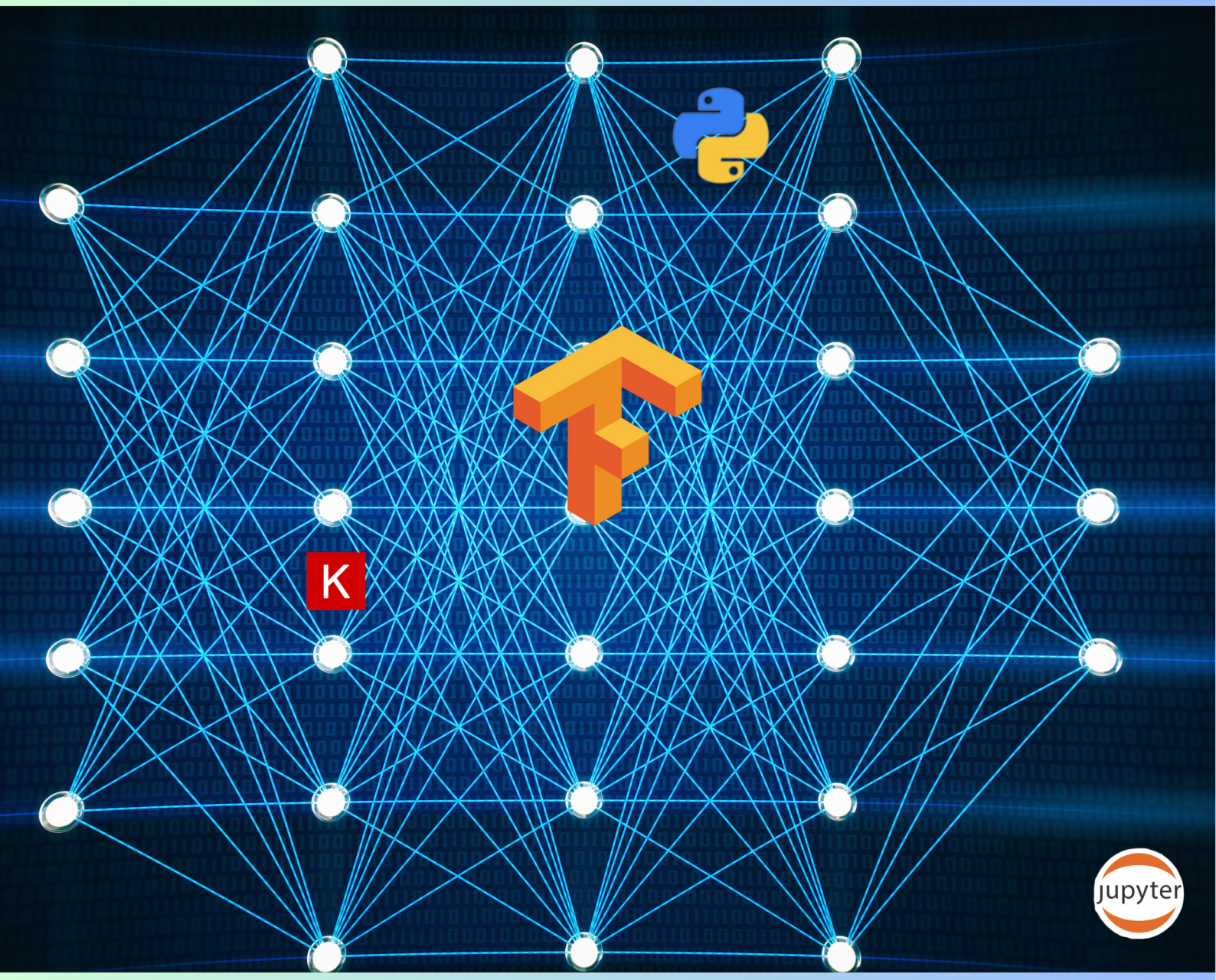
Fundamentals of Deep Learning

Build simple neural networks with TensorFlow



Overview of key functions and steps of building simple, sequential neural networks

Shivang Kainthola

Simple Neural Networks with Tensorflow

June 13, 2024

[ ]:
```
import tensorflow as tf
```

1. Building fully connected, sequential neural network sturctures :

To define a network :

[ ]:
```
from tensorflow.keras.models import Sequential
```

To define a fully connected layer :

[ ]:
```
from tensorflow.keras.layers import Dense
```

[ ]:
```
It includes the number of nodes and the activation function for that layer, for␣
↪example :
tf.keras.layers.Dense(64, activation='relu')
```

> model.summary() - Shows the structure of the model

2. Loss functions :

Can be defined with imported functions, or simply declared.

