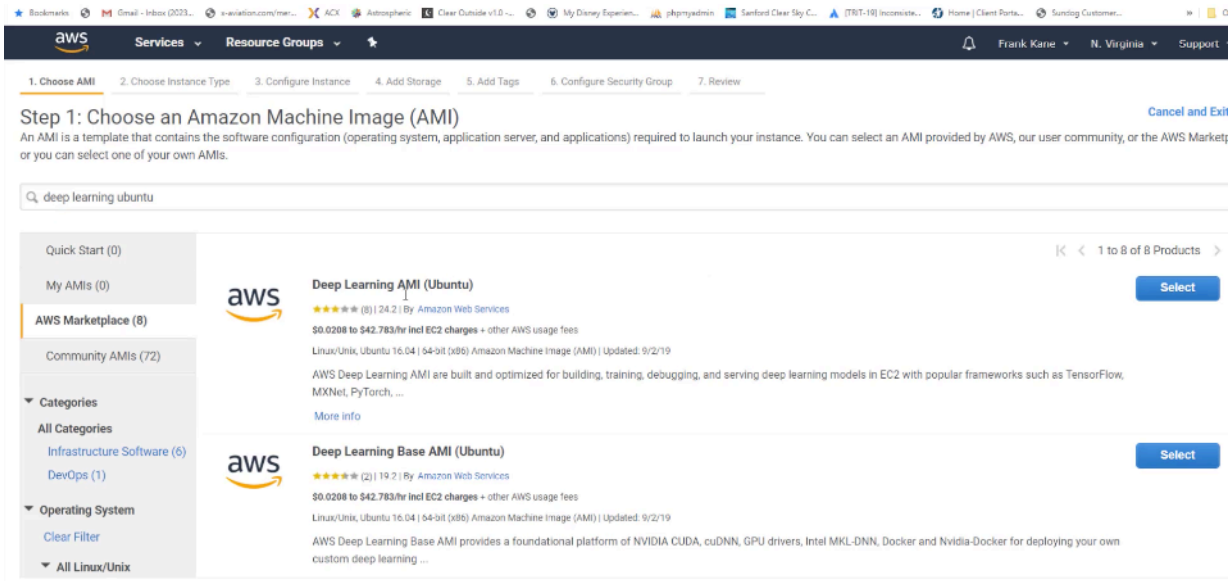
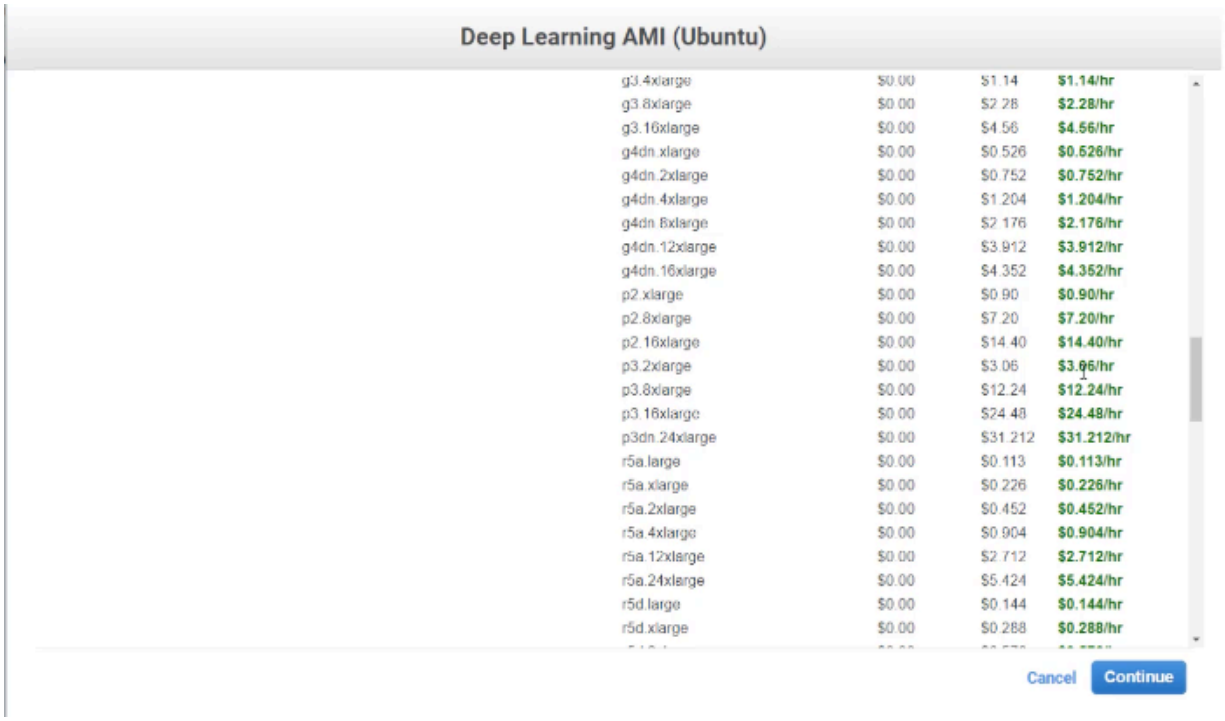
Udemy - 3 - Modeling Lab - CNN on EC2

