CSE572-Lab8-key

October 17, 2022

1 CSE 572: Lab 8

In this lab, you will practice implementing feed-forward and convolutional neural networks.

To execute and make changes to this notebook, click File > Save a copy to save your own version in your Google Drive or Github. Read the step-by-step instructions below carefully. To execute the code, click on each cell below and press the SHIFT-ENTER keys simultaneously or by clicking the Play button.

When you finish executing all code/exercises, save your notebook then download a copy (.ipynb file). Submit the following **three** things: 1. a link to your Colab notebook, 2. the .ipynb file, and 3. a pdf of the executed notebook on Canvas.

To generate a pdf of the notebook, click File > Print > Save as PDF.

[1]:
```python
# Import libraries
import tensorflow as tf
import numpy as np
import pandas as pd
from tensorflow import keras
from tensorflow.keras import layers

# Set the random seed for reproducibility
seed = 0
tf.random.set_seed(seed)
```

1.1 Feed-forward neural networks

Describe

1.1.1 Load the dataset

For this example, we will use the Cleveland Heart Disease dataset. Review the dataset documentation to learn more about the attributes and other aspects of the dataset. The dataset consists of a CSV file with 303 rows. Each row contains information about a patient. There are 14 attribute columns and one binary class column (`target`) that reports whether or not a patient had a heart disease. We will train a feed-forward neural network model to predict whether or not a given patient has a heart disease based on the attribute values.

[2]:
```
# Load the dataset
data = pd.read_csv("http://storage.googleapis.com/download.tensorflow.org/data/
 ↪heart.csv")
```

[3]:
```
# Print sample rows
data.head()
```

[3]:
```
   age  sex  cp  trestbps  chol  fbs  restecg  thalach  exang  oldpeak  slope  \
0   63    1   1       145   233    1        2      150      0      2.3      3
1   67    1   4       160   286    0        2      108      1      1.5      2
2   67    1   4       120   229    0        2      129      1      2.6      2
3   37    1   3       130   250    0        0      187      0      3.5      3
4   41    0   2       130   204    0        2      172      0      1.4      1

   ca        thal  target
0   0       fixed       0
1   3      normal       1
2   2  reversible       0
3   0      normal       0
4   0      normal       0
```

[4]:
```
# Print the number of rows and columns
data.shape
```

[4]:
```
(303, 14)
```

Split the dataset into three randomly-sampled subsets: training (60%), validation (20%), and test (20%).

[5]:
```
# YOUR CODE HERE

train, val, test = np.split(data.sample(frac=1, random_state=seed), [int(.
 ↪6*len(data)), int(.8*len(data))])
```

Print the number of samples in each of the three subsets and the number of instances from each class. For example, for the training set you might print "The training set has ___ instances (___ heart disease, ___ no disease)".

[6]:
```
# YOUR CODE HERE

print('Train set has {} instances ({} heart disease, {} no disease)'.
 ↪format(train.shape[0],                                                    ␣
 ↪train[train['target'] == 1].shape[0],                                     ␣
 ↪train[train['target'] == 0].shape[0]))
```

```
print('Validation set has {} instances ({} heart disease, {} no disease)'.
 ↪format(val.shape[0],
                                                                     ␣
 ↪val[val['target'] == 1].shape[0],
                                                                     ␣
 ↪val[val['target'] == 0].shape[0]))

print('Test set has {} instances ({} heart disease, {} no disease)'.format(test.
 ↪shape[0],
                                                                ␣
 ↪test[test['target'] == 1].shape[0],
                                                                ␣
 ↪test[test['target'] == 0].shape[0]))
```

```
Train set has 181 instances (48 heart disease, 133 no disease)
Validation set has 61 instances (18 heart disease, 43 no disease)
Test set has 61 instances (17 heart disease, 44 no disease)
```

1.1.2 Prepare the dataset

Before we can feed this dataset to our model for training and evaluation, we need to perform a few steps to get it ready: 1. Convert the dataframes to Dataset objects 2. Normalize the numerical feature values 3. Binarize the Categorical features by converting to one-hot encodings

[7]:
```
# Convert dataframes to Dataset objects
def dataframe_to_dataset(df):
    df = df.copy()
    # Remove the target column and store in a separate array
    labels = df.pop('target')
    ds = tf.data.Dataset.from_tensor_slices((dict(df), labels))
    ds = ds.shuffle(buffer_size=len(df), reshuffle_each_iteration=False)
    return ds


train_ds = dataframe_to_dataset(train)
val_ds = dataframe_to_dataset(val)
test_ds = dataframe_to_dataset(test)
```

```
2022-09-27 16:30:48.746413: I tensorflow/core/platform/cpu_feature_guard.cc:193]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
```

The Dataset object yields a tuple containing the input feature vector and target (class value): `(input, target)`. `input` is a dictionary of features and `target` is the value 0 or 1. The code below prints an example instance drawn from the training Dataset object.

[8]:
```
for x, y in train_ds.take(1):
    print("Input:", x)
    print("\nTarget:", y)
```

```
Input: {'age': <tf.Tensor: shape=(), dtype=int64, numpy=63>, 'sex': <tf.Tensor:
shape=(), dtype=int64, numpy=0>, 'cp': <tf.Tensor: shape=(), dtype=int64,
numpy=3>, 'trestbps': <tf.Tensor: shape=(), dtype=int64, numpy=135>, 'chol':
<tf.Tensor: shape=(), dtype=int64, numpy=252>, 'fbs': <tf.Tensor: shape=(),
dtype=int64, numpy=0>, 'restecg': <tf.Tensor: shape=(), dtype=int64, numpy=2>,
'thalach': <tf.Tensor: shape=(), dtype=int64, numpy=172>, 'exang': <tf.Tensor:
shape=(), dtype=int64, numpy=0>, 'oldpeak': <tf.Tensor: shape=(), dtype=float64,
numpy=0.0>, 'slope': <tf.Tensor: shape=(), dtype=int64, numpy=1>, 'ca':
<tf.Tensor: shape=(), dtype=int64, numpy=0>, 'thal': <tf.Tensor: shape=(),
dtype=string, numpy=b'normal'>}

Target: tf.Tensor(0, shape=(), dtype=int64)
```

We can use the batch() function in keras to create batches from the full dataset for passing to the model. For the training dataset, we'll define a hyperparameter `batch_size` that we will set. For the validation and test sets, we will make the batch size equivalent to the size of the subset so all samples in that subset are evaluated each time the dataset is evaluated by the model.

[9]:
```
batch_size = 32
train_ds = train_ds.batch(batch_size)
val_ds = val_ds.batch(val.shape[0])
test_ds = test_ds.batch(test.shape[0])
```

There are seven categorical features in the dataset: `sex`, `cp`, `fbs`, `restecg`, `exang`, `ca`, and `thal`. You can read more about what these features mean in the dataset documentation. All of them except `thal` have integer data type while `thal` has String data type. Below we define a function to encode these feature values as one-hot encodings using the IntegerLookup() and StringLookup() layers. These layers create look-up tables for mapping a set of arbitrary integers or strings to a one-hot encoding. We use an `is_string` argument to indicate whether we should use the `StringLookup()` for `thal` or the `IntegerLookup()` for the remaining features.

