Contents

1 Introduction **2**
1.1 My Github for the code . . . 2

2 Wisconsin Breast Cancer (Diagnostic) Data Set **2**
2.1 Plotting the features of the Wisconsin Breast Cancer Data Set . . . 3
2.1.1 Distribution of the features . . . 3
2.1.2 Features pair plotted by outcome class . . . 4
2.1.3 Correlation matrix . . . 5
2.1.4 Average values of the features . . . 6

3 The hyper tuning **6**
3.1 Keras tuner . . . 7
3.2 Parameters to tune . . . 7
3.3 Results for all trials . . . 9
3.4 Graphical result for all trials . . . 10
3.5 Results for best trail . . . 10
3.5.1 Further evaluations on the best model . . . 12

4 Summary and Conclusions **13**

Appendix A Keras documentation links **13**

Keras Hyper Tuner

Aidin Ghassemloi

November 6, 2020

1 Introduction

As I was doing projects and taking my certificate in Tensorflow I was always left with tuning my hyper parameters. Its a fun process and you always get satisfied when you reach that 1% extra accuracy. But it can also be tedious because the best model is hidden in your hyper parameters. The questions i asked myself was:

What is the optimal epoch? How many layers are optimal? What activation function is best suited? How many neurons are optimal? What dropout rate should I choose? What is the best learning rate for my optimizer? And how do I archive this without underfitting/overfitting my model?

So as any curious developer would do is to research what could be done. And what I found was great in any aspect. So to learn this the best way was to develop a project and implement a tuner with Keras. So what did I do? Well, first step was to find a solid data-set that is interesting, not to complicated to understand, used in previous research and proved to be a good data-set that suits machine learning solutions. The data set chosen was the Wisconsin Breast Cancer (Diagnostic) Data Set. The data set contain 32 features with measurements of cancer cells who are malignant or benign. And my task was classification. I also choose this data-set because the aim of this project is to implement a hyper tuner and spending to much time on feature engineering could be left for another project.

Also I will not go in to a depth on a few topics as the purpose of this paper is to demonstrate my journey for this project. I will include a section on further readings if you as a reader want to read more about the topics noted here.

1.1 My Github for the code

Before downloading the code I recommend to read the paper. The code is written in a object oriented fashion. Everything that is needed for this project is in one class for simplicity. Please take a look at the code! **Link:** `https://github.com/aidingh/Keras-Tuner-cancer-cells`

2 Wisconsin Breast Cancer (Diagnostic) Data Set

Wisconsin Breast Cancer (Diagnostic) Data Set contain 32 features with measurements of cancer cells who are malignant or benign. This data set is used in previous research to classify if a cancer cell is malignant or benign. In this section we will go through what the features are and demonstrate some plots of the features to get a understanding of the data. Feature engineering is a step I cant do before going to the next step in the process. Below is a list explaining the features. I found the data set on Kaggle and the link for it is here **Link:** `https://www.kaggle.com/uciml/breast-cancer-wisconsin-data`

Class distribution: 357 benign, 212 malignant

1. radius (mean of distances from center to points on the perimeter)
2. texture (standard deviation of gray-scale values)
3. perimeter

4. area
5. smoothness (local variation in radius lengths)
6. compactness
7. concavity (severity of concave portions of the contour)
8. concave points (number of concave portions of the contour)
9. symmetry
10. fractal dimension ("coastline approximation" - 1)

The mean, standard error and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features.

2.1 Plotting the features of the Wisconsin Breast Cancer Data Set

So now we have the data set and understand it. Now we have to plot the data and do some feature engineering to understand the data further more. In these sub section I will demonstrate some plots of the data. I will also add a code snippet of how the plots are made.