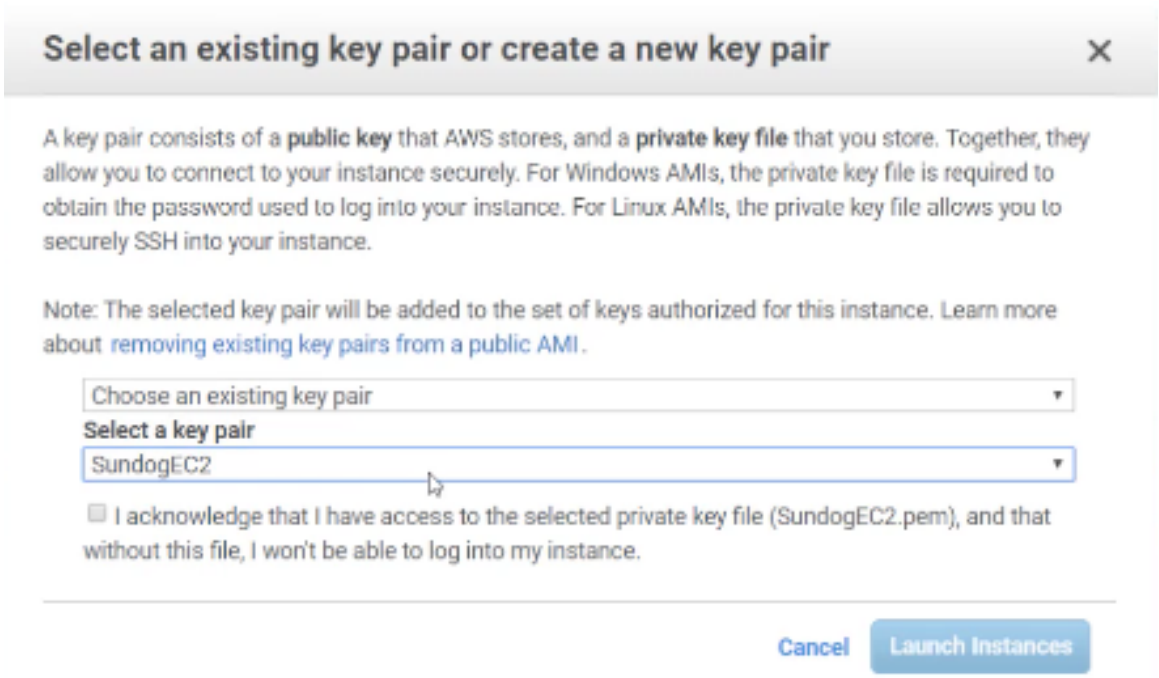
Create an EC2 using an AMI with deep learning