[10]:
```
def encode_categorical_feature(feature, name, dataset, is_string):
    from tensorflow.keras.layers import IntegerLookup
    from tensorflow.keras.layers import StringLookup

    # Create lookup layer to turn categorical features into 1-hot integer␣
 ↪encodings
    if is_string:
        lookup = StringLookup(output_mode="binary")
    else:
        lookup = IntegerLookup(output_mode="binary")

    # Prepare a Dataset that only yields the feature of interest
    feature_ds = dataset.map(lambda x, y: x[name])
    feature_ds = feature_ds.map(lambda x: tf.expand_dims(x, -1))

    # Find the set of possible values and assign them a fixed integer index
    lookup.adapt(feature_ds)

    # Turn the input into integer indices
    encoded_feature = lookup(feature)
    return encoded_feature
```

The remaining features in the dataset (`age`, `trestbps`, `chol`, `thalach`, `oldpeak`, and `slope`) are all numerical measurements. You can read more about what these features mean in the dataset documentation. We don't need to encode the numerical features, but we do want to scale them to the same range of values (e.g., using standardization or normalization). Below we define a function that uses the Normalization() layer to standardize the data (subtract the mean and divide by the standard deviation for each feature).

[11]:
```
def normalize(feature, name, dataset):
    from tensorflow.keras.layers import Normalization

    # Create a Normalization layer for our feature
    normalizer = Normalization()

    # Prepare a Dataset that only yields the feature of interest
    feature_ds = dataset.map(lambda x, y: x[name])
    feature_ds = feature_ds.map(lambda x: tf.expand_dims(x, -1))

    # Learn the statistics of the data
    normalizer.adapt(feature_ds)

    # Normalize the input feature
    norm_feature = normalizer(feature)
    return norm_feature
```

Now we can apply these functions to each of our features and return an encoded/preprocessed input layer. We first create tensor variables for each of the inputs, then apply the appropriate function, then concatenate all of these input layers for each feature together to form a single input feature layer.

[12]:
```
all_inputs = [
    keras.Input(shape=(1,), name="sex", dtype="int64"),
    keras.Input(shape=(1,), name="cp", dtype="int64"),
    keras.Input(shape=(1,), name="fbs", dtype="int64"),
    keras.Input(shape=(1,), name="restecg", dtype="int64"),
    keras.Input(shape=(1,), name="exang", dtype="int64"),
    keras.Input(shape=(1,), name="ca", dtype="int64"),
    keras.Input(shape=(1,), name="thal", dtype="string"),
    keras.Input(shape=(1,), name="age"),
    keras.Input(shape=(1,), name="trestbps"),
    keras.Input(shape=(1,), name="chol"),
    keras.Input(shape=(1,), name="thalach"),
    keras.Input(shape=(1,), name="oldpeak"),
    keras.Input(shape=(1,), name="slope"),
]
```

[13]:
```
feature_layer = layers.concatenate(
    [
        encode_categorical_feature(all_inputs[0], "sex", train_ds, False),
        encode_categorical_feature(all_inputs[1], "cp", train_ds, False),
        encode_categorical_feature(all_inputs[2], "fbs", train_ds, False),
        encode_categorical_feature(all_inputs[3], "restecg", train_ds, False),
        encode_categorical_feature(all_inputs[4], "exang", train_ds, False),
        encode_categorical_feature(all_inputs[5], "ca", train_ds, False),
        encode_categorical_feature(all_inputs[6], "thal", train_ds, True),
        normalize(all_inputs[7], "age", train_ds),
        normalize(all_inputs[8], "trestbps", train_ds),
        normalize(all_inputs[9], "chol", train_ds),
        normalize(all_inputs[10], "thalach", train_ds),
        normalize(all_inputs[11], "oldpeak", train_ds),
        normalize(all_inputs[12], "slope", train_ds)
    ]
)
```

1.1.3 Build the model

Now that we've prepared our dataset, we can construct our neural network model. We construct the model by composing Layer objects starting with the input layer (which we've already defined as `feature_layer`) and ending with the output layer (which will be the final output of the model). In this example, we will only use Dense() layers which are simple fully-connected feed-forward layers (i.e., each output from layer `i-1` is connected by a weight variable to every neuron in layer `i`). We will create a network with only one hidden layer (between the input and output layers).

The Dense() layer object allows us to specify the activation function to use using the `activation` argument. In class, we talked about several activation functions including sigmoid, sign, and tanh. Another commonly used activation function is the rectified linear unit, or "ReLU" function, which has the equation $a(z) = max(0, z)$. We will use `relu` as our activation function in this example for all layers except the final layer, which will use a `sigmoid` activation.

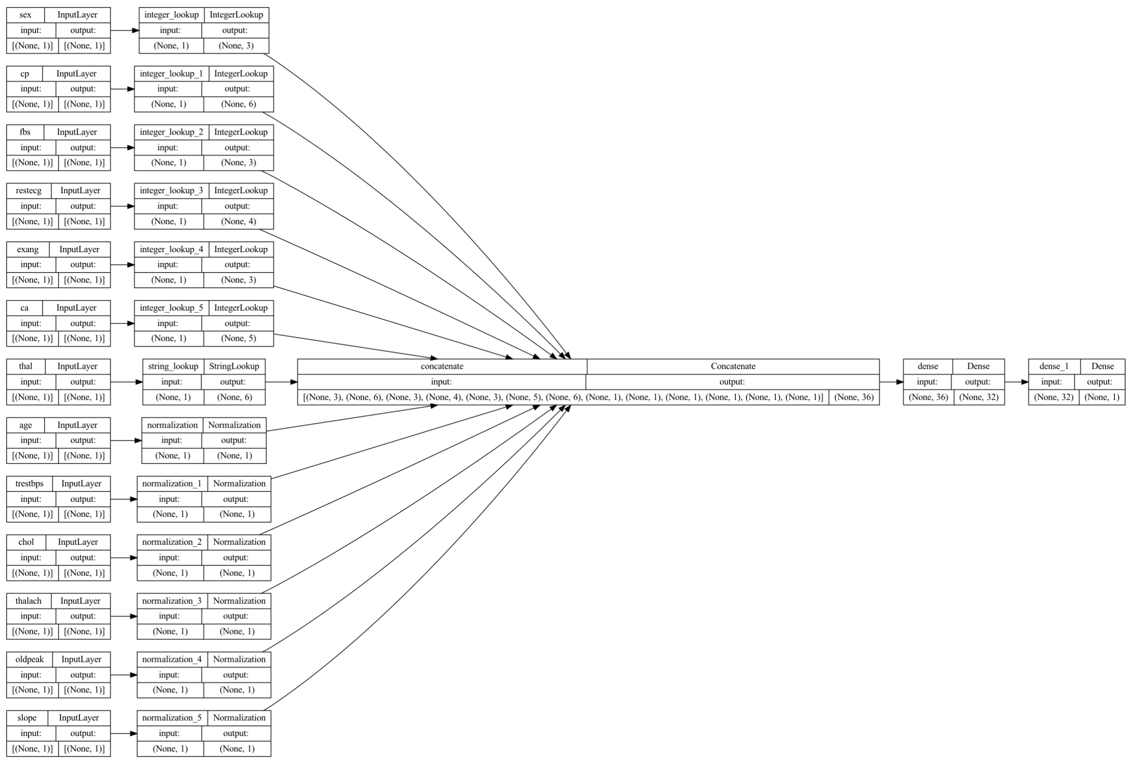
[14]:
```
# Create a variable for the number of units/neurons in the layer
h1_units = 32
# Create a Dense layer and append it to the input layer
h1_layer = layers.Dense(h1_units, activation='relu')(feature_layer)

# Create an output layer with one output representing the likelihood of
# heart disease and append it to the hidden layer
output_layer = layers.Dense(1, activation="sigmoid")(h1_layer)
```

```
# Build the model specifying the input and output layer
model = keras.Model(inputs=all_inputs, outputs=output_layer)
```

