Deep Learning
In An Afternoon

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center
Unprecedented Disruption

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

10 years ago
“Neural nets will enable real time ray tracing.”  Science Fiction.
“Neural nets will do protein folding.”  Word salad.

5 years ago
“Neural nets will do CFD.”  Well, maybe someday, but not soon.

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

Tomorrow
Skynet will kill us all. Or at least steal our jobs.
Why Now?

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

Widespread parallel computing (GPUs)

Big data training sets
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
No Hidden Layer
Linear

Network
Hidden Layers
Nonlinear

Courtesy: Chris Olah
Basic NN Architecture

Input Layer → Hidden Layer → Output Layer

- Neuron
- Synapse
In Practice

How many inputs?
For an image it could be one (or 3) per pixel.

How deep?
100+ layers have become common.

How many outputs?
Might be an entire image.
Or could be discreet set of classification possibilities.
Inference

The "forward" or thinking step
Inference
Input and Output Layers

0.5
0.9
-0.3

H1
H2
H3

1.0
0.0

Cat
Dog

Pixel
Inference
Weights or Parameters

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

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

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

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

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

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

  - Sparse activation: In a randomly initialized network, only 50% of hidden units are active.
  - Better gradient propagation: Fewer vanishing gradient problems compared to sigmoidal activation functions that saturate in both directions.
  - Efficient computation: Only comparison, maybe addition and multiplication for variants.
  - There are **Leaky** and **Noisy** variants.
Inference

Multiply, Add, do something non-linear.

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

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

H1 = 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
Then do it again.

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

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

O1 = Sigmoid(.13 * -3.0 + .96 * 1.0 + .40 * -3.0) = Sigmoid(-.63) = .35
O2 = 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)

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

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

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

\[
\text{Hidden Layer Weights} \times \text{Inputs} + \text{Bias} = \text{Hidden Layer Outputs}
\]

<table>
<thead>
<tr>
<th>Hidden Layer Weights</th>
<th>Inputs</th>
<th>Bias</th>
<th>Hidden Layer Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0  -2.0  2.0</td>
<td>0.5</td>
<td>0.1</td>
<td>-1.8  3.2  -0.3</td>
</tr>
<tr>
<td>2.0  1.0  -4.0</td>
<td>0.9</td>
<td>0.1</td>
<td>.14   .96  0.4</td>
</tr>
<tr>
<td>1.0  -1.0  0.0</td>
<td>-0.3</td>
<td>0.1</td>
<td></td>
</tr>
</tbody>
</table>
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.
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?
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.
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 \( \frac{\partial E}{\partial w_3} \).

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

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

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:

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

Load the TensorFlow 2 Container:

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

And start TensorFlow:

```bash
Singularity> python
Python 3.8.10 (default, Mar 13 2023, 10:26:41)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 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:

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

or invoke them from within the python shell:

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

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

test_images /= 255
train_images /= 255

matplotlib bonus insight

import matplotlib.pyplot as plt

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

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense_6 (Dense)</td>
<td>(None, 64)</td>
<td>50240</td>
</tr>
<tr>
<td>dense_7 (Dense)</td>
<td>(None, 64)</td>
<td>4160</td>
</tr>
<tr>
<td>dense_8 (Dense)</td>
<td>(None, 10)</td>
<td>650</td>
</tr>
</tbody>
</table>

Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0

Defining Our Network

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.
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>
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'])
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

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

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

You can also think that it "undoes" the Softmax, if you want.
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))
Why would the test accuracy *ever* be better than the training (as momentarily happens here)?

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

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

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense_18 (Dense)</td>
<td>(None, 512)</td>
<td>401920</td>
</tr>
<tr>
<td>dense_19 (Dense)</td>
<td>(None, 512)</td>
<td>262656</td>
</tr>
<tr>
<td>dense_20 (Dense)</td>
<td>(None, 10)</td>
<td>5130</td>
</tr>
</tbody>
</table>

Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0

55,050 for 64 wide Model
```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(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

- Training loss: 0.0119, accuracy: 0.9967
- Validation loss: 0.1183, accuracy: 0.9800

Model accuracy:
- Train accuracy: approximately 1.00
- Test accuracy: approximately 0.98

