0.1 Overview

This notebook introduces you to the basics of Neural Networks. You will start by implementing and training a simple neural network using Numpy. Later, you will move on to training fully connected networks using the PyTorch library. We also provide some hints about convolutional neural networks (CNNs).

The topics that will be covered are:

0.1.1 Programming Tasks

For the programming tasks you will need to replace the following comment and exception with your own code:

```
# YOUR CODE HERE
raise NotImplementedError()
```

Most programming tasks are followed by a cell with tests (using the `assert` keyword from python). You can consult these cells while developing your implementation and for validation. Note that there may be additional, hidden tests.

Note: The `@contract` decorators make sure the data types and shapes are correct for the inputs and outputs. See here (https://andreacensi.github.io/contracts/tour.html#quick-tour) for more. If you are more comfortable working without these, you can comment out the lines starting with `@contract`. However, in that case it can get tedious to locate the exact source of a bug.

0.1.2 Open Questions

The notebook also contains a few open questions. You don't get points for the open questions, they are here to improve your understanding of the topic. For the open questions you can put your answer in the cell below the question, replace the text "YOUR ANSWER HERE" with your own answer. You can later check your answer with the answer given in the solution version of the notebook.

0.1.3 Deadlines

1. All programming tasks of Part 1 need to be submitted before Tuesday 27/04/2021 17:00.
2. All programming tasks of Part 2 need to be submitted before Friday 30/04/2021 17:00.

Part 3 and Part 4 are optional.

In [1]:

```
# DO NOT INSTALL THE LIBRARIES WHEN WORKING ON ifi-europa.uibk.ac.at

# Make sure that the required libraries are installed
# If you are using Google Colab, remember to upload the requirements file before
# running this cell
# If you are running this notebook locally, the requirements file needs to be in
# the same location as this notebook
import os
running_local = True if os.getenv('JUPYTERHUB_USER') is None else False

if running_local:
    import sys
    !{sys.executable} -m pip install -r requirements_week06.txt
```

[go to top]

0.1.4 Setup

In [2]:

```
from tqdm.notebook import tqdm, trange
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from IPython.display import set_matplotlib_formats
from contracts import contract
import sklearn
from sklearn import cluster, datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

set_matplotlib_formats('svg')
%matplotlib inline
```

In [3]:

```
# Random seed for reproducability
random_seed = 123
np.random.seed(random_seed)

# Total number of data points
n_samples = 1500

# Toy dataset
moons = datasets.make_moons(n_samples=n_samples, noise=.35, random_state=random_see
```

You will be working with the moons (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html) dataset from `sklearn`. Let's load the data and split it up into train and test sets. The functions for doing this are provided to you below.

In [4]:

```
# Function for splitting into train and test sets
def split(dataset):
    """
    Splits a dataset from sklearn into train and test sets.

    :param: dataset: sklearn dataset (data, labels) (2-tuple of numpy arrays)
    :returns: x_train, x_test, y_train, y_test (4-tuple of numpy arrays)
    """

    # Get data and labels
    X,Y = dataset

    # Reshape Y to [num_points, 1]
    Y = np.expand_dims(Y, axis=1)

    # Split the data into train and test sets
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=random_s

    print("Shape of data:")
    print(f"X_train: {X_train.shape}, X_test: {X_test.shape}, Y_train: {Y_train.sha
    return X_train, X_test, Y_train, Y_test
```

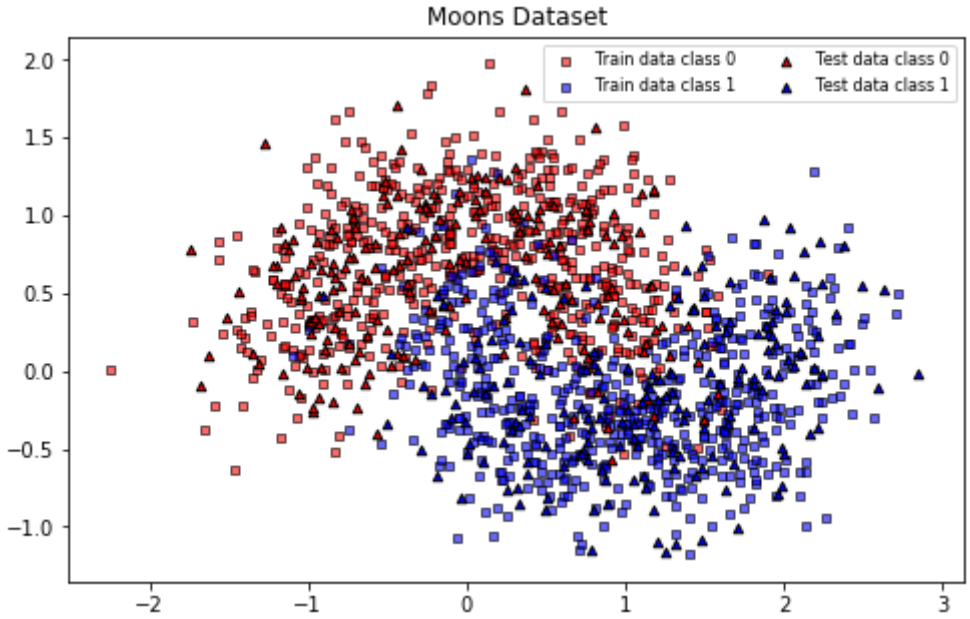
In [5]:

```
moons_x_train, moons_x_test, moons_y_train, moons_y_test = split(moons)
```

```
Shape of data:
X_train: (1125, 2), X_test: (375, 2), Y_train: (1125, 1), Y_test: (37
5, 1)
```

In [6]:

```
# Visualize the data

plt.figure(figsize=(8,5))

markers = ('s', '^', 'x', 'o', 'v')
colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
cmap = ListedColormap(colors[:len(np.unique(moons_y_train))])

for idx, yv in enumerate(np.unique(moons_y_train[:,0])):
    plt.scatter(x=moons_x_train[moons_y_train[:,0]==yv, 0],
                y=moons_x_train[moons_y_train[:,0]==yv, 1],
                alpha=0.6,
                c=[cmap(idx)],
                marker=markers[0],
                label=f"Train data class {yv}",
                edgecolors='k',
                s=20)

for idx, yv in enumerate(np.unique(moons_y_test[:,0])):
    plt.scatter(x=moons_x_test[moons_y_test[:,0]==yv, 0],
                y=moons_x_test[moons_y_test[:,0]==yv, 1],
                alpha=1.0,
                c=[cmap(idx)],
                marker=markers[1],
                label=f"Test data class {yv}",
                edgecolors='k',
                s=20)

plt.legend(ncol=2, fontsize=8)
plt.title('Moons Dataset')
plt.show()
```

