Sandheep_ComputerVision_LabAssignment4.ipynb

March 20, 2022

1 Lab Assignment 4

Computer Vision - Term 5, 2022

Instructor: Dr. Saumya Jetly TA: Ribhu Lahiri

Deadline: Wednesday, 23 March 2022 11:59 pm

Submission form link: https://forms.gle/B9m2khtKNStHLCLL8

Total points: 5

[1]:
```
# Imports
import math
import torch
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

1.0.1 Task 1: Creating the single layer perceptron (3 points)

In this lab we will focus on the perceptron, and how it can be used to model logic gates. Further, this same idea can be extended further due to perceptrons being a universal function approximator

Implement the `sigmoid` function *(1 point)*

$(x) = \frac{1}{1+e^{-x}}$

[2]:
```
def sigmoid(x):
    '''
     Applies the sigmoid function to the given input

     Parameters
     ----------
     x: torch.Tensor
         Input array/tensor

     Returns
     -------
     out: torch.Tensor
         Tensor after applying sigmoid function to it
```

```
    '''
    return 1/(torch.exp(x*-1)+1)
```

Implement the perceptron function *(1 point)*

$y' = x \bullet W^t + b$

[3]:
```
def perceptron(inputs, weights, bias):
    '''
      Defines the single layer perceptron model
    '''
    return inputs.mm(torch.transpose(weights,0,1)).clamp(min=0).add(bias)
```

Implement the binary_cross_entropy function *(1 point)*

$\text{Loss} = -\frac{1}{\text{N}} \sum_{i=1}^{\text{N}} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i)$

[4]:
```
def binary_cross_entropy(preds, targets):
    '''
      Applies binary cross entropy given predictions and ground truth
    '''
    import math
    loss=torch.tensor(0)
    temp=[]
    if len(preds)==len(targets):
        for i in range(len(preds)):
          if targets[i]==0:
            loss = torch.add(loss,torch.log2(1-preds[i]))
          else:
            loss = torch.add ( loss, torch.log2(preds[i]))
        loss = torch.multiply(torch.div(loss,len(preds)),-1)
    return loss
```

1.1 AND Gate

[15]:
```
and_data = np.array([[0,0,0],
                     [0,1,0],
                     [1,0,0],
                     [1,1,1]])

# Creating a PyTorch tensor
and_data = torch.Tensor(and_data)
```

[16]:
```
# Same slicing as np arrays
X = and_data[:,:-1]
y = and_data[:,-1:]
```

[17]:
```
W = torch.randn((1,2), requires_grad=True)
b = torch.randn((1,1), requires_grad=True)
```

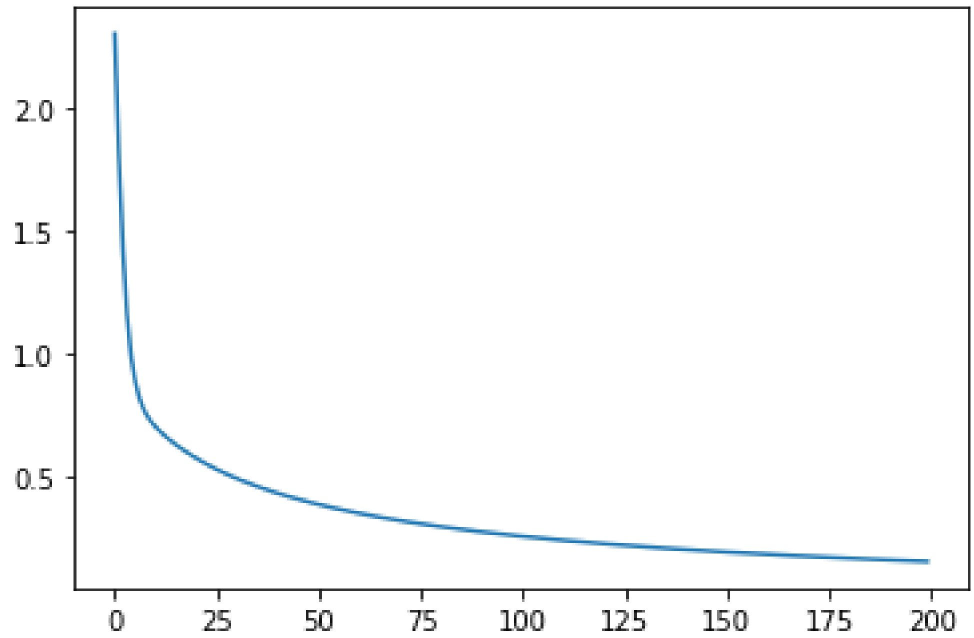
Create the training loop *(1 point)*

[18]:
```
n_epochs = 200
lr = 5e-1
losses = []
for _ in range(n_epochs):
  # Define the Training Loop here
  #
  # Get predictions
  pred = sigmoid(perceptron(X, W, b))

