



# Deep Learning: Hands On

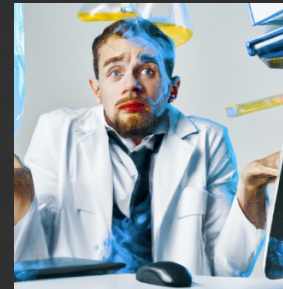
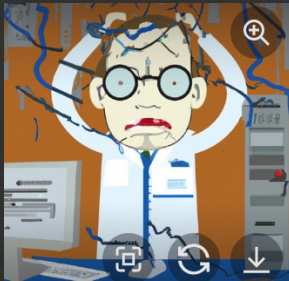
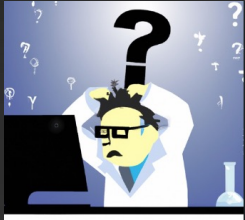
Bryon Gill  
Pittsburgh Supercomputing Center

# Our particular motivation



*Is this you?*

- Machine learning in the sciences went from “who cares” 10 years ago, to many cases of “this works way better than anything else” today.
- Meanwhile, all the knowledge is still walled off in the CS community. Usually in semester long courses.
- This panic exciting situation has left a huge knowledge gap for practicing scientists at all levels.
- So here we are.



# Unprecedented Disruption

In the history of science, I defy you to find a similarly quick paradigm shift.

*10 years ago*

“Neural nets will enable real time ray tracing.”

“Neural nets will do protein folding.”

Science Fiction.

Word salad.

*5 years ago*

“Neural nets will do CFD.”

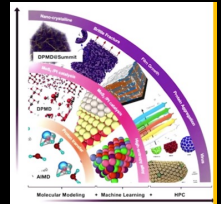
Well, maybe someday, but not soon.

*Today*

Neural net enabled algorithms are the best way to do protein folding.

*Tomorrow*

Skynet will kill us all. Or at least steal our jobs.



# Why Now?

The ideas have been around for decades. Two components came together in the past 15 years to enable astounding progress:

Widespread parallel computing (GPUs)



Big data training sets





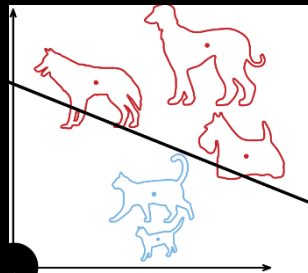
# Two Perspectives

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

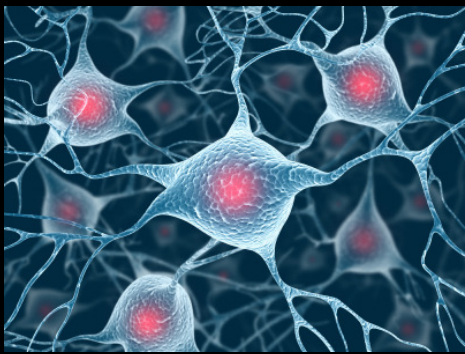
- Inspired by biological models.



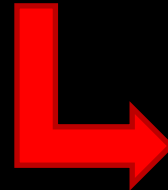
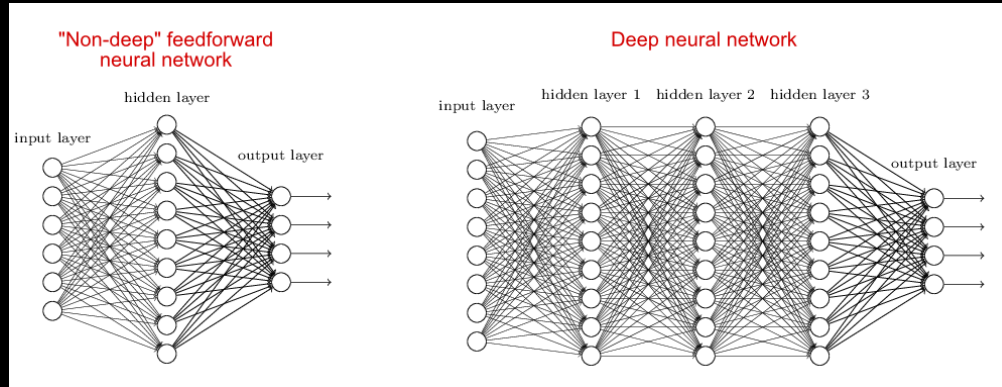
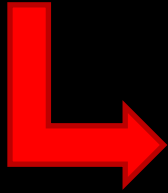
- An evolution of classic ML techniques (the perceptron).



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.



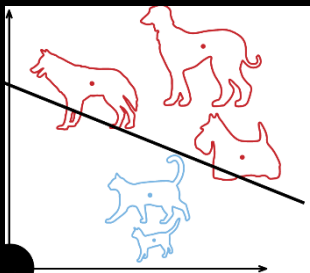
# Modeled After The Brain



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

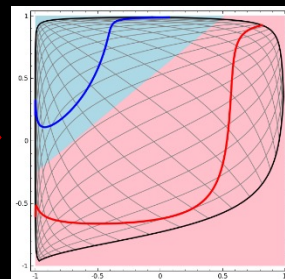
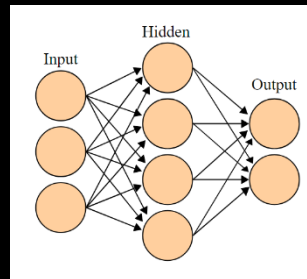
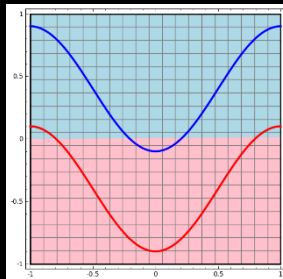
# As a Highly Dimensional Non-linear Classifier

## Perceptron



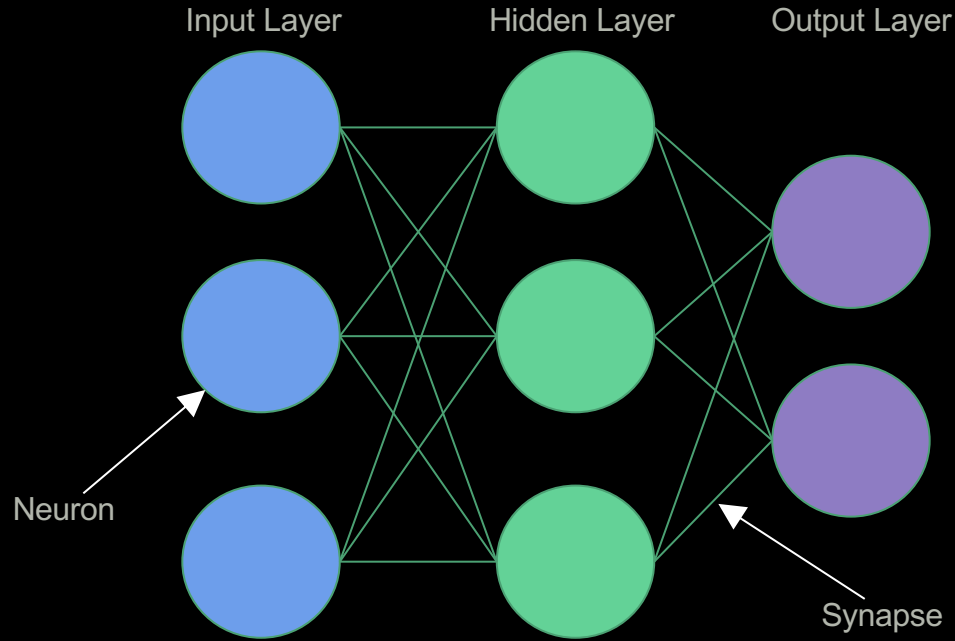
No Hidden Layer  
Linear

## Network



Hidden Layers  
Nonlinear

# Basic NN Architecture

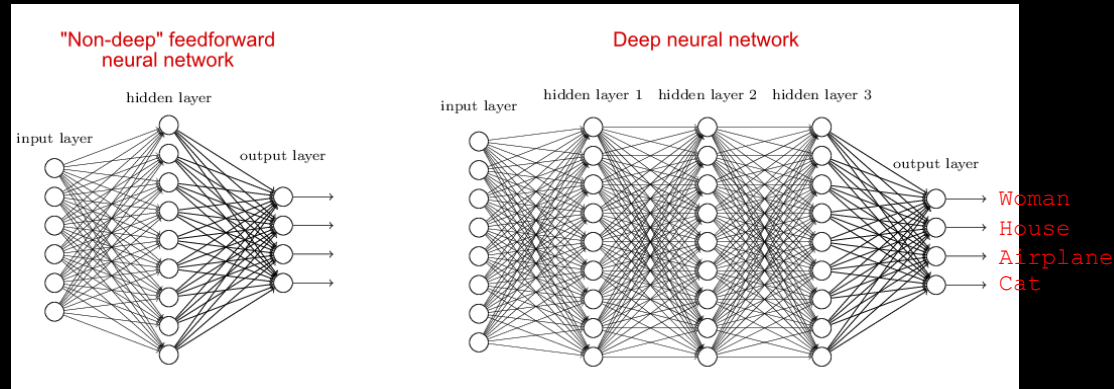


# In Practice

How many inputs?



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



How deep?

100+ layers have become common.

How many outputs?

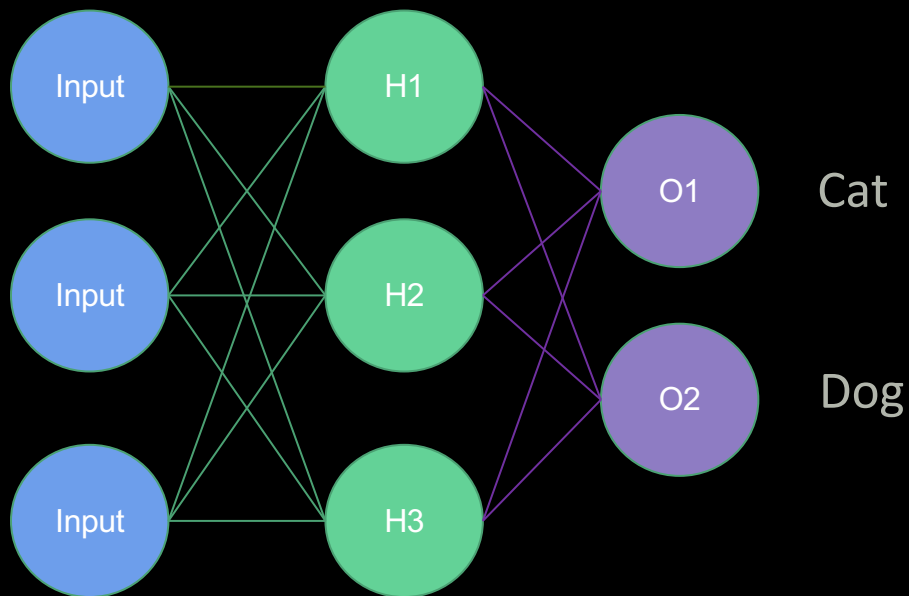


Might be an entire image.

Or could be discreet set of classification possibilities.

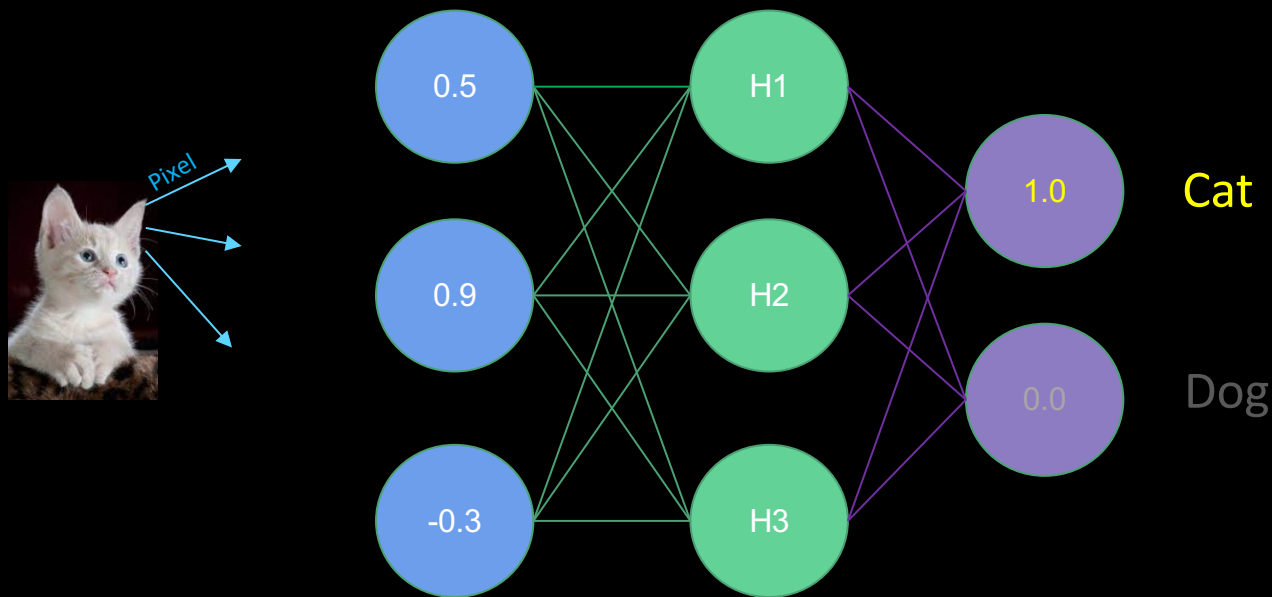
# Inference

The "forward" or thinking step



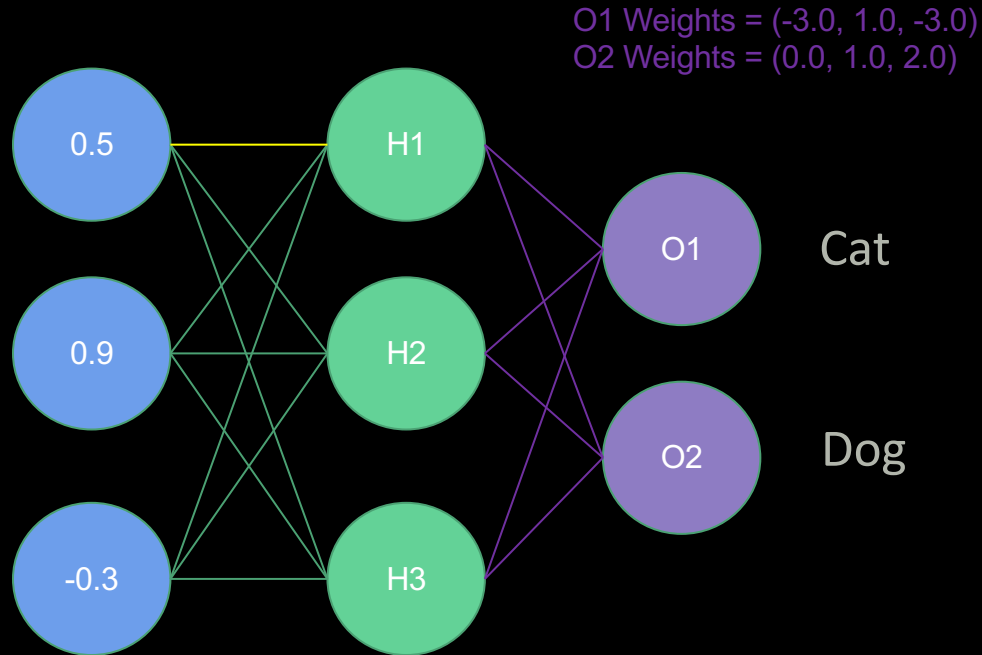
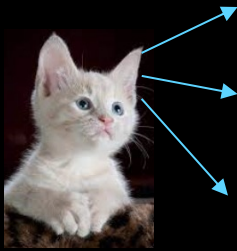
# Inference

Input and Output Layers



# Inference

Weights or Parameters



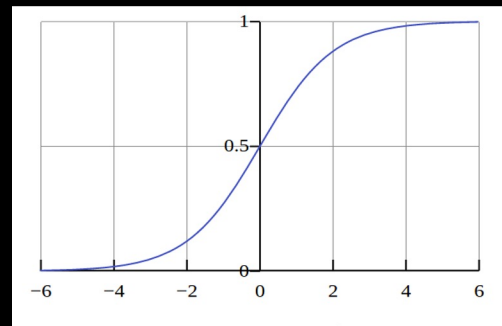
H1 Weights = (1.0, -2.0, 2.0)  
H2 Weights = (2.0, 1.0, -4.0)  
H3 Weights = (1.0, -1.0, 0.0)



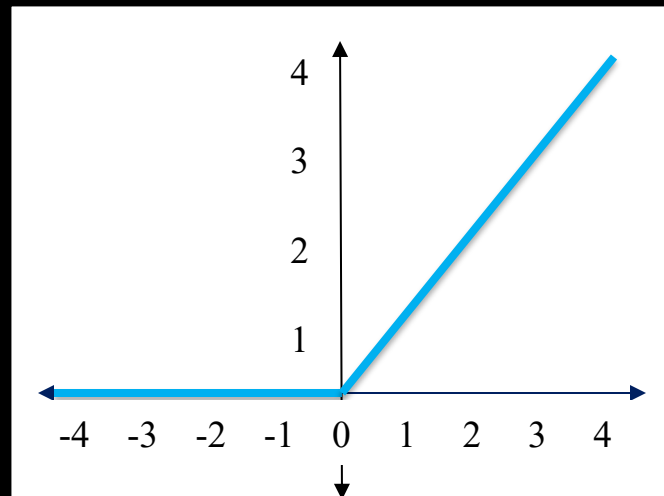
# Activation Function

Neurons apply activation functions at these summed inputs. Activation functions are typically non-linear. There are countless possibilities. In reality, there are really only a few popular families:

- 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.
- 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, maybe addition and multiplication for variants.
  - There are **Leaky** and **Noisy** variants.

