ECE695DL: Homework 7

Kevin Bridgman

26 April 2021

```
#!/usr/bin/env python

##  text_classification_with_GRU.py

"""
This script is an attempt at solving the sentiment
↪  classification problem
with a GRU based recurrence to get around the problem of
↪  vanishing gradients
that are common to neural networks with feedback.
"""

import random
import numpy
import torch
import os, sys
sys.path.append( "..//DLStudio" )
from DLStudio import *


seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)


##  watch -d -n 0.5 nvidia-smi
```

```
#great my custom class that inherits DLStudio's
↪  TextClassification class.
#ref: https://engineering.purdue.edu/kak/distDLS/

class EncodedTextClassification(DLStudio.TextClassification):

    class EncodedSentimentAnalysisDataset(DLStudio.TextClassific⌋
    ↪  ation.SentimentAnalysisDataset):
        #need to at a minimum modify __getitem__() and
        ↪  get_vocab_size()
        #more specifically, rewrite review_to_tensor() to use
        ↪  another custom function instead of
        #one_hotvec_for_word() to get input vector for word.
        ↪  Also change get_vocab_size to reflect
        #the new vector length
        def get_vocab_size(self):
            length = len(self.vocab)
            #I need to check and make sure I didn't break
            ↪  something by changing the return
            #from the true length to ceil(ln(length)). Need to
            ↪  search DL studios TextClassifcation class
            #for all calls to this function
            return int(np.ceil(np.log2(length)))

        def one_hotvec_for_word(self, word):
            word_index =  self.vocab.index(word)
            binary_representation =
            ↪  int(np.ceil(np.log2(word_index+1)))
            #hotvec = torch.zeros(1, len(self.vocab))
            #hotvec[0, word_index] = 1
            #return hotvec
            #vocab size must be less than 2^16
            #'{0:016b}'.format(binary_representation)
            input_vector = torch.zeros(1,
            ↪  int(np.ceil(np.log2(len(self.vocab)))))
            string_representation = bin(binary_representation)[2⌋
            ↪  :].zfill(self.get_vocab_size())
            for i, char in enumerate(string_representation):
                #print("index: ",i)
                #print("char: ",char)
                input_vector[0,i] = int(char)

            return(input_vector)

        def review_to_tensor(self, review):
```

```
            review_tensor = torch.zeros(len(review),
            ↪  int(np.ceil(np.log2(len(self.vocab)))))
            for i,word in enumerate(review):
                review_tensor[i,:] =
                ↪  self.one_hotvec_for_word(word)
            return review_tensor
```

```
dls = DLStudio(
#                   dataroot =
↪  "/home/kak/TextDatasets/sentiment_dataset/",
                  dataroot =
                  ↪  "./data/TextDatasets/sentiment_dataset/",
                  path_saved_model = "./saved_model",
                  momentum = 0.9,
                  learning_rate =  1e-5,
                  epochs = 5,
                  batch_size = 1,
                  classes = ('negative','positive'),
                  use_gpu = True,
              )

text_cl = EncodedTextClassification( dl_studio = dls )
dataserver_train =
↪  EncodedTextClassification.EncodedSentimentAnalysisDataset(
                                  train_or_test = 'train',
                                  dl_studio = dls,
                                  #dataset_file = "sentiment_data
                                  ↪  set_train_40.tar.gz",
                                  dataset_file = "sentiment_datas
                                  ↪  et_train_200.tar.gz",
                  )
dataserver_test =
↪  EncodedTextClassification.EncodedSentimentAnalysisDataset(
                                  train_or_test = 'test',
                                  dl_studio = dls,
                                  #dataset_file = "sentiment_data
                                  ↪  set_test_40.tar.gz",
                                  dataset_file = "sentiment_datas
                                  ↪  et_test_200.tar.gz",
                  )
text_cl.dataserver_train = dataserver_train
text_cl.dataserver_test = dataserver_test
```

```
text_cl.load_SentimentAnalysisDataset(dataserver_train,
↪  dataserver_test)

vocab_size = dataserver_train.get_vocab_size()

model = text_cl.GRUnet(vocab_size, hidden_size=512,
↪  output_size=2, n_layers=2)

number_of_learnable_params = sum(p.numel() for p in
↪  model.parameters() if p.requires_grad)

num_layers = len(list(model.parameters()))

print("\n\nThe number of layers in the model: %d" % num_layers)
print("The number of learnable parameters in the model: %d" %
↪  number_of_learnable_params)
print("The size of the input vector: %d" % vocab_size)

## TRAINING:
text_cl.run_code_for_training_for_text_classification_with_GRU(m
↪  odel,
↪  hidden_size=512)

## TESTING:
#import pymsgbox
#response = pymsgbox.confirm("Finished training.  Start testing
↪  on unseen data?")
#if response == "OK":
text_cl.run_code_for_testing_text_classification_with_GRU(model,
↪  hidden_size=512)
```

```
#!/usr/bin/env python

##  text_classification_with_GRU.py

"""
This script is an attempt at solving the sentiment
↪  classification problem
```

```
with a GRU based recurrence to get around the problem of
↪  vanishing gradients
that are common to neural networks with feedback.
"""

import random
import numpy
import torch
import os, sys
sys.path.append( "..//DLStudio" )
from DLStudio import *


seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)


##  watch -d -n 0.5 nvidia-smi

#great my custom class that inherits DLStudio's
↪  TextClassification class.
#ref: https://engineering.purdue.edu/kak/distDLS/

#also using Gabriel Loye's GRU code linked from the homework
↪  prompt.
#ref: https://github.com/gabrielloye/GRU_Prediction/blob/master/
↪  main.ipynb

class GabrielTextClassification(DLStudio.TextClassification):

    class GabrielSentimentAnalysisDataset(DLStudio.TextClassific
    ↪  ation.SentimentAnalysisDataset):
        #need to at a minimum modify __getitem__() and
        ↪  get_vocab_size()
        #more specifically, rewrite review_to_tensor() to use
        ↪  another custom function instead of
        #one_hotvec_for_word() to get input vector for word.
        ↪  Also change get_vocab_size to reflect
        #the new vector length
        def get_vocab_size(self):
```

```
        #length = len(self.vocab)
        #I need to check and make sure I didn't break
        ↪  something by changing the return
        #from the true length to ceil(ln(length)). Need to
        ↪  search DL studios TextClassifcation class
        #for all calls to this function
        return 1

    def one_hotvec_for_word(self, word):
        word_index =  self.vocab.index(word)
        binary_representation =
        ↪  int(np.ceil(np.log2(word_index+1)))
        #hotvec = torch.zeros(1, len(self.vocab))
        #hotvec[0, word_index] = 1
        #return hotvec
        #vocab size must be less than 2^16
        #'{0:016b}'.format(binary_representation)
        input_vector = torch.zeros(1,
        ↪  int(np.ceil(np.log2(len(self.vocab)))))
        string_representation = bin(binary_representation)[2
        ↪  :].zfill(self.get_vocab_size())
        for i, char in enumerate(string_representation):
            #print("index: ",i)
            #print("char: ",char)
            input_vector[0,i] = int(char)

        return(word_index)

    def review_to_tensor(self, review):
        review_tensor = torch.zeros(len(review), int(1))
        for i,word in enumerate(review):
            review_tensor[i,:] =
            ↪  self.one_hotvec_for_word(word)
        return review_tensor

class GabrielGRUnet(nn.Module): #this is taken from
↪  https://github.com/gabrielloye/GRU_Prediction/blob/maste
↪  r/main.ipynb

    def __init__(self, input_dim, hidden_dim, output_dim,
    ↪  n_layers, drop_prob=0.2):
        super(GabrielTextClassification.GabrielGRUnet,
        ↪  self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
```

```
        self.gru = nn.GRU(input_dim, hidden_dim, n_layers,
        ↪  batch_first=True, dropout=drop_prob)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x, h):
        out, h = self.gru(x, h)
        out = self.fc(self.relu(out[:,-1]))
        return out, h

    def init_hidden(self, batch_size):
        weight = next(self.parameters()).data
        hidden = weight.new(self.n_layers, batch_size,
        ↪  self.hidden_dim).zero_()
        return hidden

def run_training_for_GabrielTextClassification(self, model,
↪  hidden_dim):  #this is taken from https://github.com/gab⌋
↪  rielloye/GRU_Prediction/blob/master/main.ipynb

    model = model.to(self.dl_studio.device)
    # Defining loss function and optimizer
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(),
    ↪  lr=self.dl_studio.learning_rate)

    model.train()
    print("Starting Training of GRU model")
    epoch_times = []
    # Start training loop
    for epoch in range(self.dl_studio.epochs):
        start_time = time.clock()
        h = model.init_hidden(self.dl_studio.batch_size).to(⌋
        ↪  self.dl_studio.device)
        avg_loss = 0.
        counter = 0
        #for x, label in train_loader:
        for i, data in enumerate(self.train_dataloader):
            x,category,label = data['review'],
            ↪  data['category'], data['sentiment']
            counter += 1
            h = h.data
            model.zero_grad()

            out, h =
            ↪  model(x.to(self.dl_studio.device).float(), h)
```

```
                loss = criterion(out,
                ↪  label.to(self.dl_studio.device).float())
                loss.backward()
                optimizer.step()
                avg_loss += loss.item()
                if counter%200 == 0:
                    print("Epoch {}......Step: {}/{}.......
                    ↪  Average Loss for Epoch:
                    ↪  {}".format(epoch, counter,
                    ↪  len(self.train_dataloader),
                    ↪  avg_loss/counter))
            current_time = time.clock()
            print("Epoch {}/{} Done, Total Loss:
            ↪  {}".format(epoch, self.dl_studio.epochs,
            ↪  avg_loss/len(self.train_dataloader)))
            print("Time Elapsed for Epoch: {}
            ↪  seconds".format(str(current_time-start_time)))
            epoch_times.append(current_time-start_time)
        print("Total Training Time: {}
        ↪  seconds".format(str(sum(epoch_times))))
        self.save_model(model)

    def run_testing_for_GabrielTextClassification(self, net,
    ↪  hidden_dim):
        net.load_state_dict(torch.load(self.dl_studio.path_saved
        ↪  _model))
        classification_accuracy = 0.0
        negative_total = 0
        positive_total = 0
        confusion_matrix = torch.zeros(2,2)
        net.eval()
        with torch.no_grad():
            for i, data in enumerate(self.test_dataloader):
                review_tensor,category,sentiment =
                ↪  data['review'], data['category'],
                ↪  data['sentiment']
                hidden = net.init_hidden(1)
                #for k in range(review_tensor.shape[1]):
                    #output, hidden = net(torch.unsqueeze(torch.
                    ↪  unsqueeze(review_tensor[0,k],0),0),
                    ↪  hidden)
                output, hidden =
                ↪  net(review_tensor.to(self.dl_studio.device),
                ↪  hidden.to(self.dl_studio.device))
                predicted_idx = torch.argmax(output).item()
                gt_idx = torch.argmax(sentiment).item()
```

```
        if i % 100 == 99:
            print("   [i=%d]    predicted_label=%d
            ↪  gt_label=%d\n\n" % (i+1,
            ↪  predicted_idx,gt_idx))
        if predicted_idx == gt_idx:
            classification_accuracy += 1
        if gt_idx == 0:
            negative_total += 1
        elif gt_idx == 1:
            positive_total += 1
        confusion_matrix[gt_idx,predicted_idx] += 1
print("\nOverall classification accuracy: %0.2f%%" %
↪  (float(classification_accuracy) * 100 /float(i)))
out_percent = np.zeros((2,2), dtype='float')
out_percent[0,0] = "%.3f" % (100 * confusion_matrix[0,0]
↪  / float(negative_total))
out_percent[0,1] = "%.3f" % (100 * confusion_matrix[0,1]
↪  / float(negative_total))
out_percent[1,0] = "%.3f" % (100 * confusion_matrix[1,0]
↪  / float(positive_total))
out_percent[1,1] = "%.3f" % (100 * confusion_matrix[1,1]
↪  / float(positive_total))
print("\n\nNumber of positive reviews tested: %d" %
↪  positive_total)
print("\n\nNumber of negative reviews tested: %d" %
↪  negative_total)
print("\n\nDisplaying the confusion matrix:\n")
out_str = "                          "
out_str +=  "%18s    %18s" % ('predicted negative',
↪  'predicted positive')
print(out_str + "\n")
for i,label in enumerate(['true negative', 'true
↪  positive']):
    out_str = "%12s:  " % label
    for j in range(2):
        out_str +=  "%18s" % out_percent[i,j]
    print(out_str)
#self.run_code_for_testing_text_classification_with_GRU(
↪  model,
↪  hidden_dim)
#model.eval()
#outputs = []
#targets = []
#start_time = time.clock()
##for i in test_x.keys():
#for i, data in enumerate(self.test_dataloader):
```

```
        #    inp,category,labs = data['review'],
        ↪  data['category'], data['sentiment']
        #    #inp = torch.from_numpy(np.array(test_x[i]))
        #    #labs = torch.from_numpy(np.array(test_y[i]))
        #    h = model.init_hidden(inp.shape[0])
        #    out, h = model(inp.to(device).float(), h)
        #    outputs.append(label_scalers[i].inverse_transform(o
        ↪  ut.cpu().detach().numpy()).reshape(-1))
        #    targets.append(label_scalers[i].inverse_transform(l
        ↪  abs.numpy()).reshape(-1))
        #print("Evaluation Time:
        ↪  {}".format(str(time.clock()-start_time)))
        #sMAPE = 0
        #for i in range(len(outputs)):
        #    sMAPE += np.mean(abs(outputs[i]-targets[i])/(target
        ↪  s[i]+outputs[i])/2)/len(outputs)
