Moses Alexander
10/19/2022
Module 7: Neural Network Final Project Report

Deep Neural Decision Forests for Car Acceptability Classification

1. A ***deep neural decision forest model*** was used to classify car acceptability using the following features: car buying price, car maintenance price, number of car doors, number of people the car can carry, size of luggage boot (in American vernacular, this is trunk size), and estimated car safety.
2. The dataset contains 5184 rows, 6 feature columns (***buying, maint, doors, persons, lug_boot, safety***), and the target column ***acceptability***. The dataset contains only string columns which all are categorical in nature. It is appropriate for a classification problem due to the categorical target variable ***acceptability***. The dataset was derived from a hierarchical decision model proposed in 1988 at a conference in Avignon, France to evaluate the Hierarchy Induction Tool in 1997 in Nashville, United States. The dataset was shuffled and resampled before being applied to the model using tensorflow's **Dataset** functionality.
    1. The source URL of the data is: ***https://archive.ics.uci.edu/ml/datasets/Car+Evaluation***
    2. The variables are as follows:
        - **Target**
            - ***acceptability: unacc, acc, good, vgood***
        - **Features**
            - ***buying: low, med, high, vhigh***
            - ***maint: low, med, high, vhigh***
            - ***doors: 2, 3, 4, 5more***
            - ***persons: 2, 4, more***
            - ***lug_boot: small, med, big***
            - ***safety: low, med, high***
3. Code is attached in **Appendix A (15)**.

4. The final model chosen for this assignment was a ***deep neural decision forest model.*** The decision trees of this forest are weak classifiers which return class distributions at the leaves (the lowest layer of the tree). The neural network component of the model learns the routing probabilities of data traveling down the tree to each leaf. The decision tree component learns the class distributions at each leaf. These components are multiplied together to return a weighted average of predicted class distributions for each tree, whereas a traditional decision tree returns a standard average without accounting for the routing probabilities of the tree. Thus the relative significances of more likely class distributions is boosted and less likely class distributions dampened. This model learns the routing probabilities with a single fully-connected layer with 1 unit per decision tree leaf, activated by **Sigmoid**. **Sigmoid** represents the probability of success or failure on linear predictors of Bernoulli trials, and in this case success and failure correspond to the 2 outcomes of the tree path splitting to right or left. For each decision tree, *70%* (casted to integer) of the available features are randomly selected from the available features, and the decision tree trains using that random subset. The model iteratively travels the tree level by level to calculate and learn the routing probabilities while simultaneously learning the class distributions at each leaf. A sparse layer with **Softmax** is applied to return the class distributions for each leaf. These distributions are multiplied with the learned routing probabilities to return the expected value (weighted average) of class distributions for the entire decision tree as a probability distribution (vector). Each decision tree trains on a random subset of features and are considered weak classifiers. By summing the tree outputs and dividing by the number of trees in the forest, we aggregate the ensemble of weak classifiers into a strong classifier which incorporates information from each tree while lowering variance by averaging. Hence, the final ***deep neural decision forest model*** is a strong classifier which makes use of neural networks' nonlinear learning capacity to learn routing probabilities, tree structures to learn hierarchical conditional rules, ensemble learning to lower variance while improving accuracy, and randomness to ensure a variety of trees (class distributions at the tree leaves) are tested and ranked by the model to aggregate and return the best perceived estimate of the true class distribution of the target variable.

