Convolutional neural networks

In this notebook, we will find out makes convolutional neural networks so powerful for computer vision applications!

We will use three varieties of neural networks to classify our own handwritten digits.

```
# Colab switched to TF 2.3 in August 2020 - breaks this code
# as described in https://github.com/tensorflow/tensorflow/issues/34201
# workaround via https://github.com/googlecolab/colabtools/issues/1470#issuecomment-668897876
!pip install tensorflow~=2.1.0 tensorflow_gcs_config~=2.1.0
!pip install keras==2.3.0
```

```
Collecting tensorflow~=2.1.0
-manylinux2010_x86_64.whl (421.8MB)
ent already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.6/dist-packages (from
    tensorflow~=2.1.0) (1.1.0)
Collecting keras-applications>=1.0.8
ent already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.6/dist-packages (from
    tensorflow~=2.1.0) (3.3.0)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.6/dist-packages (from
    tensorflow~=2.1.0) (1.15.0)
Collecting tensorflow-estimator<2.2.0,>=2.1.0rc0
ator-2.1.0-py2.py3-none-any.whl (448kB)
ent already satisfied: absl-py>=0.7.0 in /usr/local/lib/python3.6/dist-packages (from
    tensorflow~=2.1.0) (0.9.0)
Collecting gast==0.2.2
  Downloading
      https://files.pythonhosted.org/packages/4e/35/11749bf99b2d4e3cceb4d55ca22590b0d7c2c62b9de38ac4a4a7f4687
Requirement already satisfied: google-pasta>=0.1.6 in /usr/local/lib/python3.6/dist-packages
    (from tensorflow~=2.1.0) (0.2.0)
Collecting tensorboard<2.2.0,>=2.1.0
ent already satisfied: wrapt>=1.11.1 in /usr/local/lib/python3.6/dist-packages (from
    tensorflow~=2.1.0) (1.12.1)
Requirement already satisfied: wheel>=0.26; python_version >= "3" in
    /usr/local/lib/python3.6/dist-packages (from tensorflow~=2.1.0) (0.34.2)
Requirement already satisfied: scipy==1.4.1; python_version >= "3" in
    /usr/local/lib/python3.6/dist-packages (from tensorflow~=2.1.0) (1.4.1)
Requirement already satisfied: protobuf>=3.8.0 in /usr/local/lib/python3.6/dist-packages
    (from tensorflow~=2.1.0) (3.12.4)
Requirement already satisfied: numpy<2.0,>=1.16.0 in /usr/local/lib/python3.6/dist-packages
    (from tensorflow~=2.1.0) (1.18.5)
Requirement already satisfied: astor>=0.6.0 in /usr/local/lib/python3.6/dist-packages (from
    tensorflow~=2.1.0) (0.8.1)
Requirement already satisfied: grpcio>=1.8.6 in /usr/local/lib/python3.6/dist-packages (from
    tensorflow~=2.1.0) (1.30.0)
Requirement already satisfied: h5py in /usr/local/lib/python3.6/dist-packages (from
    keras-applications>=1.0.8->tensorflow~=2.1.0) (2.10.0)
Requirement already satisfied: google-auth<2,>=1.6.3 in
    /usr/local/lib/python3.6/dist-packages (from
    tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (1.17.2)
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.6/dist-packages
    (from tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (1.0.1)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.6/dist-packages
    (from tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (3.2.2)
```

```
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.6/dist-packages
    (from tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (2.23.0)
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in
    /usr/local/lib/python3.6/dist-packages (from
    tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (0.4.1)
Requirement already satisfied: setuptools>=41.0.0 in /usr/local/lib/python3.6/dist-packages
    (from tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (49.2.0)
Requirement already satisfied: cachetools<5.0,>=2.0.0 in
    /usr/local/lib/python3.6/dist-packages (from
    google-auth<2,>=1.6.3->tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (4.1.1)
Requirement already satisfied: rsa<5,>=3.1.4; python_version >= "3" in
    /usr/local/lib/python3.6/dist-packages (from
    google-auth<2,>=1.6.3->tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (4.6)
Requirement already satisfied: pyasn1-modules>=0.2.1 in
    /usr/local/lib/python3.6/dist-packages (from
    google-auth<2,>=1.6.3->tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (0.2.8)
Requirement already satisfied: importlib-metadata; python_version < "3.8" in
    /usr/local/lib/python3.6/dist-packages (from
    markdown>=2.6.8->tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (1.7.0)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
    /usr/local/lib/python3.6/dist-packages (from
    requests<3,>=2.21.0->tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.6/dist-packages
    (from requests<3,>=2.21.0->tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.6/dist-packages
    (from requests<3,>=2.21.0->tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (2020.6.20)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.6/dist-packages (from
    requests<3,>=2.21.0->tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (2.10)
Requirement already satisfied: requests-oauthlib>=0.7.0 in
    /usr/local/lib/python3.6/dist-packages (from
    google-auth-oauthlib<0.5,>=0.4.1->tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (1.3.0)
Requirement already satisfied: pyasn1>=0.1.3 in /usr/local/lib/python3.6/dist-packages (from
    rsa<5,>=3.1.4; python_version >=
    "3"->google-auth<2,>=1.6.3->tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (0.4.8)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.6/dist-packages (from
    importlib-metadata; python_version <
    "3.8"->markdown>=2.6.8->tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0) (3.1.0)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.6/dist-packages
    (from
    requests-oauthlib>=0.7.0->google-auth-oauthlib<0.5,>=0.4.1->tensorboard<2.2.0,>=2.1.0->tensorflow~=2.1.0)
    (3.1.0)
Building wheels for collected packages: gast
  Building wheel for gast (setup.py) ... e=gast-0.2.2-cp36-none-any.whl size=7540
      sha256=83656ba30ed9284d896c7bd8fa8f2b8fb2f739ea290b97cf1cdd86217b99fd05
  Stored in directory:
      /root/.cache/pip/wheels/5c/2e/7e/a1d4d4fcebe6c381f378ce7743a3ced3699feb89bcfbdadadd
Successfully built gast
ERROR: tensorflow-probability 0.11.0 has requirement gast>=0.3.2, but you'll have gast 0.2.2
    which is incompatible.
Installing collected packages: keras-preprocessing, keras-applications,
    tensorflow-estimator, gast, tensorboard, tensorflow, tensorflow-gcs-config
  Found existing installation: Keras-Preprocessing 1.1.2
    Uninstalling Keras-Preprocessing-1.1.2:
      Successfully uninstalled Keras-Preprocessing-1.1.2
```

```
  Found existing installation: tensorflow-estimator 2.3.0
    Uninstalling tensorflow-estimator-2.3.0:
      Successfully uninstalled tensorflow-estimator-2.3.0
  Found existing installation: gast 0.3.3
    Uninstalling gast-0.3.3:
      Successfully uninstalled gast-0.3.3
  Found existing installation: tensorboard 2.3.0
    Uninstalling tensorboard-2.3.0:
      Successfully uninstalled tensorboard-2.3.0
  Found existing installation: tensorflow 2.3.0
    Uninstalling tensorflow-2.3.0:
      Successfully uninstalled tensorflow-2.3.0
  Found existing installation: tensorflow-gcs-config 2.3.0
    Uninstalling tensorflow-gcs-config-2.3.0:
      Successfully uninstalled tensorflow-gcs-config-2.3.0
Successfully installed gast-0.2.2 keras-applications-1.0.8 keras-preprocessing-1.1.0
    tensorboard-2.1.1 tensorflow-2.1.1 tensorflow-estimator-2.1.0 tensorflow-gcs-config-2.1.8
Collecting keras==2.3.0
ent already satisfied: scipy>=0.14 in /usr/local/lib/python3.6/dist-packages (from
    keras==2.3.0) (1.4.1)
Requirement already satisfied: numpy>=1.9.1 in /usr/local/lib/python3.6/dist-packages (from
    keras==2.3.0) (1.18.5)
Requirement already satisfied: keras-applications>=1.0.6 in
    /usr/local/lib/python3.6/dist-packages (from keras==2.3.0) (1.0.8)
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.6/dist-packages (from
    keras==2.3.0) (1.15.0)
Requirement already satisfied: keras-preprocessing>=1.0.5 in
    /usr/local/lib/python3.6/dist-packages (from keras==2.3.0) (1.1.0)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.6/dist-packages (from
    keras==2.3.0) (3.13)
Requirement already satisfied: h5py in /usr/local/lib/python3.6/dist-packages (from
    keras==2.3.0) (2.10.0)
Installing collected packages: keras
  Found existing installation: Keras 2.4.3
    Uninstalling Keras-2.4.3:
      Successfully uninstalled Keras-2.4.3
Successfully installed keras-2.3.0
```

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

import tensorflow as tf
from tensorflow.keras import optimizers
from keras.models import Sequential, Model
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.layers import Conv2D, MaxPooling2D
#from tensorflow.python.keras import backend as K
from keras import backend as K
from keras.datasets import mnist
from keras.utils import plot_model
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19: FutureWarning:
    pandas.util.testing is deprecated. Use the functions in the public API at pandas.testing
    instead.
  import pandas.util.testing as tm
Using TensorFlow backend.
```

```
print(tf.__version__) # must be 2.1!
```

```
2.1.1
```

Import data

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz
11493376/11490434 [==============================] - 2s 0us/step
```

```
X_train.shape
```

```
(60000, 28, 28)
```

```
X_test.shape
```

```
(10000, 28, 28)
```

Train a fully connected neural network on MNIST

Attribution: This section is based closely on this demo notebook by Sundeep Rangan

First, we will train a simple neural network. We have:

- One hidden layer with $N_H = 100$ units, with sigmoid activation.
- One output layer with $N_O = 10$ units, one for each of the 10 possible classes. The output activation is softmax, which is used for multi-class targets

First, we clear our session to make sure nothing is hanging around from previous models:

```
K.clear_session()
```

We will prepare our data by scaling it.

We will also separate part of the training data to use for model tuning. The accuracy on this validation set will be used to determine when to stop training the model.

```
# scale
X_train_nn = X_train/255.0
X_test_nn = X_test/255.0

# reshape
X_train_nn = X_train_nn.reshape(X_train.shape[0], X_train.shape[1]*X_train.shape[2])
```

```
X_test_nn = X_test_nn.reshape(X_test.shape[0], X_test.shape[1]*X_test.shape[2])

# split training set so we can use part of it for model tuning
X_train_nn, X_val_nn, y_train_nn, y_val_nn = train_test_split(X_train_nn, y_train,
    test_size=1.0/6.0)

print("Training data shape", X_train_nn.shape)
print("Validation data shape", X_val_nn.shape)
print("Testing data shape", X_test_nn.shape)
```

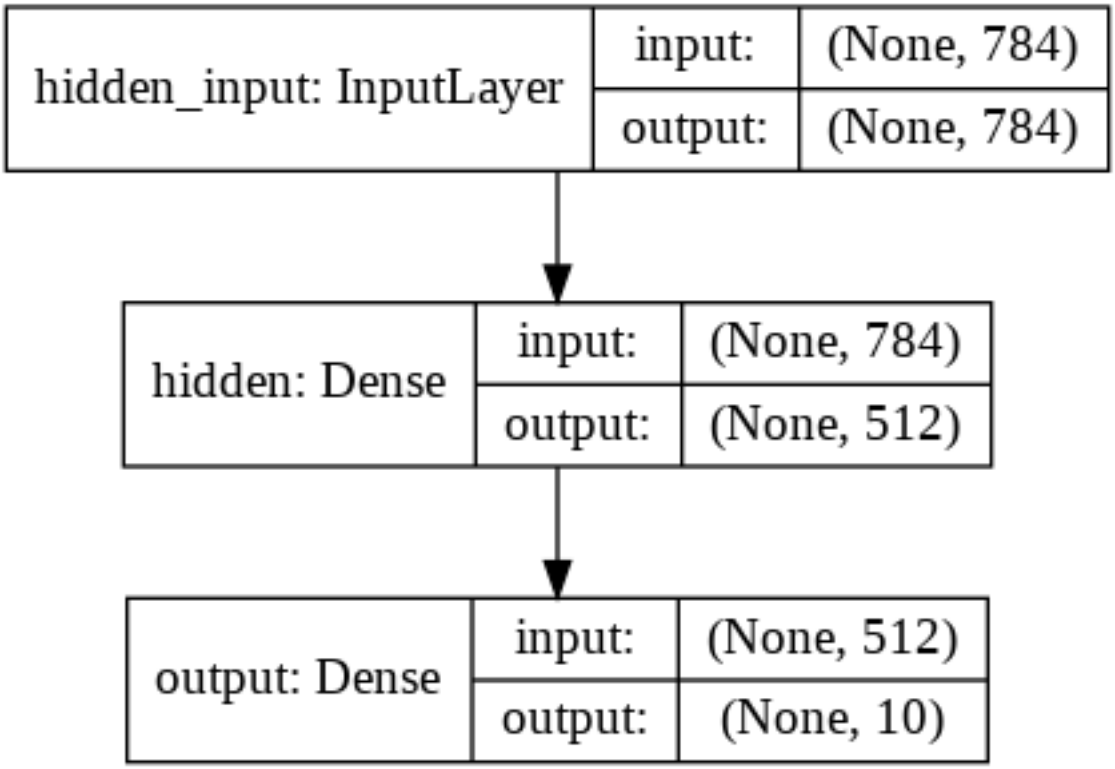
```
Training data shape (50000, 784)
Validation data shape (10000, 784)
Testing data shape (10000, 784)
```

Then, we can prepare our neural network:

```
nin = X_train_nn.shape[1]  # dimension of input data
nh = 512     # number of hidden units
nout = 10   # number of outputs
model_fc = Sequential()
model_fc.add(Dense(units=nh, input_shape=(nin,), activation='sigmoid', name='hidden'))
model_fc.add(Dense(units=nout, activation='softmax', name='output'))
model_fc.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
hidden (Dense)               (None, 512)               401920
_________________________________________________________________
output (Dense)               (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

```
plot_model(model_fc, "mnist-dense.png", show_shapes=True)
```



To train the network, we have to select an optimizer and a loss function. Since this is a multi-class classification problem, we select the `sparse_categorical_crossentropy` loss. We use the Adam optimizer for our gradient descent.

We also set the metrics that we wish to track during the optimization. In this case, we select accuracy on the training set.

