Statistics and Machine Learning II

Neural Networks

Coursework: Neural Network Training

Luis Da Silva

March 13, 2019

1 Introduction

In order to understand, or at least get further insight into how neural network models work, this coursework is designed to explore the role of the different hyperparameters it is possible to tune while training a Multi-Layer Perceptron model. In short, the main objective is to discuss a list of hyperparameters and their influence on the accuracy score obtained by classifying the test Modified National Institute of Standards and Technology (MNIST) dataset.

MNIST data consist of a training set of 60000 examples and a test set of 10000 examples of handwritten digits normalized into a grayscale 28x28 pixel box and is frequently used to test machine learning algorithms.

All the work presented here is being conducted on a Dell Inspiron 15 laptop with an NVIDEA GEFORCE GTX 960M Graphical Processing Unit. Although MNIST dataset is of reduced complexity and size (images are only $28 * 28 = 784$ pixels long), training time may grow exponentially on the number of testing values for each parameter, thus I will focus on a low and discrete search space.

2 Neural Networks

Inspired by the human brain itself, neural networks are a framework for processing data built of nodes (neurons) arranged in layers that are connected to each other in a directed way.

The first layer is the input, which are the values for the features describing an object or situation. Last layer is the output, that is the information it is intended to be obtained by implementing the neural network. Any layer is between is called a hidden layer, and it may have any number of hidden units (nodes).

In the MNIST dataset, the input layer is a set of values that describes how bright each pixel is, thus $28*28 = 784$ nodes. On the other hand, output layer contains a node for each possible outcome with a number between 0 and 1 (softmax), meaning how confident the network is of a certain image to be representing a certain number; thus, 10 nodes.

Each of the nodes in one layer may be connected to the next layer via a weight, so each node will be a linear transformation of the nodes in the previous layer. To allow neural networks to capture non-linear patterns in the data, an activation function is used, which transforms the value on the node in a non-linear way before that node is used to calculate next layer's nodes.

Weights are usually initialized by random, but they might use weights from a pre-trained neural network with similar purposes using a technique called transfer learning. Then, the weights are adjusted (usually) via back-propagation to minimize a cost function (or alternatively, maximize a score function) in a series of iterations called epochs.

Based on this framework, a huge variety of architectures and arrangements of layers, connections, etc. can be ensemble to design a customized infrastructure that will achieve a high success rate on a particular kind of task.

3 Hyper-parameter research

As Neural Networks are a highly flexible framework, customization becomes imperative. Nevertheless, it is important to note that the objective of this coursework is not to achieve the lowest possible error rate, but to get a sense of how the different hyper-parameters affect the neural network accuracy, thus research is done in a finite set of alternatives.

A Simple Perceptron Neural Network (i.e. no hidden layers) is used as the base model to be improved. The following general procedure is used to compute and compare the different neural networks trained:

1. Define a model with the objective hyper-parameters.
2. Randomly assign weights.
3. Train model on training dataset with a 20% validation split.
4. Score model on test dataset.
5. As there is some randomness in the weight adjustment procedure, repeat steps 2 to 4 four more times and then average test scores.
6. Compare score with the base model's score. If the score is better then keep this as the new base model.

Hyper-parameters to be taken into account are:

- Number of Hidden Layers.
- Number of Epochs.
- Number of Hidden Units.
- Activation function.
- Dropout rate.
- Optimizer.
- Learning Rate.

- Batch Size.

Each hyper-parameter will be tested one at a time, and whenever a better model is found it turns into the new base model. As in reality the performance of a given hyper-parameter is dependant on the others, this search method is not guaranteed to produce optimal results. In fact, the order of tuning might change the final outcome. Nonetheless, I still use it because it reduces hugely the searching space.

