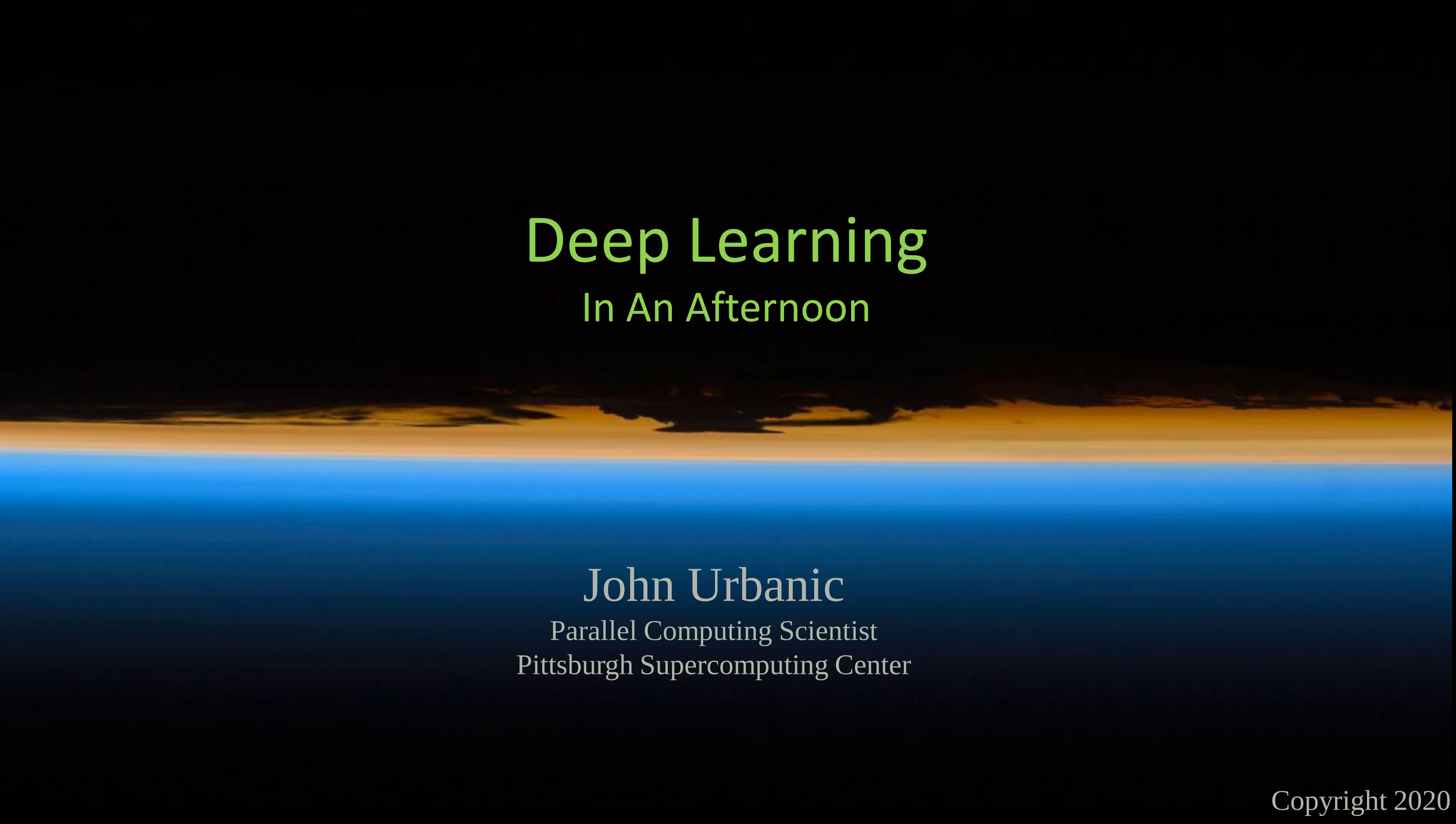


Deep Learning

In An Afternoon

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

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:

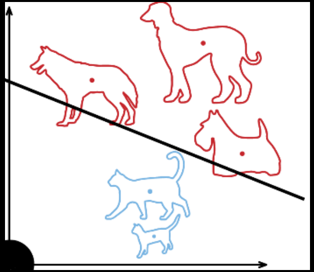
- Widespread parallel computing (GPUs)
- Big data training sets

Two Perspectives

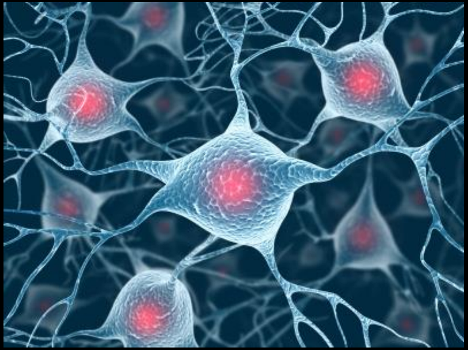
There are really two common ways to view the fundaments of deep learning.

- Inspired by biological models.

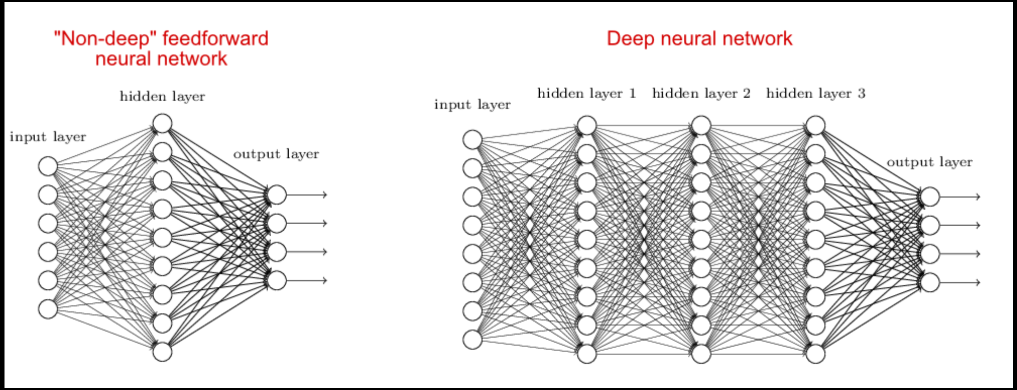
- An evolution of classic ML techniques (the perceptron).



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.



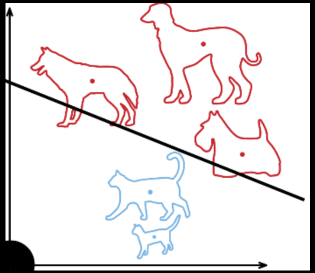
Modeled After The Brain



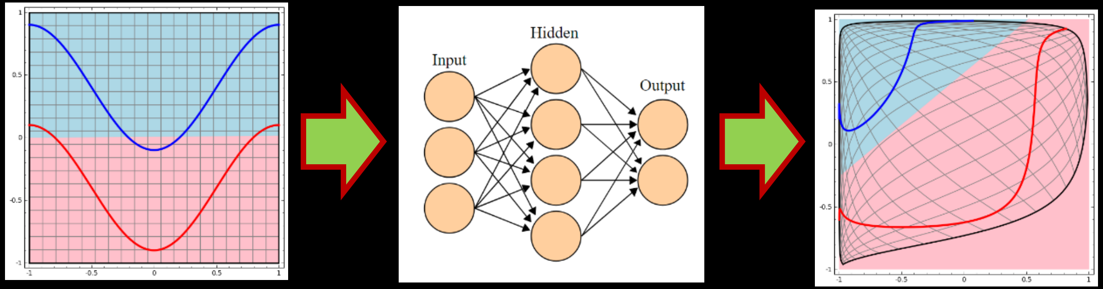
$$M = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1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

As a Highly Dimensional Non-linear Classifier

Perceptron



Network

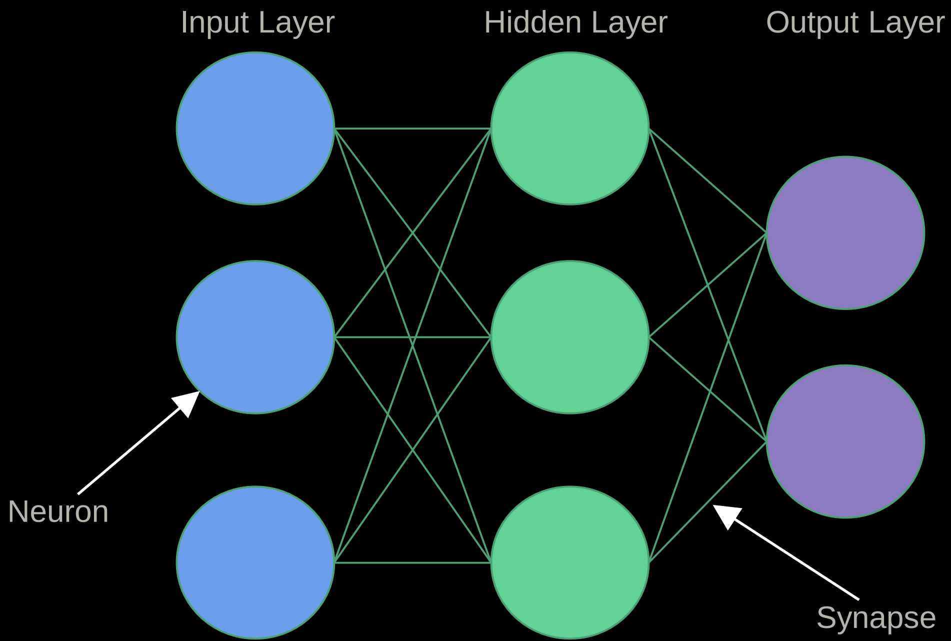


No Hidden Layer
Linear

Hidden Layers
Nonlinear

Courtesy: Chris Olah

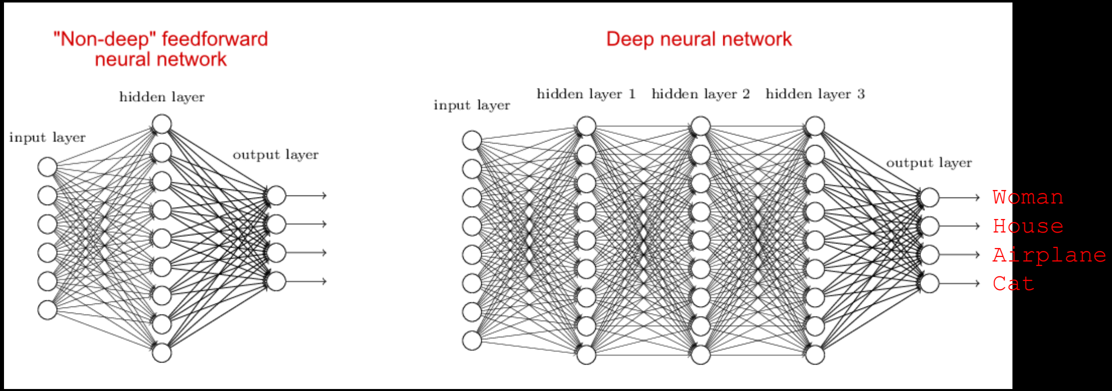
Basic NN Architecture



In Practice

How many inputs?

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



How deep?

100+ layers have become common.

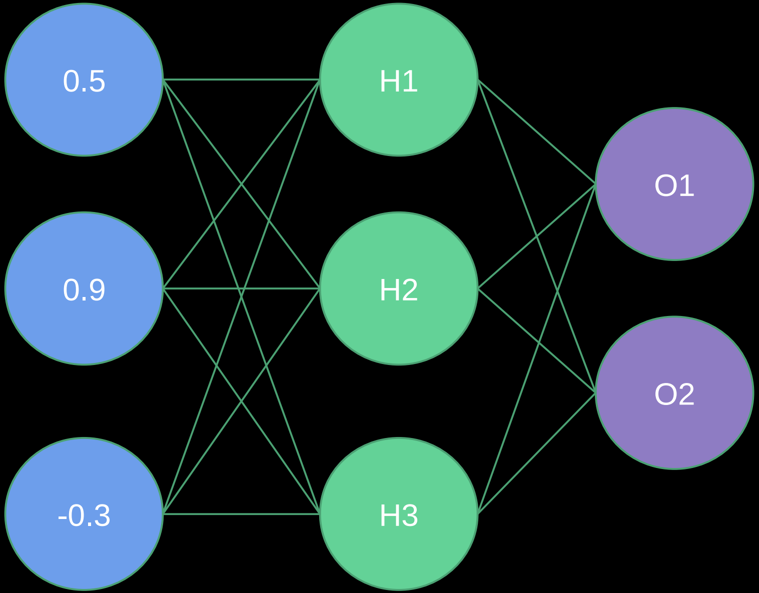
How many outputs?

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.

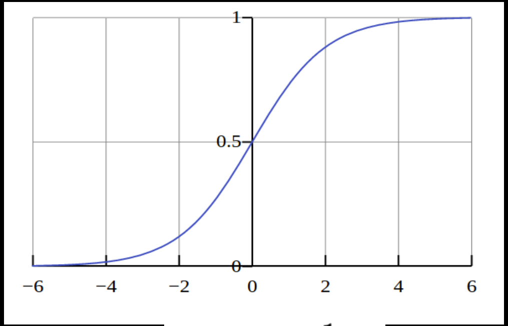
- The Rectified Linear activation function is zero when the input is negative and is equal to the input when the input is positive. Rectified Linear activation functions are currently the most popular activation function as they are more efficient than the sigmoid or hyperbolic tangent.

    - Sparse activation: In a randomly initialized network, only 50% of hidden units are active.

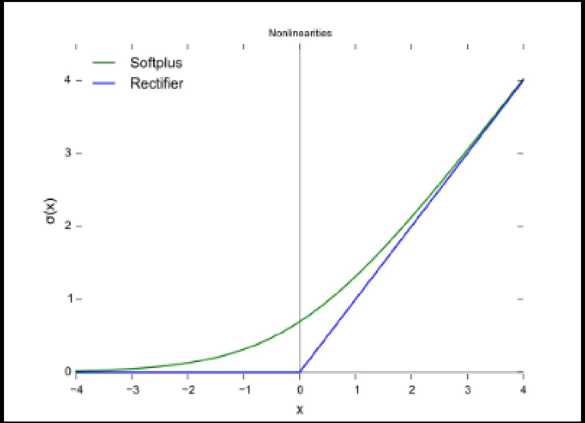
    - Better gradient propagation: Fewer vanishing gradient problems compared to sigmoidal activation functions that saturate in both directions.

    - Efficient computation: Only comparison, addition and multiplication.

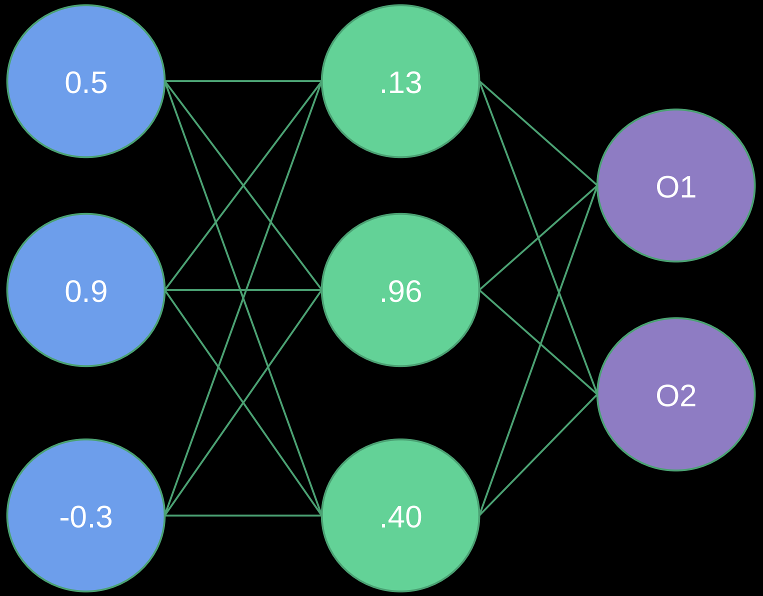
    - There are Leaky and Noisy variants.



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



Inference

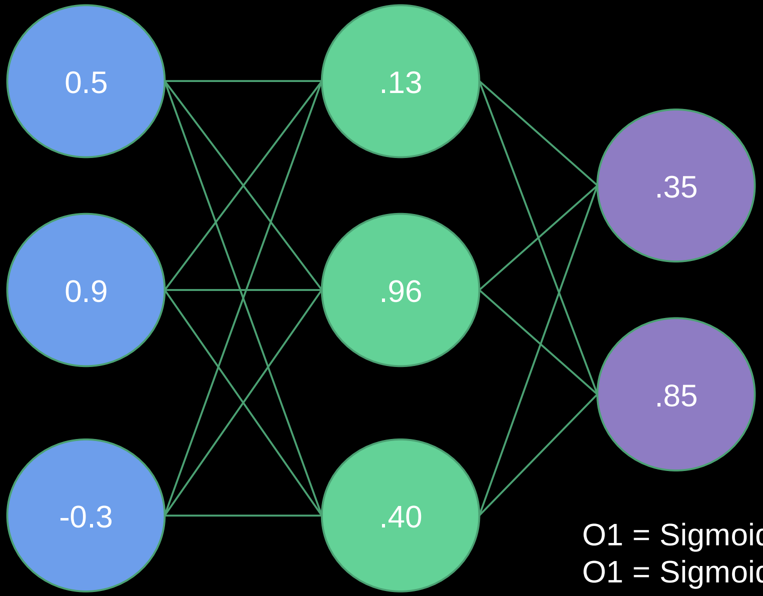


H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

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:

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

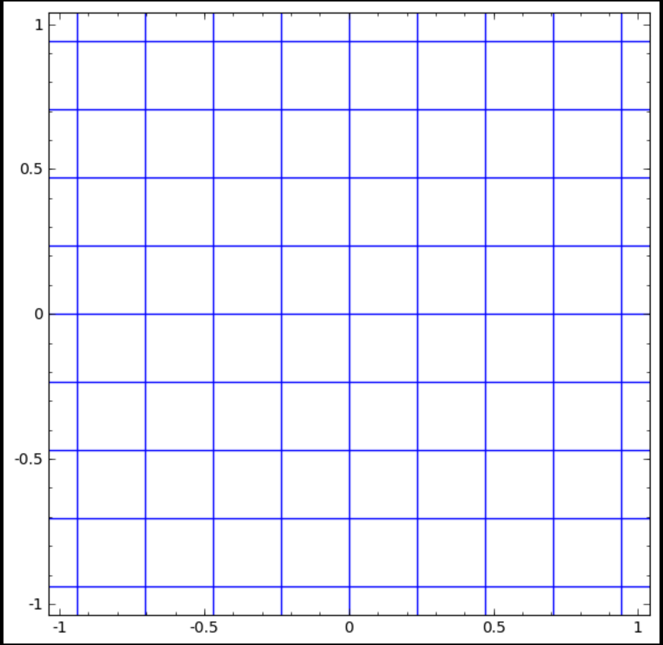
Linear + Nonlinear

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*.

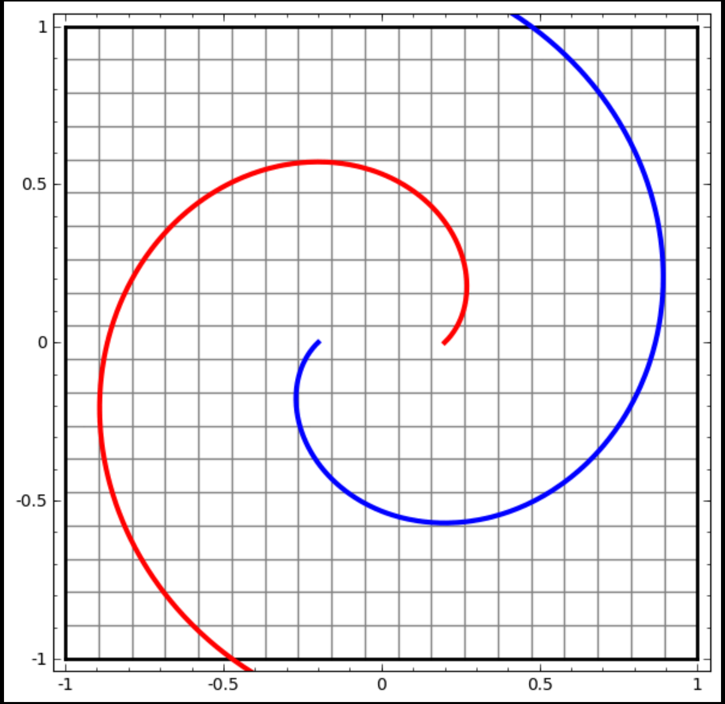
The bias provides the *translation*.