We can plot our completed model to visualize the input, hidden, and output layers.

[15]:
```
# `rankdir='LR'` is to make the graph horizontal
keras.utils.plot_model(model, show_shapes=True, rankdir="LR")
```

[15]:


sex | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
cp | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
fbs | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
restecg | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
exang | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
ca | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
thal | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
age | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
trestbps | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
chol | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
thalach | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
oldpeak | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
slope | InputLayer | input: | output: | [(None, 1)] | [(None, 1)]
integer_lookup | IntegerLookup | input: | output: | (None, 1) | (None, 3)
integer_lookup_1 | IntegerLookup | input: | output: | (None, 1) | (None, 6)
integer_lookup_2 | IntegerLookup | input: | output: | (None, 1) | (None, 3)
integer_lookup_3 | IntegerLookup | input: | output: | (None, 1) | (None, 4)
integer_lookup_4 | IntegerLookup | input: | output: | (None, 1) | (None, 3)
integer_lookup_5 | IntegerLookup | input: | output: | (None, 1) | (None, 5)
string_lookup | StringLookup | input: | output: | (None, 1) | (None, 6)
normalization | Normalization | input: | output: | (None, 1) | (None, 1)
normalization_1 | Normalization | input: | output: | (None, 1) | (None, 1)
normalization_2 | Normalization | input: | output: | (None, 1) | (None, 1)
normalization_3 | Normalization | input: | output: | (None, 1) | (None, 1)
normalization_4 | Normalization | input: | output: | (None, 1) | (None, 1)
normalization_5 | Normalization | input: | output: | (None, 1) | (None, 1)
concatenate | Concatenate | input: | output: | [(None, 3), (None, 6), (None, 3), (None, 4), (None, 3), (None, 5), (None, 6), (None, 1), (None, 1), (None, 1), (None, 1), (None, 1), (None, 1)] | (None, 36)
dense | Dense | input: | output: | (None, 36) | (None, 32)
dense_1 | Dense | input: | output: | (None, 32) | (None, 1)

Next we compile the model by specifying the optimization technique and loss function to be used in model training. We can also specify the metric(s) that will be logged during training. We will use stochastic gradient descent (`sgd`) for the optimizer and binary cross entropy (log loss) as the loss function. We will log the accuracy metric during training. We can also specify the learning rate hyperparameter here.

1.1.4 Train the model

Now that we've constructed and compiled our model, we can train the model using our training dataset. This is done in keras using the `fit()` function, which also gives us an option to provide the validation dataset which will be used to evaluate validation accuracy after every epoch.

