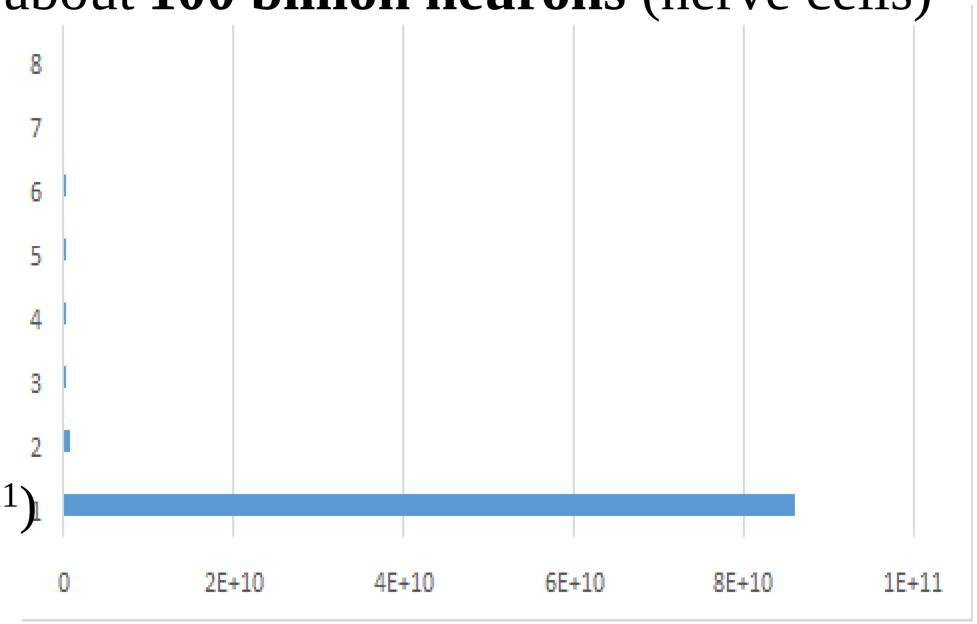
Northeastern University

INFO 6105
Data Science Eng Methods and Tools

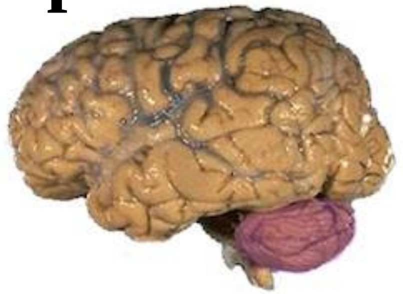
ML Lab with Keras®

Cells

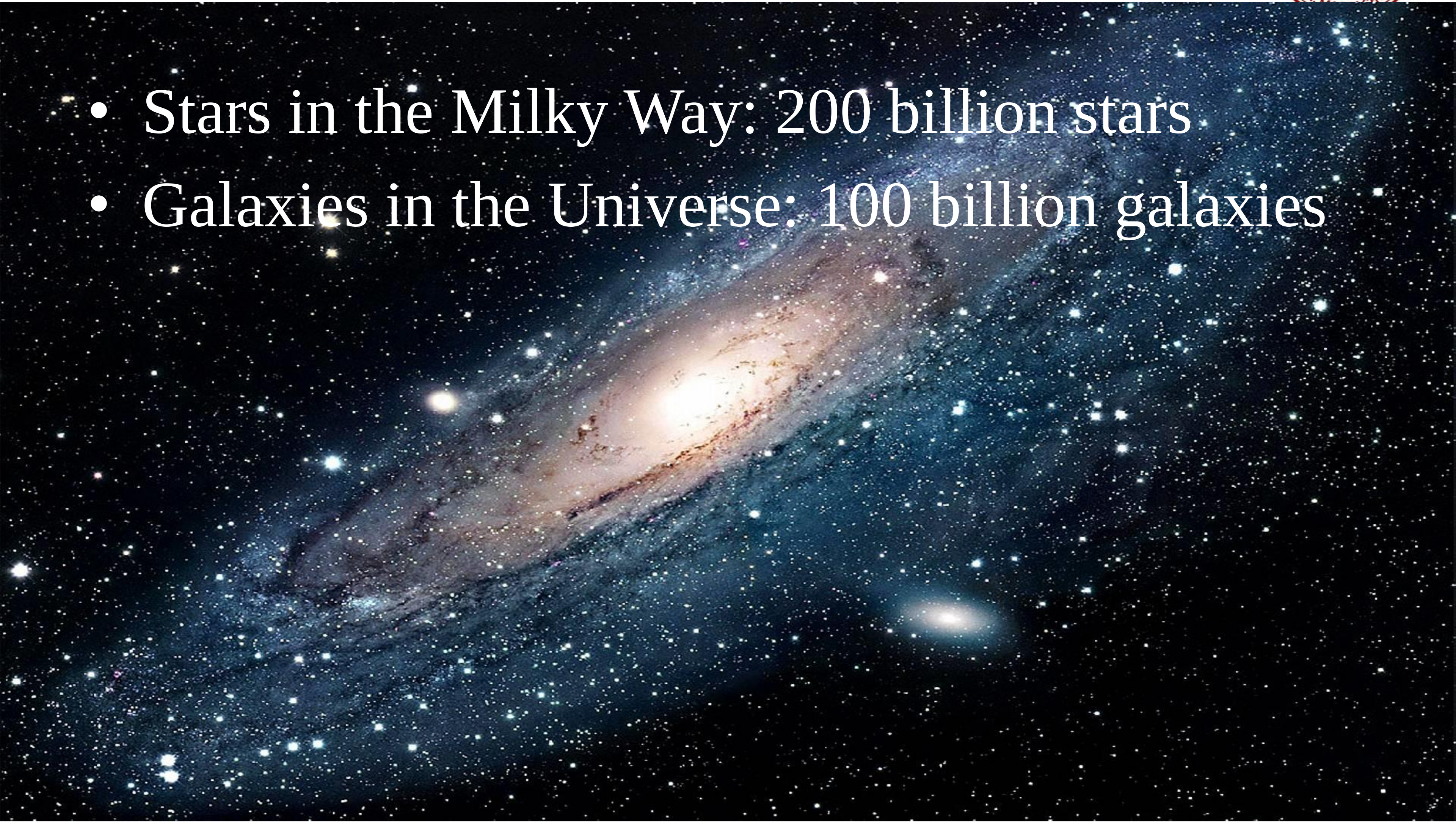
- The average human brain has about **100 billion neurons** (nerve cells)
  - Jellyfish: 800
  - Snail: 11,000
  - Fruit Fly: 100,000
  - Ant: 250,000
  - Honey Bee: 960,000
  - Rat: 200,000,000
  - Cat: 760,000,000
  - Human: 86,000,000,000 (10^{11})



- The average human brain has about **10^{15} synapses**
  - Rat: 10^{11}
  - Cat: 10^{13}
  - Human: 10^{15}



Stars



- Stars in the Milky Way: 200 billion stars
- Galaxies in the Universe: 100 billion galaxies

Of Humans, Galaxies and Atoms

- Atoms in the Universe: ~10^{80}
  - Only accounts only for the observable universe which reaches 46 billion light years in any direction, and is based on where the expansion of space has taken the most distant objects observed
  - Within this observable universe, this matter is spread homogeneously throughout space, at least when averaged over distances longer than 300 million light-years
  - On smaller scales, however, matter is observed to form into the clumps of hierarchically-organized luminous matter that we are all familiar with
  - Most atoms are condensed into stars, most stars are condensed into galaxies, most galaxies into clusters, most clusters into superclusters and, finally, into the largest-scale structures like the *Great Wall of galaxies*

Big Numbers

Million (Mega): 10^6 ← *A book of zeroes* (400 pages with 50 lines per page and 50 zeroes per line)

- Billion (Giga): 10^9 ← *Neurons in a human brain/ Stars in the Milky Way*
- Trillion (Tera): 10^{12} ← *How many \$ americans owe*

10^{11}

- Quadrillion (Peta): 10^{15}
- Quintillion (Exa): 10^{18} ← *Human neurons on Earth*
- Sextillion (Zeta): 10^{21} ← *Stars in the Universe*
- Septillion (Yotta): 10^{24}
- Googol: 10^{100} ← *Atoms in the Universe* ← *10,001st Fibonacci number*
- Googolplex: 10^{googol}

TOOLS FOR ML: THE GPU

GPU

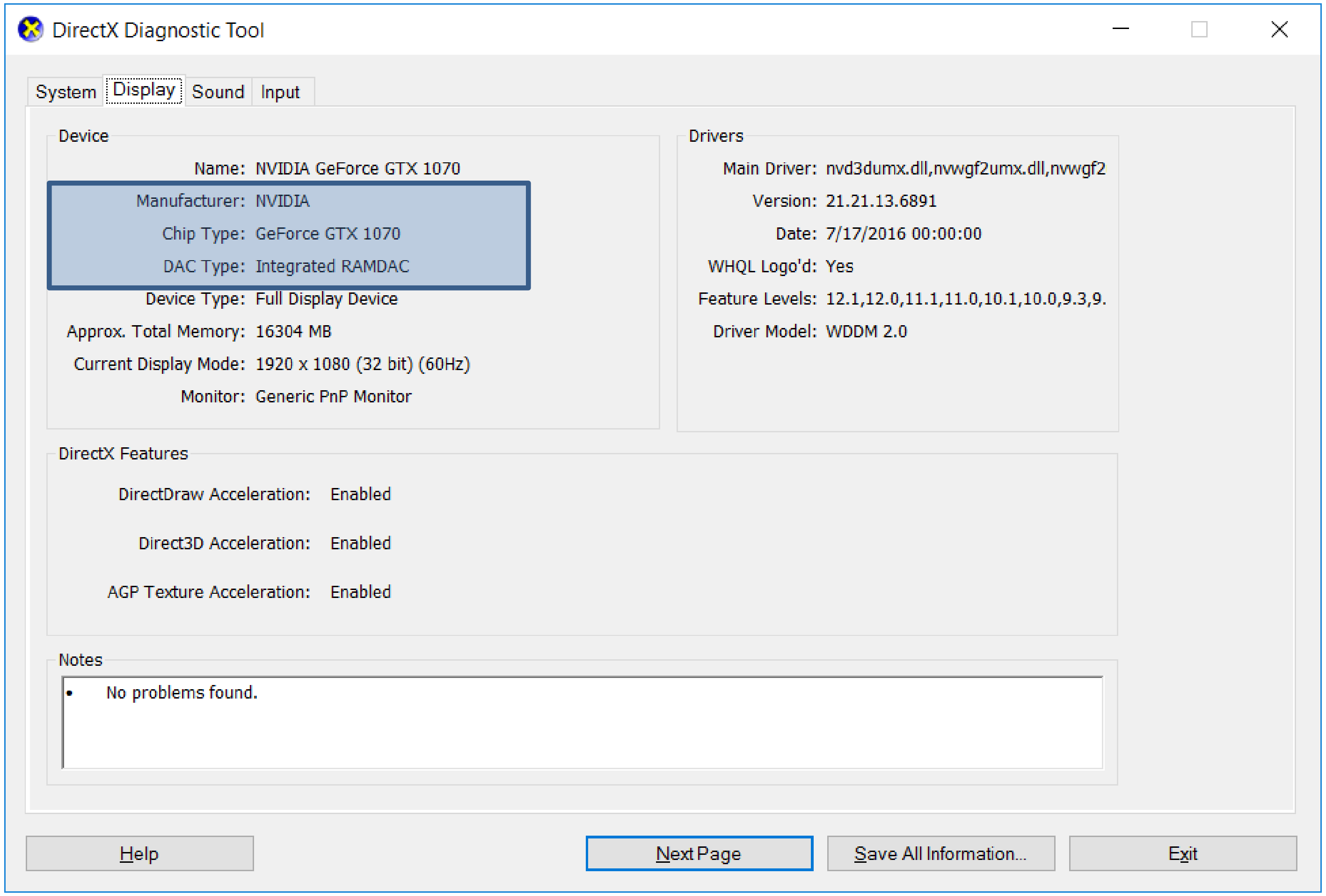
- A **graphics processing unit** (**GPU**) is a specialized chip designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display
- Modern GPUs are very efficient at manipulating computer graphics and image processing, and their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel
- 5 years ago, most Big Data processing was done with disk-based frameworks like Hadoop. Now, most Big Data processing and especially regression analysis is performed with GPUs
  - If you want to specialize in Big Data, the best investment for you would be to get a new laptop with a GPU
  - *Asus*, *MSI*, and *AlienWare* are some of the best 3rd party laptops w/GPU

OSX: Does your computer have GPU?

- Under the Apple menu select About This Mac, click the More Info … button, and then select Graphics/Displays under the Hardware list
- There you will find the vendor name and model of your graphics card
- If it is an NVIDIA card that is listed on the CUDA-supported GPUs page, your GPU is CUDA-capable
  - https://developer.nvidia.com/cuda-gpus

Windows: Does your computer have GPU?

Type in Cortana box: **`dxdiag`**



DirectX Diagnostic Tool

System | Display | Sound | Input

Device

Name:	NVIDIA GeForce GTX 1070
Manufacturer:	NVIDIA
Chip Type:	GeForce GTX 1070
DAC Type:	Integrated RAMDAC
Device Type:	Full Display Device
Approx. Total Memory:	16304 MB
Current Display Mode:	1920 x 1080 (32 bit) (60Hz)
Monitor:	Generic PnP Monitor

Drivers

Main Driver:	nvd3dumx.dll,nvwgf2umx.dll,nvwgf2
Version:	21.21.13.6891
Date:	7/17/2016 00:00:00
WHQL Logo'd:	Yes
Feature Levels:	12.1,12.0,11.1,11.0,10.1,10.0,9.3,9.
Driver Model:	WDDM 2.0

DirectX Features

DirectDraw Acceleration:	Enabled
Direct3D Acceleration:	Enabled
AGP Texture Acceleration:	Enabled

Notes

- No problems found.