  # Calculate Loss
  loss = binary_cross_entropy(pred, y)

  # Do a backward step (to calculate gradients)
  loss.backward()

  # Update Weights
  with torch.no_grad():
    W -= lr * W.grad
    b -= lr * b.grad
    W.grad.zero_()
    b.grad.zero_()

  # Append Loss
  losses.append(loss)
```

[19]:
```
losses = [loss.detach().numpy()[0] for loss in losses]
```

[20]:
```
plt.plot(losses)
```

[20]:
```
[<matplotlib.lines.Line2D at 0x7f0930f84350>]
```



[21]:
```
with torch.no_grad():
  print((perceptron(X, W, b) > 0.5).int())
```

```
tensor([[0],
        [0],
        [0],
        [1]], dtype=torch.int32)
```

1.2 OR Gate

[22]:
```
or_data = np.array([[0,0,0],
                     [0,1,1],
                     [1,0,1],
                     [1,1,1]])

# Creating a PyTorch tensor
or_data = torch.Tensor(or_data)
```

[23]:
```
# Same slicing as np arrays
X = or_data[:,:-1]
y = or_data[:,-1:]
```

[24]:
```
W = torch.randn((1,2), requires_grad=True)
b = torch.randn((1,1), requires_grad=True)
```

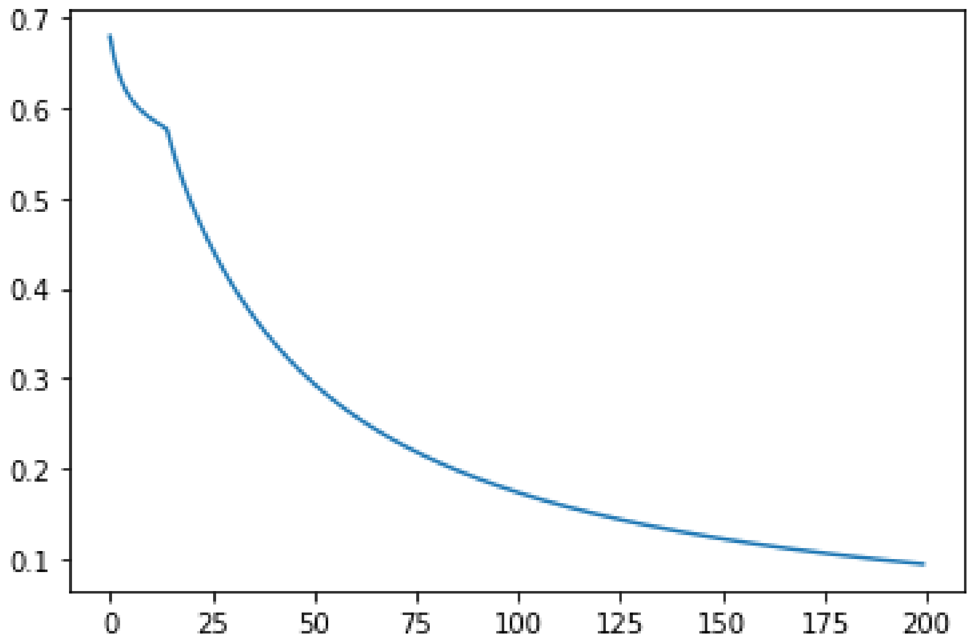
Reuse the training loop

[25]:
```
n_epochs = 200
lr = 5e-1
losses = []
for _ in range(n_epochs):
  # Define the Training Loop here
  #
  # Get predictions
  pred = sigmoid(perceptron(X, W, b))