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

Recap:
- FC 64,64: 97.5
- FC 512,512: 98.2
- FC 521,512,512: 98.0
You usually can't just brute force your way into success. Beyond the obvious time and memory costs, you are opening yourself up to

- Overfitting
- Vanishing gradients

We will have to be smarter than "bigger is better" about choosing our hyperparameters. One very smart thing to do is to choose a more appropriate architecture.
**Image Recognition Done Right: CNNs**

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

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>FULL CONNECT</td>
<td>4M</td>
</tr>
<tr>
<td>FULL 4096/ReLU</td>
<td>16M</td>
</tr>
<tr>
<td>FULL 4096/ReLU</td>
<td>37M</td>
</tr>
<tr>
<td>MAX POOLING</td>
<td>442K</td>
</tr>
<tr>
<td>CONV 3x3/ReLU</td>
<td>1.3M</td>
</tr>
<tr>
<td>CONV 3x3/ReLU</td>
<td>884K</td>
</tr>
<tr>
<td>MAX POOLING 2x2sub</td>
<td>4.42M</td>
</tr>
<tr>
<td>LOCAL CONTRAST NORM</td>
<td>3.07M</td>
</tr>
<tr>
<td>CONV 11x11/ReLU</td>
<td>3.5K</td>
</tr>
<tr>
<td>MAX POOL 2x2sub</td>
<td>35K</td>
</tr>
<tr>
<td>LOCAL CONTRAST NORM</td>
<td>7K</td>
</tr>
<tr>
<td>CONV 11x11/ReLU</td>
<td>4K</td>
</tr>
</tbody>
</table>

*Image Object Recognition [Krizhevsky, Sutskever, Hinton 2012]*
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

\[ \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \]

Edge Detector
Stacking Convolutions

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

Stride = 2
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 $W_O \times H_O \times D_O$

$$
W_O = \frac{(W_I - N + 2P)}{S+1}
$$
$$
H_O = \frac{(H_I - F + 2P)}{S+1}
$$
$$
D_O = K
$$

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.
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.
This is your brain on CNNs.
One of countless "illusions" I could provoke your own CNNs with.
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))

Let's Start Small
**Early CNN Results**

**Epoch 10/10**

60000/60000 [==============================] - 12s 198us/sample - loss: 0.0051 - accuracy: 0.9989 - val_loss: 0.0424 - val_accuracy: 0.9874

**Score Thus Far**

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

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

<table>
<thead>
<tr>
<th>Layer</th>
<th>Accuracy</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>
<tr>
<td>CNN (2 Layer)</td>
<td>99.0</td>
</tr>
</tbody>
</table>

Overfitting
Overfitting = Memorization

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

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

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

Score Thus Far

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</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>
<tr>
<td>CNN (2 Layer)</td>
<td>99.0</td>
</tr>
<tr>
<td>CNN with Dropout</td>
<td>99.2</td>
</tr>
</tbody>
</table>

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_50 (Dense)    (None, 128)             1179776
```

```
dropout_5 (Dropout) (None, 128)                   0
```

```
dense_51 (Dense)    (None, 10)                 1290
```

Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
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 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:

```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

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 is very similar to what we just did, but different enough to be interesting.

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

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

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

If you are running on Bridges login nodes, from your computer something like:
```
ssh -2 -Nf -L 6006:127.0.0.1:6006 br014.bridges2.psc.edu
```

If you are running on a Bridges compute nodes, you need to use the compute's IB address/hostname, for example:
```
ssh -2 -Nf -L 6006:r001.ib.bridges2.psc.edu:6006 br014.bridges2.psc.edu
```
TensorBoard Analysis

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

Our First Model
64 Wide FC

Our CNN
We can explore the architecture of the deep learning graphs we have constructed. And we can drill down.

Our First Model
64 Wide FC

Our CNN

Our CNN's FC Layer

Keras "Conceptual Model" View of CNN
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.

**Performance Profiling**
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

If one GPU is good, more must be better! This is largely true, and you will notice our GPU nodes are stuffed full with 4 or more GPUs each.

