Deep Learning
In An Afternoon

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Copyright 2022
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 fundamentals of deep learning.

• Inspired by biological models.
• An evolution of classic ML techniques (the perceptron).

They are both fair and useful. We’ll give each a thin slice of our attention before we move on to the actual implementation. You can decide which perspective works for you.
Modeled After The Brain
As a Highly Dimensional Non-linear Classifier

Perceptron

Network

No Hidden Layer
Linear

Hidden Layers
Nonlinear

Courtesy: Chris Olah
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)
Neurons apply activation functions at these summed inputs. Activation functions are typically non-linear. There are countless possibilities. In reality, you are likely to encounter these two:

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

- The **Rectified Linear** activation function is zero when the input is negative and is equal to the input when the input is positive. Rectified Linear activation functions are currently the most popular activation function as they are more efficient than the sigmoid or hyperbolic tangent.
  
  - Sparse activation: In a randomly initialized network, only 50% of hidden units are active.
  
  - Better gradient propagation: Fewer vanishing gradient problems compared to sigmoidal activation functions that saturate in both directions.
  
  - Efficient computation: Only comparison, maybe addition and multiplication for variants.
  
  - There are **Leaky** and **Noisy** variants.
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
H1 Weights = (1.0, -2.0, 2.0)  
H2 Weights = (2.0, 1.0, -4.0)  
H3 Weights = (1.0, -1.0, 0.0)  
O1 Weights = (-3.0, 1.0, -3.0) 
O2 Weights = (0.0, 1.0, 2.0) 

O1 = \text{Sigmoid}(0.13 \times -3.0 + 0.96 \times 1.0 + 0.40 \times -3.0) = \text{Sigmoid}(-0.63) = 0.35 
O2 = \text{Sigmoid}(0.13 \times 0.0 + 0.96 \times 1.0 + 0.40 \times 2.0) = \text{Sigmoid}(1.76) = 0.85
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)

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 \( \text{bias}=0.1 \):

\[
\text{Hidden Layer Weights} \times \text{Inputs} + \text{Bias} = \text{Sig(Hidden Layer Outputs)}
\]
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
A very underappreciated fact about networks is that the width of any layer determines how many dimensions it can work in. This is valuable even for lower dimension problems. How about trying to classify (separate) this dataset:

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

It takes at least 3 units wide to pull this off, regardless of depth. 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 \( \frac{\partial E}{\partial w_3} \).

![Diagram of a neural network with weights and outputs.]

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

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

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

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

From the fantastic *Deep Learning*, Goodfellow, Bengio and Courville.
If you have 30 minutes, and remember freshman calculus, you can understand the complete details of the algorithm. I heartily recommend one of these.

An elegant perspective on this can be found from Chris Olah at http://colah.github.io/posts/2015-08-Backprop.

With basic calculus you can readily work through the details. You can find an excellent explanation from the renowned 3Blue1Brown at https://www.youtube.com/watch?v=Ilg3gGewQ5U.

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

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

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

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

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

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

Make sure you are on a GPU node:

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

Load the TensorFlow 2 Container:

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

And start TensorFlow:

```
Singularity> python
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license"
>>> import tensorflow
...some congratulatory noise...
```
The API is well documented. That is terribly unusual. Take advantage and keep a browser open as you develop.
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.
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)

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

train_images /= 255
test_images /= 255

matplotlib bonus insight

import matplotlib.pyplot as plt

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

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

train_images /= 255
test_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'])

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:

<table>
<thead>
<tr>
<th>Digit</th>
<th>Output</th>
<th>Exponential</th>
<th>Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.8</td>
<td>121</td>
<td>.87</td>
</tr>
<tr>
<td>1</td>
<td>-2.6</td>
<td>0.07</td>
<td>.00</td>
</tr>
<tr>
<td>2</td>
<td>2.9</td>
<td>18</td>
<td>.13</td>
</tr>
</tbody>
</table>

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

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

You can think that it "undoes" the Softmax, if you want.
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))
Train on 60000 samples, validate on 10000 samples

Epoch 1/40
60000/60000 [==============================] - 1s 16us/sample
- loss: 0.3971
- accuracy: 0.8889
- val_loss: 0.2003
- val_accuracy: 0.9386

Epoch 2/40
60000/60000 [==============================] - 1s 10us/sample
- loss: 0.1696
- accuracy: 0.9503
- val_loss: 0.1430
- val_accuracy: 0.9562

Epoch 3/40
60000/60000 [==============================] - 1s 10us/sample
- loss: 0.1224
- accuracy: 0.9631
- val_loss: 0.1218
- val_accuracy: 0.9614

