ANN Classifier Assignment

Artificial Intelligence - Spring 2021

Thomas Trankle

March 4, 2021

Summary

This report describes the development of an artificial neural network to be used as a classifier for predicting the quality level of various Portuguese white wines. Overall, the best model that I trained had an accuracy of 0.90, but the average Mean Squared Error produced throughout the evaluation was around 0.87. The introduction describes the modifiable parameters of this basic artificial neural network that is built using a Sequential model. Additionally, it describes the input features/columns used in predicting the quality of wine. The body of the report steps through necessary preprocessing procedures for the model and data, as well as the main adjustments made through the development and evaluation of the model. Throughout the paper, a decision to only train the model based on the available classes in the data set provided was chosen.

Introduction

The purpose of this report is to provide a detailed explanation of creating a Artificial Neural Network to predict wine quality based on 11 features. This implementation uses a classification model since the output variable, which is the quality of the wine, can be considered a quality level between 0 and 10. However, in the data, the predication labels only contain the quality levels from 3 to 9. Therefore, we will only be trying to predict output classes in this range. Ideally, we want to be able produce prediction labels matching the actual output variables exactly. For this reason, the overall goal is maximize the prediction accuracy of our model, that is the number of correctly predicted labels over the total number of attempted predictions.

$$Accuracy = \frac{TruePredictions}{TruePredictions + FalsePredictions}$$

The data set, which can be found here , contains 4,898 observations and 11 features, namely:

1. fixed acidity
2. volatile acidity
3. citric acid
4. residual sugar
5. chlorides
6. free sulfur dioxide
7. total sulfur dioxide
8. density
9. pH
10. sulphates
11. alcohol

The objective is to train and evaluate a neural network that uses these features to accurately predict which class of wine quality a wine belongs to. To do this, I selected using Categorical Cross

Entropy as loss metric, since it is very intuitive and common metric when evaluating the performance of a multi-class classification model. The formula is defined as follows,

$$CE = -log(\frac{e^{s_p}}{\sum_j^C e^{s_j}})$$

Additionally, I used the *Softmax* as my activation function for my output layer since this is considered best practice for models with our criteria. When creating the model, there is no procedural method to define the specific parameters of our model to optimize our objective function. Instead, we have to experiment with the different parameters to determine what parameters, at least on average, can produce a sufficient result. However, we do utilize a Grid Search approach to try and best select the hyper-parameters to optimize the model. To solve this black box architecture with reasonable and efficiency and accuracy, we will tune the following parameters:

- Optimizer (using Grid Search)
- Number of Layers
- Number of Nodes per Layer
- Learning rate of the specified optimizer
- Activation Function of each respective Layer (using Grid Search)
- Batch Size (specified in compile method)
- Number of Epochs
- Regularizers
- Output Distribution

There are certainly more details and/or parameters that could be added to the above list, however, for the purpose of this exercise they were omitted since they are above the necessary scope. Additionally, a couple of common best practices and preprocessing steps were held constant when designing, tuning and evaluating the model. The first was checking to see that the data provided contained no *null* values, even though this information is specified in the data description. After this was verified, the next step was to transform our output variables using a label encoder. Simply put, since our output range is from 3 to 7, we create aliases for each of these numbers, that is

Original Class	Encoded Class
3	0
4	1
5	2
6	3
7	4
8	5
9	6

After which, we one-hot encode each encoded class as follows,

$$
\begin{aligned}
0 &\rightarrow [1,0,0,0,0,0,0] \\
1 &\rightarrow [0,1,0,0,0,0,0] \\
2 &\rightarrow [0,0,1,0,0,0,0] \\
3 &\rightarrow [0,0,0,1,0,0,0] \\
4 &\rightarrow [0,0,0,0,1,0,0] \\
5 &\rightarrow [0,0,0,0,0,1,0] \\
6 &\rightarrow [0,0,0,0,0,0,1]
\end{aligned}
$$

Next, we split the data into **train** and **test** sets. Lastly, before we pass the data to our model, we center and scale the predictor variable (X features).

