CSCE4613 – Artificial Intelligence
Spring 2025

Assignment 4: Neural Networks

Out Date: March 10, 2025
Due Date: **March. 21, 2025**

Instructions:

- **Written Format & Template:** Students can use either Google Doc or Latex
- Write your full name, email address, and student ID in the report.
- Write the number of "Late Days" the student has used in the report.
- Submission via BlackBoard
- **Policy:** Can be solved in groups (two students per group at max) & review the late-day policy.
- **Submissions:** Your submission should be a PDF for the written section & a zip file with your source code implementation.
    - **For more information, please visit the course website for the homework policy.**

In this assignment, you will study the neural network and backpropagation method. This assignment requires you to implement PyTorch. You have two options about the machine for this assignment: (1) using your computer if you think it is good enough, or (2) using Google Colab with GPU support (it's free). If you choose the first option, you must ensure you have installed **torch**, **torchvision**, **scikit-learn**, and **matplotlib** libraries. For those who use the second option (Google Colab), you don't need to install anything, Google Colab already installs these libraries for you.

Codebase organization: for those who use the first option, the code base is stored in the folder named ***Scripts***. For those who use the second option, you upload the file ***Homework4.ipynb*** in the folder ***GoogleColab*** to Google Colab.

1. **Binary Network**

   In this section, we will configure the network and learn hyper-parameters for the binary network. There are four types of binary operators: **and**, **or**, **xor**, and **nor**. The truth table for these four operators is given below.

x	y	x and y	x or y	x xor y	x nor y
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	1	0	0

   The data generator is already implemented for you as follows.

```
def generate_data(operator = "AND"):
  assert operator in ["AND", "OR", "XOR", "NOR"], "%s operator is
not valid" % operator
  data = []
  label = []
  for i in range(2):
    for j in range(2):
      data.append([i, j])
      if operator == "AND":
        label.append(i & j)
      elif operator == "OR":
        label.append(i | j)
      elif operator == "XOR":
        label.append(i ^ j)
      else:
        label.append(not (i | j))
  data = torch.as_tensor(data, dtype = torch.float32)
  label = torch.as_tensor(label, dtype = torch.float32)
  return data, label
```

Now, we define the neural network to perform the binary operator in the BinaryNetwork class.

```
class BinaryNetwork(nn.Module):
  def __init__(self):
    # YOUR CODE HERE

  def forward(self, x):
    # x has the size of (batch size x 2)
    # YOUR CODE HERE
```

Then, we will define the training and evaluation framework.

```
model = BinaryNetwork()
model.train()
print(model)
operator = "XOR"
inputs, labels = generate_data(operator = operator)

n_iters = 1000
learning_rate = 0.1

optim = torch.optim.SGD(params = model.parameters(), lr =
learning_rate, momentum=0.9)


for i in range(1, n_iters + 1):

  # WRITE YOUR CODE TO COMPUTE OUTPUTS, LOSS, ACCURACY, AND
OPTIMIZE MODEL
  # outptus = ???
  # loss = ???
  # accuracy = ???
  # optimize the model
  loss = 0.0
  accuracy = 0.0

  if i % 5 == 0:
    print("[%d/%d]. Loss: %0.4f. Accuracy: %0.2f" % (i, n_iters,
loss, accuracy))
```

```
model.eval()
# WRITE YOUR CODE TO CALCULATE THE FINAL ACCURACY
# accuracy = ???
accuracy = 0.0
print("Final Accuracy: %0.2f" % (accuracy))

torch.save(model.state_dict(), "%s_Network.pth" % operator)
  # model.load_state_dict(torch.load("%s_Network.pth" % operator))
# Load model in the next time you use
```

The code above defines the training framework for the binary network. First, we get the data from the data generator. Second, we define the optimizer, specifically, we use the Stochastic Gradient Descent (SGD). This code can configure the learning rate and the number of training steps. Next, we loop and optimize the network with respect to given data, a model, and hyper-parameters. Finally, we report the final accuracy and save the model. If you run the code successfully, you will receive the following output.

a. You need to design and implement the BinaryNetwork class for the **AND** operator. Then, you need to write the code to complete the training and evaluation framework.

b. You need to perform the training and report the training losses in the curve graph and the final accuracy. To create a training loss curve graph, you can use the plot function of matplotlib library.