```
opt = optimizers.Adam(lr=0.005)
model_fc.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Finally, we are ready to train our network. We wil specify the number of epochs and the batch size. We will also use a callback function to configure the training process to stop before the configured number of epochs, if no improvement in the validation set accuracy is observed for several epochs. In case of "early stopping", it will also the restore the weights that had the best performance on the validation set.

```
es = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', mode='max',
                                      patience=5,restore_best_weights=True )
```

Note that since the `fit` command is split across multiple lines, we cannot use the line-level magic command `%time` that we used previously to time it. Instead, we use the cell-level magic equivalent `%%time`, which reports the time to execute the entire cell

```
%%time
hist = model_fc.fit(X_train_nn, y_train_nn,
                        epochs=100, batch_size=128,
                        validation_data=(X_val_nn,y_val_nn),
                        callbacks=[es])
```

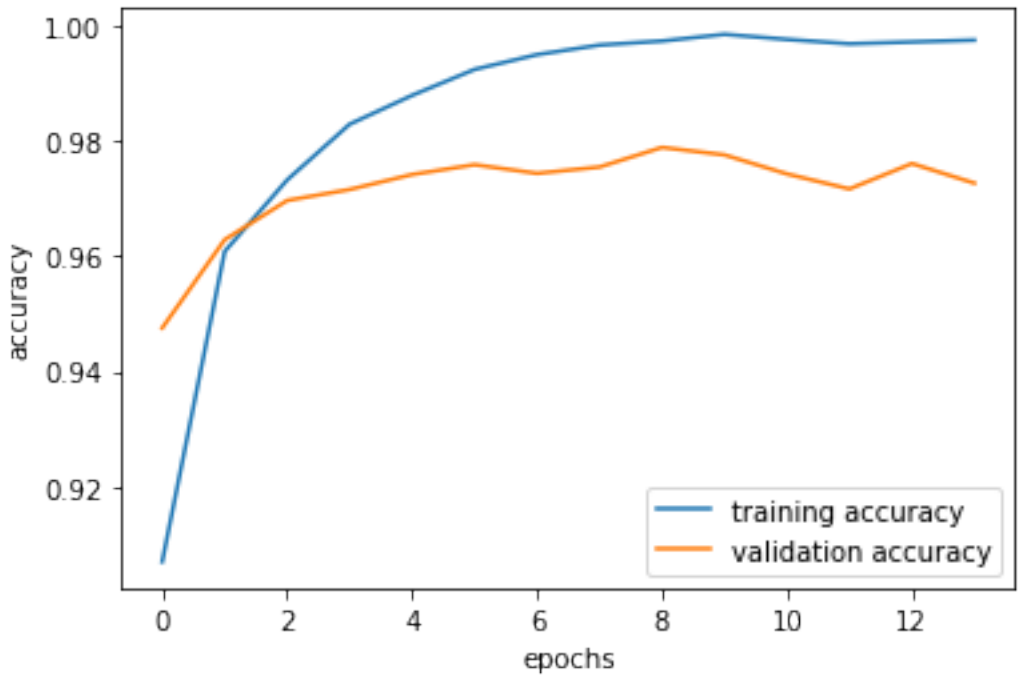
```
Train on 50000 samples, validate on 10000 samples
Epoch 1/100
50000/50000 [==============================] - 3s 59us/step - loss: 0.3118 - accuracy:
    0.9070 - val_loss: 0.1858 - val_accuracy: 0.9475
Epoch 2/100
50000/50000 [==============================] - 2s 31us/step - loss: 0.1353 - accuracy:
    0.9608 - val_loss: 0.1285 - val_accuracy: 0.9628
Epoch 3/100
50000/50000 [==============================] - 2s 32us/step - loss: 0.0873 - accuracy:
    0.9732 - val_loss: 0.1048 - val_accuracy: 0.9696
Epoch 4/100
50000/50000 [==============================] - 2s 34us/step - loss: 0.0567 - accuracy:
    0.9828 - val_loss: 0.0986 - val_accuracy: 0.9715
Epoch 5/100
50000/50000 [==============================] - 2s 35us/step - loss: 0.0392 - accuracy:
    0.9878 - val_loss: 0.0856 - val_accuracy: 0.9741
Epoch 6/100
50000/50000 [==============================] - 2s 37us/step - loss: 0.0267 - accuracy:
    0.9923 - val_loss: 0.0867 - val_accuracy: 0.9758
Epoch 7/100
50000/50000 [==============================] - 2s 36us/step - loss: 0.0184 - accuracy:
    0.9949 - val_loss: 0.0955 - val_accuracy: 0.9743
Epoch 8/100
50000/50000 [==============================] - 2s 36us/step - loss: 0.0134 - accuracy:
    0.9965 - val_loss: 0.0919 - val_accuracy: 0.9754
Epoch 9/100
50000/50000 [==============================] - 2s 37us/step - loss: 0.0104 - accuracy:
    0.9972 - val_loss: 0.0834 - val_accuracy: 0.9788
Epoch 10/100
50000/50000 [==============================] - 2s 33us/step - loss: 0.0068 - accuracy:
    0.9984 - val_loss: 0.0932 - val_accuracy: 0.9775
Epoch 11/100
50000/50000 [==============================] - 2s 32us/step - loss: 0.0091 - accuracy:
    0.9975 - val_loss: 0.1074 - val_accuracy: 0.9742
Epoch 12/100
50000/50000 [==============================] - 2s 32us/step - loss: 0.0102 - accuracy:
    0.9968 - val_loss: 0.1296 - val_accuracy: 0.9716
Epoch 13/100
50000/50000 [==============================] - 2s 32us/step - loss: 0.0096 - accuracy:
    0.9971 - val_loss: 0.1053 - val_accuracy: 0.9760
Epoch 14/100
50000/50000 [==============================] - 2s 35us/step - loss: 0.0084 - accuracy:
    0.9974 - val_loss: 0.1242 - val_accuracy: 0.9726
CPU times: user 25.7 s, sys: 3.19 s, total: 28.8 s
Wall time: 25.4 s
```

Next, we plot the training accuracy and validation accuracy vs. the epoch number. This helps us understand whether our network is overfitted; we may suspect overfitting if the training performance is improving with additional training epochs while the validation performance is getting worse.

In this case, we can see that we ”saturated” the training accuracy at 100%, while the accuracy on the test set is a bit lower than that.

```
tr_accuracy = hist.history['accuracy']
```

```
val_accuracy = hist.history['val_accuracy']

plt.plot(tr_accuracy);
plt.plot(val_accuracy);
plt.xlabel('epochs');
plt.ylabel('accuracy');
plt.legend(['training accuracy', 'validation accuracy']);
```



Now we can make predictions with our fitted model:

```
%time y_pred_prob_nn = model_fc.predict(X_test_nn)
y_pred_nn = np.argmax(y_pred_prob_nn, axis=-1)
```

```
CPU times: user 387 ms, sys: 65.3 ms, total: 452 ms
Wall time: 353 ms
```

And compute accuracy:

```
acc = accuracy_score(y_test, y_pred_nn)
acc
```

```
0.98
```

Note that we can also compute the accuracy with

```
score = model_fc.evaluate(X_test_nn, y_test)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

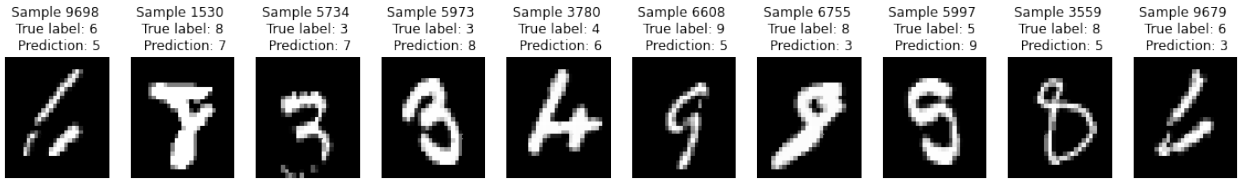
```
10000/10000 [==============================] - 1s 64us/step
Test score: 0.07270313802413293
```

```
Test accuracy: 0.9800000190734863
```

Our neural network does pretty well! Currently, the state of the art (best result) on the MNIST dataset is 0.21% classification error - you can see some of the best-performing methods at this link.

Furthermore, looking at some of the samples that are misclassified by our network, we can see that many of these samples are difficult for humans to classify as well. (Some may even be labeled incorrectly!)

```
num_samples = 10
p = plt.figure(figsize=(num_samples*2,2))
idxs_mis = np.flatnonzero(y_test!=y_pred_nn)
idxs = np.random.choice(idxs_mis, num_samples, replace=False)
for i, idx in enumerate(idxs):
  p = plt.subplot(1, num_samples, i+1);
  p = sns.heatmap(X_test[idx].astype('uint8'), cmap=plt.cm.gray,
            xticklabels=False, yticklabels=False, cbar=False)
  p = plt.axis('off');
  p = plt.title("Sample %d \n True label: %d \n Prediction: %d" % (idx, y_test[idx],
      y_pred_nn[idx]));
plt.show()
```



Try our fully connected neural network on our own test sample

Now, let's try to classify our own test sample (as in a previous homework assignment).

On a plain white piece of paper, in a black or other dark-colored pen, write a digit of your choice from 0 to 9. Take a photo of your handwritten digit.

Edit your photo (crop, rotate as needed), using a photo editor of your choice (I used Google Photos), so that your photo is approximately square, and includes only the digit and the white background. Upload your image here.

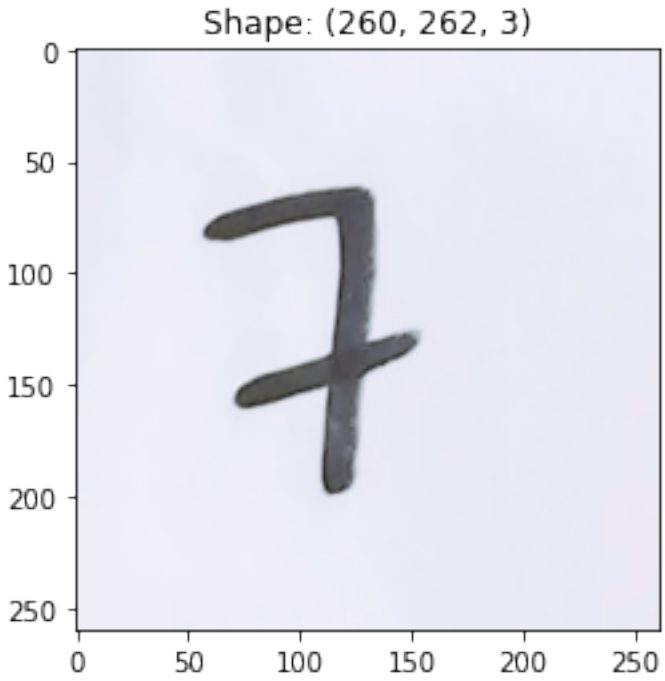
```
from google.colab import files

uploaded = files.upload()

for fn in uploaded.keys():
  print('User uploaded file "{name}" with length {length} bytes'.format(
      name=fn, length=len(uploaded[fn])))
```

```
<IPython.core.display.HTML object>
```

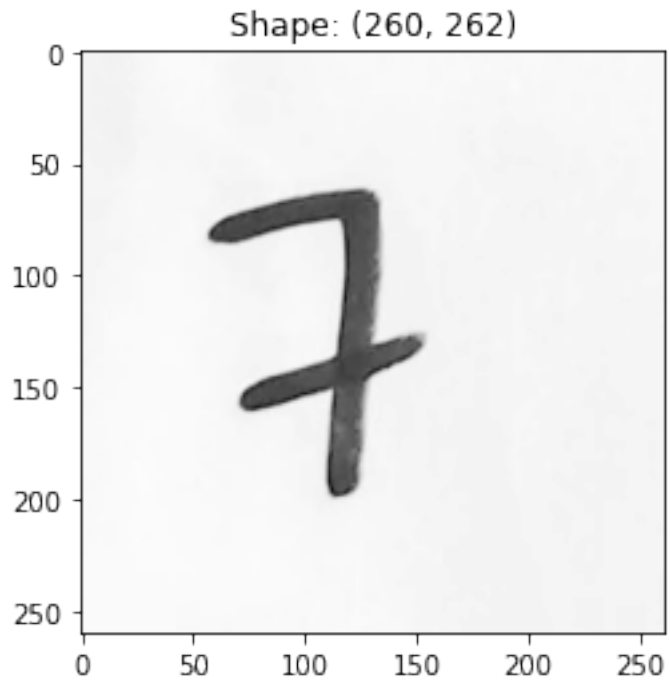
```
Saving input.png to input.png
User uploaded file "input.png" with length 24495 bytes
```

```
from PIL import Image
```

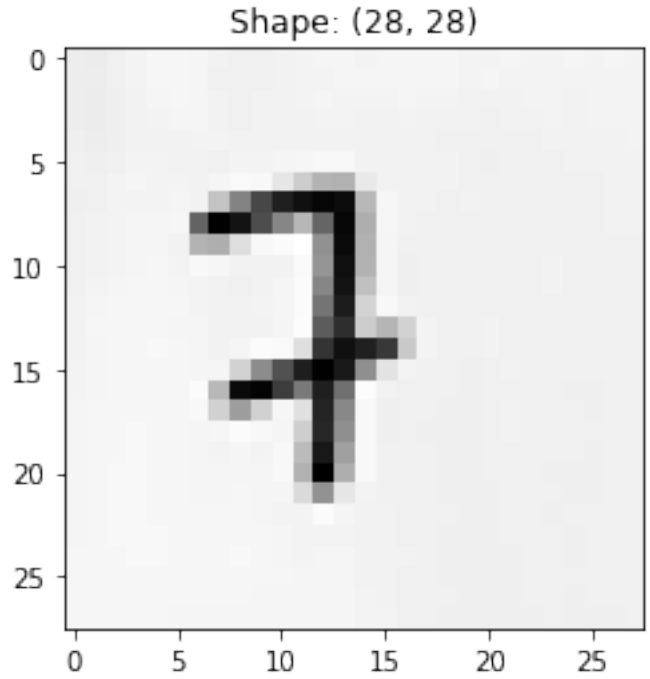
```
filename = 'input.png'

image = Image.open(filename)
p = plt.imshow(np.asarray(image), cmap=plt.cm.gray,);
p = plt.title('Shape: ' + str(np.asarray(image).shape))
```




```
# convert to grayscale image - 'L' format means each pixel is
# represented by a single value from 0 to 255
image_bw = image.convert('L')
p = plt.imshow(np.asarray(image_bw), cmap=plt.cm.gray,);
p = plt.title('Shape: ' + str(np.asarray(image_bw).shape))
```

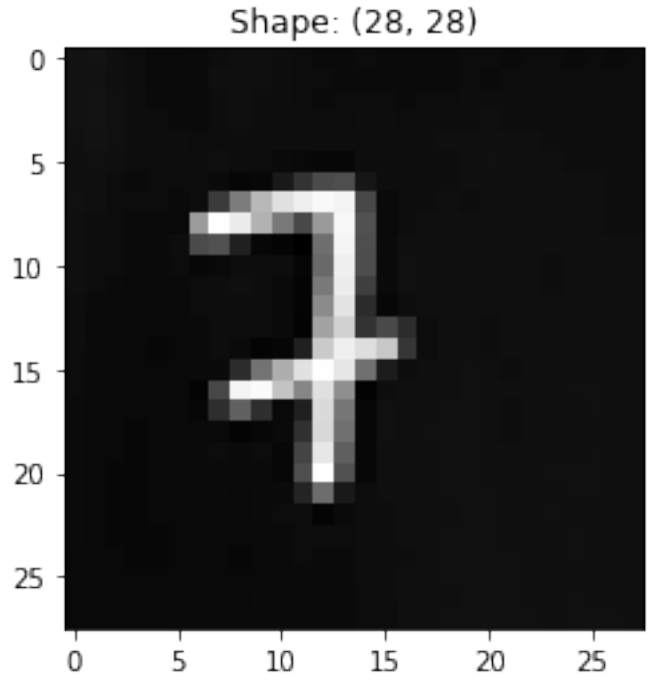



