Lecture 10

Neural Networks & Tensor Flow

Zoran B. Djordjević

References

- Material in these slides is to a good measure based on:
  - *Lectures on Machine Learning*

    by Andrew Ng of Stanford University and Coursera.
  - Hands-on Machine Learning with Scikit-Learn & TensorFlow

    by Aurelie Geron, O'Relly 2017
  - TensorFlow tutorials and API documentation, htts://www.tensorflow.org
  - Intro to TensorFlow Workshop, Singapore, 2016, https://goo.gl/AwmsrV

Introduction

- Development of Neural Networks date back to the early 1940s. It experienced an upsurge in popularity in the late 1980s. This was a result of the discovery of new techniques and developments and general advances in computer hardware technology.
- Some NNs are models of biological neural networks and some are not, but historically, much of the inspiration for the field of NNs came from the desire to produce artificial systems capable of sophisticated, perhaps intelligent, computations similar to those that the human brain routinely performs. A side goal has always been to enhance our understanding of the human brain.
- Most NNs have some sort of training rule. In other words, NNs learn from examples (as children learn to recognize dogs from examples of dogs) and exhibit some capability for generalization beyond the training data.
- Neural computing is not a competitor to conventional computing. Neural computing is complementary technology. The most successful neural solutions have been those which operate in conjunction with existing, traditional techniques.

Introduction

- Neural networks have emerged as one of key machine learning techniques.
- Neural networks were quite popular some 20 years ago, then their popularity waned. In recent year advancement in computer hardware and software, made NN much more efficient and able to tackle various problems that seemed to be more easily analyzed with other ML techniques.

Neural Network Techniques vs. Regular Computers

- Computers have to be explicitly programmed
  - Analyze the problem to be solved.
  - Write the code in a programming language.
- Neural Networks learn from examples
  - No explicit description of the problem is needed.
  - "No need for a programmer".
  - The neural computer adapts itself during a training period, based on examples of similar problems even without a desired solution to each problem.
  - After sufficient training the neural computer is able to relate the problem data to the solutions, inputs to outputs, and it is then able to offer a viable solution to a brand new problem.
  - Able to generalize or to handle incomplete data.

Digital Computers vs Neural Networks

Digital Computers

- Deductive Reasoning. We apply known rules to input data to produce output.
- Computation is centralized, synchronous, and serial.
- Memory is packetted, literally stored, and location addressable.
- Not fault tolerant. One transistor goes and it no longer works.
- Exact.
- Static connectivity.
- Applicable if well defined rules with precise input data.

Neural Networks

- Inductive Reasoning. Given input and output data (training examples), we construct the rules.
- Computation is collective, asynchronous, and parallel.
- Memory is distributed, internalized, short term and content addressable.
- Fault tolerant, redundancy, and sharing of responsibilities.
- Inexact.
- Dynamic connectivity.
- Applicable if rules are unknown or complicated, or if data are noisy or partial.

Applications off NNs

- **classification**
  - in marketing: consumer spending pattern classification
  - In defence: radar and sonar image classification
  - In agriculture & fishing: fruit and catch grading
  - In medicine: ultrasound and electrocardiogram image classification, EEGs, medical diagnosis
- **recognition and identification**
  - In general computing and telecommunications: speech, vision and handwriting recognition
  - In finance: signature verification and bank note verification
- **assessment**
  - In engineering: product inspection monitoring and control
  - In defence: target tracking
  - In security: motion detection, surveillance image analysis and fingerprint matching
- **forecasting and prediction**
  - In finance: foreign exchange rate and stock market forecasting
  - In agriculture: crop yield forecasting
  - In marketing: sales forecasting
  - In meteorology: weather prediction

What can you do with an NN and what not?

- In principle, NNs can compute any computable function, i.e., they can do everything a normal digital computer can do. Almost any mapping between vector spaces can be approximated to arbitrary precision by feedforward NNs
- In practice, NNs are especially useful for classification and function approximation problems usually when rules such as those that might be used in an expert system cannot easily be applied.
- NNs are, at least today, difficult to apply successfully to problems that concern manipulation of symbols and memory. And there are no methods for training NNs that can magically create information that is not contained in the training data.

Who is concerned with NNs?

- Computer scientists want to find out about the properties of non-symbolic information processing with neural nets and about learning systems in general.
- Statisticians use neural nets as flexible, nonlinear regression and classification models.
- Engineers of many kinds exploit the capabilities of neural networks in many areas, such as signal processing and automatic control.
- Cognitive scientists view neural networks as a possible apparatus to describe models of thinking and consciousness (High-level brain function).
- Neuro-physiologists use neural networks to describe and explore medium-level brain function (e.g. memory, sensory system, motorics).
- Physicists use neural networks to model phenomena in statistical mechanics and for a lot of other tasks.
- Biologists use Neural Networks to interpret nucleotide sequences.
- Philosophers and some other people may also be interested in Neural Networks for various reasons
- Musicians
- Graphic Artists
- Chess Players

Neurons and the brain

Hypothesis is that the brain has a single learning algorithm.
Evidence for hypothesis:

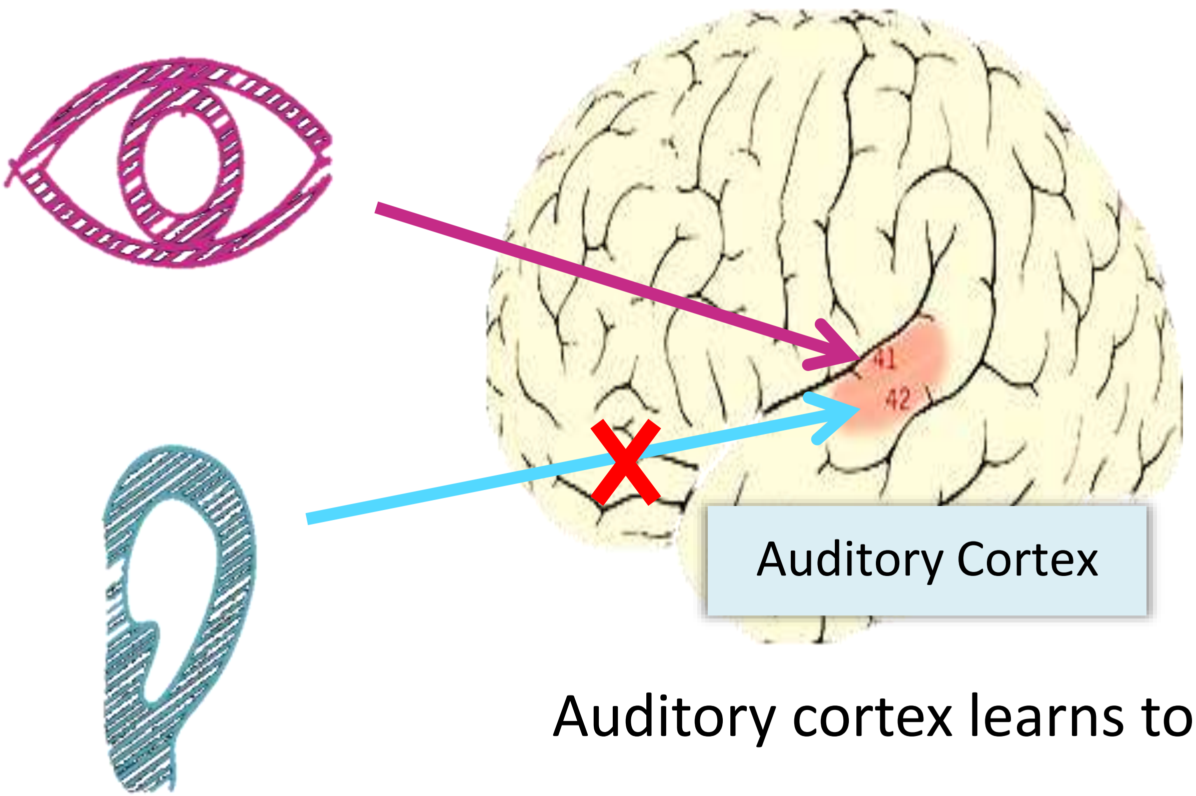
- Auditory cortex --> takes sound signals
  If you cut the wiring from the ear to the auditory cortex
  Re-route optic nerve to the auditory cortex
  - Auditory cortex learns to see
- Somatosensory context (touch processing)
  If you rewrite optic nerve to somatosensory cortex then it learns to see
  With different tissue learning to see, maybe they all learn in the same way. Brain learns by itself how to learn.

Other examples

- Seeing with your tongue
  Grayscale camera on head. Run wires to array of electrodes on tongue
  Pulses onto tongue represent image signal. People could see with their tongue
- Human echolocation
  Blind people being trained in schools to interpret sound and echo
  Lets them move around
- Haptic belt direction sense
  Belt which buzzes towards north
  Gives you a sense of direction

Brain can process and learn from data from any source

The "one learning algorithm" hypothesis

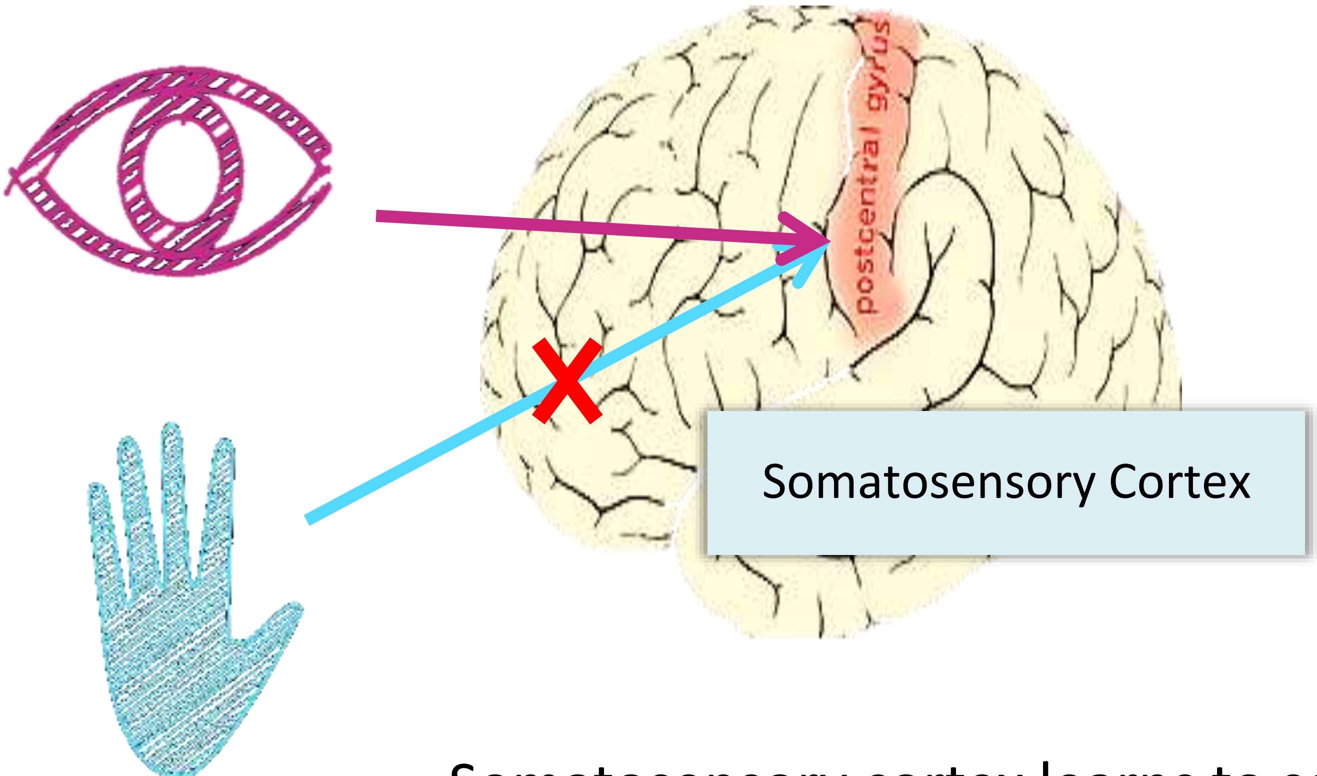


Auditory cortex learns to see

Professor Andrew Ng, originally posted on the ml-class.org

[Roe et al., 1992]

The "one learning algorithm" hypothesis



Somatosensory cortex learns to see

[Professor Andrew Ng](), originally posted on the [ml-class.org]()

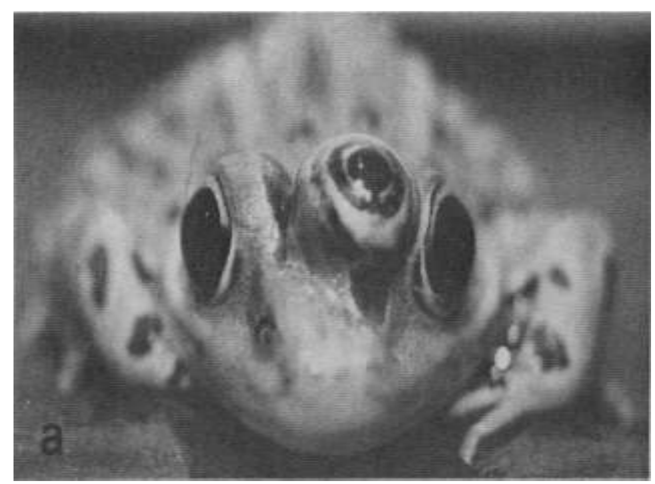
[Metin & Frost, 1989]

Sensor representations in the brain

Seeing with your tongue

Human echolocation (sonar)

