CNN Architecture II

May 4, 2020

Chad Schupbach

The default backend engine for Keras (Tensorflow) uses CUDA, an API only supported by NVIDIA GPUs. We utilize PlaidML as Keras backend engine, which has Metal support for the current device GPU (AMD Radeon Pro 5300M). We reassign Keras backend engine as PlaidML in the following two code blocks.

[1]:
```
import os
```

[2]:
```
path = '/Users/chadschupbach/opt/anaconda3/'
os.environ['KERAS_BACKEND'] = 'plaidml.keras.backend'
os.environ['RUNFILES_DIR'] = path + 'share/plaidml'
os.environ['PLAIDML_NATIVE_PATH'] = path + 'lib/libplaidml.dylib'
```

[3]:
```
import numpy as np
import pandas as pd
import keras
from keras.layers import Conv2D
from keras.layers import MaxPooling2D, BatchNormalization
from keras.layers import Flatten, Dense, Dropout
from keras.models import Sequential
from keras.callbacks import ModelCheckpoint, EarlyStopping
from keras import backend as K
from sklearn.model_selection import train_test_split
from IPython.display import clear_output
import time
from src import utils
```

```
Using plaidml.keras.backend backend.
```

1 MNIST

The MNIST digits dataset is one of the most widely used datasets for high-dimensional classification. While recent advancements in deep learning have led many to conclude MNIST digits classification is a solved problem, we use it here as introduction to deep learning. Prior to these advancements, support-vector machine classification was considered to be the optimal approach to the MNIST digits problem; achieving a maximum testing accuracy around 97.8% [Zalando Research]. In our next notebook, we dive into deep learning using the dataset released primarily as a replacement to the MNIST digits dataset. One of the reasons the MNIST digits dataset is so popular is its wide availability as part of the Keras distribution. In addition, the dataset is large enough for meaningful deep learning application, while not being so large that it requires using GPU support.

1.1 Initialization

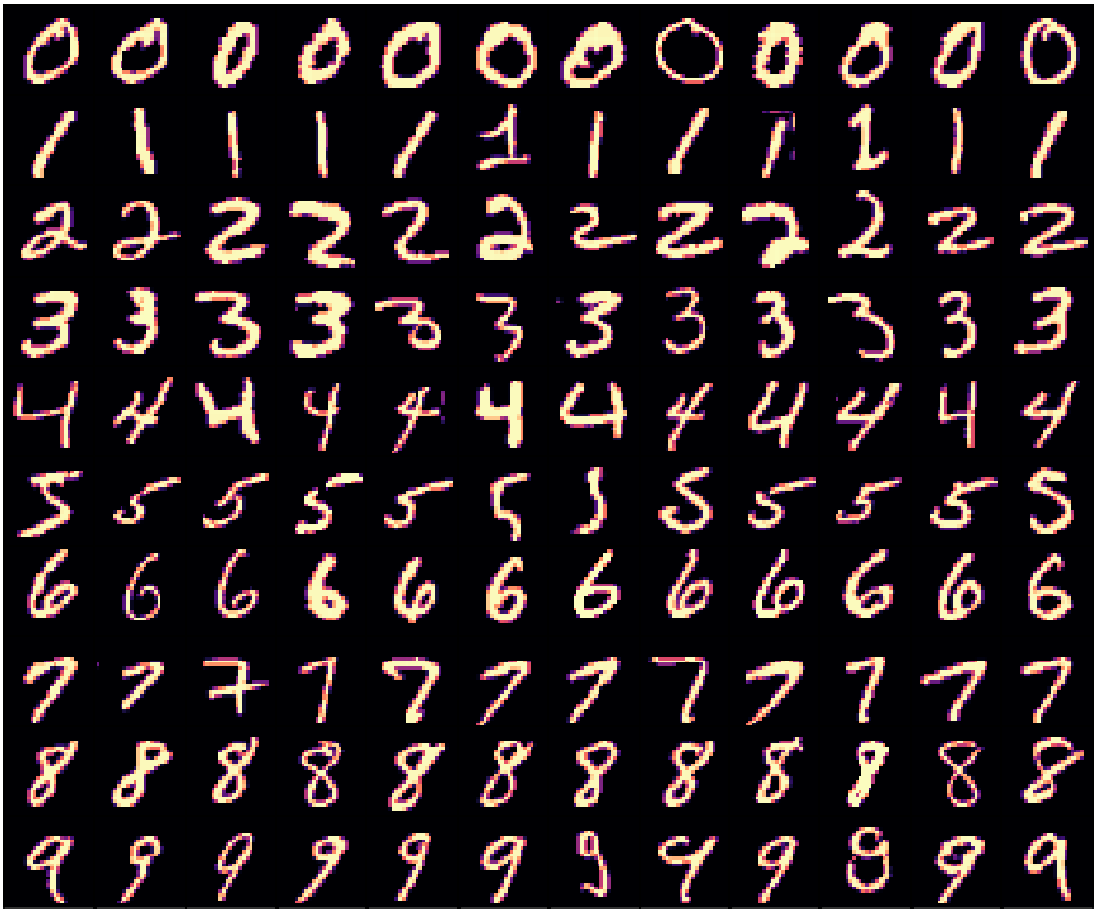
Load the entire MNIST digits dataset containing 60000 training images and 10000 testing images across 10 classes $\{0, 1, \ldots, 8, 9\}$.