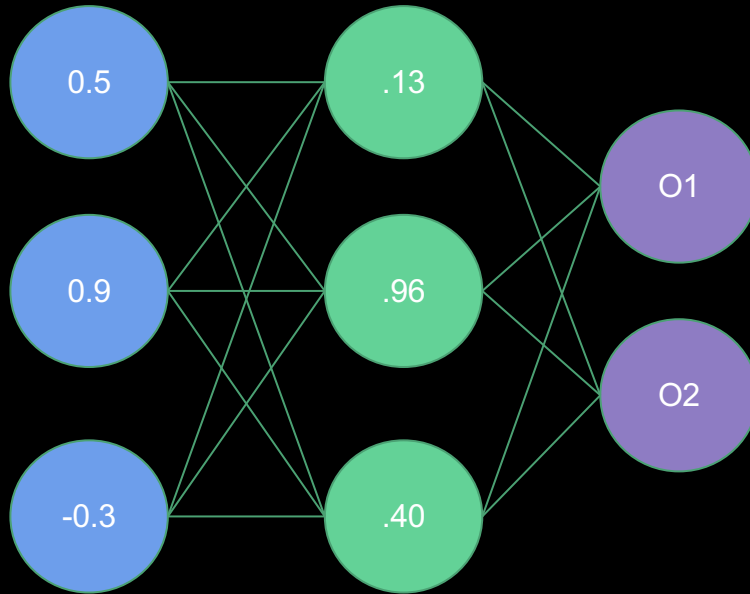


$$S(t) = \frac{1}{1 + e^{-t}}$$



# Inference

Multiply, Add, do something non-linear.



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)

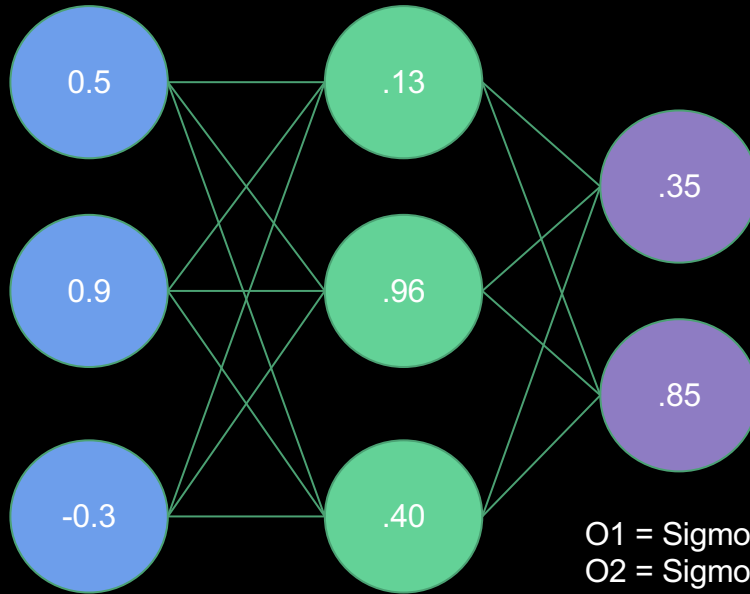
$$H1 = \text{Sigmoid}(0.5 * 1.0 + 0.9 * -2.0 + -0.3 * 2.0) = \text{Sigmoid}(-1.9) = .13$$

$$H2 = \text{Sigmoid}(0.5 * 2.0 + 0.9 * 1.0 + -0.3 * -4.0) = \text{Sigmoid}(3.1) = .96$$

$$H3 = \text{Sigmoid}(0.5 * 1.0 + 0.9 * -1.0 + -0.3 * 0.0) = \text{Sigmoid}(-0.4) = .40$$

# Inference

Then do it again.



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)

$$O1 = \text{Sigmoid}(.13 * -3.0 + .96 * 1.0 + .40 * -3.0) = \text{Sigmoid}(-.63) = .35$$

$$O2 = \text{Sigmoid}(.13 * 0.0 + .96 * 1.0 + .40 * 2.0) = \text{Sigmoid}(1.76) = .85$$

# 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)

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

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*:

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

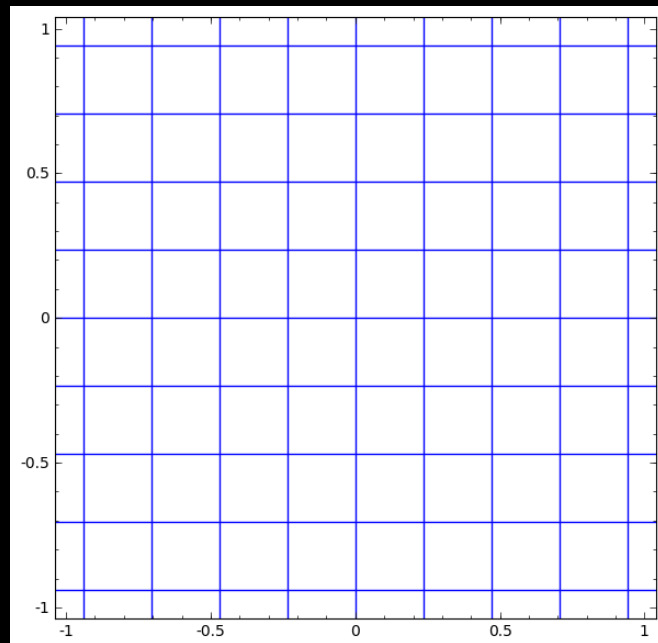
# Linear + Nonlinear

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*.

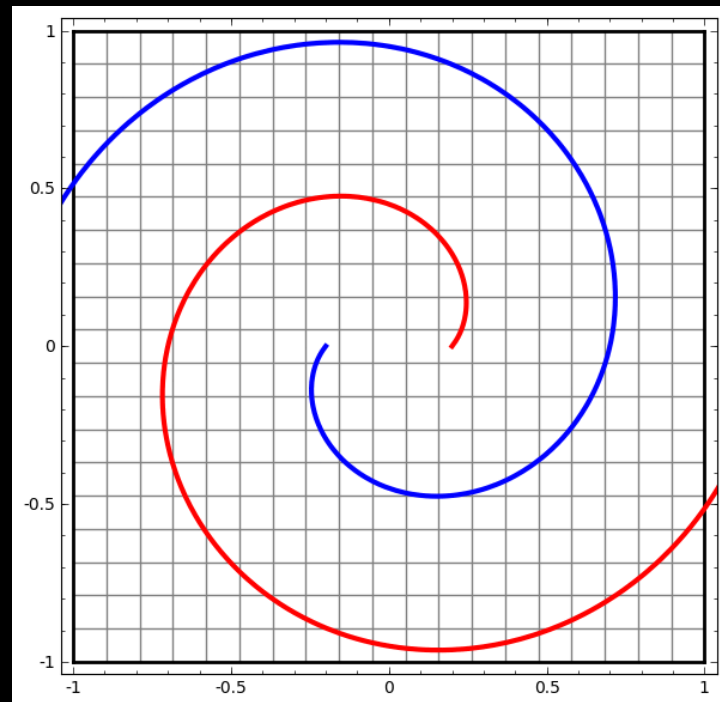
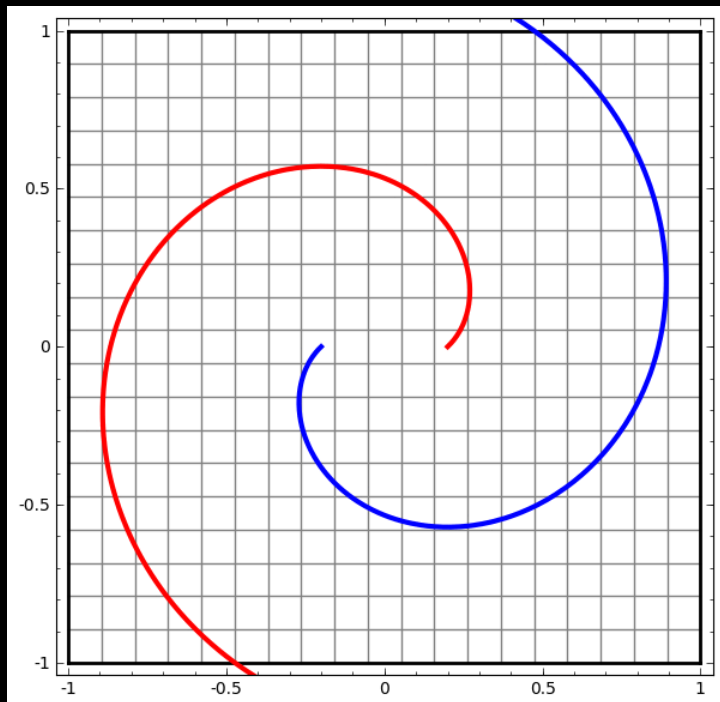
The bias provides the *translation*.

The activation function provides the *warp*.



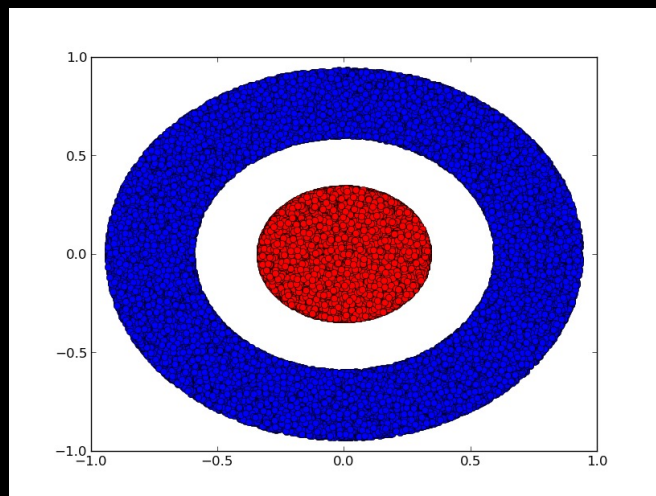
# Linear + Nonlinear

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



# 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:

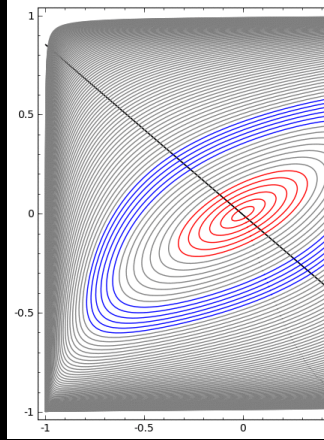


Can a neural net do this with twisting and deforming? What good does it do to have more than two dimensions with a 2D dataset?

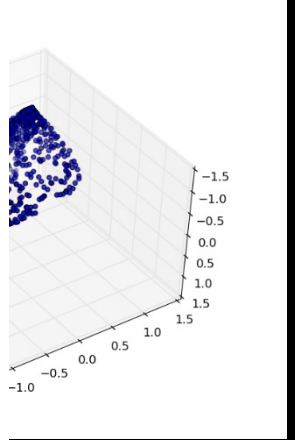
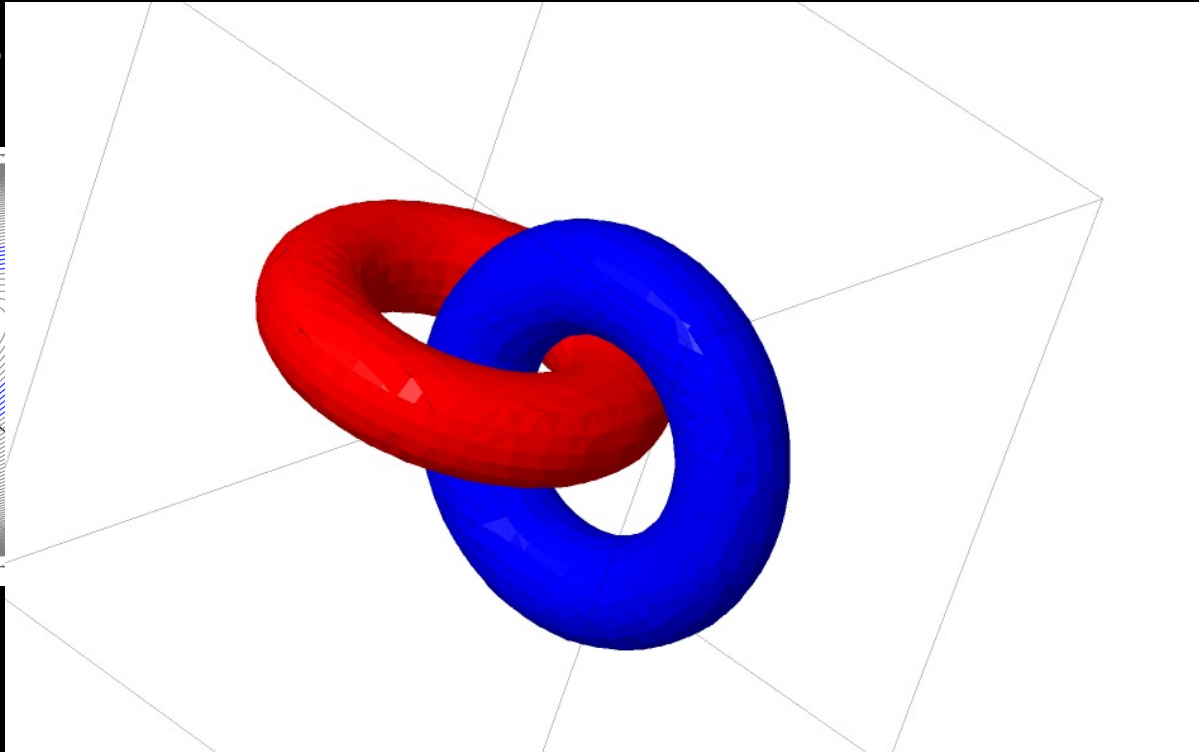


# Working In Higher Dimensions

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.

# Theoretically

*Universal Approximation Theorem:* A 1-hidden-layer feedforward network of this type can approximate any function<sup>1</sup>, given enough width<sup>2</sup>.

Not really that useful as:

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

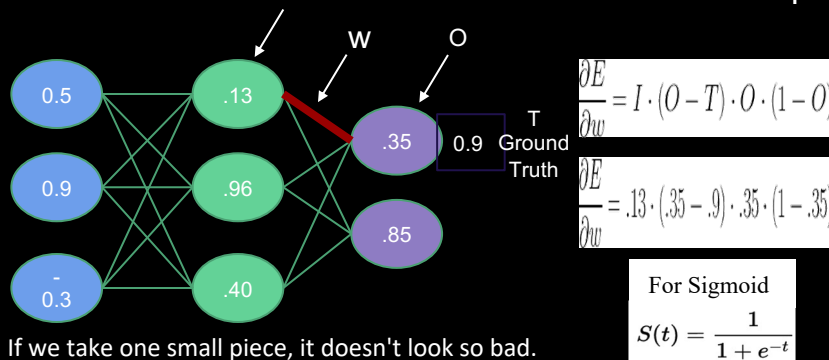
1) Borel measurable. Basically, mostly continuous and bounded.

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.

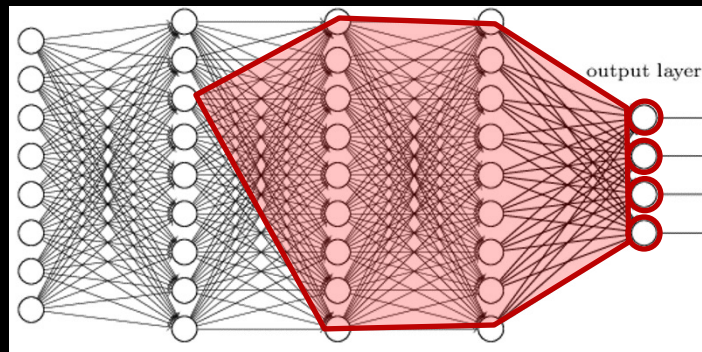
We know that the output is a function of the weights:  $E(w_1, w_2, w_3, \dots, i_1, \dots, t_1, \dots)$ . So to figure out which way, and how much, to push any particular weight, say  $w_3$ , we want to calculate  $\frac{\partial E}{\partial w_3}$



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.

# Back-Propagation

In a useful network, the chain rule results in a lot of factors for any given weight adjustment.



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.

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\text{path}(u^{(\pi_1)}, u^{(\pi_2)}, \dots, u^{(\pi_t)}, \text{ from } \pi_1=j \text{ to } \pi_t=n)} \prod_{k=2}^t \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}.$$

From the fantastic *Deep Learning*, Goodfellow, Bengio and Courville.