  # Calculate Loss
  loss = binary_cross_entropy(pred, y)

  # Do a backward step (to calculate gradients)
  loss.backward()

  # Update Weights
  with torch.no_grad():
    W -= lr * W.grad
    b -= lr * b.grad
    W.grad.zero_()
    b.grad.zero_()

  # Append Loss
  losses.append(loss)
```

[27]:
```
losses = [loss.detach().numpy()[0] for loss in losses]
```

[28]:
```
plt.plot(losses)
```

[28]:
```
[<matplotlib.lines.Line2D at 0x7f0930cbc390>]
```



[29]:
```
with torch.no_grad():
  print((perceptron(X, W, b) > 0.5).int())
```

```
tensor([[0],
        [1],
        [1],
        [1]], dtype=torch.int32)
```

1.3 XOR Gate

[63]:
```
xor_data = np.array([[0,0,0],
                     [0,1,1],
                     [1,0,1],
                     [1,1,0]])

# Creating a PyTorch tensor
xor_data = torch.Tensor(xor_data)
```

[64]:
```
# Same slicing as np arrays
X = xor_data[:,:-1]
y = xor_data[:,-1:]
```

[51]:
```
W = torch.randn((1,2), requires_grad=True)
b = torch.randn((1,1), requires_grad=True)
```

Reuse the training loop

[52]:

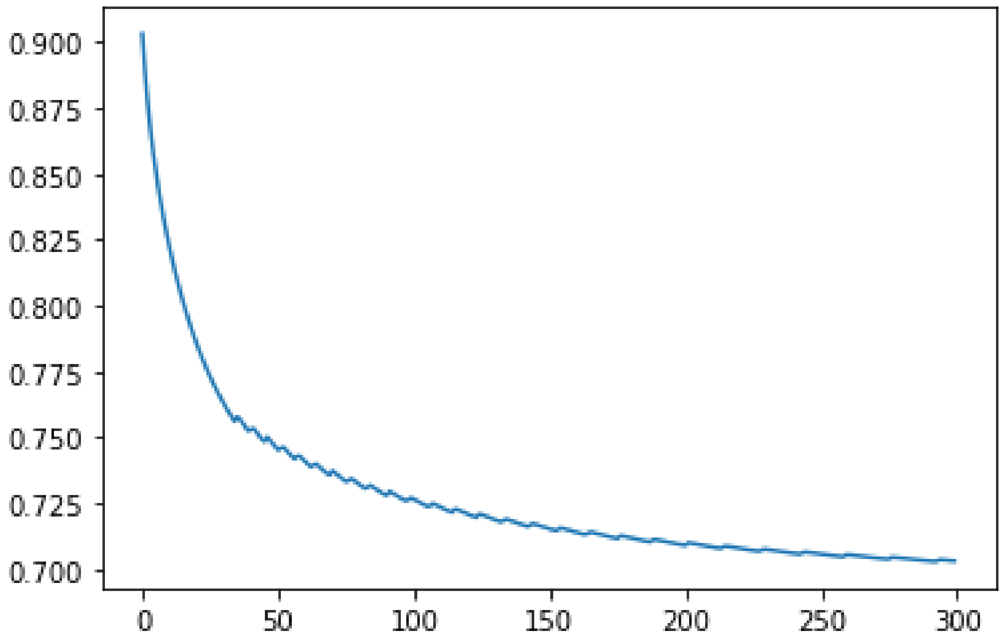
```
n_epochs = 300
lr = 5e-1
losses = []
for _ in range(n_epochs):
  # Define the Training Loop here
  #
  # Get predictions
  pred = sigmoid(perceptron(X, W, b))

  # Calculate Loss
  loss = binary_cross_entropy(pred, y)

  # Do a backward step (to calculate gradients)
  loss.backward()

  # Update Weights
  with torch.no_grad():
    W -= lr * W.grad
    b -= lr * b.grad
    W.grad.zero_()
    b.grad.zero_()

  # Append Loss
  losses.append(loss)
```

[53]:

```
losses = [loss.detach().numpy()[0] for loss in losses]
plt.plot(losses)
```

[53]:

```
[<matplotlib.lines.Line2D at 0x7f0930a96c10>]
```



[54]:
```
with torch.no_grad():
  print((perceptron(X, W, b) > 0.5).int())
```