5. The data from the ***car.data*** file was triplicated and compiled in the ***car1.csv*** file. The ***car1.csv*** file was loaded in a tensorflow **Dataset**. The features were embedded into dense vectors and concatenated. The target was encoded as integer indices. The decision tree model was created using the subclass pattern, and then the decision forest model was created using the subclass pattern also. Performance was evaluated over different combinations of hyper-parameters in this experiment by checking training, validation, and test loss and accuracy over 50 epochs. The best set of hyperparameters with the lowest overall loss and highest overall accuracy was selected for the final model. The following 2 articles inspired this experiment:
    1. Explanation of model: ***https://topos-theory.github.io/deep-neural-decision-forests/***
    2. Explanation of code: ***https://keras.io/examples/structured_data/deep_neural_decision_-forests/***
6. Each model took between 4 and 5 minutes to run for 50 epochs and about 45 hyperparameter combinations were tested, so total model experimentation plus debugging and final testing took approximately **270 minutes**. This does not include time spent on data preprocessing and original development.
7. All input variables for this model were strings and hence categorical in nature.
    1. ***buying***: car buying price
    2. ***maint***: car maintenance price
    3. ***doors***: number of car doors
    4. ***persons***: number of people the car can carry
    5. ***lug_boot***: size of luggage boot (in American vernacular, this is trunk size)
    6. ***safety***: estimated car safety
8. Before coding, the data was copied from its source ***car.data*** into a new file ***car1.csv*** which would be easier to process in Python. The ***car1.csv*** file was loaded as a tensorflow **Dataset** in the format of **(features, target class integer index)** and shuffled to give the rows a random (seeded) order. This was done for convenience of later preprocessing steps. The tensorflow **Dataset** was then split into training, validation, and test datasets with the split of **70%/20%/10%**. **Input** layers were created for each of the 6 features, which are for the purpose of

creating symbolic tensors that act as placeholders so the model knows what sort of input types and shapes to accept. For each feature, the classes of the feature were encoded as integer indices with a **StringLookup** layer and then embedded with **Embedding** layers which reduced the feature dimensionality from the number of feature classes to the square root of the number of feature classes (casted to integer). The feature embeddings were then all combined into a single tensor using a **Concatenate** layer. These steps were necessary to convert the string categorical features into a format processable by neural networks. Lastly, the embedded features were batch normalized with a **BatchNormalize** layer before being applied to the model. This recentered and rescaled the input batches before feeding them to the model, speeding up the learning process and standardizing input to the model to better facilitate generalization.

9. The loss metric used was **sparse categorical crossentropy**. Crossentropy is the appropriate metric for classification problems. Sparse categorical crossentropy is used specifically for classification problems in which the target label has more than 2 classes and the classes are encoded as integer indices. Training and validation loss and accuracy were plotted on the screen over epochs after fitting the model on the training and validation datasets, and test loss and accuracy were printed to the screen after evaluating the model on the test dataset.
10. The model was validated while training using a validation dataset containing 20% of the original dataset. The validation loss and accuracy was more or less similar to training loss and accuracy for each epoch, confirming the concurrent model validation process. The functionality of the model was also tested on other categorical datasets (*MNIST, Iris*) without hyperparameter optimization to ensure the model was a reliable classifier.
11. For the final model, the test set loss was approximately ***0.23*** and the test set accuracy was approximately ***0.97***. Because the model is a decision forest, the random splits of features for the trees will cause some variation in performance across different runs. This is reduced by aggregating the trees' outputs and dividing by the number of trees, so the final model variance is low with slight variation. For this model, a shallow forest model with many trees of low depth were used due to only having 6 features available with which to train the model. A deeper model with more depth gave worse performance due to splitting on the data too much

and over-separating classes at the leaves of each tree, thus learning many useless class distributions that diluted the classification power of the model. The crossentropy metric gives a measure of difference between the observed data distribution (training dataset) and the predicted distribution of the model in terms of *nats* (units of information). The model is, on average across many runs, performing quite well because its predicted distributions are only marginally different from the training dataset. The confusion matrix of the model predictions on the test set also shows that ***500*** items were correctly classified and ***12*** items were incorrectly classified out of a total of ***512*** items, attached in **Appendix B (16)**. Training and validation loss and accuracy plots over ***200*** epochs are attached in **Appendix B (16)**. The model took ***540.893*** seconds to train for ***200*** epochs.