```
# resize image
image_bw_resized = image_bw.resize((28,28), Image.BICUBIC)
p = plt.imshow(np.asarray(image_bw_resized), cmap=plt.cm.gray,);
p = plt.title('Shape: ' + str(np.asarray(image_bw_resized).shape))
```

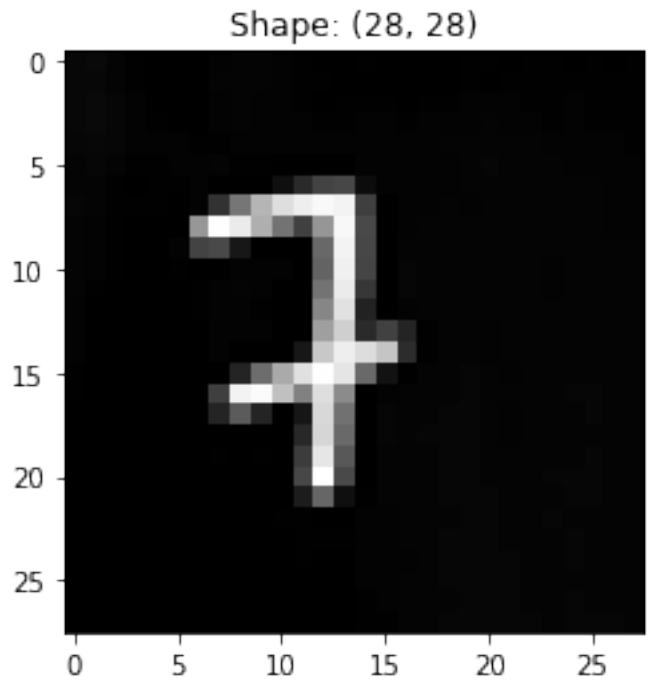



```
# invert image, to match training data
import PIL.ImageOps
```

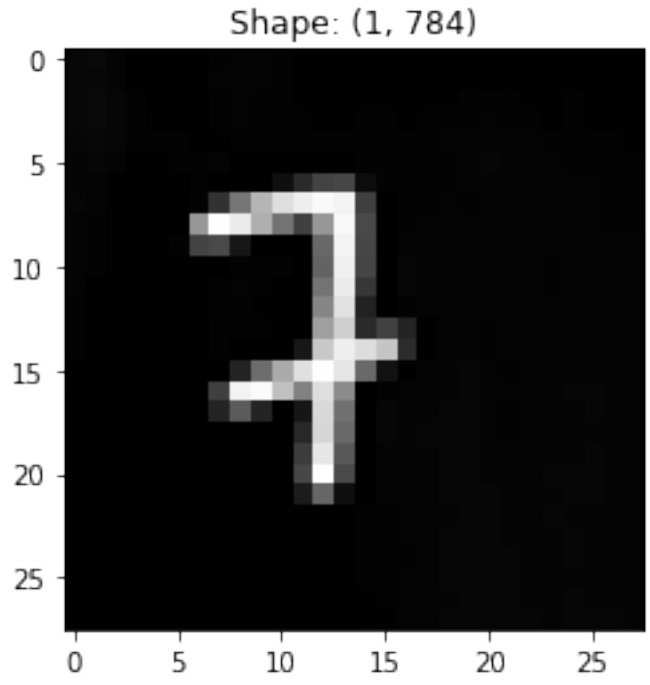
```
image_bw_resized_inverted = PIL.ImageOps.invert(image_bw_resized)
p = plt.imshow(np.asarray(image_bw_resized_inverted), cmap=plt.cm.gray,);
p = plt.title('Shape: ' + str(np.asarray(image_bw_resized_inverted).shape))
```




```
# adjust contrast and scale
pixel_filter = 20 # value from 0 to 100 - may need to adjust this manually
min_pixel = np.percentile(image_bw_resized_inverted, pixel_filter)
image_bw_resized_inverted_scaled = np.clip(image_bw_resized_inverted-min_pixel, 0, 255)
max_pixel = np.max(image_bw_resized_inverted)
image_bw_resized_inverted_scaled = np.asarray(image_bw_resized_inverted_scaled)/max_pixel
p = plt.imshow(np.asarray(image_bw_resized_inverted_scaled), cmap=plt.cm.gray,);
p = plt.title('Shape: ' + str(np.asarray(image_bw_resized_inverted_scaled).shape))
```



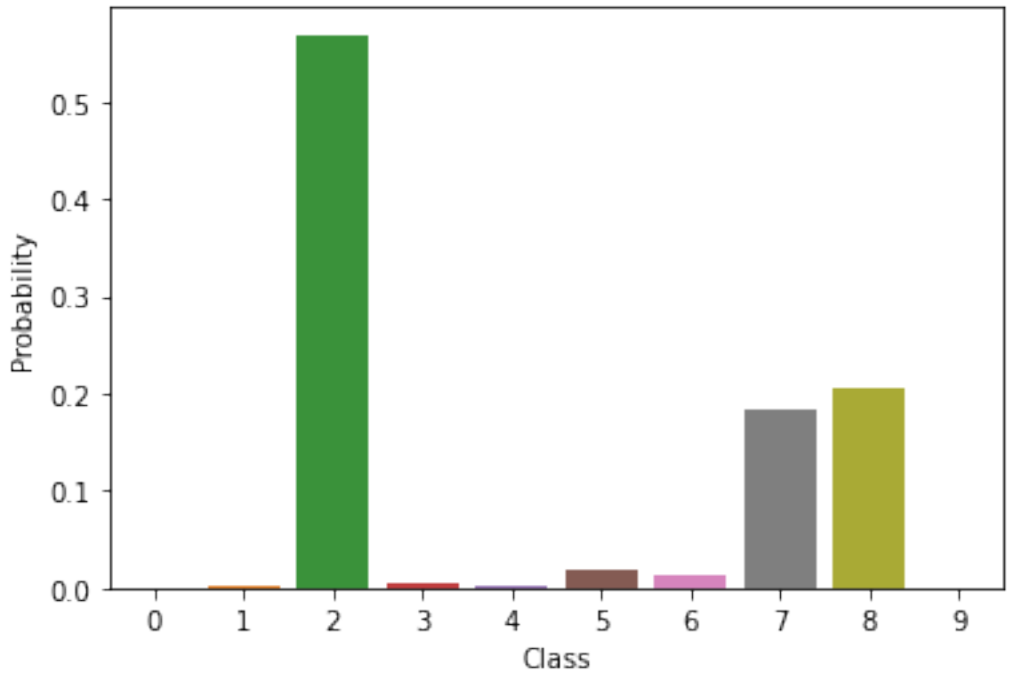

```
# finally, reshape to (1, 784) - 1 sample, 784 features
test_sample = np.array(image_bw_resized_inverted_scaled).reshape(1,784)
p = plt.imshow(np.reshape(test_sample, (28,28)), cmap=plt.cm.gray,);
p = plt.title('Shape: ' + str(test_sample.shape))
```



Now we can predict the class of this sample:

```
test_probs = model_fc.predict(test_sample)
```

```
sns.barplot(np.arange(0,10), test_probs.squeeze());
plt.ylabel("Probability");
plt.xlabel("Class");
```



Things to try

- What if we use a test sample where the image is not so well centered?

Background: Convolutional neural networks

The fully connected neural network was OK, but for images, there are two important reasons why we will often prefer a convolutional neural network instead:

- Dimension - images can have a huge number of pixels, and for image classification problems, we can also have a very large number of possible classes. A deep, fully connected network for these problems will have a *lot* of weights to learn.
- Images (and videos!) have a structure that is wasted on the fully connected network.
- Relevant features may be anywhere in the image. They may be rotated, translated, scaled...

The key idea behind convolutional neural networks is that a "neuron" is connected to a small part of image at a time (locally connected).

By having multiple locally connected neurons covering the entire image, we effectively "scan" the image.

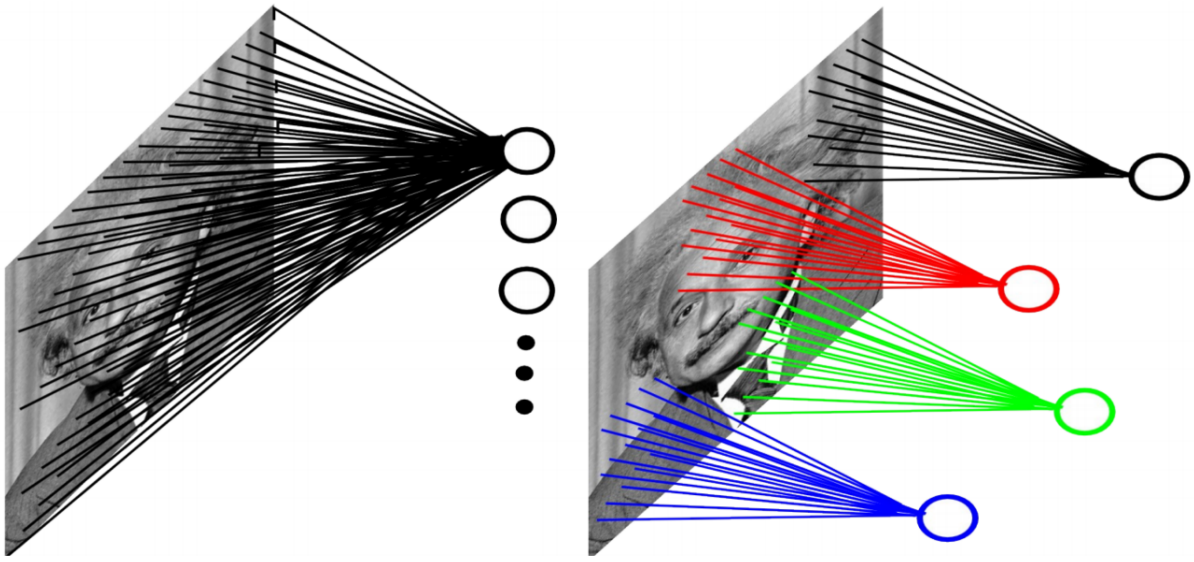


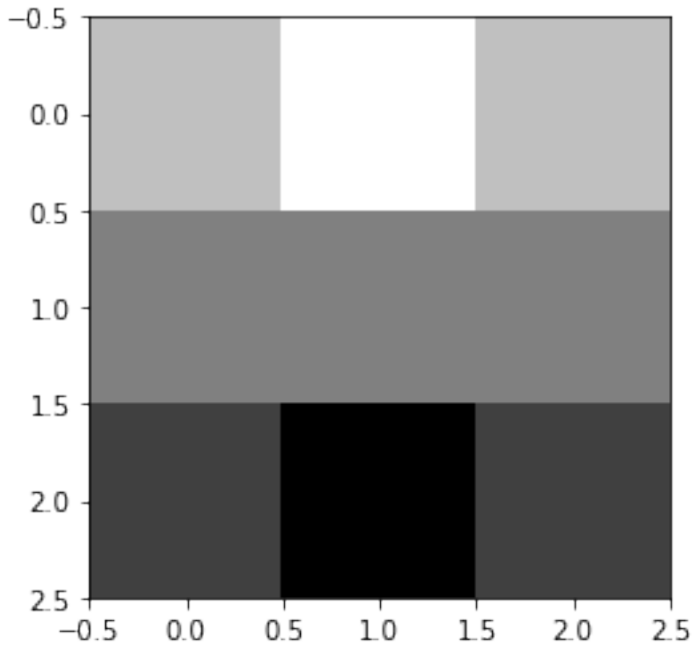
Image: 200x200 image. Fully connected network with 400,000 hidden units, 16 billion parameters. Locally connected network with 400,000 hidden units in 10x10 fields, 40 million parameters. Image credit: MA Ranzato.

What does convolution do? First, let's look at a numerical example:

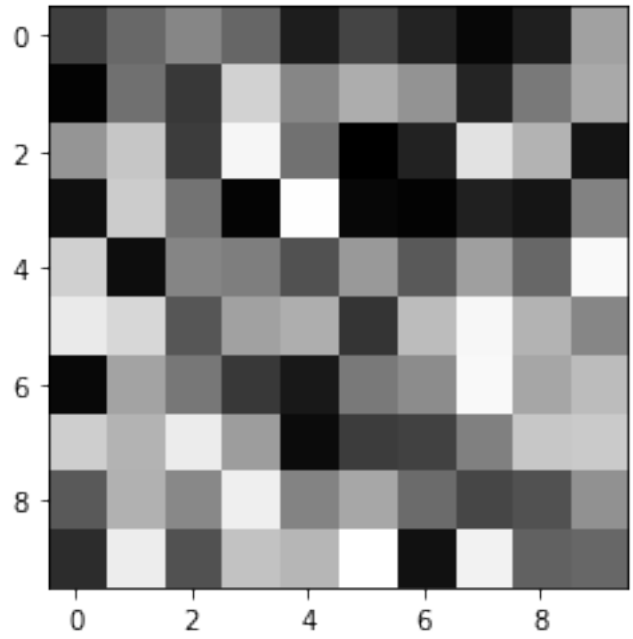
https://cs231n.github.io/assets/conv-demo/index.html

Now, let's look at a visual example.

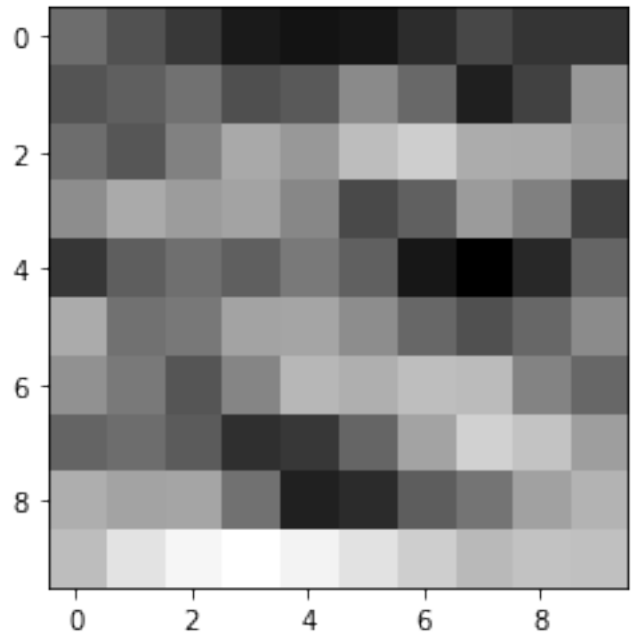
```
horizontal_sobel = np.array([[1,2,1],[0,0,0],[-1,-2,-1]])
plt.imshow(horizontal_sobel, cmap='gray');
```



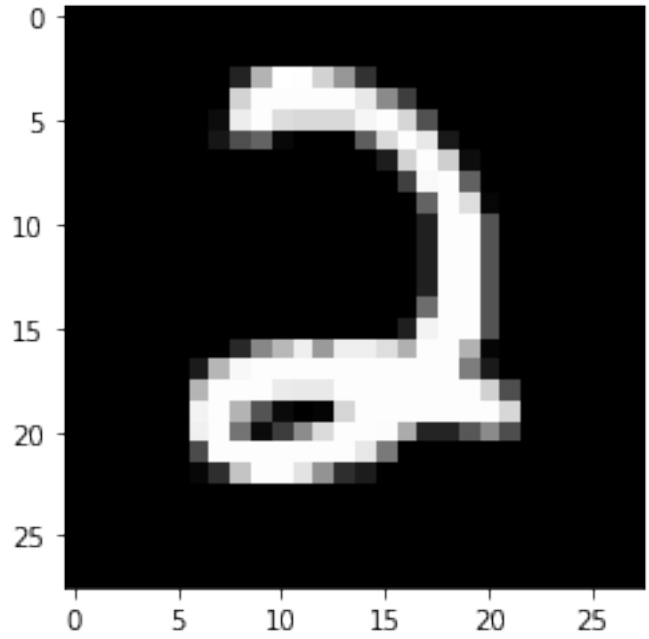

