11753 - Computational Intelligence
Intelligent Systems
Universitat de les Illes Balears

FEED FORWARD NEURAL NETWORKS

Manasut, Phornphawit
Charnota, Sofiia

Supervisor: Prof. Alberto Ortiz

1st Semester

January 13, 2024

Contents

1 Task 0

1.1 Disclaimer before we begin

Throughout my experiments, we had to restart our Colab notebook because the time ran out. I accidentally rerun every single experiments again. Due to the random nature of weight initialization, some of the results reported in this pdf will not match with the notebook (although they are similar). Since, the report has already been 90% completed, we do not wish to rerun all our experiments and change our discussions again. In hindsight, I should have just rerun the main code functions and then run the last few experiments. The responsibility is on me (Phornphawit Manasut) for messing up but I hope you are lenient on this.

1.2 Code Explanation

In this section, we aim to explain all the code since we reuse the same functions for all tasks.

We started our work from the handout 1 code (we hope that this is not considered plagiarism). We copied most of the code except the preprocessing as all the features are within the same range. We change the **do_train** function in order to input the hidden layers and the corresponding number of neurons per layer as a list. We changed the corresponding area of creating Dense layers so that it reads the list of number of hidden layers neurons and construct hidden layers as instructed. In addition, we added several arguments for optimizer, weight initializer, activation functions of hidden layers, activation of output layer, epochs, schedule callback, and batch size. This is so that we can easily perform various experiments in other task. We give the default weight initializer and optimizer to be Uniform and Stochastic Gradient Descent (SGD) respectively. In addition, we add an EarlyStopping callback to save and restore the best weights as we run the entire 300 epochs of training. Our defualt hidden layer activation function is sigmoid while softmax (basically a generalized sigmoid) is used for the output layer activation function because we have more than 2 categories to predict.

Our main functions code is listed in the appendix section since it is quite long but you can also refer to the notebook submission for clarity. **do_train** function are changed as described in previous paragraph. On the other hand, **do_show** mostly remains unchanged except for the fact that we added **extra_metrics** flag as a parameter. When this flag is true, the function will call SKlearn's **classification_report** function to provide accuracy, precision, recall and f1 metrics. The **make_step_scheduler** and **make_poly_scheduler** and **make_expo_scheduler** functions are added for dynamic learning rate.

The mathematical form of step scheduling is:

$$step\_lr = lr0 * d^{\frac{ep}{ed}} \tag{1}$$

where d is drop factor per step, ed is the every number of epoch to step down, ep is the current epoch, and $lr0$ is the initial learning rate. It should be noted that fraction $\frac{ep}{ed}$ for the power in this equation is floored (I don't know how to represent it in the equation). The mathematical form of polynomial decay scheduling is:

$$poly\_lr = lr0 * ((1 - \frac{ep}{te})^p) \tag{2}$$

where te is the total number of epochs and p is the power where 1 indicates linear decay for example.

The mathematical form of exponential decay scheduling is:

$$expo\_lr = lr0 * (e^{lr0*(ea-ep)}) \tag{3}$$

where e is the natural number and ea is the the number of epochs to wait before applying this exponential decay. In our experiments, we wait for 150 epochs before applying the above formula, the learning rate before 150 epochs is static.

Each schedule is added as a callback to the model. Figure 1 shows the learning rate schedules we used for task 5 for model m7, m8, and m9.

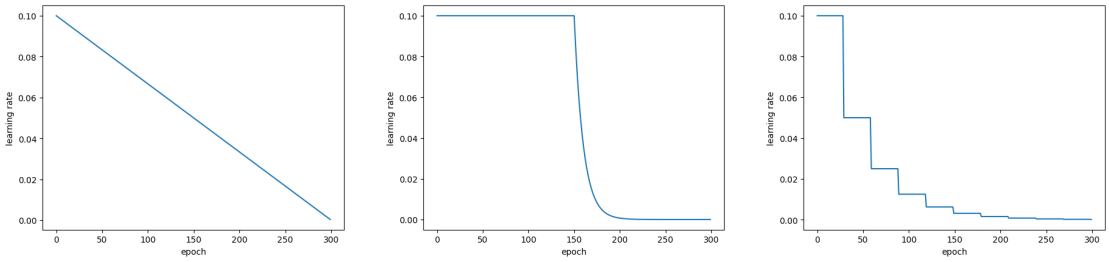

Figure 1: Different schedules used in task 5. From left to right are linear decay schedule, exponential decay schedule, and step decay schedule where the drop factor (d) is 0.5 and the drop happens every 50 steps