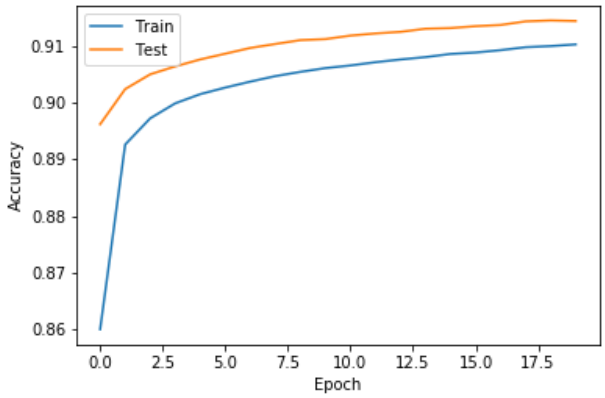
As an example, an exhausting searching for 3 possible values in 8 hyper-parameters will turn into $3^8 = 6561$ distinct models, and as they will be computed 5 times, that means 32805 optimizations (completely non-viable). On the other hand, this step-wise method only needs to fit $3 * 8 = 24$ models or 120 optimizations.

3.1 Base model: Simple Perceptron

As pointed out before, the model used as based is a Simple Perceptron (SP). It is known that the input layer will have 784 units and the output layer only 10. As there are no hidden layers, the output will be a linear combination of the input. Nevertheless, there are some hyper-parameters that need to be chosen in advance. The first one is the number of epochs, which is initially set to 20. Then the optimizer, which will be Stochastic Gradient Descent (SGD). Batch size, which is the number of samples that will be propagated across the network, will be set to 128 and finally, layers will be densely connected, thus each node in the input layer will have a connection with every node in the output layer, and a bias for each output will be added, therefore $784 * 10 + 10 = 7850$ weights will need to be tuned.

After training the network, this simple model already gets an average test accuracy of 91.482%, with a minimum of 90.61% and a maximum of 92.01%, setting the first benchmark to be beaten. Figure 1 shows the average accuracy history across epochs. It is easy to see that accuracy seems to have a positive trend at the end of the line, and thus it could get better by increasing the number of epochs. Also, the fact that the test set gets higher accuracy than the train set means that there is still a lot of room to improve these results yet.

Figure 1: Simple Perceptron performance



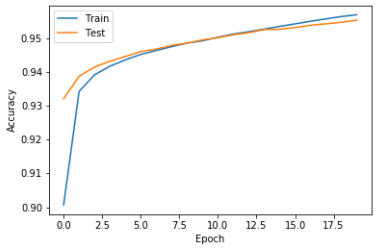
3.2 Adding hidden layers

Although increasing the number of epochs seems like a natural next step to improve the SP accuracy, I am jumping directly to selected a number of hidden (and thus turning the SP into a Multi-Layer Perceptron, MLP) layers for two reasons: first, it is known that computer vision needs some degree of non-linearity to perform well, and this is achieved by adding hidden layers to the SP model. Second, when the number of layer increases, the number of weights to be trained (thus the flexibility of the model) also increases and usually these needs a smaller number of epochs to be optimized (the marginal contribution of each weight is smaller, thus small variations in them have a non-perceptible impact in the output).

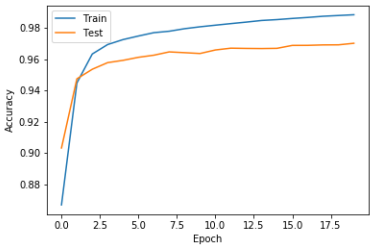
Nevertheless, adding hidden layers immediately requires the set-up of some additional hyper-parameters. As suggested in the coursework lab, the number of hidden units (i.e. the size of the layer) will be set to 128, the activation function, which is a transformation implemented to the value of each node, will be ReLU (see figure 5a) and the batch size, which is the number of samples that will be propagated through the network, will be 128. For simplicity, every hidden layer will share the same characteristics.

Even adding a simple hidden layer with these characteristics increases the number of parameters to be tuned from 7850 to $784*128+128+128*10+10 = 101770$. I am searching the space between 1 and 7 hidden layers and then