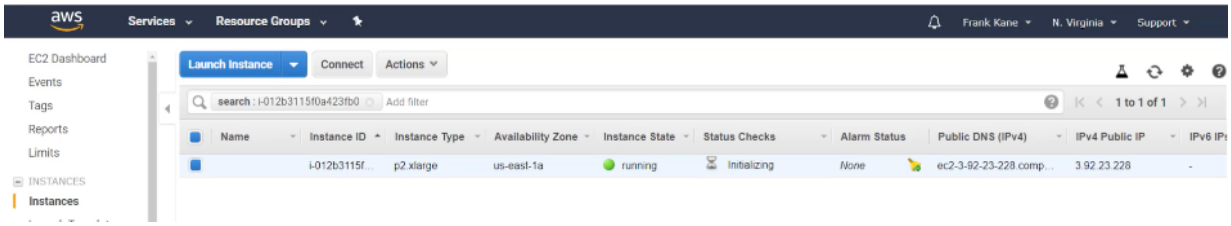


We will use a p2.2xl

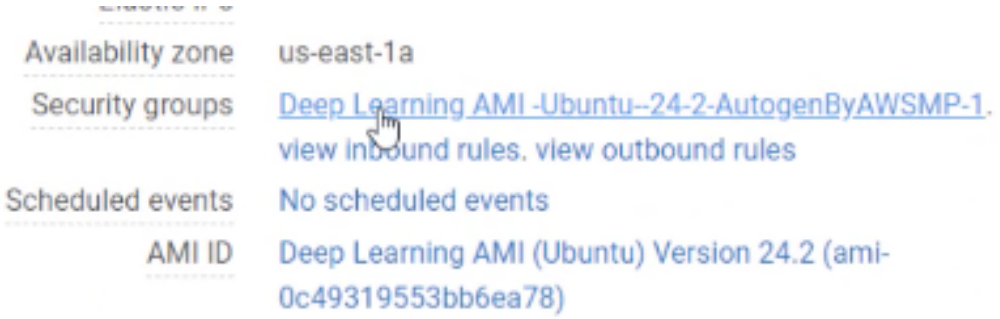


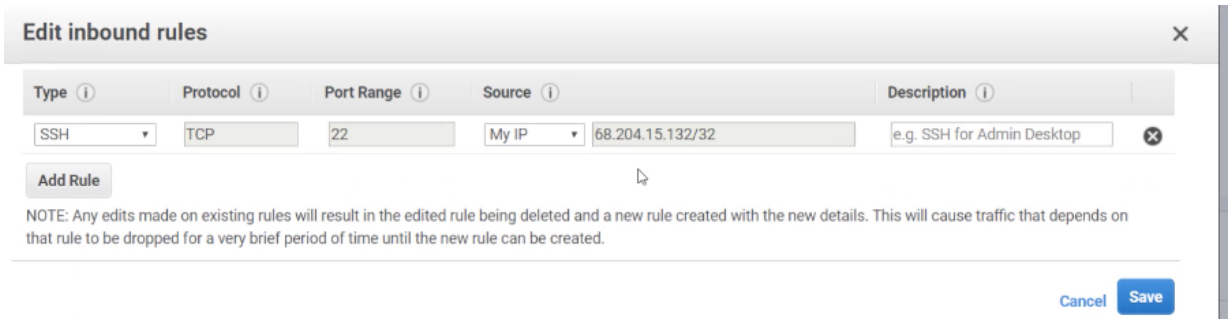
Select a key pair



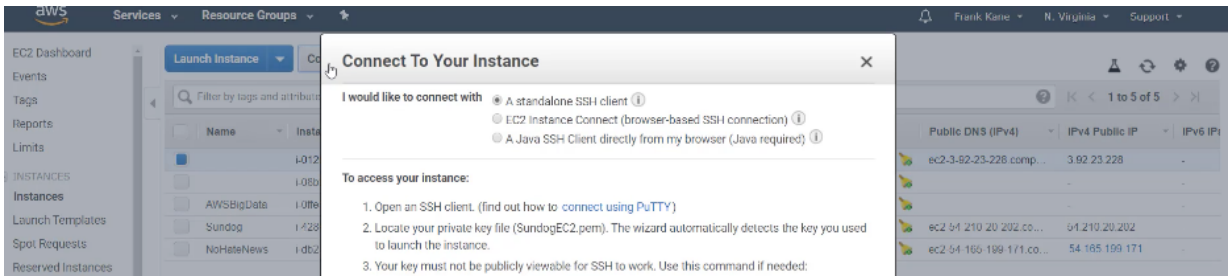


Make sure we can connect
