EMERGING METHODS FOR EARLY DETECTION OF FOREST FIRES

MODEL BUILDING

CONFIGURING THE LEARNING PROCESS

Date	02 November2022
Team ID	PNT2022TMID31052
Project Name	Emerging Methods for Early Detection of Forest Fires

Configuring The Learning Process:

With both the training data defined and model defined, it's time to configure the learning process. This is accomplished with a call to the compile () method of the Sequential model class. Compilation requires 3 arguments: an optimizer, a loss function, and a list of metrics.

```
In [ ]: #configure the Learning process
        model.compile(loss = 'binary_crossentropy',
                      optimizer = "adam",
                      metrics = ["accuracy"])
```

If you more than two classes in output put “loss = categorical_crossentropy”.

The Experiment of Forest Fires Prediction using Deep Learning:

Forest fires is one of the important catastrophic events and have great impact on environment, infrastructure and human life. For the need of an early warning detection system of forest fires, there are various methods that have been used including : physics-based model, statistical model, machine learning model and deep learning model.

A brief overview of artificial neural networks (ANN):

ANN are made of layers with an input and an output dimension. The latter is determined by the number of **neurons** (also called ‘nodes’), a computational unit that connects the weighted inputs through **activation function** (which helps the neuron to switch on/off). The **weights,** like in most of the machine learning algorithms, are randomly initialized and optimized during the training to minimize a loss function.

Here are the steps to do the experiment:

Step 1: Understanding Dataset

Before we import the dataset, we must import the required libraries.

```
import numpy as np import
pandas as pd import
matplotlib.pyplot as plt
plt.style.use('seaborn')
import seaborn as sns
from sklearn.preprocessing import LabelEncoder, StandardScaler, MinMaxScalerfrom sklearn.model_selection import train_test_split from sklearn.metrics import r2_score import tensorflow as tensorflow from keras.models import Sequential from keras.layers import Dense, Dropout from tensorflow import keras from tensorflow.keras import layers from tensorflow.keras.optimizers import SGD from tensorflow.keras.utils import to_categorical from keras.callbacks import EarlyStopping from keras.callbacks import ModelCheckpoint from keras.utils.vis_utils import plot_model
```

For importing dataset, do this following steps:

```
df = pd.read_csv('dataset.csv')
df.head(10)
```

	X	Y	month	day	FFMC	DMC	DC	ISI	temp	RH	wind	rain	area
0	7	5	mar	fri	86.2	26.2	94.3	5.1	8.2	51	6.7	0.0	0.0
1	7	4	oct	tue	90.6	35.4	669.1	6.7	18.0	33	0.9	0.0	0.0
2	7	4	oct	sat	90.6	43.7	686.9	6.7	14.6	33	1.3	0.0	0.0
3	8	6	mar	fri	91.7	33.3	77.5	9.0	8.3	97	4.0	0.2	0.0
4	8	6	mar	sun	89.3	51.3	102.2	9.6	11.4	99	1.8	0.0	0.0
5	8	6	aug	sun	92.3	85.3	488.0	14.7	22.2	29	5.4	0.0	0.0
6	8	6	aug	mon	92.3	88.9	495.6	8.5	24.1	27	3.1	0.0	0.0
7	8	6	aug	mon	91.5	145.4	608.2	10.7	8.0	86	2.2	0.0	0.0
8	8	6	sep	tue	91.0	129.5	692.6	7.0	13.1	63	5.4	0.0	0.0
9	7	5	sep	sat	92.5	88.0	698.6	7.1	22.8	40	4.0	0.0	0.0

Attribute Information:

- **X** : x-axis spatial coordinate within the Montesinho park map: 1 to 9
- **Y** : y-axis spatial coordinate within the Montesinho park map: 2 to 9
- **month** : month of the year: ‘jan’ to ‘dec’
- **day** : day of the week: ‘mon’ to ‘sun’
- **FFMC** : FFMC (Fine Fuel Moisture Code) index from the FWI system: 18.7 to 96.20
- **DMC** : DMC (Duff Moisture Code) index from the FWI system: 1.1 to 291.3
- **DC** : DC (Drought Code) index from the FWI system: 7.9 to 860.6
- **ISI** : ISI (Initial Spread Index) index from the FWI system: 0.0 to 56.10

