps=False,
    average_across_batch=True)

  loss = tf.contrib.legacy_seq2seq.sequence_loss_by_example([logits], [tf.reshape(self._targets, [-1])],
                                                    [tf.ones([batch_size * num_steps])])
self._cost = tf.reduce_sum(loss)

# Store the final state
 # Use the contrib sequence loss and average over the batches
 loss = tf.contrib.seq2seq.sequence_loss(
     logits,
     self.targets,
     tf.ones([batch_size, num_steps], dtype=tf.float32),
     average_across_timesteps=False,
     average_across_batch=True)

   loss = tf.contrib.legacy_seq2seq.sequence_loss_by_example([logits], [tf.reshape(self._targets, [-1])],
                                                     [tf.ones([batch_size * num_steps])])
 self._cost = tf.reduce_sum(loss)

 # Store the final state
 self._final_state = state

 #Everything after this point is relevant only for training
 if action_type != "is_training":
     return

 ################################################
 # Creating the Training Operation for our Model #
 ################################################
 # Create a variable for the learning rate
 self._lr = tf.Variable(0.0, trainable=False)
 # Get all TensorFlow variables marked as "trainable" (i.e. all of them except _lr, which we just created)
 tvars = tf.trainable_variables()
 # Define the gradient clipping threshold
 grads, _ = tf.clip_by_global_norm(tf.gradients(self._cost, tvars), max_grad_norm)
 # Create the gradient descent optimizer with our learning rate
 optimizer = tf.train.GradientDescentOptimizer(self.lr)
 # Create the training TensorFlow Operation through our optimizer
 self._train_op = optimizer.apply_gradients(zip(grads, tvars))

Helper functions for our LSTM RNN class
```

```
    # Assign the learning rate for this model
    def assign_lr(self, session, lr_value):
        session.run(tf.assign(self.lr, lr_value))

    # Returns the input data for this model at a point in time
    @property
    def input_data(self):
        return self._input_data



    # Returns the targets for this model at a point in time
    @property
    def targets(self):
        return self._targets

    # Returns the initial state for this model
    @property
    def initial_state(self):
        return self._initial_state

    # Returns the defined Cost
    @property
    def cost(self):
        return self._cost

    # Returns the final state for this model
    @property
    def final_state(self):
        return self._final_state

    # Returns the final output words for this model
    @property
    def final_output_words(self):
        return self._output_words

    # Returns the current learning rate for this model
    @property
    def lr(self):
        return self._lr
```

```
    # Returns the training operation defined for this model
    @property
    def train_op(self):
        return self._train_op
```

With that, the actual structure of our Recurrent Neural Network with Long Short-Term Memory is finished. What remains for us to do is to actually create the methods to run through time -- that is, the `run_epoch` method to be run at each epoch and a `main` script which ties all of this together.

What our `run_epoch` method should do is take our input data and feed it to the relevant operations. This will return at the very least the current result for the cost function.

```
###############################################################################################################
# run_one_epoch takes as parameters the current session, the model instance, the data to be fed, and the operation to be run #
###############################################################################################################
def run_one_epoch(session, m, data, eval_op, verbose=False):

    #Define the epoch size based on the length of the data, batch size and the number of steps
    epoch_size = ((len(data) // m.batch_size) - 1) // m.num_steps
    start_time = time.time()
    costs = 0.0
    iters = 0

    state = session.run(m.initial_state)

    #For each step and data point
    for step, (x, y) in enumerate(reader.ptb_iterator(data, m.batch_size, m.num_steps)):

        #Evaluate and return cost, state by running cost, final_state and the function passed as parameter
        cost, state, out_words, _ = session.run([m.cost, m.final_state, m.final_output_words, eval_op],
                                     {m.input_data: x,
                                      m.targets: y,
                                      m.initial_state: state})

        #Add returned cost to costs (which keeps track of the total costs for this epoch)
        costs += cost
```

```
        #Add returned cost to costs (which keeps track of the total costs for this epoch)
        costs += cost

        #Add number of steps to iteration counter
        iters += m.num_steps

        if verbose and step % (epoch_size // 10) == 10:
            print("Itr %d of %d, perplexity: %.3f speed: %.0f wps" % (step , epoch_size, np.exp(costs / iters), iters * m.batch_size / (time.time() - start_time)))

    # Returns the Perplexity rating for us to keep track of how the model is evolving
    return np.exp(costs / iters)
```

Now, we create the `main` method to tie everything together. The code here reads the data from the directory, using the `reader` helper module, and then trains and evaluates the model on both a testing and a validating subset of data.

```
# Reads the data and separates it into training data, validation data and testing data
raw_data = reader.ptb_raw_data(data_dir)
train_data, valid_data, test_data, _, _ = raw_data
```

```
# Initializes the Execution Graph and the Session
with tf.Graph().as_default(), tf.Session() as session:
    initializer = tf.random_uniform_initializer(-init_scale, init_scale)

    # Instantiates the model for training
    # tf.variable_scope add a prefix to the variables created with tf.get_variable
    with tf.variable_scope("model", reuse=None, initializer=initializer):
        m = PTBModel("is_training")

    # Reuses the trained parameters for the validation and testing models
    # They are different instances but use the same variables for weights and biases, they just don't change when data is input
    with tf.variable_scope("model", reuse=True, initializer=initializer):
        mvalid = PTBModel("is_validating")
        mtest = PTBModel("is_testing")

    #Initialize all variables
    tf.global_variables_initializer().run()
```

```
    for i in range(max_epoch):
        # Define the decay for this epoch
        lr_decay = decay ** max(i - max_epoch_decay_lr, 0.0)

        # Set the decayed learning rate as the learning rate for this epoch
        m.assign_lr(session, learning_rate * lr_decay)

        print("Epoch %d : Learning rate: %.3f" % (i + 1, session.run(m.lr)))

        # Run the loop for this epoch in the training model
        train_perplexity = run_one_epoch(session, m, train_data, m.train_op, verbose=True)
        print("Epoch %d : Train Perplexity: %.3f" % (i + 1, train_perplexity))

        # Run the loop for this epoch in the validation model
        valid_perplexity = run_one_epoch(session, mvalid, valid_data, tf.no_op())
        print("Epoch %d : Valid Perplexity: %.3f" % (i + 1, valid_perplexity))

    # Run the loop in the testing model to see how effective was our training
    test_perplexity = run_one_epoch(session, mtest, test_data, tf.no_op())

    print("Test Perplexity: %.3f" % test_perplexity)
```

```
Epoch 1 : Learning rate: 1.000
Itr 10 of 774, perplexity: 4146.504 speed: 208 wps
Itr 87 of 774, perplexity: 1258.985 speed: 203 wps
```

Lab Applying RNN/LSTM TO CHARACTER MODELLING

First, import the requiered libraries:

```
import tensorflow as tf
import time
import codecs
import os
import collections
from six.moves import cPickle
import numpy as np
```

Data loader

The following cell is a class that help to read data from input file.

```
class TextLoader():
    def __init__(self, data_dir, batch_size, seq_length, encoding='utf-8'):
        self.data_dir = data_dir
        self.batch_size = batch_size
        self.seq_length = seq_length
        self.encoding = encoding

        input_file = os.path.join(data_dir, "input.txt")
        vocab_file = os.path.join(data_dir, "vocab.pkl")
        tensor_file = os.path.join(data_dir, "data.npy")

        if not (os.path.exists(vocab_file) and os.path.exists(tensor_file)):
            print("reading text file")
            self.preprocess(input_file, vocab_file, tensor_file)
        else:
            print("loading preprocessed files")
            self.load_preprocessed(vocab_file, tensor_file)
        self.create_batches()
        self.reset_batch_pointer()

    def preprocess(self, input_file, vocab_file, tensor_file):
        with codecs.open(input_file, "r", encoding=self.encoding) as f:
            data = f.read()
        counter = collections.Counter(data)
        count_pairs = sorted(counter.items(), key=lambda x: -x[1])
        self.chars, _ = zip(*count_pairs)
        self.vocab_size = len(self.chars)
        self.vocab = dict(zip(self.chars, range(len(self.chars))))
        with open(vocab_file, 'wb') as f:
            cPickle.dump(self.chars, f)
        self.tensor = np.array(list(map(self.vocab.get, data)))
        np.save(tensor_file, self.tensor)
```

```
    def load_preprocessed(self, vocab_file, tensor_file):
        with open(vocab_file, 'rb') as f:
            self.chars = cPickle.load(f)
        self.vocab_size = len(self.chars)
        self.vocab = dict(zip(self.chars, range(len(self.chars))))
        self.tensor = np.load(tensor_file)
        self.num_batches = int(self.tensor.size / (self.batch_size * self.seq_length))

    def create_batches(self):
        self.num_batches = int(self.tensor.size / (self.batch_size * self.seq_length))

        # When the data (tensor) is too small, let's give them a better error message
        if self.num_batches==0:
            assert False, "Not enough data. Make seq_length and batch_size small."

        self.tensor = self.tensor[:self.num_batches * self.batch_size * self.seq_length]
        xdata = self.tensor
        ydata = np.copy(self.tensor)
        ydata[:-1] = xdata[1:]
        ydata[-1] = xdata[0]
        self.x_batches = np.split(xdata.reshape(self.batch_size, -1), self.num_batches, 1)
        self.y_batches = np.split(ydata.reshape(self.batch_size, -1), self.num_batches, 1)


    def next_batch(self):
        x, y = self.x_batches[self.pointer], self.y_batches[self.pointer]
        self.pointer += 1
        return x, y

    def reset_batch_pointer(self):
        self.pointer = 0
```

Parameters

Batch, number_of_batch, batch_size and seq_length

what is batch, number_of_batch, batch_size and seq_length in the charcter level example?

Lets assume the input is this sentence: '**here is an example**'. Then:

- txt_length = 18
- seq_length = 3
- batch_size = 2
- number_of_batchs = 18/3*2 = 3
- batch = array ([['h','e','r'],['e',' ','i']])
- sample Seq = 'her'

Ok, now, lets look at a real dataset, with real parameters.

```
seq_length = 50 # RNN sequence length
batch_size = 60  # minibatch size, i.e. size of data in each epoch
num_epochs = 125 # you should change it to 50 if you want to see a relatively good results
learning_rate = 0.002
decay_rate = 0.97
rnn_size = 128 # size of RNN hidden state (output dimension)
num_layers = 2 #number of layers in the RNN
```

***

We download the input file, and print a part of it:

```
!wget -nv -O input.txt https://ibm.box.com/shared/static/a3f9e9mbpup09toq35ut7ke3131f03hg.txt
with open('input.txt', 'r') as f:
    read_data = f.read()
    print (read_data[0:100])
f.closed
```

```
All:
Speak, speak.

First Citizen:
You
True
```

Now, we can read the data at batches using the **TextLoader** class. It will convert the characters to numbers, and represent each sequence as a vector in batches:

```
data_loader = TextLoader('', batch_size, seq_length)
vocab_size = data_loader.vocab_size
print ("vocabulary size:" ,data_loader.vocab_size)
print ("Characters:" ,data_loader.chars)
print ("vocab number of 'F':",data_loader.vocab['F'])
print ("Character sequences (first batch):", data_loader.x_batches[0])
```

```
loading preprocessed files
vocabulary size: 65
Characters: (' ', 'e', 't', 'o', 'a', 'h', 's', 'r', 'n', 'i', '\n', 'l', 'd', 'u', 'm', 'y', ',', 'w', 'f', 'c', 'g', 'I', 'b', 'p', ':', '.', 'A', 'v', 'k', 'T', "'", 'E',
', 'S', 'L', 'C', ';', 'W', 'U', 'H', 'M', 'B', '?', 'G', '!', 'D', '-', 'F', 'Y', 'P', 'K', 'V', 'j', 'q', 'x', 'z', 'J', 'Q', 'Z', 'X', '3', '&', '$')
vocab number of 'F': 49
Character sequences (first batch): [[49  9  7 ...  1  4  7]
 [19  4 14 ... 14  9 20]
 [ 8 20 10 ...  8 10 18]
 ...
 [21  2  0 ...  0 21  0]
 [ 9  7  7 ...  0  2  3]
 [ 3  7  0 ...  5  9 23]]
```

Input and output

Input and output

```
x,y = data_loader.next_batch()
x
```

```
array([[49,  9,  7, ...,  1,  4,  7],
       [19,  4, 14, ..., 14,  9, 20],
       [ 8, 20, 10, ...,  8, 10, 18],
       ...,
       [21,  2,  0, ...,  0, 21,  0],
       [ 9,  7,  7, ...,  0,  2,  3],
       [ 3,  7,  0, ...,  5,  9, 23]])
```

```
x.shape  #batch_size =60, seq_length=50
```

```
(60, 50)
```

Here, **y** is the next character for each character in **x**:

```
y
```

```
array([[ 9,  7,  6, ...,  4,  7,  0],
       [ 4, 14, 22, ...,  9, 20,  5],
       [20, 10, 29, ..., 10, 18,  4],
       ...,
       [ 2,  0,  6, ..., 21,  0,  6],
       [ 7,  7,  4, ...,  2,  3,  0],
       [ 7,  0, 33, ...,  9, 23,  0]])
```

LSTM Architecture

Each LSTM cell has 5 parts:

1. Input
2. prv_state
3. prv_output
4. new_state
5. new_output

- Each LSTM cell has an input layre, which its size is 128 units in our case. The input vector's dimension also is 128, which is the dimensionality of embedding vector, so called, dimension size of W2V/embedding, for each character/word.
- Each LSTM cell has a hidden layer, where there are some hidden units. The argument n_hidden=128 of BasicLSTMCell is the number of hidden units of the LSTM (inside A). It keeps the size of the output and state vector. It is also known as, rnn_size, num_units, num_hidden_units, and LSTM size
- An LSTM keeps two pieces of information as it propagates through time:
  - **hidden state** vector: Each LSTM cell accept a vector, called **hidden state** vector, of size n_hidden=128, and its value is returned to the LSTM cell in the next step. The **hidden state** vector, which is the memory of the LSTM, accumulates using its (forget, input, and output) gates through time. "num_units" is equivalant to "size of RNN hidden state". number of hidden units is the dimensianality of the output (= dimesianality of the state) of the LSTM cell.
  - **previous time-step output**: For each LSTM cell that we initialize, we need to supply a value (128 in this case) for the hidden dimension, or as some people like to call it, the number of units in the LSTM cell.

num_layers = 2

- number of layers in the RNN, is defined by num_layers
- An input of MultiRNNCell is **cells** which is list of RNNCells that will be composed in this order.

Defining stacked RNN Cell

BasicRNNCell is the most basic RNN cell.

```
]: cell = tf.contrib.rnn.BasicRNNCell(rnn_size)
```

Defining stacked RNN Cell

BasicRNNCell is the most basic RNN cell.

```
cell = tf.contrib.rnn.BasicRNNCell(rnn_size)
```

```
# a two layer cell
stacked_cell = tf.contrib.rnn.MultiRNNCell([cell] * num_layers)
```

```
# hidden state size
stacked_cell.output_size
```

```
128
```

state varibale keeps output and new_state of the LSTM, so it is a touple of size:

```
stacked_cell.state_size
```

```
(128, 128)
```

Lets define input data:

```
input_data = tf.placeholder(tf.int32, [batch_size, seq_length])# a 60x50
input_data
```

```
<tf.Tensor 'Placeholder:0' shape=(60, 50) dtype=int32>
```

and target data:

```
targets = tf.placeholder(tf.int32, [batch_size, seq_length]) # a 60x50
targets
```

```
<tf.Tensor 'Placeholder_1:0' shape=(60, 50) dtype=int32>
```

The memory state of the network is initialized with a vector of zeros and gets updated after reading each character.

BasicRNNCell.zero_state(batch_size, dtype) Return zero-filled state tensor(s). In this function, batch_size representing the batch size.

```
initial_state = stacked_cell.zero_state(batch_size, tf.float32) #why batch_size ? 60x128
```

Lets check the value of the input_data again:

```
session = tf.Session()
feed_dict={input_data:x, targets:y}
session.run(input_data, feed_dict)
```

```
array([[49,  9,  7, ...,  1,  4,  7],
       [19,  4, 14, ..., 14,  9, 20],
       [ 8, 20, 10, ...,  8, 10, 18],
       ...,
       [21,  2,  0, ...,  0, 21,  0],
       [ 9,  7,  7, ...,  0,  2,  3],
       [ 3,  7,  0, ...,  5,  9, 23]], dtype=int32)
```

Embedding

In this section, we build a 128-dim vector for each character. As we have 60 batches, and 50 character in each sequence, it will generate a [60,50,128] matrix.

Notice: The function `tf.get_variable()` is used to share a variable and to initialize it in one place. `tf.get_variable()` is used to get or create a variable instead of a direct call to `tf.Variable`.

