Deep Learning

In An Afternoon
For Physicists

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Who am I?

John Urbanic

Parallel Computing Scientist

Pittsburgh Supercomputing Center

&

CMU Physics

I am here to work with you on anything high performance computing related. Which includes many things - like this.

7311 Wean Hall and via urbanic@psc.edu

any time. And I look forward to resuming office hours.

NSF Monthly Workshop Series

- September 3-4 HPC Monthly Workshop: MPI
- October 1-2 HPC Monthly Workshop: Big Data
- November 5 HPC Monthly Workshop: OpenMP
- December 3-4 HPC Monthly Workshop: Big Data
- January 21 HPC Monthly Workshop: OpenMP
- February 19-20 HPC Monthly Workshop: Big Data
- March 3 HPC Monthly Workshop: OpenACC
- April 7-8 HPC Monthly Workshop: Big Data
- May 5-6 HPC Monthly Workshop: MPI
- June 2-5 Summer Boot Camp
- August 10-11 HPC Monthly Workshop: Big Data
- September 14-15 HPC Monthly Workshop: MPI
- October 5-6 HPC Monthly Workshop: Big Data
- November 3 HPC Monthly Workshop: OpenMP
- December 7-8 HPC Monthly Workshop: Big Data

Logistics

Schedule

1:00	Start
3:00	20 Minute Break
5:00	Finish

Materials

http://psc.edu/dl-for-physicists

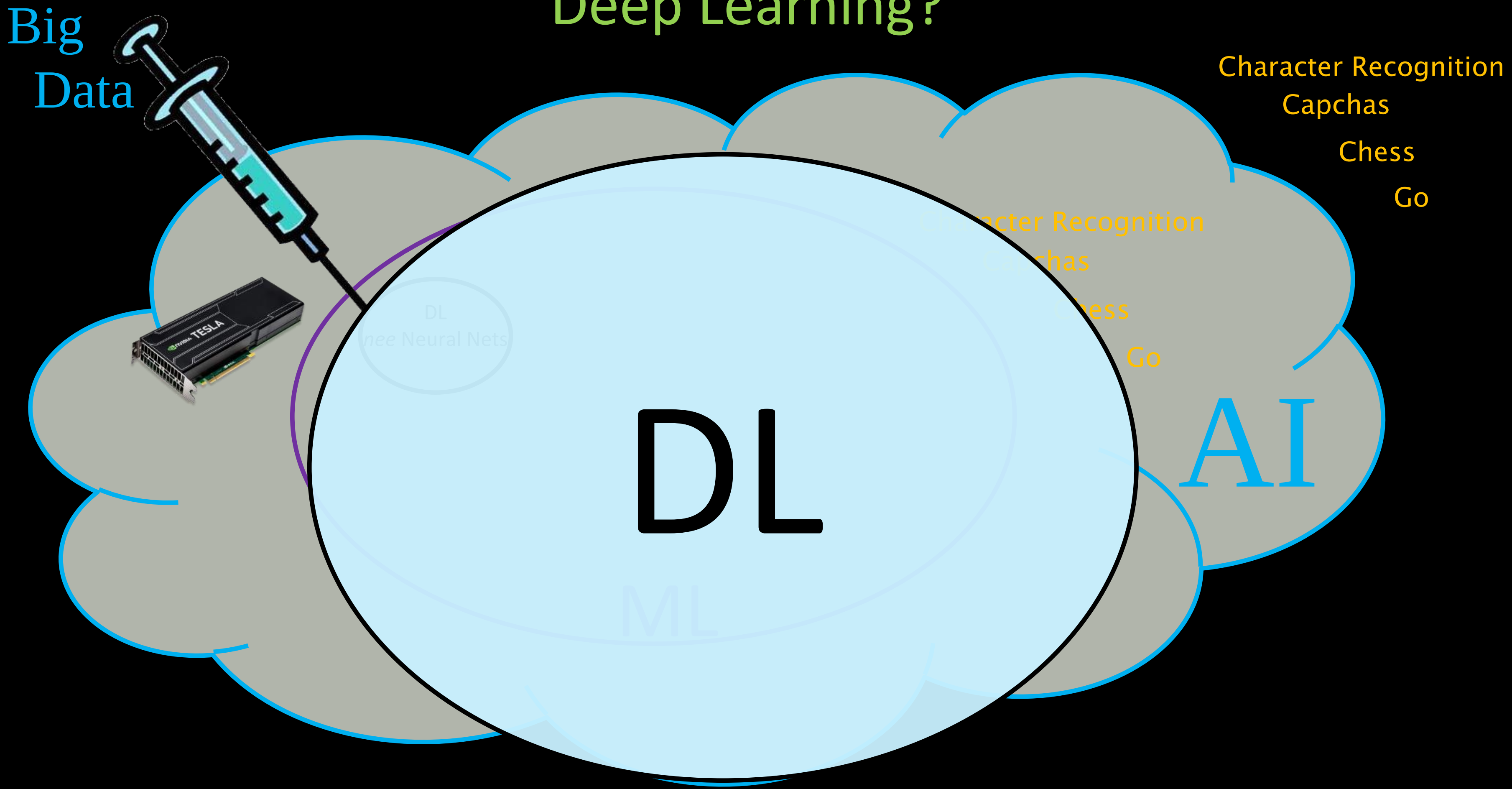
Questions

- I hope you have lots.
- Let's start with chat and hands-up for mic.
- We have a producer to help manage.

Hands-On

When we get there, but mostly after.

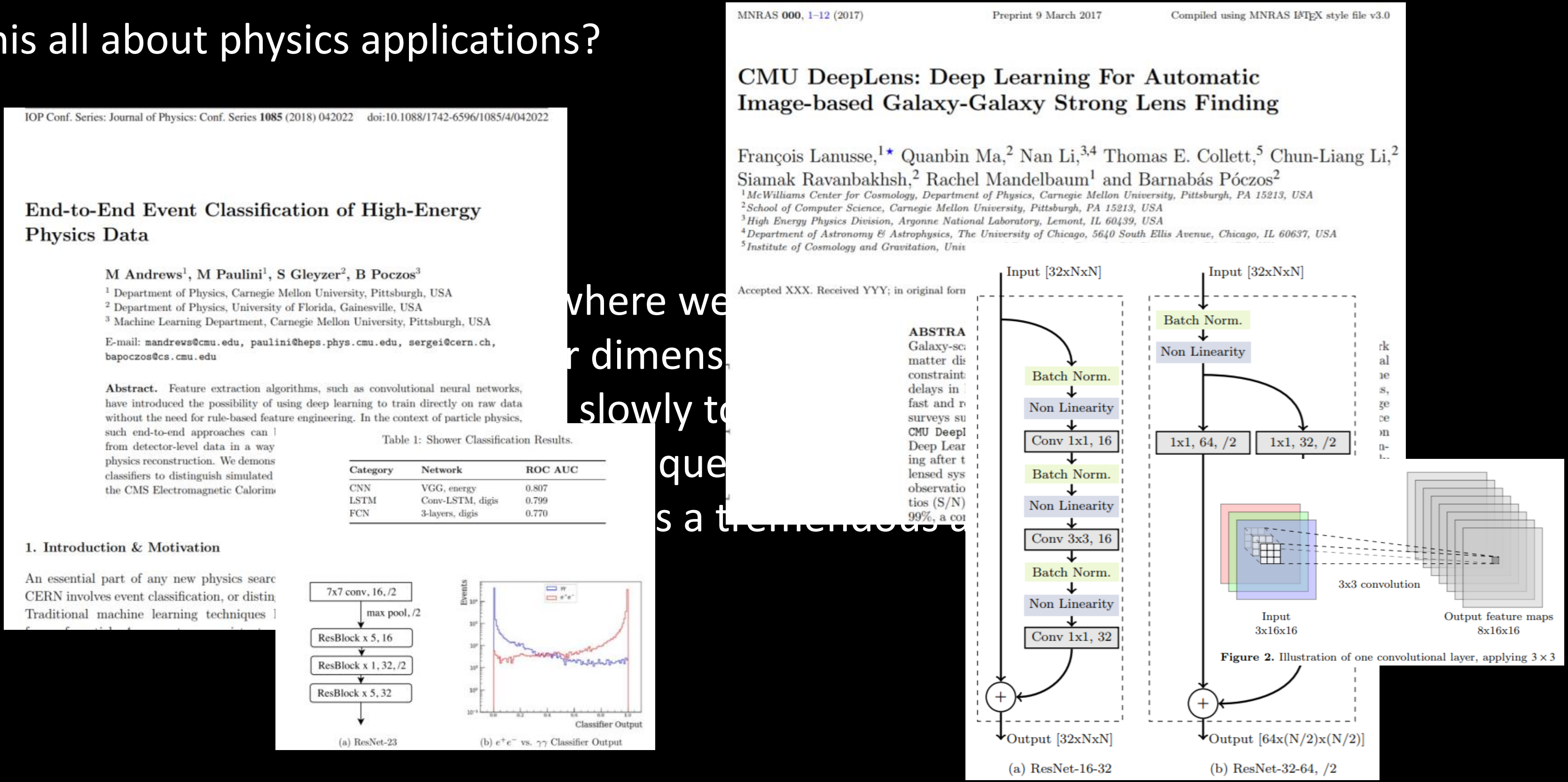
Deep Learning?



For Physicists?

Is this all about physics applications?

where we
r dimens
slowly to
que
s a tremendous u


IOP Conf. Series: Journal of Physics: Conf. Series **1085** (2018) 042022 doi:10.1088/1742-6596/1085/4/042022

End-to-End Event Classification of High-Energy Physics Data

M Andrews[1], M Paulini[1], S Gleyzer[2], B Poczos[3]
[1] Department of Physics, Carnegie Mellon University, Pittsburgh, USA
[2] Department of Physics, University of Florida, Gainesville, USA
[3] Machine Learning Department, Carnegie Mellon University, Pittsburgh, USA

E-mail: mandrews@cmu.edu, paulini@heps.phys.cmu.edu, sergei@cern.ch, bapoczos@cs.cmu.edu

Abstract. Feature extraction algorithms, such as convolutional neural networks, have introduced the possibility of using deep learning to train directly on raw data without the need for rule-based feature engineering. In the context of particle physics, such end-to-end approaches can
from detector-level data in a way
physics reconstruction. We demons
classifiers to distinguish simulated
the CMS Electromagnetic Calorim

1. Introduction & Motivation

An essential part of any new physics searc
CERN involves event classification, or distin
Traditional machine learning techniques

Table 1: Shower Classification Results.

Category	Network	ROC AUC
CNN	VGG, energy	0.807
LSTM	Conv-LSTM, digis	0.799
FCN	3-layers, digis	0.770

(a) ResNet-23
(b) e^+e^- vs. $\gamma\gamma$ Classifier Output

MNRAS **000**, 1–12 (2017) Preprint 9 March 2017 Compiled using MNRAS LaTeX style file v3.0

CMU DeepLens: Deep Learning For Automatic Image-based Galaxy-Galaxy Strong Lens Finding

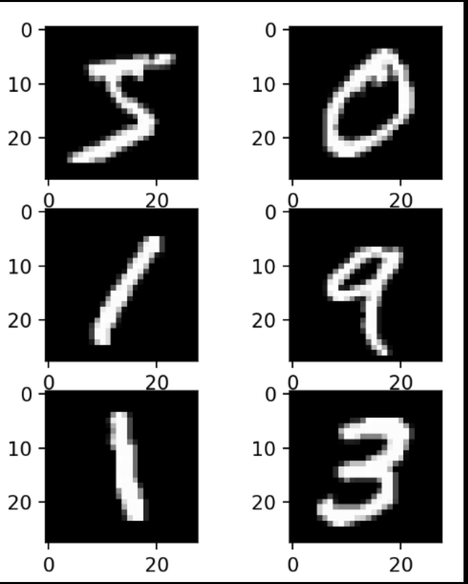
François Lanusse,[1★] Quanbin Ma,[2] Nan Li,[3,4] Thomas E. Collett,[5] Chun-Liang Li,[2] Siamak Ravanbakhsh,[2] Rachel Mandelbaum[1] and Barnabás Póczos[2]
[1] *McWilliams Center for Cosmology, Department of Physics, Carnegie Mellon University, Pittsburgh, PA 15213, USA*
[2] *School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA*
[3] *High Energy Physics Division, Argonne National Laboratory, Lemont, IL 60439, USA*
[4] *Department of Astronomy & Astrophysics, The University of Chicago, 5640 South Ellis Avenue, Chicago, IL 60637, USA*
[5] *Institute of Cosmology and Gravitation, Univ*