Epoch 4/40
60000/60000 [==============================] - 1s 9us/sample
- loss: 0.0972
- accuracy: 0.9715
- val_loss: 0.1109
- val_accuracy: 0.9657

Epoch 5/40
60000/60000 [==============================] - 1s 11us/sample
- loss: 0.0813
- accuracy: 0.9758
- val_loss: 0.0986
- val_accuracy: 0.9700

Epoch 6/40
60000/60000 [==============================] - 1s 11us/sample
- loss: 0.0683
- accuracy: 0.9796
- val_loss: 0.1035
- val_accuracy: 0.9683

Epoch 38/40
60000/60000 [==============================] - 1s 12us/sample
- loss: 0.0064
- accuracy: 0.9978
- val_loss: 0.1632
- val_accuracy: 0.9699

Epoch 39/40
60000/60000 [==============================] - 1s 10us/sample
- loss: 0.0027
- accuracy: 0.9993
- val_loss: 0.1384
- val_accuracy: 0.9750

Epoch 40/40
60000/60000 [==============================] - 1s 9us/sample
- loss: 6.8242e-04
- accuracy: 0.9999
- val_loss: 0.1390
- val_accuracy: 0.9755

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

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

Latter on we will see that regularization techniques (which are only turned on for training) also add to this effect.
Let's Go Wider

```python
import tensorflow as tf

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

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

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

test_images /= 255
train_images /= 255

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

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

model.fit(train_images, train_labels, batch_size=128, epochs=30, verbose=1, validation_data=(test_images, test_labels))
```
Epoch 30/30
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0083 - accuracy: 0.9977 - val_loss: 0.1027 - val_accuracy: 0.9821

Wider Results

```
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
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')

train_images /= 255

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

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

model.fit(train_images, train_labels, batch_size=128, epochs=30, verbose=1, validation_data=(test_images, test_labels))
Wide And Deep Results

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

**Recap**

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

**Deep and Wide**

```
model.summary()
```

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense_24 (Dense)</td>
<td>(None, 512)</td>
<td>401920</td>
</tr>
<tr>
<td>dense_25 (Dense)</td>
<td>(None, 512)</td>
<td>262656</td>
</tr>
<tr>
<td>dense_26 (Dense)</td>
<td>(None, 512)</td>
<td>262656</td>
</tr>
<tr>
<td>dense_27 (Dense)</td>
<td>(None, 10)</td>
<td>5130</td>
</tr>
</tbody>
</table>

Total params: 932,362

Trainable params: 932,362
Non-trainable params: 0

**Model accuracy**

![Accuracy Graph](image)
Brute Force Does Not Work

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

- Overfitting
- Vanishing gradients

We will have to be smarter than "bigger is better" about choosing our hyperparameters. One very smart thing to do is to choose a more appropriate architecture.
AlexNet won the 2012 ImageNet LSVRC and changed the DL world.
Convolution

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

+          =

Edge Detector
Simplest Convolution Net

Courtesy: Chris Olah
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
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.
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))
Epoch 10/10
60000/60000 [====================================] - 12s 198us/sample - loss: 0.0051 - accuracy: 0.9989 - val_loss: 0.0424 - val_accuracy: 0.9874

Early CNN Results

Score Thus Far

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC (64,64)</td>
<td>97.5</td>
</tr>
<tr>
<td>FC (512,512)</td>
<td>98.2</td>
</tr>
<tr>
<td>FC (521,512,512)</td>
<td>98.0</td>
</tr>
<tr>
<td>CNN (1 layer)</td>
<td>98.7</td>
</tr>
</tbody>
</table>

Primitive CNN

```python
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
==================================================================
Total params: 542,230
Trainable params: 542,230
Non-trainable params: 0
```
Scaling Up The CNN

```python
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
__________________________________________________
dense_43 (Dense)     (None, 10)               1290
==================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable 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
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.
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))

CNN With Dropout

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

Epoch 15/15
60000/60000 [==============================] - 40s 667us/sample - loss: 0.0187 - accuracy: 0.9935 - val_loss: 0.0301 - val_accuracy: 0.9919

### Dropout CNN

```
model.summary()
```

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv2d_12 (Conv2D)</td>
<td>(None, 26, 26, 32)</td>
<td>320</td>
</tr>
<tr>
<td>conv2d_13 (Conv2D)</td>
<td>(None, 24, 24, 64)</td>
<td>18496</td>
</tr>
<tr>
<td>max_pooling2d_7</td>
<td>(None, 12, 12, 64)</td>
<td>0</td>
</tr>
<tr>
<td>dropout_4 (Dropout)</td>
<td>(None, 12, 12, 64)</td>
<td>0</td>
</tr>
<tr>
<td>flatten_7 (Flatten)</td>
<td>(None, 9216)</td>
<td>0</td>
</tr>
<tr>
<td>dense_50 (Dense)</td>
<td>(None, 128)</td>
<td>1179776</td>
</tr>
<tr>
<td>dropout_5 (Dropout)</td>
<td>(None, 128)</td>
<td>0</td>
</tr>
<tr>
<td>dense_51 (Dense)</td>
<td>(None, 10)</td>
<td>1290</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Total params:</strong> 1,199,882</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Trainable params:</strong> 1,199,882</td>
</tr>
</tbody>
</table>