```
with tf.variable_scope('rnnlm', reuse=False):
    softmax_w = tf.get_variable("softmax_w", [rnn_size, vocab_size]) #128x65
    softmax_b = tf.get_variable("softmax_b", [vocab_size]) # 1x65)
    #with tf.device("/cpu:0"):

    # embedding variable is initialized randomely
    embedding = tf.get_variable("embedding", [vocab_size, rnn_size])  #65x128

    # embedding_lookup goes to each row of input_data, and for each character in the row, finds the correspond vector in embedding
    # it creates a 60*50*[1*128] matrix
    # so, the first elemnt of em, is a matrix of 50x128, which each row of it is vector representing that character
    em = tf.nn.embedding_lookup(embedding, input_data) # em is 60x50x[1*128]
    # split: Splits a tensor into sub tensors.
    # syntax:  tf.split(split_dim, num_split, value, name='split')
    # it will split the 60x50x[1x128] matrix into 50 matrix of 60x[1*128]
    inputs = tf.split(em, seq_length, 1)
    # It will convert the list to 50 matrix of [60x128]
    inputs = [tf.squeeze(input_, [1]) for input_ in inputs]
```

Lets take a look at the **embedding**, **em**, and **inputs** variabbles:

Embedding variable is initialized with random values:

```
session.run(tf.global_variables_initializer())
#print embedding.shape
session.run(embedding)
```

```
array([[ 0.09682302, -0.1413292 ,  0.0605492 , ..., -0.12485891,
         0.05089612,  0.06459557],
       [ 0.10604803, -0.02018537,  0.01680125, ...,  0.17529999,
         0.15361421, -0.03503613],
       [-0.12641312,  0.0973895 , -0.03902969, ..., -0.06618046,
        -0.16916679,  0.05245478],
       ...,
       [ 0.12825353, -0.05840966,  0.02873248, ..., -0.03441671,
```

The first elemnt of em, is a matrix of 50x128, which each row of it is vector representing that character

```
em = tf.nn.embedding_lookup(embedding, input_data)
emp = session.run(em,feed_dict={input_data:x})
print (emp.shape)
emp[0]
```

```
(60, 50, 128)
array([[-0.04610696,  0.03213142, -0.13474256, ...,  0.08589412,
        -0.11100204, -0.04003918],
       [ 0.04089905,  0.03810503,  0.15088399, ...,  0.14986075,
         0.10503717,  0.17262845],
       [ 0.01612227,  0.15555416,  0.05389382, ...,  0.05362642,
         0.09817319, -0.08176701],
       ...,
       [ 0.10604803, -0.02018537,  0.01680125, ...,  0.17529999,
         0.15361421, -0.03503613],
       [-0.136866  ,  0.04898758, -0.09564161, ..., -0.10230616,
        -0.13759644,  0.09764285],
       [ 0.01612227,  0.15555416,  0.05389382, ...,  0.05362642,
         0.09817319, -0.08176701]], dtype=float32)
```

Let's consider each sequence as a sentence of length 50 characters, then, the first item in **inputs** is a [60x128] vector which represents the first characters of 60 sentences.

```
inputs = tf.split(em, seq_length, 1)
inputs = [tf.squeeze(input_, [1]) for input_ in inputs]
inputs[0:5]
```

```
[<tf.Tensor 'Squeeze:0' shape=(60, 128) dtype=float32>,
 <tf.Tensor 'Squeeze_1:0' shape=(60, 128) dtype=float32>,
 <tf.Tensor 'Squeeze_2:0' shape=(60, 128) dtype=float32>,
 <tf.Tensor 'Squeeze_3:0' shape=(60, 128) dtype=float32>,
 <tf.Tensor 'Squeeze_4:0' shape=(60, 128) dtype=float32>]
```

Feeding a batch of 50 sequence to a RNN:

The feeding process for iputs is as following:

- Step 1: first character of each of the 50 sentences (in a batch) is entered in parallel.
- Step 2: second character of each of the 50 sentences is input in parallel.
- Step n: nth character of each of the 50 sentences is input in parallel.

The parallelism is only for efficiency. Each character in a batch is handled in parallel, but the network sees one character of a sequence at a time and does the computations accordingly. All the computations involving the characters of all sequences in a batch at a given time step are done in parallel.

```
session.run(inputs[0],feed_dict={input_data:x})
```

```
array([[-0.04610696,  0.03213142, -0.13474256, ...,  0.08589412,
        -0.11100204, -0.04003918],
       [-0.04281311,  0.07677947, -0.10739081, ..., -0.09417471,
         0.16230513, -0.03518304],
       [ 0.06974062, -0.07587742, -0.08997462, ..., -0.06710815,
        -0.15369378, -0.11222705],
       ...,
       [ 0.04347058,  0.12137432,  0.1711338 , ..., -0.12174603,
        -0.08534024,  0.157034  ],
       [ 0.04089905,  0.03810503,  0.15088399, ...,  0.14986075,
         0.10503717,  0.17262845],
       [-0.16674013, -0.134291  ,  0.13750212, ..., -0.16421077,
         0.17264254,  0.13595559]], dtype=float32)
```

Feeding the RNN with one batch, we can check the new output and new state of network:

Feeding the RNN with one batch, we can check the new output and new state of network:

```
#outputs is 50x[60*128]
outputs, new_state = tf.contrib.legacy_seq2seq.rnn_decoder(inputs, initial_state, stacked_cell, loop_function=None, scope='rnnlm')
new_state
```

```
(<tf.Tensor 'rnnlm_1/rnnlm/multi_rnn_cell/cell_0/basic_rnn_cell/Tanh_98:0' shape=(60, 128) dtype=float32>,
 <tf.Tensor 'rnnlm_1/rnnlm/multi_rnn_cell/cell_0/basic_rnn_cell/Tanh_99:0' shape=(60, 128) dtype=float32>)
```

```
outputs[0:5]
```

```
[<tf.Tensor 'rnnlm_1/rnnlm/multi_rnn_cell/cell_0/basic_rnn_cell/Tanh_1:0' shape=(60, 128) dtype=float32>,
 <tf.Tensor 'rnnlm_1/rnnlm/multi_rnn_cell/cell_0/basic_rnn_cell/Tanh_3:0' shape=(60, 128) dtype=float32>,
 <tf.Tensor 'rnnlm_1/rnnlm/multi_rnn_cell/cell_0/basic_rnn_cell/Tanh_5:0' shape=(60, 128) dtype=float32>,
 <tf.Tensor 'rnnlm_1/rnnlm/multi_rnn_cell/cell_0/basic_rnn_cell/Tanh_7:0' shape=(60, 128) dtype=float32>,
 <tf.Tensor 'rnnlm_1/rnnlm/multi_rnn_cell/cell_0/basic_rnn_cell/Tanh_9:0' shape=(60, 128) dtype=float32>]
```

Let's check the output of network after feeding it with first batch:

```
first_output = outputs[0]
session.run(tf.global_variables_initializer())
session.run(first_output,feed_dict={input_data:x})
```

```
array([[-0.02367019,  0.02249297, -0.07756998, ...,  0.11196241,
         0.06198768, -0.10009311],
       [ 0.00383705, -0.085026  , -0.00578733, ..., -0.09159598,
        -0.03011957,  0.08199154],
       [-0.00090939, -0.10684054,  0.01080646, ...,  0.00749919,
         0.04780088, -0.03666944],
       ...,
       [-0.14592405,  0.1025786 ,  0.08757704, ..., -0.06113817,
        -0.06668758, -0.14927606],
       [ 0.02468136, -0.02489132,  0.09907677, ...,  0.03210178,
         0.08921564,  0.10248761],
       [ 0.15641604, -0.04086293,  0.14779116, ...,  0.00040349,
         0.08794381,  0.06123935]], dtype=float32)
```

As it was explained, **outputs** variable is a 50x[60x128] tensor. We need to reshape it back to [60x50x128] to be able to calculate the probablity of the next character using the softmax. The **softmax_w** shape is [rnn_size, vocab_size],whihc is [128x65] in our case. Threfore, we have a fully connected layer on top of LSTM cells, which help us to decode the next charachter. We can use the **softmax(output * softmax_w + softmax_b)** for this purpose. The shape of the matrixis would be:

We can do it step-by-step:

```
output = tf.reshape(tf.concat( outputs,1), [-1, rnn_size])
output
```

```
<tf.Tensor 'Reshape:0' shape=(3000, 128) dtype=float32>
```

```
logits = tf.matmul(output, softmax_w) + softmax_b
logits
```

```
<tf.Tensor 'add:0' shape=(3000, 65) dtype=float32>
```

```
probs = tf.nn.softmax(logits)
probs
```

```
<tf.Tensor 'Softmax:0' shape=(3000, 65) dtype=float32>
```

Here is the probablity of the next chracter in all batches:

```
session.run(tf.global_variables_initializer())
session.run(probs,feed_dict={input_data:x})
```

```
array([[0.01528679, 0.01569048, 0.01131989, ..., 0.01269612, 0.01714724,
        0.01820261],
       [0.01431782, 0.01847589, 0.01598706, ..., 0.01111288, 0.02135488,
        0.02090689],
       [0.01336517, 0.01799161, 0.01232213, ..., 0.01540034, 0.01303841,
        0.01832603],
       ...,
       [0.01815249, 0.01892675, 0.01160855, ..., 0.00875916, 0.02111502,
        0.01363283],
       [0.01134539, 0.02074135, 0.010227  , ..., 0.01111425, 0.0178633 ,
        0.01426977],
       [0.01436835, 0.01653408, 0.01137002, ..., 0.01388402, 0.02148109,
        0.01175293]], dtype=float32)
```

Now, we are in the position to calculate the cost of training with **loss function**, and keep feedng the network to learn it. But, the question is: what the LSTM networks learn?

```
grad_clip =5.
tvars = tf.trainable_variables()
tvars
```

```
[<tf.Variable 'rnnlm/softmax_w:0' shape=(128, 65) dtype=float32_ref>,
 <tf.Variable 'rnnlm/softmax_b:0' shape=(65,) dtype=float32_ref>,
 <tf.Variable 'rnnlm/embedding:0' shape=(65, 128) dtype=float32_ref>,
 <tf.Variable 'rnnlm/multi_rnn_cell/cell_0/basic_rnn_cell/kernel:0' shape=(256, 128) dtype=float32_ref>,
 <tf.Variable 'rnnlm/multi_rnn_cell/cell_0/basic_rnn_cell/bias:0' shape=(128,) dtype=float32_ref>]
```

All together

Now, let's put all of parts together in a class, and train the model:

```
class LSTMModel():
    def __init__(self,sample=False):
        rnn_size = 128 # size of RNN hidden state vector
        batch_size = 60 # minibatch size, i.e. size of dataset in each epoch
        seq_length = 50 # RNN sequence length
        num_layers = 2 # number of layers in the RNN
        vocab_size = 65
        grad_clip = 5.
        if sample:
            print(">> sample mode:")
            batch_size = 1
            seq_length = 1
        # The core of the model consists of an LSTM cell that processes one char at a time and computes probabilities of the possible continuations of the char.
        basic_cell = tf.contrib.rnn.BasicRNNCell(rnn_size)
        # model.cell.state_size is (128, 128)
        self.stacked_cell = tf.contrib.rnn.MultiRNNCell([basic_cell] * num_layers)

        self.input_data = tf.placeholder(tf.int32, [batch_size, seq_length], name="input_data")
        self.targets = tf.placeholder(tf.int32, [batch_size, seq_length], name="targets")
        # Initial state of the LSTM memory.
        # The memory state of the network is initialized with a vector of zeros and gets updated after reading each char.
        self.initial_state = stacked_cell.zero_state(batch_size, tf.float32) #why batch_size

        with tf.variable_scope('rnnlm_class1'):
            softmax_w = tf.get_variable("softmax_w", [rnn_size, vocab_size]) #128x65
            softmax_b = tf.get_variable("softmax_b", [vocab_size]) # 1x65
            with tf.device("/cpu:0"):
                embedding = tf.get_variable("embedding", [vocab_size, rnn_size])  #65x128
                inputs = tf.split(tf.nn.embedding_lookup(embedding, self.input_data), seq_length, 1)
                inputs = [tf.squeeze(input_, [1]) for input_ in inputs]
                #inputs = tf.split(em, seq_length, 1)
```

```
        # The value of state is updated after processing each batch of chars.
        outputs, last_state = tf.contrib.legacy_seq2seq.rnn_decoder(inputs, self.initial_state, self.stacked_cell, loop_function=None, scope='rnnlm_class1')
        output = tf.reshape(tf.concat(outputs,1), [-1, rnn_size])
        self.logits = tf.matmul(output, softmax_w) + softmax_b
        self.probs = tf.nn.softmax(self.logits)
        loss = tf.contrib.legacy_seq2seq.sequence_loss_by_example([self.logits],
                [tf.reshape(self.targets, [-1])],
                [tf.ones([batch_size * seq_length])],
                vocab_size)
        self.cost = tf.reduce_sum(loss) / batch_size / seq_length
        self.final_state = last_state
        self.lr = tf.Variable(0.0, trainable=False)
        tvars = tf.trainable_variables()
        grads, _ = tf.clip_by_global_norm(tf.gradients(self.cost, tvars),grad_clip)
        optimizer = tf.train.AdamOptimizer(self.lr)
        self.train_op = optimizer.apply_gradients(zip(grads, tvars))


    def sample(self, sess, chars, vocab, num=200, prime='The ', sampling_type=1):
        state = sess.run(self.stacked_cell.zero_state(1, tf.float32))
        #print state
        for char in prime[:-1]:
            x = np.zeros((1, 1))
            x[0, 0] = vocab[char]
            feed = {self.input_data: x, self.initial_state:state}
            [state] = sess.run([self.final_state], feed)

        def weighted_pick(weights):
            t = np.cumsum(weights)
            s = np.sum(weights)
            return(int(np.searchsorted(t, np.random.rand(1)*s)))
```

```
        ret = prime
        char = prime[-1]
        for n in range(num):
            x = np.zeros((1, 1))
            x[0, 0] = vocab[char]
            feed = {self.input_data: x, self.initial_state:state}
            [probs, state] = sess.run([self.probs, self.final_state], feed)
            p = probs[0]

            if sampling_type == 0:
                sample = np.argmax(p)
            elif sampling_type == 2:
                if char == ' ':
                    sample = weighted_pick(p)
                else:
                    sample = np.argmax(p)
            else: # sampling_type == 1 default:
                sample = weighted_pick(p)