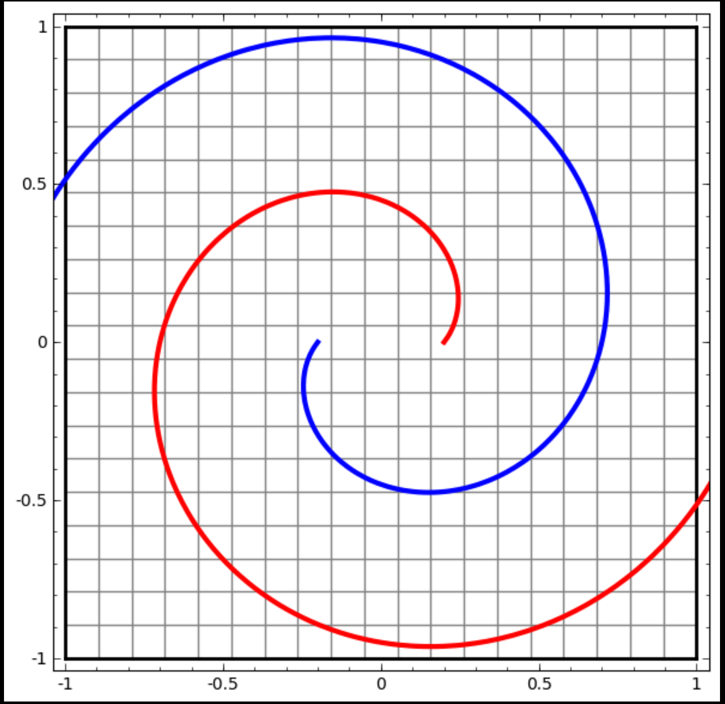
The activation function provides the *warp*.



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.

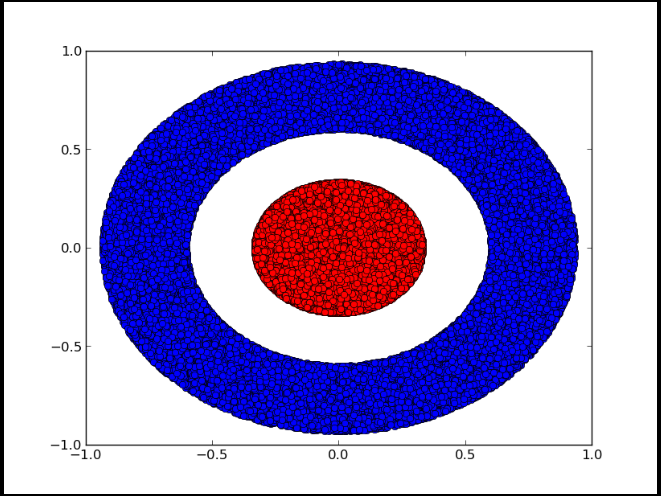




Courtesy: Chris Olah

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:

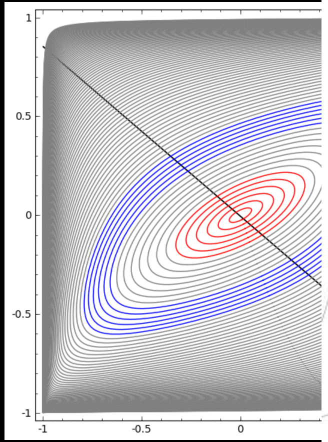


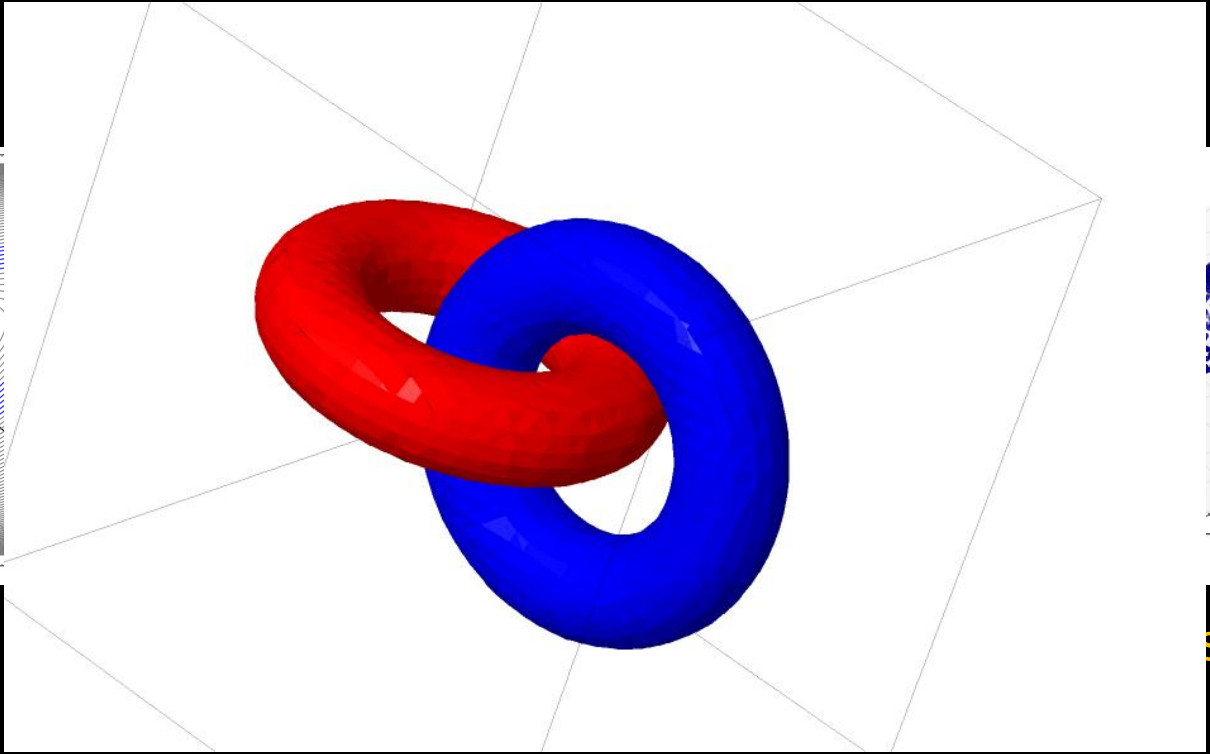
Can a neural net do this with twisting and deforming? What good does it do to have more than two dimensions with a 2D dataset?

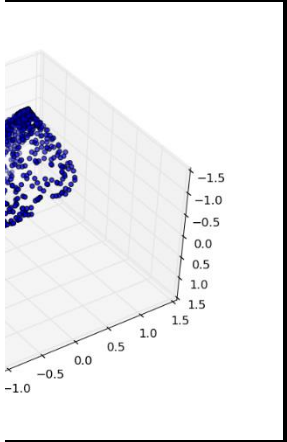
Courtesy: Chris Olah

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.

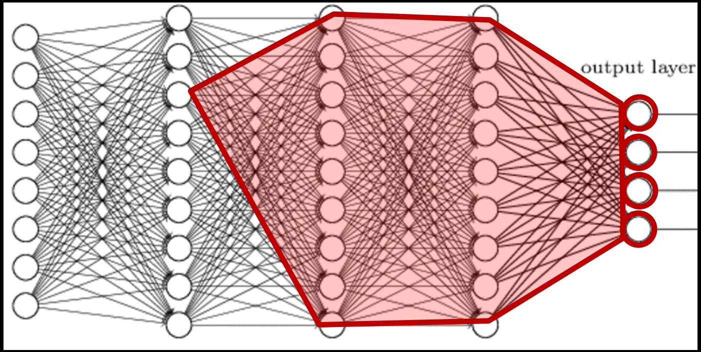
1) Borel measurable. Basically, mostly continuous and bounded.

2) Could be exponential number of hidden units, with one unit required for each distinguishable input configuration.

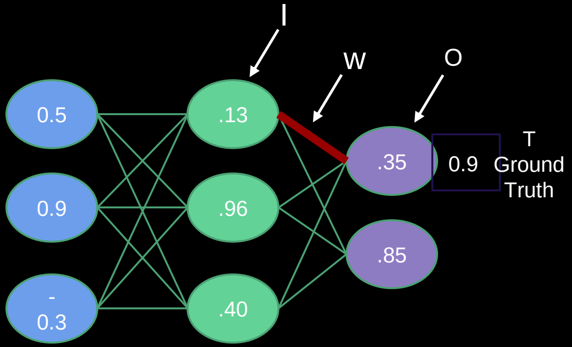
Training Neural Networks

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.

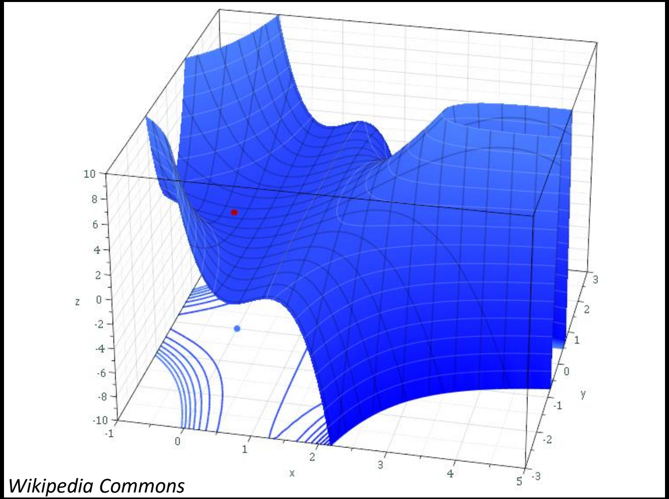
Solvers

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*.

Ready To Play Along?

Make sure you are on a GPU node:

```
br006% interact -gpu
gpu46%
```

Load the TensorFlow 2 Container:

```
[urbanic@gpu046 ~]$ module load singularity
[urbanic@gpu046 ~]$ singularity shell --nv /pylon5/containers/ngc/tensorflow_20.02-tf2-py3.sif
```

And start TensorFlow:

```
Singularity> python
Python 3.6.9 (default, Nov  7 2019, 10:44:
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "li
>>> import tensorflow
>>> ...some congratulatory noise...
>>>
```

Two Other Ways To Play Along

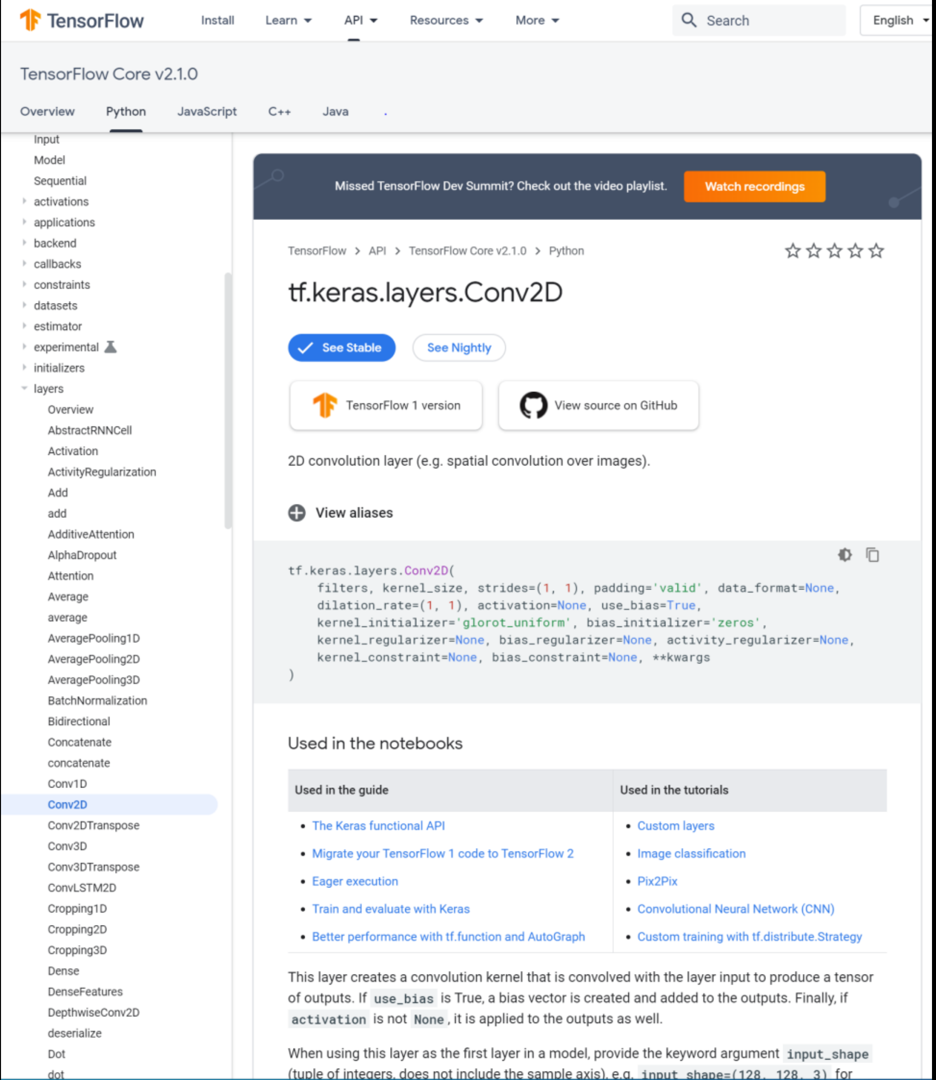
From inside the container, and in the right example directory,run the python programs from the command line:

```
gpu46% python CNN.py
```

or invoke them from within the python shell:

```
>>> exec(open("./CNN.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.

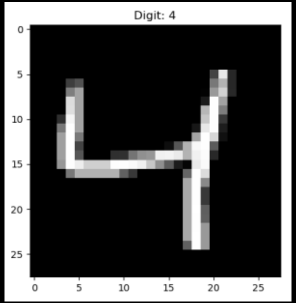
First we'll do this with the simplest possible model just to show how the TensorFlow framework functions. Then we will gradually implement our way to a quite sophisticated and accurate convolutional neural network for this same problem.

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

lt.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 -Σ 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

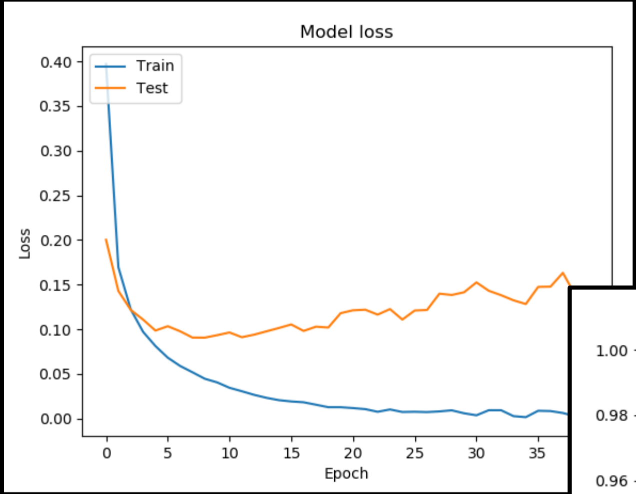
```
import tensorflow as tf

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))
```

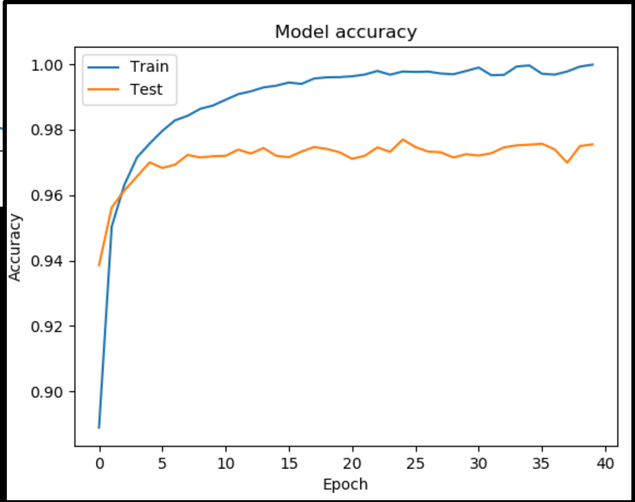
Results

matplotlib bonus insight

```
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


.9978 - val_loss: 0.1632 - val_accuracy: 0.9699

9750

0.9755
```



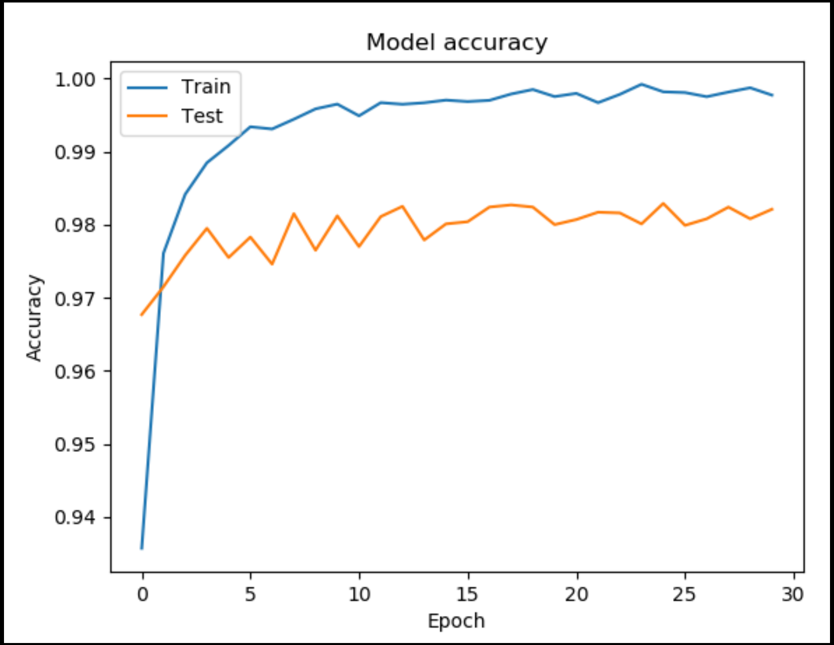


Let's Go Wider

```
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'])


history = 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
```

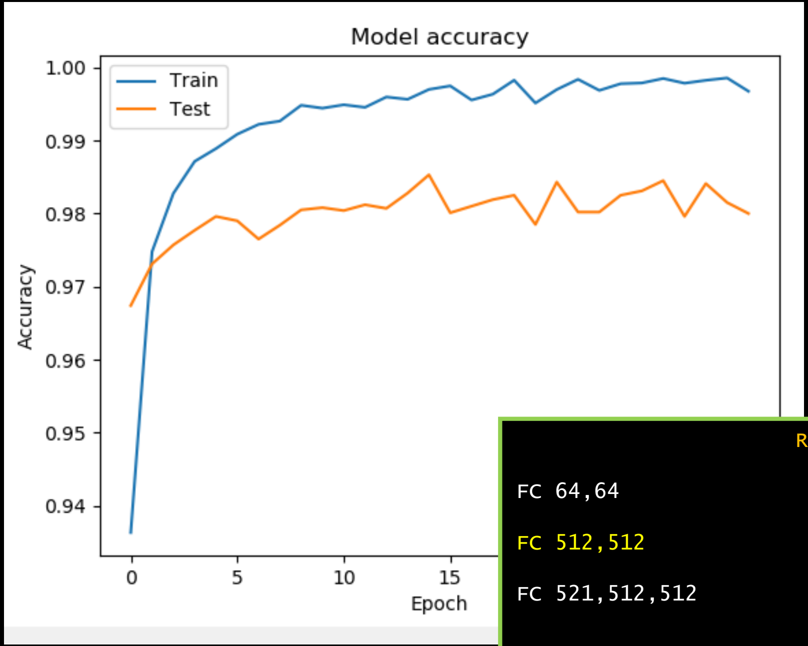
55,050 for 64 Wide Model

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'])


history = 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

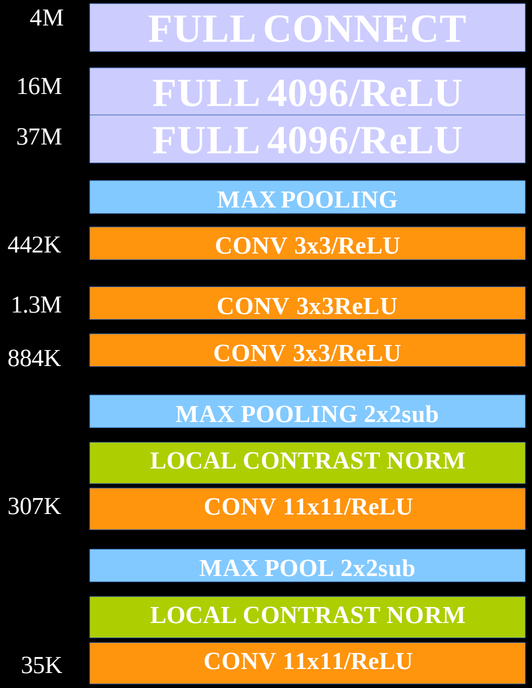
```
model.summary()
_________________________________________________________________
Layer (type)            Output Shape            Param #
=================================================================
dense_24 (Dense)         (None, 512)             401920
_________________________________________________________________
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.