        #print("sMAPE: {}%".format(sMAPE*100))
        #return outputs, targets, sMAPE

dls = DLStudio(
#                  dataroot =
↪  "/home/kak/TextDatasets/sentiment_dataset/",
                 dataroot =
                 ↪  "./data/TextDatasets/sentiment_dataset/",
                 path_saved_model = "./saved_model",
                 momentum = 0.9,
                 learning_rate =  1e-5,
                 epochs = 5,
                 batch_size = 1,
                 classes = ('negative','positive'),
                 use_gpu = True,
              )


text_cl = GabrielTextClassification( dl_studio = dls )
dataserver_train =
↪  GabrielTextClassification.GabrielSentimentAnalysisDataset(
                                train_or_test = 'train',
                                dl_studio = dls,
                                #dataset_file = "sentiment_data
                                ↪  set_train_40.tar.gz",
                                dataset_file = "sentiment_datas
                                ↪  et_train_200.tar.gz",
                 )
dataserver_test =
↪  GabrielTextClassification.GabrielSentimentAnalysisDataset(
```

```
                                    train_or_test = 'test',
                                    dl_studio = dls,
                                    #dataset_file = "sentiment_data
                                    ↪  set_test_40.tar.gz",
                                    dataset_file = "sentiment_datas
                                    ↪  et_test_200.tar.gz",
                    )
text_cl.dataserver_train = dataserver_train
text_cl.dataserver_test = dataserver_test

text_cl.load_SentimentAnalysisDataset(dataserver_train,
↪  dataserver_test)

vocab_size = dataserver_train.get_vocab_size()

model = text_cl.GabrielGRUnet(vocab_size, hidden_dim=512,
↪  output_dim=2, n_layers=2)

number_of_learnable_params = sum(p.numel() for p in
↪  model.parameters() if p.requires_grad)

num_layers = len(list(model.parameters()))

print("\n\nThe number of layers in the model: %d" % num_layers)
print("\nThe number of learnable parameters in the model: %d" %
↪  number_of_learnable_params)
print("\nThe size of the input vector: %d" % vocab_size)

## TRAINING:
text_cl.run_training_for_GabrielTextClassification(model,
↪  hidden_dim=512)

## TESTING:
#import pymsgbox
#response = pymsgbox.confirm("Finished training.  Start testing
↪  on unseen data?")
#if response == "OK":
text_cl.run_testing_for_GabrielTextClassification(model,
↪  hidden_dim=512)
```

```
#!/usr/bin/env python

##  text_classification_with_GRU.py

"""
This script is an attempt at solving the sentiment
↪  classification problem
with a GRU based recurrence to get around the problem of
↪  vanishing gradients
that are common to neural networks with feedback.
"""

import random
import numpy
import torch
import os, sys
sys.path.append( "..//DLStudio" )
from DLStudio import *


seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)


##  watch -d -n 0.5 nvidia-smi

#great my custom class that inherits DLStudio's
↪  TextClassification class.
#ref: https://engineering.purdue.edu/kak/distDLS/

class IntTextClassification(DLStudio.TextClassification):

    class IntSentimentAnalysisDataset(DLStudio.TextClassificatio⌋
    ↪  n.SentimentAnalysisDataset):
        #need to at a minimum modify __getitem__() and
        ↪  get_vocab_size()
        #more specifically, rewrite review_to_tensor() to use
        ↪  another custom function instead of
        #one_hotvec_for_word() to get input vector for word.
        ↪  Also change get_vocab_size to reflect
```

```
        #the new vector length
        def get_vocab_size(self):
            #length = len(self.vocab)
            #I need to check and make sure I didn't break
            ↪  something by changing the return
            #from the true length to ceil(ln(length)). Need to
            ↪  search DL studios TextClassifcation class
            #for all calls to this function
            return 1

        def one_hotvec_for_word(self, word):
            word_index =  self.vocab.index(word)
            binary_representation =
            ↪  int(np.ceil(np.log2(word_index+1)))
            #hotvec = torch.zeros(1, len(self.vocab))
            #hotvec[0, word_index] = 1
            #return hotvec
            #vocab size must be less than 2^16
            #'{0:016b}'.format(binary_representation)
            input_vector = torch.zeros(1,
            ↪  int(np.ceil(np.log2(len(self.vocab)))))
            string_representation = bin(binary_representation)[2
            ↪  :].zfill(self.get_vocab_size())
            for i, char in enumerate(string_representation):
                #print("index: ",i)
                #print("char: ",char)
                input_vector[0,i] = int(char)

            return(word_index)

        def review_to_tensor(self, review):
            review_tensor = torch.zeros(len(review), int(1))
            for i,word in enumerate(review):
                review_tensor[i,:] =
                ↪  self.one_hotvec_for_word(word)
            return review_tensor




dls = DLStudio(
#                  dataroot =
↪  "/home/kak/TextDatasets/sentiment_dataset/",
                 dataroot =
                 ↪  "./data/TextDatasets/sentiment_dataset/",
                 path_saved_model = "./saved_model",
```

```
                momentum = 0.9,
                learning_rate =  1e-5,
                epochs = 5,
                batch_size = 1,
                classes = ('negative','positive'),
                use_gpu = True,
            )


text_cl = IntTextClassification( dl_studio = dls )
dataserver_train =
↪  IntTextClassification.IntSentimentAnalysisDataset(
                                train_or_test = 'train',
                                dl_studio = dls,
                                #dataset_file = "sentiment_data
                                ↪  set_train_40.tar.gz",
                                dataset_file = "sentiment_datas
                                ↪  et_train_200.tar.gz",
                )
dataserver_test =
↪  IntTextClassification.IntSentimentAnalysisDataset(
                                train_or_test = 'test',
                                dl_studio = dls,
                                #dataset_file = "sentiment_data
                                ↪  set_test_40.tar.gz",
                                dataset_file = "sentiment_datas
                                ↪  et_test_200.tar.gz",
                )
text_cl.dataserver_train = dataserver_train
text_cl.dataserver_test = dataserver_test

text_cl.load_SentimentAnalysisDataset(dataserver_train,
↪  dataserver_test)

vocab_size = dataserver_train.get_vocab_size()

model = text_cl.GRUnet(vocab_size, hidden_size=512,
↪  output_size=2, n_layers=2)

number_of_learnable_params = sum(p.numel() for p in
↪  model.parameters() if p.requires_grad)

num_layers = len(list(model.parameters()))

print("\n\nThe number of layers in the model: %d" % num_layers)
```

```
print("The number of learnable parameters in the model: %d" %
↪  number_of_learnable_params)
print("The size of the input vector: %d" % vocab_size)

## TRAINING:
text_cl.run_code_for_training_for_text_classification_with_GRU(m
↪  odel,
↪  hidden_size=512)

## TESTING:
#import pymsgbox
#response = pymsgbox.confirm("Finished training.  Start testing
↪  on unseen data?")
#if response == "OK":
text_cl.run_code_for_testing_text_classification_with_GRU(model,
↪  hidden_size=512)
```

```
#!/usr/bin/env python

##  text_classification_with_GRU.py

"""
This script is an attempt at solving the sentiment
↪  classification problem
with a GRU based recurrence to get around the problem of
↪  vanishing gradients
that are common to neural networks with feedback.
"""

import random
import numpy
import torch
import os, sys
import sys
sys.path.append( "..//DLStudio" )


seed = 0
random.seed(seed)
torch.manual_seed(seed)
```

```
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)


##  watch -d -n 0.5 nvidia-smi

from DLStudio import *

dls = DLStudio(
#                   dataroot =
↪  "/home/kak/TextDatasets/sentiment_dataset/",
                  dataroot =
                  ↪  "./data/TextDatasets/sentiment_dataset/",
                  path_saved_model = "./saved_model",
                  momentum = 0.9,
                  learning_rate =  1e-5,
                  epochs = 5,
                  batch_size = 1,
                  classes = ('negative','positive'),
                  use_gpu = True,
              )


text_cl = DLStudio.TextClassification( dl_studio = dls )
dataserver_train =
↪  DLStudio.TextClassification.SentimentAnalysisDataset(
                                  train_or_test = 'train',
                                  dl_studio = dls,
                                  #dataset_file = "sentiment_data⌋
                                  ↪  set_train_40.tar.gz",
                                  dataset_file = "sentiment_datas⌋
                                  ↪  et_train_200.tar.gz",
                  )
dataserver_test =
↪  DLStudio.TextClassification.SentimentAnalysisDataset(
                                  train_or_test = 'test',
                                  dl_studio = dls,
                                  #dataset_file = "sentiment_data⌋
                                  ↪  set_test_40.tar.gz",
                                  dataset_file = "sentiment_datas⌋
                                  ↪  et_test_200.tar.gz",
                  )
text_cl.dataserver_train = dataserver_train
```

```
text_cl.dataserver_test = dataserver_test

text_cl.load_SentimentAnalysisDataset(dataserver_train,
↪  dataserver_test)

vocab_size = dataserver_train.get_vocab_size()

model = text_cl.GRUnet(vocab_size, hidden_size=512,
↪  output_size=2, n_layers=2)

number_of_learnable_params = sum(p.numel() for p in
↪  model.parameters() if p.requires_grad)

num_layers = len(list(model.parameters()))

print("\n\nThe number of layers in the model: %d" % num_layers)
print("The number of learnable parameters in the model: %d" %
↪  number_of_learnable_params)
print("The size of the input vector: %d" % vocab_size)

## TRAINING:
text_cl.run_code_for_training_for_text_classification_with_GRU(m
↪  odel,
↪  hidden_size=512)

## TESTING:
import pymsgbox
#response = pymsgbox.confirm("Finished training.  Start testing
↪  on unseen data?")
#if response == "OK":
text_cl.run_code_for_testing_text_classification_with_GRU(model,
↪  hidden_size=512)
```

```
#!/usr/bin/env python

##  text_classification_with_GRU_word2vec.py

"""
This script is an attempt at solving the sentiment
↪  classification problem
```

```
with an RNN that uses a GRU to get around the problem of
↪  vanishing gradients
that are common to neural networks with feedback.
"""

import random
import numpy
import torch
import os, sys
sys.path.append( "..//DLStudio" )
from DLStudio import *


seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)


##  watch -d -n 0.5 nvidia-smi

class MyTextClassificationWithEmbeddings(DLStudio.TextClassifica⌋
↪  tionWithEmbeddings):

    class BatchedSentimentAnalysisDataset(DLStudio.TextClassific⌋
    ↪  ationWithEmbeddings.SentimentAnalysisDataset):
        """
        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.
        """
        def __init__(self, dl_studio, train_or_test,
        ↪  dataset_file, path_to_saved_embeddings=None):
            super(DLStudio.TextClassificationWithEmbeddings.Sent⌋
            ↪  imentAnalysisDataset,
            ↪  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
self.test_max_length = 0
self.train_max_length = 0
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_t⌋
        ↪  o_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")
        ##  save to a disk-based hash table for
        ↪  faster reloading subsequently. '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_embeddi⌋
        ↪  ngs +
        ↪  '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(dataset,
        ↪  encoding='latin1')
    else:
        self.positive_reviews_train,
        ↪  self.negative_reviews_train, self.vocab
        ↪  = pickle.loads(dataset)
    self.categories = sorted(list(self.positive_revi⌋
    ↪  ews_train.keys()))
```

```
        self.category_sizes_train_pos = {category :
        ↪  len(self.positive_reviews_train[category])
        ↪  for category in self.categories}
        self.category_sizes_train_neg = {category :
        ↪  len(self.negative_reviews_train[category])
        ↪  for category in self.categories}
        self.indexed_dataset_train = []
        for category in self.positive_reviews_train:
            for review in
            ↪  self.positive_reviews_train[category]:
                self.indexed_dataset_train.append([revie⌋
                ↪  w, category,
                ↪  1])
        for category in self.negative_reviews_train:
            for review in
            ↪  self.negative_reviews_train[category]:
                self.indexed_dataset_train.append([revie⌋
                ↪  w, category,
                ↪  0])
        random.shuffle(self.indexed_dataset_train)

        #now lets find the longest review so we can make
        ↪  all reviews the same size
        for sample in self.indexed_dataset_train:
            #print(self.indexed_dataset_train[i][0])
            if len(sample[0]) > self.train_max_length:
                self.train_max_length = len(sample[0])
        #print(self.train_max_length)
    elif train_or_test == 'test':
        if sys.version_info[0] == 3:
            self.positive_reviews_test,
            ↪  self.negative_reviews_test, self.vocab =
            ↪  pickle.loads(dataset, encoding='latin1')
        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_revi⌋
        ↪  ews_test.keys()))
        self.category_sizes_test_pos = {category :
        ↪  len(self.positive_reviews_test[category])
        ↪  for category in self.categories}
        self.category_sizes_test_neg = {category :
        ↪  len(self.negative_reviews_test[category])
        ↪  for category in self.categories}
```

```
        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])