- **temp** : temperature in Celsius degrees: 2.2 to 33.30
- **RH** : relative humidity in %: 15.0 to 100
- **wind** : wind speed in km/h: 0.40 to 9.40
- **rain** : outside rain in mm/m2 : 0.0 to 6.4
- **area** : the burned area of the forest (in ha): 0.00 to 1090.84

Step 2: Data Preprocessing

1) Add a new column = size_category

For classification problem, we attempt to add a new column, namely size_category to categorize the data into two categories:

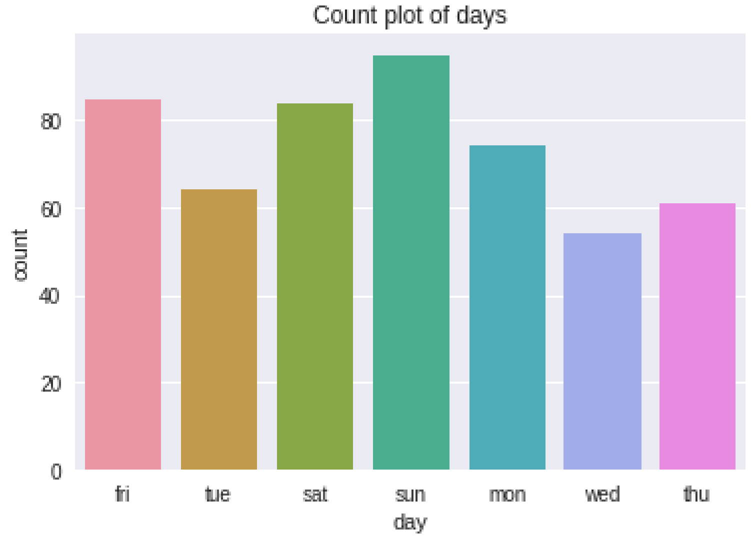
- If the value of the area < 6 then the size_category will be labeled as 0 (Small Fire)
- If the value of the area ≥ 6 then the size_category will be labeled as 1 (Wide Fire)

```
df['size_category'] = np.where(df['area']>6, '1', '0')
df['size_category']= pd.to_numeric(df['size_category']) df.tail(10)
```

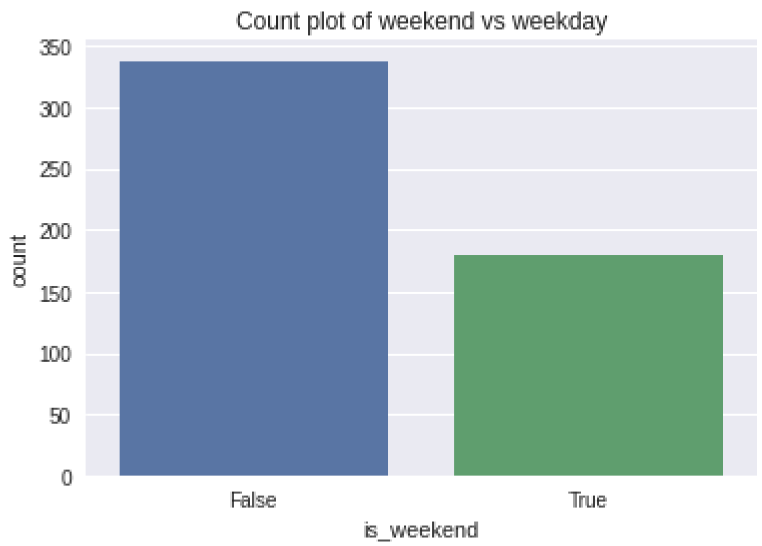
	X	Y	month	day	FFMC	DMC	DC	ISI	temp	RH	wind	rain	area	size_category
507	2	4	aug	fri	91.0	166.9	752.6	7.1	25.9	41	3.6	0.0	0.00	0
508	1	2	aug	fri	91.0	166.9	752.6	7.1	25.9	41	3.6	0.0	0.00	0
509	5	4	aug	fri	91.0	166.9	752.6	7.1	21.1	71	7.6	1.4	2.17	0
510	6	5	aug	fri	91.0	166.9	752.6	7.1	18.2	62	5.4	0.0	0.43	0
511	8	6	aug	sun	81.6	56.7	665.6	1.9	27.8	35	2.7	0.0	0.00	0
512	4	3	aug	sun	81.6	56.7	665.6	1.9	27.8	32	2.7	0.0	6.44	1
513	2	4	aug	sun	81.6	56.7	665.6	1.9	21.9	71	5.8	0.0	54.29	1
514	7	4	aug	sun	81.6	56.7	665.6	1.9	21.2	70	6.7	0.0	11.16	1
515	1	4	aug	sat	94.4	146.0	614.7	11.3	25.6	42	4.0	0.0	0.00	0
516	6	3	nov	tue	79.5	3.0	106.7	1.1	11.8	31	4.5	0.0	0.00	0