Accepted XXX. Received YYY; in original forn

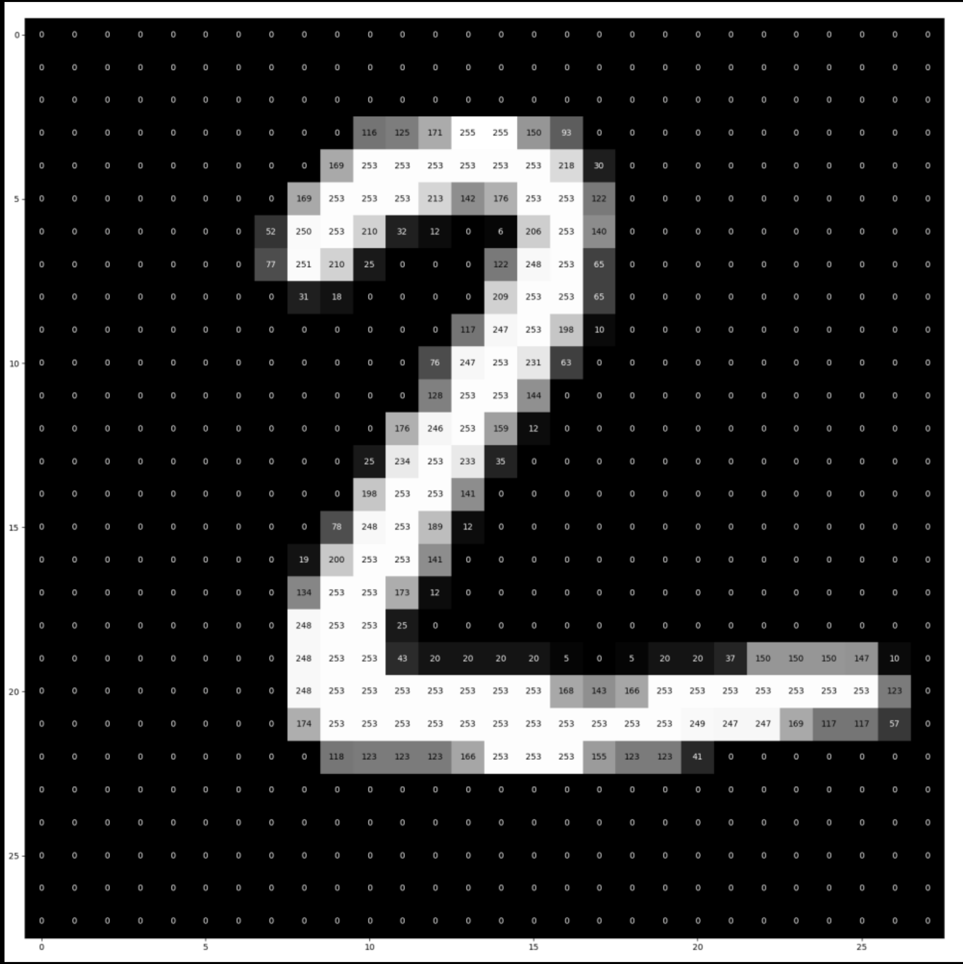
(a) ResNet-16-32
(b) ResNet-32-64, /2

Figure 2. Illustration of one convolutional layer, applying 3 × 3

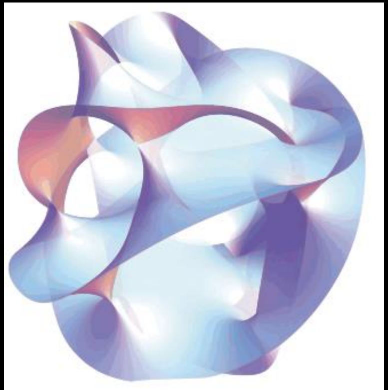
Why Would An Image Have 784 Dimensions?



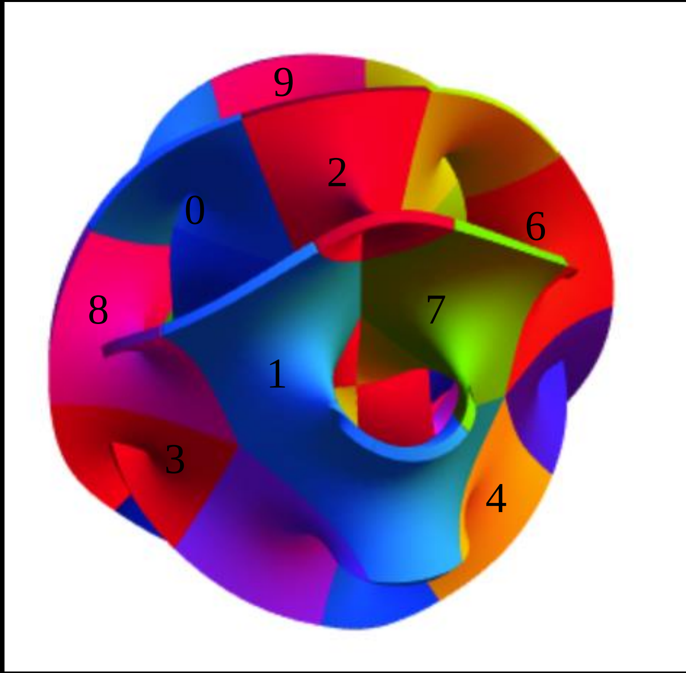
MNIST 28x28
greyscale images



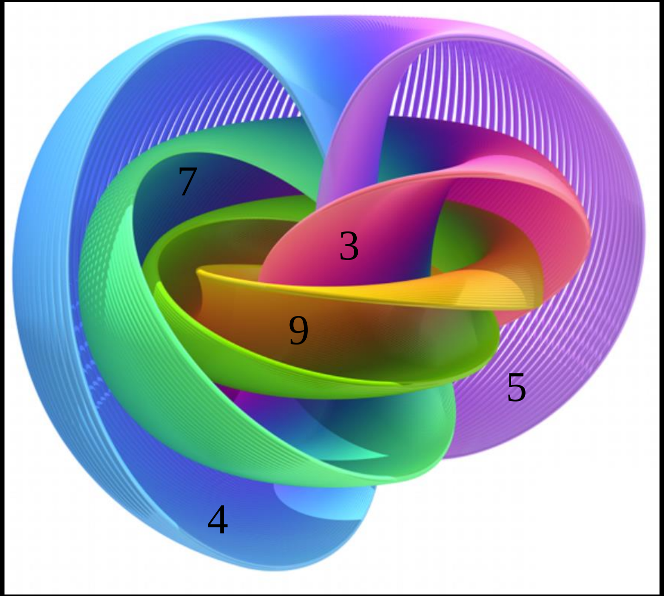
Central Hypothesis of Modern DL



Data Lives On
A Lower Dimensional
Manifold



Maybe Very Contiguous



Maybe Less So

Images from Wikipedia

Dimensionality Reduction

You will find a recurring theme throughout machine learning, not just deep learning:

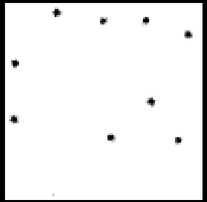
- Our data naturally resides in higher dimensions
- Reducing the dimensionality makes the problem more tractable
- And simultaneously provides us with insight

This last two bullets highlight the principle that "learning" is often finding an effective compressed representation.

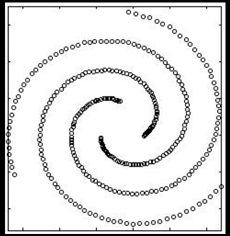
Clustering

As intuitive as clustering is, it presents challenges to implement in an efficient and robust manner.

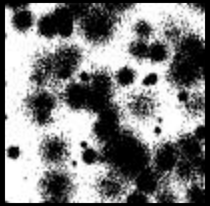
You might think this is trivial to implement in lower dimensional spaces.



But it can get tricky even there.



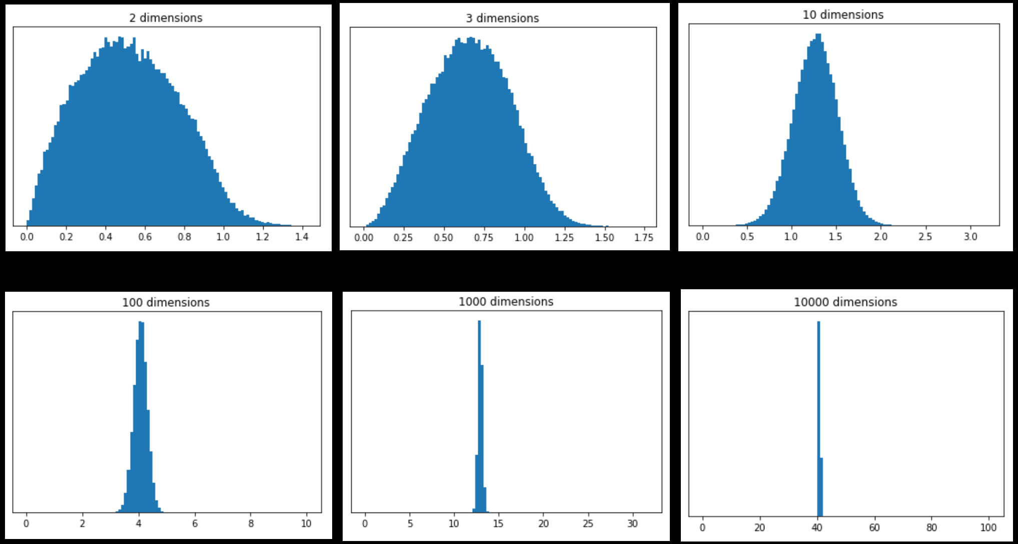
Sometimes you know how many clusters you have to start with. Often you don't.
How hard can it be to count clusters? How many are here?



And in much higher dimensional spaces we run into the *Curse of Dimensionality.*

Curse of Dimensionality

This is a good time to point out how our intuition can lead us astray as we increase the dimensionality of our problems - which we will certainly be doing - and to a great degree. There are several related aspects to this phenomenon, often referred to as the *Curse of Dimensionality*. One root cause of confusion is that our notion of Euclidian distance starts to fail in higher dimensions.



These plots show the distributions of pairwise distances between randomly distributed points within differently dimensioned unit hypercubes. Notice how all the points start to be about the same distance apart.

Once can imagine this makes life harder on a clustering algorithm!

There are other surprising effects: random vectors are almost all orthogonal; the unit sphere takes almost no volume in the unit square. These cause all kinds of problems when generalizing algorithms from our lowly 3D world.

Metrics

Even the definition of distance (the *metric*) can vary based upon application. If you are solving chess problems, you might find the Manhattan distance (or taxicab metric) to be most useful.

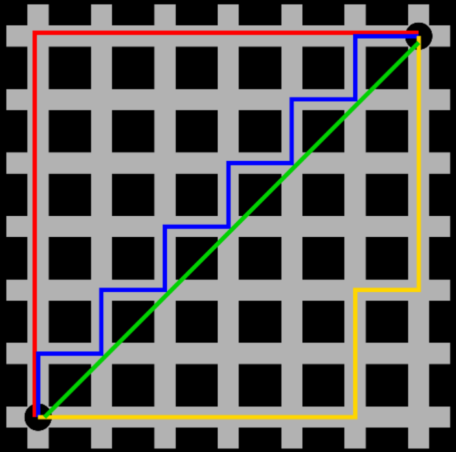


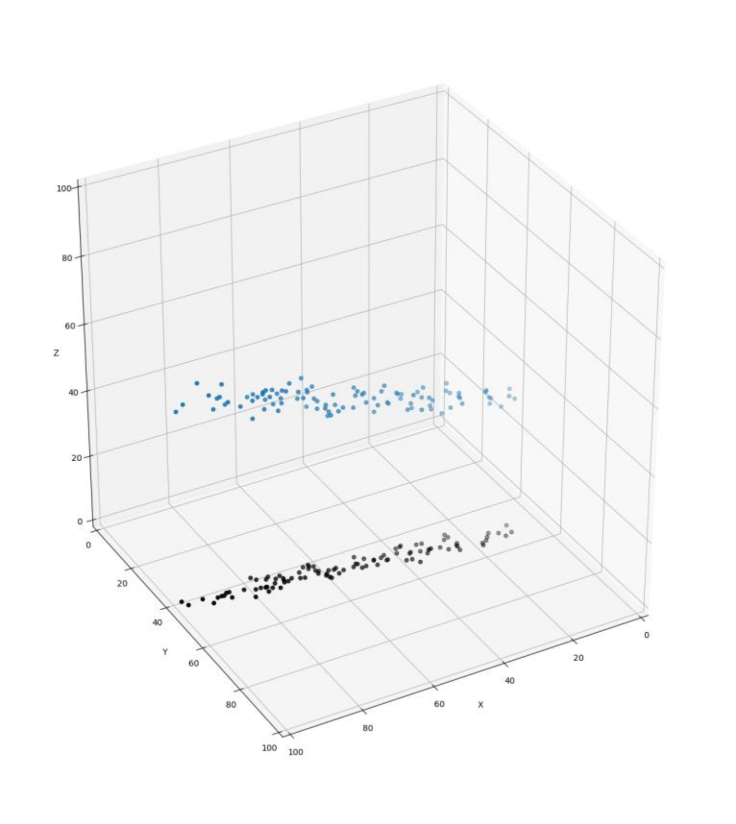
Image Source: Wikipedia

For comparing text strings, we might choose one of dozens of different metrics. For spell checking you might want one that is good for phonetic distance, or maybe edit distance. For natural language processing (NLP), you probably care more about tokens.