[4]:
```
x_train, y_train, x_test, y_test, input_shape = utils.load_mnist()
```

```
INFO:plaidml:Opening device "metal_amd_radeon_pro_5300m.0"
```

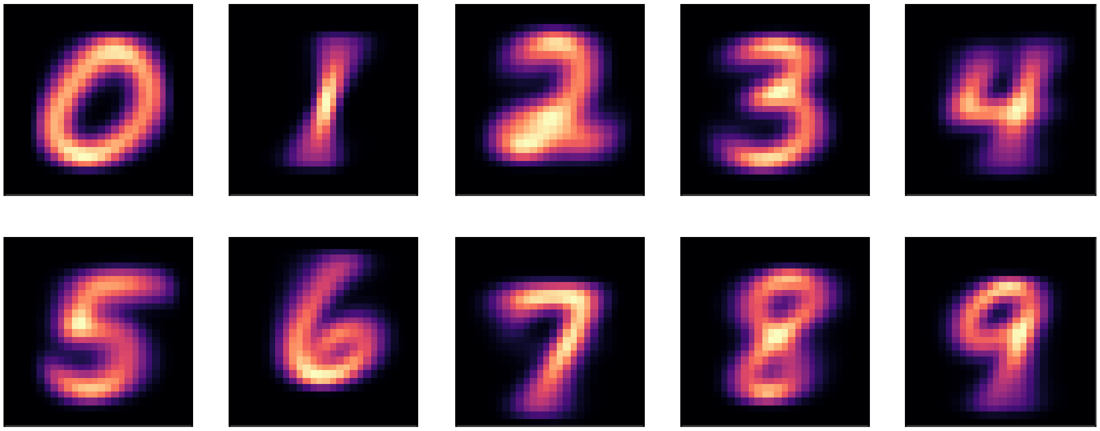
The first 12 samples from each class are shown below.

[5]:
```
utils.plot_samples(x_train, y_train)
```



We display the mean training image for each class as follows:

[6]:
```
utils.plot_class_means(x_train, y_train)
```



For the ensemble, we will train 10 models with the same architecture using a batch size of 128.