```
img = np.random.uniform(0,1,size=(10,10))
plt.imshow(img, cmap='gray');
```



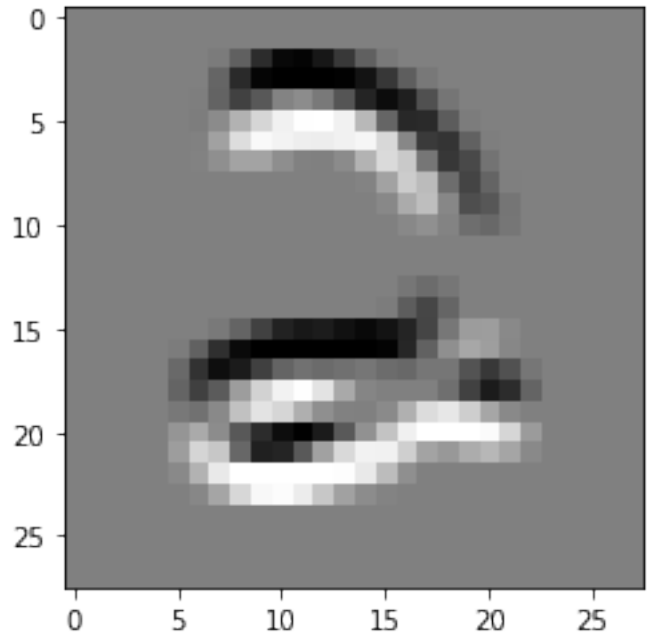

```
from scipy import signal
img_conv = signal.correlate2d(img, horizontal_sobel, mode='same')
plt.imshow(img_conv, cmap='gray');
```



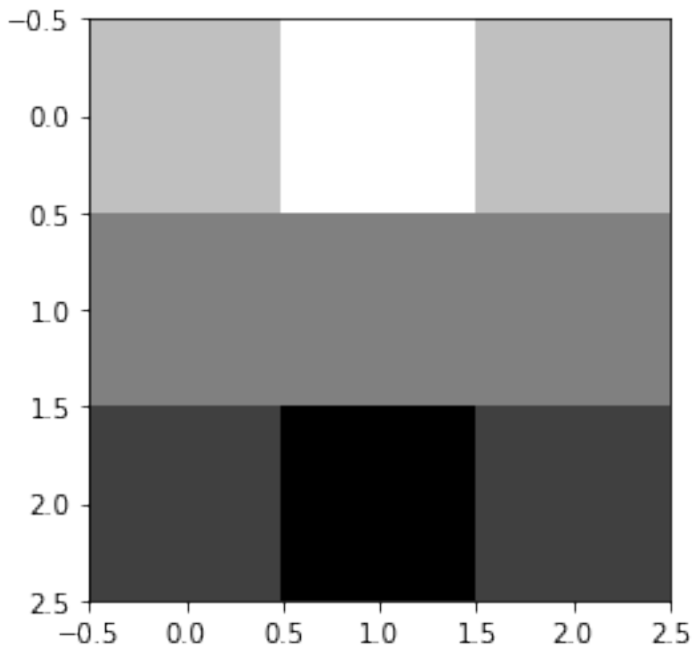

```
img_index = 3675
img = X_test[img_index]
plt.imshow(img.reshape(28,28), cmap='gray');
```



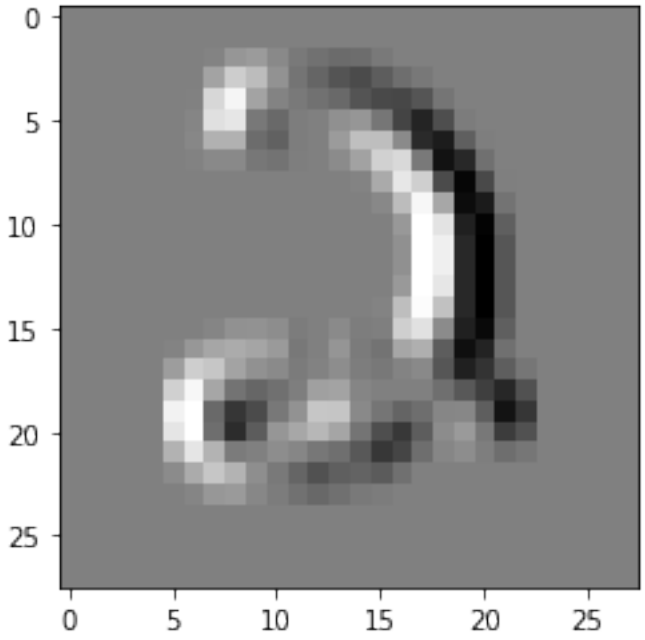

```
img_conv = signal.correlate2d(img.reshape(28,28), horizontal_sobel, mode='same')
plt.imshow(img_conv, cmap='gray');
```




```
vertical_sobel =  np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
plt.imshow(horizontal_sobel, cmap='gray');
```




```
img_conv = signal.correlate2d(img.reshape(28,28), vertical_sobel, mode='same')
plt.imshow(img_conv, cmap='gray');
```



A convolutional layer is like an array of these filters - each one ”sweeps” the image and looks for a different high-level ”feature”.

Convolutional layers are often followed by pooling layers. Here is a numerical example that shows how the two work together:

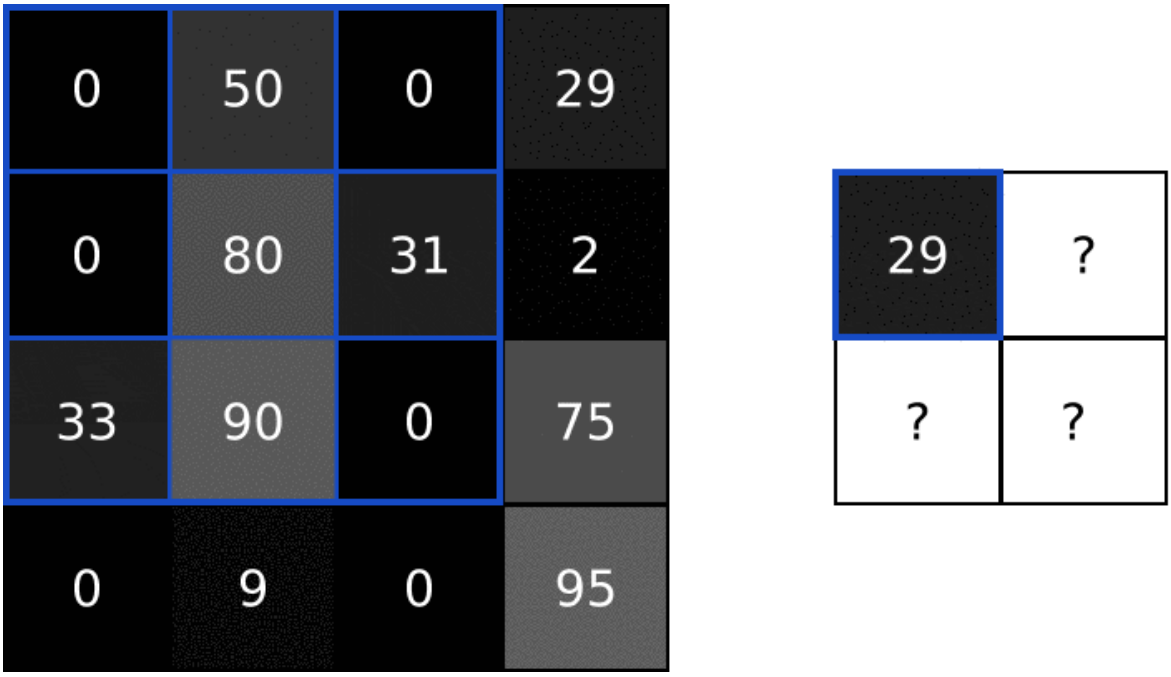


Image: Convolution. Via Victor Zhou https://victorzhou.com/blog/intro-to-cnns-part-1/.

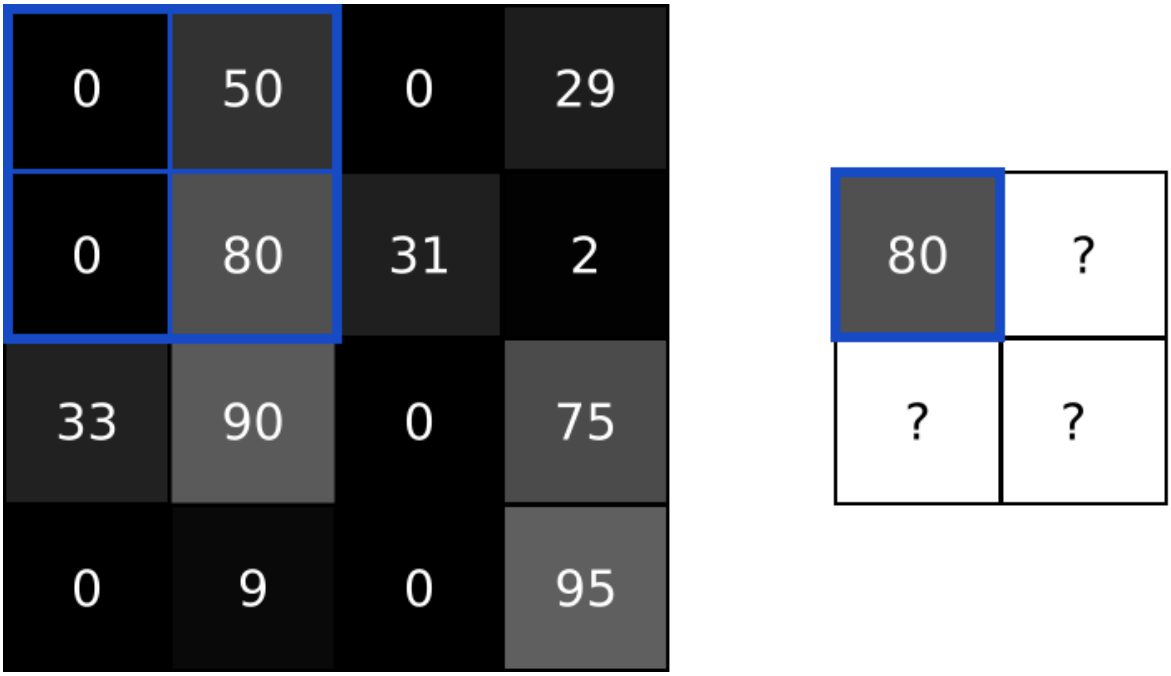


Image: Pooling. Via Victor Zhou https://victorzhou.com/blog/intro-to-cnns-part-1/.

Train a convolutional neural network on MNIST

Attribution: This section is based closely on this demo notebook by Daniel Moser.

In this next section, we will train a convolutional neural network. Also, we will try to improve performance using the following techniques:

- **Dropout layers**: Because deep networks can be prone to overfitting, we will also add *dropout* layers to our network architecture. In each training stage, a dropout layer will "zero" a random selection

of outputs (just for that stage). You can read more about this technique in this paper.
- **Batch normalization**: This technique re-scales and centers the data in the mini-batch when applied between layers.

First, we clear our session to make sure nothing is hanging around from previous models:

```
K.clear_session()
```

Then, we prepare our data. First, we reshape: the convolutional neural network requires each sample to have a 3D shape, including a depth - here, our image has only one color channel, so the depth is 1. We also scale and shift our data.

We separate part of the training data to use for model tuning. The accuracy on this validation set will be used to determine when to stop training the model.

```
# reshape input to a 28x28x1 volume
X_train_conv = X_train.reshape(X_train.shape[0], 28, 28, 1)
X_test_conv = X_test.reshape(X_test.shape[0], 28, 28, 1)

# scale
X_train_conv = 2*(X_train_conv/255 - 0.5)
X_test_conv = 2*(X_test_conv/255 - 0.5)

# convert string classes to integer equivalents
y_train = y_train.astype(np.int)
y_test  = y_test.astype(np.int)

# also add dimension to target
y_train_conv = y_train.reshape(-1,1)
y_test_conv = y_test.reshape(-1,1)

# split training set so we can use part of it for model tuning
X_train_conv, X_val_conv, y_train_conv, y_val_conv = train_test_split(X_train_conv,
    y_train_conv, test_size=1.0/6.0)

print("Training data shape", X_train_conv.shape)
print("Validation data shape", X_val_conv.shape)
print("Testing data shape", X_test_conv.shape)
```

```
Training data shape (50000, 28, 28, 1)
Validation data shape (10000, 28, 28, 1)
Testing data shape (10000, 28, 28, 1)
```

Next, we prepare our model with a sequence of `Conv2D`, `BatchNormalization`, `Activation`, `MaxPooling2D`, `Dropout`, and `Dense` layers.

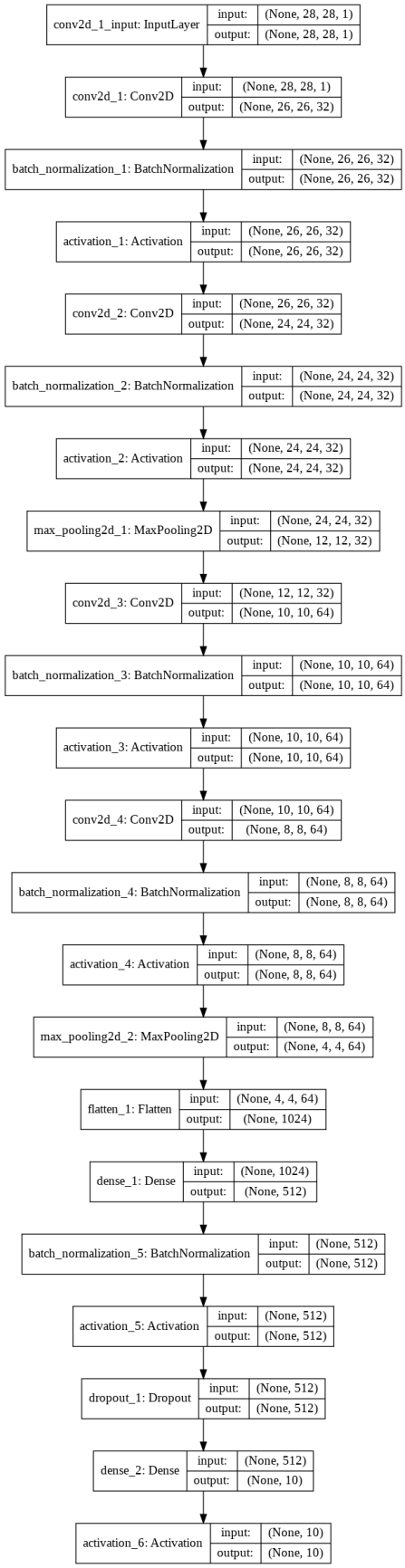
```
# Model parameters
n_filters = 32                                  # number of convolutional filters to use
pool_size = (2, 2)                              # size of pooling area for max pooling
kernel_size = (3, 3)                            # convolution kernel size
input_shape = (28, 28, 1)                       # input image volume
n_classes = 10                                  # number of classes

model_conv = Sequential()                             # Linear stacking of layers
```

