Use Wine Data from previous Lab

You can use the following code to import it to Python and print a random couple of rows:‌

import os

import numpy as np

import pandas as pd

import tensorflow as tf

import itertools

import warnings

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

tf.random.set\_seed(42)

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'

warnings.filterwarnings('ignore')

df = pd.read\_csv('data/winequalityN.csv')

df.sample(5)

We’re ignoring the warnings and changing the default TensorFlow log level just so we don’t get overwhelmed with the output.

Here’s how the dataset looks like:‌

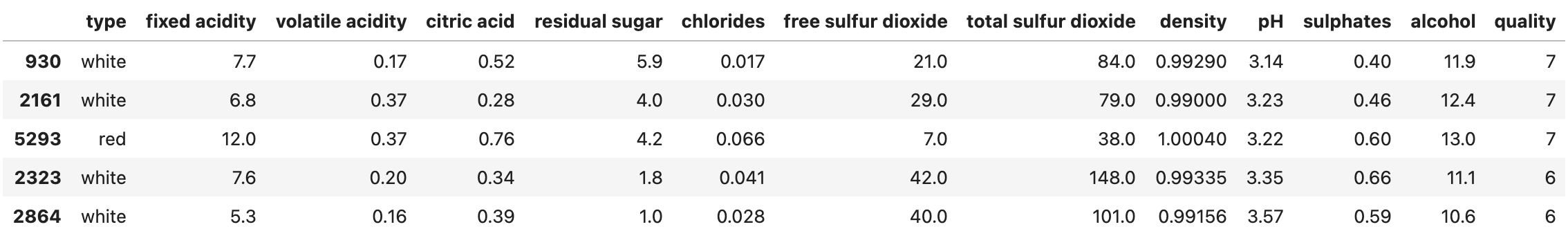


Image 2 — A random sample of the wine quality dataset (image by author)

‌The dataset is mostly clean, but isn’t designed for binary classification by default (good/bad wine). Instead, the wines are rated on a scale. We’ll address that now, with numerous other things:

* Delete missing values — There’s only a handful of them, so we won’t waste time on imputation.
* Handle categorical features — The only one is type, indicating whether the wine is white or red.
* Convert to a binary classification task — We’ll declare any wine with a grade of 6 and above as *good*, and anything below as *bad*.
* Train/test split — A classic 80:20 split.
* Scale the data — The scale between predictors differs significantly, so we’ll use the StandardScaler to bring the values closer.

Here’s the entire data preprocessing code snippet:‌

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

# Prepare the data

df = df.dropna()

df['is\_white\_wine'] = [1 if typ == 'white' else 0 for typ in df['type']]

df['is\_good\_wine'] = [1 if quality >= 6 else 0 for quality in df['quality']]

df.drop(['type', 'quality'], axis=1, inplace=True)

# Train/test split

X = df.drop('is\_good\_wine', axis=1)

y = df['is\_good\_wine']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y,

test\_size=0.2, random\_state=42

)

# Scaling

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

Once again, please refer to the [previous article](https://betterdatascience.com/how-to-train-a-classification-model-with-tensorflow-in-10-minutes/) if you want more detailed insights into the logic behind data preprocessing.

With that out of the way, let’s see how to approach optimizing neural network architectures.

## **How to approach optimizing neural network models?**

The approach to finding the optimal neural network model will have some tweakable constants. Today’s network will have 3 hidden layers, with a minimum of 64 and a maximum of 256 nodes per layer. We’ll set the step size between nodes to 64, so the possibilities are 64, 128, 192, and 256:‌

num\_layers = 3

min\_nodes\_per\_layer = 64

max\_nodes\_per\_layer = 256

node\_step\_size = 64

Let’s verify the node number possibilities. You can do so by creating a list of ranges between the minimum and maximum number of nodes, having the step size in mind:‌

node\_options = list(range(

min\_nodes\_per\_layer,

max\_nodes\_per\_layer + 1,

node\_step\_size

))

Here’s what you’ll see:‌

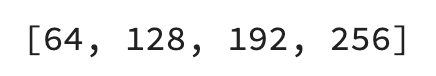


Image 3 — Node number possibilities (image by author)

‌Taking this logic to two hidden layers, you end up with the following possibilities:‌

two\_layer\_possibilities = [node\_options, node\_options]