[16]:
```
lr = 0.01
model.compile(keras.optimizers.SGD(learning_rate=lr), "binary_crossentropy",␣
  ↪metrics=["accuracy"])
```

```
[17]: model_result = model.fit(train_ds, epochs=100, validation_data=val_ds)
```

```
Epoch 1/100
6/6 [==============================] - 1s 46ms/step - loss: 0.5750 - accuracy:
0.7624 - val_loss: 0.5773 - val_accuracy: 0.7705
Epoch 2/100
6/6 [==============================] - 0s 3ms/step - loss: 0.5595 - accuracy:
0.7680 - val_loss: 0.5649 - val_accuracy: 0.7705
Epoch 3/100
6/6 [==============================] - 0s 3ms/step - loss: 0.5455 - accuracy:
0.7845 - val_loss: 0.5536 - val_accuracy: 0.7705
Epoch 4/100
6/6 [==============================] - 0s 3ms/step - loss: 0.5329 - accuracy:
0.7845 - val_loss: 0.5432 - val_accuracy: 0.7705
Epoch 5/100
6/6 [==============================] - 0s 3ms/step - loss: 0.5214 - accuracy:
0.7845 - val_loss: 0.5336 - val_accuracy: 0.7705
Epoch 6/100
6/6 [==============================] - 0s 3ms/step - loss: 0.5109 - accuracy:
0.7845 - val_loss: 0.5246 - val_accuracy: 0.7705
Epoch 7/100
6/6 [==============================] - 0s 3ms/step - loss: 0.5012 - accuracy:
0.7845 - val_loss: 0.5162 - val_accuracy: 0.7705
Epoch 8/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4923 - accuracy:
0.7845 - val_loss: 0.5083 - val_accuracy: 0.7869
Epoch 9/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4840 - accuracy:
0.7845 - val_loss: 0.5010 - val_accuracy: 0.7705
Epoch 10/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4763 - accuracy:
0.7956 - val_loss: 0.4942 - val_accuracy: 0.7705
Epoch 11/100
6/6 [==============================] - 0s 2ms/step - loss: 0.4691 - accuracy:
0.7956 - val_loss: 0.4878 - val_accuracy: 0.7705
Epoch 12/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4625 - accuracy:
0.8011 - val_loss: 0.4817 - val_accuracy: 0.7541
Epoch 13/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4562 - accuracy:
0.8011 - val_loss: 0.4760 - val_accuracy: 0.7705
Epoch 14/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4503 - accuracy:
0.8011 - val_loss: 0.4707 - val_accuracy: 0.7705
Epoch 15/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4447 - accuracy:
0.8122 - val_loss: 0.4655 - val_accuracy: 0.7705
```

```
Epoch 16/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4395 - accuracy:
0.8122 - val_loss: 0.4606 - val_accuracy: 0.7705
Epoch 17/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4345 - accuracy:
0.8122 - val_loss: 0.4560 - val_accuracy: 0.7705
Epoch 18/100
6/6 [==============================] - 0s 2ms/step - loss: 0.4297 - accuracy:
0.8122 - val_loss: 0.4515 - val_accuracy: 0.8033
Epoch 19/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4253 - accuracy:
0.8122 - val_loss: 0.4473 - val_accuracy: 0.8197
Epoch 20/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4210 - accuracy:
0.8066 - val_loss: 0.4432 - val_accuracy: 0.8197
Epoch 21/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4169 - accuracy:
0.8011 - val_loss: 0.4393 - val_accuracy: 0.8197
Epoch 22/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4129 - accuracy:
0.8011 - val_loss: 0.4356 - val_accuracy: 0.8197
Epoch 23/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4092 - accuracy:
0.8011 - val_loss: 0.4321 - val_accuracy: 0.8197
Epoch 24/100
6/6 [==============================] - 0s 2ms/step - loss: 0.4056 - accuracy:
0.8011 - val_loss: 0.4286 - val_accuracy: 0.8197
Epoch 25/100
6/6 [==============================] - 0s 3ms/step - loss: 0.4021 - accuracy:
0.8011 - val_loss: 0.4253 - val_accuracy: 0.8197
Epoch 26/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3988 - accuracy:
0.8011 - val_loss: 0.4222 - val_accuracy: 0.8197
Epoch 27/100
6/6 [==============================] - 0s 2ms/step - loss: 0.3957 - accuracy:
0.8011 - val_loss: 0.4192 - val_accuracy: 0.8197
Epoch 28/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3926 - accuracy:
0.8122 - val_loss: 0.4163 - val_accuracy: 0.8361
Epoch 29/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3896 - accuracy:
0.8177 - val_loss: 0.4135 - val_accuracy: 0.8361
Epoch 30/100
6/6 [==============================] - 0s 2ms/step - loss: 0.3868 - accuracy:
0.8122 - val_loss: 0.4108 - val_accuracy: 0.8361
Epoch 31/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3840 - accuracy:
0.8122 - val_loss: 0.4082 - val_accuracy: 0.8361
```

```
Epoch 32/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3813 - accuracy:
0.8122 - val_loss: 0.4056 - val_accuracy: 0.8361
Epoch 33/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3788 - accuracy:
0.8177 - val_loss: 0.4032 - val_accuracy: 0.8361
Epoch 34/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3762 - accuracy:
0.8232 - val_loss: 0.4008 - val_accuracy: 0.8361
Epoch 35/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3738 - accuracy:
0.8287 - val_loss: 0.3985 - val_accuracy: 0.8361
Epoch 36/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3714 - accuracy:
0.8287 - val_loss: 0.3963 - val_accuracy: 0.8361
Epoch 37/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3691 - accuracy:
0.8287 - val_loss: 0.3942 - val_accuracy: 0.8361
Epoch 38/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3669 - accuracy:
0.8287 - val_loss: 0.3922 - val_accuracy: 0.8361
Epoch 39/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3647 - accuracy:
0.8232 - val_loss: 0.3902 - val_accuracy: 0.8361
Epoch 40/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3626 - accuracy:
0.8287 - val_loss: 0.3884 - val_accuracy: 0.8361
Epoch 41/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3605 - accuracy:
0.8287 - val_loss: 0.3865 - val_accuracy: 0.8361
Epoch 42/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3585 - accuracy:
0.8287 - val_loss: 0.3847 - val_accuracy: 0.8361
Epoch 43/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3566 - accuracy:
0.8287 - val_loss: 0.3830 - val_accuracy: 0.8361
Epoch 44/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3547 - accuracy:
0.8287 - val_loss: 0.3814 - val_accuracy: 0.8361
Epoch 45/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3529 - accuracy:
0.8287 - val_loss: 0.3798 - val_accuracy: 0.8361
Epoch 46/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3511 - accuracy:
0.8287 - val_loss: 0.3782 - val_accuracy: 0.8361
Epoch 47/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3494 - accuracy:
0.8287 - val_loss: 0.3767 - val_accuracy: 0.8361
```

```
Epoch 48/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3477 - accuracy:
0.8287 - val_loss: 0.3752 - val_accuracy: 0.8361
Epoch 49/100
6/6 [==============================] - 0s 2ms/step - loss: 0.3461 - accuracy:
0.8343 - val_loss: 0.3738 - val_accuracy: 0.8361
Epoch 50/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3445 - accuracy:
0.8398 - val_loss: 0.3724 - val_accuracy: 0.8361
Epoch 51/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3429 - accuracy:
0.8398 - val_loss: 0.3711 - val_accuracy: 0.8361
Epoch 52/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3414 - accuracy:
0.8398 - val_loss: 0.3698 - val_accuracy: 0.8361
Epoch 53/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3399 - accuracy:
0.8398 - val_loss: 0.3685 - val_accuracy: 0.8361
Epoch 54/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3384 - accuracy:
0.8398 - val_loss: 0.3673 - val_accuracy: 0.8361
Epoch 55/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3370 - accuracy:
0.8398 - val_loss: 0.3660 - val_accuracy: 0.8361
Epoch 56/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3356 - accuracy:
0.8398 - val_loss: 0.3649 - val_accuracy: 0.8525
Epoch 57/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3343 - accuracy:
0.8398 - val_loss: 0.3637 - val_accuracy: 0.8525
Epoch 58/100
6/6 [==============================] - 0s 2ms/step - loss: 0.3329 - accuracy:
0.8398 - val_loss: 0.3626 - val_accuracy: 0.8525
Epoch 59/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3316 - accuracy:
0.8398 - val_loss: 0.3615 - val_accuracy: 0.8525
Epoch 60/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3304 - accuracy:
0.8398 - val_loss: 0.3604 - val_accuracy: 0.8525
Epoch 61/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3291 - accuracy:
0.8398 - val_loss: 0.3594 - val_accuracy: 0.8525
Epoch 62/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3279 - accuracy:
0.8398 - val_loss: 0.3584 - val_accuracy: 0.8525
Epoch 63/100
6/6 [==============================] - 0s 2ms/step - loss: 0.3267 - accuracy:
0.8398 - val_loss: 0.3575 - val_accuracy: 0.8525
```

```
Epoch 64/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3255 - accuracy:
0.8398 - val_loss: 0.3566 - val_accuracy: 0.8525
Epoch 65/100
6/6 [==============================] - 0s 2ms/step - loss: 0.3244 - accuracy:
0.8398 - val_loss: 0.3557 - val_accuracy: 0.8525
Epoch 66/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3233 - accuracy:
0.8398 - val_loss: 0.3548 - val_accuracy: 0.8525
Epoch 67/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3222 - accuracy:
0.8398 - val_loss: 0.3539 - val_accuracy: 0.8525
Epoch 68/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3212 - accuracy:
0.8398 - val_loss: 0.3531 - val_accuracy: 0.8525
Epoch 69/100
6/6 [==============================] - 0s 2ms/step - loss: 0.3201 - accuracy:
0.8398 - val_loss: 0.3523 - val_accuracy: 0.8525
Epoch 70/100
6/6 [==============================] - 0s 2ms/step - loss: 0.3191 - accuracy:
0.8398 - val_loss: 0.3515 - val_accuracy: 0.8525
Epoch 71/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3181 - accuracy:
0.8398 - val_loss: 0.3507 - val_accuracy: 0.8525
Epoch 72/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3172 - accuracy:
0.8398 - val_loss: 0.3499 - val_accuracy: 0.8689
Epoch 73/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3162 - accuracy:
0.8398 - val_loss: 0.3492 - val_accuracy: 0.8689
Epoch 74/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3153 - accuracy:
0.8398 - val_loss: 0.3485 - val_accuracy: 0.8689
Epoch 75/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3143 - accuracy:
0.8398 - val_loss: 0.3478 - val_accuracy: 0.8689
Epoch 76/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3134 - accuracy:
0.8453 - val_loss: 0.3471 - val_accuracy: 0.8689
Epoch 77/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3125 - accuracy:
0.8453 - val_loss: 0.3464 - val_accuracy: 0.8689
Epoch 78/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3116 - accuracy:
0.8453 - val_loss: 0.3457 - val_accuracy: 0.8689
Epoch 79/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3108 - accuracy:
0.8453 - val_loss: 0.3451 - val_accuracy: 0.8689
```

```
Epoch 80/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3099 - accuracy:
0.8508 - val_loss: 0.3445 - val_accuracy: 0.8689
Epoch 81/100
6/6 [==============================] - 0s 2ms/step - loss: 0.3091 - accuracy:
0.8508 - val_loss: 0.3439 - val_accuracy: 0.8689
Epoch 82/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3082 - accuracy:
0.8564 - val_loss: 0.3433 - val_accuracy: 0.8689
Epoch 83/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3074 - accuracy:
0.8564 - val_loss: 0.3427 - val_accuracy: 0.8689
Epoch 84/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3066 - accuracy:
0.8564 - val_loss: 0.3421 - val_accuracy: 0.8689
Epoch 85/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3058 - accuracy:
0.8619 - val_loss: 0.3416 - val_accuracy: 0.8689
Epoch 86/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3050 - accuracy:
0.8619 - val_loss: 0.3410 - val_accuracy: 0.8689
Epoch 87/100
6/6 [==============================] - 0s 2ms/step - loss: 0.3042 - accuracy:
0.8619 - val_loss: 0.3405 - val_accuracy: 0.8689
Epoch 88/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3034 - accuracy:
0.8619 - val_loss: 0.3400 - val_accuracy: 0.8689
Epoch 89/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3027 - accuracy:
0.8619 - val_loss: 0.3395 - val_accuracy: 0.8689
Epoch 90/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3019 - accuracy:
0.8619 - val_loss: 0.3390 - val_accuracy: 0.8689
Epoch 91/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3012 - accuracy:
0.8619 - val_loss: 0.3385 - val_accuracy: 0.8689
Epoch 92/100
6/6 [==============================] - 0s 3ms/step - loss: 0.3004 - accuracy:
0.8619 - val_loss: 0.3381 - val_accuracy: 0.8689
Epoch 93/100
6/6 [==============================] - 0s 3ms/step - loss: 0.2997 - accuracy:
0.8674 - val_loss: 0.3376 - val_accuracy: 0.8689
Epoch 94/100
6/6 [==============================] - 0s 3ms/step - loss: 0.2990 - accuracy:
0.8674 - val_loss: 0.3372 - val_accuracy: 0.8689
Epoch 95/100
6/6 [==============================] - 0s 3ms/step - loss: 0.2983 - accuracy:
0.8674 - val_loss: 0.3367 - val_accuracy: 0.8689
```

```
Epoch 96/100
6/6 [==============================] - 0s 3ms/step - loss: 0.2976 - accuracy:
0.8674 - val_loss: 0.3363 - val_accuracy: 0.8689
Epoch 97/100
6/6 [==============================] - 0s 2ms/step - loss: 0.2969 - accuracy:
0.8674 - val_loss: 0.3359 - val_accuracy: 0.8689
Epoch 98/100
6/6 [==============================] - 0s 3ms/step - loss: 0.2962 - accuracy:
0.8674 - val_loss: 0.3355 - val_accuracy: 0.8689
Epoch 99/100
6/6 [==============================] - 0s 3ms/step - loss: 0.2955 - accuracy:
0.8674 - val_loss: 0.3351 - val_accuracy: 0.8689
Epoch 100/100
6/6 [==============================] - 0s 3ms/step - loss: 0.2949 - accuracy:
0.8729 - val_loss: 0.3347 - val_accuracy: 0.8689
```