```
# Convolution Layer 1
model_conv.add(Conv2D(32, (3, 3), input_shape=(28,28,1))) # 32 3x3 kernels
model_conv.add(BatchNormalization(axis=-1))               # normalize
convLayer01 = Activation('relu')                      # activation
model_conv.add(convLayer01)

# Convolution Layer 2
model_conv.add(Conv2D(32, (3, 3)))                        # 32 3x3 kernels
model_conv.add(BatchNormalization(axis=-1))               # normalize
model_conv.add(Activation('relu'))                        # activation
convLayer02 = MaxPooling2D(pool_size=(2,2))           # Pool the max values over a 2x2 kernel
model_conv.add(convLayer02)

# Convolution Layer 3
model_conv.add(Conv2D(64,(3, 3)))                         # 64 3x3 kernels
model_conv.add(BatchNormalization(axis=-1))               # normalize
convLayer03 = Activation('relu')                      # activation
model_conv.add(convLayer03)

# Convolution Layer 4
model_conv.add(Conv2D(64, (3, 3)))                        # 64 3x3 kernels
model_conv.add(BatchNormalization(axis=-1))               # normalize
model_conv.add(Activation('relu'))                        # activation
convLayer04 = MaxPooling2D(pool_size=(2,2))           # Pool the max values over a 2x2 kernel
model_conv.add(convLayer04)
model_conv.add(Flatten())                                 # Flatten final 4x4x64 output
    matrix into a 1024-length vector

# Fully Connected Layer 5
model_conv.add(Dense(512))                                # 512 fully connected nodes
model_conv.add(BatchNormalization())                      # normalization
model_conv.add(Activation('relu'))                        # activation

# Fully Connected Layer 6
model_conv.add(Dropout(0.2))                              # 20% dropout of randomly selected
    nodes
model_conv.add(Dense(10))                                 # final 10 fully connected nodes
model_conv.add(Activation('softmax'))                     # softmax activation

model_conv.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
batch_normalization_1 (Batch (None, 26, 26, 32)        128
_________________________________________________________________
activation_1 (Activation)    (None, 26, 26, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 32)        9248
_________________________________________________________________
batch_normalization_2 (Batch (None, 24, 24, 32)        128
```

```
_________________________________________________________________
activation_2 (Activation)    (None, 24, 24, 32)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 10, 10, 64)        18496
_________________________________________________________________
batch_normalization_3 (Batch (None, 10, 10, 64)        256
_________________________________________________________________
activation_3 (Activation)    (None, 10, 10, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 64)          36928
_________________________________________________________________
batch_normalization_4 (Batch (None, 8, 8, 64)          256
_________________________________________________________________
activation_4 (Activation)    (None, 8, 8, 64)          0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               524800
_________________________________________________________________
batch_normalization_5 (Batch (None, 512)               2048
_________________________________________________________________
activation_5 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_6 (Activation)    (None, 10)                0
=================================================================
Total params: 597,738
Trainable params: 596,330
Non-trainable params: 1,408
_________________________________________________________________
```

```
plot_model(model_conv, "mnist-convnet.png", show_shapes=True)
```


conv2d_1_input: InputLayer | input: (None, 28, 28, 1) | output: (None, 28, 28, 1)
conv2d_1: Conv2D | input: (None, 28, 28, 1) | output: (None, 26, 26, 32)
batch_normalization_1: BatchNormalization | input: (None, 26, 26, 32) | output: (None, 26, 26, 32)
activation_1: Activation | input: (None, 26, 26, 32) | output: (None, 26, 26, 32)
conv2d_2: Conv2D | input: (None, 26, 26, 32) | output: (None, 24, 24, 32)
batch_normalization_2: BatchNormalization | input: (None, 24, 24, 32) | output: (None, 24, 24, 32)
activation_2: Activation | input: (None, 24, 24, 32) | output: (None, 24, 24, 32)
max_pooling2d_1: MaxPooling2D | input: (None, 24, 24, 32) | output: (None, 12, 12, 32)
conv2d_3: Conv2D | input: (None, 12, 12, 32) | output: (None, 10, 10, 64)
batch_normalization_3: BatchNormalization | input: (None, 10, 10, 64) | output: (None, 10, 10, 64)
activation_3: Activation | input: (None, 10, 10, 64) | output: (None, 10, 10, 64)
conv2d_4: Conv2D | input: (None, 10, 10, 64) | output: (None, 8, 8, 64)
batch_normalization_4: BatchNormalization | input: (None, 8, 8, 64) | output: (None, 8, 8, 64)
activation_4: Activation | input: (None, 8, 8, 64) | output: (None, 8, 8, 64)
max_pooling2d_2: MaxPooling2D | input: (None, 8, 8, 64) | output: (None, 4, 4, 64)
flatten_1: Flatten | input: (None, 4, 4, 64) | output: (None, 1024)
dense_1: Dense | input: (None, 1024) | output: (None, 512)
batch_normalization_5: BatchNormalization | input: (None, 512) | output: (None, 512)
activation_5: Activation | input: (None, 512) | output: (None, 512)
dropout_1: Dropout | input: (None, 512) | output: (None, 512)
dense_2: Dense | input: (None, 512) | output: (None, 10)
activation_6: Activation | input: (None, 10) | output: (None, 10)

We will use the Adam optimizer again, and compile our model with `sparse_categorical_crossentropy` loss for backpropagation and `accuracy` for a scoring metric.

```
opt = optimizers.Adam(lr=0.005)
model_conv.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Next, we prepare our Early Stopping callback. We will stop training if 5 epochs pass without an improvement in the validation accuracy, and at that point we will restore the model with the best validation accuracy seen so far.

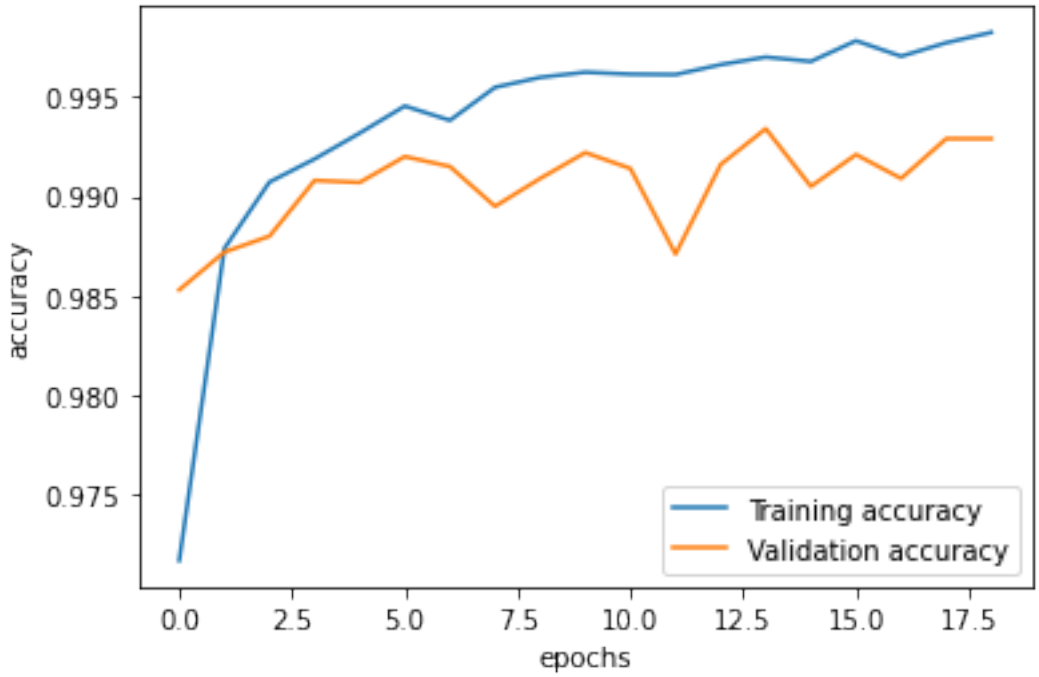
```
es = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', mode='max',
                                      patience=5,restore_best_weights=True )
```

```
%%time
# steps per epoch should be n_samples/batch_size
hist = model_conv.fit(X_train_conv, y_train_conv,
                          epochs = 20, batch_size=128,
                          validation_data=(X_val_conv, y_val_conv),
                          callbacks=[es])
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/20
50000/50000 [==============================] - 18s 366us/step - loss: 0.0926 - accuracy:
    0.9717 - val_loss: 0.0486 - val_accuracy: 0.9853
Epoch 2/20
50000/50000 [==============================] - 13s 261us/step - loss: 0.0397 - accuracy:
    0.9874 - val_loss: 0.0448 - val_accuracy: 0.9872
Epoch 3/20
50000/50000 [==============================] - 13s 259us/step - loss: 0.0290 - accuracy:
    0.9907 - val_loss: 0.0407 - val_accuracy: 0.9880
Epoch 4/20
50000/50000 [==============================] - 13s 261us/step - loss: 0.0244 - accuracy:
    0.9919 - val_loss: 0.0351 - val_accuracy: 0.9908
Epoch 5/20
50000/50000 [==============================] - 13s 263us/step - loss: 0.0212 - accuracy:
    0.9932 - val_loss: 0.0335 - val_accuracy: 0.9907
Epoch 6/20
50000/50000 [==============================] - 13s 260us/step - loss: 0.0171 - accuracy:
    0.9945 - val_loss: 0.0324 - val_accuracy: 0.9920
Epoch 7/20
50000/50000 [==============================] - 13s 259us/step - loss: 0.0182 - accuracy:
    0.9938 - val_loss: 0.0330 - val_accuracy: 0.9915
Epoch 8/20
50000/50000 [==============================] - 13s 259us/step - loss: 0.0139 - accuracy:
    0.9955 - val_loss: 0.0403 - val_accuracy: 0.9895
Epoch 9/20
50000/50000 [==============================] - 13s 257us/step - loss: 0.0129 - accuracy:
    0.9960 - val_loss: 0.0371 - val_accuracy: 0.9909
Epoch 10/20
50000/50000 [==============================] - 13s 259us/step - loss: 0.0106 - accuracy:
    0.9962 - val_loss: 0.0328 - val_accuracy: 0.9922
Epoch 11/20
```

```
50000/50000 [==============================] - 13s 258us/step - loss: 0.0117 - accuracy:
    0.9961 - val_loss: 0.0364 - val_accuracy: 0.9914
Epoch 12/20
50000/50000 [==============================] - 13s 258us/step - loss: 0.0121 - accuracy:
    0.9961 - val_loss: 0.0586 - val_accuracy: 0.9871
Epoch 13/20
50000/50000 [==============================] - 13s 257us/step - loss: 0.0097 - accuracy:
    0.9966 - val_loss: 0.0350 - val_accuracy: 0.9916
Epoch 14/20
50000/50000 [==============================] - 13s 260us/step - loss: 0.0094 - accuracy:
    0.9970 - val_loss: 0.0374 - val_accuracy: 0.9934
Epoch 15/20
50000/50000 [==============================] - 13s 257us/step - loss: 0.0095 - accuracy:
    0.9968 - val_loss: 0.0384 - val_accuracy: 0.9905
Epoch 16/20
50000/50000 [==============================] - 13s 258us/step - loss: 0.0069 - accuracy:
    0.9978 - val_loss: 0.0401 - val_accuracy: 0.9921
Epoch 17/20
50000/50000 [==============================] - 13s 261us/step - loss: 0.0101 - accuracy:
    0.9970 - val_loss: 0.0468 - val_accuracy: 0.9909
Epoch 18/20
50000/50000 [==============================] - 13s 258us/step - loss: 0.0070 - accuracy:
    0.9977 - val_loss: 0.0376 - val_accuracy: 0.9929
Epoch 19/20
50000/50000 [==============================] - 13s 258us/step - loss: 0.0060 - accuracy:
    0.9982 - val_loss: 0.0407 - val_accuracy: 0.9929
CPU times: user 3min 15s, sys: 50.7 s, total: 4min 5s
Wall time: 4min 13s
```

```
tr_accuracy = hist.history['accuracy']
val_accuracy = hist.history['val_accuracy']

plt.plot(tr_accuracy);
plt.plot(val_accuracy);
plt.xlabel('epochs');
plt.ylabel('accuracy');
plt.legend(['Training accuracy', 'Validation accuracy']);
```




```
%time y_pred_prob_conv = model_conv.predict(X_test_conv)
y_pred_conv = np.argmax(y_pred_prob_conv, axis=-1)
```

```
CPU times: user 1.62 s, sys: 171 ms, total: 1.8 s
Wall time: 1.71 s
```

```
score = model_conv.evaluate(X_test_conv, y_test)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [==============================] - 2s 165us/step
Test score: 0.029985896098984596
Test accuracy: 0.9930999875068665
```

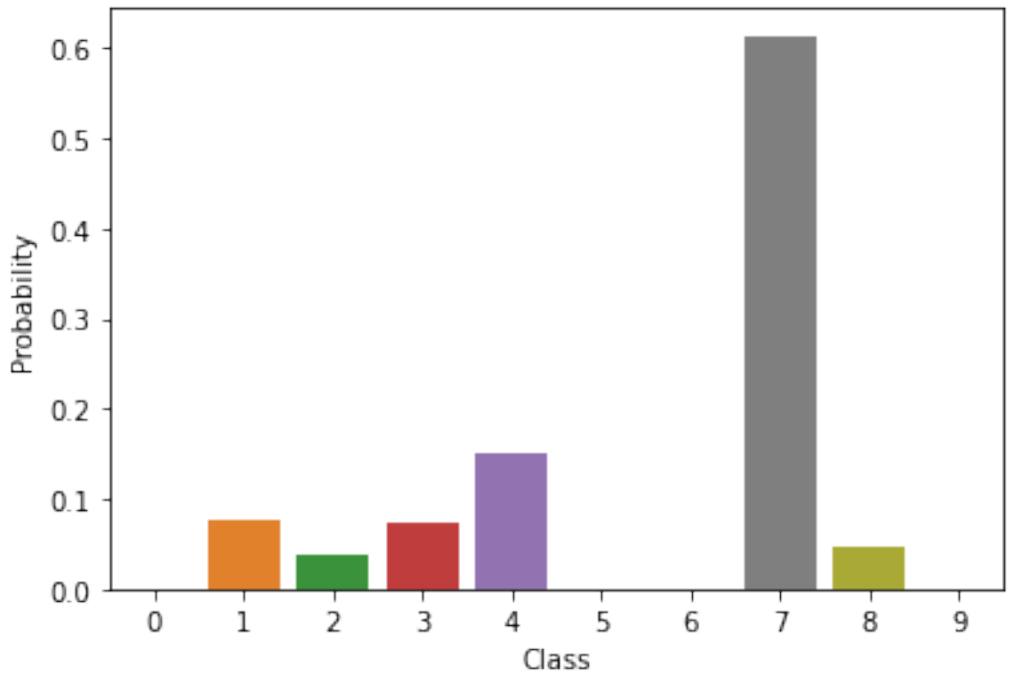
Try our convolutional neural network on our own test sample