        random.shuffle(self.indexed_dataset_test)

        #now lets find the longest review so we can make
        ↪  all reviews the same size
        for sample in self.indexed_dataset_test:
            #print(self.indexed_dataset_test[i][0])
            if len(sample[0]) > self.test_max_length:
                self.test_max_length = len(sample[0])
        #print(self.test_max_length)

    self.max_length = max(self.test_max_length,
    ↪  self.train_max_length)

def review_to_tensor(self, review):
    list_of_embeddings = []
    while len(list_of_embeddings) < self.max_length:
        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(embed
                ↪  ding))
                if len(list_of_embeddings) ==
                ↪  self.max_length:
                    break
            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_dataset_test[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

class GRUnetWithEmbeddings(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.
        """
        def __init__(self, hidden_size, output_size, n_layers,
        ↪  input_size=300, drop_prob=0.2):
            super(MyTextClassificationWithEmbeddings.GRUnetWithE⌋
            ↪  mbeddings,
            ↪  self).__init__()
            self.hidden_size = hidden_size
            self.n_layers = n_layers
            print("Batch First = False")
            self.gru = nn.GRU(input_size, hidden_size, n_layers,
            ↪  batch_first=True, dropout=drop_prob)
            self.fc = nn.Linear(hidden_size, output_size)
            self.relu = nn.ReLU()
            self.logsoftmax = nn.LogSoftmax(dim=1)

        def forward(self, x, h):
            out, h = self.gru(x, h)
            out = self.fc(self.relu(out[:,-1]))
            out = self.logsoftmax(out)
            return out, h

        def init_hidden(self, batch_size):
            weight = next(self.parameters()).data
            hidden = weight.new(self.n_layers, batch_size,
            ↪  self.hidden_size).zero_()
            return hidden
    def run_code_for_training_for_text_classification_with_MyGRU⌋
    ↪  _word2vec(self, net, hidden_size,
    ↪  batch_size):
        filename_for_out = "performance_numbers_" +
        ↪  str(self.dl_studio.epochs) + ".txt"
        FILE = open(filename_for_out, 'w')
        net = copy.deepcopy(net)
        net = net.to(self.dl_studio.device)

        ##  Note that the GREnet now produces the LogSoftmax
        ↪  output:
        #criterion = nn.MSELoss()
        criterion = torch.nn.CrossEntropyLoss()
        #criterion = nn.NLLLoss()

        #optimizer = optim.SGD(net.parameters(),
```

```
#             lr=self.dl_studio.learning_rate,
↪  momentum=self.dl_studio.momentum)
optimizer = torch.optim.Adam(net.parameters(),
↪  lr=self.dl_studio.learning_rate)

accum_times = []
training_loss_tally = []
for epoch in range(self.dl_studio.epochs):
    print("")
    print("batch size is {}".format(batch_size))
    running_loss = 0.0
    start_time = time.perf_counter()
    for i, data in enumerate(self.train_dataloader):
        review_tensor,category,sentiment =
        ↪  data['review'], data['category'],
        ↪  data['sentiment']
        review_tensor =
        ↪  review_tensor.to(self.dl_studio.device)
        sentiment = sentiment.to(self.dl_studio.device)
        ## The following type conversion needed for
        ↪  MSELoss:
        ##sentiment = sentiment.float()
        optimizer.zero_grad()
        #print(review_tensor.shape[0])
        #print(review_tensor.shape[1])
        hidden = net.init_hidden(review_tensor.shape[0])
        ↪  .to(self.dl_studio.device)
        for k in range(review_tensor.shape[1]):
            #print(torch.unsqueeze(review_tensor[:,k],0)
            ↪  .size())
            #print(hidden.size())
            output, hidden = net(torch.unsqueeze(review_
            ↪  tensor[:,k],1),
            ↪  hidden)
        #print(output.size())
        #print(torch.argmax(sentiment, 1).size())
        loss = criterion(output, torch.argmax(sentiment,
        ↪  1))
        running_loss += loss.item()
        loss.backward()
        optimizer.step()
        if i % (500/batch_size) == (500/batch_size) - 1:
            avg_loss = running_loss /
            ↪  float(500/batch_size)
            training_loss_tally.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,i+1,
                ↪  time_elapsed,avg_loss))
                accum_times.append(current_time-start_time)
                FILE.write("%.5f\n" % avg_loss)
                FILE.flush()
                running_loss = 0.0
    print("Total Training Time:
    ↪  {}".format(str(sum(accum_times))))
    print("\nFinished Training\n")
    self.save_model(net)
    plt.figure(figsize=(10,5))
    plt.title("Training Loss vs. Iterations")
    plt.plot(training_loss_tally)
    plt.xlabel("iterations")
    plt.ylabel("training loss")
    plt.legend()
    plt.savefig("training_loss.png")
    plt.show()


def run_code_for_testing_text_classification_with_GRU_word2v
↪  ec(self, net, hidden_size,
↪  batch_size):
    net.load_state_dict(torch.load(self.dl_studio.path_saved
    ↪  _model))
    classification_accuracy = 0.0
    negative_total = 0
    positive_total = 0
    confusion_matrix = torch.zeros(2,2)
    with torch.no_grad():
        for i, data in enumerate(self.test_dataloader):
            review_tensor,category,sentiment =
            ↪  data['review'], data['category'],
            ↪  data['sentiment']
            hidden = net.init_hidden(review_tensor.shape[0])
            for k in range(review_tensor.shape[1]):
                output, hidden = net(torch.unsqueeze(review_
                ↪  tensor[:,k],1),
                ↪  hidden)
            #print(output.size())
            predicted_idx = torch.argmax(output,1)
            gt_idx = torch.argmax(sentiment,1)
            #print(predicted_idx)
            #print(gt_idx)
```

```
                if i % 100 == 99:
                    print("   [i=%d]    predicted_label=%d
                    ↪  gt_label=%d" % (i+1,
                    ↪  predicted_idx[0],gt_idx[0]))
                for j in range(len(gt_idx)):
                    if predicted_idx[j] == gt_idx[j]:
                        classification_accuracy += 1
                    if gt_idx[j] == 0:
                        negative_total += 1
                    elif gt_idx[j] == 1:
                        positive_total += 1
                    confusion_matrix[gt_idx[j],predicted_idx[j]]
                    ↪  += 1
            print("\nOverall classification accuracy: %0.2f%%" %
            ↪  (float(classification_accuracy) * 100
            ↪  /float(negative_total+positive_total)))
            out_percent = np.zeros((2,2), dtype='float')
            out_percent[0,0] = "%.3f" % (100 * confusion_matrix[0,0]
            ↪  / float(negative_total))
            out_percent[0,1] = "%.3f" % (100 * confusion_matrix[0,1]
            ↪  / float(negative_total))
            out_percent[1,0] = "%.3f" % (100 * confusion_matrix[1,0]
            ↪  / float(positive_total))
            out_percent[1,1] = "%.3f" % (100 * confusion_matrix[1,1]
            ↪  / float(positive_total))
            print("\n\nNumber of positive reviews tested: %d" %
            ↪  positive_total)
            print("\n\nNumber of negative reviews tested: %d" %
            ↪  negative_total)
            print("\n\nDisplaying the confusion matrix:\n")
            out_str = "                        "
            out_str +=  "%18s    %18s" % ('predicted negative',
            ↪  'predicted positive')
            print(out_str + "\n")
            for i,label in enumerate(['true negative', 'true
            ↪  positive']):
                out_str = "%12s:  " % label
                for j in range(2):
                    out_str +=  "%18s%%" % out_percent[i,j]
                print(out_str)


dls = DLStudio(
                  dataroot =
                  ↪  "./data/TextDatasets/sentiment_dataset/",
                  path_saved_model = "./saved_model",
```

```
                momentum = 0.9,
                learning_rate =  1e-5,
                epochs = 5,
                batch_size = 10,
                classes = ('negative','positive'),
                use_gpu = True,
            )

text_cl = MyTextClassificationWithEmbeddings( dl_studio = dls )

dataserver_train = MyTextClassificationWithEmbeddings.BatchedSen⌋
↪  timentAnalysisDataset(
                                train_or_test = 'train',
                                dl_studio = dls,
                                #dataset_file = "sentiment_data⌋
                                ↪  set_train_40.tar.gz",
                                dataset_file = "sentiment_datas⌋
                                ↪  et_train_200.tar.gz",
                                path_to_saved_embeddings = "./d⌋
                                ↪  ata/TextDatasets/word2vec/",
                 )
dataserver_test = MyTextClassificationWithEmbeddings.BatchedSent⌋
↪  imentAnalysisDataset(
                                train_or_test = 'test',
                                dl_studio = dls,
                                #dataset_file = "sentiment_data⌋
                                ↪  set_test_40.tar.gz",
                                dataset_file = "sentiment_datas⌋
                                ↪  et_test_200.tar.gz",
                                path_to_saved_embeddings = "./d⌋
                                ↪  ata/TextDatasets/word2vec/",
                )
text_cl.dataserver_train = dataserver_train
text_cl.dataserver_test = dataserver_test

text_cl.load_SentimentAnalysisDataset(dataserver_train,
↪  dataserver_test)

model = text_cl.GRUnetWithEmbeddings(hidden_size=512,
↪  output_size=2, n_layers=2)

number_of_learnable_params = sum(p.numel() for p in
↪  model.parameters() if p.requires_grad)

num_layers = len(list(model.parameters()))
```

```
print("\n\nThe number of layers in the model: %d" % num_layers)
print("\nThe number of learnable parameters in the model: %d" %
↪  number_of_learnable_params)

## TRAINING:
print("\nStarting training\n")

text_cl.run_code_for_training_for_text_classification_with_MyGRU
↪  _word2vec(model, hidden_size=512, batch_size =
↪  dls.batch_size)

## TESTING:
print("starting testing")
text_cl.run_code_for_testing_text_classification_with_GRU_word2v
↪  ec(model, hidden_size=512, batch_size =
↪  dls.batch_size)
```

```
#!/usr/bin/env python

##  text_classification_with_GRU_word2vec.py

"""
This script is an attempt at solving the sentiment
↪  classification problem
with an RNN that uses a GRU to get around the problem of
↪  vanishing gradients
that are common to neural networks with feedback.
"""

import random
import numpy
import torch
import os, sys
sys.path.append( "..//DLStudio" )
from DLStudio import *


seed = 0
random.seed(seed)
torch.manual_seed(seed)
```

```
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)


##  watch -d -n 0.5 nvidia-smi

class MyTextClassificationWithEmbeddings(DLStudio.TextClassifica
↪  tionWithEmbeddings):

    def run_code_for_training_for_text_classification_with_MyGRU
    ↪  _word2vec(self, net,
    ↪  hidden_size):
        filename_for_out = "performance_numbers_" +
        ↪  str(self.dl_studio.epochs) + ".txt"
        FILE = open(filename_for_out, 'w')
        net = copy.deepcopy(net)
        net = net.to(self.dl_studio.device)

        ##  Note that the GREnet now produces the LogSoftmax
        ↪  output:
        #criterion = nn.MSELoss()
        criterion = torch.nn.CrossEntropyLoss()
        #criterion = nn.NLLLoss()

        #optimizer = optim.SGD(net.parameters(),
        #             lr=self.dl_studio.learning_rate,
        ↪  momentum=self.dl_studio.momentum)
        optimizer = torch.optim.Adam(net.parameters(),
        ↪  lr=self.dl_studio.learning_rate)

        accum_times = []
        training_loss_tally = []
        for epoch in range(self.dl_studio.epochs):
            print("")
            running_loss = 0.0
            start_time = time.perf_counter()
            for i, data in enumerate(self.train_dataloader):
                review_tensor,category,sentiment =
                ↪  data['review'], data['category'],
                ↪  data['sentiment']
                review_tensor =
                ↪  review_tensor.to(self.dl_studio.device)
                sentiment = sentiment.to(self.dl_studio.device)
```

```
                ## The following type conversion needed for
                ↪  MSELoss:
                ##sentiment = sentiment.float()
                optimizer.zero_grad()
                hidden =
                ↪  net.init_hidden(1).to(self.dl_studio.device)
                for k in range(review_tensor.shape[1]):
                    output, hidden = net(torch.unsqueeze(torch.u⌋
                    ↪  nsqueeze(review_tensor[0,k],0),0),
                    ↪  hidden)
                loss = criterion(output, torch.argmax(sentiment,
                ↪  1))
                running_loss += loss.item()
                loss.backward()
                optimizer.step()
                if i % 500 == 499:
                    avg_loss = running_loss / float(500)
                    training_loss_tally.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,i+1,
                    ↪  time_elapsed,avg_loss))
                    accum_times.append(current_time-start_time)
                    FILE.write("%.5f\n" % avg_loss)
                    FILE.flush()
                    running_loss = 0.0
        print("Total Training Time:
        ↪  {}".format(str(sum(accum_times))))
        print("\nFinished Training\n")
        self.save_model(net)
        plt.figure(figsize=(10,5))
        plt.title("Training Loss vs. Iterations")
        plt.plot(training_loss_tally)
        plt.xlabel("iterations")
        plt.ylabel("training loss")
        plt.legend()
        plt.savefig("training_loss.png")
        plt.show()


dls = DLStudio(
#                  dataroot =
↪  "/home/kak/TextDatasets/sentiment_dataset/",
                 dataroot =
                 ↪  "./data/TextDatasets/sentiment_dataset/",
```

