Instituto Superior Técnico

Aprendizagem Automática

Laboratory 3 - Neural Networks

Authors:
Diogo Moura - nº 86976
Diogo Alves - nº 86980

Lab Shift:
Tuesday 17h-18h30m

16th of November of 2019

Theoretical Introduction

Multi Layer Perceptron

A multilayer perceptron (MLP) is a class of feedforward artificial neural network (ANN).
MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers can distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable, since it can use non-linear activation functions.

Each unit i is connected to a unit j of the next layer trough a weight w_{ij}.

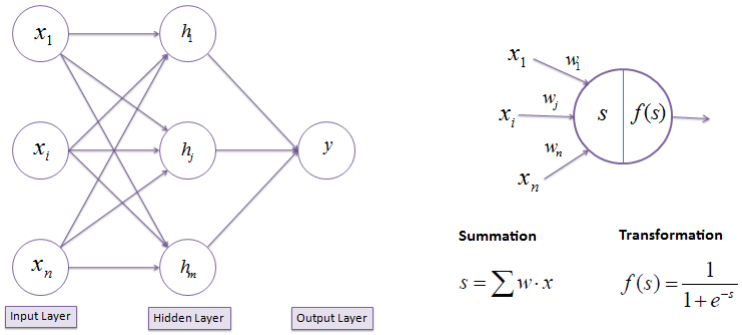

Figure 1: Single Perceptron with logistic activation function (on the right) and Multi Layer Perceptron (on the left)

Output $z(j)$ for input layers with activation function g(s):

$$s_j = w_{oj} + \sum_{j \in input} w_{ij} x_j \tag{1}$$

$$z(j) = g(s_j) \tag{2}$$

Output $z(j)$ for output and hidden layers with activation function g(s):

$$s_j = w_{oj} + \sum_{j \in previouslayer} w_{ij} z_i \tag{3}$$

$$z(j) = g(s_j) \tag{4}$$

There are several ways to update the network's weights ($w_{ij}^{t+1} = w_{ij}^t + \Delta w_{ij}^t$):

- Batch Mode, which updates the network's weights with all the training patterns' information:

$$\Delta w_{ij} = -\eta \frac{dR}{dw_{ij}} = -\frac{\eta}{n} \sum_k \frac{dL(y^{(k)}, \hat{y}^{(k)})}{dw_{ij}} \tag{5}$$

- Mini-batch Mode, which updates the weights using a subset of training patterns.

- Online-mode, which uses only one training pattern's information to update the network each epoch.

$$\Delta w_{ij} = -\eta \frac{dL^k}{dw_{ij}} \tag{6}$$

To calculate the loss, the backpropagation algorithm is used:
ϵ_j is obtained from the back propagation network, which propagates the error back.
z_i is obtained from the forward network.

$$\frac{dL}{dw_{ij}} = z_i \epsilon_j \tag{7}$$

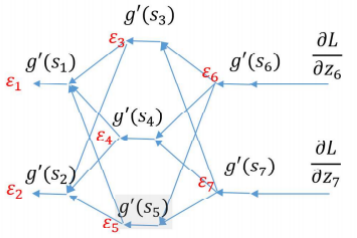

Figure 2: Graphical representation of the backpropagation algorithm

After calculating the loss for each perceptron, the network is updated through one of the modes described above, either the online mode, batch mode or mini-batch mode.

A confusion matrix is a matrix that is used to describe the performance of a classifier on a set of test data for which the true values are known. It allows the visualization of the performance of an algorithm. It allows easy identification of confusion between classes e.g. one class is mislabeled another.
Each entry of the confusion matrix (C_{ij}) is given by the following equation:

$$C_{ij} = P(\hat{y} = j | y = i) \tag{8}$$

Using this, one can easily calculate the probability of error of classification with M classes, with the following equation:

$$P_{error} = 1 - P_{correct\_classification} = 1 - \sum_{i=0}^{M} C_{ii} P(y_i) \tag{9}$$

The Confusion Matrix can also be expressed in absolute terms, rather than probabilistic. In this case, in each entry C_{ij}, we would put the number of examples that belong to class y_i and were classified in class $\hat{y}_j$

Convolutional Neural Network

A Convolutional Neural Network (CNN) is a type of Multi Layer Perceptron, which operates with a convolution operation and is faster and more efficient in image classification than typical multi layer perceptrons.
A convolutional neural network receives an input image and predicts an output image or a label, based on a sequence of internal representations that extract useful information (features) from the image.
Image representations are obtained by a concatenation of layers, including:

- convolutional layers
- pooling layers
- fully connected layers

Convolutional Layers

A convolutional layer receives a 3D input, convolves it with a set of kernels (filters) and applies an activation function (typically RELU) to the filter outputs.
Each filter produces a 2D output known as a feature map. Stacking the feature maps produced by multiple filters leads to a 3D array.

$$\text{3D input: } z_{ijk}^{l-1} \text{ , } l-1 \text{ : number of input layer} \tag{10}$$

$$\text{3D kernel: } h_{ijk}^{l} \tag{11}$$

$$\text{2D output: } s_{ij}^{l} = \sum_{p} \sum_{q} \sum_{r} h_{pqr}^{l} z_{i+p,j+q,0+r}^{l-1} \tag{12}$$

$$z_{ij}^{l} = g(s_{ij}^{l}) \tag{13}$$

Pooling Layers

Pooling reduces the size of a 3D array.
Each channel is separately processed. Each channel is divided into non-overlaping cells and then each cell is replaced by a numeric value (e.g., its maximum or mean).

3D input:

$$z_{ijk}^{l-1} \ , \ l \ - \ number\, of\, input\, layer \tag{14}$$

3D output, replacing each cell by its maximum:

$$z_{ijk}^{l} = \underset{p,q \in \{0,\ldots\Delta-1\}}{max} \{z_{\Delta i+p,\Delta j+q,k^{l-1}}\} \tag{15}$$

Fully Connected Layers

The fully connected layer is used when the image representation is converted into a 1D array.
It is often used as an output layer in classification problems.

1 Digit Recognition

In this section we are going to make digit recognition with the models previously described, using Tensorflow. First we load and analyze the data (section 1.1), then we use a Multi-Layer Perceptron to recognize the digits (section 1.2), then we use a Convolutional Neural Network (section 1.3), and then we analyze the results for each approach (section 1.4).

1.1 Data

1) Load the data files and check the size of inputs X and labels y.

```
X_train = np.load('mnist_train_data.npy')
print("X train:", X_train.shape)
y_train = np.load('mnist_train_labels.npy')
print("y train:", y_train.shape)

X_test = np.load('mnist_test_data.npy')
print("X test:", X_test.shape)
y_test = np.load('mnist_test_labels.npy')
print("y test:", y_test.shape)
```

```
X train: (3000, 28, 28, 1)
y train: (3000,)
X test: (500, 28, 28, 1)
y test: (500,)
```

Figure 3: Dimensions of the trainning and test datasets (output from the program)

As we can see from the program output, the trainning dataset has the size of 3000 and the test dataset has 500 the size of 500.
Each image has the dimensions of 28×28 pixels.

2) Display some of the digits in the train and test data.

```
fig=plt.figure(figsize=(8, 8))
columns = 3
rows = 3
for i in range(1, columns*rows +1):
    fig.add_subplot(rows, columns, i)
    plt.imshow(np.squeeze(X_train, axis=3)[i])
plt.show()
```

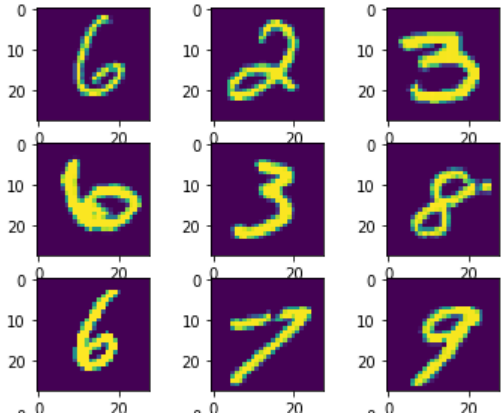

Figure 4: Some digits belonging to the test dataset

3) Divide data by 255 to get floating point values in the 0–1 range.

```
X_train = X_train/255
X_test = X_test/255
```

4) Convert labels to one-hot encoding. In this representation, the label matrix will have 10 elements.

```
y_test = keras.utils.to_categorical(y_test, num_classes=10)
y_train = keras.utils.to_categorical(y_train, num_classes=10)
```

5) Split train data into two subsets, one for actual training and the other for validation.

```
X_train, X_validation, y_train,
y_validation = ms.train_test_split(X_train, y_train, test_size=0.3)
```