We can use this convolutional neural network to predict the class of the test sample we uploaded previously.

```
test_sample_conv = test_sample.reshape(1, 28, 28, 1)
test_sample_conv = 2*(test_sample_conv - 0.5)
```

```
test_probs = model_conv.predict(test_sample_conv)
```

```
sns.barplot(np.arange(0,10), test_probs.squeeze());
plt.ylabel("Probability");
plt.xlabel("Class");
```

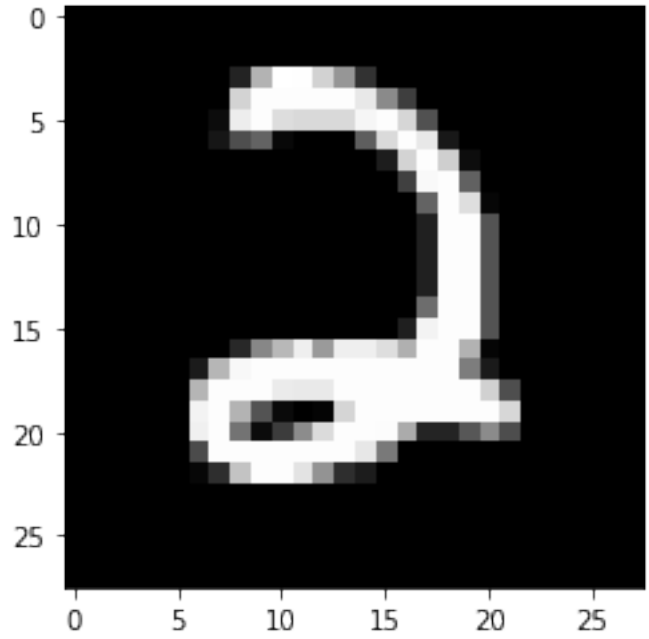


Looking at output of convolutional layers

Because deep learning is so complex, it can be difficult to understand why it makes the decisions it does. One way to better understand the behavior of a neural network is to visualize the output of each layer for a given input.

We will select one input to examine:

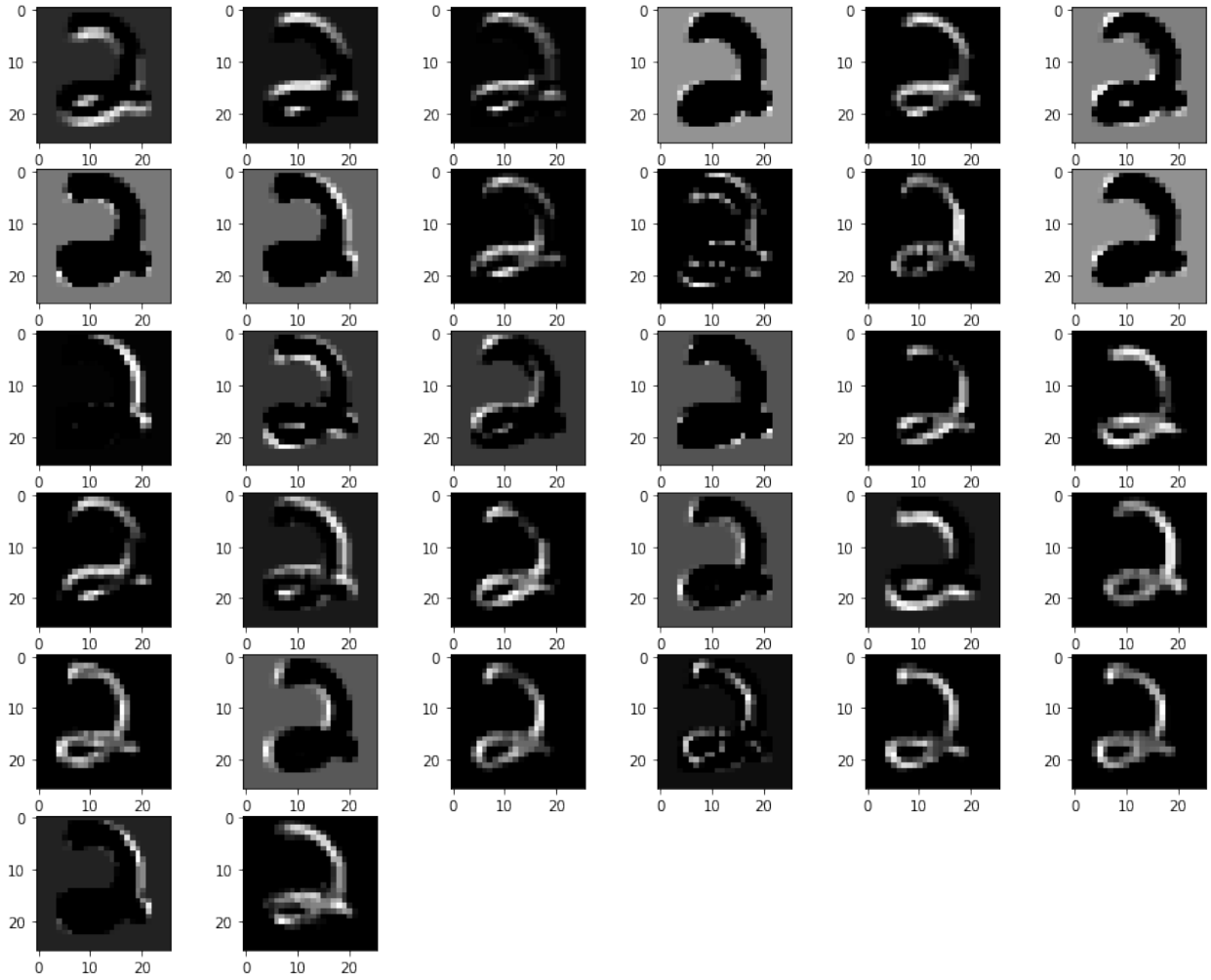
```
# choose an image to explore
img_index = 3675
img = X_test_conv[img_index]
# add an extra dimension to it so it is in 4D
img = img.reshape(1,28,28,1)
plt.figure();
plt.imshow(img.reshape(28,28), cmap='gray', interpolation='none');
```




```
# helper function to plot a convolution layer
def visualize(layer):
    inputs = [K.learning_phase()] + model_conv.inputs
    _convout1_f = K.function(inputs, [layer.output])

    def convout1_f(X):
        return _convout1_f([0] + [X])

    convolutions = convout1_f(img)
    convolutions = np.squeeze(convolutions)

    print ('Shape of conv:', convolutions.shape)

    m = convolutions.shape[2]
    n = int(np.ceil(np.sqrt(m)))

    # Visualization of each filter of the layer
    fig = plt.figure(figsize=(15,12))
    for i in range(m):
        ax = fig.add_subplot(n,n,i+1)
        ax.imshow(convolutions[:,:,i], cmap='gray')

# note: to use this function, we have to have imported the backend from keras
# from keras import backend as K
# and not from tensorflow.keras as
# import tensorflow.keras.backend as K
```

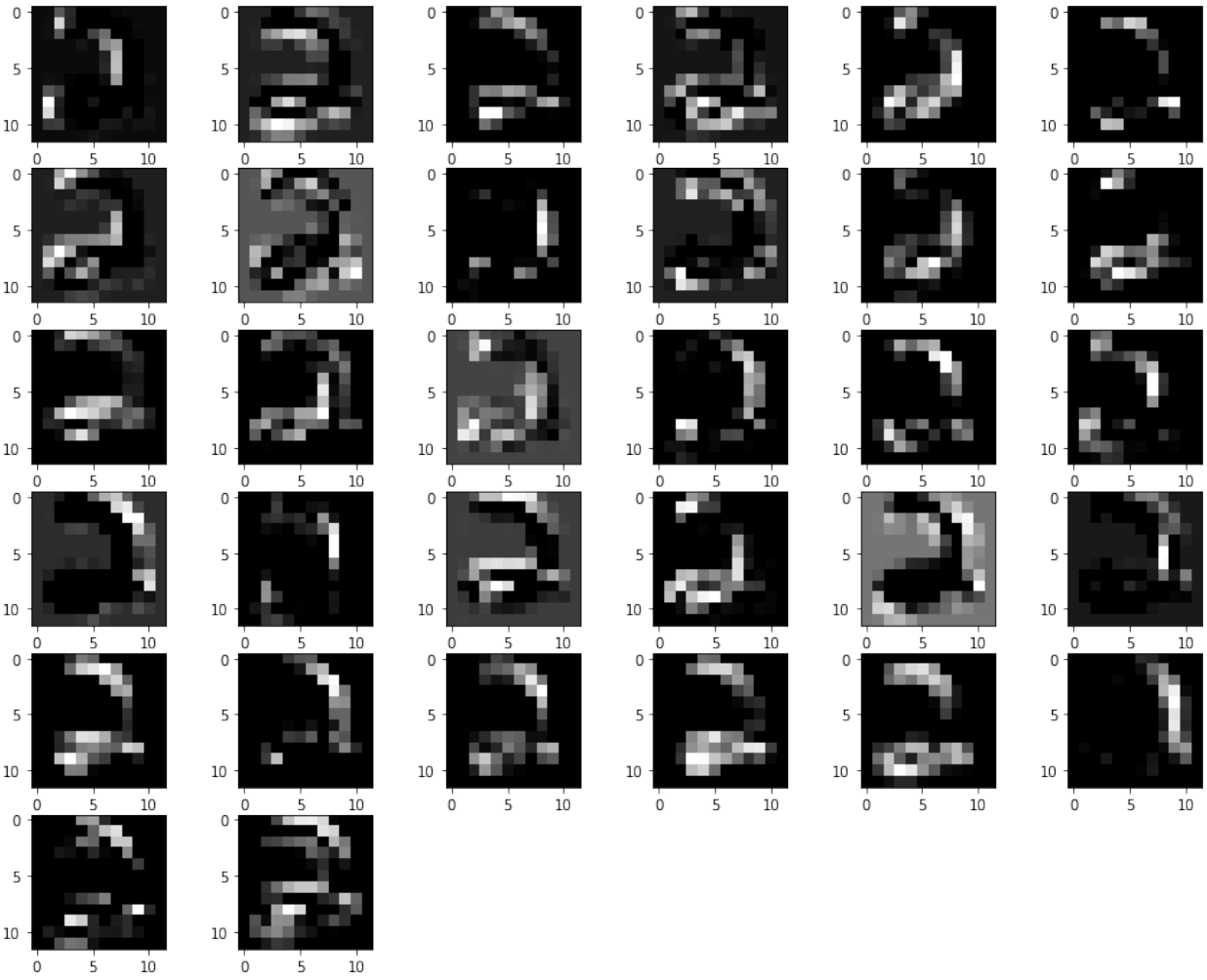
```
visualize(convLayer01)
```

```
Shape of conv: (26, 26, 32)
```



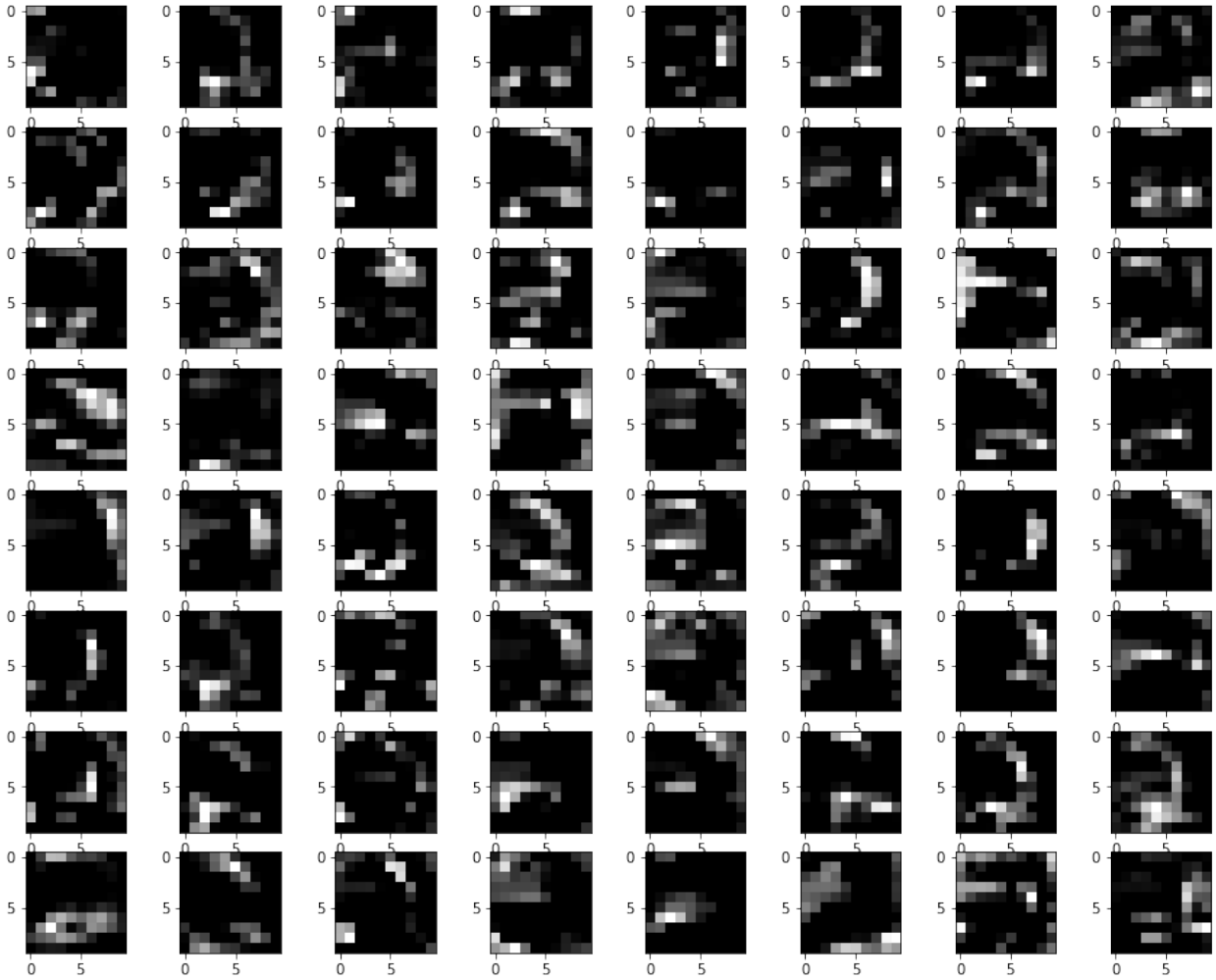
```
visualize(convLayer02)
```

```
Shape of conv: (12, 12, 32)
```



