In this lab, we aim to predict whether product reviews from Amazon have a positive or negative sentiment about the product. The neural network architecture is a recurrent neural network, specifically, a Gated Recurrent Network (GRU) that keeps feeding into itself outputs from the previous run. The lab gives details on the exact prediction mechanism a GRU uses. In this lab, three models are written, trained, and tested. First, a GRU is built from scratch and then wrapped in a recurrent network. Second, PyTorch's default GRU is used to predict sentiments. Lastly, PyTorch's bi-directional GRU is used, which adds a boost to the prediction score on the same dataset.

Part 1: Sentiment Analysis with Own GRU

Product reviews were put into PyTorch's dataloader utility using the DLStudio code. The review words are "translated" to 300 dimensional embedding vectors, and each review is loaded as a tensor with the first axis representing the number of words in the review and the second axis representing the embedding dimension (300). All the following results will use this same format for loading data.

Code for own GRU, logic, and mitigating vanishing gradients:

This part of the lab required implementing our own GRU logic instead of using the module provided by PyTorch. The gated unit logic, specifically, the computation of the hidden gate (z_t), reset gate (r_t), the candidate hidden state ($h_{t\sim}$), and the new hidden state (h_t) is below:

```
class GruNet(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(GruNet, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        #weights to create the hidden gate, zt
        self.fc_hidden_gate = nn.Linear(input_size + hidden_size, hidden_size)

        #weights to create the reset gate
        self.fc_reset_gate = nn.Linear(input_size + hidden_size, hidden_size)

        #weights to create the candidate hidden state
        self.fc_hidden_state = nn.Linear(input_size + hidden_size, hidden_size)

        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()



    def forward(self, x, ht_minus1):
        #create zt, the hidden gate
        concat_xt_ht_minus1 = torch.cat((x, ht_minus1), dim = 0)
        pre_act_update_gate = self.fc_hidden_gate(concat_xt_ht_minus1)
        zt = sigmoid(pre_act_update_gate) #zt, update gate, created

        #create rt, the reset gate
        pre_act_reset_gate = self.fc_reset_gate(concat_xt_ht_minus1)
        rt = sigmoid(pre_act_reset_gate)
        #print(rt.shape)

        #create ht_tilde, the candidate hidden state
        pre_act_hidden_state = self.fc_hidden_state(concat_xt_ht_minus1)
        ht_candidate = self.tanh(pre_act_hidden_state)
        #print(ht_candidate.shape)

        #create new hidden state, which weights 1 - hidden gate element wise multiplied by prev hidden state
        # added to hidden gate element wise multipled by the candidate hidden
        ht_new = (((1 - zt) * ht_minus1) + (zt * ht_candidate))

        return  ht_new
```

The hidden gate is computed by first concatenating x_t and h_{t-1} , resulting in a 1 x 400 vector. This 1 x 400 vector is multiplied by a weight matrix that is 400 x 100, resulting in a 1 x 100 vector. After a sigmoid is applied to that vector, we have the resulting z_t. In a very similar manner, r_t , the reset gate is computed, resulting in another 1 x 100 vector. This reset gate is factored into computing a candidate hidden state ($h_{t\sim}$). Now, both the hidden gate (z_t) and the candidate hidden state ($h_{t\sim}$) are accessible in the computation of the actual new hidden state in the form of $(1 - z_t)* h_{t-1} + z_t * h_{t\sim}$. We can see that if z_t is close to one, then, the old hidden state (h_{t-1}) will be replaced by the candidate hidden state ($h_{t\sim}$). If z_t is close to 0, then the old hidden state (h_{t-1}) will dominate the output, and it would be like the new word has not contributed much to learning of the semantic meaning (sentiment, in this case) of the review. This mechanism basically obviates the need to hold many hidden states and create an extremely high layer network, which is how the problem of vanishing gradients is someone mitigated in the GRU implementation.

