Day 6 - Building Complex Models Using the Functional API - Part 1

Tags	Complex Models, Functional API, Wide and Deep Neural Network Architecture

Building Complex Models Using the Functional API - Part 1

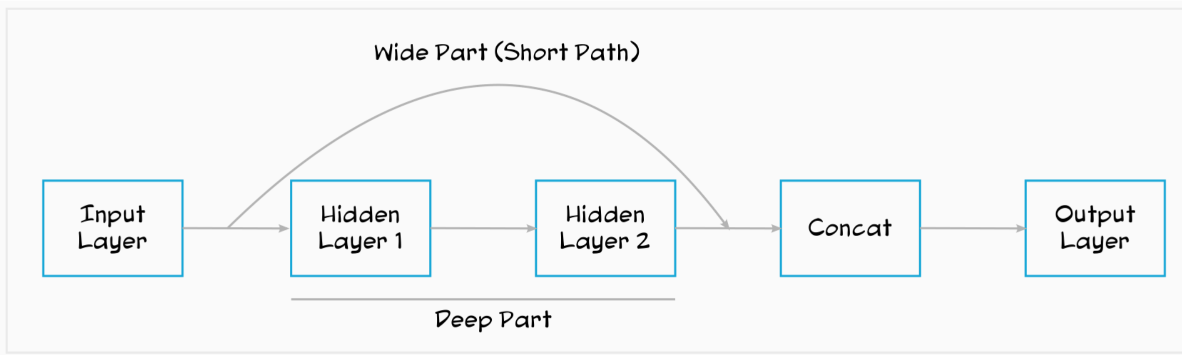
Author: **Chandan Kumar**

`enchandan.com`

Wide & Deep Neural Network Architecture

- So far we have used Sequential network to build our model
- For more complex models, we can create non-sequential network like Wide & Deep Neural Network

- In this architecture, few or all of the inputs are directly connected to the Output layer



- Heng-Tze came up with this architecture and claims that it can learn both deep and simple patterns in the dataset efficiently.
- How?
  Deep path learns the deep patterns and short path learns the simple rules in the given dataset.
- Another advantage is that, simple patterns or rules get distorted when they are passed through deep network. It's a good idea to separate these features out and let them escape the deep route.

- Let's go back to California Housing dataset and process the dataset using a wide & deep neural network architecture created in above picture

Implementation

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation='relu')(input_)
hidden2 = keras.layers.Dense(30, activation='relu')(hidden1)
concat = keras.layers.Concatenate()([input_, hidden2])
output = keras.layers.Dense(1)(concat)

model = keras.Model(inputs=[input_], outputs=[output])
```

- Create a Input layer and specify the input_shape
- Create a Dense layer (part of hidden layers) with 30 neurons, using the ReLU activation
  - We call this dense layer like a function, that's why it's called a functional api approach

- By passing the Input layer, we are telling `keras` how to connect the layers
- Then we create the second Hidden layer, and pass the output of the first hidden layer
- Next, we create a `Concatenate` layer and we use it like a function to concatenate Input layer directly and output of the second Hidden layer

```
model.summary()

>>>
Model: "functional_1"
__________________________________________________________________________________________________
_
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
=
input_1 (InputLayer)            [(None, 8)]          0
__________________________________________________________________________________________________
_
dense (Dense)                   (None, 30)           270         input_1[0][0]
__________________________________________________________________________________________________
_
dense_1 (Dense)                 (None, 30)           930         dense[0][0]
__________________________________________________________________________________________________
_
concatenate (Concatenate)       (None, 38)           0           input_1[0][0]
                                                                 dense_1[0][0]
__________________________________________________________________________________________________
_
dense_2 (Dense)                 (None, 1)            39          concatenate[0][0]
==================================================================================================
=
Total params: 1,239
Trainable params: 1,239
Non-trainable params: 0
```

Train the Model with different variations

- From notes of Day 4 - Performance of a simple model with Sequential API

```
Epoch 30/30
363/363 [==============================] - 0s 1ms/step - loss: 0.3510 - val_loss: 0.4280