Go to Security group and allow our IP



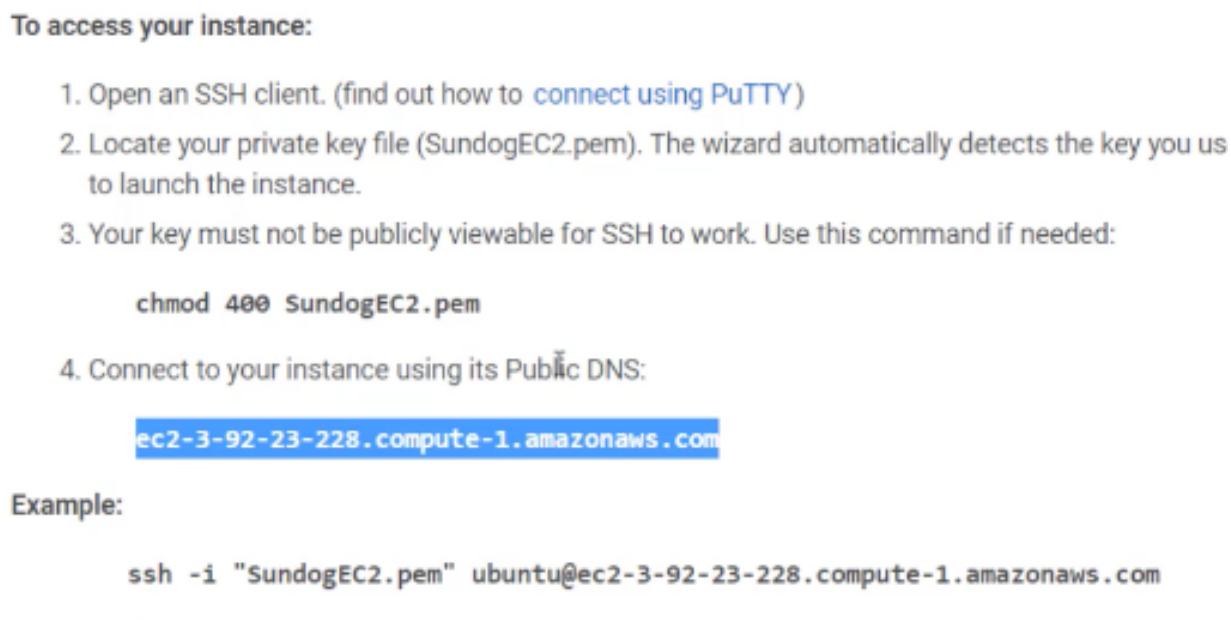


Now connect

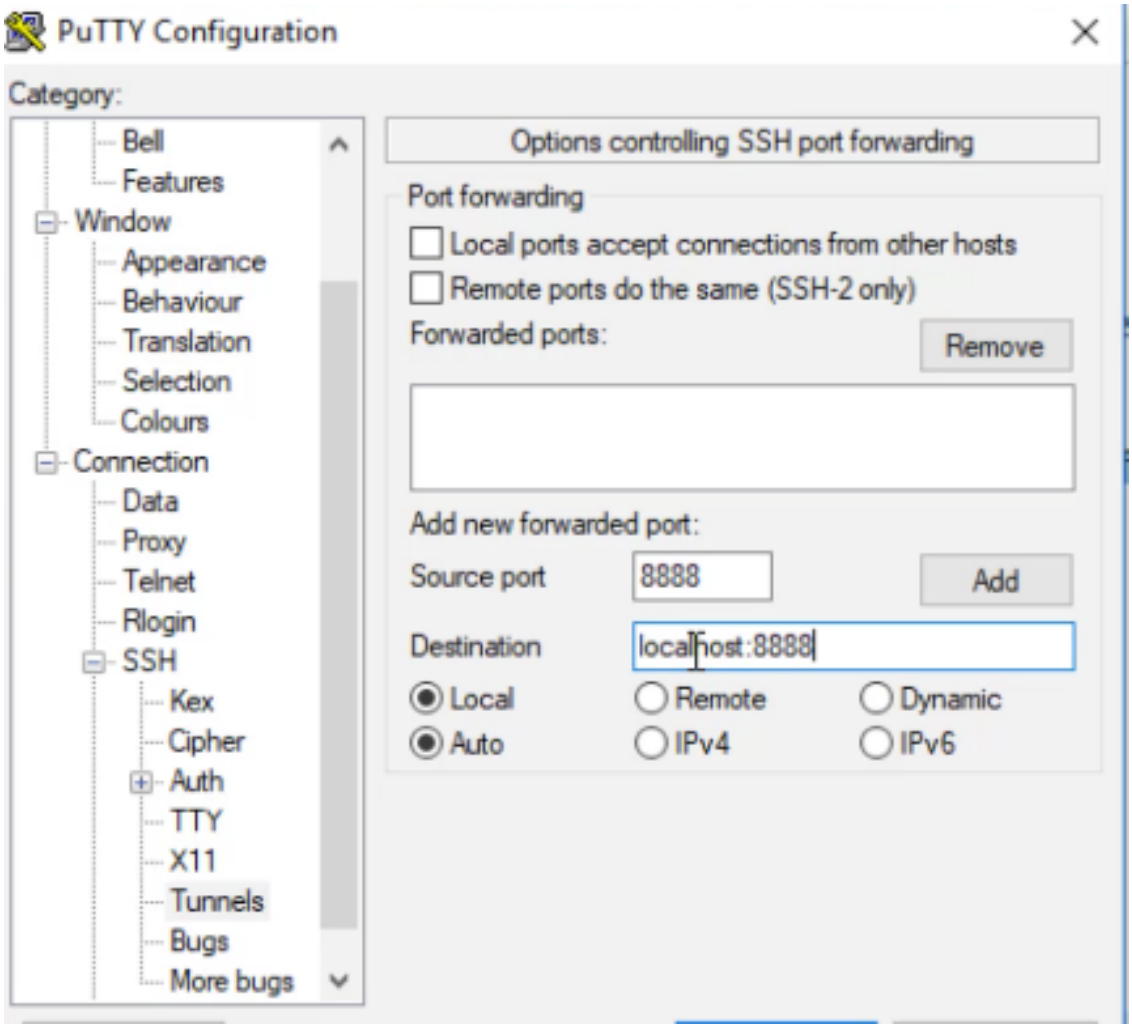


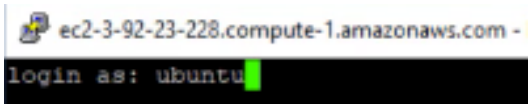
We want to use a Jupyter notebook and tunnel through it