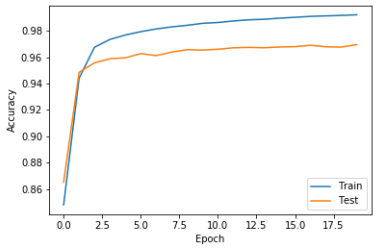
Figure 2: Adding Hidden Layers



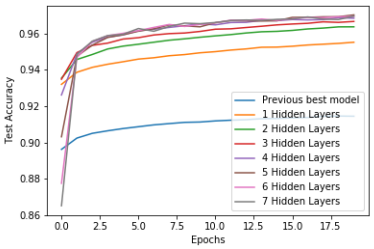
(a) 1 Hidden Layer Performance



(b) 5 Hidden Layers Performance



(c) 7 Hidden Layers Performance



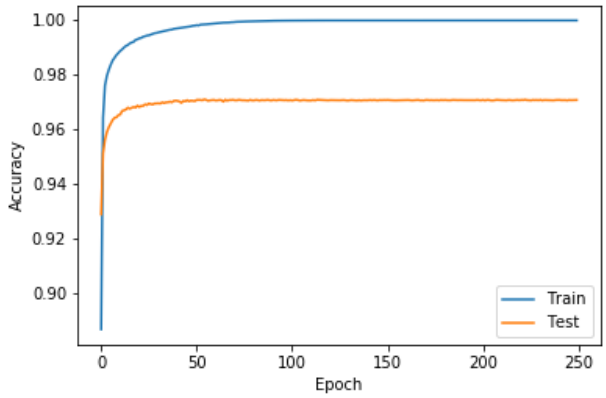
(d) Hidden Layers comparison. Previous best model is SP.

choosing the on that gives the best average test result after 20 epochs.

A single hidden layer improves test accuracy up to 95.328% (i.e. +3.846 percentage points), but adding 5 hidden layers improves accuracy up until 97.036%! This new 7 layer network (input + 5 hidden + output) needs to train 167818 weights. If one adds two more layers accuracy suffers a little, going down to 96.63%, meaning that the extra complexity is not helping the network generalize to new data. Figure 2d shows the test scores for 1 to 7 hidden layers.

Next step in the search for a better network is to tune the number of epochs.

Figure 3: 250 epochs test and train accuracy



3.3 Changing the number of epochs

The number of epochs could be defined as the number of times each individual observation is used for training the network. After all the training observations has been used to optimize weights, these may be used again because neural networks use an iterative approach to optimization instead of an analytical one (like a linear model does).

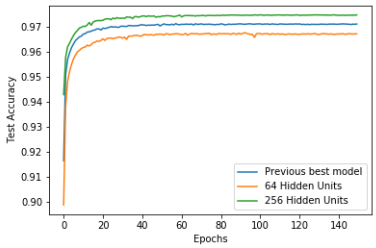
Although the number of epochs does not add any more weights to be optimized, it affects computational time needed to complete the training. More epochs will directly translate into more time, thus one does not want to do more than it is necessary.

I tested the 7 layer MLP with 150 and 250 epochs and, as figure 3 shows, the test improvement curve becomes flat around 50 epochs, thus 150 is selected with a better score of 97.244% (+0.208 percentage points). Although one may reduce the number to about 50 epochs and reduce training time to a third, I allowed it to stay at 150 just in case extra flexibility is needed later.

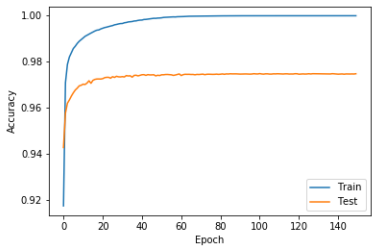
3.4 Changing the number of hidden units

Another way to change the complexity in a neural network is to change the number of hidden units, or nodes, each of the hidden layer has. If one reduces

Figure 4: Changing the number of hidden units



(a) Comparison of test performances. Previous best model had 128 hidden units.



(b) Performance with 256 Hidden Units

the number then the network is forced to generalize more, and may underfit. On the other hand, a higher number of nodes allows more complex functions to be modelled, but it may produce overfitting.

As the starting number of hidden units was 128, I tested 64 (half) and 256 (double). As shown in figure 4a, the latter configuration increased mean test accuracy from the beginning and up to 97.698% (+0.454 percentage points). Figure 4b also shows that train acurracy achieved 100% at around 60 epochs, showing a slightly overfitted model.

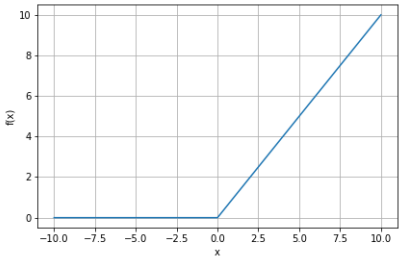
Now, the new network with 7 layers densely connected ad 5 hidden layers of 256 nodes each has 466698 weights to train.