            pred = chars[sample]
            ret += pred
            char = pred
        return ret
```

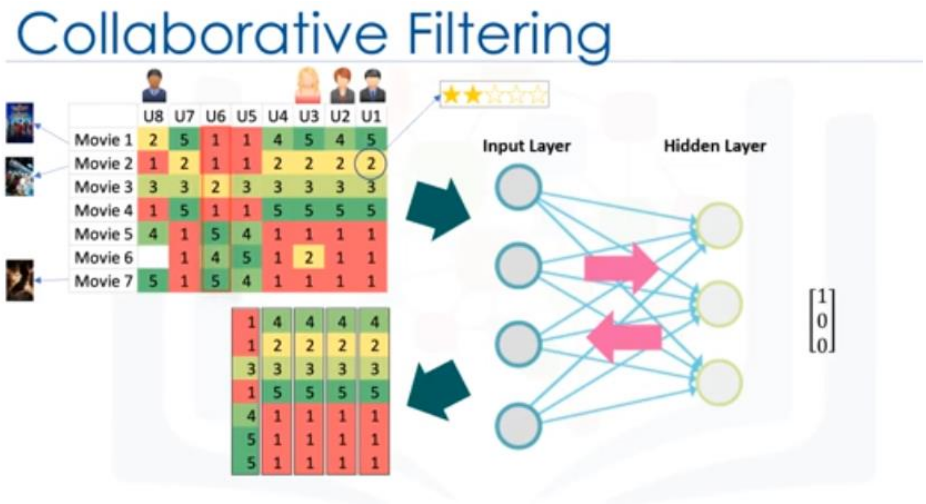
Train usinng LSTMModel class

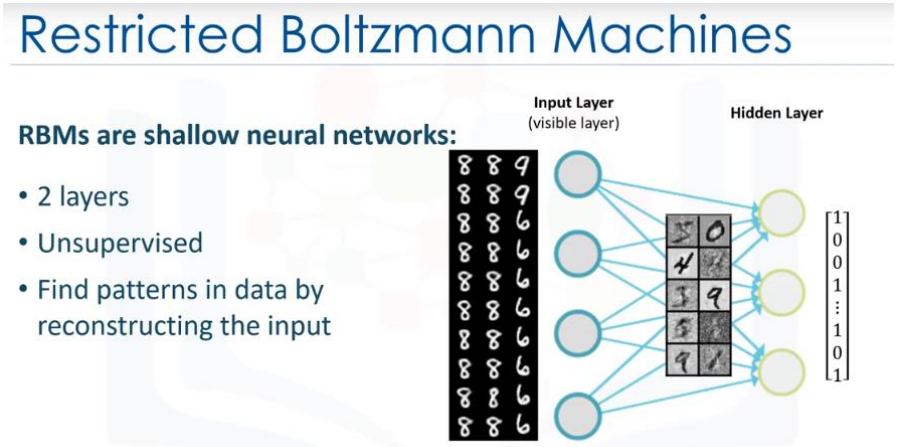
We can train our model through feeding batches:

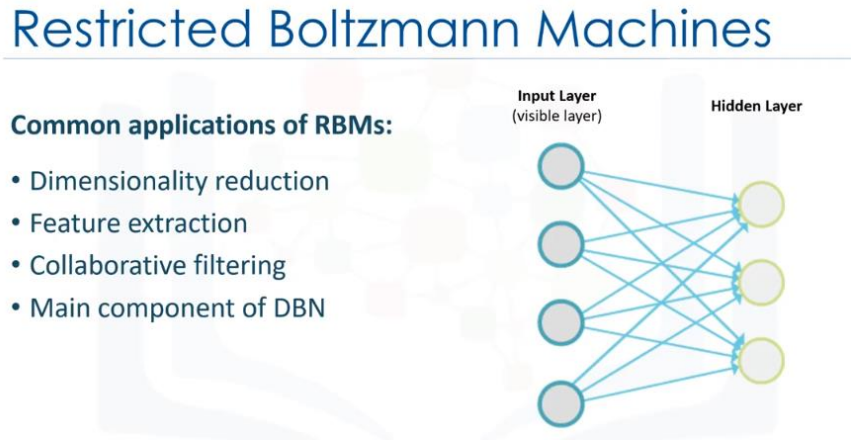
```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for e in range(num_epochs): # num_epochs is 5 for test, but should be higher
        sess.run(tf.assign(model.lr, learning_rate * (decay_rate ** e)))
        data_loader.reset_batch_pointer()
        state = sess.run(model.initial_state) # (2x[60x128])
        for b in range(data_loader.num_batches): #for each batch
            start = time.time()
            x, y = data_loader.next_batch()
            feed = {model.input_data: x, model.targets: y, model.initial_state:state}
            train_loss, state, _ = sess.run([model.cost, model.final_state, model.train_op], feed)
            end = time.time()
        print("{}/{} (epoch {}), train_loss = {:.3f}, time/batch = {:.3f}" \
                .format(e * data_loader.num_batches + b, num_epochs * data_loader.num_batches, e, train_loss, end - start))
        with tf.variable_scope("rnn", reuse=True):
            sample_model = LSTMModel(sample=True)
            print (sample_model.sample(sess, data_loader.chars , data_loader.vocab, num=50, prime='The ', sampling_type=1))
            print ('---------------------------------')
```

MODULE 4 – RESTRICTED BOLZMANN MACHINES (RBM)

INTRO TO RBMs



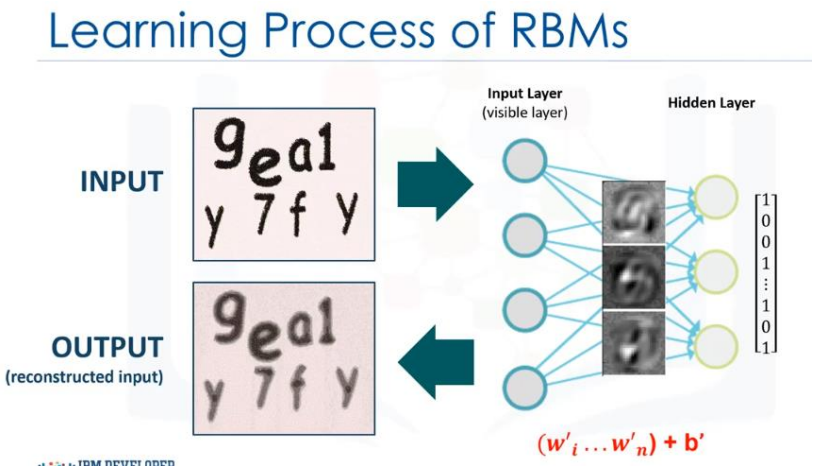


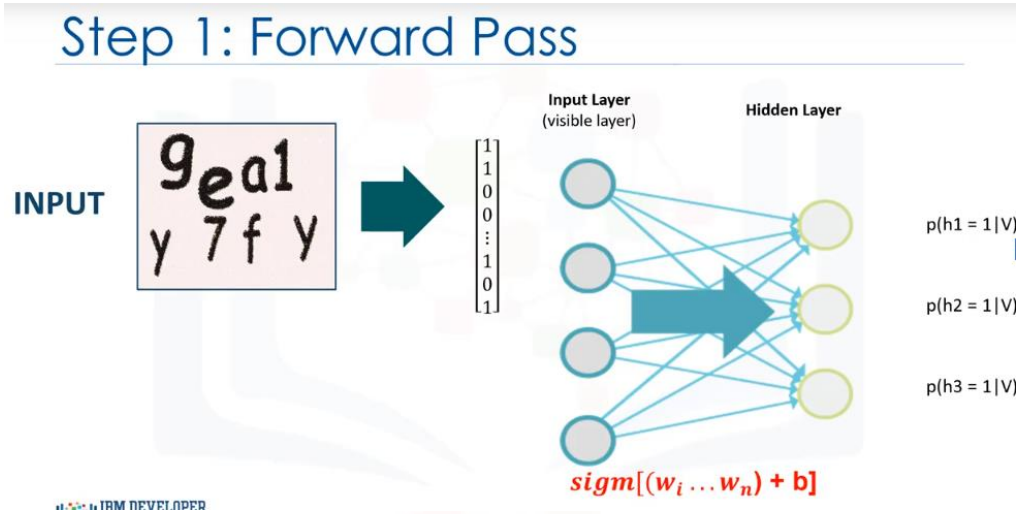


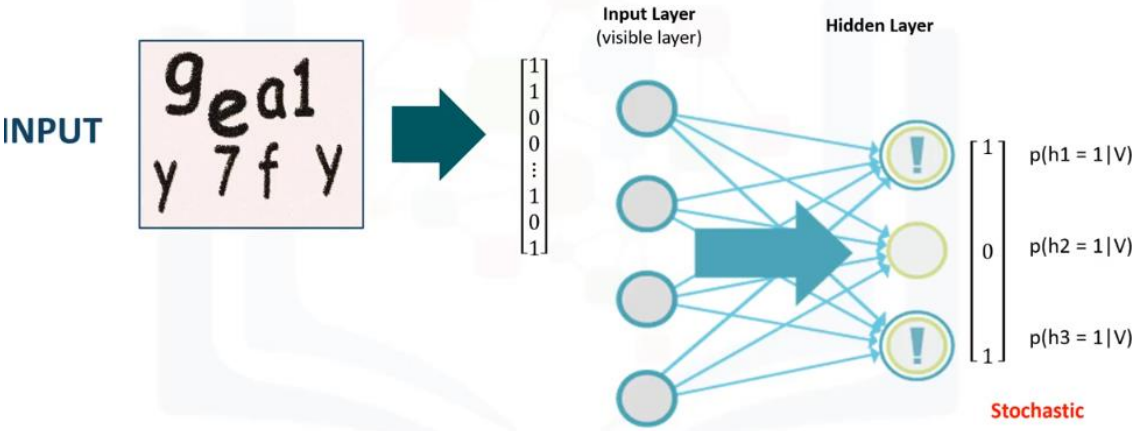
Start of transcript. Skip to the end.
Hello, and welcome! In this video we'll be providing an introduction to Restricted Boltzmann Machines. So let's get started!
Imagine that we have access to a matrix of the viewer ratings of a certain number of movies on Netflix, where each row corresponds to a movie, and each column corresponds to a user's rating. For the sake of simplicity, let's say we're only examining 8 users and their ratings of 7 movies.
The value in each cell shows the score that the user has given to the movie after watching it, and is based on a rating scale of 1 to 5.
Also, imagine that we have a type of neural network, that has only 2 layers, the input layer and the hidden layer. Let's also assume that this network has learned in such a way that it can reconstruct the input vectors.
For example, when you feed the first user vector into the network, it goes through the network, and finally fires up some units in the hidden layer.
Then, the values of the hidden layer will be fed back into the network, and a vector, which is almost the same as the input vector, is reconstructed as output.
We can think of it as making guesses about the input data.
You feed the second user's ratings, which are not very different from first user, and thus, the same hidden units will be turned on, and the network output would be the same as the first reconstructed vector. We can repeat it for the third user.
And for user number 4. Now, let's feed a user that has a completely different idea about these movies. As you can see, this particular user was not a fan of the first movie. When we feed the respective rating values into the network, different hidden units get turned on, and a different vector is reconstructed,

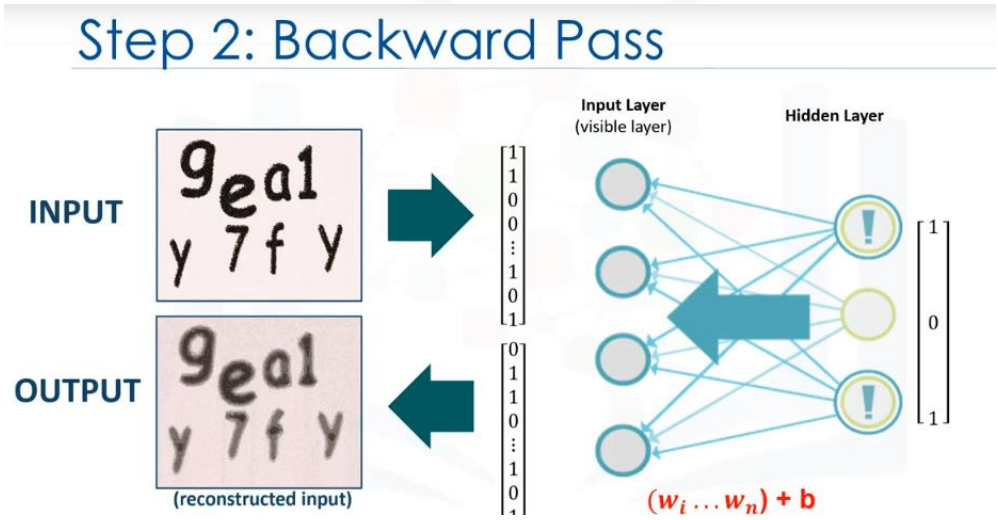
which is almost the same as User number 5's preferences. It is the same for user number 6.
And the process can be repeated for the other users.
Now, let's look at User number 8. He hasn't watched movie 6, but does have
some preferences that are almost the same as Users 5 and 6, right?
Let's feed this vector into our network. It'll fire up the same hidden units as Users
5 and 6. And, feeding back these values, we'll reconstruct
a new vector. Look at the expected value for movie 6 in
the reconstructed vector. Using this value, it's not hard to imagine
that user number 8 might be interested in this movie, even though he hasn't watched it
yet. Maybe we can even recommend it to him, yes?
In fact, it is a way of solving collaborative filtering, which is a type of recommender
system engine. And the network that can make such a model
is called a Restricted Boltzmann Machine.
Restricted Boltzmann Machines (or RBMs, for short), are shallow neural networks that only
have two layers. They are an unsupervised method used to find
patterns in data by reconstructing the input. The first layer of the RBM is called the visible
layer, and the second layer is the hidden layer.
We say that they are "restricted" because neurons within the same layer are not connected.
Feeding the input data, the network learns its weights.
Then, feeding an input image, the values that appear in the hidden layer can be considered
as features learned automatically from the input data.
And, as there are a smaller number of units in the hidden units of an RBM, we can tell
that the values in the hidden units are a good representation of data that are lower
in dimensionality when compared to the original data.
Restricted Boltzmann Machines are useful in many applications like dimensionality
reduction,
feature extraction, and collaborative filtering, just to name a few.
On top of that, RBMs are used as the main block of another type of Deep Neural Network,
which is called, Deep Belief Networks, which we'll be talking about later.
By now, you should have basic knowledge about RBMs and their applications.
Thanks for watching this video.
End of transcript. Skip to the start.

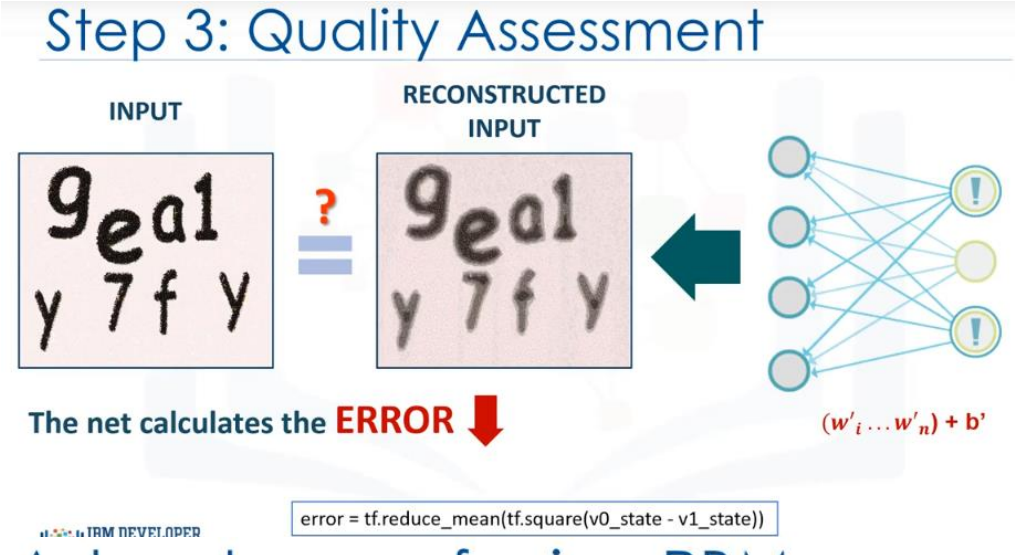
RESRICTED BOLZMANN MACHINE

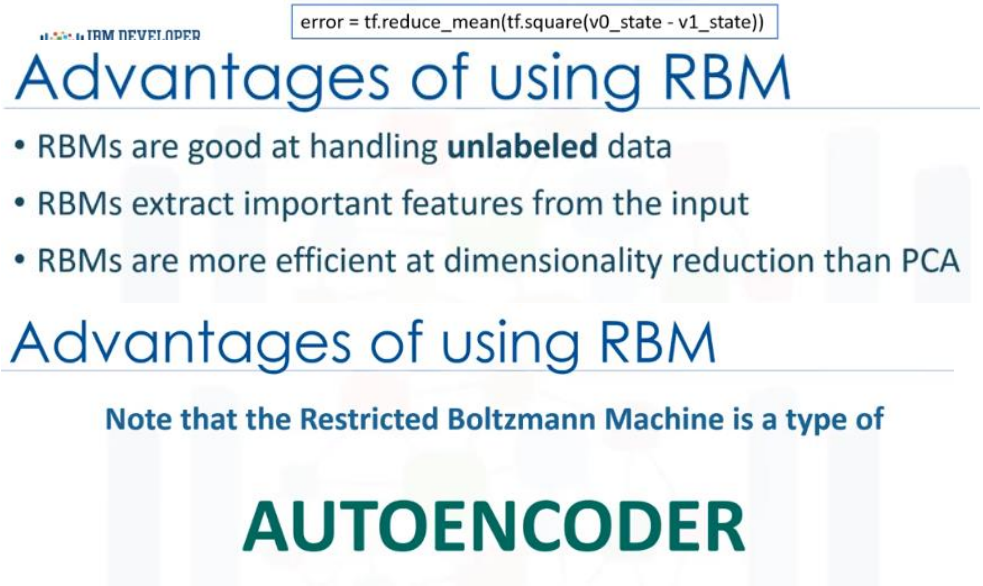



Step 1: Forward Pass
INPUT
Input Layer
(visible layer)
Hidden Layer
p(h1 = 1|V)
p(h2 = 1|V)
p(h3 = 1|V)
$sigm[(w_i \dots w_n) + b]$


INPUT
Input Layer
(visible layer)
Hidden Layer
p(h1 = 1|V)
p(h2 = 1|V)
p(h3 = 1|V)
Stochastic


Step 2: Backward Pass
INPUT
OUTPUT
(reconstructed input)
Input Layer
(visible layer)
Hidden Layer
$(w_i \dots w_n) + b$





Hello, and welcome! In this video we will provide an overview
of RBMs, with a focus on understanding how they work.
Let's get started
Let's say that we provide an image as input to an RBM.
The pixels are processed by the input layer, which is also known as the visible layer.
RBMs learn patterns and extract important features in data by reconstructing the input.
So, the learning process consists of several forward and backward passes, where the RBM tries to reconstruct the input data. The weights of the neural net are adjusted
in such a way that the RBM can find the relationships among input features, and then determines
which features are relevant.
After training is complete, the net is able to reconstruct the input based on what it learned. The reconstructed image, here, is only a representation
of what happens. What's important to take from this example,
though, is that the RBM can automatically extract meaningful features from a given input in the training process. In fact, a trained RBM can reveal which features
are the most important ones when detecting patterns.
It can also represent each image with some hidden values, also referred to, as latent values. Now let's look at an RBM's training process,
in which, three major steps are repeated.

The first step is the forward pass. In the forward pass, the input image is converted to binary values, and then, the vector input is fed into the network, where its values are multiplied by weights, and an overall bias, in each hidden unit.
Then, the result goes to an activation function, such as the sigmoid function, which represents the probability of turning each individual hidden-unit on, or in other words, the probability of the node activation.
Then, a sample is drawn from this probability distribution, and it finds which neurons may or may not activate. This means, it makes stochastic decisions about whether or not to transmit that hidden data.
The intuition behind the sampling, is that there are some random hidden variables, and by sampling from the hidden layer, you can reproduce sample variants encountered during training. So, as you can see, the forward pass translates the inputs into a set of binary values that get represented in the hidden layer.
Then we get to step 2: the backward pass. In the backward pass, the activated neurons in the hidden layer send the results back to the visible layer, where the input will be reconstructed. During this step, the data that is passed backwards is also combined with the same weights and overall bias that were used in the forward pass. So, once the information gets to the visible layer, it is in the shape of the probability distribution of the input values, given the hidden values. And sampling the distribution, the input is reconstructed. So, as you can see, the Backward pass is about making guesses about the probability distribution of the original input.
Now, let's look at step 3.
Step 3 consists of assessing the quality of the reconstruction by comparing it to the original data. The RBM then calculates the error and adjusts the weights and bias in order to minimize it.
That is, in each epoch, we compute the "error" as a sum of the squared difference between step 1 and the next step. These 3 steps are repeated until the error is deemed sufficiently low.
Now, let's touch on a few reasons why RBMs are such a great tool.
A big advantage of RBMS is that they excel when working with unlabeled data.
Many important real-world datasets are unlabeled, like videos, photos, and audio files, so RBMs provide a lot of benefit in these types of unsupervised learning problems.
Another advantage is that during the learning process, the RBM extracts features from the input data, decides which features are relevant, and how to best combine them to form patterns.
RBMs are also generally more efficient at dimensionality reduction when compared to principal component analysis, which is considered a popular alternative.
Finally, RBMs learn from the data, they actually encode their own structure.
This is why they're grouped into a larger family of models known as the Autoencoders.
However, Restricted Boltzmann Machines differ from Autoencoders in that they use a stochastic approach, rather than a deterministic approach.
This concludes this video, Thanks for watching!
End of transcript. Skip to the start.

>> LAB
Lab RBM MNIST

RESTRICTED BOLTZMANN MACHINES

Introduction

Restricted Boltzmann Machine (RBM): RBMs are shallow neural nets that learn to reconstruct data by themselves in an unsupervised fashion.

Why are RBMs important?

It can automatically extract **meaningful** features from a given input.

How does it work?

RBM is a 2 layer neural network. Simply, RBM takes the inputs and translates those into a set of binary values that represents them in the hidden layer. Then, these numbers can be translated back to reconstruct the inputs. Through several forward and backward passes, the RBM will be trained, and a trained RBM can reveal which features are the most important ones when detecting patterns.

What are the applications of RBM?

RBM is useful for Collaborative Filtering, dimensionality reduction, classification, regression, feature learning, topic modeling and even **Deep Belief Networks**.

Is RBM a generative or Discriminative model?

RBM is a generative model. Let me explain it by first, see what is different between discriminative and generative models:
Discriminative: Consider a classification problem in which we want to learn to distinguish between Sedan cars (y = 1) and SUV cars (y = 0), based on some features of cars. Given a training set, an algorithm like logistic regression tries to find a straight line—that is, a decision boundary—that separates the suv and sedan.
Generative: looking at cars, we can build a model of what Sedan cars look like. Then, looking at SUVs, we can build a separate model of what SUV cars look like. Finally, to classify a new car, we can match the new car against the Sedan model, and match it against the SUV model, to see whether the new car looks more like the SUV or Sedan.

Generative Models specify a probability distribution over a dataset of input vectors. We can do both supervise and unsupervised tasks with generative models:

- In an unsupervised task, we try to form a model for P(x), where P is the probability given x as an input vector.
- In the supervised task, we first form a model for P(x|y), where P is the probability of x given y(the label for x). For example, if y = 0 indicates whether a car is a SUV or y = 1 indicates indicate a car is a Sedan, then p(x|y = 0) models the distribution of SUVs' features, and p(x|y = 1) models the distribution of Sedans' features. If we manage to find P(x|y) and P(y), then we can use `Bayes rule` to estimate P(y|x), because:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

Now the question is, can we build a generative model, and then use it to create synthetic data by directly sampling from the modeled probability distributions? Lets see.

Initialization

First we have to load the utility file which contains different utility functions that are not connected in any way to the networks presented in the tutorials, but rather help in processing the outputs into a more understandable way.

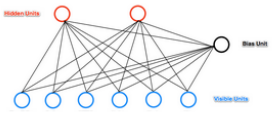
```
import urllib.request
with urllib.request.urlopen("http://deeplearning.net/tutorial/code/utils.py") as url:
    response = url.read()
target = open('utils.py', 'w')
target.write(response.decode('utf-8'))
target.close()
```

Now, we load in all the packages that we use to create the net including the TensorFlow package:

```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
#!pip install pillow
from PIL import Image
from utils import tile_raster_images
import matplotlib.pyplot as plt
%matplotlib inline
```

RBM layers

An RBM has two layers. The first layer of the RBM is called the **visible** (or input layer). Imagine that our toy example, has only vectors with 7 values, so the visible layer must have j=7 input nodes. The second layer is the **hidden** layer, which possesses i neurons in our case. Each hidden node can have either 0 or 1 values (i.e., si = 1 or si = 0) with a probability that is a logistic function of the inputs it receives from the other j visible units, called for example, p(si = 1). For our toy sample, we'll use 2 nodes in the hidden layer, so i = 2.



Each node in the first layer also has a **bias**. We will denote the bias as "v_bias" for the visible units. The **v_bias** is shared among all visible units.

Here we define the **bias** of second layer as well. We will denote the bias as "h_bias" for the hidden units. The **h_bias** is shared among all hidden units

```
v_bias = tf.placeholder("float", [7])
h_bias = tf.placeholder("float", [2])
```

We have to define weights among the input layer and hidden layer nodes. In the weight matrix, the number of rows are equal to the input nodes, and the number of columns are equal to the output nodes. Let **W** be the Tensor of 7x2 (7 - number of visible neurons, 2 - number of hidden neurons) that represents weights between neurons.

```
W = tf.constant(np.random.normal(loc=0.0, scale=1.0, size=(7, 2)).astype(np.float32))
```

What RBM can do after training?

Think RBM as a model that has been trained based on images of a dataset of many SUV and Sedan cars. Also, imagine that the RBM network has only two hidden nodes, one for the weight and, and one for the size of cars, which in a sense, their different configurations represent different cars, one represent SUV cars and one for Sedan. In a training process, through many forward and backward passes, RBM adjust its weights to send a stronger signal to either the SUV node (0, 1) or the Sedan node (1, 0) in the hidden layer, given the pixels of images. Now, given a SUV in hidden layer, which distribution of pixels should we expect? RBM can give you 2 things. First, it encodes your images in hidden layer. Second, it gives you the probability of observing a case, given some hidden values.

How to inference?

RBM has two phases:

- Forward Pass
- Backward Pass or Reconstruction

Phase 1) Forward pass: Input one training sample (one image) **X** through all visible nodes, and pass it to all hidden nodes. Processing happens in each node in the hidden layer. This computation begins by making stochastic decisions about whether to transmit that input or not (i.e. to determine the state of each hidden layer). At the hidden layer's nodes, **X** is multiplied by a W_{ij} and added to **h_bias**. The result of those two operations is fed into the sigmoid function, which produces the node's output, $p(h_j)$, where j is the unit number.

$p(h_j) = \sigma(\sum_i w_{ij} x_i)$, where $\sigma()$ is the logistic function.

Now lets see what $p(h_j)$ represents. In fact, it is the probabilities of the hidden units. And, all values together are called **probability distribution**. That is, RBM uses inputs x to make predictions about hidden node activations. For example, imagine that the values of h_p for the first training item is [0.51 0.84]. It tells you what is the conditional probability for each hidden neuron to be at Phase 1):

- $p(h_1 = 1|V) = 0.51$
- $(h_2 = 1|V) = 0.84$

As a result, for each row in the training set, **a vector/tensor** is generated, which in our case it is of size [1x2], and totally n vectors ($p(h)$=[nx2]).

We then turn unit h_j on with probability $p(h_j|V)$, and turn it off with probability $1 - p(h_j|V)$.

Therefore, the conditional probability of a configuration of h given v (for a training sample) is:

$$p(\mathbf{h} \mid \mathbf{v}) = \prod_{j=0}^{H} p(h_j \mid \mathbf{v})$$

Now, sample a hidden activation vector **h** from this probability distribution $p(h_j)$. That is, we sample the activation vector from the probability distribution of hidden layer values.

Before we go further, let's look at a toy example for one case out of all input. Assume that we have a trained RBM, and a very simple input vector such as [1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0], lets see what would be the output of forward pass:

```
sess = tf.Session()
X = tf.constant([[1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]])
v_state = X
print ("Input: ", sess.run(v_state))

h_bias = tf.constant([0.1, 0.1])
print ("hb: ", sess.run(h_bias))
print ("w: ", sess.run(W))

# Calculate the probabilities of turning the hidden units on:
h_prob = tf.nn.sigmoid(tf.matmul(v_state, W) + h_bias)  #probabilities of the hidden units
print ("p(h|v): ", sess.run(h_prob))

# Draw samples from the distribution:
h_state = tf.nn.relu(tf.sign(h_prob - tf.random_uniform(tf.shape(h_prob)))) #states
print ("h0 states:", sess.run(h_state))
```

```
Input:  [[1. 0. 0. 1. 0. 0. 0.]]
hb:  [0.1 0.1]
w:  [[-1.3240469  -0.65724415]
 [ 1.0220742  -0.16399099]
 [-0.7053166   0.10699695]
 [-0.00470642 -0.5750789 ]
 [ 0.7478376   1.0539196 ]
 [-1.5976839  -0.8808901 ]
 [ 0.8178901  -0.90882224]]
p(h|v):  [[0.22639973 0.24373265]]
h0 states: [[0. 0.]]
```

Phase 2) Backward Pass (Reconstruction): The RBM reconstructs data by making several forward and backward passes between the visible and hidden layers.

So, in the second phase (i.e. reconstruction phase), the samples from the hidden layer (i.e. h) play the role of input. That is, **h** becomes the input in the backward pass. The same weight matrix and visible layer biases are used to go through the sigmoid function. The produced output is a reconstruction which is an approximation of the original input.

```
vb = tf.constant([0.1, 0.2, 0.1, 0.1, 0.1, 0.2, 0.1])
print ("b: ", sess.run(vb))
v_prob = sess.run(tf.nn.sigmoid(tf.matmul(h_state, tf.transpose(W)) + vb))
print ("p(v|h): ", v_prob)
v_state = tf.nn.relu(tf.sign(v_prob - tf.random_uniform(tf.shape(v_prob))))
print ("v probability states: ", sess.run(v_state))
```

```
b:  [0.1 0.2 0.1 0.1 0.1 0.2 0.1]
p(v|h):  [[0.5249792  0.54983395 0.5249792  0.5249792  0.5249792  0.54983395
  0.5249792 ]]
v probability states:  [[0. 1. 1. 0. 0. 0. 1.]]
```

RBM learns a probability distribution over the input, and then, after being trained, the RBM can generate new samples from the learned probability distribution. As you know, **probability distribution**, is a mathematical function that provides the probabilities of occurrence of different possible outcomes in an experiment.

The (conditional) probability distribution over the visible units v is given by

$$p(\mathbf{v} \mid \mathbf{h}) = \prod_{i=0}^{V} p(v_i \mid \mathbf{h}),$$

where,

$$p(v_i \mid \mathbf{h}) = \sigma\left(a_i + \sum_{j=0}^{H} w_{ji} h_j\right)$$

so, given current state of hidden units and weights, what is the probability of generating [1. 0. 0. 1. 0. 0. 0.] in reconstruction phase, based on the above **probability distribution** function?

```
inp = sess.run(X)
print(inp)
print(v_prob[0])
v_probability = 1
for elm, p in zip(inp[0],v_prob[0]) :
    if elm ==1:
        v_probability *= p
    else:
        v_probability *= (1-p)
v_probability
```

```
[[1. 0. 0. 1. 0. 0. 0.]]
[0.5249792  0.54983395 0.5249792  0.5249792  0.5249792  0.54983395
 0.5249792 ]
0.005986425374900215
```

How similar X and V vectors are? Of course, the reconstructed values most likely will not look anything like the input vector because our network has not trained yet. Our objective is to train the model in such a way that the input vector and reconstructed vector to be same. Therefore, based on how different the input values look to the ones that we just reconstructed, the weights are adjusted.

MNIST

We will be using the MNIST dataset to practice the usage of RBMs. The following cell loads the MNIST dataset.

```
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mnist.test.images, mnist.test.labels
```

```
WARNING:tensorflow:From <ipython-input-8-a0c1bc5755ed>:1: read_data_sets (from tensorflow.contrib.learn.python.learn.datasets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use alternatives such as official/mnist/dataset.py from tensorflow/models.
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.py:260: maybe_download (from tensorflow.contrib.learn.python.learn.datasets.base)
is deprecated and will be removed in a future version.
Instructions for updating:
Please write your own downloading logic.
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.py:262: extract_images (from tensorflow.contrib.learn.python.learn.datasets.mnist)
is deprecated and will be removed in a future version.
Instructions for updating:
Please use tf.data to implement this functionality.
Extracting MNIST_data/train-images-idx3-ubyte.gz
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.py:267: extract_labels (from tensorflow.contrib.learn.python.learn.datasets.mnist)
is deprecated and will be removed in a future version.
Instructions for updating:
Please use tf.data to implement this functionality.
Extracting MNIST_data/train-labels-idx1-ubyte.gz
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.py:110: dense_to_one_hot (from tensorflow.contrib.learn.python.learn.datasets.mnis
t) is deprecated and will be removed in a future version.
Instructions for updating:
Please use tf.one_hot on tensors.
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.py:290: DataSet.__init__ (from tensorflow.contrib.learn.python.learn.datasets.mnis
t) is deprecated and will be removed in a future version.
Instructions for updating:
Please use alternatives such as official/mnist/dataset.py from tensorflow/models.
```

Lets look at the dimension of the images.

```
(784,)
```

MNIST images have 784 pixels, so the visible layer must have 784 input nodes. For our case, we'll use 50 nodes in the hidden layer, so i = 50.

```
vb = tf.placeholder("float", [784])
hb = tf.placeholder("float", [50])
```

Let **W** be the Tensor of 784x50 (784 - number of visible neurons, 50 - number of hidden neurons) that represents weights between the neurons.

```
W = tf.placeholder("float", [784, 50])
```

Lets define the visible layer:

```
v0_state = tf.placeholder("float", [None, 784])
```

Now, we can define hidden layer:

```
h0_prob = tf.nn.sigmoid(tf.matmul(v0_state, W) + hb)  #probabilities of the hidden units
h0_state = tf.nn.relu(tf.sign(h0_prob - tf.random_uniform(tf.shape(h0_prob)))) #sample_h_given_X
```

Now, we define reconstruction part:

```
v1_prob = tf.nn.sigmoid(tf.matmul(h0_state, tf.transpose(W)) + vb)
v1_state = tf.nn.relu(tf.sign(v1_prob - tf.random_uniform(tf.shape(v1_prob)))) #sample_v_given_h
```

What is objective function?

Goal: Maximize the likelihood of our data being drawn from that distribution

Calculate error:
In each epoch, we compute the "error" as a sum of the squared difference between step 1 and step n, e.g the error shows the difference between the data and its reconstruction.

Note: tf.reduce_mean computes the mean of elements across dimensions of a tensor.

How to train the model?

Warning!! The following part discuss how to train the model which needs some algebra background. Still, you can skip this part and run the next cells.
As mentioned, we want to give a high probability to the input data we train on. So, in order to train an RBM, we have to maximize the product of probabilities assigned to all rows v (images) in the training set V (a matrix, where each row of it is treated as a visible vector v):

$$\arg\max_W \prod_{v=V} P(v)$$

Which is equivalent, maximizing the expected log probability of V:

$$\arg\max_W \mathbb{E}\left[\sum_{v=V} \log P(v)\right]$$

So, we have to update the weights wij to increase p(v) for all v in our training data during training. So we have to calculate the derivative:

$$\frac{\partial \log p(\mathbf{v})}{\partial w_{ij}}$$

This cannot be easily done by typical **gradient descent (SGD)**, so we can use another approach, which has 2 steps:

1. Gibbs Sampling
2. Contrastive Divergence

Gibbs Sampling

First, given an input vector v we are using p(h|v) for prediction of the hidden values h.

- $p(h|v) = sigmoid(X \otimes W + hb)$
- h0 = sampleProb(h0)

Then, knowing the hidden values, we use p(v|h) for reconstructing of new input values v.

- $p(v|h) = sigmoid(h0 \otimes transpose(W) + vb)$
- $v1 = sampleProb(v1)$ (Sample v given h)

This process is repeated k times. After k iterations we obtain an other input vector vk which was recreated from original input values v0 or X.

Reconstruction steps:

- Get one data point from data set, like *x*, and pass it through the net
- Pass 0: (x) ⇒ (h0) ⇒ (v1) (v1 is reconstruction of the first pass)
- Pass 1: (v1) ⇒ (h1) ⇒ (v2) (v2 is reconstruction of the second pass)
- Pass 2: (v2) ⇒ (h2) ⇒ (v3) (v3 is reconstruction of the third pass)
- Pass n: (vk) ⇒ (hk+1) ⇒ (vk+1)(vk is reconstruction of the nth pass)

What is sampling here (sampleProb)?

In forward pass: We randomly set the values of each hi to be 1 with probability $sigmoid(v \otimes W + hb)$.

- To sample h given v means to sample from the conditional probability distribution P(h|v). It means that you are asking what are the probabilities of getting a specific set of values for the hidden neurons, given the values v for the visible neurons, and sampling from this probability distribution. In reconstruction: We randomly set the values of each vi to be 1 with probability $sigmoid(h \otimes transpose(W) + vb)$.

contrastive divergence (CD-k)

The update of the weight matrix is done during the Contrastive Divergence step.
Vectors v0 and vk are used to calculate the activation probabilities for hidden values h0 and hk. The difference between the outer products of those probabilities with input vectors v0 and vk results in the update matrix:

$$\Delta W = v0 \otimes h0 - vk \otimes hk$$

Contrastive Divergence is actually matrix of values that is computed and used to adjust values of the W matrix. Changing W incrementally leads to training of W values. Then on each step (epoch), W is updated to a new value W' through the equation below:

$$W' = W + alpha * \Delta W$$

What is Alpha?

Here, alpha is some small step rate and is also known as the "learning rate".

Ok, lets assume that k=1, that is we just get one more step:

```
h1_prob = tf.nn.sigmoid(tf.matmul(v1_state, W) + hb)
h1_state = tf.nn.relu(tf.sign(h1_prob - tf.random_uniform(tf.shape(h1_prob)))) #sample_h_given_X
```

```
alpha = 0.01
W_Delta = tf.matmul(tf.transpose(v0_state), h0_prob) - tf.matmul(tf.transpose(v1_state), h1_prob)
update_w = W + alpha * W_Delta
update_vb = vb + alpha * tf.reduce_mean(v0_state - v1_state, 0)
update_hb = hb + alpha * tf.reduce_mean(h0_state - h1_state, 0)
```

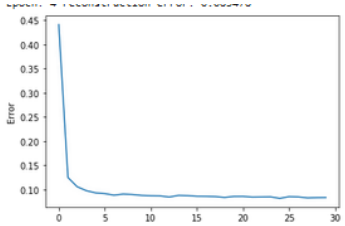
Let's start a session and initialize the variables:

```
cur_w = np.zeros([784, 50], np.float32)
cur_vb = np.zeros([784], np.float32)
cur_hb = np.zeros([50], np.float32)
prv_w = np.zeros([784, 50], np.float32)
prv_vb = np.zeros([784], np.float32)
prv_hb = np.zeros([50], np.float32)
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
```

Lets look at the error of the first run:

```
sess.run(err, feed_dict={v0_state: trX, W: prv_w, vb: prv_vb, hb: prv_hb})
```

```
0.4814648
```




```
#Parameters
epochs = 5
batchsize = 100
weights = []
errors = []

for epoch in range(epochs):
    for start, end in zip( range(0, len(trX), batchsize), range(batchsize, len(trX), batchsize)):
        batch = trX[start:end]
        cur_w = sess.run(update_w, feed_dict={ v0_state: batch, W: prv_w, vb: prv_vb, hb: prv_hb})
        cur_vb = sess.run(update_vb, feed_dict={v0_state: batch, W: prv_w, vb: prv_vb, hb: prv_hb})
        cur_hb = sess.run(update_hb, feed_dict={ v0_state: batch, W: prv_w, vb: prv_vb, hb: prv_hb})
        prv_w = cur_w
        prv_vb = cur_vb
        prv_hb = cur_hb
        if start % 10000 == 0:
            errors.append(sess.run(err, feed_dict={v0_state: trX, W: cur_w, vb: cur_vb, hb: cur_hb}))
            weights.append(cur_w)
    print ('Epoch: %d' % epoch,'reconstruction error: %f' % errors[-1])
plt.plot(errors)
plt.xlabel("Batch Number")
plt.ylabel("Error")
plt.show()
```

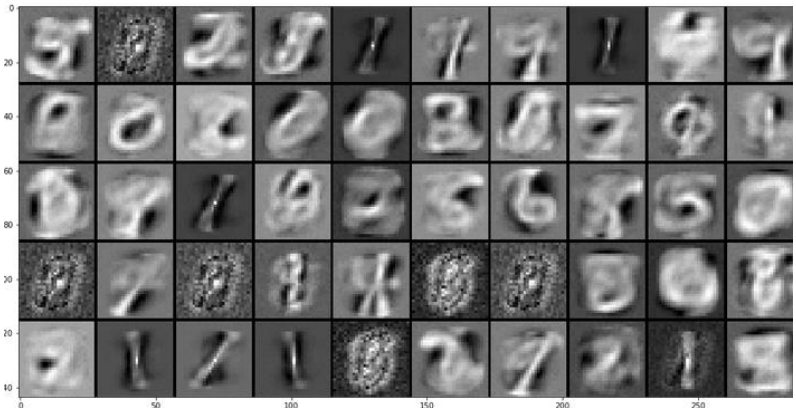
```
Epoch: 0 reconstruction error: 0.091645
Epoch: 1 reconstruction error: 0.086844
Epoch: 2 reconstruction error: 0.085454
Epoch: 3 reconstruction error: 0.084869
```

What is the final weight after training?

