ECE 60146 Deep Learning Homework 8

Mehmet Berk Sahin, sahinm@purdue.edu, 34740048

April 8, 2023

1 Introduction

In the previous homeworks, we mostly dealt with image data but there is also language data which is very important for applications like language translation, chatbots, text summarization, software retrieval, etc. In this homework, we have a language data from Amazon reviews instead of an image data. The objectives of the homework are the following. **i)** gaining insights into the internal mechanism of Recurrent Neural Networks (RNN), which are very important for language modeling, sequence-to-sequence learning, and time-series data prediction. **ii)** comprehending the gating mechanism in Gated Recurrent Units (GRU) to mitigate the problem of vanishing gradients. **iii)** designing a GRU-based RNNs for modeling variable-length product reviews for automatic classification as either positive or negative review. Before explaining the implementation, I want to explain vanilla RNNs and their severe vanishing gradient problem. Then, I will explain the GRU logic and explain how its gating mechanism mitigates the vanishing gradient problem.

1.1 RNN Logic

In image datasets, as image pixels consist of densities from 0 to 255, images can be represented numerically. However, there is no feature like that for words. They are just string-type objects. One way of representing them is using one-hot-encodings for each word. However, as we covered in class, one-hot-encoding results in large dimensional tensors. This requires model to learn more parameters and for that, model needs to have more training data (positive & negative reviews). However, more training data increases the dimensionality of the one-hot-encodings so it will require more parameters to learn and so on. This is like a chicken-egg problem. To overcome this issue and fix the dimensionality of the word vectors on the basis of the similarity of the word contexts, word embeddings were introduced. Most famous of them are `word2vec` and `fastText`. I will explain the RNN logic assuming the inputs or reviews are in the form of word embeddings.

Humans learn the meaning of a sentence by looking at each word and understanding the connection between them. So, it is important capture the relation between the last word of a sentence and the previous words. In addition to that, length of the sentences are not fixes. So, as oppossed to an image, language data has variable-length, which creates another problem. RNN overcome these challenges to some extent thanks to its feedback mechanism. Single-layer of a fully connected neural network is as follows:

$$h = g(Wx) \tag{1}$$

where h, $g(.)$, x, and W represent the output vector, nonlinear activation function, input vector, and weights of the network respectively. Different than that, one RNN cell performs the following

operation:

$$h_t = g(Wx_t + Uh_{t-1}) \tag{2}$$

for each $t \in 1, 2, \cdots, T$ where T is the length of a sequence. Input data type can be time-series or sentences. And since each series/sequence may differ, it has variable-length so T probably changes for each sentence input in our application. Thus, x_t, h_t, and h_{t-1} represent the word at index/time t, latent vector at time t, and latent vector from the previous time $t-1$. As you may notice intuitively, latent vectors are like summary of the words that are seen up to that time. Doing forward propagation via equation (2), the goal is scanning all the words and output a summary of a sentence. And by doing back-propagation through the RNN cells for each sequence, they learn the relations between the words and hopefully capture the underlying meaning of a sentence for the purpose of the task, which is semantic analysis in our case.

1.2 Vanishing Gradient Problem

As explained previously, RNN learns long chains of dependencies thanks to its feedback mechanism. However, downside of this is long chains of dependencies create gradients equal to long chains of partial derivatives. This results in either vanishing gradients or exploding gradients. For example, if partial derivatives are in between 0 and 1, then as the sequence length increases, their multiplication will approach to 0. Thus, effects (gradients) of the first words will be dominated by the effects (gradients) of the last words in a sentence. Vice versa can happen for gradients larger than 1 and it is called exploding gradients. To mitigate that problem and to learn from all words in a sequence, Gated Recurrent Units (GRUs) were introduced, which will be explained next.

1.3 GRU Logic

In this section, I will explain the GRU logic and how it mitigates vanishing gradient problem. GRU cells were first introduced to mitigate the vanishing gradient problem in RNN cells. Demonstration of a GRU cell can be seen in Figure 1. To explain its logic, I will first give the cell equations and

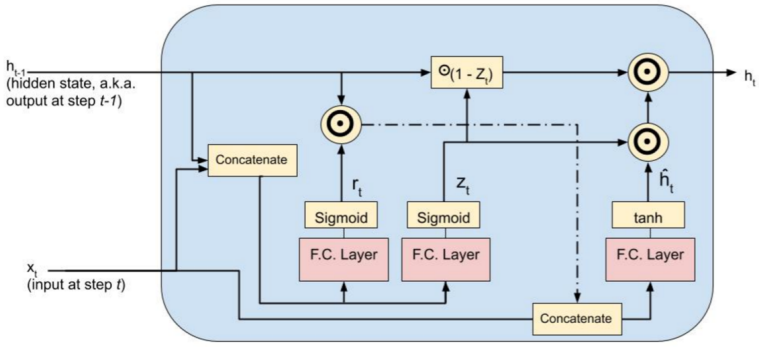


Figure 1: Demonstration of a GRU cell

then explain the meaning of the equations. They are given in equation (3), where $\cdot$ represents element-wise product.

$$\begin{aligned} z_t &= \sigma(W_z x_t + U_z h_{t-1}) \\ r_t &= \sigma(W_r x_t + U_r h_{t-1}) \\ \tilde{h}_t &= tanh(W_h x_t + U_h(r_t \cdot h_{t-1})) \\ h_t &= (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_{t-1} \end{aligned} \tag{3}$$