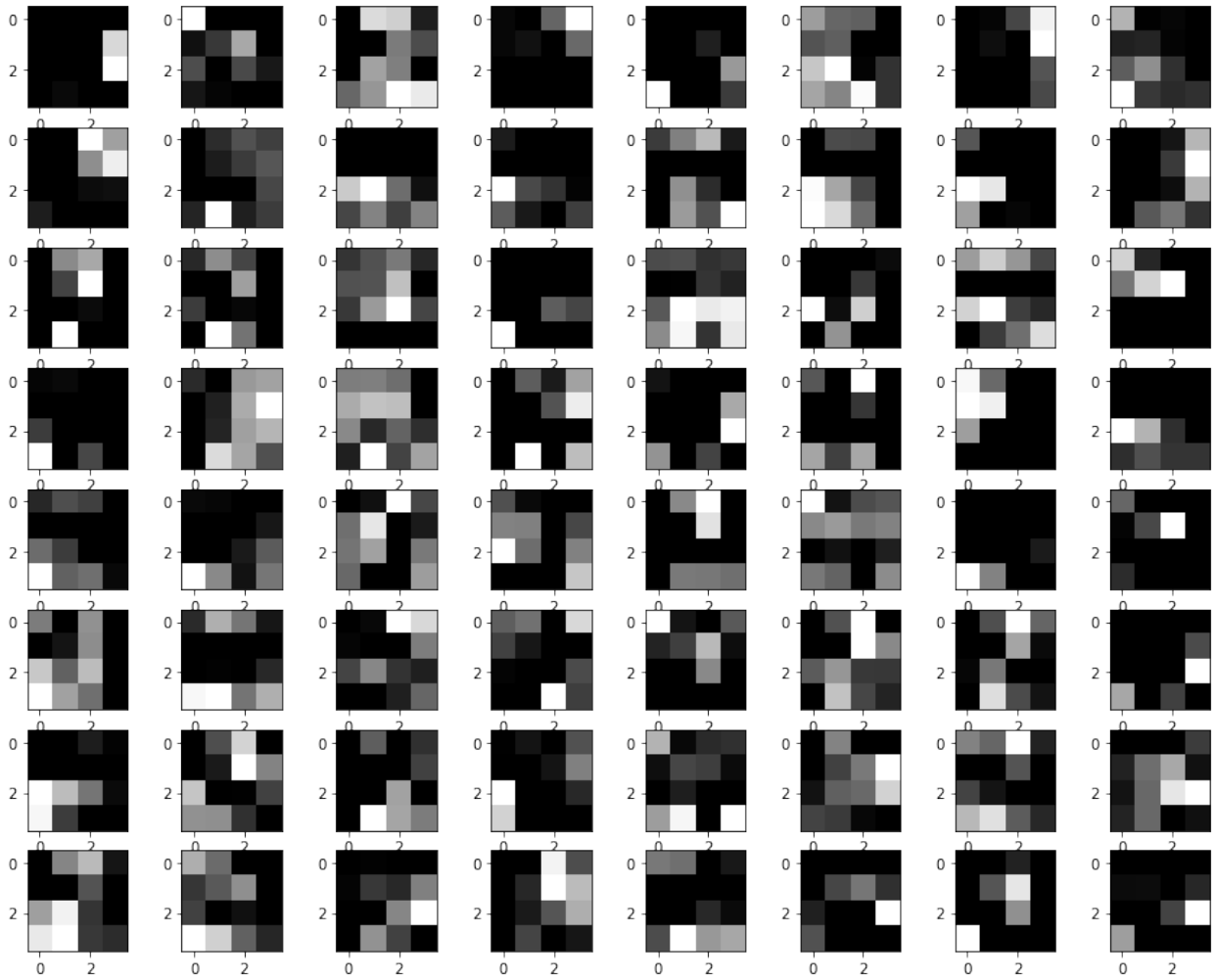
```
visualize(convLayer03)
```

```
Shape of conv: (10, 10, 64)
```



```
visualize(convLayer04)
```

```
Shape of conv: (4, 4, 64)
```



Generally, the convolutional layers close to the input capture small details, while those close to the output of the model capture more general features that are less sensitive to local variations in the input image. We can see this characteristic in the visualizations above.

Saving and restoring a model

Since this model took a long time to train, it may be useful to save the results, so that we can re-use the model later without having to re-train. We can save the model in an `hd5` file:

```
model_conv.save("mnist_conv_mod.h5")
```

```
/usr/local/lib/python3.6/dist-packages/keras/engine/saving.py:165: UserWarning: TensorFlow
    optimizers do not make it possible to access optimizer attributes or optimizer state
    after instantiation. As a result, we cannot save the optimizer as part of the model save
    file.You will have to compile your model again after loading it. Prefer using a Keras
    optimizer instead (see keras.io/optimizers).
  'TensorFlow optimizers do not '
```

Now, if you click on the folder icon in the menu on the left side of the Colab window, you can see this file in your workspace. You can download the file for later use.

To use the model again in the future, you can load it using `load_model`, then use it to make predictions without having to train it.

```
from tensorflow.keras.models import load_model

model2 = load_model("mnist_conv_mod.h5")
opt = optimizers.Adam(lr=0.005)
model2.compile(optimizer=opt,
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

# use saved model to predict new samples
y_pred_prob_conv2 = model2.predict(X_test_conv)
y_pred_conv2 = np.argmax(y_pred_prob_conv, axis=-1)
acc = accuracy_score(y_test, y_pred_conv2)
print("Accuracy of saved model on test set: %f" % acc)
```

```
WARNING:tensorflow:No training configuration found in save file: the model was *not*
    compiled. Compile it manually.
Accuracy of saved model on test set: 0.993100
```

With data augmentation

We can try one more way to improve the model performance:

- **Data augmentation**: To supply more training samples, we can provide slightly modified versions of training samples - for example, samples with a small rotation applied - on which to train the model.

```
K.clear_session()
```

```
# Model parameters
n_filters = 32                                  # number of convolutional filters to use
pool_size = (2, 2)                              # size of pooling area for max pooling
kernel_size = (3, 3)                            # convolution kernel size
input_shape = (28, 28, 1)                       # input image volume
n_classes = 10                                  # number of classes

model_aug = Sequential()                             # Linear stacking of layers

# Convolution Layer 1
model_aug.add(Conv2D(32, (3, 3), input_shape=(28,28,1))) # 32 3x3 kernels
model_aug.add(BatchNormalization(axis=-1))               # normalize
convLayer01 = Activation('relu')                    # activation
model_aug.add(convLayer01)

# Convolution Layer 2
model_aug.add(Conv2D(32, (3, 3)))                        # 32 3x3 kernels
model_aug.add(BatchNormalization(axis=-1))               # normalize
model_aug.add(Activation('relu'))                        # activation
convLayer02 = MaxPooling2D(pool_size=(2,2))         # Pool the max values over a 2x2 kernel
model_aug.add(convLayer02)

# Convolution Layer 3
model_aug.add(Conv2D(64,(3, 3)))                         # 64 3x3 kernels
```

```
model_aug.add(BatchNormalization(axis=-1))                    # normalize
convLayer03 = Activation('relu')                            # activation
model_aug.add(convLayer03)

# Convolution Layer 4
model_aug.add(Conv2D(64, (3, 3)))                             # 64 3x3 kernels
model_aug.add(BatchNormalization(axis=-1))                    # normalize
model_aug.add(Activation('relu'))                             # activation
convLayer04 = MaxPooling2D(pool_size=(2,2))                 # Pool the max values over a 2x2 kernel
model_aug.add(convLayer04)
model_aug.add(Flatten())                                      # Flatten final 4x4x64 output
    matrix into a 1024-length vector

# Fully Connected Layer 5
model_aug.add(Dense(512))                                     # 512 fully connected nodes
model_aug.add(BatchNormalization())                           # normalization
model_aug.add(Activation('relu'))                             # activation

# Fully Connected Layer 6
model_aug.add(Dropout(0.2))                                   # 20% dropout of randomly selected
    nodes
model_aug.add(Dense(10))                                      # final 10 fully connected nodes
model_aug.add(Activation('softmax'))                          # softmax activation

model_aug.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
batch_normalization_1 (Batch (None, 26, 26, 32)        128
_________________________________________________________________
activation_1 (Activation)    (None, 26, 26, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 32)        9248
_________________________________________________________________
batch_normalization_2 (Batch (None, 24, 24, 32)        128
_________________________________________________________________
activation_2 (Activation)    (None, 24, 24, 32)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 10, 10, 64)        18496
_________________________________________________________________
batch_normalization_3 (Batch (None, 10, 10, 64)        256
_________________________________________________________________
activation_3 (Activation)    (None, 10, 10, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 64)          36928
_________________________________________________________________
batch_normalization_4 (Batch (None, 8, 8, 64)          256
_________________________________________________________________
```

```
activation_4 (Activation)    (None, 8, 8, 64)          0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               524800
_________________________________________________________________
batch_normalization_5 (Batch (None, 512)               2048
_________________________________________________________________
activation_5 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_6 (Activation)    (None, 10)                0
=================================================================
Total params: 597,738
Trainable params: 596,330
Non-trainable params: 1,408
_________________________________________________________________
```

```
opt = optimizers.Adam(lr=0.005)
model_aug.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

In the following cell, we will use the `ImageDataGenerator` in `keras` for data augmentation. This function will generate versions of the training images that have some image effects applied: rotation, shift, shear, zoom.

```
from keras.preprocessing.image import ImageDataGenerator

train_gen = ImageDataGenerator(rotation_range=8, width_shift_range=0.08, shear_range=0.3,
                         height_shift_range=0.08, zoom_range=0.08)
train_generator = train_gen.flow(X_train_conv, y_train_conv, batch_size=128)

val_gen = ImageDataGenerator()
val_generator = val_gen.flow(X_val_conv, y_val_conv, batch_size=128)
```

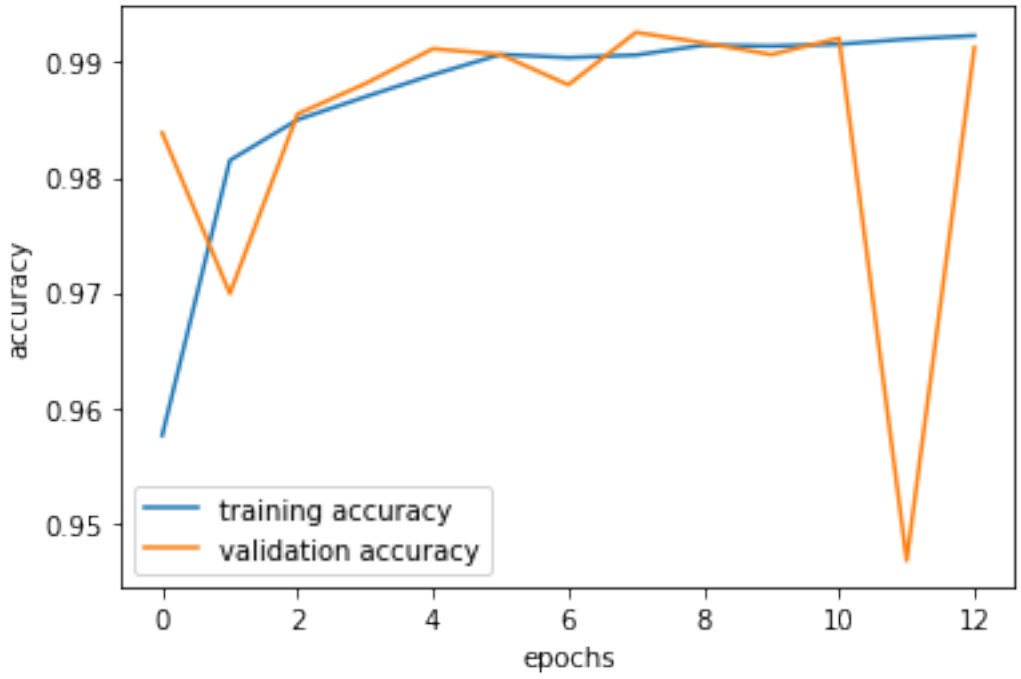
To train our model with data augmentation, we will use the `fit_generator` function, and specify the number of steps per epoch as the number of samples divided by the batch size.

```
%%time
# steps per epoch should be n_samples/batch_size
hist = model_aug.fit_generator(train_generator,
                           epochs = 20, steps_per_epoch=X_train_conv.shape[0]//128,
                           validation_data=val_generator,
                           validation_steps=X_val_conv.shape[0]//128,
                           callbacks=[es])
```

```
Epoch 1/20
```

```
390/390 [==============================] - 27s 69ms/step - loss: 0.1359 - accuracy: 0.9576 -
    val_loss: 0.0561 - val_accuracy: 0.9839
Epoch 2/20
390/390 [==============================] - 26s 66ms/step - loss: 0.0584 - accuracy: 0.9815 -
    val_loss: 0.1415 - val_accuracy: 0.9699
Epoch 3/20
390/390 [==============================] - 27s 68ms/step - loss: 0.0476 - accuracy: 0.9850 -
    val_loss: 0.0592 - val_accuracy: 0.9855
Epoch 4/20
390/390 [==============================] - 25s 64ms/step - loss: 0.0413 - accuracy: 0.9870 -
    val_loss: 0.0474 - val_accuracy: 0.9881
Epoch 5/20
390/390 [==============================] - 26s 66ms/step - loss: 0.0347 - accuracy: 0.9889 -
    val_loss: 0.0463 - val_accuracy: 0.9912
Epoch 6/20
390/390 [==============================] - 25s 65ms/step - loss: 0.0313 - accuracy: 0.9907 -
    val_loss: 0.0039 - val_accuracy: 0.9907
Epoch 7/20
390/390 [==============================] - 25s 65ms/step - loss: 0.0321 - accuracy: 0.9904 -
    val_loss: 0.0389 - val_accuracy: 0.9880
Epoch 8/20
390/390 [==============================] - 25s 65ms/step - loss: 0.0309 - accuracy: 0.9906 -
    val_loss: 0.0279 - val_accuracy: 0.9926
Epoch 9/20
390/390 [==============================] - 25s 65ms/step - loss: 0.0277 - accuracy: 0.9915 -
    val_loss: 0.0941 - val_accuracy: 0.9917
Epoch 10/20
390/390 [==============================] - 25s 64ms/step - loss: 0.0275 - accuracy: 0.9914 -
    val_loss: 1.5510e-04 - val_accuracy: 0.9907
Epoch 11/20
390/390 [==============================] - 24s 62ms/step - loss: 0.0262 - accuracy: 0.9916 -
    val_loss: 0.0601 - val_accuracy: 0.9921
Epoch 12/20
390/390 [==============================] - 25s 63ms/step - loss: 0.0247 - accuracy: 0.9920 -
    val_loss: 0.1184 - val_accuracy: 0.9467
Epoch 13/20
390/390 [==============================] - 25s 64ms/step - loss: 0.0243 - accuracy: 0.9923 -
    val_loss: 0.0024 - val_accuracy: 0.9913
CPU times: user 7min 23s, sys: 41.8 s, total: 8min 5s
Wall time: 5min 31s
```

```
tr_accuracy = hist.history['accuracy']
val_accuracy = hist.history['val_accuracy']

plt.plot(tr_accuracy);
plt.plot(val_accuracy);
plt.xlabel('epochs');
plt.ylabel('accuracy');
plt.legend(['training accuracy', 'validation accuracy']);
```




