Deep Learning Assignment Report

Karthik Babu Nambiar

2320702

1 Question 1

Which loss function, out of Cross Entropy and Mean Squared Error, works best with logistic regression because it guarantees a single best answer (no room for confusion)? Explain why this is important and maybe even show how it affects the model's training process.

Logisitc function for a single independent variable x is given as

$$\sigma_{\beta_0,\beta_1}(x) = \frac{1}{1 + \exp\left(-\beta_0 - \beta_1 x\right)} \tag{1}$$

The Mean Squared Error (MSE) cost function measures the distances between model and data and is given by

$$h = \sum_{i=1}^{N} (y_i - \sigma_{\beta_0,\beta_1}(x_i))^2 = \sum_{i=1}^{N} (y_i - \sigma_i)^2 \tag{2}$$

where $\sigma_i = \sigma(f(x_i)) = \sigma(\beta_0 + \beta_1 x_i)$.

Even though MSE cost function is convex for linear regression problem, it is non-convex for logistic regression.

Proof: Mean squared error is not convex as cost function for logistic regression

From 2, calculate the first order derivative of summand

$$\frac{\partial (y_i - \sigma_i)^2}{\partial \beta_k} = -2(y_i - \sigma_i)\frac{\partial \sigma_i}{\partial \beta_k} \tag{3}$$

The last factor is according to the chain rule

$$\frac{\partial \sigma(f)}{\partial \beta_k} = \frac{\partial \sigma(f)}{\partial f}\frac{\partial f}{\partial \beta_k} \tag{4}$$

We know that

$$\frac{\partial \sigma(f)}{\partial f} = \sigma(f)(1 - \sigma(f)) \tag{5}$$

Therefore 3 becomes,

$$\begin{aligned}
\frac{\partial (y_i - \sigma_i)^2}{\partial \beta_k} &= -2(y_i - \sigma_i)\frac{\partial \sigma}{\partial f}\frac{\partial f}{\partial \beta_k} && (6)\\
&= -2(y_i - \sigma_i)\sigma_i(1 - \sigma_i)\frac{\partial f}{\partial \beta_k} && (7)\\
&= -2(y_i\sigma_i - y_i\sigma_i^2 - \sigma_i^2 + \sigma_i^3)\frac{\partial f}{\partial \beta_k} && (8)
\end{aligned}$$

Now calculate the second order derivatice of summand

$$
\begin{aligned}
\frac{\partial}{\partial \beta_k}\left(\frac{\partial (y_i-\sigma_i)^2}{\partial \beta_k}\right) &= \frac{\partial}{\partial \beta_k}\left(-2(y_i\sigma_i - y_i\sigma_i^2 - \sigma_i^2 + \sigma_i^3)\frac{\partial f}{\partial \beta_k}\right) && (9)\\
&= -2\left(y_i\frac{\partial \sigma_i}{\partial \beta_k} - y_i\frac{\partial \sigma_i^2}{\partial \beta_k} - \frac{\partial \sigma_i^2}{\partial \beta_k} + \frac{\partial \sigma_i^3}{\partial \beta_k}\right)\frac{\partial f}{\partial \beta_k} && (10)\\
&\quad + -2(y_i\sigma_i - y_i\sigma_i^2 - \sigma_i^2 + \sigma_i^3)\cancelto{0}{\frac{\partial^2 f}{\partial \beta_k^2}} && (11)\\
&= -2(y_i - 2y_i\sigma_i - 2\sigma_i + 3\sigma_i^2)\frac{\partial \sigma_i}{\partial \beta_k}\frac{\partial f}{\partial \beta_k} && (12)\\
&= -2(y_i - 2y_i\sigma_i - 2\sigma_i + 3\sigma_i^2)\frac{\partial \sigma_i}{\partial f}\left(\frac{\partial f}{\partial \beta_k}\right)^2 && (13)\\
&= -2(y_i - 2y_i\sigma_i - 2\sigma_i + 3\sigma_i^2)\sigma_i(1-\sigma_i)\left(\frac{\partial f}{\partial \beta_k}\right)^2 && (14)
\end{aligned}
$$

