Homework 8

Yuxin Sun

Results

1. My Own Implementation

Figure 1: `MyGRU`, `lr=1e-3`, `betas=(0.85,0.9)`

```
[epoch:4/4] [iter:14200] elapsed time:1630 secs   loss: 0.34897
model.state_dict() saved to  /home/parry/gitRepos/homeworks_ECE_60146/hw8_
Total training time:  6415 secs
Accuracy: 86.11%
```

The accuracy is 86.11%.

Figure 2: Loss

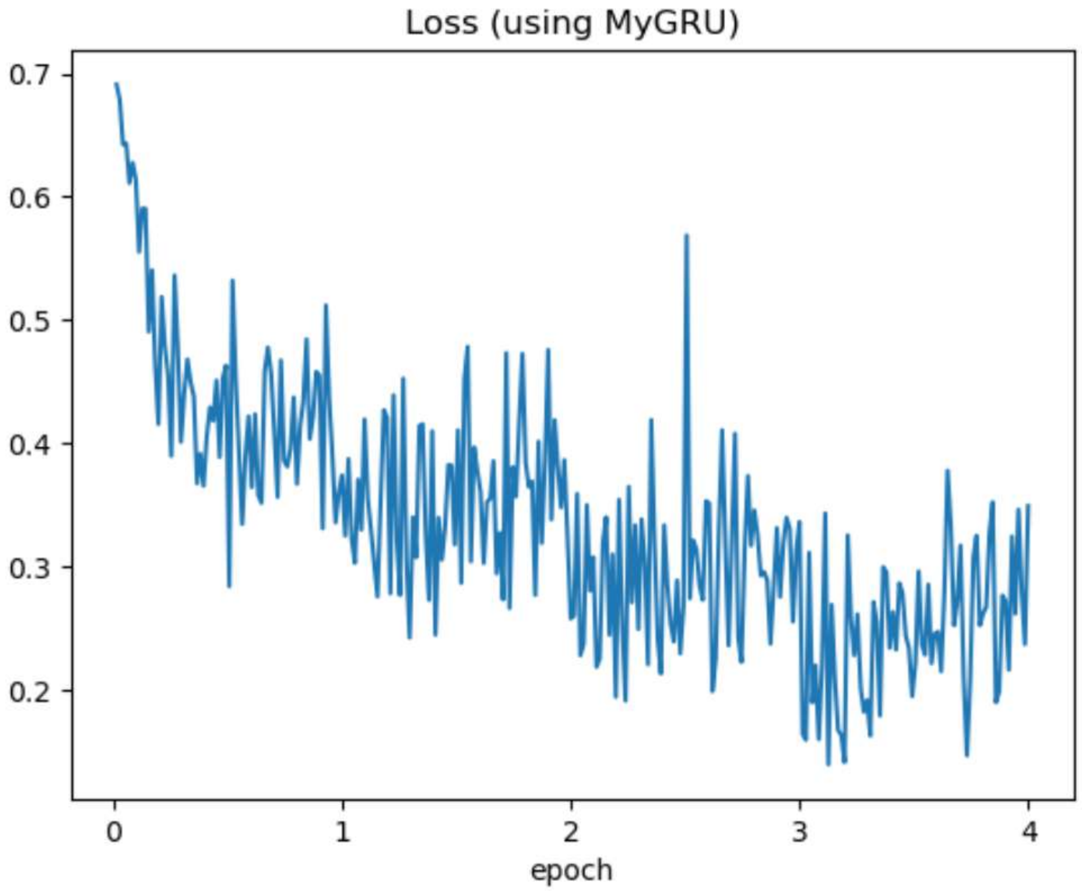
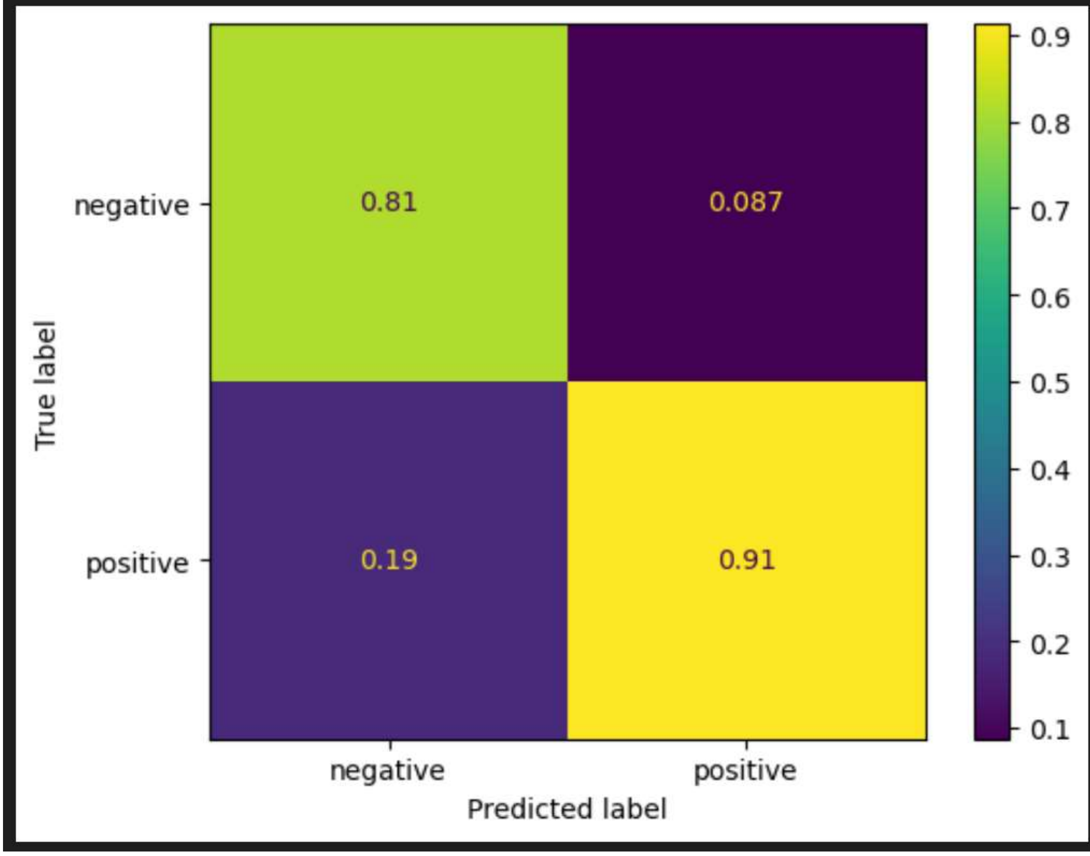