[ ]:
```
from tensorflow.keras.losses
```

within which we access the loss functions like :

[ ]:
```
from tensorflow.keras.losses import MeanSquaredError
```

[ ]:
```
from tensorflow.keras.losses import MeanAbsoluteError
```

[ ]:
```
from tensorflow.keras.losses import BinaryCrossentropy
```

[ ]:
```
from tensorflow.keras.losses import CategoricalCrossentropy
```

[ ]:
```
from tensorflow.keras.losses import SparseCategoricalCrossentropy
```

[ ]:
```
from tensorflow.keras.losses import Hinge
```

3. Optimizers :

```
[ ]: import tensorflow.keras.optimizers
```

from which, we can access various optimizers like :

```
[ ]: from tensorflow.keras.optimizers import SGD
     from tensorflow.keras.optimizers import Adam
     from tensorflow.keras.optimizers import RMSprop
     from tensorflow.keras.optimizers import Adagrad
```

More functions exist for building parallel, convolutional and recurrent neural networks.

4. Compiling and fitting neural networks :

After building the network structure, you compile it with the following parameters :

1) optimizer : The optimizer to be used, also takes the initial learning rate as argument
2) loss : Takes the loss function to be used
3) metrics : Specify the metrics that the model will calculate for its performance

```
[ ]: model.compile(optimizer= '<>', loss = '<>', metrics = ['<>',])
```

Once the neural net is compiled, you fit/train it on the training data with the following parameters :

1) epochs : Number of epochs
2) validation_data : Pass the test/validation split
3) callbacks : Specify callback features for learning rate schedulers, early stopping etc.
4) verbose : As the model is fitted, for every epoch the model can report its fitting time, accuracy, etc. If verbose is set to 1, this history is printed as output. The entire history log can also be saved by assigning to a variable.

```
[ ]: history = model.fit(X_train, y_train, epochs = <>, validation_data =␣
      ↪(x_valid,y_valid), callbacks ='')
```

The 'histroy' variable contains the logs of the model's fitting history, and can be printed, or accessed.

It is used to visualise the model's performance with the number of epochs.

5. Generating predictions and evaluating the neural net :

model.predict(<>) : Returns model's output for the given input data

```
[ ]: predictions = model.predict(input)
```

model.evaluate(X,y) : Returns the metrics calculated for the model's output for the given data

6. Saving and loading the model

A neural network model can be saved with the function :

[ ]:
```
model.save('/directory/<name>.h5')
```

[ ]:
```
This saves the model with its architecture and weights in the directory of the
↪given path.
```

A saved model can be loaded with the function :

tensorflow.keras.models.load_model('model.h5')

[ ]:
```
model_b = tensorflow.keras.models.load_model('model.h5')
```

Putting it all together with an example :

We will use the Boston Housing dataset for price prediction, which is a regression problem.

Loading and preparing data :

[1]:
```
import tensorflow as tf
from tensorflow.keras.datasets import boston_housing
import numpy as np
import matplotlib.pyplot as plt
```

[3]:
```
(x_train, y_train), (x_test, y_test) = boston_housing.load_data()
```

[5]:
```
mean = x_train.mean(axis=0)
std = x_train.std(axis=0)
x_train = (x_train - mean) / std
x_test = (x_test - mean) / std
```

a) Build a model structure :

1) We are building a simple model with one hidden layer.
2) The last layer has one parameter, because our output is one value - the predicted price. In cases of classification problems, the output layer has a specific type of activation function.

[7]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model1 = Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(13,)),
    tf.keras.layers.Dense(32,activation='relu'),
```

```
    tf.keras.layers.Dense(1)
])
```

```
C:\Users\Asus\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

b) Compile the model with the appropriate loss functions, optimizer

The problem is predicting a value, so we can use the Mean Absolute Error loss function.

The Adam optimizer is used, and the learning rate is set to 0.001 initially.

The metric passed is Mean absolute error, which is appropriate for this regression problem.

[9]:
```
from tensorflow.keras.optimizers import Adam

model1.compile(optimizer=Adam(learning_rate=0.001),
               loss='mean_squared_error',
               metrics=['mean_absolute_error'])
```

c) Fitting the model for 10 epochs

We are not defining any callbacks like learning rate schedulers, because this problem does not merit using them.