Haptic belt: Direction sense

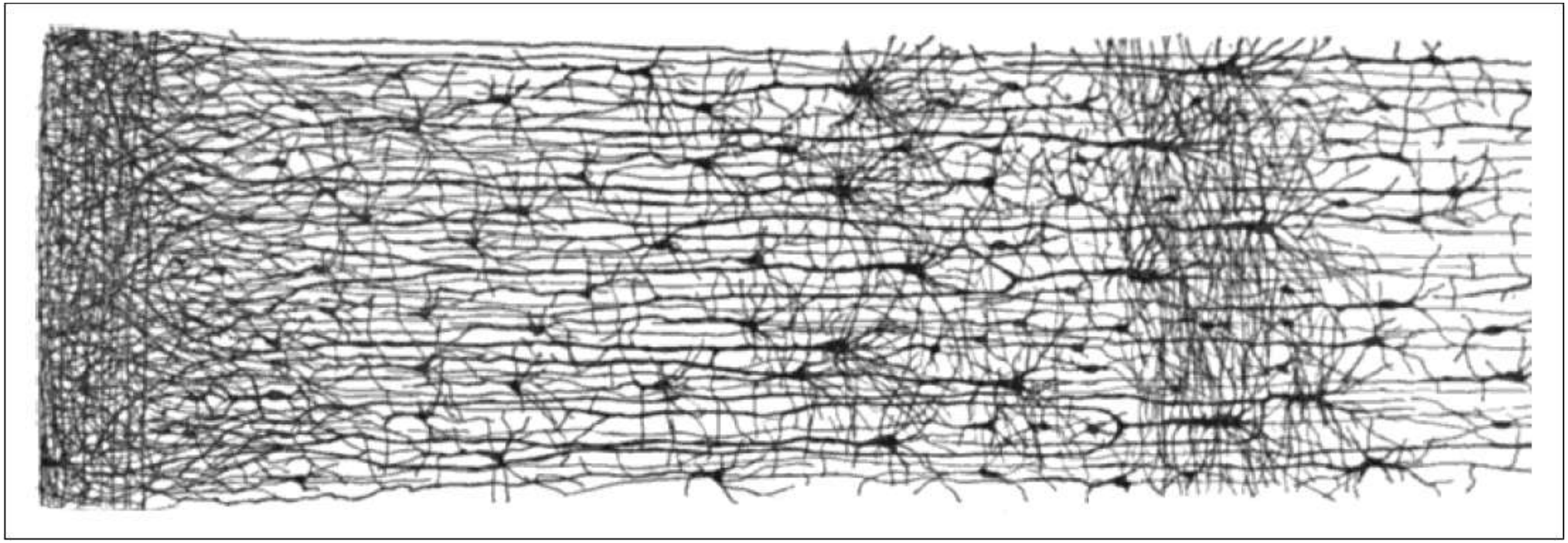


Implanting a 3rd eye

[BrainPort; Welsh & Blasch, 1997; Nagel et al., 2005; Constantine-Paton & Law, 2009]

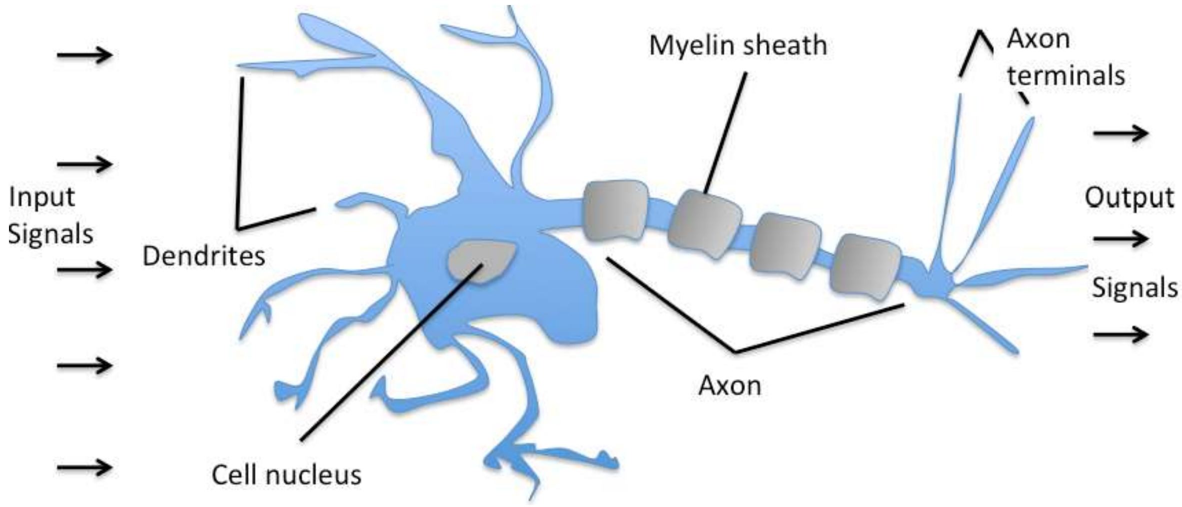
One Learning Algorithm Hypothesis

- What we have just described suggests that brain is not made of very different structures serving different purposes but is rather made of one and the same structure, the smallest unit being the neuron, which could adjust and perform different functions. Neurons appear to be organized in layers.
- Attempts to understand how brain work are much older. A Spanish scientist Santiago Ramón y Cajal made extensive work & discoveries in late 1800-s.
- Image from "Texture of the Nervous System of Man and the Vertebrates" by Santiago Ramón y Cajal (1899–1904). Read about y Cajal's discoveries at:
- https://www.nytimes.com/2017/02/17/science/santiago-ramon-y-cajal-beautiful-brain.html

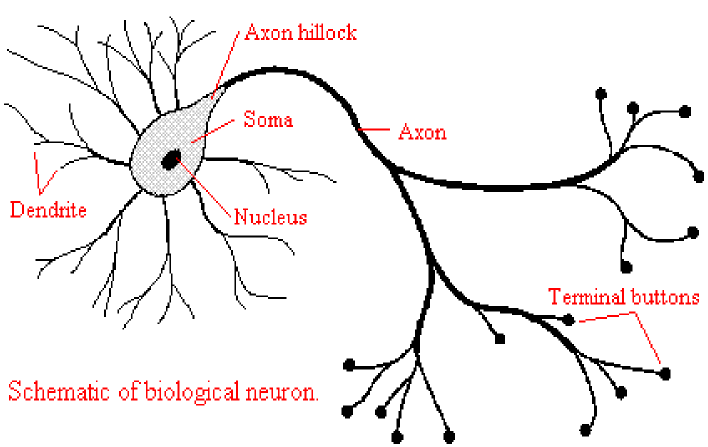


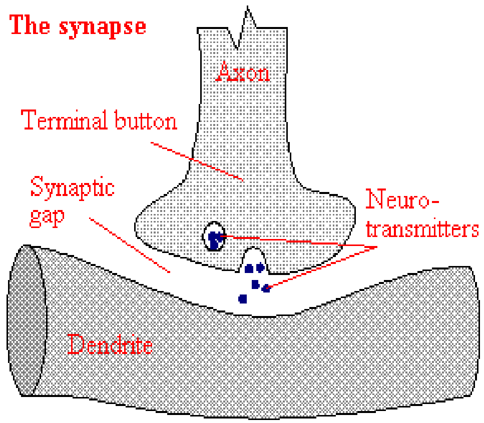
Artificial Neurons and the McCulloch-Pitts Model

- The initial idea of the perceptron dates back to the work of Warren McCulloch and Walter Pitts in 1943, who drew an analogy between biological neurons and simple logic gates with binary outputs..
- In more intuitive terms, neurons can be understood as the subunits of a neural network in a biological brain. Here, the signals of variable magnitudes arrive at the dendrites. Those input signals are then accumulated in the cell body of the neuron, and if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.



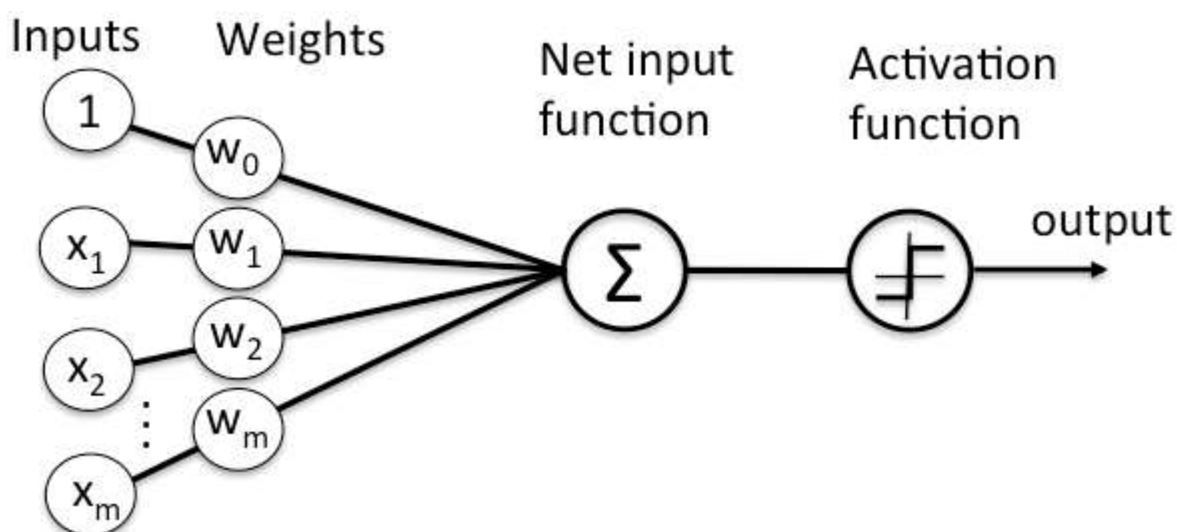
The Biological Neuron





- Our brain is a collection of about 10 billion interconnected neurons. Each neuron is a cell that uses biochemical reactions to receive, process and transmit information.
- Each terminal button is connected to other neurons across a small gap called a synapse.
- A neuron's dendritic tree is connected to a thousand neighbouring neurons. When one of those neurons fire, a positive or negative charge is received by one of the dendrites. The strengths of all the received charges are added together through the processes of spatial and temporal summation.

Perceptron Learning Rule

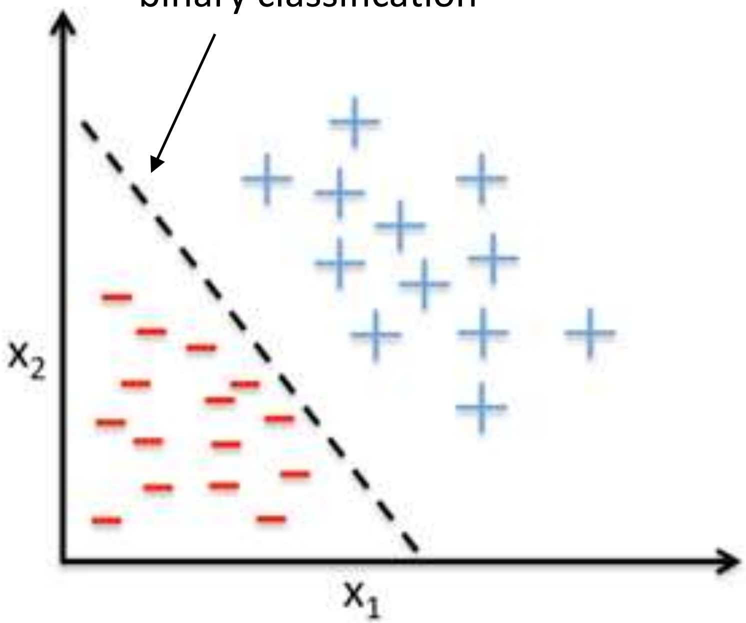
- Mathematicians started speculated about the mechanism used by neurons to make a decision. One of early mathematical models of neurons was Frank Rosenblatt's Perceptron with learning rules.
- The idea behind this "thresholded" perceptron was to mimic how a single neuron in the brain works: It either "fires" or not.
- A perceptron receives multiple input signals, and if the sum of the input signals exceed a certain threshold it either returns a signal or remains "silent" otherwise.
- What made this a "machine learning" algorithm was the idea of the perceptron learning rule:
  - The perceptron algorithm is about learning the weights for the input signals in order to draw linear decision boundary that allows us to discriminate between the two linearly separable classes +1 and -1.
- 

Rosenblatt's perceptron

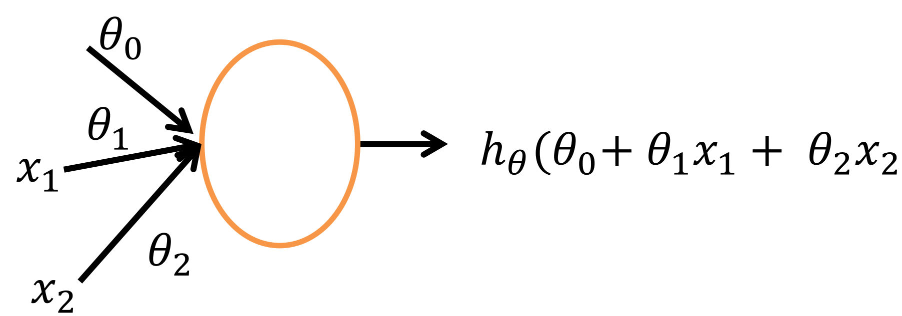
Perceptron as Classifier

- In context of pattern classification, such an algorithm could be useful to determine if a sample belongs to one class or the other.

Linear decision boundary for binary classification



- The perceptron belongs to the category of supervised learning algorithms, single-layer binary linear classifiers to be more specific.
- The task is to predict to which of two possible categories a certain data point belongs based on a set of input variables



We are basically looking for a single line and 3 parameters: θ_0, θ_1 *and* θ_2
Perceptron can perform linear classification.

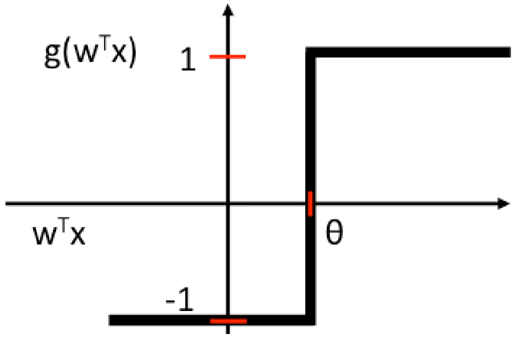
Unit Step Function

- If we are performing binary classification, we usually label the *positive* and *negative* class in our binary classification as "1" and "-1", respectively.
- We define an activation function $g(z)$ that takes a linear combination of the input values x and weights w as input ($z = w_1 x_1 + \cdots + wmxm$). If $g(z)$ is greater than a defined threshold θ we predict 1 and -1 otherwise.
- This activation function g is a simple "unit step function," which is sometimes also called "Heaviside step function."

$$g(z) = 1 \; if \; z \geq \; \theta, \qquad -1 \; if \; z < \; \theta$$

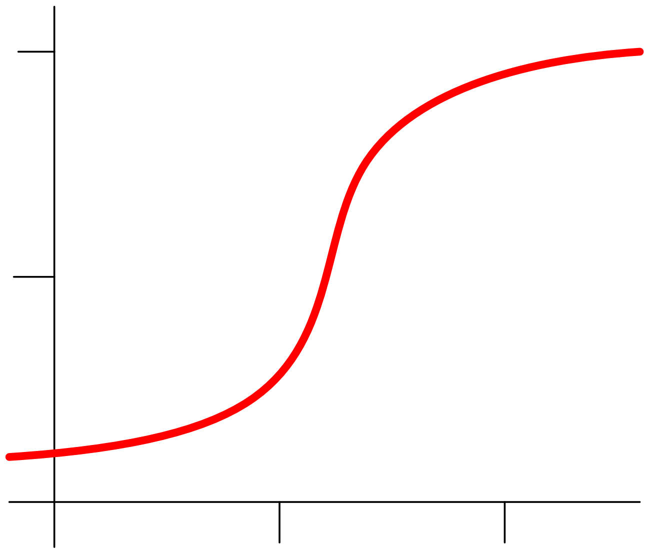
- We can express z as a dot product of vectors w and x, $z = \sum_{j=1}^{m} x_j w_j = \; w^T x$
- w is the vector of weights
- x is an m-dimensional sample

  from the training dataset.