MSE loss on Test data
>>>
0.34973791241645813
```

1. With `sgd` and `mean_squared_error`

- Compile the Model

```
model.compile(optimizer='sgd', loss='mean_squared_error')
```

- Train the Model

```
history = model.fit(X_train, y_train, epochs=30, validation_data=(X_valid, y_valid))


>>>
Epoch 1/30
363/363 [==============================] - 1s 1ms/step - loss: 0.8421 - val_loss: 12.7057
Epoch 2/30
363/363 [==============================] - 0s 1ms/step - loss: 0.9727 - val_loss: 2.2081
Epoch 3/30
363/363 [==============================] - 0s 1ms/step - loss: 0.4301 - val_loss: 13.6059
.
.
Epoch 17/30
363/363 [==============================] - 0s 1ms/step - loss: 6.0010 - val_loss: 1042.3330
Epoch 18/30
363/363 [==============================] - 0s 1ms/step - loss: 6405.5396 - val_loss: 295.5580
.
.
Epoch 30/30
363/363 [==============================] - 0s 1ms/step - loss: 0.6182 - val_loss: 1.0933
```

- A lot of fluctuations in loss function with `optimizer='sgd'` and `loss='mean_squared_error'`
- Probably, because of high learning rate (0.01 by default in sgd)

```
https://keras.io/api/optimizers/sgd/

tf.keras.optimizers.SGD(
    learning_rate=0.01, momentum=0.0, nesterov=False, name="SGD", **kwargs
)
```

- From https://stackoverflow.com/questions/37232782/nan-loss-when-training-regression-network

```
'''
Historically, one key solution to exploding gradients was to reduce the learning rate, but with the advent of per-parameter adaptive learning rate algorithms like Adam, you no longer need to set a learning rate to get good performance. There is very little reason to use SGD with momentum any more
'''
```

2. With `adam` and `mean_squared_error`

- `Adam` has default learning rate as 0.001 slightly smaller than `sgd`

```
tf.keras.optimizers.Adam(
    learning_rate=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    name="Adam",
    **kwargs
)
```

- Compile and train the model

```
model.compile(optimizer='adam', loss='mean_squared_error',)
history = model.fit(X_train, y_train, epochs=30, validation_data=(X_valid, y_valid))

>>>
Epoch 1/30
363/363 [==============================] - 1s 2ms/step - loss: 1.2900 - val_loss: 0.7870
Epoch 2/30
363/363 [==============================] - 1s 1ms/step - loss: 0.4763 - val_loss: 0.4150
.
.
.
Epoch 29/30
363/363 [==============================] - 0s 1ms/step - loss: 0.2884 - val_loss: 0.3280
Epoch 30/30
363/363 [==============================] - 0s 1ms/step - loss: 0.2824 - val_loss: 0.8150
```

- Evaluate the Model

```
mse_test = model.evaluate(X_test, y_test)
mse_test

>>>
0.30665069818496704
```

3. With `adam` and `huber_loss`

- Compile and train the model

```
model.compile(optimizer='adam', loss='huber_loss')
history = model.fit(X_train, y_train, epochs=30, validation_data=(X_valid, y_valid))

>>>
Epoch 1/30
363/363 [==============================] - 1s 2ms/step - loss: 0.3898 - val_loss: 0.1890
Epoch 2/30
363/363 [==============================] - 0s 1ms/step - loss: 0.1755 - val_loss: 0.1636
.
.
.
Epoch 30/30
363/363 [==============================] - 0s 1ms/step - loss: 0.1238 - val_loss: 0.1280
```

- Evaluate the Model

```
hloss_test = model.evaluate(X_test, y_test)
hloss_test

>>>
0.12629371881484985
```

- Pretty impressive!
- `huber_loss` works well especially if your dataset is skewed

Summary

- Using wide and deep neural network architecture, we make our model to learn the complex patterns as well as simple rules
- optimizer `adam` performs well, we'll learn about this algorithm in detail in coming sections
- `huber_loss` improves the performance especially if our dataset is skewed (which happens all the time in real world data)

References

https://keras.io/guides/functional_api/

https://keras.io/api/optimizers/adam/