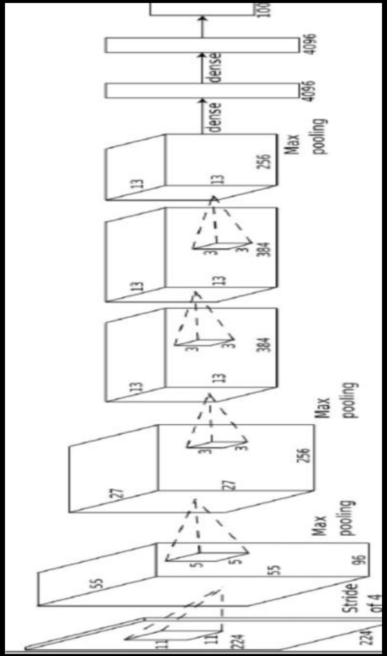
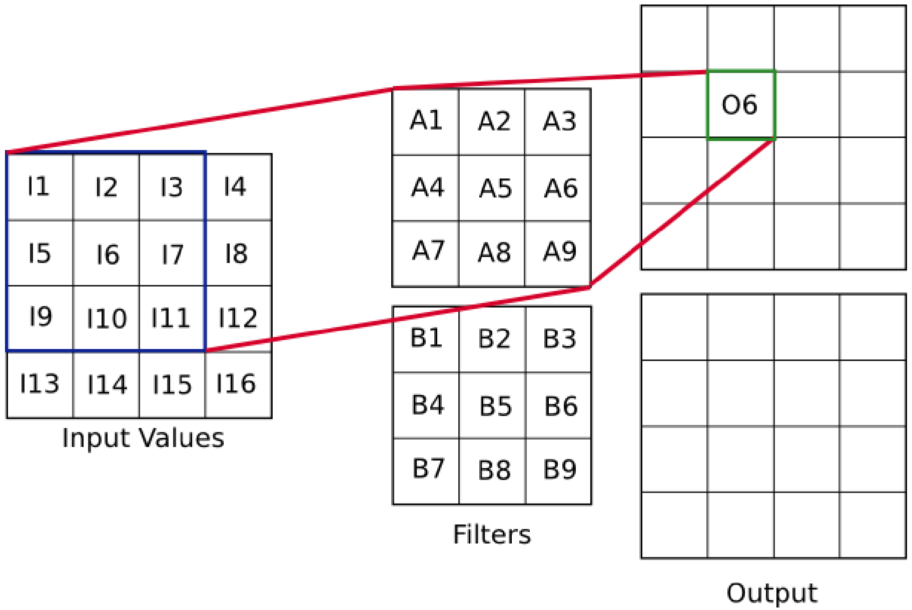


Image Object Recognition [Krizhevsky, Sutskever, Hinton 2012]

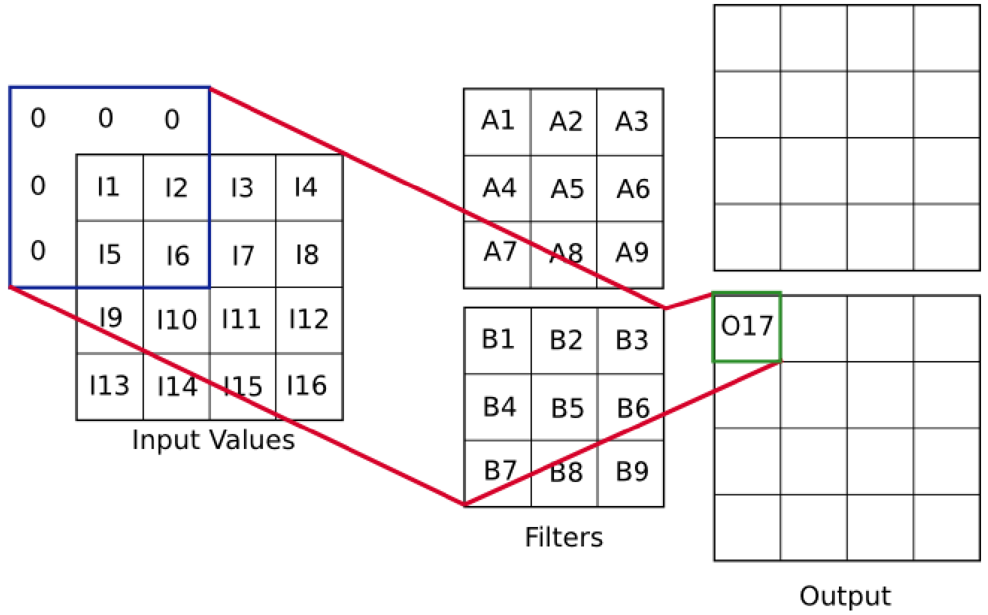
Convolution



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

Convolution

Boundary and Index Accounting



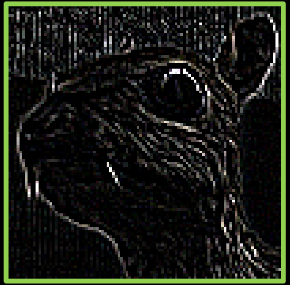
$$O_{17} = B_5 \cdot I_1 + B_6 \cdot I_2 + B_8 \cdot I_5 + B_9 \cdot I_6$$

Straight Convolution

+

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

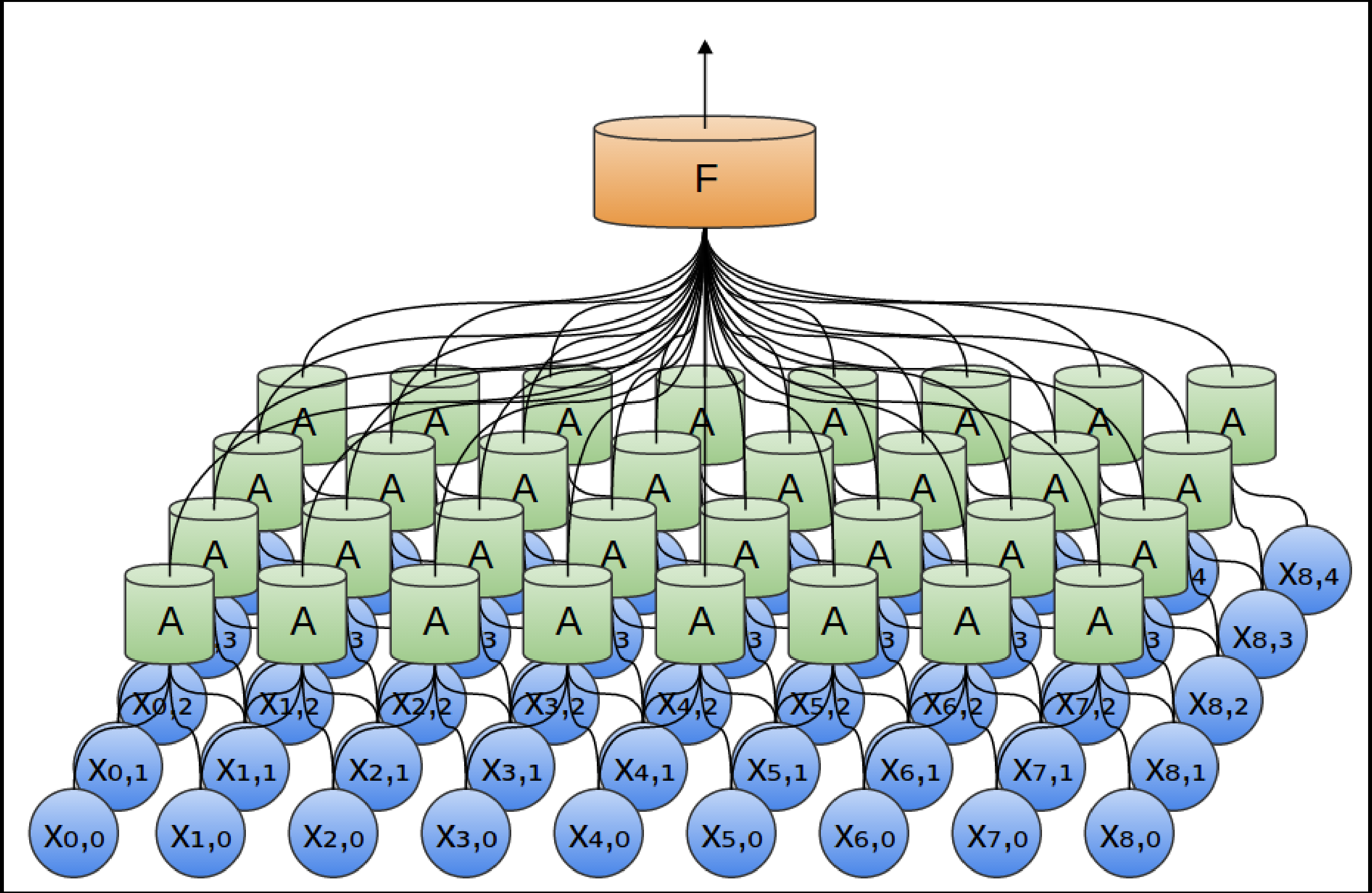
=



Edge Detector

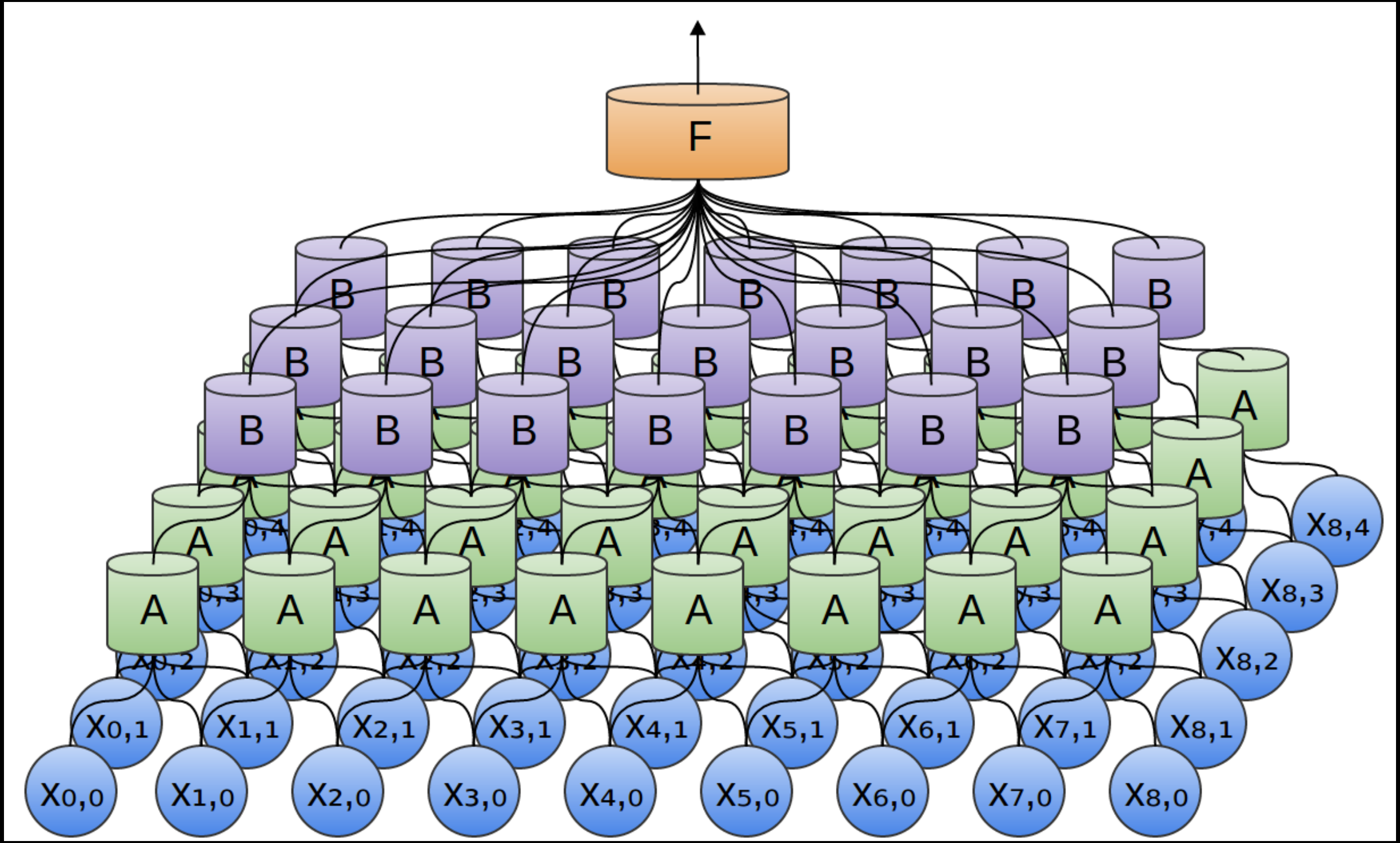
Images: Wikipedia

Simplest Convolution Net


F
A
X0,0 X1,0 X2,0 X3,0 X4,0 X5,0 X6,0 X7,0 X8,0
X0,1 X1,1 X2,1 X3,1 X4,1 X5,1 X6,1 X7,1 X8,1
X0,2 X1,2 X2,2 X3,2 X4,2 X5,2 X6,2 X7,2 X8,2
X8,3
X8,4

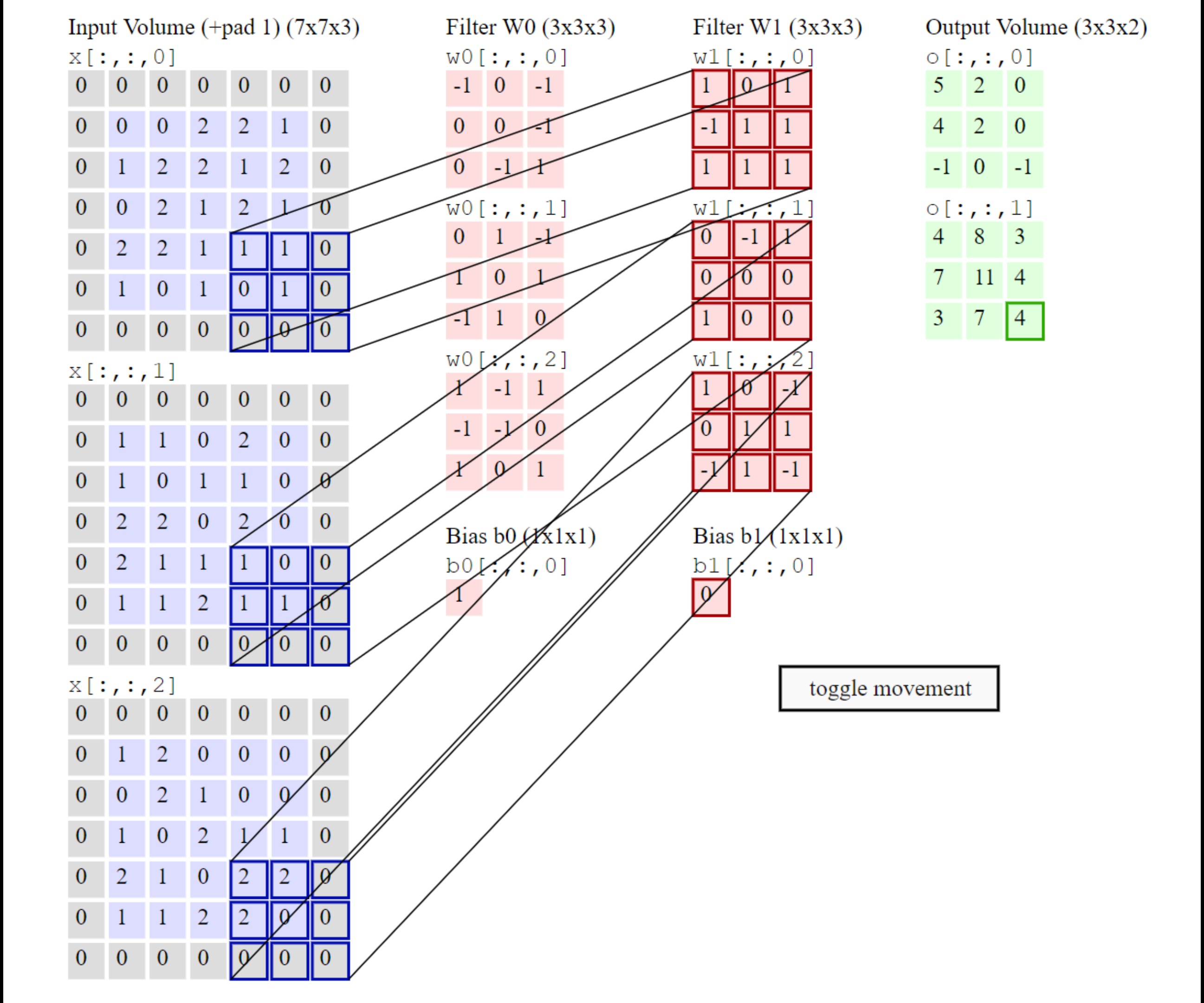
Courtesy: Chris Olah

Stacking Convolutions



Courtesy: Chris Olah

Convolution



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	0	0	2	2	1	0
0	1	2	2	1	2	0
0	0	2	1	2	1	0
0	2	2	1	1	1	0
0	1	0	1	0	1	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	1	1	0	2	0	0
0	1	0	1	1	0	0
0	2	2	0	2	0	0
0	2	1	1	1	0	0
0	1	1	2	1	1	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	1	2	0	0	0	0
0	0	2	1	0	0	0
0	1	0	2	1	1	0
0	2	1	0	2	2	0
0	1	1	2	2	0	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

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

w0[:,:,1]

0	1	-1
1	0	1
-1	1	0

w0[:,:,2]

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

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

1	0	1
-1	1	1
1	1	1

w1[:,:,1]

0	-1	1
0	0	0
1	0	0

w1[:,:,2]

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

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

5	2	0
4	2	0
-1	0	-1

o[:,:,1]

4	8	3
7	11	4
3	7	4

toggle movement

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 $W_O \times H_O \times D_O$

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

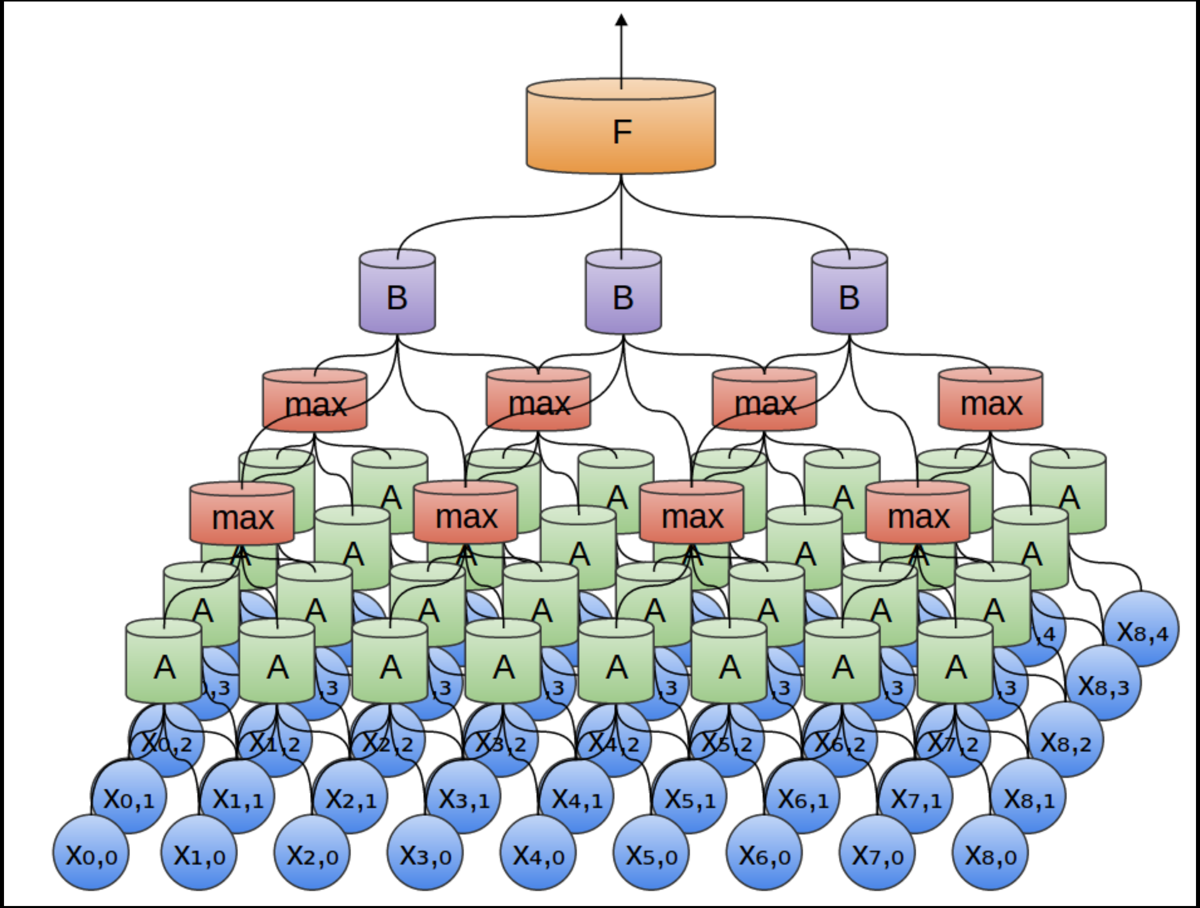
$H_O = (H_I - F + 2P) / S+1$

$D_O = K$

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.