Figure 3: Confusion Matrix

```
tensor([[1502.,  150.],
        [ 345., 1566.]])
```



2. Using `nn.GRU`

Figure `nn.GRU(bidirectional=False)`

Figure 4: `nn.GRU`, `lr=1e-3`, `betas=(0.85,0.9)`

```
[epoch:4/4] [iter:14200] elapsed time:  70 secs   loss: 0.24921
model.state_dict() saved to  /content/drive/MyDrive/Colab Notebooks/saved_model_Apr_19_18:49:36
Total training time:   288 secs
Accuracy: 87.26%
```

The accuracy is 87.26%.

Figure 5: Loss

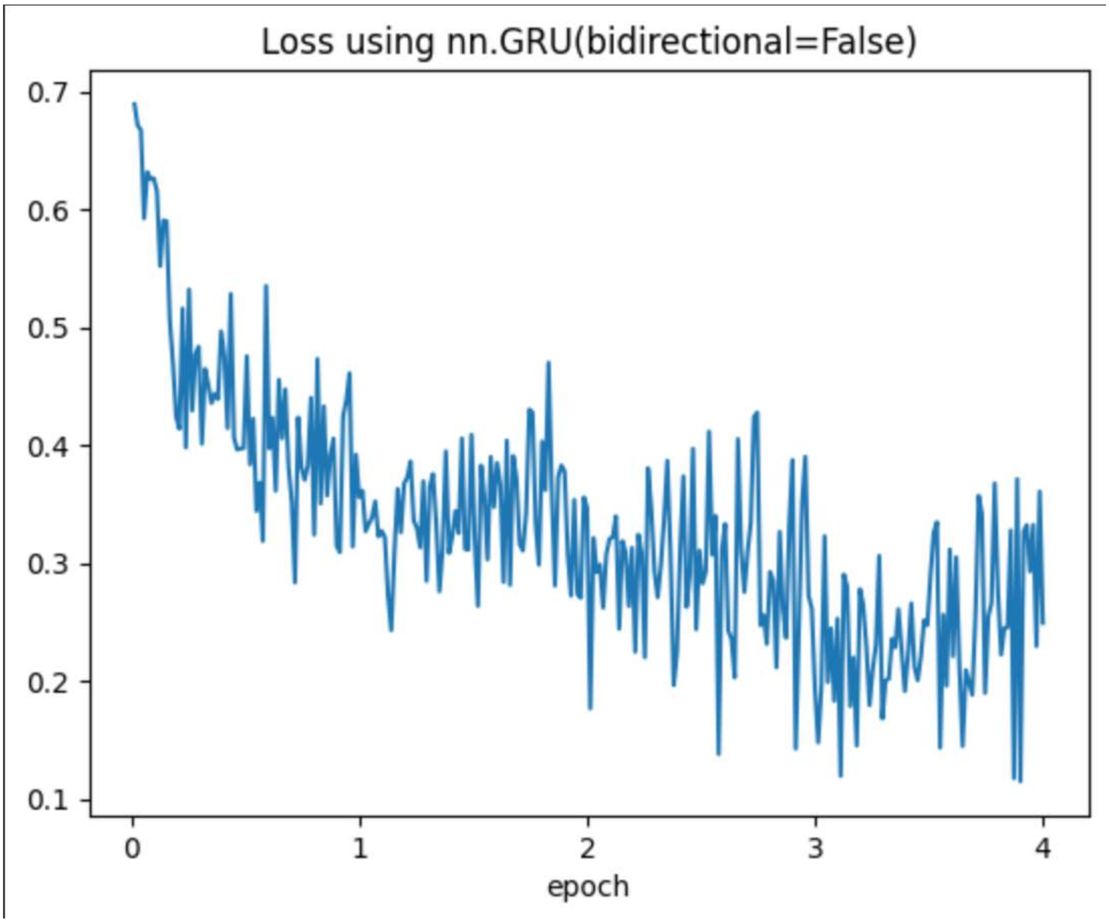
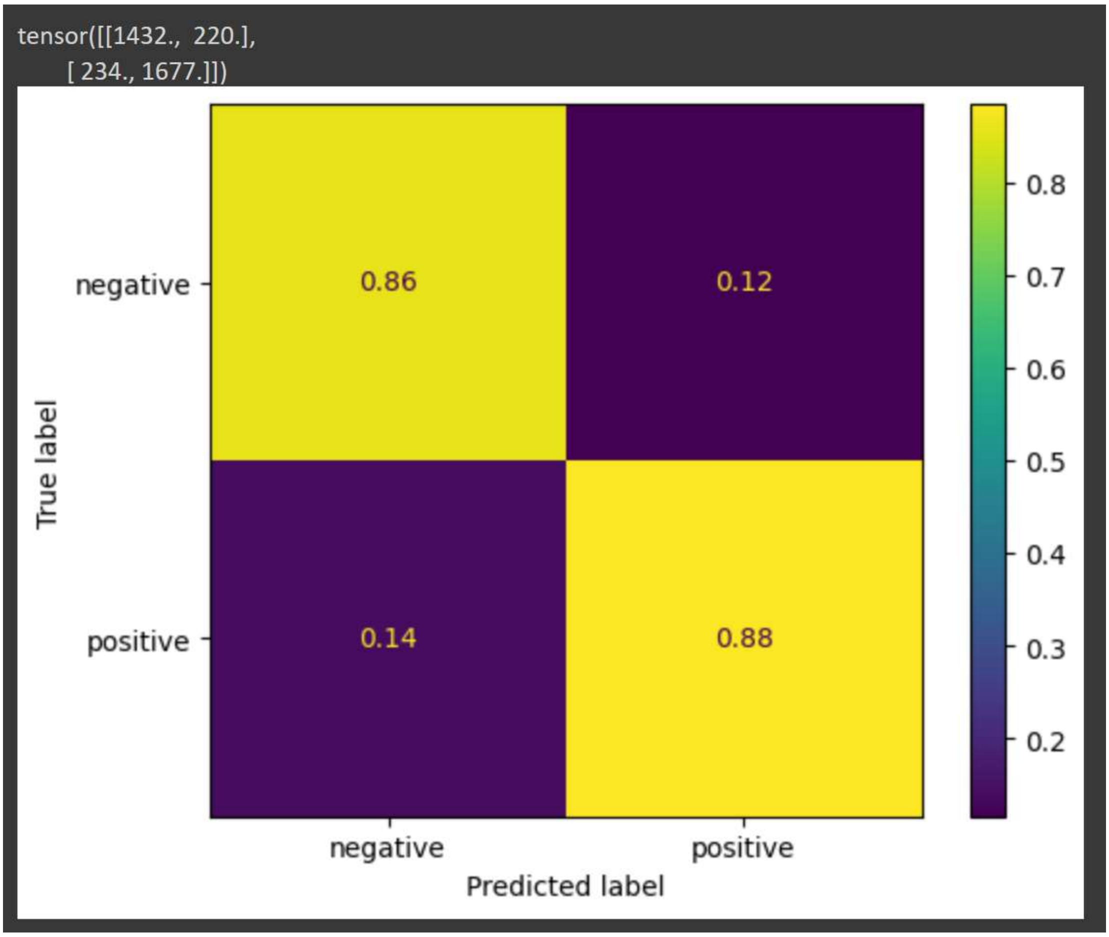