Or visually:‌

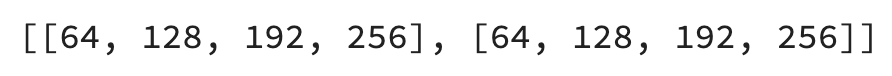
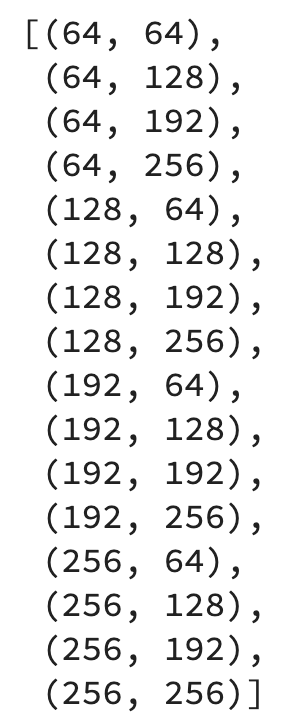


Image 4 — Node number possibilities for two hidden layers (image by author)

‌To get every possible permutation of the options among two layers, you can use the product() function from itertools:‌

list(itertools.product(\*two\_layer\_possibilities))

Here’s the output:‌

Image 5 — Two layer deep neural network architecture permutations (image by author)

The goal is to optimize a 3-layer-deep neural network, so we’ll end up with a bit more permutations. You can declare the possibilities by first multiplying the list of node options with num\_layers and then calculate the permutations:‌

layer\_possibilities = [node\_options] \* num\_layers

layer\_node\_permutations = list(itertools.product(\*layer\_possibilities))

It’s a lot of options — 64 in total. During optimization, we’ll iterate over the permutations and then iterate again over the values of the individual permutation to get the node counts for each hidden layer.

In short, we’ll have two for loops. Here’s the logic for the first two permutations:‌

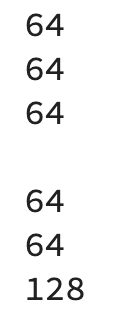
for permutation in layer\_node\_permutations[:2]:

for nodes\_at\_layer in permutation:

print(nodes\_at\_layer)

print()

The second print statement is here just to make a gap between models, so don’t think too much of it. Here’s the output:‌

Image 6 — Number of nodes at each layer (image by author)

We’ll create a new tf.keras.Sequential model at each iteration and add a tf.keras.layers.InputLayer to it with a shape of a single training row ((12,)). Then, we’ll iterate over the items in a single permutation and add a tf.keras.layers.Dense layer to the model with the number of nodes set to the current value of the single permutation. Finally, we’ll add a tf.keras.layers.Dense output layer.

It’s a good idea to set the name to the model, so it’s easier to compare them later. We’ll hardcode the input shape and the activation functions for no, and set these parts as dynamic in the next section.

Here’s the code:‌

models = []

for permutation in layer\_node\_permutations:

model = tf.keras.Sequential()

model.add(tf.keras.layers.InputLayer(input\_shape=(12,)))

model\_name = ''

for nodes\_at\_layer in permutation:

model.add(tf.keras.layers.Dense(nodes\_at\_layer, activation='relu'))

model\_name += f'dense{nodes\_at\_layer}\_'