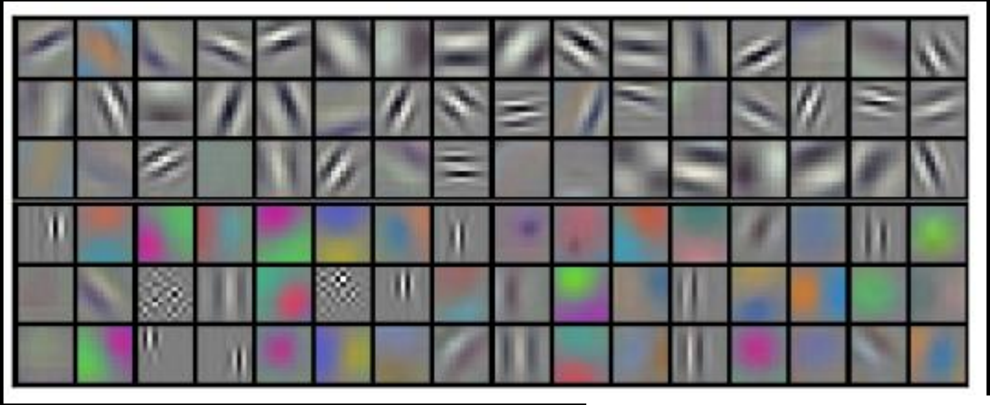
Pooling



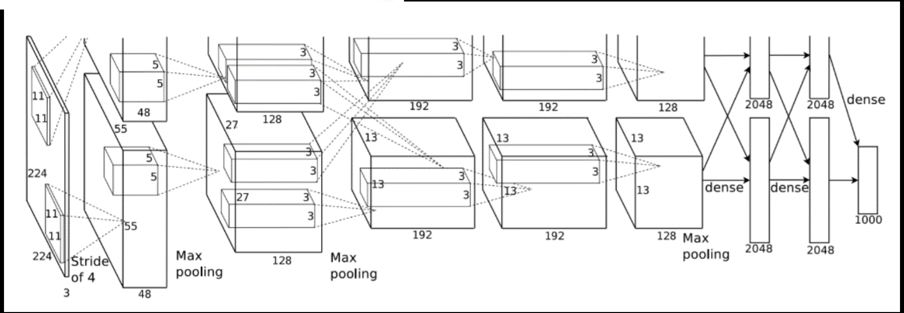
Courtesy: Chris Olah

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.

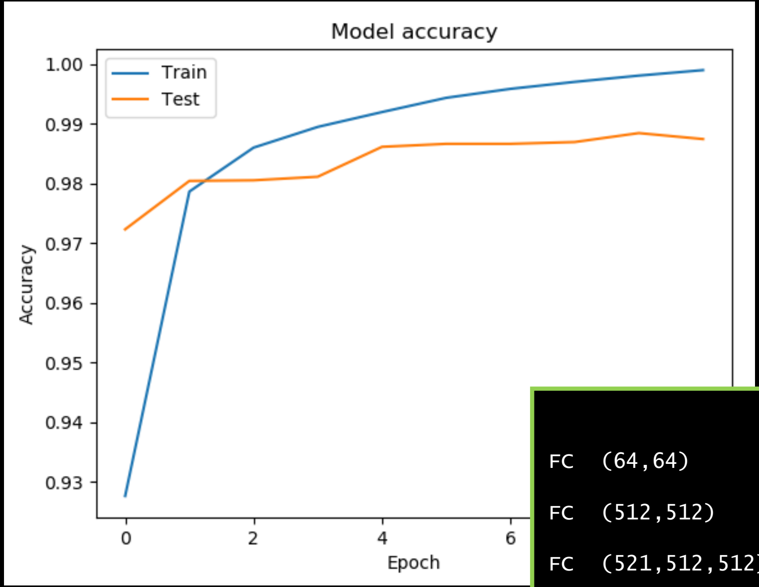


Let's Start Small

```
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
```



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
ble params: 0
```

Score Thus Far

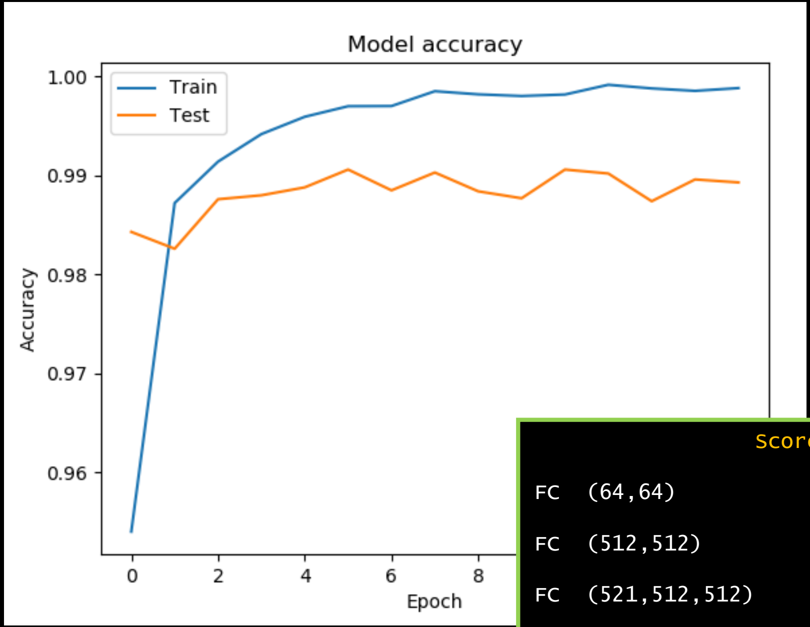
```
FC   (64,64)             97.5
FC   (512,512)           98.2
FC   (521,512,512)       98.0
CNN  (1 layer)           98.7
```

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
```



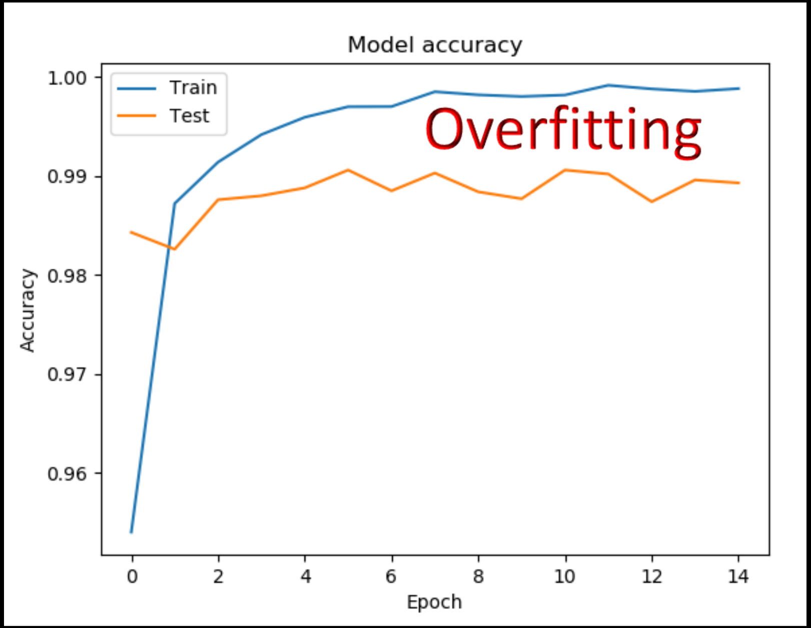
Deeper CNN

```
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
                =================================================
                ,199,882
                s: 1,199,882
                arams: 0
```

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

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.

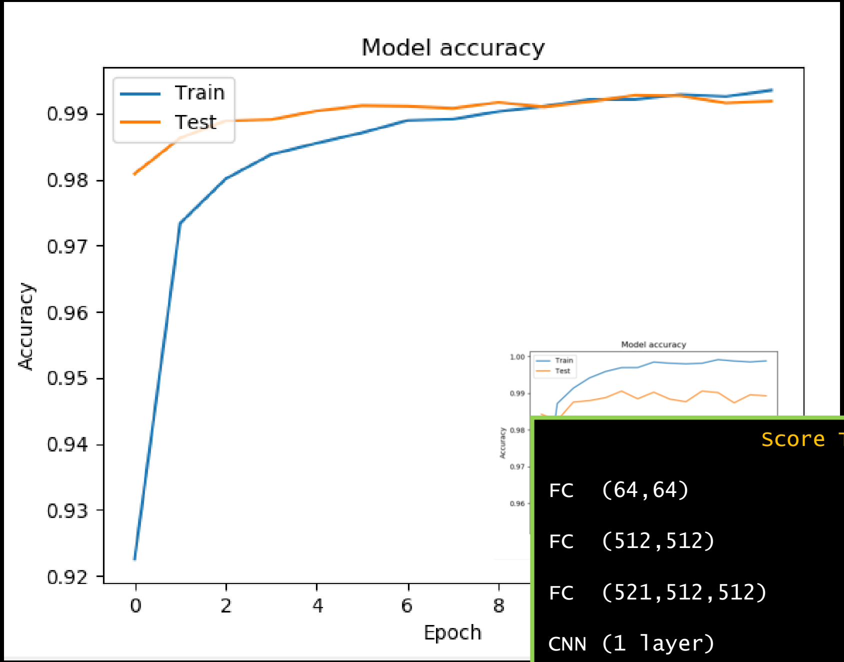
CNN With Dropout

```
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*.

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
```



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
=================================================================
         ams: 1,199,882
          params: 1,199,882
         able params: 0
```

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

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 of 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))
```

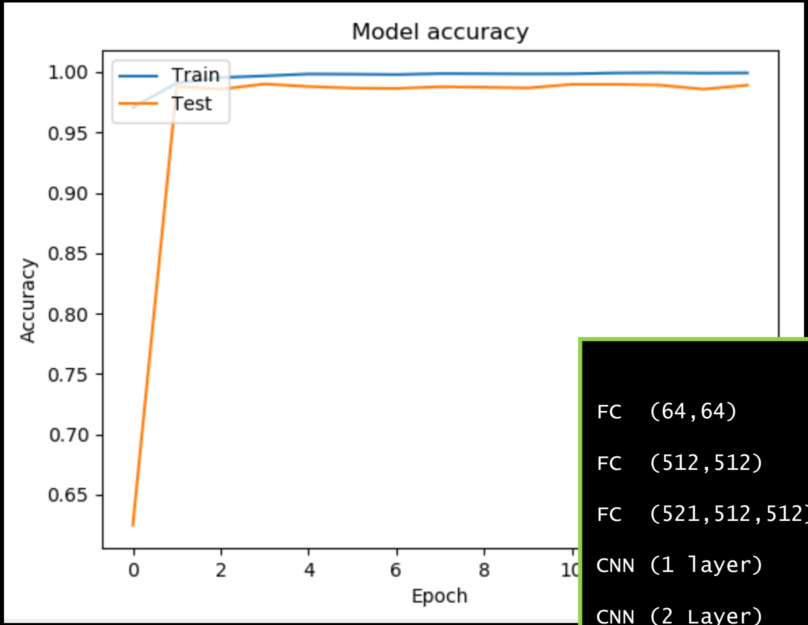
There are also normalizations that work on single samples instead of batches, so better for recurrent networks. In TensorFlow we have Group Normalization, Instance Normalization and Layer Normalization.

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
```



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
_________________________________________________________________
                 ation_1     (None, 12, 12, 64)        256
_________________________________________________________________
                 tten)       (None, 9216)              0
_________________________________________________________________
                 )           (None, 128)               1179776
_________________________________________________________________
                 ation_2     (Batch (None, 128)        512
_________________________________________________________________
                 )           (None, 10)                1290
=================================================================
                  1,200,778
                 ms: 1,200,330
                  params: 448
```

Score Thus Far

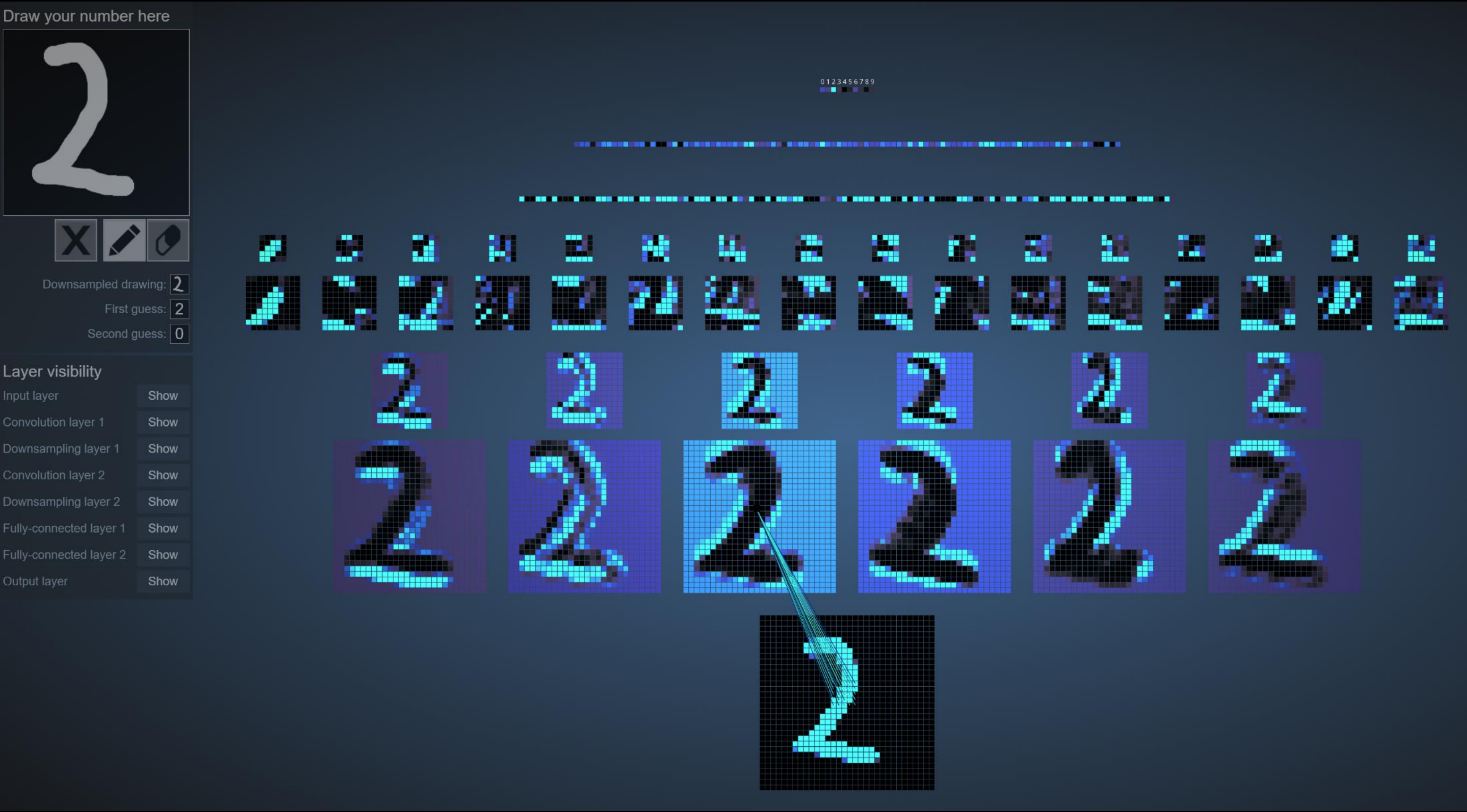
FC (64,64)	97.5
FC (512,512)	98.2
FC (521,512,512)	98.0
CNN (1 layer)	98.7
CNN (2 Layer)	99.0
CNN with Dropout	99.2
Batch Normalization	98.9

Real Time Demo

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

http://scs.ryerson.ca/~aharley/vis/conv/flat.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

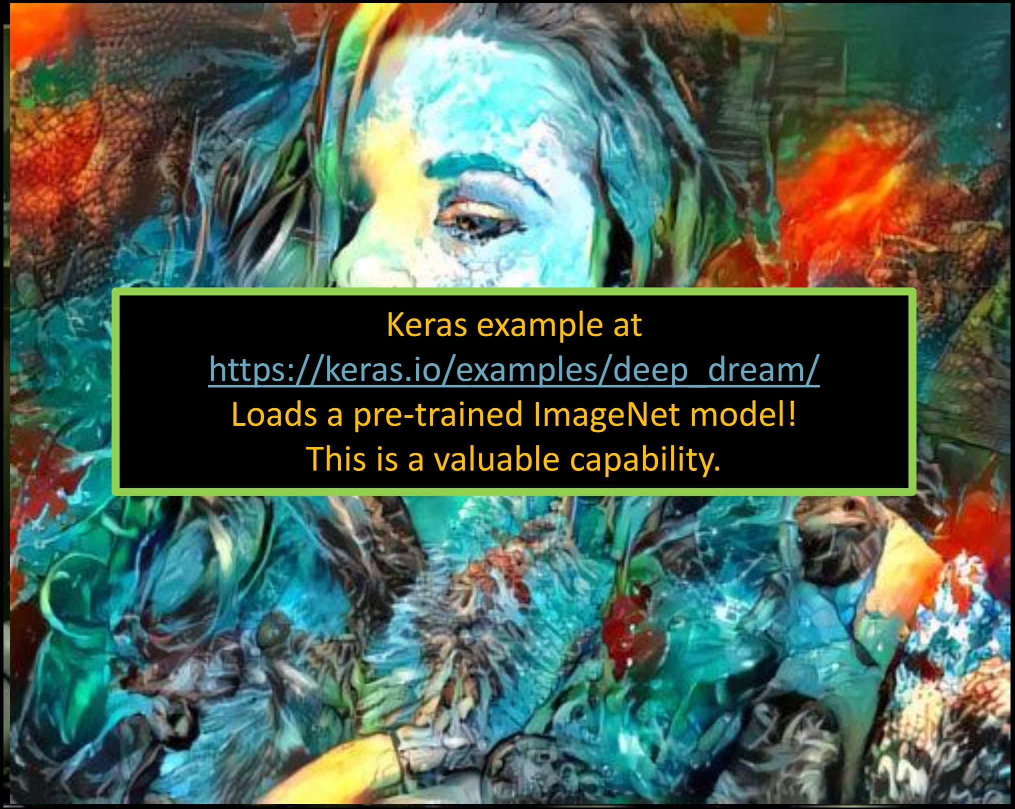
Layer visibility

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

Style vs. Content

Deep Dream Generator

https://deepdreamgenerator.com/feed

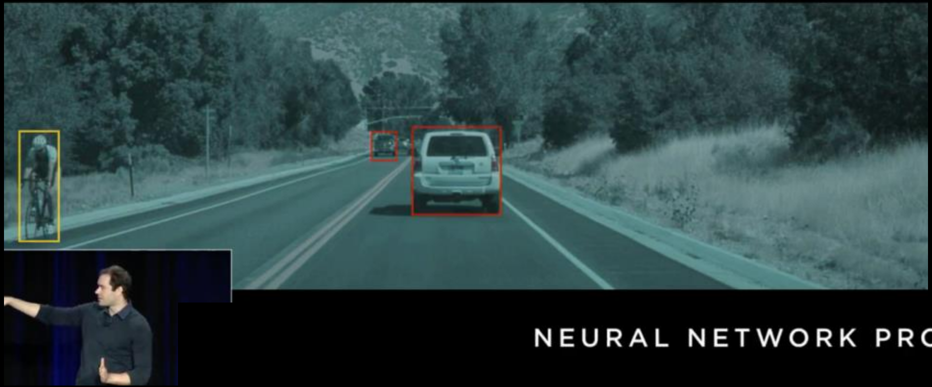


Keras example at
https://keras.io/examples/deep_dream/
Loads a pre-trained ImageNet model!
This is a valuable capability.

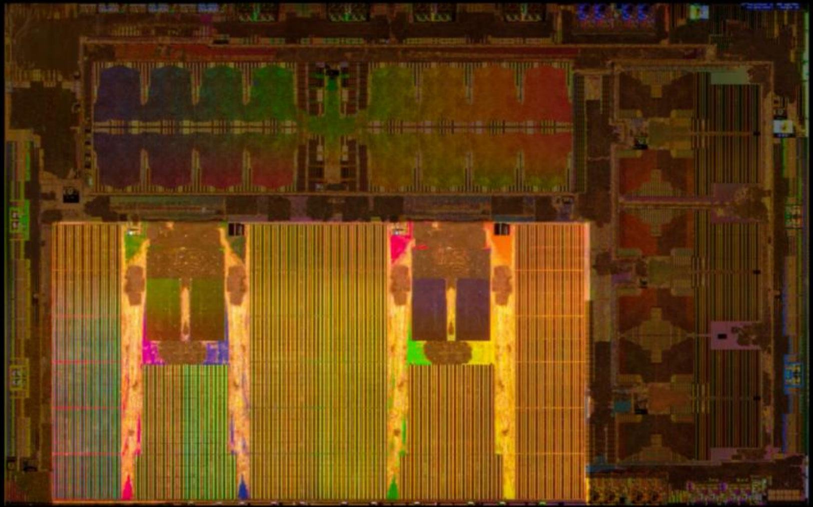
Inference Is Fast

Perceptual Labs

iPhone Demo



NEURAL NETWORK PROCESSOR



32MB SRAM

96x96 Mul/Add array

ReLU hardware

Pooling hardware

36 TOPS @ 2 GHz

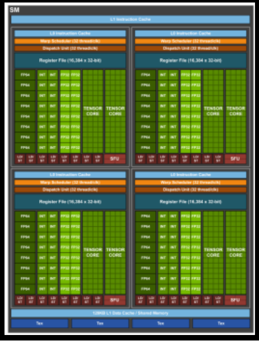
2 per chip, 72 TOPS total

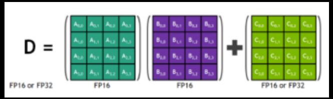
TESLA LIVE

Everyone Doing Specialized Hardware

NVIDIA

Volta/Turing
Tensor Cores

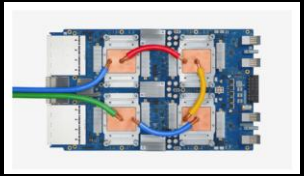




4x4 matrix mixed precision matrix multiply machines. 125 FP16 TFlops.