Since the number of paths from one node to a distant node can grow exponentially in the length of these paths, the number of terms in the above sum, which is the number of such paths, can grow exponentially with depth. A large cost would be incurred because the same computation for the subfactors would be redone many times. To avoid such recomputation, back-propagation works as a table-filling algorithm that stores intermediate results and avoids repeating many common subexpressions.

# Back-propagation Full Story

If you have 30 minutes, and remember freshman calculus, you can understand the complete details of the algorithm. I heartily recommend one of these.

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> .

To be honest, many people are happy to leave the details to TensorFlow, or whatever package they are using. Just don't think it is beyond your understanding.

# Solvers

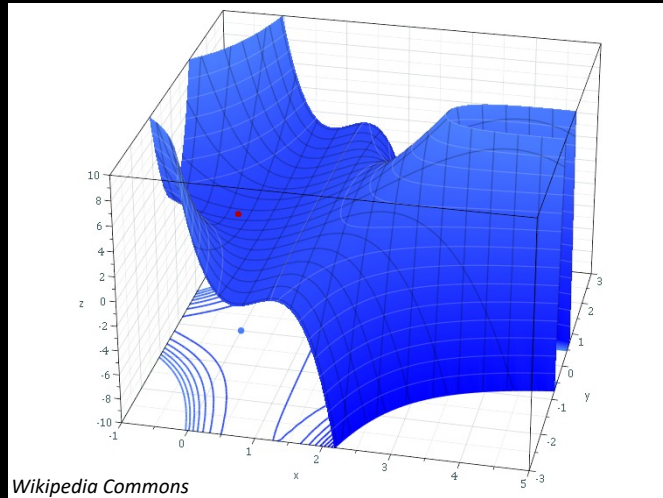
However, even this efficient technique leaves us with potentially many millions of simultaneous equations to solve (real nets have a lot of weights). And the solution space is non-convex. Fortunately, this isn't a new problem created by deep learning, so we have options from the world of numerical methods.

The standard has been *gradient descent*. Variations of this 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* (or *learning rate*) are the two *hyperparameters* we will encounter later.

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

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 -R BigGPUJul11  
v001%
```

Load the TensorFlow 2 Container:

```
v001% singularity shell --nv /ocean/containers/ngc/tensorflow/tensorflow_23.04-tf2-py3.sif
```

And start TensorFlow:

```
Singularity> python  
Python 3.8.10 (default, Mar 13 2023, 10:26:41)  
[GCC 9.4.0] 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())
```

TensorFlow

Install Learn API Resources More

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
<ul style="list-style-type: none"><li><a href="#">The Keras functional API</a></li><li><a href="#">Migrate your TensorFlow 1 code to TensorFlow 2</a></li><li><a href="#">Eager execution</a></li><li><a href="#">Train and evaluate with Keras</a></li><li><a href="#">Better performance with tf.function and AutoGraph</a></li></ul>	<ul style="list-style-type: none"><li><a href="#">Custom layers</a></li><li><a href="#">Image classification</a></li><li><a href="#">Pix2Pix</a></li><li><a href="#">Convolutional Neural Network (CNN)</a></li><li><a href="#">Custom training with tf.distribute.Strategy</a></li></ul>

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

# Documentation

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)

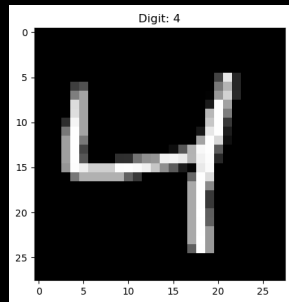
test_images = test_images.astype('float32')
train_images = train_images.astype('float32')

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'),
])
```

## 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		

# 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

# Solving For Weights

```
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'])
```

# Defining our error

In ML, defining the *error* (or *loss*, or *cost*) is often the core of defining the objective solution. Once we define the error, we can usually plug it into a canned solver which can minimize it. Defining the error can be obvious, or very subtle, or have multiple acceptable methods.

**Clustering:** For k-means we simply used the geometrical distance. It was actually the sum of the squared distances, but you get the idea.

**Image Recognition:** If our algorithm tags a picture of a cat as a dog, is that a larger error than if it tags it as a horse? Or a car? How would you quantify these?

How about if our self driving car mistakes a crosswalk for an on-ramp?!

**Regression:** Do you want to penalize a lot of medium errors more than an occasional large error? If you are predicting stock prices, you most likely care more about the average, and an occasional bad call is OK. If you are projecting drug doses, that large error could kill!

**Rare events:** Want to make a 99% accurate tornado warning algorithm? Just put a piece of paper up saying "No tornado today." How do you weigh your error to deal with the significance of false negatives or positives?

# GPT-4

Many emerging applications can be challenging to quantify. Some of the things GPT-4 does are quite easy to grade. Literally.

Exam	GPT-4	GPT-4 (no vision)	GPT-3.5
Uniform Bar Exam (MBE+MEE+MPT)	298 / 400 (~90th)	298 / 400 (~90th)	213 / 400 (~10th)
LSAT	163 (~88th)	161 (~83rd)	149 (~40th)
SAT Evidence-Based Reading & Writing	710 / 800 (~93rd)	710 / 800 (~93rd)	670 / 800 (~87th)
SAT Math	700 / 800 (~89th)	690 / 800 (~89th)	590 / 800 (~70th)
Graduate Record Examination (GRE) Quantitative	163 / 170 (~80th)	157 / 170 (~62nd)	147 / 170 (~25th)
Graduate Record Examination (GRE) Verbal	169 / 170 (~99th)	165 / 170 (~96th)	154 / 170 (~63rd)
Graduate Record Examination (GRE) Writing	4 / 6 (~54th)	4 / 6 (~54th)	4 / 6 (~54th)
USABO Semifinal Exam 2020	87 / 150 (99th - 100th)	87 / 150 (99th - 100th)	43 / 150 (31st - 33rd)
USNCO Local Section Exam 2022	36 / 60	38 / 60	24 / 60
Medical Knowledge Self-Assessment Program	75 %	75 %	53 %
Codeforces Rating	392 (below 5th)	392 (below 5th)	260 (below 5th)
AP Art History	5 (86th - 100th)	5 (86th - 100th)	5 (86th - 100th)
AP Biology	5 (85th - 100th)	5 (85th - 100th)	4 (62nd - 85th)
AP Calculus BC	4 (43rd - 59th)	4 (43rd - 59th)	1 (0th - 7th)
AP Chemistry	4 (71st - 88th)	4 (71st - 88th)	2 (22nd - 46th)
AP English Language and Composition	2 (14th - 44th)	2 (14th - 44th)	2 (14th - 44th)
AP English Literature and Composition	2 (8th - 22nd)	2 (8th - 22nd)	2 (8th - 22nd)
AP Environmental Science	5 (91st - 100th)	5 (91st - 100th)	5 (91st - 100th)
AP Macroeconomics	5 (84th - 100th)	5 (84th - 100th)	2 (33rd - 48th)
AP Microeconomics	5 (82nd - 100th)	4 (60th - 82nd)	4 (60th - 82nd)
AP Physics 2	4 (66th - 84th)	4 (66th - 84th)	3 (30th - 66th)
AP Psychology	5 (83rd - 100th)	5 (83rd - 100th)	5 (83rd - 100th)
AP Statistics	5 (85th - 100th)	5 (85th - 100th)	3 (40th - 63rd)
AP US Government	5 (88th - 100th)	5 (88th - 100th)	4 (77th - 88th)
AP US History	5 (89th - 100th)	4 (74th - 89th)	4 (74th - 89th)
AP World History	4 (65th - 87th)	4 (65th - 87th)	4 (65th - 87th)
AMC 10 <sup>3</sup>	30 / 150 (6th - 12th)	36 / 150 (10th - 19th)	36 / 150 (10th - 19th)
AMC 12 <sup>3</sup>	60 / 150 (45th - 66th)	48 / 150 (19th - 40th)	30 / 150 (4th - 8th)
Introductory Sommelier (theory knowledge)	92 %	92 %	80 %
Certified Sommelier (theory knowledge)	86 %	86 %	58 %
Advanced Sommelier (theory knowledge)	77 %	77 %	46 %
Leetcode (easy)	31 / 41	31 / 41	12 / 41
Leetcode (medium)	21 / 80	21 / 80	8 / 80
Leetcode (hard)	3 / 45	3 / 45	0 / 45

**Table 1.** GPT performance on academic and professional exams. In each case, we simulate the conditions and scoring of the real exam. We report GPT-4's final score graded according to exam-specific rubrics, as well as the percentile of test-takers achieving GPT-4's score.

# How do we even?

But some of the capabilities, especially the *generative* ones, defy simple quantification.

User

What is funny about this image? Describe it panel by panel.



Source: <https://www.reddit.com/r/hmmm/comments/ubab5v/hmmm/>

GPT-4

The image shows a package for a "Lightning Cable" adapter with three panels.

Panel 1: A smartphone with a VGA connector (a large, blue, 15-pin connector typically used for computer monitors) plugged into its charging port.

Panel 2: The package for the "Lightning Cable" adapter with a picture of a VGA connector on it.

Panel 3: A close-up of the VGA connector with a small Lightning connector (used for charging iPhones and other Apple devices) at the end.

The humor in this image comes from the absurdity of plugging a large, outdated VGA connector into a small, modern smartphone charging port.



write a poem about traffic lights



In the heart of a bustling city's night,  
A sentinel stands, glowing with light,  
With colors of red, yellow, and green,  
A triad of hues, a silent machine.

A beacon of order midst chaos and speed,  
Guiding the masses, an unspoken creed,  
Whispers of safety, a language of hue,  
The traffic lights speak, their purpose imbued.

Red, a guardian with a stern, watchful eye,

Perhaps most interesting,  
some of these capabilities  
weren't entirely anticipated,  
and were *emergent*!

In unity they stand, those traffic light sentries,  
A dance of transitions, a cycle of entry,  
And so, in the chaos, a pattern ignites,  
The rhythm of life, in colors of traffic lights.

Paint a photo of Disneyland in the style of Vincent Van Gogh



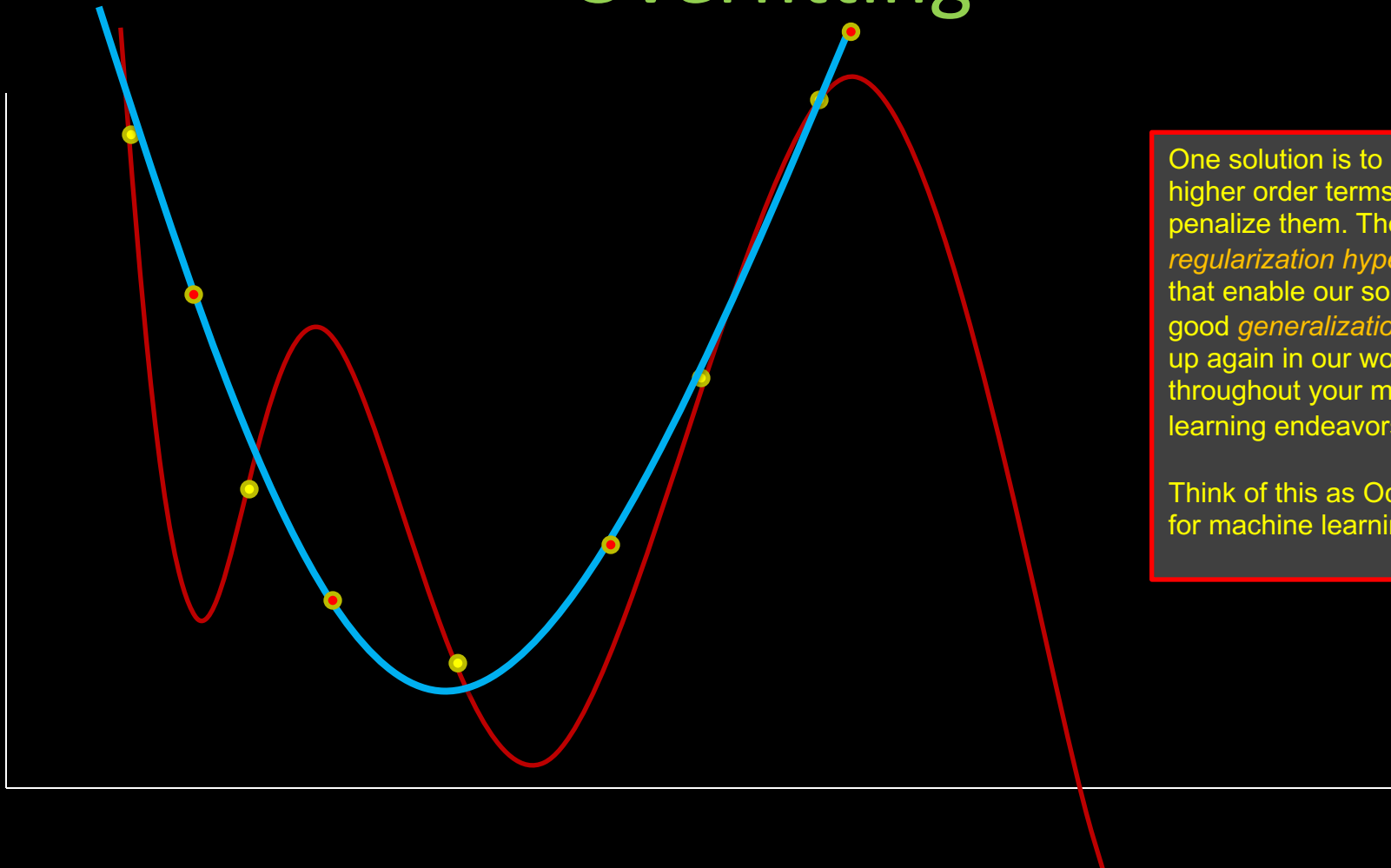
Do we need a comedian with a calculator?

Or a poetry reviewer?

Or an art critic?



# Overfitting



One solution is to keep using higher order terms, but to penalize them. These *regularization hyperparameters* that enable our solution to have good *generalization* will show up again in our workshop, and throughout your machine learning endeavors.

Think of this as Occam's Razor for machine learning.

# Cross Entropy

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  $-\sum y_- \log y$ , or if this really is a "0",  $y_=(1,0,0)$ , and

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

It somewhat penalizes a slightly wrong guess, or an "unconfident" right guess, and greatly penalizes a very wrong guess.

You can also 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

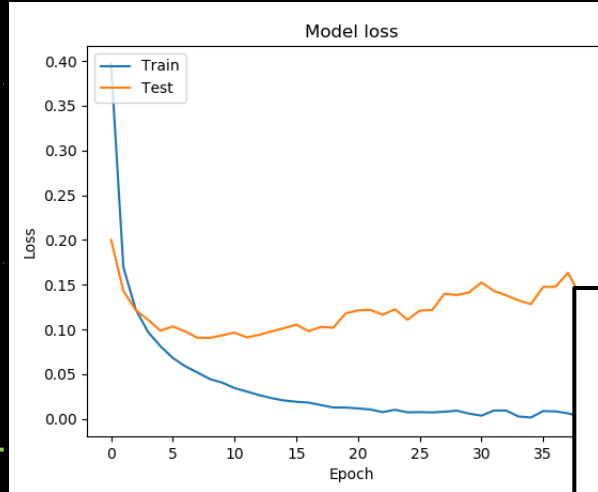
```
history = model.fit(train_images, ..., ...)
```

```
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
```

```
right')
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
right')
plt.show()
```

/sample - loss: 0.3971 - accuracy:

/sample - loss: 0.1696 - accuracy:



why would the test accuracy *ever* be better than the training (as momentarily happens here)?

The training value is the average over each batch, and the test value is only at the end of the epoch, when the model tends to be at least slightly better.

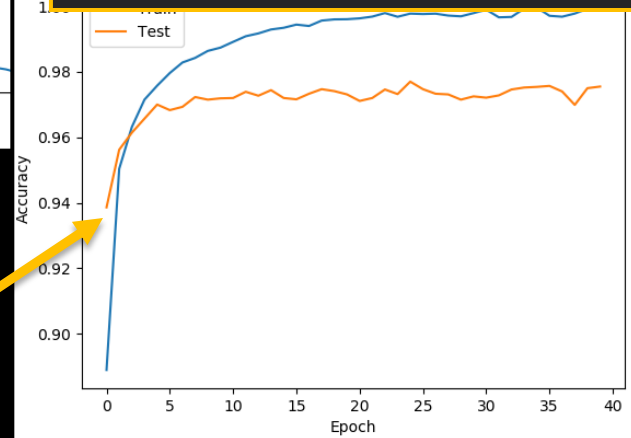
Later on we will see that regularization techniques (which are only turned on for training) also add to this effect.

## Accuracy or Loss?

*Loss* is the "mathematical" value we have specified in our model to use for parameter fitting.

*Accuracy* is simply how many we get right when we test our model as an application. It might not apply to a non-classification problem (think *Stable Diffusion*) and it doesn't capture how much right or wrong we are (we could be very confident that a dog is a cat).

The two are normally closely related and track each other. We will choose Accuracy for our graphs. Any user understands what accuracy represents.