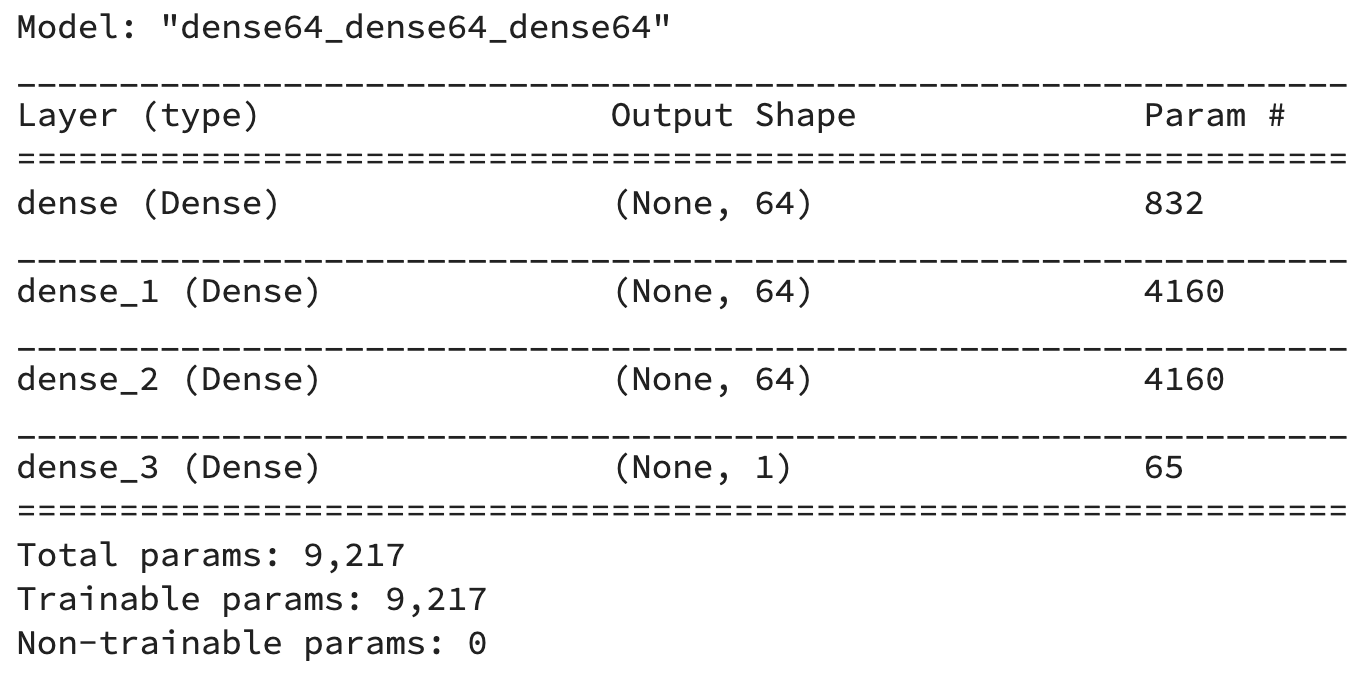
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

model.\_name = model\_name[:-1]

models.append(model)

And now let’s inspect how a single model looks like:‌

models[0].summary()

Image 7 — Single model architecture (image by author)

That’s the logic we’ll go with. There’s a way to improve it, though, as it’s not convenient to run dozens of notebook cells every time you want to run the optimization. It’s also not the best idea to hardcode values for activation functions, input shape, and so on.

For that reason, we’ll declare a function for generating Sequential models next.

## **Model generation function for optimizing neural networks**

The function accepts a lot of parameters but doesn’t contain anything we didn’t cover previously. It gives you the option to change the input shape, activation function for the hidden and output layer, and the number of nodes at the output layer.

Here’s the code:‌

def get\_models(num\_layers: int,

min\_nodes\_per\_layer: int,

max\_nodes\_per\_layer: int,

node\_step\_size: int,

input\_shape: tuple,

hidden\_layer\_activation: str = 'relu',

num\_nodes\_at\_output: int = 1,

output\_layer\_activation: str = 'sigmoid') -> list:

node\_options = list(range(min\_nodes\_per\_layer, max\_nodes\_per\_layer + 1, node\_step\_size))

layer\_possibilities = [node\_options] \* num\_layers

layer\_node\_permutations = list(itertools.product(\*layer\_possibilities))

models = []

for permutation in layer\_node\_permutations:

model = tf.keras.Sequential()

model.add(tf.keras.layers.InputLayer(input\_shape=input\_shape))

model\_name = ''

for nodes\_at\_layer in permutation:

model.add(tf.keras.layers.Dense(nodes\_at\_layer, activation=hidden\_layer\_activation))

model\_name += f'dense{nodes\_at\_layer}\_'

model.add(tf.keras.layers.Dense(num\_nodes\_at\_output, activation=output\_layer\_activation))

model.\_name = model\_name[:-1]

models.append(model)

return models

Let’s test it — we’ll stick to a model with three hidden layers, each having a minimum of 64 and a maximum of 256 nodes:‌

all\_models = get\_models(

num\_layers=3,

min\_nodes\_per\_layer=64,

max\_nodes\_per\_layer=256,

node\_step\_size=64,

input\_shape=(12,)

)

Feel free to inspect the values of the all\_models list. It contains 64 Sequential models, each having a unique name and architecture. Training so many models will take time, so let’s make things extra simple by writing yet another helper function.

## **Model training function for optimizing neural networks**

This one accepts the list of models, training and testing data, and optionally a number of epochs and the verbosity level. It’s advised to set verbosity to 0, so you don’t get overwhelmed with the console output. The function returns a Pandas DataFrame containing the performance metrics on the test set, measured in accuracy, precision, recall, and F1.