2.1.1 Distribution of the features

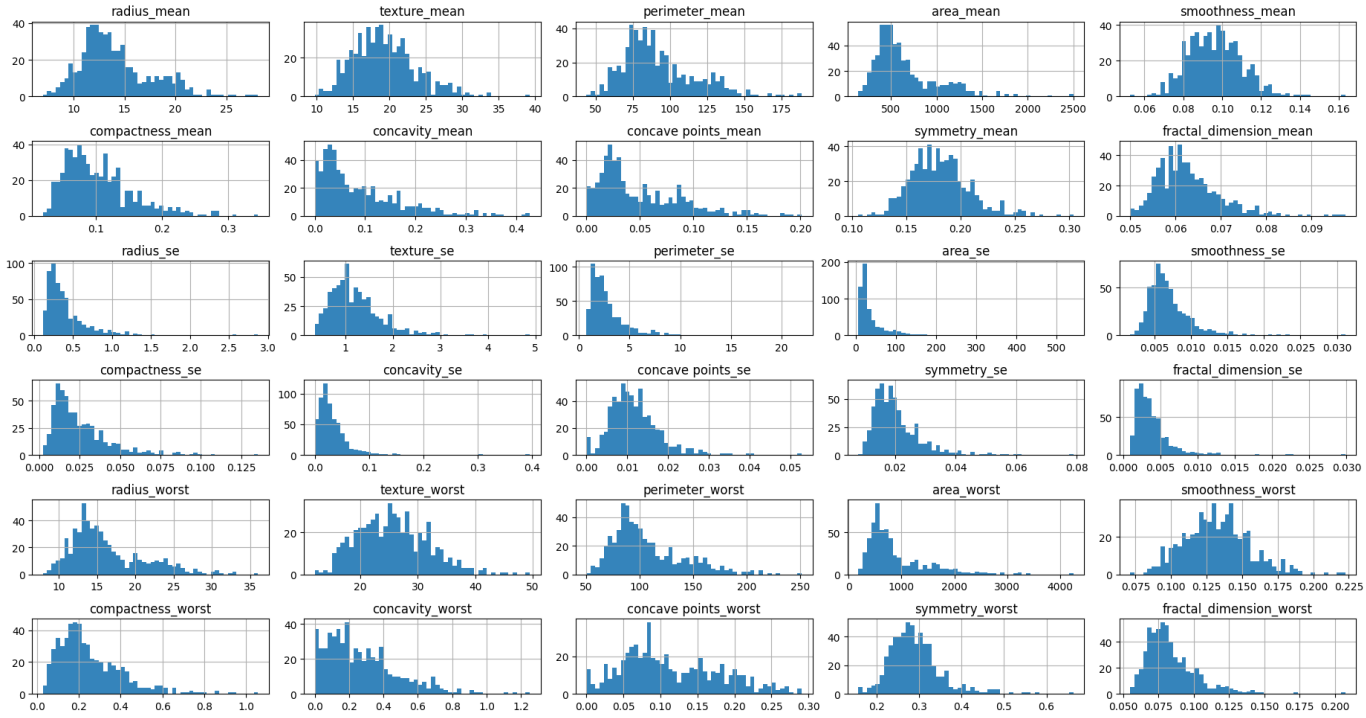


Figure 1: Distribution of the features

Figure 1: Shows how the values are distributed in a feature and to understand were if a feature comes from a specified distribution. Also its a good analysis if there is a choice of scaling the features.

```
    def plot_bins(self):
        data_frame.hist(bins=50, alpha=0.9, figsize=(20, 10))
        plt.tight_layout()
        plt.show()
```

2.1.2 Features pair plotted by outcome class

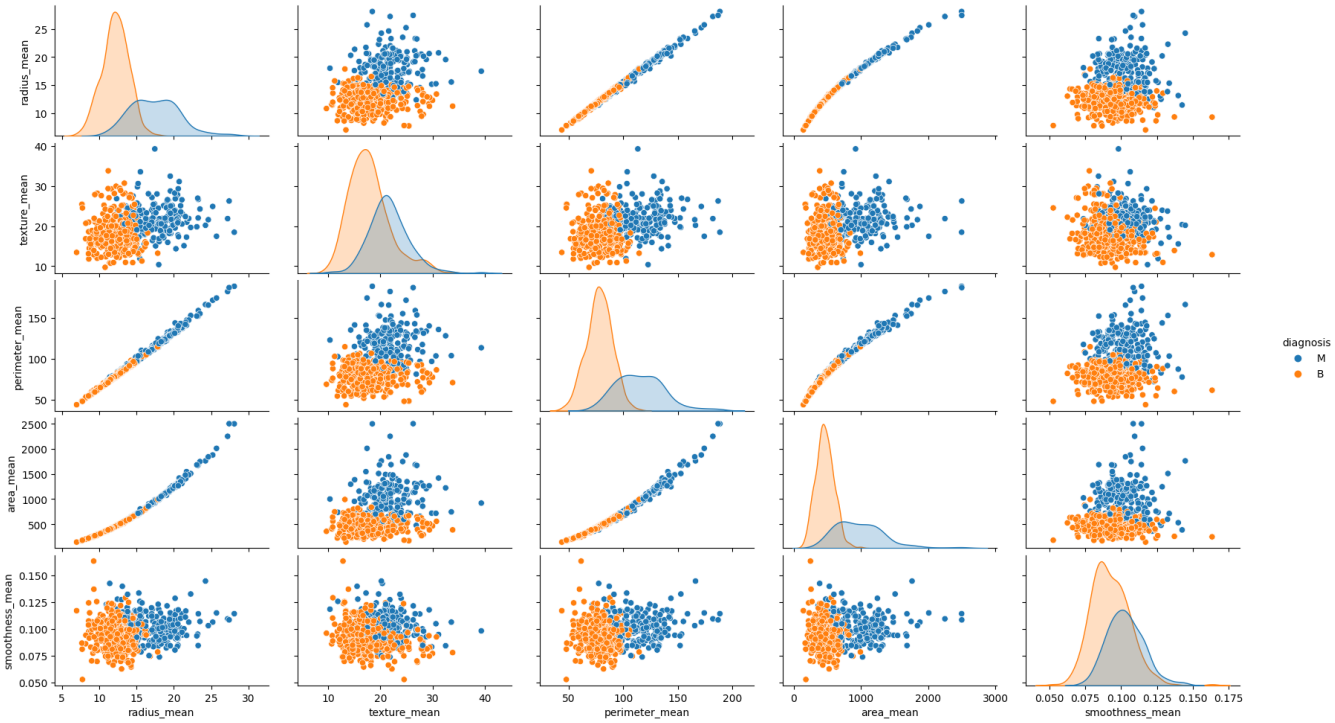


Figure 2: Features pair plotted by outcome class

Figure 2: This plot is the one that is the ice-breaker for me. I choose all the mean features and plotted them by pair to see the difference in the classes. It clearly shows that there is difference in x and y values for the classes we are aiming to classify.

```
1     def plot_by_mean(self):
2         sn.pairplot(data_frame, hue='diagnosis', vars=['radius_mean', 'texture_mean', '
      perimeter_mean', 'area_mean', 'smoothness_mean'])
3         plt.show()
```