[11]:
```
history = model1.fit(x_train, y_train, epochs=10, validation_data=(x_test,␣
↪y_test), verbose = 1)
```

```
Epoch 1/10
13/13             4s 41ms/step -
loss: 572.5272 - mean_absolute_error: 22.0269 - val_loss: 573.9330 -
val_mean_absolute_error: 22.1665
Epoch 2/10
13/13             0s 8ms/step - loss:
535.7487 - mean_absolute_error: 21.3350 - val_loss: 520.8567 -
val_mean_absolute_error: 20.9692
Epoch 3/10
13/13             0s 7ms/step - loss:
460.9495 - mean_absolute_error: 19.6534 - val_loss: 454.6360 -
val_mean_absolute_error: 19.3768
Epoch 4/10
13/13             0s 7ms/step - loss:
411.2466 - mean_absolute_error: 18.2183 - val_loss: 372.1741 -
val_mean_absolute_error: 17.2398
Epoch 5/10
13/13             0s 7ms/step - loss:
```

```
309.1825 - mean_absolute_error: 15.6262 - val_loss: 277.9571 -
val_mean_absolute_error: 14.6537
Epoch 6/10
13/13             0s 7ms/step - loss:
232.6413 - mean_absolute_error: 13.1835 - val_loss: 187.0619 -
val_mean_absolute_error: 11.7838
Epoch 7/10
13/13             0s 7ms/step - loss:
148.3632 - mean_absolute_error: 9.8449 - val_loss: 119.1460 -
val_mean_absolute_error: 9.2493
Epoch 8/10
13/13             0s 7ms/step - loss:
95.1376 - mean_absolute_error: 7.7539 - val_loss: 79.4885 -
val_mean_absolute_error: 7.2479
Epoch 9/10
13/13             0s 7ms/step - loss:
67.5665 - mean_absolute_error: 6.1287 - val_loss: 60.8438 -
val_mean_absolute_error: 6.0864
Epoch 10/10
13/13             0s 7ms/step - loss:
44.6259 - mean_absolute_error: 5.1322 - val_loss: 49.3962 -
val_mean_absolute_error: 5.4726
```

d) Visualise the model's peformance and evaluate it

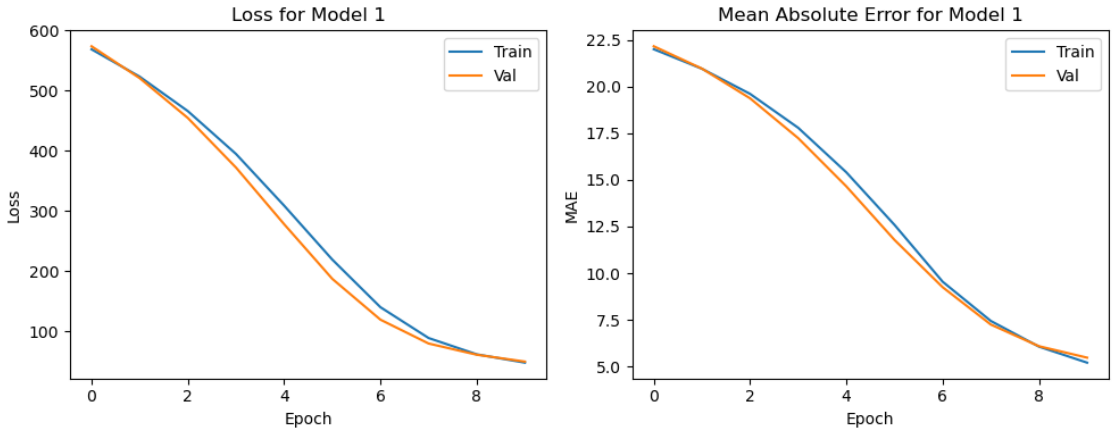
We can access the 'history' variable to visualise the model's training performance

[15]:
```
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Loss for Model 1')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Val'])

plt.subplot(1, 2, 2)
plt.plot(history.history['mean_absolute_error'])
plt.plot(history.history['val_mean_absolute_error'])
plt.title('Mean Absolute Error for Model 1')
plt.xlabel('Epoch')
plt.ylabel('MAE')
plt.legend(['Train', 'Val'])

plt.tight_layout()
plt.show()
```



We can see that the mean absolute error and the loss are both decreasing as the number of epochs increase, so training it for more epochs will better its performance.

