COMP4660/8420 Lab 2
PyTorch on Classification

Q1. Calculate the output from the neuron below using the following activation functions.

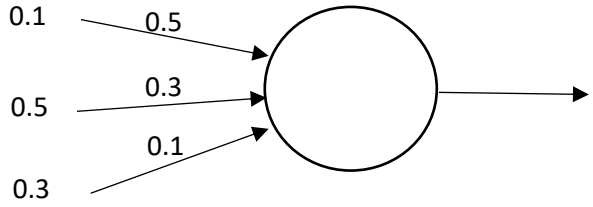


Figure 2. A simple neuron with inputs and weights labelled

a. Linear: $purelin(n) = n$

b. Hard limit: $hardlim(n) = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases}$

c. Sigmoid: $sig(n) = \dfrac{1}{1 + e^{-n}}$

d. Tan-sigmoid: $tansig(n) = \dfrac{2}{1 + e^{-2n}} - 1$

Q2. A neural network is a collection of neurons. An example of a multilayer neural network is shown in Figure 3.

Why is the middle layer referred to as the hidden layer?

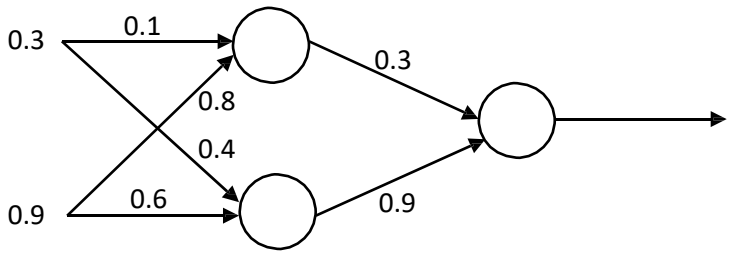


Figure 3. A multilayer feedforward neural network

Q3. Calculate the output from the neural network shown in Figure 3 using the sigmoid function as the activation function for all the neurons.

Task 1: A basic classification task in PyTorch

Build a neural network for a classification task. The dataset we are using is the Glass Identification data set located at http://archive.ics.uci.edu/ml/datasets/Glass+Identification

Q1. How many instances are in this data set?

Q2. How many attributes (features) are there?

Q3. What was this data set originally used for?

Q4. What is the output attribute? How many output values are there?

Download the data set "glass.data" and save it as a CSV file, i.e. "glass.csv".

We will begin by simplifying the dataset so that it is only two classes. Make a copy of the original file and name it "glass_binary.csv".

Q5. Line 21 removes all the data in the column labelled "Id number" . Why do you think we should not use this data to build a model?

Q6. What does line 27 do and why? Hint: it sets all the values in the final column to either 0 or 1.

Q7. Implement a neural network that classifies the data set based on whether the type of glass is a Window glass or a Non-window glass by filling in the sections marked TODO. The inputs for the neural network will be the refractive index and measurements for sodium, magnesium, aluminium, silicon, potassium, calcium, barium and iron.

Hint: Please refer to task2.py in Lab 1 or *Appendix. Introduction to PyTorch Basics* if you don't know how to start.

Advanced Steps:

1. We encourage you to experiment with different ways of accomplishing the task.
2. Explore the normalisation and pre-processing techniques discussed in the lectures and investigate its impact on the performance of the classification.
3. Investigate the performance of the neural network classification by changing the various characteristics of the neural network such as:
    a. Number of neurons in each layer
    b. Number of layers
    c. Number of epochs
    d. Learning rate

Task 2: An advanced classification task in PyTorch

Now we will work with a more complicated classification task, the original glass data set.

Load the unmodified data set "glass.csv" into PyTorch and perform the same classification task as above.

Q8. How many classes are you predicting now?

Q9. How will you represent these classes and how will you calculate the error of classification?

Appendix. Introduction to PyTorch Basics

Before delving deeper into the building neural networks with PyTorch, it is helpful to know some basic manipulations of PyTorch elements. The following examples in task 1 are designed to give you a very brief introduction to operations in PyTorch and will be sufficient for the course, but if you are interested in increasing your knowledge we encourage you to look at the PyTorch tutorials and play around further:
http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

All the following examples in Appendix are also provided in "pytorch_basics.py". Please download "pytorch_basics.py" and play around with it. To execute the script, navigate to your directory and type the following command in your terminal (if you are using one of the lab machines in CSIT, please use Anaconda3 Shell as your terminal) :

```
python3 pytorch_basics.py
```

What is PyTorch?