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

text_cl = MyTextClassificationWithEmbeddings( dl_studio = dls )

dataserver_train =
↪  MyTextClassificationWithEmbeddings.SentimentAnalysisDataset(
                                train_or_test = 'train',
                                dl_studio = dls,
                                #dataset_file = "sentiment_data
                                ↪  set_train_40.tar.gz",
                                dataset_file = "sentiment_datas
                                ↪  et_train_200.tar.gz",
#                                path_to_saved_embeddings =
↪  "/home/kak/TextDatasets/word2vec/",
                                path_to_saved_embeddings = "./d
                                ↪  ata/TextDatasets/word2vec/",
                    )
dataserver_test =
↪  MyTextClassificationWithEmbeddings.SentimentAnalysisDataset(
                                train_or_test = 'test',
                                dl_studio = dls,
                                #dataset_file = "sentiment_data
                                ↪  set_test_40.tar.gz",
                                dataset_file = "sentiment_datas
                                ↪  et_test_200.tar.gz",
#                                path_to_saved_embeddings =
↪  "/home/kak/TextDatasets/word2vec/",
                                path_to_saved_embeddings = "./d
                                ↪  ata/TextDatasets/word2vec/",
                    )
text_cl.dataserver_train = dataserver_train
text_cl.dataserver_test = dataserver_test

text_cl.load_SentimentAnalysisDataset(dataserver_train,
↪  dataserver_test)

model = text_cl.GRUnetWithEmbeddings(hidden_size=512,
↪  output_size=2, n_layers=2)
```