Input vectors of the cell are x_t and h_{t-1}, which represent the word embedding at time t and previous latent vector from the previous cell respectively. The output of the cell is h_t and if the current cell is not the last cell, it is transmitted to the next cell. Otherwise, it is used as an output or summary of a sentence. The key components that mitigates the vanishing gradient problem are **update** and **reset** gates, which are given in the first two equations of equation (3). They decide what information should be passed to the next cell. Their elements consist of weights in the range of 0 and 1. I called them weights because in the third equation, candidate for the next hidden state, $\tilde{h}_t$, is created by summing up the output of two fully connected layers whose inputs are embedding and weighted (thanks to the reset gate) previous latent vector. It is weighted because we want to pass the important elements from previous steps to the next state. Similarly, we calculate the next hidden state by taking the weighted (thanks to the update gate) summation of the previous state and the candidate state. That approach maintains the long chain dependencies and let the important information flow through the end of the GRU cells sequence. There is also simplified version of the equation (3), which is called Poor Man's GRU. It is given below:

$$\begin{aligned} f_t &= \sigma(W_f x_t + U_f h_{t-1}) \\ \tilde{h}_t &= tanh(W_h x_t + U_h(f_t \cdot h_{t-1})) \\ h_t &= (1 - f_t) \cdot h_{t-1} + f_t \cdot \tilde{h}_t \end{aligned} \tag{4}$$

Different than equation (3), number of gates was reduced to one. Instead of using different gates for update and reset, we have one gate, which is represented by f_t. It is a substitute for both update and reset gates. This implementation is for reducing the computational cost of the GRU.

Rest of the homework solution is structured as follows. Section 2 explains the structure of the dataset and how custom dataset function is constructed. Section 3 explains the implementation of GRU logic by using both hand-crafted GRU and `nn.GRU`. Section 4 discusses the results. Section 5 includes instructions to how to run the code. Section 6 consists of lessons learned from the homework. Lastly, Appendix includes the source code. For details, which are not provided in code snippets, reader can check there. I believe code is clean and well-commented.

2 Sentiment Analysis Dataset

Before embarking on finding the solution, I discovered the dataset structure, which is explained in Prof. Kak's slides. For training and testing, I used the following files:

`sentiment_dataset_train_400.tar.gz`
`sentiment_dataset_test_400.tar.gz`

and vocabulary size is 64,350. I will not explain the data structure again because it is already explained in Prof. Kak's slides. Custom dataset was implemented to extract the reviews from those files in `dataset.py` module in `SentimentDataset` class. I will not explain the details of the code because most of it were taken from Prof. Kak's `SentimentAnalysisDataset` class in `DLStudio`. I removed the unnecessary parts and changed some parts for our specific task. When the custom dataset instance initialized, it creates a list for reviews. Each element is a list whose elements are reviews, category of the review, and label (either positive or negative review). When an element was requested from the dataset, each review is converted to a tensor whose first dimension equals to the length of the sequence and second dimension equals to the size of the word embeddings. And word embeddings are size of 300 and pre-trained `word2vec` embeddings were retrieved from `gensim`

package. Each sentiment is converted to a tensor having a size of 2. If it is a negative sentiment, first element is 1, else second element is 1. Other details can be seen in the source code.

3 Implementation

In this section, I am going to explain the implementation of networks with hand-crafted GRU cell, `nn.GRU` cells and `nn.GRU(bidirectional=True)`. Also, I am going to explain how `nn.GRU` work in PyTorch API. Let's start with the GRU cell.

3.1 `GRU Cell`

Reader can see my implementation of a GRU cell in Figure 2 I call it a GRU cell because it is designed for taking one element of a sequence at a time. It does the same job as in the depiction

```python
class GRU_cell(nn.Module):

    def __init__(self, args):
        super(GRU_cell, self).__init__()
        # vector sizes
        self.hidden_size = args.hidden_size
        self.input_size = args.input_size
        self.batch_size = args.batch_size
        self.pm = args.pm # switch for poor man's GRU
        # poor man's GRU (only one forget gate)
        if args.pm:
            self.WU_f = nn.Sequential(nn.Linear(self.input_size + self.hidden_size, self.hidden_size), nn.Sigmoid())
        # real GRU (with update and reset gate)
        else:
            self.WU_z = nn.Sequential(nn.Linear(self.input_size + self.hidden_size, self.hidden_size), nn.Sigmoid())
            self.WU_r = nn.Sequential(nn.Linear(self.input_size + self.hidden_size, self.hidden_size), nn.Sigmoid())
        # layer for determining candidate hidden state
        self.WU_h = nn.Sequential(nn.Linear(self.input_size + self.hidden_size, self.hidden_size), nn.Tanh())

    def forward(self, x, h):
        inp_hid = torch.cat((x, h), dim=1)
        if self.pm:
            forget = self.WU_f(inp_hid) # forget gate
            inp_fhid = torch.cat((x, forget * h), dim=1) # input for cand. hid
            h_cand = self.WU_h(inp_fhid) # candidate hidden state
            new_h = (1 - forget) * h + forget * h_cand # next hidden state
        else:
            update = self.WU_z(inp_hid) # update gate
            reset = self.WU_r(inp_hid) # reset gate
            inp_rhid = torch.cat((x, reset * h), dim=1) # input for cand. hid
            h_cand = self.WU_h(inp_rhid) # candidate hidden state
            new_h = (1 - update) * h + update * h_cand # next hidden state
        return new_h, new_h
```

Figure 2: My implementation of a GRU cell

in Figure 1. `GRU_cell` is implemented in `models.py` module and its scalar attributes are hidden state size, input size, batch size, which is 1, and switch for pmGRU and GRU. Additionally, it has tensor attributes, which are fully connected layers for reset and update gates, and layer for candidate hidden state if `args.pm` is `False`. Otherwise, instead of reset and update gates, there is just a forget gate and a layer for candidate hidden state. To calculate the next state, `forward` function should be called until all the words are scanned. Instead of calculating each term inside $g(\cdot)$ in equation (3) and (4), I concatenated the inputs and then passed that input to the network. That's why input channels of the gates equal to `input_size` + `hidden_size`. Lastly, hidden state is returned as an output of `forward()` function. Next, I will explain the main network's class, which is `GRUNet`.