What if you don't have a GPU?

- Amazon offers an EC2 instance that provides access to the GPU for computation purposes
- This instance is named the **g2.2xlarge instance** and costs approximately *$0.65 per hour*
  - The GPU included on the system is a K520 with 4GB of memory and 1,536 cores
- You can also upgrade to the **g2.8xlarge instance** (*$2.60 per hour*) to obtain *four* K520 GPUs (for a grand total of 16GB of memory)
- The g2.8xlarge is a bit expensive, especially if you're only doing deep learning for a class or as a hobby
- On the other hand, the g2.2xlarge instance is a totally reasonable option!
- http://www.pyimagesearch.com/2014/10/13/deep-learning-amazon-ec2-gpu-python-nolearn/
- http://markus.com/install-theano-on-aws/

Nvidia

- Nvidia was first to produce a chip capable of programmable shading, the *GeForce 3*
  - Each pixel could now be processed by a short program that could include additional image textures as inputs, and each geometric vertex could likewise be processed by a short program before it was projected onto the screen
  - Used in the Xbox console, it competed with the PlayStation 2 (which used a custom vector DSP for hardware accelerated vertex processing)
- By October 2002, with the introduction of the ATI *Radeon 9700* (also known as R300), the world's first Direct3D 9.0 accelerator, pixel and vertex shaders could implement looping and lengthy floating point math, and were quickly becoming as flexible as CPUs, yet orders of magnitude faster for image-array operations
- Nvidia's ***CUDA*** platform, first introduced in 2007, was the earliest widely adopted programming model for GPU computing
- More recently ***OpenCL*** has become broadly supported
  - An open standard defined by the Khronos Group which allows for the development of code for both GPUs and CPUs with an emphasis on portability
  - OpenCL solutions are supported by Intel, AMD, Nvidia, and ARM, and according to a recent report by Evan's Data, OpenCL is the GPGPU development platform most widely used by developers in both the US and Asia Pacific
- CUDA is specifically for NVIDIA GPUs while OpenCL is designed to work across a multitude of architectures including GPU, CPU and DSP

GPGPUs

- It is becoming increasingly common to use a general purpose graphics processing unit (GPGPU) as a modified form of stream processor (or vector processor), running compute kernels
- This concept turns the massive computational power of a modern graphics accelerator's shader pipeline into general-purpose computing power, as opposed to being hard wired solely to do graphical operations
- What *we* do, with deep learning!

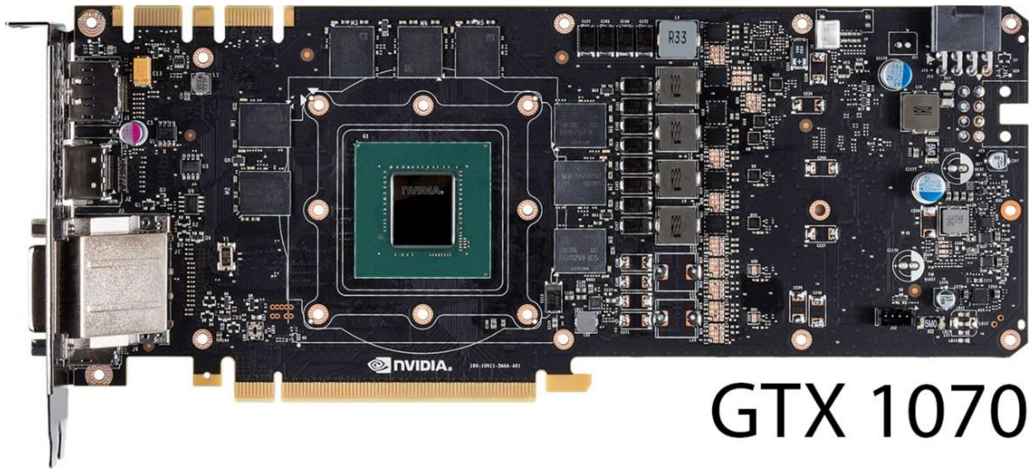
Nvidia GPUs

- ***Kepler*** line of GPUs was followed by
- ***Maxwell*** line, manufactured on the same process
  - 28 nm chips by Nvidia were manufactured by TSMC, the Taiwan Semiconductor Manufacturing Company
- ***Pascal*** was the next generation, released in 2016
  - The GeForce 10 series of cards are under this generation of graphics cards
  - They are made using the 16 nm manufacturing process
- ***Turing*** is the newest generation of graphics cards

Comparison, Nvidia Pascal GPUs

- **Nvidia TITAN X**: $700
  - 11 TFLOPS of parallel computing performance
  - 3584 NVIDIA CUDA cores and 12GB GDDR5X memory
- **Nvidia GTX 1080**: $500
  - 8873 GFLOPS, 7.2 billion transistors
  - 2560 NVIDIA CUDA Cores and 8GB GDDR5X
- **Nvidia GTX 1070**: $350
  - 1920 CUDA cores and 8GB GDDR5X

Nvidia GeForce GTX 1070 & 1080



Almost the same..

http://www.geforce.com/hardware/10series/architecture

Nvidia GeForce GTX 1070

- Looks almost exactly like the 1080
- Borrows *lots* from its big brother, electrically & mechanically
  - GeForce GTX 1080 comes equipped with a full GP104 die (measures ~ 300mm^2, features 8 Gbps GDDR5 Memory) sporting 20 Streaming Multiprocessors across four Graphics Processing Clusters
  - GEForce GTX 1070 sheds a complete GPC, losing five SMs in the process
    - That leaves it with 15 SMs, or 1920 CUDA cores (vs. 2560) and 120 texture units (vs. 160)
  - Nvidia further detunes GTX 1070 by dialing the GPU’s base frequency to 1506MHz and its specified GPU Boost clock rate to 1683MHz (1607MHz and 1733MHz, respectively, in the GTX 1080)

Nvidia GeForce GTX 1070 Back End

- Eight 32-bit memory controllers with eight *render output units* (ROP) and 256KB of L2 cache bound to each
  - In total, that's 64 ROPs and 2MB of L2
  - Whereas the GeForce GTX 1080 sports 8GB of 10 Gb/s GDDR5X, 1070 gets 8GB of 8 Gb/s GDDR5 from Samsung
  - Note: ROPs are pixel pipelines that take *pixel* and *texel* (texture elmnt.) information and process it, via specific matrix and vector operations, into a final pixel or depth value. This process is called *rasterization*
- Memory bandwidth peaks at 256 GB/s
  - The GeForce GTX 980 Ti and Titan X actually benefit from more throughput than the 1070 due to their 384-bit interfaces (as do several AMD cards with 384- and 512-bit buses)
  - However, Nvidia maintains that the improved delta color compression in GTX 1070/1080 yields 20% more effective bandwidth by reducing bytes fetched

Under the shroud

- Whereas the 1080 employs a *vapor chamber* solution, 1070 sports an aluminum heat sink with three embedded copper heat pipes
  - Cost-cutting measure related to 1070’s 150W TDP
  - A lower-power card doesn’t *need* such a beefy cooler, even if it would undoubtedly help GeForce GTX 1070 overcome some of the thermal limits we saw the 1080 hit

Software for Directly Programming a GPU

- **CUDA**: GPU programming API by NVIDIA based on extension to C (CUDA C)
  - Vendor-specific
  - Numeric libraries (BLAS, RNG, FFT) are maturing.
- **OpenCL**: multi-vendor version of CUDA
  - More general, standardized.
  - Fewer libraries, lesser spread.
- **PyCUDA**: Python bindings to CUDA driver interface allow to access Nvidia's CUDA parallel computation API from Python
  - Convenience:
  - Makes it easy to do GPU meta-programming from within Python.
  - Abstractions to compile low-level CUDA code from Python (pycuda.driver.SourceModule).
  - GPU memory buffer (pycuda.gpuarray.GPUArray)
  - Automatic error checking: All CUDA errors are automatically translated into Python exceptions.
  - Speed: PyCUDA's base layer is written in C++
  - Good memory management of GPU objects: Object cleanup tied to lifetime of objects (RAII, 'Resource Acquisition Is Initialization')
  - Makes it easier to write correct, leak- and crash-free code
- **PyOpenCL**: PyCUDA for OpenCL

Compute Unified Device Architecture (CUDA)

- **CUDA** is a parallel computing platform and API model created by Nvidia
- It allows software developers to use a CUDA-enabled GPU for general purpose processing
  - Like Nvidia 1070, 1080, or TitanX
- **CUDA** *supports* programming frameworks:
  - **OpenACC** (http://www.openacc.org/)
    - Compiler directives to specify loops and regions of code in standard C, C++ to be offloaded from a host CPU to an attached GPU
  - **OpenCL** (https://www.khronos.org/opencl/)
    - Framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs, field-programmable gate arrays (FPGAs) and other hardware accelerators

What can be accelerated with a GPU

- Only computations with *float32* data-type can be accelerated
  - Better support for *float64* is expected in upcoming hardware but *float64* computations are still relatively slow (Jan 2010).
- Matrix multiplication, convolution, and large element-wise operations can be accelerated a lot (5-50x) when arguments are large enough to keep 30 processors busy
- Indexing, dimension-shuffling and constant-time reshaping will be equally fast on GPU as on CPU
- Summation over rows/columns of tensors can be a little slower on the GPU than on the CPU
- Copying of large quantities of data to and from a device is relatively slow, and often cancels most of the advantage of one or two accelerated functions on that data. Getting GPU performance largely hinges on making data transfer to the device pay off

Popular DL Frameworks

- BidMach
- Blocks
- **Caffe** ← UCal Berkeley
- Chainer
- **CNTK** ← Microsoft
- cuda-convnet
- cuda-convnet2
- Deeplearning4j
- Kaldi
- **Keras** ← Chollet, (uses Theano or TensorFlow) user friendly
- **Lasagne** ← Dieleman, (uses Theano)
- Marvin
- MatConvNet
- **Mxnet** ← Amazon high performance
- **TensorFlow** ← Google big user community
- **Theano** ← UMontreal
- **Torch , PyTorch** ← Facebook research

Torch

- Created/Used by NYU, Facebook, Google DeepMind
- De rigueur for deep learning research
- Its language is Lua, NOT Python
  - Lua's syntax is somewhat Pythonic
  - Arguably, *most popular* ML framework today in university ML research
- Torch's main strengths are its features
- http://torch.ch/
- https://github.com/torch/torch7
- http://bit.ly/1KzuFhd

Caffe

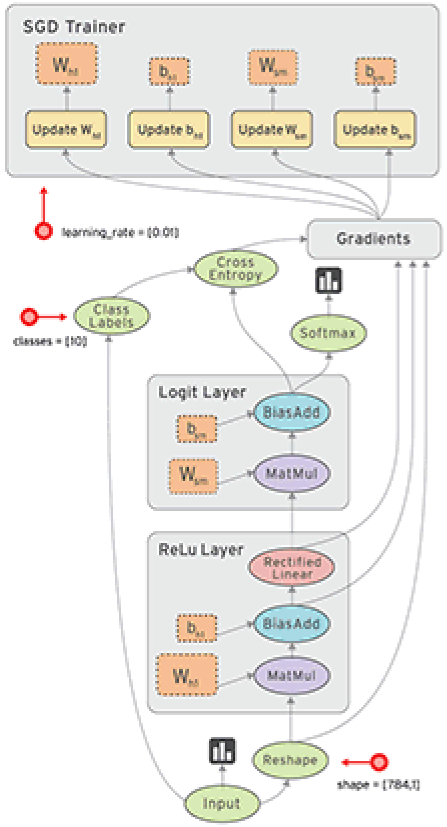
- Created/Used by Berkeley, Google
- Best tool to get started with:
  - Lots of pre-trained reference models
  - Lots of standard deep learning datasets
  - Easy to configure networks with config files
  - Not really free..
- http://bit.ly/1Db2bHT

GraphLab-Create

- Created by *Dato*, *Inc*.
- User friendly, picks intelligent defaults
- Tons of features
- Blazing fast out-of-core computations on small/medium/big data
- Pythonic API, great documentation
- http://bit.ly/1LZVqLS