   ```
   import matplotlib.pyplot as plt
   # losses = [ ] # log of training losses
   plt.plot(losses)
   ```

   You are free to define your network and hyper-parameter so that the final model and accuracy are as optimal as possible.

c. Design and train the binary network for **OR** operator. Report the training losses in the curve graph and the final accuracy.

d. Design and train the binary network for **XOR** operator. Report the training losses in the curve graph and the final accuracy.

e. Design and train the binary network for **NOR** operator. Report the training losses in the curve graph and the final accuracy.

2. Digit Classification

The MNIST (Lecun et al, 1998) handwritten digit classification problem is a standard dataset used in computer vision and deep learning. Although the dataset is effectively solved, it can be used as the basis for learning and practicing how to develop, evaluate, and use neural networks for digit classification from scratch. The dataset has 60,000 training samples and 10,000 testing samples at a resolution of 28x28. Both training and testing samples have already been manually annotated.

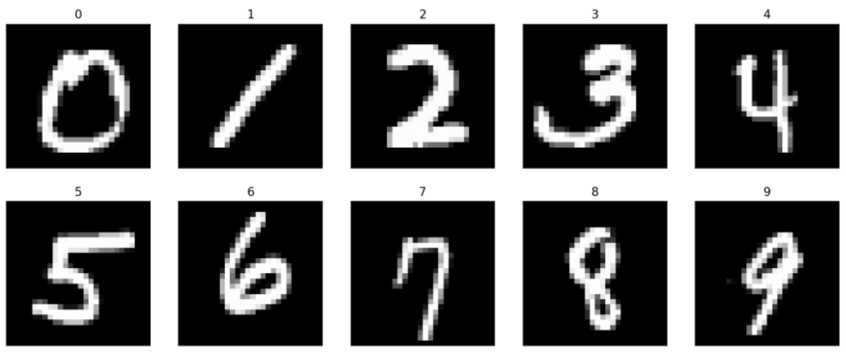

Figure 1: Examples of the MNIST dataset

Initially, we define the training and testing data loader as follows.

```
def create_data_generator(batch_size = 32, root = "data"):
  train_dataset = torchvision.datasets.MNIST(root = root,
                                             train = True,
                                             transform =
torchvision.transforms.ToTensor(),
                                             download = True)
  test_dataset = torchvision.datasets.MNIST(root = root,
                                             train = False,
                                             transform =
torchvision.transforms.ToTensor(),
                                             download = True)
  train_loader = torch.utils.data.DataLoader(train_dataset,
                                             batch_size =
batch_size,
                                             shuffle = True)
  test_loader = torch.utils.data.DataLoader(test_dataset,
                                             batch_size =
batch_size,
                                             shuffle = False)
  return train_loader, test_loader
```

In the above function, we can configurate the batch size upon network size and (GPU) memory of your computer. Next, we define the model to classify an image as follows.

```
class DigitNetwork(nn.Module):
  def __init__(self):
    super(DigitNetwork, self).__init__()
    # YOUR CODE HERE

  def forward(self, x):
    # x has the size of (batch size x 1 x height x height)
    # YOUR CODE HERE
```

In the `DigitNetwork` class, we need to design and implement the neural network to classify the digit of the given image.

```
cuda = torch.cuda.is_available()
batch_size = 32
train_loader, test_loader = create_data_generator(batch_size)
model = DigitNetwork()
print(model)
if cuda:
  model.cuda()
n_epochs = 1
learning_rate = 0.1
optim = torch.optim.SGD(params = model.parameters(), lr =
learning_rate, momentum=0.9)
loss_fn = nn.CrossEntropyLoss()

model.train()
for epoch in range(1, n_epochs + 1):
  for idx, (images, labels) in enumerate(train_loader):
    # WRITE YOUR CODE TO COMPUTE OUTPUTS, LOSS, ACCURACY, AND
OPTIMIZE MODEL
    # outptus = ???
    # loss = ???
    # accuracy = ???
    # optimize the model
    loss = 0.0
    accuracy = 0.0

    if idx % 100 == 0:
      print("Epoch [%d/%d]. Iter [%d/%d]. Loss: %0.2f. Accuracy:
%0.2f" % (epoch, n_epochs, idx + 1, len(train_loader), loss,
accuracy))

torch.save(model.state_dict(), "MNIST_Network.pth")
```