Model Development and Paramter Tuning

Preprocessing

The data is provided in a *.csv* file. However, the delimiter used to separate the columns is a semi-colon. Therefore, the first step in the preprocessing phase was to split and clean the data to make sure it was in the appropriate format. To do this, I simply imported the data file, called *winequality-white.csv* using a very popular Python package called *pandas*. The package has a *read_csv* function that I used to load the data into my environment. Next I made use of an in-built Python tool, list comprehensions, to split the data into the appropriate index positions (as seen in the data). The predictor features are assigned to the *X* variable which is a list of list while the outcome is assigned to the *Y* variable that is just a single list.

```
X = np.array([[float(x) for x in dataframe.values[i][0].split(';')[0:11]]
                                        for i in range(len(dataframe))])
Y = [int(data.values[i][0].split(';')[-1]) for i in range(len(data))]
```

The package *keras*, we use to create a neural network, prefers to deal with an array data type as opposed to a list, in order to optimize its computational efficiency. So, we import and use another popular Python package called *numpy* to transform the X and Y variables into array data types (matrix) and ensure they are both floating values.

```
X = np.array(X).astype('float32')
Y = np.array(Y).astype('float32')
```

Next, we use a common machine learning package called *sklearn* , to create nominal target labels with values between 0 and 6, and split our data into train and test sets.

```
Y = preprocessing.LabelEncoder().fit_transform(Y)
X_train, X_test, Y_train, Y_test = train_test_split(X,Y,test_size = 0.2)
```

Following this, we one-hot encode our Y sets and import a normalizer called StandardScaler which centers and scales our input columns by removing the mean and scaling to unit variance.

```
Y_train = to_categorical(Y)
Y_test = to_categorical(Y_test)
scaler = StandardScaler()
X_train = scaler.fit_transform(X)
X_test = scaler.fit_transform(X_test)
```

*A preprocessing step that was later implemented during the model development, was stratifying the Y variables during the split and using class weights to adjust for the relative distribution of Y variables. Stratifying aims to partition to test and train data with similar label distributions. Class weights is used during model training to penalize the loss function for incorrectly predicting classes with smaller frequencies in the data set. This was done to prevent classes being left out of the test set and prevent the model from underfitting by not being able to train on certain classes. To implement the above, we slightly alter our partition statement and calculate the class distributions using *numpy*

```
X_train, X_test, Y_train, Y_test = train_test_split(X,Y,test_size = 0.
 ↪11,random_state=6464, stratify = Y)
unique, counts = np.unique(Y_train,axis=0, return_counts=True)
counts = counts.max()/counts
class_weight = unique[i]:counts[i] for i in range(len(counts))
```

Course of Action and Model Development

After all the preprocessing steps had been implemented, I decided to create an initial model with one hidden layer. This was done to make sure I was able to pass all the data through in the correct format.

```
## All other parameters have been defined outside this code chunk ##
model = Sequential()
model.add(Dense(11, input_dim=num_cols, activation = 'relu'))
model.add(Dense(20,  activation = 'relu'))
model.add(Dense(7, activation = 'softmax'))
```

This gave me an accuracy of 0.54 using the optimizer Adam, 75 Epochs, ReLu activation function for two layers, and default parameters for the remaining parameter inputs mentioned in the introduction. This initial model took 31.51 seconds to run.

After creating an initial starting point, I wanted to determine the best optimizer to use going forward. So, I ran Grid Search with higher node values and more hidden layers. Additionally, the number of Epochs was set to 100 and we used 10-fold cross validation.

```
## All other parameters have been defined outside this code chunk ##
num_folds = 10
batch = 256
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']

def create_model(optimizer='adam'):
    model = Sequential()
```

```
    model.add(Dense(500, input_dim = 11, activation = activation_func))
    model.add(Dense(254,  activation = 'relu'))
    model.add(Dense(254,  activation = 'relu'))
    model.add(Dense(254,  activation = 'relu'))
    model.add(Dense(7, activation = 'softmax'))

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

# create model
model = KerasClassifier(build_fn=create_model, epochs=num_epochs,␣
 ↪batch_size=batch, verbose=1)
# define the grid search parameters
param_grid = dict(optimizer=optimizer)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=1,
                    cv=num_folds)
```

This resulted in the following results,

Best: 0.624299 using 'optimizer': 'RMSprop'

$0.559974(0.022548)\ with: \{'optimizer' :' SGD'\}$
$0.624299(0.018763)\ with: \{'optimizer' :' RMSprop'\}$
$0.516587(0.031607)\ with: \{'optimizer' :' Adagrad'\}$
$0.455338(0.036370)\ with: \{'optimizer' :' Adadelta'\}$
$0.618684(0.019814)\ with: \{'optimizer' :' Adam'\}$
$0.617414(0.026986)\ with: \{'optimizer' :' Adamax'\}$
$0.614346(0.030033)\ with: \{'optimizer' :' Nadam'\}$

Time required for training: 0:09:57.663054

As you can see, the best optimizer is RMSprop and this search took approximately 10 minutes to run.