[7]:
```
n_classes = y_test.shape[-1]
n_models = 10
batch_size = 128
```

The architecture of each CNN model is as follows:

[8]:
```
model = [None] * n_models
for i in range(n_models):
    model[i] = Sequential()
    model[i].add(Conv2D(16, 3, padding='same', activation='relu',
                        input_shape=(28, 28, 1)))
    model[i].add(Conv2D(16, 3, padding='same', activation='relu'))
    model[i].add(MaxPooling2D(pool_size=(2, 2)))
    model[i].add(Conv2D(32, 3, padding='same', activation='relu'))
    model[i].add(Conv2D(32, 3, padding='same', activation='relu'))
    model[i].add(MaxPooling2D(pool_size=(2, 2)))
    model[i].add(Conv2D(64, 3, padding='same', activation='relu'))
    model[i].add(Conv2D(64, 3, padding='same', activation='relu'))
    model[i].add(Conv2D(64, 3, activation='relu'))
    model[i].add(Flatten())
    model[i].add(Dropout(0.25))
    model[i].add(Dense(batch_size, activation='relu'))
    model[i].add(Dropout(0.5))
    model[i].add(Dense(batch_size, activation='relu'))
    model[i].add(Dropout(0.5))
    model[i].add(Dense(n_classes, activation='softmax'))
    model[i].compile(optimizer='nadam', loss='categorical_crossentropy',
                     metrics=['accuracy'])
```

[9]:
```
model[0].summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 16)        160
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 28, 28, 16)        2320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 14, 14, 32)        4640
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 14, 14, 32)        9248
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 32)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 7, 7, 64)          18496
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 7, 7, 64)          36928
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 5, 5, 64)          36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 1600)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1600)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               204928
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
=================================================================
Total params: 331,450
Trainable params: 331,450
Non-trainable params: 0
_________________________________________________________________
```

We assign model checkpoints and early stopping criterion below. The checkpoints save parameter weights of the best training epoch based on validation accuracy to files in the `models/mnist/` directory. We also set early stopping criterion indicating convergence when no decrease in validation loss is observed over 7 epochs.

[10]:
```python
checkpoint = []
earlystop = []
for i in range(n_models):
    checkpoint += [ModelCheckpoint(filepath=f'models/mnist/best_weights_{i}.
 ↪hdf5',
                                    monitor='val_acc', save_best_only=True,
                                    save_weights_only=True, mode='max')]
    earlystop += [EarlyStopping(monitor='val_loss', patience=10, mode='min')]
```

If the early stopping criterion is not met, we end the training session of each model after 20 epochs. Because we need a validation set for the checkpoints and convergence criterion, we generate seeds (cell 11) to randomly split the training data into 50000 training samples and 10000 validation samples prior to each training session. It's not uncommon to see similar models implemented using the testing set for model validation in an effort to fully utilize the available training data. While it's tempting given that the validation set is not actually being used in training, using this strategy will result in model selection bias and should be avoided.

We fit each CNN model as follows:

[11]:
```python
start = time.time()
epochs = 25
ledger = []
for i in range(n_models):
    clear_output(wait=True)
    print(f'CNN_{i+1}\n' + '='*50)
    x, x_valid, y, y_valid = train_test_split(x_train, y_train,
                                              test_size=0.2)
    ledger += [model[i].fit(x, y, batch_size=batch_size, epochs=epochs,
                            verbose=1, callbacks=[checkpoint[i], earlystop[i]],
                            validation_data=(x_valid, y_valid))]
runtime = time.time() - start
```

```
CNN_10
==================================================
Train on 48000 samples, validate on 12000 samples
Epoch 1/25
48000/48000 [==============================] - 29s 611us/step - loss: 0.3849 -
acc: 0.8803 - val_loss: 0.0652 - val_acc: 0.9816
Epoch 2/25
48000/48000 [==============================] - 28s 592us/step - loss: 0.0976 -
acc: 0.9754 - val_loss: 0.0477 - val_acc: 0.9868
Epoch 3/25
48000/48000 [==============================] - 28s 590us/step - loss: 0.0688 -
acc: 0.9822 - val_loss: 0.0514 - val_acc: 0.9868
Epoch 4/25
48000/48000 [==============================] - 28s 589us/step - loss: 0.0592 -
acc: 0.9857 - val_loss: 0.0361 - val_acc: 0.9905
Epoch 5/25
```

```
48000/48000 [==============================] - 28s 588us/step - loss: 0.0501 -
acc: 0.9879 - val_loss: 0.0373 - val_acc: 0.9907
Epoch 6/25
48000/48000 [==============================] - 28s 581us/step - loss: 0.0444 -
acc: 0.9890 - val_loss: 0.0391 - val_acc: 0.9901
Epoch 7/25
48000/48000 [==============================] - 28s 592us/step - loss: 0.0392 -
acc: 0.9895 - val_loss: 0.0296 - val_acc: 0.9920
Epoch 8/25
48000/48000 [==============================] - 28s 588us/step - loss: 0.0379 -
acc: 0.9901 - val_loss: 0.0407 - val_acc: 0.9898
Epoch 9/25
48000/48000 [==============================] - 28s 582us/step - loss: 0.0399 -
acc: 0.9904 - val_loss: 0.0395 - val_acc: 0.9903
Epoch 10/25
48000/48000 [==============================] - 29s 597us/step - loss: 0.0405 -
acc: 0.9902 - val_loss: 0.0347 - val_acc: 0.9918
Epoch 11/25
48000/48000 [==============================] - 28s 592us/step - loss: 0.0346 -
acc: 0.9913 - val_loss: 0.0376 - val_acc: 0.9898
Epoch 12/25
48000/48000 [==============================] - 28s 590us/step - loss: 0.0290 -
acc: 0.9924 - val_loss: 0.0318 - val_acc: 0.9919
Epoch 13/25
48000/48000 [==============================] - 28s 579us/step - loss: 0.0285 -
acc: 0.9931 - val_loss: 0.0301 - val_acc: 0.9923
Epoch 14/25
48000/48000 [==============================] - 29s 594us/step - loss: 0.0291 -
acc: 0.9931 - val_loss: 0.0445 - val_acc: 0.9899
Epoch 15/25
48000/48000 [==============================] - 28s 592us/step - loss: 0.0292 -
acc: 0.9926 - val_loss: 0.0460 - val_acc: 0.9914
Epoch 16/25
48000/48000 [==============================] - 28s 588us/step - loss: 0.0272 -
acc: 0.9933 - val_loss: 0.0441 - val_acc: 0.9920
Epoch 17/25
48000/48000 [==============================] - 28s 591us/step - loss: 0.0292 -
acc: 0.9930 - val_loss: 0.0526 - val_acc: 0.9906
```

[12]:
```python
print('Total runtime: {:.2f} min'.format(runtime / 60))
```

```
Total runtime: 91.28 min
```

The training epochs of the final model are shown as output to cell 12. In addition, we observe a total runtime of just over 104 minutes for fitting all models. The training summary is given below.

[13]:
```python
utils.training_summary(model, ledger, x_test, y_test)
```

[13]:

	Epoch	Train Loss	Val Loss	Test Loss	Train Acc	Val Acc	Test Acc
Model 0	9.0	0.02905	0.03882	0.02953	0.9928	0.9911	0.9918
Model 1	13.0	0.02466	0.03280	0.03238	0.9938	0.9939	0.9932
Model 2	24.0	0.01968	0.03148	0.05521	0.9952	0.9931	0.9919
Model 3	10.0	0.02399	0.03629	0.02270	0.9944	0.9913	0.9938
Model 4	7.0	0.02560	0.03674	0.02665	0.9938	0.9912	0.9929
Model 5	19.0	0.02471	0.04275	0.03671	0.9940	0.9907	0.9927
Model 6	13.0	0.02305	0.03189	0.03093	0.9943	0.9927	0.9928
Model 7	10.0	0.02509	0.04039	0.02839	0.9936	0.9911	0.9932
Model 8	19.0	0.01759	0.03923	0.03865	0.9952	0.9915	0.9915
Model 9	13.0	0.02718	0.02960	0.03278	0.9933	0.9922	0.9924
Average	13.7	0.02406	0.03600	0.03339	0.9941	0.9919	0.9926