Nonlinear Activation Functions

- Eventually, computer scientist learned that Sigmoid function is much more easier to work with and is now the most frequently used neuron activation function.



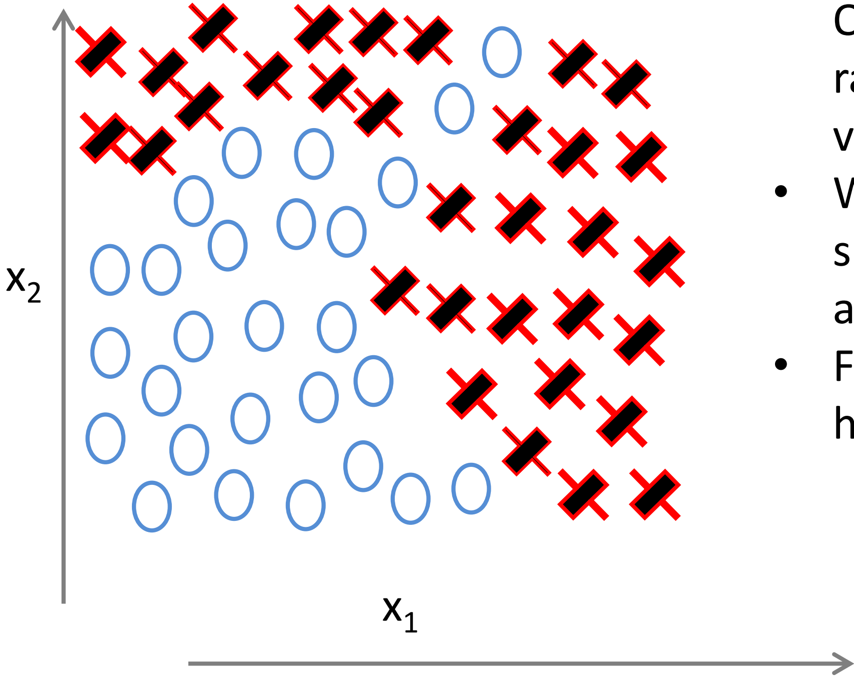
Sigmoid

$$y_{hid}(u) = \frac{1}{1+e^{-u}}$$

- A single neuron with either of activation function can quite well identify classes in a binary problem where the boundary between two classes is close to a straight line or multi-dimensional plane.

Need for
Non-linear Hypotheses

Non-linear Classification



- Not all problems are binary or simply linear. Quite often the boundary between classes is a rather complex function of independent variables: x_1 , $x_2, \ldots, x_n$.
- We know from mathematics that we can simulate fairly arbitrary function if we use an arbitrary polynomial of higher order.
- For example, a polynomial of the third order the hypothesis could look like this:

$$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 \\ + \theta_3 x_1 x_2 + \theta_4 x_1^2 x_2 \\ + \theta_5 x_1^3 x_2 + \theta_6 x_1 x_2^2 + \ldots)$$

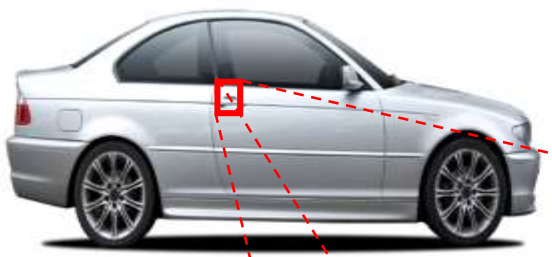
$x_1 =$ siz
$x_2 =$ # bedrooms
$x_3 =$ # floors
$x_4 =$ age
$\ldots$
x_{100}

- If our problem is complex and has a large number of variables, we have an issue. Polynomial of 2nd order of n-variables has ½ n(n-1) terms ~ n^2 features. Polynomial of 3rd order has approximately n^3 features. If n = 100, number of features and unknown θ_i will be of the order of 10,000 and 1,000,000 respectively.
- Fitting that many parameters using linear regression is very slow and error prone.

[Professor Andrew Ng](), originally posted on the [ml-class.org]()

Detecting a car in an image

You see this:



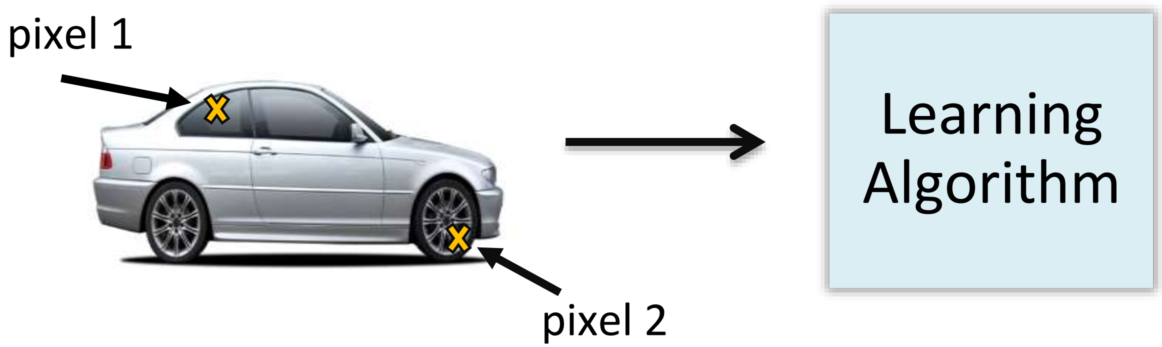
The number of variables n is the number of pixels in the image:
50x50= 2,500 independent variables
If image is gray. If RGB, then 7500.

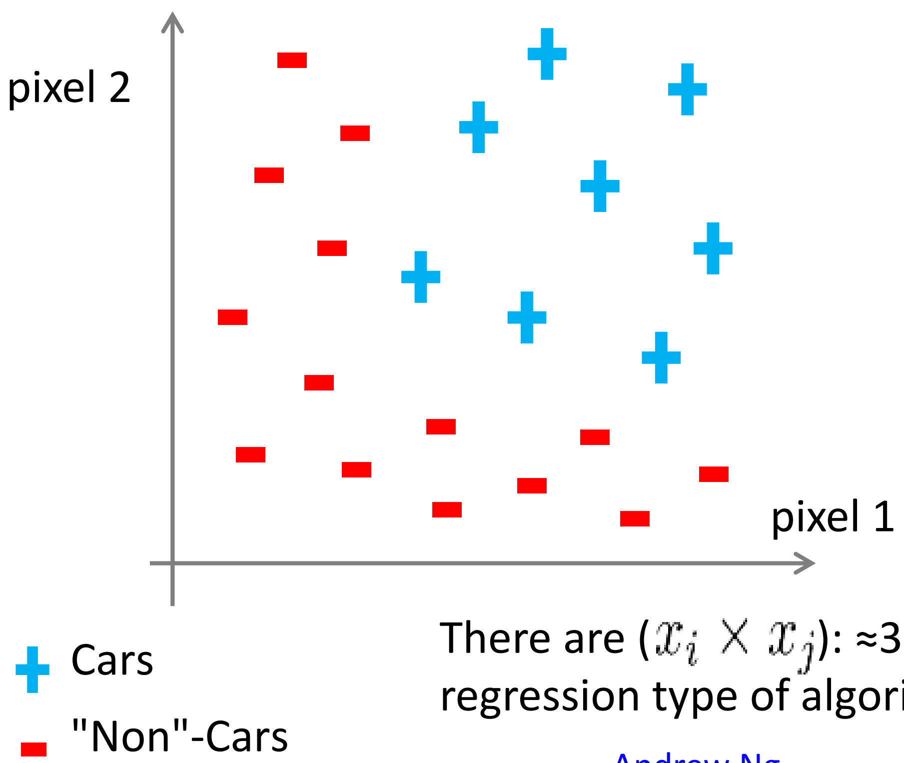
But the camera sees this:

194	210	201	212	199	213	215	195	178	158	182	209
180	189	190	221	209	205	191	167	147	115	129	163
114	126	140	188	176	165	152	140	170	106	78	88
87	103	115	154	143	142	149	153	173	101	57	57
102	112	106	131	122	138	152	147	128	84	58	66
94	95	79	104	105	124	129	113	107	87	69	67
68	71	69	98	89	92	98	95	89	88	76	67
41	56	68	99	63	45	60	82	58	76	75	65
20	43	69	75	56	41	51	73	55	70	63	44
50	50	57	69	75	75	73	74	53	68	59	37
72	59	53	66	84	92	84	74	57	72	63	42
67	61	58	65	75	78	76	73	59	75	69	50

Professor Andrew Ng, originally posted on the ml-class.org

Detecting a car with a quadratic feature list





50 x 50 pixel images→ 2500 pixels (7500 if RGB)

$n = 2500$

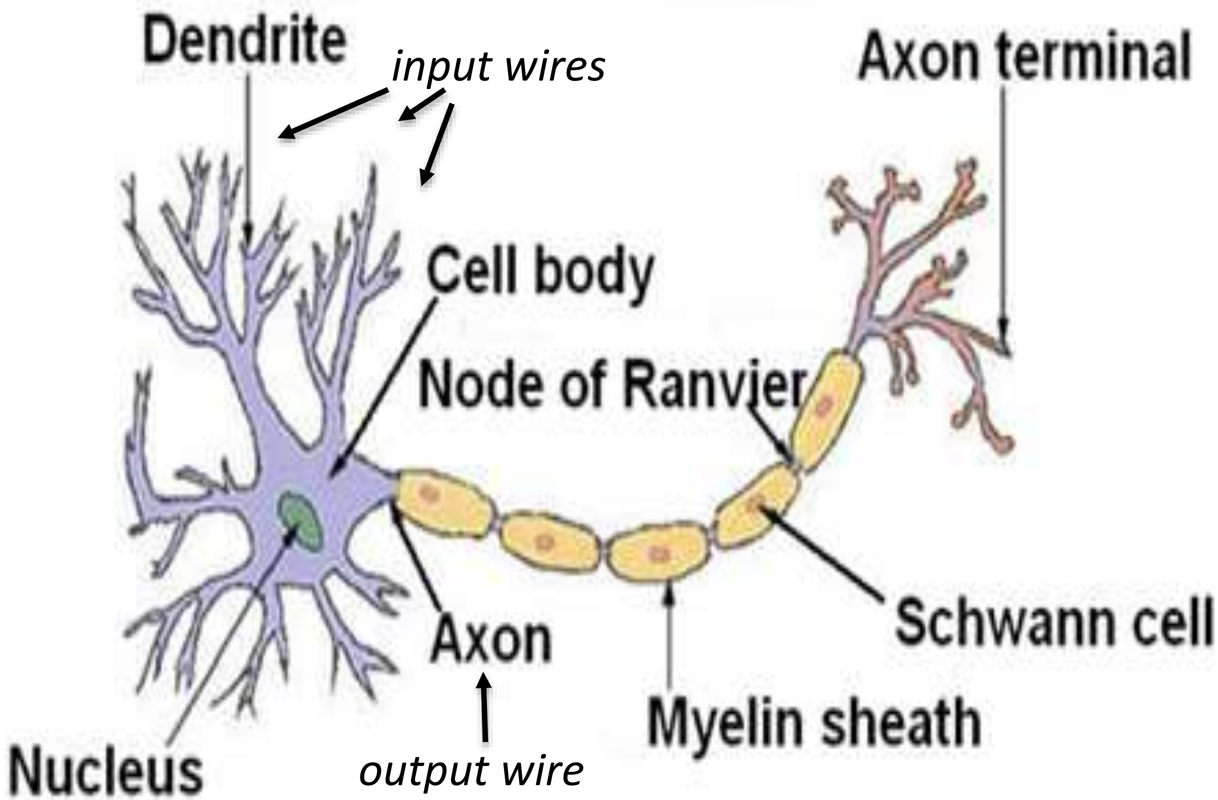
$$x = \begin{bmatrix} \text{Pixel 1 intensity} \\ \text{Pixel 2 intensity} \\ \vdots \\ \text{Pixel 2500 intensity} \end{bmatrix}$$

There are $(x_i \times x_j)$: ≈3 million quadratic features. Linear regression type of algorithm will not work (efficiently).