2.1.3 Correlation matrix

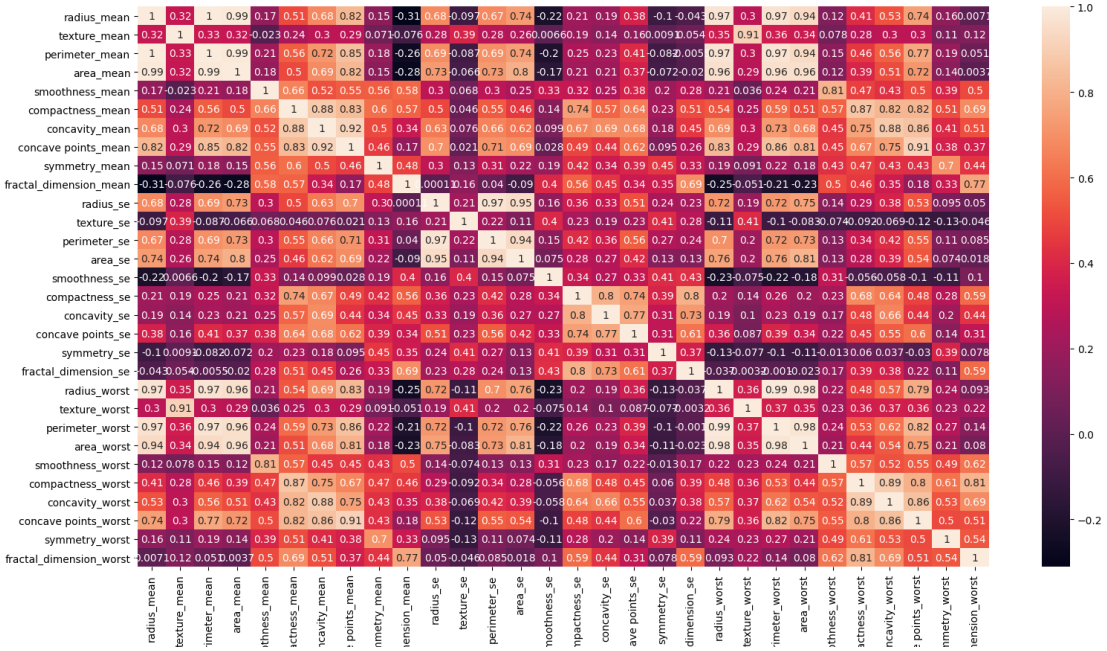


Figure 3: Correlation matrix of all the features correlated to each other

Figure 3: Old habits die hard. I have a habit of always make a correlation matrix to check how the features are correlated to dig further. In some cases low correlated values may be dropped, specially when analyzing what feature is important. The elements that black in the matrix can be analyzed further. But in our case all our features have importance.

```
    def plot_corr_matrix(self):
        corr_val = data_frame.corr()
        print(corr_val)
        plt.figure(figsize=(20, 10))
        sn.heatmap(corr_val, annot=True)
        sn.set_style("darkgrid")
        sn.set_context("paper")
        plt.show()
```