For the loading of data, since the files are in csv, we use Pandas to load these data as Dataframe object. The Dataframe class is a very flexible idea as it can transform the data to regular list or numpy objects and provides various other functionalities. We have 4 sets of data: combined, first split, second split, and third split. The combined data basically concatenate all dataframes of first, second and third splits together. The code is listed below:

Listing 1: Code for loading data.

```
import zipfile
import pandas as pd


with zipfile.ZipFile("ds05.zip", "r") as zip_ref:
      zip_ref.extractall('./dataset')

train_split_1 = pd.read_csv('dataset/ds1-05-1-nn-tr.csv',
    index_col=None)
test_split_1 = pd.read_csv('dataset/ds1-05-1-nn-te.csv',
    index_col=None)

train_split_2 = pd.read_csv('dataset/ds1-05-2-nn-tr.csv',
    index_col=None)
test_split_2 = pd.read_csv('dataset/ds1-05-2-nn-te.csv',
    index_col=None)
```

```
13
14 train_split_3 = pd.read_csv('dataset/ds1-05-3-nn-tr.csv',
       index_col=None)
15 test_split_3 = pd.read_csv('dataset/ds1-05-3-nn-te.csv',
       index_col=None)
16
17 train_df = pd.concat(
18     [pd.read_csv('dataset/ds1-05-1-nn-tr.csv', index_col=None),
19      pd.read_csv('dataset/ds1-05-2-nn-tr.csv', index_col=None),
20      pd.read_csv('dataset/ds1-05-3-nn-tr.csv', index_col=None)]
21 )
22 test_df = pd.concat(
23     [pd.read_csv('dataset/ds1-05-1-nn-te.csv', index_col=None),
24      pd.read_csv('dataset/ds1-05-2-nn-te.csv', index_col=None),
25      pd.read_csv('dataset/ds1-05-3-nn-te.csv', index_col=None)]
26 )
27
28 train_X = train_df[train_df.columns[:-1]]
29 train_Y = train_df[train_df.columns[-1]]
30 test_X = test_df[test_df.columns[:-1]]
31 test_Y = test_df[test_df.columns[-1]]
32
33 train_X1 = train_split_1[train_split_1.columns[:-1]]
34 train_Y1 = train_split_1[train_split_1.columns[-1]]
35 test_X1 = test_split_1[test_split_1.columns[:-1]]
36 test_Y1 = test_split_1[test_split_1.columns[-1]]
37
38 train_X2 = train_split_2[train_split_2.columns[:-1]]
39 train_Y2 = train_split_2[train_split_2.columns[-1]]
40 test_X2 = test_split_2[test_split_2.columns[:-1]]
41 test_Y2 = test_split_2[test_split_2.columns[-1]]
42
43 train_X3 = train_split_3[train_split_3.columns[:-1]]
44 train_Y3 = train_split_3[train_split_3.columns[-1]]
45 test_X3 = test_split_3[test_split_3.columns[:-1]]
46 test_Y3 = test_split_3[test_split_3.columns[-1]]
```

2 Task 1

We begin with single layers experiment with default uniform weight initializer, stochastic gradient descent optimizer, a batch size of 10, a learning rate of 0.1, and the number of epochs of 300. The 3 models have different number of neuron for a single hidden layer. They are 8, 12 and 32 respectively. The number 12 is chosen because the size of the input layer (or the number of features) is 125. According to the slides in the class, $\frac{n}{10}$ is an empirically good number of neurons for single hidden layers. Meanwhile, 8 and 32 were chosen because they are powers of 2.

The Table 1 shows that the test accuracy does not differ much between each model and the best models is m1. However, we use the config of m3 instead because from our extended testing, m1 does not perform as well in our further tasks.