After determining my optimizer, I decided to run another Grid Search to determine the best activation function(s) for my model. I decided, in the interest of time, to reduce the number of hidden layers back down to one. I did this because I wanted to determine if it was best to proceed with the same activation function for all layers (except output layer) or if a potential combination could be more beneficial.

```
## All other parameters have been defined outside this code chunk ##
batch = 64
num_folds=10
activation = ['relu','selu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear']
activation2=['relu','selu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear']

def create_model(activation='relu',activation2 ='relu'):
```

```
    # create model
    model = Sequential()
    model.add(Dense(500, input_dim = 11, activation = activation))
    model.add(Dense(254,  activation = activation2))
    model.add(Dense(7, activation = 'softmax'))
    optimizer = 'RMSprop'
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer=optimizer,
                  metrics=['accuracy'])
    return model

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

# create model
model = KerasClassifier(build_fn=create_model, epochs=num_epochs,
                        batch_size=batch, verbose=1)
# define the grid search parameters
param_grid = dict(activation=activation, activation2 = activation2)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=1,
                    cv=num_folds)
```

The results were as follows,

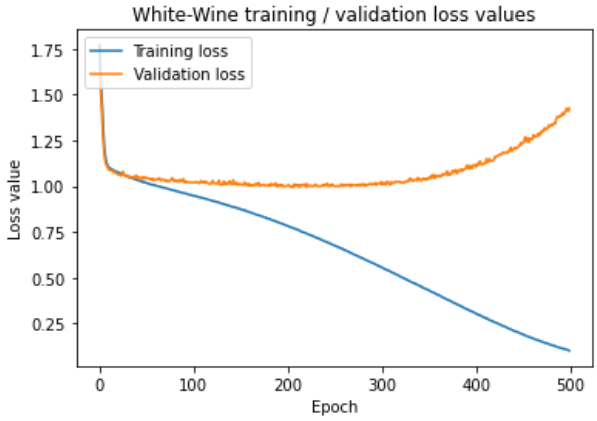
Best: 0.631972 using 'activation': 'tanh', 'activation2': 'tanh'

$$0.606946(0.021184)\ with: {}'activation' :' relu',' activation2' :' relu'$$
$$0.564061(0.028114)\ with: {}'activation' :' relu',' activation2' :' selu'$$
$$0.578609(0.032436)\ with: {}'activation' :' relu',' activation2' :' tanh'$$
$$\vdots$$
$$0.524755(0.029325)\ with: {}'activation' :' linear',' activation2' :' linear'$$

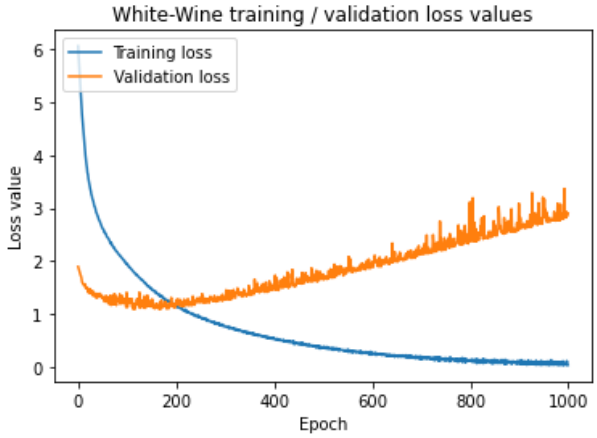
Time required for training: 2:45:14.389007

As you can see, it is seemingly better to stick with the same activation function for this model. After a search run-time of 2 hours and 45 minutes we see that the best activation function was the Hyperbolic Tangent.