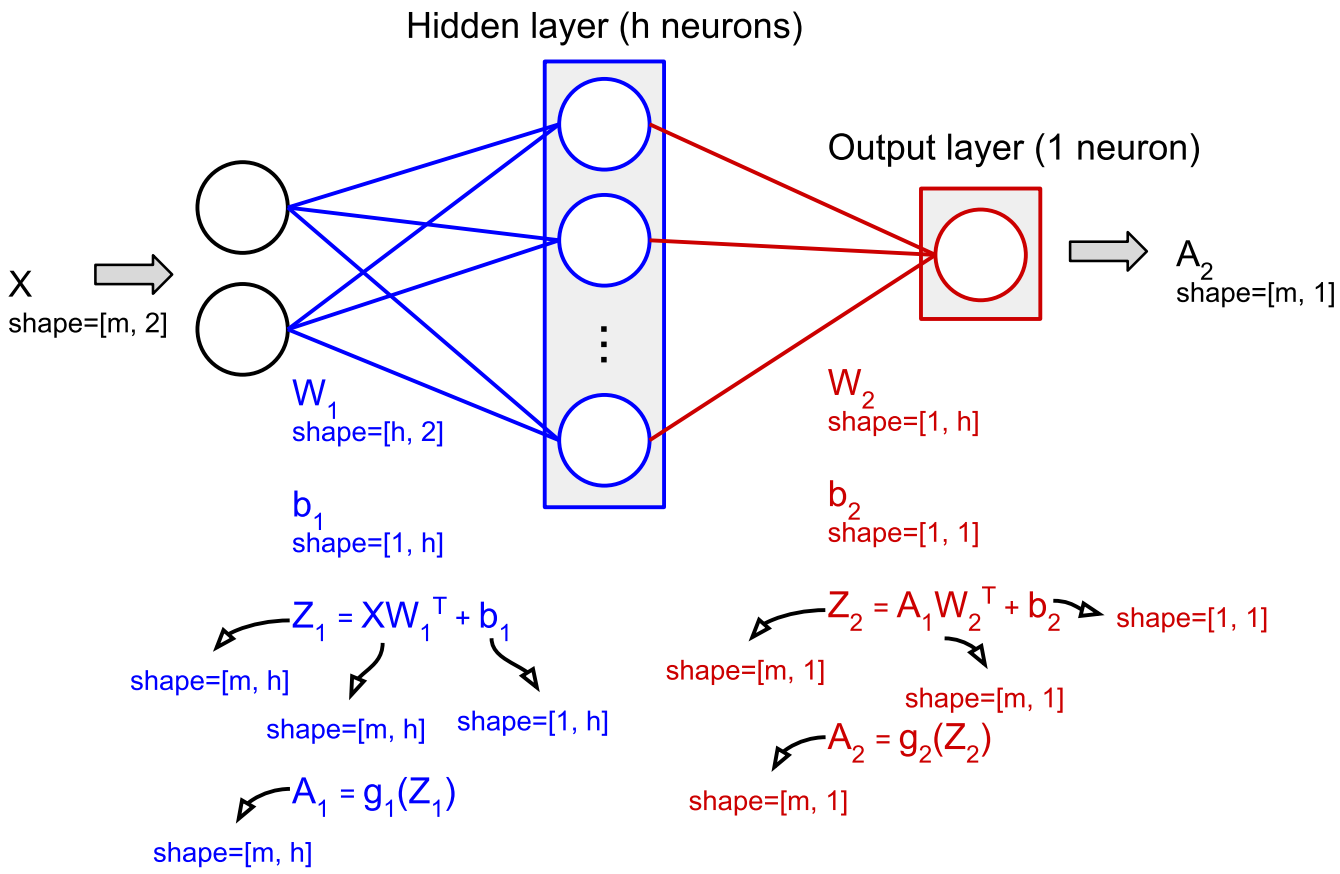


[go to top]

1 Part 1: Binary Classification with a Simple Neural Network (using Numpy)

For this task you will be building and training a simple neural network (consisting of 1 hidden layer) using numpy. A diagram of this simple neural network is provided below, along with the shapes of all the numpy arrays that you will be using.

- In particular, note that data will be fed into the network in batches.
- For example, in the figure below, the input X is of shape $[m, 2]$, which means that each data point in X is 2-dimensional and there are m such data points (batch size is m).
- X is then combined with the weights W_1 and biases b_1 to produce the linear output Z_1.
- Z_1 is then passed through g_1, the activation function of the hidden layer to produce the hidden layer output A_1.
- A_1 now forms the input to the output layer, and is combined with W_2 and b_2 to produce Z_2.
- Z_2 is fed in to g_2, the activation function of the output layer to produce A_2, which is the network's prediction for X.
- Since this network will be used for binary classification, g_2 is the sigmoid function and A_2 has the shape $[m, 1]$.



For building and training the network, the following steps need to be performed:

Initialize → Forward → Calculate Loss → Backward → Weight Update

1.1 Numpy Network: Initialization

The first step of building the network shown above is to initialize the weights and biases of the network. For the weights, you need to construct numpy arrays which are initialized randomly. But these random weights should be close to zero (but not all zero). You can use the [np.random.randn (https://numpy.org/doc/stable/reference/random/generated/numpy.random.randn.html)](https://numpy.org/doc/stable/reference/random/generated/numpy.random.randn.html) function with the correct shape for this, but DO NOT `np.random.randn` samples numbers from the "standard normal" distribution. If you use exactly the returned values, there is a high chance of numerical overflow while training the network. You will

Type: string, but 100%

have to modify the returned values so that they are much smaller (e.g. divide by a factor of 100). For the biases, you can use `np.zeros` with the proper shape. Complete the missing parts of the function `init_model` for initializing the network.

In [7]:

```
@contract(X='array[MxD],M>0,D>0',
          Y='array[Mx1],M>0',
          n_hidden='int,>0')
def init_model(X, Y, n_hidden):
    """
    Function for initializing the parameters (weights and biases) of our neural net
    We are working with a network which has 1 hidden layer and 1 output layer.

    param: X: Input data (numpy array of shape [m,input_dim])
    param: Y: Target labels (numpy array of shape [m,output_dim])
    param: n_hidden: Number of hidden units (int)

    returns: parameters:  python dictionary containing the initialized parameters:
                          W1 - numpy array of shape (n_hidden, input_dim)
                          b1 - numpy array of shape (1, n_hidden)
                          W2 - numpy array of shape (output_dim, n_hidden)
                          b2 - numpy array of shape (1, output_dim)
    """

    # Calculate the number of nodes in the input and output layers
    input_dim = X.shape[1]
    output_dim = Y.shape[1]

    # Initialize the weights and biases
    # assign to the variables W1, b1, W2 and b2

    # YOUR CODE HERE
    W1 = np.random.randn(n_hidden,input_dim) / 100
    b1 = np.zeros((1, n_hidden))
    W2 = np.random.randn(output_dim, n_hidden) / 100
    b2 = np.zeros((1, output_dim))

    parameters = {'W1': W1,
                  'b1': b1,
                  'W2': W2,
                  'b2': b2}

    return parameters
```

In [8]:

```
# Test the init function
m = 250
X = np.random.rand(m, 2)
Y = np.random.rand(m, 1)
n_hidden = 45
parameters = init_model(X, Y, n_hidden)

assert parameters["W1"].shape == (n_hidden, 2), \
f"The shape of W1 is incorrect, it should be ({n_hidden}, 2) but it is {parameters[
assert parameters["b1"].shape == (1, n_hidden), \
f"The shape of b1 is incorrect, it should be (1, {n_hidden}) but it is {parameters[
assert parameters["W2"].shape == (1, n_hidden), \
f"The shape of W2 is incorrect, it should be (1, {n_hidden}) but it is {parameters[
assert parameters["b2"].shape == (1, 1), \
f"The shape of b2 is incorrect, it should be (1, 1) but it is {parameters['b2'].sha
```

1.2 Numpy Network: Forward Propagation

The steps for forward propagating input data through the network has already been described here. These steps are summarized again below. Complete the functions `sigmoid` (https://en.wikipedia.org/wiki/Sigmoid_function) for the output activation function. Keep in mind that `sigmoid` should be able to handle both single numbers as well as arrays. Then complete the function `forward_propagation` by implementing these steps:

In [9]:

```
@contract(z='array[AxB],A>0,B>0|float',
          returns='array[AxB],A>0,B>0|float,>=0.0,<=1.0')
def sigmoid(z):
    """
    Computes the sigmoid function.
    Capable of vectorizing (works for both single floats as well as numpy arrays)

    param: z: Input (float or numpy array)
    returns: Sigmoid of input (float or numpy array)
    """

    # YOUR CODE HERE
    sigmoid_value = 1 / (1 + np.exp(-z))

    return sigmoid_value
```

Use `np.tanh` for g_1 and the `sigmoid` function for g_2.