Copy the Public DNS

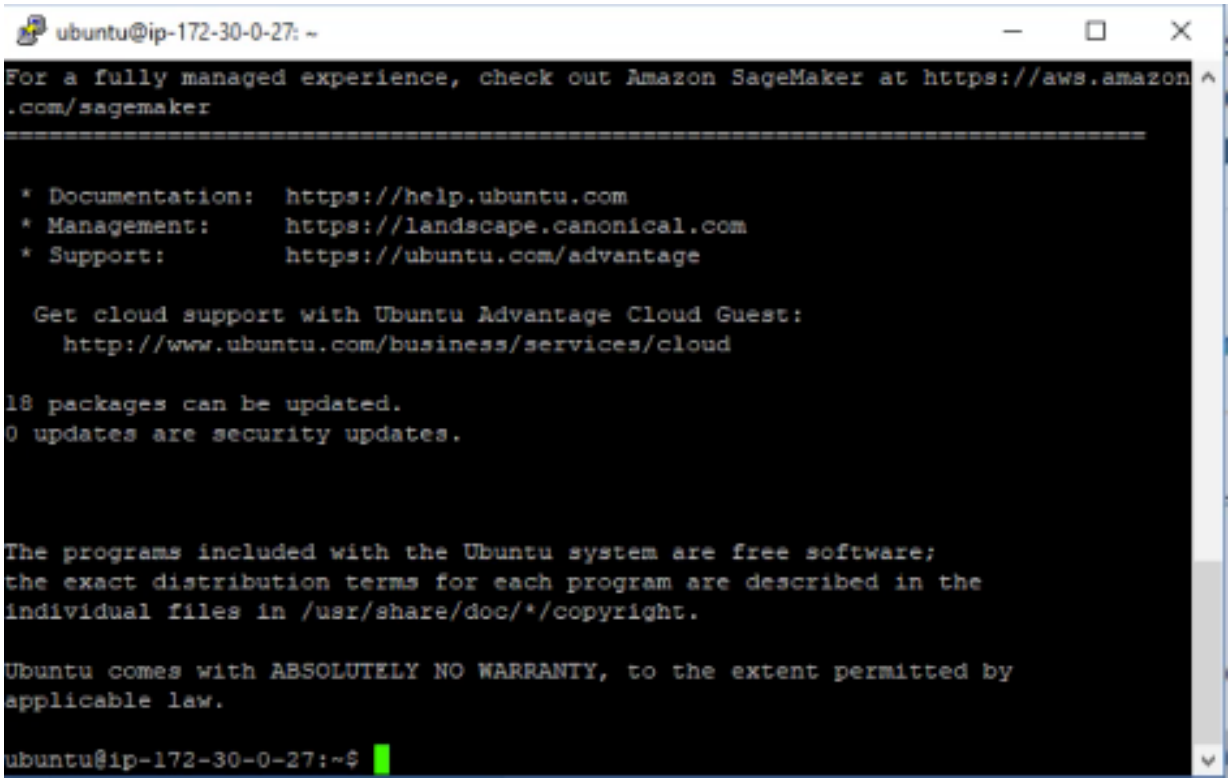


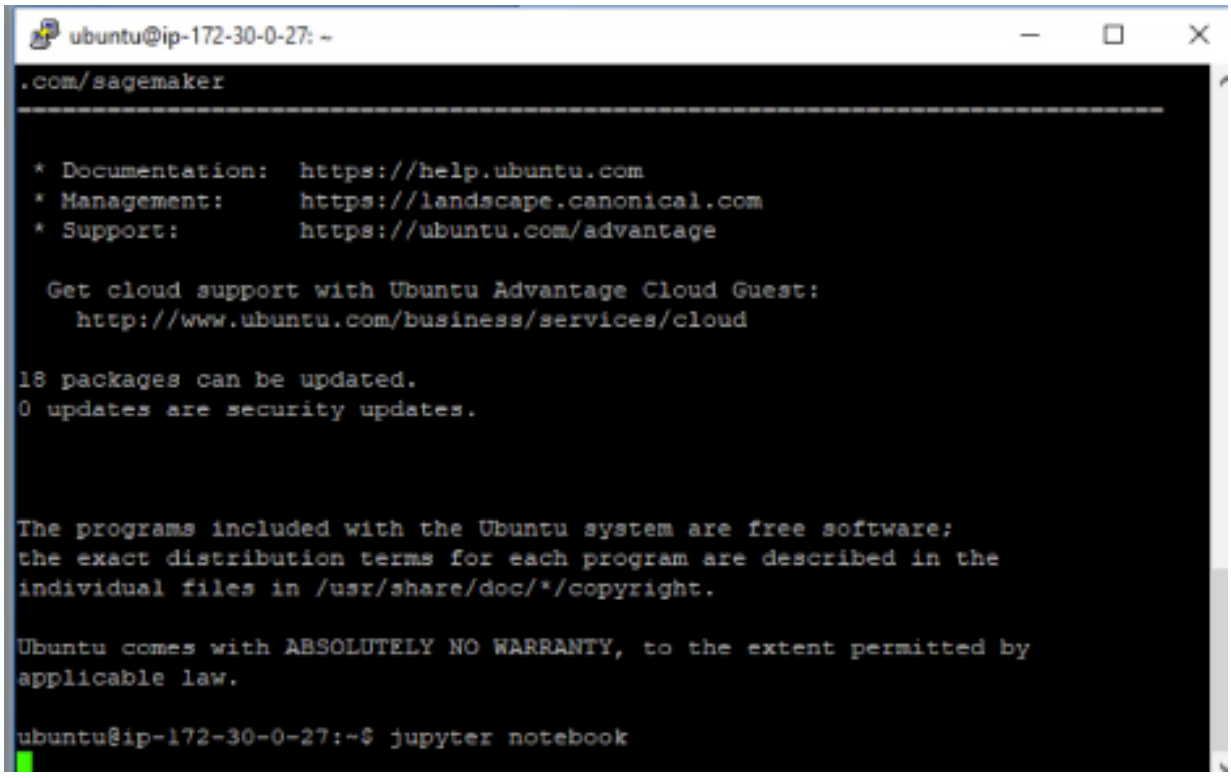
Set up tunnel



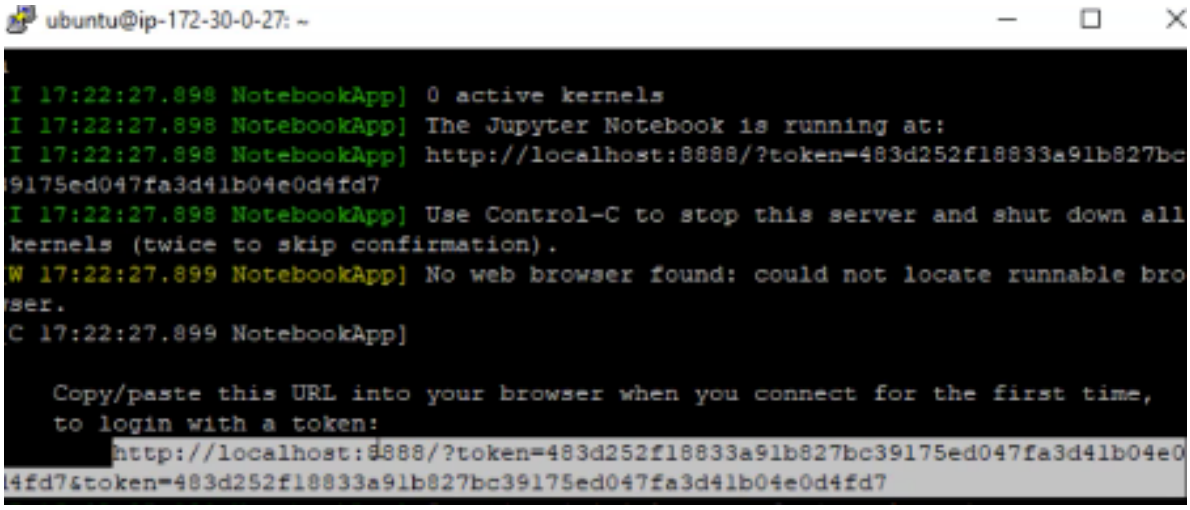


Now we are on the instance



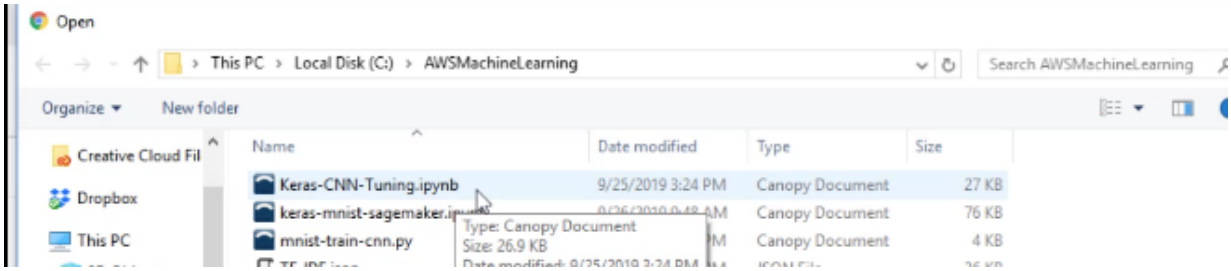


Copy the url and paste into browser

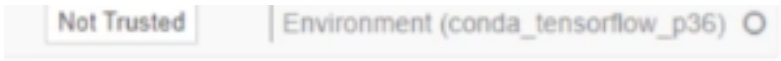


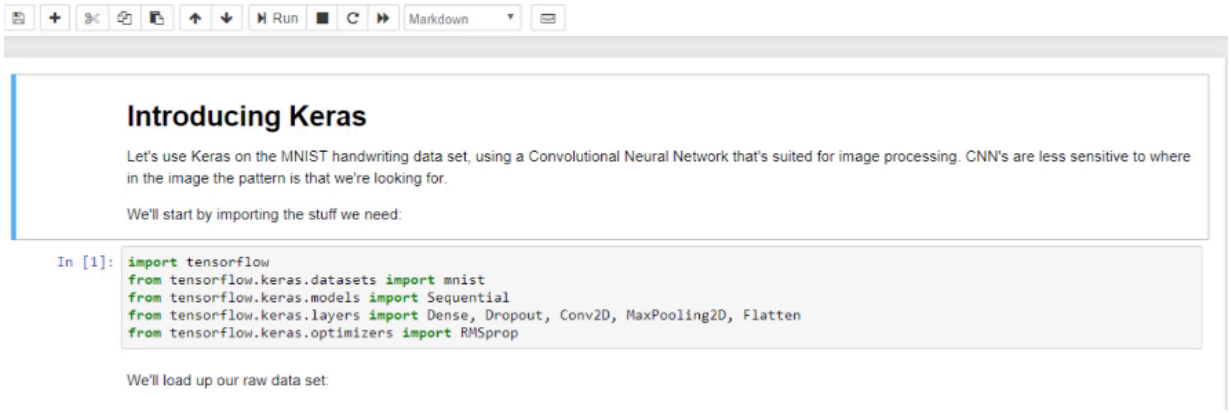
Now connected to EC2 via a tunnel.

Upload notebook



Use tensorflow kernel:





Introducing Keras

Let's use Keras on the MNIST handwriting data set, using a Convolutional Neural Network that's suited for image processing. CNN's are less sensitive to where in the image the pattern is that we're looking for.

We'll start by importing the stuff we need:

In [1]:
```
import tensorflow
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.optimizers import RMSprop
```

We'll load up our raw data set:

Load the data

In [2]:
```
(mnist_train_images, mnist_train_labels), (mnist_test_images, mnist_test_labels) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
```

Transform the data: reshape 1 dimensional pixel data into 2 dimensional arrays of 28x28 pixels
Also need to scale the data down (/ 255)

Since we're treating the data as 2D images of 28x28 pixels, we need to shape it accordingly. Depending on the data format Keras is set up for, this may be 1x28x28 or 28x28x1 (the "1" indicates a single color channel, as this is just grayscale. If we were dealing with color images, it would be 3 instead of 1 since we'd have red, green, and blue color channels)

In [3]:
```
from tensorflow.keras import backend as K

if K.image_data_format() == 'channels_first':