Model Name	Hidden Layers	Test Accuracy
m1	8	0.952095
m2	12	0.940119
m3	32	0.946107

Table 1: Best testing accuracy obtained for different models (m1, m2, m3) for task 1.

3 Task 2

Since m1 and m3 have extremely similar performance, we selected m3 as the model to continue since it has larger number of neurons. For this task, we experimented with different activation functions for the hidden layers: Relu, Tanh, Leaky Relu. We use Relu to avoid vanishing gradient problems while trying out Tanh and Leaky Relu as alternatives in case Relu does not offer better performance.

Model m4 (Relu) and m5 (Tanh) offer the best performance and an improvement from the previous model, m3. For this test, we kept the epochs to be at 300 while the learning rate is lowered to 0.025 after some testing. In addition, for the sake of fair comparison, we also added m3x which is m3 but trained at the learning rate of 0.025. It should be noted that the result of m4 that we are continuing with is very similar to m1 in the first task. Our extended testing lead us to believe that m1 is less robust due to lower number of neurons and this is simply a statistical anomaly that it manages to get a better result than m3 and therefore we opted to go with 32 number of neurons for hidden layers for the next few tasks.

Relu basically changes the negative values to 0 and keep positive values as is and can be represented as follows:

$$f(x) = max(0, x) \tag{4}$$

Tanh is similar to sigmoid but the range is from -1 to 1 instead of 0 to 1:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{5}$$

Leaky Relu, instead of changing negative values to 0, multiply them by 0.01 and can be represented as follows:

$$f(x) = max(0.01x, x) \tag{6}$$

Model Name	Hidden Layers	HL Activation Function	Test Accuracy	lr0
m3	32	Sigmoid	0.946107	0.1
m3x	32	Sigmoid	0.910179	0.025
m4	32	Relu	0.952095	0.025
m5	32	Tanh	0.952095	0.025
m6	32	Leaky Relu	0.937125	0.025

Table 2: Best testing accuracy obtained for different models (m4, m5, m6) for task 2. (HL = Hidden Layers)

4 Task 3

Since model m4 has the best performance in task 2, we shall continue with this parameter. In this section, we experimented with 3 kinds of schedules for dynamic learning rate as detailed in task 0 and Figure 1 represents the schedules we are using: linear decay, cube decay, and step decay. Our extensive results in Table 3 shows learning rate scheduling do not improve performance for our case. Therefore, we keep m4 and move forward to the next task. We also perform extended experiments it learning rate of 0.1 because we believe that the learning rate was decaying too quickly for m7 and m9. However, the result here remains inferior to the m4 and m1 result.

Model Name	Learning Rate Schedule Type	lr0	Test Accuracy
m7	Linear	0.025	0.865269
m8	Cube Decay	0.025	0.94910
m9	Step	0.025	0.886227
m7x	Linear	0.1	0.934131
m8x	Cube Decay	0.1	0.949101
m9x	Step	0.1	0.94011

Table 3: Best testing accuracy obtained for different models (m7, m8, m9) for task 3.

5 Task 4

In this experiment, we continue using exponential decay schedule for learning rate. Our first model, m10, changes SGD to add momentum (nesterov) 0.1 with decay rate of 0.1. Our second model, m11, uses RMSProp with rho of 0.9 and a momentum of 0.1. Our third model, m12, uses Adam. Finally, our final model, m13, uses Adagrad. We also change the learning rate for each experiment because they have different convergence time to the problem. The initial learning rate is thus reported in the table. Our result shows that m11 is the best and shows minor improvement on the test accuracy, therefore we will continue to the next task with m11 configuration.

Model Name	Optimizer	lr0	Test Accuracy
m10	SGD Momentum	0.025	0.934131
m11	RMSProp	0.001	0.958083
m12	Adam	0.001	0.949101
m13	Adagrad	0.01	0.92814

Table 4: Best testing accuracy obtained for different models (m10, m11, m12, m13) for task 4.