It's a Python based scientific computing package targeted at two sets of audiences:

- A replacement for NumPy to use the power of GPUs
- a deep learning research platform that provides maximum flexibility and speed

To use PyTorch, you need to import the library by writing:

```
import torch
```

1. Tensors

1.1 What are Tensors?

Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

Construct a randomly initialized matrix:

```
# create a randomly initialized matrix
x = torch.rand(5, 3)
print(x)
# get matrix size
print(x.size())
```

1.2 Tensors operations

Tensors support basic operations such as addition, subtraction, multiplication, and division. There are multiply multiple ways of specifying operations. In the following example, we will take a look at the addition operation.

```
# Addition: method 1
x = torch.rand(5, 3)
y = torach.rand(5, 3)
print(x + y)

# Addition: method 2
x = torch.rand(5, 3)
y = torch.rand(5, 3)
print(torch.add(x, y))

# Addition: in-place
x = torch.rand(5, 3)
y = torch.rand(5, 3)
print(y)
y = add_(x)
print(y)
```

Note 1: Any operation that mutates a tensor in-place is post-fixed with an _. For example: `x.copy_(y)`, `x.t_()`, will change `x`.

Indexing is also NumPy-like.

```
# access the second column
print(x[:, 1])

# access the first row
print(x[0, :])
```

1.3 Tensors <-> NumPy arrays

Tensors can be converted to NumPy's ndarrays,

```
# import numpy library
import numpy as np
# create a randomly initialized tensor matrix
x = torch.rand(5, 3)
# convert tensors to numpy array
y = x.numpy()
print(y)
```

and can be formed by NumPy's ndarrays.

```
# create a numpy array
x = np.array([3,4], [3,5])
# convert numpy array to tensor
y = torch.from_numpy(x)
print(y)
```

Note 2: The Torch Tensor and NumPy array will share their underlying memory locations, so changing one will change the other.

```
# create a numpy array
a = torch.ones(5)
b = a.numpy()
# all elements in a add 1
a.add_(1)
```

```
# b will also be updated
print(a)
print(b)
```

2. Autograd: automatic differentiation

Central to all neural networks in PyTorch is the `autograd` package. It provides automatic differentiation for all operations on Tensors.

Autograd is reverse automatic differentiation system. Conceptually, `autograd` records a graph recording all of the operations that created the data as you execute operations, giving you a directed acyclic graph whose leaves are the input variables and roots are the output variables. By tracing this graph from roots to leaves, you can automatically compute the gradients using the chain rule.

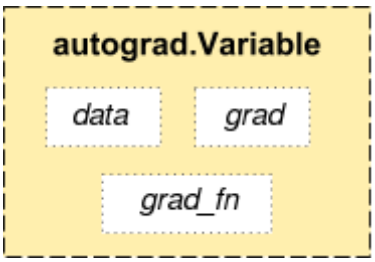
Internally, `autograd` represents this graph as a graph of Function objects (really expressions), which can be `apply()` ed to compute the result of evaluating the graph. When computing the forwards pass, `autograd` simultaneously performs the requested computations and builds up a graph representing the function that computes the gradient (the `.grad_fn` attribute of each `Variable` is an entry point into this graph). When the forwards pass is completed, we evaluate this graph in the backwards pass to compute the gradients.

2.1 Variable (Deprecated)

This class has been deprecated as of version 0.4. All functionalities of the class Variable are now supported directly by the class Tensor. The below section is useful to understand what Variable does (if you are using an old Pytorch version) and what Tensor can do now (the functions are directly applicable to Tensor now).

`autograd.Variables` is the central class of the package. It wraps a Tensor, and supports nearly all operations defined on it. Once you finish your computation, you can call `.backward()` and have all the gradients computed automatically.

There are three basic attributes in a Variable:



Variable

You can access the raw tensor through the `.data` attribute, the gradient with respect to this variable is accumulated into `.grad`. Each variable has a `.grad_fn` attribute which references a `Function` that has created the `Variable`, except for Variables created by the user – their `grad_fn is None`.

If you want to compute the derivatives, you can call `.backward()` on a Variable. If Variable is a scalar (i.e. it holds a one element data), you don't need to specify any arguments to

`backward()`, however if it has more elements, you need to specify a `gradient` argument that is a tensor of matching shape.

```
# Create variables
x = Variable(torch.Tensor([1]), requires_grad=True)
w = Variable(torch.Tensor([2]), requires_grad=True)
b = Variable(torch.Tensor([3]), requires_grad=True)

# Define a function
y = w * x + b            # y = 2 * x + 3

# Compute gradients.
y.backward()             # equal to y.backward(torch.Tensor([1.0]))

# Print out the gradients.
print('dy/dx: {}'.format(x.grad.data))   # x.grad = 2
print('dy/dw: {}'.format(w.grad.data))   # w.grad = 1
print('dy/db: {}'.format(b.grad.data))   # b.grad = 1
```

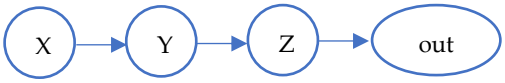
2.2 Gradients

Let's now look at the following example to understand how gradients are calculated.

```
import torch
from torch.autograd import Variable

x=torch.Tensor([[1.,2.,3.],[4.,5.,6.]])
x=Variable(x,requires_grad=True)
y=x+2
z=y*y*3
out=z.mean()
```