3.5 Using different activation functions

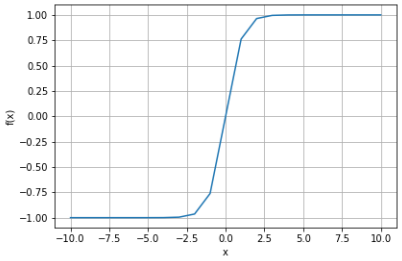
In addition to ReLU, there are a of activation functions commonly used in neural networks. The ones I picked for testing are:

- ReLU (figure 5a): used from the beginning of this coursework, is defined by $f(x) = max(0, x)$.
- Tanh (figure 5b): the hyperbolic analogue of a tangent function.
- Sigmoid (figure 5c): defined as $\frac{1}{1+e^{-x}}$.

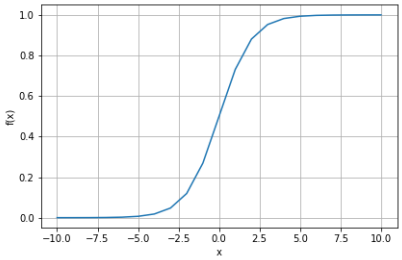
Figure 5: Activation functions



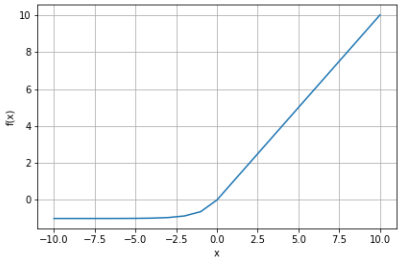
(a) Rectified Linear Unit (ReLU)



(b) Hyperbolic Tangent (tanh)



(c) Sigmoid Function



(d) Exponential Linear Unit (ELU)

- ELU (figure 5d): defined as $f(x) = x$ if $x > 0$ and $f(x) = \alpha(e^x - 1)$ if $x < 0$. $\alpha = 1$ is used.

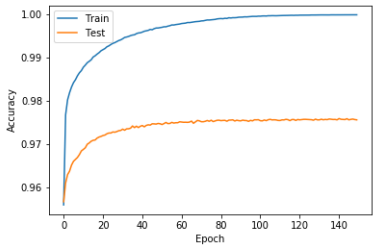
After training the network on each type of activation function, the tanh activation achieved an accuracy score of 97.776%, slightly better than the network using ReLU activation (+0.078 percentage points), altough it is worth noticing that this type of activation is slower.

Also, by looking at figure 6b, it is interesting to notice that the sigmoid activation presents, by far, the slowest learning path.

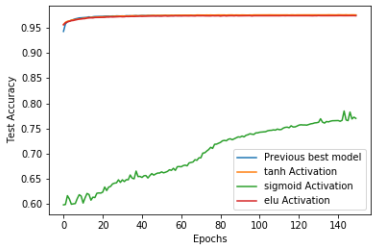
3.6 Adding dropout

It is possible to "shut down" randomly a proportion of the nodes in a given layer during an epoch with the purpose of "forcing" the neural network to be robust on its predictions. It is a interesting feature as it helps prevent over

Figure 6: Activation functions



(a) Accuracy performance using Tanh Activation



(b) All function's test accuracy. Previous best model uses ReLU.

fitting. The bas side is that it increases significantly the number of iterations required for a network to converge, although epoch training time is smaller.

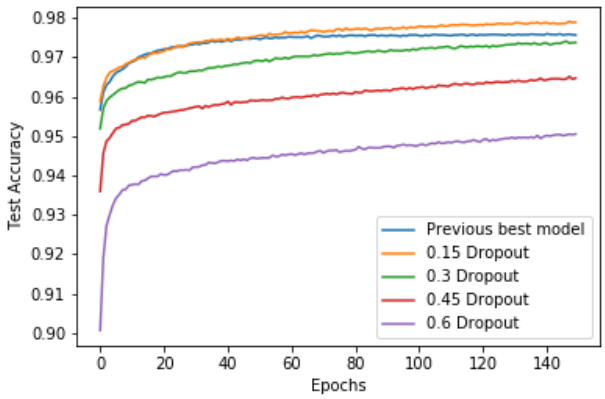
As the MLP so far has 0% dropout rate, I tested 4 increments of 15%. As seen in figure 7, a dropout rate of 15% improves the model's accuracy and sets it in 97.982%. Higher dropout rates drops too much of the model's complexity and thus leaves to underfitting. Also, it seems that the model could even get better by increasing the number of epochs (it hasn't converged yet) but let's keep on with this result as some other optimizer may converge quicker.