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'),
])

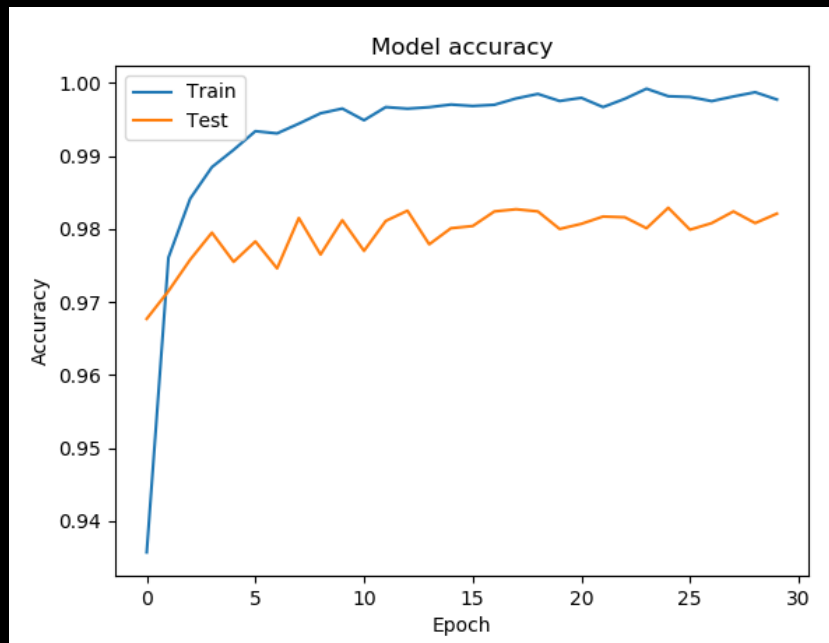
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

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'),
])

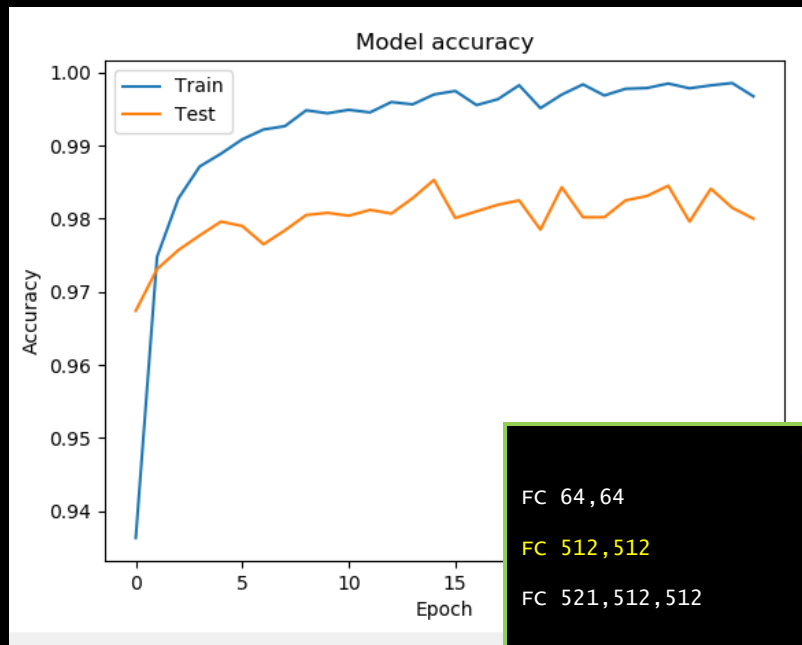
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

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



# Brute Force Does Not Work

You usually can't just brute force your way into success. Beyond the obvious time and memory costs, you are opening yourself up to

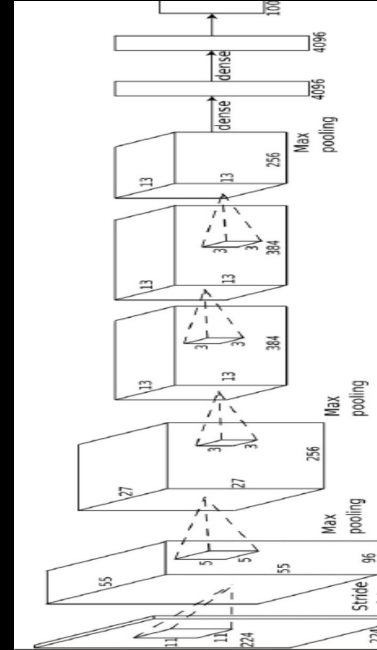
- Overfitting
- Vanishing gradients

We will have to be smarter than "bigger is better" about choosing our hyperparameters. One very smart thing to do is to choose a more appropriate architecture.

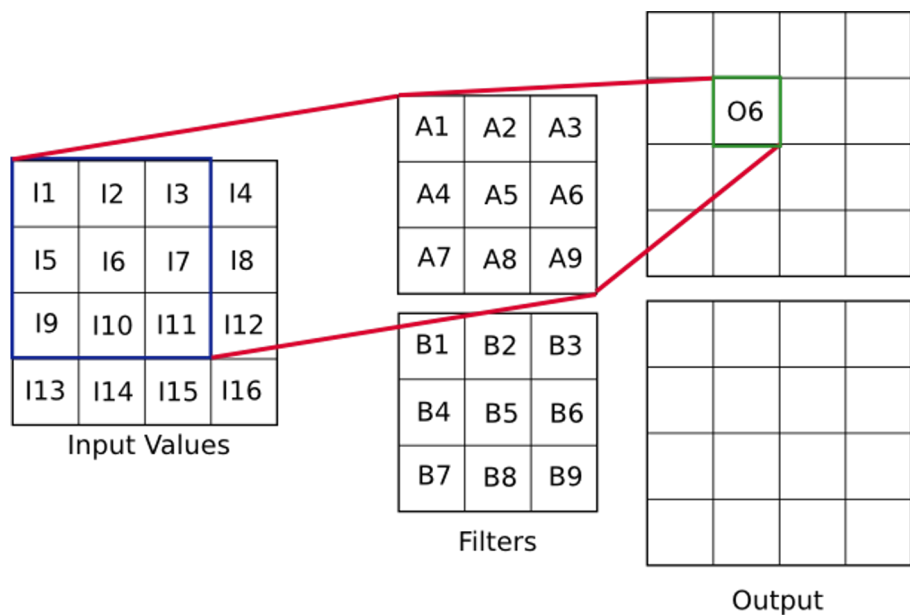
# Image Recognition Done Right: CNNs

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

4M	FULL CONNECT
16M	FULL 4096/ReLU
37M	FULL 4096/ReLU
	MAX POOLING
442K	CONV 3x3/ReLU
1.3M	CONV 3x3/ReLU
884K	CONV 3x3/ReLU
	MAX POOLING 2x2sub
	LOCAL CONTRAST NORM
307K	CONV 11x11/ReLU
	MAX POOL 2x2sub
	LOCAL CONTRAST NORM
35K	CONV 11x11/ReLU



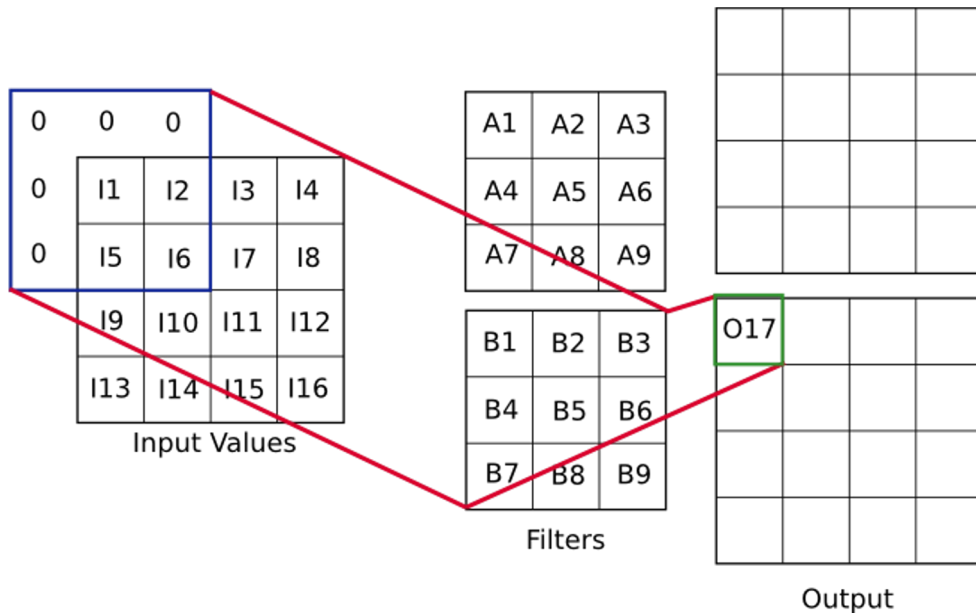
# Convolution



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

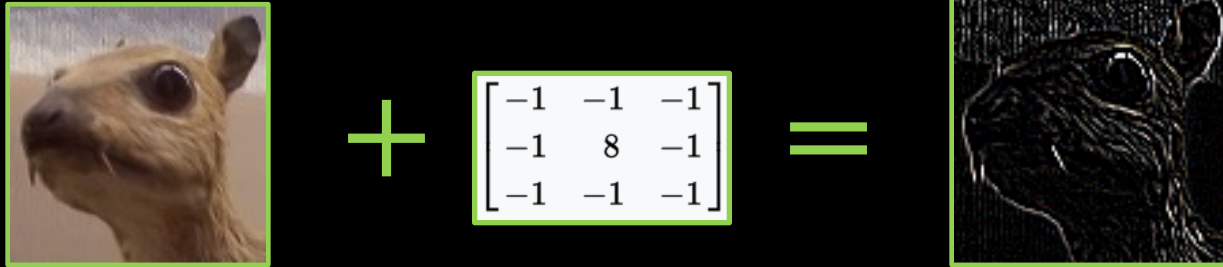
# Convolution

## Boundary and Index Accounting



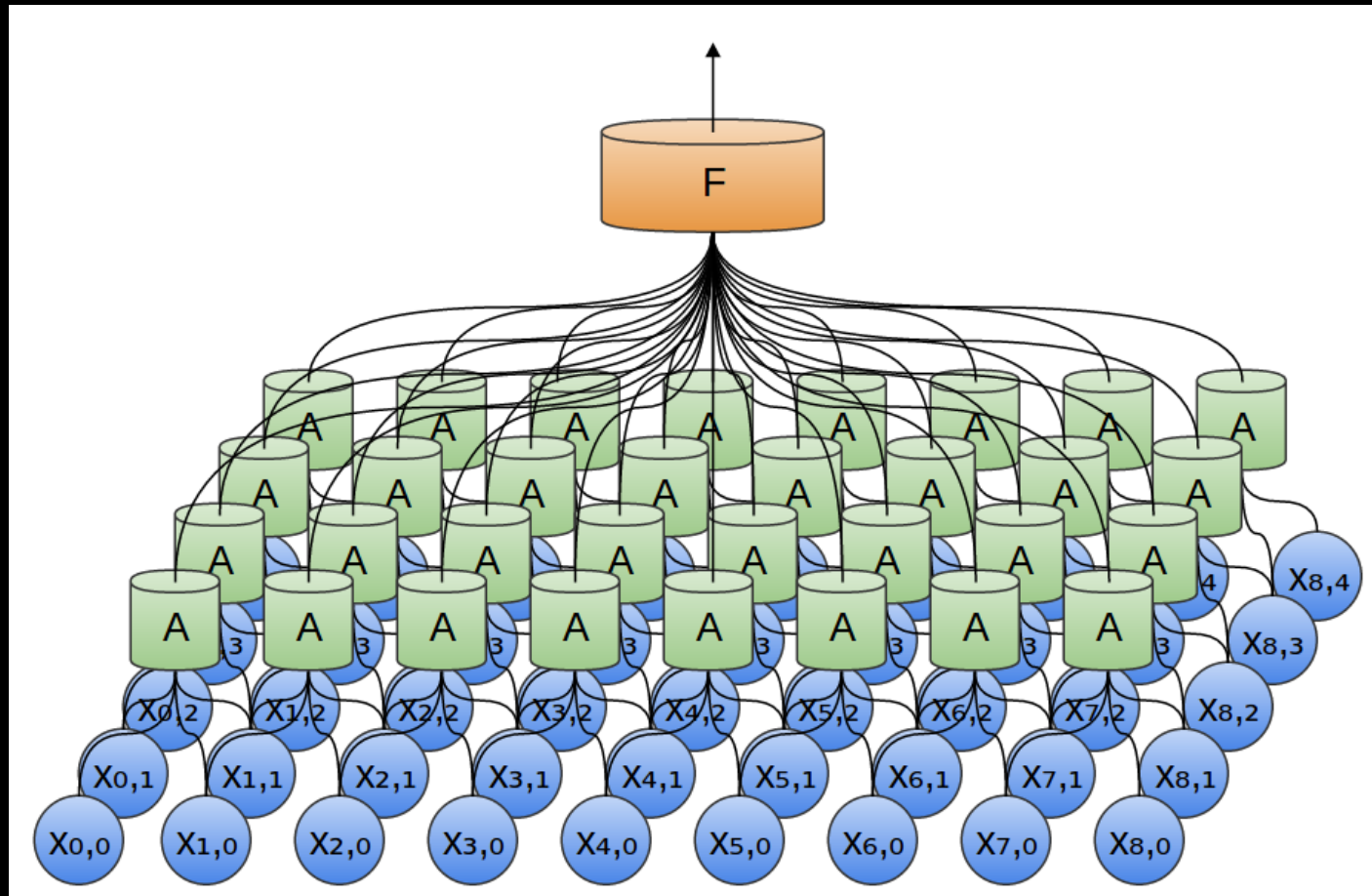
$$O_{17} = B_5 \cdot I_1 + B_6 \cdot I_2 + B_8 \cdot I_5 + B_9 \cdot I_6$$