An equivalent computation graph of the above code is:



Now, if you follow the computation direction by using its `.grad_fn` attribute, i.e. by printing out the `.grad_fn` of each variable as follows,

```
print(x.grad_fn)    # None
print(y.grad_fn)    # <AddBackward0 object at 0x00000219A888E860>
print(z.grad_fn)    # <MulBackward0 object at 0x00000219A888E860>
print(out.grad_fn)  # <MeanBackward1 object at 0x00000219A888E860>
```

you will see a graph of computations that looks like this:

```
x -> add -> multiply -> mean -> out
```

To see the gradient of `out` with respect to `x`, $\frac{\partial out}{\partial x}$, we can do

```
out.backward()      # equivalent to out.backward(torch.Tensor([1]))
print(x.grad)
```

Similarly, to see the gradient of `z` with respect to `x`, $\frac{\partial z}{\partial x}$, we can do

```
# need to specify a gradient argument that is a tensor of matching shape.
z.backward(torch.Tensor([[1, 1, 1], [1, 1, 1]]))
print(x.grad)
```

3. Neural Networks

Neural networks can be constructed using the `torch.nn` packages. `nn` depends on `autograd` to define models and differentiate them. An `nn.Module` contains layers, and a method `forward(input)` that returns the `output`.

A typical training procedure for a neural network is as follows:

- Define the neural network that has some learnable parameters (or weights)
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss (how far is the output from being correct)
- Propagate gradients back into the network's parameters
- Update the weights of the network, typically using a simple update rule:
  `weight = weight - learning_rate * gradient`

3.1 Define the network

A template for defining a neural network is:

```
import torch.nn as nn
import torch.nn.functional as F

# a template for defining a neural network
Class net_name(nn.Module):
    def __init__(self):
        super(net_name, self).__init()
        # add layers here
        self.layer1 =
            nn.Linear(n_input, n_hidden)  #change nn.Linear if it is not linear
        self.layer2 = …
        # more layers…

    # define the process of performing forward pass,
    # that is how to return a Variable of output data
    # from a Variable of input data x
    def forward(self, x):
        x = F.some_function1(x)        # calling some functions in torch.nn.functional
        x = F.some_function2(x)        # calling some functions in torch.nn.functional
        x = self.layer1(x)             # apply pre-define layer1
        … …
        return x

# define a neural network using the customised structure
net = Net()
```

You just need to define the `forward` function, and the `backward` function (where gradients are computed) is automatically defined for you using `autograd`. You can use any of the Tensor operations in the `forward` function. The input to the forward is an `autograd.Variable`, and so is the output.

```
# perform forward pass to get the actual output
output = net(input)
```

The learnable parameters of a model are returned by `net.parameters`().

Here as an example, we define a simple neural network with one hidden layer using the

above template:

```
# define a simple neural network with one sigmoid hidden layer
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 output layer output
        y_pred = self.out(h_output)

        return y_pred

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

3.2 Loss Function

A loss function takes the (output, target) pair of inputs, and computes a value that estimates how far away the output is from the target.

There are several different loss functions under the `nn` package . A simple loss is: `nn.MSELoss` which computes the mean-squared error between the input and the target.

For example:

```
# perform forward pass to get the actual output
output = net(input)

# define loss function
loss_func = nn.MSELoss()

# compute loss
loss = loss_func (output, target)
print(loss)
```

So, when we call `loss.backward()`, the whole computational graph is differentiated with respect to the loss, and all Variables in the graph will have their `.grad` Variable accumulated with the gradient.

3.3 Back propagation

To backpropagate the error all we have to do is to `loss.backward()`. You need to clear the existing gradients though, else gradients will be accumulated to existing gradients.

```
# clear gradient buffers of all parameters
```

```
net.zero_grad()

# perform backward pass: compute gradients of the loss with respect to
# all the learnable parameters of the model.
loss.backward()
```

3.4 Update the weights

The simplest update rule used in practice is the Stochastic Gradient Descent (SGD):

```
weight = weight - learning_rate * gradient
```

We can implement this using simple python code:

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

However, you may want to use various different update rules such as SGD, Nesterov-SGD, Adam, RMSProp, etc. To enable this, we can use a small package `torch.optim` that implements all these methods. Using it is very simple:

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = loss_func(output, target)
loss.backward()
optimizer.step()    # does the update
```

3.5 Save and load a model

Sometimes, you may want to save the trained model and load it later. There are two approaches for this.

The first (recommended) saves and loads only the model parameters:

```
torch.save(the_model.state_dict(), PATH)
```

Then later:

```
the_model = TheModelClass(*args, **kwargs)
the_model.load_state_dict(torch.load(PATH))
```

The second saves and loads the entire model:

```
torch.save(the_model, PATH)
```

Then later:

```
the_model = torch.load(PATH)
```