2) Data Preprocessing for Days

The distribution for the day seems pretty. We will, instead of encoding 7 variables, separate these into weekend (True) or not weekend (False). With the assumption, if the amount of area burned in a fire is also related to how the fire fighters responded to the flame. During the weekend, the amount of firefighters or the response in general may be different compared during the weekday.

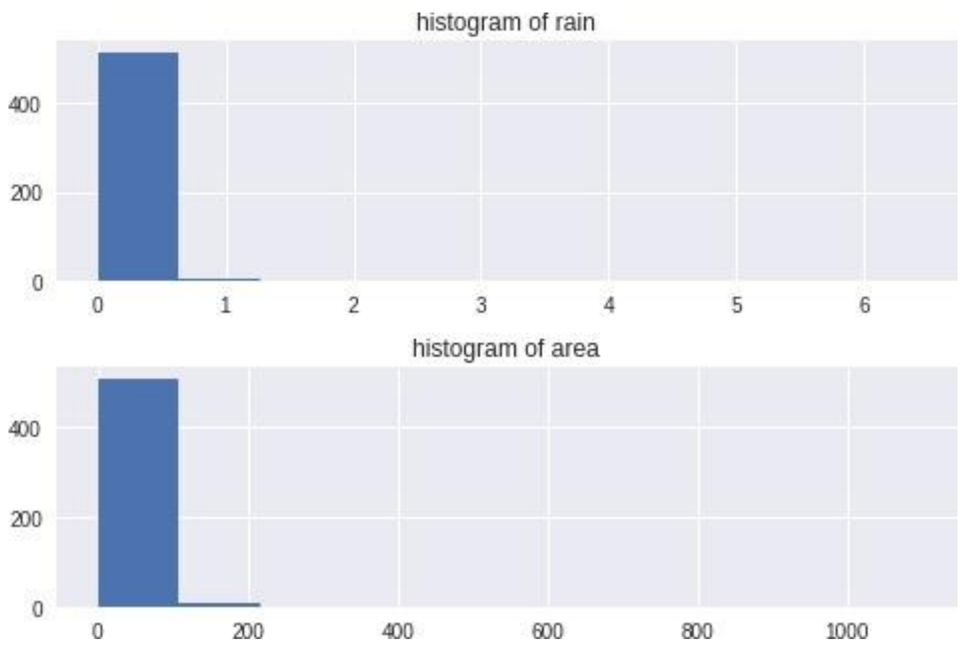


 # converting to is weekend df['day'] = ((df['day'] == 'sun') | (df['day'] == 'sat'))**# renaming column** df = df.rename(columns = {'day' : 'is_weekend'})**# visualizing** sns.countplot(df['is_weekend']) plt.title('Count plot of weekend vs weekday')



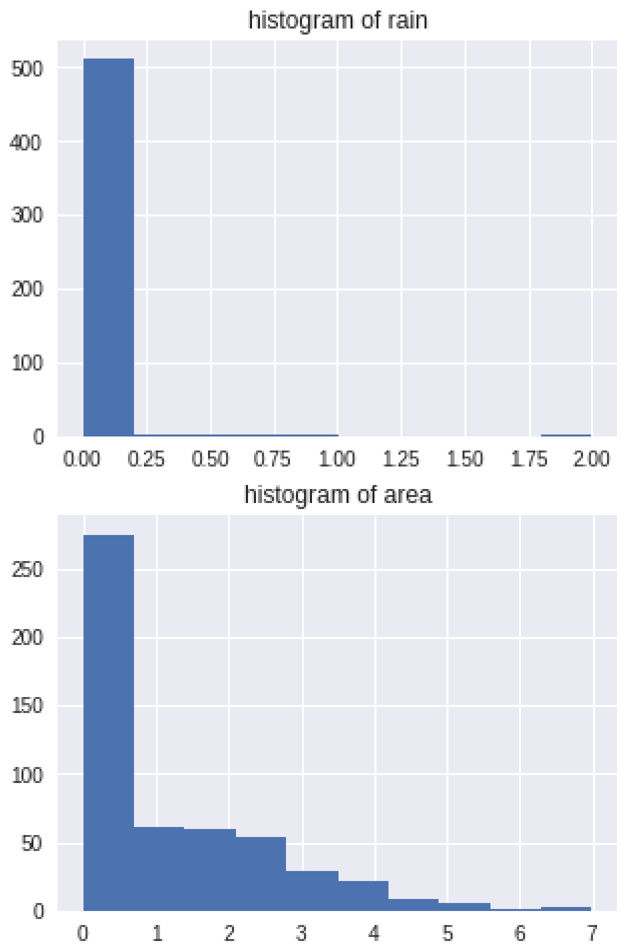
The skew is not too large so we are happy with this conversion.