3.2 `GRUNet`

In this section, I will explain the GRU network designed for hand-crafted GRU cells, `nn.GRU` and `nn.GRU(bidirectional=True)`. `GRUNet` is a network designed for classifying the sentences as either positive sentiment or negative sentiment. And it does that by using GRU cells with fully connected layers. I will explain the implementation step by step by going through the important parts. For the full version, reader can check the source code. First, I initialize the sizes of the input, hidden

```python
class GRUNet(nn.Module):

    def __init__(self, args):
        super(GRUNet, self).__init__()
        """
        Some important tensor shapes (assuming batch_first=True):
        GRU input tensor: ( batch_size, sequence length, input_size )
        GRU output tensor: ( batch_size, sequence length, D*hidden_size )
        hidden tensor: ( D * num_layers, batch_size, hidden_size )
        """
        # GRU parameters
        self.input_size = args.input_size
        self.hidden_size = args.hidden_size
        self.num_layers = args.num_layers
        self.output_size = args.output_size
        self.D = 2 if args.bidirectional else 1
        # GRU layers
        if args.torch_gru:
            self.gru = nn.GRU(args.input_size, args.hidden_size, num_layers=args.num_layers,
                              bidirectional=args.bidirectional, batch_first=True, dropout=args.dropout)
        else:
            self.gru_cell = GRU_cell(args)
        # output layer
        self.fc = nn.Linear(self.D * args.hidden_size, args.output_size)
        self.logsoftmax = nn.LogSoftmax(dim=1) # logsoftmax layer for cross-entropy

    def zero_hidden(self, batch_size):
        # return a zero vector with the same data type as weights
        weight = next(self.parameters())
        hidden = weight.new(self.D * self.num_layers, batch_size, self.hidden_size).zero_()
        return hidden
```

Figure 3: `GRUNet` implementation part 1

state and output vectors. Moreover, there is also `num_layers` and `D` that represent number of GRU layers and switch for bidirectionality of the GRU layer. There is also `args.torch_gru` switch for using either hand-crafted GRU or `nn.GRU`, which can be seen in the `if-else` statement. After the sequence/sentence is passed through GRU layer, its output is passed to a fully connected layer and logsoftmax layer for classification. Later, in the training, `nn.LogSoftmax` will be combined with the `nn.NLLLoss` to calculate the cross-entropy loss for the sentiment predictions. I set the `num_layers` to 1 because even with a single layer, hand-crafted GRU works very slowly compared to PyTorch's GRU. So, I did not increase the layer count for fair comparison. Furthermore, I implemented an instance-method, which is `zero_hidden()`. It returns a hidden state with the same type as weight tensors. As explained in the comment, hidden state shape is as follows:

```
(D, batch_size, hidden_size)
```

That implies if `nn.GRU(bidirectional=True)`, its first axis is 2 dimensional. In fact, it returns the **last** outputs/hidden states in the forward direction and in the backward direction in time/index. As we covered in class, `nn.GRU`'s functionality depends on the shape of the input. If sequence length is 1, it operates as a GRU cell, otherwise, it operates as a GRU layer so that all sequence is processed at once. I used the second option with `batch_first=True` so input shape is as follows:

```
(batch_size, sequence_length, embedding_size)
```

I used `batch_size=1` because sequences in the dataset have varying lengths and there was not any end-of-sentence or start-of-sentence tags for reviews so I could not use fixed-length input sequences. Also, `embedding_size=300` as `word2vec` embeddings are 300 dimensional. As I used the `nn.GRU` with the second option, each output for every time index was returned so output size is as follows:

`(batch_size, sequence_length, D*hidden_size)`

The reason for `hidden_size` gets multiplied by `D` is that if `nn.GRU` is bidirectional, than outputs of each time index from different directions are concatenated and vector size is multiplied by 2. As a result of this, classifier, which is a fully connected layer, needs to double its input channel as well. Due to this concatenation, I needed to do an adjustment for the forward pass of `nn.GRU(bidirectional=True)`. As outputs at the same time index are concatenated, I cannot use the last output of `nn.GRU` because the last output with respect to the backward direction is the output at the first time index. So, I extract those last outputs and concatenate each other to get a better output vector from a sentence. Otherwise, I would concatenate the hidden state of full sentence in forward direction with the last word's hidden state. Forward pass implementatio can be seen in Figure 4. It can perform three types of forward pass. If `torch_gru=True` and `D=1`, then

```
def forward(self, x, h):
    if args.torch_gru:
        # torch's GRU
        out, new_h = self.gru(x, h)
        if self.D == 1:
            # pick the last output
            out = self.fc(F.relu(out[:,-1]))
        elif self.D == 2:
            # pick the last output of the forward direction
            out_forward = out[:,-1,:self.hidden_size]
            # pick the last output of the backward direction
            out_backward = out[:,0,self.hidden_size:]
            # concat them
            out = torch.cat((out_forward, out_backward), dim=1)
            out = self.fc(F.relu(out))
    else:
        # iterate through sequences for hand-crafted GRU
        seq_length, hidden = x.shape[1], h.squeeze(dim=1)
        # forward pass through time/words
        for word_idx in range(seq_length):
            out, hidden = self.gru_cell(x[:,word_idx], hidden)
        out = self.fc(F.relu(out))
    return self.logsoftmax(out)
```

Figure 4: `GRUNet` implementation part 2

after passing the input sequence and initial hidden state vector through `nn.GRU`, it extracts the last output and passes it through `ReLU`, fully-connected layer and `LogSoftmax` layer. If `torch_gru=True` but `D=2`, then last outputs with respect to forward and backward directions are extracted from the output of `nn.GRU` and they are concatenated as I explained previously. Then, the new vector is passed to `ReLU`, fully connected layer and `LogSoftmax` layer. If `torch_gru=False`, then each element (word embedding) of a sequence is passed to a `GRU_cell` with the previous hidden state, which has all zero elements initially. This for loop iterates until all sequence elements are passed to the cell. After that, last output is passed through `ReLU`, fully connected layer and `LogSoftmax` layer.