    train_images = mnist_train_images.reshape(mnist_train_images.shape[0], 1, 28, 28)
    test_images = mnist_test_images.reshape(mnist_test_images.shape[0], 1, 28, 28)
    input_shape = (1, 28, 28)
else:
    train_images = mnist_train_images.reshape(mnist_train_images.shape[0], 28, 28, 1)
    test_images = mnist_test_images.reshape(mnist_test_images.shape[0], 28, 28, 1)
    input_shape = (28, 28, 1)

train_images = train_images.astype('float32')
test_images = test_images.astype('float32')
train_images /= 255
test_images /= 255
```

Convert data with One-hot encoding

We need to convert our train and test labels to be categorical in one-hot format:

In [4]:
```
train_labels = tensorflow.keras.utils.to_categorical(mnist_train_labels, 10)
test_labels = tensorflow.keras.utils.to_categorical(mnist_test_labels, 10)
```

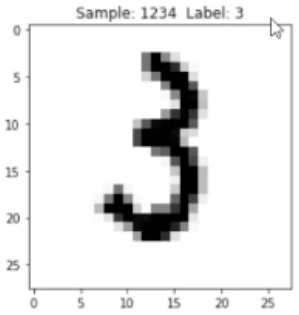
As a sanity check let's print out one of the training images with its label:

In [5]:
```
%matplotlib inline
import matplotlib.pyplot as plt

def display_sample(num):
    #Print the one-hot array of this sample's label
    print(train_labels[num])
    #Print the label converted back to a number
    label = train_labels[num].argmax(axis=0)
    #Reshape the 768 values to a 28x28 image
    image = train_images[num].reshape([28,28])
    plt.title('Sample: %d  Label: %d' % (num, label))
    plt.imshow(image, cmap=plt.get_cmap('gray_r'))
    plt.show()

display_sample(1234)
```

```
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
```



Now for the meat of the problem. Setting up a convolutional neural network involves more layers.