```
uw = weights[-1].T
print (uw) # a weight matrix of shape (50,784)
```

Learned features

We can take each hidden unit and visualize the connections between that hidden unit and each element in the input vector. In our case, we have 50 hidden units. Lets visualize those.



Let's plot the current weights: **tile_raster_images** helps in generating an easy to grasp image from a set of samples or weights. It transform the **uw** (with one flattened image per row of size 784), into an array (of size 25×20) in which images are reshaped and laid out like tiles on a floor.

```
tile_raster_images(X=cur_w.T, img_shape=(28, 28), tile_shape=(5, 10), tile_spacing=(1, 1))
import matplotlib.pyplot as plt
from PIL import Image
%matplotlib inline
image = Image.fromarray(tile_raster_images(X=cur_w.T, img_shape=(28, 28) ,tile_shape=(5, 10), tile_spacing=(1, 1)))
### Plot image
plt.rcParams['figure.figsize'] = (18.0, 18.0)
imgplot = plt.imshow(image)
imgplot.set_cmap('gray')
```

Each tile in the above visualization corresponds to a vector of connections between a hidden unit and visible layer's units.

Let's look at one of the learned weights corresponding to one of hidden units for example. In this particular square, the gray color represents weight = 0, and the whiter it is, the more positive the weights are (closer to 1). Conversely, the darker pixels are, the more negative the weights. The positive pixels will increase the probability of activation in hidden units (after multiplying by input/visible pixels), and negative pixels will decrease the probability of a unit hidden to be 1 (activated). So, why is this important? So we can see that this specific square (hidden unit) can detect a feature (e.g. a "/" shape) and if it exists in the input.

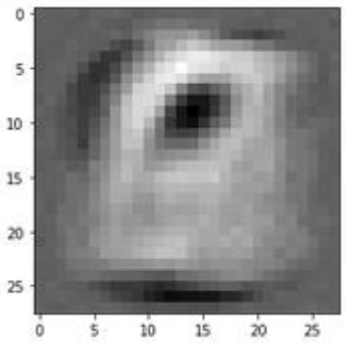
```
from PIL import Image
image = Image.fromarray(tile_raster_images(X =cur_w.T[10:11], img_shape=(28, 28),tile_shape=(1, 1), tile_spacing=(1, 1)))
### Plot image
plt.rcParams['figure.figsize'] = (4.0, 4.0)
imgplot = plt.imshow(image)
imgplot.set_cmap('gray')
```

Let's look at the reconstruction of an image now. Imagine that we have a destructed image of figure 3. Lets see if our trained network can fix it:

First we plot the image:

```
!wget -O destructed3.jpg  https://ibm.box.com/shared/static/vvm1b63uvuxq88vbw9znpwu5ol380mco.jpg
img = Image.open('destructed3.jpg')
img
```

Now let's pass this image through the net:



```
# convert the image to a 1d numpy array
sample_case = np.array(img.convert('I').resize((28,28))).ravel().reshape((1, -1))/255.0
```

Feed the sample case into the network and reconstruct the output:

```
hh0_p = tf.nn.sigmoid(tf.matmul(v0_state, W) + hb)
#hh0_s = tf.nn.relu(tf.sign(hh0_p - tf.random_uniform(tf.shape(hh0_p))))
hh0_s = tf.round(hh0_p)
hh0_p_val,hh0_s_val  = sess.run((hh0_p, hh0_s), feed_dict={ v0_state: sample_case, W: prv_w, hb: prv_hb})
print("Probability nodes in hidden layer:" ,hh0_p_val)
print("activated nodes in hidden layer:" ,hh0_s_val)

# reconstruct
vv1_p = tf.nn.sigmoid(tf.matmul(hh0_s_val, tf.transpose(W)) + vb)
rec_prob = sess.run(vv1_p, feed_dict={ hh0_s: hh0_s_val, W: prv_w, vb: prv_vb})
```

Here we plot the reconstructed image:

```
img = Image.fromarray(tile_raster_images(X=rec_prob, img_shape=(28, 28),tile_shape=(1, 1), tile_spacing=(1, 1)))
plt.rcParams['figure.figsize'] = (4.0, 4.0)
imgplot = plt.imshow(img)
imgplot.set_cmap('gray')
```

Lab Collaborative Filtering With RBM

RECOMMENDATION SYSTEM WITH A RESTRICTED BOLTZMANN MACHINE

Welcome to the **Recommendation System with a Restricted Boltzmann Machine** notebook. In this notebook, we study and go over the usage of a Restricted Boltzmann Machine (RBM) in a Collaborative Filtering based recommendation system. This system is an algorithm that recommends items by trying to find users that are similar to each other based on their item ratings. By the end of this notebook, you should have a deeper understanding of how Restricted Boltzmann Machines are applied, and how to build one using TensorFlow.

Table of Contents

Acquiring the Data

To start, we need to download the data we are going to use for our system. The datasets we are going to use were acquired by GroupLens and contain movies, users and movie ratings by these users.

After downloading the data, we will extract the datasets to a directory that is easily accessible.

```
!wget -O ./data/moviedataset.zip http://files.grouplens.org/datasets/movielens/ml-1m.zip
!unzip -o ./data/moviedataset.zip -d ./data
```

```
--2020-05-21 05:54:56--  http://files.grouplens.org/datasets/movielens/ml-1m.zip
Resolving files.grouplens.org (files.grouplens.org)... 128.101.65.152
Connecting to files.grouplens.org (files.grouplens.org)|128.101.65.152|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5917549 (5.6M) [application/zip]
Saving to: ‘./data/moviedataset.zip’

./data/moviedataset 100%[===================>]   5.64M  11.9MB/s    in 0.5s

2020-05-21 05:54:57 (11.9 MB/s) - ‘./data/moviedataset.zip’ saved [5917549/5917549]

Archive:  ./data/moviedataset.zip
   creating: ./data/ml-1m/
  inflating: ./data/ml-1m/movies.dat
  inflating: ./data/ml-1m/ratings.dat
  inflating: ./data/ml-1m/README
  inflating: ./data/ml-1m/users.dat
```

With the datasets in place, let's now import the necessary libraries. We will be using Tensorflow and Numpy together to model and initialize our Restricted Boltzmann Machine and Pandas to manipulate our datasets. To import these libraries, run the code cell below.

With the datasets in place, let's now import the necessary libraries. We will be using Tensorflow and Numpy together to model and initialize our Restricted Boltzmann Machine and Pandas to manipulate our datasets. To import these libraries, run the code cell below.

```
#Tensorflow library. Used to implement machine learning models
import tensorflow as tf
#Numpy contains helpful functions for efficient mathematical calculations
import numpy as np
#Dataframe manipulation library
import pandas as pd
#Graph plotting library
import matplotlib.pyplot as plt
%matplotlib inline
```

Loading in the Data

Let's begin by loading in our data with Pandas. The .dat files containing our data are similar to CSV files, but instead of using the ',' (comma) character to separate entries, it uses '::' (two colons) characters instead. To let Pandas know that it should separate data points at every '::', we have to specify the `sep='::'` parameter when calling the function.

Additionally, we also pass it the `header=None` parameter due to the fact that our files don't contain any headers.

Let's start with the movies.dat file and take a look at its structure:

```
#Loading in the movies dataset
movies_df = pd.read_csv('./data/ml-1m/movies.dat', sep='::', header=None, engine='python')
movies_df.head()
```

	0	1	2
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy

We can do the same for the ratings.dat file:

```
#Loading in the ratings dataset
ratings_df = pd.read_csv('./data/ml-1m/ratings.dat', sep='::', header=None, engine='python')
ratings_df.head()
```

	0	1	2	3
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275

So our **movies_df** variable contains a dataframe that stores a movie's unique ID number, title and genres, while our **ratings_df** variable stores a unique User ID number, a movie's ID that the user has watched, the user's rating to said movie and when the user rated that movie.

Let's now rename the columns in these dataframes so we can better convey their data more intuitively:

```
movies_df.columns = ['MovieID', 'Title', 'Genres']
movies_df.head()
```

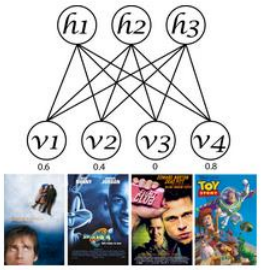
	MovieID	Title	Genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy

And our final ratings_df:

```
ratings_df.columns = ['UserID', 'MovieID', 'Rating', 'Timestamp']
ratings_df.head()
```

	UserID	MovieID	Rating	Timestamp
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291

The Restricted Boltzmann Machine model



The Restricted Boltzmann Machine model has two layers of neurons, one of which is what we call a visible input layer and the other is called a hidden layer. The hidden layer is used to learn features from the information fed through the input layer. For our model, the input is going to contain X neurons, where X is the amount of movies in our dataset. Each of these neurons will possess a normalized rating value varying from 0 to 1, where 0 meaning that a user has not watched that movie and the closer the value is to 1, the more the user likes the movie that neuron's representing. These normalized values, of course, will be extracted and normalized from the ratings dataset.
After passing in the input, we train the RBM on it and have the hidden layer learn its features. These features are what we use to reconstruct the input, which in our case, will predict the ratings for movies that user hasn't watched, which is exactly what we can use to recommend movies!

We will now begin to format our dataset to follow the model's expected input.

Formatting the Data

First let's see how many movies we have and see if the movie ID's correspond with that value:

```
len(movies_df)
```

3883

Now, we can start formatting the data into input for the RBM. We're going to store the normalized users ratings into as a matrix of user-rating called trX, and normalize the values.

```
user_rating_df = ratings_df.pivot(index='UserID', columns='MovieID', values='Rating')
user_rating_df.head()
```

MovieID	1	2	3	4	5	6	7	8	9	10	...	3943	3944	3945	3946	3947	3948	3949	3950	3951	3952
UserID																					
1	5.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	NaN	2.0	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

5 rows × 3706 columns

Lets normalize it now:

```
norm_user_rating_df = user_rating_df.fillna(0) / 5.0
trX = norm_user_rating_df.values
trX[0:5]
```

```
array([[1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

Setting the Model's Parameters

Next, let's start building our RBM with TensorFlow. We'll begin by first determining the number of neurons in the hidden layers and then creating placeholder variables for storing our visible layer biases, hidden layer biases and weights that connects the hidden layer with the visible layer. We will be arbitrarily setting the number of neurons in the hidden layers to 20. You can freely set this value to any number you want since each neuron in the hidden layer will end up learning a feature.

```
hiddenUnits = 20
visibleUnits =  len(user_rating_df.columns)
vb = tf.placeholder("float", [visibleUnits]) #Number of unique movies
hb = tf.placeholder("float", [hiddenUnits]) #Number of features we're going to learn
W = tf.placeholder("float", [visibleUnits, hiddenUnits])
```

We then move on to creating the visible and hidden layer units and setting their activation functions. In this case, we will be using the `tf.sigmoid` and `tf.relu` functions as nonlinear activations since it is commonly used in RBM's.

```
#Phase 1: Input Processing
v0 = tf.placeholder("float", [None, visibleUnits])
_h0 = tf.nn.sigmoid(tf.matmul(v0, W) + hb)
h0 = tf.nn.relu(tf.sign(_h0 - tf.random_uniform(tf.shape(_h0))))
#Phase 2: Reconstruction
_v1 = tf.nn.sigmoid(tf.matmul(h0, tf.transpose(W)) + vb)
v1 = tf.nn.relu(tf.sign(_v1 - tf.random_uniform(tf.shape(_v1))))
h1 = tf.nn.sigmoid(tf.matmul(v1, W) + hb)
```

Now we set the RBM training parameters and functions.

```
#Learning rate
alpha = 1.0
#Create the gradients
w_pos_grad = tf.matmul(tf.transpose(v0), h0)
w_neg_grad = tf.matmul(tf.transpose(v1), h1)
#Calculate the Contrastive Divergence to maximize
CD = (w_pos_grad - w_neg_grad) / tf.to_float(tf.shape(v0)[0])
#Create methods to update the weights and biases
update_w = W + alpha * CD
update_vb = vb + alpha * tf.reduce_mean(v0 - v1, 0)
update_hb = hb + alpha * tf.reduce_mean(h0 - h1, 0)
```

And set the error function, which in this case will be the Mean Absolute Error Function.

```
err = v0 - v1
err_sum = tf.reduce_mean(err * err)
```

We also have to initialize our variables. Thankfully, NumPy has a handy ,code>zeros function for this. We use it like so:

```
#Current weight
cur_w = np.zeros([visibleUnits, hiddenUnits], np.float32)
#Current visible unit biases
cur_vb = np.zeros([visibleUnits], np.float32)
#Current hidden unit biases
cur_hb = np.zeros([hiddenUnits], np.float32)
#Previous weight
prv_w = np.zeros([visibleUnits, hiddenUnits], np.float32)
#Previous visible unit biases
prv_vb = np.zeros([visibleUnits], np.float32)
#Previous hidden unit biases
prv_hb = np.zeros([hiddenUnits], np.float32)
sess = tf.Session()
sess.run(tf.global_variables_initializer())
```

Now we train the RBM with 15 epochs with each epoch using 10 batches with size 100. After training, we print out a graph with the error by epoch.

```
epochs = 15
batchsize = 100
errors = []
for i in range(epochs):
    for start, end in zip( range(0, len(trX), batchsize), range(batchsize, len(trX), batchsize)):
        batch = trX[start:end]
        cur_w = sess.run(update_w, feed_dict={v0: batch, W: prv_w, vb: prv_vb, hb: prv_hb})
        cur_vb = sess.run(update_vb, feed_dict={v0: batch, W: prv_w, vb: prv_vb, hb: prv_hb})
        cur_nb = sess.run(update_hb, feed_dict={v0: batch, W: prv_w, vb: prv_vb, hb: prv_hb})
        prv_w = cur_w
        prv_vb = cur_vb
        prv_hb = cur_hb
    errors.append(sess.run(err_sum, feed_dict={v0: trX, W: cur_w, vb: cur_vb, hb: cur_hb}))
    print (errors[-1])
plt.plot(errors)
plt.ylabel('Error')
plt.xlabel('Epoch')
plt.show()
```

```
0.059504203
0.050683744
0.04907286
0.047362775
0.04681908
0.046252735
0.04597408
0.045728832
0.045538023
0.0453459
```

Recommendation

We can now predict movies that an arbitrarily selected user might like. This can be accomplished by feeding in the user's watched movie preferences into the RBM and then reconstructing the input. The values that the RBM gives us will attempt to estimate the user's preferences for movies that he hasn't watched based on the preferences of the users that the RBM was trained on.

Lets first select a **User ID** of our mock user:

```
mock_user_id = 215
```

```
#Selecting the input user
inputUser = trX[mock_user_id-1].reshape(1, -1)
inputUser[0:5]
```

```
#Feeding in the user and reconstructing the input
hh0 = tf.nn.sigmoid(tf.matmul(v0, W) + hb)
vv1 = tf.nn.sigmoid(tf.matmul(hh0, tf.transpose(W)) + vb)
feed = sess.run(hh0, feed_dict={ v0: inputUser, W: prv_w, hb: prv_hb})
rec = sess.run(vv1, feed_dict={ hh0: feed, W: prv_w, vb: prv_vb})
print(rec)
```

We can then list the 20 most recommended movies for our mock user by sorting it by their scores given by our model.

```
scored_movies_df_mock = movies_df[movies_df['MovieID'].isin(user_rating_df.columns)]
scored_movies_df_mock = scored_movies_df_mock.assign(RecommendationScore = rec[0])
scored_movies_df_mock.sort_values(["RecommendationScore"], ascending=False).head(20)
```

So, how to recommend the movies that the user has not watched yet?