# Straight Convolution

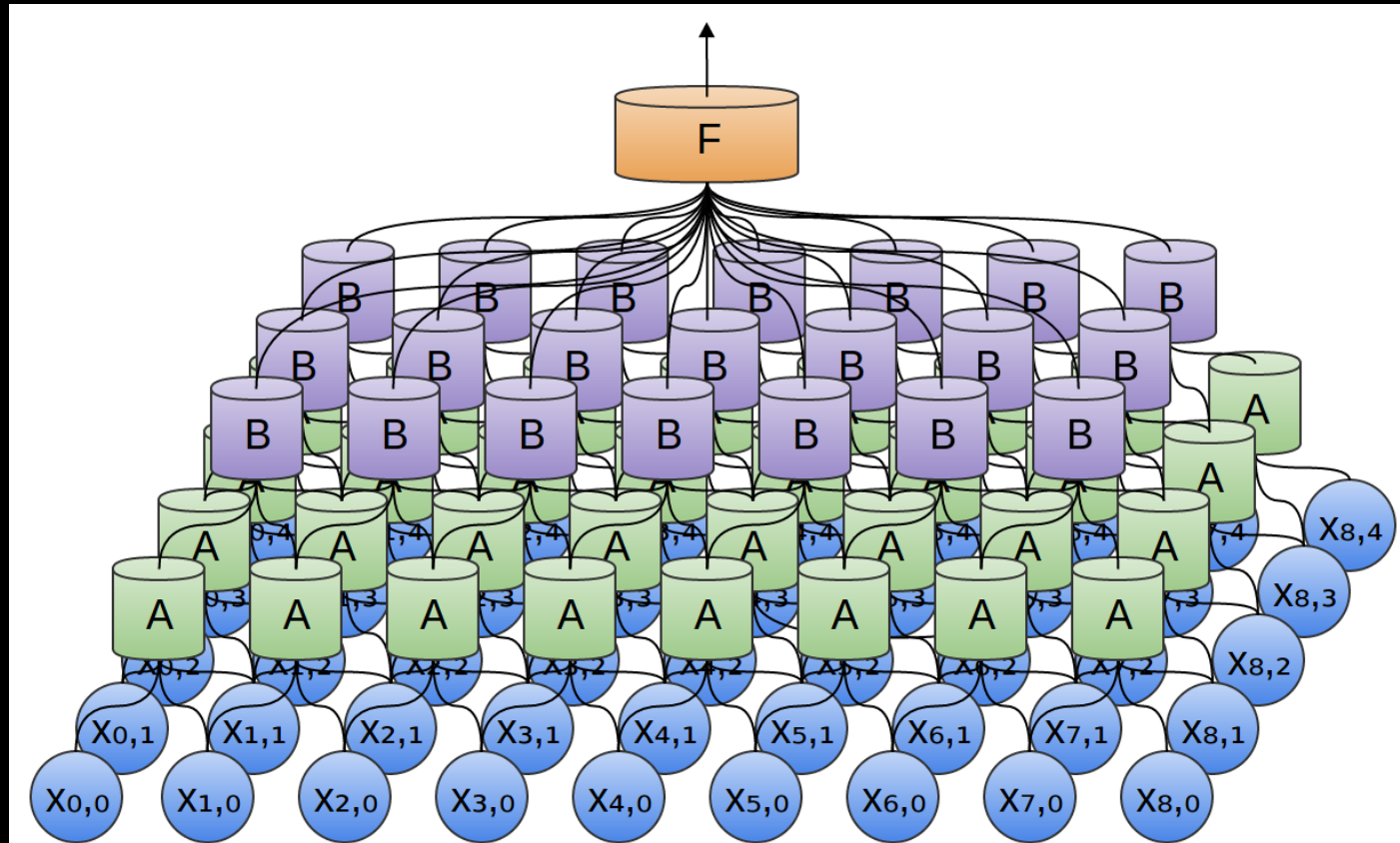

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Edge Detector

# Simplest Convolution Net



# Stacking Convolutions



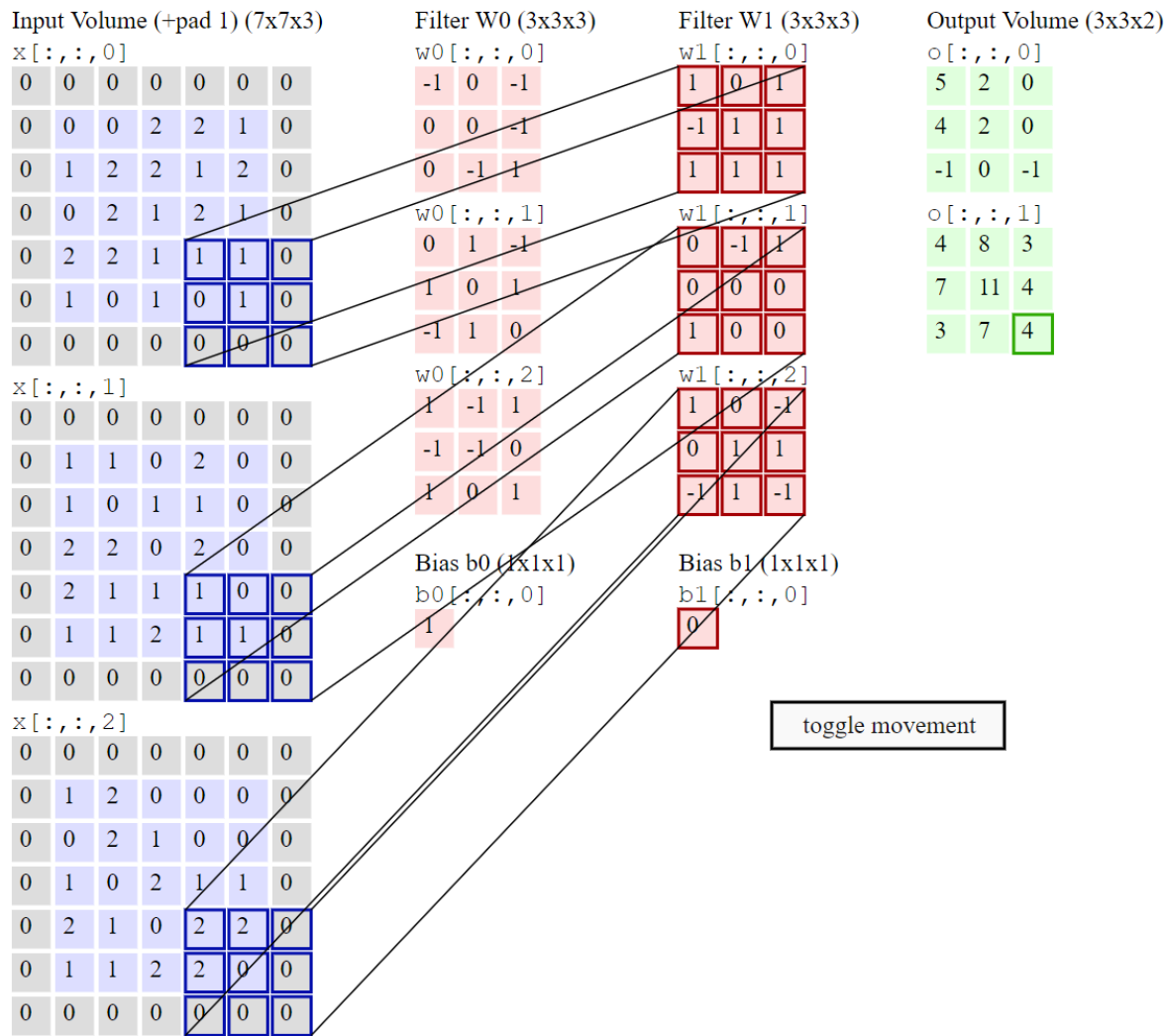
# Demos

Style vs. Content: Your own algorithms





# 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  $W_O \times H_O \times D_O$

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

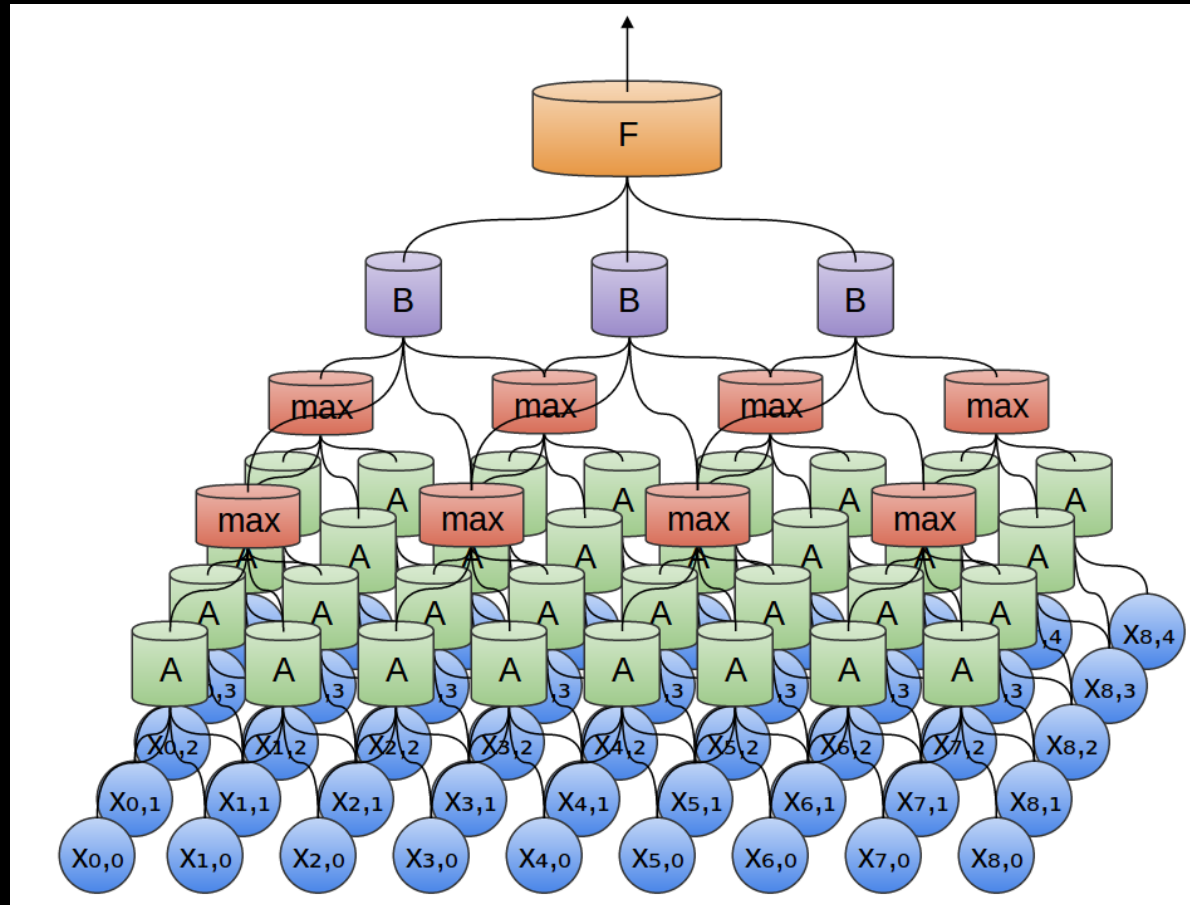
$$H_O = (H_I - F + 2P) / S + 1$$

$$D_O = K$$

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

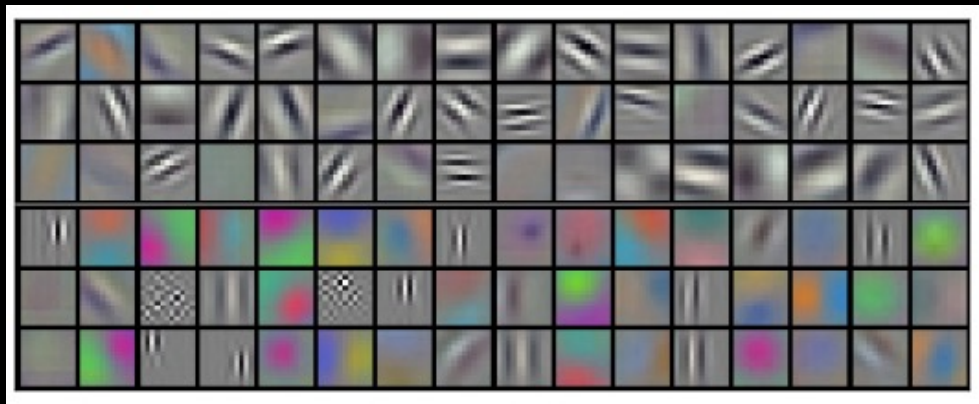
In the output volume, the d-th depth slice (of size  $W_O \times H_O$ ) is the result of performing a convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

# Pooling

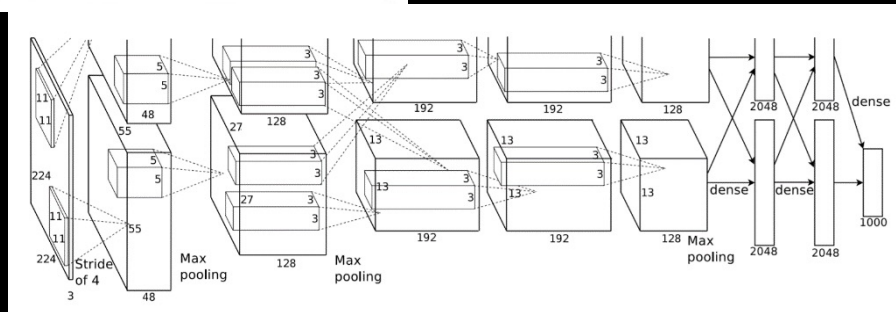


# A Groundbreaking Example

These are the 96 first layer 11x11 (x3, RGB, stacked here) filters from AlexNet.



Among the several novel techniques combined in this work (such as aggressive use of ReLU), they used dual GPUs, with different flows for each, communicating only at certain layers. A result is that the bottom GPU consistently specialized on color information, and the top did not.



# Let's Start Small

```
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, 28, 28, 1)
test_images = test_images.reshape(10000, 28, 28, 1)
train_images, test_images = train_images/255, test_images/255

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

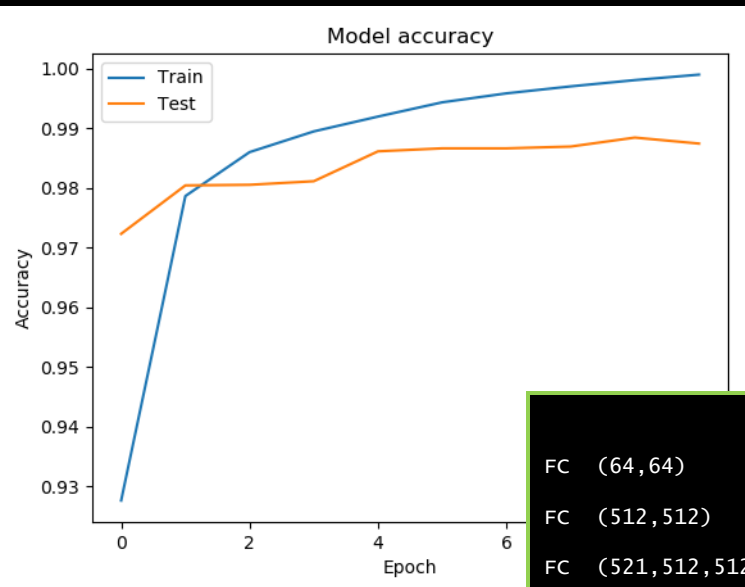
model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.01, momentum=0.9), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.fit(train_images, train_labels, batch_size=32, epochs=10, verbose=1, validation_data=(test_images, test_labels))
```

# Early CNN Results

....  
....

Epoch 10/10  
60000/60000 [=====] - 12s 198us/sample - loss: 0.0051 - accuracy: 0.9989 - val\_loss: 0.0424 - val\_accuracy: 0.9874



## Score Thus Far

FC (64,64)	97.5
FC (512,512)	98.2
FC (521,512,512)	98.0
CNN (1 layer)	98.7

## Primitive CNN

```
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_1	(None, 13, 13, 32)	0
flatten_1 (Flatten)	(None, 5408)	0
dense_38 (Dense)	(None, 100)	540900
dense_39 (Dense)	(None, 10)	1010

```
ms: 542,230  
params: 542,230  
able params: 0
```

# Scaling Up The CNN

```
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, 28, 28, 1)
test_images = test_images.reshape(10000, 28, 28, 1)
train_images, test_images = train_images/255, test_images/255

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

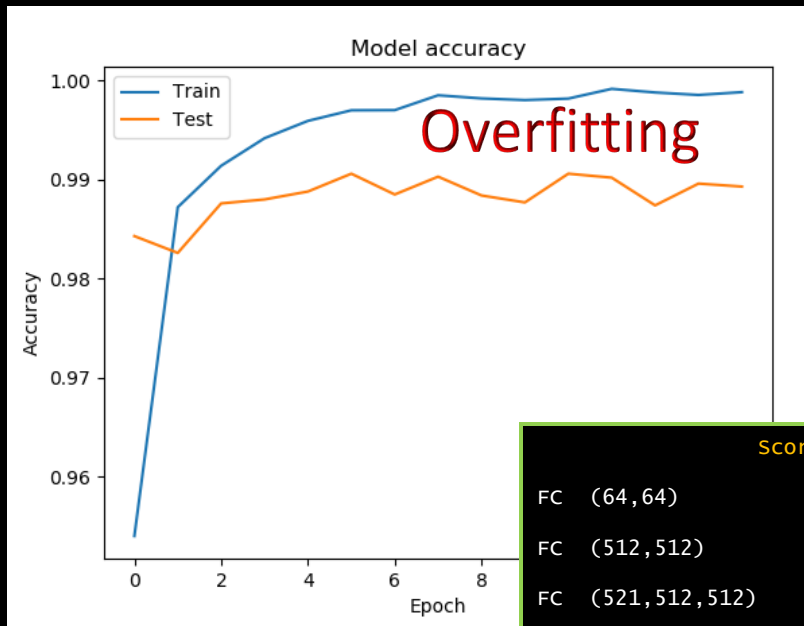
model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.01, momentum=0.9), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.fit(train_images, train_labels, batch_size=32, epochs=10, verbose=1, validation_data=(test_images, test_labels))
```

# Deeper CNN Results

....  
....

Epoch 15/15  
60000/60000 [=====] - 34s 566us/sample - loss: 0.0052 - accuracy: 0.9985 - val\_loss: 0.0342 - val\_accuracy: 0.9903



## Score Thus Far

FC (64,64)	97.5
FC (512,512)	98.2
FC (521,512,512)	98.0
CNN (1 layer)	98.7
CNN (2 Layer)	99.0

## Deeper CNN

```
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 26, 26, 32)	320
conv2d_5 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_3	(None, 12, 12, 64)	0
flatten_3 (Flatten)	(None, 9216)	0
dense_42 (Dense)	(None, 128)	1179776
	(None, 10)	1290

=====

1,199,882

s: 1,199,882

arams: 0



# Overfitting = Memorization

We now have enough parameters that the network is prone to memorizing instead of learning. This will only get worse as our larger and smarter networks grow into billions of parameters.



Cat



Dog



Dog



Cat



Cat



Dog



Dog



Dog



Dog



Dog

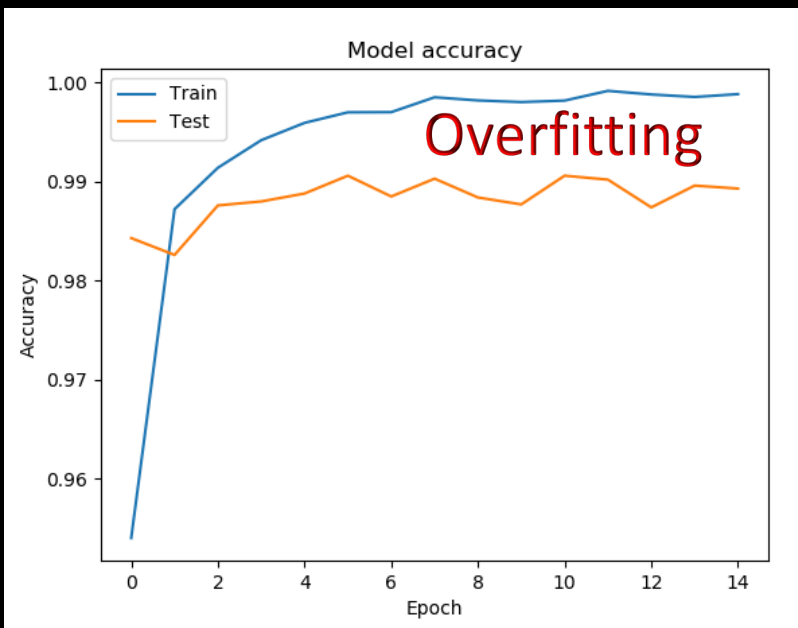


Cat



Cat

# Dropout



As we know by now, we need some form of regularization to help with the overfitting. One seemingly crazy way to do this is the relatively new technique (introduced by the venerable Geoffrey Hinton in 2012) of Dropout.



