COMP4660/8420 Lab 1
Neural Networks

Part 1: Introduction to Neural Networks

The first part of the lab this week will be reviewing the basics of neural networks.

Q1. Briefly justify how artificial neural networks are biologically inspired.

Artificial neural networks are inspired by the biological brain, and are designed to mimic how learning happens in the brain through a series of interconnected neurons. ANNs are engineered to duplicate the computational principles behind the brain and hence duplicate its functionality, as the brain is evidence that intelligent behaviour is possible through such a structure.

Q2. Below is a diagram of the simplest neural network: a single neuron.

Label the diagram and describe each of the components

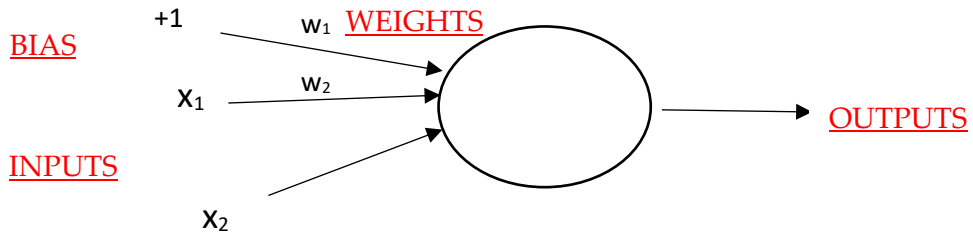
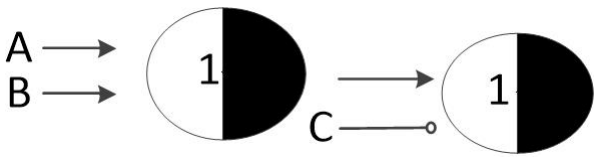


Figure 1. A simple neuron

INPUTS – features of a sample that contribute toward its classification into some category
WEIGHTS - a modification to the input value which determines how much effect that particular feature has on the neuron
BIAS – a value that allows you to shift the activation function to the left or right to produce the desired output, so that the decision hyperplane (where the value is 0.5) does not have to go through the origin
OUTPUTS – the resulting class/result that the ANN has predicted

Q3. Draw two McCulloch and Pitts cells joined together that represent A or B but not C.

First cell has Excitatory weights (arrows) for A and B and a threshold of 1. Second cell has an excitatory weight (arrow) coming from the first cell and an inhibitory weight (line with circle) for C with a threshold of 1.



Part 2: Introduction to PyTorch

During the semester, we will use PyTorch (http://pytorch.org/) as a tool to implement the algorithms we will learn. It is an open source machine learning library for Python.

In the first lab, make yourself familiar with Python and PyTorch.

TASK1

Download the python script "task1.py" and the dataset folder from Wattle, and save them under the same directory.

This python script demonstrates a quick way to build a basic neural network using PyTorch built-in nn package on the iris dataset (For more details about the dataset, please refer to https://archive.ics.uci.edu/ml/datasets/iris). It uses 80% of the original iris data (saved as iris_train.csv) to train a neural network, and uses 20% of the original iris data (saved as iris_test.csv) to test the neural network. The neural network contains one hidden layer with 10 hidden sigmoid neurons. The neural network is trained with Stochastic Gradient Descent (SGD) as an optimiser, and its performance is evaluated using cross-entropy.

In PyTorch, the nn package defines a set of Modules, which are roughly equivalent to neural network layers. A Module receives input Variables and computes output Variables, but may also hold internal state such as Variables containing learnable parameters. The nn package also defines a set of useful loss functions that are commonly used when training neural networks.

Execute the script, i.e. type the following command in your terminal:

```
python3 task1.py
```

Q1. Report on the error and the accuracy of the training and testing

Q2. Report on the confusion matrices.

Now adjust the number of hidden neurons to a value of your choice, and record the error and accuracy of the change. Adjust the number of neurons again, making sure that you have tried values both above and below the initial value of 10.

Q3. Does adding more neurons always import results? Not always

TASK2

While the nn package is useful for quickly building up a neural network, the real power comes when you can specify models that are more complex than a sequence of existing Modules. In order words, you can perform in-depth customization on your neural networks, such as adding multiple hidden layers of neurons, changing the activation functions or changing the learning algorithm. As you will be expected to write your own code for assignments, we will now introduce you to building a neural network with customised nn modules.

Download the python script "task2.py" and the dataset folder from Wattle, and save them under the same directory.

To execute the script, you can i.e. type the following command in your terminal:

```
python3 task2.py
```

The python script "task2.py" is an alternative implementation of "task1.py", but it demonstrates a way to define your own neural network by subclassing nn.Module and defining a forward function which receives input Variables and produces output Variables. Similar to the previous script, it uses 80% of the original iris data (saved as iris_train.csv) to train a neural network, and uses 20% of the original iris data (saved as iris_test.csv) to test the neural network. The neural network contains one hidden layer with 10 hidden sigmoid neurons. The neural network is trained with Stochastic Gradient Descent (SGD) as an optimiser, and its performance is evaluated using cross-entropy.