```
tensor([[0],
        [0],
        [1],
        [0]], dtype=torch.int32)
```

1.4 Need for MLP

As seen above, we are unable to model the XOR gate using a single layer perceptron, so we need to add a hidden layer.

[119]:
```
# Same slicing as np arrays
X = xor_data[:,:-1]
y = xor_data[:,-1:]
```

[120]:
```
W1 = torch.randn((10,2), requires_grad=True)
W2 = torch.randn((1,10), requires_grad=True)
b1 = torch.randn((1,10), requires_grad=True)
b2 = torch.randn((1,1), requires_grad=True)
```

Implement the `mlp` function *(1 point)*

[121]:
```
def mlp(inputs, W1, W2, b1, b2):
  '''
    Defines the multi-layer perceptron model
```

```
    Note: Only 1 hidden layer
  '''
  layer1 = inputs.mm(torch.transpose(W1,0,1)).clamp(min=0).add(b1)
  layer1 = sigmoid(layer1)
  hidden_layer = layer1.mm(torch.transpose(W2,0,1)).clamp(min=0).add(b2)
  output = sigmoid ( hidden_layer)
  return output
```

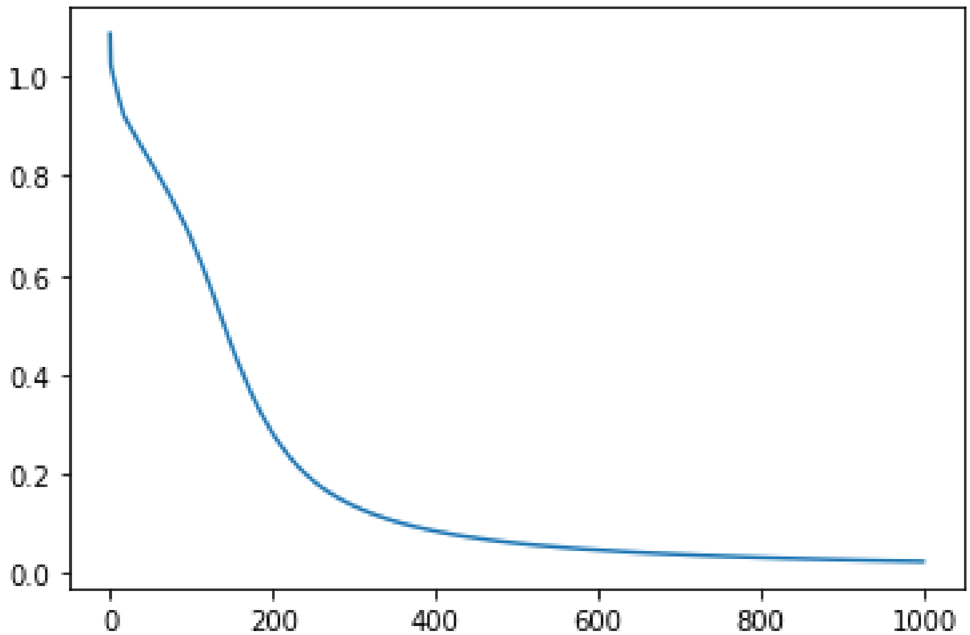
Reuse the training loop

NOTE: It will require slight modification due to the hidden layer

[122]:
```
mlpn_epochs = 100
lr = 5e-1
losses = []
for _ in range(n_epochs):
  # Define the Training Loop here
  #
  # Get predictions
  pred = mlp(X, W1, W2, b1, b2)

  # Calculate Loss
  loss = binary_cross_entropy(pred, y)

  # Do a backward step (to calculate gradients)
  loss.backward()

  # Update Weights
  with torch.no_grad():
    W1 -= lr * W1.grad
    b1 -= lr * b1.grad
    W2 -= lr * W2.grad
    b2 -= lr * b2.grad

    W1.grad.zero_()
    b1.grad.zero_()
    W2.grad.zero_()
    b2.grad.zero_()

  # Append Loss
  losses.append(loss)
```

[125]:
```
losses = [loss.detach().numpy()[0] for loss in losses]
plt.plot(losses)
```

[125]: [<matplotlib.lines.Line2D at 0x7f092e7bd990>]



[126]:
```
with torch.no_grad():
  print((mlp(X, W1, W2, b1, b2) > 0.5).int())
```

```
tensor([[0],
        [1],
        [1],
        [0]], dtype=torch.int32)
```