3.7 Changing the optimizer

Keras has 7 built-in optimizers for the weights of a neural network. So far everything has been trained following a Stochastic Gradient Descent (SGD), which is an iterative method to optimize a function, but I am going to test two popular ones: Adam, which is an extension of SGD, means Adaptive Moment Estimation and computes adaptive learning rates using the mean and the variance of past improvements in loss function; and RMSProp, short for Root Mean Square Propagation, which is another adaptive learning rate method but only based on the recent magnitudes of the weights.

Although both Adam and RMSProp are much more volatile than SGD, they seem to have learnt much faster to recognize digit numbers on this network setting, needing only about 20 epochs to converge, and that's even

Figure 7: Dropout comparison. Previous best model has a dropout rate of 0.



given that they use an smaller default learning rate of 0.001 instead of 0.01.

Figure 8 shows that the RMSProp optimizer achieves a new best test accuracy of 98.18% (+0.198 percentage points).

3.8 Changing the learning rate

Maybe it is possible to improve this result further by allowing the network to learn slower (decreasing its volatility) when it starts to converge. That is the function of the learning rate (LR) of the optimizer: a higher value will increase volatility but will speed up convergence, while a smaller value will be more careful in finding the minimum in the loss function, but it will take longer and may get caught in local minima.

Remember that the RMSProp optimizer that the current network is using has a default learning rate of 1e−3, so I tested a smaller rate of 1e−4. Figure 9a shows that, effectively, a smaller learning rate takes longer to converge, but once it does, its behaviour is much more stable and it actually achieves a higher test score of 98.234% (+0.054 percentage points). On the other hand, figure 9b shows what happens when a too high learning rate of 0.01 is used: volatility shoots and convergence occurs fast, but at a very low score.

Figure 8: Optimizers comparison. Previous best model uses SGD.

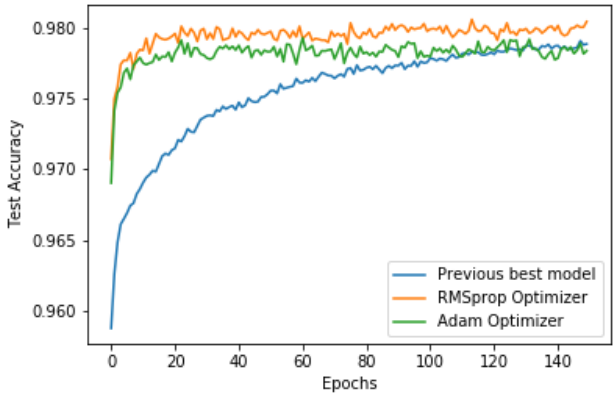
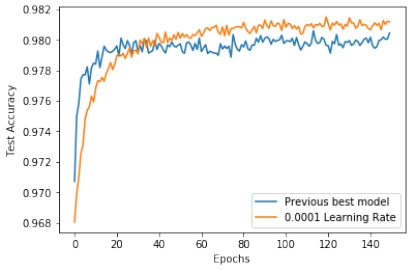
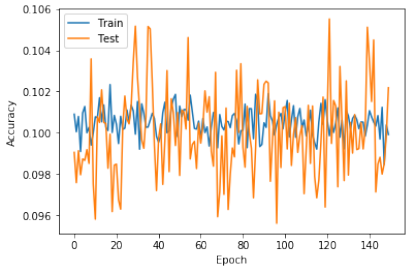