TensorFlow

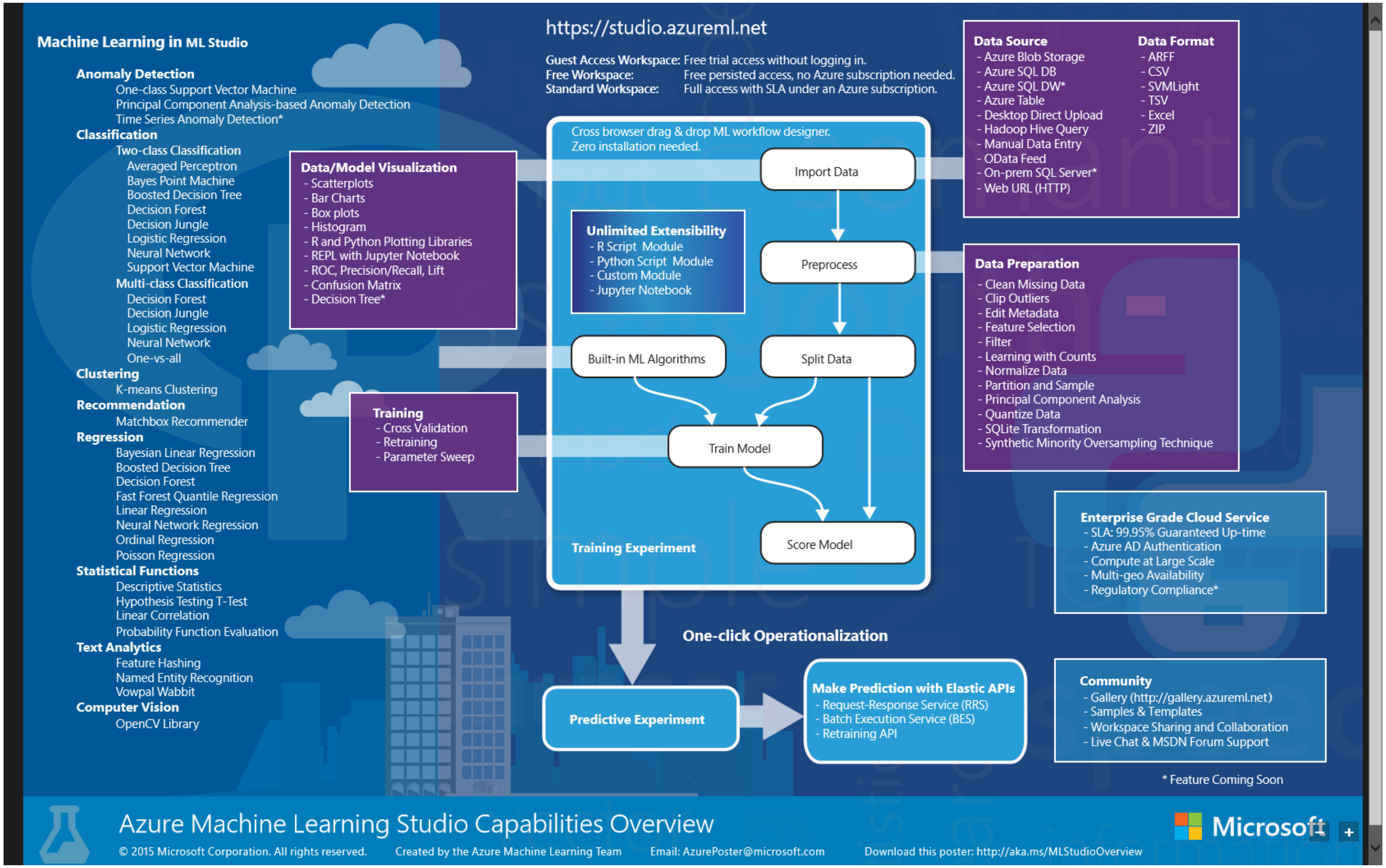
- Open source software library for numerical computation using data flow graphs, *from Google*
  - Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them (like AzureML)
  - We will revisit this concept when we study Monads
  - Flexible architecture allows computation to deploy to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API (like hadoop)
  - Originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization
  - https://www.tensorflow.org/
  - http://googleresearch.blogspot.com/2015/11/tensorflow-googles-latest-machine_9.html



Azure ML

- ***Visual*** IDE to develop a predictive analytics model and operationalize it on the Azure cloud
  - Number of machine learning algorithms available, along with modules that help with data input, output, preparation, and visualization
  - Using these components one can develop a predictive analytics experiment, iterate on it, and use it to train your model
  - Then with one click you can operationalize your model in the Azure cloud so that it can be used to score new data
- https://studio.azureml.net/
- We may study this when we safari on Azure

Azure ML



Machine Learning in ML Studio

Anomaly Detection
- One-class Support Vector Machine
- Principal Component Analysis-based Anomaly Detection
- Time Series Anomaly Detection*

Classification
- Two-class Classification
  - Averaged Perceptron
  - Bayes Point Machine
  - Boosted Decision Tree
  - Decision Forest
  - Decision Jungle
  - Logistic Regression
  - Neural Network
  - Support Vector Machine
- Multi-class Classification
  - Decision Forest
  - Decision Jungle
  - Logistic Regression
  - Neural Network
  - One-vs-all

Clustering
- K-means Clustering

Recommendation
- Matchbox Recommender

Regression
- Bayesian Linear Regression
- Boosted Decision Tree
- Decision Forest
- Fast Forest Quantile Regression
- Linear Regression
- Neural Network Regression
- Ordinal Regression
- Poisson Regression

Statistical Functions
- Descriptive Statistics
- Hypothesis Testing T-Test
- Linear Correlation
- Probability Function Evaluation

Text Analytics
- Feature Hashing
- Named Entity Recognition
- Vowpal Wabbit

Computer Vision
- OpenCV Library

Data/Model Visualization
- Scatterplots
- Bar Charts
- Box plots
- Histogram
- R and Python Plotting Libraries
- REPL with Jupyter Notebook
- ROC, Precision/Recall, Lift
- Confusion Matrix
- Decision Tree*

Training
- Cross Validation
- Retraining
- Parameter Sweep

https://studio.azureml.net

Guest Access Workspace:	Free trial access without logging in.
Free Workspace:	Free persisted access, no Azure subscription needed.
Standard Workspace:	Full access with SLA under an Azure subscription.

Cross browser drag & drop ML workflow designer.
Zero installation needed.

Unlimited Extensibility
- R Script Module
- Python Script Module
- Custom Module
- Jupyter Notebook

Data Source
- Azure Blob Storage
- Azure SQL DB
- Azure SQL DW*
- Azure Table
- Desktop Direct Upload
- Hadoop Hive Query
- Manual Data Entry
- OData Feed
- On-prem SQL Server*
- Web URL (HTTP)

Data Format
- ARFF
- CSV
- SVMLight
- TSV
- Excel
- ZIP

Data Preparation
- Clean Missing Data
- Clip Outliers
- Edit Metadata
- Feature Selection
- Filter
- Learning with Counts
- Normalize Data
- Partition and Sample
- Principal Component Analysis
- Quantize Data
- SQLite Transformation
- Synthetic Minority Oversampling Technique

Enterprise Grade Cloud Service
- SLA: 99.95% Guaranteed Up-time
- Azure AD Authentication
- Compute at Large Scale
- Multi-geo Availability
- Regulatory Compliance*

Make Prediction with Elastic APIs
- Request-Response Service (RRS)
- Batch Execution Service (BES)
- Retraining API

Community
- Gallery (http://gallery.azureml.net)
- Samples & Templates
- Workspace Sharing and Collaboration
- Live Chat & MSDN Forum Support

* Feature Coming Soon

Azure Machine Learning Studio Capabilities Overview

© 2015 Microsoft Corporation. All rights reserved. Created by the Azure Machine Learning Team Email: AzurePoster@microsoft.com Download this poster: http://aka.ms/MLStudioOverview

Microsoft

CMTK/DMTK

- Open source distributed Machine Learning library from *Microsoft Research*
  - Framework for training models on multiple servers, topic modeling algorithm, and word-embedding algorithm for natural language processing
  - http://www.dmtk.io/
  - https://github.com/Microsoft/DMTK

Theano

- A numerical computation library for Python
  - Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently
  - In **Theano**, computations are expressed using a NumPy-like syntax and compiled to run efficiently on either CPU or GPU architectures
  - **Theano** is an open source project primarily developed by a machine learning group at the *Université de Montréal*
  - http://deeplearning.net/software/theano/
- http://www.mila.umontreal.ca/Home

Keras

- Minimalist, modular Neural Networks library, written in Python and capable of running on top of either *TensorFlow* or *Theano*
  – Developed with a focus on going from idea to result with the least possible delay
  – Fast prototyping (modularity, minimalism, extensibility)
  – Supports both convolutional networks and recurrent networks, as well as combinations
  – Initially developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System)
  – Very user friendly
  – *Now official API of Tensorflow 2.0!*

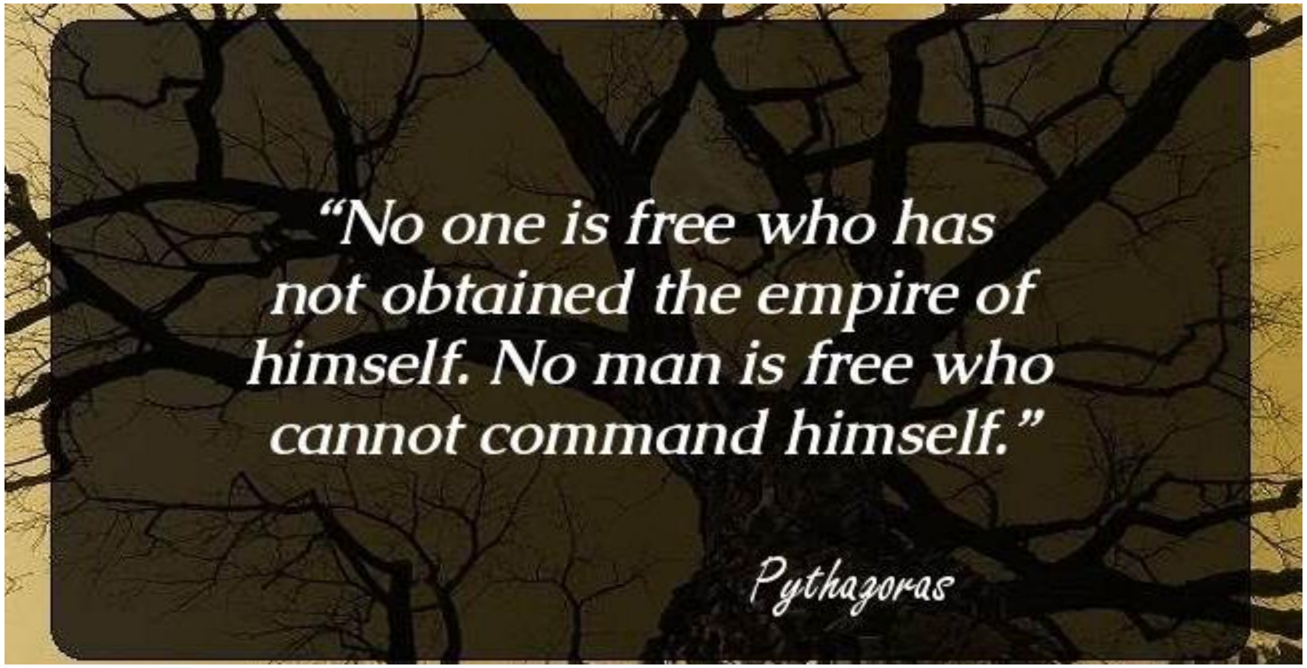
Yoshua Bengio

- Université de Montreal

- Theano
  - – http://deeplearning.net/software/theano/install.html

Theano

- Theano (Θεανώ; 6th-century BC) or *Theano of Croton*, Pythagorean philosopher and wife of Pythagoras

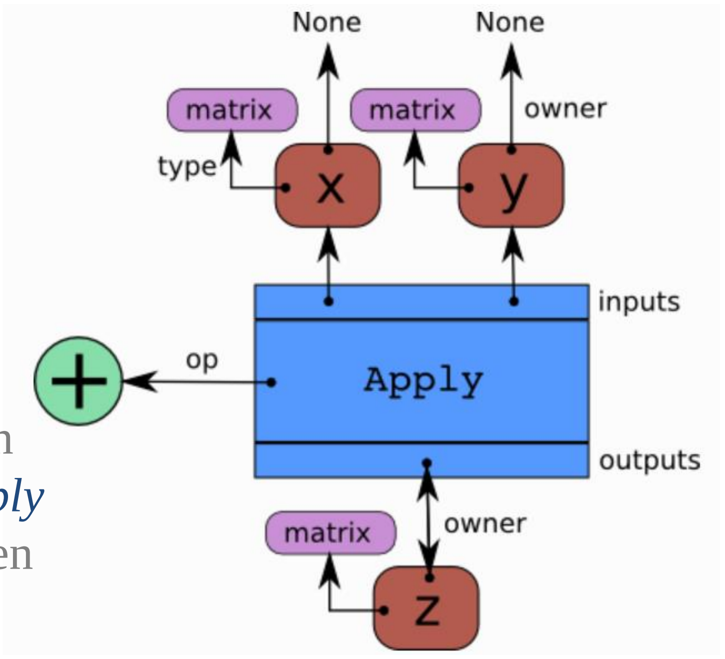

"No one is free who has not obtained the empire of himself. No man is free who cannot command himself."

Pythagoras

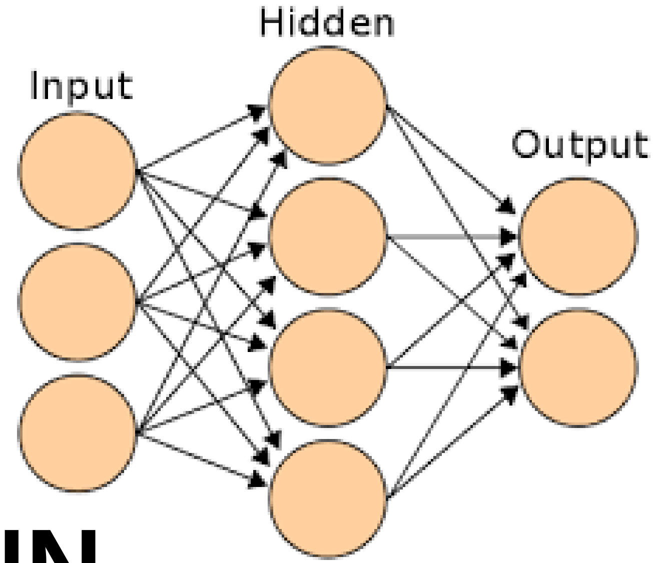
Understanding Theano

- Theano represents symbolic mathematical computations as *graphs*
  - graphs are composed of interconnected *Apply*, *Variable* and *Op* nodes
  - *Apply* node represents the application of an *op* to some *variables*