Note: the following wo blocks are for debugging/demonstration purposes.

[18]:
```
# First epochs = 100
for x, y in train_ds.take(1):
    print("Input:", x)
    print("\nTarget:", y)
```

```
Input: {'age': <tf.Tensor: shape=(32,), dtype=int64, numpy=
array([63, 63, 63, 59, 58, 56, 57, 39, 64, 61, 35, 48, 45, 49, 43, 42, 44,
       44, 52, 57, 54, 58, 59, 67, 49, 57, 41, 46, 56, 40, 42, 56])>, 'sex':
<tf.Tensor: shape=(32,), dtype=int64, numpy=
array([0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0,
       1, 0, 1, 1, 0, 1, 1, 1, 1, 1])>, 'cp': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([3, 4, 4, 1, 3, 3, 4, 4, 3, 4, 2, 2, 2, 2, 3, 3, 4, 2, 2, 2, 3, 4,
       3, 3, 3, 4, 2, 4, 2, 4, 1, 2])>, 'trestbps': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([135, 140, 150, 160, 120, 130, 140, 118, 140, 148, 122, 110, 130,
       130, 122, 120, 112, 120, 134, 124, 160, 130, 150, 115, 120, 110,
       126, 140, 120, 110, 148, 130])>, 'chol': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([252, 187, 407, 273, 340, 256, 192, 219, 313, 203, 192, 229, 234,
       266, 213, 240, 290, 220, 201, 261, 201, 197, 212, 564, 188, 335,
       306, 311, 236, 167, 244, 221])>, 'fbs': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 0, 0, 0, 0, 0])>, 'restecg': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([2, 2, 2, 2, 0, 2, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0,
       0, 2, 0, 0, 0, 0, 0, 2, 2, 2])>, 'thalach': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([172, 144, 154, 125, 172, 142, 148, 140, 133, 161, 174, 168, 175,
       171, 165, 194, 153, 170, 158, 141, 163, 131, 157, 160, 139, 143,
       163, 120, 178, 114, 178, 163])>, 'exang': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 1, 0, 0])>, 'oldpeak': <tf.Tensor: shape=(32,),
dtype=float64, numpy=
array([0. , 4. , 4. , 0. , 0. , 0.6, 0.4, 1.2, 0.2, 0. , 0. , 1. , 0.6,
       0.6, 0.2, 0.8, 0. , 0. , 0.8, 0.3, 0. , 0.6, 1.6, 1.6, 2. , 3. ,
       0. , 1.8, 0.8, 2. , 0.8, 0. ])>, 'slope': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([1, 1, 2, 1, 1, 2, 2, 2, 1, 1, 1, 3, 2, 1, 2, 3, 1, 1, 1, 1, 1, 2,
       1, 2, 2, 2, 1, 2, 1, 2, 1, 1])>, 'ca': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([0, 2, 3, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0,
       0, 0, 3, 1, 0, 2, 0, 0, 2, 0])>, 'thal': <tf.Tensor: shape=(32,),
dtype=string, numpy=
array([b'normal', b'reversible', b'reversible', b'normal', b'normal',
       b'fixed', b'fixed', b'reversible', b'reversible', b'reversible',
       b'normal', b'reversible', b'normal', b'normal', b'normal',
       b'reversible', b'normal', b'normal', b'normal', b'reversible',
       b'normal', b'normal', b'normal', b'reversible', b'reversible',
       b'reversible', b'normal', b'reversible', b'normal', b'reversible',
       b'normal', b'reversible'], dtype=object)>}

Target: tf.Tensor([0 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 1 0 1 0
0], shape=(32,), dtype=int64)
```