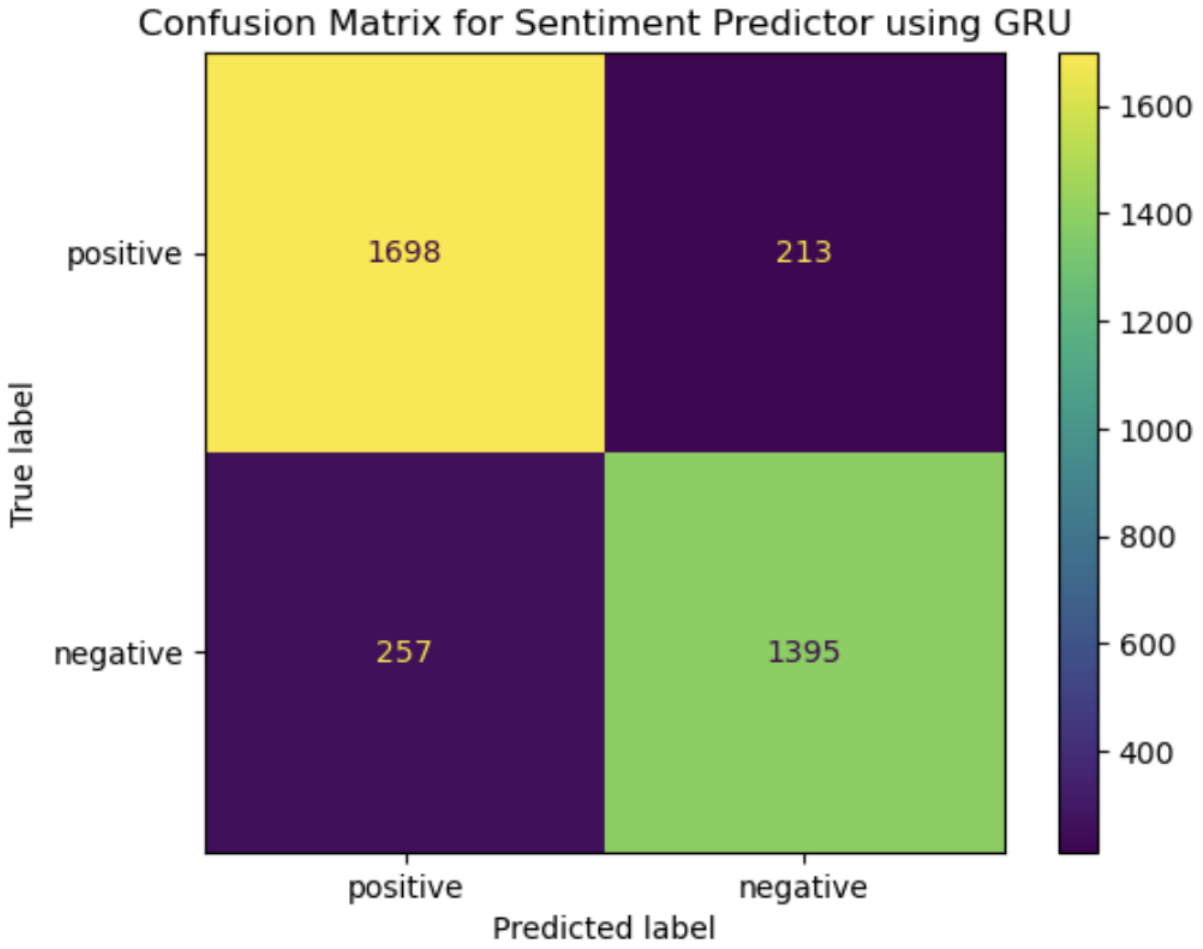
Then, the recurrent network is basically akin to using the new hidden state as input to the GRU network posted above. This is handled in another RNN class, where there is a loop in the forward model to produce and output a two dimensional vector after the last hidden state and word embedding has been passed to the GRU. This recurrent network is posted below:

```
45 class RNN(nn.Module):
46
47     def __init__(self, input_size, hidden_size):
48         super(RNN, self).__init__()
49         self.gru = GruNet(input_size, hidden_size) #instantiate GRU
50         self.fc = nn.Linear(100, 2)
51         self.relu = nn.ReLU()
52         self.logsoftmax = nn.LogSoftmax(dim = 0)
53
54     def forward(self, x):
55         for i, word_emb in enumerate(range(x.shape[0])): #this loop runs GRU for each embedding and hidden vector
56             if i == 0: #no hidden state yet, init to 0:
57                 ht_new = torch.zeros(100)
58                 ht_new = self.gru(x[word_emb], ht_new)
59
60             else:
61                 ht_new = self.gru(x[word_emb], ht_new)
62
63         out = self.fc(self.relu(ht_new)) #take last hidden state vector to output size (2)
64         out = self.logsoftmax(out) #apply logsoftmax activation (before computing NLL Loss outside of the class)
65
66         return out
67
```

Results on own GRU:

Training loss progression:

Confusion Matrix for own GRU:



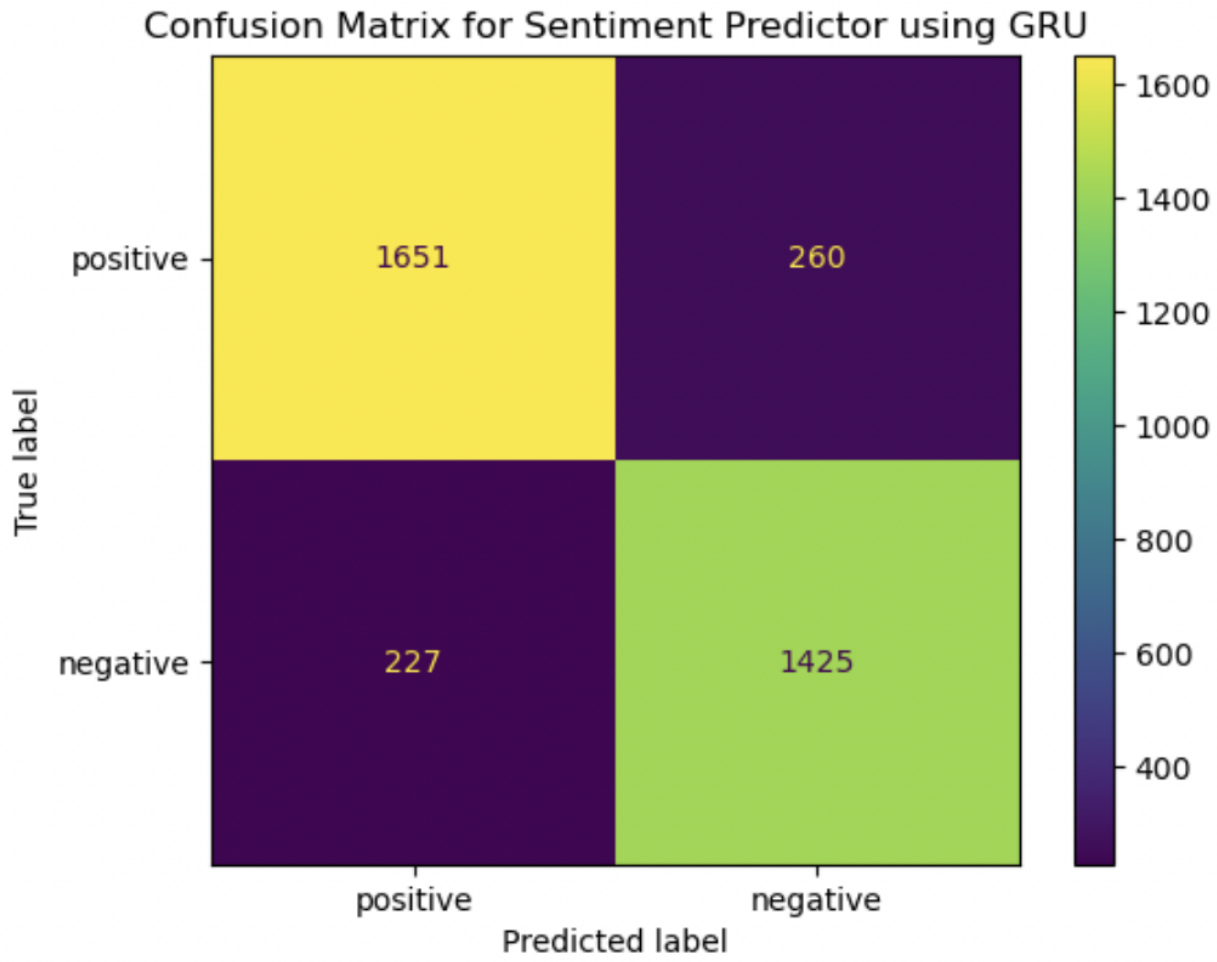
Total Accuracy for own GRU:

```
total acc for pos:        total acc for neg:
0.8885400313971743           0.8444309927360775
total accuracy: 0.868088689306764
```

Part 2: Sentiment Analysis using torch.nn.GRU

Training Loss curve for torch.nn.GRU (bidirectional=False)

Confusion Matrix for nn.GRU (non-bidirectional variant)