3.3 `train()` & `inference()`

Training function of `GRUNet` is implemented in `run_train()` instance-method that can be seen in Figure 5. Different than previous homeworks' criterions, I used `nn.NLLLoss` to combine it with

```python
def run_train(self, args, train_loader):
    # load model to device
    device = f"cuda:{args.cuda_idx}" if torch.cuda.is_available() else "cpu"
    self = self.to(device)
    print(f"model name: {args.model_name}")
    print(f"training is running on {device}!")
    # Adam optimizer with Negative Log-Likelihood Loss
    optimizer = torch.optim.Adam(self.parameters(), lr=args.lr, betas=(0.9, 0.999))
    criterion = nn.NLLLoss()
    ce_hist = [] # save cross-entropy losses
    for epoch in range(1, args.epochs + 1):
        ce_loss = 0.0
        for i, data in enumerate(train_loader, 1):
            # get the data
            embeddings, category, sentiment = data["review"], data["category"], data["sentiment"]
            # reset the optimizer
            optimizer.zero_grad()
            # load the tensors to device
            embeddings, sentiment = embeddings.to(device), sentiment.to(device)
            # set the hidden vector to zero
            hidden = model.zero_hidden(batch_size=1).to(device)
            preds = model(embeddings, hidden)
            # calculate cross-entropy
            loss = criterion(preds, torch.argmax(sentiment, dim=1))
            loss.backward() # back-propagation
            optimizer.step()
            ce_loss += loss.item()
```

Figure 5: `GRUNet` implementation part 3, `train()`

`nn.LogSoftmax` layer to calculate the cross-entropy loss. I extract the word embeddings for the words in a review, its sentiment label, and the category of the review. After setting the optimizer gradients to zero and loading the models to GPU, I called the `forward()` function of `GRUNet` by writing `model()`. Its output is a prediction for the given review so I pass it to the loss function with its label. And I back-propagate the cross-entropy loss through the network, update the network weights and save the loss. In the rest of the function, which is not shown in Figure 5, I calculated the mean cross-entropy loss of each 100 iteration and save it into a list. After the training is done, model and loss history are saved to a local disk.

To test the performance of the model on test dataset, I implemented `inference()` instance-method. It can be seen in Figure 6. It takes args and test dataloader as arguments. It only runs for one epoch and different than training, it saves predictions and ground truths for each review to use them in confusion matrix. After forward pass, predicted class indices and ground truths class indices are saved to lists by `extend()` function. Mean cross-entropy of each 100 iteration is calculated, displayed and saved. After all reviews are propagated through the network, confusion matrix is drawn with the predictions and ground truths. Its implementation can be seen in Figure 7. I use the `confusion_matrix` function of `scikit-learn`. After scaling the number of correct and false classifications to 0 and 1, I created a dataframe object from `pandas`. Then, I create the confusion matrix (heatmap) with `seaborn` and save it to a local disk. Lastly, `inference()` returns an array of cross-entropy losses for each 100 iteration.

4 Results & Discussion

In this section, I am going to present the results and discuss them. I tested three types of RNN which are my GRU implementation, `nn.GRU()`, and `nn.GRU(bidirectional=True)`. I used the following hyperparameters learning rate is 10^{-3}, optimizer is Adam with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, batch size is 1, number of epochs is 6, and cross-entropy losses are saved at each 100 iteration of the algorithm. I used the same random seed for every experiment to make a fair comparison. First, I will present the results of each experiment then, I will compare them. Henceforth, the terms **GRUh**, **GRU**, and **GRUb** refer to my GRU cell implementation, `nn.GRU(bidirectional=False)`, and

```python
def inference(self, args, test_loader):
    device = f"cuda:{args.cuda_idx}" if torch.cuda.is_available() else "cpu"
    self.load_state_dict(torch.load(args.model_name)) # load the saved model
    self = self.to(device)
    print(f"model name: {args.model_name}")
    print(f"inference is running on {device}")
    # evaluation mode
    self.eval()
    criterion = nn.NLLLoss()
    ce_hist, y_preds, y_trues = [], [], []
    ce_loss = 0
    for i, data in enumerate(test_loader, 1):
        # get the data
        embeddings, category, sentiment = data["review"], data["category"], data["sentiment"]
        # load the tensors to device
        embeddings, sentiment = embeddings.to(device), sentiment.to(device)
        # set the hidden vector to zero
        hidden = model.zero_hidden(batch_size=1).to(device)
        preds = model(embeddings, hidden)
        # save the predictions
        y_pred, y_true = torch.argmax(preds, dim=1), torch.argmax(sentiment, dim=1)
        y_preds.extend(y_pred.cpu())
        y_trues.extend(y_true.cpu())
        # calculate cross-entropy
        loss = criterion(preds, y_true)
        ce_loss += loss.item()
        if i % 100 == 0:
            mean_loss = ce_loss / i
            print(f"[ITER: {i}/{len(test_loader)}] cross-entropy loss: {round(mean_loss, 4)}")
            ce_hist.append(mean_loss)
```

Figure 6: `GRUNet` implementation part 4, `inference()`

```python
# draw & save confusion matrix
classes = ["negative", "positive"]
cm = confusion_matrix(y_trues, y_preds)
df_cm = pd.DataFrame(cm / np.sum(cm, axis=1)[:, None], index = [i for i in classes],
                     columns = [i for i in classes])
plt.figure()
sns.heatmap(df_cm, annot=True)
plt.savefig('confusion_matrix.png')
# save the loss history
np.save(f"{args.model_name}_hist", np.array(ce_hist))
return np.array(ce_hist)
```

