

▼ CSE 572: Lab 9

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.

```
# 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)
```

▼ Feed-forward neural networks

▼ 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.

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

# Print sample rows
data.head()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpea
0	63	1	1	145	233	1	2	150	0	2.
1	67	1	4	160	286	0	2	108	1	1.
2	67	1	4	120	229	0	2	129	1	2.
3	37	1	3	130	250	0	0	187	0	3.

```
# Print the number of rows and columns
data.shape

(303, 14)
```

Split the dataset into three randomly-sampled subsets: training (60%), validation (20%), and test (20%). Use the `seed` variable for the `random_state`.

```
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)".

```
# YOUR CODE HERE
print('Train set: {} instances ,heart disease: {}, no disease: {}'.format(train.shape[0],train[train['target'] == 1].shape[0],train[train['target'] == 0].shape[0]))
print('Validation set: {} instances ,heart disease: {}, no disease: {}'.format(val.shape[0],val[val['target'] == 1].shape[0],val[val['target'] == 0].shape[0]))
print('Test set: {} instances ,heart disease: {}, no disease: {}'.format(test.shape[0],test[test['target'] == 1].shape[0],test[test['target'] == 0].shape[0]))

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

▼ 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

```
# 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)
```

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.

```
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=1>, 'trestbps': <tf.Tensor: shape=(), dtype=int64, numpy=130>, 'chol': <tf.Tensor: shape=(), dtype=int64, numpy=240>, 'fbs': <tf.Tensor: shape=(), dtype=int64, numpy=0>, 'restecg': <tf.Tensor: shape=(), dtype=int64, numpy=0>, 'exang': <tf.Tensor: shape=(), dtype=int64, numpy=0>, 'ca': <tf.Tensor: shape=(), dtype=int64, numpy=0>, 'thal': <tf.Tensor: shape=(), dtype=int64, numpy=0>}
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.

```
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.

```
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).

```

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.

```

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"),
]

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)
    ]
)

```

▼ 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. 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.

```

# 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)

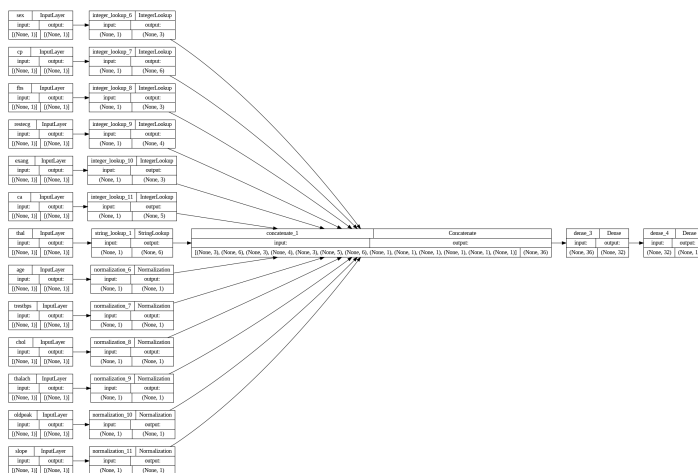
```

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

```

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

```



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.

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

▼ 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.

```
model_result = model.fit(train_ds, epochs=100, validation_data=val_ds)
```

```

6/6 [=====] - 0s 6ms/step - loss: 0.4414 - accuracy: 0.7845 - val_loss: 0.4716 - val_accuracy: 0.8033
Epoch 39/100
6/6 [=====] - 0s 6ms/step - loss: 0.4383 - accuracy: 0.7901 - val_loss: 0.4688 - val_accuracy: 0.8033
Epoch 40/100
6/6 [=====] - 0s 5ms/step - loss: 0.4353 - accuracy: 0.7901 - val_loss: 0.4661 - val_accuracy: 0.8197
Epoch 41/100
6/6 [=====] - 0s 6ms/step - loss: 0.4324 - accuracy: 0.7901 - val_loss: 0.4634 - val_accuracy: 0.8197
Epoch 42/100
6/6 [=====] - 0s 8ms/step - loss: 0.4295 - accuracy: 0.7956 - val_loss: 0.4608 - val_accuracy: 0.8361
Epoch 43/100
6/6 [=====] - 0s 7ms/step - loss: 0.4267 - accuracy: 0.7956 - val_loss: 0.4582 - val_accuracy: 0.8361
Epoch 44/100
6/6 [=====] - 0s 7ms/step - loss: 0.4240 - accuracy: 0.8011 - val_loss: 0.4558 - val_accuracy: 0.8361
Epoch 45/100
6/6 [=====] - 0s 9ms/step - loss: 0.4214 - accuracy: 0.8011 - val_loss: 0.4534 - val_accuracy: 0.8361
Epoch 46/100
6/6 [=====] - 0s 6ms/step - loss: 0.4188 - accuracy: 0.8011 - val_loss: 0.4511 - val_accuracy: 0.8361
Epoch 47/100
6/6 [=====] - 0s 6ms/step - loss: 0.4162 - accuracy: 0.8011 - val_loss: 0.4488 - val_accuracy: 0.8361
Epoch 48/100
6/6 [=====] - 0s 6ms/step - loss: 0.4138 - accuracy: 0.8011 - val_loss: 0.4466 - val_accuracy: 0.8197
Epoch 49/100
6/6 [=====] - 0s 5ms/step - loss: 0.4114 - accuracy: 0.8066 - val_loss: 0.4445 - val_accuracy: 0.8197
Epoch 50/100
6/6 [=====] - 0s 5ms/step - loss: 0.4090 - accuracy: 0.8066 - val_loss: 0.4424 - val_accuracy: 0.8361
Epoch 51/100
6/6 [=====] - 0s 9ms/step - loss: 0.4067 - accuracy: 0.8066 - val_loss: 0.4404 - val_accuracy: 0.8361
Epoch 52/100
6/6 [=====] - 0s 6ms/step - loss: 0.4045 - accuracy: 0.8232 - val_loss: 0.4384 - val_accuracy: 0.8361
Epoch 53/100
6/6 [=====] - 0s 7ms/step - loss: 0.4023 - accuracy: 0.8287 - val_loss: 0.4364 - val_accuracy: 0.8361
Epoch 54/100
6/6 [=====] - 0s 6ms/step - loss: 0.4002 - accuracy: 0.8287 - val_loss: 0.4345 - val_accuracy: 0.8361
Epoch 55/100
6/6 [=====] - 0s 6ms/step - loss: 0.3981 - accuracy: 0.8287 - val_loss: 0.4327 - val_accuracy: 0.8361
Epoch 56/100
6/6 [=====] - 0s 5ms/step - loss: 0.3960 - accuracy: 0.8287 - val_loss: 0.4309 - val_accuracy: 0.8525
Epoch 57/100