Steps for Forward Propagation

1. $Z_1 = XW_1^T + b_1$
2. $A_1 = g_1(Z_1)$ (where g_1 is the activation function of layer 1)
3. $Z_2 = A_1 W_2^T + b_2$
4. $A_2 = g_2(Z_2)$ (where g_2 is the sigmoid function)

Typesetting math: 100%

In [10]:

```
@contract(X='array[MxD],M>0,D>0')
def forward_propagation(X, parameters):
    """
    Forward-propagates the data through the neural network

    param: X: Input data (numpy array of shape [m,input_dim])
    param: python dictionary containing the initialized parameters:
                          W1 - numpy array of shape (n_hidden, input_dim)
                          b1 - numpy array of shape (1, n_hidden)
                          W2 - numpy array of shape (output_dim, n_hidden)
                          b2 - numpy array of shape (1, output_dim)

    returns: A2: The network prediction (numpy array of shape [m,output_dim])
             cache: python dictionary containing Z1, A1, Z2 and A2
    """

    # Retrieve weights and biases
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']


    # Implement Forward Propagation to calculate A2 (probabilities)
    #   assign to variables Z1, A1, Z2, A2
    #
    #   use `np.tanh` for A1
    #   use the `sigmoid` function for A2

    # YOUR CODE HERE
    Z1 = X.dot(W1.T) + b1
    A1 = np.tanh(Z1)
    Z2 = A1.dot(W2.T) + b2
    A2 = sigmoid(Z2)

    cache = {'Z1': Z1,
             'A1': A1,
             'Z2': Z2,
             'A2': A2}

    return A2, cache  # Think: Why do we return the cache?
```

In [11]:

```
# Test the forward function using the variables from the previous test block
A2, cache = forward_propagation(X, parameters)

assert A2.shape == (X.shape[0], 1), \
f"The shape of A2 is incorrect, it should be ({X.shape[0]}, 1) but it is {A2.shape}

assert cache["Z1"].shape == (m, n_hidden), \
f"The shape of cache['Z1'] is incorrect, it should be ({m}, {n_hidden}) but it is {

assert cache["A1"].shape == (m, n_hidden), \
f"The shape of cache['A1'] is incorrect, it should be ({m}, {n_hidden}) but it is {

assert cache["Z2"].shape == (m, 1), \
f"The shape of cache['Z2'] is incorrect, it should be ({m}, 1) but it is {cache['Z2

assert cache["A2"].shape == (m, 1), \
f"The shape of cache['A2'] is incorrect, it should be ({m}, 1) but it is {cache['A2
```

1.3 Numpy Network: Binary Cross Entropy Loss

Complete the function `loss` which computes the binary cross entropy loss J, given by the following formula:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(Y^{(i)}\log\left(A_2^{(i)}\right) + (1-Y^{(i)})\log\left(1-A_2^{(i)}\right)\right)$$

where m is the number of data points, $Y^{(i)}$ is the i^{th} target label, and $A_2^{(i)}$ is the i^{th} prediction.

Hint: Since $\log(0) \to \infty$ use of the logarithm might lead to numerical issues (for example for very small numbers). To avoid this issue, add small number `eps` in the log functions $\log(x + \text{eps})$.

In [12]:

```
@contract(A2='array[MxO],M>0,O>0',
          Y='array[MxO],M>0,O>0',
          returns='float, >=0')
def loss(A2, Y):
    """
    Computes the binary cross entropy loss

    param: A2: The network prediction (numpy array of shape [m,output_dim])
    param: Y: Target labels (numpy array of shape [m,output_dim])

    returns: loss: cross-entropy loss (float)
    """

    eps = 1e-15

    # Number of data points
    num_data_points = Y.shape[0]

    # Compute the cross-entropy loss
    #
    #   Make sure to return a scalar!

    # YOUR CODE HERE
    sum_m = 0
    for i in range(num_data_points):
        sum_m += (Y[i] @ np.log(A2[i] + eps) + (1 - Y[i]) @ np.log(1 - A2[i] + eps)

    loss_value = -(sum_m / num_data_points)

    return loss_value
```

In [13]:

```
# Test the loss function
loss_value = loss(A2, Y)
assert isinstance(loss_value, float) and loss_value>=0.0, "loss_value should be a f
```

1.4 Numpy Network: Backward Propagation

The backward function is used for computing the gradients of the trainable parameters (the weights and the biases) of the network. The steps for the backward computation are provided below. These formulas were derived assuming that g_2, the output activation function is sigmoid. All the symbols have the same meaning as in earlier cells. The backward function is provided to you, but you are encouraged to study and understand it.

A detailed proof is provided here (https://iis.uibk.ac.at/public/auddy/proof/Backpropagation_Proof.pdf).

Steps for Backward Propagation

where $\odot$ denotes the Hadamard product.

($\text{d}Z_{2}$ stands for $\cfrac{\partial{J}}{\partial{Z_{2}}}$ and so on)

1. $\text{d}Z_{2} = A_{2} - Y$ (assuming sigmoid function is used in the output layer)
2. $\text{d}W_{2} = \cfrac{1}{m} \text{d}Z_{2}^{T} A_{1}$
3. $\text{d}b_{2} = \cfrac{1}{m}$np.sum($\text{d}Z_{2}$, axis=0, keepdims=True)

Type $\text{d}W_{2} = \cfrac{1}{m}

4. $\text{d}Z_{1} = (\text{d}Z_{2}W_{2}) \odot (g'_{1}(Z_{1}))$
5. $\text{d}W_{1} = \cfrac{1}{m} \text{d}Z_{1}^{T}X$
6. $\text{d}b_{1} = \cfrac{1}{m}$np.sum($\text{d}Z_{1}$, axis=0, keepdims=True)

In [14]:

```
def backward_propagation(parameters, cache, X, Y):
    """
    Backward function for computing gradients

    param: parameters: python dictionary containing parameters W1, b1, W2 and b2
    param: cache: python dictionary containing Z1, A1, Z2 and A2
    param: X: Input data (numpy array of shape [m,input_dim])
    param: Y: Target labels (numpy array of shape [m,output_dim])

    returns: grads: python dictionary containing gradients dW1, db1, dW2 and db2
    """

    m = X.shape[1]

    # First, retrieve W1 and W2 from the dictionary 'parameters'
    W1 = parameters['W1']
    W2 = parameters['W2']

    # Retrieve also A1 and A2 from dictionary 'cache'
    A1 = cache['A1']
    A2 = cache['A2']

    # Backward propagation: calculate dW1, db1, dW2, db2.
    dZ2 = A2 - Y
    dW2 = np.dot(dZ2.T, A1)/m
    db2 = np.sum(dZ2, axis=0, keepdims=True)/m
    dZ1 = np.multiply(np.dot(dZ2, W2), (1 - np.power(A1, 2)))
    dW1 = np.dot(dZ1.T, X)/m
    db1 = np.sum(dZ1, axis=0, keepdims=True)/m

    grads = {'dW1': dW1,
             'db1': db1,
             'dW2': dW2,
             'db2': db2}

    return grads
```

1.5 Numpy Network: Updating Weights

After the gradient computation, the weights and biases need to be updated in the direction of the negative gradient by using the learning rate. The weight update is done using the following formula: $$ \theta = \theta - \alpha \frac{\partial J }{ \partial \theta }$$ where α is the learning rate and θ represents a trainable parameter (weight or bias).