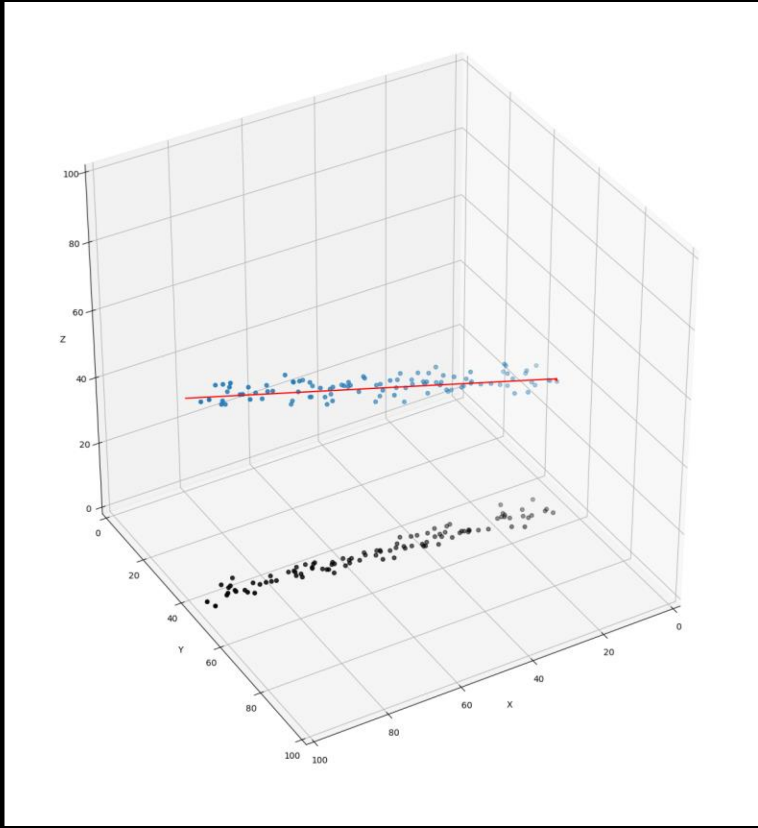
For genomics, you might care more about string sequences.

Some useful measures don't even qualify as metrics (usually because they fail the triangle inequality: $a + b \geq c$).

Alternative DR: Principal Component Analysis

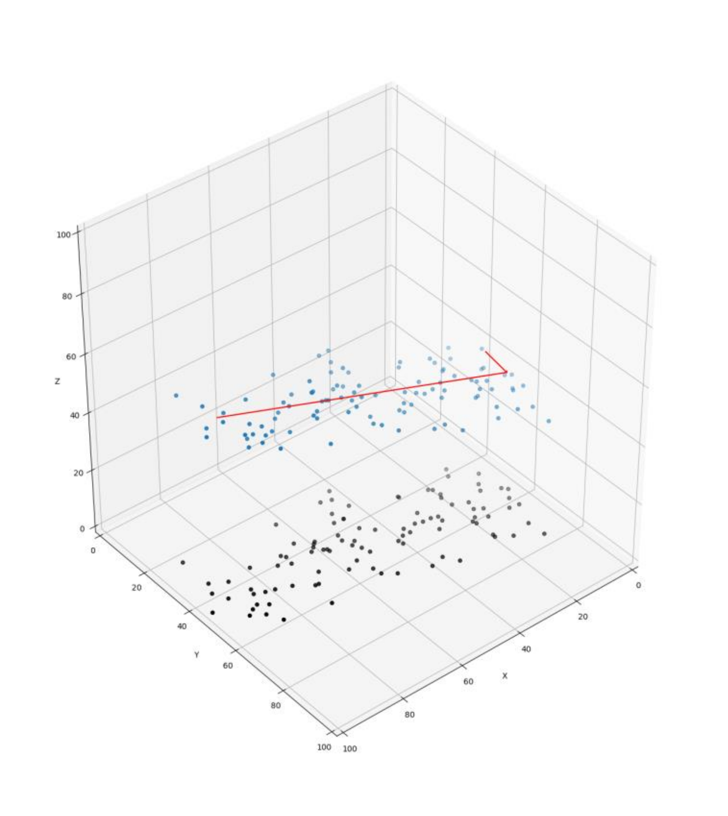


3D Data Set

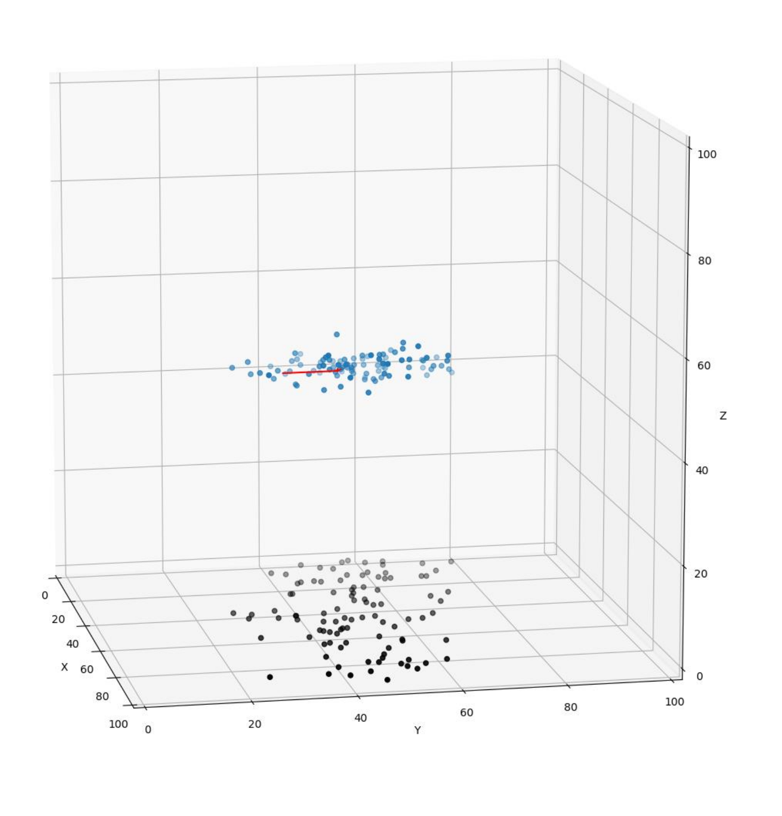


Maybe mostly 1D!

Alternative DR: Principal Component Analysis



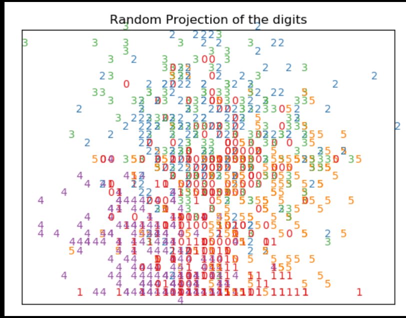
Flatter 2D-ish Data Set

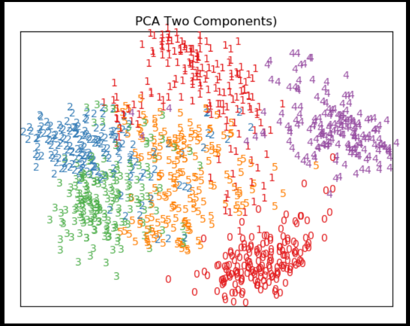


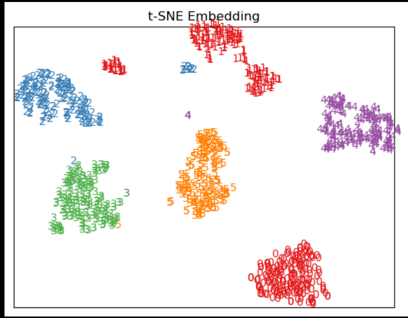
View down the 1st Princ. Comp.

Testing These Ideas With Scikit-learn

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import (datasets, decomposition, manifold, random_projection)

def draw(X, title):
    plt.figure()
    plt.xlim(X.min(0)[0],X.max(0)[0]); plt.ylim(X.min(0)[1],X.max(0)[1])
    plt.xticks([]); plt.yticks([])
    plt.title(title)
    for i in range(X.shape[0]):
        plt.text(X[i, 0], X[i, 1], str(y[i]), color=plt.cm.Set1(y[i] / 10.) )

digits = datasets.load_digits(n_class=6)
X = digits.data
y = digits.target

rp = random_projection.SparseRandomProjection(n_components=2, random_state=42)
X_projected = rp.fit_transform(X)
draw(X_projected, "Random Projection of the digits")

X_pca = decomposition.PCA(n_components=2).fit_transform(X)
draw(X_pca, "PCA (Two Components)")

tsne = manifold.TSNE(n_components=2, init='pca', random_state=0)
X_tsne = tsne.fit_transform(X)
draw(X_tsne, "t-SNE Embedding")