Figure 9: Learning Rates

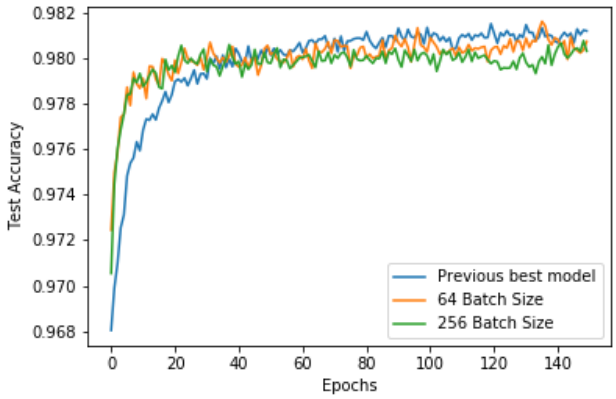


(a) Comparison between $LR =$ 1e$-$3 and $LR =$ 1e$-$4



(b) A high $LR = 0.01$ value

Figure 10: Batch size performance comparison. Previous best model uses 128 samples per batch.



3.9 Changing the batch size

Finally, the last change I am testing is to change the batch size. As stated earlier, a batch is the number of samples a neural network will take into account before adjusting its weights. Therefore, the smaller the batch size less memory is required for the network to be trained, but also the estimate of the gradient will become less accurate (the network will have less information before making a decision on the weights) for the training data. Another issue with large batch sizes is that they are more prone to overfit due to the presence of minimizers in the training function.

Figure 10 shows the training trajectories of the network with 64, 128 and 256 batch sizes. In fact, as training set has 48000 images, those are all relatively low batch sizes, thus their performance is very similar.

4 Summary

This particular way of searching the space of the huge amount of different neural network settings arrived to the conclusion that a 7 layer Neural network with 256 nodes in each of its 5 hidden layers, each with tanh as their activation function, a dropout rate of 15%, a batch size of 128 samples and

optimized with a RMSProp with $LR = 1e{-}4$ achieves the best test score of all possibilities tested. Mean test accuracy score got to 98.234%, which means an improvement of +6.752 percentage points from the simple perceptron model or +3.564 percentage points from the MLP network presented in lab.

Nevertheless, external research shows that much better results may be achieved by further finetuning the network. More precisely, Ciresan et al. (2010) trained a 7 layer Neural Network with 2500, 2000, 1500, 1000 and 500 nodes in each of their 5 hidden layers and achieved an accuracy score of 99.65%.

Related techniques, such as Convolutional Neural Networks, have achieved accuracy scores as high as 99.77% (Ciresa et al., 2012).

Appendix: Python Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import datetime as dt
from joblib import dump, load
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Dropout
from keras.optimizers import SGD, Adam, RMSprop
from keras.datasets import mnist
from keras.utils import np_utils

np.random.seed(4526)

def read_transform_mnist():
    mn = {}
    # Read dataset
    (input_X_train, output_Y_train), (input_X_test, output_Y_test) =
     mnist.load_data()
    mn['nclasses'] = len(np.unique(output_Y_test))
    mn['dimmension'] = input_X_train.shape[1] * input_X_train.shape[2]
```

```
    # Flatten
    mn['input_X_train'] = input_X_train.reshape(input_X_train.shape[0],
     mn['dimmension']).astype('float32')
    mn['input_X_test'] = input_X_test.reshape(input_X_test.shape[0],
     mn['dimmension']).astype('float32')

    # Now rescale to values between 0 and 1
    mn['input_X_train'] /= mn['input_X_train'].max()
    mn['input_X_test'] /= mn['input_X_test'].max()

    # Convert to categorical
    mn['output_Y_train'] = np_utils.to_categorical(output_Y_train,
     mn['nclasses'])
    mn['output_Y_test'] = np_utils.to_categorical(output_Y_test,
    mn['nclasses'])

    return mn

mn = read_transform_mnist()

# # Defining functions to make training models easier
def train_evaluate(model, mn, epochs = 20, batch_size=128,
validation_split=0.2,
verbose='one', repeat=10, plot=True, img_name=None, model_name='auto'):
    results = pd.DataFrame()
    test_score = []
    test_acc = []
    ini = dt.datetime.now()

    if verbose == 'results':
        v = 0
    else:
        v = 1

    for i in range(repeat):
        model.reset_states() # Start with a random model
        rini = dt.datetime.now()
        if verbose == 'one' and i!=0:
            v = 0
```