However, simply fitting the model for more epochs will make it go through data it has already processed. So, we will make a copy of the original model.

e) Training a copy of the model for 20 more epochs than previous model.

[17]:
```
model2 = Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(13,)),
    tf.keras.layers.Dense(32,activation='relu'),
    tf.keras.layers.Dense(1)
])
```

[19]:
```
model2.compile(optimizer=Adam(learning_rate=0.001),
               loss='mean_squared_error',
               metrics=['mean_absolute_error'])
```

We will train it for 30 epochs :

[21]:
```
history2 = model2.fit(x_train, y_train, epochs=30, validation_data=(x_test,␣
 ↪y_test), verbose = 1)
```

```
Epoch 1/30
13/13            3s 41ms/step -
loss: 594.1405 - mean_absolute_error: 22.5879 - val_loss: 594.0067 -
val_mean_absolute_error: 22.6239
Epoch 2/30
13/13            0s 8ms/step - loss:
609.6848 - mean_absolute_error: 22.6915 - val_loss: 551.3209 -
val_mean_absolute_error: 21.6766
Epoch 3/30
```

```
13/13            0s 7ms/step - loss:
492.2885 - mean_absolute_error: 20.3524 - val_loss: 504.6153 -
val_mean_absolute_error: 20.5737
Epoch 4/30
13/13            0s 7ms/step - loss:
463.9328 - mean_absolute_error: 19.6265 - val_loss: 444.7930 -
val_mean_absolute_error: 19.0904
Epoch 5/30
13/13            0s 7ms/step - loss:
418.8248 - mean_absolute_error: 18.2134 - val_loss: 368.8014 -
val_mean_absolute_error: 17.1096
Epoch 6/30
13/13            0s 8ms/step - loss:
324.6862 - mean_absolute_error: 15.8964 - val_loss: 280.3506 -
val_mean_absolute_error: 14.5779
Epoch 7/30
13/13            0s 10ms/step -
loss: 239.6012 - mean_absolute_error: 13.3595 - val_loss: 193.6961 -
val_mean_absolute_error: 11.8011
Epoch 8/30
13/13            0s 11ms/step -
loss: 171.9527 - mean_absolute_error: 10.7745 - val_loss: 126.9877 -
val_mean_absolute_error: 9.2474
Epoch 9/30
13/13            0s 12ms/step -
loss: 106.5818 - mean_absolute_error: 8.0888 - val_loss: 90.7710 -
val_mean_absolute_error: 7.5779
Epoch 10/30
13/13            0s 10ms/step -
loss: 79.9429 - mean_absolute_error: 6.8769 - val_loss: 72.5531 -
val_mean_absolute_error: 6.7393
Epoch 11/30
13/13            0s 17ms/step -
loss: 61.6628 - mean_absolute_error: 6.1812 - val_loss: 59.0805 -
val_mean_absolute_error: 6.1540
Epoch 12/30
13/13            0s 14ms/step -
loss: 48.0221 - mean_absolute_error: 5.2373 - val_loss: 49.4832 -
val_mean_absolute_error: 5.6655
Epoch 13/30
13/13            0s 10ms/step -
loss: 49.5146 - mean_absolute_error: 5.1804 - val_loss: 42.8540 -
val_mean_absolute_error: 5.2577
Epoch 14/30
13/13            0s 9ms/step - loss:
34.5063 - mean_absolute_error: 4.2571 - val_loss: 38.0195 -
val_mean_absolute_error: 4.8931
Epoch 15/30
```

```
13/13            0s 9ms/step - loss:
27.3481 - mean_absolute_error: 3.9027 - val_loss: 34.6767 -
val_mean_absolute_error: 4.6626
Epoch 16/30
13/13            0s 9ms/step - loss:
23.6294 - mean_absolute_error: 3.5204 - val_loss: 32.6898 -
val_mean_absolute_error: 4.5154
Epoch 17/30
13/13            0s 8ms/step - loss:
23.7159 - mean_absolute_error: 3.6145 - val_loss: 31.1923 -
val_mean_absolute_error: 4.4084
Epoch 18/30
13/13            0s 9ms/step - loss:
22.9391 - mean_absolute_error: 3.5567 - val_loss: 30.2274 -
val_mean_absolute_error: 4.3066
Epoch 19/30
13/13            0s 9ms/step - loss:
20.0341 - mean_absolute_error: 3.2688 - val_loss: 28.9498 -
val_mean_absolute_error: 4.1907
Epoch 20/30
13/13            0s 8ms/step - loss:
19.9055 - mean_absolute_error: 3.2805 - val_loss: 28.1945 -
val_mean_absolute_error: 4.1231
Epoch 21/30
13/13            0s 9ms/step - loss:
18.4579 - mean_absolute_error: 3.1652 - val_loss: 27.5585 -
val_mean_absolute_error: 4.0467
Epoch 22/30
13/13            0s 10ms/step -
loss: 19.8132 - mean_absolute_error: 3.1984 - val_loss: 27.3886 -
val_mean_absolute_error: 4.0000
Epoch 23/30
13/13            0s 15ms/step -
loss: 19.6061 - mean_absolute_error: 3.0565 - val_loss: 26.6281 -
val_mean_absolute_error: 3.9217
Epoch 24/30
13/13            0s 14ms/step -
loss: 17.0582 - mean_absolute_error: 2.9404 - val_loss: 26.1154 -
val_mean_absolute_error: 3.8596
Epoch 25/30
13/13            0s 8ms/step - loss:
20.6256 - mean_absolute_error: 3.1839 - val_loss: 26.3129 -
val_mean_absolute_error: 3.8516
Epoch 26/30
13/13            0s 9ms/step - loss:
14.0676 - mean_absolute_error: 2.7849 - val_loss: 25.5456 -
val_mean_absolute_error: 3.7802
Epoch 27/30
```