2.1.4 Average values of the features

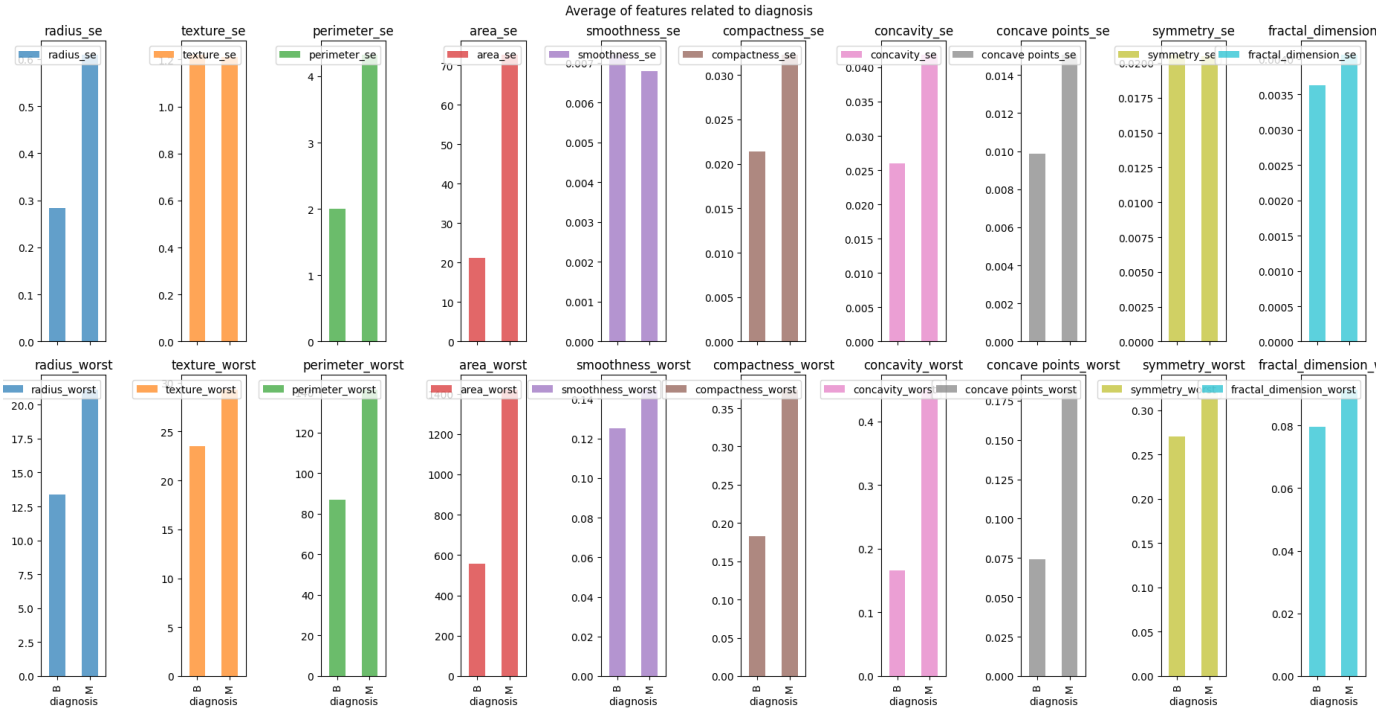


Figure 4: Average values of the features plotted against the labels

Figure 4: The features who are not in the mean category are plotted by their average against our outcome labels.

```
    def plot_group_by_mean(self):
        data_frame.drop(data_frame.iloc[:, 1:11], axis=1, inplace=True)
        data_frame.groupby('diagnosis').mean().plot(kind="bar", subplots=True, sharex=True,
    sharey=False, figsize=(20, 10), layout=(2, 10), alpha=0.7, title="Average of features
    related to diagnosis")
        plt.tight_layout()
        plt.show()
```

3 The hyper tuning

As I implement a model the first thing is to make the model work. The input shape for the model is in order, the data is scaled properly and a split of the data-set that contains a train and test data. I split 80% for training and 20% for test. What I do is to make the model produce any decent result or any at all. It does not matter if the model is underfitted or overfitted. My main focus is to produce any result.

After that I plot the validation accuracy and train accuracy and the same goes for the loss. So lets say my model produced a 88.03% validation accuracy hypothetically. Its not bad but I know there is potential for more. Also the model is overfitting.

So now we have a few options. We can analyze our loss and accuracy plotted against the epochs and try to improve by tweaking the hyper parameters available to us. But if you ask me there is a better way. Let me introduce to you the Keras tuner.

Lets just put somethings in perspective. If you have 10 potential values and 5 parameters that's 100000 combinations we need to try(10^5).

3.1 Keras tuner

There are three ways to tune your Neural Network with Keras. Random search, Hyperband and Bayesian optimization. In this paper I will explain the Random search optimizer. So lets go through what the Random search does. The Random search optimizer randomly sample hyper parameter combinations depending on a objective. In my case it is the validation accuracy and produce the best model based on that. So the metric to produce the best model on is the validation accuracy but can be different depending on your task. So why the validation accuracy as metric? Well, the best model is based on the validation metric.
There are three main parameters to consider when creating the Random search object.

1. objective
2. max_trials
3. execution_per_trail

1) The objective parameter is the metric we tune on.
2) The max_trials parameter is the total number of model configurations to test at most.
3) The execution_per_trail parameter is the number of models that should be built for each trial. This is to create robustness.

```
    def init_random_search_keras_tuner(self, max_trials_int, executions_per_trial_int,
    verbose):

        tuner = RandomSearch(
            self.build_model,
            objective='val_accuracy',
            max_trials=max_trials_int,
            executions_per_trial=executions_per_trial_int,
            directory=self.log_model_dir,
            project_name='Model_Logs_Keras_Tuner'
        )
        if verbose:
            tuner_summary = tuner.search_space_summary()
            print(tuner_summary)
            return tuner
        else:
            return tuner
```

3.2 Parameters to tune

So in my case I wanted to tune as many hyper parameters I could. Specially to stop the model when convergence have happened in the loss metric. That means to stop the model based on the epochs. So if the epoch number is 100 and convergence is on 50 epochs it should stop there. I will come back to how I do this. Now lets go through what I tuned for this project and how I did it. Of course you can tune even more, but I tuned the parameters I thought was most important for this project.

- Epochs
- Number of neurons
- Number of layers
- Activation functions
- Learning rate for optimizer
- Number of neuron dropout

When tuning these hyper parameters in code I use HyperParameters class from Keras. To further demonstrate this I will provide code to make it more clear. After creating the random search object "tuner" I use the tuner.search() method to run the hyper parameter optimization. The tuner.search() is equivalent to a model.fit(). Here is were you define you callbacks, epochs, batch size, validation data and X/Y training data.

```
    def run_keras_tuner(self, tuner,  X_train, X_test, y_train, y_test, epoch_int,
    batch_size_int, tensorboard_callbacks):

        if tensorboard_callbacks:
            tuner.search(X_train, y_train, validation_data=(X_test, y_test), epochs=
    epoch_int, batch_size=batch_size_int, callbacks=[tf.keras.callbacks.TensorBoard(log_dir=
    self.log_tensorboard_model_dir), tf.keras.callbacks.EarlyStopping('val_loss', patience
    =3)])
            command = self.os_tensorboard_log_dirr + self.log_tensorboard_model_dir
            os.system("gnome-terminal -e 'bash -c \"" + command + ";bash\"'")
        else:
            tuner.search(X_train, y_train, validation_data=(X_test, y_test), epochs=
    epoch_int, batch_size=batch_size_int, callbacks=[tf.keras.callbacks.EarlyStopping('
    val_loss', patience=5)])

        best_hps = tuner.get_best_hyperparameters()[0]
        models = tuner.get_best_models(num_models=1)
        best_model = models[0]

        return best_model, best_hps, tuner
```