1.2 MLP

1) Create a sequential model and start by adding a flatten layer to convert the 2D images to 1D.

```
model = keras.Sequential()
model.add(keras.layers.Flatten(input_shape=(28,28,1)))
```

2) Add two hidden layers to your MLP, the first with 64 and the second with 128 neurons. Use 'relu' as activation function for all neurons in the hidden layers.

```
model.add(keras.layers.Dense(64, activation='relu'))
model.add(keras.layers.Dense(128, activation='relu'))
```

3) End the network with a softmax layer. This is a dense layer that will return an array of 10 probability scores, one per class, summing to 1.

```
model.add(keras.layers.Dense(10, activation='softmax'))
```

4) Get the summary of your network to check it is correct.

```
print(model.summary())
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_4 (Dense)              (None, 128)               8320
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1290
=================================================================
Total params: 59,850
Trainable params: 59,850
Non-trainable params: 0
```

Figure 5: MLP summary

5) Create an Early stopping monitor that will stop training when the validation loss is not improving.

```
earlyStop = keras.callbacks.EarlyStopping(patience=15, restore_best_weights=True)
```

6) Fit the MLP to your training and validation data using *categorical_crossentropy* as the loss function, a batch size of 300 and Adam as the optimizer . Choose, as stopping criterion, the number of iterations reaching 400.

```
adams = keras.optimizers.Adam(lr=0.01, clipnorm=1)

model.compile(optimizer=adams, loss='categorical_crossentropy')

history = model.fit(x=X_train, y=y_train, validation_data=(X_validation, y_validation),
batch_size=300, callbacks = [earlyStop], epochs=400, verbose = 0)
```

7) Plot the evolution of the training loss and the validation loss. Note that the call to fit() returns a History object where metrics monitored during training are kept.

```
plt.figure()
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss (Com Early Stop)')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

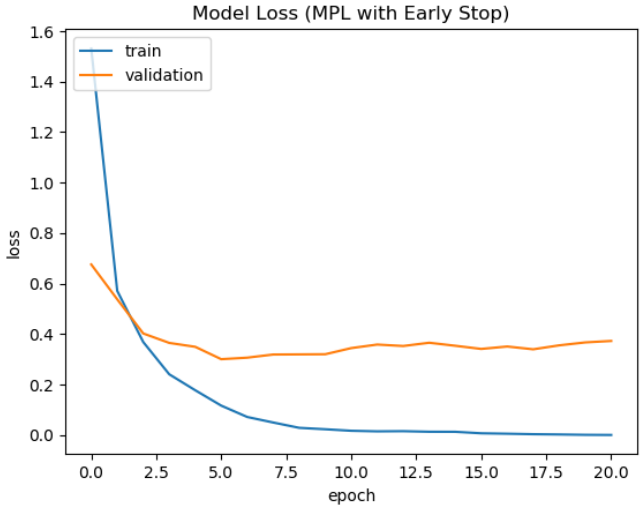

Figure 6: Loss evolution with early stop in function of number of epochs

8) To get an idea of how well the model generalizes to new, unseen data, evaluate performance (accuracy and confusion matrix) on the test data.

```
y_predict = model.predict(X_test, batch_size=300)

print("Accuracy (Com ES):", skmetrics.accuracy_score(np.argmax(y_test,
axis=1), np.argmax(y_predict, axis=1)))

print("Confusion Matrix (Com ES):\n", skmetrics.confusion_matrix(np.argmax(
y_test, axis=1), np.argmax(y_predict, axis=1)))
```

```
Accuracy (Com ES): 0.924
Confusion Matrix (Com ES):
 [[43  0  0  0  0  0  0  0  0  0]
 [ 0 51  1  1  0  0  0  0  0  0]
 [ 0  1 56  2  0  0  0  1  0  0]
 [ 0  0  0 53  0  1  0  1  2  1]
 [ 0  0  0  0 43  0  2  0  0  1]
 [ 0  0  0  1  0 35  3  1  0  2]
 [ 0  0  1  0  0  0 40  0  0  0]
 [ 0  0  3  0  1  0  0 49  0  1]
 [ 0  1  0  0  1  0  1  1 49  1]
 [ 0  0  0  0  4  0  0  2  0 43]]
```

Figure 7: Accuracy and Confusion Matrix with early stop

The accuracy can also be calculated from the Confusion Matrix by dividing the number of correct classifications by the total number of examples: $Accuracy = \frac{\sum_i C_{ii}}{\sum_{i,j} C_{ij}} = \frac{462}{500} = 92.4\%$

This way, we can see that the accuracy obtained with the python libraries is the same as the one obtained with the confusion matrix.

9) Repeat the previous items without Early Stopping