Andrew Ng

Artificial Neural Networks
Modeling of Neurons

Neuron in the brain

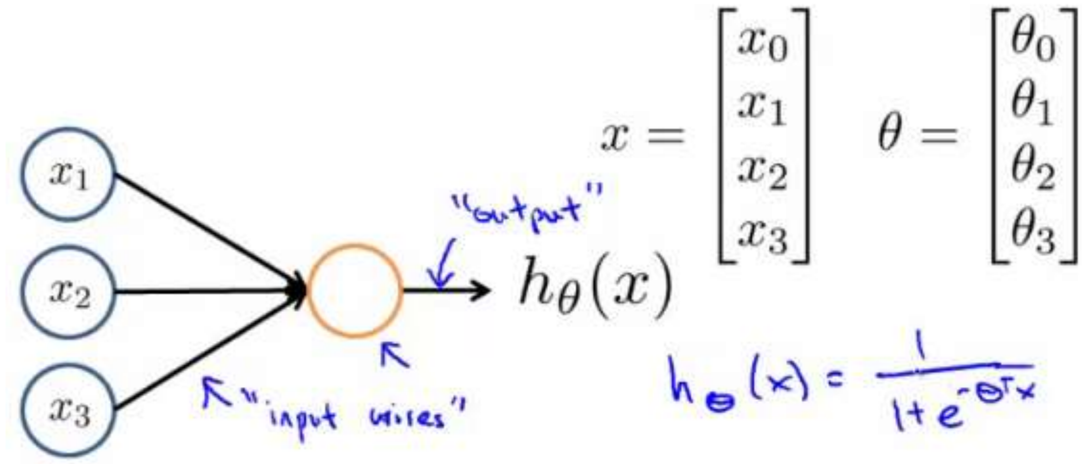


Three things to notice
- Cell body
- Number of input wires (dendrites)
- Output wire (axon)

Simple level
- Neurone gets one or more inputs through dendrites
- Does processing
- Sends output down axon

Neurons communicate through electric spikes
- Pulse of electricity via axon to another neuron

Logistic unit represents a Neuron



In an artificial neural network, a neuron is a logistic unit

- Feed input via input wires
- Logistic unit does computation
- Sends output down output wires

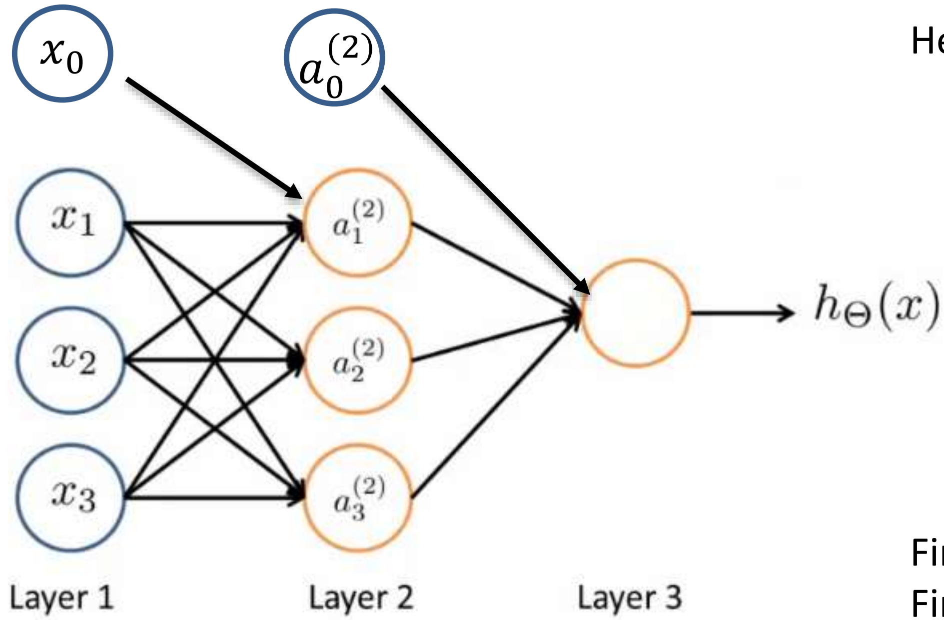
That logistic computation is just like the logistic regression hypothesis calculation

- This is an artificial neuron with a sigmoid (logistic) activation function
  - ϴ vector may also be called the **weights** of a model
- The above diagram is a single neuron.
- As we have seen this single neuron could do some useful thing.
- More complex classifications or problems might require models with several neurons.

Andrew Ng

Multi layer Neural Networks

- Below we have a group of neurons strung together



$a_i^{(j)}$ - activation of unit *i* in layer *j*

So, a_1^2 - is the **activation** of the 1st unit in the second layer

By activation, we mean the value which is computed and output by that node

$a_0^{(j)}$ bias term of layer j.

Here, input is x_1, x_2 and x_3

We could also call input activation on the first layer - i.e. (a_1^1, a_2^1 and a_3^1)

Three neurones in layer 2 (a_1^2, a_2^2 and a_3^2)

Final fourth neurone which produces the output

Which again we *could* call a_1^3

First layer is the **input layer**

Final layer is the **output layer** - produces value computed by a hypothesis

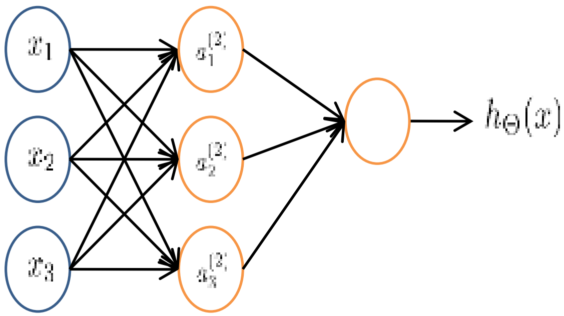
Middle layer(s) are called the **hidden layers**

You don't observe the values processed in the hidden layer

Not a great name

Can have many hidden layers

Neural Network, Notation



$a_i^{(j)} =$ "activation" of unit i in layer j

$\Theta^{(j)} =$ matrix of weights controlling function mapping from layer j to layer $j+1$

Column length of matrix Θ is the number of units in the following layer
Row length is the number of units in the current layer + 1 (because we have to map the bias unit). If we had two layers - 101 and 21 units in each, $\Theta^{(j)}$ =[21x102] matrix

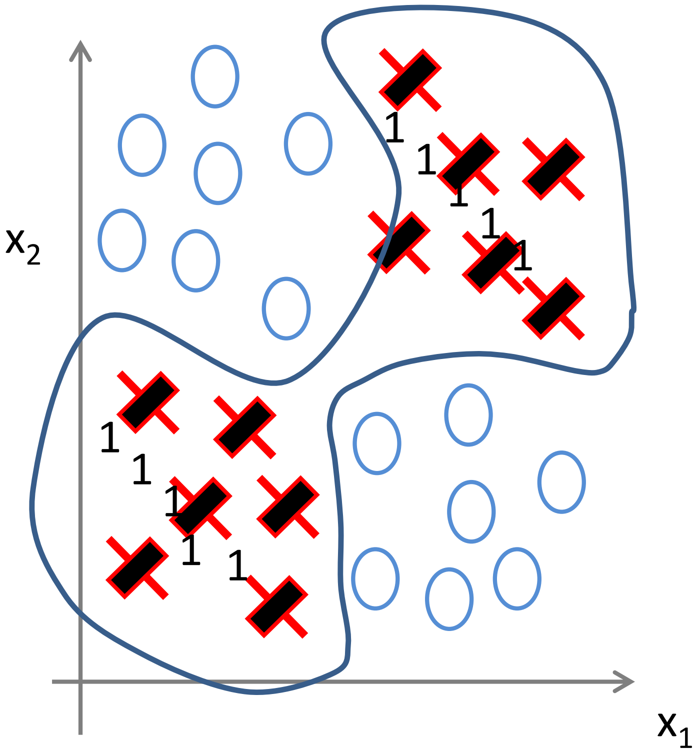
$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

If network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$.

Neural Networks: Examples

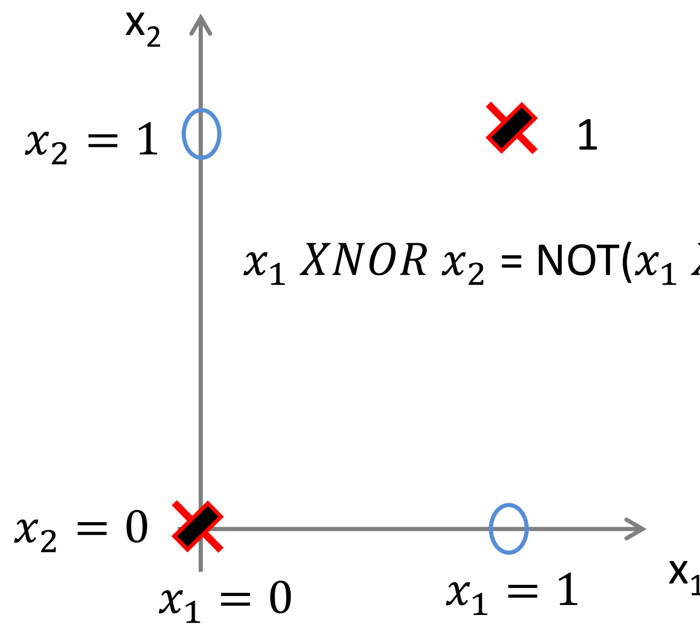
Non-linear classification example: XOR/XNOR

We would like to find the decision boundary between these two classes. A single straigh line would not do it.



We could roughly approximate this problem on the left with a simple logical problem: XNOR function.

x_1, x_2 are binary (0 or 1).

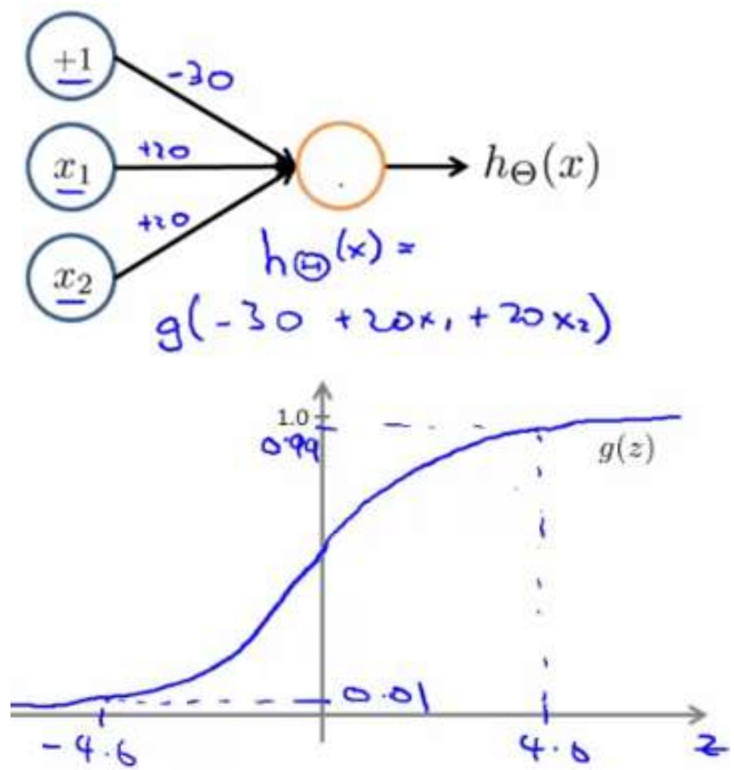


$x_1\ XNOR\ x_2$ = NOT($x_1\ XOR\ x_2$)

We will try to implement a neural net circuit for the "computation" on the right.
We start by looking at even simpler calculations.

Neural network AND Logical Function

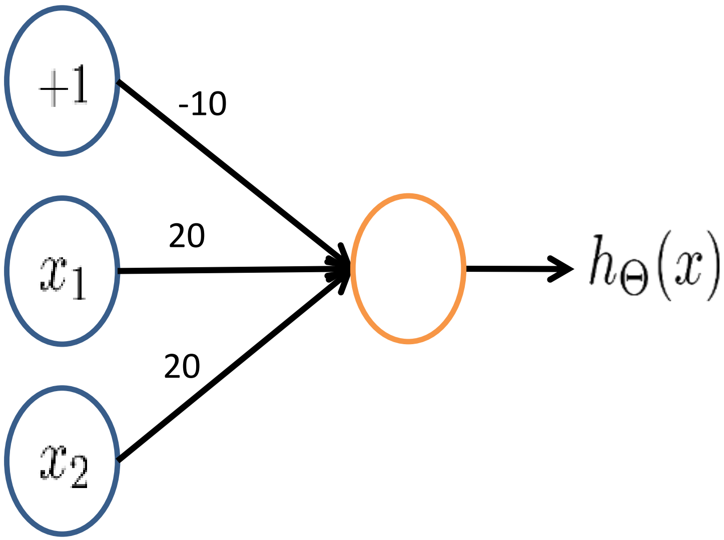
- Can we get a one-unit neural network to compute this logical AND function? Add a bias unit
  - Add some weights for the networks
- What are weights?
  - Weights are the parameter values which multiply into the input nodes (i.e. Θ)
- Sometimes it's convenient to add the weights into the diagram
  - These values are just the Θ parameters so
    - $\Theta_{10}^{1} = -30$
    - $\Theta_{11}^{1} = 20$
    - $\Theta_{12}^{1} = 20$
- Look at the four input values.
- The table on the right is called "logical table"
- Sigmoid function at +/-4.6 is 0.01 = 1% away from its saturation value.