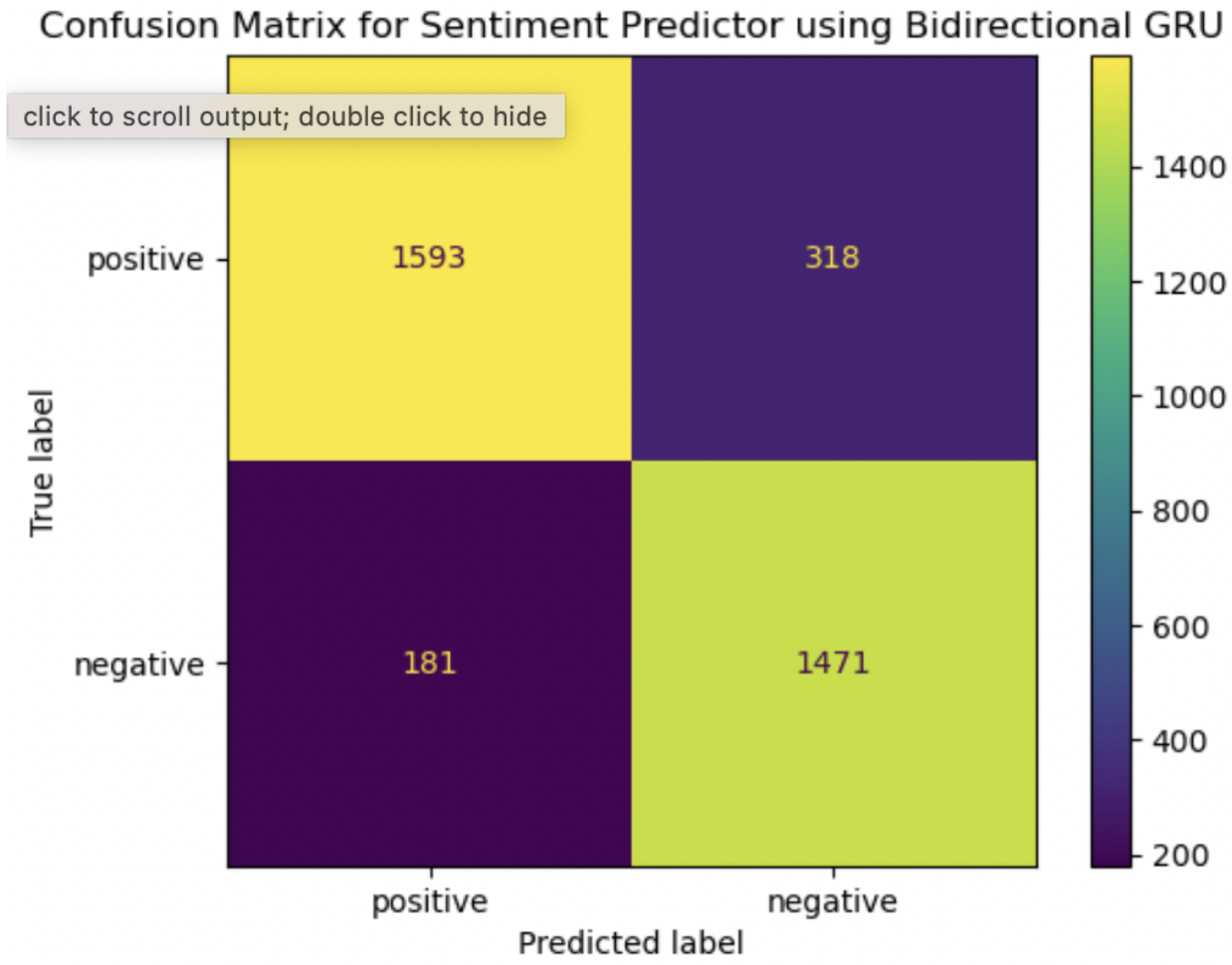


Overall Accuracy for nn.GRU:

```
total acc for pos:        total acc for neg:
0.8639455782312925           0.862590799031477
total accuracy: 0.8633174291327533
```

Part 3: Sentiment Analysis using nn.GRU bidirectional


Loss over Training for nn.GRU bidirectional mode
Loss
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
50
100
150
200
250
Training Iteration



Overall Accuracy for nn.GRU bidirectional:

```
total acc for pos:        total acc for neg:
0.8335949764521193           0.8904358353510896
total accuracy: 0.8599494807746281
```

Part 4: Comparison of all RNNs

Total Accuracy Comparison		
Own GRU	**nn.GRU**	**nn.GRU, bidirectional**
86.8 %	86.3%	85.9%

Every network was trained for four epochs. Overall, all the RNNs achieve an extremely similar total accuracy of around 86%. If one cares most about overall accuracy in identifying positive sentiments correctly, the bidirectional variant would be the best while if one cares most about correctly identifying negative sentiments, the hand crafted RNN may be best. However, the results are so similar that multiple test runs should be run to truly understand if there is a model that sticks out as the “best” RNN.