There is a lot going on in this code, so let me explain a bit before we go further. I have define two scenarios in this section of code. The first If statement is defined with two callbacks. The first callback is the EarlyStopping object to stop at the best epoch number as possible. So at the best scenario stop at the generalization gap. The next callback is the Tensorboard callback, the purpose for this is to be able to visualize all the trials and inspect the loss and accuracy of the models. And with Tensorboard we can isolate the best model by trail id and inspect its results. The next code snippet demonstrates the tuner model and how its implemented.

```
    def build_model(self, hp):

        model = Sequential()
        model.add(Dense(units=hp.Int('units', min_value=32, max_value=512, step=32),
    activation=hp.Choice('activations', ['relu', 'softmax']), input_shape=(self.
    static_shape_len,)))
        model.add(keras.layers.BatchNormalization())
        model.add(keras.layers.Dropout(rate=hp.Choice('dropout_rate', [0.0, 0.1, 0.2, 0.3,
    0.4, 0.5])))

        for i in range(hp.Int("Layers", min_value=0, max_value=3)):
            model.add(tf.keras.layers.Dense(units=hp.Int('units', min_value=32, max_value
    =512, step=32), activation=hp.Choice('activations', ['relu', 'softmax'])))
            model.add(keras.layers.BatchNormalization())
            model.add(keras.layers.Dropout(rate=hp.Choice('dropout_rate', [0.0, 0.1, 0.2,
    0.3, 0.4, 0.5])))

        model.add(Dense(1, activation='sigmoid'))

        model.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.Adam(hp.
    Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])), metrics=['accuracy'])

        return model
```

Here is were I use the HyperParameters class from Keras as hp. So lets go through what this class can provide. To understand more I recommend to read the documentation about this class which I will provide as a link. So for number of neurons to optimize I use the hp.Int() method that lets me define a min neuron choice and max. And I let the function increase the neuron number of the layer by 32. The hp.Choice() makes the optimizer to make trials with the options defined. For example for the activation functions, were the optimizer tries the relu and softmax options defined. The for-loop that is implemented is to optimize

the number of layers.

Summary
So lets summarize a little to make it more clear. If we set max_trials to 5 and execution_per_trail to 1, the optimizer will random search 5 trials on the validation metric. It will try different hyper parameters options until it reaches a result based on the parameters it has as options to tune. In the next section I will go further in to this topic to explain more.

3.3 Results for all trials

So after we have defined the tuner and the code its time evaluate the results that optimizer produced. This is why I used the Tensorboard callback to be able to visualize all the trial and the best trial. Are the model that are optimized any good? Is the best trail underfitted/overfitted? So for this paper I set the max_trials to 5 and execution_per_trail to 1. Now lets check the result summary.