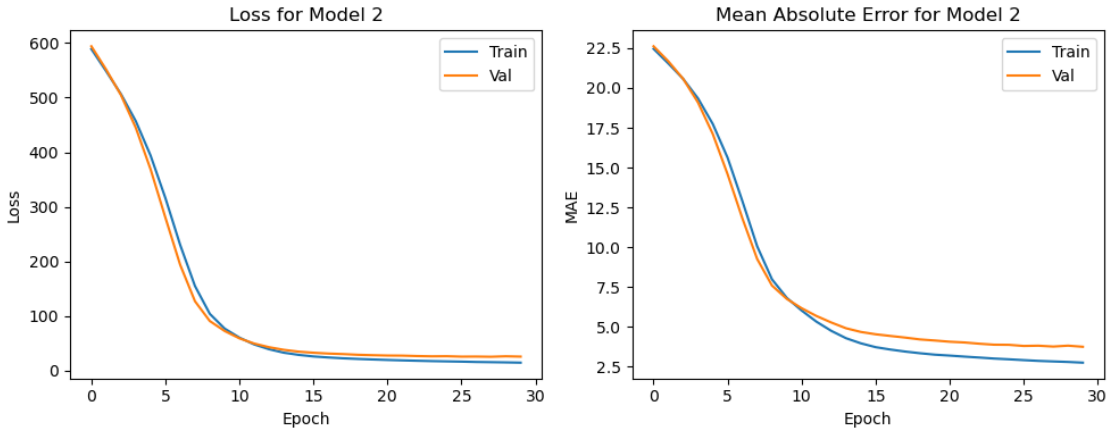
```
13/13             0s 9ms/step - loss:
14.3410 - mean_absolute_error: 2.7250 - val_loss: 25.6759 -
val_mean_absolute_error: 3.7961
Epoch 28/30
13/13             0s 10ms/step -
loss: 13.4167 - mean_absolute_error: 2.7098 - val_loss: 25.3253 -
val_mean_absolute_error: 3.7427
Epoch 29/30
13/13             0s 9ms/step - loss:
15.5074 - mean_absolute_error: 2.8120 - val_loss: 26.1840 -
val_mean_absolute_error: 3.7986
Epoch 30/30
13/13             0s 9ms/step - loss:
14.3130 - mean_absolute_error: 2.7016 - val_loss: 25.5900 -
val_mean_absolute_error: 3.7284
```

[23]:
```
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.plot(history2.history['loss'])
plt.plot(history2.history['val_loss'])
plt.title('Loss for Model 2')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Val'])

plt.subplot(1, 2, 2)
plt.plot(history2.history['mean_absolute_error'])
plt.plot(history2.history['val_mean_absolute_error'])
plt.title('Mean Absolute Error for Model 2')
plt.xlabel('Epoch')
plt.ylabel('MAE')
plt.legend(['Train', 'Val'])

plt.tight_layout()
plt.show()
```



On training for 30 epochs, the neural network has improved significantly, reaching mean absolute error value of 2.5, which is fantastic.

Can this model be improved even further by training for 50 epochs ?