```
adams = keras.optimizers.Adam(lr=0.01, clipnorm=1)

model2.compile(optimizer=adams, loss='categorical_crossentropy')

history2 = model2.fit(x=X_train, y=y_train, validation_data=(X_validation, y_validation),
batch_size=300, epochs=400, verbose = 0)
```

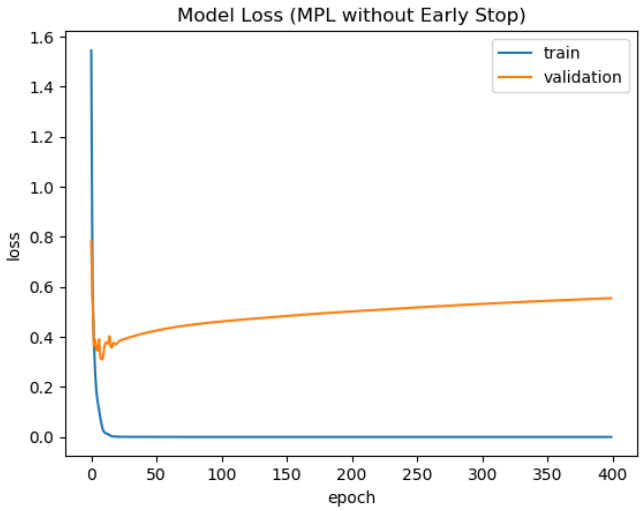

Figure 8: Loss evolution without early stop in function of number of epochs

```
Accuracy (Sem ES): 0.94
Confusion Matrix (Sem ES):
 [[43  0  0  0  0  0  0  0  0  0  0]
 [ 0 50  1  1  0  0  0  0  1  0]
 [ 0  0 58  1  0  0  0  0  1  0]
 [ 0  0  1 55  0  1  0  1  0  0]
 [ 1  0  0  0 43  0  1  0  0  1]
 [ 0  0  0  1  2 36  1  1  0  1]
 [ 0  0  1  0  0  0 40  0  0  0]
 [ 0  0  3  0  1  0  0 50  0  0]
 [ 1  0  0  0  0  2  0  0 51  0]
 [ 0  0  0  1  1  0  0  2  1 44]]
```

Figure 9: Accuracy and Confusion Matrix without Early Stopping

1.3 CNN

1) Create a Convolutional Neural Network (CNN) with two alternated Convolutional (with relu activation and 3x3 filters) and MaxPooling2D layers (2x2). Use 16 filters in the first conv layer and 32 in the second. Add a flatten layer and then a dense layer with 64 units (with relu activation). End the network with a softmax layer.

```
modelCNN = keras.Sequential()

modelCNN.add(keras.layers.Conv2D(16,(3,3), activation='relu',
input_shape=(28,28,1)))

modelCNN.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))

modelCNN.add(keras.layers.Conv2D(32,(3,3), activation='relu'))
modelCNN.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))

modelCNN.add(keras.layers.Flatten())
modelCNN.add(keras.layers.Dense(64, activation='relu'))

modelCNN.add(keras.layers.Dense(10, activation='softmax'))
```

2) Get the summary of your network to check it is correct.

```
print(modelCNN.summary())
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 16)        160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 32)        4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 32)          0
_________________________________________________________________
flatten (Flatten)            (None, 800)               0
_________________________________________________________________
dense (Dense)                (None, 64)                51264
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 56,714
Trainable params: 56,714
Non-trainable params: 0
```

Figure 10: CNN summary

3) Fit the CNN to your training and validation data. Use the same loss function, batch size, optimizer and Early Stopping callback that were used for the MLP.

```
earlyStop = keras.callbacks.EarlyStopping(patience=15, restore_best_weights=True)

adams = keras.optimizers.Adam(lr=0.01, clipnorm=1)

modelCNN.compile(optimizer=adams, loss='categorical_crossentropy')

historyCNN = modelCNN.fit(x=X_train, y=y_train, validation_data=(X_validation, y_validation),
batch_size=300, callbacks = [earlyStop], epochs=400, verbose = 0)
```

4) Plot the evolution of the training loss and the validation loss.