Now, we can find all the movies that our mock user has watched before:

```
movies_df_mock = ratings_df[ratings_df['UserID'] == mock_user_id]
movies_df_mock.head()
```

In the next cell, we merge all the movies that our mock users has watched with the predicted scores based on his historical data:

```
#Merging movies_df with ratings_df by MovieID
merged_df_mock = scored_movies_df_mock.merge(movies_df_mock, on='MovieID', how='outer')
```

lets sort it and take a look at the first 20 rows:

```
merged_df_mock.sort_values(["RecommendationScore"], ascending=False).head(20)
```

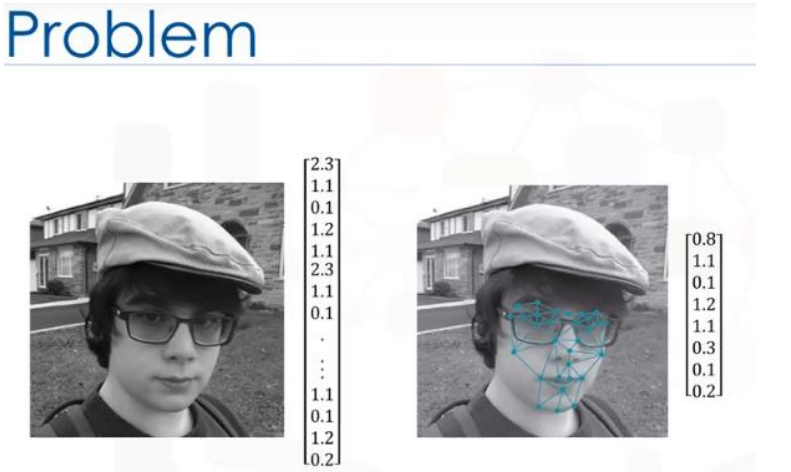
As you can see, there are some movies that user has not watched yet and has high score based on our model. So, we can recommend them to the user.

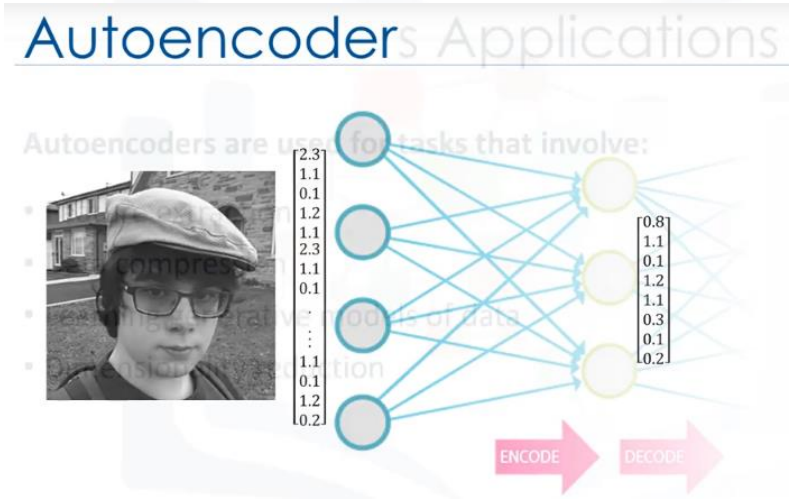
This is the end of the module. If you want, you can try to change the parameters in the code -- adding more units to the hidden layer, changing the loss functions or maybe something else to see if it changes anything. Does the model perform better? Does it take longer to compute?

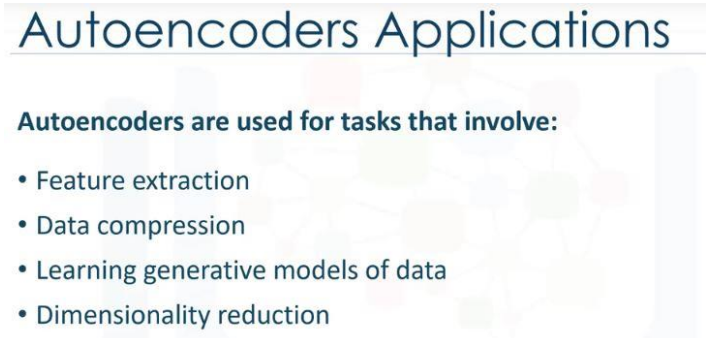
Thank you for reading this notebook. Hopefully, you now have a little more understanding of the RBM model, its applications and how it works with TensorFlow.

MODULE 5 – AUTENCODERS

INTRODUCTION





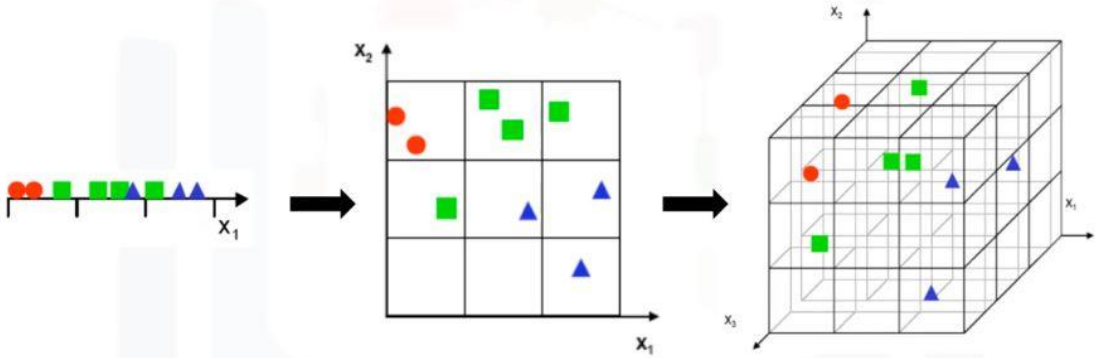


Curse of Dimensionality

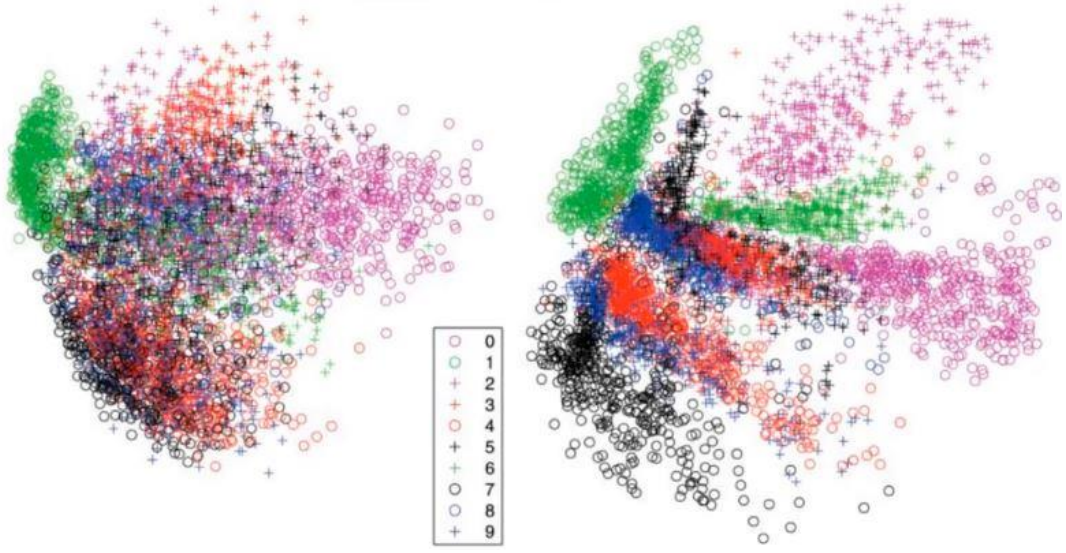
$$m^{-p/(2p+d)}$$

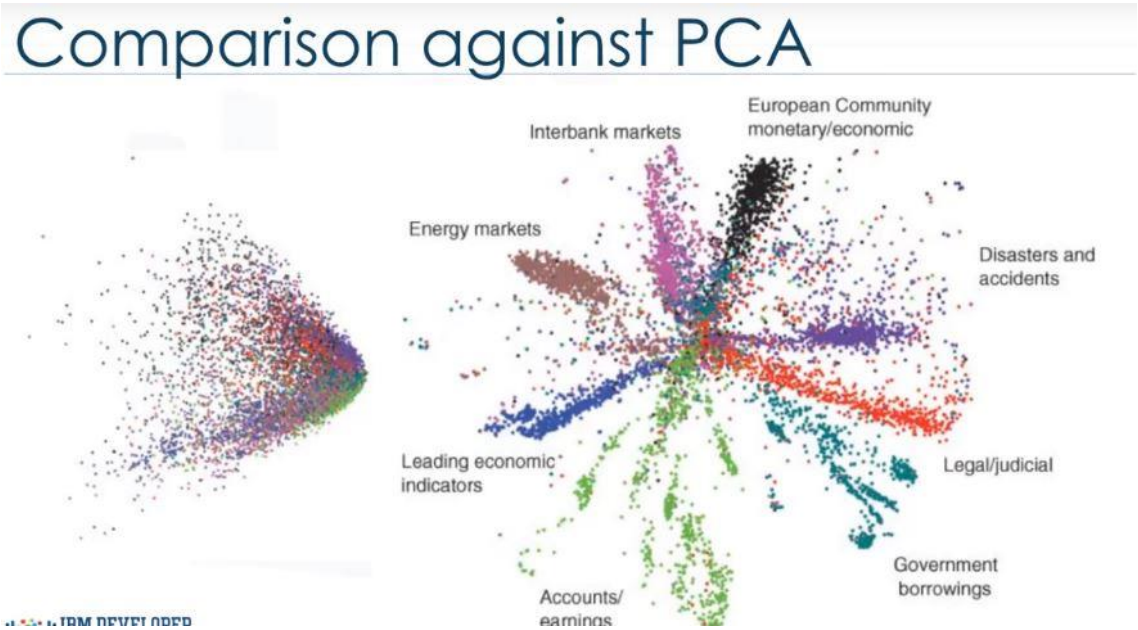
Being:
m: Number of data points
d: Dimensionality of the data
p: Parameter that depends on the model

Sparsity



Comparison against PCA

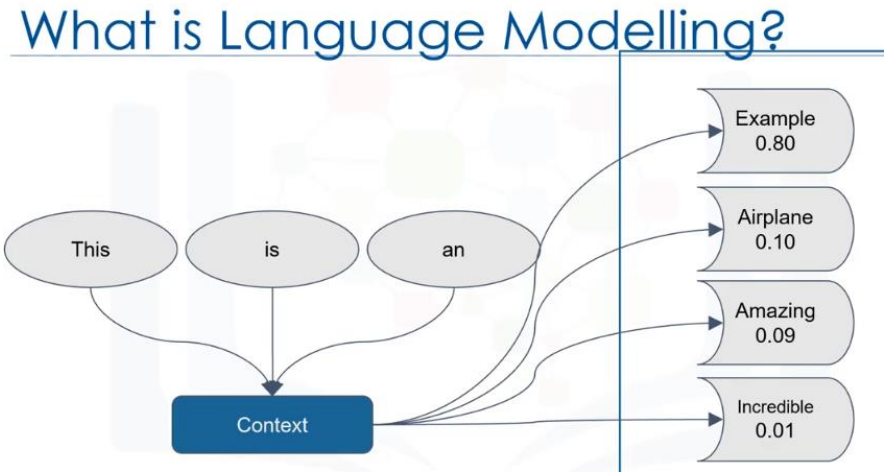




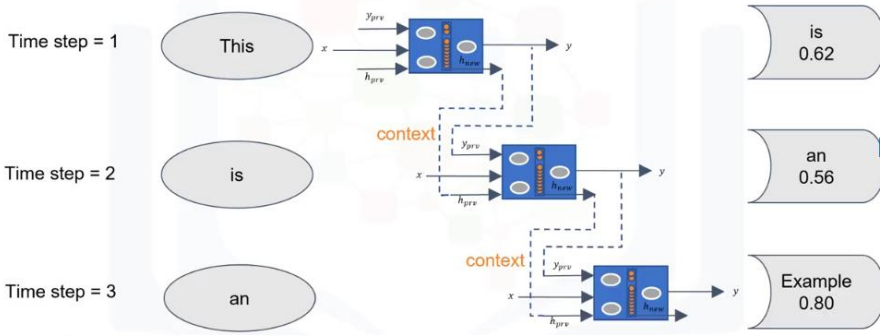
Hello, and welcome! In this video, we'll be covering the basic
concepts and the motivation behind autoencoders, a type of neural network used in unsupervised
machine learning.
Let's say that you want to extract the feeling or emotion of a person in a photograph.
The photograph that you see here is a small image that's just 256 by 256 pixels…
But, this means that there are over 65 thousand pixels in play in defining the dimension of the input! As we increase the dimensionality, the time to deal with data increases exponentially,
in order to train and fit the raw data into a neural network that can detect the emotion.
So, we need a way to extract the most important features of a face, and represent each image with those features which are of lower dimensions. An autoencoder works well for this type of problem.
An autoencoder is a type of unsupervised learning algorithm that will find patterns in a dataset by detecting key features. It is a type of neural net that analyzes all
of the images in your dataset and extracts some useful features automatically in such
a way that it can distinguish images using those features.
Generally speaking, autoencoders excel in tasks that involve feature learning or extraction, data compression, and learning generative models of data and dimensionality reduction.
Let me talk more about Dimension Reduction.
High-dimensional data is a significant problem for machine learning tasks.
In fact, data scientists commonly refer to it as the "CURSE OF DIMENSIONALITY".
Many common problems that we want to target have a large number of dimensions.
Even that 256 x 256 pixel image that we saw earlier, would have over 65 thousand dimensions; or in other words, one dimension for each pixel.
A high-resolution image from a smart phone would be even larger.
According to a study, the time to fit a model is, at best, the function you see here, which is a function of the number of points, dimensionality, and parameters.
So, as we increase the dimensionality, our time to fit our model will increase exponentially.
So, what happens when we increase or reduce the dimension of data?
Well, if we have a huge number of dimensions, our data will start to get sparse, which results

in an over-allocation of memory and slow training time.
We run into additional problems when we try to reduce the dimensions.
If we have a small number of dimensions, our data could overlap, resulting in a loss of data characteristics. You can see how that looks in one dimension
and three dimensions. Overlap and sparsity make it difficult to
determine the underlying patterns. However, with the proper number of dimensions, the patterns become much clearer.
It's important to know that an Autoencoder is not the only dimension reduction method in Machine Leaning. Principal Component Analysis (or PCA) has
been around for a long time, and is a classic algorithm for dimensionality reduction.
Take a look at the data separation outputs for the MNIST dataset of handwritten digits.
These charts show the image data after reducing their dimensions to 2 dimensions.
The left output is from PCA. The output on the right is from an autoencoder.
As you can see, it's easier to discern the data with the autoencoder's output.
Here is another comparison between PCA and an autoencoder, now applied to news stories.
The separability of the autoencoder is far better than the PCA, in this case.
Since separability is important for applying clustering algorithms, this difference in quality is significant. By now, you should understand that an autoencoder
can extract key image features, improve training times of other networks, and improve the separability
of reduced datasets when compared to other methods.
For these reasons, the autoencoder was a breakthrough in the unsupervised learning research field.
Thanks for watching this video.
End of transcript. Skip to the start.

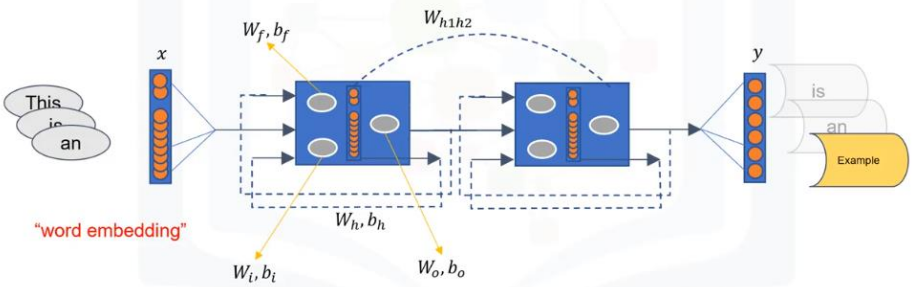
APPYING RECURRENT NETWORKS TO LANGUAGE MODELLING



Unfolded LSTM network



Training LSTM



Word Embedding

n-dimensional (e.g. 200)

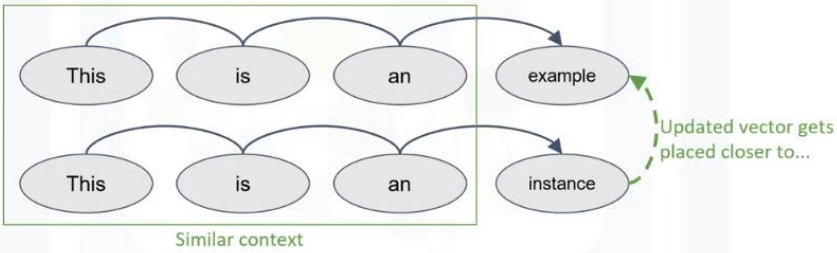
Vocab size
10000 words

here	0.571	0.384	0.619	0.603	0.685	...	0.618	0.046	0.025	0.572
book	0.717	0.173	0.934	0.393	0.763	...	0.529	0.321	0.964	0.969
is	0.123	0.46	0.799	0.088	0.983	...	0.225	0.298	0.846	0.755
man	0.613	0.253	0.712	0.113	0.777	...	0.88	0.828	0.425	0.793
this	0.486	0.836	0.844	0.693	0.305	...	0.844	0.682	0.315	0.525
boy	0.85	0.326	0.616	0.505	0.965	...	0.588	0.45	0.892	0.777
girl	0.828	0.895	0.078	0.053	0.645	...	0.47	0.331	0.518	0.074
work	0.862	0.496	0.686	0.33	0.603	...	0.32	0.245	0.038	0.833
example	0.225	0.006	0.578	0.465	0.792	...	0.283	0.856	0.243	0.118
...	..	..	..	...	..	..	..	...	..	..
watch	0.995	0.695	0.637	0.703	0.546	...	0.95	0.068	0.335	0.701
are	0.323	0.563	0.559	0.708	0.442	...	0.029	0.406	0.387	0.291
a	0.114	0.364	0.496	0.226	0.904	...	0.38	0.818	0.024	0.356
you	0.851	0.537	0.552	0.757	0.11	...	0.99	0.388	0.235	0.912
went	0.935	0.859	0.555	0.279	0.792	...	0.767	0.944	0.548	0.837
me	0.83	0.907	0.719	0.204	0.45	...	0.661	0.535	0.245	0.681