```

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:

```

model_result.history.keys()

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.

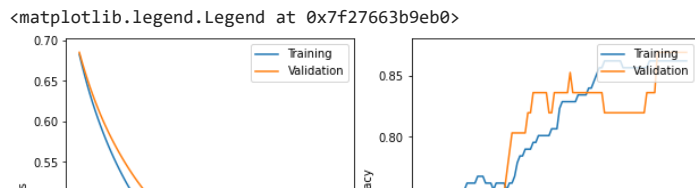
```

import matplotlib.pyplot as plt

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

# YOUR CODE HERE
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')

```



▼ Test the model

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

```
preds = model.predict(test_ds)

1/1 [=====] - 0s 224ms/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.

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

Compute and print the test accuracy.

```
from sklearn.metrics import accuracy_score

# YOUR CODE HERE
print('Test Acc: {}'.format(accuracy_score([y.numpy() for x, y in test_ds][0], pred_classes)))

Test Acc: 0.8032786885245902
```

Question 1: There is a large difference between the training and validation accuracies and the test accuracy. What do you think could explain this difference?

Answer:

Training Accuracy: 0.83

Validation Accuracy: 0.83

Test Accuracy: 0.803

I think there is not any large difference among the accuracies. But if there would have been an difference, it is most likely because model should have overfitted.

▼ 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.

▼ 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

```
(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
```

```
# 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,)
```

```
# 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.

```
# 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.

```
# 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)
```

▼ 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 invariance to small translations.

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.

```
# 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)

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

# 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.

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



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.

```
model.summary()

Model: "model_3"

```

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d_2 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_2 (MaxPooling 2D)	(None, 13, 13, 32)	0
conv2d_3 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_3 (MaxPooling 2D)	(None, 5, 5, 64)	0
flatten_1 (Flatten)	(None, 1600)	0
dense_5 (Dense)	(None, 10)	16010

```

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

```

▼ 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_crossentropy](#) 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.

```
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)
```

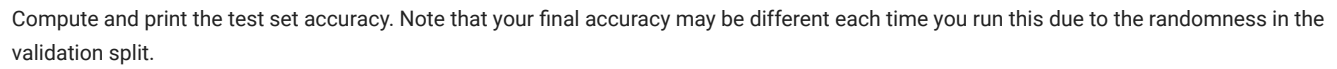
```
Epoch 1/10
94/94 [=====] - 29s 306ms/step - loss: 37.4940 - accuracy: 0.1100 - val_loss: 2.2615 - val_accuracy: 0.1086
Epoch 2/10
94/94 [=====] - 30s 320ms/step - loss: 2.1020 - accuracy: 0.2029 - val_loss: 1.8471 - val_accuracy: 0.3252
Epoch 3/10
94/94 [=====] - 28s 300ms/step - loss: 1.4641 - accuracy: 0.5234 - val_loss: 1.2303 - val_accuracy: 0.6118
Epoch 4/10
94/94 [=====] - 28s 299ms/step - loss: 1.1534 - accuracy: 0.6545 - val_loss: 1.1211 - val_accuracy: 0.6364
Epoch 5/10
94/94 [=====] - 29s 310ms/step - loss: 1.0586 - accuracy: 0.6724 - val_loss: 1.0508 - val_accuracy: 0.6584
Epoch 6/10
94/94 [=====] - 31s 331ms/step - loss: 0.9586 - accuracy: 0.6767 - val_loss: 0.8598 - val_accuracy: 0.7025
Epoch 7/10
94/94 [=====] - 32s 344ms/step - loss: 0.7798 - accuracy: 0.7348 - val_loss: 0.6647 - val_accuracy: 0.7728
Epoch 8/10
94/94 [=====] - 29s 304ms/step - loss: 0.6613 - accuracy: 0.7695 - val_loss: 0.6197 - val_accuracy: 0.7820
Epoch 9/10
94/94 [=====] - 30s 315ms/step - loss: 0.6171 - accuracy: 0.7818 - val_loss: 0.6127 - val_accuracy: 0.7800
Epoch 10/10
94/94 [=====] - 28s 301ms/step - loss: 0.5963 - accuracy: 0.7886 - val_loss: 0.5854 - val_accuracy: 0.7919
```

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.

```
import matplotlib.pyplot as plt

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

# YOUR CODE HERE
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')
23
ax2.set_ylabel('Accuracy')
ax2.legend(loc='upper right')
```



XXXXXXXXXX OF XXXXXXXXXX