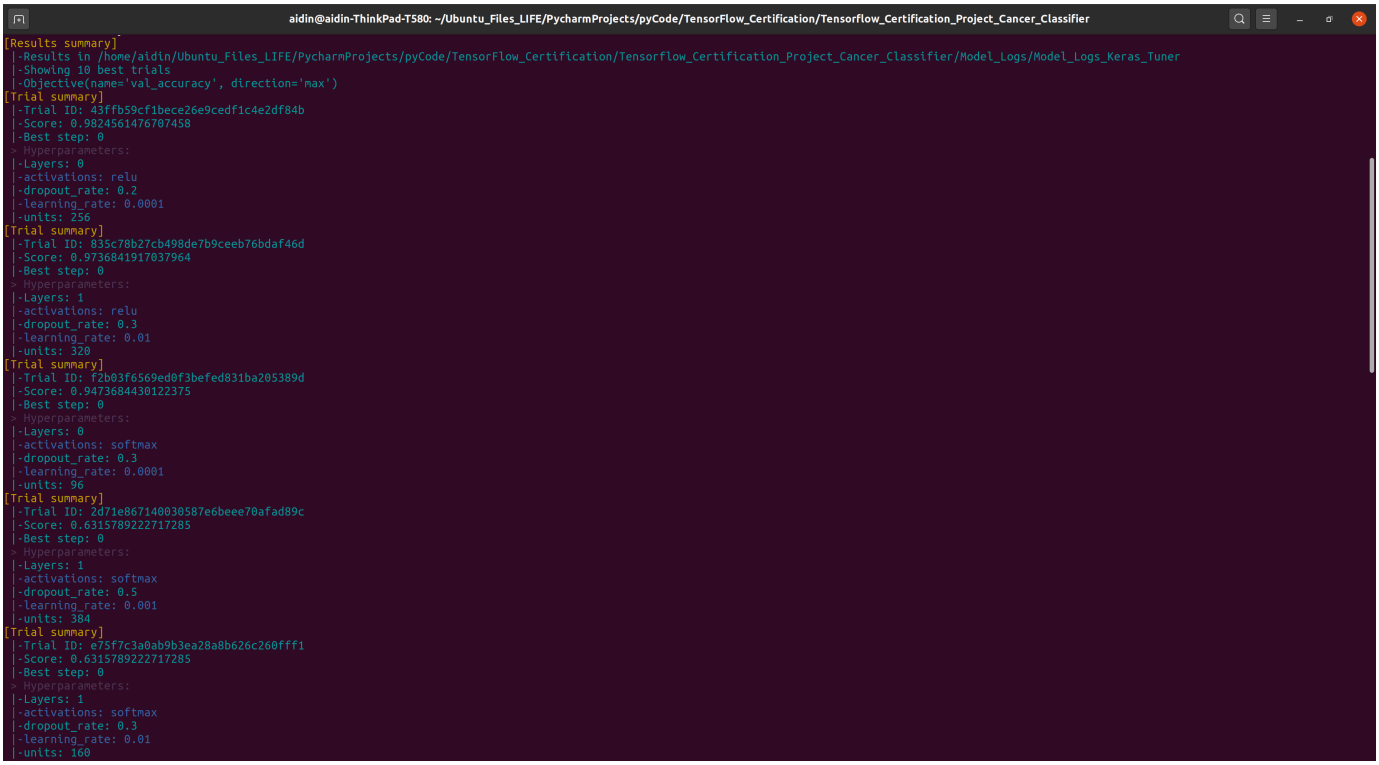


Figure 5: Result summary of the Random search optimizer

Figure 5: Demonstrates the 10 best trials the optimizer was able to produce with the hyper parameter options it were given.
The summary provides us with 10 of the best trials. There are some parameters returned by the optimizer. The first trial summary contains the best model. This is the model we will isolate and make a section for to evaluate by it self. But before I do that, I will explain some of the parameters return by the summary. Note that the optimizer stores the results in a given directory with the train and validation so we can plot loss and accuracy on the train and validation data.

Trail Summary
1) Trail Id to check that specific trial.
2) Score is the validation accuracy score. So if Score is = 0.97 the validation accuracy is 97%

Hyper parameters
Then we have the hyper parameter defined for that specific trail and score.
1) Layers: If layers is = 0 that actually means that it needs 1 layer because the for-loo starts from index 0.
2) Activation's: What type of activation function was best suited for the neurons.
3) Dropout rate: What dropout rate what best suited for that specific model to avoid some overfitting.
4) Learning rate: Best learning rate suited for the optimizer.
5) Units: How many neurons need to be defined for the layers to create the model.

3.4 Graphical result for all trials

All trails are plotted for loss and accuracy with a smoothing factor of 0.6.