Google

TPU



Cloud TPU v3
420 teraflops
128 GB HBM.

Intel

Loihi

128-core, 130,000 artificial neurons, and 130 million synapses + 3 managing Lakemont cores.

Also new AVX512_VNNI (Vector Neural Network) instructions like an FMA instruction for 8-bit multiplies with 32-bit accumulates on new processors.

Neuromorphic

IBM, ...

Brain only uses 20W.

Analog, pruning, spiking, lots of new directions.

We are also continuously learning how little we know about how biological mechanisms work.

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:

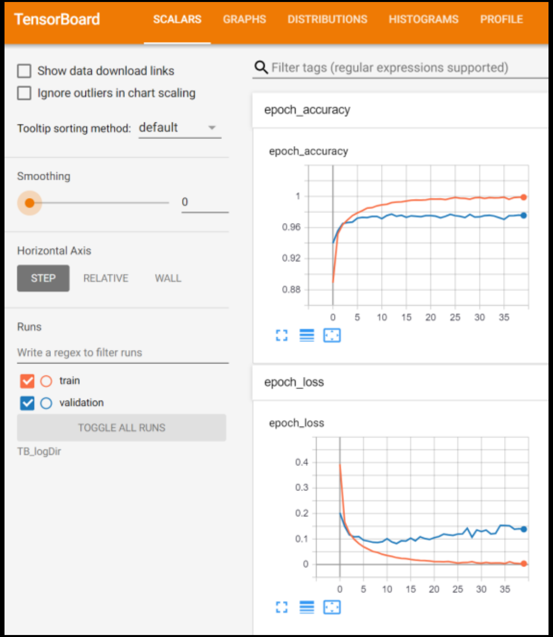
```
tensorboard --logdir=TB_logD
```

Somewhere else:

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

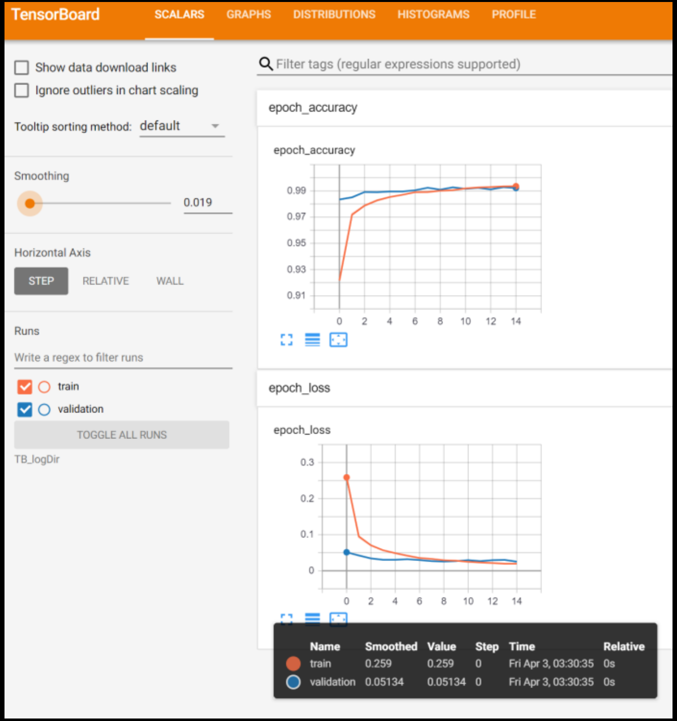
TensorBoard Analysis

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.



Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0

Our First Model
64 Wide FC



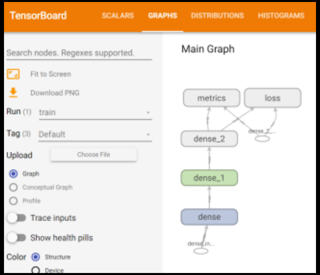
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.019

	Name	Smoothed	Value	Step	Time	Relative
●	train	0.259	0.259	0	Fri Apr 3, 03:30:35	0s
●	validation	0.05134	0.05134	0	Fri Apr 3, 03:30:35	0s

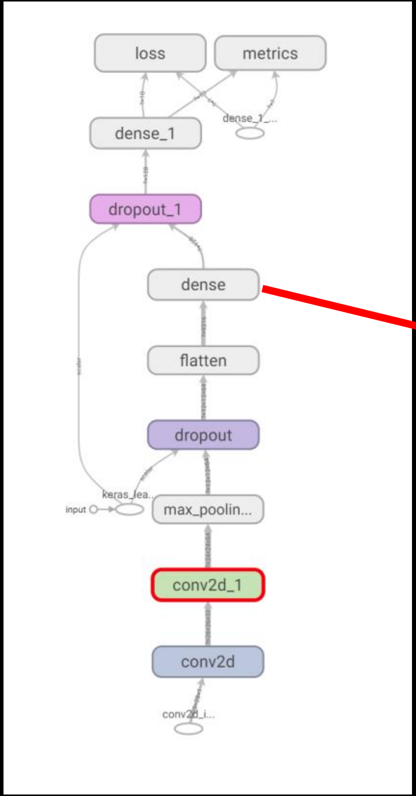
Our CNN

TensorBoard Graph Views

We can explore the architecture of the deep learning graphs we have constructed.

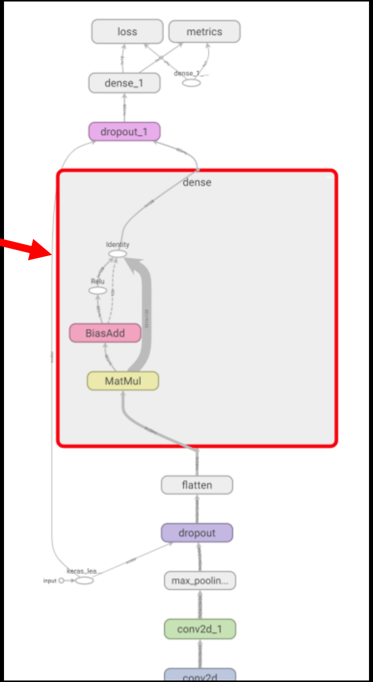


Our First Model
64 Wide FC

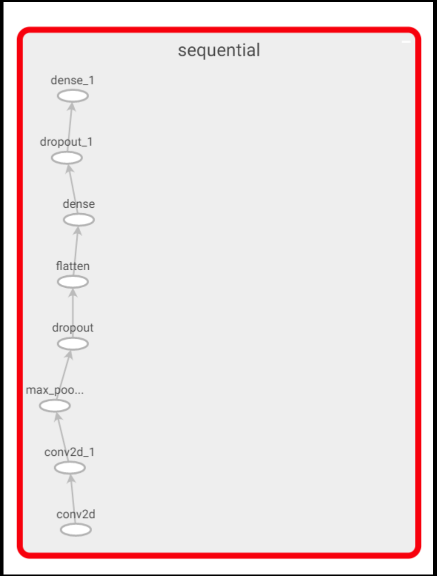


Our CNN

And we can drill down.



Our CNN's
FC Layer

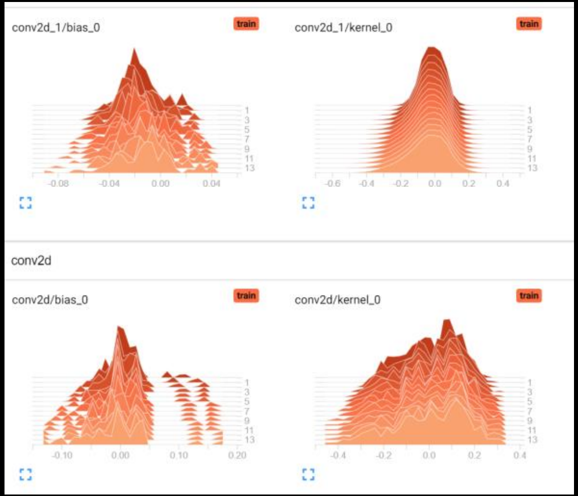


Keras
"Conceptual
Model"
View
of CNN

TensorBoard Parameter Visualization

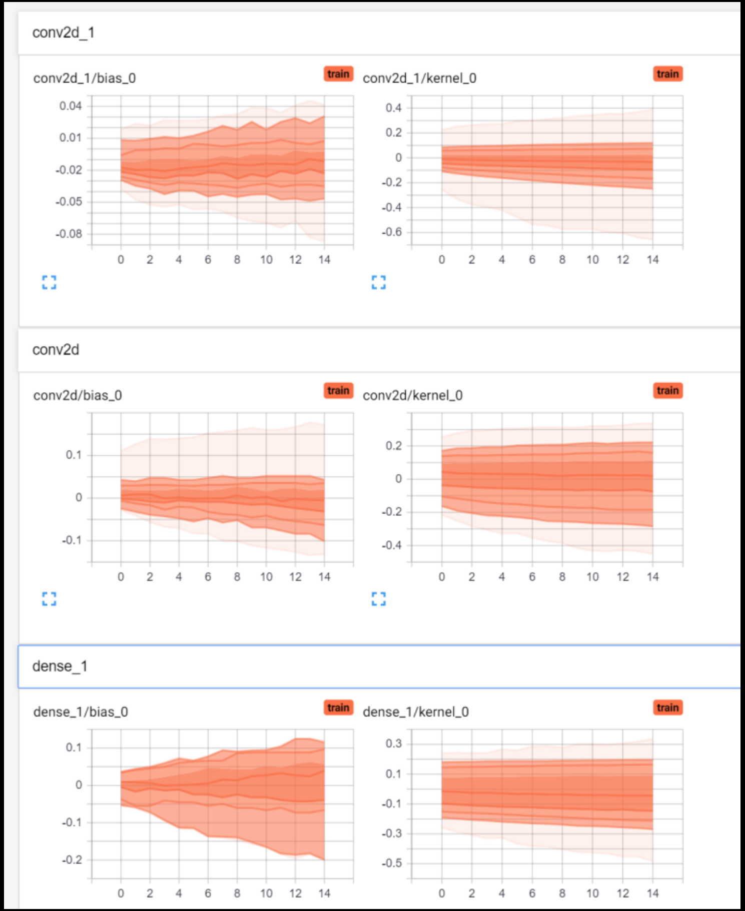
And we can observe the time evolution of our weights and biases, or at least their distributions.

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



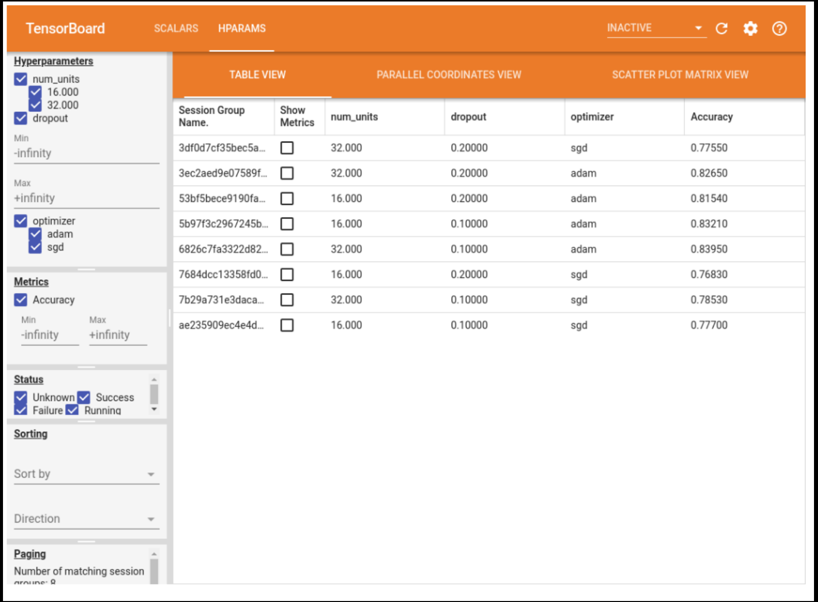
Histogram View

Distribution View



TensorBoard Add Ons

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



Hyperparameters
- [✓] num_units
  - [✓] 16.000
  - [✓] 32.000
- [✓] dropout
  - Min -infinity
  - Max +infinity
- [✓] optimizer
  - [✓] adam
  - [✓] sgd

Metrics
- [✓] Accuracy
  - Min -infinity
  - Max +infinity

Status
- [✓] Unknown [✓] Success
- [✓] Failure [✓] Running

Sorting
- Sort by
- Direction

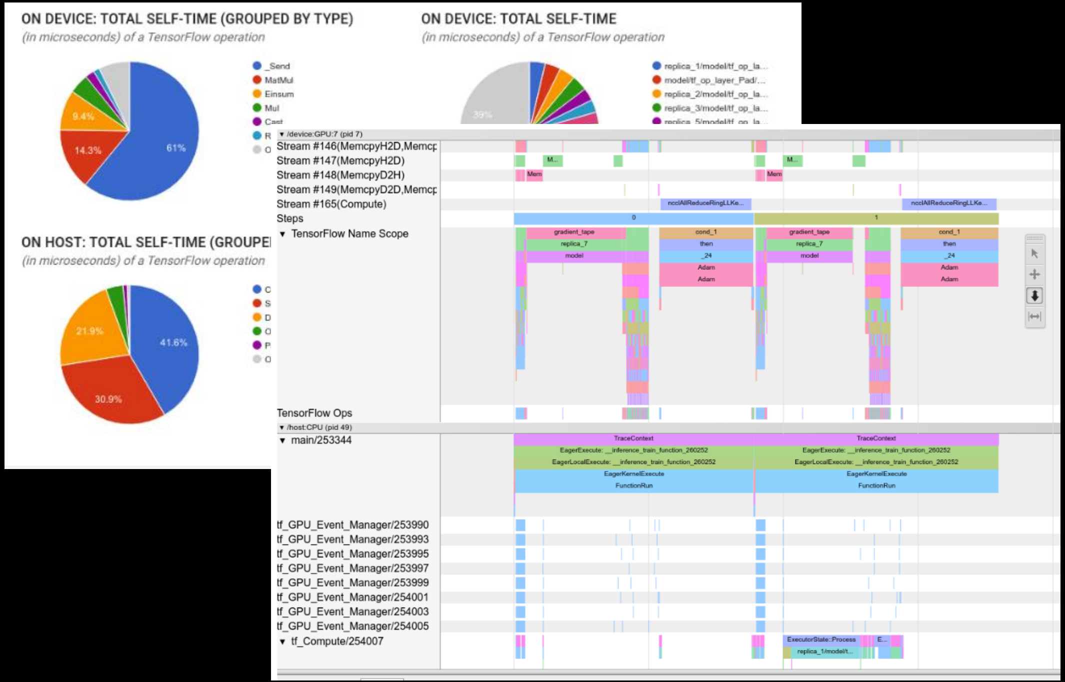
Paging
- Number of matching session groups: 8

Session Group Name.	Show Metrics	num_units	dropout	optimizer	Accuracy
3df0d7cf35bec5a...	☐	32.000	0.20000	sgd	0.77550
3ec2aed9e07589f...	☐	32.000	0.20000	adam	0.82650
53bf5bece9190fa...	☐	16.000	0.20000	adam	0.81540
5b97f3c2967245b...	☐	16.000	0.10000	adam	0.83210
6826c7fa3322d82...	☐	32.000	0.10000	adam	0.83950
7684dcc13358fd0...	☐	16.000	0.20000	sgd	0.76830
7b29a731e3daca...	☐	32.000	0.10000	sgd	0.78530
ae235909ec4e4d...	☐	16.000	0.10000	sgd	0.77700

Hyperparameter Search

Requires some scripting on your part. Look at https://www.tensorflow.org/tensorboard/hyperparameter_tuning_with_hparams for a good introduction.

Performance Profiling


ON DEVICE: TOTAL SELF-TIME (GROUPED BY TYPE)
(in microseconds) of a TensorFlow operation
_Send
MatMul
Einsum
Mul
Cast
61%
14.3%
9.4%
ON DEVICE: TOTAL SELF-TIME
(in microseconds) of a TensorFlow operation
ON HOST: TOTAL SELF-TIME (GROUPE
(in microseconds) of a TensorFlow operation
41.6%
30.9%
21.9%
/device:GPU:7 (pid 7)
Stream #146(MemcpyH2D,Memcp
Stream #147(MemcpyH2D)
Stream #148(MemcpyD2H)
Stream #149(MemcpyD2D,Memcp
Stream #165(Compute)
Steps
TensorFlow Name Scope
TensorFlow Ops
/host:CPU (pid 49)
main/253344
TraceContext
EagerExecute
EagerKernelExecute
FunctionRun
tf_GPU_Event_Manager/253990
tf_GPU_Event_Manager/253993
tf_GPU_Event_Manager/253995
tf_GPU_Event_Manager/253997
tf_GPU_Event_Manager/253999
tf_GPU_Event_Manager/254001
tf_GPU_Event_Manager/254003
tf_GPU_Event_Manager/254005
tf_Compute/254007

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

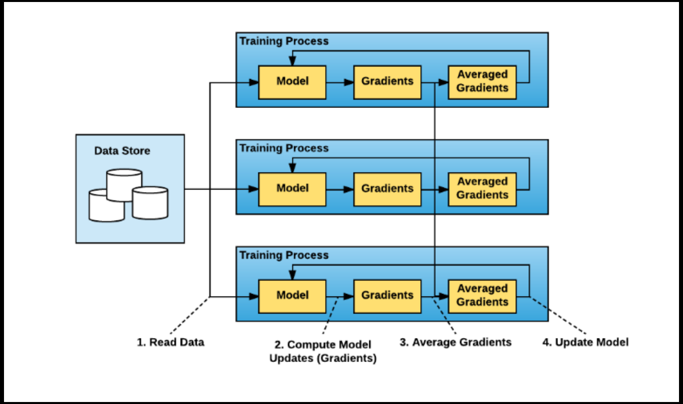
You may have the idea that deep learning has a voracious appetite for GPU cycles. That is absolutely the case, and the leading edge of research is currently limited by available resources. Researchers routinely use many GPUs to train a model. Conversely, the largest resources demand that you use them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

```
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    model = tf.keras.Sequential([
        tf.keras.layers.Dropout(rate=0.2, input_shape=X.shape[1:]),
        tf.keras.layers.Dense(units=64, activation='relu'),
        ...
    ])
    model.compile(...)
model.fit(...)
```

An alternative that has proven itself at extreme scale is *Horovod*.

MNIST with Horovod!