3) Scaling Area and Rain



The distributions of rain and areaare too skewed and have large outliers so we will scale it to even out the distribution.

```
# natural logarithm scaling (+1 to prevent errors at 0) df.loc[:, ['rain', 'area']]
= df.loc[:, ['rain', 'area']].apply(lambda x: np.log(x + 1), axis
= 1)# visualizing fig, ax =
plt.subplots(2, figsize = (5, 8))
ax[0].hist(df['rain'])
ax[0].title.set_text('histogram of
rain') ax[1].hist(df['area'])
ax[1].title.set_text('histogram of
area')
```



The distribution for rain is not good but the distribution for areais highly improved. Now we scale the entire dataset. Note that we plan on testing a neural network on the dataset so we will scale the area as a preventative measure against an exploding gradient.

First we will split the data into **train and test splits** so that we can scale the train set and then scale the test set based on the train set. Then we will scale everything.

4) Train Test Split

Data is randomly splitted into training data (80 %) and testing data(20%).

```
features = df.drop(['size_category'], axis = 1) labels = df['size_category'].values.reshape(-1, 1)X_train, X_test, y_train, y_test = train_test_split(features,labels, test_size = 0.2, random_state = 42)
```

5) Feature Scaling: StandardScaler

Apply the feature scaling : the standardscaler to the data

```
# fitting scaler
sc_features = StandardScaler()# transforming features X_test = sc_features.fit_transform(X_test)
X_train = sc_features.transform(X_train)# features
X_test = pd.DataFrame(X_test, columns = features.columns) X_train = pd.DataFrame(X_train, columns = features.columns)# labels y_test = pd.DataFrame(y_test, columns = ['size_category']) y_train = pd.DataFrame(y_train, columns = ['size_category'])X_train.head()
```

	X	Y	is_summer	is_weekend	FFMC	DMC	DC	ISI	temp	RH	wind	rain	area
0	-0.293766	-0.927776	0.53287	1.404076	0.342959	-0.060220	0.867158	-0.249015	0.784070	-1.071784	-0.054362	-0.137348	0.219260
1	-0.293766	-0.161993	0.53287	-0.712212	-0.057456	0.370353	0.600021	-0.465731	-0.203920	-0.279375	-1.042369	-0.137348	0.177491
2	-1.130796	0.603791	0.53287	-0.712212	0.312158	0.834731	0.483714	0.664288	0.221938	0.248898	-0.054362	-0.137348	0.180797
3	-0.712281	-0.161993	0.53287	-0.712212	1.544206	1.341810	0.537142	0.664288	2.283090	-1.071784	-1.042369	-0.137348	1.087255
4	0.124750	-0.161993	-1.87663	1.404076	-1.320305	-1.602809	-2.022846	-0.945602	-1.106740	0.645102	-1.042369	-0.137348	0.578923

Step 3: Hyperparameter/ Experiment Results

1) Experiment 1 : Base Model

Here we are going to create our ANN object by using a certain class of Keras named Sequential. Once we initialize our ANN, we are now going to create layers. Here we are going to create a base model network that will have :

- 1 input layer
- 2 hidden layers
- 1 dropout layer
- 1 output layer

Here we have created our first hidden layer by using the Dense class which is part of the layers module. This class accepts 2 inputs:

- **units**: number of neurons that will be present in the respective layer
- **activation**: specify which activation function to be used