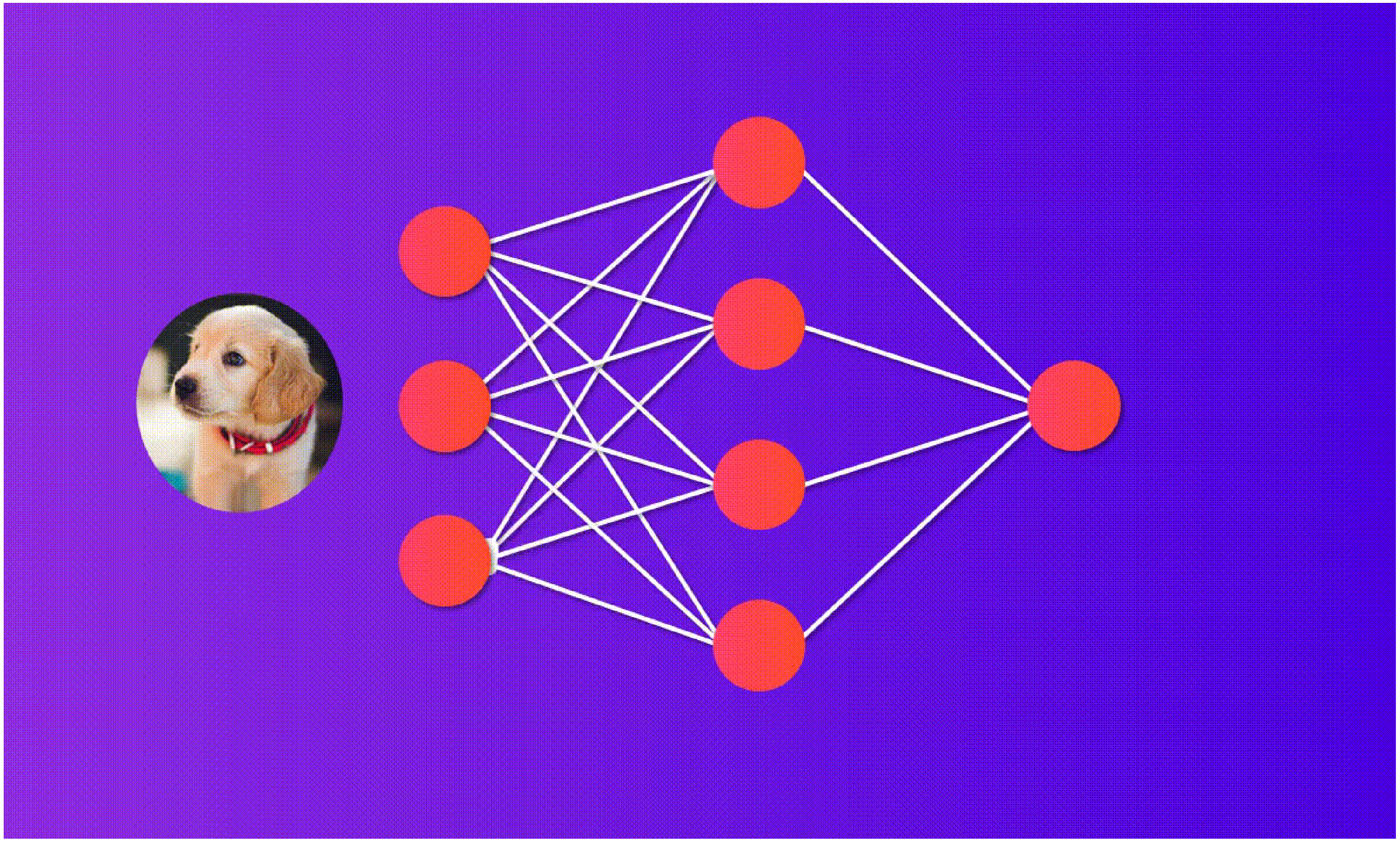
```python
import theano.tensor as T

x = T.dmatrix('x')
y = T.dmatrix('y')
z = x + y
```



Arrows represent references to the Python objects pointed at. The blue box is an *Apply* node. Red boxes are *Variable* nodes. Green circles are *Ops*. Purple boxes are *Types*



ANN

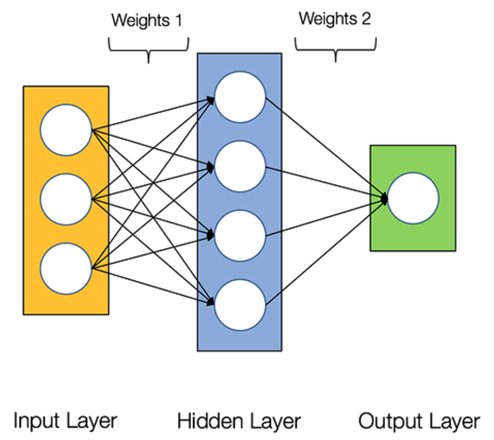


Neural Networks

- Neural Networks consist of the following components
  - An **input layer** accepting $\boldsymbol{x}$
  - An *arbitrary* amount of **hidden layers**
  - An **output layer** producing $\hat{\boldsymbol{y}}$
  - A set of **weights** and **biases** between each layer, ***W and b***
  - A choice of **activation function** for each hidden layer, $\boldsymbol{\sigma}$
    - We'll use a Sigmoid activation function
      - https://en.wikipedia.org/wiki/Sigmoid_function

Weights & Training

- Weights are the most important thing
  - Because you are essentially reducing a huge dataset of all trained inputs (going through the input layer)
  - To a set of weights of the neural network
- The weights of the neural network are essentially equivalent to all the data that you used to train the network
- Training the network is adjusting the weights



Neural Network Lab

- Create a new python notebook, and add the following code:

```
class NeuralNetwork:
    def __init__(self, x, y):
        self.input      = x
        self.weights1   = np.random.rand(self.input.shape[1],4)
        self.weights2   = np.random.rand(4,1)
        self.y          = y
        self.output     = np.zeros(y.shape)
```

Training the Neural Network

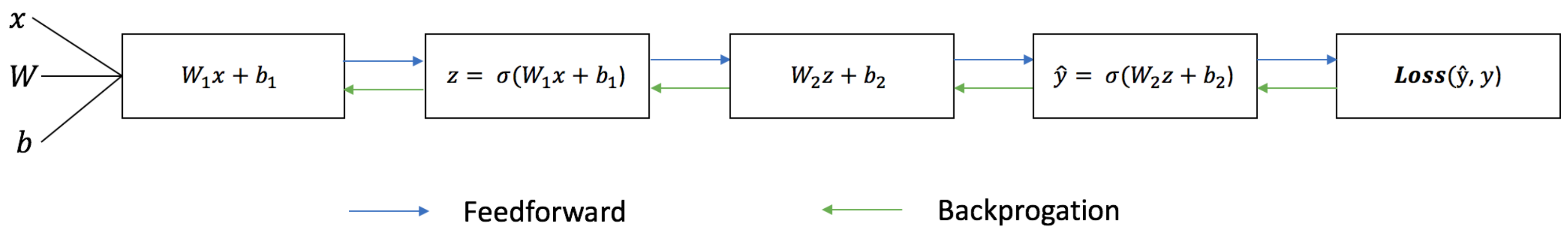
- The output $\hat{y}$ of a simple 2-layer Neural Network is:

$$\hat{y} = \sigma(W_2 \sigma(W_1 x + b_1) + b_2)$$

- Where W_1 is the matrix of weights between input ad hidden layer
- W_2 is the matrix of weights between hidden and output layer
- x is a 3D vector
- $\hat{y}$ is a scalar

Training process

- Each iteration of the training process consists of the following steps:
  - Calculating the predicted output $\hat{y}$, known as **feedforward**
  - Updating the weights and biases, known as **backpropagation**
- The sequential graph below illustrates the process



Feedforward

- Add a feedforward function in your python
- For simplicity, assume the biases to be 0

```
class NeuralNetwork:
    def __init__(self, x, y):
        self.input      = x
        self.weights1   = np.random.rand(self.input.shape[1],4)
        self.weights2   = np.random.rand(4,1)
        self.y          = y
        self.output     = np.zeros(self.y.shape)

    def feedforward(self):
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.output = sigmoid(np.dot(self.layer1, self.weights2))
```

Loss function

- We need a way to evaluate the “goodness” of our predictions (i.e. how far off are our predictions)?
- The **Loss Function** allows us to do that
- The sum-of-squares error is simply the sum of the difference between each predicted value and the actual value

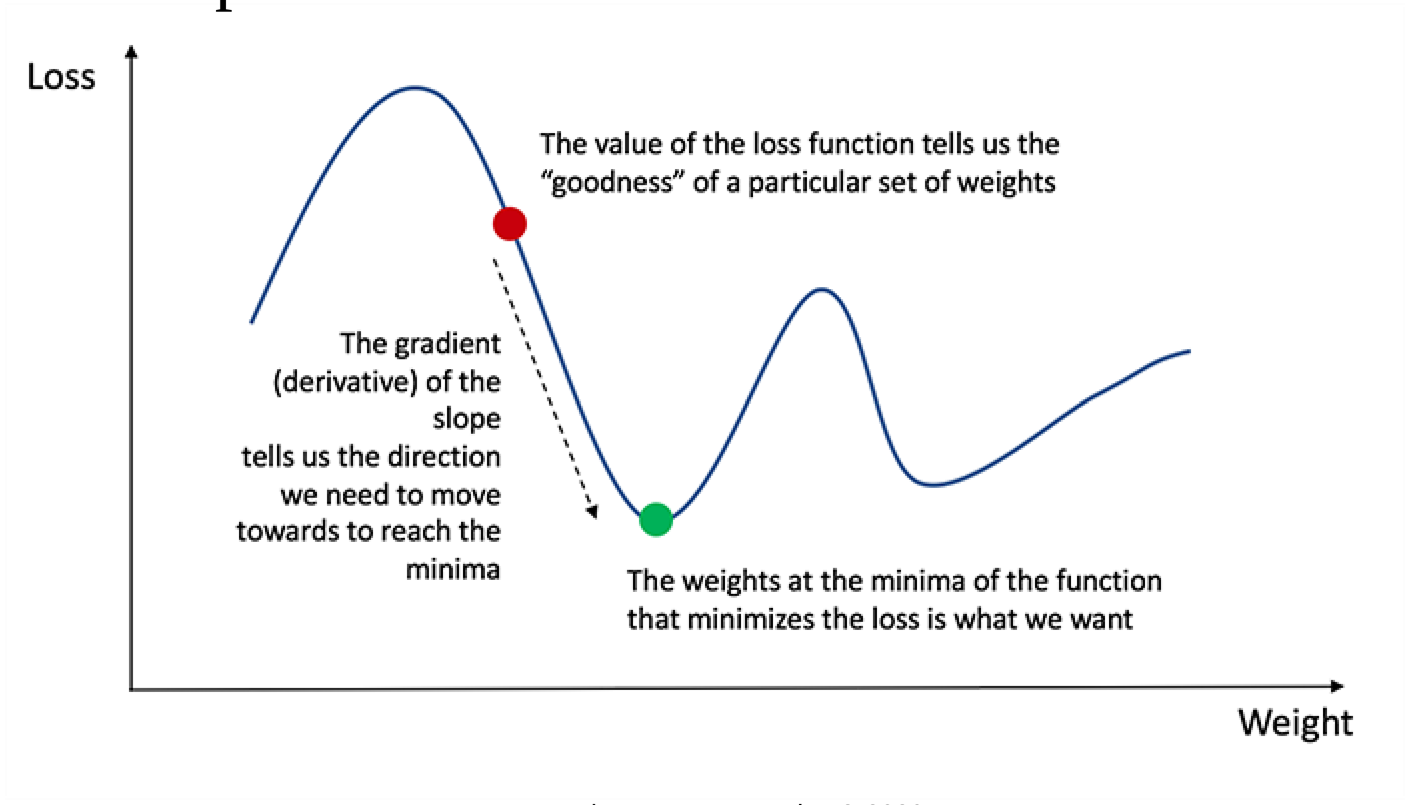
$$Sum-of-Squares\ Error = \sum_{i=1}^{n}(y - \hat{y})^2$$

Backpropagation

- Now that we've measured the error of our prediction (loss), we need to find a way to **propagate** the error back, and to update our weights and biases
- In order to know the appropriate amount to adjust the weights and biases by, we need to know the **derivative of the loss function with respect to the weights and biases**

Derivatives minimize a function

- The derivative of a function is simply the slope of the function
- The slope is 0 at the minima


Loss
The value of the loss function tells us the "goodness" of a particular set of weights
The gradient (derivative) of the slope tells us the direction we need to move towards to reach the minima
The weights at the minima of the function that minimizes the loss is what we want
Weight

Gradient descent

- If we have the derivative, we can simply update the weights (increase them or reduce them) in order to minimize the Loss function
- This is known as ***gradient descent***
- **Because the Loss function may be complex and have many minima, this is not as easy as it sounds**
- Think about coming down a mountain
  - There are many ways to go down in many dimensions. Which way to go?

Gradient Descent

Loss function derivative

- To calculate the derivative of the loss function with respect to the weights and biases, we need the **chain rule** to help us

$$Loss(y,\hat{y}) = \sum_{i=1}^{n}(y - \hat{y})^2$$

$$\frac{\partial\, Loss(y,\hat{y})}{\partial W} = \frac{\partial Loss(y,\hat{y})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W} \qquad \text{where } z = Wx + b$$

$$= 2(y - \hat{y}) * \text{derivative of sigmoid function} * x$$

$$= 2(y - \hat{y}) * \text{z(1-z)} * x$$

Not really correct, but that's what books show ya!