Complete the function `update_parameters` for performing weight updates.

In [15]:

```
def update_parameters(parameters, grads, learning_rate = 0.01):
    """
    Updates parameters using the gradient descent update rule given above

    param: parameters: python dictionary containing parameters W1, b1, W2 and b2
    param: grads: python dictionary containing gradients dW1, db1, dW2 and db2

    returns: parameters: python dictionary containing updated parameters W1, b1, W2
    """

    # Steps:
    #   1. Retrieve each parameter (W1, b1, W2, b2) from the dictionary "parameters
    #   2. Retrieve each gradient (dW1, db1, dW2, db2) from the dictionary "grads"
    #   3. Update each parameter (W1, b1, W2, b2)

    # YOUR CODE HERE

    # retrieve weights and biases
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    # retrieve gradients
    dW1 = grads['dW1']
    db1 = grads['db1']
    dW2 = grads['dW2']
    db2 = grads['db2']

    # update weights and biases
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2

    parameters = {'W1': W1,
                  'b1': b1,
                  'W2': W2,
                  'b2': b2}

    return parameters
```

In [16]:

```
# Test the update function
grads = backward_propagation(parameters, cache, X, Y)
updated_parameters = update_parameters(parameters, grads, learning_rate = 0.01)

for k, v in parameters.items():
    assert updated_parameters[k].shape == v.shape, \
    f"Shape of updated_parameters['{k}']: {updated_parameters[k].shape} is not equa
```

1.6 Numpy Network: Accuracy and Prediction

Complete the functions `predict` for making predictions and `calc_accuracy` for computing the accuracy of predictions. The prediction function should assign outputs of the network which are > 0.5 to class 1 and the rest to class 0.

In [17]:

```
@contract(X='array[MxD],M>0,D>0',
          returns='array[Mx1],M>0')
def predict(parameters, X):
    """
    Make predictions (class 0 or 1) using the learned parameters

    param: parameters: python dictionary containing parameters W1, b1, W2 and b2
    param: X: Input data for which label is to be predicted (numpy array of shape [

    returns: predictions: Predictions of our model (numpy array of shape [m,1] cont
    """

    # Compute probabilities using forward propagation, and classify to 0/1 using 0.

    # YOUR CODE HERE
    probs = forward_propagation(X, parameters)[0]
    predictions = np.where(probs < 0.5, 0, 1)

    return predictions.astype(int)
```

In [18]:

```
# Test the predict function
predictions = predict(parameters, X)

assert predictions.shape == (X.shape[0], 1), \
f"Shape of predictions is {predictions.shape}, but it must be {(X.shape[0], 1)}"

# only labels 0, 1 are allowed
assert set(np.unique(predictions)).issubset(set([0, 1])), \
f"Predictions can only contain 0s and 1s but they contain {np.unique(predictions)}"
```

In [19]:

```
@contract(Y_pred='array[Mx1],M>0',
          Y='array[Mx1],M>0',
          returns='float,>=0.0,<=1.0')
def calc_accuracy(Y_pred, Y):
    """
    Calculates the accuracy of the predictions against the true labels
    (What percent of the predicted labels Y_pred matches the true labels in Y)

    param: Y_pred: Predictions of our model (numpy array of shape [m,1] containing
    param: Y: Target labels (numpy array of shape [m,output_dim])

    returns: accuracy (float between 0.0 and 1.0)

    """
    # YOUR CODE HERE
    accuracy = np.sum(Y_pred == Y) / Y.shape[0]

    return accuracy
```

In [20]:

```
# Test the accuracy function
accuracy = calc_accuracy(A2, Y)
assert isinstance(accuracy, float) and accuracy >= 0.0 and accuracy <= 1.0, \
"Accuracy must be a float between 0.0 and 1.0"
```

1.7 Numpy Network: Putting Everything Together

The function `train_neural_network` uses batch gradient descent (entire training data is used at the same time to compute gradients) for training our numpy neural network. Complete this function by using the functions that you have implemented till now.

In [25]:

```
@contract(X_train='array[MxD],M>0,D>0',
          Y_train='array[Mx1],M>0',
          n_hidden='int,>0',
          learning_rate='float,>0.0',
          num_iterations='int,>0')
def train_neural_network(X_train, Y_train, n_hidden, learning_rate=0.01, num_iterat
    """
    Trains the network using batch gradient descent

    param: X_train: Training data (numpy array of shape [m,input_dim])
    param: Y_train: Training data labels (numpy array of shape [m,output_dim])
    param: n_hidden: Number of neurons in the hidden layer (int)
    param: num_iterations: Number of iterations in gradient descent loop (int)
    param: learning_rate: Learning rate for gradient descent (float)

    returns: parameters: Learned parameters - python dictionary containing W1, b1,
    """

    losses = list()


    # Follow these steps:

    # Initialize parameters, then retrieve W1, b1, W2, b2.
    # For each iteration (till `num_iterations` is reached):
    #     Forward propagation: Compute A2, cache using `X_train` and `parameters`
    #     Loss. Inputs: `A2`, `Y_train`. Outputs: `loss_value`. Append `loss_value`
    #     Backpropagation. Inputs: `parameters`, `cache`, `X_train`, `Y_train`. Out
    #     Parameter update. Inputs: `parameters`, `grads`, `learning_rate`. Outputs

    # YOUR CODE HERE
    parameters = init_model(X_train, Y_train, n_hidden)
    for i in range(num_iterations):
        (A2, cache) = forward_propagation(X_train, parameters)
        loss_value = loss(A2, Y_train)
        losses.append(loss_value)
        grads = backward_propagation(parameters, cache, X_train, Y_train)
        parameters = update_parameters(parameters, grads, learning_rate)

    return parameters, losses
```

In [26]:

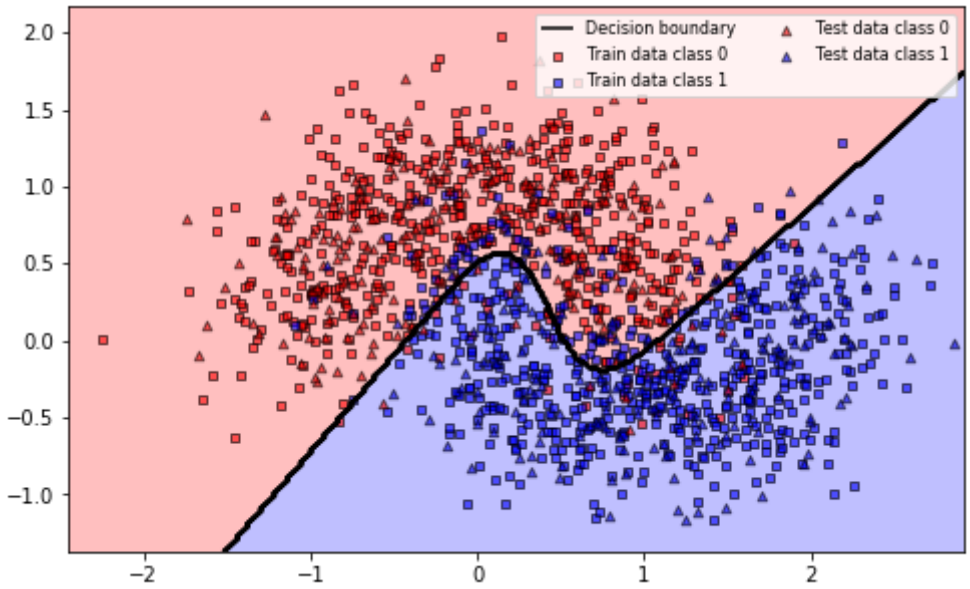
```
# Lets train a network with 3 units in the hidden layer using the moon dataset
params_moons_3, losses_moons_3 = train_neural_network(moons_x_train,
                                                      moons_y_train,
                                                      3,
                                                      learning_rate=0.01,
                                                      num_iterations=5000)
```