We know that

$$\sigma_i(1-\sigma_i)\left(\frac{\partial f}{\partial \beta_k}\right)^2 := g \geq 0 | \sigma_i \in [0,1] \tag{15}$$

For binary classification problem $y_i \in \{0,1\}$
for $y_i = 0$

$$\frac{\partial^2 (y_i-\sigma_i)^2}{\partial \beta_k^2} = -2(-2+3\sigma_i)\sigma_i g \tag{16}$$

for $y_i = 1$

$$\frac{\partial^2 (y_i-\sigma_i)^2}{\partial \beta_k^2} = -2(1-4\sigma+3\sigma_i^2)g = -2(1-\sigma)(1-3\sigma)g \tag{17}$$

Hence there is a sign change and $\frac{\partial^2 h_i^2}{\partial \beta_k^2} \ngeq 0$.
Proof: Cross Entropy is convex as cost function for logistic regression

$$
\begin{aligned}
h_i &= -y_i\log(\sigma_i) - (1-y_i)\log(1-\sigma_i) && (18)\\
\frac{\partial h_i}{\partial \beta_k} &= -y_i\frac{\partial \log(\sigma_i)}{\partial \beta_k} - (1-y_i)\frac{\partial \log(1-\sigma_i)}{\partial \beta_k} && (19)\\
&= -y_i\frac{\partial \sigma_i}{\partial \beta_k}\frac{1}{\sigma_i} + (1-y_i)\frac{\partial \sigma_i}{\partial \beta_k}\frac{1}{1-\sigma_i} && (20)\\
\textbf{with}\frac{\partial \sigma_i}{\partial \beta_k} &= \sigma(f)(1-\sigma(f))\frac{\partial f}{\partial \beta_k} && (21)\\
\frac{\partial h_i}{\partial \beta_k} &= (-y_i(1-\sigma_i) + (1-y_i)\sigma_i)\frac{\partial f}{\partial \beta_k} && (22)\\
\frac{\partial h_i}{\partial \beta_k} &= (\sigma_i - y_i)\frac{\partial f}{\partial \beta_k} && (23)\\
\frac{\partial^2 h_i}{\partial \beta_k^2} &= \frac{\partial \sigma_i}{\partial \beta_k}\frac{\partial f}{\partial \beta_k} + \sigma_i\cancelto{0}{\frac{\partial^2 f}{\partial \beta_k^2}} && (24)\\
\frac{\partial^2 h_i}{\partial \beta_k^2} &= \sigma_i(1-\sigma_i)\left(\frac{\partial f}{\partial \beta_k}\right)^2 && (25)\\
\frac{\partial^2 h_i}{\partial \beta_k^2} &\geq 0 && (26)
\end{aligned}
$$

Hence Cross Entropy works best with logistic regression when compared to Mean Squared Error and guarentees a single best answer because of its convexity. This will help during training as it will take less time to converge to the minimum.

2 Question 2

For a binary classification task with a deep neural network (containing at least one hidden layer) equipped with linear activation functions, which of the following loss functions guarantees a convex optimization problem? Justify your answer with a formal proof or a clear argument. (a) CE (b) MSE (c) Both (A) and (B) (d) None

A linear neural network can be denoted as

$$f(x) = W^T x = \beta_0 + \beta_1 x \tag{27}$$

Consider MSE loss function

$$h_i = (y_i - f_i)^2 \tag{28}$$

$$\frac{\partial h_i}{\partial \beta_k} = -2(y_i - f_i)\frac{\partial f_i}{\partial \beta_k} \tag{29}$$

$$\frac{\partial^2 h_i}{\partial \beta_k^2} = -2\cancelto{0}{\frac{\partial y_i}{\partial \beta_k}}\frac{\partial f_i}{\partial \beta_k} + 2\left(\frac{\partial f_i}{\partial \beta_k}\right)^2 - 2(y_i - f_i)\cancelto{0}{\frac{\partial^2 f_i}{\partial \beta_k^2}} \tag{30}$$