12. The various components of the model were determined as follows:
    1. The number of units was determined by the model structure and its associated paper, so there was no leeway to modify or test other structures of units.
    2. A ***deep neural decision forest model*** (feedforward fully-connected single-layer network accepting batch-normalized categorical feature embeddings, connected to a decision tree, connected to a sparse softmax layer for outputting class distributions) was chosen because it seemed interesting to implement.
    3. Supervised training was chosen because the chosen model expected labeled data.
    4. The proportion of training, validation, and test datasets was **70%/20%/10%**. This split was chosen because it is a standard split for many classification problems and seemed reasonable for this task given the amount and types of data.
    5. The number of input units was determined by the number of features and the dimensions of the embedded features, so there was no leeway for modifying the number of input units. The embedding dimensions were the square root of the number of classes of the features, chosen because it was a standard embedding dimension for most categorical embeddings. The number of output units was determined by the number of classes for the target variable, so there was no leeway for modifying the number of output units.
    6. The number and size of hidden layers was determined by the model structure and its associated paper, so there was no leeway to modify or test other structures of hidden layers.

7. The number of epochs chosen was 200. When testing hyperparameters, model performance converged anywhere from 55 to 70 epochs (depending on the particular run and its associated decision trees). Because of the double descent phenomenon, which shows that training models beyond the overfitting (interpolation) threshold can facilitate learning more nuanced relationships in the data, 200 was chosen as the number of epochs for model training.
8. The choice of activation function for learning the decision tree routing probabilities to each leaf was **Sigmoid**, because it outputs a probability between 0 and 1. Furthermore, it "squashes" values at the extremities closer to 0 or 1 respectively, giving more unambiguous predictions of probability. Its domain is negative infinity to infinity so any input value will return an output between 0 and 1. Because the model uses many shallow decision trees, the gradients are unlikely to diminish to 0 and result in the vanishing gradient problem. The choice of activation function for learning the class distributions at the leaves of each decision tree is S**oftmax**, because it outputs a probability distribution which reduces the lower probabilities and emphasizes the highest probability. This gives an unambiguous prediction of the most likely class.
9. The size of the dataset was originally 1728 records, but to meet the requirement of 4000 records, the dataset was triplicated and shuffled. The dataset used to split into training, validation, and test datasets contained 5184 rows and 7 columns (6 features and 1 target).
10. The learning rates tested for this model were 0.001, 0.005, 0.01, 0.05, and 0.1. Of these 5 options, 0.01 performed the best on this model and data. The learning rates of 0.001, 0.005, and 0.1 performed considerably poorly. The learning rate of 0.05 offered similar but slightly worse performance across many comparison tests, so 0.01 was chosen as the final learning rate. An exponential decay schedule was also tested with the optimizer, and the decaying learning rate schedule outperformed the default learning rate schedule (no exponential decay) for all learning rates tested. Thus, the learning rate of 0.01 was chosen with an exponential decay schedule.

11. The optimizer chosen was Adam (adaptive moment estimation), a momentum-based optimizer. Stochastic gradient descent, which does not incorporate momentum, and Adamax were also tested for this model, but Adam outperformed them.

13. Overall, the model performed quite well on the data with low loss and high accuracy on the training, validation, and test datasets across many runs. In the future, I would like to try this on more complex data with many more features. Because this dataset only contained 5184 records and 6 features, a shallow forest model with many trees of low depth provided good performance. On a larger dataset, I would want to test deeper trees and different random feature selection rates, perhaps even randomly setting the rates per tree to evaluate the performance, compared to the constant feature selection rate employed in this model. I would also perhaps experiment with adding a bias term which influences the weighted averages of the decision trees to see if that would improve performance given a particular dataset and classification problem. I might also apply k-fold cross-validation and automated hyperparameter tuning methods which were not needed for this project due to the small amount of data and shallow model structure. The model as-is provided decent classification performance, so it serves as a solid baseline for further experimentation and application on other datasets and classification problems. The process of implementing this complex model was enjoyable, and I hope to be able to make use of this experience for future projects.