I then began to test my optimizer model using the new found activation function and Epochs set to 500. This resulted in a training accuracy of 0.64 and had a run-time of 1 minute and 10 seconds. Unfortunately, even though this was my best accuracy result so far, I noticed that my model was overfitting very quickly.

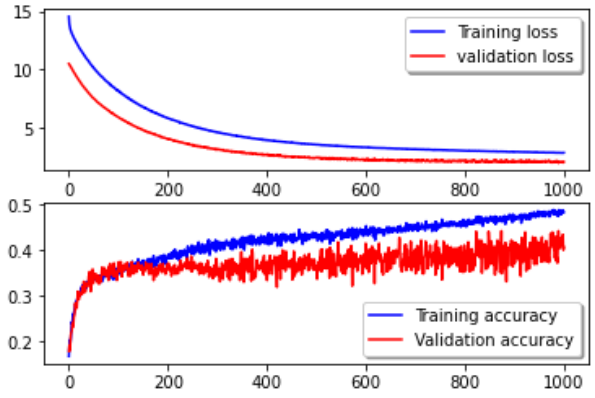


According to our textbook, one of the suggested solutions to this early onset of overfitting, is to decrease the node sizes. However, I attempted this and noticed minimal changes. So, I returned to my old model and decreased my learning rate of my optimizer by an order of 10. To compensate for this change, I increased the Epochs to 1000. This resulted in a run-time of 1 minute and 53.95 seconds, with an accuracy of 0.67. Even though we have only a slight improvement, we notice that the rate at which overfittig happens, does not seem to be a drastic.



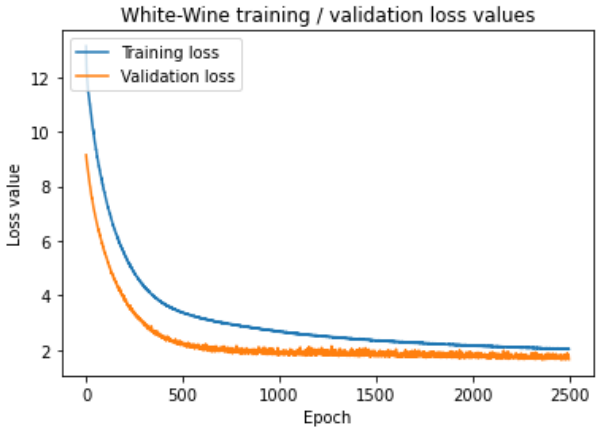
Next, I decided to explore the addition of regularizers in the model, which is another suggested solution in the textbook. Using the standard L2 normalizer, my models accuracy decreased heavily to 0.40, but the the new trend between my training and validation loss showed no signs of overfitting. This model took 2 minutes and 42 seconds to run.

```
num_epochs = 1000
regularizer = regularizers.l2()
model = Sequential()
model.add(Dense(500, input_dim = 11, activation = 'tanh',
                kernel_regularizer = regularizer))
model.add(Dense(254,  activation = 'tanh',kernel_regularizer = regularizer))
model.add(Dense(254,  activation = 'tanh',kernel_regularizer = regularizer))
model.add(Dense(254,  activation = 'tanh',kernel_regularizer = regularizer))
model.add(Dense(7, activation = 'softmax'))
```



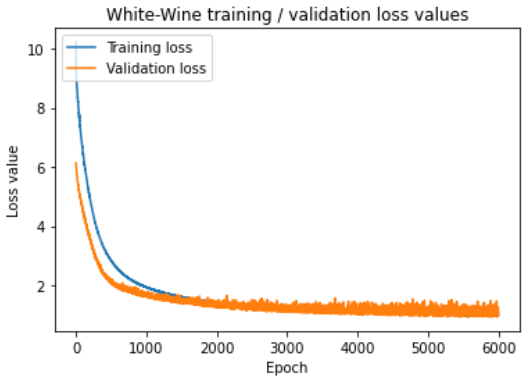
At this point, I liked the look of my new model and determined it simply needed more time to train in order for the loss' functions to converge. So I increased the my Epochs to 2500. This improved my accuracy to 0.47 and increased my run-time to 6 minutes and 44 seconds.