```
plt.figure()
plt.plot(historyCNN.history['loss'])
plt.plot(historyCNN.history['val_loss'])
plt.title('model loss (CNN)')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

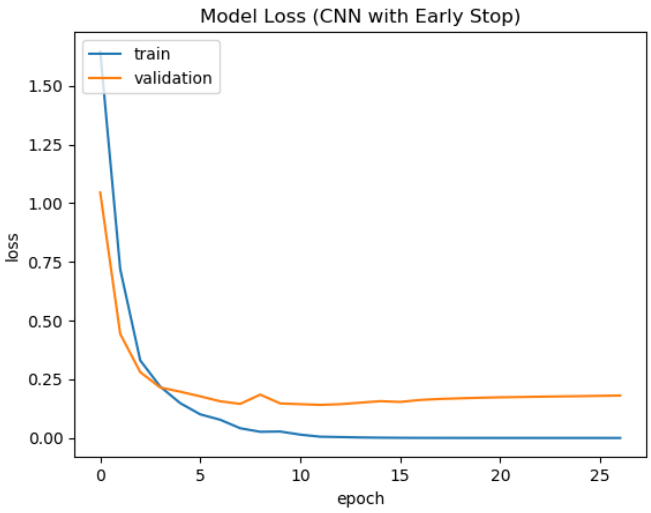

Figure 11: Model Loss (CNN with Early Stop)

5) Evaluate performance (accuracy and confusion matrix) on the test data.

```
y_predictCNN = modelCNN.predict(X_test, batch_size=300)

print("Accuracy (CNN):", skmetrics.accuracy_score(np.argmax(y_test, axis=1),
np.argmax(y_predictCNN, axis=1)))

print("Confusion Matrix (CNN):\n", skmetrics.confusion_matrix(np.argmax(y_test,
axis=1), np.argmax(y_predictCNN, axis=1)))
```

```
Accuracy (CNN): 0.976
Confusion Matrix (CNN):
 [[43  0  0  0  0  0  0  0  0  0]
 [ 0 50  1  1  0  0  0  0  1  0]
 [ 0  0 59  0  0  0  0  0  1  0]
 [ 0  0  0 57  0  0  0  1  0  0]
 [ 1  0  0  0 44  0  1  0  0  0]
 [ 0  0  0  0  0 41  1  0  0  0]
 [ 0  0  0  0  0  0 41  0  0  0]
 [ 0  0  1  0  0  0  0 53  0  0]
 [ 0  0  0  0  0  1  0  0 52  1]
 [ 0  0  0  0  1  0  0  0  0 48]]
```

Figure 12: CNN confusion matrix

1.4 Comments

1) Explain in your own words how Early Stopping works and what is the goal.

During training, weights in the neural networks are updated so the model better fits the training data. For a while, improvements on the training set correlate positively with improvements on the validation set. However, there comes a point where you begin to overfit on the training data and further "improvements" will result in lower generalization performance. This is known as overfitting. After that point, the validation performance gets worse (loss increases and accuracy decreases). Early stopping is introduced to terminate the training before overfitting happens.

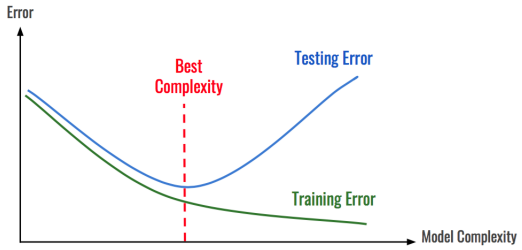

Figure 13: Overfitting

```
keras.callbacks.callbacks.EarlyStopping(patience=15, restore_best_weights=True)
```

patience: number of epochs that produced the monitored quantity with no improvement after which training will be stopped.
restore_best_weights: whether to restore model weights from the epoch with the best value of the monitored quantity. If False, the model weights obtained at the last step of training are used.

In our case we set patience to 15 and restore_best_weights to True. This way, after 15 epochs, if there is no improvement in the accuracy we restore the weights of the network, to the best fit for accuracy of those 15 weights, so that we assure that "overfitting" did not occur.

2) Comment on the differences in loss evolution, execution time and test set performance between the MLP model with and without Early Stopping.