14. Comments are attached to the code in **Appendix A (15)**.

15. Appendix A:

(python3)

```
# imports
from math import sqrt
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow.keras import layers
from time import perf_counter, sleep

""" SETUP """
# set path to data file
filename = str(os.path.join(os.getcwd(), "car1.csv"))
# add whitespace for readability of output
print("\n\n\n\n", filename, "\n\n\n\n")
# read data from .csv
df = pd.read_csv(filename)
# seed for reproducibility of randomness
seed = 1
# define columns as specified in .csv header
# target
target_col = ["acceptability"]
# features
feature_cols = [c for c in df.columns.values if c not in target_col]
# combine target and features
```

```
cols = feature_cols + target_col
# number of rows in data
df_size = df.shape[0]
# number of target classes
num_classes = df[target_col[0]].unique().shape[0]

""" DATA INFO """
# add whitespace for readability of output
print("\n\nDATA:\n\n")
# print data
print(df)
# add whitespace for readability of output
print("\n\nFEATURE DISTRIBUTIONS:\n\n")
# print distributions of columns
[print(df[col].value_counts()) for col in feature_cols]
# add whitespace for readability of output
print("\n\nTARGET DISTRIBUTION:\n\n")
[print(df[col].value_counts()) for col in target_col]
# add whitespace for readability of output
print("\n\nNUMBER OF TARGET CLASSES:\n\n")
# print number of target classes
print(num_classes)
# add whitespace for readability of output
print("\n\nSAMPLE OF DATA:\n\n")
# get random sample of data
sample = df.sample(20)
# print sample of data
print(sample)
# add whitespace for readability of output
```

```
print("\n\nSHAPE OF DATA:\n\n")
# print shape of data
print(df.shape)
# add whitespace for readability of output
print("\n\n\n\n\n")

""" MODEL HYPERPARAMETERS """
# number of training iterations
epochs = 200
# batch size per training step
# 64 performed better on this model than 4, 8, 16, 32
# batches larger than 64 are invalid for this model
batch_size = 64
# determines frequency of weight updates per training iteration
# 0.01 performed better on this model than 0.1, 0.05, 0.01, 0.005, 0.001
lr = 0.01
# objective loss function to minimize
# since class labels are encoded as integers and there are more than 2 target classes
loss = "sparse_categorical_crossentropy"
# exponentially decay learning rate over training steps, gives better performance for this model
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=1e-2, decay_steps=10000, decay_rate=0.9)
# optimizer for minimization
# Adam performed better on this model than SGD and Adamax
opt = tf.keras.optimizers.Adam(learning_rate=lr_schedule)
# convert target classes to integer indices, with no out of vocabulary items
target_lookup = \
  layers.StringLookup(vocabulary=df[target_col[0]].unique(), mask_token=None,
num_oov_indices=0)
```

```
""" HELPER FUNCTIONS """
# create tf dataset from .csv
def get_dataset_from_csv(csv_file_path):
    # initialize tf dataset from .csv file with batch size 64 replicated 3 times
    # shuffle dataset with seed, use .csv header, use acceptability as target column
    # add "NA" as default value and replace missing values with "NA"
    # map target lookup onto target column and store it in dataset
    dataset = tf.data.experimental.make_csv_dataset(
        csv_file_path, batch_size=batch_size, column_names=cols, num_epochs=1,
        column_defaults=["NA" for _ in range(len(cols))], label_name=target_col[0],
        header=True, na_value="NA", shuffle=True, shuffle_seed=seed
        ).map(lambda features, target: (features, target_lookup(target)))
    # return
    return dataset.cache()

# tf autograph cannot transform lambda functions currently
@tf.autograph.experimental.do_not_convert
# split dataset into training and validation and test datasets using indices of shuffled dataset
def split(x):
  # return 70/20/10 split for training/validation/test
  return \
    x.enumerate().filter(lambda x, y: x % 10 < 7).map(lambda x,y: y), \