We'll start with a 2D convolution of the image - it's set up to take 32 windows, or "filters", of each image, each filter being 3x3 in size.

We then run a second convolution on top of that with 64 3x3 windows - this topology is just what comes recommended within Keras's own examples. Again you want to re-use previous research whenever possible while tuning CNN's, as it is hard to do.

Next we apply a MaxPooling2D layer that takes the maximum of each 2x2 result to distill the results down into something more manageable.

Next we flatten the 2D layer we have at this stage into a 1D layer. So at this point we can just pretend we have a traditional multi-layer perceptron...

... and feed that into a hidden, flat layer of 128 units.

And finally, we feed that into our final 10 units where softmax is applied to choose our category of 0-9.

In [6]:
```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
# 64 3x3 kernels
model.add(Conv2D(64, (3, 3), activation='relu'))
# Reduce by taking the max of each 2x2 block
model.add(MaxPooling2D(pool_size=(2, 2)))
# Flatten the results to one dimension for passing into our final layer
model.add(Flatten())
# A hidden layer to learn with
model.add(Dense(128, activation='relu'))
# Final categorization from 0-9 with softmax
model.add(Dense(10, activation='softmax'))
```

```
In [7]: model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 9216)              0
_________________________________________________________________
dense (Dense)                (None, 128)               1179776
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
```

We are doing multiple categorization, so categorical_crossentropy is still the right loss function to use. We'll use the Adam optimizer, although the example provided with Keras uses RMSProp. You might want to try both if you have time.

```
In [8]: model.compile(loss='categorical_crossentropy',
                    optimizer='adam',
                    metrics=['accuracy'])
```

Now we can train the model

Warning

This will take a few minutes to run on a p3.large instance.

```
In [9]: history = model.fit(train_images, train_labels,
                            batch_size=32,
                            epochs=10,
                            verbose=2,
                            validation_data=(test_images, test_labels))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 - 13s - loss: 0.1098 - acc: 0.9671 - val_loss: 0.0340 - val_acc: 0.9885
Epoch 2/10
60000/60000 - 12s - loss: 0.0379 - acc: 0.9875 - val_loss: 0.0295 - val_acc: 0.9904
Epoch 3/10
60000/60000 - 12s - loss: 0.0236 - acc: 0.9928 - val_loss: 0.0340 - val_acc: 0.9888
Epoch 4/10
60000/60000 - 12s - loss: 0.0157 - acc: 0.9950 - val_loss: 0.0371 - val_acc: 0.9886
Epoch 5/10
60000/60000 - 12s - loss: 0.0116 - acc: 0.9961 - val_loss: 0.0409 - val_acc: 0.9889
```

Accuracy on training data after epoch 3 its increasing quicker than with validation set
Sign we are over-fitting

=> Use Dropout layers to regularize.

So, you can see that we started overfitting pretty early on, as our accuracy on the test set started exceeding our accuracy on the validation set. Our validation accuracy maxed out at around 99.0% after just a couple of epochs, while our accuracy on the test set kept climbing.

To prevent overfitting, we need to perform some sort of regularization. Dropout layers are one such technique in deep learning; they work by "dropping out" neurons on each pass to force learning to spread itself out across the network as a whole.

In [10]:
```
def MakeModel():
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3),
                     activation='relu',
                     input_shape=input_shape))
    # 64 3x3 kernels
    model.add(Conv2D(64, (3, 3), activation='relu'))
    # Reduce by taking the max of each 2x2 block
    model.add(MaxPooling2D(pool_size=(2, 2)))
    # Dropout to avoid overfitting
    model.add(Dropout(0.25))
    # Flatten the results to one dimension for passing into our final layer
    model.add(Flatten())
    # A hidden layer to learn with
    model.add(Dense(128, activation='relu'))
    # Another dropout
    model.add(Dropout(0.5))
    # Final categorization from 0-9 with softmax
    model.add(Dense(10, activation='softmax'))
    return model

model = MakeModel()
```

Now results are better

Let's run it again with those two dropout layers added in.