Figure 6: Confusion Matrix



3. Using `nn.GRU` with `bidirectional=True`

Figure 7: `nn.GRU(bidirectional=True)`, `lr=1e-3`, `betas=(0.85,0.9)`

```
[epoch:4/4] [iter:14200] elapsed time:  77 secs   loss: 0.13840
model.state_dict() saved to  /content/drive/MyDrive/Colab Notebooks/saved_model_Apr_19_18:10:22
Total training time:   312 secs
Accuracy: 86.70%
```

The accuracy is 86.70%.

Figure 8: Loss

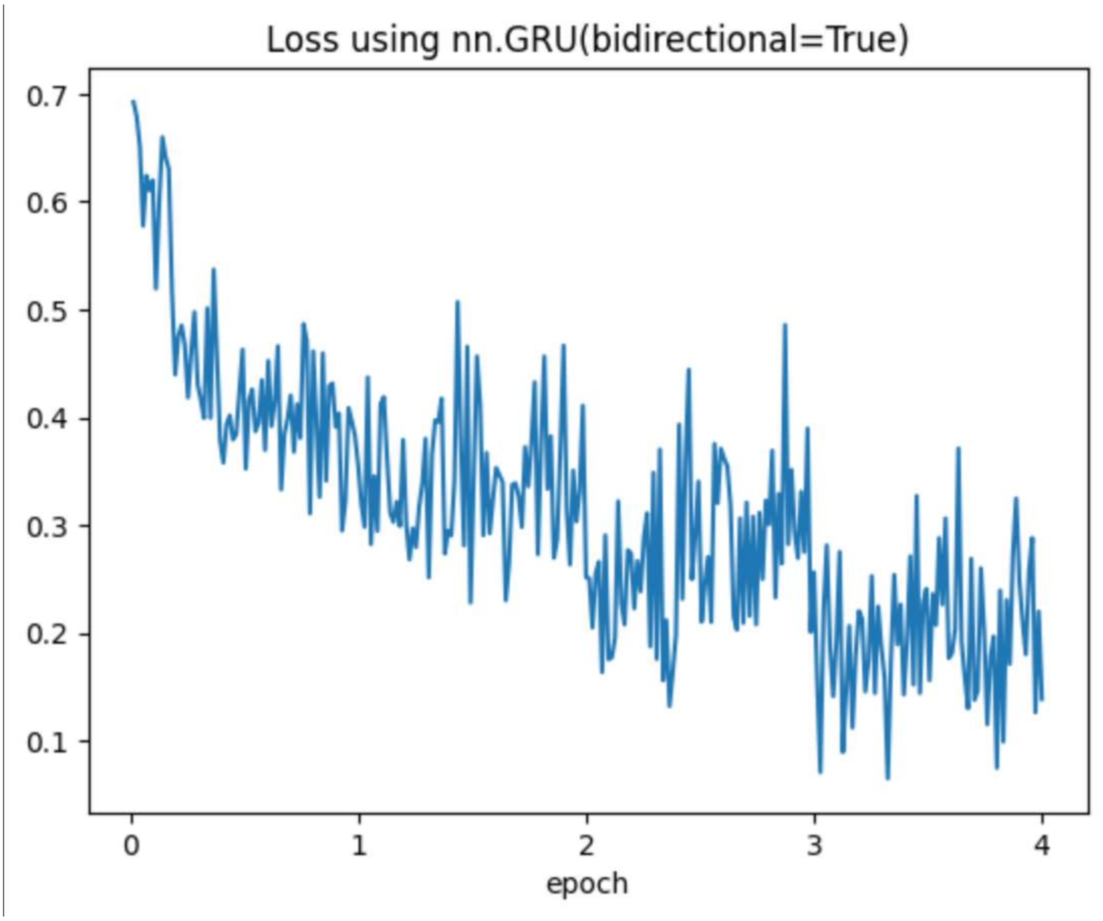
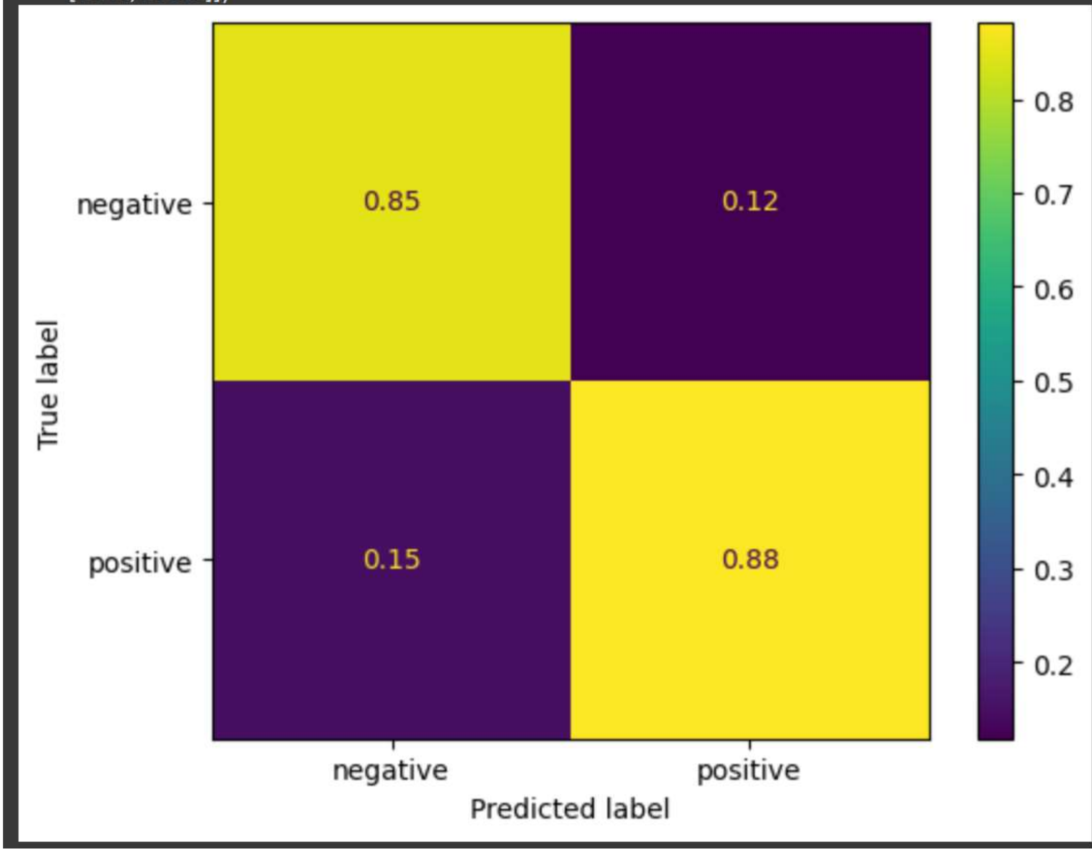