We create a sequence of layers to define the neural network and define each layer by initializing weights, defining the activation function and selecting the nodes per hidden layer.

model = Sequential()**# input layer + 1st hidden layer** model.add(Dense(6, input_dim=13, activation='relu'))**# 2nd hidden layer** model.add(Dense(6, activation='relu'))**# output layer** model.add(Dense(6, activation='sigmoid'))

model.add(Dropout(0.2))

model.add(Dense(1, activation = 'relu'))model.summary()

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 6)                 84

 dense_1 (Dense)             (None, 6)                 42

 dense_2 (Dense)             (None, 6)                 42

 dropout (Dropout)           (None, 6)                 0

 dense_3 (Dense)             (None, 1)                 7

=================================================================
Total params: 175
Trainable params: 175
Non-trainable params: 0
_________________________________________________________________
```

The next step, we will compile our ANN with using the hyperparameter below:

Hyperparameter	Value
Epoch	100
Batch Size	10
Activation Function	Relu, sigmoid
Loss Function	binary_crossentropy
Learning Rate	-
Optimizers	Adam

- **Epoch** : how many times neural networks will be trained
- **Batch Size** : how many observations should be there in the batch.
- **Activation Function** : The primary role of the activation function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output.
- **Loss Function** : loss functions are used to determine the error between the output of our algorithms and the given target value.
- **Learning Rate :** learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated.
- **Optimizers** : optimizers are algorithms or methods used to minimize an error function(loss function)or to maximize the efficiency of production.

To check the performance of the methods, we calculate the accuracy measure.

 # Compile Model model.compile(optimizer = 'adam', metrics=['accuracy'], loss ='binary_crossentropy')**# Train Model** history = model.fit(X_train, y_train, validation_data = (X_test, y_test), batch_size = 10, epochs = 100)

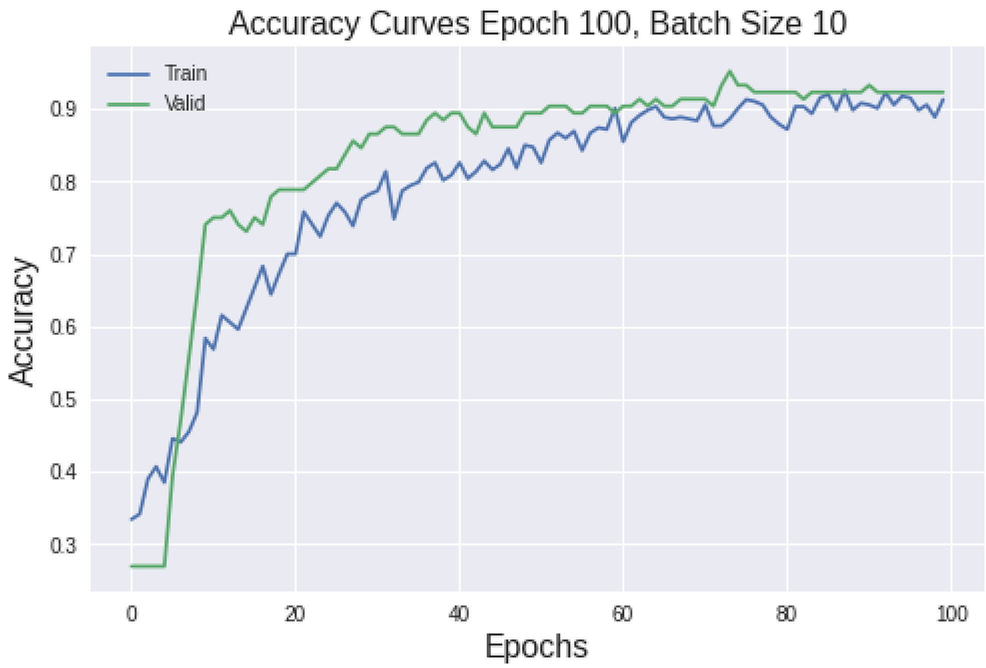
```
Epoch 91/100
42/42 [==============================] - 0s 7ms/step - loss: 0.2995 - accuracy: 0.9056 - val_loss: 0.1458 - val_accuracy: 0.9327
Epoch 92/100
42/42 [==============================] - 0s 6ms/step - loss: 0.2005 - accuracy: 0.9007 - val_loss: 0.1459 - val_accuracy: 0.9231
Epoch 93/100
42/42 [==============================] - 0s 5ms/step - loss: 0.1701 - accuracy: 0.9225 - val_loss: 0.1456 - val_accuracy: 0.9231
Epoch 94/100
42/42 [==============================] - 0s 6ms/step - loss: 0.1970 - accuracy: 0.9056 - val_loss: 0.1451 - val_accuracy: 0.9231
Epoch 95/100
42/42 [==============================] - 0s 5ms/step - loss: 0.2490 - accuracy: 0.9177 - val_loss: 0.1445 - val_accuracy: 0.9231
Epoch 96/100
42/42 [==============================] - 0s 6ms/step - loss: 0.2952 - accuracy: 0.9153 - val_loss: 0.1440 - val_accuracy: 0.9231
Epoch 97/100
42/42 [==============================] - 0s 5ms/step - loss: 0.1885 - accuracy: 0.8983 - val_loss: 0.1434 - val_accuracy: 0.9231
Epoch 98/100
42/42 [==============================] - 0s 6ms/step - loss: 0.2722 - accuracy: 0.9056 - val_loss: 0.1428 - val_accuracy: 0.9231
Epoch 99/100
42/42 [==============================] - 0s 6ms/step - loss: 0.1989 - accuracy: 0.8886 - val_loss: 0.1423 - val_accuracy: 0.9231
Epoch 100/100
42/42 [==============================] - 0s 6ms/step - loss: 0.2983 - accuracy: 0.9128 - val_loss: 0.1419 - val_accuracy: 0.9231
```

_, train_acc = model.evaluate(X_train, y_train, verbose=0) _, valid_acc = model.evaluate(X_test, y_test, verbose=0) print('Train: %.3f, Valid: %.3f' % (train_acc, valid_acc))