    x.enumerate().filter(lambda x, y: x % 10 > 7).map(lambda x,y: y), \
    x.enumerate().filter(lambda x, y: x % 10 == 7).map(lambda x,y: y)

# define function for plotting loss or accuracy over epochs
def plot_f(model, s):
  # verify "loss" or "accuracy" are the keys passed to the function to plot
```

```
  if s not in ["loss", "accuracy"]:  print("Invalid plot label."); return None
  # loss plot y axis limit
  if s == "loss":  y_lim = 1.5
  # accuracy plot y axis limit
  if s == "accuracy":  y_lim = 1.5
  # plot loss or accuracy over epochs on training data using model history
  plt.plot(model.history[s], label=s)
  # plot loss or accuracy over epochs on validation data using model history
  plt.plot(model.history[f"val_{s}"], label=f"val_{s}")
  # label X axis
  plt.xlabel("EPOCHS")
  # label Y axis
  plt.ylabel(s.upper())
  # set plot X axis bounds
  plt.xlim(0, epochs + (0.2 * epochs))
  # set plot Y axis bounds
  plt.ylim(0, y_lim)
  # create plot legend
  plt.legend(["train", "val"], loc="upper left")
  # add gridlines to plot
  plt.grid(True)
  # save plot as .png with random filename to prevent overwriting
  plt.show()
  # clear plots
  plt.close()


# create feature embeddings
def embed_features(inputs):
  # initialize embedded features list
```

```
  embedded_features = []
  # iterate through features in df
  for col in inputs:
    # get count of classes for each feature
    ct = df[col].unique().shape[0]
    # convert strings to integer indices, with no out of vocabulary items
    lookup = layers.StringLookup(vocabulary=df[col].unique(), num_oov_indices=0)
    # create embedding for each feature and add to embedded features list
    # input dim = number of classes for catching other values
    # output dim = square root of number of classes
    embedding = layers.Embedding(ct, int(sqrt(ct)))
    # lookup and embed feature and add to embedded features list
    embedded_features.append(embedding(lookup(inputs[col])))
  # concatenated embedded features into single embedded feature tensor
  embedded_features = layers.Concatenate()(embedded_features)
  # return
  return embedded_features

""" MODEL DEFINITIONS """
# decision tree model class
class NeuralDecisionTree(tf.keras.Model):
  # initialize decision tree model
  def __init__(self, depth, num_features, select_features, num_classes):
    # inherit from tf.keras.Model
    super().__init__()
    # depth of tree
    self.depth = depth
    # number of leaves is 2 ^ tree depth
    self.num_leaves = 2 ** depth
```

```
  # number of classes for target
  self.num_classes = num_classes
  # select number of features to use in this tree
  num_selected_features = int(num_features * select_features)
  # create identity matrix for features
  one_hot = np.eye(num_features)
  # select random features to use in this tree with one hot vectors of randomly selected features
  self.selected_features_mask = \
    one_hot[np.random.choice(np.arange(num_features), num_selected_features, replace=False)]
  # initialize weights for class distribution of tree
  # represents class distribution of leaves of tree
  self.pi = tf.Variable(
    initial_value=tf.random_normal_initializer()(shape=[self.num_leaves, self.num_classes]),
    dtype="float32", trainable=True)
  # layer outputting routing probabilities (probability of traveling to each leaf)
  # units = number of leaves, activation function = sigmoid
  self.decision = layers.Dense(units=self.num_leaves, activation="sigmoid", name="decision")
# call decision tree model
def call(self, inputs):
  # apply mask and get selected random features
  # transpose selected features mask before multiplication
  # shape = (batch size, number of selected features)
  features = tf.matmul(inputs, self.selected_features_mask, transpose_b=True)
  # compute routing probabilities by applying fully-connected layer to selected random features
  # creates 3d tensor by adding 1 additional dimension of size 1 ("depth")
  # shape = (batch size, number of leaves, 1)
  decisions = tf.expand_dims(self.decision(features), axis=2)