```
number_of_learnable_params = sum(p.numel() for p in
↪  model.parameters() if p.requires_grad)

num_layers = len(list(model.parameters()))

print("\n\nThe number of layers in the model: %d" % num_layers)
print("\nThe number of learnable parameters in the model: %d" %
↪  number_of_learnable_params)

## TRAINING:
print("\nStarting training\n")

text_cl.run_code_for_training_for_text_classification_with_MyGRU
↪  _word2vec(model,
↪  hidden_size=512)

## TESTING:
#import pymsgbox
#response = pymsgbox.confirm("Finished training.  Start testing
↪  on unseen data?")
#if response == "OK":
text_cl.run_code_for_testing_text_classification_with_GRU_word2v
↪  ec(model,
↪  hidden_size=512)
```

```
#!/usr/bin/env python

##  text_classification_with_TEXTnetOrder2_word2vec.py

"""This script uses an embeddings version of text classification
↪  class TEXTnetOrder2.

Read the comment block at the beginning of

         text_classification_with_TEXTnetOrder2.py

to see why the TEXTnetOrder2 class is a stepping stone to
↪  working with a GRU.  In
the same manner, you can think of the class
↪  TEXTnetOrder2WithEmbeddings also as
```

```
a stepping stone to using recurrence throush a GRU.
"""

import random
import numpy
import torch
import os, sys
sys.path.append( "..//DLStudio" )
from DLStudio import *


seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)


##  watch -d -n 0.5 nvidia-smi

class MyGRUTextClassificationWithEmbeddings(DLStudio.TextClassif⌋
↪  icationWithEmbeddings):

    class GatedTEXTnetOrder2WithEmbeddings(nn.Module):
        """
        In this variant of the TEXTnet network, the value of
        ↪  hidden as used at
        each time step also includes its value at the previous
        ↪  time step.  This
        fact, not directly apparent by the definition of the
        ↪  class shown below,
        is made possible by the last parameter, cell, in the
        ↪  header of forward().
        All you can see here, at the end of forward(), is that
        ↪  the value of cell
        goes through a linear layer and through a sigmoid
        ↪  nonlinearity. By the way,
        since the sigmoid saturates at 0 and 1, it can act like
        ↪  a switch. Later
        when I use this class in the training function, you will
        ↪  see the cell
        values being used in such a manner that the hidden state
        ↪  at each time
```

```
step is mixed with the hidden state at the previous time
↪  step.

Location: Inner class TextClassification
"""
def __init__(self, hidden_size, output_size,
↪  input_size=300):
    print("Using updated TEXTnetOrder2 with gating!")
    super(MyGRUTextClassificationWithEmbeddings.GatedTEX
    ↪  TnetOrder2WithEmbeddings,
    ↪  self).__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.output_size = output_size
    self.combined_to_hidden = nn.Linear(input_size +
    ↪  2*hidden_size, hidden_size)
    self.combined_to_middle = nn.Linear(input_size +
    ↪  2*hidden_size, 100)
    self.middle_to_out = nn.Linear(100, output_size)
    self.logsoftmax = nn.LogSoftmax(dim=1)
    self.dropout = nn.Dropout(p=0.1)
    # for the cell
    self.linear_for_cell = nn.Linear(hidden_size,
    ↪  hidden_size)

    #self.linear_for_input_reset = nn.Linear(input_size,
    ↪  hidden_size)
    #self.linear_for_input_update =
    ↪  nn.Linear(input_size, hidden_size)
    #self.linear_for_hidden_reset =
    ↪  nn.Linear(hidden_size, hidden_size)
    #self.linear_for_hidden_update =
    ↪  nn.Linear(hidden_size, hidden_size)
    #self.linear_for_input_vector_r =
    ↪  nn.Linear(input_size, hidden_size)
    #self.linear_for_hidden_vector_r =
    ↪  nn.Linear(hidden_size, hidden_size)
    #self.linear_for_hidden_vector_r =
    ↪  nn.Linear(hidden_size, hidden_size)
    #self.linear_for_output = nn.Linear(hidden_size,
    ↪  output_size)

def forward(self, input_vector, hidden, cell):
    combined = torch.cat((input_vector, hidden, cell), 1)
    hidden = self.combined_to_hidden(combined)
    hidden = torch.tanh(hidden)
```

```
            output = self.combined_to_middle(combined)
            output = torch.nn.functional.relu(output)
            output = self.dropout(output)
            output = self.middle_to_out(output)
            output = self.logsoftmax(output)
            hidden_clone = hidden.clone()
            #cell =
            ↪  torch.tanh(self.linear_for_cell(hidden_clone))
            cell =
            ↪  torch.sigmoid(self.linear_for_cell(hidden_clone))
            hidden_new = hidden * cell

            #input_1 = input_vector.clone()
            #input_2 = input_vector.clone()
            #input_3 = input_vector.clone()

            #reset = torch.sigmoid(self.linear_for_input_reset(i
            ↪  nput_1) +
            ↪  self.linear_for_hidden_reset(hidden))
            #vector_r = torch.tanh(reset *
            ↪  (self.linear_for_input_vector_r(input_2) +
            ↪  self.linear_for_hidden_vector_r(hidden)))


            #update = torch.sigmoid(self.linear_for_input_update
            ↪  (input_3) +
            ↪  self.linear_for_hidden_update(hidden))
            #vector_u = update * hidden

            #hidden_new = vector_r * (1 - update) + vector_u

            #output = self.linear_for_output(hidden_new)

            return output,hidden_new,cell

        def initialize_cell(self, batch_size):
            weight = next(self.linear_for_cell.parameters()).data
            cell = weight.new(1, self.hidden_size).zero_()
            return cell

dls = DLStudio(
#                  dataroot =
↪  "/home/kak/TextDatasets/sentiment_dataset/",
                 dataroot =
                 ↪  "./data/TextDatasets/sentiment_dataset/",
                 path_saved_model = "./saved_model",
```

```
                momentum = 0.9,
                learning_rate =  1e-5,
                epochs = 5,
                batch_size = 1,
                classes = ('negative','positive'),
                use_gpu = True,
            )

text_cl = MyGRUTextClassificationWithEmbeddings( dl_studio = dls
↪  )

dataserver_train = MyGRUTextClassificationWithEmbeddings.Sentime
↪  ntAnalysisDataset(
                                train_or_test = 'train',
                                dl_studio = dls,
                                #dataset_file = "sentiment_data
                                ↪  set_train_40.tar.gz",
                                dataset_file = "sentiment_datas
                                ↪  et_train_200.tar.gz",
#                                path_to_saved_embeddings =
↪  "/home/kak/TextDatasets/word2vec/",
                                path_to_saved_embeddings = "./d
                                ↪  ata/TextDatasets/word2vec/",
                  )
dataserver_test = MyGRUTextClassificationWithEmbeddings.Sentimen
↪  tAnalysisDataset(
                                train_or_test = 'test',
                                dl_studio = dls,
                                #dataset_file = "sentiment_data
                                ↪  set_test_40.tar.gz",
                                dataset_file = "sentiment_datas
                                ↪  et_test_200.tar.gz",
#                                path_to_saved_embeddings =
↪  "/home/kak/TextDatasets/word2vec/",
                                path_to_saved_embeddings = "./d
                                ↪  ata/TextDatasets/word2vec/",
                  )
text_cl.dataserver_train = dataserver_train
text_cl.dataserver_test = dataserver_test

text_cl.load_SentimentAnalysisDataset(dataserver_train,
↪  dataserver_test)

model =
↪  text_cl.GatedTEXTnetOrder2WithEmbeddings(hidden_size=512,
↪  output_size=2)
```

```
number_of_learnable_params = sum(p.numel() for p in
↪  model.parameters() if p.requires_grad)
num_layers = len(list(model.parameters()))

print("\n\nThe number of layers in the model: %d" % num_layers)
print("\nThe number of learnable parameters in the model: %d" %
↪  number_of_learnable_params)

text_cl.run_code_for_training_with_TEXTnetOrder2_word2vec(model,
↪  hidden_size=512)

#import pymsgbox
#response = pymsgbox.confirm("Finished training.  Start testing
↪  on unseen data?")
#if response == "OK":
text_cl.run_code_for_testing_with_TEXTnetOrder2_word2vec(model,
↪  hidden_size=512)
```

```
#!/usr/bin/env python

##  text_classification_with_TEXTnetOrder2_word2vec.py

"""This script uses an embeddings version of text classification
↪  class TEXTnetOrder2.

Read the comment block at the beginning of

         text_classification_with_TEXTnetOrder2.py

to see why the TEXTnetOrder2 class is a stepping stone to
↪  working with a GRU.  In
the same manner, you can think of the class
↪  TEXTnetOrder2WithEmbeddings also as
a stepping stone to using recurrence throush a GRU.
"""

import random
import numpy
import torch
```

```
import os, sys
sys.path.append( "..//DLStudio" )
from DLStudio import *


seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)


##  watch -d -n 0.5 nvidia-smi

class MyGRUTextClassificationWithEmbeddings(DLStudio.TextClassif
↪  icationWithEmbeddings):

    class GatedTEXTnetOrder2WithEmbeddings(nn.Module):
        """
        In this variant of the TEXTnet network, the value of
        ↪  hidden as used at
        each time step also includes its value at the previous
        ↪  time step.  This
        fact, not directly apparent by the definition of the
        ↪  class shown below,
        is made possible by the last parameter, cell, in the
        ↪  header of forward().
        All you can see here, at the end of forward(), is that
        ↪  the value of cell
        goes through a linear layer and through a sigmoid
        ↪  nonlinearity. By the way,
        since the sigmoid saturates at 0 and 1, it can act like
        ↪  a switch. Later
        when I use this class in the training function, you will
        ↪  see the cell
        values being used in such a manner that the hidden state
        ↪  at each time
        step is mixed with the hidden state at the previous time
        ↪  step.