```
Train: 0.969, Valid: 0.923
```

Based on the results of experiment 1, which used the the hyperparameter of the base model, the accuracy score of the train data is 96% and the accuracy score of the valid or the test data is 92%.

plt.figure(figsize=[8,5]) plt.plot(history.history['accuracy'], label='Train') plt.plot(history.history['val_accuracy'], label='Valid') plt.legend() plt.xlabel('Epochs', fontsize=16) plt.ylabel('Accuracy', fontsize=16) plt.title('Accuracy Curves Epoch 100, Batch Size 10', fontsize=16) plt.show()



Based on the output of the accuracy graph, the model begins to show the stability at epochs 60 to 100.

2) Experiment 2: Batch Size: 4, 6, 10, 16, 32, 64, 128, 260

For the experiment 2, we will do the ANN's modelling with hyperparameter details as below:

Hyperparameter	Value
Epoch	100
Batch Size	**4, 6, 10, 16, 32, 64, 128, 260**
Activation Function	Relu, sigmoid
Loss Function	binary_crossentropy
Learning Rate	-
Optimizers	Adam

 # Fit a model and plot learning curve def fit_model(X_train, y_train, X_test, y_test, n_batch):**# Define Model** model = Sequential() model.add(Dense(6, input_dim=13, activation='relu')) model.add(Dense(6, activation='relu')) model.add(Dense(6, activation='sigmoid')) model.add(Dropout(0.2))