  # concatenate routing probabilities and their complements in 3d tensor
  # complement: probability of going to any other leaf in tree
```

```
# shape = (batch size, number of leaves, 2)
decisions = layers.concatenate([decisions, 1-decisions], axis=2)
# initialize probabilities of input data reaching leaves (all 1s)
mu = tf.ones([batch_size, 1, 1])
# initialize starting and ending indices
stt, end = 1, 2
# breadth-first tree traversal
# iterate through tree one level at a time
for i in range(self.depth):
  if mu.shape[0] == batch_size:
    # reshape mu into 3d tensor
    # shape: (batch size, 2 ^ tree level, 1)
    mu = tf.reshape(mu, [batch_size, -1, 1])
    # replicate 3d tensor and add 1 to "depth"
    # # shape: (batch size, 2 ^ tree level, 2)
    mu = tf.tile(mu, (1, 1, 2))
    # get routing probabilities for all nodes in current tree level
    # shape: (batch size, 2 ^ tree level, 2)
    tree_level_decisions = decisions[:, stt:end, :]
    # multiply routing probabilities of input data batch by
    # probabilities of routing to each node in current tree level
    # shape: (batch size, 2 ^ tree level, 2)
    mu *= tree_level_decisions
    # set start index to node of first index of next tree level
    # since slicing is not inclusive at the end: [start: end)
    stt = end
    # set ending index to node of first index of next tree level's next tree level
    # since slicing is not inclusive at the end: [start: end)
    end = stt + 2 ** (i + 1)
```

```
    # reshape routing probabilities into 2d tensor
    # ie multiplying the replicated "depth" by the number of nodes in the 2nd-to-last level
    # 2 * 2 ^ final tree level - 1 = number of nodes in final tree level = number of leaves
    # shape: (batch size, number of leaves)
    mu = tf.reshape(mu, [batch_size, self.num_leaves])
    # apply softmax to get class distribution for leaf
    probabilities = tf.keras.activations.softmax(self.pi)
    # multiply routing probabilities for each leaf by class distribution for each leaf
    outputs = tf.matmul(mu, probabilities)
    # return
    return outputs

# decision forest model class
class NeuralDecisionForest(tf.keras.Model):
  # initialize decision forest model
  def __init__(self, num_trees, depth, num_features, select_features, num_classes):
    # inherit from tf.keras.Model
    super().__init__()
    # number of classes
    self.num_classes = num_classes
    # create list of neural decision tree objects
    # each tree will split on a set of randomly selected features
    self.trees = [NeuralDecisionTree(depth, num_features, select_features, num_classes)
        for _ in range(num_trees)]
  # call decision forest model
  def call(self, inputs):
    # initialize outputs with a zero matrix
    # shape: (batch size, number of classes)
    tree_outputs = tf.zeros([batch_size, self.num_classes])
```

```
    # aggregate decision tree outputs for all trees in forest
    for tree in self.trees:  tree_outputs += tree(inputs)
    # get average of aggregated tree outputs
    tree_outputs /= len(self.trees)
    # return
    return tree_outputs

""" DECISION FOREST HYPER PARAMETERS """
# number of trees in forest
# 20 performed better on this model than 5, 10, 40
num_trees = 20
# depth of trees
# 10 performed better on this model than 5, and 20 requires too much memory
depth = 10
# proportion of features used in each tree
# 0.7 performed better on this model than 0.3 and 0.5
select_features = 0.7

# define model in functional form
# define input layers for each feature
inputs = {col: layers.Input(name=col, shape=(), dtype=tf.string) for col in feature_cols}
# create feature embeddings
embedded_features = embed_features(inputs)
# batch normalize embedded input data features
features = layers.BatchNormalization()(embedded_features)
# create neural decision forest model with previously specified decision forest hyperparameters
forest = NeuralDecisionForest(num_trees, depth, features.shape[1], select_features, num_class-
es)
# get forest model outputs
```