plt.show()
```

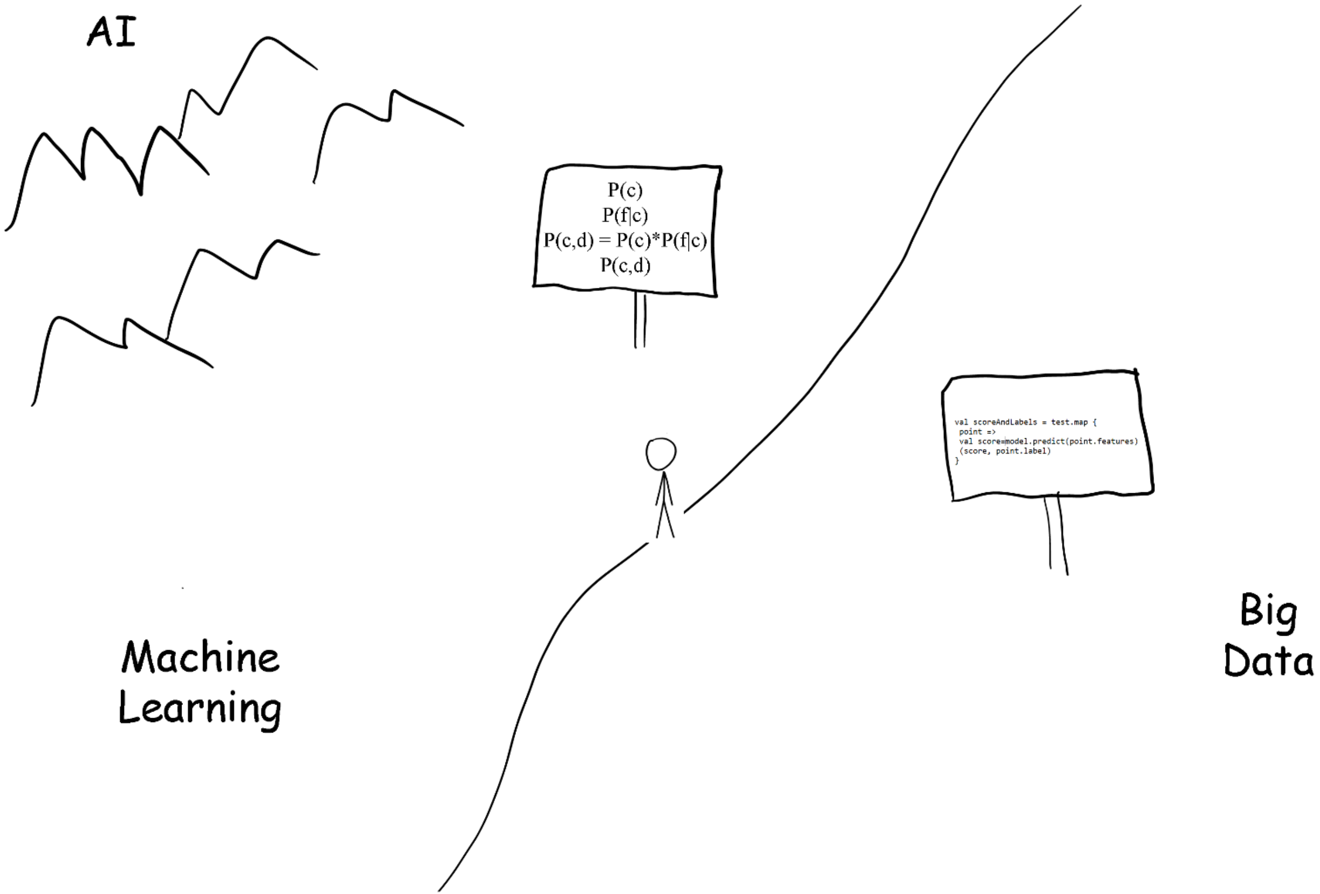

Random Projection of the digits


PCA Two Components)


t-SNE Embedding


Sample of 64-dimensional digits dataset

The Journey Ahead



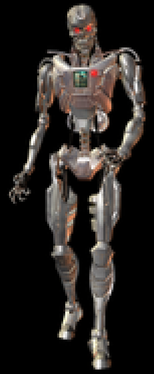
As the Data Scientist wanders across the ill-defined boundary between Data Science and Machine Learning, in search of the fabled land of Artificial Intelligence, they find that the language changes from programming to a creole of linear algebra and probablity and statistics.

Deep Learning / Neural Nets

Without question the biggest thing in ML and computer science right now. Is the hype real? Can you learn anything meaningful in an afternoon? How did we get to this point?

The ideas have been around for decades. Two components came together in the past decade to enable astounding progress:

- Widespread parallel computing (GPUs)
- Big data training sets

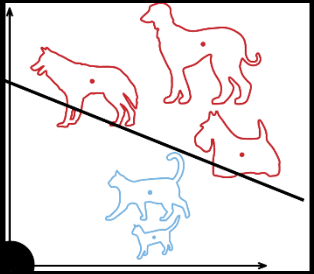


Two Perspectives

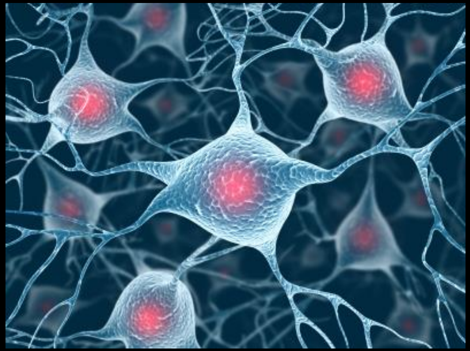
There are really two common ways to view the fundaments of deep learning.

- Inspired by biological models.

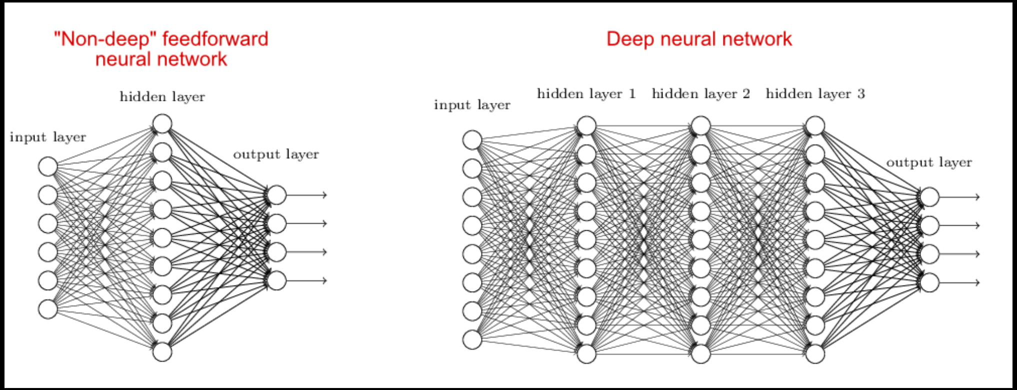
- An evolution of classic ML techniques (the perceptron).



They are both fair and useful. We'll give each a thin slice of our attention before we move on to the actual implementation. You can decide which perspective works for you.



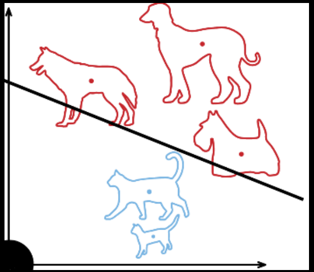
Modeled After The Brain



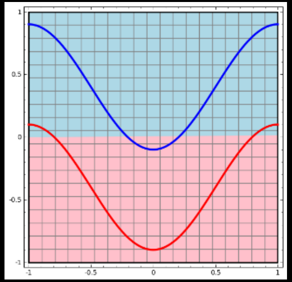
$$
M = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1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

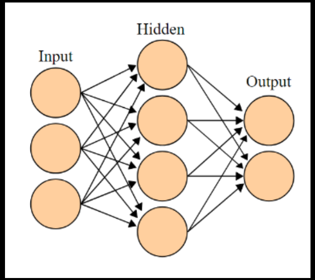
As a Highly Dimensional Non-linear Classifier

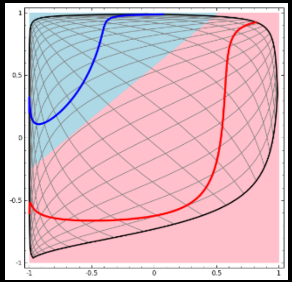
Perceptron



Network





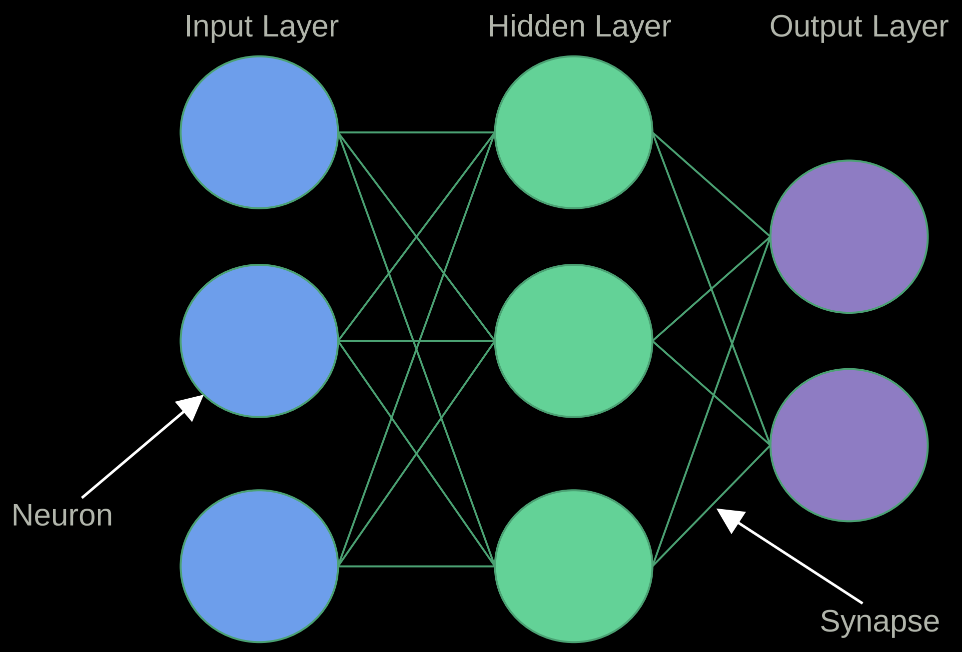


No Hidden Layer
Linear

Hidden Layers
Nonlinear

Courtesy: Chris Olah

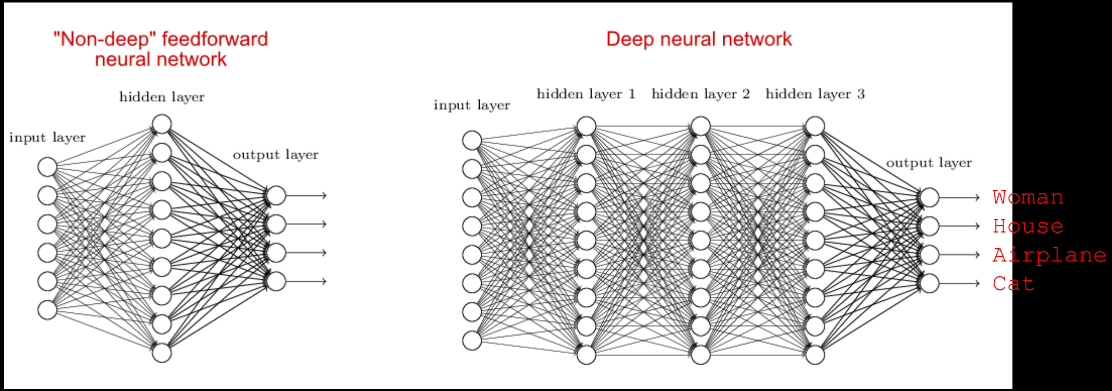
Basic NN Architecture



In Practice

How many inputs?

For an image it could be one (or 3) per pixel.



How deep?

100+ layers have become common.

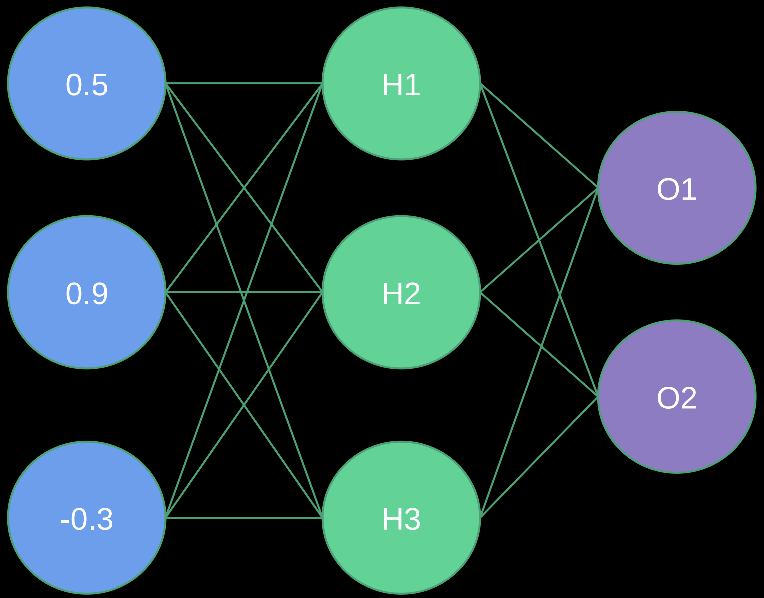
How many outputs?

Might be an entire image.

Or could be discreet set of classification possibilities.

Inference

The "forward" or thinking step



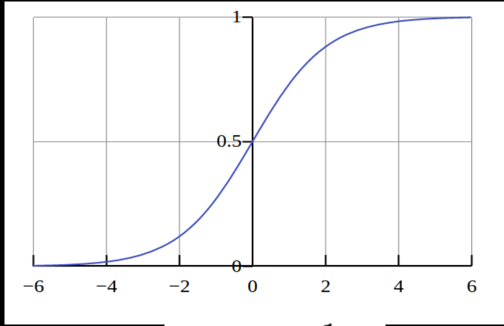
H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

Activation Function

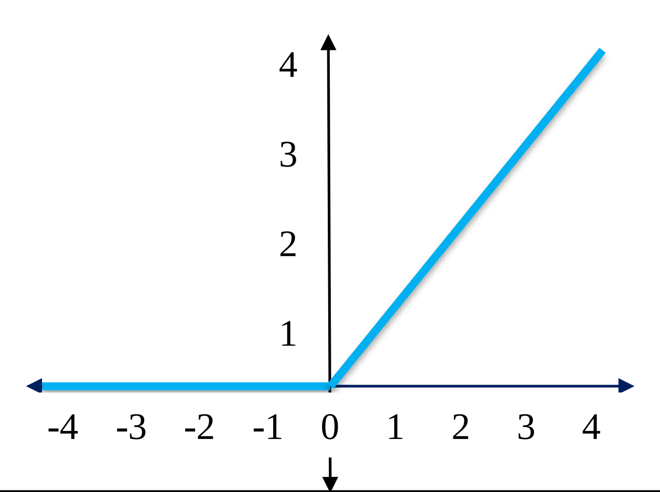
Neurons apply activation functions at these summed inputs. Activation functions are typically non-linear.

- The Sigmoid function produces a value between 0 and 1, so it is intuitive when a probability is desired, and was almost standard for many years.

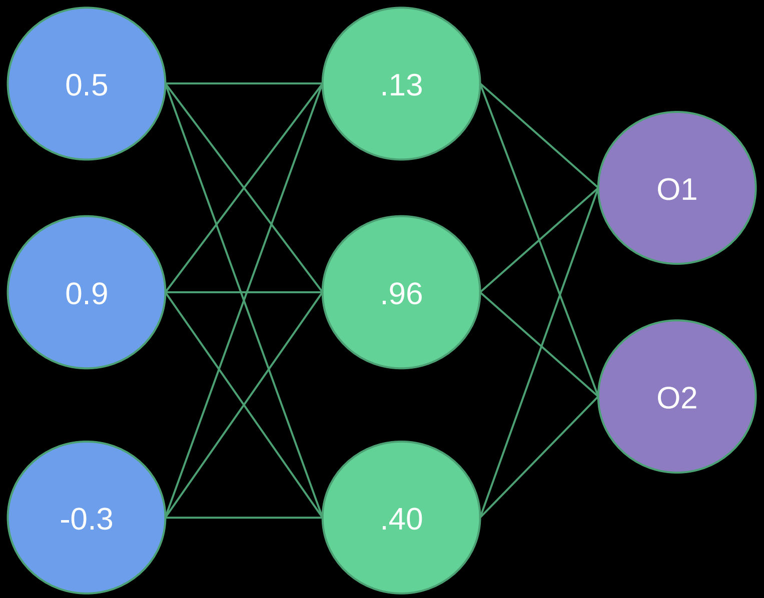


$$S(t) = \frac{1}{1+e^{-t}}$$

- The Rectified Linear activation function is zero when the input is negative and is equal to the input when the input is positive. Rectified Linear activation functions are currently the most popular activation function as they are more efficient than the sigmoid or hyperbolic tangent.
  - Sparse activation: In a randomly initialized network, only 50% of hidden units are active.
  - Better gradient propagation: Fewer vanishing gradient problems compared to sigmoidal activation functions that saturate in both directions.
  - Efficient computation: Only comparison, addition and multiplication.
  - There are Leaky and Noisy variants.



Inference

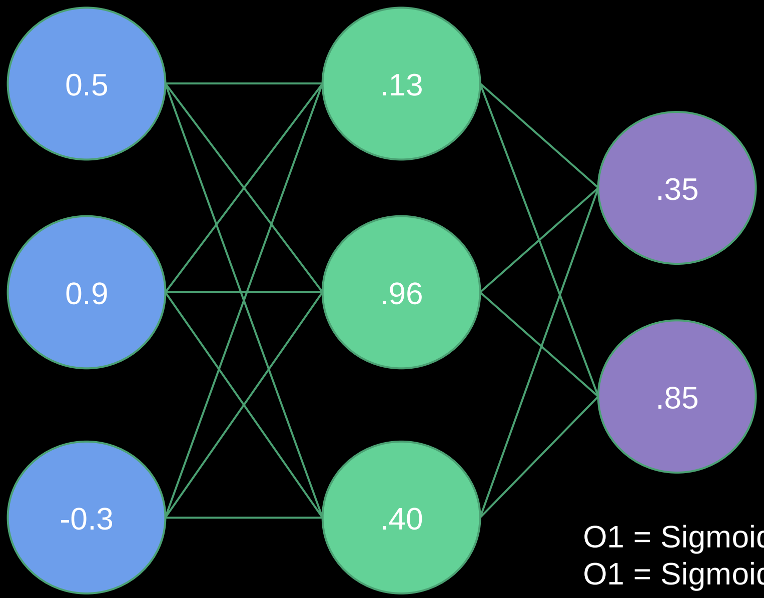


H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

H1 = Sigmoid(0.5 * 1.0 + 0.9 * -2.0 + -0.3 * 2.0) = Sigmoid(-1.9) = .13
H2 = Sigmoid(0.5 * 2.0 + 0.9 * 1.0 + -0.3 * -4.0) = Sigmoid(3.1) = .96
H3 = Sigmoid(0.5 * 1.0 + 0.9 * -1.0 + -0.3 * 0.0) = Sigmoid(-0.4) = .40

Inference



H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

O1 = Sigmoid(.13 * -3.0 + .96 * 1.0 + .40 * -3.0) = Sigmoid(-.63) = .35
O1 = Sigmoid(.13 * 0.0 + .96 * 1.0 + .40 * 2.0) = Sigmoid(1.76) = .85

As A Matrix Operation

H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

$$\mathrm{Sig}\left(\overset{\text{Hidden Layer Weights}}{\begin{bmatrix} 1.0 & -2.0 & 2.0 \\ 2.0 & 1.0 & -4.0 \\ 1.0 & -1.0 & 0.0 \end{bmatrix}} * \overset{\text{Inputs}}{\begin{bmatrix} 0.5 \\ 0.9 \\ -0.3 \end{bmatrix}}\right) = \mathrm{Sig}\left(\begin{bmatrix} -1.9 & 3.1 & -0.4 \end{bmatrix}\right) = \overset{\text{Hidden Layer Outputs}}{\begin{bmatrix} .13 & .96 & 0.4 \end{bmatrix}}$$

Now this looks like something that we can pump through a GPU.

Biases

It is also very useful to be able to offset our inputs by some constant. You can think of this as centering the activation function, or translating the solution (next slide). We will call this constant the *bias*, and it there will often be one value per layer.

Our math for the previously calculated layer now looks like this with bias=0.1:

$$\mathrm{Sig}\left(\overset{\text{Hidden Layer Weights}}{\begin{bmatrix} 1.0 & -2.0 & 2.0 \\ 2.0 & 1.0 & -4.0 \\ 1.0 & -1.0 & 0.0 \end{bmatrix}} * \overset{\text{Inputs}}{\begin{bmatrix} 0.5 \\ 0.9 \\ -0.3 \end{bmatrix}} + \overset{\text{Bias}}{\begin{bmatrix} 0.1 \\ 0.1 \\ 0.1 \end{bmatrix}} \right) = \mathrm{Sig}\left(\begin{bmatrix} -1.8 & 3.2 & -0.3 \end{bmatrix} \right) = \overset{\text{Hidden Layer Outputs}}{\begin{bmatrix} .14 & .96 & 0.4 \end{bmatrix}}$$