$$\frac{\partial^2 h_i}{\partial \beta_k^2} \geq 0 \tag{31}$$

Consider CE loss

$$h_i = -y_i \log(f_i) - (1 - y_i)\log(1 - f_i) \tag{32}$$

$$\frac{\partial h_i}{\partial \beta_k} = -y_i \frac{\partial \log(f_i)}{\partial \beta_k} - (1 - y_i)\frac{\partial \log(1 - f_i)}{\partial \beta_k} \tag{33}$$

$$= -y_i \frac{\partial f_i}{\partial \beta_k}\frac{1}{f_i} + (1 - y_i)\frac{\partial f_i}{\partial \beta_k}\frac{1}{1 - f_i} \tag{34}$$

$$= \left(\frac{-y_i}{f_i} + \frac{1 - y_i}{1 - f_i}\right)\frac{\partial f_i}{\partial \beta_k} \tag{35}$$

$$\frac{\partial^2 h_i}{\partial \beta_k^2} = \left(\frac{y_i}{f_i^2} + \frac{1 - y_i}{(1 - f_i)^2}\right)\left(\frac{\partial f_i}{\partial \beta_k}\right)^2 + \left(\frac{-y_i}{f_i} + \frac{1 - y_i}{1 - f_i}\right)\cancelto{0}{\frac{\partial^2 f}{\partial \beta_k^2}} \tag{36}$$

$$\frac{\partial^2 h_i}{\partial \beta_k^2} \geq 0 | y_i \in \{0, 1\} \tag{37}$$

Therefore both CE and MSE loss functions guarentees a convex optimization problem in this case.

3 Question 3

Dense Neural Network: Implement a feedforward neural network with dense layers only. Specify the number of hidden layers, neurons per layer, and activation functions. How will you preprocess the input images? Consider hyperparameter tuning strategies.

Importing all necessary libraries.

```python
    import torch
    import torch.nn as nn
    import torchvision
    import torchvision.transforms as transforms
    from torch.utils.data import DataLoader, random_split
```

Defining the neural network architecture. It has one input layer, one output layer and four hidden layers. The model has the following number of neurons : [784, 256, 256, 128, 128, 10]. The activation function used is ReLU.

```python
# Define the neural network architecture
class NeuralNetwork(nn.Module):
```

```python
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 2*hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(2*hidden_size, 2*hidden_size)
        self.fc3 = nn.Linear(2*hidden_size, hidden_size)
        self.fc4 = nn.Linear(hidden_size, hidden_size)
        self.fc5 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        out = self.relu(out)
        out = self.fc4(out)
        out = self.relu(out)
        out = self.fc5(out)
        return out
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyperparameters
input_size = 28 * 28  # MNIST image size is 28x28
hidden_size = 128  # Number of neurons in the hidden layers
num_classes = 10  # Number of output classes (digits 0-9)
batch_size = 100
num_epochs = 5
```

The original images range from [0, 1] is transformed into [-1, 1]

```python
# Load and preprocess the MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),  # Convert images to tensors
    transforms.Normalize((0.5,), (0.5,))  # Normalize the pixel values to the range [-1, 1]
])
```

Dataloader for training on MNIST dataset.

```python
train_dataset = torchvision.datasets.MNIST(root='./data', train=True, transform=transform, download=True)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False, transform=transform)

# Split the dataset into training and validation sets
train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_dataset, val_dataset = random_split(train_dataset, [train_size, val_size])

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(dataset=val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```

Hyperparamter tuning of learning rate using Grid Search strategy.

```python
# Hyperparameter tuning - Grid Search for learning rate
learning_rates = [0.001, 0.01, 0.1]
best_accuracy = 0
best_learning_rate = None

for lr in learning_rates:
    # Initialize the model
    model = NeuralNetwork(input_size, hidden_size, num_classes).to(device)
```

```python

    # Loss and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    # Training the model
    for epoch in range(num_epochs):
        model.train()
        for i, (images, labels) in enumerate(train_loader):
            images = images.reshape(-1, 28 * 28).to(device)
            labels = labels.to(device)

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in val_loader:
            images = images.reshape(-1, 28 * 28).to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        accuracy = correct / total
        print(f'Learning Rate: {lr}, Validation Accuracy: {accuracy}')

        # Update best learning rate if current accuracy is higher
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_learning_rate = lr

print(f'Best Learning Rate: {best_learning_rate}, Best Validation Accuracy: {best_accuracy}')
```

Training the model with the best learning rate from the hyperparameter tuning.