6 Task 5

In this experiment, we tested Sparse Categorical Cross Entropy (SCCE) where keras expects our ground truth to be provided as integer instead of one-hot encoding. This means that we did not convert our label to one-hot encoding in the code. This is indicated on the if clause before the fit function called. Sparse Categorical Cross Entropy is good in the case where we need to save our memory (RAM) usage at the cost of label preciseness. CCE is generally preferred for its numerical stability and precision since the model's output probabilities are directly compared against clear fixed standard for each class. This also means that we expect the model with this loss function to perform worse than the one with categorical cross entropy (CCE). The result is shown in Table 5 and it is as we expect. m14 which uses Sparse Categorical Cross Entropy performs at approximately 1.5% worse in accuracy than m11 which uses categorical cross entropy.
CCE follows the following formula:

$$H(y, \hat{y}) = -\sum_{i}^{C} y_i \log(\hat{y}_i) \tag{7}$$

where:

- y is the true label in one-hot encoded form.
- $\hat{y}$ is the predicted probability distribution.
- y_i is the true probability for class i.
- $\hat{y}_i$ is the predicted probability for class i
- C is basically all classes.

SCCE follows the following formula:

$$H(y, \hat{y}) = -\sum_{i} \delta(y, i) \log(\hat{y}_i) \tag{8}$$

where:

- y is the true label as a single integer.
- $\hat{y}$ is the predicted probability distribution.
- $\delta(y, i)$ is the Kronecker delta function, which is 1 when $y = i$ and 0 otherwise.

- $\hat{y}_i$ is the predicted probability for class i.
- C is basically all classes.

Model Name	Loss function	Test Accuracy
m11	Categorical Cross Entropy	0.958083
m14	Sparse Categorical Cross Entropy	0.943113

Table 5: Best testing accuracy for m14 compared with m11.

7 Task 6

Here, we continue to use the setting of m11 but we change the number of hidden layers. Table 7 show the result of different models when added second layers of varying sizes. The format is of the hidden layers are indicated as follow: [number of neurons in hidden layer 1, number of neurons in hidden layer 2]. Since we are extending from m11, the number of neurons in hidden layer 1 is kept the same at 32. The result is that none of the new configurations perform better than m11.

In this task, Early Stopping comes very handy since convergence is reached early (please refer to the notebook for loss graphs).

Model Name	Hidden Layer Sizes	Test Accuracy
m11	[32]	0.958083
m15	[32, 16]	0.901197
m16	[32, 24]	0.922155
m17	[32, 32]	0.919161

Table 6: Best testing accuracy obtained for different models (m15, m16, m17) when added second layers to m11 for task 6.

8 Task 7

In task 7, we uses the following batch sizes: 8, 16, 32 , 64. Our default batch size is 10. We try to use power of 2 again because we hope it can fit the computer memory better. It is surprising that non of them provides better accuracy than batch size of 10. This goes to show that while batch size of power of 2 provides computational speed up due to better RAM alignment, they may not provide better accuracy. Since we did not get many improvements, we added m22 which uses HeNormal weight initializer and obtain an improvement. The below table shows the accuracy comparison:

Model Name	Batch Size	Initializer	Test Accuracy
m11	10	Uniform	0.958083
m18	8	Uniform	0.9461078
m19	16	Uniform	0.919161
m20	32	Uniform	0.952095
m21	64	Uniform	0.937125
m22	10	HeNormal	0.961077

Table 7: Best testing accuracy obtained for different models (m18, m19, m20, m21, m22) when changing batch size (and for m22 - initializer) for task 7.

9 Task 8

9.1 Task 8.1

Here, we train each split and named them m23, m24, and m25 for split 1, 2, and 3 respectively. Our convergence is shown in the 3 Figures below in the order of m23, m24, and m25. We know that the convergence is achieved because the validation loss curve on the left image of each plot has plateu. The accuracy graph for each plot is to provide more information.