Part 5: Listing of Source Code

Extracting embeddings via dataloader. Note, most of this is done with the code provided in DL Studio

```
#Implement logic to get the Word2Vec embeddings:
# Use DL studio to get sentiment data we want to train:
import gzip
import os, sys
import pickle
print(os.getcwd())
dataset_archive_train = "data/sentiment_dataset_train_400.tar.gz"
dataset_archive_test =  "data/sentiment_dataset_test_400.tar.gz"
path_to_saved_embeddings = "word2vec_saved_embeddings/"
dataroot = "/Users/alim/Documents/ECE60146/hw8/"




class MiniDLStudio():
    def __init__(self,dataroot, path_saved_model, momentum, learning_rate,
                 epochs, batch_size, classes, use_gpu):
        self.dataroot = dataroot
        self.path_saved_model = path_saved_model
        self.momentum = momentum
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.batch_size = batch_size
        self.classes = classes
        self.use_gpu = use_gpu

dls = MiniDLStudio(
                  dataroot = dataroot,
                  path_saved_model = "./saved_model",
                  momentum = 0.9,
                  learning_rate =  1e-5,
                  epochs = 1,
                  batch_size = 1,
                  classes = ('negative','positive'),
                  use_gpu = False
              )
```

```
class TextClassificationWithEmbeddings(nn.Module):
        """
        The text processing class described previously, TextClassification, was based on
        using one-hot vectors for representing the words.  The main challenge we faced
        with one-hot vectors was that the larger the size of the training dataset, the
        larger the size of the vocabulary, and, therefore, the larger the size of the
        one-hot vectors.  The increase in the size of the one-hot vectors led to a
        model with a significantly larger number of learnable parameters --- and, that,
        in turn, created a need for a still larger training dataset.  Sounds like a classic
        example of a vicious circle.  In this section, I use the idea of word embeddings
        to break out of this vicious circle.

        Word embeddings are fixed-sized numerical representations for words that are
        learned on the basis of the similarity of word contexts.  The original and still
        the most famous of these representations are known as the word2vec
        embeddings. The embeddings that I use in this section consist of pre-trained
        300-element word vectors for 3 million words and phrases as learned from Google
        News reports.  I access these embeddings through the popular Gensim library.

        Class Path:  DLStudio -> TextClassificationWithEmbeddings
        """
        def __init__(self, dl_studio, dataserver_train=None,dataserver_test=None,dataset_file_train=None,dataset_f
            super(TextClassificationWithEmbeddings, self).__init__()
            self.dl_studio = dl_studio
            self.dataserver_train = dataserver_train
            self.dataserver_test = dataserver_test

        class SentimentAnalysisDataset(torch.utils.data.Dataset):
            """
            In relation to the SentimentAnalysisDataset defined for the TextClassification section of
            DLStudio, the __getitem__() method of the dataloader must now fetch the embeddings from
            the word2vec word vectors.

            Class Path:  DLStudio -> TextClassificationWithEmbeddings -> SentimentAnalysisDataset
            """
            def __init__(self, dl_studio, train_or_test, dataset_file, path_to_saved_embeddings=None):
                super(TextClassificationWithEmbeddings.SentimentAnalysisDataset, self).__init__()
                import gensim.downloader as gen_api
#                self.word_vectors = gen_api.load("word2vec-google-news-300")
                self.path_to_saved_embeddings = path_to_saved_embeddings
                self.train_or_test = train_or_test
                root_dir = dl_studio.dataroot
                f = gzip.open(root_dir + dataset_file, 'rb')
```

```
                dataset = f.read()
                if path_to_saved_embeddings is not None:
                    import gensim.downloader as genapi
                    from gensim.models import KeyedVectors
                    if os.path.exists(path_to_saved_embeddings + 'vectors.kv'):
                        self.word_vectors = KeyedVectors.load(path_to_saved_embeddings + 'vectors.kv')
                    else:
                        print("""\n\nSince this is your first time to install the word2vec embeddings, it may take
                              """\na couple of minutes. The embeddings occupy around 3.6GB of your disk space.\n\n
                        self.word_vectors = genapi.load("word2vec-google-news-300")
                        ##  'kv' stands for  "KeyedVectors", a special datatype used by gensim because it
                        ##  has a smaller footprint than dict
                        self.word_vectors.save(path_to_saved_embeddings + 'vectors.kv')
                if train_or_test == 'train':
                    if sys.version_info[0] == 3:
                        self.positive_reviews_train, self.negative_reviews_train, self.vocab = pickle.loads(datase
                    else:
                        self.positive_reviews_train, self.negative_reviews_train, self.vocab = pickle.loads(datase
                    self.categories = sorted(list(self.positive_reviews_train.keys()))
                    self.category_sizes_train_pos = {category : len(self.positive_reviews_train[category]) for cat
                    self.category_sizes_train_neg = {category : len(self.negative_reviews_train[category]) for cat
                    self.indexed_dataset_train = []
                    for category in self.positive_reviews_train:
                        for review in self.positive_reviews_train[category]:
                            self.indexed_dataset_train.append([review, category, 1])
                    for category in self.negative_reviews_train:
                        for review in self.negative_reviews_train[category]:
                            self.indexed_dataset_train.append([review, category, 0])
                    random.shuffle(self.indexed_dataset_train)
                elif train_or_test == 'test':
                    if sys.version_info[0] == 3:
                        self.positive_reviews_test, self.negative_reviews_test, self.vocab = pickle.loads(dataset,
                    else:
                        self.positive_reviews_test, self.negative_reviews_test, self.vocab = pickle.loads(dataset)
                    self.vocab = sorted(self.vocab)
                    self.categories = sorted(list(self.positive_reviews_test.keys()))
                    self.category_sizes_test_pos = {category : len(self.positive_reviews_test[category]) for categ
                    self.category_sizes_test_neg = {category : len(self.negative_reviews_test[category]) for categ
                    self.indexed_dataset_test = []
                    for category in self.positive_reviews_test:
                        for review in self.positive_reviews_test[category]:
                            self.indexed_dataset_test.append([review, category, 1])
                    for category in self.negative_reviews_test:
                        for review in self.negative_reviews_test[category]:
                            self.indexed_dataset_test.append([review, category, 0])
```

```
                    else:
                        next
                review_tensor = torch.FloatTensor( list_of_embeddings )
                return review_tensor

            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 __len__(self):
                if self.train_or_test == 'train':
                    return len(self.indexed_dataset_train)
                elif self.train_or_test == 'test':
                    return len(self.indexed_dataset_test)

            def __getitem__(self, idx):
                sample = self.indexed_dataset_train[idx] if self.train_or_test == 'train' else self.indexed_datase
                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

        def load_SentimentAnalysisDataset(self, dataserver_train, dataserver_test ):
            self.train_dataloader = torch.utils.data.DataLoader(dataserver_train,
                        batch_size=self.dl_studio.batch_size,shuffle=True, num_workers=2) #wraps the dataset in th
            self.test_dataloader = torch.utils.data.DataLoader(dataserver_test,
                                batch_size=self.dl_studio.batch_size,shuffle=False, num_workers=2)
```

```
        def load_SentimentAnalysisDataset(self, dataserver_train, dataserver_test ):
            self.train_dataloader = torch.utils.data.DataLoader(dataserver_train,
                        batch_size=self.dl_studio.batch_size,shuffle=True, num_workers=2) #wraps the dataset in th
            self.test_dataloader = torch.utils.data.DataLoader(dataserver_test,
                                batch_size=self.dl_studio.batch_size,shuffle=False, num_workers=2)


text_cl = TextClassificationWithEmbeddings(dl_studio = dls)
dataserver_train = text_cl.SentimentAnalysisDataset(dl_studio = dls, train_or_test = 'train',
                                                    dataset_file = dataset_archive_train,
                                                     path_to_saved_embeddings = path_to_saved_embeddings)

dataserver_test = TextClassificationWithEmbeddings.SentimentAnalysisDataset(
                                train_or_test = 'test',
                                dl_studio = dls,
                                dataset_file = dataset_archive_test,
                                path_to_saved_embeddings = path_to_saved_embeddings
                  )
text_cl.dataserver_train = dataserver_train #assigns the dataserver_train to self.dataserver_train
text_cl.dataserver_test = dataserver_test #assigns the dataserver_test to self.dataserver_test

text_cl.load_SentimentAnalysisDataset(dataserver_train, dataserver_test)
```