Looking at figures 6 and 8, without Early Stopping, the loss in the training set is always decreasing, but in the validation set it starts decreasing, then hits a minimum and starts increasing. With Early Stopping this doesn't happen, since the execution is terminated before the loss starts increasing, So, the loss in the situation without Early Stopping is lower in the training set and higher in the validation set, compared to the situation with Early Stopping. This is due to the occurrence of overfitting in the training set, in the situation without Early Stopping.

The execution time is much lower in the situation with Early Stopping, since it only iterates through about 20 epochs instead of the full 400.

The values obtained for accuracy in the test set are: 0.924 with Early Stop and 0.94 without Early Stop. As we can see, although these values are very close, the performance of an independent set is higher in the model in which Early Stopping was not used, due to the fact that we are trying to minimize the cross-entropy and not the loss and also due to the fact that the test dataset is almost two times smaller than the validation dataset, wich makes the result accuracy quite unpredictable on the test dataset.

3) Comment on the differences between MLP and CNN with Early Stopping, in what regards performance and number of parameters.

As we can see from figures 6 and 11, loss is smaller in the validation dataset, using the CNN.

```
Accuracy (MPL with Early Stop): 0.924
Accuracy (CNN with Early Stop): 0.976
```

Accuracy in the test dataset is also better (bigger) using a CNN, despite using less parameters (56714) compared to the MLP (59850).

This shows that CNN significantly improves the performance, compared to the MLP when it comes to image classification, as expected.
Despite this fact, in terms of computational resources, CNN is more demanding, since it requires the use of convolution, and has more layers. This is evident, since the execution is much bigger with the use of the CNN.

Final Considerations

In the following figure, we can see some of the images belonging to the test dataset that were misclassified by all three neural networks.

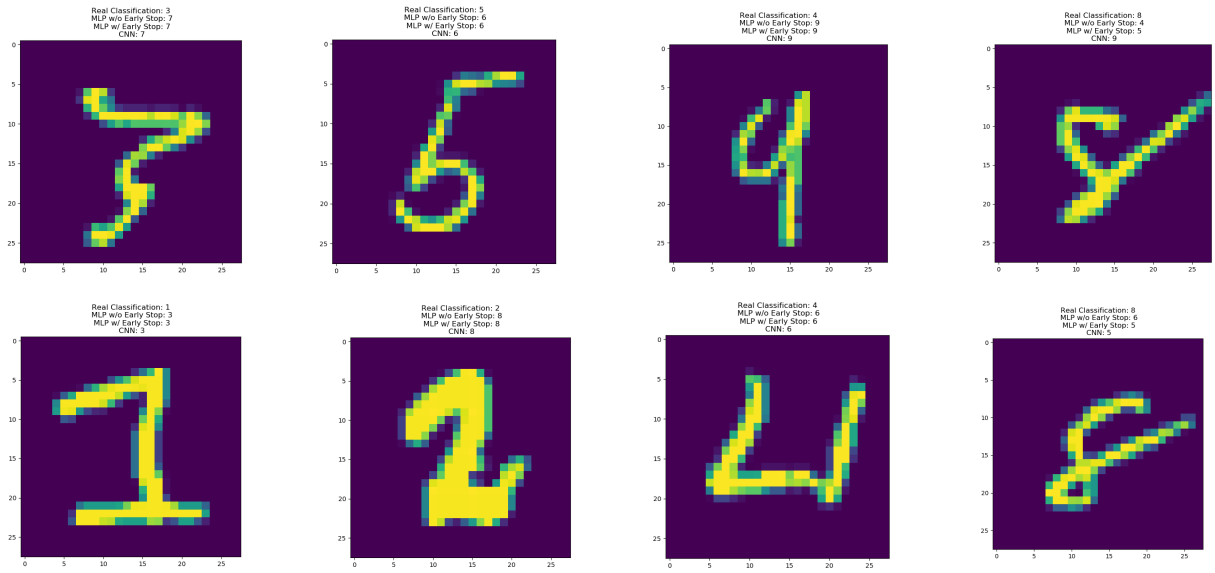

Figure 14: Images belonging to the test dataset that were misclassified by all three neural networks

As we can see, most of these images are really hard to classify, to the point that even a human would have a hard time doing it. With this in mind, it's remarkable that all three neural networks achieved surprisingly high scores for accuracy (over 90% for all of them), considering that the training set had only about two thousand training images.