Some view it as an ensemble method that trains multiple data models simultaneously. One neat perspective of this analysis-defying technique comes from Jürgen Schmidhuber, another innovator in the field; under certain circumstances, it could also be viewed as a form of training set augmentation: effectively, more and more informative complex features are removed from the training data.

# CNN With Dropout

```
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, 28, 28, 1)
test_images = test_images.reshape(10000, 28, 28, 1)
train_images, test_images = train_images/255, test_images/255

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.01, momentum=0.9), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.fit(train_images, train_labels, batch_size=32, epochs=10, verbose=1, validation_data=(test_images, test_labels))
```

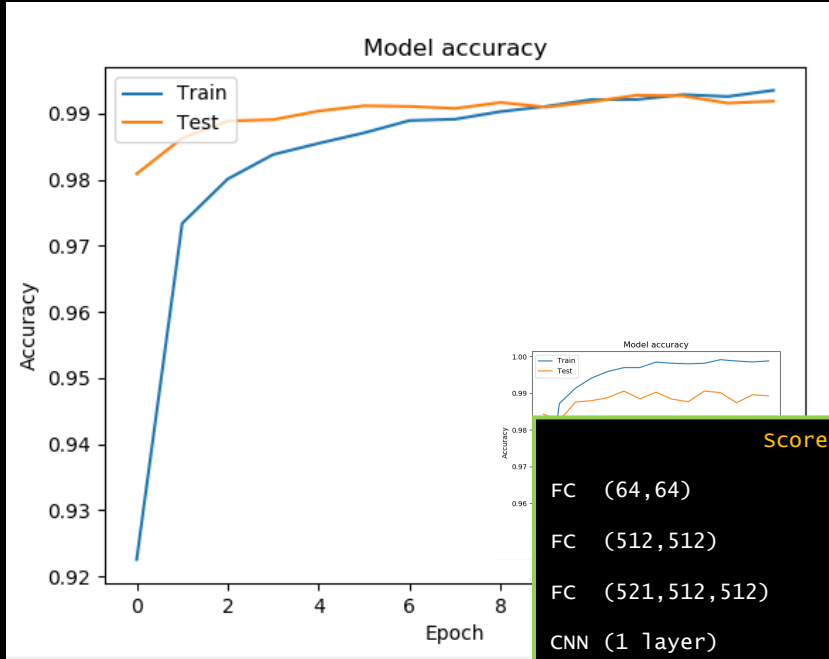
Parameter is fraction to drop.

Drop out is not used in the final, trained, network. Similarly, it is automatically disabled here during testing.

# Help From Dropout

....  
....

Epoch 15/15  
60000/60000 [=====] - 40s 667us/sample - loss: 0.0187 - accuracy: 0.9935 - val\_loss: 0.0301 - val\_accuracy: 0.9919



## Score Thus Far

FC (64,64)	97.5
FC (512,512)	98.2
FC (521,512,512)	98.0
CNN (1 layer)	98.7
CNN (2 Layer)	99.0
CNN with Dropout	99.2

## Dropout CNN

```
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 26, 26, 32)	320
conv2d_13 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_7	(None, 12, 12, 64)	0
dropout_4 (Dropout)	(None, 12, 12, 64)	0
flatten_7 (Flatten)	(None, 9216)	0
(Dense)	(None, 128)	1179776
(Dropout)	(None, 128)	0
(Dense)	(None, 10)	1290

params: 1,199,882  
params: 1,199,882  
able params: 0

# Batch Normalization

Another "between layers" layer that is quite popular is Batch Normalization. This technique really helps with vanishing or exploding gradients. So it is better with deeper networks.

- Maybe not so compatible with Dropout, but the subject of research (and debate).
- Maybe Apply Dropout after all BN layers: <https://arxiv.org/pdf/1801.05134.pdf>
- Before or after non-linear activation function? Oddly, also open to debate. But, it may be more appropriate after the activation function if for s-shaped functions like the hyperbolic tangent and logistic function, and before the activation function for activations that result in non-Gaussian distributions like ReLU.

How could we apply it before or after our activation function if we wanted to? We haven't been peeling our layers apart, but we can micro-manage more if we want to:

```
model.add(tf.keras.layers.Conv2D(64, (3, 3), use_bias=False))  
model.add(tf.keras.layers.BatchNormalization())  
model.add(tf.keras.layers.Activation("relu"))
```

```
model.add(tf.keras.layers.Conv2D(64, kernel_size=3, strides=2, padding="same"))  
model.add(tf.keras.layers.LeakyReLU(alpha=0.2))  
model.add(tf.keras.layers.BatchNormalization(momentum=0.8))
```

There are also normalizations that work on single samples instead of batches, so better for recurrent networks. In TensorFlow we have Group Normalization, Instance Normalization and Layer Normalization.

# Trying Batch Normalization

```
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, 28, 28, 1)
test_images = test_images.reshape(10000, 28, 28, 1)
train_images, test_images = train_images/255, test_images/255

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(10, activation='softmax')
])

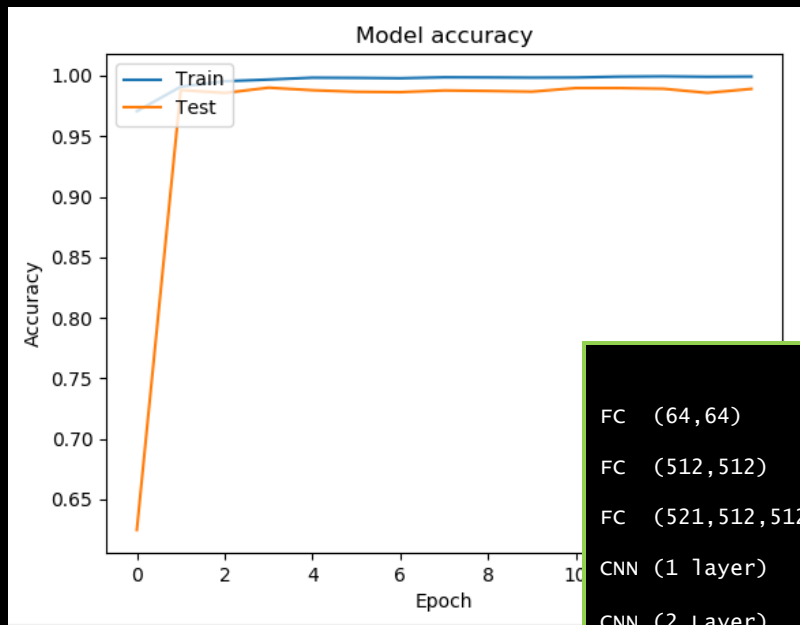
model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.01, momentum=0.9), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.fit(train_images, train_labels, batch_size=32, epochs=10, verbose=1, validation_data=(test_images, test_labels))
```

# Not So Helpful

....  
....

Epoch 15/15  
60000/60000 [=====] - 50s 834us/sample - loss: 0.0027 - accuracy: 0.9993 - val\_loss: 0.0385 - val\_accuracy: 0.9891



## Score Thus Far

FC (64,64)	97.5
FC (512,512)	98.2
FC (521,512,512)	98.0
CNN (1 layer)	98.7
CNN (2 Layer)	99.0
CNN with Dropout	99.2
Batch Normalization	98.9

## Batch Normalization CNN

```
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 26, 26, 32)	320
batch_normalization	(None, 26, 26, 32)	128
conv2d_3 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_1	(None, 12, 12, 64)	0
batch_normalization_1	(None, 12, 12, 64)	256
flatten	(None, 9216)	0
conv2d_4 (Conv2D)	(None, 128)	1179776
batch_normalization_2	(Batch Normalization)	512
conv2d_5 (Conv2D)	(None, 10)	1290
Total params: 1,200,778		
Trainable params: 1,200,330		
Non-trainable params: 448		

# Real Time Demo

This *amazing, stunning, beautiful* demo from Adam Harley is very similar to what we just did, but different enough to be interesting.

[https://aharley.github.io/nn\\_vis/cnn/2d.html](https://aharley.github.io/nn_vis/cnn/2d.html)

It is worth experiment with. Note that this is an excellent demonstration of how efficient the forward network is. You are getting very real-time analysis from a lightweight web program. Training it took some time.



Draw your number here



Downsampled drawing: 2

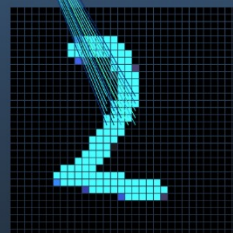
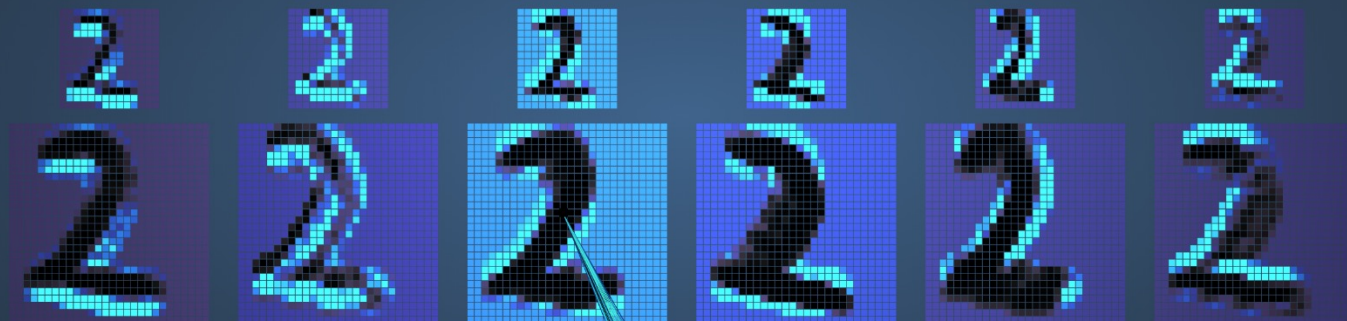
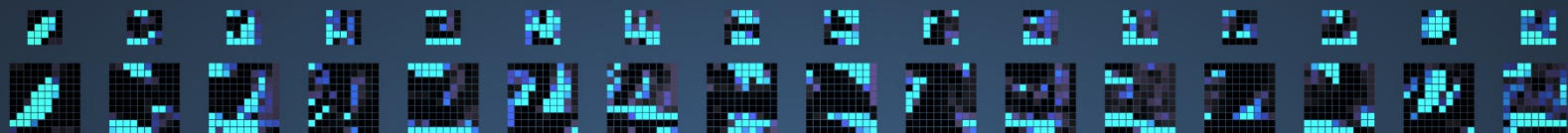
First guess: 2

Second guess: 0

### Layer visibility

Input layer	Show
Convolution layer 1	Show
Downsampling layer 1	Show
Convolution layer 2	Show
Downsampling layer 2	Show
Fully-connected layer 1	Show
Fully-connected layer 2	Show
Output layer	Show

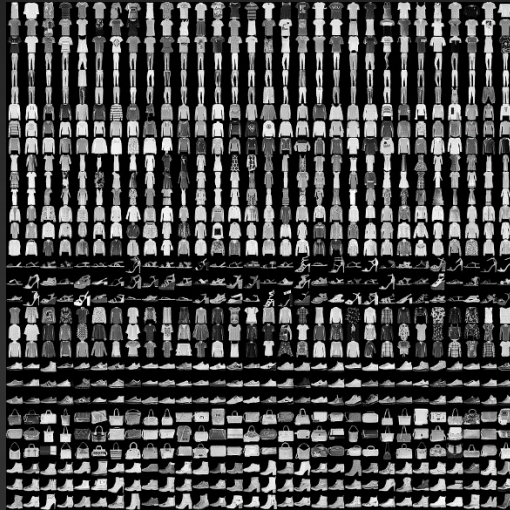
0123456789



# Hands-On Exercise: Fashion MNIST

We have done very well with MNIST, and trying to improve it much more will have very diminishing returns. Let us find a more challenging problem.

We have a very obvious and incremental next step with Fashion MNIST. These are 60,000 training images, and 10,000 test images of 10 types of clothing, in 28x28 greyscale. Sound familiar?



# Exercises

We are going to leave you with a few substantial problems that you are now equipped to tackle. Feel free to use your extended workshop access to work on these, and remember that additional time is an easy Startup Allocation away. Of course everything we have done is standard and you can work on these problems in any reasonable environment.

You may have wondered what else was to be found at [tf.keras.datasets](https://tf.keras.datasets.fashion). The answer is many interesting problems. The obvious follow-on is:

## Fashion MNIST

These are 60,000 training images, and 10,000 test images of 10 types of clothing, in 28x28 greyscale. Sound familiar? A more challenging drop-in for MNIST.

This will be our exercise. Tweak the hyperparameters of our `CNN_dropout.py` script to try to get a better categorization of the MNIST clothing dataset than what we started with. Change values, add layers, play around and think through what's happening and we'll review some suggested solutions at the end.



# Fashion MNIST



It is very easy to just change one line of code and we are using this new and more challenging dataset:

```
mnist = tf.keras.datasets.fashion_mnist
```

And, if you run using our current networks, you even get reasonable results. **You should consider this your "baseline".** However, we now have a lot more room for improvement.

Your job is to find a better network, and get those improvements. You have plenty of hyperparameters to play with here, so you won't be short of options to explore.

You can change some of the obvious ones we have just investigated in obvious ways (brute force, bigger, bigger!) or you can peruse the documentation for inspiration.

This is an open-ended research project. There is no right answer, and you may even surpass my own solutions. However, if you put in a reasonable effort you should have no problem getting a measurable improvement over the baseline. And if you want to determine what a reasonable ceiling might be, I suggest a little Googling. ***A literature search is the essential starting point for any deep learning project!***

# Adding TensorBoard To Your Code

TensorBoard is a very versatile tool that allows us multiple types of insight into our TensorFlow codes. We need only add a callback into the model to activate the necessary logging.

```
...  
...  
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])  
  
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir='TB_logDir', histogram_freq=1)  
  
history = model.fit(train_images, train_labels, batch_size=128, epochs=15, verbose=1,  
                    validation_data=(test_images, test_labels), callbacks=[tensorboard_callback])  
...  
...
```

TensorBoard runs as a server, because it has useful run-time capabilities, and requires you to start it separately, and to access it via a browser.

Somewhere else:

```
tensorboard --logdir=TB_logD
```

Somewhere else:

Start your Browser and point it at port 6006: <http://localhost:6006/>

If you are running on Bridges login nodes, from your computer something like:

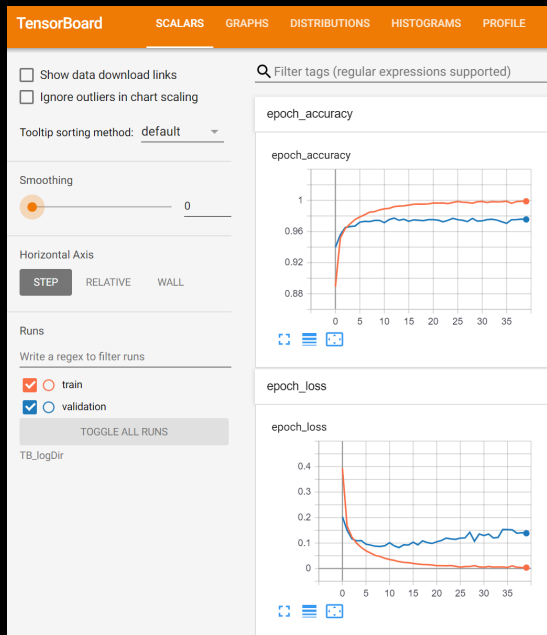
```
ssh -2 -Nf -L 6006:127.0.0.1:6006 br014.bridges2.psc.edu
```

If you are running on a Bridges compute nodes, you need to use the compute's IB address/hostname, for example:

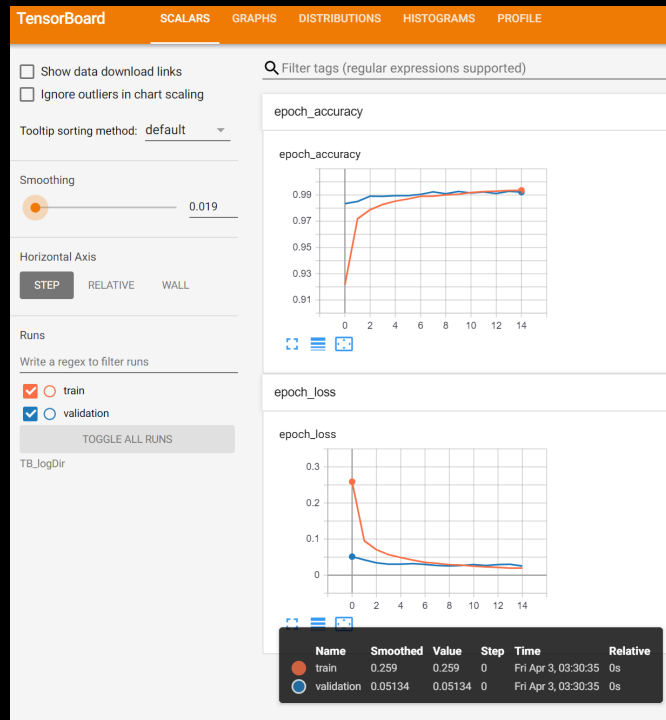
```
ssh -2 -Nf -L 6006:r001.ib.bridges2.psc.edu:6006 br014.bridges2.psc.edu
```

# TensorBoard Analysis

The most obvious thing we can do is to look at our training loss. Note that TB is happy to do this in real-time as the model runs. This can be very useful for you to monitor overfitting.