Training and Eval Code for Own GRU:

```python
#Training Loop for handcrafted GRU:
#GRUNet = GruNet(input_size = 300, hidden_size = 100)
rnn = RNN(input_size =300, hidden_size =100)
epochs = 4
criterion = nn.NLLLoss()
optimizer = optim.Adam(rnn.parameters(),  lr=dls.learning_rate)
running_loss = 0.0
training_loss_tally_ownGRU = []
start_time = time.perf_counter()
for epoch in range(epochs):
    for n, data in enumerate(text_cl.dataserver_train):
        review = data['review'] #review is a tensor of embeddings
        sentiment = data['sentiment'] #sentiment is a tensor of GT sentiment:
        out = rnn(review)
        target = torch.argmax(sentiment)
        loss = criterion(out, target)
        running_loss += loss.item()
        loss.backward()
        optimizer.step()
        if n % 200 == 199:
            avg_loss = running_loss / float(200)
            training_loss_tally_ownGRU.append(avg_loss)
            current_time = time.perf_counter()
            time_elapsed = current_time-start_time
            print("[epoch:%d  iter:%4d  elapsed_time:%4d secs]     loss: %.5f" % (epoch+1,n+1, time_elapsed,avg_los
            running_loss = 0.0

```

```python
### Prediction / Eval Code for my own GRU:
correct_pos_sentiment = 0.0
total_pos_sentiment = 0.0
correct_neg_sentiment = 0.0
total_neg_sentiment = 0.0
sentiment_pred = []
sentiment_label = []
mapping = { 0 : 'negative', 1: 'positive'}
with torch.no_grad():
    for n, data in enumerate(text_cl.dataserver_test):

            #print('n is', n)
            sentiment = data['sentiment']
            review = data['review']
            out = rnn(review) #goes through one sequence of embeddings (i.e one review)

            GT = torch.argmax(sentiment).item()
            sentiment_label.append(mapping[GT])
            pred = torch.argmax(out).item()
            sentiment_pred.append(mapping[pred])
            if (GT == 1): #compute positive sentiment accuracy
                total_pos_sentiment += 1
                if pred == GT:
                    #print('accurate pos sentiment pred')
                    correct_pos_sentiment += 1

            if (GT == 0): #compute negative sentiment accuracy
                total_neg_sentiment += 1
                if pred == GT:
                    #print('accurate neg sentiment pred')
                    correct_neg_sentiment += 1


    print("total acc for pos:        total acc for neg:")
    print(correct_pos_sentiment / total_pos_sentiment, '         ', correct_neg_sentiment / total_neg_sentiment)
    print('total accuracy:', (correct_pos_sentiment + correct_neg_sentiment) / len(sentiment_label))

from sklearn.metrics import confusion_matrix
y_true = sentiment_label
y_pred = sentiment_pred
print(len(y_true), len(y_pred))
confusion_matrix=confusion_matrix(y_true, y_pred, labels = [ "positive", "negative"])
disp = ConfusionMatrixDisplay(confusion_matrix, display_labels = [ "positive", "negative"])
disp.plot()
disp.ax_.set_title("Confusion Matrix for Sentiment Predictor using own GRU")
plt.show()