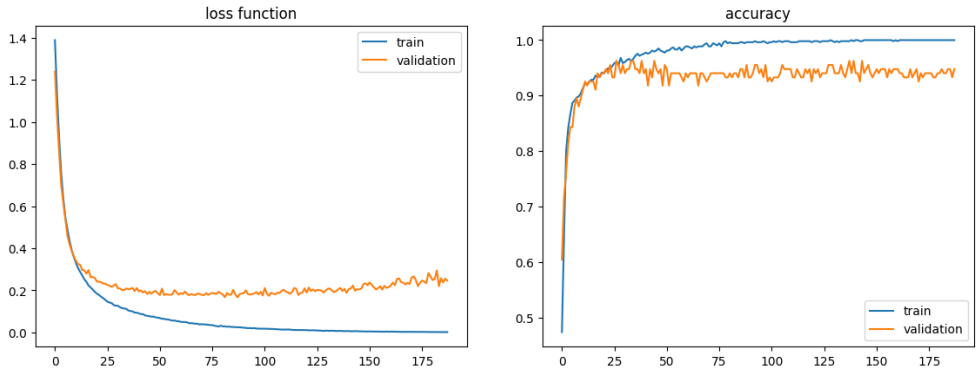

Figure 2: Model m23's loss(left) and accuracy(right) graphs where m23 is trained on split 1.

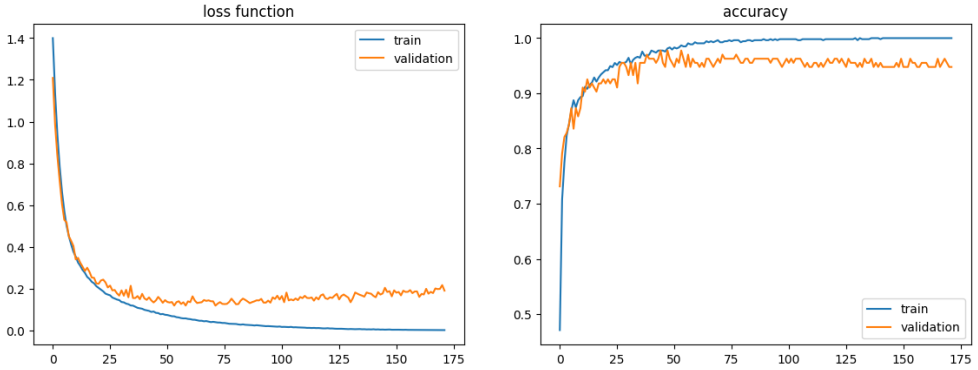

Figure 3: Model m24's loss(left) and accuracy(right) graphs where m24 is trained on split 2.

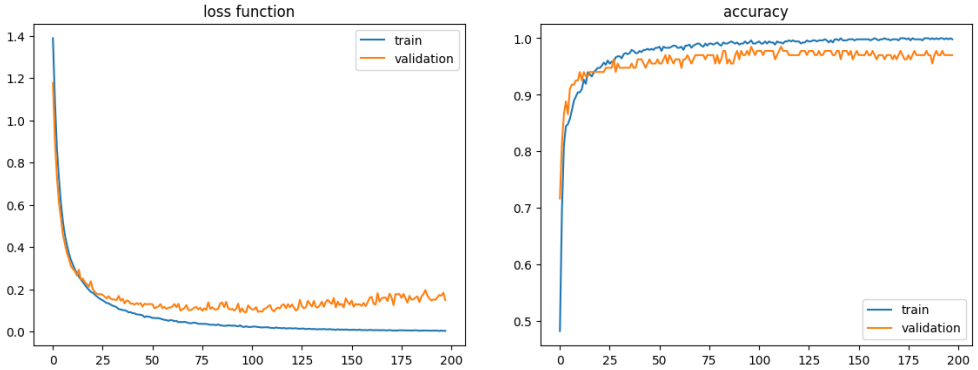

Figure 4: Model m25's loss(left) and accuracy(right) graphs where m25 is trained on split 3.

9.2 Task 8.2

Table 8, 9, 10, 11, 12, and 13 show the confusion matrix and classification report. The reported accuracy is reported for overall while recall, precision, and f1-score is reported for each label and overall average.

Ground Truth	Predicted Labels				
	0	1	2	3	4
0	62	4	0	0	0
1	3	56	5	3	0
2	0	0	65	0	1
3	0	0	0	67	0
4	0	0	2	0	65

Table 8: Confusion Matrix for m23, ith row being the ground truth and jth column being the predicted labels.