```
model.add(Dense(1, activation = 'relu'))# Compile Model model.compile(optimizer = 'adam', metrics=['accuracy'], loss = 'binary_crossentropy')# Fit Model history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=100, verbose=0, batch_size=n_batch)# Plot Learning Curves plt.plot(history.history['accuracy'], label='train') plt.plot(history.history['val_accuracy'], label='test') plt.title('batch='+str(n_batch)) plt.legend()# Create learning curves for different batch sizes batch_sizes = [4, 6, 10, 16, 32, 64, 128, 260]plt.figure(figsize=(10,15)) for i in range(len(batch_sizes)):# Determine the Plot Number plot_no = 420 + (i+1) plt.subplot(plot_no)# Fit model and plot learning curves for a batch size fit_model(X_train, y_train, X_test, y_test, batch_sizes[i])# Show learning curves plt.show()
```

Based on the accuracy graph above, the model that is good enough to show stability is the model which **batch = 6.**

3) Experiment 3: Batch Size = 6, Epochs = 20, 50, 100, 120, 150, 200, 300, 400

For the experiment 3, we will do the ANN’s modelling with hyperparameter details as below:

Hyperparameter	Value
Epoch	**20, 50, 100, 120, 150, 200, 300, 400**
Batch Size	6
Activation Function	Relu, sigmoid
Loss Function	binary_crossentropy
Learning Rate	-
Optimizers	Adam

 # fit a model and plot learning curve def fit_model(trainX, trainy, validX, validy, n_epoch):**# define model** model = Sequential() model.add(Dense(6, input_dim=13, activation='relu'))

model.add(Dense(6, activation='relu')) model.add(Dense(6, activation='sigmoid')) model.add(Dropout(0.2)) model.add(Dense(1, activation = 'relu'))**# compile model** model.compile(optimizer ='adam', metrics=['accuracy'], loss =

'binary_crossentropy')**# fit model** history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=n_epoch, verbose=0, batch_size=6)**# plot learning curves** plt.plot(history.history['accuracy'], label='train') plt.plot(history.history['val_accuracy'], label='test') plt.title('epoch='+str(n_epoch)) plt.legend()**# Create learning curves for different batch sizes** epochs = [20, 50, 100, 120, 150, 200, 300, 400]plt.figure(figsize=(10,15)) for i in range(len(batch_sizes)):**# Determine the Plot Number** plot_no = 420 + (i+1) plt.subplot(plot_no)**# Fit model and plot learning curves for a batch size** fit_model(X_train, y_train, X_test, y_test, epochs[i])**# Show learning curves** plt.show()

Based on the accuracy graph above, the model that is good enough to show stability is the model which epoch = 200,300 and 400**.**

4) Experiment 4

Batch Size = 6, Early Stopping (Patience, Model Checkpoint)

For the experiment 4, we will do the ANN's modelling with hyperparameter details as below:

Hyperparameter	Value
Epoch (Input)	**250**
Batch Size	6
Activation Function	Relu, sigmoid
Loss Function	binary_crossentropy
Learning Rate	-
Optimizers	Adam
Patience	**150**
Early Stopping (Epoch)	**225**

def init_model():# **define model** model = Sequential() model.add(Dense(6, input_dim=13, activation='relu')) model.add(Dense(6, activation='relu'))model.add(Dense(6, activation='sigmoid')) model.add(Dropout(0.2)) model.add(Dense(1, activation = 'relu')) model.compile(optimizer ='adam', metrics=['accuracy'], loss = 'binary_crossentropy')return model# **init model** model = init_model()# **simple early stopping** es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=150)# **model checkpoint** mc = ModelCheckpoint('best_model.h5', monitor='val_accuracy', mode='max', verbose=1, save_best_only=True)#

fitting model history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=250, verbose=0, batch_size=6, callbacks=[es, mc])