        Location: Inner class TextClassification
        """
```

```
def __init__(self, hidden_size, output_size,
↪  input_size=300):
    print("Using updated TEXTnetOrder2 with GRU!")
    super(MyGRUTextClassificationWithEmbeddings.GatedTEX
    ↪  TnetOrder2WithEmbeddings,
    ↪  self).__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.output_size = output_size
    #self.combined_to_hidden = nn.Linear(input_size +
    ↪  2*hidden_size, hidden_size)
    #self.combined_to_middle = nn.Linear(input_size +
    ↪  2*hidden_size, 100)
    #self.middle_to_out = nn.Linear(100, output_size)
    self.logsoftmax = nn.LogSoftmax(dim=1)
    #self.dropout = nn.Dropout(p=0.1)
    # for the cell
    self.linear_for_cell = nn.Linear(hidden_size,
    ↪  hidden_size)

    self.linear_for_input_reset = nn.Linear(input_size,
    ↪  hidden_size)
    self.linear_for_input_update = nn.Linear(input_size,
    ↪  hidden_size)
    self.linear_for_hidden_reset =
    ↪  nn.Linear(hidden_size, hidden_size)
    self.linear_for_hidden_update =
    ↪  nn.Linear(hidden_size, hidden_size)
    self.linear_for_input_vector_r =
    ↪  nn.Linear(input_size, hidden_size)
    self.linear_for_hidden_vector_r =
    ↪  nn.Linear(hidden_size, hidden_size)
    self.linear_for_hidden_vector_r =
    ↪  nn.Linear(hidden_size, hidden_size)
    self.linear_for_output = nn.Linear(hidden_size,
    ↪  output_size)

def forward(self, input_vector, hidden, cell):
    #combined = torch.cat((input_vector, hidden, cell),
    ↪  1)
    #hidden = self.combined_to_hidden(combined)
    #hidden = torch.tanh(hidden)
    #output = self.combined_to_middle(combined)
    #output = torch.nn.functional.relu(output)
    #output = self.dropout(output)
    #output = self.middle_to_out(output)
```

```
            #output = self.logsoftmax(output)
            #hidden_clone = hidden.clone()
            ##cell =
            ↪  torch.tanh(self.linear_for_cell(hidden_clone))
            #cell =
            ↪  torch.sigmoid(self.linear_for_cell(hidden_clone))
            #hidden_new = hidden * cell

            input_1 = input_vector.clone()
            input_2 = input_vector.clone()
            input_3 = input_vector.clone()

            reset = torch.sigmoid(self.linear_for_input_reset(in
            ↪  put_1) +
            ↪  self.linear_for_hidden_reset(hidden))
            vector_r = torch.tanh(reset *
            ↪  (self.linear_for_input_vector_r(input_2) +
            ↪  self.linear_for_hidden_vector_r(hidden)))


            update = torch.sigmoid(self.linear_for_input_update(
            ↪  input_3) +
            ↪  self.linear_for_hidden_update(hidden))
            vector_u = update * hidden

            hidden_new = vector_r * (1 - update) + vector_u

            output = self.linear_for_output(hidden_new)
            output = self.logsoftmax(output)

            return output,hidden_new,cell

        def initialize_cell(self, batch_size):
            weight = next(self.linear_for_cell.parameters()).data
            cell = weight.new(1, self.hidden_size).zero_()
            return cell

dls = DLStudio(
#                   dataroot =
↪  "/home/kak/TextDatasets/sentiment_dataset/",
                  dataroot =
                  ↪  "./data/TextDatasets/sentiment_dataset/",
                  path_saved_model = "./saved_model",
                  momentum = 0.9,
                  learning_rate =  1e-5,
                  epochs = 5,
```

```
                batch_size = 1,
                classes = ('negative','positive'),
                use_gpu = True,
            )

text_cl = MyGRUTextClassificationWithEmbeddings( dl_studio = dls
↪  )

dataserver_train = MyGRUTextClassificationWithEmbeddings.Sentime
↪  ntAnalysisDataset(
                                train_or_test = 'train',
                                dl_studio = dls,
                                #dataset_file = "sentiment_data
                                ↪  set_train_40.tar.gz",
                                dataset_file = "sentiment_datas
                                ↪  et_train_200.tar.gz",
#                                path_to_saved_embeddings =
↪  "/home/kak/TextDatasets/word2vec/",
                                path_to_saved_embeddings = "./d
                                ↪  ata/TextDatasets/word2vec/",
                )
dataserver_test = MyGRUTextClassificationWithEmbeddings.Sentimen
↪  tAnalysisDataset(
                                train_or_test = 'test',
                                dl_studio = dls,
                                #dataset_file = "sentiment_data
                                ↪  set_test_40.tar.gz",
                                dataset_file = "sentiment_datas
                                ↪  et_test_200.tar.gz",
#                                path_to_saved_embeddings =
↪  "/home/kak/TextDatasets/word2vec/",
                                path_to_saved_embeddings = "./d
                                ↪  ata/TextDatasets/word2vec/",
                )
text_cl.dataserver_train = dataserver_train
text_cl.dataserver_test = dataserver_test

text_cl.load_SentimentAnalysisDataset(dataserver_train,
↪  dataserver_test)

model =
↪  text_cl.GatedTEXTnetOrder2WithEmbeddings(hidden_size=512,
↪  output_size=2)

number_of_learnable_params = sum(p.numel() for p in
↪  model.parameters() if p.requires_grad)
```

```
num_layers = len(list(model.parameters()))

print("\n\nThe number of layers in the model: %d" % num_layers)
print("\nThe number of learnable parameters in the model: %d" %
↪  number_of_learnable_params)

text_cl.run_code_for_training_with_TEXTnetOrder2_word2vec(model,
↪  hidden_size=512)

#import pymsgbox
#response = pymsgbox.confirm("Finished training.  Start testing
↪  on unseen data?")
#if response == "OK":
text_cl.run_code_for_testing_with_TEXTnetOrder2_word2vec(model,
↪  hidden_size=512)
```

```
#!/usr/bin/env python

##  text_classification_with_TEXTnetOrder2_word2vec.py

"""This script uses an embeddings version of text classification
↪  class TEXTnetOrder2.

Read the comment block at the beginning of

         text_classification_with_TEXTnetOrder2.py

to see why the TEXTnetOrder2 class is a stepping stone to
↪  working with a GRU.  In
the same manner, you can think of the class
↪  TEXTnetOrder2WithEmbeddings also as
a stepping stone to using recurrence throush a GRU.
"""

import random
import numpy
import torch
import os, sys
sys.path.append( "..//DLStudio" )
```

```
seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)


##  watch -d -n 0.5 nvidia-smi

from DLStudio import *

dls = DLStudio(
#                   dataroot =
↪  "/home/kak/TextDatasets/sentiment_dataset/",
                  dataroot =
                  ↪  "./data/TextDatasets/sentiment_dataset/",
                  path_saved_model = "./saved_model",
                  momentum = 0.9,
                  learning_rate =  1e-5,
                  epochs = 5,
                  batch_size = 1,
                  classes = ('negative','positive'),
                  use_gpu = True,
              )

text_cl = DLStudio.TextClassificationWithEmbeddings( dl_studio =
↪  dls )

dataserver_train = DLStudio.TextClassificationWithEmbeddings.Sen⌋
↪  timentAnalysisDataset(
                                  train_or_test = 'train',
                                  dl_studio = dls,
                                  #dataset_file = "sentiment_data⌋
                                  ↪  set_train_40.tar.gz",
                                  dataset_file = "sentiment_datas⌋
                                  ↪  et_train_200.tar.gz",
#                                  path_to_saved_embeddings =
↪  "/home/kak/TextDatasets/word2vec/",
                                  path_to_saved_embeddings = "./d⌋
                                  ↪  ata/TextDatasets/word2vec/",
                   )
```

```
dataserver_test = DLStudio.TextClassificationWithEmbeddings.Sent⌋
↪  imentAnalysisDataset(
                                train_or_test = 'test',
                                dl_studio = dls,
                                #dataset_file = "sentiment_data⌋
                                ↪  set_test_40.tar.gz",
                                dataset_file = "sentiment_datas⌋
                                ↪  et_test_200.tar.gz",
#                                path_to_saved_embeddings =
↪  "/home/kak/TextDatasets/word2vec/",
                                path_to_saved_embeddings = "./d⌋
                                ↪  ata/TextDatasets/word2vec/",
                  )
text_cl.dataserver_train = dataserver_train
text_cl.dataserver_test = dataserver_test

text_cl.load_SentimentAnalysisDataset(dataserver_train,
↪  dataserver_test)

model = text_cl.TEXTnetOrder2WithEmbeddings(hidden_size=512,
↪  output_size=2)

number_of_learnable_params = sum(p.numel() for p in
↪  model.parameters() if p.requires_grad)
num_layers = len(list(model.parameters()))

print("\n\nThe number of layers in the model: %d" % num_layers)
print("\nThe number of learnable parameters in the model: %d" %
↪  number_of_learnable_params)

text_cl.run_code_for_training_with_TEXTnetOrder2_word2vec(model,
↪  hidden_size=512)