In [11]:
```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

In [12]:
```
history = model.fit(train_images, train_labels,
                    batch_size=32,
                    epochs=10,
                    verbose=2,
                    validation_data=(test_images, test_labels))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 - 14s - loss: 0.1909 - acc: 0.9413 - val_loss: 0.0449 - val_acc: 0.9854
Epoch 2/10
60000/60000 - 13s - loss: 0.0791 - acc: 0.9758 - val_loss: 0.0300 - val_acc: 0.9902
Epoch 3/10
60000/60000 - 13s - loss: 0.0620 - acc: 0.9808 - val_loss: 0.0356 - val_acc: 0.9883
Epoch 4/10
60000/60000 - 13s - loss: 0.0511 - acc: 0.9845 - val_loss: 0.0288 - val_acc: 0.9914
Epoch 5/10
60000/60000 - 13s - loss: 0.0429 - acc: 0.9869 - val_loss: 0.0270 - val_acc: 0.9917
Epoch 6/10
60000/60000 - 13s - loss: 0.0356 - acc: 0.9890 - val_loss: 0.0266 - val_acc: 0.9918
Epoch 7/10
60000/60000 - 13s - loss: 0.0331 - acc: 0.9890 - val_loss: 0.0279 - val_acc: 0.9923
Epoch 8/10
60000/60000 - 13s - loss: 0.0300 - acc: 0.9902 - val_loss: 0.0270 - val_acc: 0.9927
Epoch 9/10
60000/60000 - 13s - loss: 0.0266 - acc: 0.9913 - val_loss: 0.0303 - val_acc: 0.9920
Epoch 10/10
60000/60000 - 13s - loss: 0.0261 - acc: 0.9914 - val_loss: 0.0289 - val_acc: 0.9923
```

That's better; our train and test accuracy ended up about the same, at 99.2%. There may still be a tiny bit of overfitting going on, but it's a lot better.

Let's also explore the effect the batch size has; as an experiment, let's increase it up to 1000:

Let’s now explore batch size of 1000 instead of 32

In [13]:
```
model = MakeModel()

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

history = model.fit(train_images, train_labels,
                    batch_size=1000,
                    epochs=10,
                    verbose=2,
                    validation_data=(test_images, test_labels))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 - 6s - loss: 0.5625 - acc: 0.8303 - val_loss: 0.1380 - val_acc: 0.9585
Epoch 2/10
60000/60000 - 5s - loss: 0.1605 - acc: 0.9540 - val_loss: 0.0676 - val_acc: 0.9799
Epoch 3/10
60000/60000 - 5s - loss: 0.1009 - acc: 0.9707 - val_loss: 0.0480 - val_acc: 0.9848
Epoch 4/10
60000/60000 - 5s - loss: 0.0779 - acc: 0.9770 - val_loss: 0.0417 - val_acc: 0.9859
Epoch 5/10
60000/60000 - 5s - loss: 0.0667 - acc: 0.9801 - val_loss: 0.0382 - val_acc: 0.9869
Epoch 6/10
60000/60000 - 5s - loss: 0.0568 - acc: 0.9829 - val_loss: 0.0323 - val_acc: 0.9885
Epoch 7/10
60000/60000 - 5s - loss: 0.0497 - acc: 0.9848 - val_loss: 0.0302 - val_acc: 0.9899
Epoch 8/10
60000/60000 - 5s - loss: 0.0457 - acc: 0.9861 - val_loss: 0.0300 - val_acc: 0.9894
Epoch 9/10
60000/60000 - 5s - loss: 0.0397 - acc: 0.9876 - val_loss: 0.0296 - val_acc: 0.9897
60000/60000 - 5s - loss: 0.0380 - acc: 0.9885 - val_loss: 0.0288 - val_acc: 0.9906
```

click to scroll output; double click to hide

Results are not as good
Pb of too large batch size -> can get stuck in local minima

> If you run this block a few times, you'll probably get very different results. Large batch sizes tend to get stuck in "local minima", and converge on the wrong solution at random. Smaller batch sizes also have a regularization effect. Sometimes you'll get lucky and the large batch will converge on a good solution; other times, not so much.

Now let’s explroe learning rate impact buy increasing it

Let's explore the effect of the learning rate. The default learning rate for Adam is 0.001; let's see what happens if we increase it by an order of magnitude to 0.01:

In [19]:
```
model = MakeModel()

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

adam = tensorflow.keras.optimizers.Adam(learning_rate=0.01)

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

Results are bad

In [15]:
```
history = model.fit(train_images, train_labels,
                    batch_size=32,
                    epochs=10,
                    verbose=2,
                    validation_data=(test_images, test_labels))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 - 14s - loss: 0.2704 - acc: 0.9195 - val_loss: 0.0812 - val_acc: 0.9757
Epoch 2/10
60000/60000 - 13s - loss: 0.1964 - acc: 0.9443 - val_loss: 0.0779 - val_acc: 0.9768
Epoch 3/10
60000/60000 - 13s - loss: 0.1840 - acc: 0.9493 - val_loss: 0.0679 - val_acc: 0.9801
Epoch 4/10
60000/60000 - 13s - loss: 0.1837 - acc: 0.9490 - val_loss: 0.0923 - val_acc: 0.9772
Epoch 5/10
60000/60000 - 13s - loss: 0.1781 - acc: 0.9524 - val_loss: 0.0695 - val_acc: 0.9797
Epoch 6/10
60000/60000 - 13s - loss: 0.1794 - acc: 0.9523 - val_loss: 0.0923 - val_acc: 0.9786
Epoch 7/10
60000/60000 - 13s - loss: 0.1821 - acc: 0.9521 - val_loss: 0.0743 - val_acc: 0.9774
Epoch 8/10
60000/60000 - 13s - loss: 0.1673 - acc: 0.9562 - val_loss: 0.0723 - val_acc: 0.9797
Epoch 9/10
60000/60000 - 13s - loss: 0.1755 - acc: 0.9542 - val_loss: 0.0757 - val_acc: 0.9804
Epoch 10/10
60000/60000 - 14s - loss: 0.1773 - acc: 0.9540 - val_loss: 0.0892 - val_acc: 0.9804
```

Yikes! That had a huge, and terrible, effect on the results. Small batch sizes are best paired with low learning rates, and large learning rates have a tendency to overshoot the correct solution entirely - which is probably what happened here. The learning rate is an example of a hyperparameter that you might want to tune by just trying different values; we'll see more of that later in the course.

Its because large learning rate has a tendency of over shooting correct minima