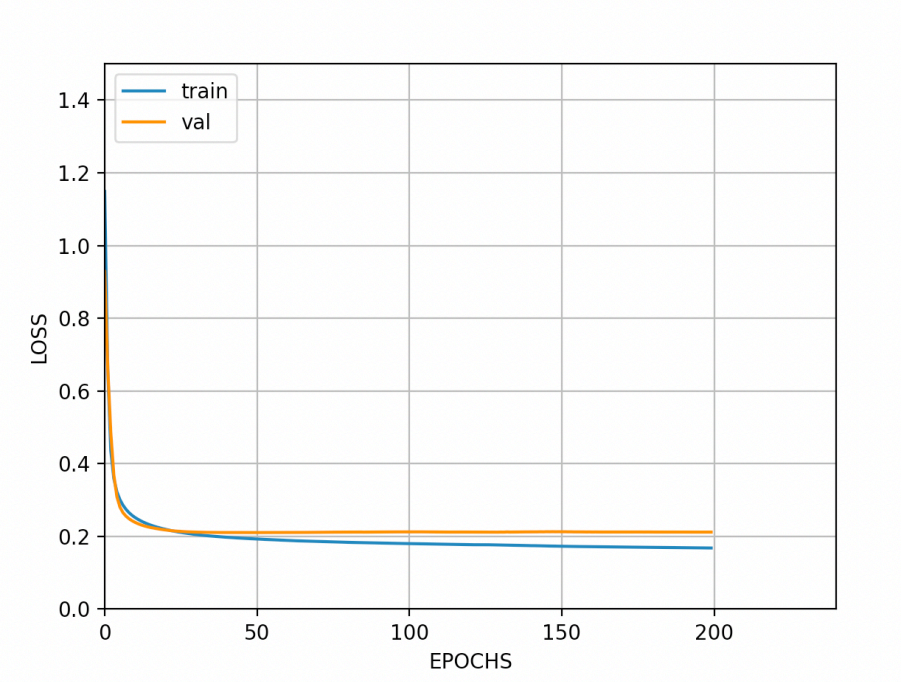
```
outputs = forest(features)
# define final model
model = tf.keras.Model(inputs=inputs, outputs=outputs)
# show model summary
model.summary()

""" COMPILATION, TRAINING, EVALUATION """
# create tf dataset from .csv
dataset = get_dataset_from_csv(filename)
# training: 70%, validation: 20%, test: 10%
training, validation, test = split(dataset)
# compile model with previously specified hyperparameters
model.compile(optimizer=opt, loss=loss, metrics=["accuracy"])
# start timing
start = perf_counter()
# fit model on training dataset for 100 epochs
model.fit(training, epochs=epochs, validation_data=validation)
# end timing
end = perf_counter()
# plot training loss over epochs
plot_f(model.history, "loss")
# plot training accuracy over epochs
plot_f(model.history, "accuracy")
# print model training time in seconds
print(f"\nModel took {round(end-start, 3)} seconds to train for {epochs} epochs.\n")
# evaluate model on test dataset
results = model.evaluate(test)
# add whitespace for readability of output
print("\n\n\n\n")
```

```
# print test loss and accuracy
print(f"\nTEST SET LOSS: {results[0]}\nTEST SET ACCURACY: {results[1]}\n")
# get model predictions on test set
predictions = model.predict(test)
# get test set labels from tensorflow dataset
labels = [i[1] for i in tfds.as_numpy(test)]
# concatenate labels from 2d array into 1d array
labels = np.concatenate(labels)
# get index of class with highest predicted probability
predictions = [np.argmax(p) for p in predictions]
# calculate confusion matrix
confusion_matrix = tf.math.confusion_matrix(labels, predictions, num_classes=num_classes)
# add whitespace for readability of output
print("\n\nCONFUSION MATRIX:\n\n")
# print confusion matrix
print(confusion_matrix)
# add whitespace for readability of output
print("\n\nTOTAL NUMBER OF ITEMS CLASSIFIED:\n\n")
# print sum of confusion matrix diagonal
print(len(labels))
# add whitespace for readability of output
print("\n\nNUMBER CORRECTLY CLASSIFIED:\n\n")
# print sum of confusion matrix diagonal
print(int(tf.linalg.trace(confusion_matrix)))
# add whitespace for readability of output
print("\n\nNUMBER INCORRECTLY CLASSIFIED:\n\n")
# print length of test label list minus sum of confusion matrix diagonal
print(len(labels) - int(tf.linalg.trace(confusion_matrix)))
```