Figure 9: Confusion Matrix

```
tensor([[1429.,  223.],
      [ 251., 1660.]])
```



Code

MyGRU

By following equations on Prof. Kak's slides, I have:

```python
class MyGRU(nn.Module):
    def __init__(self, input_size, hidden_size) -> None:
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        # bias = True
        self.Wz = nn.Linear(self.input_size, self.hidden_size)
        self.Wr = nn.Linear(self.input_size, self.hidden_size)
        self.Wh = nn.Linear(self.input_size, self.hidden_size)
        self.Uz = nn.Linear(self.hidden_size, self.hidden_size)
        self.Ur = nn.Linear(self.hidden_size, self.hidden_size)
        self.Uh = nn.Linear(self.hidden_size, self.hidden_size)
        self.sigm = nn.Sigmoid()
        self.tanh = nn.Tanh()

    def forward(self, x, h):
        z = self.sigm(self.Wz(x) + self.Uz(h))
        r = self.sigm(self.Wr(x) + self.Ur(h))
        ht= self.tanh(self.Wh(x) + self.Uh(h*r))          # ht means 
\tilde{h}
        h_next = h + z*(ht-h)
```

```
        output = h_next
        return output, h_next
```

RNN

I wrap my GRU into a RNN class. It takes in all the words in a review and returns a 2D sentiment vector.

```
class RNN_1(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, *,
device="cpu"):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.device = device

        self.gru = MyGRU(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()

    def forward(self, input):
        hidden = self.init_hidden()
        for k in range(input.shape[1]):
            output, hidden = self.gru(input[0,k].unsqueeze(dim=0),
hidden)
        output = self.fc(self.relu(output))
        # there is no LogSoftMax here,
        # so we should use CrossEntropyLoss as our criterion.
        return output

    # from Prof Kak's code
    def init_hidden(self):
        # hidden = weight.new(1, self.hidden_size).zero_()
        hidden = torch.zeros(1, self.hidden_size, dtype=torch.float,
device=self.device)
        return hidden
```

The `nn.GRU` is also wrapped into a RNN:

```
class RNN_2(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, *,
device="cpu", bidirectional=False):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.device = device
        self.num_layers = 1
        self.bidirectional = bidirectional

        # self.gru = MyGRU(input_size, hidden_size)
        self.gru = nn.GRU(input_size, hidden_size, self.num_layers,
batch_first=True, bidirectional=bidirectional)
        self.fc = nn.Linear(hidden_size * (2 if self.bidirectional else
1), output_size)
```

```
        self.relu = nn.ReLU()

    def forward(self, input):
        hidden = self.init_hidden()
        output, hidden = self.gru(input, hidden)
        hidden = hidden.reshape(1,-1)
        output = self.fc(self.relu(hidden))
        return output
```

Discussion

1. how I think the problem of vanishing gradients is mitigated with the gating mechanism?

`h_next = h + z*(ht-h)`

Above formula is similar to `ResNet` which uses original information `h` plus certain new information `z*(ht-h)` from a block. The original information can go deep into the network and shorten the effective depth of the network.

Besides, there's a simple observation that if gate z is close to zero, then the previous hidden state `h` will remain almost unchanged. I guess the hidden state will not change much if we feed in an irrelevant word. This behavior may also make the effective depth of RNN smaller.

2. Does using bidirectional scan make a difference?

There is no big difference in my homework.

3. performance of the three RNNs.

All these three RNNs have an accuracy around 86%. So we can say they have the same accuracy.

I found that `nn.GRU` runs much faster than my own implementation. There may be many optimizations in `nn.GRU`, or that it is just better if we feed in an entire sequence.