```

Network, Training and Eval Code for nn.GRU:

```python
class GRUNetPyTorch(nn.Module):
    """
    For this embeddings adapted version of the GRUnet shown earlier, we can assume that
    the 'input_size' for a tensor representing a word is always 300.
    Source: https://blog.floydhub.com/gru-with-pytorch/
    with the only modification that the final output of forward() is now
    routed through LogSoftmax activation.

    Class Path:  DLStudio -> TextClassificationWithEmbeddings -> GRUnetWithEmbeddings
    """
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        """
        -- input_size is the size of the tensor for each word in a sequence of words.  If you word2vec
              embedding, the value of this variable will always be equal to 300.
        -- hidden_size is the size of the hidden state in the RNN
        -- output_size is the size of output of the RNN.  For binary classification of
              input text, output_size is 2.
        -- num_layers creates a stack of GRUs
        """
        super(GRUNetPyTorch, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.output_size = output_size
        self.gru = nn.GRU(self.input_size, self.hidden_size, self.num_layers)
        self.fc = nn.Linear(self.hidden_size, self.output_size)
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x, h):
        #print("in input: x shp and then h shape:", x.shape, h.shape)
        out, h = self.gru(x, h)
        #print("after running through self.gru:", out.shape, h.shape)
        #print("out[:,-1].shape", out[:,-1].shape)
        out = self.fc(self.relu(out[:,-1]))

        #print("after relu applied to out [:,-1] and a fc(hidden size --> output size):", out.shape)
        out = self.logsoftmax(out)
        #print("output after logsoftmax:", out.shape)
        return out, h

    def init_hidden(self):
        weight = next(self.parameters()).data
        #                  num_layers  batch_size    hidden_size
        hidden = weight.new(  2,          1,         self.hidden_size    ).zero_()
        return hidden

```

```
model = GRUNetPyTorch(input_size = 300, hidden_size = 100, output_size = 2, num_layers=2)
epochs = 4
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(),  lr=dls.learning_rate)
running_loss = 0.0
training_loss_tally_nnGRU = []
start_time = time.perf_counter()

for epoch in range(epochs):
    for n, data in enumerate(text_cl.dataserver_train):
        review = data['review']
        sentiment = data['sentiment']
        h = model.init_hidden()
        #print('h.shape:',h.shape)
        #input to an instance of torch.nn.GRU = tensor with shape seq len, batchsize, input_size

        review = torch.unsqueeze(review, 1) #reshape review, which originally is in seq len, input size shape
        #to have a first axis with size 1.

        out, h = model(review, h)
        target = torch.argmax(sentiment)
        #print(out.shape, h.shape)
        #compute loss on output from GRU
        loss = criterion(out[-1,:], target)
        running_loss += loss.item()
        loss.backward()
        optimizer.step()
        if n % 200 == 199:
            avg_loss = running_loss / float(200)
            training_loss_tally_nnGRU.append(avg_loss)
            current_time = time.perf_counter()
            time_elapsed = current_time-start_time
            print("[epoch:%d  iter:%4d  elapsed_time:%4d secs]     loss: %.5f" % (epoch+1,n+1, time_elapsed,avg_los
            running_loss = 0.0

```

```python
### Prediction / Eval Code for nn.GRU model
correct_pos_sentiment = 0.0
total_pos_sentiment = 0.0
correct_neg_sentiment = 0.0
total_neg_sentiment = 0.0
sentiment_pred = []
sentiment_label = []
mapping = { 0 : 'negative', 1: 'positive'}
with torch.no_grad():
    for n, data in enumerate(text_cl.dataserver_test):

            #print('n is', n)
            sentiment = data['sentiment']
            review = data['review']
            h = model.init_hidden()
            review = torch.unsqueeze(review, 1)
            out, h = model(review, h) #goes through one sequence of embeddings (i.e one review)

            GT = torch.argmax(sentiment).item()
            sentiment_label.append(mapping[GT])
            pred = torch.argmax(out[-1, :]).item()
            sentiment_pred.append(mapping[pred])
            if (GT == 1): #compute positive sentiment accuracy
                total_pos_sentiment += 1
                if pred == GT:
                    #print('accurate pos sentiment pred')
                    correct_pos_sentiment += 1

            if (GT == 0): #compute negative sentiment accuracy
                total_neg_sentiment += 1
                if pred == GT:
                    #print('accurate neg sentiment pred')
                    correct_neg_sentiment += 1


    print("total acc for pos:        total acc for neg:")
    print(correct_pos_sentiment / total_pos_sentiment, '         ', correct_neg_sentiment / total_neg_sentiment)
    print('total accuracy:', (correct_pos_sentiment + correct_neg_sentiment) / len(sentiment_label))

from sklearn.metrics import confusion_matrix
y_true = sentiment_label
y_pred = sentiment_pred
print(len(y_true), len(y_pred))
confusion_matrix=confusion_matrix(y_true, y_pred, labels = [ "positive", "negative"])
disp = ConfusionMatrixDisplay(confusion_matrix, display_labels = [ "positive", "negative"])
disp.plot()
disp.ax_.set_title("Confusion Matrix for Sentiment Predictor using nn.GRU - bidirectional=False")
plt.show()
```