Backpropagation function

- Add the backpropagation function into your python code

```
 def backprop(self):
     # application of the chain rule to find derivative of the loss function with respect to weights2 and weights1
     d_weights2 = np.dot(self.layer1.T, (2*(self.y - self.output) * sigmoid_derivative(self.output)))
     d_weights1 = np.dot(self.input.T,  (np.dot(2*(self.y - self.output) * sigmoid_derivative(self.output), self.weights2.T) * 
sigmoid_derivative(self.layer1)))

     # update the weights with the derivative (slope) of the loss function
     self.weights1 += d_weights1
     self.weights2 += d_weights2
```

- For more on backpropagation
  - https://youtu.be/tIeHLnjs5U8

Putting it all together

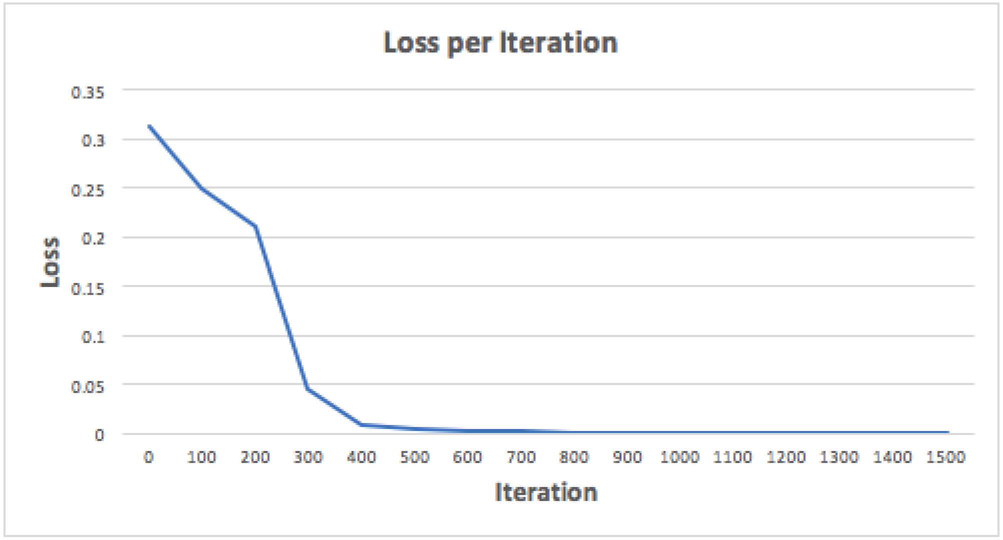
- Now that we have our complete python code for doing feedforward and backpropagation, let's apply our Neural Network on an example

X1	X2	X3	Y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

- Our Neural Network should learn the ideal set of weights to represent this function

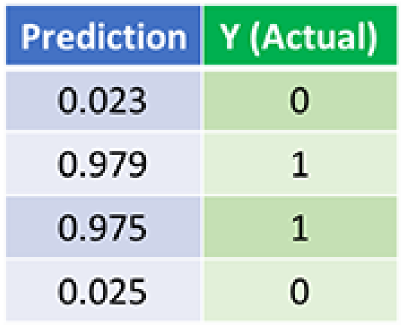
Goal

- Train the Neural Network for 1500 iterations and see what happens
- You should see the loss **monotonically decreasing towards a minimum**



Result

- Final prediction (output) from the Neural Network after 1500 iterations:



Prediction	Y (Actual)
0.023	0
0.979	1
0.975	1
0.025	0

- Predictions converged on the true values.
- Note that there's a slight difference between the predictions and the actual values
  - This is desirable, as it prevents **overfitting** and allows the Neural Network to **generalize** better to *unseen* data

Reference

- https://www.amazon.com/Neural-Network-Projects-Python-ultimate-ebook/dp/B07P77QWW7/

KERAS

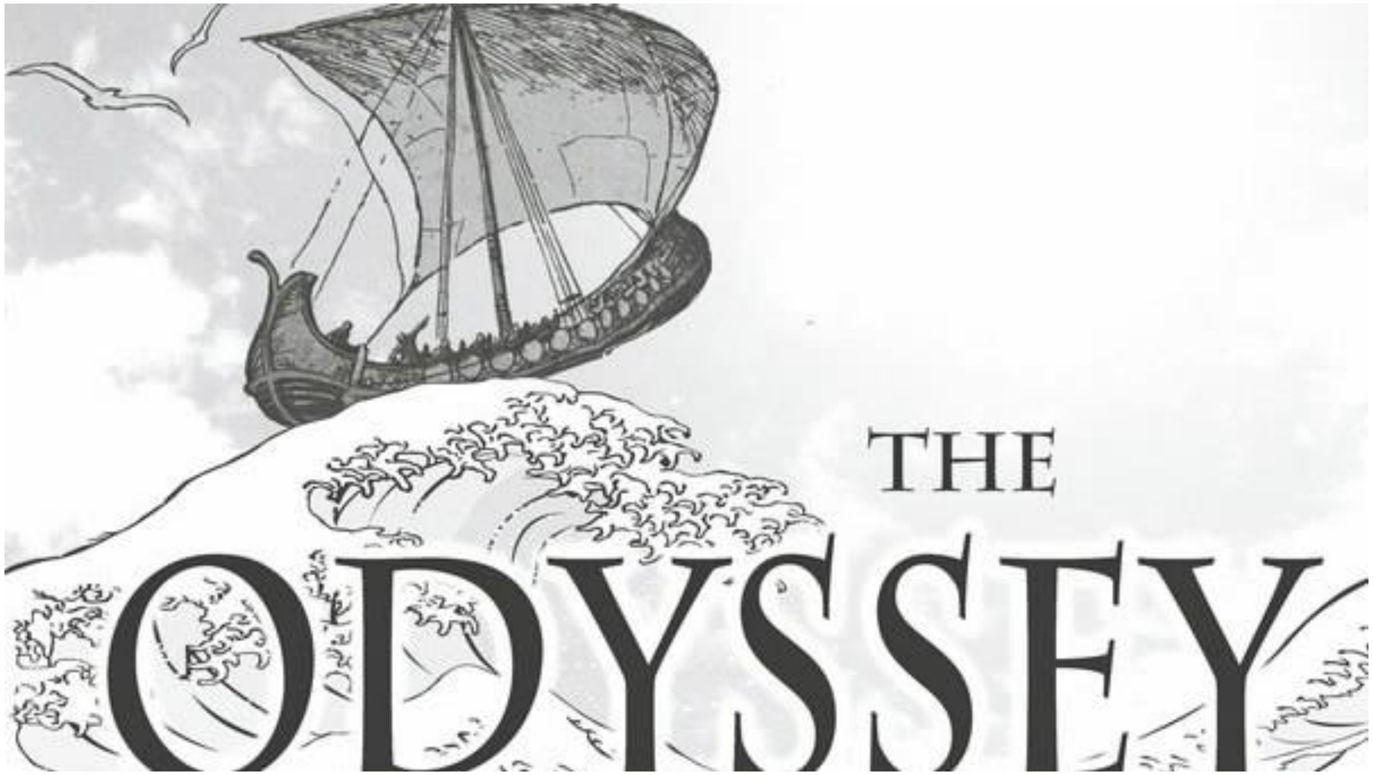
Keras

- Minimalist, modular Neural Networks library, written in Python and capable of running on top of either *TensorFlow* or *Theano*
  - Developed with a focus on going from idea to result with the least possible delay
  - Fast prototyping (modularity, minimalism, extensibility)
  - Supports both convolutional networks and recurrent networks, as well as combinations
  - Supports arbitrary connectivity schemes (including multi-input and multi-output training)
  - Runs seamlessly on CPU and GPU
  - Initially developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System)
  - http://keras.io/
  - http://keras.io/documentation/
  - http://robotfuture.net
  - https://github.com/fchollet/keras
- `pip install keras`

Keras

- Keras (κέρας) means *horn* in Greek
  - Reference to a literary image from ancient Greek and Latin literature, first found in the *Odyssey*, where dream spirits (*Oneiroi*, singular *Oneiros*) are divided between those who deceive with false visions and arrive to Earth through a gate of ivory (blue pill in the Matrix®), and those who announce a future that will come to pass and arrive through a gate of horn (red pill in the Matrix®)
  - Play on the greek words κέρας (horn) / κραίνω (fulfill), and ἐλέφας (ivory) / ἐλεφαίρομαι (deceive)

Homer’s Odyssey

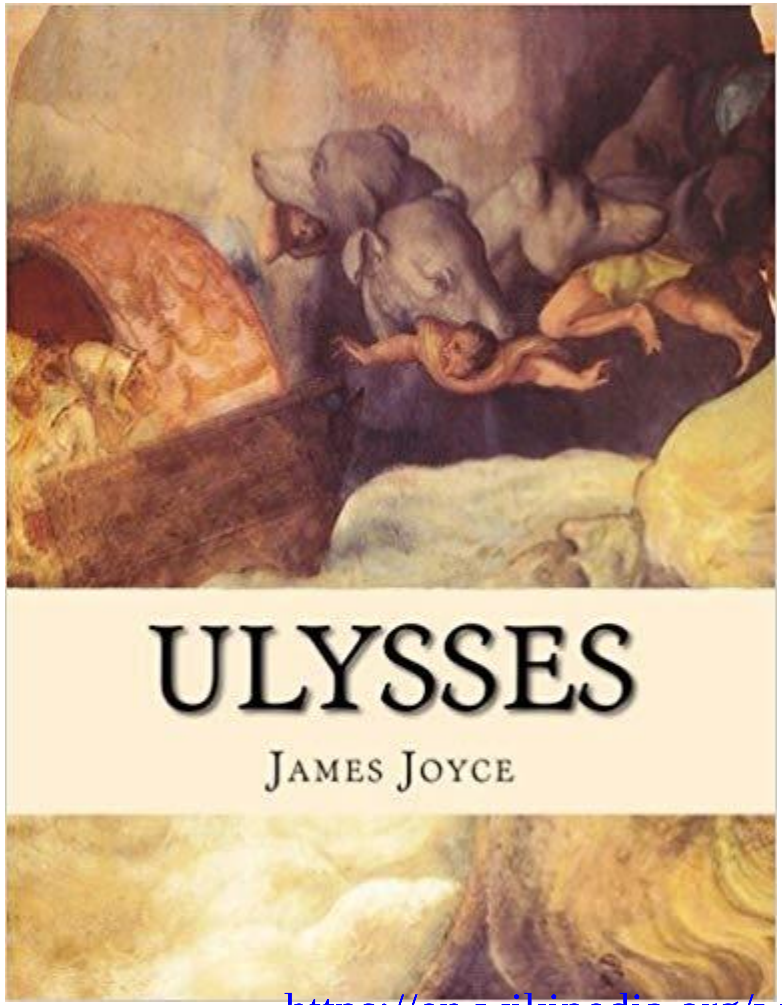


https://en.wikipedia.org/wiki/Odyssey

Ulysses

AMBRUS/AETHON

James Joyce's Ulysses

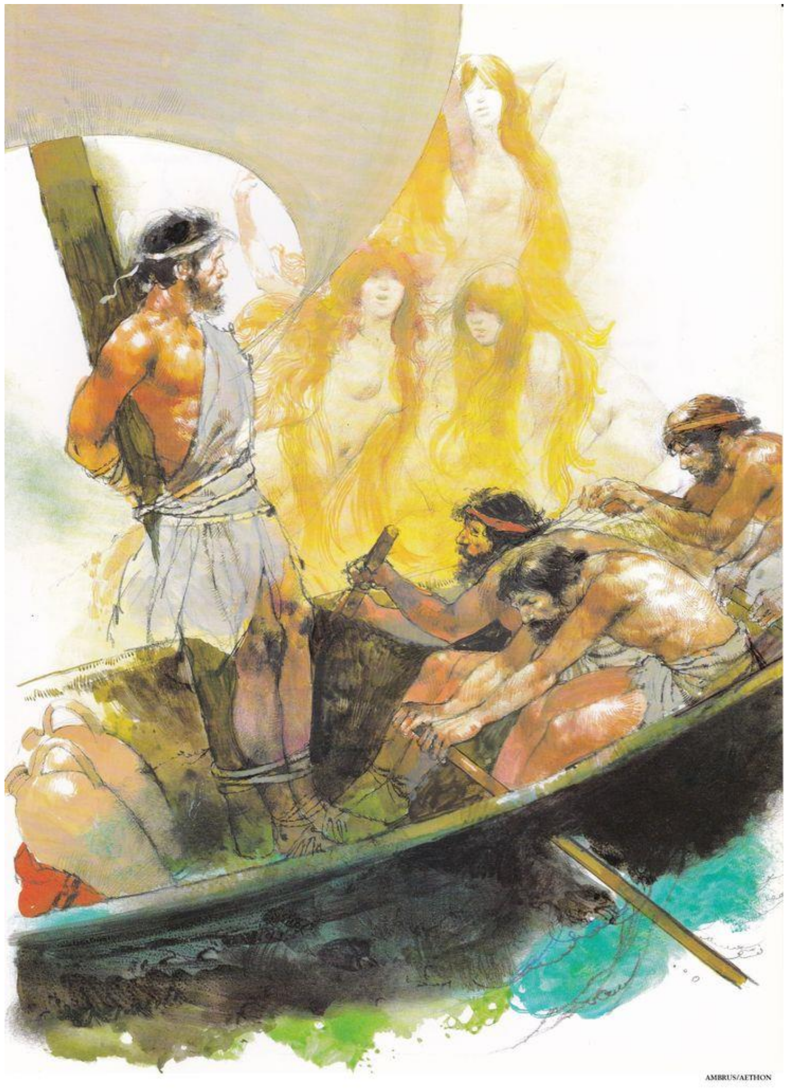


https://en.wikipedia.org/wiki/Ulysses_(novel)

Keras

- *"Oneiroi are beyond our unravelling --who can be sure what tale they tell? Not all that men look for comes to pass. Two gates there are that give passage to fleeting Oneiroi; one is made of horn, one of ivory. The Oneiroi that pass through sawn ivory are deceitful, bearing a message that will not be fulfilled; those that come out through polished horn have truth behind them, to be accomplished for men who see them".*
  - Homer, Odyssey 19 (Shewring translation)

Oneiroi



Behind Keras

- *François Chollet*
  - Author of Keras
  - Founder of Wysp, social network & learning platform for artists - wysp.ws
  - Recently launched QuickAnswers, natural language question-answering engine - quickanswers.io
- *Morgan Quigley*
  - https://www.technologyreview.com/lists/innovators-under-35/2013/inventor/morgan-quigley/

Sequential Model

- http://keras.io
- The Sequential model is a linear stack of layers
  - You create a Sequential model by passing a list of layer instances to the constructor:

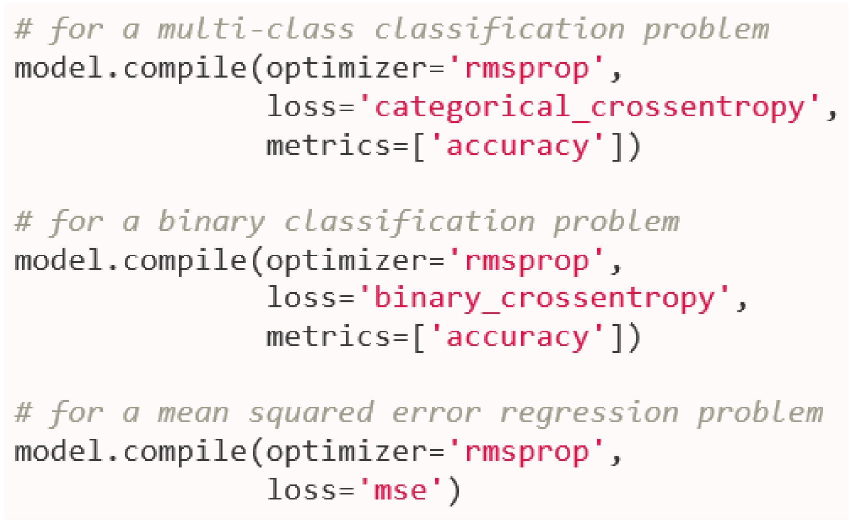
```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_dim=784),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

  - You can also simply add layers via the .add() method:

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

Compilation

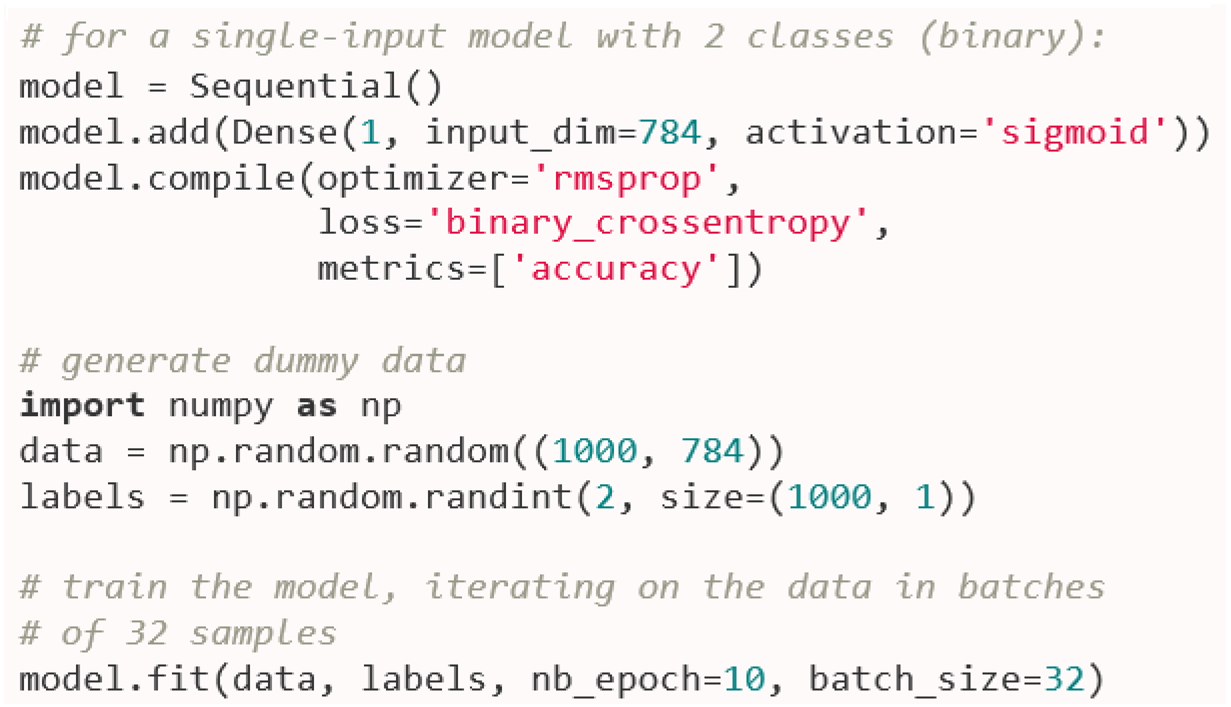
- Before training a model, you configure the learning process with **3** parameters:
  - Optimizer: This could be the string identifier of an existing optimizer (such as rmsprop or adagrad), or an instance of the Optimizer class
  - Loss function: This is the objective that the model will try to minimize. It can be the string identifier of an existing loss function (such as categorical_crossentropy or mse), or it can be an objective function
  - List of metrics: For any classification problem you will want to set this to metrics=['accuracy']
    - A metric could be the string identifier of an existing metric or a custom metric function



```
# for a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# for a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# for a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')
```

Training

- Keras models are trained on `NumPy` arrays of input data and labels

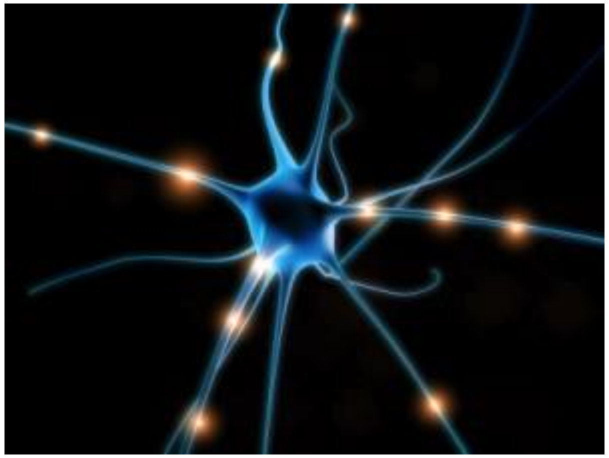


```
# for a single-input model with 2 classes (binary):
model = Sequential()
model.add(Dense(1, input_dim=784, activation='sigmoid'))
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# generate dummy data
import numpy as np
data = np.random.random((1000, 784))
labels = np.random.randint(2, size=(1000, 1))

# train the model, iterating on the data in batches
# of 32 samples
model.fit(data, labels, nb_epoch=10, batch_size=32)
```

Datasets on Keras

- http://keras.io/datasets/
- **CIFAR10 small image classification**
  - keras.datasets.cifar10
  - Dataset of 50,000 32x32 color training images, labeled over 10 categories, and 10,000 test images
  - (X_train, y_train), (X_test, y_test) = cifar10.load_data()
  - Returns: 2 tuples:
    - X_train, X_test: uint8 array of RGB image data with shape (nb_samples, 3, 32, 32)
    - y_train, y_test: uint8 array of category labels (integers in range 0-9) with shape (nb_samples,)
- Others:
  - **CIFAR100 small image classification**
    - Dataset of 50,000 32x32 color training images, labeled over 10 categories, and 10,000 test images.
  - **MNIST database of handwritten digits**
    - Dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images.
  - **IMDB Movie reviews sentiment classification**
    - Dataset of 25,000 movies reviews from IMDB, labeled by sentiment (positive/negative)
  - **Reuters newswire topics classification**
    - Dataset of 11,228 newswires from Reuters, labeled over 46 topics

One neuron



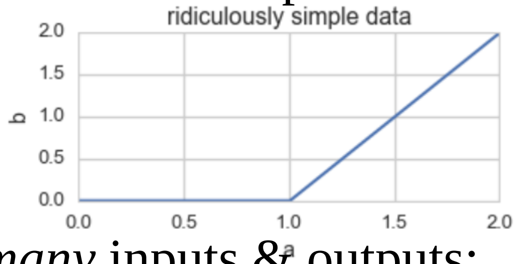
LEARNING KERAS

Prerequisites

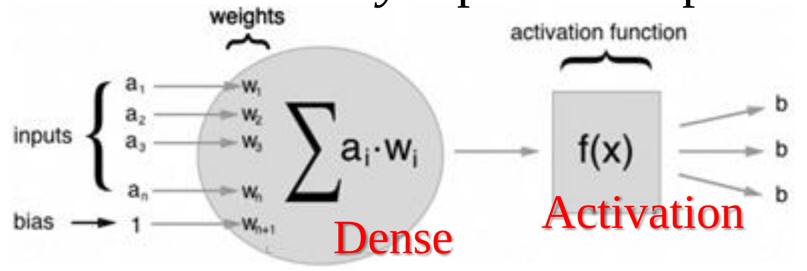
- To build animations:
  - Download ffmpeg (install at `C:\ffmpeg`)
    - For Windows: https://ffmpeg.zeranoe.com/builds/
      - Directions: http://www.wikihow.com/Install-FFmpeg-on-Windows
    - For OSX: http://www.renevolution.com/ffmpeg/2013/03/16/how-to-install-ffmpeg-on-mac-os-x.html
- To reset your PATH environment variable without killing and restarting your command console:
  - `SET PATH=%PATH%;C:\ffmpeg\bin`

How simple of an ANN?

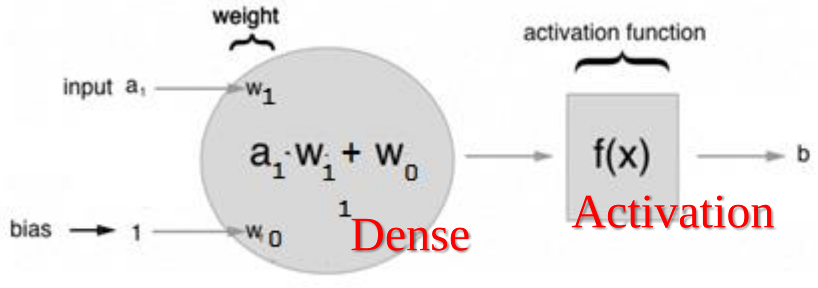
- *Single* neuron, with the simplest nonlinear activation function (ReLU)
- Observations:



- Neuron with *many* inputs & outputs:



- Neuron with *one* input & one output:



Observation data

- Create subfolders **`videos`** and **`images`**
- Run this code in file **`one.py`** (**`python one.py`**):

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import math
sns.set_style('whitegrid')
sns.set_context('talk')

n_points = 200
x = np.linspace(0, 2, n_points)
y = np.array([0] * int(n_points / 2) + list(x[:int(n_points / 2)])) * 2

plt.figure(figsize=(5, 2))
plt.plot(x, y, linewidth=2)
plt.title('ridiculously simple data')
plt.xlabel('a')
plt.ylabel('b')
plt.show()
import pdb;pdb.set_trace()
```

- Type **`continue`** at the command line..

Network with *one* Neuron

- Choices:
  - Initialization of the weights: Randomly drawn from a normal distribution
  - Activation function: ReLu function looks similar to our data
  - Continue code:

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
import numpy as np

np.random.seed(0)
model = Sequential()
model.add(Dense(output_dim=1, input_dim=1, init="normal"))
model.add(Activation("relu"))
model.compile(loss='mean_squared_error', optimizer='sgd')

# print initial weigths
weights = model.layers[0].get_weights()
w0 = weights[0][0][0]
w1 = weights[1][0]
print('neural net initialized with weigths w0: {w0:.2f}, w1: {w1:.2f}'.format(**locals()))
pdb.set_trace()
```

Problems?

- If you have a problem with the very first line
  - `from keras.models import Sequential`
- Then do:
  - `conda install mingw libpython`

Training preliminaries

```
from keras.callbacks import Callback

class TrainingHistory(Callback):
    def on_train_begin(self, logs={}):
        self.losses = []
        self.predictions = []
        self.i = 0
        self.save_every = 50

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))
        self.i += 1
        if self.i % self.save_every == 0:
            pred = model.predict(X_train)
            self.predictions.append(pred)

history = TrainingHistory()
pdb.set_trace()
```

Training

- Training of our neural network is done using *back propagation* of error
- By default error metric in Keras is the *mean squared error*
- We train our neuron on the data by calling the `fit()` method on our model

```
X_train = np.array(x, ndmin=2).T
Y_train = np.array(y, ndmin=2).T
model.fit(X_train,
        Y_train,
        nb_epoch=2000,
        verbose=0,
        callbacks=[history])

# print trained weights
weights = model.layers[0].get_weights()
w0 = weights[0][0][0]
w1 = weights[1][0]
print('neural net weights after training w0: {w0:.2f}, w1: {w1:.2f}'.format(**locals()))
pdb.set_trace()
```

```
neural net weigths after training w0: 1.98, w1: -1.98
```

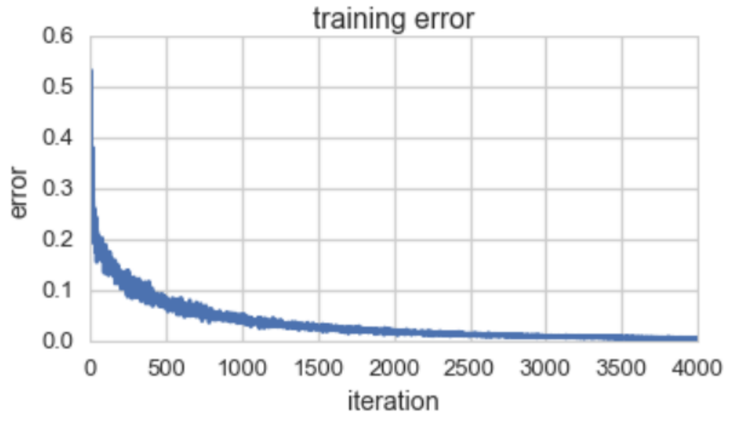
Training history

- It takes *many* iterations to learn!
- *Keras* is using a learning rate of 0.01 by default
- This means in every step it just changes the weights by 1% of the actual change from plain gradient descent
  - It prevents *overfitting* (remember our L1 & L2 labs?)
  - The net learns slower, but gets better at ignoring noise

Analysis