### 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
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).
- Before or after non-linear activation function? Oddly, also open to debate. But, it may be more appropriate after the activation function if for s-shaped functions like the hyperbolic tangent and logistic function, and before the activation function for activations that result in non-Gaussian distributions like ReLU.

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

```python
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.
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))
Epoch 15/15
60000/60000 [==============================================] - 50s 834us/sample - loss: 0.0027 - accuracy: 0.9993 - val_loss: 0.0385 - val_accuracy: 0.9891

Not So Helpful

Batch Normalization CNN

model.summary()

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv2d_2 (Conv2D)</td>
<td>(None, 26, 26, 32)</td>
<td>320</td>
</tr>
<tr>
<td>batch_normalization</td>
<td>(None, 26, 26, 32)</td>
<td>128</td>
</tr>
<tr>
<td>conv2d_3 (Conv2D)</td>
<td>(None, 24, 24, 64)</td>
<td>18496</td>
</tr>
<tr>
<td>max_pooling2d_1</td>
<td>(None, 12, 12, 64)</td>
<td>0</td>
</tr>
<tr>
<td>batch_normalization_1</td>
<td>(None, 12, 12, 64)</td>
<td>256</td>
</tr>
<tr>
<td>dense_2 (Dense)</td>
<td>(None, 128)</td>
<td>1290</td>
</tr>
<tr>
<td>batch_normalization_2</td>
<td>(Batch (None, 128)</td>
<td>512</td>
</tr>
<tr>
<td>dense_3 (Dense)</td>
<td>(None, 10)</td>
<td>1290</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,200,778</td>
</tr>
<tr>
<td></td>
<td></td>
<td>params: 1,200,330</td>
</tr>
<tr>
<td></td>
<td></td>
<td>non-trainable params: 448</td>
</tr>
</tbody>
</table>

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
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.
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.

```python
...  
...  
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:

```bash
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.

Our First Model
64 Wide FC

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.
TensorBoard Add Ons

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

Hyperparameter Search

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

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

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

```python
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.

```python
# 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'])
```

You can find a full example of using Horovod with a Keras MNIST code at:

https://horovod.readthedocs.io/en/latest/keras.html

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

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

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

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

What if we wanted to teach it:

![Images of handwritten digits](image)

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.

How many samples do we need?

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

Note that *tf.image* makes many of these processes very convenient.
```python
def test(args, model, device, test_loader):
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)
    print('Test set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)
'.format(test_loss, correct, len(test_loader.dataset), 100. * correct / len(test_loader.dataset)))

def main():
    # Bunch of parsed training inputs...
    ...
    torch.manual_seed(args.seed)
    device = torch.device("cuda" if use_cuda else "cpu")
    kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
    train_loader = torch.utils.data.DataLoader(datasets.MNIST('../data', train=True, download=True,
                  transform=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])),
                  batch_size=args.batch_size, shuffle=True, **kwargs)
    test_loader = torch.utils.data.DataLoader(datasets.MNIST('../data', train=False, transform=
                  transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])),
                  batch_size=args.test_batch_size, shuffle=True, **kwargs)
    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()
```

---

**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](https://github.com/pytorch/examples/blob/master/mnist/main.py)
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.
**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.

**Reuters**

46 topics in newswire form.

---

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...

---

Its December acquisition of space co it expects earnings per share in 1987 of 1.15 to 1.30 dlr per share up from 70 cts in 1986 the company said pretax net should rise to nine to 10 mln dlr from six mln dlr in 1986 and rental operation revenues to 19 to 22 mln dlr from 12.5 mln dlr it said cash flow per share this year should be 2.50 to three dlr 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:

- **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 classification
  - $50,000
  - 2,106 teams

- **SIIM-ACR Pneumothorax Segmentation**
  - Identify Pneumothorax disease in chest x-rays
  - Featured: a month to go
  - 1,281 teams
  - $30,000

- **Predicting Molecular Properties**
  - Predict the properties of molecules
  - $30,000
  - 1,281 teams

Including this one:

- **Digit Recognizer**
  - Learn computer vision fundamentals with the famous MNIST data
  - Knowledge
  - 3,008 teams