[19]:
```python
# Second epochs = 100
for x, y in train_ds.take(1):
    print("Input:", x)
    print("\nTarget:", y)
```

```
Input: {'age': <tf.Tensor: shape=(32,), dtype=int64, numpy=
array([63, 63, 63, 59, 58, 56, 57, 39, 64, 61, 35, 48, 45, 49, 43, 42, 44,
       44, 52, 57, 54, 58, 59, 67, 49, 57, 41, 46, 56, 40, 42, 56])>, 'sex':
<tf.Tensor: shape=(32,), dtype=int64, numpy=
array([0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0,
       1, 0, 1, 1, 0, 1, 1, 1, 1, 1])>, 'cp': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([3, 4, 4, 1, 3, 3, 4, 4, 3, 4, 2, 2, 2, 2, 3, 3, 4, 2, 2, 2, 3, 4,
       3, 3, 3, 4, 2, 4, 2, 4, 1, 2])>, 'trestbps': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([135, 140, 150, 160, 120, 130, 140, 118, 140, 148, 122, 110, 130,
       130, 122, 120, 112, 120, 134, 124, 160, 130, 150, 115, 120, 110,
       126, 140, 120, 110, 148, 130])>, 'chol': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([252, 187, 407, 273, 340, 256, 192, 219, 313, 203, 192, 229, 234,
       266, 213, 240, 290, 220, 201, 261, 201, 197, 212, 564, 188, 335,
          306, 311, 236, 167, 244, 221])>, 'fbs': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 0, 0, 0, 0, 0])>, 'restecg': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([2, 2, 2, 2, 0, 2, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0,
       0, 2, 0, 0, 0, 0, 0, 2, 2, 2])>, 'thalach': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([172, 144, 154, 125, 172, 142, 148, 140, 133, 161, 174, 168, 175,
       171, 165, 194, 153, 170, 158, 141, 163, 131, 157, 160, 139, 143,
       163, 120, 178, 114, 178, 163])>, 'exang': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 1, 0, 0])>, 'oldpeak': <tf.Tensor: shape=(32,),
dtype=float64, numpy=
array([0. , 4. , 4. , 0. , 0. , 0.6, 0.4, 1.2, 0.2, 0. , 0. , 1. , 0.6,
       0.6, 0.2, 0.8, 0. , 0. , 0.8, 0.3, 0. , 0.6, 1.6, 1.6, 2. , 3. ,
       0. , 1.8, 0.8, 2. , 0.8, 0. ])>, 'slope': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([1, 1, 2, 1, 1, 2, 2, 2, 1, 1, 1, 3, 2, 1, 2, 3, 1, 1, 1, 1, 1, 2,
       1, 2, 2, 2, 1, 2, 1, 2, 1, 1])>, 'ca': <tf.Tensor: shape=(32,),
dtype=int64, numpy=
array([0, 2, 3, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0,
       0, 0, 3, 1, 0, 2, 0, 0, 2, 0])>, 'thal': <tf.Tensor: shape=(32,),
dtype=string, numpy=
array([b'normal', b'reversible', b'reversible', b'normal', b'normal',
       b'fixed', b'fixed', b'reversible', b'reversible', b'reversible',
       b'normal', b'reversible', b'normal', b'normal', b'normal',
       b'reversible', b'normal', b'normal', b'normal', b'reversible',
       b'normal', b'normal', b'normal', b'reversible', b'reversible',
       b'reversible', b'normal', b'reversible', b'normal', b'reversible',
       b'normal', b'reversible'], dtype=object)>}

Target: tf.Tensor([0 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 0 1 0
0], shape=(32,), dtype=int64)
```

The `fit()` function returns a history attribute that gives the metrics recorded during training as a dictionary. We can print the dictionary keys to see which metrics were stored:

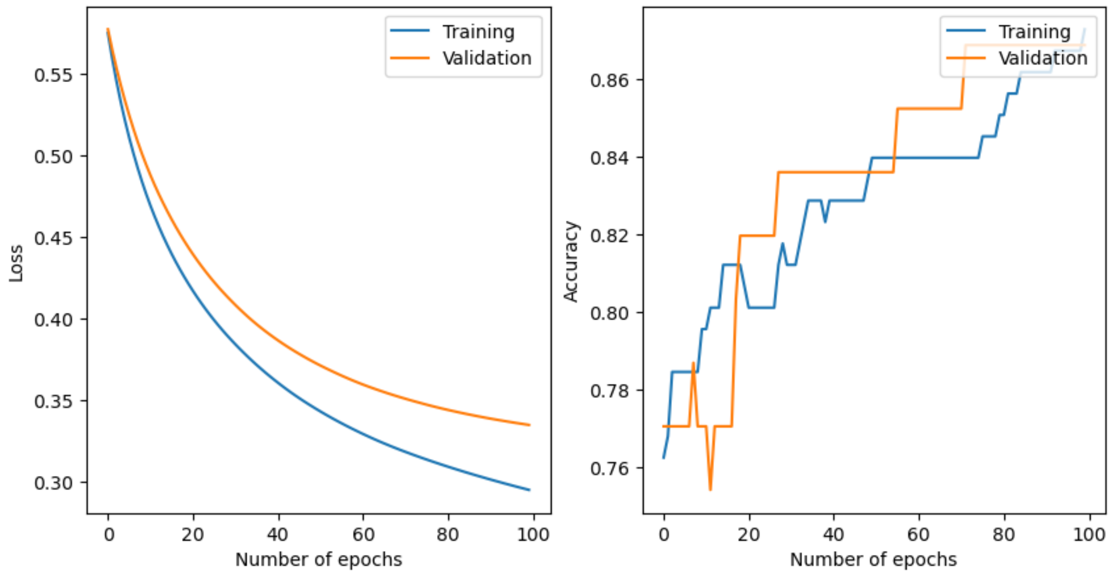
```
[20]: model_result.history.keys()
```

```
[20]: dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

Create a figure with two subplots. The first subplot should plot the training and validation loss (`loss` and `val_loss`) and the second subplot should plot the training and validation accuracy (`accuracy` and `val_accuracy`). Make sure you include the axis labels and a legend in each plot.

