Hoin Jung
jung414@purdue.edu
Purdue University

BME646 and ECE60146: Homework 9

Last update: 04-10-2024

Introduction

In HW9, I aim to predict sentiment of text using "Financial Sentiment Analysis" dataset. For easier implementation, I employ GRU in Pytorch, torch.nn.GRU.

Data Preparation

For the data preprocessing, I follow the homework instruction's tokenization and word embedding. More importantly, I run the experiments respecitvely to check which one is better among 'word level' and 'subword level tokenization'.

Training

Data Loading

I split the train-test into 80:20. Also, I made a one-hot vector of 3 dimensions corresponding to each emotion.

```

# Split data. Using Word-level embedding or subword-level embedding is user's choice.

# X_train, X_test, y_train, y_test = train_test_split(word_embeddings, sentiments, test_size=0.2, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(subword_embeddings, sentiments, test_size=0.2)

# Convert sentiments to one-hot vectors
le = LabelEncoder()
y_train_encoded = le.fit_transform(y_train)
y_train_one_hot = torch.nn.functional.one_hot(torch.tensor(y_train_encoded), num_classes=3)

y_test_encoded = le.transform(y_test)
y_test_one_hot = torch.nn.functional.one_hot(torch.tensor(y_test_encoded), num_classes=3)


class SentimentDataset(Dataset):
    def __init__(self, embeddings, sentiments):
        self.embeddings = embeddings
        self.sentiments = sentiments

    def __len__(self):
        return len(self.embeddings)

    def __getitem__(self, idx):
        embedding = self.embeddings[idx].squeeze(0)
        return embedding, self.sentiments[idx]

train_dataset = SentimentDataset(X_train, y_train_one_hot)
test_dataset = SentimentDataset(X_test, y_test_one_hot)

train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)
```

Network

My architecture uses a GRU (Gated Recurrent Unit) layer to process the input text sequence, capturing sequential dependencies. The GRU can be unidirectional or bidirectional, depending on the bidirectional parameter. The last hidden state of the GRU is then passed through a fully connected layer with ReLU activation to obtain the output logits. Finally, a softmax function is applied to obtain the predicted probabilities for each sentiment class. This architecture is suitable for sentiment analysis tasks where the goal is to classify the sentiment of a given text into predefined categories.

```
class MyGRU(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers, bidirectional):
        super(MyGRU, self).__init__()
        self.gru = nn.GRU(input_dim, hidden_dim, num_layers, batch_first=True, bidirectional=
    bidirectional)
        self.fc = nn.Linear(hidden_dim * (2 if bidirectional else 1), output_dim)
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        out, hidden = self.gru(x)

        out = self.fc(self.relu(out[:,-1]))
        return self.softmax(out)
```

Training

I'm optimizing unidirectional model and bidirectional model simultaneously, using different optimizer. Each model consists of 256 hidden dimension with two layers. Instead of NLL loss with LogSoftmax, I used cross-entropy loss with softmax, which is equivalent each other. I run the training 100 epochs without validation, and use the last model for the evaluation.

```


def train(model, train_loader, optimizer, criterion, device):
    model.train()
    total_loss = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs.float())

        loss = criterion(outputs, torch.argmax(labels, 1))

        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    return total_loss / len(train_loader)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

unidirectional_model = MyGRU(input_dim=768, hidden_dim=256, output_dim=3, num_layers=2,
    bidirectional=False).to(device)
bidirectional_model = MyGRU(input_dim=768, hidden_dim=256, output_dim=3, num_layers=2,
    bidirectional=True).to(device)

# optimizer_uni = optim.SGD(unidirectional_model.parameters(), lr=0.0001,momentum=0.9)
# optimizer_bi = optim.SGD(bidirectional_model.parameters(), lr=0.0001,momentum=0.9)
optimizer_uni = optim.Adam(unidirectional_model.parameters(), lr=0.001)
optimizer_bi = optim.Adam(bidirectional_model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

num_epochs = 100

```

```
uni_losses = []
bi_losses = []

for epoch in range(num_epochs):
    uni_loss = train(unidirectional_model, train_loader, optimizer_uni, criterion, device)
    uni_losses.append(uni_loss)

    bi_loss = train(bidirectional_model, train_loader, optimizer_bi, criterion, device)
    bi_losses.append(bi_loss)

    print(f'Epoch {epoch+1}/{num_epochs}, Unidirectional Loss: {uni_loss:.4f}, Bidirectional Loss:
     {bi_loss:.4f}')
```