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

# Horovod: pin GPU to be used to process local rank (one GPU per process)
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
config.gpu_options.visible_device_list = str(hvd.local_rank())
K.set_session(tf.Session(config=config))
...
# Horovod: adjust number of epochs based on number of GPUs.
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'))
```

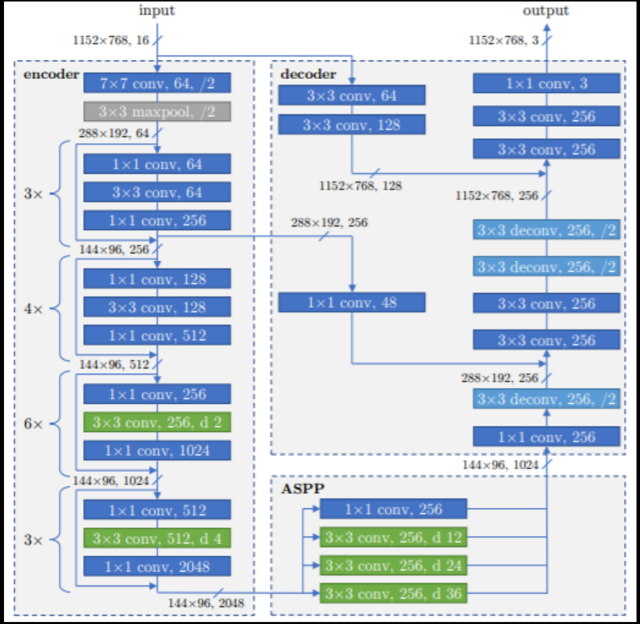


Horovod: fast and easy distributed deep learning in TensorFlow
Alexander Sergeev, Mike Del Balso

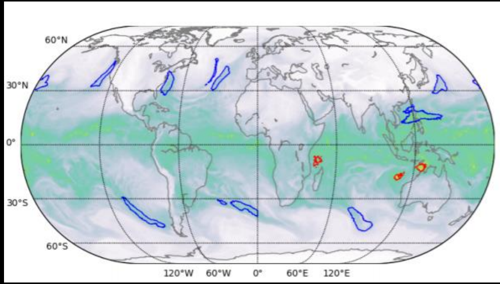
You can find a full example of using Horovod with a Keras MNIST code at
https://horovod.readthedocs.io/en/latest/keras.html

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*



Exascale Deep Learning for Climate Analytics
Kurth, et. al.

Other Tasks And Their Architectures

So far we have focused on images, and their classification. You know that deep learning has had success across a wide, and rapidly expanding, number of domains. Even our digit recognition task could be more sophisticated:

• Classification	(What we did)
• Localization	(Where is the digit?)
• Detection	(Are there digits? How many?)
• Segmentation	(Which pixels are the digits?)

These tasks would call for different network designs. This is where our Day 3 would begin, and we would use some other building blocks.

We don't have a Day 3, but we do have a good foundation to at least introduce the other important building blocks in current use.

Building Blocks

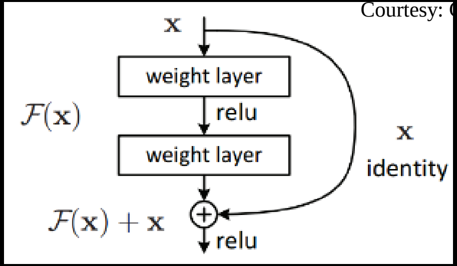
So far, we have used Fully Connected and Convolutional layers. These are ubiquitous, but there are many others:

Fully Connected	(FC)
Convolutional	(CNN)
Residual	(ResNet) [Feed forward]
Recurrent	(RNN), [Feedback, but has vanishing gradients so...]
Long Short Term Memory	(LSTM)
Transformer	(Attention based)
Bidirectional RNN	
Restricted Boltzmann Machine	

Several of these are particularly common...

Residual Neural Nets

We've mentioned that disappearing gradients can be an issue, and we know that deeper networks are more powerful. How do we reconcile these two phenomenae? One, very successful, method is to use some feedforward.



- Helps preserve reasonable gradients for very deep networks
- Very effective at imagery
- Used by AlphaGo Zero (40 residual CNN layers) in place of previous complex dual network
- 100s of layers common, Pushing 1000

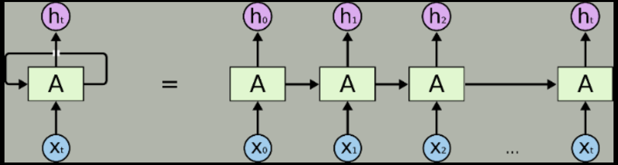
```
#Example: input 3-channel 256x256 image
x = Input(shape=(256, 256, 3))
y = Conv2D(3, (3, 3))(x)
z = keras.layers.add([x, y])
```

Haven't all of our Keras networks been built as strict layers in a *sequential* method? Indeed, but Keras supports a *functional* API that provides the ability to define network that branch in other ways. It is easy and here (https://www.tensorflow.org/guide/keras/functional) is an MNIST example with a 3 dense layers.

More to our current point, here (https://www.kaggle.com/yadavsarthak/residual-networks-and-mnist) is a neat experiment that uses *15*(!) residual layers to do MNIST. Not the most effective approach, but it works and illustrates the concept beautifully.

Recurrent Networks (RNNs)

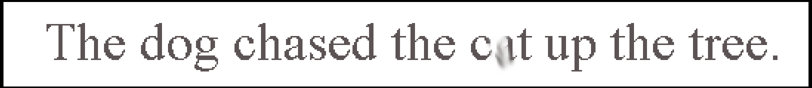
If feedforward is useful, is there a place for feedback? Indeed, it is currently at the center of the many of the most effective techniques in deep learning.



Many problems occur in some context. Our MNIST characters are just pulled from a hat. However most character recognition has some context that can greatly aid the interpretation, as suggested by the following - not quite true - text.

"Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoatnt tihng is taht the frist and lsat ltteers be at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe."

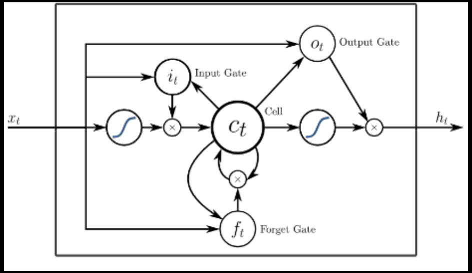
To pick a less confounding example. The following smudged character is pretty obvious by its context. If our network can "look back" to the previous words, it has a good chance at guessing the, otherwise unreadable, "a".



The dog chased the c[illegible]t up the tree.

LSTMs

This RNN idea seems an awful lot like "memory", and suggests that we might actually incorporate a memory into networks. While the Long Short Term Memory (LSTM) idea was first formally proposed in 1997 by Hochreiter and Schmidhuber, it has taken on many variants since. This is often not explained and can be confusing if you aren't aware. I recommend "LSTM: A Search Space Odyssey" (Greff, et. al.) to help.



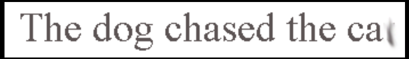
The basic design involves a memory cell, and some method of triggering a forget. *tf.keras.layers.LSTM* takes care of the details for us (but has a *lot* of options).

The Keras folks even provide us with an MNIST version (https://keras.io/examples/mnist_hierarchical_rnn/), although I think it is confusing as we are now killing a fly with a bazooka.

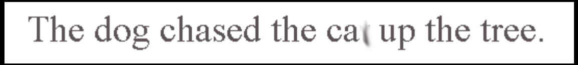
I recommend https://keras.io/examples/conv_lstm/, which uses network is used to predict the next frame of an artificially generated movie which contains moving squares. A much more natural fit.

Bi-directional LSTMs

Often, and especially in language processing, it is helpful to see both forward and backward. Take this example:

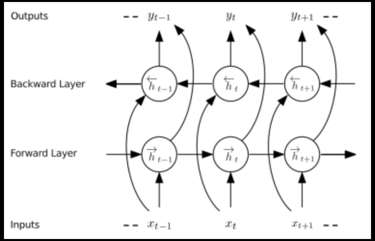

The dog chased the ca[illegible]

Is the dog chasing a cat, or a car? If we read the rest of the sentence, it is obvious:


The dog chased the ca[illegible] up the tree.

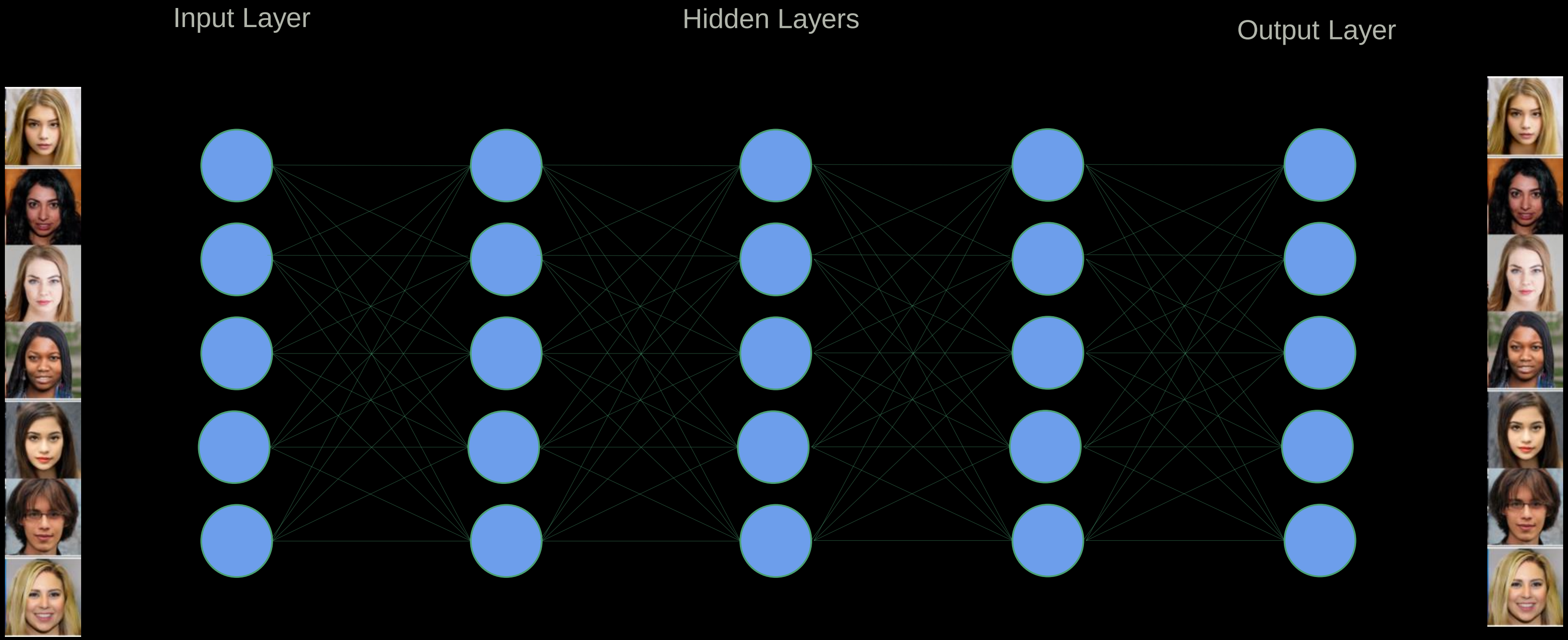
Adding even this very sophisticated type of network is easy in TF. Here is the network definition from the Keras *IMDB movie review sentiment analysis* example (https://www.tensorflow.org/tutorials/text/text_classification_rnn).

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(encoder.vocab_size, 64),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,  return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1)
])
```

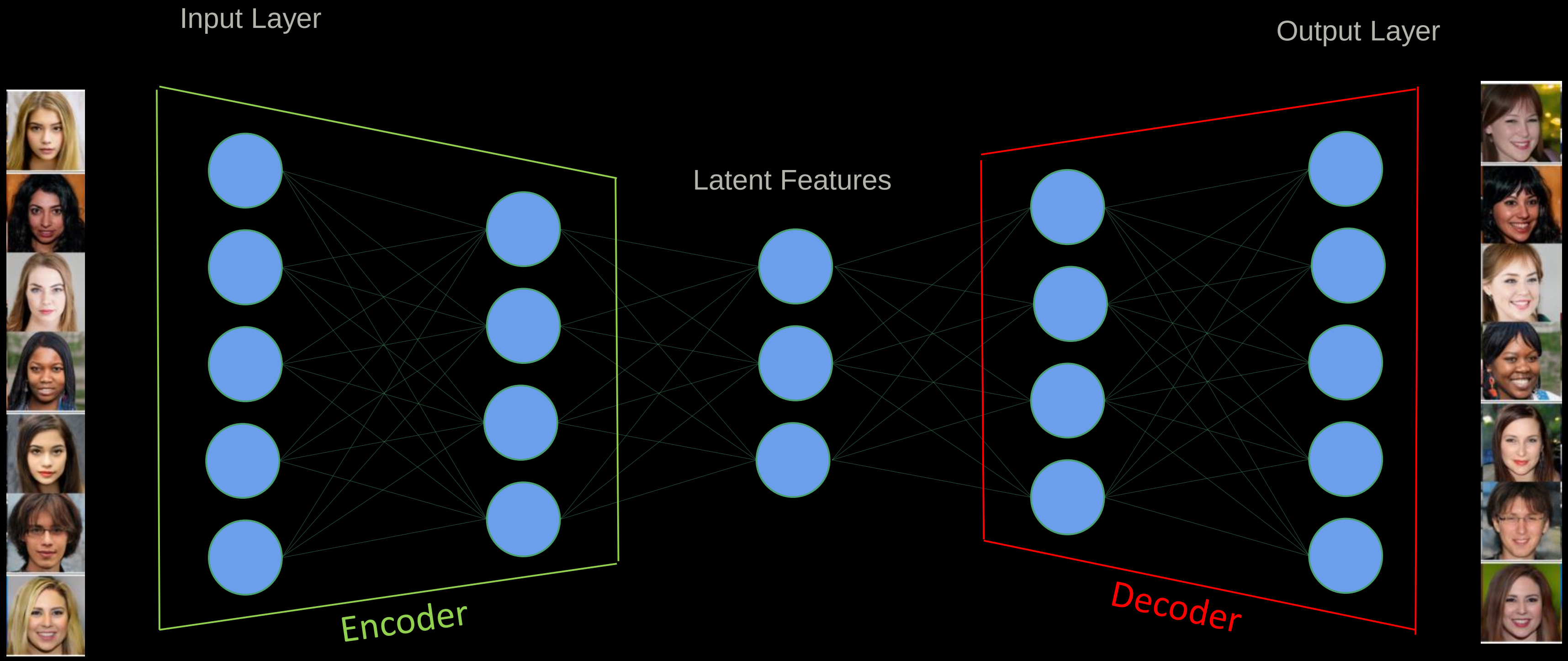


The first, embedding, layer introduces the concept of word embeddings - of central importance to any of you interested in natural language processing, and related to our running theme of dimensionality reduction. To oversimplify, here we are asking TF to reduce our vocabulary of vocab_size, so that every word's meaning is represented by a 64 dimensional vector.

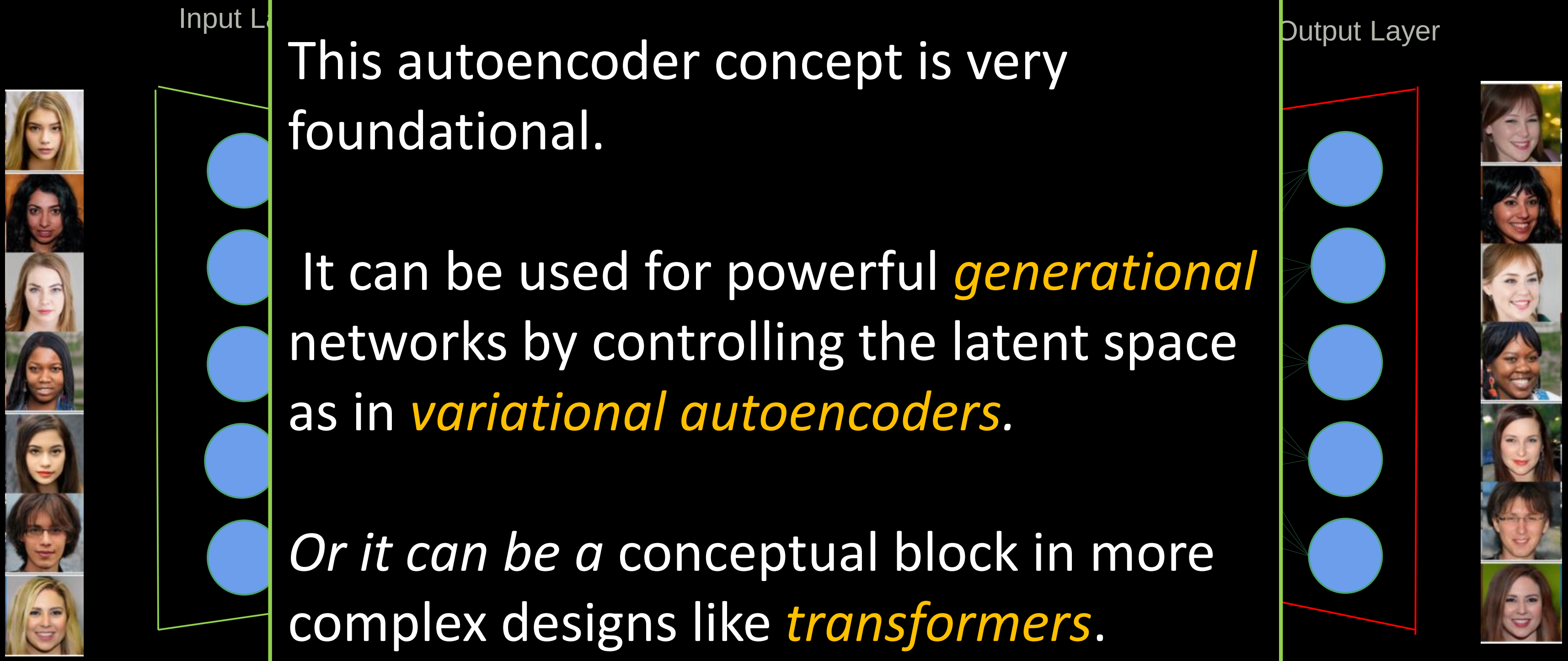
Autoencoder



Autoencoder



Autoencoder

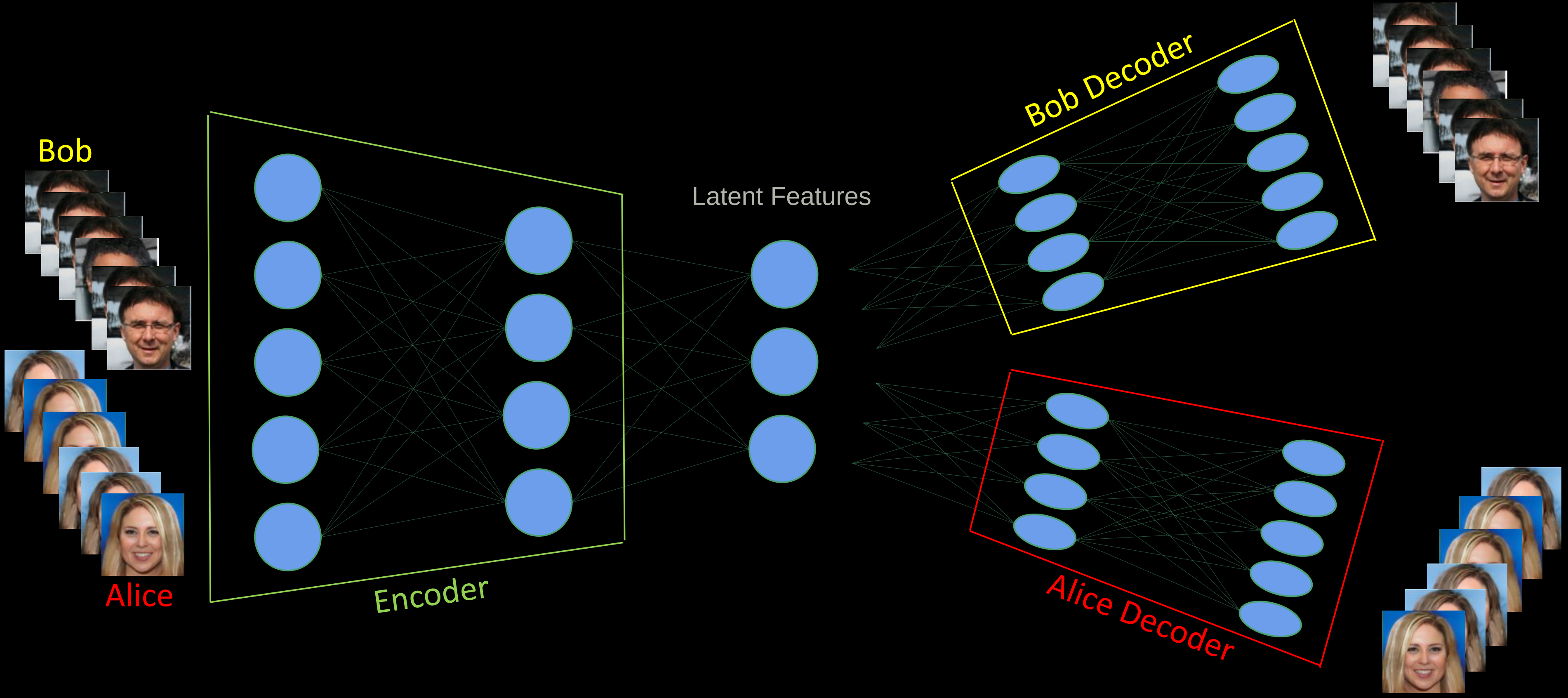


This autoencoder concept is very foundational.

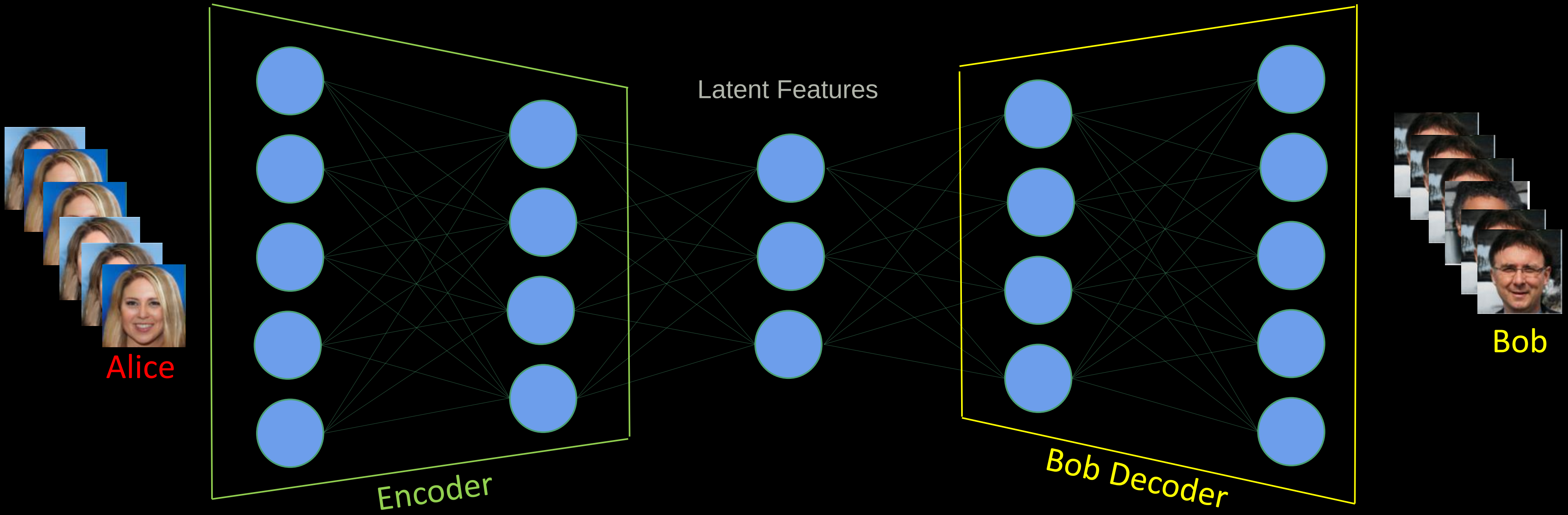
It can be used for powerful *generational* networks by controlling the latent space as in *variational autoencoders*.