Our First Model  
64 Wide FC



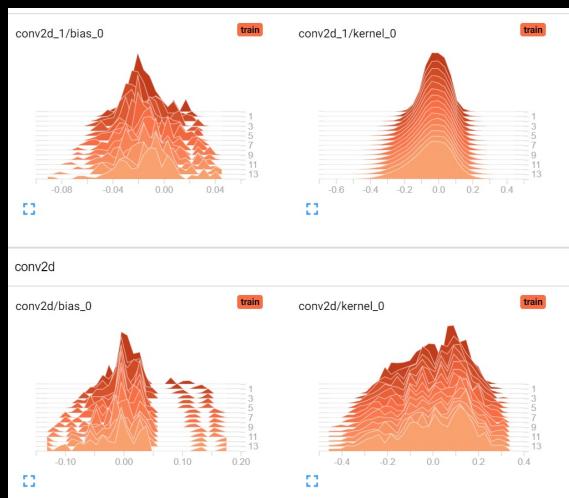
Our CNN

# TensorBoard Parameter Visualization

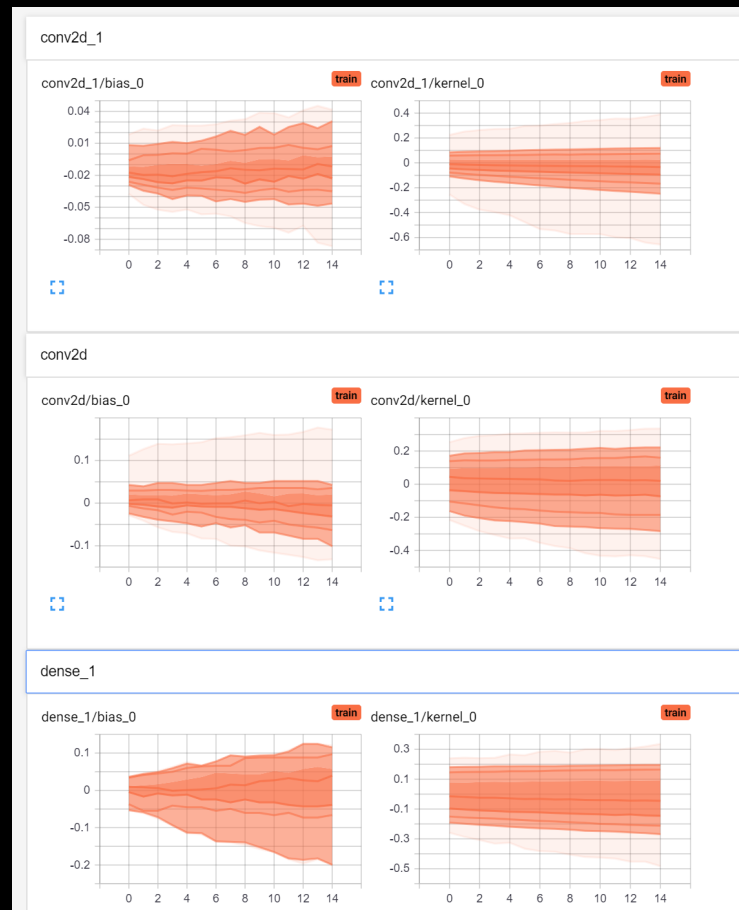
## Distribution View

And we can observe the time evolution of our weights and biases, or at least their distributions.

This can be very telling, but requires some deeper application and architecture dependent understanding.



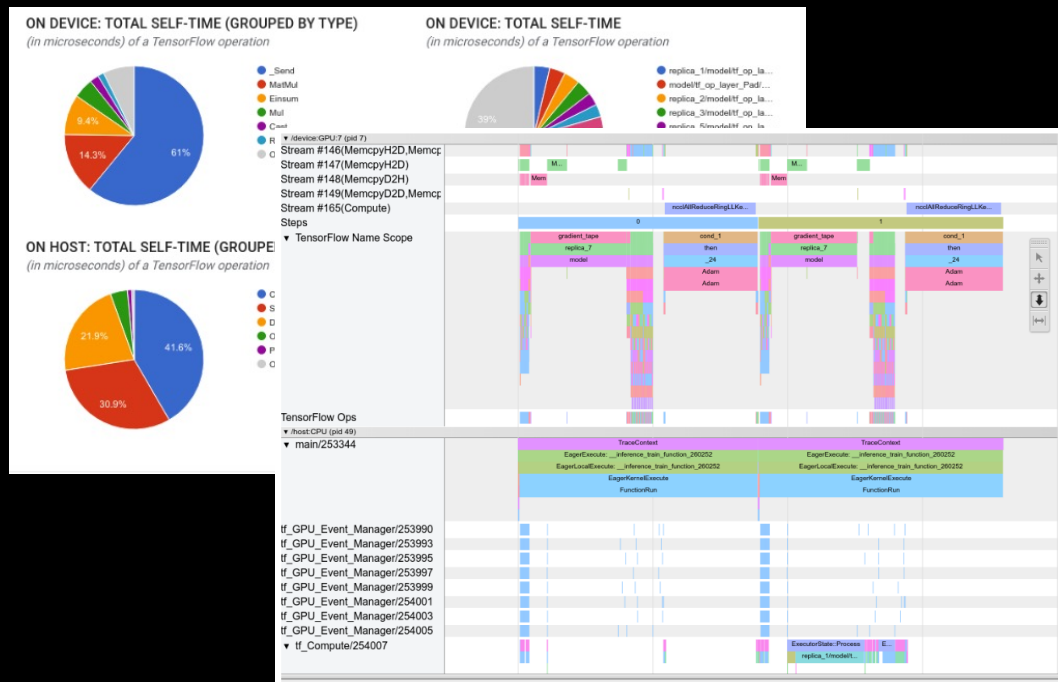
## Histogram View



# TensorBoard Add Ons

TensorBoard has lots of extended capabilities. Two particularly useful and powerful ones are Hyperparameter Search and Performance Profiling.

## Performance Profiling



## Hyperparameter Search

Requires some scripting on your part. Look at [https://www.tensorflow.org/tensorboard/hyperparameter\\_tuning\\_with\\_hparams](https://www.tensorflow.org/tensorboard/hyperparameter_tuning_with_hparams) for a good introduction.

Going beyond basics, like **IO time**, requires integration of hardware specific tools. This is well covered if you are using NVIDIA, otherwise you may have a little experimentation to do. The end result is a user friendly interface and valuable guidance.



# Scaling Up

You may have the idea that deep learning has a voracious appetite for GPU cycles. That is absolutely the case, and the leading edge of research is currently limited by available resources. Researchers routinely use many GPUs to train a model. Conversely, the largest resources demand that you use them in a parallel fashion. There are capabilities built into TensorFlow, the **MirroredStrategy**:

```
strategy = tf.distribute.Mirrored
with strategy.scope():
    model = tf.keras.Sequential
        tf.keras.layers.Dropout
        tf.keras.layers.Dense(u
        ...
    ])
    model.compile(...)
    model.fit(...)
```

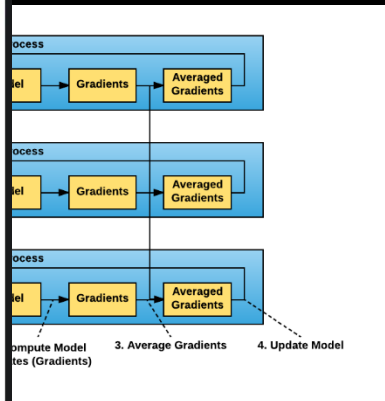
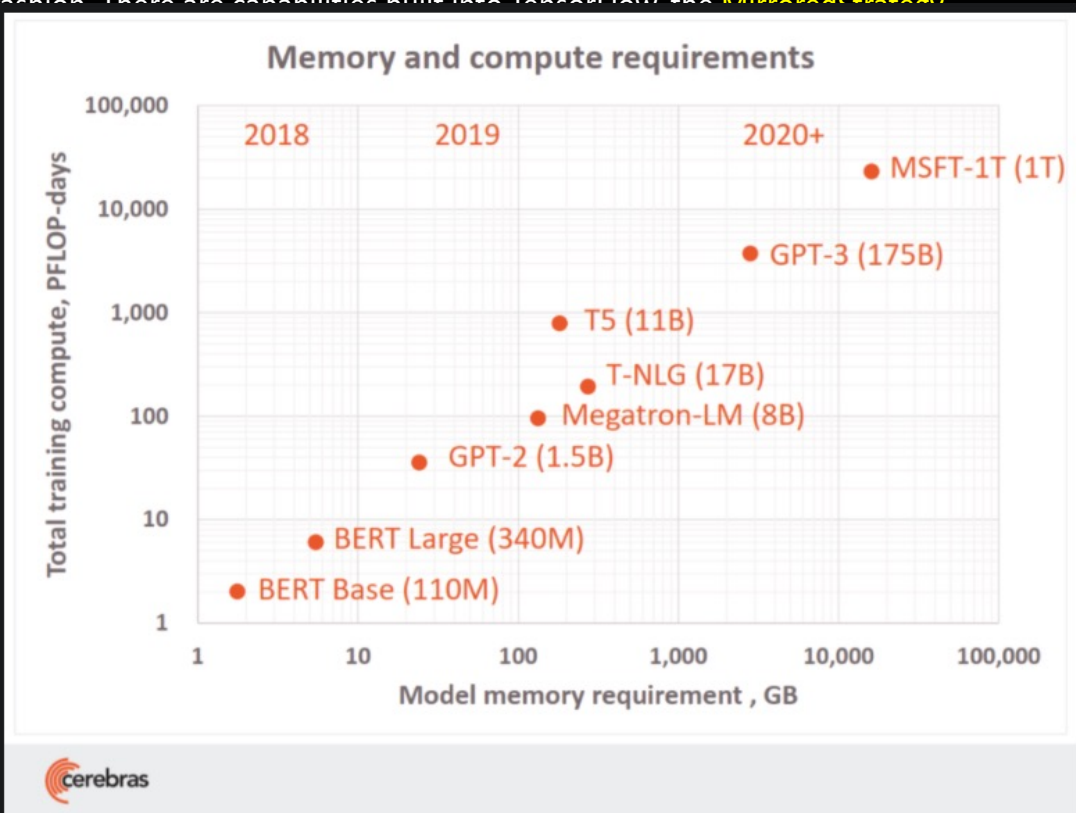
An alternative that has proven it

**MNIST**

```
# Horovod: initialize Horovod.
hvd.init()

# Horovod: pin GPU to be used to process 1c
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
config.gpu_options.visible_device_list = str(K.set_session(tf.Session(config=config)))
...
# Horovod: adjust number of epochs based on hvd.size()
epochs = int(math.ceil(12.0 / hvd.size()))
...
# Horovod: adjust learning rate based on number of GPUs
opt = keras.optimizers.Adadelta(1.0 * hvd.size())
...
# Horovod: add Horovod Distributed Optimizer
opt = hvd.DistributedOptimizer(opt)
...
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=opt,
              metrics=['accuracy'])

callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback(0)]
if hvd.rank() == 0: callbacks.append(keras.callbacks.ModelCheckpoint("./checkpoint-{epoch}.h5"))
```

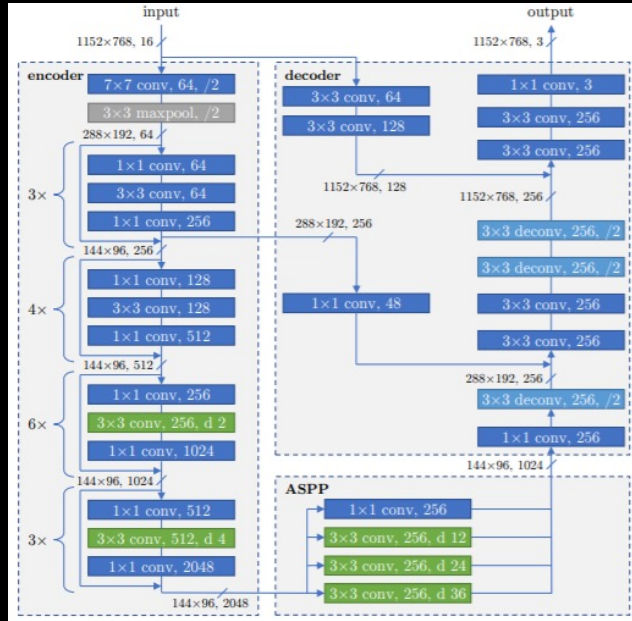


*distributed deep learning in TensorFlow*  
Del Balso

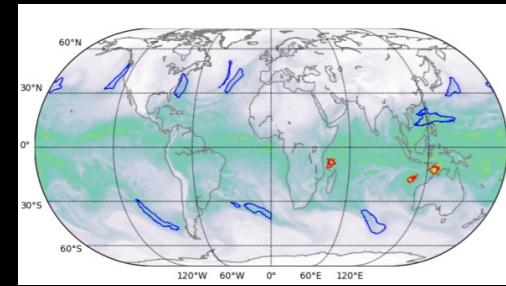
using Horovod with a Keras MNIST code at:  
<https://keras.io/latest/keras.html>

# Scaling Up Massively

*Horovod* demonstrates its excellent scalability with a Climate Analytics code that won the Gordon Bell prize in 2018. It predicts Tropical Cyclones and Atmospheric River events based upon climate models. It shows not only the reach of deep learning in the sciences, but the scale at which networks can be trained.



- *1.13 ExaFlops (mixed precision) peak training performance*
- *On 4560 6 GPU nodes (27,360 GPUs total)*
- *High-accuracy (harder when predicting "no hurricane today" is 98% accurate), solved with weighted loss function.*
- *Layers each have different learning rate*



# Data Augmentation

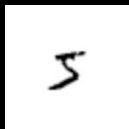
As I've mentioned, labeled data is valuable. This type of *supervised learning* often requires human-labeled data. Getting more out of our expensive data is very desirable. More datapoints generally equals better accuracy. The process of generating more training data from our existing pool is called *Data Augmentation*, and is an extremely common technique, especially for classification

Our MNIST network has learned to recognize ve

What if we wanted to teach it:

## How many samples do we need?

This is another hyperparameter (yes), where we can only offer a vague rule of thumb. And that suggestion is about 5000 per category for competence, 10 million for a real task with human performance.



Scale Invariance



Rotation Invariance



Noise Tolerance



Translation Invariance

You can see how straightforward and mechanical this is. And yet very effective. You will often see detailed explanations of the data augmentation techniques employed in any given project.

Note that `tf.image` makes many of these processes very convenient.

```

from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.optim.lr_scheduler import StepLR

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output

def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{}] ({:.0f}%) \tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))

def test(args, model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item() # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True) # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('Test set: Average loss: {:.4f}, Accuracy: {:.0f}%\n'.format(
        test_loss, 100. * correct / len(test_loader.dataset)))

if __name__ == '__main__':
    main()

```

# PyTorch CNN MNIST

Not a fair comparison of terseness as this version has a lot of extra flexibility.

From:

<https://github.com/pytorch/examples/blob/master/mnist/main.py>

# More tf.keras.datasets Fun

## Boston Housing

Predict housing prices base upon crime, zoning, pollution, etc.

CRIM	per capita crime rate by town
ZN	proportion of residential land
INDUS	proportion of non-retail busine
CHAS	Charles River dummy variable (-
NOX	nitric oxides concentration (pa
RM	average number of rooms per dwe
AGE	proportion of owner-occupied ur
DIS	weighted distances to five Bost
RAD	index of accessibility to radia
TAX	full-value property-tax rate pe
PTRATIO	pupil-teacher ratio by town
B	$1000(B_k - 0.63)^2$ where $B_k$ is t
LSTAT	% lower status of the populatio
MEDV	Median value of owner-occupied

## CIFAR10

32x32 color images in 10 classes.



## CIFAR100

Like CIFAR10 but with 100 non-overlapping classes.



## IMDB

1 sentence positive or negative reviews.

*I have been known to fall asleep during films, but this...*  
Mann photographs the Alberta Rocky Mountains in a superb fashion...  
This is the kind of film for a snowy Sunday afternoon...