	Precision	Recall	F1-Score	Support
0	0.97	0.96	0.96	67
1	0.94	0.93	0.93	67
2	0.94	0.95	0.95	66
3	0.97	0.99	0.98	67
4	0.99	0.99	0.99	67
Macro avg	0.96	0.96	0.96	334
Accuracy			0.96	334

Table 9: Reporting of m23's precision, recall, and f1-score per label and overall average (macro avg). Accuracy is reported as overall.

Ground Truth	Predicted Labels				
	0	1	2	3	4
0	63	2	2	0	0
1	0	64	2	0	0
2	0	1	62	1	3
3	0	0	0	67	0
4	1	1	2	0	59

Table 10: Confusion Matrix for m24, ith row being the ground truth and jth column being the predicted labels.

	Precision	Recall	F1-Score	Support
0	0.98	0.94	0.96	67
1	0.94	0.97	0.96	66
2	0.87	0.93	0.90	67
3	0.99	1.00	0.99	67
4	0.95	0.89	0.92	66
Macro avg	0.95	0.95	0.95	333
Accuracy			0.95	333

Table 11: Reporting of m24's precision, recall, and f1-score per label and overall average (macro avg). Accuracy is reported as overall.

Ground Truth	Predicted Labels				
	0	1	2	3	4
0	64	1	1	0	0
1	1	54	6	3	0
2	0	1	66	0	0
3	2	0	0	64	0
4	0	0	2	0	65

Table 12: Confusion Matrix for m25, ith row being the ground truth and jth column being the predicted labels.

	Precision	Recall	F1-Score	Support
0	0.96	0.97	0.96	66
1	0.97	0.85	0.90	67
2	0.89	0.99	0.94	67
3	0.94	0.97	0.96	66
4	1.00	0.97	0.98	67
Macro avg	0.95	0.95	0.95	333
Accuracy			0.95	333

Table 13: Reporting of m25's precision, recall, and f1-score per label and overall average (macro avg). Accuracy is reported as overall.

9.3 Task 8.3

The average for each metric across all splits are:

- Accuracy: 0.9520
- Precision: 0.9530
- Recall: 0.9520
- F1-score: 0.9519

The code for getting average metric is reported below:

Listing 2: Code to calculate average metric across 3 splits.

```
sum_metrics = {
    'accuracy':0.0,
    'precision': 0.0,
    'recall': 0.0,
    'f1-score':0.0
    }


# Parse classification reports and calculate average
# cr1, cr2, cr3 are classification report returned from do_show with
# extra_metrics=True
for cr in [cr1, cr2, cr3]:
  sum_metrics['accuracy'] += cr['accuracy']
  for metric_name, metric_value in cr['macro avg'].items():
    if metric_name == 'support':  # Skip support
      continue

    sum_metrics[metric_name] += metric_value

average_metrics = {key: value / 3 for key, value in
    sum_metrics.items()}

# print
print('Average metrics for all splits:')
for key, value in average_metrics.items():
    print(f'   {key.capitalize()}: {value:.4f}')
```

10 Conclusion

This project shows that, depending on the task, simple single hidden layer FFN can achieve a very good score compared to their multiple hidden layers counterpart. In addition, because of randomization of initial weights, different reruns can achieve different accuracy. While our report is not according to your template, we believe that this is a better way to report on the project (although it did take a lot of work to make this). We hope that you are lenient in us not following the report template.

For the code, as mentioned before, the appendix shows the main functions used in this code. We did not list all the function calls for each experiment because they are only 2-3 lines of code per experiment due to how the code is structured and there are nothing to comment for these experiment runs. We believe it would be easier for you to check the notebook.

11 Appendix

11.1 Main Functions

Listing 3: Main Functions used for all tasks.

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

from sklearn.datasets import load_iris
from sklearn.metrics import confusion_matrix, accuracy_score,
    classification_report
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

import tensorflow
from tensorflow import keras
from tensorflow.keras import layers

from keras.utils import to_categorical


def do_train(X_train, y_train, X_test, y_test, nh,
             initializer=None,
             opt=None,
             hidden_layer_activation_function='sigmoid',
             output_layer_activation_function='softmax',
             loss='categorical_crossentropy',
             epochs=100,
             batch_size=10,
             schedule_callback=None,
             learning_rate=0.01):
    # nh has to be number of neurons for each layer, len(nh) > 0
    assert len(nh) > 0, "there is no hidden layers indicated!"