Or it can be a conceptual block in more complex designs like *transformers*.

Deepfake Training



Deepfake At Work



Zao Does DiCaprio

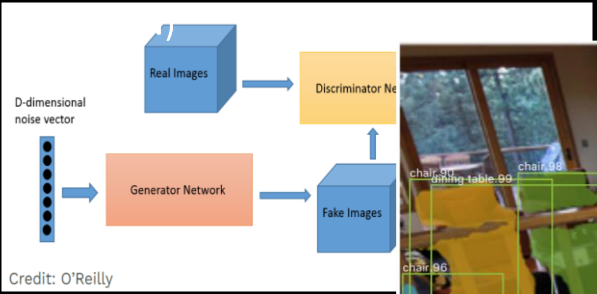
The Chinese app Zao did the below in 8 seconds from one photo.

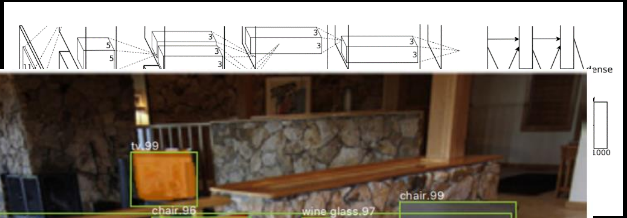
twitter.com/AllanXia/status/1168049059413643265

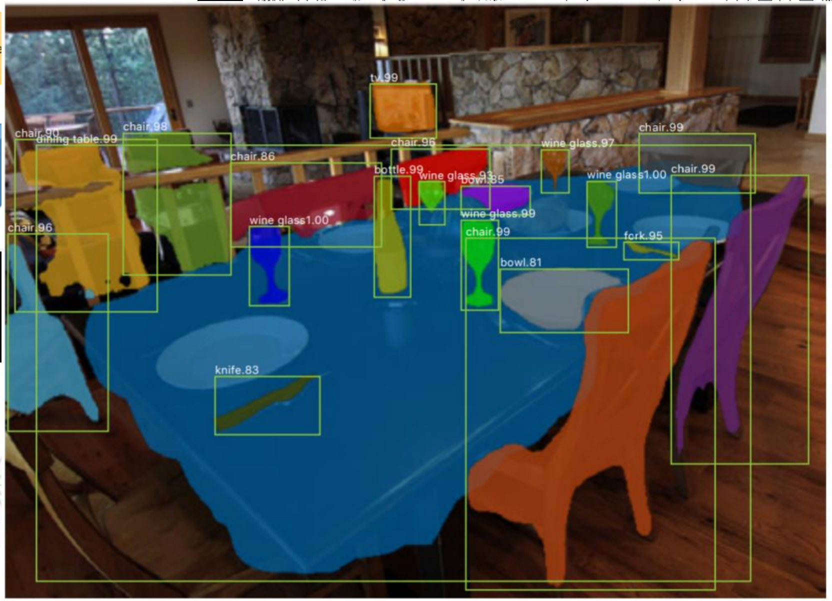
Architectures

With these layers, we can build countless different networks (and use TensorFlow to define them). Again, this is "3rd day" material, but we present them here and you should feel competent to research them yourself.

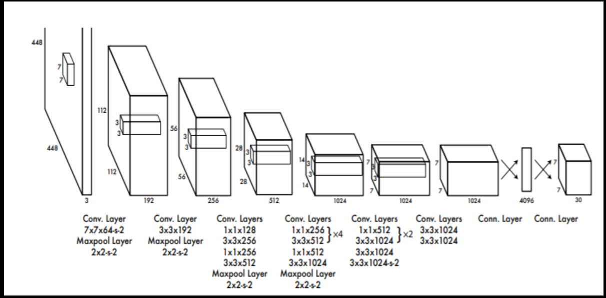
Generative Adversarial Network

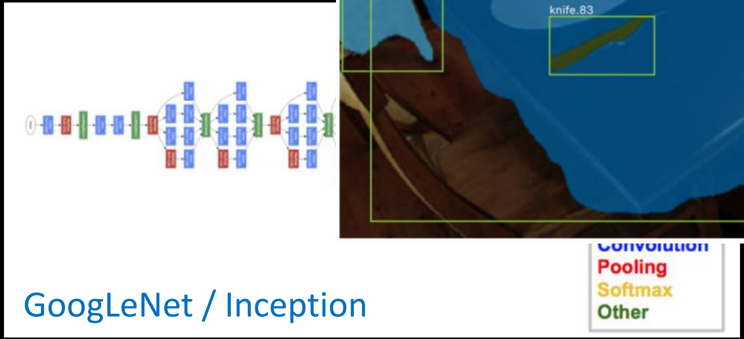




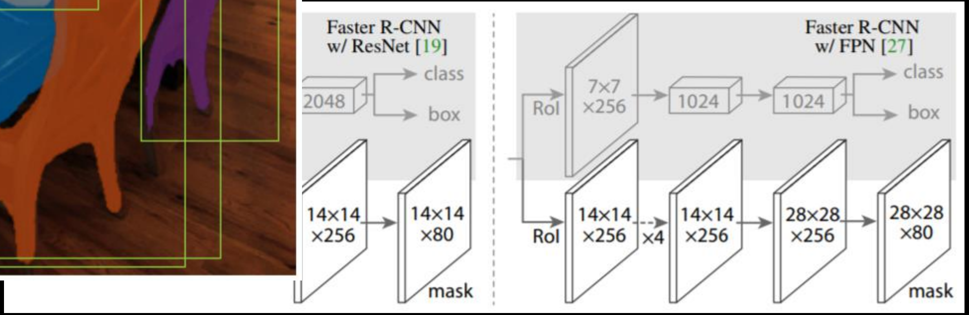


YOLO (You Only Look Once)


448
448
7
7
112
112
3
192
56
56
256
28
28
512
14
14
1024
7
7
1024
7
7
1024
4096
7
7
30
Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2
Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x256 3x3x512 ×4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x512 3x3x1024 ×2 3x3x1024 3x3x1024-s-2
Conv. Layers 3x3x1024 3x3x1024
Conn. Layer
Conn. Layer



GoogLeNet / Inception



Mask R-CNN

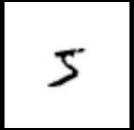
Images from original papers

Data Augmentation

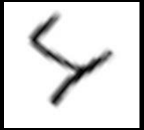
As I've mentioned, labeled data is valuable. This type of *supervised learning* often requires human-labeled data. Getting more out of our expensive data is very desirable. More datapoints generally equals better accuracy. The process of generating more training data from our existing pool is called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

Our MNIST network has learned to recognize very uniformly formatted characters:

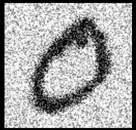
What if we wanted to teach it:



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.

Learning Approaches

Supervised Learning

How you learned colors.
What we have been doing just now.
Used for: image recognition, tumor identification, segmentation.
Requires labeled data.
Lots of it. Augmenting helps.

Reinforcement Learning

How you learned to walk.
Requires goals (maybe long term, i.e. arbitrary delays between action and reward).
Used for: Go (AlphaGo Zero), robot motion, video games.
Don't just read data, but *interact* with it!

Unsupervised Learning

(Maybe) how you learned to see.
What we did earlier with clustering and our recommender, and Deepfake.
Find patterns in data, compress data into model, find reducible representation of data.
Used for: Learning from unlabeled data.

All of these have been done with and without deep learning. DL has moved to the forefront of all of these.

“Theoretician’s Nightmare”

That is paraphrasing Yann LeCun, the godfather of Deep Learning.

If it feels like this is an oddly empirical branch of computer science, you are spot on.

Many of these techniques were developed through experimentation, and many of them are not amenable to classical analysis. A theoretician would suggest that non-convex loss functions are at the heart of the matter, and that situation isn’t getting better as many of the latest techniques have made this much worse.

You may also have noticed that many of the techniques we have used today have very recent provenance. This is true throughout the field. Rarely is the undergraduate researcher so reliant upon results groundbreaking papers of a few years ago.

My own humble observation: Deep Learning looks a lot like late 19th century chemistry. There is a weak theoretical basis, but significant experimental breakthroughs of great utility. The lesson from that era was "expect a lot more perspiration than inspiration."

You Now Have A Toolbox

The reason that we have attempted this ridiculously ambitious workshop is that the field has reached a level of maturity where the tools can encapsulate much of the complexity in black boxes.

One should not be ashamed to use a well-designed black box. Indeed it would be foolish for you to write your own FFT or eigensolver math routines. Besides wasting time, you won’t reach the efficiency of a professionally tuned tool.

On the other hand, most programmers using those tools have been exposed to the basics of the theory, and could dig out their old textbook explanation of how to cook up an FFT. This provides some baseline level of judgement in using tools provided by others.

You are treading on newer ground. However this means there are still major discoveries to be made using these tools in fresh applications.

One particularly exciting aspect of this whole situation is that exploring hyperparameters has been very fruitful. The toolbox allows you to do just that.

Other Toolboxes

You have a plethora of alternatives available as well. You are now in a position to appreciate some comparisons.

Package	Applications	Language	Strengths
TensorFlow	Neural Nets	Python, C++	Very popular.
Caffe	Neural Nets	Python, C++	Caffe2 rolled into PyTorch.
Spark MLLIB	Classification, Regression, Clustering, etc.	Python, Scala, Java, R	Very scalable. Widely used in serious applications. Lots of plugins to DL frameworks: TensorFrames, TF on Spark, CaffeOnSpark, Keras Elephas.
Scikit-Learn	Classification, Regression, Clustering	Python	Integrates well with TF to create powerful workflows.
cuDNN	Neural Nets	C++, GPU-based	Used in many other frameworks: TF, Caffe, etc.
Theano	Neural Nets	Python	Lower level numerical routines. NumPy-esque. Kinda obsolete.
PyTorch (Torch)	Neural Nets	Python (Lua)	Popular. Was dynamic graphs, eager execution (now in TF).
Keras	Neural Nets	Python (on top of TF, Theano)	Now completely absorbed into TF.
Digits	Neural Nets	“Caffe”, GPU-based	Used with other frameworks (only Caffe at moment).

TensorFlow 1 Version

```
from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf

mnist = input_data.read_data_sets(".", one_hot=True)

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

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

W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
b_conv1 = tf.Variable(tf.constant(0.1,shape=[32]))
h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1,strides=[1, 1, 1, 1], padding='SAME') + b_conv1)
h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))
b_conv2 = tf.Variable(tf.constant(0.1,shape=[64]))
h_conv2 = tf.nn.relu(tf.nn.conv2d(h_pool1, W_conv2,strides=[1, 1, 1, 1], padding='SAME') + b_conv2)
h_pool2 = tf.nn.max_pool(h_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

W_fc1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1024], stddev=0.1))
b_fc1 = tf.Variable(tf.constant(0.1,shape=[1024]))
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

W_fc2 = tf.Variable(tf.truncated_normal([1024, 10], stddev=0.1))
b_fc2 = tf.Variable(tf.constant(0.1,shape=[10]))
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(labels=y_, logits=y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

sess = tf.InteractiveSession()

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

print("test accuracy %g"%accuracy.eval(feed_dict={ x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
test accuracy 0.9915
```

32 kernel, 5x5 convolutional layer with 2x2 pooling.

```
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)

                                                                              n'.format(
```

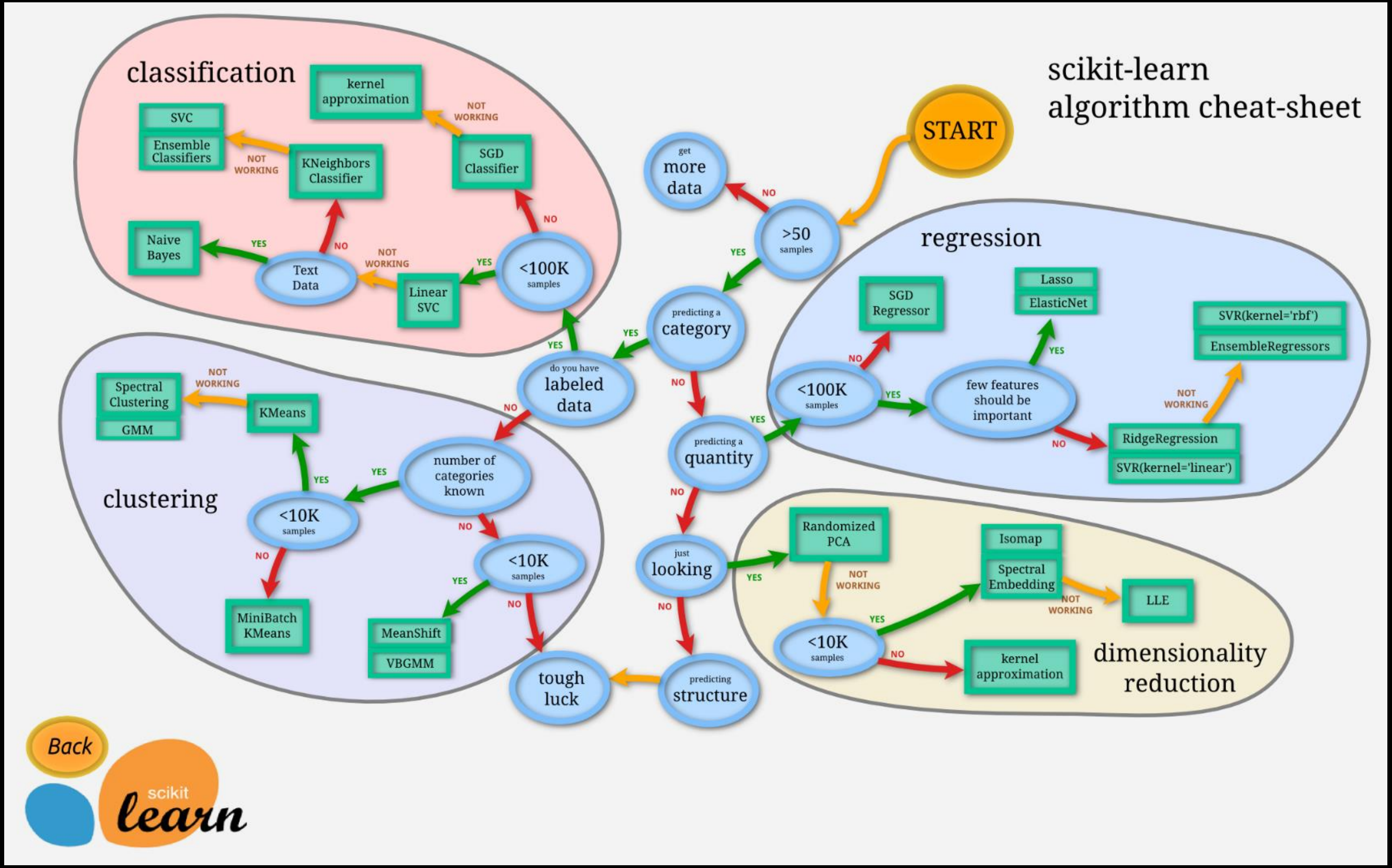
PyTorch CNN MNIST

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

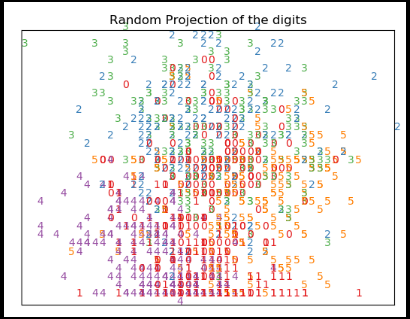
```
                                                                              ose([

    model = Net().to(device)
    optimizer = optim.Adadelta(model.parameters(), lr=args.lr)

    scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
    for epoch in range(1, args.epochs + 1):
        train(args, model, device, train_loader, optimizer, epoch)
        test(args, model, device, test_loader)
        scheduler.step()

    if args.save_model:
        torch.save(model.state_dict(), "mnist_cnn.pt")


if __name__ == '__main__':
    main()
```

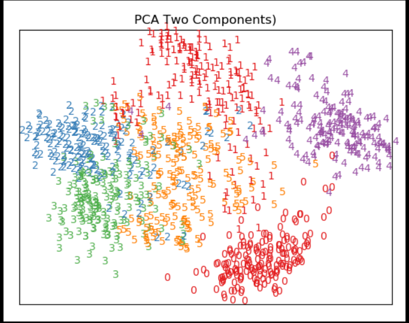
Scikit-learn

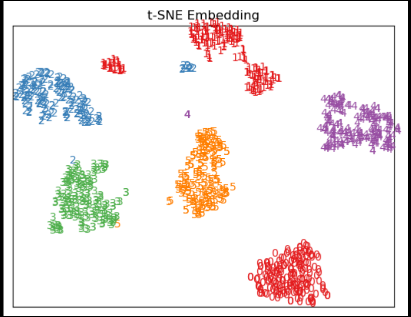


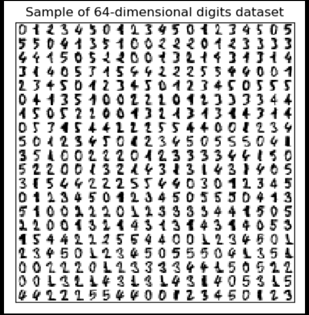
Return To DR With Scikit-learn

```
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()
```









Exercises

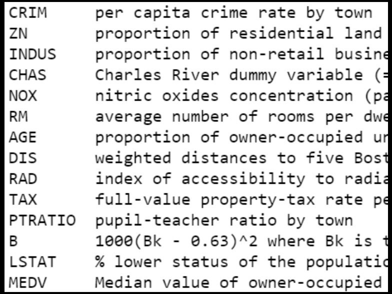
We are going to leave you with a few substantial problems that you are now equipped to tackle. Feel free to use your extended workshop access to work on these, and remember that additional time is an easy Startup Allocation away. Of course everything we have done is standard and you can work on these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

You may have wondered what else was to be found at tf.keras.datasets. 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.

More tf.keras.datasets Fun



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 un
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(Bk - 0.63)^2 where Bk is t
LSTAT	% lower status of the populatio
MEDV	Median value of owner-occupied

Boston Housing — Predict housing prices base upon crime, zoning, pollution, etc.

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...

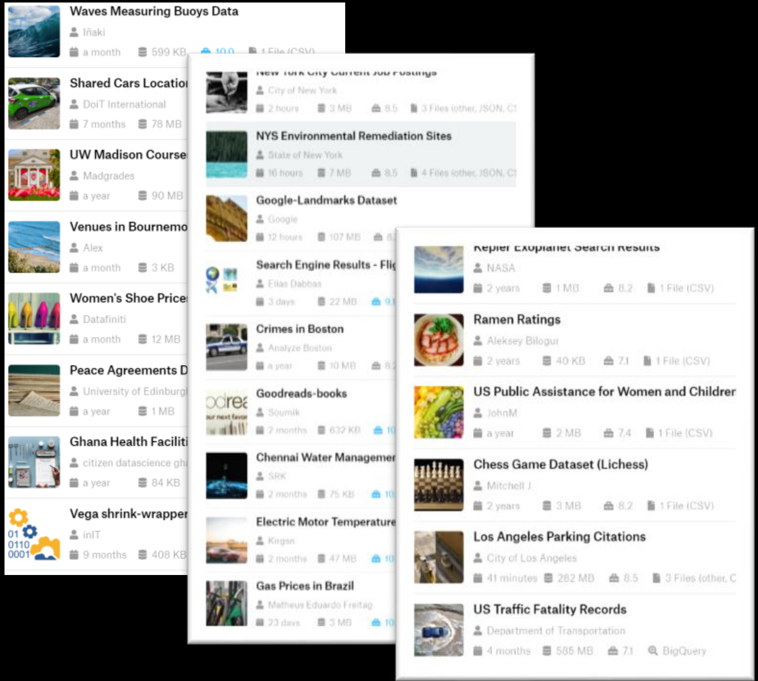
Reuters — 46 topics in newswire form.

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:



Waves Measuring Buoys Data
Iñaki
a month · 599 KB · 10.0 · 1 File (CSV)

Shared Cars Locatio
DoiT International
7 months · 78 MB

UW Madison Course
Madgrades
a year · 90 MB

Venues in Bournemo
Alex
a month · 3 KB

Women's Shoe Price
Datafiniti
a month · 12 MB

Peace Agreements D
University of Edinburg
a year · 1 MB

Ghana Health Faciliti
citizen datascience gh
a year · 84 KB

Vega shrink-wrapper
inIT
9 months · 408 KB

New York City Current Job Postings
City of New York
2 hours · 3 MB · 8.5 · 3 Files (other, JSON, C

NYS Environmental Remediation Sites
State of New York
16 hours · 7 MB · 8.5 · 4 Files (other, JSON, C

Google-Landmarks Dataset
Google
12 hours · 107 MB · 8.

Search Engine Results - Fli
Elias Dabbas
3 days · 22 MB · 9.

Crimes in Boston
Analyze Boston
a year · 10 MB · 8.