## Reuters

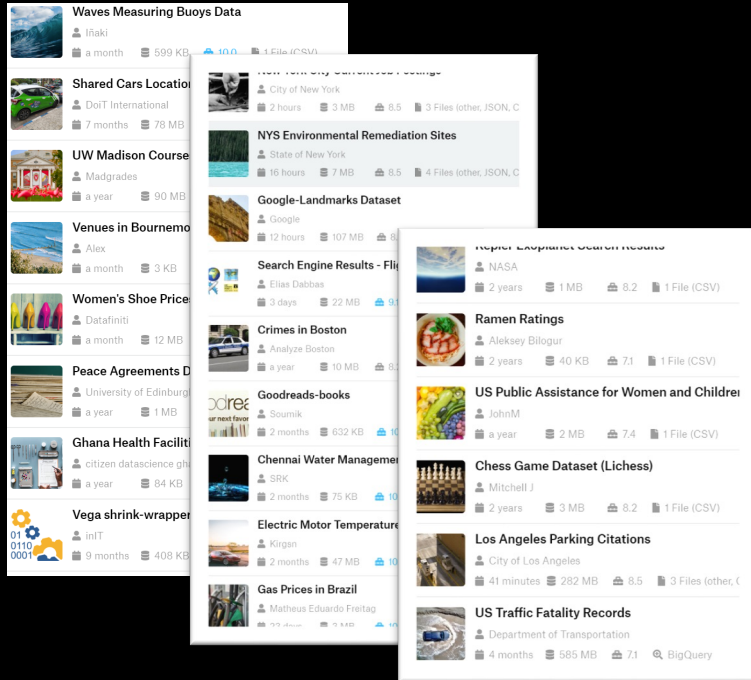
46 topics in newswire form.

Its december acquisition of space co it expects earnings per share in 1987 of 1 15 to 1 30 dlrs per share up from 70 cts in 1986 the company said pretax net should rise to nine to 10 mln dlrs from six mln dlrs in 1986 and rental operation revenues to 19 to 22 mln dlrs from 12 5 mln dlrs it said cash flow per share this year should be 2 50 to three dlrs reuters...






# Endless Exercises

## Kaggle Challenge


The benchmark driven nature of deep learning research, and its competitive consequences, have found a nexus at Kaggle.com. There you can find over 20,000 datasets:



and competitions:

	<b>Severstal: Steel Defect Detection</b> Can you detect and classify defects in steel? <i>Featured</i> · Kernels Competition · 3 months to go · manufacturing, image data	\$120,000 299 teams
	<b>Two Sigma: Using News to Predict Stock Movements</b> Use news analytics to predict stock price performance <i>Featured</i> · Kernels Competition · a day to go · news agencies, time series, finance, money	\$100,000 2,927 teams
	<b>APTOS 2019 Blindness Detection</b> Detect diabetic retinopathy to stop blindness before it's too late <i>Featured</i> · Kernels Competition · a month to go · healthcare, medicine, image data, multiclass classi...	\$50,000 2,106 teams
	<b>SIIM-ACR Pneumothorax Segmentation</b> Identify Pneumothorax disease in chest x-rays <i>Featured</i> · a month to go · image data, object segmentation	\$30,000 1,281 teams
	<b>Predicting Molecular Properties</b>	\$30,000

Including this one:

	<b>Digit Recognizer</b> Learn computer vision fundamentals with the famous MNIST data <i>Getting Started</i> · Ongoing · tabular data, image data, multiclass classification, object identification	Knowledge 3,008 teams
---	---	--------------------------

# Some Hands-On Problem Solutions



## Where to start?

**Step 1:** What do I mean by "better network"? This sounds like an actual research assignment! Well, if I run this

```
import tensorflow as tf

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

train_images = train_images.reshape(60000, 28, 28, 1)
test_images = test_images.reshape(10000, 28, 28, 1)
train_images, test_images = train_images/255, test_images/255

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.01, momentum=0.9), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.fit(train_images, train_labels, batch_size=32, epochs=10, verbose=1, validation_data=(test_images, test_labels))
```

we get this for our baseline.

Epoch 15/15  
469/469 [=====] - 1s 3ms/step - loss: 0.1213 - accuracy: 0.9555 - val\_loss: 0.2384 - val\_accuracy: **0.9284**



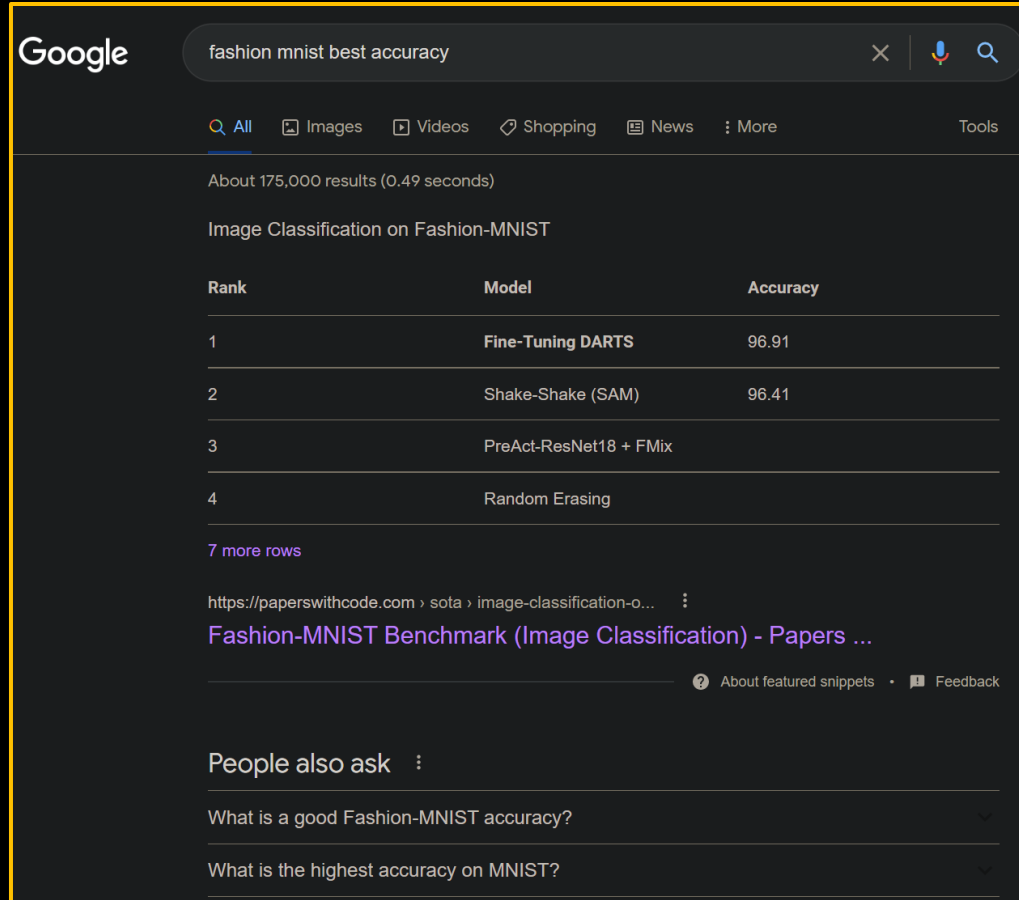
## How far to go?

**Step 2:** As per my hint (and I hope your developing instincts), a good place to gain some perspective is always a little research.

It looks like some networks are getting 96%.

It also looks like those are some fancy networks.

Maybe I'll click a few more of those links...



A screenshot of a Google search results page for the query "fashion mnist best accuracy". The search bar at the top shows the query and the Google logo. Below the search bar, there are tabs for "All", "Images", "Videos", "Shopping", "News", and "More". The search results show "About 175,000 results (0.49 seconds)". The main content area displays a table titled "Image Classification on Fashion-MNIST". The table has three columns: "Rank", "Model", and "Accuracy". The top four results are:

Rank	Model	Accuracy
1	Fine-Tuning DARTS	96.91
2	Shake-Shake (SAM)	96.41
3	PreAct-ResNet18 + FMix	
4	Random Erasing	

Below the table, there is a link to "7 more rows". The search results also include a snippet from "https://paperswithcode.com › sota › image-classification-o..." titled "Fashion-MNIST Benchmark (Image Classification) - Papers ...". At the bottom, there is a section titled "People also ask" with two questions: "What is a good Fashion-MNIST accuracy?" and "What is the highest accuracy on MNIST?".

## Still researching.

This top link happens to provide a nice summary of the state-of-the-art in MNIST. That "kmnist" is a Japanese character version.

We know Alexnet is a big complex and powerful network, and yet it isn't doing as well as our baseline. This is a clue that bigger is not better.

Those other networks are also pretty big and fancy. So it looks like getting even 94% here is a big ask.

And we know I'm not an unreasonable fellow. So let's set our sights appropriately.

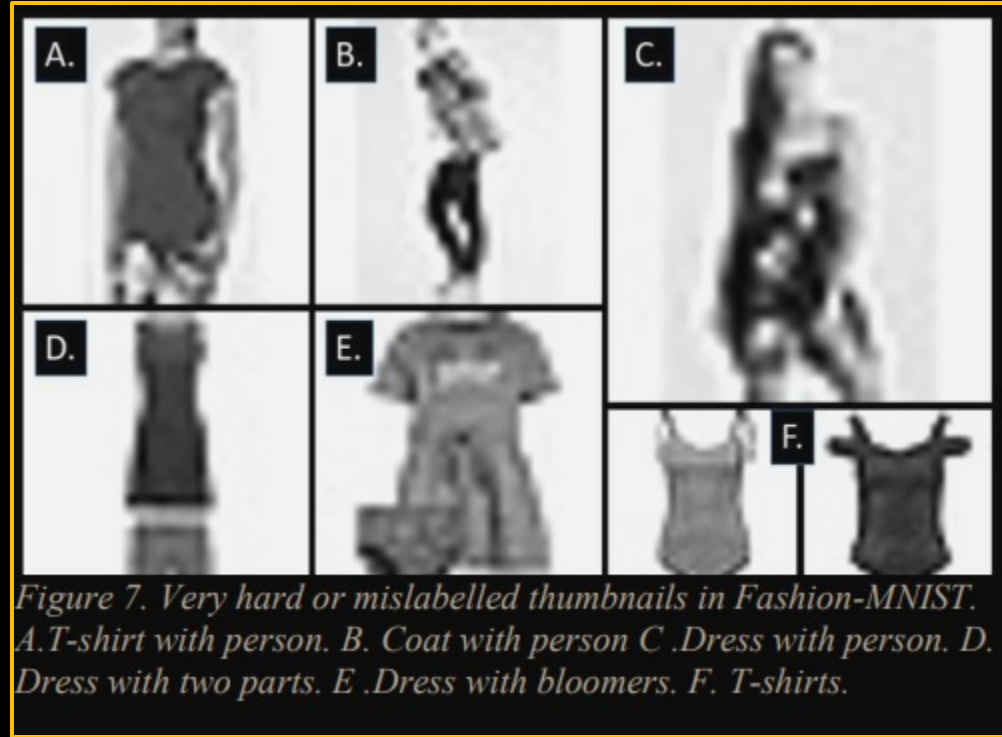
Dataset	Model	Best Epoch	Train Loss	Train Acc.	Val. Loss	Val. Acc.
kmnist	mobilenet_v2	89	0.036	99.988	0.095	98.468
kmnist	resnet101	90	0.020	99.993	0.097	98.147
kmnist	resnet14	73	0.021	99.975	0.070	98.748
kmnist	resnet152	85	0.020	99.988	0.090	98.197
kmnist	resnet18	73	0.020	99.997	0.077	98.628
kmnist	resnet34	83	0.018	99.998	0.081	98.538
kmnist	resnet50	80	0.020	99.982	0.097	98.067
kmnist	resnet9	54	0.008	99.997	0.069	98.528
kmnist	vgg11_bn	62	0.016	99.998	0.078	98.427
kmnist	vgg13_bn	54	0.016	99.980	0.069	98.698
kmnist	vgg16_bn	79	0.015	99.998	0.070	98.698
kmnist	vgg19_bn	99	0.015	99.998	0.078	98.518
fashionmnist	alexnet	97	0.296	89.248	0.308	89.042
fashionmnist	densenet121	92	0.037	99.947	0.266	93.950
fashionmnist	densenet161	93	0.038	99.937	0.264	94.171
fashionmnist	densenet169	97	0.038	99.933	0.258	94.291
fashionmnist	googlenet	96	0.044	99.973	0.240	93.439
fashionmnist	inception_v3	97	0.050	99.861	0.244	94.441
fashionmnist	mobilenet_v2	98	0.040	99.913	0.252	93.860
fashionmnist	resnet101	94	0.019	99.985	0.281	93.740
fashionmnist	resnet14	89	0.021	99.997	0.228	94.040
fashionmnist	resnet152	91	0.020	99.970	0.286	93.770
fashionmnist	resnet18	86	0.020	99.998	0.228	93.970
fashionmnist	resnet34	96	0.018	99.993	0.261	93.910
fashionmnist	resnet50	89	0.019	99.985	0.261	93.810
fashionmnist	resnet9	63	0.009	99.998	0.203	94.071
fashionmnist	vgg11_bn	91	0.017	100.000	0.229	93.600
fashionmnist	vgg13_bn	80	0.017	99.998	0.211	94.111
fashionmnist	vgg16_bn	86	0.018	99.995	0.226	94.030
fashionmnist	vgg19_bn	95	0.018	99.987	0.244	93.960

## Why so hard?

As per this analysis of the dataset, there are a lot debatable labels, and some that are apparently wrong.

This is just a trickier problem. Especially at this resolution.

I could not find the number for "superhuman" on this dataset, but it is clearly lower than for the digits.



## Get to work!

**Step 3:** There is no substitute for trial and error with the hyperparameters. Maybe we can hope to spot a trend.

This is a great example of the kind of approach to take.

This made it to over 94%.

*Nesterov is just a fancier version of gradient descent (it "looks ahead") that you can select.*

Change from Base Code (Acc = 0.8832)	Accuracy
Wider (512)	0.8927
Deeper (512,512,512)	0.8896
CNN (1 layer) - 100 Epochs	0.8790
CNN (2 layers) (32, 64)	0.9223
CNN (2 layers) (64, 64)	0.9217
Drop (0.25,0.5) - 100 Epochs	0.9322
Drop (0.5,0.5) - 80 Epochs	0.9338
Batch Normalization	0.9279
Changing Activation Function ("sigmoid")	0.8678
Conv2D strides = (5,5), lr = 0.02, momentum = 0.9	0.9252
Conv2D strides = (3,3), lr = 0.02, momentum = 0.8	0.9256
lr = 0.02, momentum = 0.9	0.9301
Changing (padding="same") - 40 epochs	0.9364
Changing (padding="same") - 80 epochs	0.9372
nesterov=True - 40 epochs	0.9362
maxPooling(padding="same")	0.9359
nesterov=False	0.9334
nesterov=False - 120 epochs	0.9381
Dropout (0.5,0.6), Epochs = 40	0.9314 (Max: 0.9388)
Dropout (0.6,0.6), Epochs = 40	0.9356
Dropout (0.75,0.75), Epochs = 40	0.9266
Dropout (0.2,0.6), Epochs = 40	0.9336
Dropout (0.2,0.6), Epochs = 100	0.9355
Dropout (0.6,0.6), Epochs = 100	0.9384 (Max: 0.9422)
nesterov=True, Epochs = 100	0.9418

Another fine pursuit of good hyperparameters.

93.7% and a systematic two-steps-forward-one-step-back attack.

#	Model	Train	Test
1	Baseline w/ Dropout (trained for 20 epochs)	0.9427	0.918
2	Changed optimizer to ADAM	0.9619	0.9267
3	Increase first conv layer to 64 filters	0.9612	0.9315
4	Increase second conv layer to 128 filters	0.968	0.9303
5	Increase dense layer to 192	0.9746	0.9335
6	Increase dense layer to 256	0.9814	0.9264
7	Increase dropout after conv layer to 0.4	0.971	0.932
8	Increase dropout after conv layer to 0.5	0.9627	0.9317
9	Add conv layer with 96 filters between existing conv layers	0.9634	0.9318
10	Removed new layer, added third conv layer with 192 filters	0.9667	0.9343
11	Add dense layer with 256 neurons, dropout 0.5	0.9467	0.931
12	Reduce dropout of new layer to 0.25	0.9653	0.9308
13	Remove dropout of new layer	0.9817	0.9256
14	Change new layer to have 320 neurons, reintroduce 0.25 dropout	0.9702	0.9303
15	Revert to model 10, changed third conv layer to have 224 filters	0.9648	0.9348
16	Revert to model 10, changed second conv layer to have 192 filters	0.9626	0.9356
17	Changed first conv layer to have 128 filters	0.9655	0.932
18	Revert to model 16, increase dense layer to have 320 neurons	0.9677	0.9293
19	Revert to model 16, train for 30 epochs	0.9688	0.9338
20	Changed third conv layer to have 256 filters, train for 25 epochs	0.9691	0.9303
21	Changed dense layer to have 320 neurons	0.9733	0.9331
22	Changed dense layer to have 512 neurons	0.9749	0.9357
23	Add BatchNorm after all conv layers before pooling	0.9691	0.9316
24	Add first dense layer with 1024 neurons, dropout 0.5	0.952	0.9342
25	Add conv layer with 64 filters and BatchNorm after first layer	0.948	0.9335
26	Revert to model 16, add BatchNorm after all conv layers before pooling	0.9654	0.9368