To help you better understand this, a step-by-step explanation is provided below:

Step1. Load and setup training dataset

First load the dataset you are going to use for training your neural network. To do this, read the dataset file by providing the path of the dataset, i.e.

```
# load training data
data_train = pd.read_csv('dataset/iris_train.csv')
```

Take a look at the training dataset by printing it out, i.e.

```
print(data_train)
```

Q4. How many features (inputs) are there? 4

Q5. How many classes (outputs) are there? 3

Q6. How many data instances are there? 120

You might notice that the loaded data isn't quite what you would have expected so you may want to massage the data. For example, in this case, the target values are string but we need numeric values for training a neural network. We would also like to split features and targets out. Therefore, for this case, the following lines are added:

```
# convert string target values to numeric values
#   class 0: Iris-setosa
#   class 1: Iris-versicolor
#   class 2: Iris-virginica
data_train.at[data_train['species'] == 'Iris-setosa', ['species']] = 0
data_train.at[data_train['species'] == 'Iris-versicolor', ['species']] = 1
data_train.at[data_train['species'] == 'Iris-virginica', ['species']] = 2
data_train = data_train.apply(pd.to_numeric)

# convert pandas dataframe to array
# the first 4 columns are features, the last column is target
data_train_array = data_train.as_matrix()

# split x (features) and y (targets)
x_array = data_train_array[: , : 4]
```

```
y_array = data_train_array[: , 4]
```

Finally, as PyTorch only trains neural networks on Variables, we need to create Tensors to hold inputs and outputs, and wrap them in Variables, as follow:

```
# create Tensors to hold inputs and outputs, and wrap them in Variables
X = Variable(torch.Tensor(x_array).float())
Y = Variable(torch.Tensor(y_array).long())
```

Step2. Define and train a neural network

Now create a neural network. To create a single hidden layer neural network with 10 sigmoid hidden neurons, we can define a class `TwoLayerNet` by subclassing `torch.nn.Module`, and define our own forward pass function as follow:

```
# define a customised neural network structure
class TwoLayerNet(torch.nn.Module):
    def __init__(self, n_input, n_hidden, n_output):
        super(TwoLayerNet, self).__init__()
        # define linear hidden layer output
        self.hidden = torch.nn.Linear(n_input, n_hidden)
        # define linear output layer output
        self.out= torch.nn.Linear(n_hidden, n_output)

    def forward(self, x):
        # In the forward function, we define the process of performing
        # forward pass, that is to accept a Variable of input data, x,
        # and return a Variable of output data, y_pred

        # get hidden layer input
        h_input = self.hidden(x)
        # define activation function for hidden layer
        h_output = F.sigmoid(h_input)
        # get hidden layer input
        y_pred = self.out(h_output)

        return y_pred

# define the number of neurons for each layer
input_neurons = 4
hidden_neurons = 10
output_neurons = 3

# define a neural network using the customised structure
net = TwoLayerNet(input_neurons, hidden_neurons, output_neurons)
```

Now we are going to train the neural network using our training data.

```
# define loss function, learning rate, optimiser and training epoch
loss_func = torch.nn.CrossEntropyLoss()
learning_rate = 0.01
optimiser = torch.optim.SGD(net.parameters(), lr=learning_rate)
num_epoch = 500

# train a neural network
for epoch in range(num_epoch):
    # train a neural network
    Y_pred = net(X)

    # compute loss
    loss = loss_func(Y_pred, Y)

    # clear the gradients before running the backward pass
    net.zero_grad()
```

```
    # perform backward pass
    loss.backward()
```

Step3. Load and setup testing dataset

Similar to step1, now we can load the test data and get it ready for testing our neural network.

```
# load testing data
data_test = pd.read_csv('dataset/iris_test.csv')

# convert string target values to numeric values
#   class 0: Iris-setosa
#   class 1: Iris-versicolor
#   class 2: Iris-virginica
data_test.at[data_test ['species'] == 'Iris-setosa', ['species']] = 0
data_test.at[data_test ['species'] == 'Iris-versicolor', ['species']] = 1
data_test.at[data_test['species'] == 'Iris-virginica', ['species']] = 2
data_test = data_test.apply(pd.to_numeric)

# convert pandas dataframe to array
# the first 4 columns are features, the last column is target
data_test_array = data_test.as_matrix()

# split x (features) and y (targets)
x_test_array = data_test_array[: , : 4]
y_test_array = data_test_array[: , 4]

# create Tensors to hold inputs and outputs, and wrap them in Variables
X_test = Variable(torch.Tensor(x_test_array).float())
Y_test = Variable(torch.Tensor(y_test_array).long())
```

Step4. Test the neural network

Now, test the neural network with the testing dataset and calculate the error.

```
# test the neural network using testing data
# please note that Y_pred_test contains three columns, where the index of the maximum
column value indicates the class of the instance
Y_pred_test = net(X)


# get prediction
_, predicted_test = torch.max(F.softmx(Y_pred_test), 1)


# calculate accuracy
total_test = predicted_test.size(0)
correct_test = sum(predicted_Test.data.numpy() == Y.test_data.numpy())

print('Testing Accuracy: %.2f %%' % (100 * correct_test / total_test))
```

Q7. What is the accuracy of the classification?

Once again, try different numbers of hidden neurons and report back on the effect it has on the error and accuracy. Run your neural network several times using different parameters.

Q8. What was the best accuracy you were able to achieve? What were the parameters of the neural network?

Q9. Run the neural network again using the same parameters as your best result. Did you receive the exact same result again? Why might it different?

Different initialisation parameters (weights) and different division of data into

training/testing sets.

Extending your neural network

If you have finished the other tasks, try extending the functionality of the neural network and playing around with the parameters, such as the number of hidden neurons and the number of hidden layers. You can try changing the activation functions to others to see what effect this has on the output and error. You can also look into the other types of neural networks and learning algorithms that PyTorch has available.

Want to learn more about PyTorch?

If you would like to learn more about PyTorch, look at the tutorials:
http://pytorch.org/tutorials/