What if we add one more hidden layer ?

Let us try.

f) Training a copy of the model, with an added layer, for 50 epochs :

[25]:
```
model3 = Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(13,)),
    tf.keras.layers.Dense(32,activation='relu'),
    tf.keras.layers.Dense(32,activation='relu'),
    tf.keras.layers.Dense(1)
])
```

[27]:
```
model3.compile(optimizer=Adam(learning_rate=0.001),
               loss='mean_squared_error',
               metrics=['mean_absolute_error'])
```

[29]:
```
history3 = model3.fit(x_train, y_train, epochs=50, validation_data=(x_test,␣
↪y_test), verbose = 1)
```

```
Epoch 1/50
13/13             4s 38ms/step -
loss: 557.6638 - mean_absolute_error: 21.7952 - val_loss: 571.7007 -
val_mean_absolute_error: 22.1264
Epoch 2/50
13/13             0s 7ms/step - loss:
554.6521 - mean_absolute_error: 21.5740 - val_loss: 510.0485 -
val_mean_absolute_error: 20.7613
```

```
Epoch 3/50
13/13             0s 7ms/step - loss:
468.8989 - mean_absolute_error: 19.6830 - val_loss: 404.4833 -
val_mean_absolute_error: 18.2275
Epoch 4/50
13/13             0s 7ms/step - loss:
363.0786 - mean_absolute_error: 17.0869 - val_loss: 248.0837 -
val_mean_absolute_error: 13.7734
Epoch 5/50
13/13             0s 7ms/step - loss:
176.7868 - mean_absolute_error: 11.2650 - val_loss: 105.0300 -
val_mean_absolute_error: 8.6081
Epoch 6/50
13/13             0s 7ms/step - loss:
96.4662 - mean_absolute_error: 7.3362 - val_loss: 70.5858 -
val_mean_absolute_error: 6.5212
Epoch 7/50
13/13             0s 7ms/step - loss:
57.8828 - mean_absolute_error: 5.6669 - val_loss: 51.9977 -
val_mean_absolute_error: 5.5904
Epoch 8/50
13/13             0s 7ms/step - loss:
49.8437 - mean_absolute_error: 5.2117 - val_loss: 41.9657 -
val_mean_absolute_error: 5.0673
Epoch 9/50
13/13             0s 7ms/step - loss:
35.4682 - mean_absolute_error: 4.2101 - val_loss: 35.6017 -
val_mean_absolute_error: 4.6157
Epoch 10/50
13/13             0s 7ms/step - loss:
24.8939 - mean_absolute_error: 3.6114 - val_loss: 32.1190 -
val_mean_absolute_error: 4.3893
Epoch 11/50
13/13             0s 8ms/step - loss:
22.9367 - mean_absolute_error: 3.4621 - val_loss: 29.8565 -
val_mean_absolute_error: 4.2134
Epoch 12/50
13/13             0s 7ms/step - loss:
19.8005 - mean_absolute_error: 3.2623 - val_loss: 28.2047 -
val_mean_absolute_error: 4.0393
Epoch 13/50
13/13             0s 7ms/step - loss:
20.6227 - mean_absolute_error: 3.1297 - val_loss: 27.6167 -
val_mean_absolute_error: 3.9710
Epoch 14/50
13/13             0s 7ms/step - loss:
24.4384 - mean_absolute_error: 3.3367 - val_loss: 26.3295 -
val_mean_absolute_error: 3.7922
```

```
Epoch 15/50
13/13            0s 7ms/step - loss:
19.4644 - mean_absolute_error: 3.0773 - val_loss: 25.7097 -
val_mean_absolute_error: 3.7166
Epoch 16/50
13/13            0s 7ms/step - loss:
20.2172 - mean_absolute_error: 3.1641 - val_loss: 25.2208 -
val_mean_absolute_error: 3.6495
Epoch 17/50
13/13            0s 7ms/step - loss:
18.9150 - mean_absolute_error: 2.9942 - val_loss: 25.1570 -