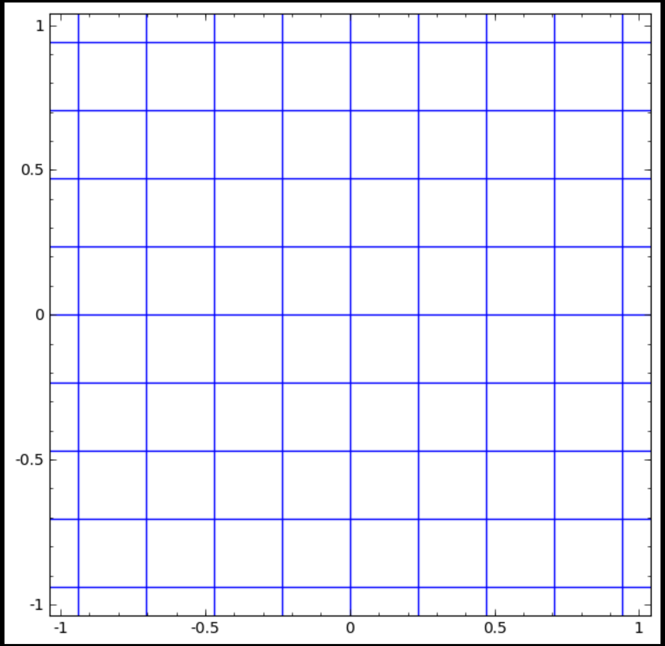
Linear + Nonlinear

The magic formula for a neural net is that, at each layer, we apply linear operations (which look naturally like linear algebra matrix operations) and then pipe the final result through some kind of final nonlinear activation function. The combination of the two allows us to do very general transforms.

The matrix multiply provides the *skew, rotation* and *scale*.

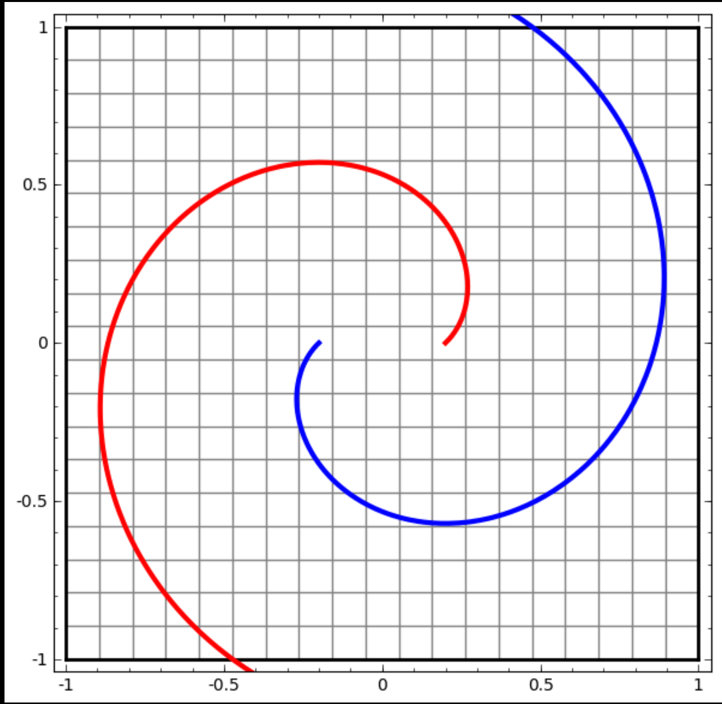
The bias provides the *translation*.

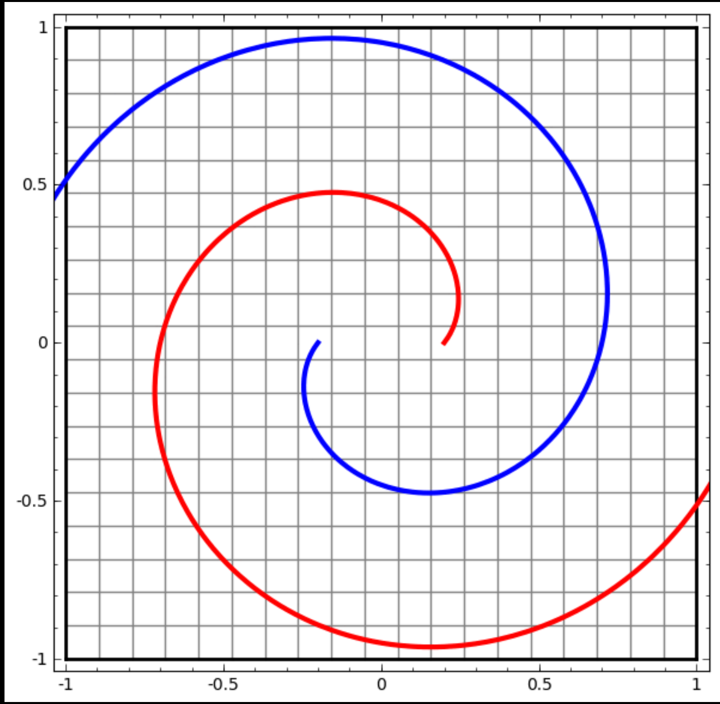
The activation function provides the *warp*.



Linear + Nonlinear

These are two very simple networks untangling spirals. Note that the second does not succeed. With more substantial networks these would both be trivial.

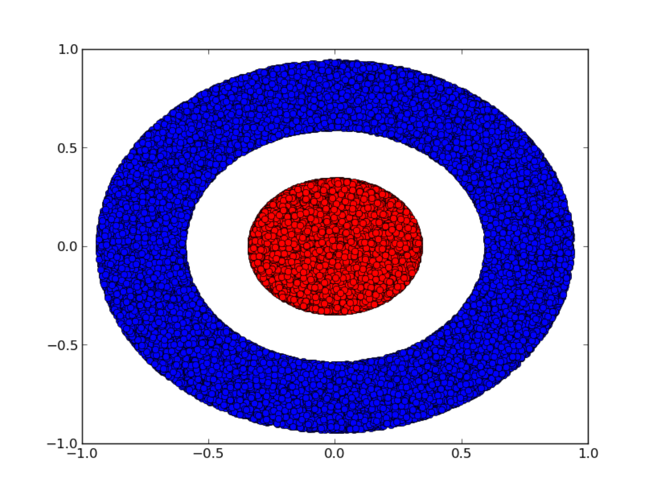




Courtesy: Chris Olah

Width of Network

A very underappreciated fact about networks is that the width of any layer determines how many dimensions it can work in. This is valuable even for lower dimension problems. How about trying to classify (separate) this dataset:

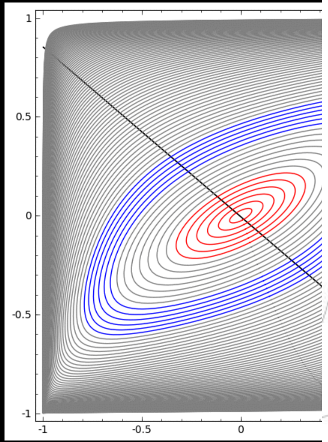


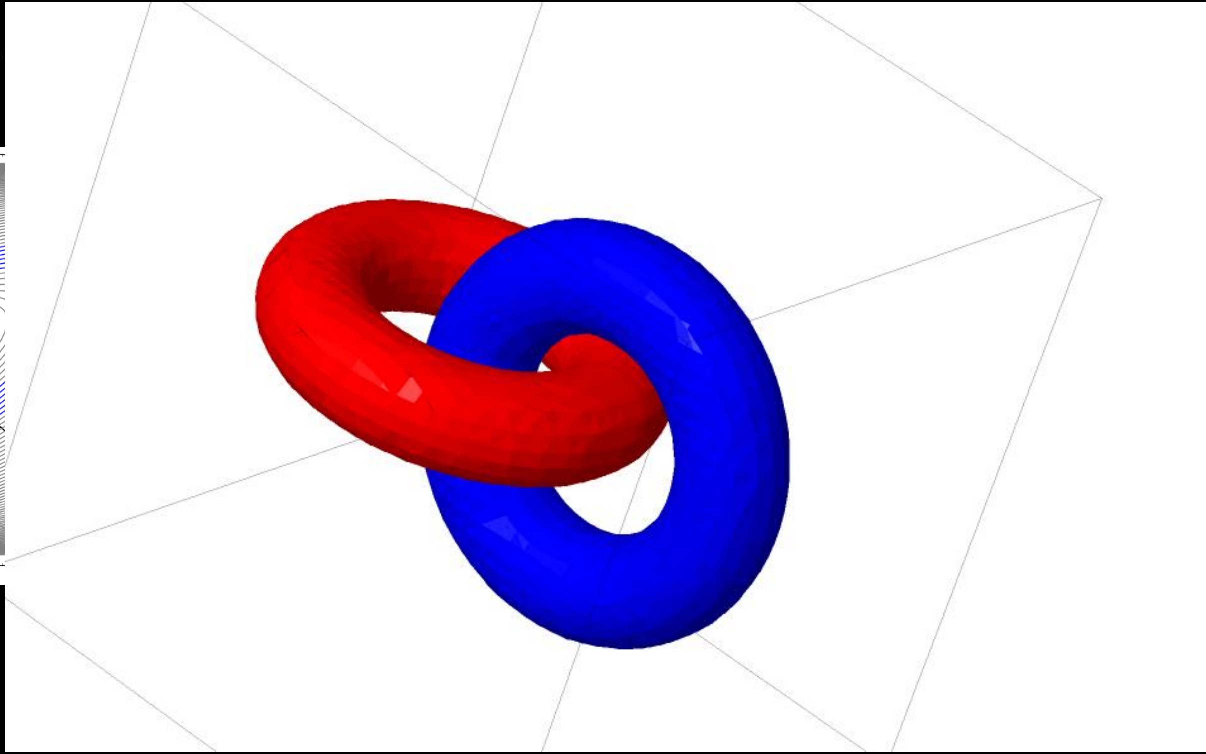
Can a neural net do this with twisting and deforming? What good does it do to have more than two dimensions with a 2D dataset?

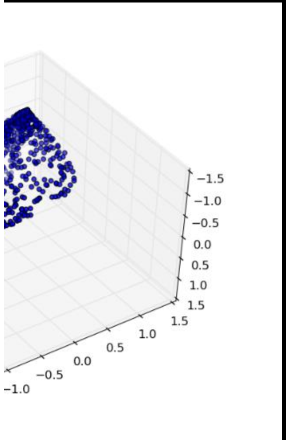
Courtesy: Chris Olah

Working In Higher Dimensions

It takes at least 3







Trying

s in 3D

Greater depth allows us to stack these operations, and can be very effective. The gains from depth are harder to characterize.

Courtesy: Chris Olah

Theoretically

Universal Approximation Theorem: A 1-hidden-layer feedforward network of this type can approximate any function[1], given enough width[2].

Not really that useful as:

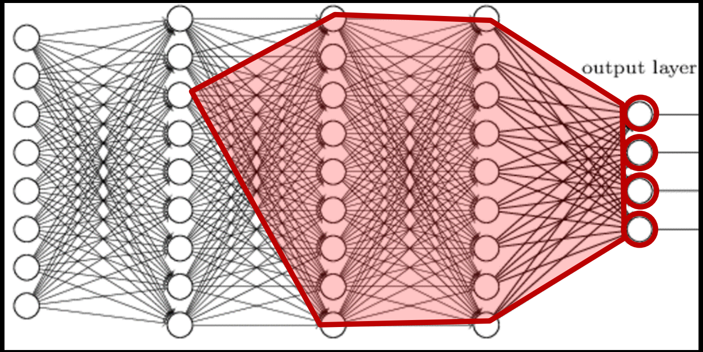
- Width could be enormous.
- Doesn't tell us how to find the correct weights.

1) Borel measurable. Basically, mostly continuous and bounded.
2) Could be exponential number of hidden units, with one unit required for each distinguishable input configuration.

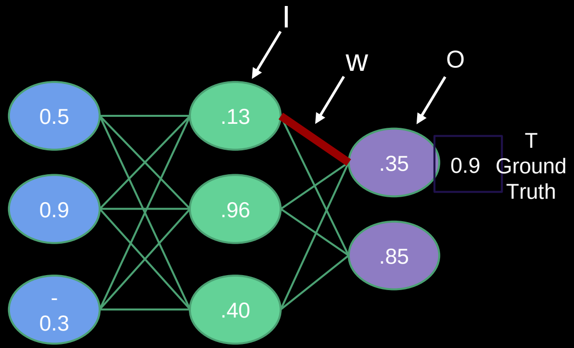
Training Neural Networks

So how do we find these magic weights? We want to minimize the error on our training data. Given labeled inputs, select weights that generate the smallest average error on the outputs.

We know that the output is a function of the weights: $E(w_1,w_2,w_3,\ldots i_1,\ldots t_1,\ldots)$. So to figure out which way, and how much, to push any particular weight, say w_3, we want to calculate $\frac{\partial E}{\partial w_3}$



There are a lot of dependencies going on here. It isn't obvious that there is a viable way to do this in very large networks.



If we take one small piece, it doesn't look so bad.

$$\frac{\partial E}{\partial w} = I \cdot (O - T) \cdot O \cdot (1 - O)$$

$$\frac{\partial E}{\partial w} = .13 \cdot (.35 - .9) \cdot .35 \cdot (1 - .35)$$

For Sigmoid

$$S(t) = \frac{1}{1 + e^{-t}}$$

Note that the role of the gradient, $\frac{\partial E}{\partial w_3}$, here means that it becomes a problem if it vanishes. This is an issue for very deep networks.

Backpropagation

If we use the chain rule repeatedly across layers we can work our way backwards from the output error through the weights, adjusting them as we go. Note that this is where the requirement that activation functions must have nicely behaved derivatives comes from.

This technique makes the weight inter-dependencies much more tractable. An elegant perspective on this can be found from Chris Olah at

http://colah.github.io/posts/2015-08-Backprop .

With basic calculus you can readily work through the details. You can find an excellent explanation from the renowned *3Blue1Brown* at

https://www.youtube.com/watch?v=Ilg3gGewQ5U .

You don't need to know the details, and this is all we have time to say, but you certainly can understand this fully if your freshman calculus isn't too rusty and you have some spare time.

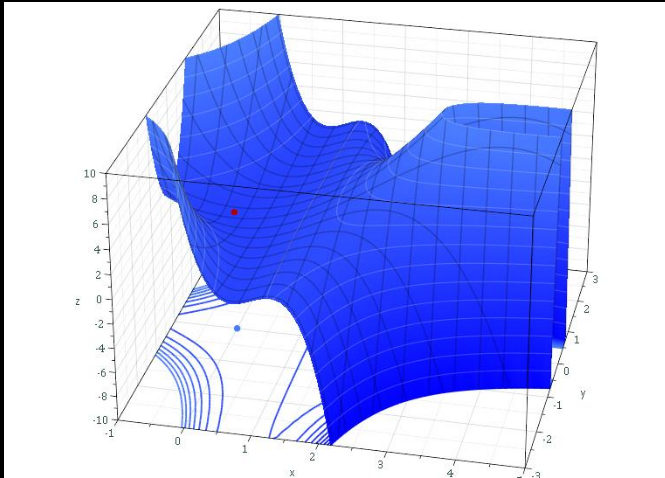
Solvers

However, even this efficient process leaves us with potentially many millions of simultaneous equations to solve (real nets have a lot of weights). They are non-linear to boot. Fortunately, this isn't a new problem created by deep learning, so we have options from the world of numerical methods.

The standard has been *gradient descent*. Methods, often similar, have arisen that perform better for deep learning applications. TensorFlow will allow us to use these interchangeably - and we will.

Most interesting recent methods incorporate *momentum* to help get over a local minimum. Momentum and *step size* are the two *hyperparameters* we will encounter later.

Nevertheless, we don't expect to ever find the actual global minimum.



Wikipedia Commons

We could/should find the error for all the training data before updating the weights (an *epoch*). However it is usually much more efficient to use a *stochastic* approach, sampling a random subset of the data, updating the weights, and then repeating with another *mini-batch*.

Going To Play Along?

Make sure you are on a GPU node:

```
bridges2-login014% interact -gpu
v001%
```

Load the TensorFlow 2 Container:

```
v001% singularity shell --nv  /ocean/containers/ngc/tensorflow/tensorflow_21.02-tf2-py3.sif
```

And start TensorFlow:

```
Singularity> python
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license"
>>> import tensorflow
>>> ...some congratulatory noise...
>>>
```

Two Other Ways To Play Along

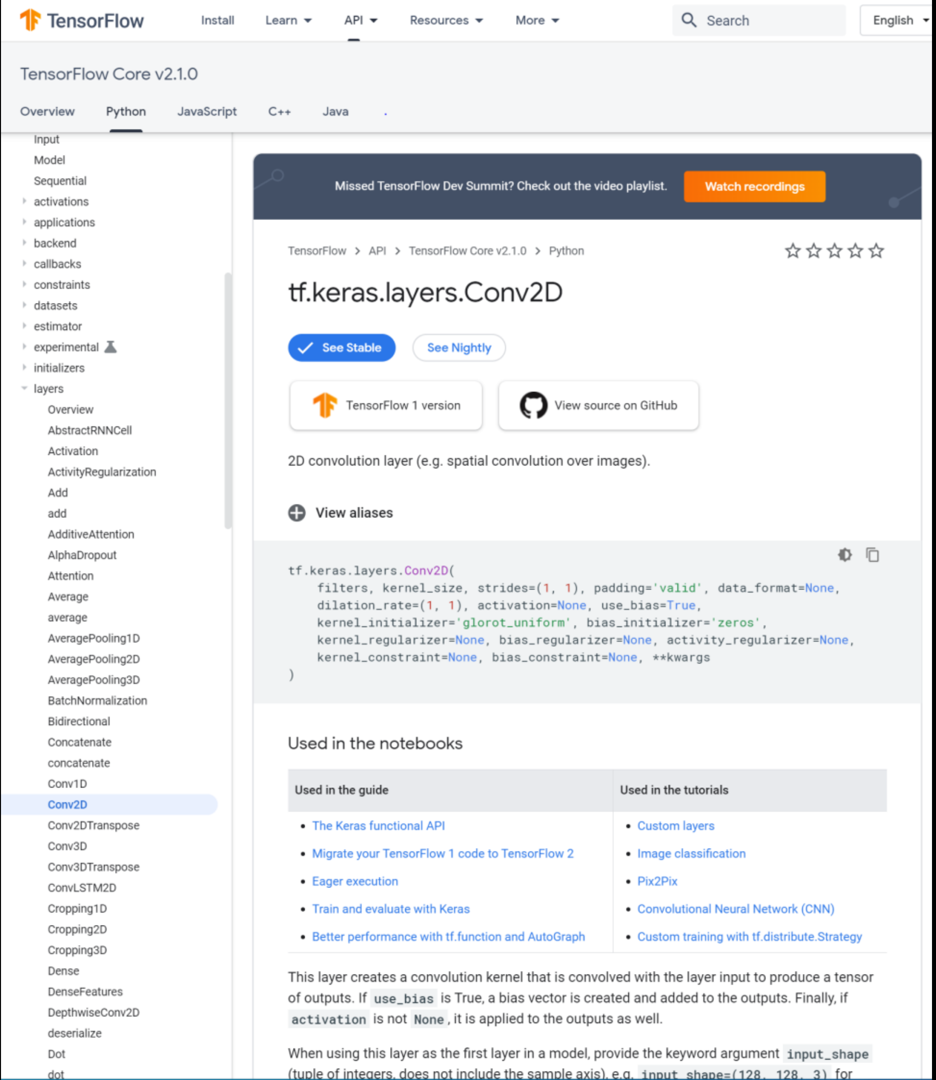
From inside the container, and in the right example directory,run the python programs from the command line:

```
Singularity> python CNN_Dropout.py
```

or invoke them from within the python shell:

```
>>> exec(open("./CNN_Dropout.py").read())
```

Documentation


TensorFlow Install Learn API Resources More Search English

TensorFlow Core v2.1.0

Overview Python JavaScript C++ Java

Input
Model
Sequential
activations
applications
backend
callbacks
constraints
datasets
estimator
experimental
initializers
layers
Overview
AbstractRNNCell
Activation
ActivityRegularization
Add
add
AdditiveAttention
AlphaDropout
Attention
Average
average
AveragePooling1D
AveragePooling2D
AveragePooling3D
BatchNormalization
Bidirectional
Concatenate
concatenate
Conv1D
Conv2D
Conv2DTranspose
Conv3D
Conv3DTranspose
ConvLSTM2D
Cropping1D
Cropping2D
Cropping3D
Dense
DenseFeatures
DepthwiseConv2D
deserialize
Dot
dot

Missed TensorFlow Dev Summit? Check out the video playlist. Watch recordings

TensorFlow > API > TensorFlow Core v2.1.0 > Python

tf.keras.layers.Conv2D

See Stable See Nightly

TensorFlow 1 version View source on GitHub

2D convolution layer (e.g. spatial convolution over images).

View aliases

```
tf.keras.layers.Conv2D(
    filters, kernel_size, strides=(1, 1), padding='valid', data_format=None,
    dilation_rate=(1, 1), activation=None, use_bias=True,
    kernel_initializer='glorot_uniform', bias_initializer='zeros',
    kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
    kernel_constraint=None, bias_constraint=None, **kwargs
)
```

Used in the notebooks

Used in the guide	Used in the tutorials
• The Keras functional API	• Custom layers
• Migrate your TensorFlow 1 code to TensorFlow 2	• Image classification
• Eager execution	• Pix2Pix
• Train and evaluate with Keras	• Convolutional Neural Network (CNN)
• Better performance with tf.function and AutoGraph	• Custom training with tf.distribute.Strategy

This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. If `use_bias` is True, a bias vector is created and added to the outputs. Finally, if `activation` is not `None`, it is applied to the outputs as well.

When using this layer as the first layer in a model, provide the keyword argument `input_shape` (tuple of integers, does not include the sample axis), e.g. `input_shape=(128, 128, 3)` for

The API is well documented.

That is terribly unusual.

Take advantage and keep a browser open as you develop.

MNIST

We now know enough to attempt a problem. Only because the TensorFlow framework, and the Keras API, fills in a lot of the details that we have glossed over. That is one of its functions.

Our problem will be character recognition. We will learn to read handwritten digits by training on a large set of 28x28 greyscale samples.

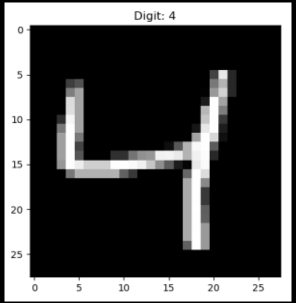
First we'll do this with the simplest possible model just to show how the TensorFlow framework functions. Then we will gradually implement our way to a quite sophisticated and accurate convolutional neural network for this same problem.

Getting Into MNIST

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()


train_images = train_images.reshape(60000, 784)
test_images  = test_images.reshape(10000, 784)

test_images  = test_images.astype('float32')
train_images = train_images.astype('float32')

test_images  /= 255
train_images /= 255
```

matplotlib bonus insight

```
import matplotlib.pyplot as plt

plt.imshow(train_images[2], cmap=plt.get_cmap('gray'),
interpolation='none')
plt.title("Digit: {}".format(train_labels[2]))
```



Defining Our Network

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()


train_images = train_images.reshape(60000, 784)
test_images  = test_images.reshape(10000, 784)

test_images  = test_images.astype('float32')
train_images = train_images.astype('float32')

test_images  /= 255
train_images /= 255


model = tf.keras.Sequential([
  tf.keras.layers.Dense(64, activation='relu', input_shape=(784,)),
  tf.keras.layers.Dense(64, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax'),
])


model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accurac
```

Starting from zero?

In general, initialization values are hard to pin down analytically. Values might help optimization but hurt generalization, or vice versa.

The only certainty is you need to have different values to break the symmetry, or else units in the same layer, with the same inputs, would track each other.

Practically, we just pick some "reasonable" values.