Figure 7: `GRUNet` implementation part 5

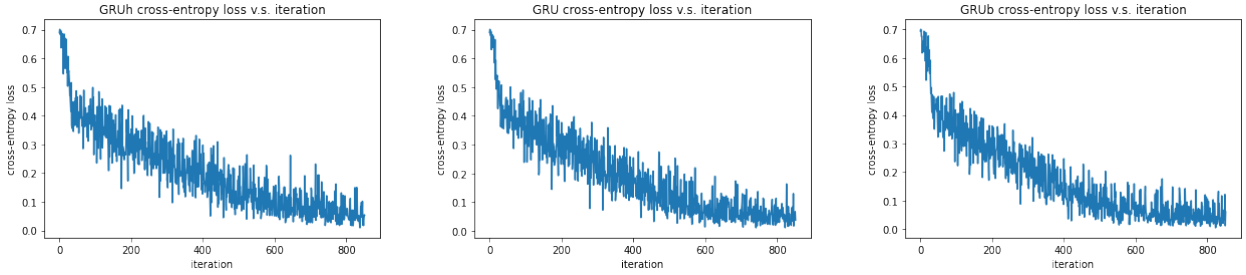
`nn.GRU(bidirectional=True)` respectively. Summary of the results of three different experiments with same parameters are given in Table 1.

Model Name	Time	Train Accuracy	Test Accuracy	Train Cross-Entropy	Test Cross-Entropy
GRUh	5hr 31m 48s	96.27	85.41	0.0644	0.0729
GRU	5m 17s	96.9	85.46	0.0556	0.0707
GRUb	7m 38s	97.56	86.64	0.0505	0.0636

Table 1: Summary of the results for three different experiments

Results suggest that test accuracies of the models are as follows $GRUb > GRU > GRUh$. Test accuracy of my GRU implementation and PyTorch's GRU are very close to each other, which implies I implemented the GRU correctly. However, since it is not optimized as PyTorch does, there is a significant time cost between my implementation of GRU and PyTorch's GRUs. My implementation took roughly 5.5 hour but others took approximately 5 to 7 minutes. Because I used loops to iterate through all sequence elements and PyTorch's codebase is C++, so PyTorch's GRU implementation is much faster than my implementation. However, as they both do the same job, accuracies are almost the same. And the time difference between bidirectional GRU and normal GRU is because the first one performs approximately twice multiplication of the second one due to additional backward direction. Another observation is that train accuracy of all models approached to %100, which implies that model is powerful enough to learn the data. I did not increase the number of epochs to increase the performance of GRU and GRUb because I think

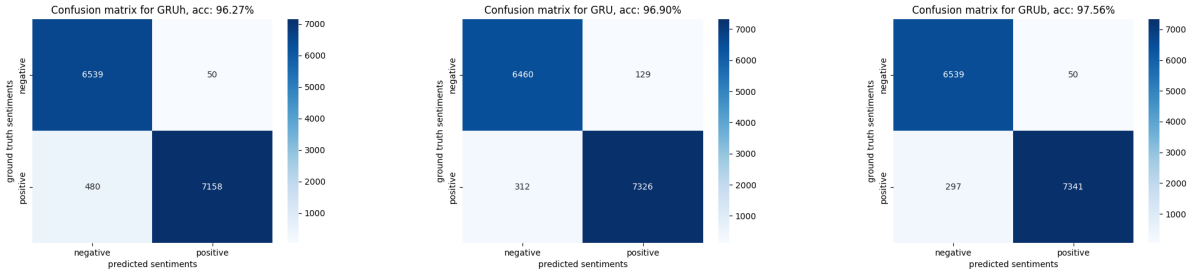
current performances are sufficient and I did not want to wait too long for GRUh. Lastly, cross-entropy losses for train and test sets are very close 0, which is an another indication for model learned how to review from the data. Cross-entropy train plot for each 100 iteration for different models can be seen in Figure 8. Train plots are noisy because I literally did SGD (batch size equals



(a) GRUh train loss (b) GRU train loss (c) GRUb train loss

Figure 8: Train losses for GRUh, GRU, and GRUb per 100 iteration

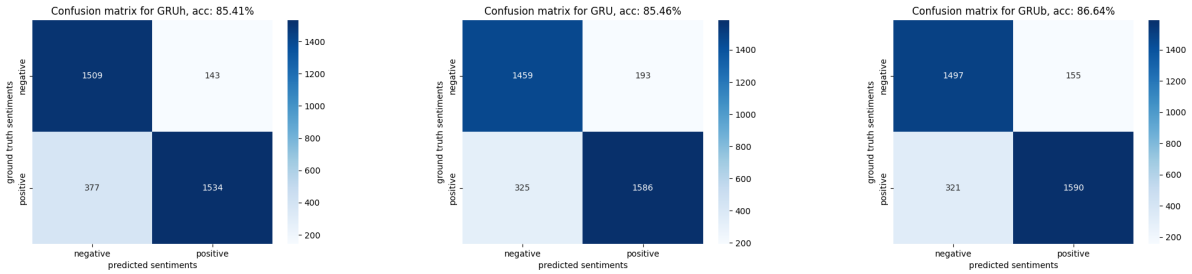
to 1). All trainings are converged to 0. This is also apparent in Table 1 from the mean cross-entropy losses for train set. I wanted to display the confusion matrices for train set to validate the accuracy of the training. They can be seen in Figure 9. Rows are for ground truths and columns



(a) GRUh confusion matrix (b) GRU confusion matrix (c) GRUb confusion matrix

Figure 9: Train confusion matrices of GRUh, GRU, and GRUb

are predicted sentiments. So, diagonal elements have correct number of predictions and other terms represent misclassifications. As it can be seen from above figure, most of the reviews are classified correctly. Confusion matrices of three models evaluate on test set can be seen in Figure 10. Test



(a) GRUh confusion matrix (b) GRU confusion matrix (c) GRUb confusion matrix

Figure 10: Test confusion matrices of GRUh, GRU, and GRUb

results show that all GRU models generalize the data well. And there is no imbalance between the

misclassifications. That is, number of misclassifications for positive and negative sentiments are very close to each other. In both train and test sets, bidirectional GRU's performance is higher than the other models. I think this is because bidirectional GRU scans the sentence in both ways, which are forward and backward, so it learns more from a sentence. However, it takes longer than `nn.GRU(bidirectional=False)` to perform forward pass. This may not scale well. For example, if we have a more complex dataset, we may need to increase the number of GRU layers and if we make the GRU bidirectional, then it will perform forward and backward pass for both directions at each layer. This may increase the computational cost tremendously.