Evaluation

```


def evaluate(model, test_loader, criterion, device):
    model.eval()
    total_loss = 0
    correct = 0
    total = 0
    all_predicted = []
    all_labels = []
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs.float())
            loss = criterion(outputs, torch.argmax(labels, 1))
            total_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            _, labels = torch.max(labels.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            all_predicted.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
    return total_loss / len(test_loader), correct / total, confusion_matrix(all_labels,
    all_predicted)

uni_test_loss, uni_accuracy, uni_conf_matrix = evaluate(unidirectional_model, test_loader,
    criterion, device)
bi_test_loss, bi_accuracy, bi_conf_matrix = evaluate(bidirectional_model, test_loader, criterion,
    device)
```

Result

In the case of training loss with subword-level embedding, bi-directional GRU shows faster convergence and slightly better accuracy compared to unidirectional GRU as shown in Figure 1 and 2. I guess bi-directional GRU can capture information from both past and future contexts within a sequence, thereby is beneficial for understanding the context. This can lead to faster convergence and improved accuracy.

As shown in Figure 3 and 4, when I change the experiments to the word-level embedding, it doesn't converge well, and shows poor test performance. It implies that the subword-level embedding is suitable for current network.

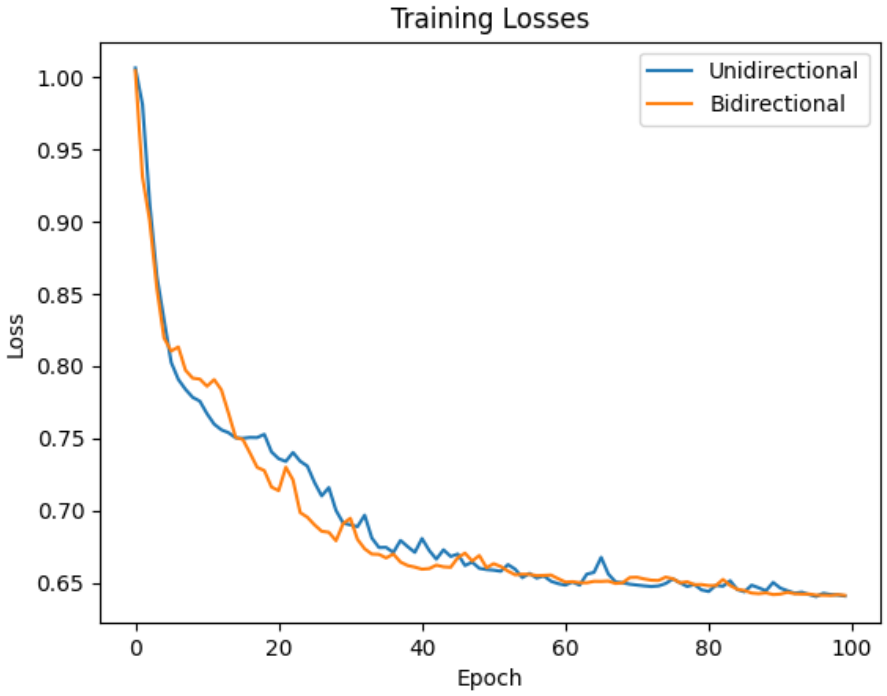

Figure 1: GRU training loss with subword-level embedding.

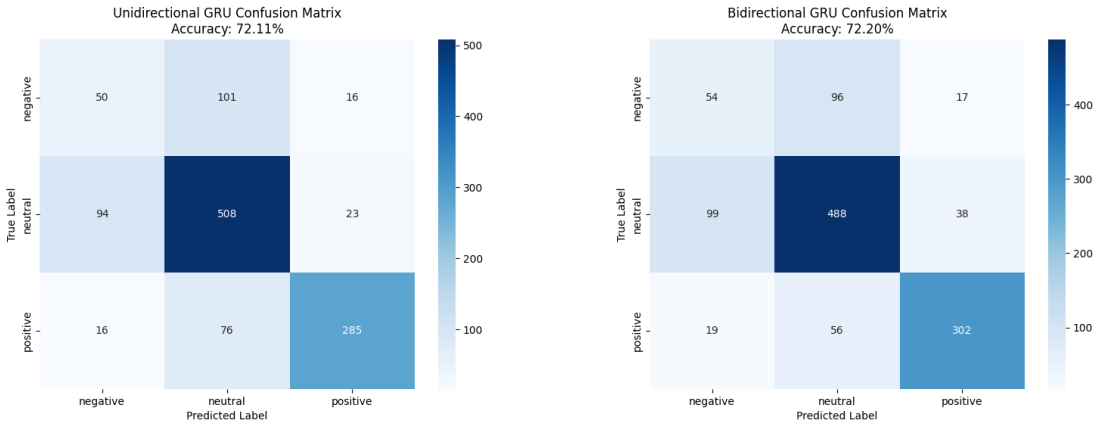

Figure 2: GRU confusion matrix and accuracy with subword-level embedding.