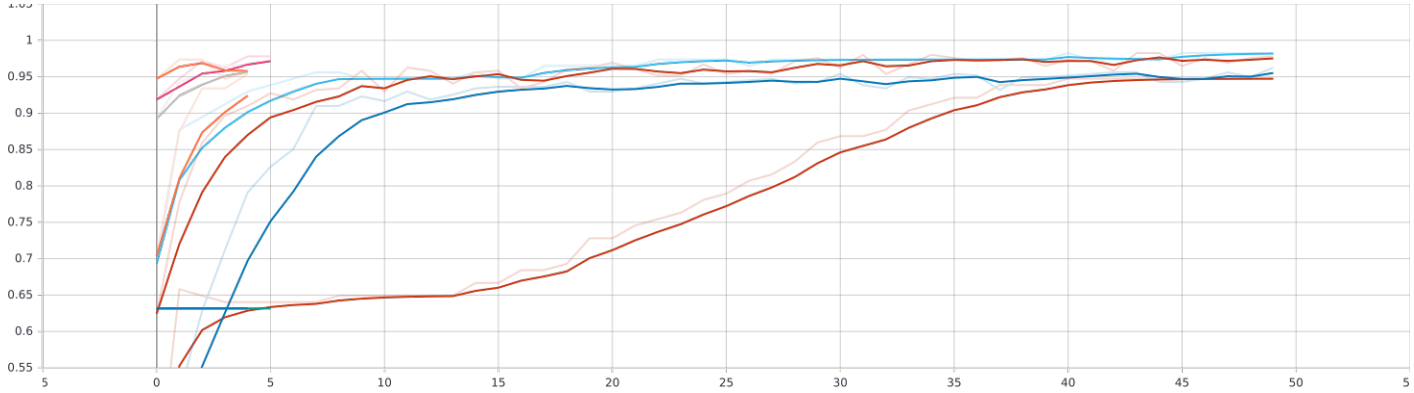


Figure 6: Train and validation accuracy plotted against epochs

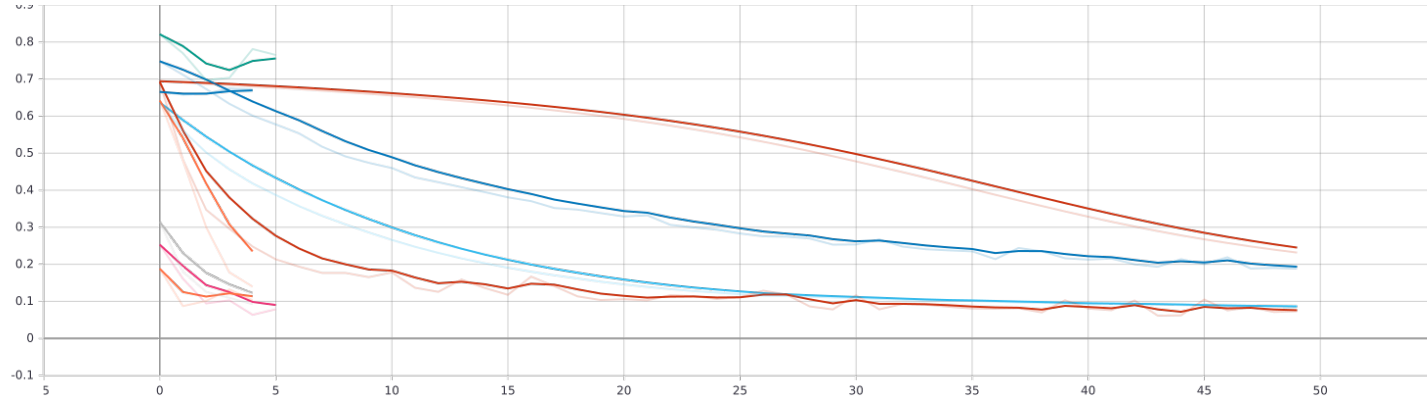


Figure 7: Train and validation loss plotted against epochs

Summary for plots
The plots shows that some on the models stop at certain epochs because the parameters tried by the optimizer have a very large generalization gap and do not improve on their metrics. This will be more clear as we isolate the best model to analyze its results.

3.5 Results for best trail

In this section of this paper we isolate the best trail to evaluate it further. For this project our aim is to get the best model, evaluate it and use it. First we check the result summary gained from the optimizer.

```
[Trial summary]
 |-Trial ID: 43ffb59cf1bece26e9cedf1c4e2df84b
 |-Score: 0.9824561476707458
 |-Best step: 0
 > Hyperparameters:
 |-Layers: 0
 |-activations: relu
 |-dropout_rate: 0.2
 |-learning_rate: 0.0001
 |-units: 256
```

Figure 8: This is the trial summary for the best trial

So now I have identified the best model and its hyper parameters. The validation accuracy is 0.982 and validation loss is 0.0925. So the next step is to plot the validation and train data for the accuracy and loss metric to check how it performed.

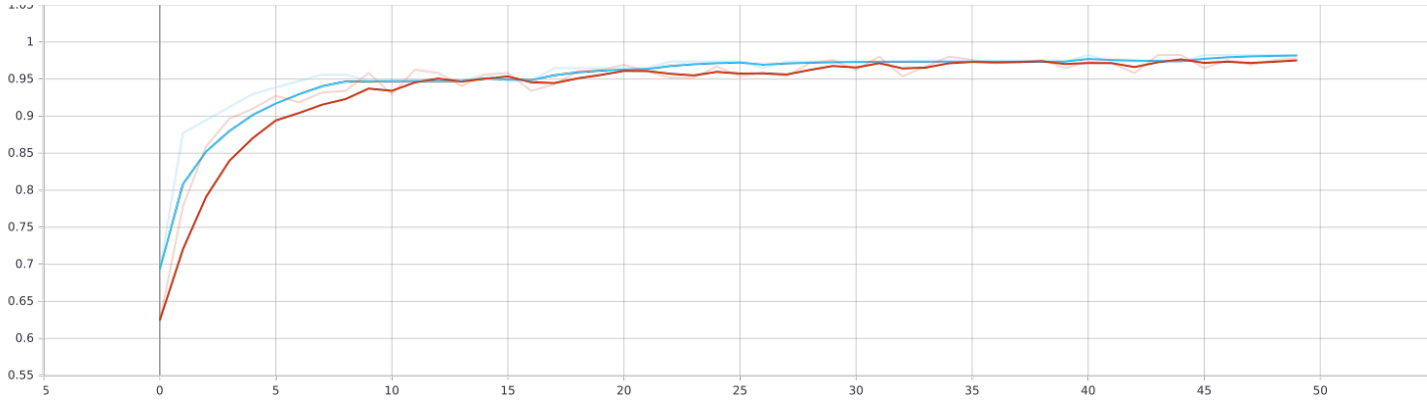


Figure 9: Train and validation accuracy plotted against epochs
Red: Train data
Blue: Validation data

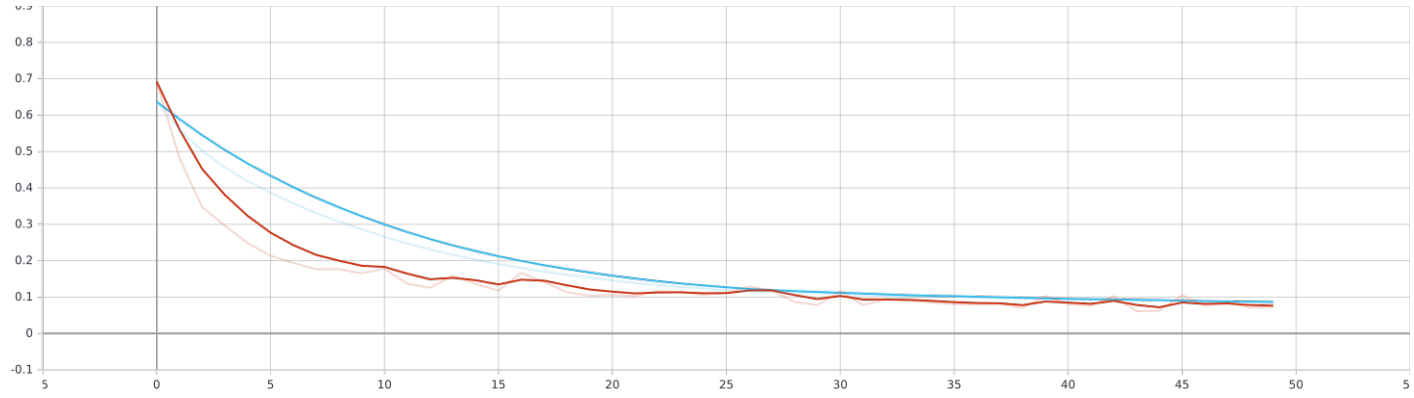


Figure 10: Train and validation loss plotted against epochs
Red: Train data
Blue: Validation data