```
[21]: import matplotlib.pyplot as plt

      fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(10,5))

      # YOUR CODE HERE

      # Loss plot
      ax1.plot(model_result.history['loss'], label='Training')
      ax1.plot(model_result.history['val_loss'], label='Validation')
      ax1.set_xlabel('Number of epochs')
      ax1.set_ylabel('Loss')
      ax1.legend(loc='upper right')

      # Accuracy plot
      ax2.plot(model_result.history['accuracy'], label='Training')
      ax2.plot(model_result.history['val_accuracy'], label='Validation')
      ax2.set_xlabel('Number of epochs')
      ax2.set_ylabel('Accuracy')
      ax2.legend(loc='upper right')
```

```
[21]: <matplotlib.legend.Legend at 0x1d0066f70>
```



Question 1: If we train our model for too long (too many epochs), we run the risk of overfitting to the training data which will have a negative effect on our model's generalization to the validation dataset. Try changing the number of epochs to 1000 and re-training. Around how many epochs do you see the model begin to overfit?

Answer:

YOUR ANSWER HERE

The validation loss goes down until around 200 epochs and then starts to increase again, which indicates the model has begun overfitting. In the accuracy plot, we can see that the validation accuracy increases until around 200 epochs as well, after which it begins to plateau and then decrease even though the training accuracy continues to increase.

1.1.5 Test the model

Finally, we evaluate our trained model on the held-out test set. First we predict the outputs for the test data.

[22]:
```
preds = model.predict(test_ds)
```

```
1/1 [==============================] - 0s 166ms/step
```

The model output from the final sigmoid layer is a value between 0 and 1 representing the likelihood that a given sample patient has heart disease. To get the predicted classes, we predict 1 if the output was >= 0.5 and 0 otherwise.

[23]:
```
pred_classes = [1 if p >= 0.5 else 0 for p in preds]
```

Compute and print the test accuracy.

[24]:
```
from sklearn.metrics import accuracy_score

# YOUR CODE HERE
print('Accuracy on test set: %.5f' % (accuracy_score([y.numpy() for x, y in␣
  ↪test_ds][0], pred_classes)))
```

```
Accuracy on test set: 0.78689
```

Question 2: There is a large difference between the training and validation accuracies and the test accuracy. What do you think could explain this difference?

Answer:

YOUR ANSWER HERE (graders: any complete/non-empty answer can be accepted for this question)

The random split could have yielded a split in which the training and validation data are more similar than the test data, so even though the model does not appear to have overfit yet after 100 epochs, it could still generalize poorly to the test data because the test data contains patterns not yet seen in the training or validation data.

1.2 Convolutional neural networks

The previous example used a structured dataset of pre-determined features to predict the likelihood of heart disease in a patient. This type of dataset is well suited for a feed-forward neural network (FFNN) architecture. However, other datasets may not be as well suited for this type of architecture. For example, to train a FFNN to classify an image dataset, we would need to convert each image (which has dimension $N \times M \times K$ where N and M are the image dimensions and K is the number of color channels - 3 for RGB) to a feature vector representation. We can do this by flattening the image to a 1D feature vector so the new dimension would be $N * M * K \times 1$, but this would remove the 2D spatial structure.

Convolutional neural networks (CNNs) are another type of neural network architecture that enables spatial relationships to be learned from the data. The features learned in each hidden layer have 2D structures that act as filters on the spatial inputs, rather than scalar activations like in a FFNN. In this example, we'll train a CNN to classify an image dataset.

1.2.1 Load the dataset

For this example we'll use the FashionMNIST dataset. This is a dataset of 60,000 28x28 color training images and 10,000 test images with 10 class labels:

0. t-shirt
1. trouser
2. pullover
3. dress
4. coat
5. sandal
6. shirt
7. sneaker
8. bag
9. boot

[25]:
```
(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
```

[26]:
```
# Print the dimensions of the datasets

print('Training data input dim: ', x_train.shape)
print('Training data label dim: ', y_train.shape)
print('Test data input dim: ', x_test.shape)
print('Test data label dim: ', y_test.shape)
```

```
Training data input dim:  (60000, 28, 28)
Training data label dim:  (60000,)
Test data input dim:  (10000, 28, 28)
Test data label dim:  (10000,)
```

[27]:
```
# Set variable for number of classes
num_classes = 10
```

Next we'll plot a few random images from the training set to get an idea of what the data look like. You can run this as many times as you want to see a different random subset of images.

[28]:
```
# number of images to plot
n = 5

# create a dictionary for the label names
```

```
labels = {
    0: 't-shirt',
    1: 'trouser',
    2: 'pullover',
    3: 'dress',
    4: 'coat',
    5: 'sandal',
    6: 'shirt',
    7: 'sneaker',
    8: 'bag',
    9: 'boot'
}


# Randomly sample n sample indices
rand_inds = np.random.randint(0, x_train.shape[0], n)

fig, axes = plt.subplots(ncols=n)
for i, ax in enumerate(axes):
    ax.imshow(x_train[rand_inds[i]], cmap='gray')
    ax.set_title(labels[y_train[rand_inds[i]]])
    ax.axis('off')
```

The class is currently a categorical variable with values 0 to 9. We need to convert this to a one-hot integer encoding.

[29]:
```
# convert class category to one hot vector
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

1.2.2 Build the model

We will construct a model in the same way as the previous example by composing layers starting with the input layer and ending with the output layer. However in this example, we will use Conv2D() layers instead of Dense() layers as our hidden layers. In addition to specifying the number of neurons/filters and activation function for each Conv2D() layer, we need to specify the kernel size (the size of the convolutional kernel/filter). We will use 3×3 convolutions for all layers in this example. Each Conv2D() layer is followed by a MaxPool2D() layer to enable translation invariance.

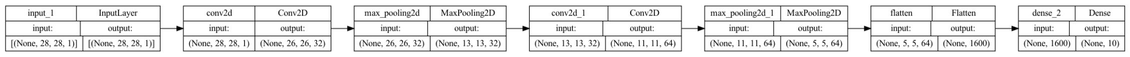
We will still use a Dense() layer for the final output layer. Previously we used the sigmoid activation in our final layer, but this is only used for binary classification outputs (0 to 1). Since this is a multi-class classification, we will use the softmax function which returns the log odds for the multi-class case.

[30]:
```
# Define the input layer
input_layer = keras.Input(shape=[x_train.shape[1], x_train.shape[2], 1])

# Define one hidden layer of Conv2D + Max Pooling
h1 = layers.Conv2D(32, kernel_size=(3, 3), activation="relu")(input_layer)
h1 = layers.MaxPooling2D(pool_size=(2, 2))(h1)

# Define a second Conv2D layer with 64 units, 3x3 kernel, and relu activation
h2 = layers.Conv2D(64, kernel_size=(3, 3), activation="relu")(h1) # YOUR CODE␣
 ↪HERE

# Define a second MaxPooling2D layer with 2x2 pool size
h2 = layers.MaxPooling2D(pool_size=(2, 2))(h2) # YOUR CODE HERE

# Flatten the output from the last hidden layer so it can be passed to Dense␣
 ↪layer
h2 = layers.Flatten()(h2)

# Final output layer
output = layers.Dense(num_classes, activation="softmax")(h2)

# Build the model specifying the input and output layer
model = keras.Model(inputs=input_layer, outputs=output)
```

We can plot our completed model to visualize the input, hidden, and output layers.