Goodreads-books
Soumik
2 months · 632 KB · 10

Chennai Water Managemen
SRK
2 months · 75 KB · 10

Electric Motor Temperature
Kirgsn
2 months · 47 MB · 10

Gas Prices in Brazil
Matheus Eduardo Freitag
23 days · 3 MB · 10

Kepler Exoplanet Search Results
NASA
2 years · 1 MB · 8.2 · 1 File (CSV)

Ramen Ratings
Aleksey Bilogur
2 years · 40 KB · 7.1 · 1 File (CSV)

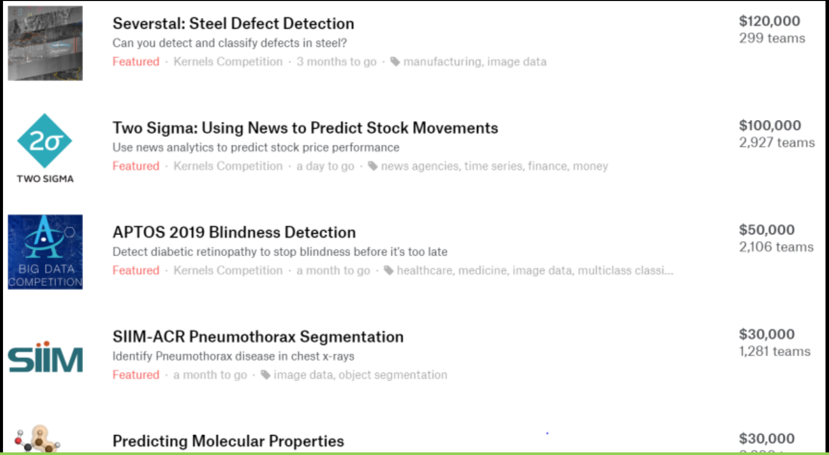
US Public Assistance for Women and Childrer
JohnM
a year · 2 MB · 7.4 · 1 File (CSV)

Chess Game Dataset (Lichess)
Mitchell J
2 years · 3 MB · 8.2 · 1 File (CSV)

Los Angeles Parking Citations
City of Los Angeles
41 minutes · 282 MB · 8.5 · 3 Files (other, C

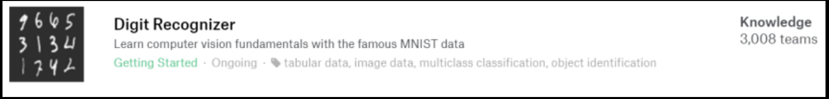
US Traffic Fatality Records
Department of Transportation
4 months · 585 MB · 7.1 · BigQuery

and competitions:



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

Including this one:



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

Demos

Ray-traced videogames soon? Recurrent CNN.

http://research.nvidia.com/sites/default/files/publications/dnn_denoise_author.pdf

Demos & Discussion

A wise man once (not that long ago) told me "John, I don't need a neural net to rediscover conservation of energy."

Model-Free Prediction of Large Spatiotemporally Chaotic Systems from Data: A Reservoir Computing Approach

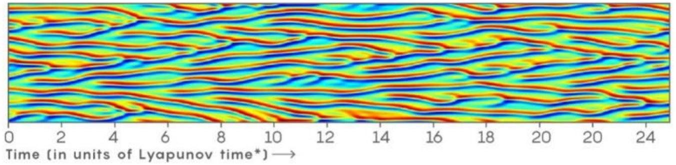
Jaideep Pathak, Brian Hunt, Michelle Girvan, Zhixin Lu, and Edward Ott
Phys. Rev. Lett. 120, 024102 – Published 12 January 2018

Training Computers to Tame Chaos

A machine-learning algorithm has been shown to accurately predict a chaotic system far further into the future than previously possible.

A Chaos Model

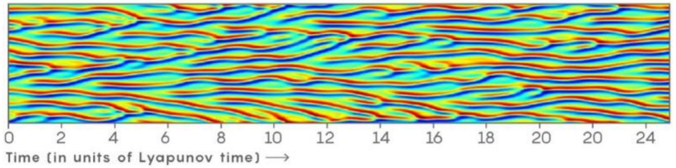
Researchers started with the evolving solution to the Kuramoto-Sivashinsky equation, which models propagating flames:



* Lyapunov time = Length of time before a small difference in the system's initial state begins to diverge exponentially. It typically sets the horizon of predictability, which varies from system to system.

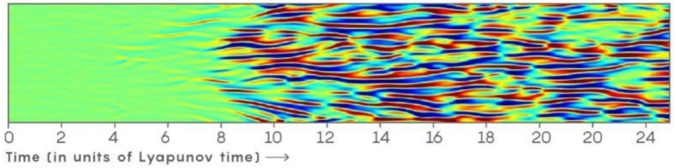
B Machine Learning

After training itself on data from the past evolution of the Kuramoto-Sivashinsky system, the "reservoir computing" algorithm predicts its future evolution:



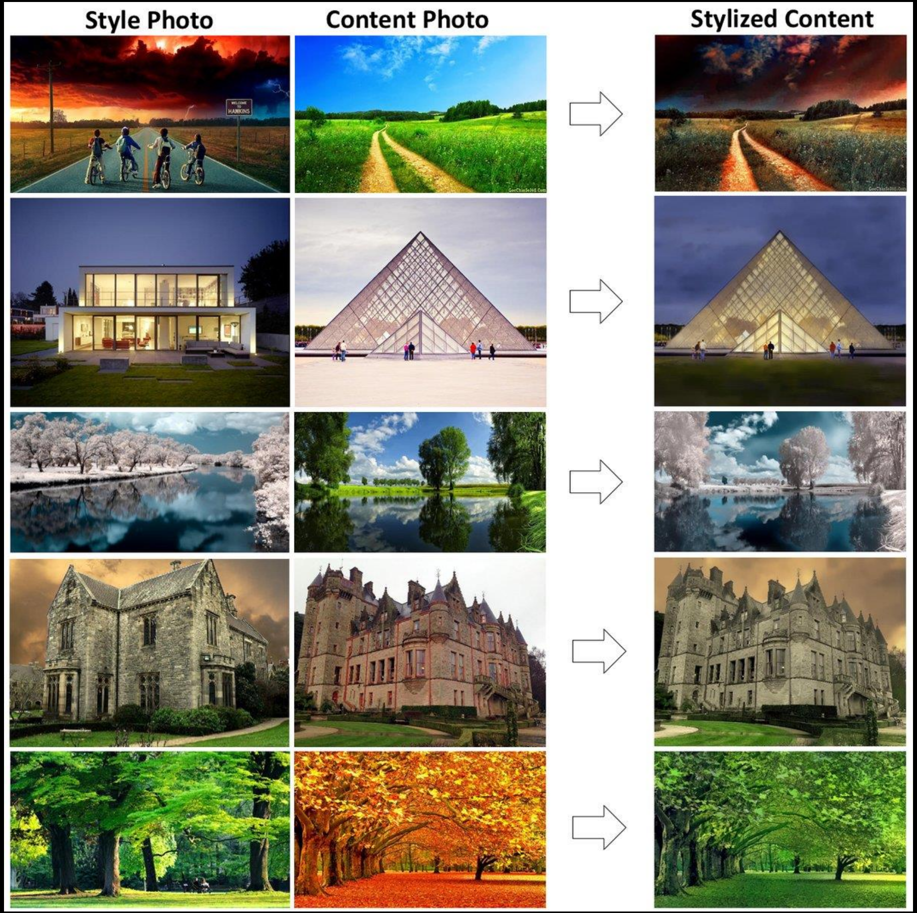
A–B Do They Match?

Subtracting B from A shows that the algorithm accurately predicts the model out to an impressive 8 Lyapunov times, before chaos ultimately prevails:



Demos

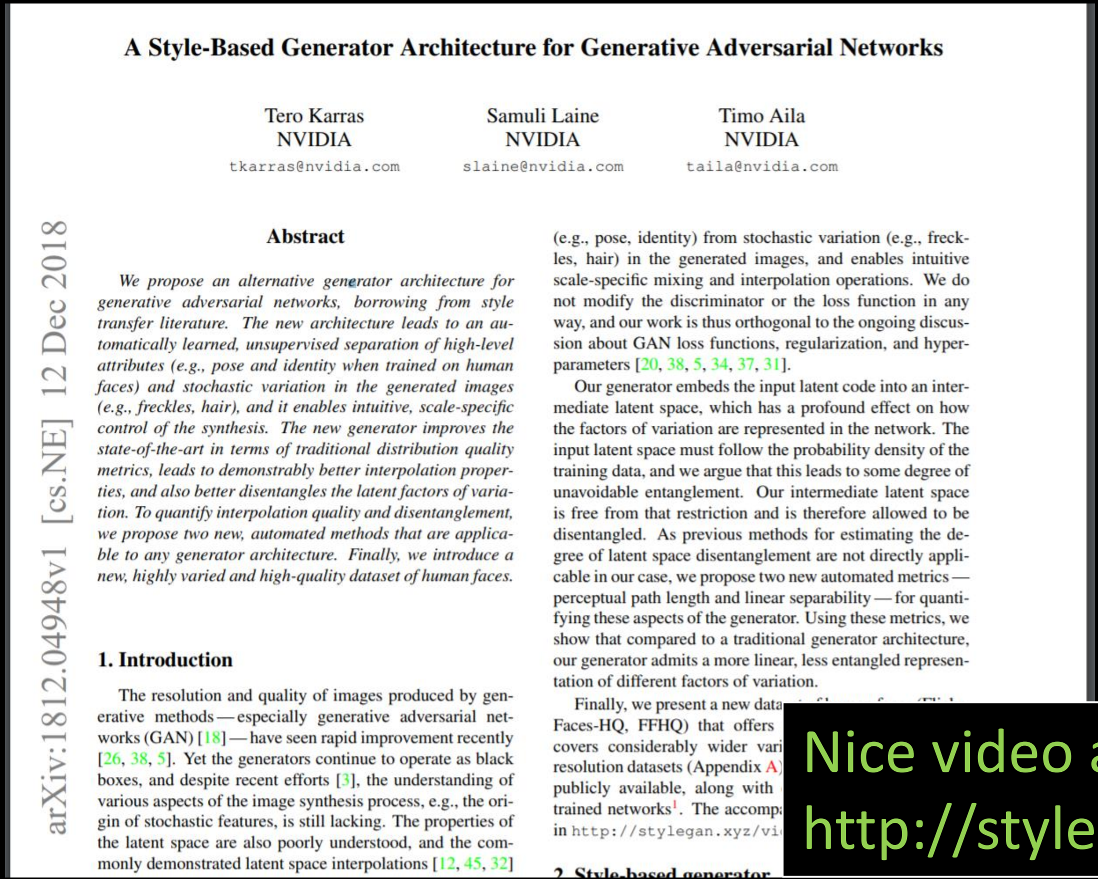
Style vs. Content: A little more subtle



Grab it at https://github.com/NVIDIA/FastPhotoStyle

Tomorrow

If Only...



arXiv:1812.04948v1 [cs.NE] 12 Dec 2018

A Style-Based Generator Architecture for Generative Adversarial Networks

Tero Karras
NVIDIA
tkarras@nvidia.com

Samuli Laine
NVIDIA
slaine@nvidia.com

Timo Aila
NVIDIA
taila@nvidia.com

Abstract

We propose an alternative generator architecture for generative adversarial networks, borrowing from style transfer literature. The new architecture leads to an automatically learned, unsupervised separation of high-level attributes (e.g., pose and identity when trained on human faces) and stochastic variation in the generated images (e.g., freckles, hair), and it enables intuitive, scale-specific control of the synthesis. The new generator improves the state-of-the-art in terms of traditional distribution quality metrics, leads to demonstrably better interpolation properties, and also better disentangles the latent factors of variation. To quantify interpolation quality and disentanglement, we propose two new, automated methods that are applicable to any generator architecture. Finally, we introduce a new, highly varied and high-quality dataset of human faces.

1. Introduction

The resolution and quality of images produced by generative methods — especially generative adversarial networks (GAN) [18] — have seen rapid improvement recently [26, 38, 5]. Yet the generators continue to operate as black boxes, and despite recent efforts [3], the understanding of various aspects of the image synthesis process, e.g., the origin of stochastic features, is still lacking. The properties of the latent space are also poorly understood, and the commonly demonstrated latent space interpolations [12, 45, 32] (e.g., pose, identity) from stochastic variation (e.g., freckles, hair) in the generated images, and enables intuitive scale-specific mixing and interpolation operations. We do not modify the discriminator or the loss function in any way, and our work is thus orthogonal to the ongoing discussion about GAN loss functions, regularization, and hyperparameters [20, 38, 5, 34, 37, 31].

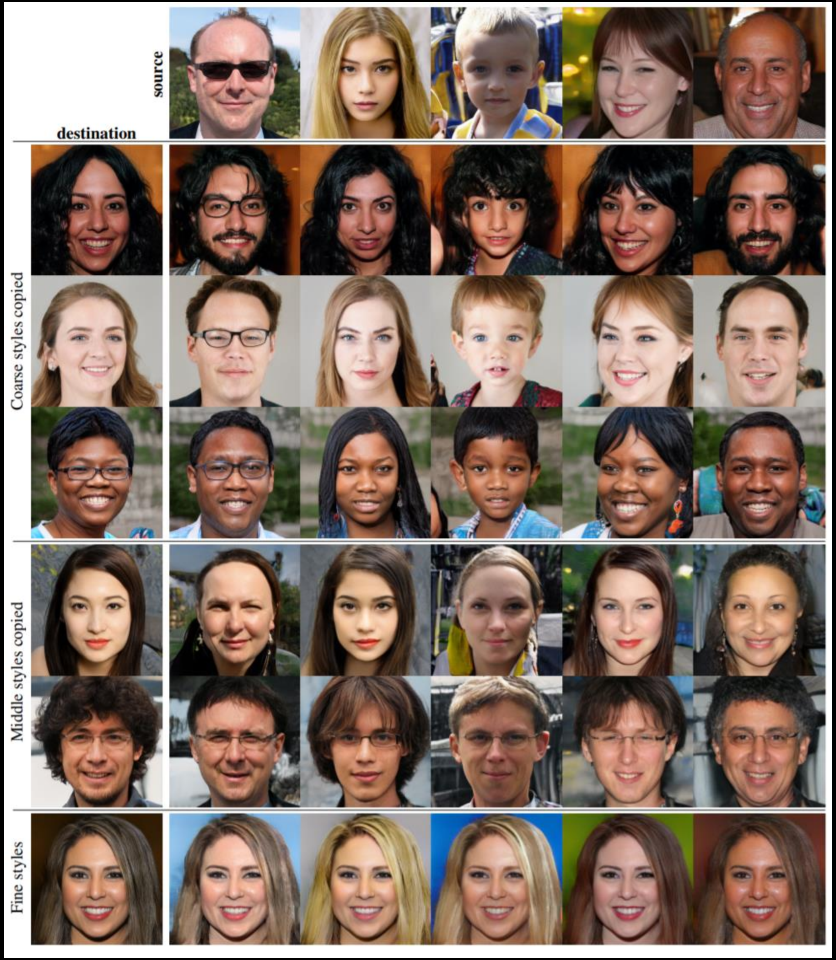
Our generator embeds the input latent code into an intermediate latent space, which has a profound effect on how the factors of variation are represented in the network. The input latent space must follow the probability density of the training data, and we argue that this leads to some degree of unavoidable entanglement. Our intermediate latent space is free from that restriction and is therefore allowed to be disentangled. As previous methods for estimating the degree of latent space disentanglement are not directly applicable in our case, we propose two new automated metrics — perceptual path length and linear separability — for quantifying these aspects of the generator. Using these metrics, we show that compared to a traditional generator architecture, our generator admits a more linear, less entangled representation of different factors of variation.

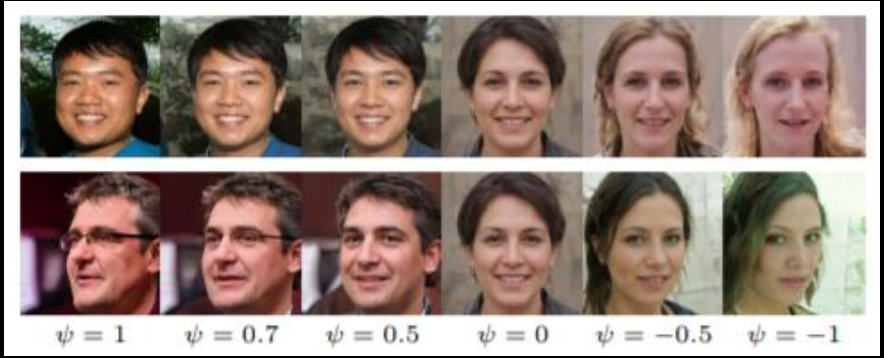
Finally, we present a new data
Faces-HQ, FFHQ) that offers
covers considerably wider vari
resolution datasets (Appendix A)
publicly available, along with
trained networks[1]. The accompa
in http://stylegan.xyz/vi

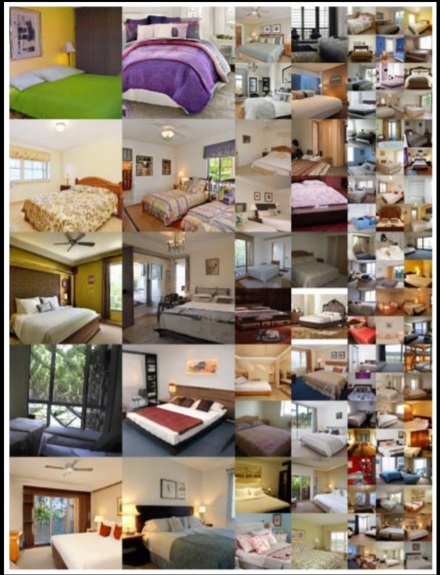
2. Style-based generator

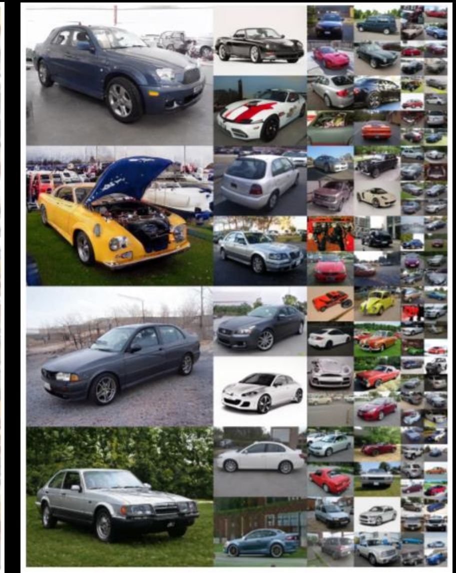
Nice video at http://stylegan.xyz/video

What is reality?

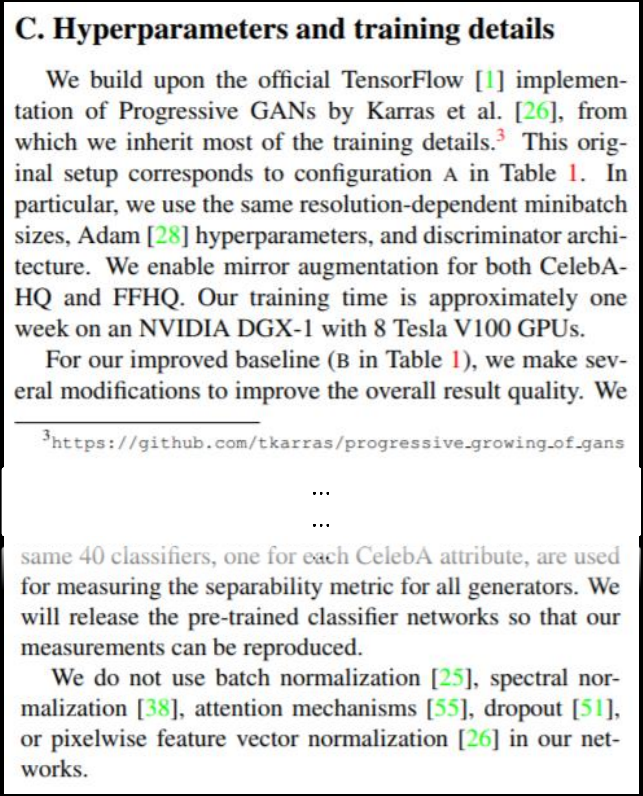








Where did they get their hyperparameters?



C. Hyperparameters and training details

We build upon the official TensorFlow [1] implementation of Progressive GANs by Karras et al. [26], from which we inherit most of the training details.[3] This original setup corresponds to configuration A in Table 1. In particular, we use the same resolution-dependent minibatch sizes, Adam [28] hyperparameters, and discriminator architecture. We enable mirror augmentation for both CelebA-HQ and FFHQ. Our training time is approximately one week on an NVIDIA DGX-1 with 8 Tesla V100 GPUs.

For our improved baseline (B in Table 1), we make several modifications to improve the overall result quality. We

[3]https://github.com/tkarras/progressive_growing_of_gans

…

…

same 40 classifiers, one for each CelebA attribute, are used for measuring the separability metric for all generators. We will release the pre-trained classifier networks so that our measurements can be reproduced.

We do not use batch normalization [25], spectral normalization [38], attention mechanisms [55], dropout [51], or pixelwise feature vector normalization [26] in our networks.