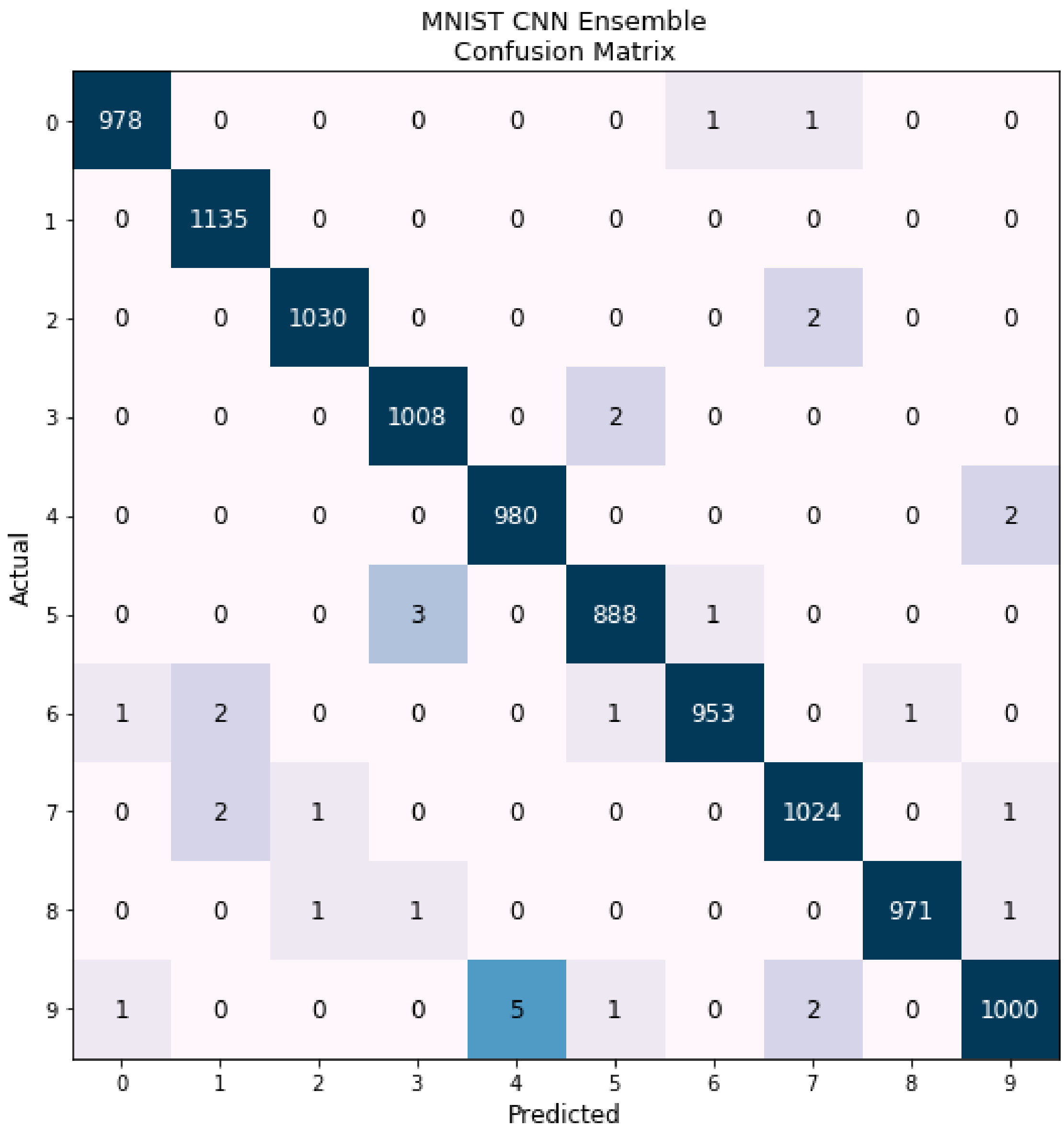
Taking a look at the average of all models above, we observe a mean training, validation, and testing accuracies of 94.08%, 92.37%, and 91.63%, respectively. The small decrease in testing accuracy relative to validation accuracy reveals a small amount of overfitting not accounted for by the three dropout layers. It's plausible that reducing model complexity with the addition of an L1 regularizer might help account for a fraction of the overfitting. We combine the results of all models to form the ensemble CNN as follows:

[14]:
```
test_act, test_pred, test_res = utils.ensemble_results(model, x_train, y_train,
                                                       x_test, y_test)
```

```
Ensemble Train Accuracy: 0.9991
Ensemble Test Accuracy: 0.9967
```

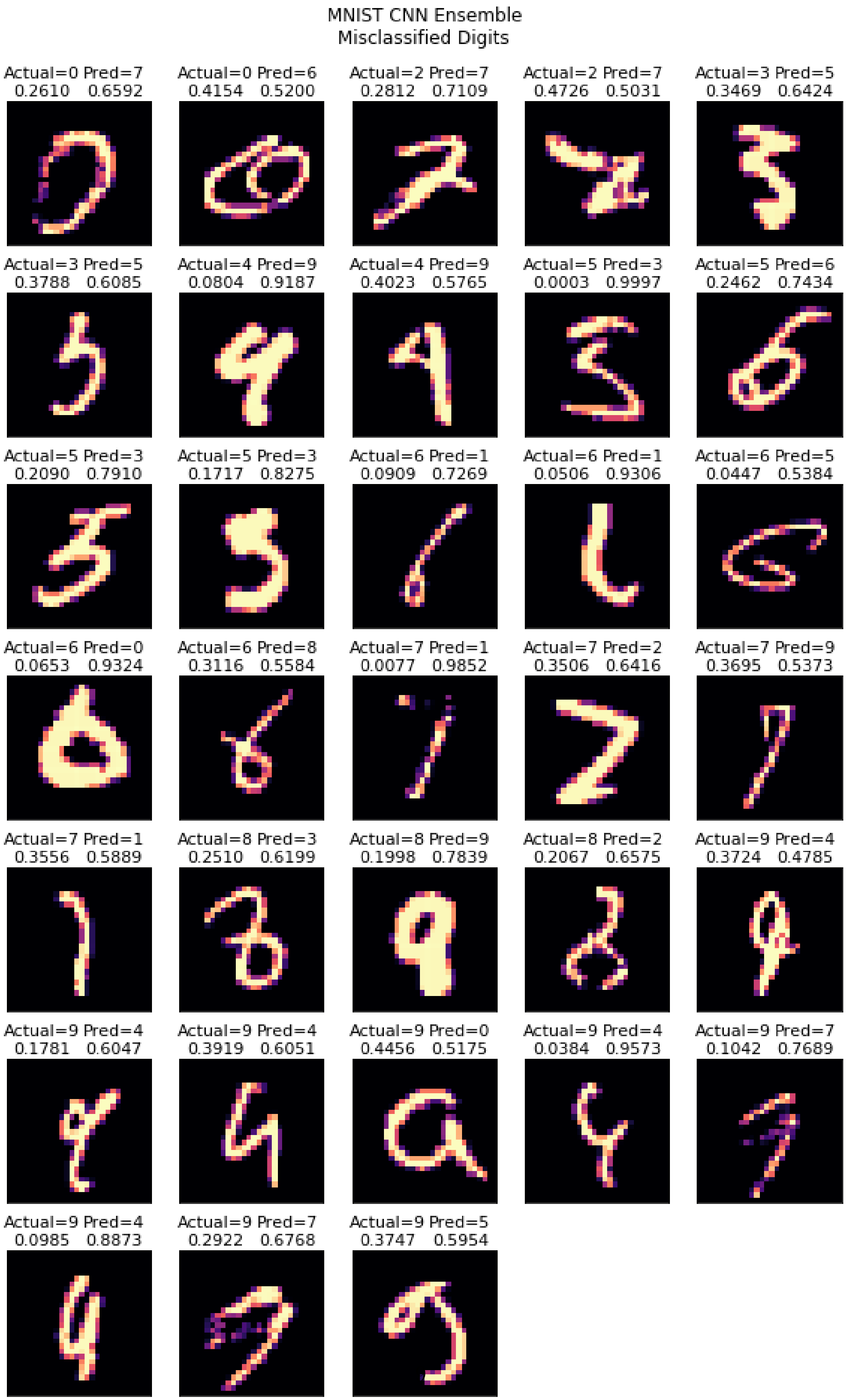
Shown above, the ensemble CNN results in a testing accuracy of 93.43%. We observe increases in testing accuracy of 1.80% and 1.49% relative to the average testing accuracy across all models (91.63%) and the model with the best testing accuracy (Model 4, 91.94%), respectively. We display the confusion matrix for the ensemble as follows:

[15]:
```
utils.plot_confusion(test_act, test_pred)
```



The confusion matrix above reveals poor performance of the model when classifying shirts, where items labelled `Shirt` are frequently misclassified as `T-shirt`, and vice versa. We also observe items labelled `Shirt` misclassified as `Pullover` and `Coat` at a relatively high rate. Some other item pairs that are difficult for the model to tell apart are `Sneaker`/`Boot` and `Coat`/`Pullover`. Below, we display the first 50 misclassified fashion items from the testing data with their respective actual and predicted labels. For each label, we also display the probability that the label is true according to the trained ensemble. Doing this provides some insight into how close the CNN ensemble was to predicting the labels of misclassified items correctly.

[16]:
```
utils.plot_misclassified(x_test, test_pred, test_act, test_res, _sort=True)
```


MNIST CNN Ensemble
Misclassified Digits
Actual=0 Pred=7
0.2610 0.6592
Actual=0 Pred=6
0.4154 0.5200
Actual=2 Pred=7
0.2812 0.7109
Actual=2 Pred=7
0.4726 0.5031
Actual=3 Pred=5
0.3469 0.6424
Actual=3 Pred=5
0.3788 0.6085
Actual=4 Pred=9
0.0804 0.9187
Actual=4 Pred=9
0.4023 0.5765
Actual=5 Pred=3
0.0003 0.9997
Actual=5 Pred=6
0.2462 0.7434
Actual=5 Pred=3
0.2090 0.7910
Actual=5 Pred=3
0.1717 0.8275
Actual=6 Pred=1
0.0909 0.7269
Actual=6 Pred=1
0.0506 0.9306
Actual=6 Pred=5
0.0447 0.5384
Actual=6 Pred=0
0.0653 0.9324
Actual=6 Pred=8
0.3116 0.5584
Actual=7 Pred=1
0.0077 0.9852
Actual=7 Pred=2
0.3506 0.6416
Actual=7 Pred=9
0.3695 0.5373
Actual=7 Pred=1
0.3556 0.5889
Actual=8 Pred=3
0.2510 0.6199
Actual=8 Pred=9
0.1998 0.7839
Actual=8 Pred=2
0.2067 0.6575
Actual=9 Pred=4
0.3724 0.4785
Actual=9 Pred=4
0.1781 0.6047
Actual=9 Pred=4
0.3919 0.6051
Actual=9 Pred=0
0.4456 0.5175
Actual=9 Pred=4
0.0384 0.9573
Actual=9 Pred=7
0.1042 0.7689
Actual=9 Pred=4
0.0985 0.8873
Actual=9 Pred=7
0.2922 0.6768
Actual=9 Pred=5
0.3747 0.5954