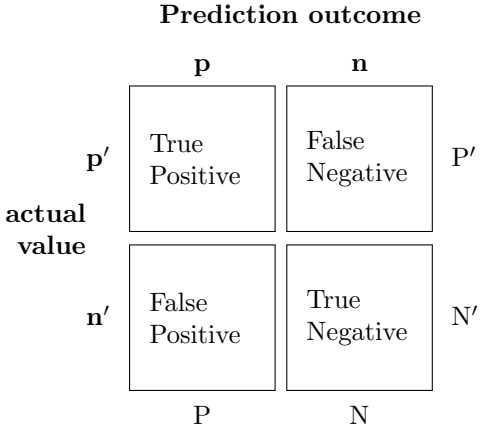
Summary for plots

If you ask me this is a great result. There is no sign of underfitting and the generalization gap is very small in the loss function. And the train and validation converge nicely. This is a result I take seriously and do

further evaluation tests on. Because my task was classification is suited to make a confusion matrix and ROC (operator curve) analysis to evaluate the model further.

3.5.1 Further evaluations on the best model

To be sure that a model is in a good state there should be a lot of evaluations made. For this paper I will demonstrate a confusion matrix produced by the best model created by the optimizer and a ROC evaluation.



This are the results the best model produced for the confusion matrix.
True Positive: 72
False negative: 0
False positive: 2
True negative: 40

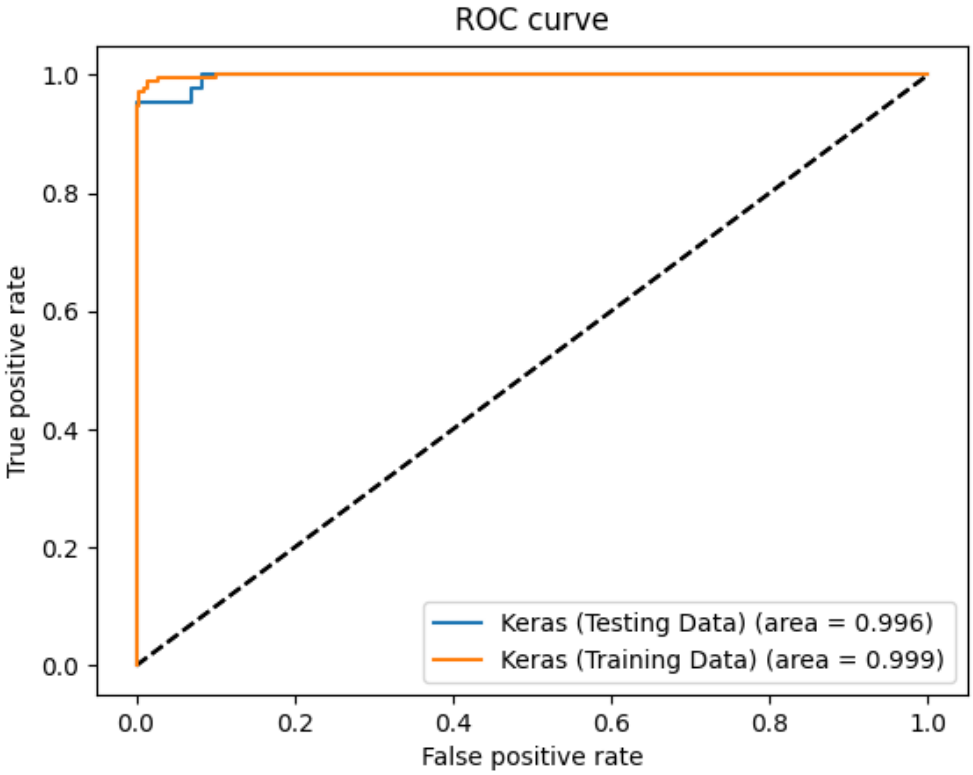


Figure 11: Operator curve on train and test data for the best model

4 Summary and Conclusions

When should I take usage of such a thing as a tuning optimizer? If you ask me its when you already have a working model providing you with decent results already. And you know you could find extra validation accuracy. I think this is always a need to do when you are at you final step at finalising your model or specially working with CNN's. And its a great way to test different model configuration. You could always be surprised too. The third best model was using softmax as its activation function to create a model with 94% validation accuracy, that's not to bad. Also combining the tuning with Tensorboard as a callback gives a very good intuition of how your progress is going. And give a great visualisation tool.

Using a optimizer like this is also depending on you task. I usually work with univariate and multivariate regression tasks. The way to analyze the results for the accuracy may wary so that's something to be take in consideration.

Another thing to mention is that our data already had great features to work with. In real life situation the data processing part would require more effort and time to create. But as I mentioned in the beginning of this paper the project was to demonstrate what a tuner could provide.

Appendix A Keras documentation links

Link: `https://keras-team.github.io/keras-tuner/documentation/tuners/#randomsearch-class`
Link: `https://keras-team.github.io/keras-tuner/documentation/hyperparameters/`