In [27]:

```
def plot_decision_boundary_nn(ax, predict_fn, params, X_train, Y_train, X_test, Y_t
    """
    Plots the decision boundary predicted by the neural network
    Don't worry about the details of this function
    """

    markers = ('s', '^', 'x', 'o', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(Y_train))])

    # For constructing the grid limits
    h = 0.02
    x_min, x_max = X_train[:,0].min() - 10*h, X_train[:,0].max() + 10*h
    y_min, y_max = X_train[:,1].min() - 10*h, X_train[:,1].max() + 10*h
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # Make predictions for each value inside the grid and reshape
    Z = predict_fn(params, np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    ax.contourf(xx, yy, Z, cmap=cmap, alpha=0.25)
    cs = ax.contour(xx, yy, Z, colors='k', alpha=1.0)
    cs.collections[0].set_label("Decision boundary")

    for idx, yv in enumerate(np.unique(Y_train[:,0])):
        ax.scatter(x=X_train[Y_train[:,0]==yv, 0],
                   y=X_train[Y_train[:,0]==yv, 1],
                   alpha=0.6,
                   c=[cmap(idx)],
                   marker=markers[0],
                   s=20,
                   label=f"Train data class {yv}",
                   edgecolors='k')

    for idx, yv in enumerate(np.unique(Y_test[:,0])):
        ax.scatter(x=X_test[Y_test[:,0]==yv, 0],
                   y=X_test[Y_test[:,0]==yv, 1],
                   alpha=0.6,
                   c=[cmap(idx)],
                   marker=markers[1],
                   s=20,
                   label=f"Test data class {yv}",
                   edgecolors='k')
    ax.legend(ncol=2, fontsize=8)
```

In [28]:

```
# Plot the decision boundary for network we just trained
fig, ax = plt.subplots(1, 1, figsize=(8,5))
plot_decision_boundary_nn(ax,
                          predict,
                          params_moons_3,
                          moons_x_train,
                          moons_y_train,
                          moons_x_test,
                          moons_y_test)
```



Try training another network with a larger number of hidden units and check what the decision boundary looks like.

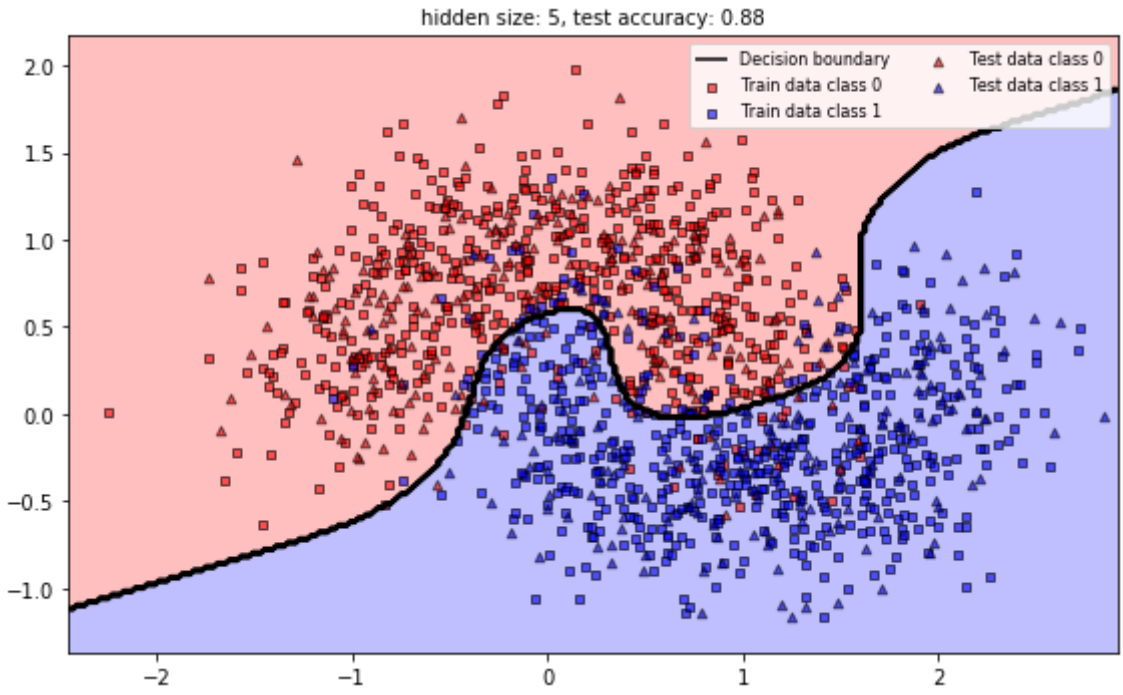
In [29]:

```
hidden = [5]
params_moons_dict = dict()

for h in hidden:
    print(f"Training for hidden size {h}")
    params, losses = train_neural_network(moons_x_train,
                                          moons_y_train,
                                          h,
                                          learning_rate=0.01,
                                          num_iterations=5000)

    train_acc = calc_accuracy(predict(params, moons_x_train), moons_y_train)
    test_acc = calc_accuracy(predict(params, moons_x_test), moons_y_test)

    params_moons_dict[h] = {"params": params,
                            "losses": losses,
                            "train_acc": train_acc,
                            "test_acc": test_acc}
```

```
Training for hidden size 5
```

In [30]:

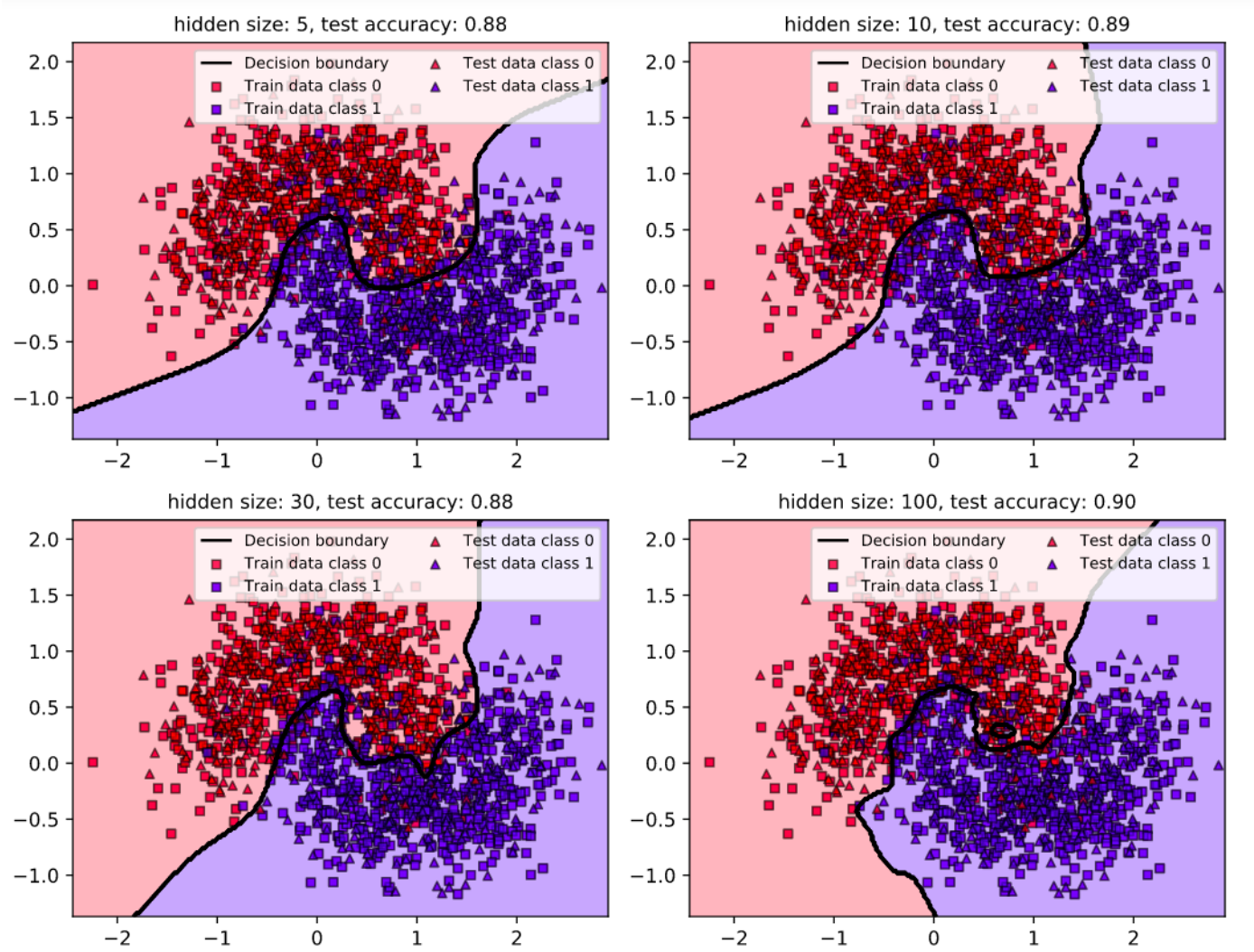
```
# Plotting the decision boundary

fig, ax = plt.subplots(1, 1, figsize=(8,5))

for i, (h, value) in enumerate(params_moons_dict.items()):
    plot_decision_boundary_nn(ax,
                              predict,
                              value["params"],
                              moons_x_train,
                              moons_y_train,
                              moons_x_test,
                              moons_y_test)
    ax.set_title(f"hidden size: {h}, test accuracy: {value['test_acc']:.2f}",
                 fontsize=10)
plt.tight_layout()
```



If everything goes well, you should be able to observe that the decision boundary becomes much more non-linear.

Hint: If the decision boundary appears as a straight line, it may be because that network was not able to learn. Try changing the hyperparameters, such as the number of hidden units, the learning rate, or the number of iterations, and try again.

Given below is an image showing how the decision boundary becomes more non-linear as the number of hidden units is increased. For a very high number of hiddent units, the classifier overfits the training data.



.

[go to top]

2 Part 2: PyTorch Basics

PyTorch (https://pytorch.org/) is a Python-based scientific computing library that is commonly used for deep learning. It provides helpful features such as automatic differentitaion, the ability to utilize GPUs, and a large collection of deep-learning related algorithms. There are several other alternatives to PyTorch, such as Tensorflow or Caffe, but for this notebook, you will be using PyTorch.

If you are unfamiliar with PyTorch, we recommend you work through the PyTorch 60minutes Blitz (https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html) first.

For the exercises below, no prior knowledge of PyTorch is assumed.

Visit this link (https://pytorch.org/tutorials/index.html) for some PyTorch tutorials.

First, let us use the same dataset that was used till now, but this time you will create a multilayered fully connected neural network using PyTorch for classifying the points of the dataset.

In [42]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchsummary import summary

# Check https://pytorch.org/docs/stable/notes/randomness.html#reproducibility
torch.manual_seed(random_seed)
```

Out[42]:

```
<torch._C.Generator at 0x7fe49c6e4ed0>
```

2.1 PyTorch Basics: Network Definition

Given below is the class definition for a fully connected network with 2 hidden layers and 1 output layer. Complete the missing parts. Comments are provided to guide you. **It may be helpful to get acquainted with the different activation functions in PyTorch (https://pytorch.org/docs/stable/nn.functional.html#non-linear-activation-functions).** The module is imported as `F`.

In [83]:

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Net, self).__init__()
        self.hidden_size = hidden_size
        self.input_size = input_size
        self.output_size = output_size
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # Follow these steps:
        #
        # Flatten the input x keeping the batch dimension the same
        # Use the relu activation on the output of self.fc1(x)
        # Use the relu activation on the output of self.fc2(x)
        # Pass x through fc3 but do not apply any activation function (think why no


        # YOUR CODE HERE
        x = x.view(-1, self.input_size)
        print(x.shape)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x  # Return x (logits)
```

In [84]:

```
# Verify the network class
testnet = Net(2, 20, 1)
ip = torch.rand((350,2))
# This is not the best way to pass the data through the net
# In the next task you will need to find the better way
op = testnet.forward(ip)
assert op.shape==(350,1), f"Output shape must be (350,1) but it is {op.shape}"
```

```
torch.Size([350, 2])
```

2.2 PyTorch Basics: Training

Once the network is defined, it can be trained by following roughly the same sequence of steps as the numpy network you trained earlier. This is done in the function `train_neural_network_pytorch` given below. Complete the missing parts of this function.

Note:

- The weights and biases of the network are initialized when the network object is created (outside the training function)
- PyTorch operates on Tensors (https://pytorch.org/docs/stable/tensors.html) which are multi-dimensional matrices containing elements of a single data type. Although tensors look similar to numpy arrays, they have several additional properties. Therefore, any data or labels that are used with a PyTorch network need to be converted into Tensors.
- PyTorch includes a large number of training algorithms (optimizers) for training networks. See here for details of how to use an optimizer (https://pytorch.org/docs/stable/optim.html) and their different types. The code for using an optimizer for this notebook is provided to you.
- Several types of loss functions (**criterion**) are also available (https://pytorch.org/docs/stable/nn.html#loss-functions).

In [65]:

```
# Define hyperparameters
LEARNING_RATE = 0.001
MOMENTUM = 0.9
MAX_ITERATIONS = 5000
INPUT_SIZE = 2
HIDDEN_SIZE = 10
OUTPUT_SIZE = 1
```

We use the BCEWithLogitsLoss function (https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html#torch.nn.BCEWithLogitsLoss) so familiarize yourself with this type of function.

In [46]:

```
# Initialize the network
net = Net(INPUT_SIZE, HIDDEN_SIZE, OUTPUT_SIZE)

# Define the loss criterion and the training algorithm
criterion = nn.BCEWithLogitsLoss()  # Be careful, use binary cross entropy for bina
optimizer = optim.SGD(net.parameters(), lr=LEARNING_RATE, momentum=MOMENTUM)
```

In [47]:

```
@contract(inputs='array[MxD],M>0,D>0',
          labels='array[Mx1],M>0',
          iterations='int,>0')
def train_neural_network_pytorch(net, inputs, labels, optimizer, criterion, iterati
    """
    Function for training the PyTorch network.

    :param net: the neural network object
    :param inputs: numpy array of training data values
    :param labels: numpy array of training data labels
    :param optimizer: PyTorch optimizer instance
    :param criterion: PyTorch loss function
    :param iterations: number of training steps
    """
    net.train()  # Before training, set the network to training mode

    for iter in trange(iterations):  # loop over the dataset multiple times

        # It is a common practice to track the losses during training
        # Feel free to do so if you want

        # Get the inputs; data is a list of [inputs, labels]
        # Convert to tensors if data is in the form of numpy arrays
        if not torch.is_tensor(inputs):
            inputs = torch.from_numpy(inputs.astype(np.float32))

        if not torch.is_tensor(labels):
            labels = torch.from_numpy(labels.astype(np.float32))


        # Follow these steps:
        # 1. Reset gradients: Zero the parameter gradients (Check the link for opti
        #                     above to find the correct function)
        # 2. Forward: Pass `inputs` through the network. This can be done calling
        #             the `forward` function of `net` explicitly but there is an
        #             easier way that is more commonly used
        # 3. Compute the loss: Use `criterion` and pass it the `outputs` and `label
        #                      Check the link in the text cell above for details
        # 4. Backward: Call the `backward` function in `loss`
        # 5. Update parameters: This is done using the optimizer's `step` function.
        #                       Check the link provided for details.

        # YOUR CODE HERE
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    print('Finished Training')
```

2.3 PyTorch Basics: Activation Function after Final FC layer output

In the `forward` function of the `Net` class, we did not use any non-linear activation function on the final output. Why?