#import pymsgbox
#response = pymsgbox.confirm("Finished training.  Start testing
↪  on unseen data?")
#if response == "OK":
text_cl.run_code_for_testing_with_TEXTnetOrder2_word2vec(model,
↪  hidden_size=512)
```

RNN Networks

Kevin Bridgman

BME695/ECE695 Deep Learning

Homework #7

4/26/2021

1 Introduction

The goal of this homework assignment is to explore various aspects of recurrent neural networks to gain understanding of the following:

1) Recurrent Neural Networks (RNN) and how they incorporate feedback
2) Issues with using one-hot vector inputs for RNNs
3) Gating and GRUs to combat the vanishing gradient problem in RNNs
4) How to use a RNN to predict sentiments (positive/negative) of Amazon reviews

2 Overview

2.1 Submission Files

The submission zip file contains the following files.

DL_HW7_Report.PDF	This file
text_classification_with_GRU_onehot.py	Baseline for task 1
text_classification_with_GRU_encoded.py	Encoding implementation for task 1
text_classification_with_GRU_Gabriel.py	Gabriel's implementation for task 1
text_classification_with_GRU_int.py	Integer implementation for task 1
text_classification_with_TEXTnetOrder2_word2vec.py	Baseline for task 2
text_classification_with_TEXTnetOrder2_word2vec_gated.py	My implementation for task 2
text_classification_with_TEXTnetOrder2_word2vec_GRU.py	My GRU implementation for task 2
text_classification_with_MyGRU_word2vec.py	Baseline for task 3
text_classification_with_MyGRU_word2vec_batched.py	My batched implementation or task 3
HW07.ipynb	Colab work space with all results

2.2 Training conditions and Requirements

Unless otherwise specified, all training was performed with pytorch's NLLLoss criterion and pytorch's SGD optimizer with a learning rate of 1e-5 and momentum of 0.9.

Trainings were run for 5 epochs with the following training file: *sentiment_dataset_train_200.tar.gz*

Testing was performed with the following file: *sentiment_dataset_test_200.tar.gz*

2.3 References and Resources

I used code from Dr Kak's DLStudio 2.0.8 and Gabriel Loye's RNN git repo. I also used Gabriel's blog as a reference for my GRU implementation.

https://engineering.purdue.edu/kak/distDLS/

https://github.com/gabrielloye/GRU_Prediction

https://blog.floydhub.com/gru-with-pytorch/

3 Task 1

Task 1 is focused on the deficiencies of one-hot vector representation for words as the input to RNNs. The main deficiency being the size of the resulting network from the large one-hot vector and the resulting training time.

The objective is to find and evaluate alternate representations for words.

I used DLStudio's example GRU network as a baseline for performance. Then I modified the GRU network in the following two ways.

1) I changed the dataloader and network to use a binary encoding scheme with vector length ceiling(log2(vocab_size). This resulted in an input vector of size 16 for the dataset *sentiment_dataset_X_200.tar.gz.*
2) I changed the dataloader and network to use an integer index for the word. This resulted in an input vector of size 1.

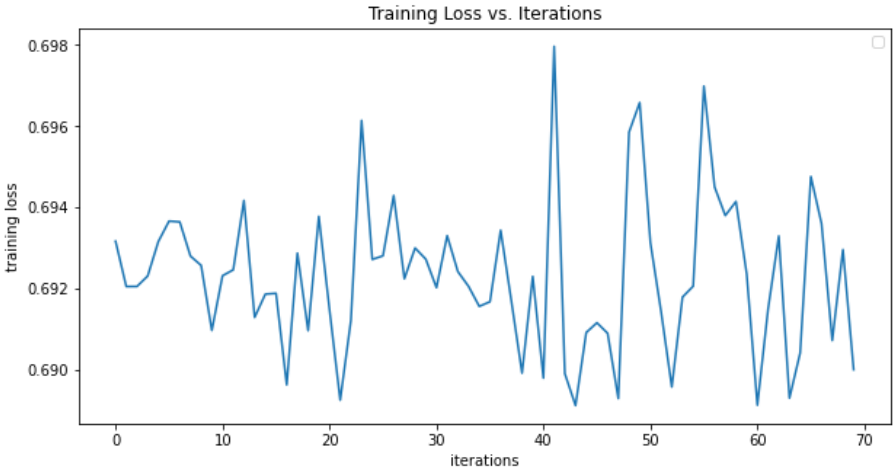
I also used Gabriel Loye's code from github (linked above) with an integer input.

The following table shows the associated parameter count and training times for each different model.

Model	Seconds per Epoch	Parameters
text_classification_with_GRU_onehot.py	~4000	68852226
text_classification_with_GRU_encoded.py	~630	2391042
text_classification_with_GRU_int.py	~600	2368002
text_classification_with_GRU_Gabriel.py	~117	2368002

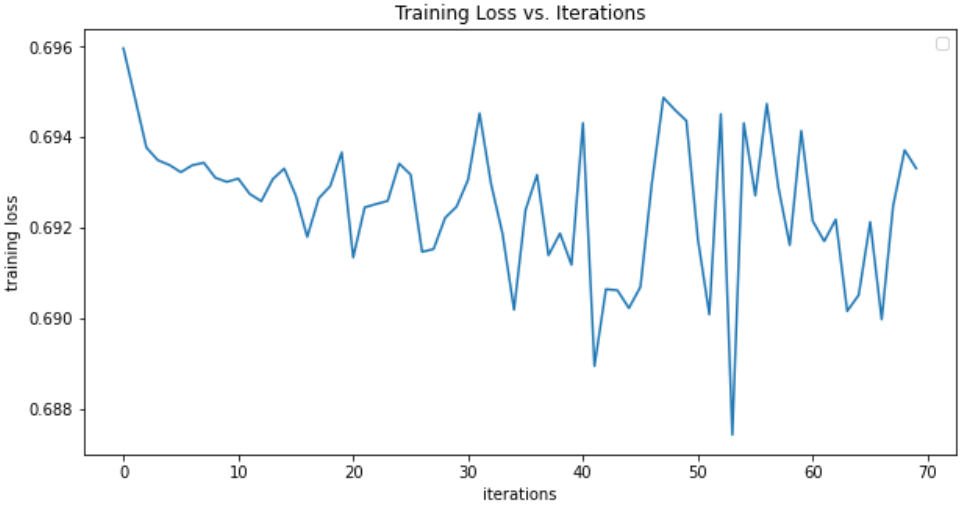
While training time decreased significantly, none of the networks were able to reduce loss or achieve good accuracy. The training and testing results for each model are listed below.

3.1 One Hot Results



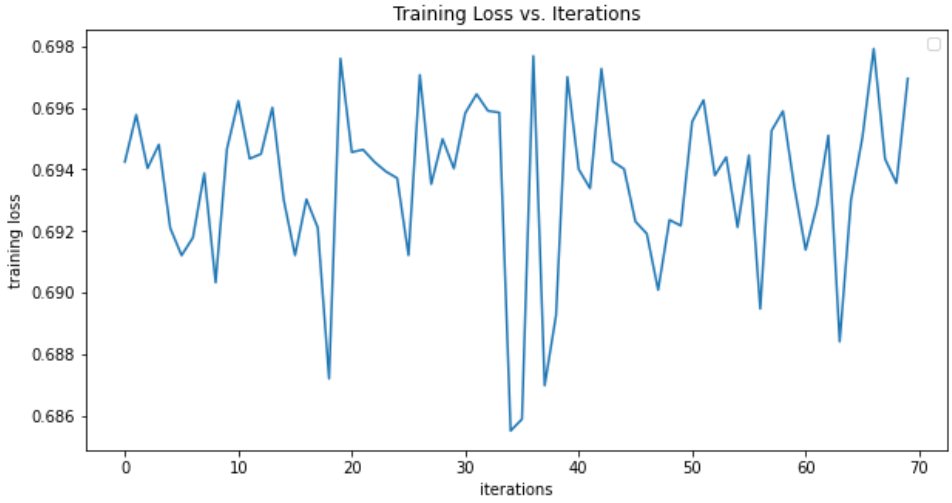

```
Overall classification accuracy: 52.24%

Number of positive reviews tested: 980

Number of negative reviews tested: 897

Displaying the confusion matrix:

                    predicted negative    predicted positive

true negative:                 0.0              100.0
true positive:                 0.0              100.0
```

3.2 ENCODED RESULTS




```
Overall classification accuracy: 52.24%

Number of positive reviews tested: 980

Number of negative reviews tested: 897

Displaying the confusion matrix:

                    predicted negative    predicted positive

true negative:              0.0                 100.0
true positive:              0.0                 100.0
```

3.3 INTEGER RESULTS



```
Overall classification accuracy: 51.97%

Number of positive reviews tested: 980

Number of negative reviews tested: 897

Displaying the confusion matrix:

                     predicted negative    predicted positive

true negative:              5.574              94.426
true positive:              5.612              94.388
```

3.4 GABRIEL'S RESULTS

Gabriel's code does not provide a loss vs interations graph. Loss decreased from 0.27 to 0.25 over 5 epochs. See the provided Jupiter python notebook for full results.

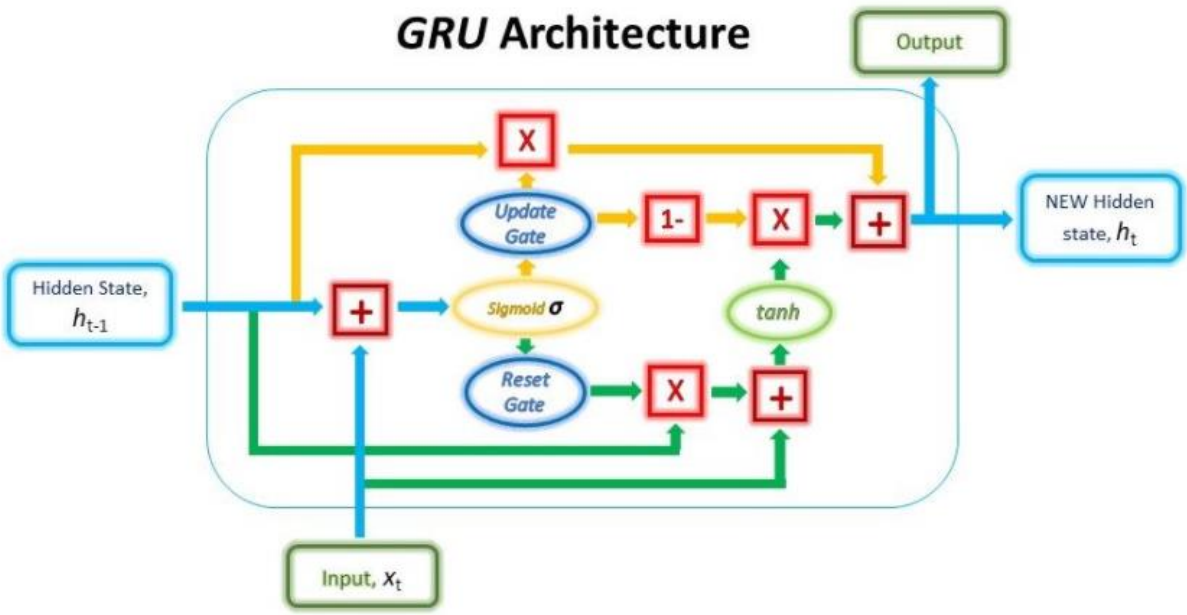
```
Overall classification accuracy: 53.46%