```python

# Testing with best learning rate
# Initialize the model with the best learning rate
model = NeuralNetwork(input_size, hidden_size, num_classes).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=best_learning_rate)

# Training the model
for epoch in range(num_epochs):
    model.train()
    for i, (images, labels) in enumerate(train_loader):
        images = images.reshape(-1, 28 * 28).to(device)
        labels = labels.to(device)

        # Forward pass
```

```
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# Testing
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.reshape(-1, 28 * 28).to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print(f'Test Accuracy with Best Learning Rate: {100 * correct / total}%')
```

Result

```
Learning Rate: 0.001, Validation Accuracy: 0.9658333333333333
Learning Rate: 0.01, Validation Accuracy: 0.9268333333333333
Learning Rate: 0.1, Validation Accuracy: 0.11583333333333333
Best Learning Rate: 0.001, Best Validation Accuracy: 0.9658333333333333
Test Accuracy with Best Learning Rate: 97.05%
```

4 Question 4

Build a classifier for Street View House Numbers (SVHN) (Dataset) using pretrained model weights from PyTorch. Try multiple models like LeNet-5, AlexNet, VGG, or ResNet(18, 50, 101). Compare performance comment why a particular model is well suited for SVHN dataset. (You can use a subset of dataset (25%) in case you do not have enough compute.)

Importing the necessary libraries

```
import torch
import torchvision
from torchvision import transforms, datasets
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from sklearn.metrics import accuracy_score
```

Dataloader and data transformation of SVHN dataset

```
# Step 1: Load and preprocess the SVHN dataset
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

train_data = datasets.SVHN(root='./data', split='train', download=True, transform=transform)
test_data = datasets.SVHN(root='./data', split='test', download=True, transform=transform)

```

```python
# Split the dataset into train and validation sets
train_size = int(0.75 * len(train_data))
val_size = len(train_data) - train_size
train_dataset, val_dataset = torch.utils.data.random_split(train_data, [train_size, val_size])

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
```

training loop

```python
# Step 3: Fine-tune the pretrained models
def train_model(model, criterion, optimizer, train_loader, val_loader, num_epochs=5):
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for inputs, labels in train_loader:
            inputs, labels = inputs.cuda(), labels.cuda()
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item() * inputs.size(0)

        epoch_loss = running_loss / len(train_loader.dataset)
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}")

        # Validate the model
        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.cuda(), labels.cuda()
                outputs = model(inputs)
                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        val_accuracy = correct / total
        print(f"Validation Accuracy: {val_accuracy:.4f}")
```

Comparing result of alexnet, vgg, resnet-18, resnet-50, resnet-101. Only the last FC layer is being fune tuned

```python
# AlexNet
alexnet = torchvision.models.alexnet(pretrained=True)
for param in alexnet.parameters():
    param.requires_grad = False
alexnet.classifier[6] = nn.Linear(4096, 10)  # Modify the last fully connected layer for 10 classes
alexnet.classifier[6].requires_grad = True  # Set requires_grad to True for the fully connected layer
alexnet = alexnet.cuda()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(alexnet.classifier[6].parameters(), lr=0.001)  # Only optimize parameters of the fully
     connected layer
train_model(alexnet, criterion, optimizer, train_loader, val_loader)

# VGG
vgg = torchvision.models.vgg16(pretrained=True)
for param in vgg.parameters():
```

```
    param.requires_grad = False
vgg.classifier[6] = nn.Linear(4096, 10)  # Modify the last fully connected layer for 10 classes
vgg.classifier[6].requires_grad = True  # Set requires_grad to True for the fully connected layer
vgg = vgg.cuda()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(vgg.classifier[6].parameters(), lr=0.001)  # Only optimize parameters of the fully connected layer
train_model(vgg, criterion, optimizer, train_loader, val_loader)