In the code above, we first the cuda (GPU) is available or not, followed by defining the batch size and getting the data loader. Next, we define the number of epochs, learning rate, loss function, and the optimizer. Then, we will loop over epoch, for each epoch, we loop over the training set to train the model. Finally, we define the evaluation framework to evaluate our learned model.

```
cuda = torch.cuda.is_available()
batch_size = 1
train_loader, test_loader = create_data_generator(batch_size)
model = DigitNetwork()
if cuda:
  model.cuda()
model.eval()
model.load_state_dict(torch.load("MNIST_Network.pth"))

total_accuracy = 0.0
for idx, (images, labels) in enumerate(test_loader):
  # WRITE YOUR CODE TO COMPUTE ACCURACY
  # accuracy = ???
  accuracy = 0.0

  total_accuracy += accuracy

  if idx % 2000 == 0:
    print("Iter [%d/%d]. Accuracy: %0.2f" % (idx + 1,
len(test_loader), accuracy))

print("Final Accuracy: %0.2f" % (total_accuracy /
len(test_loader)))
```

In the script above, we define the model and load the model from the checkpoint that we save from the training framework. Then, we loop over the testing set and calculate the accuracy for each iteration. Finally, we report the final accuracy. If you run the script successfully, the output should look as follows.

a. Design and implement the neural network for the `DigitNetwork` class. Calculate the number of parameters of the model with your configuration (report in each layer).
b. Implement the missing code in the training and evaluation framework. Then, Configure your learning hyper-parameters (batch size, learning rate, number of epochs). Finally,

train the model with the configuration you have chosen for the model and hyper-parameters. Report training losses and training accuracies in the curve graph. Report the average time of each iteration (including forward and backward passes) and the total time training the model.

c. Report your final accuracy. How long is your inference time of the model in the testing mode (consider one sample, i.e. batch size is one)? Visualize the confusion matrix of the final results.

   The confusion matrix is a 10x10 matrix where the row indicates the truth label and the column indicates the predicted label. The cell *(i, j)* in the confusion matrix illustrates the number of samples with label i, predicted to label ***j*** by your model. You can read here (Scikit-Learn Team) for more information about the confusion matrix.

d. Visualize your correct and incorrect predicted samples (10 samples for each, if you don't have any incorrectly predicted samples, you can ignore them). For each sample, you report the confidence score provided by the model for this sample. (You can visualize look like Figure 1 above.)

3. Backpropagation

Backpropagation is an algorithm for supervised learning of neural networks using gradient descent. Given a neural network and a loss function, the method calculates the gradient of the loss function to the neural network's weights. In the backpropagation method, we have to calculate derivatives of the loss with respect to the weights and the loss with respect to the input and output (of each layer). In this section, we will implement a function, including forward and backward passes, that calculates the output of the forward pass and the functions' derivatives (with respect to input and weights). Let us consider the ReLU activation to understand the format of the function as follows.

```
#
https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#
pytorch-defining-new-autograd-functions
class MyReLU(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by
subclassing
    torch.autograd.Function and implementing the forward and
backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the
input and return
        a Tensor containing the output. ctx is a context object
that can be used
        to stash information for backward computation. You can
cache arbitrary
        objects for use in the backward pass using the
ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the
gradient of the loss
        with respect to the output, and we need to compute the
gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input
```

To define a new function, we will define a class for this function which inherits the base class `torch.autograd.Function` of pytorch. We will implement two static methods: (1) forward and (2) backward. In the `MyReLU` class, the forward function receives a ctx and an input and return the output the ReLU layer. The ctx argument is context of the function, which allows stored needed values for the backward pass. For example, in the forward pass of the `MyReLU` class, we will store the input for the backward pass. Next, in the backward pass, we will calculate the gradient of the input (derivative of loss with respect to an input) and the gradient of weights (derivative of loss with respect to weights). In the example of `MyReLU` class, since we don't have any weights (parameters), this function, therefore, doesn't calculate the derivative of loss with respect to weights. This function receives the ctx and the grad_output values. The grad_output is the derivative of the loss with respect to the output. From the ctx, you can unveil the input (stored from the forward pass). The backward function has to return the gradient of the input and gradients of weights.

a. In the question 1, you already define the sigmoid function and its derivative. In this question, you must implement forward and backward functions for the sigmoid layer.

```
class MySigmoid(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        # input is a N x C tensor, N is the batch size, C is
the dimension of input
        ctx.save_for_backward(input)
        # YOUR CODE HERE
        # return output of sigmoid function

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        # YOUR CODE HERE
        # return grad_input
```

b. You are required to implement forward and backward functions for a fully connected layer (dense layer).