```
        # Train model
        history = model.fit(mn['input_X_train'], mn['output_Y_train'], ba
                            verbose=v, validation_split=validation_split)
        results['acc_{}'.format(i)] = history.history['acc']
        results['val_acc_{}'.format(i)] = history.history['val_acc']

        # Score on validation set
        score = model.evaluate(mn['input_X_test'], mn['output_Y_test'],
         verbose=v)
        test_score.append(score[0])
        test_acc.append(score[1])

        #Print round results
        now = dt.datetime.now()
        print('*--'*30)
        print('Round: {}. Time: {}. Total time: {}. Expected: {}'.format(
         now-rini, now-ini, (now-ini)*repeat/(1+i)))
        print("Test score:", score[0])
        print("Test accuracy:", score[1])

    # Plot
    if plot:
        if model_name == 'auto':
            model_name = model.__name__

        acc = ['acc_{}'.format(i) for i in range(repeat)]
        val_acc = ['val_acc_{}'.format(i) for i in range(repeat)]
        acc_mean = results[acc].mean(axis=1)
        val_acc_mean = results[val_acc].mean(axis=1)

        plt.plot(acc_mean, label='Train')
        plt.plot(val_acc_mean, label='Test')
        plt.ylabel('Accuracy')
        plt.xlabel('Epoch')
        plt.legend()
        if model_name is not None:
            plt.title('Model {} accuracy'.format(model_name))
        if img_name is not None:
            plt.savefig('Graphs/{}.png'.format(img_name))
```

```
            plt.show()

        print('*--'*30)
        print('Mean test acc:', np.mean(test_acc))
        return {'history_acc':acc_mean, 'history_val_acc':val_acc_mean,'test_
         'test_acc':test_acc}

    class Model:
        def __init__(self, classes, dimmension, data):
            self.classes = classes
            self.dimmension = dimmension
            self.hlayers = 0
            self.epochs = 20
            self.hidden_units = 128
            self.activation = 'relu'
            self.dropout = None
            self.optimizer = SGD
            self.optimizer_name = type(self.optimizer).__name__
            self.optimizer_params = {}
            self.batch_size = 128
            self.acc = 0
            self.data = data
            self.previous_history = None

        def create_model(self, hlayers, hidden_units, activation,
         dropout, optimizer):
            self.model = Sequential()

            # Add layers
            for i in range(hlayers):
                self.model.add(Dense(hidden_units, input_shape=
                (self.dimmension,)))
                self.model.add(Activation(activation))
                if dropout is not None:
                    self.model.add(Dropout(dropout))