```
# save the animation
import matplotlib.animation as animation

fig = plt.figure(figsize=(5, 2.5))
plt.plot(x, y,  label='data')
line, = plt.plot(x, history.predictions[0],  label='prediction')
plt.legend(loc='upper left')

def update_line(num):
    plt.title('iteration: {0}'.format((history.save_every * (num + 1))))
    line.set_xdata(x)
    line.set_ydata(history.predictions[num])
    return []

ani = animation.FuncAnimation(fig, update_line, len(history.predictions),
                              interval=50, blit=True)
ani.save('videos/neuron.mp4', fps=30, extra_args=['-vcodec', 'libx264', '-pix_fmt','yuv420p'])
plt.close()

plt.figure(figsize=(5, 2.5))
plt.plot(x, y, label='data')
plt.plot(x, history.predictions[0], label='prediction')
plt.legend(loc='upper left')
plt.title('iteration: 0')
plt.savefig('images/neuron_start.png')
plt.close()

plt.figure(figsize=(6, 3))
plt.plot(history.losses)
plt.ylabel('error')
plt.xlabel('iteration')
plt.title('training error')
plt.show()
pdb.set_trace()
```

Training jitters



- In some iterations the error actually gets worse!
  - Keras doesn't do plain *gradient descent*. For large amounts of data this would be to computationally expensive
  - Instead Keras uses *stochastic gradient descent*. It <u>randomly</u> selects a subset of the data for each iteration and does a gradient descent on the error on this subset
  - By default Keras uses 128 data point on each iteration
    - Sample size for stochastic gradient descent is a parameter to the `Model.fit()` method called `batch_size`

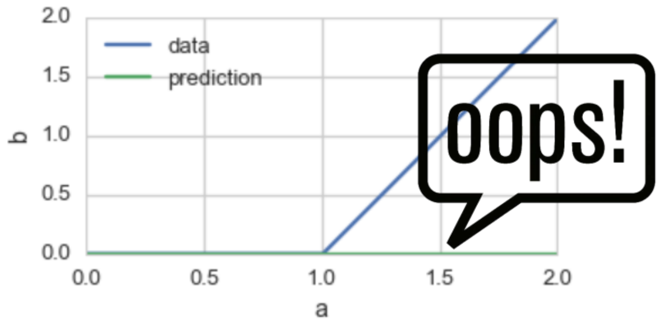
Changing batch_size

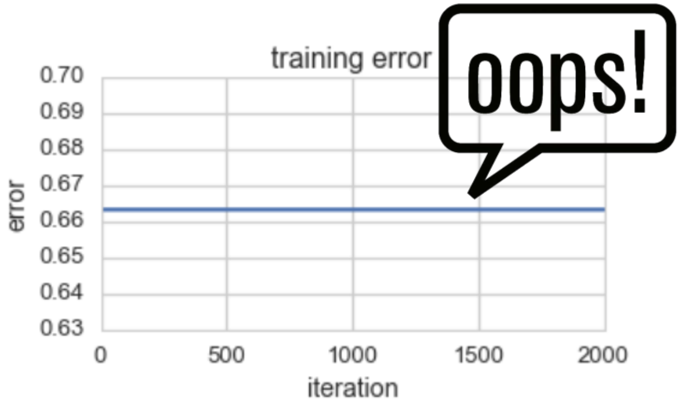
```
history = TrainingHistory()
model = Sequential()
model.add(Dense(output_dim=1, input_dim=1, init="normal"))
model.add(Activation("relu"))
model.compile(loss='mean_squared_error', optimizer='sgd')
model.fit(X_train,
        Y_train,
        batch_size=200,
        nb_epoch=2000,
        verbose=0,
        callbacks=[history])

plt.figure(figsize=(6, 3))
plt.plot(history.losses)
plt.ylabel('error')
plt.xlabel('iteration')
plt.title('training error')
plt.show()
pdb.set_trace()
```

Changing initial weights

```
hnp.random.seed(2)
history = TrainingHistory()
model = Sequential()
model.add(Dense(output_dim=1, input_dim=1, init="normal"))
model.add(Activation("relu"))
model.compile(loss='mean_squared_error', optimizer='sgd')

weights = model.layers[0].get_weights()
w0 = weights[0][0][0]
w1 = weights[1][0]
print('neural net initialized with weigths w0: {w0:.2f}, w1: {w1:.2f}'.format(**locals()))

model.fit(X_train,
        Y_train,
        batch_size=200,
        nb_epoch=2000,
        verbose=0,
        callbacks=[history])

weights = model.layers[0].get_weights()
w0 = weights[0][0][0]
w1 = weights[1][0]
print('neural net weigths after training w0: {w0:.2f}, w1: {w1:.2f}'.format(**locals()))

fig = plt.figure(figsize=(5, 2.5))
plt.plot(x, y,  label='data')
line, = plt.plot(x, history.predictions[0],  label='prediction')
plt.xlabel('a')
plt.ylabel('b')
plt.legend(loc='upper left')

plt.figure(figsize=(5, 2.5))
plt.plot(history.losses)
plt.ylabel('error')
plt.xlabel('iteration')
plt.title('training error')
plt.show()
```

```
neural net initialized with weigths w0: -0.02, w1: 0.00
neural net weigths after training w0: -0.02, w1: 0.00
```





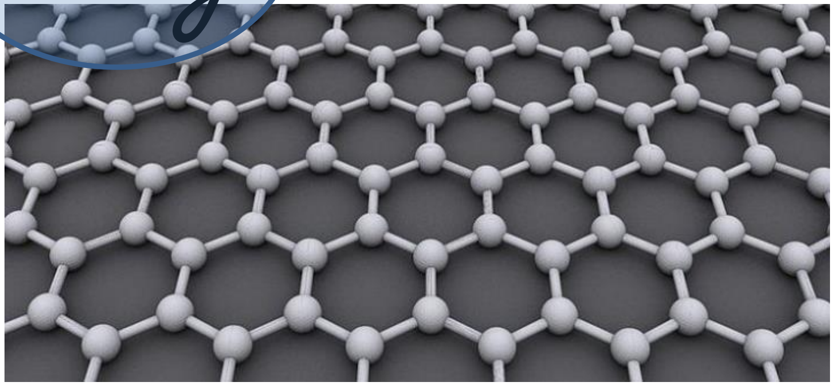
Dying ReLu problem!

- The neuron's weights never get updated during training!
- This is known as the *dying ReLu problem*
  - If the initial weights map all our sample points to values smaller than 0, the ReLu maps everything to 0!
  - Even with small changes in the weights the result is still 0
    - This means the gradient is 0 and the weights never get updated!
  - https://machinelearning.wtf/terms/dying-relu/

Dying ReLu problem!

- A "dead" ReLU always outputs the same value (zero as it happens, but that is not important) for any input
  - Most often because of learning a large negative bias term for its weights
- In turn, that means that neuron takes no role in discriminating between inputs
  - For classification, this is a decision plane outside of all possible input data
- Once a ReLU ends up in this state, it is unlikely to recover, because the function gradient at 0 is also 0, so gradient descent learning will not alter the weights
  - "Leaky" ReLUs with a small positive gradient for negative inputs ($y=0.01x$ when $x < 0$ say) are one attempt to address this issue and give a chance to recover
- You may find that as much as 40% of your network can be "dead" (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high
  - With a proper setting of the learning rate this is less frequently an issue
- Sigmoid and tanh neurons suffer from similar problems as their values saturate, but there is always at least a small gradient allowing them to recover in the long term
- http://datascience.stackexchange.com/questions/5706/what-is-the-dying-relu-problem-in-neural-networks

Sixty neurons

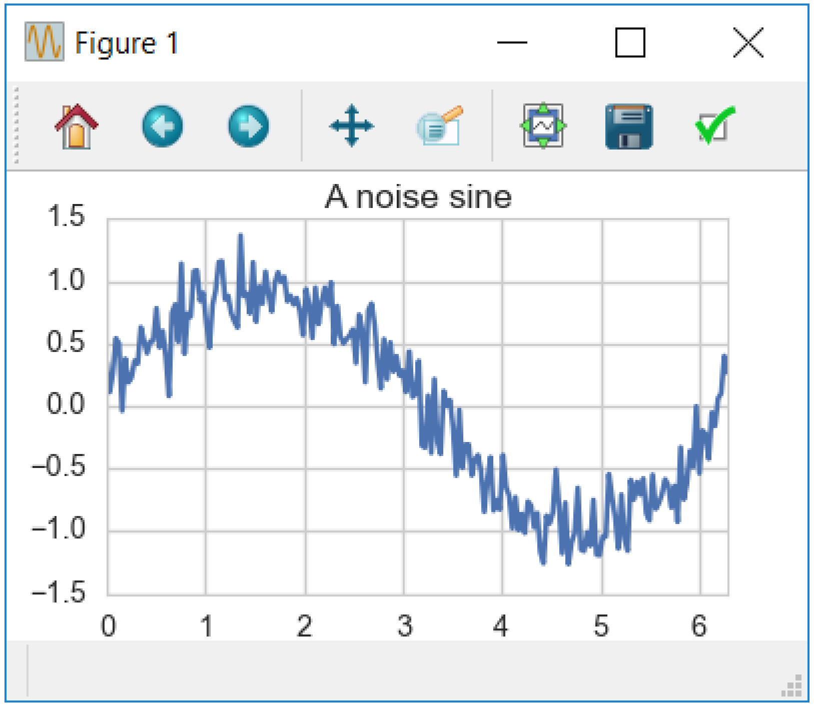


STILL A LOT LESS THAN GOOGLE BRAIN'S DNN..

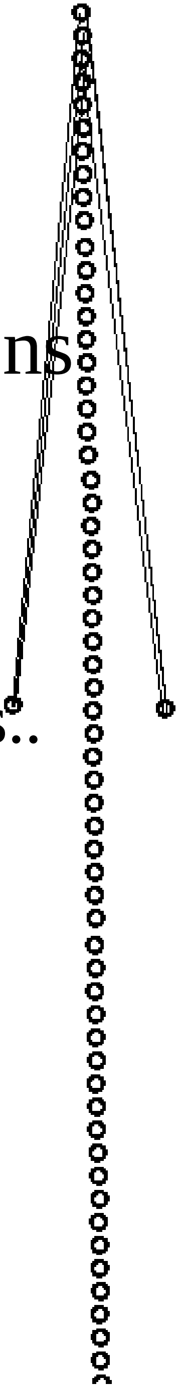
Observations: A Noisy sine

- Create subfolders **`videos`** and **`images`**

```
# Part 1
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import math

sns.set_style('whitegrid')
sns.set_context('talk')

np.random.seed(0)
x = np.linspace(0, 2 * math.pi, 200)
sine = np.sin(x)
err = np.random.normal(0, 0.2, len(sine))
y = sine + err
plt.figure(figsize=(5, 3))
plt.plot(x, y)
plt.xlim([0, 2 * math.pi])
plt.title('A noise sine')
plt.show()
import pdb;pdb.set_trace()
```



Sixty neurons



- 1-60-1 ANN: 1 hidden layer of 60 neurons
  – Model with 120 parameters: 60 weights from the single input to the 60 neurons and 60 weights from the neurons to the single output
  – *Already* a bigger model than our C# models..

```
# Part 2
from keras.models import Sequential
from keras.layers.core import Dense, Activation

n_conn = 60
model = Sequential()
model.add(Dense(output_dim=n_conn, input_dim=1))
model.add(Activation("relu"))
model.add(Dense(output_dim=1))
model.compile(loss='mean_squared_error', optimizer='sgd')
pdb.set_trace()
```

TrainingHistory class

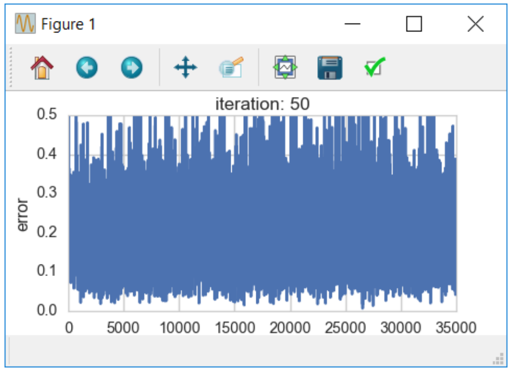
```
# Part 3
from keras.callbacks import Callback

class TrainingHistory(Callback):
    def on_train_begin(self, logs={}):
        self.losses = []
        self.predictions = []
        self.i = 0
        self.save_every = 50

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))
        self.i += 1
        if self.i % self.save_every == 0:
            pred = model.predict(X_train)
            self.predictions.append(pred)

X_train = np.array(x, ndmin=2).T
Y_train = np.array(y, ndmin=2).T
pdb.set_trace()
```

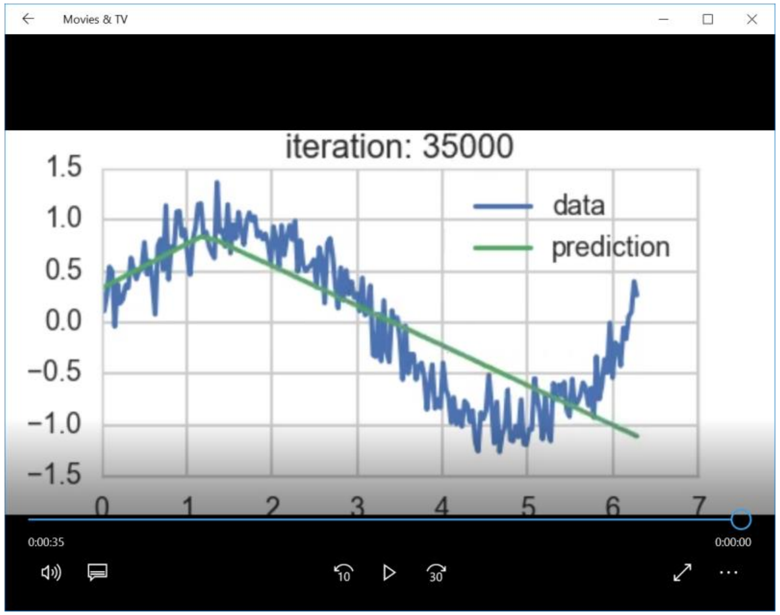
Training