```
model.summary()
_________________________________________________________________
Layer (type)             Output Shape              Param #
=================================================================
dense_6 (Dense)          (None, 64)                50240
_________________________________________________________________
dense_7 (Dense)          (None, 64)                4160
_________________________________________________________________
dense_8 (Dense)          (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
```

Cross Entropy Loss & Softmax

Why Softmax?

The values coming out of our matrix operations can have large, and negative values. We would like our solution vector to be conventional probabilities that sum to 1.0. An effective way to normalize our outputs is to use the popular *Softmax* function. Let's look at an example with just three possible digits:

Digit	Output	Exponential	Normalized
0	4.8	121	.87
1	-2.6	0.07	.00
2	2.9	18	.13

Given the sensible way we have constructed these outputs, the Cross Entropy Loss function is a good way to define the error across all possibilities. Better than squared error, which we have been using until now. It is defined as $-\Sigma$ y_ log y, or if this really is a 0, y_=(1,0,0), and

$$-1\log(0.87) - 0\log(0.0001) - 0\log(0.13) = -\log(0.87) = -0.13$$

You can think that it "undoes" the Softmax, if you want.

Training

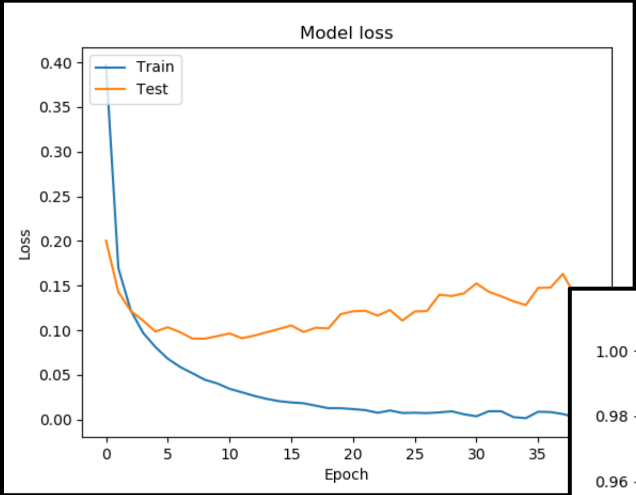
```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()


train_images = train_images.reshape(60000, 784)
test_images  = test_images.reshape(10000, 784)

test_images  = test_images.astype('float32')
train_images = train_images.astype('float32')

test_images  /= 255
train_images /= 255



model = tf.keras.Sequential([
  tf.keras.layers.Dense(64, activation='relu', input_shape=(784,)),
  tf.keras.layers.Dense(64, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax'),
])


model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])



history = model.fit(train_images, train_labels, batch_size=128, epochs=40, verbose=1, validation_data=(test_images, test_labels))
```

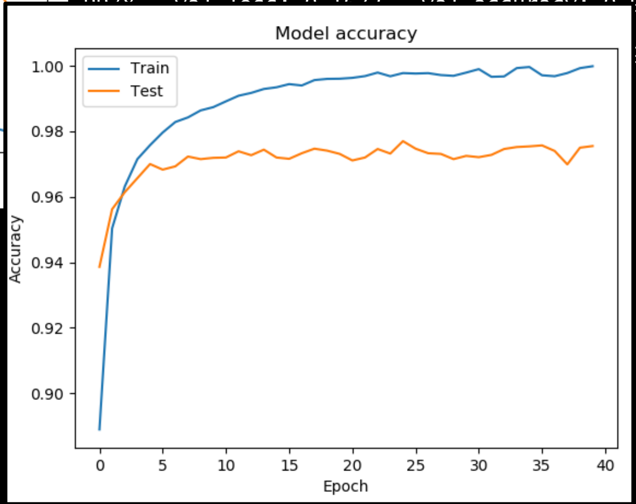
Results

matplotlib bonus insight

```
history = model.fit(train_images, ..., ...)

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper
plt.show()
```

```
/sample - loss: 0.3971 - accuracy: 0.8889 - val_loss: 0.2003 - val_accuracy: 0.9386
/sample - loss: 0.1696 - accuracy: 0.9503 - val_loss: 0.1430 - val_accuracy: 0.9562
.9631 - val_loss: 0.1218 - val_accuracy: 0.9614
9715 - val_loss: 0.1109 - val_accuracy: 0.9657
.9758 - val_loss: 0.0986 - val_accuracy: 0.9700
.9796 - val_loss: 0.1035 - val_accuracy: 0.9683
9699
9750
0.9755
```





Let's Go Wider

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape(60000, 784)
test_images  = test_images.reshape(10000, 784)

test_images  = test_images.astype('float32')
train_images = train_images.astype('float32')

test_images  /= 255
train_images /= 255

model = tf.keras.Sequential([
  tf.keras.layers.Dense(512, activation='relu', input_shape=(784,)),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax'),
])


model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])


model.fit(train_images, train_labels, batch_size=128, epochs=30, verbose=1, validation_data=(test_images, test_labels))
```

Wider Results

```
....
....
Epoch 30/30
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0083 - accuracy: 0.9977 - val_loss: 0.1027 - val_accuracy: 0.9821
```

Wider

```
model.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_18 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_19 (Dense)             (None, 512)               262656
_________________________________________________________________
dense_20 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

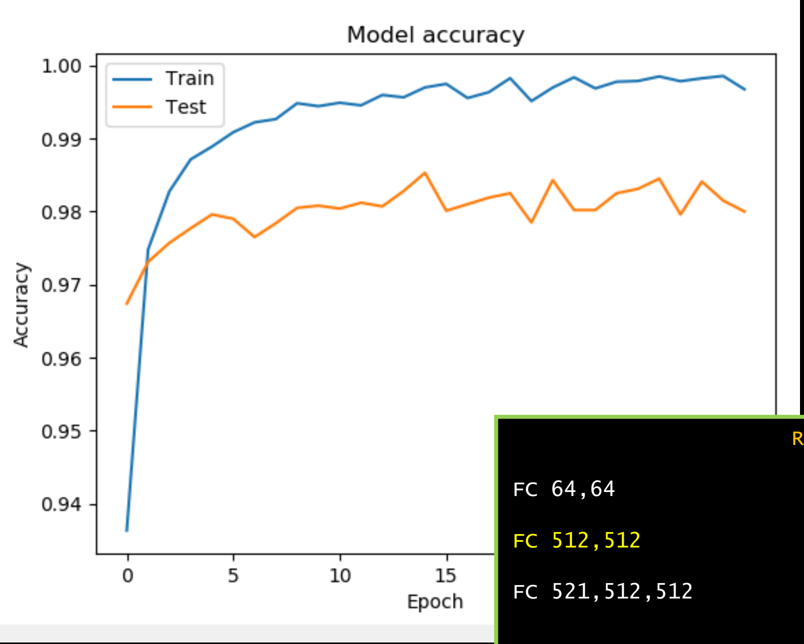
55,050 for 64 Wide Model

Maybe Deeper?

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape(60000, 784)
test_images  = test_images.reshape(10000, 784)

test_images  = test_images.astype('float32')
train_images = train_images.astype('float32')

test_images  /= 255
train_images /= 255

model = tf.keras.Sequential([
  tf.keras.layers.Dense(512, activation='relu', input_shape=(784,)),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax'),
])


model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])


model.fit(train_images, train_labels, batch_size=128, epochs=30, verbose=1, validation_data=(test_images, test_labels))
```

Wide And Deep Results

```
....
....
60000/60000 [==============================] - 3s 45us/sample - loss: 0.0119 - accuracy: 0.9967 - val_loss: 0.1183 - val_accuracy: 0.9800
```



Deep and wide

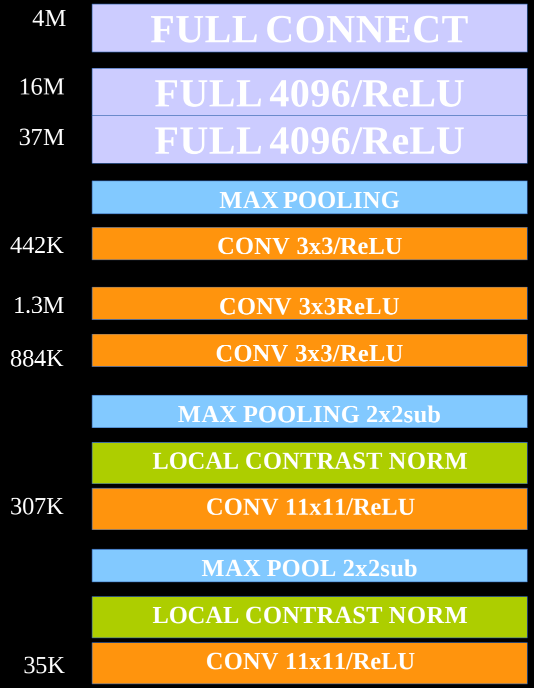
```
model.summary()
_________________________________________________
Layer (type)          Output Shape          Param #
=================================================
dense_24 (Dense)      (None, 512)           401920
_________________________________________________
dense_25 (Dense)      (None, 512)           262656
_________________________________________________
dense_26 (Dense)      (None, 512)           262656
_________________________________________________
dense_27 (Dense)      (None, 10)            5130
=================================================
Total params: 932,362
s: 932,362
arams: 0
```

Recap

FC 64,64	97.5
FC 512,512	98.2
FC 521,512,512	98.0

Image Recognition Done Right: CNNs

AlexNet won the 2012 ImageNet LSVRC and changed the DL world.



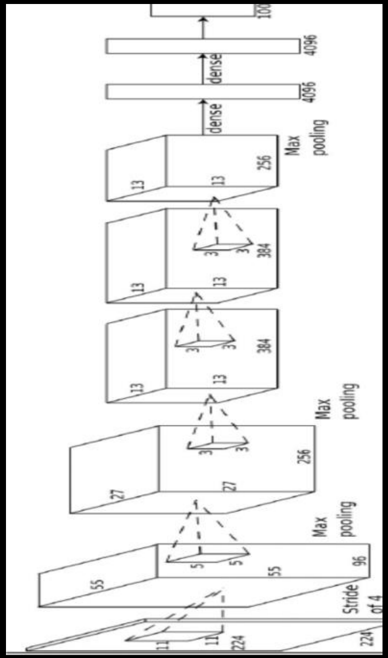
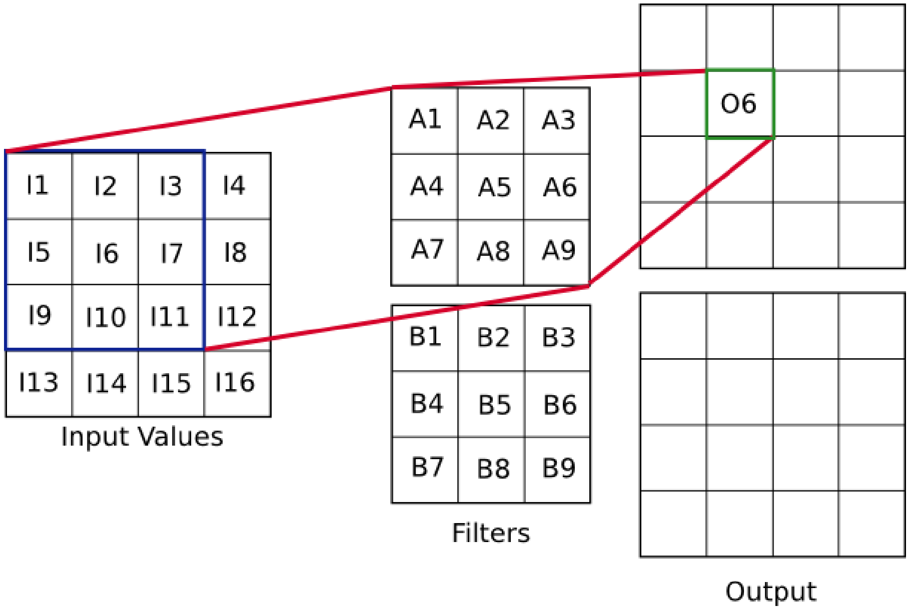


Image Object Recognition [Krizhevsky, Sutskever, Hinton 2012]

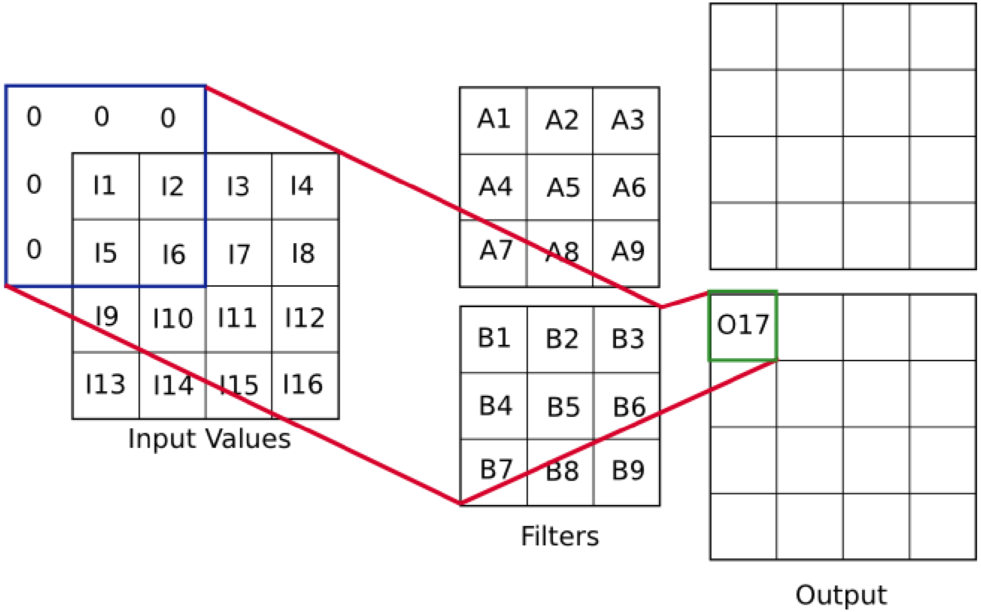
Convolution



$$\begin{aligned} O_6 = A_1 \cdot I_1 + A_2 \cdot I_2 + A_3 \cdot I_3 \\ + A_4 \cdot I_5 + A_5 \cdot I_6 + A_6 \cdot I_7 \\ + A_7 \cdot I_9 + A_8 \cdot I_{10} + A_9 \cdot I_{11} \end{aligned}$$