            # Add output layer
            self.model.add(Dense(self.classes))
            self.model.add(Activation('softmax'))
```

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

    def best_model_summary(self):
        print('Hidden Layers: {}'.format(self.hlayers))
        print('Hidden Units: {}'.format(self.hidden_units))
        print('Epochs: {}'.format(self.epochs))
        print('Activation: {}'.format(self.activation))
        print('Dropout: {}'.format(self.dropout))
        print('Optimizer: {}'.format(self.optimizer_name))
        print('Batch size: {}'.format(self.batch_size))
        print('Accuracy: {}'.format(self.acc))
        self.model.summary()

    def create_train(self, repeat=5, epochs=None, hlayers=None,
                                    hidden_units=None,
                     activation=None, dropout=None, optimizer=None,
                      batch_size=None,
                     optimizer_params={}):
        hlayers = self.hlayers if hlayers is None else hlayers
        hidden_units = self.hidden_units if hidden_units is None else
         hidden_units
        activation = self.activation if activation is None else
        activation
        dropout = self.dropout if dropout is None else dropout
        optimizer_name = self.optimizer_name if optimizer is None
        else type(optimizer).__name__
        optimizer = self.optimizer if optimizer is None else optimizer
        optimizer_params = self.optimizer_params if optimizer_params
        is None else optimizer_params

        call_optimizer = optimizer(**optimizer_params)

        self.create_model(hlayers, hidden_units, activation, dropout,
         call_optimizer)

        epochs = self.epochs if epochs is None else epochs
        batch_size = self.batch_size if batch_size is None else
```

```
            batch_size

            name = '{}HL_{}HU_{}␣activation_{}␣dropout_{}␣optimizer_{}
        ␣␣␣␣␣␣␣␣␣epochs_{}␣batch'.format(hlayers, hidden_units, activation,
              dropout, optimizer_name, epochs, batch_size)
            results = train_evaluate(self.model, self.data, epochs =
            epochs, batch_size=batch_size,
            validation_split=0.2, verbose='one', repeat=repeat, img_name=name
                                                model_name=None)

            test_result = np.mean(results['test_acc'])
            if test_result > self.acc:
                self.hlayers = hlayers
                self.epochs = epochs
                self.hidden_units = hidden_units
                self.activation = activation
                self.dropout = dropout
                self.optimizer = optimizer
                self.batch_size = batch_size
                self.acc = test_result
                self.optimizer_name = type(optimizer).__name__
                self.best_history = results['history_val_acc']
                print("Better␣model␣found!")

            self.model.summary()
            return results

        def update_history(self):
            self.previous_history = self.best_history

        def plot_comparison(self, results, keys, names, suffix, abbr):
            plt.plot(self.previous_history, label='Previous␣best␣model')
            for key, name in zip(keys, names):
                label = '{}␣{}'.format(name, suffix)
                plt.plot(results[key]['history_val_acc'], label=label)
            plt.xlabel('Epochs')
            plt.ylabel('Test␣Accuracy')
            plt.legend()
            plt.savefig('Graphs/comparison_{}.png'.format(abbr))
            plt.show()
```

```
        self.update_history()

def dump_all(model, results):
    dump(model, 'results/model.joblib')
    dump(results, 'results/results.joblib')

def evaluate_hp(model, results, param, variations, param_name, param_abbr
    keys = ['MLP_{}{}'.format(i, param_abbr) for i in variations]
    for variation, key in zip(variations, keys):
        results[key] = model.create_train(**{param:variation})

    model.best_model_summary()
    model.plot_comparison(results, keys, variations, param_name,
    param_abbr)
    dump_all(model, results)

    return results

model = Model(mn['nclasses'], mn['dimmension'], mn)

# # Exploring the effect of each parameter

# Simple Perceptron
results = {}
results['SP'] = model.create_train()
model.update_history()
dump_all(model, results)

# Adding Hidden Layers
results = evaluate_hp(model, results, 'hlayers', range(1,8),
'Hidden␣Layers', 'HL')

# Change the number of epochs
results = evaluate_hp(model, results, 'epochs', (150,250),
'Epochs', 'E')

# Change the number of Hidden Units
results = evaluate_hp(model, results, 'hidden_units',
(64,256), 'Hidden␣Units',
```

```
 'HU')

# Change the activation functions
results = evaluate_hp(model, results, 'activation',
('tanh', 'sigmoid', 'elu'),
 'Activation', 'Act')

# Change the dropout rate
results = evaluate_hp(model, results, 'dropout',
(0.15, 0.3, 0.45, 0.6), 'Dropout', 'drop')

# Change the optimizer
opts = (RMSprop, Adam)
opts_names = [i.__name__ for i in opts]
keys = ['MLP_{}_optimizer'.format(opt) for opt in opts_names]
for opt, key in zip(opts, keys):
    results[key] = model.create_train(optimizer=opt)

model.plot_comparison(results, keys, opts_names,
'Optimizer', 'Opt')
dump_all(model, results)

# Change the learning rate
lrs = (0.0001, 0.001, 0.01)
keys = ['MLP_{}_LR'.format(lr) for lr in lrs]
for lr, key in zip(lrs, keys):
    results[key] = model.create_train(optimizer_params={'lr':lr})

model.plot_comparison(results, keys, lrs, 'Learning␣Rate', 'LR')
dump_all(model, results)

lrs = (0.0001,)
keys = ['MLP_{}_LR'.format(lr) for lr in lrs]
for lr, key in zip(lrs, keys):
    results[key] = model.create_train(optimizer_params={'lr':lr})

model.plot_comparison(results, keys, lrs, 'Learning␣Rate', 'LR')
dump_all(model, results)

# Change batch size
```

```
results = evaluate_hp(model, results, 'batch_size', (64,256),
'Batch Size', 'BS')

model.best_model_summary()
```