x_1	x_2	$h_\Theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

$h_\Theta(x) \approx x_1$ AND x_2

Neural network OR function

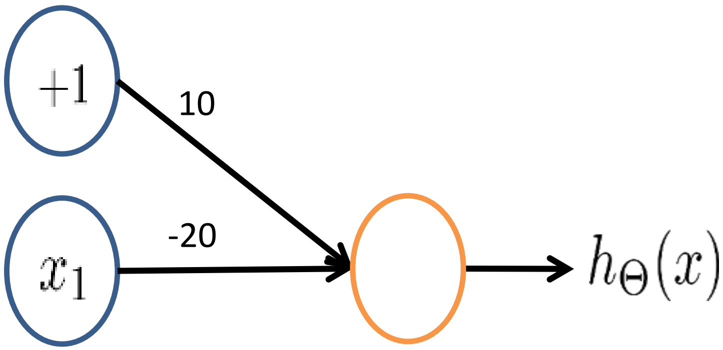


x_1	x_2	$h_\Theta(x)$
0	0	$g(-10) \sim 0$
0	1	$g(10) \sim 1$
1	0	~ 1
1	1	~ 1

$$g(-10 + 20x_1 + 20\,x_2)$$

Neural Network NOT function

- Negation is achieved by putting a large negative weight in front of the variable you want to negative



$$g(10\ -20x_1)$$

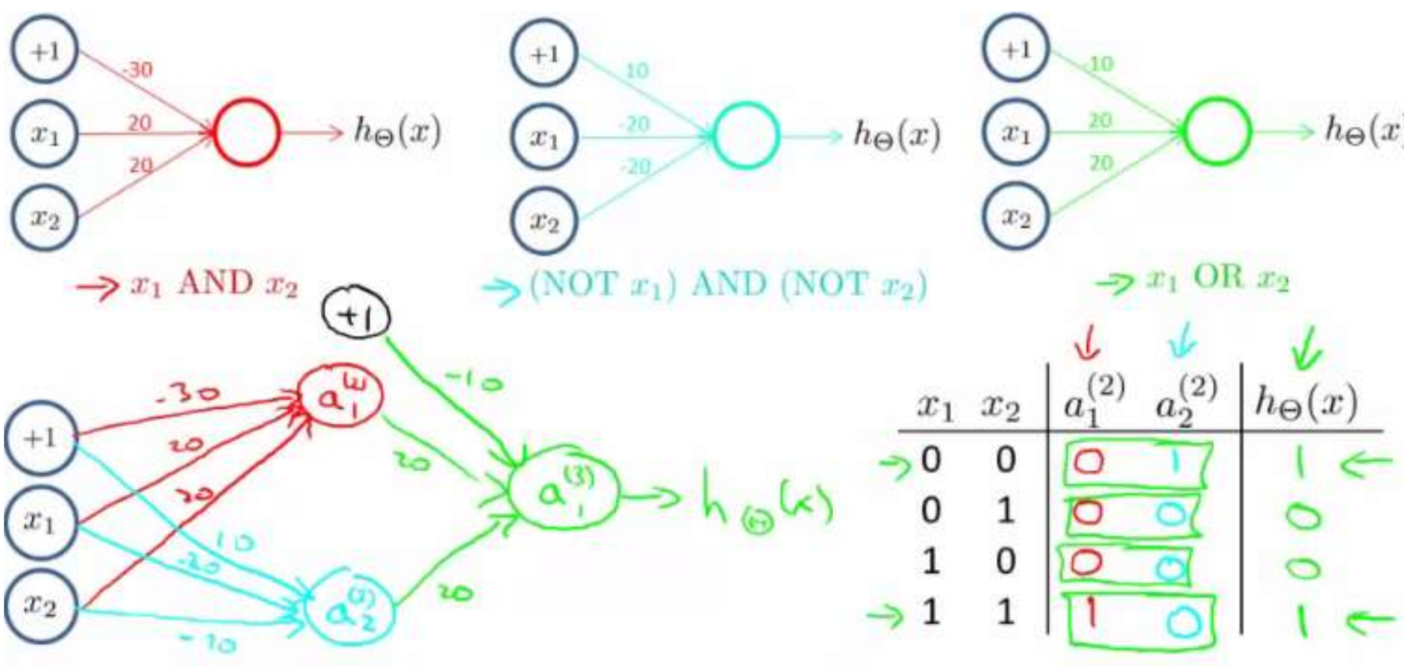
x_1	$h_\Theta(x)$
0	$g(10) \sim 1$
1	$g(-10) \sim 0$

Neural Network XNOR function

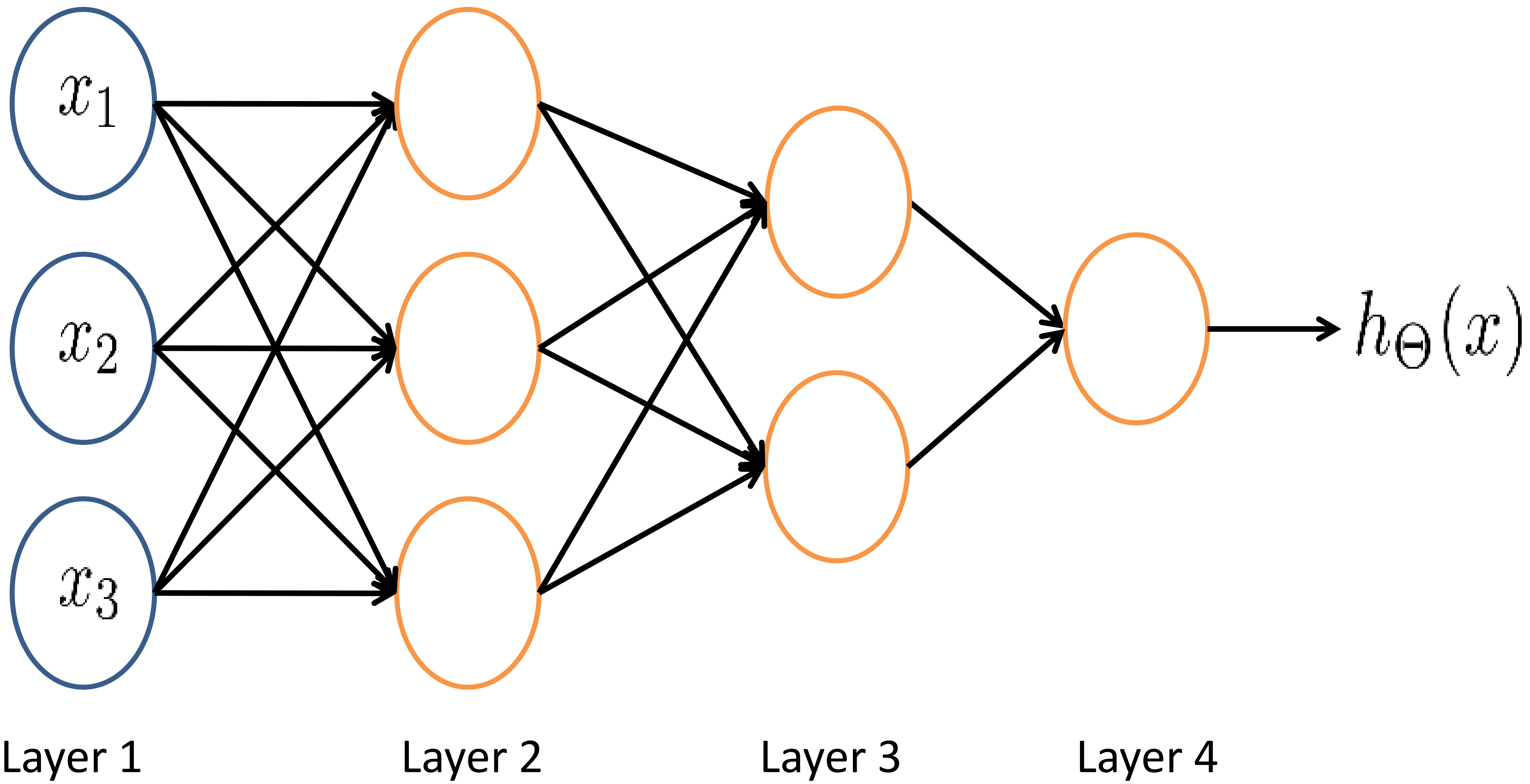
- XNOR is short for NOT XOR
  - i.e. NOT an exclusive or, so either (1,1) or (0,0)
- So we want to structure this so the input which produce a positive output are
  - AND (i.e. both true)

    OR Neither (which we can shortcut by saying not only one being true)

So we combine these into a neural network as shown below;



Neural Network intuition



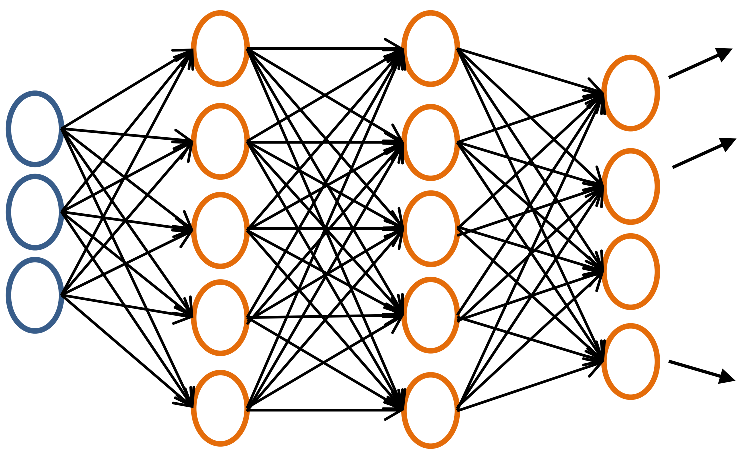
Multiple output units: One-vs-all.

Pedestrian

Car

Motorcycle

Truck

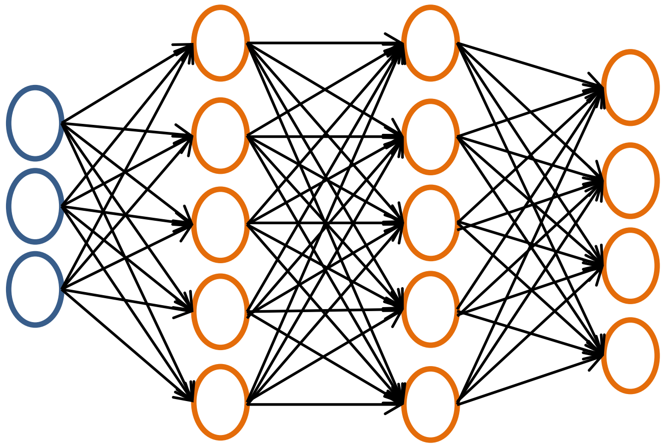


$$h_\Theta(x) \in \mathbb{R}^4$$

Want $h_\Theta(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$, etc.

when pedestrian when car when motorcycle

Multiclass classification



$$h_\Theta(x) \in \mathbb{R}^4$$

Want $h_\Theta(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$, etc.

when pedestrian when car when motorcycle

Training set: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$

$y^{(i)}$ one of $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

pedestrian car motorcycle truck

- $x^{(j)}$ is image j.
- It represents vector $y^{(j)}$

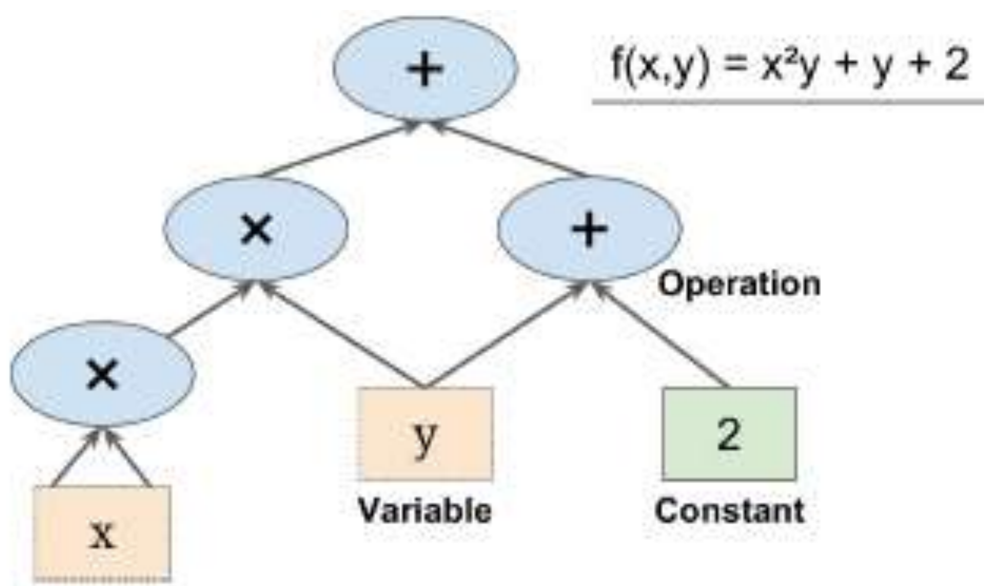
TensorFlow

TensorFlow

- TensorFlow™ is an open source software library for numerical computation using data flow graphs.
- Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them.
- The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.
- TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

Computational Graphs

- *TensorFlow* is an open source software library for numerical computation, particularly well suited and fine-tuned for large scale Machine Learning.
- Its basic principle is simple: you first define in a graph of computations to perform, then TensorFlow takes that graph and runs it efficiently using optimized C++ code.



Creating and running a graph

- Previous Graph is implemented in the following TensorFlow code:

```
import tensorflow as tf
reset_graph()
x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
f
```

- We wanted to see what is the value of f and got the type instead .

```
<tf.Tensor 'add_1:0' shape=() dtype=int32>
```

- In order to “see” calculated value of variable `f`, we have to initialize it and then use method `sess.run(f)` to calculate the actual value.

```
sess = tf.Session()
sess.run(x.initializer)
sess.run(y.initializer)
result = sess.run(f)
print(result)
42
sess.close()
```

Installation

- TensorFlow has API-s in four languages
  - Python
  - C++
  - Java
  - Go
- Main API is in Python.
- One has a impression that Google developers love Ubuntu. It might be prudent to create an Ubuntu VM and install TensorFlow on that VM.
- TensorFlow installs on other Linux OS-s, Mac OS X and Windows.
- Windows installation requires Python 3.5. According to TF site 3.6, as well.
- Tensor flow works with GPU cards. If you have NVIDIA CUDA processing unit, TensorFlow will be executed much faster. You have to check whether TF supports CUDA card you want to buy. Not all are supported.
- TensorFlow works on CPU only machines as well.
- For installation, please follow instructions on TensorFlow site:

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

Other Neural Networks Packages

- There are several other Neural Network packages of equal quality and similar performance as TensorFlow.
- Caffe Python, C++, Matlab Linux, OS X, Windows Y.Jia, UC Berkeley (BVLC) 2013
- Deeplearning4j Java, Scala, Clojure Linux, OS X, Windows, Android A.Gibson, Skymind 2014
- H2O Python, R Linux, OS X, Windows H2O.ai 2014
- Apache MXNet Python, C++, others Linux, OS X, Windows, iOS, Android DMLC 2015
- Theano Python Linux, OS X, iOS University of Montreal 2010
- Torch C++, Lua Linux, OS X, iOS, Android R.Collobert, K.Kavukcuoglu,C. Farabet 2002

Some Features of TensorFlow

- TF runs not only on Linux and MacOSX, but also on mobile devices, including both iOS and Android.
- TF provides a very simple python API called TF.Learn 2 (tensorflow.contrib.learn), compatible with Scikit-Learn: as you will see, you can use it to train various types of neural networks in just a few lines of code.
- TF provides another simple API called TF-slim (tensorflow.contrib.slim) to simplify building, training and evaluating neural networks.
- Several other high level APIs have been built independently on top of TensorFlow, such as Keras or Pretty Tensor.
- TF's main python API offers much more flexibility (at the cost of higher complexity) to create all sorts of computations, including any neural network architecture you can think of.
- TF includes highly efficient C++ implementations of many ML operations, particularly those needed to build neural networks. There is also a C++ API to define your own high-performance operations.
- TF provides several advanced optimization nodes to search for the parameters that minimize a cost function. These are very easy to use since TensorFlow automatically takes care of computing the gradients of the functions you define. This is called automatic differentiating (or autodiff).
- TF comes with a great visualization tool called TensorBoard that allows you to browse through the computation graph, view learning curves, and more.
- Google launched a cloud service to run TF graphs (https://cloud.google.com/ml/).
- TF has a dedicated team of passionate and helpful developers. Resources can be found online at https://www.tensorflow.org/, or https://github.com/jtoy/awesome-tensorflow).