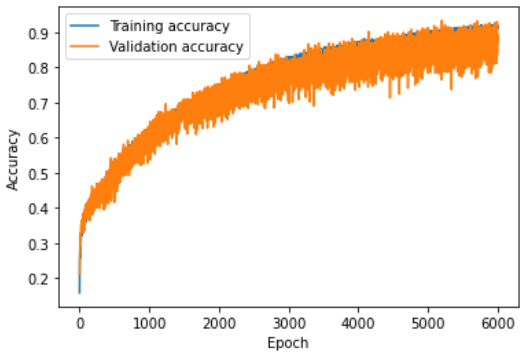
To determine the severity/constraint of adding the regularizer to my model, I decided to decrease it default penalty paramter l2 from 0.01 to 0.0085. This gave me training accuracy of 0.64 and a run-time of 7 minutes and 44 seconds



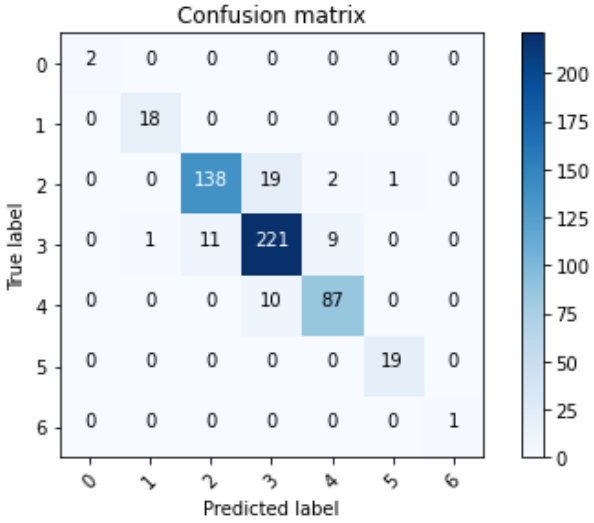
As you can see, the model starts to generate more noise but it is still preventing overfitting. So, I continued to systematically decrease the l2 penalty paramter and alter my Epochs according to allow for sufficient training time. This resulted in the following results:

Accuracy	Epochs	l2 Parameter	Run-time(m:s)
0.706	3500	0.0078	11:01
0.727	5000	0.0078	15:27
0.794	6000	0.0078	20:16
0.814	8000	0.0076	26:23
0.808	6000	0.0074	20:18
0.836	6000	0.0070	23:48
0.871	6000	0.0060	19:38
0.872	6000	0.0050	18:35
0.902	4000	0.0050	15:32





As you can see from the above graph, it was difficult to tell, because of the validation loss noise, if the model was starting to overfit. So I decreased my Epochs to 4000 which resulted in my best training accuracy of 0.901 with a run-time of 15 minutes and 32 seconds. The confusion matrix below shows the results of the best model. The model was able to successfully predict 90% of the output labels but incorrectly assigned 34 labels in the test set.



Conclusion

My final model had 3 hidden layers each with 254 nodes. The input and output layers had 500 and 7 nodes respectfully. Even though we are trying to predict a single class for each observation, it is necessary to have seven output nodes since we are trying to predict a seven dimensional vector. This output vector contains primarily 0's in all cells except one of them that contains 1. Each layer, besides the output layer, made use of the tanh activation function. This function was chosen because it seemed to perform the best in our grid search. The optimizer I selected was RMSProp with a learning rate of 0.0001. The optimizer was chosen as it also performed the best in our grid search for optimizers. Additionally, my final model had a batch size of 256 and 4,000 Epochs. To validate my model held in the general case, I ran it with different train test splits times and complete randomization. The accuracy range seemed to be around 0.86 without training any of the other parameters.

Recommendations and Comments

Originally I was very confused how a model was seemingly overfitting was out performing a converging model. This took me a while to wrap my head around but once I understood how the regularizers work, I was able to get a better model, despite the training time. I do acknowledge that it is difficult to determine how my model would perform on a data set with 10 classes. I was not sure how to fix the range of the label encoder or even whether or not that is considered best practice since we only have 7 classes in this specific data set.