You might also notice that most of the machines in the "Top 10" have a lot of GPUs in them. They deliver most of the FLOPS for scientific codes, but are also an enviable Deep Learning resource.

You might have noticed that most of the interesting leading-edge research seems to involve a lot of GPUs these days.

And the very public battles in the Large Language Model space seem to be about who can get their hands on the largest GPU clusters.

How might you reach these levels of capability?

And what about those scaling limitations I mentioned earlier?

Actually a Crypto miner. We hate these guys for hoarding our GPUs!!!
One early technique to utilize multiple GPUs was to independently train an *ensemble* of GPUs on the same task, and then have them vote on the answer. This method does work, but then the end user needs to have an ensemble of their own GPUs. This is not ideal for an application that you wish to run on your phone, or in a self-driving car.

It would be better if we could use a lot of GPUs for the training step, but end up with one great set of parameters that will fit on a single GPU when we are done. Then our users don't need to own supercomputers.

One technique to achieve this is to use *Data Parallelism* so that each GPU trains on a separate batch of data, and at the end of that batch we average the collective wisdom of all of these GPUs to arrive at our new and improved parameters.

Now when we finish we have one super set of parameters that fits on a single GPU.

This gradient averaging requires an *all-reduce*, which can be quite expensive given the number of weights involved.
TensorFlow Scalability

This is very straightforward to implement in TensorFlow using the MirroredStrategy on a single node with multiple GPUs, or MultiWorkerMirroredStrategy across multiple nodes.

```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.

MNIST with 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))
```

```python
# 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'])
```

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

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

You can find a full example of using Horovod with a Keras MNIST code at:  
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**
What about all these LLMs you have been hearing about that use *trillions* of parameters? Now, we don't have enough memory to fit the whole model on one GPU.

Instead we spread the parts of the model around (mostly their parameters, but could also be different sections of a more complex model) by using *Model Parallelism*.

The most popular way to do this in TensorFlow is via the *Mesh TensorFlow* API.

And, we can mix the way we distribute these parameters, layers, pipelines and model branches in various hybrid methods as well.
For LLMs that have been designed to scale well (avoiding overfitting and vanishing gradients, for example), we find that performance is a predictable function of:

- The dataset size
- The number of parameters

And these curves show no signs of ending yet.

So, in these applications we do expect better performance through brute force scaling.
An emergent ability as an ability that is not present in small models but develops as the model is scaled. These might be unanticipated.

Some areas of science are familiar with this idea (physics and biology, for sure). In computer science this concept has largely been a novelty (Conway’s Game of Life is a notable example). In Deep Learning, it has become a very significant phenomena.

Once again, bear in mind that many of the principles we have mentioned (overfitting, vanishing gradients, etc.) mean that brute force scaling is not going to be a default route to better performance.

Instead, understanding of those principles will allow you the option to scale.

https://www.jasonwei.net/blog/emergence
Also a good associated paper.
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:

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.
Stupid Neural Nets

Why can't they learn like we do? Can't I just tell you a fact, or an algorithm, and you can just "get it" it without countless iteration?

One-Shot Learning

On the other hand, maybe I could teach adult you what a platypus is with one example. And, if you want to spot a particular mad bomber with your airport facial recognition system, you may only have one photo.

There is such a thing as "one-shot" (or N-shot) learning. But it is harder, requires more specialized techniques, and is straying into the area of unsupervised learning. We will come back to this, but it is no magic bullet for sparse data.
def test(args, model, device, test_loader):
    model.eval()
    test_loss = 0
correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)
    print('
Test set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)'.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()
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

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 dls per share up from 70 cts in 1986 the company said pretax net should rise to nine to 10 mln dls from six mln dls in 1986 and rental operation revenues to 19 to 22 mln dls from 12 5 mln dls it said cash flow per share this year should be 2 50 to three dls reuters...
Endless Exercises

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

and competitions:

- **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 - image data, object segmentation
  - $30,000
  - 1,281 teams

- **Predicting Molecular Properties**
  - Featured - knowledge based
  - $30,000
  - 1,043 teams

Including this one:

- **Digit Recognizer**
  - Learn computer vision fundamentals with the famous MNIST data
  - Knowledge
  - 3,003 teams