Network, Training, and Evaluation code for Bidirectional GRU:

```python
### Train using PyTorch's bidirectional GRU:

class GRUNetPyTorch_BD(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        """
        -- input_size is the size of the tensor for each word in a sequence of words.  If you word2vec
               embedding, the value of this variable will always be equal to 300.
        -- hidden_size is the size of the hidden state in the RNN
        -- output_size is the size of output of the RNN.  For binary classification of
               input text, output_size is 2.
        -- num_layers creates a stack of GRUs
        """
        super(GRUNetPyTorch_BD, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.output_size = output_size
        self.gru = nn.GRU(self.input_size, self.hidden_size, self.num_layers, bidirectional = True)
        self.fc = nn.Linear(self.hidden_size * 2, self.output_size)
        self.relu = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x, h):
        #print("in input: x shp and then h shape:", x.shape, h.shape)
        out, h = self.gru(x, h)
        #print("after running through self.gru:", out.shape, h.shape)
        #print("out[:,-1].shape", out[:,-1].shape)
        out = self.fc(self.relu(out[:,-1]))

        #print("after relu applied to out [:,-1] and a fc(hidden size --> output size):", out.shape)
        out = self.logsoftmax(out)
        #print("output after logsoftmax:", out.shape)
        return out, h

    def init_hidden(self):
        weight = next(self.parameters()).data
        #                 num_layers  batch_size    hidden_size
        # I think that the first argument in this case refers to num_dirs
        hidden = weight.new(  self.num_layers *2,          1,         self.hidden_size    ).zero_()
        return hidden

```

```
model_bidirectional = GRUNetPyTorch_BD(input_size = 300, hidden_size = 100, output_size = 2, num_layers=2)
epochs = 4
criterion = nn.NLLLoss()
optimizer = optim.Adam(model_bidirectional.parameters(),  lr=dls.learning_rate)
running_loss = 0.0
training_loss_tally_nnGRU_bidirectional = []
start_time = time.perf_counter()

for epoch in range(epochs):
    for n, data in enumerate(text_cl.dataserver_train):
        review = data['review']
        sentiment = data['sentiment']

        #input to a GRU instance will have the shape num of embeddings, batch size, embedding size
        review = torch.unsqueeze(review, 1)
        #print(review.shape)

        #init hidden state. When using bidirectional mode, we need to have hidden be the shape of num dirs, batch s
        hidden = model_bidirectional.init_hidden()
        #print(hidden.shape)

        out, h = model_bidirectional(review, hidden)
        #print("out shape:", out.shape)
        #print("h shape", h.shape)
        #print(out[:,-1].shape)

        target = torch.argmax(sentiment)

        #compute loss:
        loss = criterion(out[-1,:] ,target)
        running_loss += loss.item()

        #compute gradients, backprop
        loss.backward()

        #gradient step:
        optimizer.step()

        if (n % 200) == 199:
            avg_loss = running_loss / float(200)
            training_loss_tally_nnGRU_bidirectional.append(avg_loss)
            current_time = time.perf_counter()
            time_elapsed = current_time-start_time
            print("[epoch:%d  iter:%4d  elapsed_time:%4d secs]     loss: %.5f" % (epoch+1,n+1, time_elapsed,avg_los
            running_loss = 0.0
```

```
### Prediction / Eval Code for nn.GRU model
correct_pos_sentiment = 0.0
total_pos_sentiment = 0.0
correct_neg_sentiment = 0.0
total_neg_sentiment = 0.0
sentiment_pred = []
sentiment_label = []
mapping = { 0 : 'negative', 1: 'positive'}
with torch.no_grad():
    for n, data in enumerate(text_cl.dataserver_test):

            #print('n is', n)
            sentiment = data['sentiment']
            review = data['review']
            h = model_bidirectional.init_hidden()
            review = torch.unsqueeze(review, 1)
            out, h = model_bidirectional(review, h) #goes through one sequence of embeddings (i.e one review)

            GT = torch.argmax(sentiment).item()
            sentiment_label.append(mapping[GT])
            pred = torch.argmax(out[-1, :]).item()
            sentiment_pred.append(mapping[pred])
            if (GT == 1): #compute positive sentiment accuracy
                total_pos_sentiment += 1
                if pred == GT:
                    #print('accurate pos sentiment pred')
                    correct_pos_sentiment += 1

            if (GT == 0): #compute negative sentiment accuracy
                total_neg_sentiment += 1
                if pred == GT:
                    #print('accurate neg sentiment pred')
                    correct_neg_sentiment += 1


    print("total acc for pos:        total acc for neg:")
    print(correct_pos_sentiment / total_pos_sentiment, '         ', correct_neg_sentiment / total_neg_sentiment)
    print('total accuracy:', (correct_pos_sentiment + correct_neg_sentiment) / len(sentiment_label))

from sklearn.metrics import confusion_matrix
y_true = sentiment_label
y_pred = sentiment_pred
print(len(y_true), len(y_pred))
confusion_matrix=confusion_matrix(y_true, y_pred, labels = [ "positive", "negative"])
disp = ConfusionMatrixDisplay(confusion_matrix, display_labels = [ "positive", "negative"])
disp.plot()
disp.ax_.set_title("Confusion Matrix for Sentiment Predictor using Bidirectional GRU")
plt.show()
```