Final Code

```
"""
Import Libraries Section
"""
import datetime
import numpy as np
import pandas as pd
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn import preprocessing
from tensorflow.keras.utils import to_categorical
from sklearn.metrics import confusion_matrix
import itertools
from tensorflow.keras import regularizers

"""
Parameters Section
"""
# first layer is # neurons and # number of inputs -- 10 in 1 out
# set random seed for reproducibility
seed = 6464
np.random.seed(seed)
csv_file='winequality-white.csv' #10 X 11 Y
num_cols=11
num_epochs=4000
optimizer=keras.optimizers.RMSprop(learning_rate=0.0001)
batch = 256
regularizer = regularizers.l2(0.005)

"""
Load Data Section
"""
# Load dataset
dataframe = pd.read_csv(csv_file, header = 0)

"""
Pretreat Data Section
"""
dataframe = dataframe.replace(np.nan,0)

X = [[x for x in dataframe.values[i][0].split(';')[0:11]]
     for i in range(len(dataframe))]
```

```
Y = [dataframe.values[i][0].split(';')[-1] for i in range(len(dataframe))]
X = np.array(X).astype('float32')
Y = np.array(Y).astype('float32')

Y = preprocessing.LabelEncoder().fit_transform(Y)


X_train, X_test, Y_train, Y_test = train_test_split(X,Y,test_size = 0.11,
                                        random_state=6464, stratify = Y)

unique, counts = np.unique(Y_train,axis=0, return_counts=True)
counts = counts.max()/counts
class_weight = {unique[i]:counts[i] for i in range(len(counts))}

Y_train = to_categorical(Y)
Y_test = to_categorical(Y_test)

scaler = StandardScaler()
X_train = scaler.fit_transform(X)
X_test = scaler.fit_transform(X_test)
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Define Model Section
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
start_time = datetime.datetime.now()

model = Sequential()
model.add(Dense(500, input_dim = 11, activation = 'tanh',
                kernel_regularizer = regularizer))
model.add(Dense(254,  activation = 'tanh',kernel_regularizer = regularizer))
model.add(Dense(254,  activation = 'tanh',kernel_regularizer = regularizer))
model.add(Dense(254,  activation = 'tanh',kernel_regularizer = regularizer))
model.add(Dense(7, activation = 'softmax'))

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

model.summary()
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Train Model Section
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
# fix random number seed for repeatability class_weight = class_weight
history = model.fit(X_train, Y_train, epochs=num_epochs, batch_size = batch,
                    verbose=1,class_weight = class_weight,
                    validation_data=(X_test,Y_test))
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Show output Section
```

```
"""
stop_time = datetime.datetime.now()

print('Time required for training:',stop_time - start_time)

# Plot the loss and accuracy curves for training and validation
plt.figure()
fig, ax = plt.subplots(2,1)
ax[0].plot(history.history['loss'], color='b', label="Training loss")
ax[0].plot(history.history['val_loss'],
           color='r', label="validation loss",axes =ax[0])
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['accuracy'], color='b',
           label="Training accuracy")
ax[1].plot(history.history['val_accuracy'], color='r',
           label="Validation accuracy")
legend = ax[1].legend(loc='best', shadow=True)


# Visualize model history - diagnostic plot
plt.figure()
plt.plot(history.history['accuracy'], label='Training accuracy')
plt.plot(history.history['val_accuracy'], label='Validation accuracy')
#plt.title('Training / validation accuracies')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc="upper left")
plt.show()

plt.figure()
plt.plot(history.history['loss'], label='Training loss')
plt.plot(history.history['val_loss'], label='Validation loss')
plt.title('White-Wine training / validation loss values')
plt.ylabel('Loss value')
plt.xlabel('Epoch')
plt.legend(loc="upper left")
plt.show()

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
```

```
    plt.figure()
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Predict the values from the validation dataset
pred_label = model.predict(X_test)
# Convert predictions classes to one hot vectors
pred_label_classes = np.argmax(pred_label,axis = 1)
# Convert validation observations to one hot vectors
label_true = np.argmax(Y_test,axis = 1)
# compute the confusion matrix
confusion_mtx = confusion_matrix(label_true, pred_label_classes)
print(confusion_mtx)
# plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes = range(7))

print('Accuracy: ', sum(confusion_mtx.diagonal())/confusion_mtx.sum())
```

Appebdix

HYPER search for Optimizer

Best: 0.624299 using 'optimizer': 'RMSprop'

0.559974 (0.022548) with: 'optimizer': 'SGD'