Note on Suppressing Debugging Messages

- When running TensorFlow code you're able to set the logging verbosity to either DEBUG, INFO, WARN, ERROR, or FATAL.
- For example:

```
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)
```

- This tells TensorFlow to emit only real errors and not saturate your console with informations and warnings .

Missing dll-s

- When you run TensorFlow code, at least on Windows, you might see a moderately large number of Errors telling that you are missing several Cuda libraries: `cudnn64_5.dll, cubas64_80.dll, cufft64_80.dll, curand64_80.dll.`
- First, note that cuDNN is not distributed with the rest of the CUDA toolkit, so you will need to download it separately from [the NVIDIA website](https://developer.nvidia.com/cudnn).
- You have to visit https://developer.nvidia.com/rdp/cudnn-download and select your operating system.
- On Windows, it is distributed as a ZIP archive. You must expand it and find the directory containing `cudnn64_5.dll`. For example, if you extract it to `C:\tools\cuda`, the DLL will be in `C:\tools\cuda\bin\cudnn64_5.dll`. Place that directory in your `Path` environmental variable. On Linux and Mac OS do it by typing the following at the command prompt:

```
C:\> set PATH=%PATH%;C:\tools\cuda\bin
C:\> python ...
>>> import tensorflow as tf
```

Starting `TensorBoard`

- TensorFlow comes with a convenient graphical utility called TensorBoard which allows you to see created computational graphs and to monitor progress of the training of your neural network.
- TensorBoard needs to know where to read graphs created by Python code.
- To export those graphs, so that they could be imported into the TensorBoard, add the following line after crating the Session:

```
file_writer = tf.summary.FileWriter("E:/code/output", sess.graph)
```

- TensorBoard is started by pointing to directory "`output`". This directory could have any name and reside anywhere on your operating system. If TB would not start try providing the absolute path to that directory as the value of option `–logdir "…"` of the `tensorboard` command

```
C:\> tensorboard --logdir "E:/code/output"
```

- If your graphs are not found add `--debug` flag to the above command.
- Once TensorBoard server starts, you can see a web page on port 6006 of your localhost. Once on the page go to Graphs

Basic Operations

- Like in MatLab where everything is a matrix, in TensorFlow everything is a tensor.
- Import TensorFlow and Numpy:

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

- Now, define a 2x2 matrix in different ways:

```
m1 = [[1.0, 2.0],
      [3.0, 4.0]]
m2 = np.array([[1.0, 2.0],
               [3.0, 4.0]], dtype=np.float32)
m3 = tf.constant([[1.0, 2.0],
                  [3.0, 4.0]])
print(type(m1))
print(type(m2))
print(type(m3))
<class 'list'>
<class 'numpy.ndarray'>
<class 'tensorflow.python.framework.ops.Tensor'>
```

- If you really want only tensors, you can do this

```
t1 = tf.convert_to_tensor(m1, dtype=tf.float32)
t2 = tf.convert_to_tensor(m2, dtype=tf.float32)
t3 = tf.convert_to_tensor(m3, dtype=tf.float32)
print(type(m1))
<class 'tensorflow.python.framework.ops.Tensor'>
```

[[-1 -2]]
)

Evaluate an object, `run()`

- Start again:

```
import tensorflow as tf
x = tf.constant([[1, 2]])        # create a 1x2 matrix, oops, tensor.
```

- Let's negate it. Define the negation op to be run on the matrix:

```
neg_x = tf.negative(x)
```

- Let us see new value:

```
print(neg_x)
Tensor("Neg_3:0", shape=(1, 2),dtype=int32) # this is not [-1,-2]
```

- You need to summon a session so you can launch the negation op:

```
with tf.Session() as sess:
    result = sess.run(neg_x, x)
    print(result)


[[-1 -2]]
```

- Moral of the story. You must create an object of type Session and then invoke method `run()` on that object. To method `run()` you pass whatever object you want to calculate. No calculation takes place before method `run ()` is called.

Interactive Session, `eval()`

- Interactive sessions are different from regular Session is that an `InteractiveSession` installs itself as the default session on construction. The methods tf.Tensor.eval and tf.Operation.run will use that session to run ops.
- This is convenient in interactive shells and IPython notebooks, as it avoids having to pass an explicit Session object to run ops.

```
import tensorflow as tf
sess = tf.InteractiveSession()
```

- We have a matrix we want to invert:

```
x = tf.constant([[1., 2.]])
neg_op = tf.negative(x)
```

- `neg_op` is an operation. We are in an interactive session, so we can just call the `eval()` method on the op.

```
result = neg_op.eval()
print(result)
[[-1. -2.]]
```

- That code's a little cleaner when using Jupyter notebooks (like this one).
- Don't forget to close the session:

```
sess.close()
```

Variables

- We need a better understanding of variables. Start with a session:

```
import tensorflow as tf
sess = tf.InteractiveSession()
```

- Below is a series of numbers.

```
raw_data = [1., 2., 8., -1., 0., 5.5, 6., 13]
```

- Create a boolean variable called spike to detect a sudden increase in the values.
- All variables must be initialized. Go ahead and initialize the variable by calling run() on its initializer:

```
spike = tf.Variable(False)
spike.initializer.run()
```

- Loop through the data and update the spike variable when there is a significant increase:

```
for i in range(1, len(raw_data)):
    if raw_data[i] - raw_data[i-1] > 5:
        updater = tf.assign(spike, tf.constant(True))
        updater.eval()
    else:
        tf.assign(spike, False).eval()
    print("Spike", spike.eval())
Spike False
Spike True
Spike False
Spike False
Spike True
Spike False
Spike True
sess.close()
```

Alternative ways of running a graph

- Quite often we define a `sess` object within a `with` scope.
- We also can initialize all variables using

```
init = tf.global_variables_initializer()

with tf.Session() as sess:
    init.run()
    result = f.eval()

result
42
```

- `sess` object does not have to be wrapped in a `with` block

```
init = tf.global_variables_initializer()
sess = tf.InteractiveSession()
init.run()
result = f.eval()
print(result)
42
sess.close()
result
42
```

Managing graphs

```
x1 = tf.Variable(1)
x1.graph is tf.get_default_graph()
True
graph = tf.Graph()
with graph.as_default():
    x2 = tf.Variable(2)
x2.graph is graph
True
x2.graph is tf.get_default_graph()
False
w = tf.constant(3)
x = w + 2
y = x + 5
z = x * 3
with tf.Session() as sess:
    print(y.eval())  # 10
10
with tf.Session() as sess:
    y_val, z_val = sess.run([y, z])
print(z_val)  # 15
15
```

Graphs, Name Scopes

TF Session

- Sessions, as seen in the previous exercise, are responsible for graph execution. The constructor *`https://www.tensorflow.org/versions/master/api_docs/python/`*
- *`client.html#Session.init`*`[tf.Session()]` takes in three optional parameters:
- *`target`* specifies the execution engine to use. For most applications, this will be left at its default empty string value. When using sessions in a distributed setting, this parameter is used to connect to `tf.train.Server` instances
- *`graph`* specifies the Graph object that will be launched in the Session. The default value is None, which indicates that the current default graph should be used. When using multiple graphs, it's best to explicitly pass in the Graph you'd like to run (instead of creating the Session inside of a with block).
- *`config`* allows users to specify options to configure the session, such as limiting the number of CPUs or GPUs to use, setting optimization parameters for graphs, and logging options.

Initialize Session

- In a typical TensorFlow program, Session objects will be created without changing any of the default construction parameters.

```
import tensorflow as tf
# Create Operations, Tensors, etc (using the default graph)
a = tf.add(2, 5)
b = tf.mul(a, 3)
# Start up a `Session` using the default graph
sess = tf.Session()
```

- Note that these two calls are identical:

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

- Once a Session is opened, you can use its primary method, `run()`, to calculate the value of a desired Tensor output:

```
sess.run(b) #
```

- `Session.run()` takes in one required parameter, which is called `fetches`, as well as three optional parameters: `feed_dict`, `options`, and `run_metadata`.
- We won't cover `options or run_metadata`, as they are still experimental (thus prone to being changed) and are of limited use at this time.
- `feed_dict`, however, is important to understand

Closing Session

- After you are finished using the Session, call its close() method to release unneeded resources. Open session

```
sess = tf.Session()
# Run the graph, write summary statistics, etc.
...
# Close the graph, release its resources
sess.close()
```

- As an alternative, you can also use the Session as a context manager, which will automatically close when the code exits its scope:

```
with tf.Session() as sess:
   # Run graph, write summary statistics, etc.
   ...
   # The Session closes automatically
```

- We can also use a Session as a context manager by using its `as_default()` method. Similarly to how Graph objects can be used implicitly by certain Operations, you can set a session to be used automatically by certain functions. The most common of such functions are `Operation.run()` and `Tensor.eval()`, which act as if you had passed them in to `Session.run()` directly. For example, define simple constant

```
a = tf.constant(5)
# Open up a Session
sess = tf.Session()
# Use the Session as a default inside of `with` block
with sess.as_default():
    a.eval()

# Have to close Session manually.
sess.close()
```

Fetches

- `Session.run()` takes in one required parameter, `fetches`, as well as three optional parameters: `feed_dict`, `options`, and `run_metadata`.
- In previous examples, we set `fetches` to a tensor, e.g. b (the output of the `tf.mul` Operation). This tells TensorFlow that the Session should find all of the nodes necessary to compute the value of `b`, execute them in order, and output the value of b.
- We can also pass in a list of graph elements:

```
sess.run([a, b])
```

- When `fetches` is a list, the output of run() will be a list with values corresponding to the output of the requested elements. In this example, we ask for the values of a and b, in that order. Since both a and b are tensors, we receive their values as output.
- In addition using `fetches` to get Tensor outputs, you'll also see examples where we give `fetches` a direct handle to an Operation which is a useful side-effect when run. An example of this is `tf.global_variables_initializer()`, which prepares all TensorFlow Variable objects to be used.
- We still pass the Op as the `fetches` parameter, but the result of `Session.run()` will be None:

```
# Performs the computations needed to initialize Variables, but returns `None`
sess.run(tf.global_variables_initializer())
```

feed_dict(ionary)

- The parameter `feed_dict` is used to override Tensor values in the graph, and it expects a Python dictionary object as input. The keys in the dictionary are handles to Tensor objects that should be overridden, while the values can be numbers, strings, lists, or NumPy arrays. The values must be of the same type (or able to be converted to the same type) as the Tensor key.
- Let's see how we can use `feed_dict` to overwrite the value of a in a graph:

```
import tensorflow as tf
# Create Operations, Tensors, etc (using the default graph)
a = tf.add(2, 5)
b = tf.multiply(a, 3)
# Start up a `Session` using the default graph
sess = tf.Session()
# Define a dictionary that says to replace the value of `a` with 15
replace_dict = {a: 15}
# Run the session, passing in `replace_dict` as the value to `feed_dict`
sess.run(b, feed_dict=replace_dict) # returns 45
```

- Notice that even though Variable `a` would normally evaluate to 7, the dictionary we passed into `feed_dict` replaced that value with 15.
- `feed_dict` can be extremely useful in a number of situations. Because the value of a tensor is provided up front, the graph no longer needs to compute any of the tensor's normal dependencies. This means that if you have a large graph and want to test out part of it with dummy values, TensorFlow won't waste time with unnecessary computations.
- `feed_dict` is also useful for specifying input values, as can be seen when dealing with placeholders.

`run()` vs. `eval()`

- What is the difference between `Session.run()` and `Tensor.eval()`?
- If `t` is a Tensor object, `t.eval()` is shorthand for `sess.run(t)` (where `sess` is the current default session. The two following snippets of code are equivalent:
- ```
  # Using `Session.run()`.
  sess = tf.Session()
  c = tf.constant(5.0)
  print sess.run(c)

  # Using `Tensor.eval()`.
  c = tf.constant(5.0)
  with tf.Session():
    print c.eval()
  ```
- In the second example, the session acts as a context manager, which has the effect of installing it as the default session for the lifetime of the with block.
- The context manager approach can lead to more concise code for simple use cases (like unit tests); if your code deals with multiple graphs and sessions, it may be more straightforward to make explicit calls to `Session.run()`.
- `c.eval()` is equivalent to running calculation of tensor `c` on the default graph. `c.eval()` returns one tensor, `c`, only.
- `Session.run([a,b,..])` could return multiple tensors.
  ```

Graphical Graphs

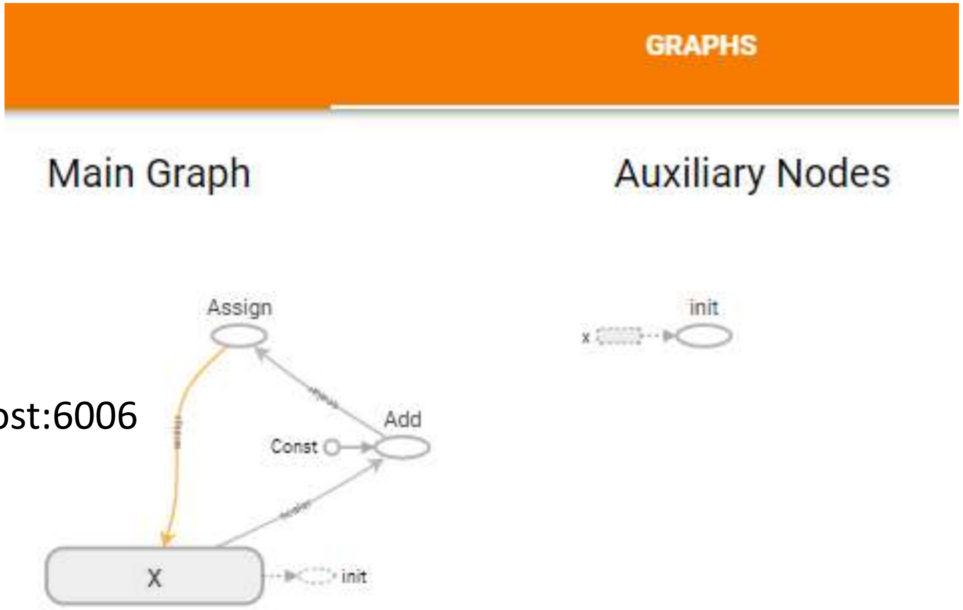
- Run this code:

```
import tensorflow as tf
with tf.Graph().as_default() as g:
   x = tf.Variable(1.0, name="x")
   add_op = tf.add(x, tf.constant(1.5))
   assign_op = tf.assign(x, add_op)
   init = tf.global_variables_initializer()
   sess = tf.Session()
   file_writer = tf.summary.FileWriter("output", sess.graph)
   sess.run(init)
   sess.run(assign_op)
   print(sess.run(x))
```

- Then go to the local directory and run:

```
C:\..> tensorboard --logdir output
```

- Then open a breowser and go to localhost:6006



Organize Graph with Name Scopes

- So far, we've only worked with toy graphs containing a few nodes and small tensors, but real world models can contain dozens or hundreds of nodes, as well as millions of parameters. In order to manage this level of complexity, TensorFlow currently offers a mechanism to help organize your graphs: *name scopes*.
- Name scopes allow you to group Operations into larger, named blocks. Then, when you launch your graph with TensorBoard, each name scope will encapsulate its own Ops and Variables, making the visualization much more digestible.
- For basic name scope usage, simply add your Operations in a

  `with tf.name_scope(<name>)` block

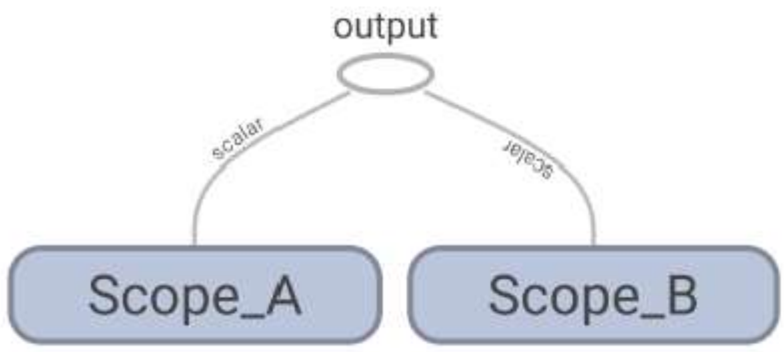
```
import tensorflow as tf
with tf.name_scope("Scope_A"):
   a = tf.add(1, 2, name="A_add")
   b = tf.mul(a, 3, name="A_mul")

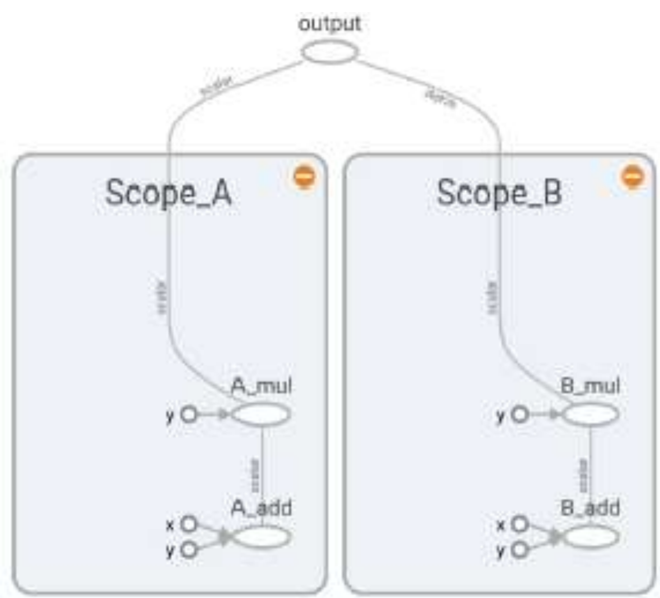
with tf.name_scope("Scope_B"):
   c = tf.add(4, 5, name="B_add")
   d = tf.mul(c, 6, name="B_mul")
   e = tf.add(b, d, name="output")

writer = tf.summary.FileWriter('name_scope_1',graph=tf.get_default_graph())
writer.close()
```

Resulting Graph

- Graph generated as the result of previous code should look like this:





- You’ll notice that the add and multiply Operations we added to the graph aren’t immediately visible. Instead, we see their enclosing name scopes. You can expand the name scope boxes by clicking on the plus + icon in their upper right corner.
- Inside of each scope, you’ll see the individual Operations you’ve added to the graph.

Nested Name Scopes

- You can also nest name scopes within other name scopes:

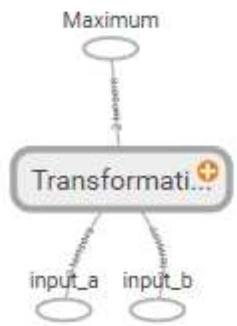
```
graph = tf.Graph()
with graph.as_default():
   in_1 = tf.placeholder(tf.float32, shape=[],name="input_a")
   in_2 = tf.placeholder(tf.float32, shape=[],name="input_b")
   const = tf.constant(3, dtype=tf.float32,name="static_value")
with tf.name_scope("Transformation"):
   with tf.name_scope("A"):
      A_mul = tf.multiply(in_1, const)
      A_out = tf.subtract(A_mul, in_1)
   with tf.name_scope("B"):
      B_mul = tf.multiply(in_2, const)
      A_out = tf.subtract(B_mul, in_2)
   with tf.name_scope("C"):
      C_div = tf.divide(A_out, B_out)
      C_out = tf.add(C_div, const)
   with tf.name_scope("D"):
      D_div = tf.divide(B_out, A_out)
      D_out = tf.add(D_div, const)

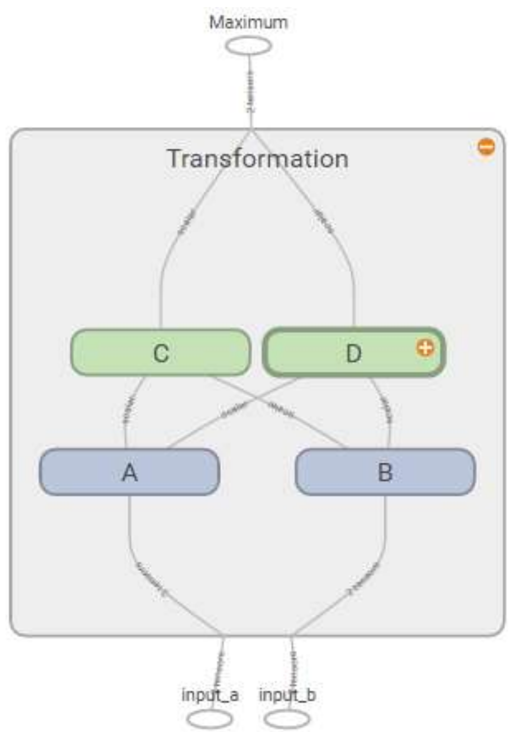
out = tf.maximum(C_out, D_out)
writer = tf.summary.FileWriter('name_scope_2', graph=graph)
#writer.close()
# c:\> tensorboard -logdir name_scope2
```

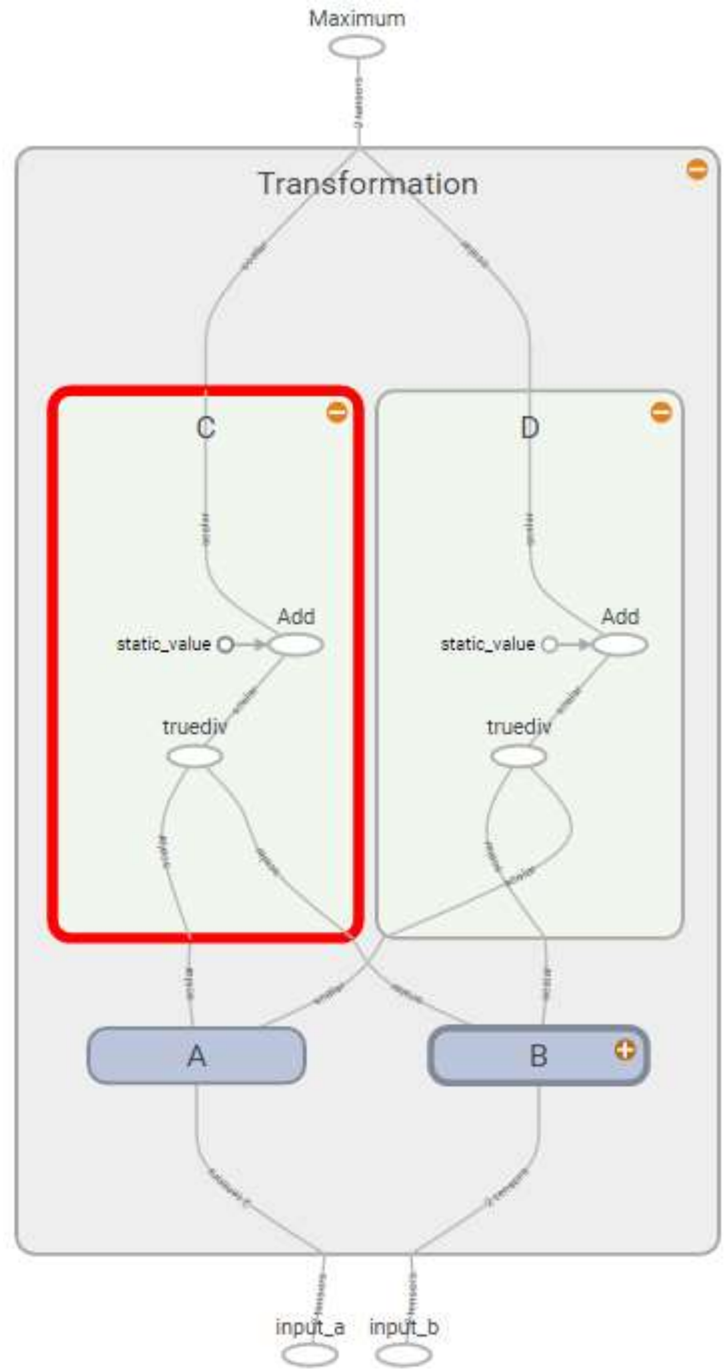
Composite Graph

- In TensorBoard graph, overall name scope `Transformations` shows first.
- If we click on the + sign in its upper right corner you will see internal scopes A, B, C and D.
- When we click the + sign on any other internal name_scope, they expand revealing internal operations, constants and variables.
- Subgraphs with same color have identical operations content.
- Separating a huge graph into meaningful clusters can make understanding the model much easier.

Main Graph







Converting Older TF API to 1.x

- The APIs in TensorFlow 1.0 have changed in ways that are not all backwards compatible. That is, TensorFlow programs that worked on TensorFlow 0.n won't necessarily work on TensorFlow 1.0. These API changes are made to ensure an internally-consistent API. There are no plans for backwards-breaking changes throughout the 1.N lifecycle.
- If you would like to automatically port your code to 1.0, you can try our `tf_upgrade.py` script. Manual changes are sometimes necessary. Get this script from our https://github.com/tensorflow/tensorflow, directory `tensorflow/tensorflow/tools/compatibility`
- To convert a single 0.n TensorFlow source file to 1.0, enter a command of the following format:
- $ **python tf_upgrade.py --infile** *InputFile* **--outfile** *OutputFile* For example,
- $ **python tf_upgrade.py --infile test.py --outfile test_1.0.py**
- The tf_upgrade.py script also generates a file named report.txt, which details all the changes it performed and makes additional suggestions about changes you might need to make manually.
- To upgrade a whole directory of 0.n TensorFlow programs to 1.0, enter a command having the following format:
- $ **python tf_upgrade.py --intree** *InputDir* **--outtree** *OutputDir* For example, the following command converts all the 0.n TensorFlow programs in the /home/user/cool directory, creating their 1.0 equivalents in the /home/user/cool_1.0 directory:
- $ **python tf_upgrade.py --intree /home/user/cool --outtree /home/user/cool_1.0** Limitations
- There are a few things to watch out for. Specifically:
- You must manually fix any instances of tf.reverse(). The tf_upgrade.py script will warn you about tf.reverse() in stdout and in the report.txt file.
- On reordered arguments, tf_upgrade.py tries to minimally reformat your code, so it cannot automatically change the actual argument order. Instead, tf_upgrade.py makes your function invocations order-independent by introducing keyword arguments.
- Constructions like tf.get_variable_scope().reuse_variables() will likely not work. We recommend deleting those lines and replacing them with lines such as the following:
- with tf.variable_scope(tf.get_variable_scope(), reuse=True):

Summary Operations

- There are 7 summary operations:
- `tf.summary.tensor_summary`
- `tf.summary.scalar`
- `tf.summary.histogram`
- `tf.summary.audio`
- `tf.summary.image`
- `tf.summary.merge`
- `tf.summary.merge_all`

`tf.summary.scalar(name, tensor, collections=None)`

- Outputs a Summary protocol buffer containing a single scalar value.
- The generated Summary has a Tensor.proto containing the input Tensor.
- Args:
  - **name**: A name for the generated node. Will also serve as the series name in TensorBoard.
  - **tensor**: A real numeric Tensor containing a single value.
  - **collections**: Optional list of graph collections keys. The new summary op is added to these collections. Defaults to [GraphKeys.SUMMARIES].
- Returns:
- A scalar Tensor of type string. Which contains a Summary protobuf.

tf.summary.tensor_summary()

```
tf.summary.tensor_summary(name,tensor,summary_description=None,
collections=None)
```

- Outputs a Summary protocol buffer with a serialized tensor.proto.
- The generated Summary has one summary value containing the input tensor.
- Args:
  - **name**: A name for the generated node. Will also serve as the series name in TensorBoard.
  - **tensor**: A tensor of any type and shape to serialize.
  - **summary_description**: Optional summary_pb2.SummaryDescription()
  - **collections**: Optional list of graph collections keys. The new summary op is added to these collections. Defaults to [GraphKeys.SUMMARIES].
- Returns:
- A scalar Tensor of type string. The serialized Summary protocol buffer.
- Defined in tensorflow/python/ops/summary_ops.py.

`tf.div`

`tf.div(x, y, name=None)`

- Divides x / y elementwise (using Python 2 division operator semantics).
- NOTE: Prefer using the Tensor division operator or `tf.divide` which obey Python division operator semantics.
- This function divides x and y, forcing Python 2.7 semantics. That is, if one of x or y is a float, then the result will be a float. Otherwise, the output will be an integer type. Flooring semantics are used for integer division.
- Args:
  - **x**: Tensor numerator of real numeric type.
  - **y**: Tensor denominator of real numeric type.
  - **name**: A name for the operation (optional). Returns: x / y returns the quotient of x and y

runnable_graph.py