val_mean_absolute_error: 3.6230
Epoch 18/50
13/13            0s 7ms/step - loss:
15.4115 - mean_absolute_error: 2.9243 - val_loss: 25.4953 -
val_mean_absolute_error: 3.6081
Epoch 19/50
13/13            0s 7ms/step - loss:
12.7681 - mean_absolute_error: 2.6322 - val_loss: 25.0407 -
val_mean_absolute_error: 3.5445
Epoch 20/50
13/13            0s 7ms/step - loss:
15.3113 - mean_absolute_error: 2.6546 - val_loss: 25.9372 -
val_mean_absolute_error: 3.5822
Epoch 21/50
13/13            0s 7ms/step - loss:
13.2084 - mean_absolute_error: 2.6525 - val_loss: 24.6031 -
val_mean_absolute_error: 3.4524
Epoch 22/50
13/13            0s 7ms/step - loss:
13.2268 - mean_absolute_error: 2.6529 - val_loss: 24.6303 -
val_mean_absolute_error: 3.4372
Epoch 23/50
13/13            0s 7ms/step - loss:
14.1178 - mean_absolute_error: 2.6365 - val_loss: 26.6394 -
val_mean_absolute_error: 3.5421
Epoch 24/50
13/13            0s 7ms/step - loss:
12.1641 - mean_absolute_error: 2.4718 - val_loss: 25.2400 -
val_mean_absolute_error: 3.4183
Epoch 25/50
13/13            0s 7ms/step - loss:
10.9408 - mean_absolute_error: 2.3490 - val_loss: 25.1490 -
val_mean_absolute_error: 3.3917
Epoch 26/50
13/13            0s 9ms/step - loss:
12.9544 - mean_absolute_error: 2.5838 - val_loss: 25.8251 -
val_mean_absolute_error: 3.4142
```

```
Epoch 27/50
13/13             0s 9ms/step - loss:
14.5606 - mean_absolute_error: 2.5484 - val_loss: 24.8753 -
val_mean_absolute_error: 3.3316
Epoch 28/50
13/13             0s 7ms/step - loss:
12.0952 - mean_absolute_error: 2.4736 - val_loss: 24.9819 -
val_mean_absolute_error: 3.3215
Epoch 29/50
13/13             0s 7ms/step - loss:
10.4339 - mean_absolute_error: 2.2949 - val_loss: 25.1865 -
val_mean_absolute_error: 3.3169
Epoch 30/50
13/13             0s 7ms/step - loss:
13.5150 - mean_absolute_error: 2.5116 - val_loss: 25.5886 -
val_mean_absolute_error: 3.3242
Epoch 31/50
13/13             0s 7ms/step - loss:
9.7424 - mean_absolute_error: 2.4091 - val_loss: 25.1225 -
val_mean_absolute_error: 3.2867
Epoch 32/50
13/13             0s 7ms/step - loss:
10.3168 - mean_absolute_error: 2.3536 - val_loss: 25.8026 -
val_mean_absolute_error: 3.3204
Epoch 33/50
13/13             0s 7ms/step - loss:
10.3549 - mean_absolute_error: 2.3842 - val_loss: 26.0048 -
val_mean_absolute_error: 3.3170
Epoch 34/50
13/13             0s 7ms/step - loss:
8.8075 - mean_absolute_error: 2.2522 - val_loss: 24.6992 -
val_mean_absolute_error: 3.2154
Epoch 35/50
13/13             0s 7ms/step - loss:
8.2160 - mean_absolute_error: 2.1921 - val_loss: 25.5725 -
val_mean_absolute_error: 3.2624
Epoch 36/50
13/13             0s 7ms/step - loss:
8.2835 - mean_absolute_error: 2.1827 - val_loss: 25.5163 -
val_mean_absolute_error: 3.2418
Epoch 37/50
13/13             0s 7ms/step - loss:
10.0586 - mean_absolute_error: 2.3746 - val_loss: 26.5473 -
val_mean_absolute_error: 3.2867
Epoch 38/50
13/13             0s 7ms/step - loss:
9.5378 - mean_absolute_error: 2.2890 - val_loss: 24.6196 -
val_mean_absolute_error: 3.1490
```