Convolution

Boundary and Index Accounting



$$O_{17} = B_5 \cdot I_1 + B_6 \cdot I_2 + B_8 \cdot I_5 + B_9 \cdot I_6$$

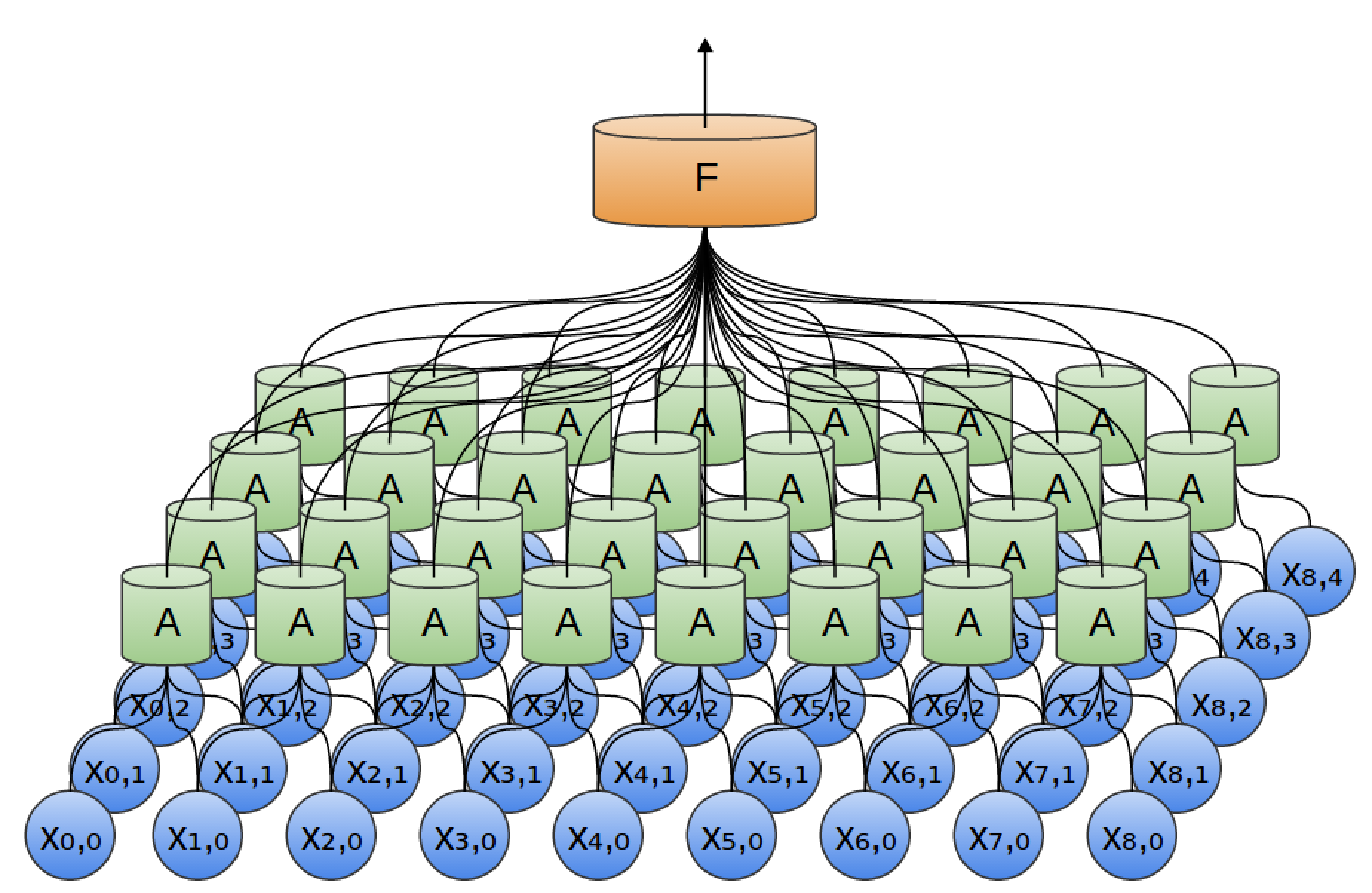
Straight Convolution

$$+ \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} =$$

Edge Detector

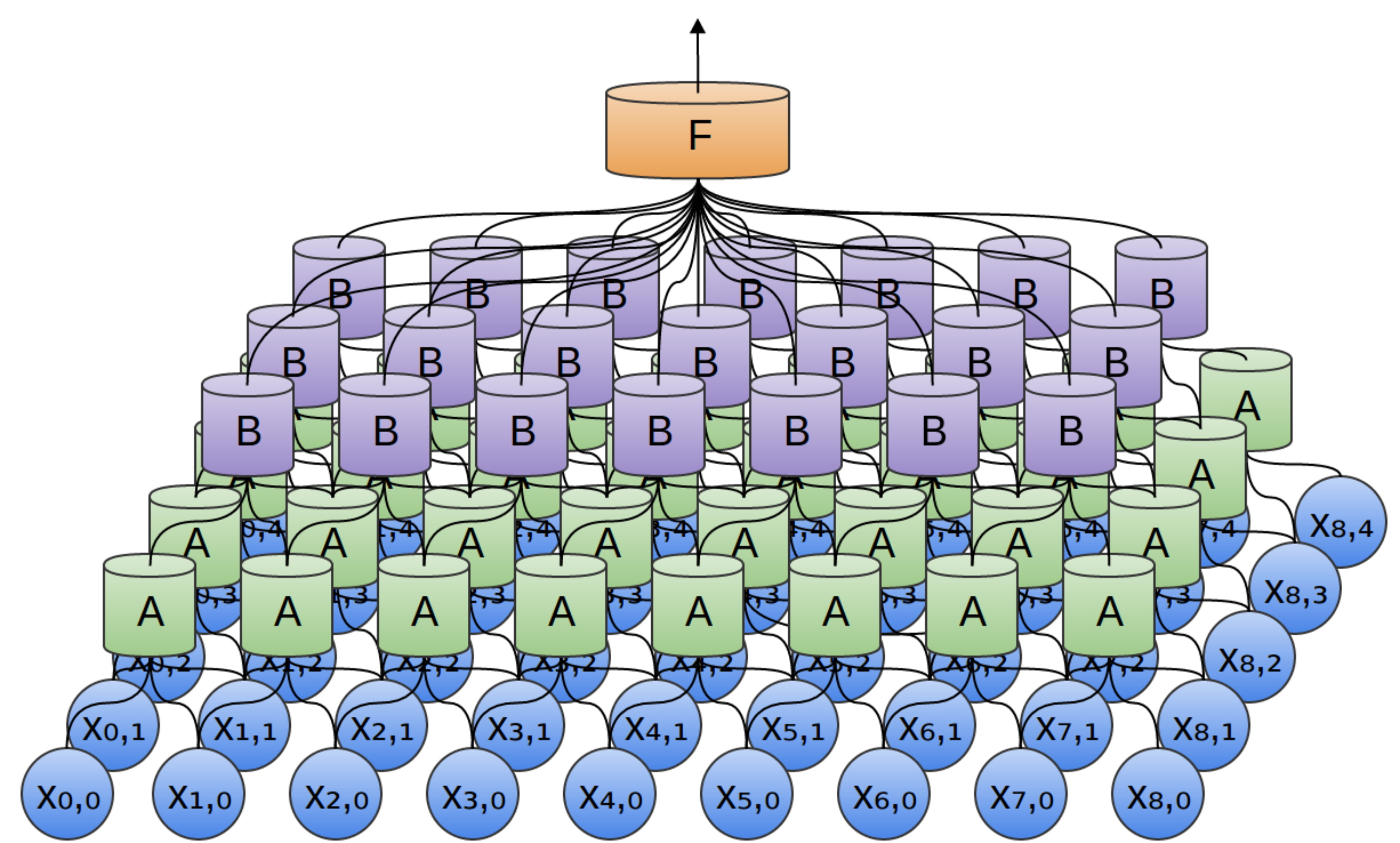
Images: Wikipedia

Simplest Convolution Net



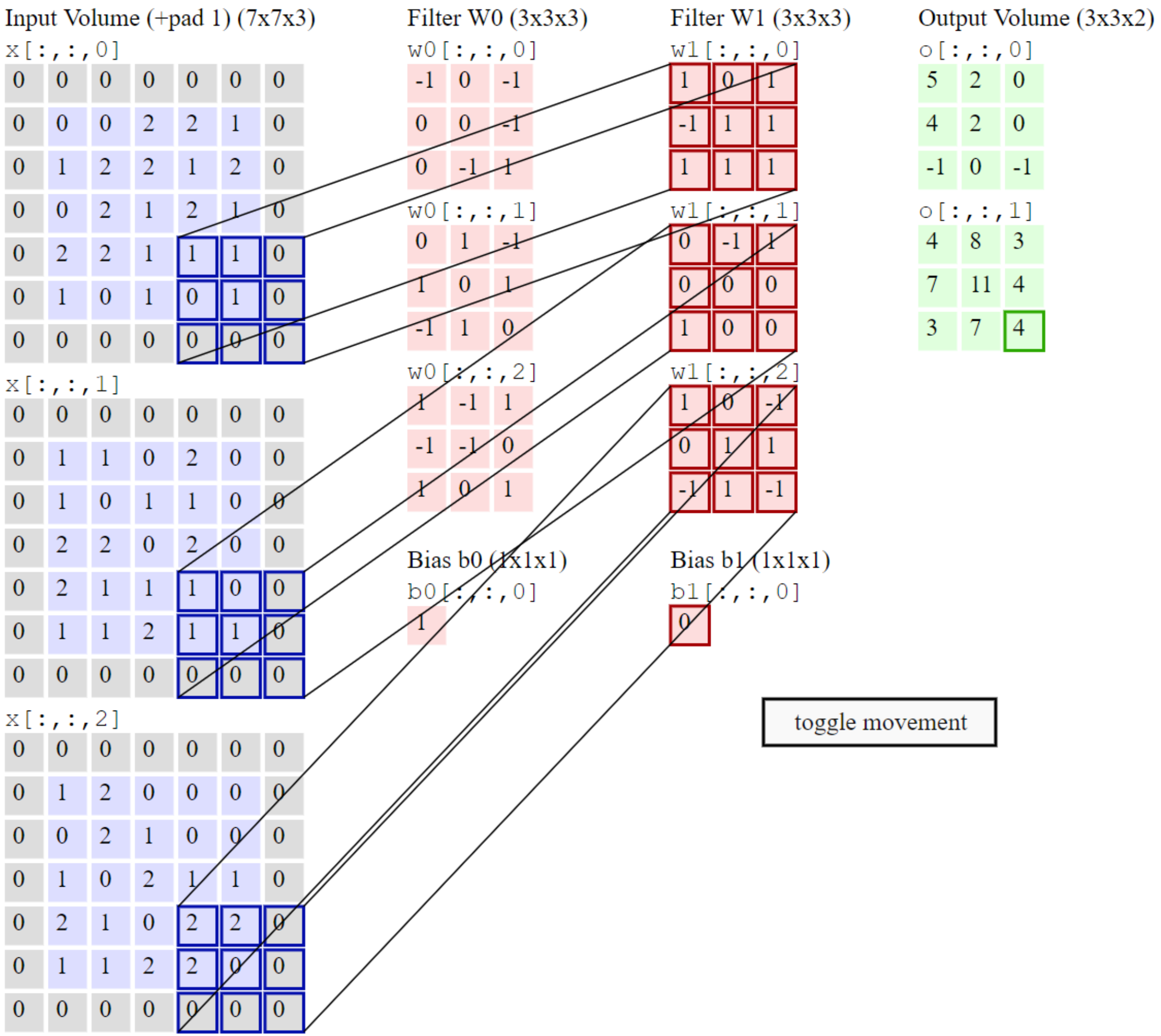
Courtesy: Chris Olah

Stacking Convolutions



Courtesy: Chris Olah

Convolution



From the very nice Stanford CS231n course at http://cs231n.github.io/convolutional-networks/

Stride = 2

Convolution Math

Each Convolutional Layer:

Inputs a volume of size $W_I \times H_I \times D_I$ (D is depth)

Requires four hyperparameters:

Number of filters K

their spatial extent N

the stride S

the amount of padding P

Produces a volume of size $W_O \times H_O \times D_O$

$W_O = (W_I - N + 2P) / S+1$

$H_O = (H_I - F + 2P) / S+1$

$D_O = K$

This requires $N \cdot N \cdot D_I$ weights per filter, for a total of $N \cdot N \cdot D_I \cdot K$ weights and K biases

In the output volume, the d-th depth slice (of size $W_O \times H_O$) is the result of performing a convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.