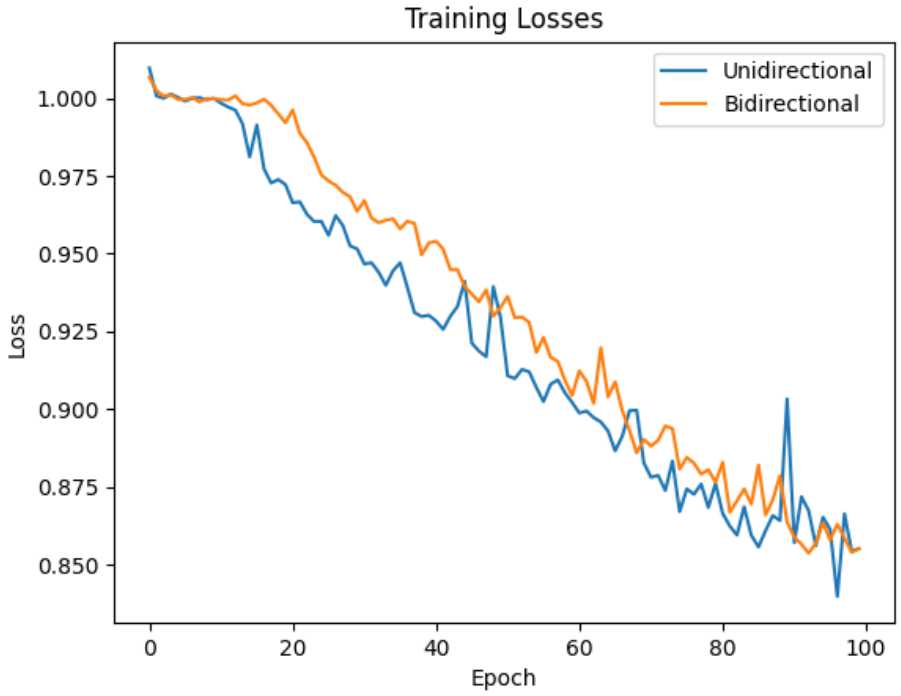

Figure 3: GRU training loss with word-level embedding.

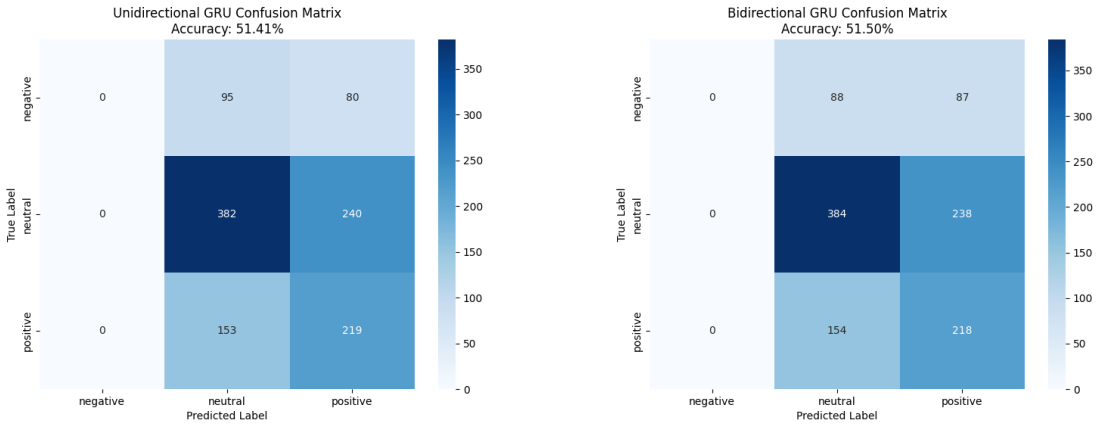

Figure 4: GRU confusion matrix and accuracy with word-level embedding.