Without any activation function, the output of the forward function are the logits. This was done since the criterion we have used is `BCEWithLogitsLoss`, which applies log and sigmoid to the logits and then computes the binary cross entropy with the target labels. If we had used the sigmoid activation on the final output, we could have used the `BCELoss` loss instead. However, `BCEWithLogitsLoss` is preferred over `BCELoss` because it is numerically more stable.

In [48]:

```
# Check the network stats
summary(net, input_size=(2,), device="cpu")
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                   [-1, 10]              30
            Linear-2                   [-1, 10]             110
            Linear-3                    [-1, 1]              11
================================================================
Total params: 151
Trainable params: 151
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
----------------------------------------------------------------
```

2.4 PyTorch Basics: Verify Number of Trainable Parameters

The `summary` function from `torchsummary` is useful for finding the statistics of network parameters layers and output shapes. Calculate the number of parameters for each layer (W and b separately) by hand and verify your result with the number of parameters shown in the cell above.

How many parameters are there?

- Layer 1 (first hidden layer): Weights [2,10] = 20
- Layer 1 (first hidden layer): biases [1,10] = 10 ...

Total = ?

- Layer 1 (first hidden layer): Weights [2,10] = 20
- Layer 1 (first hidden layer): biases [1,10] = 10
- Layer 2 (second hidden layer): Weights [10,10] = 100
- Layer 2 (second hidden layer): biases [1,10] = 10
- Layer 3 (output): Weights [10,1] = 10
- Layer 3 (output): biases [1,1] = 1

Total = 151

In [49]:

```
# Train the PyTorch network
train_neural_network_pytorch(net, moons_x_train, moons_y_train, optimizer, criterio
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

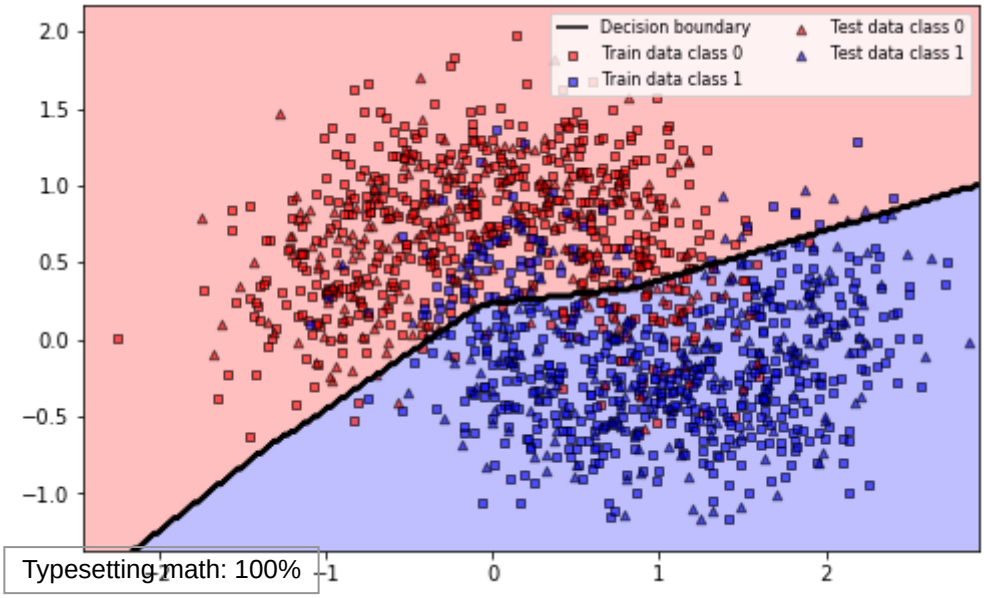
```
Finished Training
```

In [50]:

```
def predict_pytorch(net, X):
    """
    Function for producing network predictions
    """

    net.eval()

    # Make predictions (class 0 or 1) using the learned parameters

    # Computes probabilities using forward propagation, and classifies to 0/1 using
    X = torch.from_numpy(X.astype(np.float32))
    logits = net(X)
    predictions = torch.sigmoid(logits) > 0.5

    return predictions
```

In [51]:

```
# Plot the decision boundary learned by the neural network
fig, ax = plt.subplots(1,1,figsize=(8,5))
plot_decision_boundary_nn(ax,
                          predict_pytorch,
                          net,
                          moons_x_train,
                          moons_y_train,
                          moons_x_test,
                          moons_y_test)
```



In [52]:

```
# Calculate the accuracies on the training and test data
# You should be able to get accuracies around 0.9
train_acc = calc_accuracy(predict_pytorch(net, moons_x_train).data.numpy(), moons_y
test_acc = calc_accuracy(predict_pytorch(net, moons_x_test).data.numpy(), moons_y_t
print(f"Train accuracy: {train_acc:.2f}, Test accuracy: {test_acc:.2f}")

assert train_acc>0.7 and test_acc>0.7, "Rerun with different hyperparameters to get
```

```
Train accuracy: 0.86, Test accuracy: 0.88
```

[go to top]

3 Part 3: Multiclass Classification with PyTorch

Till now, we had been training networks for performing binary classification. Now, we move on to the problem of multiclass classification, where for a given input, the network needs to predict one out of C possible classes. Additionally, till now, we had been training the network using batch gradient descent, where the entire training dataset is used to compute gradients in each iteration. Now, we will use minibatch gradient descent, where a minibatch (smaller subset of the train set) is used in each iteration.

In [53]:

```
from torchvision import datasets, transforms
```

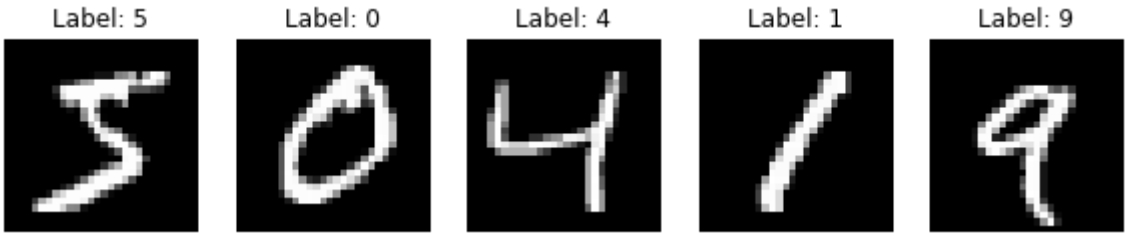
For this part of the notebook, and the next, we will be using the MNIST (https://pytorch.org/vision/stable/datasets.html#mnist) dataset from torchvision.datasets (https://pytorch.org/vision/stable/datasets.html). First, we need to create dataset objects for the training and test data. The first time the next cell is run, the data is downloaded and stored in the `data` directory. But since this notebook is run on the Jupyter-hub, the data is already downloaded and available. Standard transformations are applied to the data tenstors.

In [54]:

```
train_dataset = datasets.MNIST('/data',
                               train=True,
                               download=running_local,
                               transform=transforms.Compose([transforms.ToTensor(),
                                                             transforms.Normalize((
                                                            ]))

test_dataset = datasets.MNIST('/data',
                              train=False,
                              transform=transforms.Compose([transforms.ToTensor(),
                                                            transforms.Normalize((0
                                                           ]))
```

In [55]:

```
# View sample data and targets
fig, ax = plt.subplots(1,5,figsize=(10,2))
for i in range(5):
    ax[i].imshow(train_dataset[i][0].data.numpy().squeeze(0), cmap='gray')
    ax[i].set_title(f"Label: {train_dataset[i][1]}")
    ax[i].axis('off')
```



In [56]:

```
# Define hyperparameters
LEARNING_RATE = 0.001
MOMENTUM = 0.9
NUM_EPOCHS = 1
HIDDEN_SIZE = 100
TRAIN_BATCH_SIZE = 64
TEST_BATCH_SIZE = 64
INPUT_SIZE = 784  # Do not change this
OUTPUT_SIZE = 10  # Do not change this
```

3.1 Multiclass Classification: Using DataLoaders

For training using minibatches, PyTorch provides the DataLoader (https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader) utility which combines a dataset and a sampler, and provides an iterable over the given dataset. Once a data loader is created, it is really easy to fetch a minibatch of data from it in the actual training loop. Complete the missing parts in the code cell below to create the train and test data loaders.

In [57]:

```
# Create the train data loader
#train_loader = torch.utils.data.DataLoader(dataset= ,           # Pass the appropr
#                                           batch_size= ,        # Use the correct
#                                           shuffle= ,           # Shuffling the da
#                                           drop_last= )         # Ignore the last

# Create the test data loader in the same way
# Take care of using the correct dataset and batch size
#test_loader = torch.utils.data.DataLoader(dataset= ,           # Pass the appropr
#                                          batch_size= ,        # Use the correct
#                                          shuffle= ,           # No need to shuff
#                                          drop_last= )         # Do not ignore th

# YOUR CODE HERE
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=TRAIN_BATCH_SIZE,
                                           shuffle=True,
                                           drop_last=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=TEST_BATCH_SIZE,
                                          shuffle=False,
                                          drop_last=False)
```

3.2 Multiclass Classification: Number of Minibatches

Find the number of minibatches in the train and test data loaders.

In [58]:

```
# How many minibatches are in the train and test data loaders?
# Assign to variables `num_train_batches` and `num_test_batches`
# Note that data loaders are iterators

# YOUR CODE HERE
num_train_batches = len(train_loader)
num_test_batches = len(test_loader)

print(f"num_train_batches: {num_train_batches}, num_test_batches: {num_test_batches
```

```
num_train_batches: 937, num_test_batches: 157
```

3.3 Multiclass Classification: Cross Entropy Loss

In [59]:

```
# Instantiate the network and set up the loss function and optimizer
net_mnist = Net(INPUT_SIZE, HIDDEN_SIZE, OUTPUT_SIZE)

criterion = nn.CrossEntropyLoss()  # Be careful, use BCE for binary, CrossEntropy f
optimizer = optim.SGD(net_mnist.parameters(), lr=LEARNING_RATE, momentum=MOMENTUM)
```

In [60]:

```
# Check the parameter statistics
summary(net_mnist, input_size=(INPUT_SIZE,), device="cpu")
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                  [-1, 100]          78,500
            Linear-2                  [-1, 100]          10,100
            Linear-3                   [-1, 10]           1,010
================================================================
Total params: 89,610
Trainable params: 89,610
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.34
Estimated Total Size (MB): 0.35
----------------------------------------------------------------
```

3.4 Multiclass Classification: Training

In the function `train_neural_network_pytorch_minibatch`, minibatch gradient descent is to be used. Complete the missing parts in the function, so that a minibatch of data can be loaded using the `train_loader`. Check here (https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#train-the-network) for a possible way of using a data loader. Once the minibatch is loaded, complete the steps indicated by the comments (same as before).

In [85]:

```
def train_neural_network_pytorch_minibatch(net, train_loader, optimizer, criterion,

    net.train()  # Set the network in training mode

    for epoch in range(num_epochs):
        for batch_idx, (data, target) in enumerate(tqdm(train_loader)):

            # Follow these steps inside the loop here:
            # 1. Zero parameter gradients
            # 2. Forward
            # 3. Compute loss
            # 4. Backward
            # 5. Update step

            # YOUR CODE HERE
            optimizer.zero_grad()
            outputs = net(data)
            loss = criterion(outputs, target)
            loss.backward()
            optimizer.step()
```

In [86]:

```
def calc_accuracy_minibatch(net, data_loader):
    """
    Calculates the overall accuracy by using minibatches
    """
    net.eval()
    correct = 0
    with torch.no_grad():
        for data, target in data_loader:
            output = net(data)
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max l
            correct += pred.eq(target.view_as(pred)).sum().item()

    accuracy = correct/len(data_loader.dataset)
    return accuracy
```

In [87]:

```
train_neural_network_pytorch_minibatch(net_mnist, train_loader, optimizer, criterio
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

In [ ]:

```
# Check the train and test accuracies
train_accuracy = calc_accuracy_minibatch(net_mnist, train_loader)
test_accuracy = calc_accuracy_minibatch(net_mnist, test_loader)
print(f"Train accuracy: {train_accuracy: .2f}, Test accuracy: {test_accuracy: .2f}"

assert train_accuracy>0.8 and test_accuracy>0.8, \
"Rerun with different hyperparameters to get better accuracies"
```

[go to top]

4 Part 4: Convolutional Neural Networks with PyTorch

If you replace the fully connected network with a convolutional neural network (CNN), it will be possible to get a much better test accuracy (for MNIST, only a few epochs of training can result in a test accuracy of above 97%).

Provided below is the definition of a possible CNN for classification which can be used for the MNIST example (there are many ways to construct a network, here we only show a simple example).

In [165]:

```
class CNN(nn.Module):
    def __init__(self, output_size):
        super(CNN, self).__init__()

        # Use a 2D Conv layer with 1 input channel, 32 output channels, filter size
        self.conv1 = nn.Conv2d(1, 32, 3, 1)

        # Use a 2D Conv layer with 32 input channel, 64 output channels, filter siz
        self.conv2 = nn.Conv2d(32, 64, 3, 1)

        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, output_size)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        return x
```

In [166]:

```
# Instantiate the CNN
net_cnn = CNN(output_size=10)

# Check the number of parameters
summary(net_cnn, input_size=(1,28,28), device="cpu")
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 26, 26]             320
            Conv2d-2           [-1, 64, 24, 24]          18,496
            Linear-3                  [-1, 128]       1,179,776
================================================================
Total params: 1,198,592
Trainable params: 1,198,592
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.45
Params size (MB): 4.57
Estimated Total Size (MB): 5.02
----------------------------------------------------------------
```

Due to performance reasons, you are not required to train a CNN in this notebook. Instead, here we provide links to useful online resources which you can consult to build and train your own CNN.

- Tutorial on CNN with PyTorch (https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#train-the-network)
- Accessible illustrations on how CNNs work (https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks) **(highly recommended)**
- CNN layers in PyTorch (https://pytorch.org/docs/stable/nn.html#convolution-layers)