```
import tensorflow as tf
import numpy as np
# Explicitly create a Graph object
graph = tf.Graph()
with graph.as_default():
    with tf.name_scope("variables"):
        # Variable to keep track of how many times the graph has been run
        global_step = tf.Variable(0, dtype=tf.int32, name="global_step")
        # Variable that keeps track of the sum of all output values over time
        total_output = tf.Variable(0,dtype=tf.int32,trainable=False,
name="total_output")
    # Primary transformation Operations
    with tf.name_scope("transformation"):
        # Separate input layer
        with tf.name_scope("input"):
            # Create input placeholder- takes in a Vector
            a = tf.placeholder(tf.float32, shape=[None], name="input_placeholder_a")
            # Create input placeholder
            a=tf.placeholder(tf.int32,shape=[None],name="input_placeholder_a")

        # Separate middle layer
        with tf.name_scope("intermediate_layer"):
            b = tf.reduce_prod(a, name="product_b")
            c = tf.reduce_sum(a, name="sum_c")
```

runnable_graph.py, continued

```
    # Separate output layer
    with tf.name_scope("output"):
        output = tf.add(b, c, name="outpu")

    with tf.name_scope("update"):
        # Increment the total_output Variable by the latest output
        update_total = total_output.assign_add(output)
        # Increment the above global_step, should run whenever the Graph is run
        increment_step = global_step.assign_add(1)

# Summary Operations
with tf.name_scope("summaries"):
    avg = tf.div(update_total, increment_step,name="average")
    # Creates summary for output node
    tf.summary.scalar(name="output",tensor=output)
    tf.summary.scalar(name="update_total",tensor=update_total)
    tf.summary.scalar(tensor=avg,name="average_summary")
```

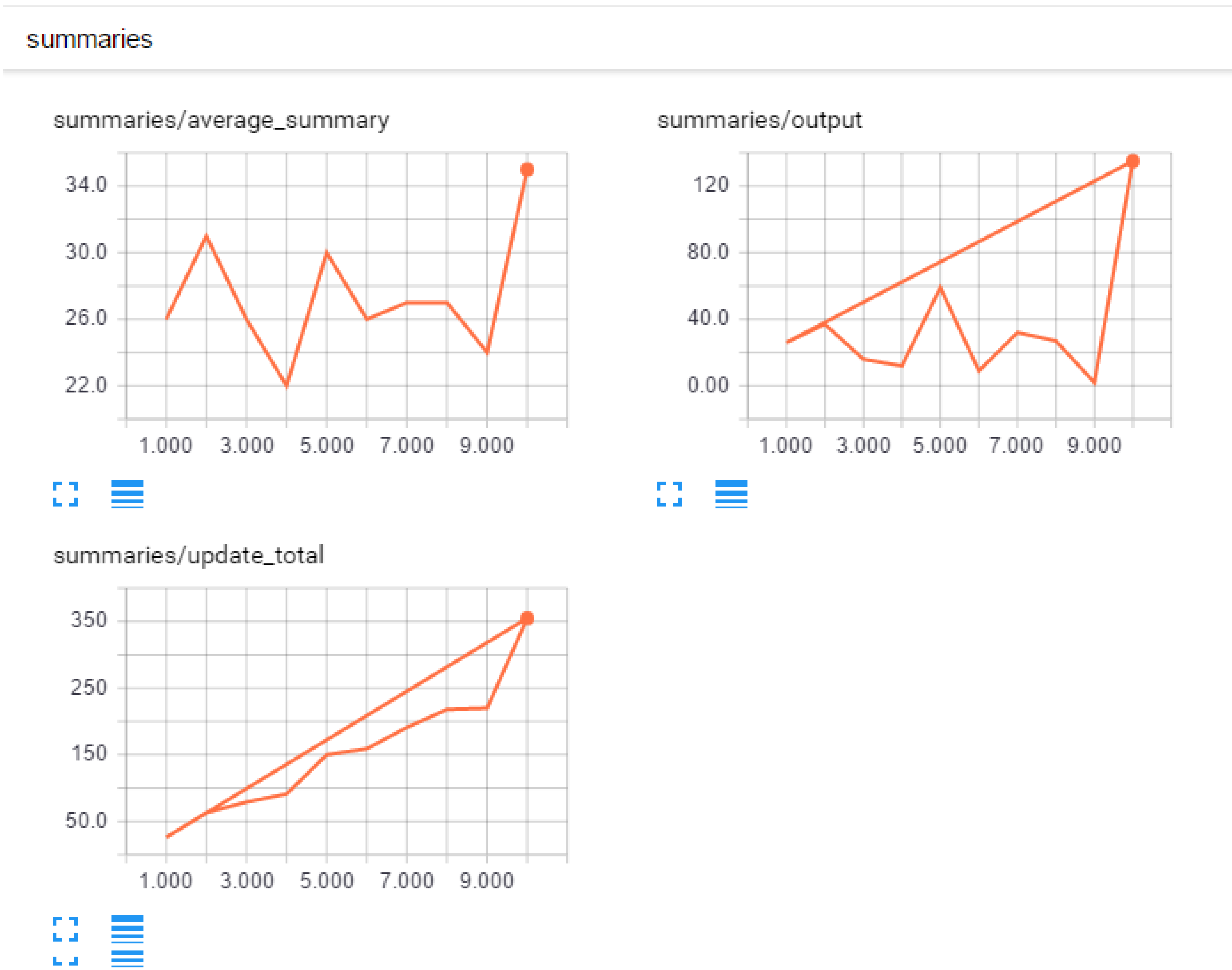
runnable_graph.py, continued

```
    # Global Variables and Operations
    with tf.name_scope("global_ops"):
        # Initialization Op
        init = tf.global_variables_initializer()
        # Collect all summary Ops in graph
        merged_summaries = tf.summary.merge_all()

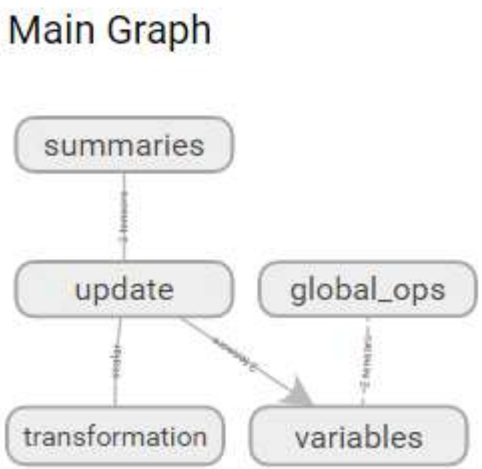
# Start a Session, using the explicitly created Graph
sess = tf.Session(graph=graph)
# Open a SummaryWriter to save summaries
writer = tf.summary.FileWriter('improved_graph', graph)
# Initialize Variables
sess.run(init)
def run_graph(input_tensor):
    "Helper function; runs the graph with given input tensor and saves summaries "
    feed_dict = {a: input_tensor}
    _,step, summary = sess.run([output, increment_step, merged_summaries],
feed_dict=feed_dict)
    writer.add_summary(summary, global_step=step)

# Run the graph with various inputs
run_graph([2,8]); run_graph([3,1,3,3])
run_graph([8]) ; run_graph([1,2,3])
run_graph([11,4]) ; run_graph([4,1])
run_graph([7,3,1]) ; run_graph([6,3])
run_graph([0,2]) ; run_graph([4,5,6])
```

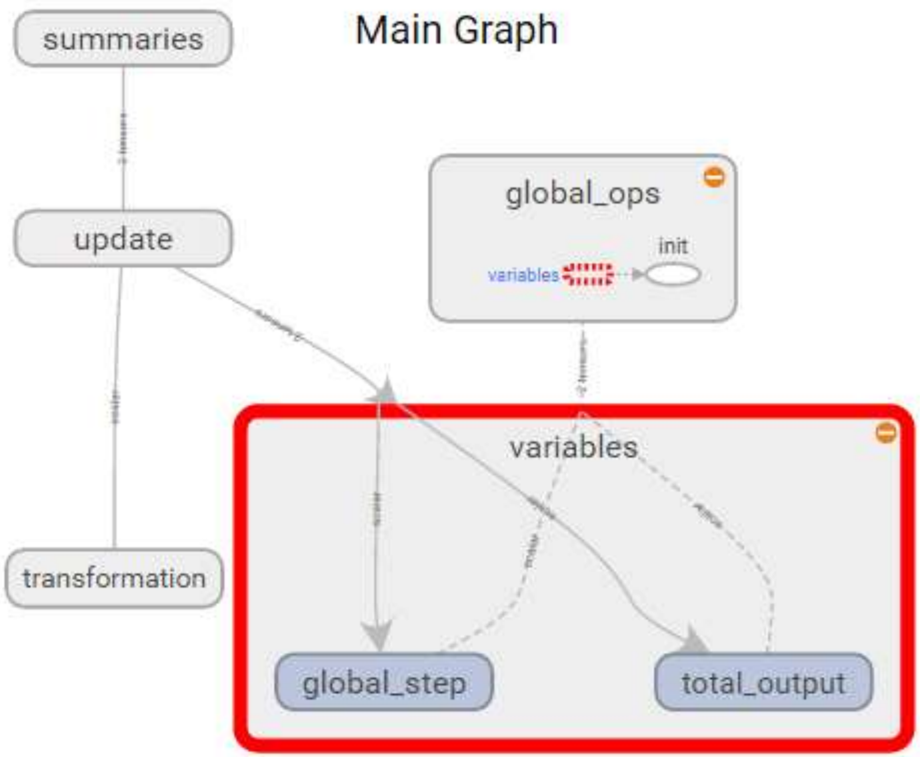
Summaries


summaries
summaries/average_summary
34.0
30.0
26.0
22.0
1.000
3.000
5.000
7.000
9.000
summaries/output
120
80.0
40.0
0.00
1.000
3.000
5.000
7.000
9.000
summaries/update_total
350
250
150
50.0
1.000
3.000
5.000
7.000
9.000

Graph

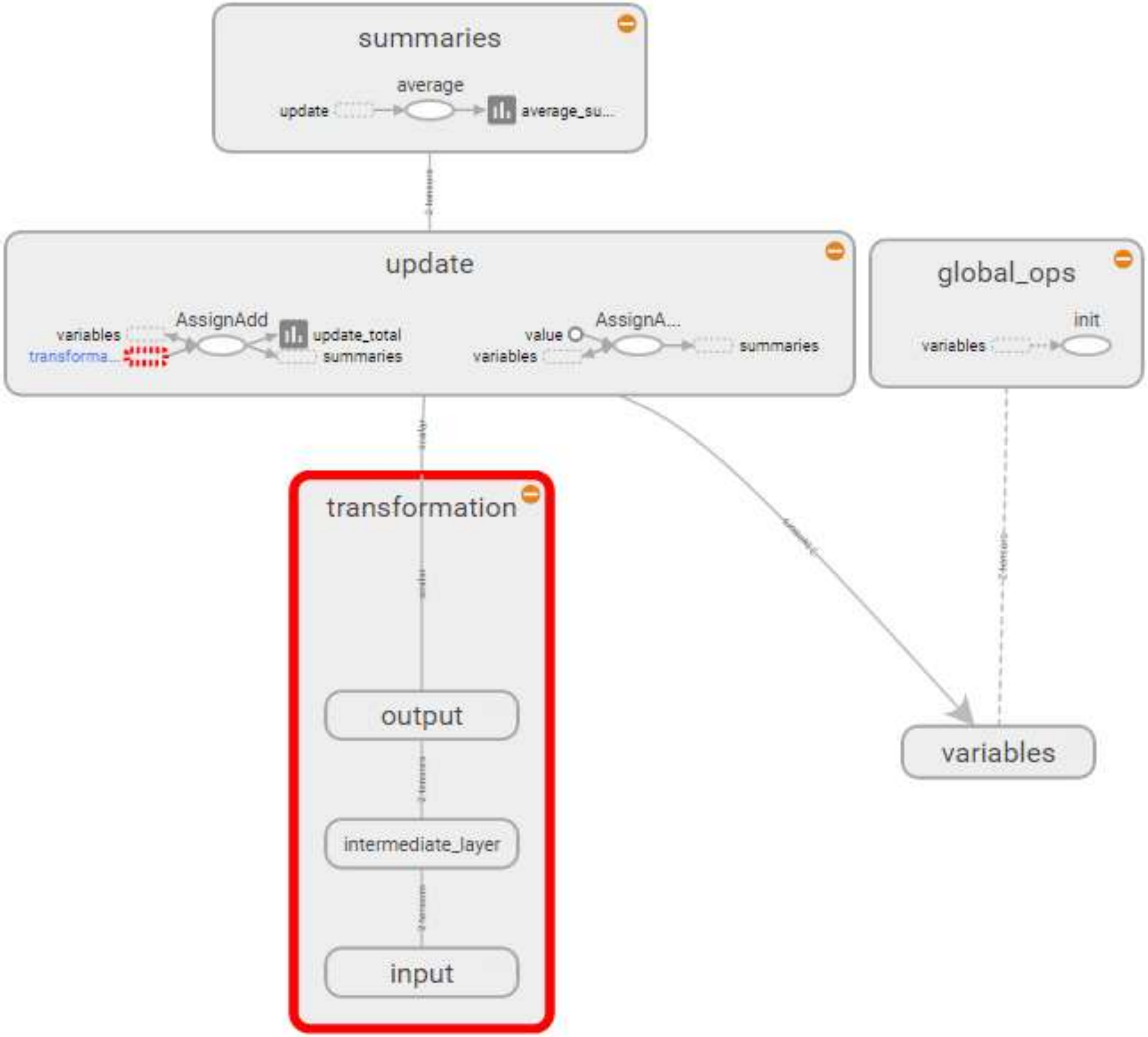


Main Graph
summaries
update
global_ops
transformation
variables



summaries
Main Graph
global_ops
init
variables
update
variables
transformation
global_step
total_output

Partially Expanded Graph



Partially Expanded Graph