I experimented with the following learning rates $[10^{-2}, 10^{-3}, 10^{-4}]$ and the first one did not converge. The third one's convergence was very slow. Second one fits very well as can be seen from the results. I did not try different batch size because input data has variable-length and I did not see any SOS or EOS tags before and after each review.

5 How to run the code?

Solution of this homework consists of two modules, which are `dataset.py` and `hw8_MehmetBerkSahin.py`. The latter one is the main module, which is used for training and testing. Before training, first, "data" folder should be created at the main directory and it should include `sentiment_dataset_test_400.tar.gz` and `sentiment_dataset_train_400.tar.gz` files. I followed the instructions in the homework to obtain these files. Then, to train different models, one can run the following commands to the command line:

GRUh (my implementation of GRU)

```
python hw8_MehmetBerkSahin.py --torch_gru False --model_name GRUh
```

GRU (PyTorch's GRU)

```
python hw8_MehmetBerkSahin.py --torch_gru True --model_name GRU
```

GRUb (PyTorch's bidirectional GRU)

```
python hw8_MehmetBerkSahin.py --torch_gru True --model_name GRUb --bidirectional
True
```

These commands will train the corresponding models for 6 epochs. If reader wants to change the number of epochs, s/he can change it by using the `--epochs` option. To evaluate the performance of the models, reader can run the comments given above with the addition of "`--train False`". This will evaluate the model on the test set, save the cross-entropy losses and it will save the confusion matrices to the current directory, which is "hw8_MehmetBerkSahin".

6 Lessons Learned

In this homework, I learned how to use a Recurrent Neural Network for sequential data and how to implement the feedback mechanism. I learned how word embeddings are used as a language models in RNNs. I understood the gating mechanism in GRU and by conducting experiments, I observe how it mitigates the vanishing gradient problem with its update and reset gates. I learned the different

modes of `nn.GRU()` for a given input, and since it is faster, I use the second option, which accepts the full sequence instead of just one element. I learned how to handle variable-length data type. Lastly, I learned how to extract the correct output from outputs of `nn.GRU(bidirectional=False)` and `nn.GRU(bidirectional=True)`. It was a great homework!

7 Appendix

7.1 Source Code

7.1.1 dataset.py

```
# pytorch imports
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
# other stuff
import os
import sys
import gzip
import pickle
import random
import gensim.downloader as genapi
from gensim.models import KeyedVectors
import numpy as np

class SentimentDataset(Dataset):

    """
    MOST OF THE PART OF THIS CLASS WAS TAKEN FROM PROF. KAK'S SentimentAnalysisDataset CLASS!
    I REMOVE THE UNNECESSARY COMPONENTS AND MAKE SOME CHANGES FOR THE HOMEWORK
    """

    def __init__(self, mode="train", path="data"):
        super(SentimentDataset, self).__init__()

        self.path = path
        self.mode = mode
        # get the word2vec pre-trained embeddings
        if os.path.exists('vectors.kv'):
            self.word_vectors = KeyedVectors.load('vectors.kv')
        else:
            self.word_vectors = genapi.load("word2vec-google-news-300")
            self.word_vectors.save('vectors.kv')
        # dataset file name
        file_name = f"sentiment_dataset_{mode}_400.tar.gz"
        f = gzip.open(os.path.join(path, file_name), "rb")
        dataset = f.read()
        # get positive & negative reviews and vocabulary
        if sys.version_info[0] == 3:
            self.pos_reviews, self.neg_reviews, self.vocab = pickle.loads(dataset, encoding='latin1')
        else:
            self.pos_reviews, self.neg_reviews, self.vocab = pickle.loads(dataset)
        # get categories
        self.categories = sorted(list(self.pos_reviews.keys()))
        # positive negative review category sizes
        self.cat_sizes_pos = {category: len(self.pos_reviews[category]) for category in self.categories}
        self.cat_sizes_neg = {category: len(self.neg_reviews[category]) for category in self.categories}
        # dataset consisting of pos./neg. reviews
        self.indexed_dataset = []
        for category in self.pos_reviews:
            for review in self.pos_reviews[category]:
                self.indexed_dataset.append([review, category, 1])
        for category in self.neg_reviews:
            for review in self.neg_reviews[category]:
                self.indexed_dataset.append([review, category, 0])
        # shuffle the dataset
        random.shuffle(self.indexed_dataset)

    def get_vocab_size(self):
        return len(self.vocab)

    # gets one-hot vector for the given word
    def one_hotvec_for_word(self, word):
        word_index = self.vocab.index(word)
        hotvec = torch.zeros(1, len(self.vocab))
        hotvec[0, word_index] = 1
        return hotvec

    def sentiment_to_tensor(self, sentiment):
        """
        Sentiment is ordinarily just a binary valued thing.  It is 0 for negative
        sentiment and 1 for positive sentiment.  We need to pack this value in a
        two-element tensor.
        """
```

```
        sentiment_tensor = torch.zeros(2)
        if sentiment == 1:
            sentiment_tensor[1] = 1
        elif sentiment == 0:
            sentiment_tensor[0] = 1
        sentiment_tensor = sentiment_tensor.type(torch.long)
        return sentiment_tensor

    def review_to_tensor(self, review):
        list_of_embeddings = []
        for i, word in enumerate(review):
            if word in self.word_vectors.key_to_index:
                embedding = self.word_vectors[word]
                list_of_embeddings.append(np.array(embedding))
            else:
                next
        review_tensor = torch.FloatTensor(np.array(list_of_embeddings))
        return review_tensor

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

    def __getitem__(self, idx):
        sample = self.indexed_dataset[idx]
        review = sample[0]
        review_category = sample[1]
        review_sentiment = sample[2]
        review_sentiment = self.sentiment_to_tensor(review_sentiment)
        review_tensor = self.review_to_tensor(review)
        category_index = self.categories.index(review_category)
        sample = {'review': review_tensor,
                  'category': category_index,  # should be converted to tensor, but not yet used
                  'sentiment': review_sentiment}
        return sample

# test code
if __name__ == "__main__":

    train_data = SentimentDataset()
    train_loader = DataLoader(train_data, batch_size=1)
    train_loader = iter(train_loader)
    print("Data size:", len(train_data))
```