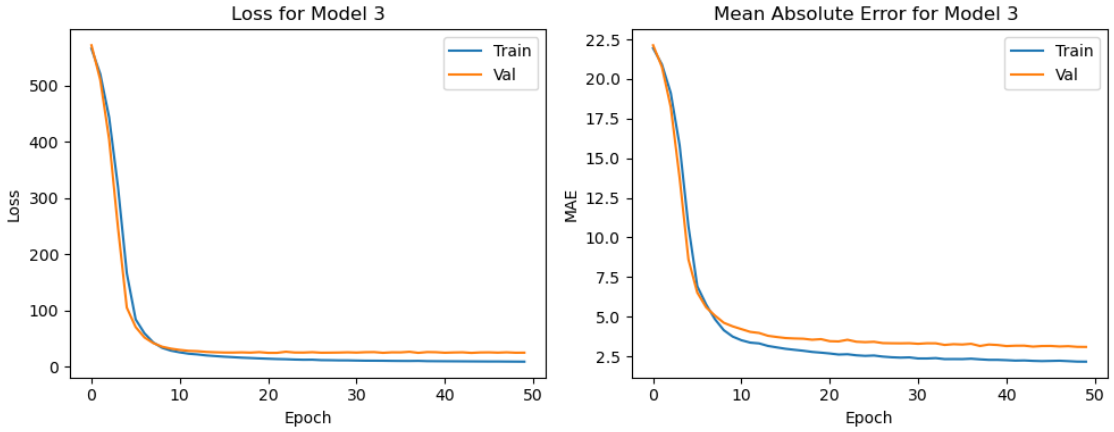
```
Epoch 39/50
13/13            0s 7ms/step - loss:
9.5785 - mean_absolute_error: 2.3022 - val_loss: 26.0603 -
val_mean_absolute_error: 3.2415
Epoch 40/50
13/13            0s 7ms/step - loss:
9.6618 - mean_absolute_error: 2.2449 - val_loss: 25.8001 -
val_mean_absolute_error: 3.2109
Epoch 41/50
13/13            0s 7ms/step - loss:
10.1419 - mean_absolute_error: 2.2317 - val_loss: 24.8331 -
val_mean_absolute_error: 3.1440
Epoch 42/50
13/13            0s 7ms/step - loss:
8.2919 - mean_absolute_error: 2.0765 - val_loss: 25.3194 -
val_mean_absolute_error: 3.1671
Epoch 43/50
13/13            0s 7ms/step - loss:
8.0678 - mean_absolute_error: 2.1168 - val_loss: 25.6029 -
val_mean_absolute_error: 3.1700
Epoch 44/50
13/13            0s 7ms/step - loss:
9.2098 - mean_absolute_error: 2.1834 - val_loss: 24.5555 -
val_mean_absolute_error: 3.1108
Epoch 45/50
13/13            0s 7ms/step - loss:
8.7439 - mean_absolute_error: 2.2366 - val_loss: 25.3514 -
val_mean_absolute_error: 3.1469
Epoch 46/50
13/13            0s 8ms/step - loss:
9.2554 - mean_absolute_error: 2.2469 - val_loss: 25.5336 -
val_mean_absolute_error: 3.1520
Epoch 47/50
13/13            0s 7ms/step - loss:
10.4546 - mean_absolute_error: 2.2804 - val_loss: 25.0111 -
val_mean_absolute_error: 3.1183
Epoch 48/50
13/13            0s 7ms/step - loss:
7.6064 - mean_absolute_error: 2.0647 - val_loss: 25.5887 -
val_mean_absolute_error: 3.1375
Epoch 49/50
13/13            0s 7ms/step - loss:
9.2823 - mean_absolute_error: 2.1685 - val_loss: 24.8675 -
val_mean_absolute_error: 3.0951
Epoch 50/50
13/13            0s 7ms/step - loss:
9.6315 - mean_absolute_error: 2.1830 - val_loss: 24.9395 -
val_mean_absolute_error: 3.0901
```

[31]:
```
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.plot(history3.history['loss'])
plt.plot(history3.history['val_loss'])
plt.title('Loss for Model 3')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Val'])

plt.subplot(1, 2, 2)
plt.plot(history3.history['mean_absolute_error'])
plt.plot(history3.history['val_mean_absolute_error'])
plt.title('Mean Absolute Error for Model 3 ')
plt.xlabel('Epoch')
plt.ylabel('MAE')
plt.legend(['Train', 'Val'])

plt.tight_layout()
plt.show()
```



We can observe that the model's performance plateaus after the 30th epoch, and further fitting will be pointless.

g) Saving the neural network :

[33]:
```
model3.save('model3.h5')
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or
`keras.saving.save_model(model)`. This file format is considered legacy. We
recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')` or `keras.saving.save_model(model,
```

'my_model.keras')`.

Thank You !

If you found this article helpful, please do leave a like and comment any feedback you feel I could use.

While I’m working on the next project, I’ll be over at :

www.linkedin.com/in/shivang-kainthola-2835151b9/

shivang.kainthola64@gmail.com

https://shivangkainthola28.medium.com/

https://github.com/HeadHunter28