```
%time y_pred_prob_conv = model_aug.predict(X_test_conv)
y_pred_conv = np.argmax(y_pred_prob_conv, axis=-1)
```

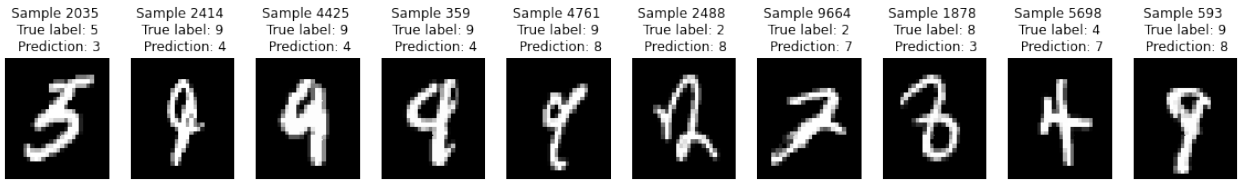
```
CPU times: user 1.47 s, sys: 126 ms, total: 1.59 s
Wall time: 1.59 s
```

```
score = model_aug.evaluate(X_test_conv, y_test)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [==============================] - 2s 194us/step
Test score: 0.02858658466657871
Test accuracy: 0.9912999868392944
```

These are some misclassified samples of this network:

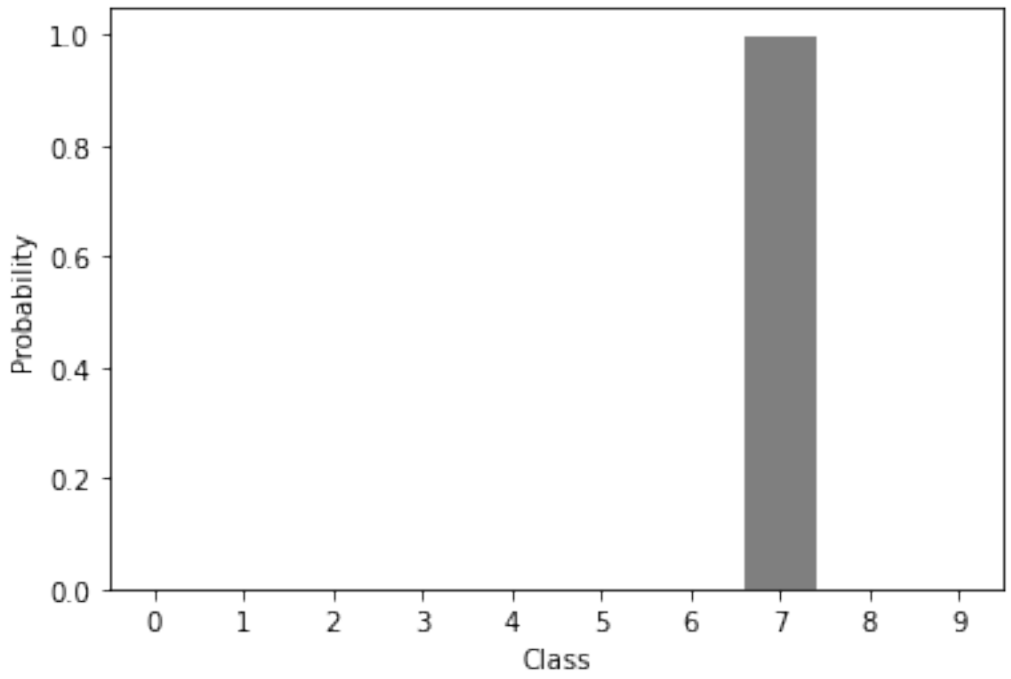
```
num_samples = 10
p = plt.figure(figsize=(num_samples*2,2))
idxs_mis = np.flatnonzero(y_test!=y_pred_conv)
idxs = np.random.choice(idxs_mis, num_samples, replace=False)
for i, idx in enumerate(idxs):
  p = plt.subplot(1, num_samples, i+1);
  p = sns.heatmap(X_test[idx].astype('uint8'), cmap=plt.cm.gray,
            xticklabels=False, yticklabels=False, cbar=False)
  p = plt.axis('off');
  p = plt.title("Sample %d \n True label: %d \n Prediction: %d" % (idx, y_test[idx],
      y_pred_conv[idx]));
plt.show()
```



Now, let's see its performance on our own test sample:

```
test_probs = model_aug.predict(test_sample_conv)
```

```
sns.barplot(np.arange(0,10), test_probs.squeeze());
plt.ylabel("Probability");
plt.xlabel("Class");
```



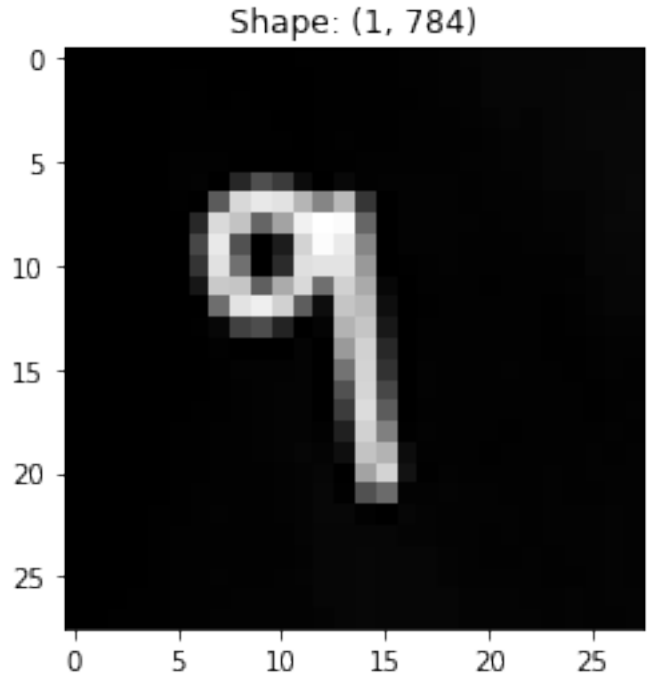
Try more of your own test samples!

```
from google.colab import files

uploaded = files.upload()

for fn in uploaded.keys():
  print('User uploaded file "{name}" with length {length} bytes'.format(
      name=fn, length=len(uploaded[fn])))
```

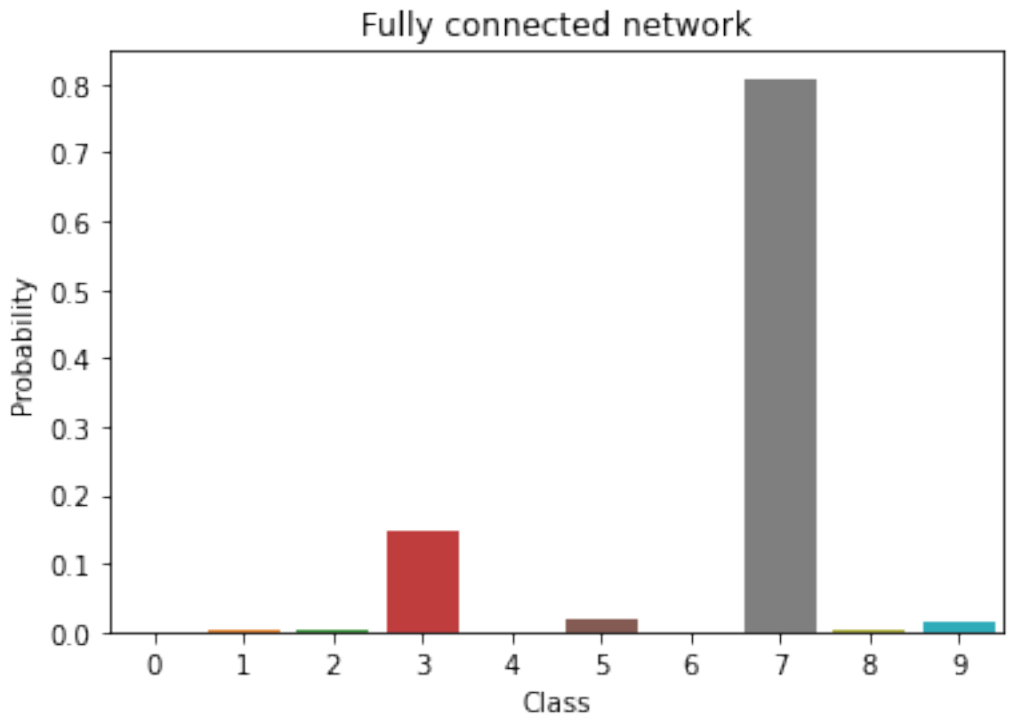
```
<IPython.core.display.HTML object>
```

```
Saving input.png to input.png
User uploaded file "input.png" with length 23665 bytes
```

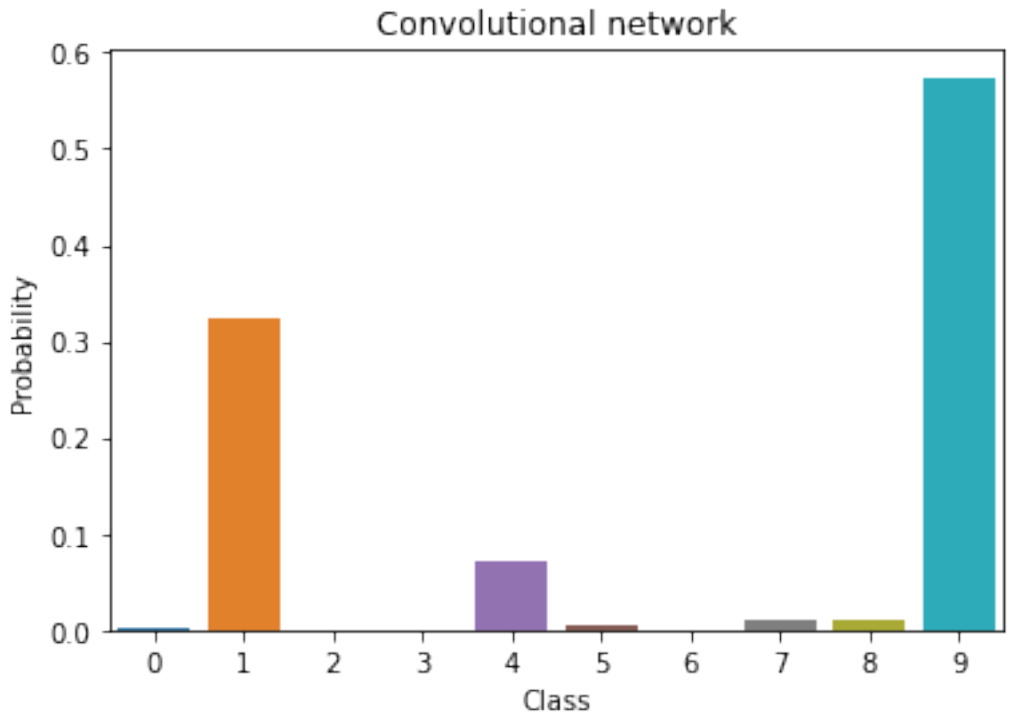
```
from PIL import Image

filename = 'input2.png'

image = Image.open(filename)
image_bw = image.convert('L')
image_bw_resized = image_bw.resize((28,28), Image.BICUBIC)
image_bw_resized_inverted = PIL.ImageOps.invert(image_bw_resized)
# adjust contrast and scale
min_pixel = np.percentile(image_bw_resized_inverted, pixel_filter)
image_bw_resized_inverted_scaled = np.clip(image_bw_resized_inverted-min_pixel, 0, 255)
max_pixel = np.max(image_bw_resized_inverted)
image_bw_resized_inverted_scaled = np.asarray(image_bw_resized_inverted_scaled)/max_pixel
test_sample = np.array(image_bw_resized_inverted_scaled).reshape(1,784)
test_sample_conv = test_sample.reshape(1, 28, 28, 1)
test_sample_conv = 2*(test_sample_conv - 0.5)
p = plt.imshow(np.reshape(test_sample, (28,28)), cmap=plt.cm.gray,);
p = plt.title('Shape: ' + str(test_sample.shape))
```



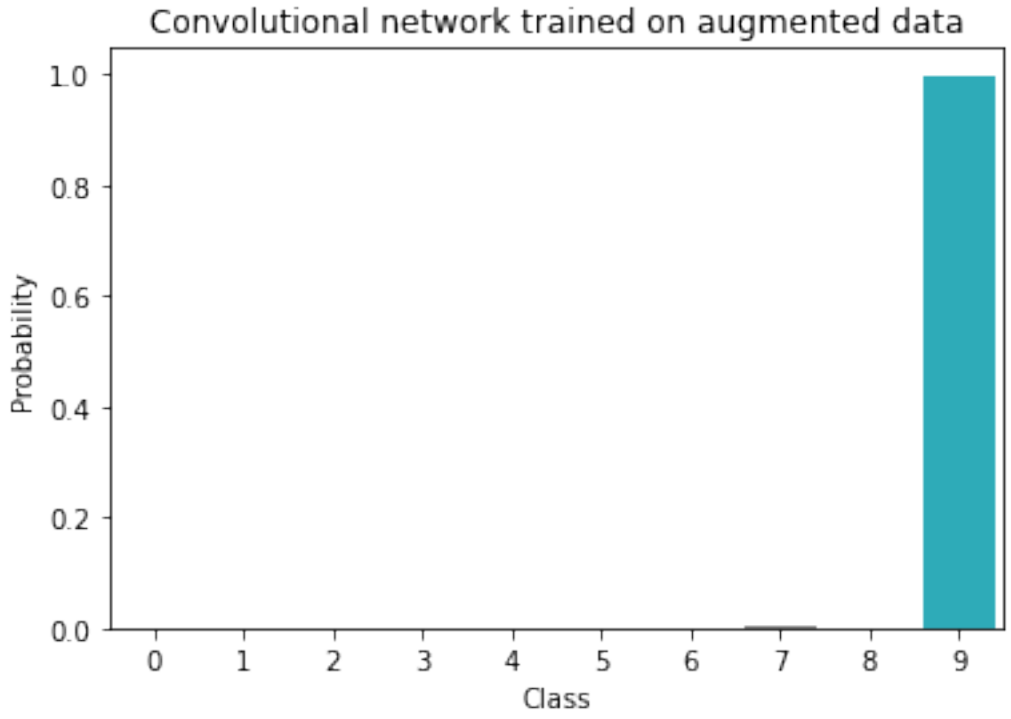

```
test_probs = model_fc.predict(test_sample)
sns.barplot(np.arange(0,10), test_probs.squeeze());
plt.ylabel("Probability");
plt.xlabel("Class");
plt.title("Fully connected network");
```




```
test_probs = model_conv.predict(test_sample_conv)
sns.barplot(np.arange(0,10), test_probs.squeeze());
plt.ylabel("Probability");
plt.xlabel("Class");
plt.title("Convolutional network");
```



```
test_probs = model_aug.predict(test_sample_conv)
sns.barplot(np.arange(0,10), test_probs.squeeze());
plt.ylabel("Probability");
plt.xlabel("Class");
plt.title("Convolutional network trained on augmented data");
```



Things to try

- This notebook runs using a free GPU on Colab! Try changing the runtime to CPU: Runtime > Change Runtime Type and change Hardware Accelerator to CPU. Then run the notebook again. How much speedup did you get with the GPU, relative to CPU?