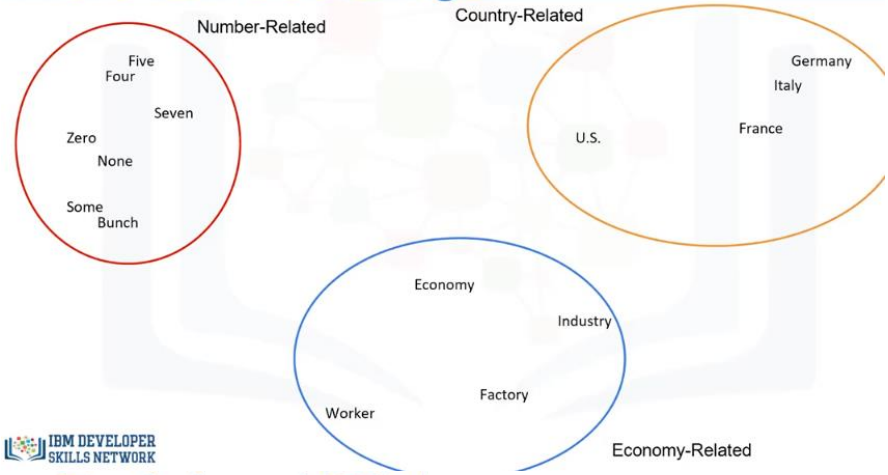
IBM DEVELOPER

```
embedding = tf.get_variable("embedding", [vocab_size, 200])
```

Word Embeddings

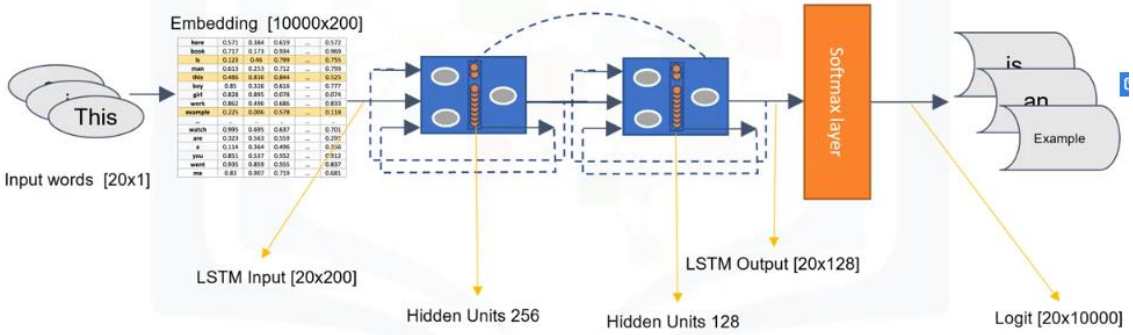


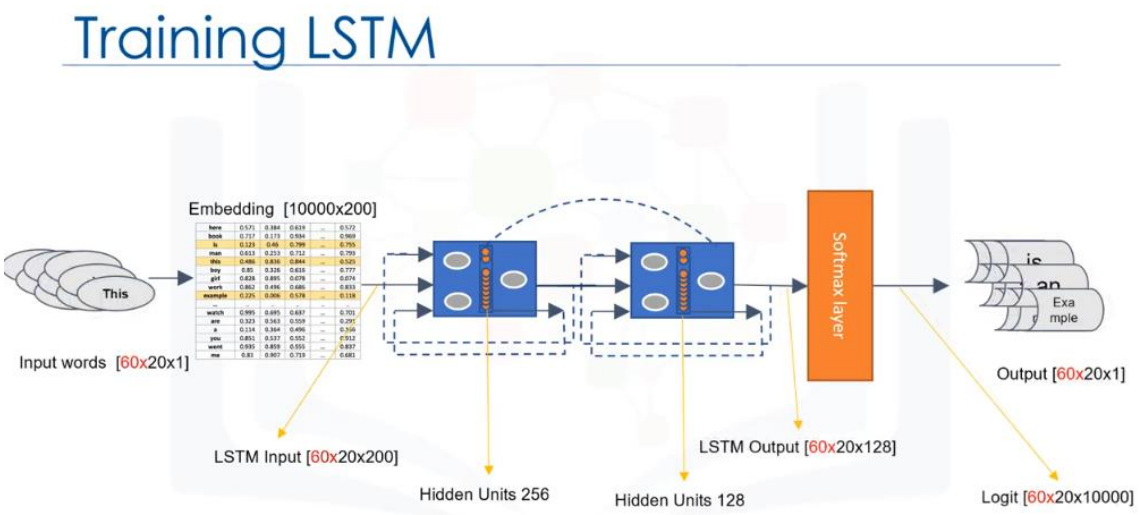
Word Embeddings

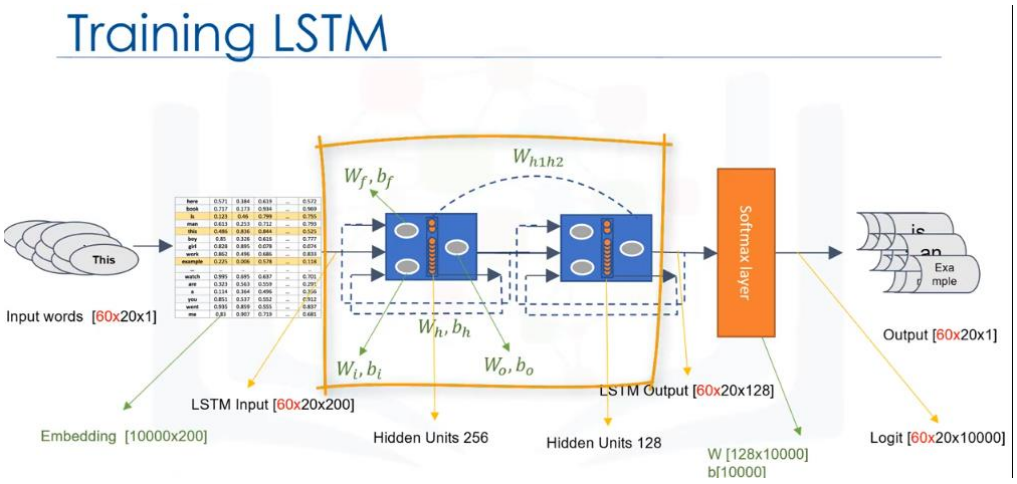


IBM DEVELOPER SKILLS NETWORK

Training LSTM







Start of transcript. Skip to the end.
Hello, and welcome. In this video, we'll be reviewing how to apply recurrent neural networks to language modelling.
Language modelling is a gateway into many exciting deep learning applications like speech recognition, machine translation, and image captioning.
At its simplest, language modelling is the process of assigning probabilities to sequences of words. So for example, a language model could analyze a sequence of words, and predict which word is most likely to follow.
So with the sequence, "This is an", which you see here, a language model might predict what the next word might be. Clearly, there are many options for what word could be used as the next one in the string. But a trained model might predict, with an 80 percent probability of being correct, that the word "example" is most likely to follow.
This boils down to a sequential data analysis problem.
The sequence of words forms the context, and the most recent word is the input data.
Using these two pieces of information, you need to output both a predicted word, and a new context that contains the input word.
Recurrent neural networks are a great fit for this type of problem.
At the first time step, a recurrent net can receive a word as input along the initial context. It generates an output.
The output word with the current sequence of words as the context will then be re-fed into

the network in the second time step. A new word would be predicted.
And, these steps are repeated until the sentence is complete.
Now, let's take closer look at an LSTM network for modeling the language.
In this network, we will use an RNN network with two stacked LSTM units.
For training such a network, we have to pass each word of the sentence to the network, and let the network generate an output. For example, after passing the words "this" and "is", if we pass the word "an" in the third time step, we expect the network to generate the word, "example," as output. But notice that we cannot easily pass a word to the network. We have to convert it into a vector of numbers somehow. We can use "word embedding" for this purpose.
Let's quickly examine what happens in word embedding.
An interesting way to process words is through a structure known as a Word Embedding.
A word embedding is an n-dimensional vector of real numbers for each word.
The vector is typically large, for example, 200 length.
You can see what that might look like with the word "example" here.
You think of word embedding as a type of encoding for text-to-numbers.
Now the question is, how do we find the proper values for these vectors?
In our RNN model, the vectors (also known as the matrix for the vocabulary) are initialized randomly for all the words that we are going to use for training.
Then, during the recurrent network's training, the vector values are updated based on the context into which the word is being inserted. So, words that are used in similar contexts end up with similar positions in the vector space.
This can be visualized by utilizing a dimensionality-reduction algorithm.
Take a look at the example shown here. After training the RNN, if we visualize the words based on their embedding vectors, the words are grouped together either because they're synonyms, or they're used in similar places within a sentence.
For example, the words "zero" and "none" are close semantically, so it's natural for them to be placed close together. And while "Italy" and "Germany" aren't synonyms, they can be interchanged in several sentences without distorting the grammar.
Ok, now let's look back at the RNN that we've been using.
Imagine that the input data is a batch with only one sequence of words.
Think of it as a batch that includes one sentence only - one that includes 20 words.
Assume that the vocabulary size of the words is 10,000 words, and the length of each embedding vector is 200. We have to look up those 20 words in the randomly initialized embedding matrix, and then feed them into the first LSTM unit.
Please notice that only one word in each time step is fed into the network, and one word would be the output. But, during 20 time steps, the output would be 20 words. In our network, we have 2 LSTM units, with arbitrary hidden sizes of 256 and 128. So, the output of the second LSTM unit would be a matrix of size 20-by-128. Now, we need a softmax layer to calculate the probability of the output words. It "squashes" the 128-dimensional vector of real values to a 10,000-dimensional vector, which is our vocabulary size.
This means that the output of the network at each time step is a probability vector of length 10,000. So, the output word is the one with maximum probability value in the vector. Now, we can compare the sequence of 20 output words with the ground truth words. And finally, calculate the discrepancy as a quantitative value, so called loss value, and back-propagate the errors into the network.
And of course, we will not train the model using only one sequence.
We will use a batch of sequences to train it and calculate the error.
So, instead of feeding one sequence, we can feed the network in many iterations - perhaps even a batch of 60 sentences, for example.

Now, the key question to be asked is: "What does the network learn when the error is propagated
back, in each iteration?" Well, as previously noted, the weights keep
updating based on the error of the network-in-training. First, the embedding matrix will be updated
in each iteration. Second, there are a bunch of weight matrices
related to the gates in the LSTM units which will be changed.
And finally, the weights related to the Softmax layer, which somehow plays the decoding role
for encoded words in the embedding layer. By now, you should have a good understanding
of how to use LSTM for language modeling.
Thanks for watching this video.
End of transcript. Skip to the start.

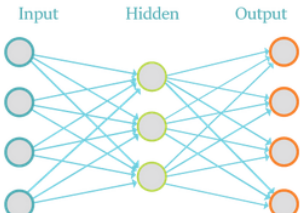
>> LAB

Lab Auencoders

Introduction

An autoencoder, also known as autoassociator or Diabolo networks, is an artificial neural network employed to recreate the given input. It takes a set of **unlabeled** inputs, encodes them and then tries to extract the most valuable information from them. They are used for feature extraction, learning generative models of data, dimensionality reduction and can be used for compression.
A 2006 paper named **Reducing the Dimensionality of Data with Neural Networks, done by G. E. Hinton and R. R. Salakhutdinov**, showed better results than years of refining other types of network, and was a breakthrough in the field of Neural Networks, a field that was "stagnant" for 10 years.

Now, autoencoders, based on Restricted Boltzmann Machines, are employed in some of the largest deep learning applications. They are the building blocks of Deep Belief Networks (DBN).



Feature Extraction and Dimensionality Reduction

An example given by Nikhil Buduma in KdNuggets (link) which gave an excellent explanation of the utility of this type of Neural Network.

Say that you want to extract what emotion the person in a photography is feeling. Using the following 256x256 pixel grayscale picture as an example:

But when use this picture we start running into a bottleneck! Because this image being 256x256 pixels in size correspond with an input vector of 65536 dimensions! If we used an image produced with conventional cellphone cameras, that generates images of 4000 x 3000 pixels, we would have 12 million dimensions to analyze.

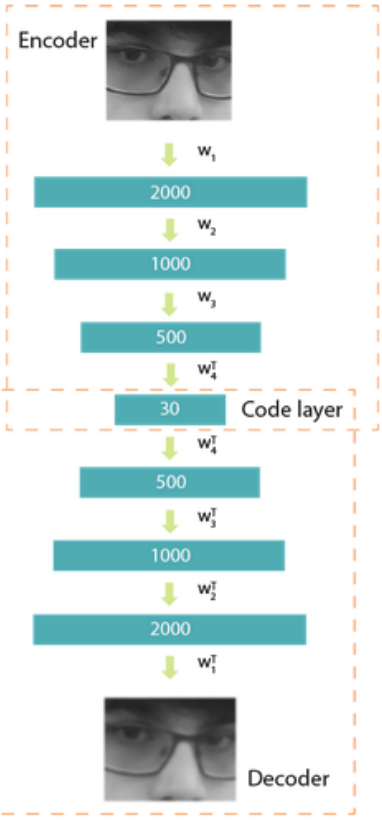
This bottleneck is further problematized as the difficulty of a machine learning problem is increased as more dimensions are involved. According to a 1982 study by C.J. Stone (link), the time to fit a model, is optimal if:

$$m^{-p/(2p+d)}$$

Where:
m: Number of data points
d: Dimensionality of the data
p: Parameter that depends on the model

As you can see, it increases exponentially! Returning to our example, we don't need to use all of the 65,536 dimensions to classify an emotion. A human identify emotions according to some specific facial expression, some **key features**, like the shape of the mouth and eyebrows.

Autoencoder Structure



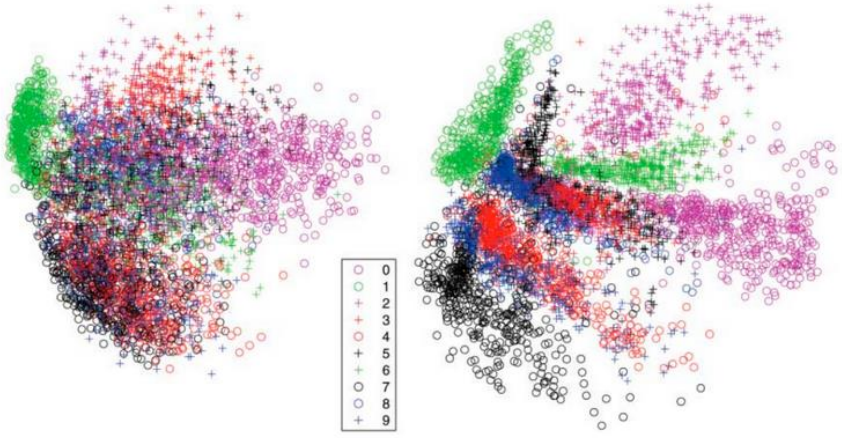
An autoencoder can be divided in two parts, the **encoder** and the **decoder**.

The encoder needs to compress the representation of an input. In this case we are going to reduce the dimension the face of our actor, from 2000 dimensions to only 30 dimensions, by running the data through layers of our encoder.

The decoder works like encoder network in reverse. It works to recreate the input, as closely as possible. This plays an important role during training, because it forces the autoencoder to select the most important features in the compressed representation.

Performance

After the training has been done, you can use the encoded data as a reliable dimensionality-reduced data, applying it to any problems where dimensionality reduction seems appropriate.



This image was extracted from the G. E. Hinton and R. R. Salakhutdinovcomparing's paper, on the two-dimensional reduction for 500 digits of the MNIST, with PCA on the left and autoencoder on the right. We can see that the autoencoder provided us with a better separation of data.

Training: Loss function

An autoencoder uses the Loss function to properly train the network. The Loss function will calculate the differences between our output and the expected results. After that, we can minimize this error with gradient descent. There are more than one type of Loss function, it depends on the type of data.

Binary Values:

$$l(f(x)) = -\sum_k (x_k log(\hat{x}_k) + (1 - x_k) \log(1 - \hat{x}_k))$$

For binary values, we can use an equation based on the sum of Bernoulli's cross-entropy.

x_k is one of our inputs and $\hat{x}_k$ is the respective output.

We use this function so that if x_k equals to one, we want to push $\hat{x}_k$ as close as possible to one. The same if x_k equals to zero.

If the value is one, we just need to calculate the first part of the formula, that is, $-x_k log(\hat{x}_k)$. Which, turns out to just calculate $-log(\hat{x}_k)$.

And if the value is zero, we need to calculate just the second part, $(1 - x_k) \log(1 - \hat{x}_k))$ - which turns out to be $log(1 - \hat{x}_k)$.

Real values:

$$l(f(x)) = -\frac{1}{2}\sum_k (\hat{x}_k - x_k)^2$$

As the above function would behave badly with inputs that are not 0 or 1, we can use the sum of squared differences for our Loss function. If you use this loss function, it's necessary that you use a linear activation function for the output layer.

As it was with the above example, x_k is one of our inputs and $\hat{x}_k$ is the respective output, and we want to make our output as similar as possible to our input.

Loss Gradient:

$$\nabla_{a(x,w)} l(f(x^{(i)})) = \hat{x}^{(i)} - x^{(i)}$$

We use the gradient descent to reach the local minimum of our function $l(f(x^{(i)}))$, taking steps towards the negative of the gradient of the function in the current point.

Our function about the gradient ($\nabla_{a(x,w)}$) of the loss of $l(f(x^{(i)}))$ in the preactivation of the output layer.

It's actually a simple formula, it is done by calculating the difference between our output $\hat{x}^{(i)}$ and our input $x^{(i)}$.

Then our network backpropagates our gradient $\nabla_{a(x,w)} l(f(x^{(i)}))$ through the network using **backpropagation**.

Code

For this part, we walk through a lot of Python 2.7.11 code. We are going to use the MNIST dataset for our example. The following code was created by Aymeric Damien. You can find some of his code in here. We made some modifications for us to import the datasets to Jupyter Notebooks.