    # feature size
    L = X_train.shape[1]
    # output size
    M = len(np.unique(y_train))

    print(f'hidden layer sizes {nh}')
    print('%d classes, %d features' % (M, L))

    # weight initializer object
    if initializer is None:
        # default most basic initializer learnt from class (Uniform)
        initializer = keras.initializers.RandomUniform(minval=-0.05,
            maxval=0.05, seed=123)

    # MODEL SPECIFICATION
```

```
    # input layers
    inputs = keras.Input(shape=(L,))
    current_layer = inputs
    # hidden layers
    for layer_size in nh:
      current_layer = layers.Dense(layer_size,
          activation=hidden_layer_activation_function,
          kernel_initializer=initializer)(current_layer)
    outputs = layers.Dense(M,
        activation=output_layer_activation_function,
        kernel_initializer=initializer)(current_layer)

    model = keras.Model(inputs, outputs)
    model.summary()

    # CALLBACKS FOR DYNAMIC SCHEDULING
    # add early stopping to save the best weights
    early_stopping_monitor = EarlyStopping(
        monitor='val_loss',
        min_delta=0,
        patience=100,
        verbose=0,
        mode='auto',
        baseline=None,
        restore_best_weights=True
    )
    callbacks = [early_stopping_monitor]
    if schedule_callback is not None:
      callbacks.append(keras.callbacks.LearningRateScheduler(schedule_callback))



    # OPTIMIZER CREATION
    if opt is None:
        opt = keras.optimizers.SGD(learning_rate=learning_rate)

    # MODEL COMPILATION
    model.compile(optimizer=opt, loss=loss, metrics=['accuracy'])

    # MODEL FITTING
    if loss=='categorical_crossentropy':
      y_train = to_categorical(y_train)
    verbosity = 0 # "auto"

    XTraining, XValidation, YTraining, YValidation =
        train_test_split(X_train,y_train,test_size=0.2,
        random_state=123)

```

```
85      history = model.fit(XTraining, YTraining, epochs=epochs,
            batch_size=batch_size,
86                          verbose=verbosity,validation_data=(XValidation,
                                YValidation),
87                          workers=16, use_multiprocessing=True,
                                callbacks=callbacks)
88
89      return model, history
90
91  # polynomial scheduler
92  def make_poly_scheduler(total_epoch, power):
93      def poly_schduler(epoch, lr):
94          # reference:
                https://stackoverflow.com/questions/69899602/linear-decay-as-learning-rate-sch
95          return lr * ((1 - float(epoch) / total_epoch) ** power)
96      return poly_schduler
97
98  # step scheduler
99  def make_step_scheduler(drop, epoch_drop):
100     def step_scheduler(epoch, lr):
101         # reference:
                https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-r
102         return lr * math.pow(drop, math.floor((1+epoch)/epoch_drop))
103     return step_scheduler
104
105 # exponential scheduler
106 def make_expo_scheduler(epoch_to_apply):
107     def expo_scheduler(epoch, lr):
108         if epoch <= epoch_to_apply:
109             return lr
110         else:
111             return lr * tensorflow.math.exp(lr * (epoch_to_apply -
                    epoch))
112     return expo_scheduler
113
114
115 def to_label(M, y):
116     if M == 2:
117         # round predictions
118         y_ = [round(x[0]) for x in y]
119     else:
120         # take max output
121         y_ = np.argmax(y, axis=1)
122     return y_
123
124 def plot_class(c, X, y):
125     # plot samples of 'X' for class 'c'
126
127     i = np.where(y == c)[0]
```