A fully connected is formulated as follows.

$$y = xW + b$$

where x is the input, y is the output, $W \in \mathbb{R}^{CxD}$ is the weights, $b \in \mathbb{R}^{D}$ and is the bias (C, D are the input and output dimension, respectively).

```
class MyLinearFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weights, bias):
        # input is a N x C tensor, N is the batch size, C is
the dimension of input
        # weights is a C x D tensor, C and D are the dimension
out input and ouput
        # bias is D tensor
        ctx.save_for_backward(input, weights, bias)
        # YOUR CODE HERE
        # return output of linear function

    @staticmethod
    def backward(ctx, grad_output):
        input, weights, bias = ctx.saved_tensors
        # YOUR CODE HERE
        # return grad_input, grad_weights, and grad_bias

class MyLinearLayer(nn.Module):
  # You don't modify this layer
  def __init__(self, in_features = 2, out_features = 4):
    self.weights = nn.Parameter(torch.randn(in_features,
out_features))
    self.bias = nn.Parameter(torch.zeros(out_features))
    self.linear_fn = MyLinearFunction.apply

  def forward(self, input):
    return self.linear_fn(x, self.weights, self.bias)
```

c. Testing your implementation. We will reuse the training and testing framework of question 2 to test your implementation. Report your final accuracy for this question. Initially, we define the simple neural network using your sigmoid function and fully connected layer.

```
class MyLinearNetwork(nn.Module):
  def __init__(self):
    super(MyLinearNetwork, self).__init__()
    self.linear_1 = MyLinearLayer(28 * 28, 128)
    self.sigmoid_fn = MySigmoid.apply
    self.linear_2 = MyLinearLayer(128, 10)
    self.softmax_fn = nn.Softmax(dim=1)

  def forward(self, x):
    size = x.size()
    x = x.reshape(size[0], -1) # Flatten images
    x = self.linear_1(x)
    x = self.sigmoid_fn(x)
    x = self.linear_2(x)
    if self.training == False:
      x = self.softmax_fn(x)
    return x
```

Finally, complete the missing code in the training and evaluation frameworks and train your model and report the final accuracy. (You are free to modify the bath size, the number of epochs, and learning rate).

```
cuda = torch.cuda.is_available()
batch_size = 32
train_loader, test_loader = create_data_generator(batch_size)
model = MyLinearNetwork()
print(model)
if cuda:
  model.cuda()
n_epochs = 3
learning_rate = 0.1
optim = torch.optim.SGD(params = model.parameters(), lr =
learning_rate, momentum=0.9)
loss_fn = nn.CrossEntropyLoss()

model.train()
for epoch in range(1, n_epochs + 1):
  for idx, (images, labels) in enumerate(train_loader):
    # WRITE YOUR CODE TO COMPUTE OUTPUTS, LOSS, ACCURACY, AND
OPTIMIZE MODEL
    # outptus = ???
    # loss = ???
    # accuracy = ???
    # optimize the model
    loss = 0.0
    accuracy = 0.0

    if idx % 100 == 0:
      print("Epoch [%d/%d]. Iter [%d/%d]. Loss: %0.2f.
Accuracy: %0.2f" % (epoch, n_epochs, idx + 1,
len(train_loader), loss, accuracy))

total_accuracy = 0.0
model.eval()
for idx, (images, labels) in enumerate(test_loader):
  # WRITE YOUR CODE TO COMPUTE ACCURACY
  # accuracy = ???
  accuracy = 0.0

  total_accuracy += accuracy

  if idx % 2000 == 0:
    print("Iter [%d/%d]. Accuracy: %0.2f" % (idx + 1,
len(test_loader), accuracy))

print("Final Accuracy: %0.2f" % (total_accuracy /
len(test_loader)))
```

References

Y. LeCun, L. Bottou, Y. Bengio and P. Haffner: Gradient-Based Learning Applied to Document Recognition, Proceedings of the IEEE, 86(11):2278-2324, November 1998.

Scikit-Learn Team. Confusion Matrix. https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html