[31]:
```
# `rankdir='LR'` is to make the graph horizontal
keras.utils.plot_model(model, show_shapes=True, rankdir="LR")
```

[31]:


input_1 | InputLayer
input: | output:
[(None, 28, 28, 1)] | [(None, 28, 28, 1)]
conv2d | Conv2D
input: | output:
(None, 28, 28, 1) | (None, 26, 26, 32)
max_pooling2d | MaxPooling2D
input: | output:
(None, 26, 26, 32) | (None, 13, 13, 32)
conv2d_1 | Conv2D
input: | output:
(None, 13, 13, 32) | (None, 11, 11, 64)
max_pooling2d_1 | MaxPooling2D
input: | output:
(None, 11, 11, 64) | (None, 5, 5, 64)
flatten | Flatten
input: | output:
(None, 5, 5, 64) | (None, 1600)
dense_2 | Dense
input: | output:
(None, 1600) | (None, 10)

Another way to visualize our model is to print a table of the layers and the shape of the representation at each layer using the `model.summary()` function.

[32]:
```
model.summary()
```

```
Model: "model_1"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 flatten (Flatten)           (None, 1600)              0

 dense_2 (Dense)             (None, 10)                16010

=================================================================
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
_________________________________________________________________
```

1.2.3 Train the model

Now that we've constructed our model, we can compile and train the model as in the previous example. Since we are doing a multi-class classification, we use the categorical cross-entropy loss function instead of binary-cross entropy.

In the first example, we created our own train/val/test subsets. In this example, the FashionMNIST dataset provided a train/test split but not a validation subset. We can have keras automatically split out a fraction of the training data using the `validation_split` argument to `model.fit()` as shown below.

[33]:
```python
batch_size = 512
epochs = 10

model.compile(loss="categorical_crossentropy", optimizer="sgd",␣
 ↪metrics=["accuracy"])

model_result = model.fit(x_train, y_train,
                         batch_size=batch_size,
                         epochs=epochs,
                         validation_split=0.2)
```

```
Epoch 1/10
94/94 [==============================] - 9s 91ms/step - loss: 13.9536 -
accuracy: 0.2250 - val_loss: 1.9482 - val_accuracy: 0.3830
Epoch 2/10
```

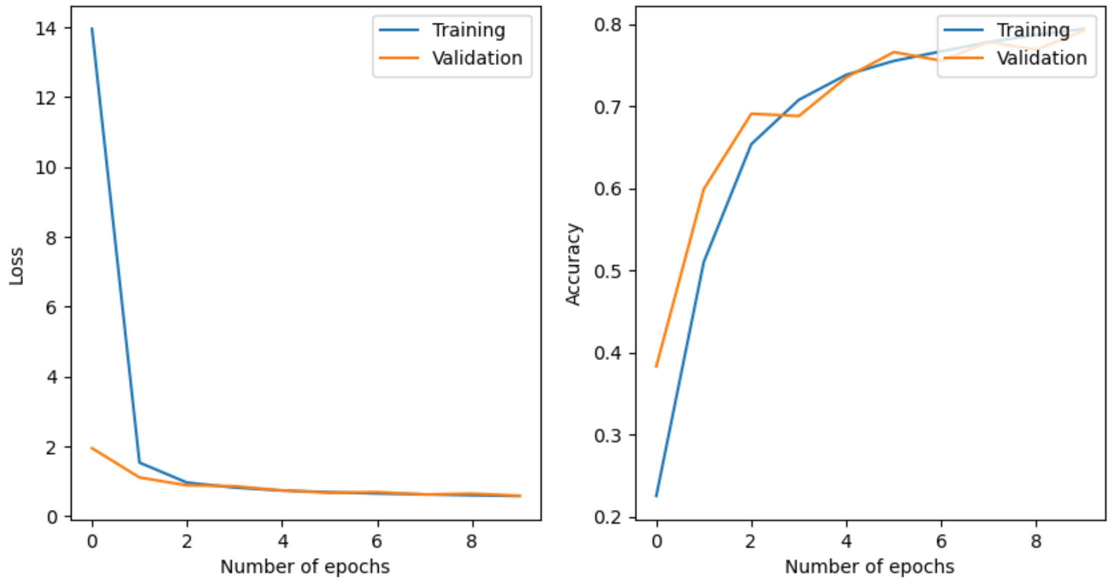
```
94/94 [==============================] - 8s 88ms/step - loss: 1.5363 - accuracy:
0.5101 - val_loss: 1.1110 - val_accuracy: 0.5989
Epoch 3/10
94/94 [==============================] - 8s 87ms/step - loss: 0.9643 - accuracy:
0.6536 - val_loss: 0.8857 - val_accuracy: 0.6907
Epoch 4/10
94/94 [==============================] - 8s 86ms/step - loss: 0.8221 - accuracy:
0.7075 - val_loss: 0.8627 - val_accuracy: 0.6878
Epoch 5/10
94/94 [==============================] - 8s 87ms/step - loss: 0.7377 - accuracy:
0.7381 - val_loss: 0.7404 - val_accuracy: 0.7349
Epoch 6/10
94/94 [==============================] - 8s 88ms/step - loss: 0.6889 - accuracy:
0.7550 - val_loss: 0.6711 - val_accuracy: 0.7657
Epoch 7/10
94/94 [==============================] - 10s 103ms/step - loss: 0.6528 -
accuracy: 0.7666 - val_loss: 0.6942 - val_accuracy: 0.7552
Epoch 8/10
94/94 [==============================] - 24s 259ms/step - loss: 0.6272 -
accuracy: 0.7779 - val_loss: 0.6242 - val_accuracy: 0.7780
Epoch 9/10
94/94 [==============================] - 30s 316ms/step - loss: 0.6016 -
accuracy: 0.7867 - val_loss: 0.6484 - val_accuracy: 0.7686
Epoch 10/10
94/94 [==============================] - 11s 115ms/step - loss: 0.5802 -
accuracy: 0.7941 - val_loss: 0.5898 - val_accuracy: 0.7921
```

Create a figure with two subplots. The first subplot should plot the training and validation loss (`loss` and `val_loss`) and the second subplot should plot the training and validation accuracy (`accuracy` and `val_accuracy`). Make sure you include the axis labels and a legend in each plot.

[34]:
```python
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(10,5))

# YOUR CODE HERE

# Loss plot
ax1.plot(model_result.history['loss'], label='Training')
ax1.plot(model_result.history['val_loss'], label='Validation')
ax1.set_xlabel('Number of epochs')
ax1.set_ylabel('Loss')
ax1.legend(loc='upper right')

# Accuracy plot
ax2.plot(model_result.history['accuracy'], label='Training')
ax2.plot(model_result.history['val_accuracy'], label='Validation')
ax2.set_xlabel('Number of epochs')
```

```
ax2.set_ylabel('Accuracy')
ax2.legend(loc='upper right')
```

[34]:
```
<matplotlib.legend.Legend at 0x1d23de700>
```



Compute and print the test set accuracy. Note that your final accuracy may be different each time you run this due to the randomness in the validation split.

[35]:
```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)

print("Test accuracy:", test_acc)
```

```
Test accuracy: 0.7903000116348267
```