```
# Part 4
history = TrainingHistory()
res = model.fit(X_train,
        Y_train,
        nb_epoch=5000,
        verbose=0,
        callbacks=[history])
pdb.set_trace()
```

Training Visualization

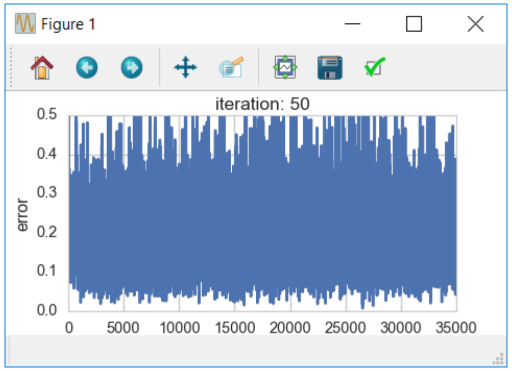
```
# Part 5
# save the animation
import matplotlib.animation as animation

def visualize_training(history, name):
    fig = plt.figure(figsize=(5, 2.5))
    plt.plot(x, y, label='data')
    line, = plt.plot(x, history.predictions[0],  label='prediction')
    plt.legend()

    def update_line(num):
        plt.title('iteration: {0}'.format((history.save_every * (num + 1))))
        line.set_xdata(x)
        line.set_ydata(history.predictions[num])
        return []

    ani = animation.FuncAnimation(fig, update_line, len(history.predictions),
                                  interval=50, blit=True)
    ani.save('videos/{0}.mp4'.format(name), dpi=100, extra_args=['-vcodec', 'libx264', '-pix_fmt','yuv420p'])
    plt.close()

    plt.figure(figsize=(5, 2.5))
    plt.plot(x, y, label='data')
    plt.plot(x, history.predictions[0], label='prediction')
    plt.legend()
    plt.title('iteration: 0')
    plt.savefig('images/{0}.png'.format(name))
    plt.close()

    plt.figure(figsize=(6, 3))
    plt.plot(history.losses)
    plt.ylabel('error')
    plt.xlabel('iteration')
    plt.ylim([0, 0.5])
    plt.title('training error')
    plt.show()

visualize_training(history, 'noisy-sine-one-layer')
pdb.set_trace()
```





Results

- The ANN quickly learns the first hill of the sine function
  - Training error decreasing rapidly during the first few iterations



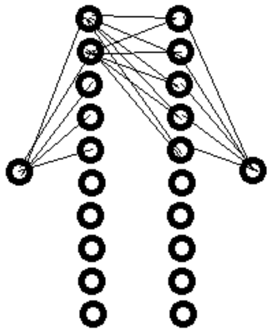
- Then it gets <u>stuck</u> in a local optimum
  - The training error remains in the range 0.2 without any signs of improvement over the last 17000 iterations!
- Why?
  - One layer ANN: Not enough representational power!

2 Hidden layers

- With the same number of weights
  - How?

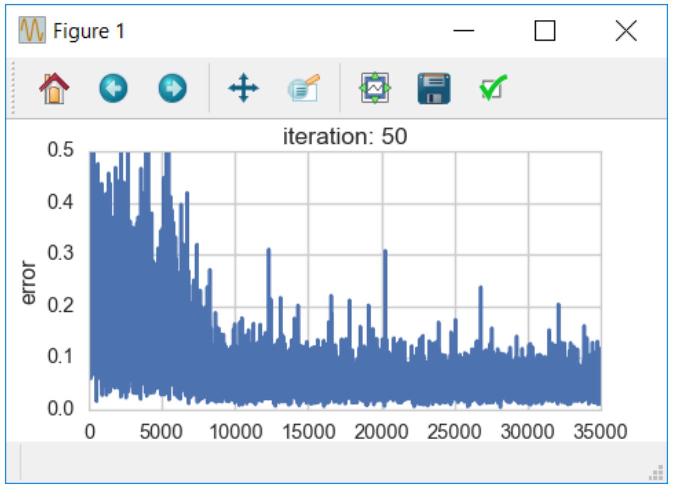
Improvement? 2 Hidden Layers

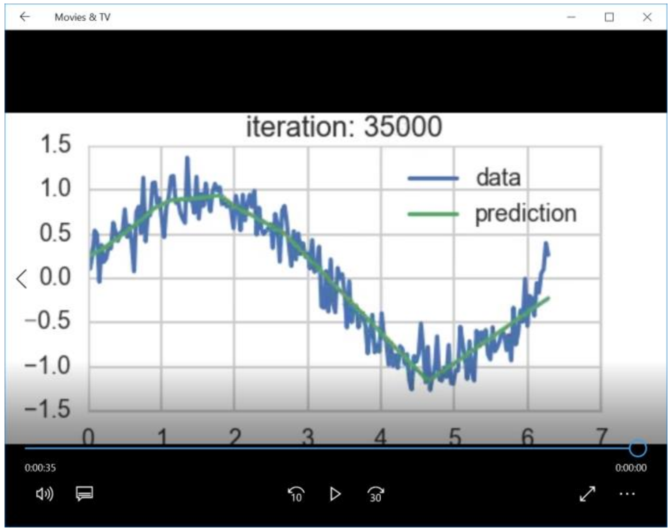
- 1-10-10-1 ANN: Two hidden layers of 10 neurons each
  - Again, 120 weights! Why?
    - 10 weights from the input to the first 10 neurons
    - Weights for the 100 connections from the first 10 to the second 10 neurons
    - 10 from the second neurons to the output



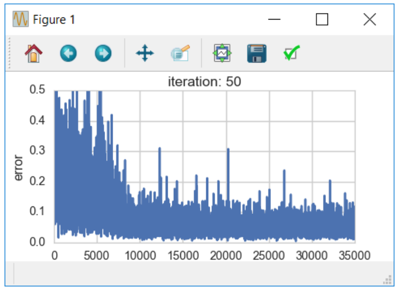
Code

```
# Part 6
n_conn = 10
model = Sequential()
model.add(Dense(output_dim=n_conn, input_dim=1))
model.add(Activation("relu"))
model.add(Dense(output_dim=n_conn))
model.add(Activation("relu"))
model.add(Dense(output_dim=1))
model.compile(loss='mean_squared_error', optimizer='sgd')

X_train = np.array(x, ndmin=2).T
Y_train = np.array(y, ndmin=2).T
history = TrainingHistory()
model.fit(X_train,
        Y_train,
        nb_epoch=5000,
        verbose=0,
        callbacks=[history])

visualize_training(history, 'tiny-sine-two-layer')
pdb.set_trace()
```





Results

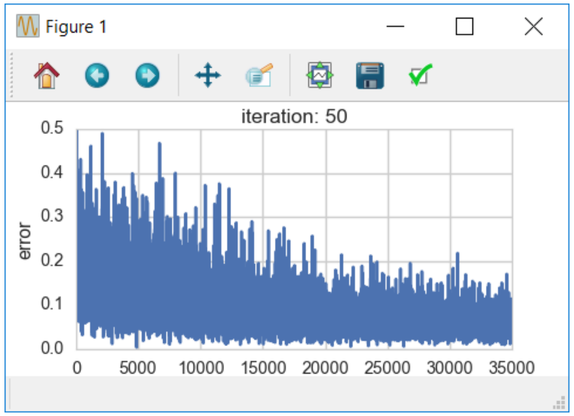


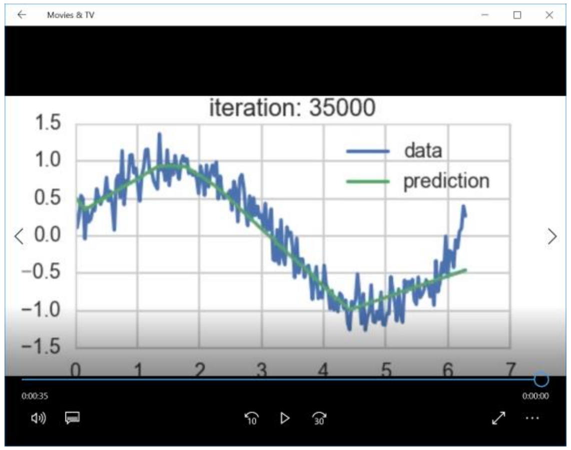
- Better!
- Training error looks improved, but it also looks like it’s gotten stuck again..
- Once again a problem of a local optimum
  - Let's try using a different optimizer
    - Recommended optimizer at Stanford CS231n is Adam..

Adam Optimizer with 1-60-1 ANN

- Let's try the new optimizer with the original 1-60-1 ANN, to compare apples with apples..

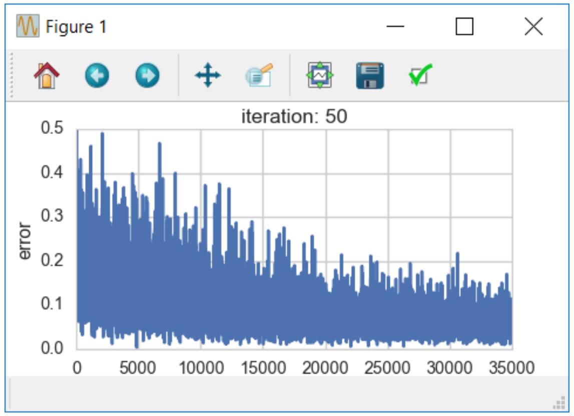
```
# Part 7
from keras.optimizers import Adam

n_conn = 60
model = Sequential()
model.add(Dense(output_dim=n_conn, input_dim=1))
model.add(Activation("relu"))
model.add(Dense(output_dim=1))
adam = Adam()
model.compile(loss='mean_squared_error', optimizer=adam)

X_train = np.array(x, ndmin=2).T
Y_train = np.array(y, ndmin=2).T
history = TrainingHistory()
model.fit(X_train,
        Y_train,
        nb_epoch=5000,
        verbose=0,
        callbacks=[history])

visualize_training(history, 'tiny-sine-one-layer-adam')
pdb.set_trace()
```





Results

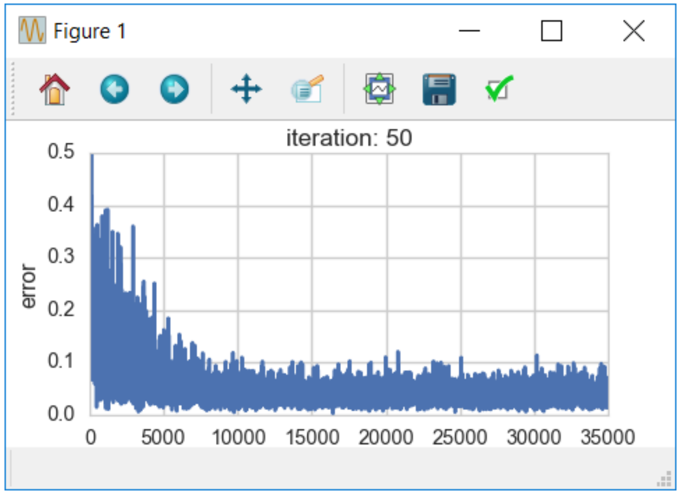
- *Steady* decrease in the training error
  - Final prediction looks close to optimal..

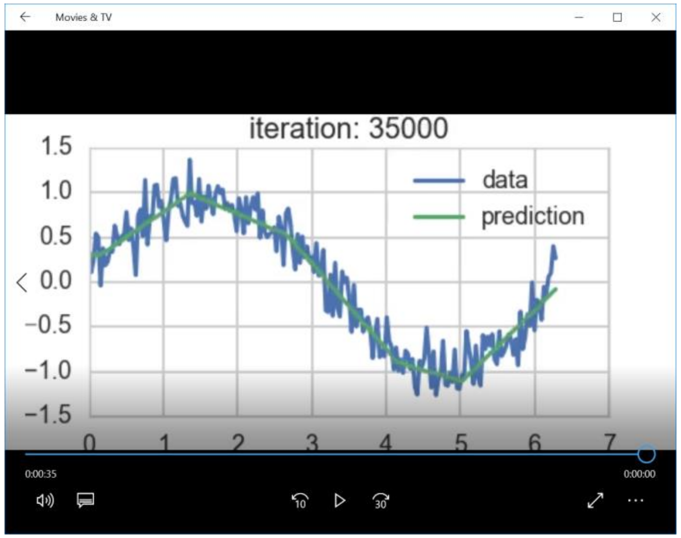


- Let’s try the Adam optimizer with the 1-10-10-1 ANN..

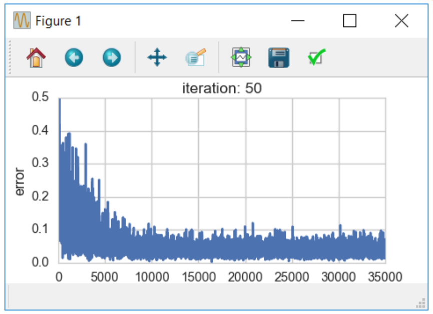
Adam optimizer with 1-10-10-1 ANN

```
# Part 8
n_conn = 10
model = Sequential()
model.add(Dense(output_dim=n_conn, input_dim=1))
model.add(Activation("relu"))
model.add(Dense(output_dim=n_conn))
model.add(Activation("relu"))
model.add(Dense(output_dim=1))
adam = Adam()
model.compile(loss='mean_squared_error', optimizer=adam)

X_train = np.array(x, ndmin=2).T
Y_train = np.array(y, ndmin=2).T
history = TrainingHistory()
model.fit(X_train,
        Y_train,
        nb_epoch=5000,
        verbose=0,
        callbacks=[history])

visualize_training(history, 'tiny-sine-two-layer-adam')pdb.set_trace()
```





Results



- Even better, but close to 1-60-1 ANN with Adam
- Here, the choice of optimizer was *more* important than the number of layers!
- Simple example, but it shows how complex (and powerful) ANNs can be
- And now with *Keras*, you're not limited to few layers and few nodes, you can do hundreds of nodes on a CPU, and thousands on a GPU, with pretty simple programs