```
Epoch 00215: val_accuracy did not improve from 0.99038

Epoch 00216: val_accuracy did not improve from 0.99038

Epoch 00217: val_accuracy did not improve from 0.99038

Epoch 00218: val_accuracy did not improve from 0.99038

Epoch 00219: val_accuracy did not improve from 0.99038

Epoch 00220: val_accuracy did not improve from 0.99038

Epoch 00221: val_accuracy did not improve from 0.99038

Epoch 00222: val_accuracy did not improve from 0.99038

Epoch 00223: val_accuracy did not improve from 0.99038

Epoch 00224: val_accuracy did not improve from 0.99038

Epoch 00225: val_accuracy did not improve from 0.99038
Epoch 00225: early stopping
```

 # plot training history

plt.plot(history.history['loss'], label='train')

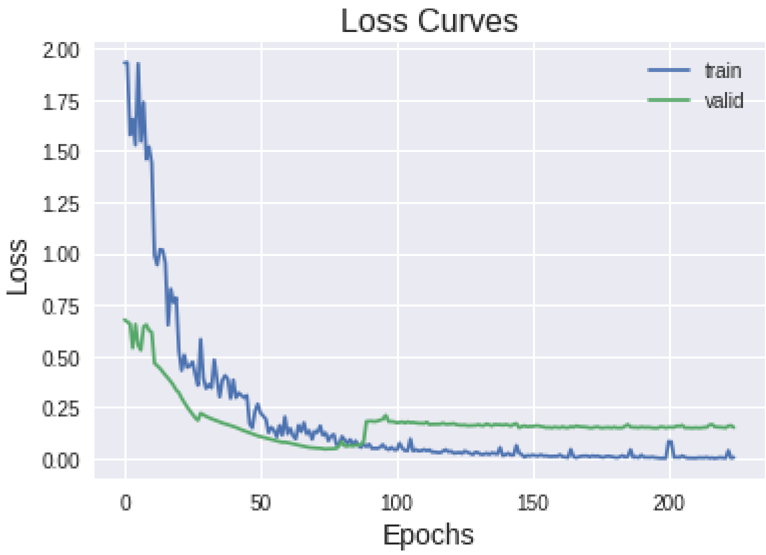
plt.plot(history.history['val_loss'],

label='valid') plt.legend() plt.xlabel('Epochs',

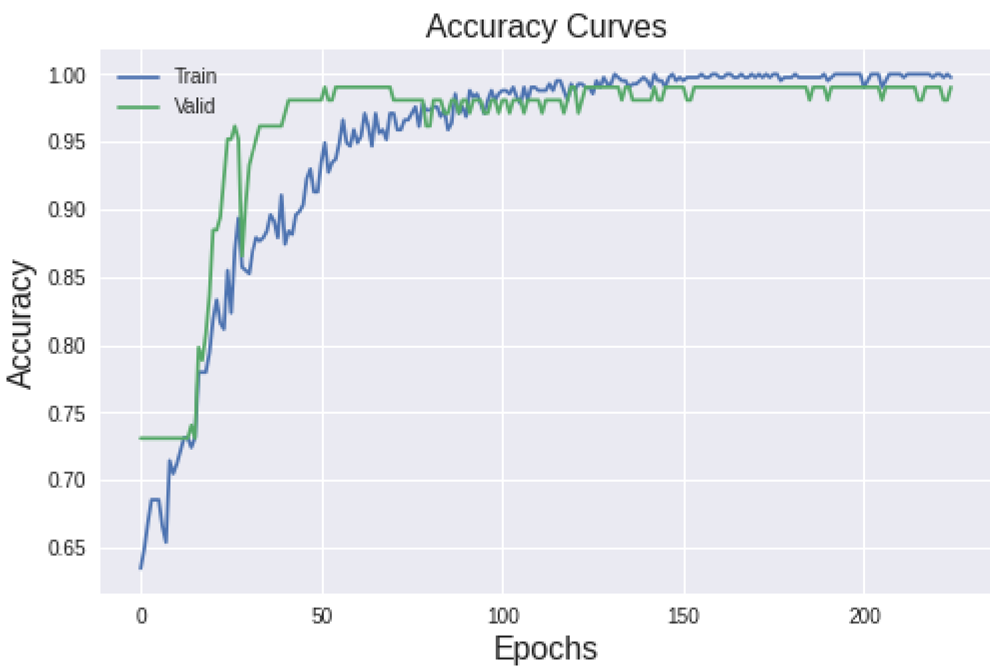
fontsize=14) plt.ylabel('Loss', fontsize=14)

plt.title('Loss Curves', fontsize=16)

plt.show()




```
plt.figure(figsize=[8,5])
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'],
label='Valid') plt.legend() plt.xlabel('Epochs',
fontsize=16) plt.ylabel('Accuracy', fontsize=16)
plt.title('Accuracy Curves', fontsize=16)
plt.show()
```



```
_, train_acc = model.evaluate(X_train, y_train, verbose=0) _, valid_acc = model.evaluate(X_test, y_test, verbose=0) print('Train: %.3f, Valid: %.3f' % (train_acc, valid_acc))
```

```
Train: 0.973, Valid: 0.990
```

Based on the results of experiment 4, which used the early stopping with patience and model checkpoint method, the accuracy score of the train data is 97% and the accuracy score of the valid or the test data is 99%.

Conclusion and Discussion

One of the key success for controlling the forest fire is the early detection of the fire. In this article we conduct the hyperparameter tuning experiment for predicting the burned area of the forest fires specifically in the northeast region of Portugal, based on the spatial, temporal and weather variables where the fire is spotted using deep learning.

To find another best method, we suggest to use the other options of the data preprocessing and try to use machine learning algorithm such as Support Vector Machines (SVM), Decision Tree, Random Forest Classifier, Naive Bayes Classifier etc.