Let's call our imports and make the MNIST data available to use.

```
#from __future__ import division, print_function, absolute_import

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
learning_rate = 0.01
training_epochs = 20
batch_size = 256
display_step = 1
examples_to_show = 10

# Network Parameters
n_hidden_1 = 256 # 1st layer num features
n_hidden_2 = 128 # 2nd layer num features
n_input = 784 # MNIST data input (img shape: 28*28)

# tf Graph input (only pictures)
X = tf.placeholder("float", [None, n_input])

weights = {
    'encoder_h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'encoder_h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'decoder_h1': tf.Variable(tf.random_normal([n_hidden_2, n_hidden_1])),
    'decoder_h2': tf.Variable(tf.random_normal([n_hidden_1, n_input])),
}
biases = {
    'encoder_b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'encoder_b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'decoder_b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'decoder_b2': tf.Variable(tf.random_normal([n_input])),
}
```

Now we need to create our encoder. For this, we are going to use sigmoidal functions. Sigmoidal functions delivers great results with this type of network. This is due to having a good derivative that is well-suited to backpropagation. We can create our encoder using the sigmoidal function like this:

```
# Building the encoder
def encoder(x):
    # Encoder first layer with sigmoid activation #1
    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['encoder_h1']), biases['encoder_b1']))
    # Encoder second layer with sigmoid activation #2
    layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['encoder_h2']), biases['encoder_b2']))
    return layer_2
```

And the decoder:

You can see that the layer_1 in the encoder is the layer_2 in the decoder and vice-versa.

```
# Building the decoder
def decoder(x):
    # Decoder first layer with sigmoid activation #1
    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['decoder_h1']),biases['decoder_b1']))
    # Decoder second layer with sigmoid activation #2
    layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['decoder_h2']), biases['decoder_b2']))
    return layer_2
```

Let's construct our model. In the variable `cost` we have the loss function and in the `optimizer` variable we have our gradient used for backpropagation.

```
# Construct model
encoder_op = encoder(X)
decoder_op = decoder(encoder_op)

# Reconstructed Images
y_pred = decoder_op
# Targets (Labels) are the input data.
y_true = X

# Define loss and optimizer, minimize the squared error
cost = tf.reduce_mean(tf.pow(y_true - y_pred, 2))
optimizer = tf.train.RMSPropOptimizer(learning_rate).minimize(cost)

# Initializing the variables
init = tf.global_variables_initializer()
```

For training we will run for 20 epochs.

```
# Launch the graph
# Using InteractiveSession (more convenient while using Notebooks)
sess = tf.InteractiveSession()
sess.run(init)

total_batch = int(mnist.train.num_examples / batch_size)
# Training cycle
for epoch in range(training_epochs):
    # Loop over all batches
    for i in range(total_batch):
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop) and cost op (to get loss value)
        _, c = sess.run([optimizer, cost], feed_dict={X: batch_xs})
    # Display logs per epoch step
    if epoch % display_step == 0:
        print("Epoch:", '%04d' % (epoch+1),
              "cost=", "{:.9f}".format(c))

print("Optimization Finished!")
```

```
Epoch: 0001 cost= 0.218729958
Epoch: 0002 cost= 0.181405053
Epoch: 0003 cost= 0.171680182
Epoch: 0004 cost= 0.157407790
Epoch: 0005 cost= 0.150844395
Epoch: 0006 cost= 0.151500776
Epoch: 0007 cost= 0.146300241
Epoch: 0008 cost= 0.139467284
Epoch: 0009 cost= 0.136919573
Epoch: 0010 cost= 0.138647676
Epoch: 0011 cost= 0.132326111
Epoch: 0012 cost= 0.130863369
Epoch: 0013 cost= 0.129828766
Epoch: 0014 cost= 0.130005032
Epoch: 0015 cost= 0.129694894
Epoch: 0016 cost= 0.124201685
Epoch: 0017 cost= 0.124395795
Epoch: 0018 cost= 0.123281069
Epoch: 0019 cost= 0.124811962
Epoch: 0020 cost= 0.122014269
Optimization Finished!
```

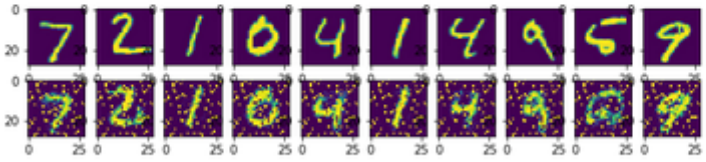
Now, let's apply encoder and decoder for our tests.

Now, let's apply encoder and decoder for our tests.

```
# Applying encode and decode over test set
encode_decode = sess.run(
    y_pred, feed_dict={X: mnist.test.images[:examples_to_show]})
```

Let's simply visualize our graphs!

```
# Compare original images with their reconstructions
f, a = plt.subplots(2, 10, figsize=(10, 2))
for i in range(examples_to_show):
    a[0][i].imshow(np.reshape(mnist.test.images[i], (28, 28)))
    a[1][i].imshow(np.reshape(encode_decode[i], (28, 28)))
```



As you can see, the reconstructions were successful. It can be seen that some noise were added to the image.

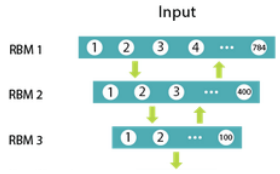
Lab DBN MNIST

Deep Belief Network

One problem with traditional multilayer perceptrons/artificial neural networks is that backpropagation can often lead to "local minima". This is when your "error surface" contains multiple grooves and you fall into a groove that is not lowest possible groove as you perform gradient descent.

Deep belief networks solve this problem by using an extra step called **pre-training**. Pre-training is done before backpropagation and can lead to an error rate not far from optimal. This puts us in the "neighborhood" of the final solution. Then we use backpropagation to slowly reduce the error rate from there.

DBNs can be divided in two major parts. The first one are multiple layers of Restricted Boltzmann Machines (RBMs) to pre-train our network. The second one is a feed-forward backpropagation network, that will further refine the results from the RBM stack.



Let's begin by importing the necessary libraries and utilities functions to implement a Deep Belief Network.

```
#urllib is used to download the utils file from deeplearning.net

import urllib.request
with urllib.request.urlopen("http://deeplearning.net/tutorial/code/utils.py") as url:
    response = url.read()
target = open("utils.py", "w")
target.write(response.decode("utf-8"))
target.close()
```

```
#Import the math function for calculations
import math
#Tensorflow library. Used to implement machine learning models
import tensorflow as tf
#Numpy contains helpful functions for efficient mathematical calculations
import numpy as np
#Image library for image manipulation
from PIL import Image
#import Image
#Utils file
from utils import tile_raster_images
```

Constructing the Layers of RBMs ¶

First of all, let's detail Restricted Boltzmann Machines.

What are Restricted Boltzmann Machines?

RBMs are shallow neural nets that learn to reconstruct data by themselves in an unsupervised fashion.

How it works?

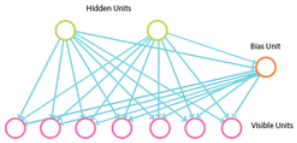
Simply, RBM takes the inputs and translates them to a set of numbers that represents them. Then, these numbers can be translated back to reconstruct the inputs. Through several forward and backward passes, the RBM will be trained, and a trained RBM can reveal which features are the most important ones when detecting patterns.

Why are RBMs important?

It can automatically extract **meaningful** features from a given input.

What's the RBM's structure?

It only possesses two layers: A visible input layer, and a hidden layer where the features are learned.



To implement DBNs in TensorFlow, we will implement a class for the Restricted Boltzmann Machines (RBM). The class below implements an intuitive way of creating and using RBM's.

```
#Class that defines the behavior of the RBM
class RBM(object):

    def __init__(self, input_size, output_size):
        #Defining the hyperparameters
        self._input_size = input_size #Size of input
        self._output_size = output_size #Size of output
        self.epochs = 5 #Amount of training iterations
        self.learning_rate = 1.0 #The step used in gradient descent
        self.batchsize = 100 #The size of how much data will be used for training per sub iteration

        #Initializing weights and biases as matrices full of zeroes
        self.w = np.zeros([input_size, output_size], np.float32) #Creates and initializes the weights with 0
        self.hb = np.zeros([output_size], np.float32) #Creates and initializes the hidden biases with 0
        self.vb = np.zeros([input_size], np.float32) #Creates and initializes the visible biases with 0


    #Fits the result from the weighted visible layer plus the bias into a sigmoid curve
    def prob_h_given_v(self, visible, w, hb):
        #Sigmoid
        return tf.nn.sigmoid(tf.matmul(visible, w) + hb)

    #Fits the result from the weighted hidden layer plus the bias into a sigmoid curve
    def prob_v_given_h(self, hidden, w, vb):
        return tf.nn.sigmoid(tf.matmul(hidden, tf.transpose(w)) + vb)

    #Generate the sample probability
    def sample_prob(self, probs):
        return tf.nn.relu(tf.sign(probs - tf.random_uniform(tf.shape(probs))))
```

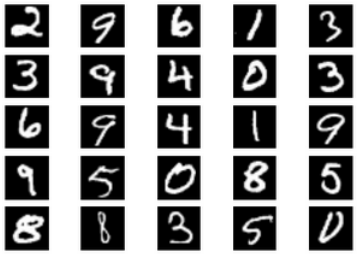
```
#Training method for the model
def train(self, X):
    #Create the placeholders for our parameters
    _w = tf.placeholder("float", [self._input_size, self._output_size])
    _hb = tf.placeholder("float", [self._output_size])
    _vb = tf.placeholder("float", [self._input_size])

    prv_w = np.zeros([self._input_size, self._output_size], np.float32) #Creates and initializes the weights with 0
    prv_hb = np.zeros([self._output_size], np.float32) #Creates and initializes the hidden biases with 0
    prv_vb = np.zeros([self._input_size], np.float32) #Creates and initializes the visible biases with 0


    cur_w = np.zeros([self._input_size, self._output_size], np.float32)
    cur_hb = np.zeros([self._output_size], np.float32)
    cur_vb = np.zeros([self._input_size], np.float32)
    v0 = tf.placeholder("float", [None, self._input_size])

    #Initialize with sample probabilities
    h0 = self.sample_prob(self.prob_h_given_v(v0, _w, _hb))
    v1 = self.sample_prob(self.prob_v_given_h(h0, _w, _vb))
    h1 = self.prob_h_given_v(v1, _w, _hb)

    #Create the Gradients
    positive_grad = tf.matmul(tf.transpose(v0), h0)
    negative_grad = tf.matmul(tf.transpose(v1), h1)

    #Update learning rates for the layers
    update_w = _w + self.learning_rate *(positive_grad - negative_grad) / tf.to_float(tf.shape(v0)[0])
    update_vb = _vb +  self.learning_rate * tf.reduce_mean(v0 - v1, 0)
    update_hb = _hb +  self.learning_rate * tf.reduce_mean(h0 - h1, 0)

    #Find the error rate
    err = tf.reduce_mean(tf.square(v0 - v1))
```

```
        #Training Loop
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            #For each epoch
            for epoch in range(self.epochs):
                #For each step/batch
                for start, end in zip(range(0, len(X), self.batchsize),range(self.batchsize,len(X), self.batchsize)):
                    batch = X[start:end]
                    #Update the rates
                    cur_W = sess.run(update_W, feed_dict={v0: batch, _W: prv_W, _hb: prv_hb, _vb: prv_vb})
                    cur_hb = sess.run(update_hb, feed_dict={v0: batch, _W: prv_W, _hb: prv_hb, _vb: prv_vb})
                    cur_vb = sess.run(update_vb, feed_dict={v0: batch, _W: prv_W, _hb: prv_hb, _vb: prv_vb})
                    prv_W = cur_W
                    prv_hb = cur_hb
                    prv_vb = cur_vb
                error = sess.run(err, feed_dict={v0: X, _W: cur_W, _vb: cur_vb, _hb: cur_hb})
                print ('Epoch: %d' % epoch,'reconstruction error: %f' % error)
            self.W = prv_W
            self.hb = prv_hb
            self.vb = prv_vb

    #Create expected output for our DBN
    def rbm_outpt(self, X):
        input_X = tf.constant(X)
        _W = tf.constant(self.W)
        _hb = tf.constant(self.hb)
        out = tf.nn.sigmoid(tf.matmul(input_X, _W) + _hb)
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            return sess.run(out)
```

The MNIST Dataset

We will be using the MNIST dataset, which is a commonly used dataset used for model benchmarking comprised of handwritten digits. We will import the images using "One Hot Encoding" to encode the handwritten images into values varying from 0 to 1.

Random Sampling of MNIST



```
#Getting the MNIST data provided by Tensorflow
from tensorflow.examples.tutorials.mnist import input_data

#Loading in the mnist data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Creating the Deep Belief Network

With the RBM class created and MNIST Datasets loaded in, we can start creating the DBN. For our example, we are going to use a 3 RBMs, one with 500 hidden units, the second one with 200 and the last one with 50. We are generating a **deep hierarchical representation of the training data**. The cell below accomplishes this:

```
RBM_hidden_sizes = [500, 200 , 50 ] #create 2 layers of RBM with size 400 and 100

#Since we are training, set input as training data
inpX = trX

#Create list to hold our RBMs
rbm_list = []

#Size of inputs is the number of inputs in the training set
input_size = inpX.shape[1]

#For each RBM we want to generate
for i, size in enumerate(RBM_hidden_sizes):
    print ('RBM: ',i,' ',input_size,'->', size)
    rbm_list.append(RBM(input_size, size))
    input_size = size
```

```
RBM:  0  784 -> 500
RBM:  1  500 -> 200
RBM:  2  200 -> 50
```

RBM Train

We will now begin the pre-training step and train each of the RBMs in our stack by individually calling the train function, getting the current RBMs output and using it as the next RBM's input.

```
#For each RBM in our list
for rbm in rbm_list:
    print ('New RBM:')
    #Train a new one
    rbm.train(inpX)
    #Return the output layer
    inpX = rbm.rbm_outpt(inpX)
```

```
New RBM:
Epoch: 0 reconstruction error: 0.060874
Epoch: 1 reconstruction error: 0.052762
Epoch: 2 reconstruction error: 0.045600
```

Now we can convert the learned representation of input data into a supervised prediction, e.g. a linear classifier. Specifically, we use the output of the last hidden layer of the DBN to classify digits using a shallow Neural Network.

Neural Network ¶

The class below implements the Neural Network that makes use of the pre-trained RBMs from above.

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


class NN(object):

    def __init__(self, sizes, X, Y):
        #Initialize hyperparameters
        self._sizes = sizes
        self._X = X
        self._Y = Y
        self.w_list = []
        self.b_list = []
        self._learning_rate =  1.0
        self._momentum = 0.0
        self._epoches = 10
        self._batchsize = 100
        input_size = X.shape[1]

        #initialization loop
        for size in self._sizes + [Y.shape[1]]:
            #Define upper limit for the uniform distribution range
            max_range = 4 * math.sqrt(6. / (input_size + size))

            #Initialize weights through a random uniform distribution
            self.w_list.append(
                np.random.uniform( -max_range, max_range, [input_size, size]).astype(np.float32))

            #Initialize bias as zeroes
            self.b_list.append(np.zeros([size], np.float32))
            input_size = size
```

```
#Load data from rbm
def load_from_rbms(self, dbn_sizes,rbm_list):
    #Check if expected sizes are correct
    assert len(dbn_sizes) == len(self._sizes)

    for i in range(len(self._sizes)):
        #Check if for each RBN the expected sizes are correct
        assert dbn_sizes[i] == self._sizes[i]

    #If everything is correct, bring over the weights and biases
    for i in range(len(self._sizes)):
        self.w_list[i] = rbm_list[i].w
        self.b_list[i] = rbm_list[i].hb

#Training method
def train(self):
    #Create placeholders for input, weights, biases, output
    _a = [None] * (len(self._sizes) + 2)
    _w = [None] * (len(self._sizes) + 1)
    _b = [None] * (len(self._sizes) + 1)
    _a[0] = tf.placeholder("float", [None, self._X.shape[1]])
    y = tf.placeholder("float", [None, self._Y.shape[1]])

    #Define variables and activation functoin
    for i in range(len(self._sizes) + 1):
        _w[i] = tf.Variable(self.w_list[i])
        _b[i] = tf.Variable(self.b_list[i])
    for i in range(1, len(self._sizes) + 2):
        _a[i] = tf.nn.sigmoid(tf.matmul(_a[i - 1], _w[i - 1]) + _b[i - 1])

    #Define the cost function
    cost = tf.reduce_mean(tf.square(_a[-1] - y))

    #Define the training operation (Momentum Optimizer minimizing the Cost function)
    train_op = tf.train.MomentumOptimizer(self._learning_rate, self._momentum).minimize(cost)

    #Prediction operation
    predict_op = tf.argmax(_a[-1], 1)

    #Training Loop
    with tf.Session() as sess:
        #Initialize Variables
        sess.run(tf.global_variables_initializer())
```

```
            #For each epoch
            for i in range(self._epoches):

                #For each step
                for start, end in zip(range(0, len(self._X), self._batchsize), range(self._batchsize, len(self._X), self._batchsize)):

                    #Run the training operation on the input data
                    sess.run(train_op, feed_dict={ _a[0]: self._X[start:end], y: self._Y[start:end]})

                for j in range(len(self._sizes) + 1):
                    #Retrieve weights and biases
                    self.w_list[j] = sess.run(_w[j])
                    self.b_list[j] = sess.run(_b[j])

                print ("Accuracy rating for epoch " + str(i) + ": " + str(np.mean(np.argmax(self._Y, axis=1) == sess.run(predict_op, feed_dict={_a[0]: self._X, y: self._Y}))))
```

Now let's execute our code:

```
nNet = NN(RBM_hidden_sizes, trX, trY)
nNet.load_from_rbms(RBM_hidden_sizes,rbm_list)
nNet.train()
```