7.1.2 hw8_MehmetBerkSahin.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
import argparse
from dataset import SentimentDataset
import random
# for confusion matrix
import numpy as np
from sklearn.metrics import confusion_matrix
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import time


class GRU_cell(nn.Module):

    def __init__(self, args):
        super(GRU_cell, self).__init__()
        # vector sizes
        self.hidden_size = args.hidden_size
        self.input_size = args.input_size
        self.batch_size = args.batch_size
        self.pm = args.pm # switch for poor man's GRU
        # poor man's GRU (only one forget gate)
        if args.pm:
            self.WU_f = nn.Sequential(nn.Linear(self.input_size + self.hidden_size, self.hidden_size), nn.Sigmoid())
        # real GRU (with update and reset gate)
        else:
            self.WU_z = nn.Sequential(nn.Linear(self.input_size + self.hidden_size, self.hidden_size), nn.Sigmoid())
            self.WU_r = nn.Sequential(nn.Linear(self.input_size + self.hidden_size, self.hidden_size), nn.Sigmoid())
        # layer for determining candidate hidden state
        self.WU_h = nn.Sequential(nn.Linear(self.input_size + self.hidden_size, self.hidden_size), nn.Tanh())

    def forward(self, x, h):
        inp_hid = torch.cat((x, h), dim=1)
        if self.pm:
            forget = self.WU_f(inp_hid) # forget gate
            inp_fhid = torch.cat((x, forget * h), dim=1) # input for cand. hid
            h_cand = self.WU_h(inp_fhid) # candidate hidden state
            new_h = (1 - forget) * h + forget * h_cand # next hidden state
        else:
            update = self.WU_z(inp_hid) # update gate
            reset = self.WU_r(inp_hid) # reset gate
            inp_rhid = torch.cat((x, reset * h), dim=1) # input for cand. hid
```

```
            h_cand = self.WU_h(inp_rhid) # candidate hidden state
            new_h = (1 - update) * h + update * h_cand # next hidden state
        return new_h, new_h

class GRUNet(nn.Module):

    def __init__(self, args):
        super(GRUNet, self).__init__()
        """
        Some important tensor shapes (assuming batch_first=True):
        GRU input tensor: ( batch_size, sequence length, input_size )
        GRU output tensor: ( batch_size, sequence length, D*hidden_size )
        hidden tensor: ( D * num_layers, batch_size, hidden_size )
        """
        # GRU parameters
        self.input_size = args.input_size
        self.hidden_size = args.hidden_size
        self.num_layers = args.num_layers
        self.output_size = args.output_size
        self.D = 2 if args.bidirectional else 1
        # GRU layers
        if args.torch_gru:
            self.gru = nn.GRU(args.input_size, args.hidden_size, num_layers=args.num_layers,
                              bidirectional=args.bidirectional, batch_first=True, dropout=args.dropout)
        else:
            self.gru_cell = GRU_cell(args)
        # output layer
        self.fc = nn.Linear(self.D * args.hidden_size, args.output_size)
        self.logsoftmax = nn.LogSoftmax(dim=1) # logsoftmax layer for cross-entropy

    def zero_hidden(self, batch_size):
        # return a zero vector with the same data type as weights
        weight = next(self.parameters())
        hidden = weight.new(self.D * self.num_layers, batch_size, self.hidden_size).zero_()
        return hidden

    def forward(self, x, h):
        if args.torch_gru:
            # torch's GRU
            out, new_h = self.gru(x, h)
            if self.D == 1:
                # pick the last output
                out = self.fc(F.relu(out[:,-1]))
            elif self.D == 2:
                # pick the last output of the forward direction
                out_forward = out[:,-1,:self.hidden_size]
                # pick the last output of the backward direction
                out_backward = out[:,0,self.hidden_size:]
                # concat them
                out = torch.cat((out_forward, out_backward), dim=1)
                out = self.fc(F.relu(out))
        else:
            # iterate through sequences for hand-crafted GRU
            seq_length, hidden = x.shape[1], h.squeeze(dim=1)
            # forward pass through time/words
            for word_idx in range(seq_length):
                out, hidden = self.gru_cell(x[:,word_idx], hidden)
            out = self.fc(F.relu(out))
        return self.logsoftmax(out)