```
128     plt.scatter(X[i,0],X[i,1])
129
130 def show_class_map_(model, X, y):
131     # this function does not do anything for our current problem and
            can be removed
132
133     if X.shape[1] != 2:
134         print('This dataset is not two-dimensional ...')
135         return
136
137     M = len(np.unique(y))
138
139     # plot data
140     plt.figure()
141     for c in range(M):
142         plot_class(c, X, y)
143     plt.axis('equal')
144
145     # plot the decision boundaries
146     ax = plt.gca()
147     xlim = ax.get_xlim()
148     ylim = ax.get_ylim()
149
150     # create the grid to evaluate the model at discrete feature space
            points
151     xx = np.linspace(xlim[0], xlim[1], 100)
152     yy = np.linspace(ylim[0], ylim[1], 100)
153     YY, XX = np.meshgrid(yy, xx)
154     xy = np.vstack([XX.ravel(), YY.ravel()]).T
155     ZZ_ = model.predict(xy)
156     ZZ  = to_label(M, ZZ_)
157
158     # plot the boundaries
159     for c in range(M):
160         ax.contour(XX, YY, ZZ_[:,c].reshape(XX.shape), levels=[0.5],
                alpha=0.5, linestyles=['--'])
161
162     plt.xlabel('x1')
163     plt.ylabel('x2')
164     plt.axis('equal')
165     plt.title('contour map')
166     plt.savefig('results/ic_lab1_c.png')
167     plt.show()
168
169     # plot the classification map
170     plt.figure()
171     plt.imshow(ZZ.reshape(XX.shape).T, origin='lower',
            extent=(xlim[0], xlim[1], ylim[0], ylim[1]), cmap='RdYlGn')
172     plt.colorbar()
```

```
173     for c in range(M):
174         plot_class(c, X, y)
175     plt.xlabel('x1')
176     plt.ylabel('x2')
177     plt.axis('equal')
178     plt.title('classification␣map')
179     plt.savefig('results/ic_lab1_m.png')
180     plt.show()
181 
182 def do_show(model, history, X_train, y_train, X_test, y_test,
        y_to_categorical=True, extra_metrics=False):
183 
184     # get loss and accuracy from training history object
185     loss = history.history['loss']
186     val_loss = history.history['val_loss']
187     accuracy = history.history['accuracy']
188     val_accuracy = history.history['val_accuracy']
189     print(accuracy[-1], val_accuracy[-1])
190 
191     # plot losses
192     plt.figure()
193     plt.plot(loss, label='train')
194     plt.plot(val_loss, label='validation')
195     plt.title('loss␣function')
196     plt.legend()
197     plt.show(block=False)
198 
199     # plot accuracy
200     plt.figure()
201     plt.plot(accuracy, label='train')
202     plt.plot(val_accuracy, label='validation')
203     plt.title('accuracy')
204     plt.legend()
205     plt.show(block=False)
206 
207     # predict and print confusion matrix on training set
208     M = len(np.unique(y_train))
209 
210     y_pred_ = model.predict(X_train)
211     y_pred = to_label(M, y_pred_)
212     cm = confusion_matrix(y_train, y_pred)
213     print('Confusion␣matrix␣for␣train:')
214     print(cm)
215     print('accuracy␣=␣%f' % (accuracy_score(y_train, y_pred)))
216 
217     y_train_ = to_categorical(y_train) if y_to_categorical else
            y_train
218 
219     score = model.evaluate(X_train, y_train_, verbose=0)
```

```
    print("Train␣loss:", score[0])
    print("Train␣accuracy:", score[1])

    show_class_map_(model, X_train, y_train)

    # predict and print confusion matrix on test set
    y_pred_ = model.predict(X_test)
    y_pred = to_label(M, y_pred_)
    cm = confusion_matrix(y_test, y_pred)
    print('Confusion␣matrix␣for␣test:')
    print(cm)
    print('accuracy␣=␣%f' % (accuracy_score(y_test, y_pred)))

    y_test_ = to_categorical(y_test) if y_to_categorical else y_test

    score = model.evaluate(X_test, y_test_, verbose=0)
    print("Test␣loss:", score[0])
    print("Test␣accuracy:", score[1])

    if extra_metrics:
      print('\nClassification␣Report:')
      print(classification_report(y_test, y_pred))

      return classification_report(y_test, y_pred, output_dict=True)
```