Here’s the code:‌

def optimize(models: list,

X\_train: np.array,

y\_train: np.array,

X\_test: np.array,

y\_test: np.array,

epochs: int = 50,

verbose: int = 0) -> pd.DataFrame:

# We'll store the results here

results = []

def train(model: tf.keras.Sequential) -> dict:

# Change this however you want

# We're not optimizing this part today

model.compile(

loss=tf.keras.losses.binary\_crossentropy,

optimizer=tf.keras.optimizers.Adam(),

metrics=[

tf.keras.metrics.BinaryAccuracy(name='accuracy')

]

)

# Train the model

model.fit(

X\_train,

y\_train,

epochs=epochs,

verbose=verbose

)

# Make predictions on the test set

preds = model.predict(X\_test)

prediction\_classes = [1 if prob > 0.5 else 0 for prob in np.ravel(preds)]

# Return evaluation metrics on the test set

return {

'model\_name': model.name,

'test\_accuracy': accuracy\_score(y\_test, prediction\_classes),

'test\_precision': precision\_score(y\_test, prediction\_classes),

'test\_recall': recall\_score(y\_test, prediction\_classes),

'test\_f1': f1\_score(y\_test, prediction\_classes)

}

# Train every model and save results

for model in models:

try:

print(model.name, end=' ... ')

res = train(model=model)

results.append(res)

except Exception as e:

print(f'{model.name} --> {str(e)}')

return pd.DataFrame(results)

And now, let’s finally start the optimization.

## **Running the optimization**

Keep in mind — the optimization will take some time, as we’re training 64 models for 50 epochs. Here’s how to start the process:‌

optimization\_results = optimize(

models=all\_models,

X\_train=X\_train\_scaled,

y\_train=y\_train,

X\_test=X\_test\_scaled,

y\_test=y\_test

)

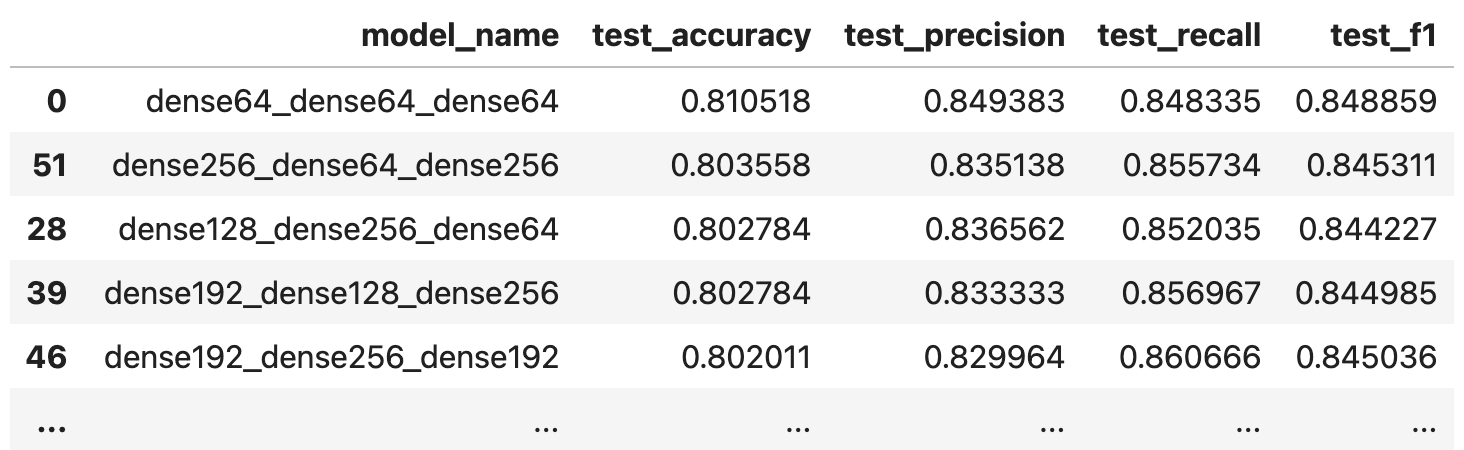
The optimization ran for 34 minutes on my machine (M1 MacBook Pro) and printed the following:‌

Image 8 — Optimization output (image by author)

You’re seeing this output because of the print() statement in the optimize() function. It’s there to give you a sense of the progress.

We now have a DataFrame we can sort either by accuracy, precision, recall, or F1. Here’s how to sort it by accuracy in descending order, so the model with the highest value is displayed first:‌

optimization\_results.sort\_values(by='test\_accuracy', ascending=False)

Image 9 — Model optimization results (image by author)

It looks like the simplest model resulted in the best accuracy. You could also test the optimization for models with two and four hidden layers, or even more, but I’ll leave that up to you. It’s just a matter of calling the get\_models() function and passing in different parameter values.