    def run_train(self, args, train_loader):
        # load model to device
        device = f"cuda:{args.cuda_idx}" if torch.cuda.is_available() else "cpu"
        self = self.to(device)
        print(f"model name: {args.model_name}")
        print(f"training is running on {device}!")
        # Adam optimizer with Negative Log-Likelihood Loss
        optimizer = torch.optim.Adam(self.parameters(), lr=args.lr, betas=(0.9, 0.999))
        criterion = nn.NLLLoss()
        ce_hist = [] # save cross-entropy losses
        start_time = time.perf_counter()
        for epoch in range(1, args.epochs + 1):
            ce_loss = 0.0
            for i, data in enumerate(train_loader, 1):
                # get the data
                embeddings, category, sentiment = data["review"], data["category"], data["sentiment"]
                # reset the optimizer
                optimizer.zero_grad()
                # load the tensors to device
                embeddings, sentiment = embeddings.to(device), sentiment.to(device)
                # set the hidden vector to zero
                hidden = model.zero_hidden(batch_size=1).to(device)
                preds = model(embeddings, hidden)
                # calculate cross-entropy
                loss = criterion(preds, torch.argmax(sentiment, dim=1))
                loss.backward() # back-propagation
                optimizer.step()
                ce_loss += loss.item()

                if i % 100 == 0:
                    mean_loss = ce_loss / 100
                    print(f"[EPOCH: {epoch}/{args.epochs}, ITER: {i}/{len(train_loader)}] cross-entropy loss: {round(mean_loss, 4)}")
```

```
                    ce_hist.append(mean_loss)
                    ce_loss = 0.0

        end_time = time.perf_counter()
        print(f"time elapsed for training: {round(end_time - start_time, 4)}")
        # save the model
        torch.save(self.state_dict(), args.model_name)
        # save the loss history
        np.save(f"{args.model_name}_train_hist", np.array(ce_hist))
        return np.array(ce_hist)

    def inference(self, args, test_loader):
        device = f"cuda:{args.cuda_idx}" if torch.cuda.is_available() else "cpu"
        self.load_state_dict(torch.load(args.model_name)) # load the saved model
        self = self.to(device)
        print(f"model name: {args.model_name}")
        print(f"inference is running on {device}")
        # evaluation mode
        self.eval()
        criterion = nn.NLLLoss()
        ce_hist, y_preds, y_trues = [], [], []
        ce_loss = 0
        for i, data in enumerate(test_loader, 1):
            # get the data
            embeddings, category, sentiment = data["review"], data["category"], data["sentiment"]
            # load the tensors to device
            embeddings, sentiment = embeddings.to(device), sentiment.to(device)
            # set the hidden vector to zero
            hidden = model.zero_hidden(batch_size=1).to(device)
            preds = model(embeddings, hidden)
            # save the predictions
            y_pred, y_true = torch.argmax(preds, dim=1), torch.argmax(sentiment, dim=1)
            y_preds.extend(y_pred.cpu())
            y_trues.extend(y_true.cpu())
            # calculate cross-entropy
            loss = criterion(preds, y_true)
            ce_loss += loss.item()
            if i % 100 == 0:
                mean_loss = ce_loss / i
                print(f"[ITER: {i}/{len(test_loader)}] cross-entropy loss: {round(mean_loss, 4)}")
                ce_hist.append(mean_loss)
                ce_loss = 0

        # draw & save confusion matrix
        classes = ["negative", "positive"]
        cm = confusion_matrix(y_trues, y_preds)
        df_cm = pd.DataFrame(cm, index = [i for i in classes], columns = [i for i in classes])
        acc = (np.trace(cm) / np.sum(cm)) * 100
        plt.figure()
        sns.heatmap(df_cm, annot=True, fmt='g', cmap="Blues")
        plt.title(f"Confusion matrix for {args.model_name}, acc: {acc:.2f}%")
        plt.xlabel("predicted sentiments")
        plt.ylabel("ground truth sentiments")
        plt.savefig(f'{args.model_name}_confusion_matrix.png')
        # save the loss history
        np.save(f"{args.model_name}_test_hist", np.array(ce_hist))
        return np.array(ce_hist)


# test code
if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    # training details
    parser.add_argument("--batch_size", default=1, type=int, help="batch size for training")
    parser.add_argument("--epochs", default=6, type=int, help="number of epochs for training")
    parser.add_argument("--bidirectional", default=False, type=eval, help="switch for bidirectional GRU")
    parser.add_argument("--torch_gru", default=True, type=eval, help="switch for using torch's GRU")
    parser.add_argument("--cuda_idx", default=0, type=int, help="cuda idx for GPU training")
    parser.add_argument("--lr", default=1e-3, type=float, help="learning rate")
    parser.add_argument("--dropout", default=0, type=float, help="dropout probability")

    # other
    parser.add_argument("--model_name", default="GRUb", type=str, help="model will be saved & loaded with this name")
    parser.add_argument("--train", default=True, type=eval, help="switch for train or test")

    # GRU parameters
    parser.add_argument("--input_size", default=300, type=int, help="input size for GRU")
    parser.add_argument("--hidden_size", default=100, type=int, help="dimension of hidden vector")
    parser.add_argument("--output_size", default=2, type=int, help="dimension of output vector")
    parser.add_argument("--num_layers", default=1, type=int, help="number of layers for GRU")
    parser.add_argument("--pm", default=False, type=eval, help="poor man's GRU")
    args = parser.parse_args()

    # for reproducible results
    seed = 1
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic=True
    torch.backends.cudnn.benchmarks=False
```

```
torch.autograd.set_detect_anomaly(True)

"""
Recommended parameters for GRU models:

Torch's GRU
lr              :1e-3
epochs          :6
batch_size      :1
bidirectional   :False
input_size      :300
"""

# train GRUNet without bidirection
model = GRUNet(args)
# load dataset
train_data = SentimentDataset(mode="train")
test_data = SentimentDataset(mode="test")
train_loader = DataLoader(train_data, batch_size=1)
test_loader = DataLoader(test_data, batch_size=1)
if args.train:
    model.run_train(args, train_loader)
    print("training is succesfull!")
else:
    model.inference(args, test_loader)
    print("inference is succesfull!")
```