Number of positive reviews tested: 980

Number of negative reviews tested: 897

Displaying the confusion matrix:

                     predicted negative    predicted positive

true negative:             37.458              62.542
true positive:             31.939              68.061
```

4 TASK 2

The objective of task 2 is to gain a better understanding on the gating logic used in GRU networks by implementing some custom gating in DLStudio's TEXTnetOrder2 network. To improve training time, I elected to use the *text_classification_with_TEXTnetOrder2_word2vec.py* script to take advantage of word embeddings.

I attempted 2 different gating mechanisms for this task.

1) I created a very simple gating effect by simply multiplying the new hidden state by a sigmoid of the previous hidden state. See my code in *text_classification_with_TEXTnetOrder2_gated.py* for more detail.
2) I implemented the full gating mechanism as described in Gabriel Loye's blog linked above. The architecture is shown in the following diagram. See my code in *text_classification_with_TEXTnetOrder2_GRU* for more details.

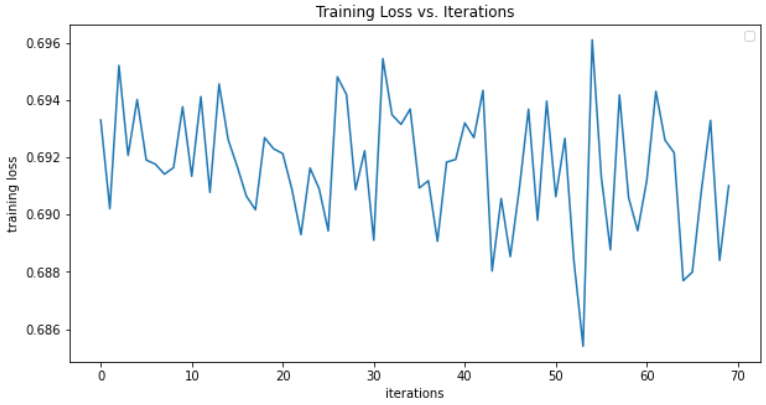


The following table shows the associated parameter count and training times for each different model.

Model	Seconds per Epoch	Parameter Count
text_classification_with_TEXTnetOrder2_word2vec.py	~310	1073758
text_classification_with_TEXTnetOrder2_word2vec_gated.py	~350	1073758
text_classification_with_TEXTnetOrder2_word2vec_GRU.py	~980	1513986

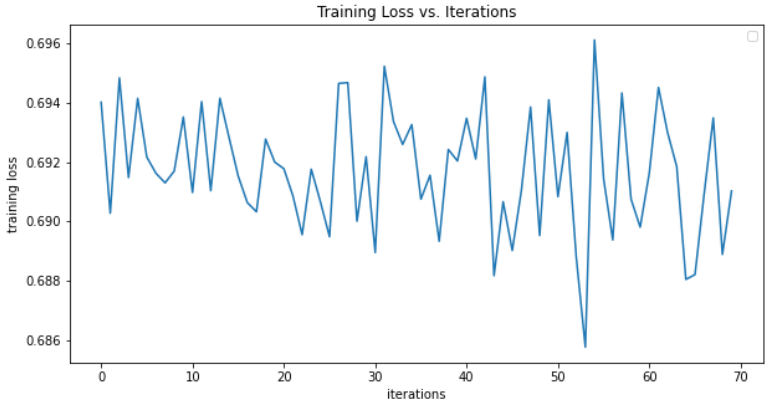
Unfortunately, both of my attempts to improve performance with gating only reduced accuracy and caused a greater mode collapse. The results are shown below.

4.1 TEXTnetOrder2 Baseline



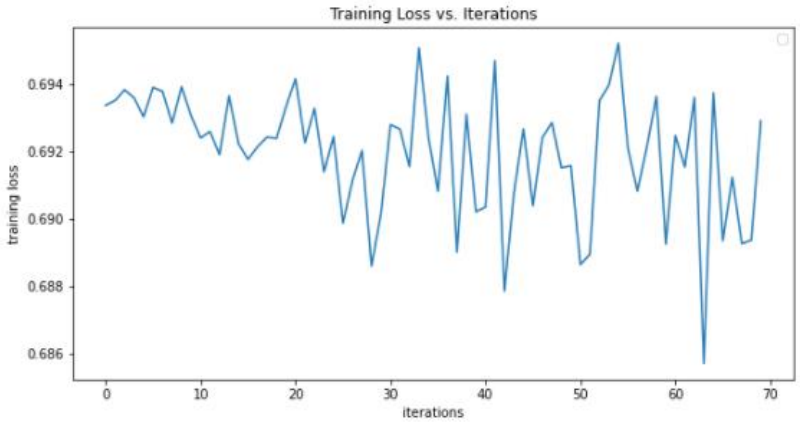
```
Overall classification accuracy: 53.20%

Number of positive reviews tested: 980

Number of negative reviews tested: 897

Displaying the confusion matrix:

                     predicted negative    predicted positive

true negative:              5.017              94.983
true positive:              2.755              97.245
```

4.2 TEXTNETORDER2 WITH GATING




```
Overall classification accuracy: 52.99%

Number of positive reviews tested: 980

Number of negative reviews tested: 897

Displaying the confusion matrix:

                     predicted negative    predicted positive

true negative:              3.679              96.321
true positive:              1.939              98.061
```

4.3 TEXTNETORDER2 WITH GRU




```
Overall classification accuracy: 52.19%

Number of positive reviews tested: 980

Number of negative reviews tested: 897

Displaying the confusion matrix:

                    predicted negative    predicted positive

true negative:              0.111              99.889
true positive:              0.204              99.796
```

5 TASK 3

The objective of task 3 is to add batching capability to evaluate its impact on performance.

Up until now all the models have had accuracies in the low 50s which has made it difficult to see the effects of my code modifications. As such, I decided to spend some time to create a better baseline model. By using DLStudio's GRU with embeddings along with the cross-entropy loss function and ADAM optimizer from Gabriel's code, I was able to create a model with 80% accuracy after 5 training epochs with the *sentiment_dataset_train_200.tar.gz* data. This new model is what I will use for the remainder of task 3.

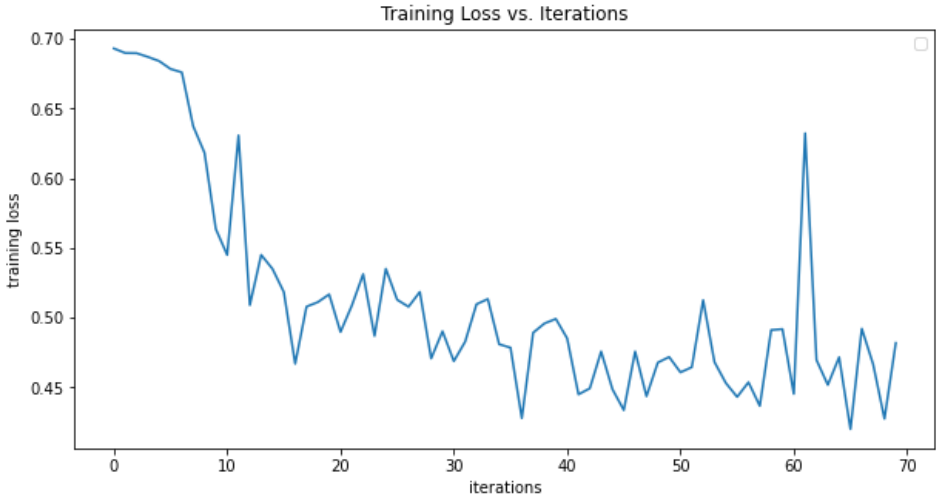
In order to batch, each input vector in the batch needs to be the same length. I made all the inputs the same length by first finding the longest length input. Then I increased the size of all shorter inputs by repeating them until they were the desired length. See my code in text_classification_with_MyGRU_word2vec_batched.py for more details.

Increasing all of the inputs to the max size resulted in an increase in training time even with a batch size of 10. The following table shows the associated parameter count and training times for the unbatched and batched models.

Model	Seconds per Epoch	Parameter Count
text_classification_with_MyGRU_word2vec.py	~440	2827266
text_classification_with_MyGRU_word2vec_batched.py	~1360	2827266

The batch and unbatch model's accuracy were very close; 79.80 vs 79.06. The results are shown below.

5.1 BATCH OF 1




```
Overall classification accuracy: 79.80%

Number of positive reviews tested: 980

Number of negative reviews tested: 897

Displaying the confusion matrix:

                    predicted negative    predicted positive

true negative:              79.487%            20.513%
true positive:                20.0%              80.0%
```

5.2 BATCH OF 10


```
Overall classification accuracy: 79.06%

Number of positive reviews tested: 980

Number of negative reviews tested: 897

Displaying the confusion matrix:

                    predicted negative    predicted positive

true negative:            84.169%              15.831%
true positive:            25.612%              74.388%
```

6 CONCLUSION

While I was unable to get performance improvements for tasks 2 and 3, I still learned a lot about RNN and GRUs. Here is a by-task summary.

- Task 1: I was able to significantly reduce the time needed for training by using shorter encodings instead of the one-hot input vectors!
- Task 2: I was not able to improve performance by adding more gating into the TEXTnetOrder2 code. However, I learned a lot about GRU gating while implementing my own!
- Task 3: I was not able to increase performance by batching. However, I did significantly improve the performance by switching the loss function and optimizer!

Here are some things I wanted to do but ran out of time.

- Redo task 1 and task 2 with the improved model from task 3
- Try using FastTest embeddings for task 1
- Try different length matching methods and batch sizes for task 3
- Run more training on task 3 to see if batching can result in lower loss