0.624299 (0.018763) with: 'optimizer': 'RMSprop'

0.516587 (0.031607) with: 'optimizer': 'Adagrad'

0.455338 (0.036370) with: 'optimizer': 'Adadelta'

0.618684 (0.019814) with: 'optimizer': 'Adam'

0.617414 (0.026986) with: 'optimizer': 'Adamax'

0.614346 (0.030033) with: 'optimizer': 'Nadam'

Time required for training: 0:09:57.663054

HYPER search for Activation

Best: 0.631972 using 'activation': 'tanh', 'activation2': 'tanh'

0.606946 (0.021184) with: 'activation': 'relu', 'activation2': 'relu'

0.564061 (0.028114) with: 'activation': 'relu', 'activation2': 'selu'

0.578609 (0.032436) with: 'activation': 'relu', 'activation2': 'tanh'

0.550800 (0.024072) with: 'activation': 'relu', 'activation2': 'sigmoid'

0.544156 (0.020088) with: 'activation': 'relu', 'activation2': '$\text{hard}_s igmoid'$

0.536514 (0.041454) with: 'activation': 'relu', 'activation2': 'linear'

0.570706 (0.023258) with: 'activation': 'selu', 'activation2': 'relu'

0.521701 (0.022679) with: 'activation': 'selu', 'activation2': 'selu'

0.622011 (0.017180) with: 'activation': 'selu', 'activation2': 'tanh'

0.623542 (0.025932) with: 'activation': 'selu', 'activation2': 'sigmoid'

0.537524 (0.033910) with: 'activation': 'selu', 'activation2': '$\text{hard}_s igmoid'$

0.491589 (0.027878) with: 'activation': 'selu', 'activation2': 'linear'

0.559986 (0.021851) with: 'activation': 'tanh', 'activation2': 'relu'

0.530373 (0.022422) with: 'activation': 'tanh', 'activation2': 'selu'

0.631972 (0.026659) with: 'activation': 'tanh', 'activation2': 'tanh'

0.613068 (0.013563) with: 'activation': 'tanh', 'activation2': 'sigmoid'

0.549269 (0.015567) with: 'activation': 'tanh', 'activation2': '$\text{hard}_s igmoid'$

0.549254 (0.016965) with: 'activation': 'tanh', 'activation2': 'linear'

0.538030 (0.018215) with: 'activation': 'sigmoid', 'activation2': 'relu'

0.485201 (0.051323) with: 'activation': 'sigmoid', 'activation2': 'selu'

0.524243 (0.037214) with: 'activation': 'sigmoid', 'activation2': 'tanh'

0.546707 (0.018137) with: 'activation': 'sigmoid', 'activation2': 'sigmoid'

0.552055 (0.026275) with: 'activation': 'sigmoid', 'activation2': '$\text{hard}_s igmoid'$

0.512253 (0.031978) with: 'activation': 'sigmoid', 'activation2': 'linear'

0.528083 (0.026023) with: 'activation': '$\text{hard}_s igmoid', activation2' : relu'$

0.521956 (0.040223) with: 'activation': 'hard$_s$*igmoid*',' *activation*2' :' *selu*'

0.509441 (0.041987) with: 'activation': 'hard$_s$*igmoid*',' *activation*2' :' *tanh*'

0.544660 (0.021571) with: 'activation': 'hard$_s$*igmoid*',' *activation*2' :' *sigmoid*'

0.547982 (0.026884) with: 'activation': 'hard$_s$*igmoid*',' *activation*2' :' *hard$_s$igmoid*'

0.518365 (0.023329) with: 'activation': 'hard$_s$*igmoid*',' *activation*2' :' *linear*'

0.560997 (0.034741) with: 'activation': 'linear', 'activation2': 'relu'

0.524526 (0.046707) with: 'activation': 'linear', 'activation2': 'selu'

0.611036 (0.028737) with: 'activation': 'linear', 'activation2': 'tanh'

0.610785 (0.024471) with: 'activation': 'linear', 'activation2': 'sigmoid'

0.554634 (0.028700) with: 'activation': 'linear', 'activation2': 'hard$_s$*igmoid*'

0.524755 (0.029325) with: 'activation': 'linear', 'activation2': 'linear'

Time required for training: 2:45:14.389007