# ResNet-18
resnet18 = torchvision.models.resnet18(pretrained=True)
for param in resnet18.parameters():
    param.requires_grad = False
resnet18.fc = nn.Linear(resnet18.fc.in_features, 10)  # Modify the last fully connected layer for 10 classes
resnet18.fc.requires_grad = True  # Set requires_grad to True for the fully connected layer
resnet18 = resnet18.cuda()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(resnet18.fc.parameters(), lr=0.001)  # Only optimize parameters of the fully connected layer
train_model(resnet18, criterion, optimizer, train_loader, val_loader)

# ResNet-50
resnet50 = torchvision.models.resnet50(pretrained=True)
for param in resnet50.parameters():
    param.requires_grad = False
resnet50.fc = nn.Linear(resnet50.fc.in_features, 10)  # Modify the last fully connected layer for 10 classes
resnet50.fc.requires_grad = True  # Set requires_grad to True for the fully connected layer
resnet50 = resnet50.cuda()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(resnet50.fc.parameters(), lr=0.001)  # Only optimize parameters of the fully connected layer
train_model(resnet50, criterion, optimizer, train_loader, val_loader)

# ResNet-101
resnet101 = torchvision.models.resnet101(pretrained=True)
for param in resnet101.parameters():
    param.requires_grad = False
resnet101.fc = nn.Linear(resnet101.fc.in_features, 10)  # Modify the last fully connected layer for 10 classes
resnet101.fc.requires_grad = True  # Set requires_grad to True for the fully connected layer
resnet101 = resnet101.cuda()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(resnet101.fc.parameters(), lr=0.001)  # Only optimize parameters of the fully connected layer
train_model(resnet101, criterion, optimizer, train_loader, val_loader)
```

Results

Alexnet

```
Epoch 1/5, Loss: 1.7448
Validation Accuracy: 0.5084
Epoch 2/5, Loss: 1.6341
Validation Accuracy: 0.5263
Epoch 3/5, Loss: 1.6068
Validation Accuracy: 0.5163
Epoch 4/5, Loss: 1.6011
Validation Accuracy: 0.5335
Epoch 5/5, Loss: 1.5893
Validation Accuracy: 0.5411
```

VGG

```
Epoch 1/5, Loss: 1.7812
Validation Accuracy: 0.4805
Epoch 2/5, Loss: 1.6833
Validation Accuracy: 0.5086
Epoch 3/5, Loss: 1.6733
Validation Accuracy: 0.5195
Epoch 4/5, Loss: 1.6636
Validation Accuracy: 0.5343
Epoch 5/5, Loss: 1.6614
Validation Accuracy: 0.5080
```

Resnet-18

```
Epoch 1/5, Loss: 1.8437
Validation Accuracy: 0.4131
Epoch 2/5, Loss: 1.6680
Validation Accuracy: 0.4353
Epoch 3/5, Loss: 1.6341
Validation Accuracy: 0.4480
Epoch 4/5, Loss: 1.6121
Validation Accuracy: 0.4530
Epoch 5/5, Loss: 1.6033
Validation Accuracy: 0.4631
```

Resnet-50

```
Epoch 1/5, Loss: 1.8692
Validation Accuracy: 0.3840
Epoch 2/5, Loss: 1.7183
Validation Accuracy: 0.3951
Epoch 3/5, Loss: 1.6795
Validation Accuracy: 0.4355
Epoch 4/5, Loss: 1.6445
Validation Accuracy: 0.4414
Epoch 5/5, Loss: 1.6166
Validation Accuracy: 0.4413
```

Resnet-101

```
Epoch 1/5, Loss: 1.8666
Validation Accuracy: 0.3762
Epoch 2/5, Loss: 1.7301
Validation Accuracy: 0.4121
Epoch 3/5, Loss: 1.6848
Validation Accuracy: 0.4319
Epoch 4/5, Loss: 1.6553
Validation Accuracy: 0.4110
Epoch 5/5, Loss: 1.6310
Validation Accuracy: 0.4245
```

Typically, deeper models like VGG, ResNet-50, and ResNet-101 tend to perform better on complex datasets like SVHN due to their deeper architectures which allow them to learn more intricate features. These models have shown great success in various image classification tasks and are well-suited for datasets like SVHN.