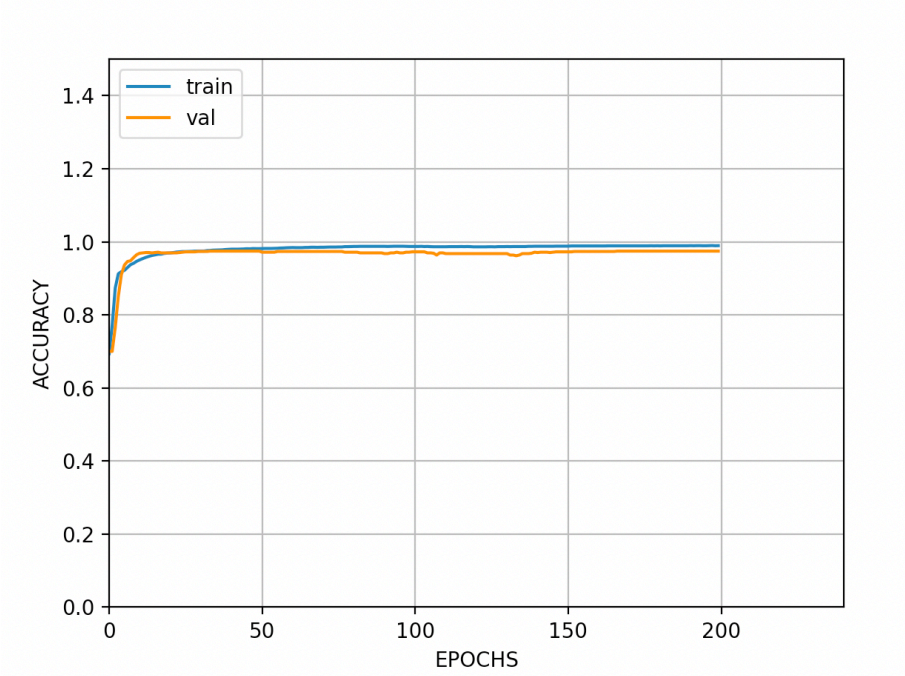
```
# add whitespace for readability of output
print("\n\n\n\n")
""" END """
# end
print("\n\ndone.\n\n")
```

16. Appendix B:

1. Loss plot attached here:



2. Accuracy plot attached here:



3. Distribution of target classes attached here:

TARGET DISTRIBUTION:

unacc 3630

acc 1152

good 207

vgood 195

NUMBER OF TARGET CLASSES: 4

4. Confusion matrix attached here:

CONFUSION MATRIX:

([329, 7, 0, 0,]
[2, 132, 0, 3,]
[0, 0, 27, 0,]
[0, 0, 0, 12,])

TOTAL NUMBER OF ITEMS CLASSIFIED: 512
NUMBER CORRECTLY CLASSIFIED: 500
NUMBER INCORRECTLY CLASSIFIED: 12