Recurrent Neural Networks for Text Classification

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Wednesday 6th May, 2020 10:29

Preamble

In a neural network with feedback, each output creates a context for the next input.

This can be useful in data processing applications that produce variable-length input data for a neural network. Such applications include text processing, natural language translation, speech understanding, etc.

Focusing on text processing, the meaning of a word in a sentence is often ambiguous if the word is considered in isolation from what came before it. In general, as you are reading text, in order to understand what a word or a phrase is saying, you have to bring to bear on it what you have already looked at. That is, as you read text, you understand each word and each phrase in the context created by what you have read so far.

As you will see in this lecture, such contexts can be created by a neural network with feedback and the estimated contexts used to interpret each new input.

Preamble (contd.)

Neural networks with feedback are more commonly known as Recurrent Neural Networks. The `torch.nn` module now also comes with a class named `torch.nn.RNN` that saves you the bother of having to be explicit about the feedback required by such networks. Here is the documentation page for this class:

`https://pytorch.org/docs/stable/nn.html#recurrent-layers`

The goal of this lecture is to:

- Introduce you to recurrent neural networks with the help of a couple of examples;
- Point to the fact that, in general, the vanishing gradient problem is even more acute for recurrent neural networks on account of the long chains of dependencies created by the feedback; and
- To talk about the gating mechanisms that are used to get around this challenge.

In particular, I'll focus on the problem of what is known as the Sentiment Analytics when highlighting the challenges of using RNNs for this application.

Preamble (contd.)

Version 1.1.4 of DLStudio includes the code shown in this lecture and also the Sentiment Analysis dataset I have constructed from the publicly available user-feedback data provided by Amazon for the year 2007.

The over 1 GB compressed archive made available by Amazon contains user feedback on 25 product categories. Each product category contains a file with the positive feedback comments and a file with the negative feedback comments. These have been marked so by human annotators. These are in addition to a lot of other information made available by Amazon. I have separated out just a designated number of the positive and the negative comments for training RNNs.

Since the ultimate motivation for a novel engineering solution is the problem itself, I'll start this lecture by first talking about the problem before delving into how it may be solved with a neural network with feedback.

However, before starting the main lecture, on the next slide I want to bring to your attention a couple of datasets at a PyTorch website that I believe are highly relevant to this lecture.

Preamble (contd.)

The following webpage at PyTorch:

`https://pytorch.org/text/datasets.html`

mentions the following two datasets:

1. AmazonReviewPolarity
2. AmazonReviewFull

that, in all likelihood, are based on the same Amazon archive that I mentioned on the previous slide.

You may want to click on the first item in particular because — I think — it may be curated in the same manner as the datasets I provide through version 1.1.4 of DLStudio. If that's the case, I'd love to hear from you. I'd rather you use an "official" PyTorch dataset along with its dataloader rather than the version I provide along with my customization of the dataloader.

Outline

Outline

What is Sentiment Analytics?

- [Gartner:] Social analytics is monitoring, analyzing, measuring and interpreting digital interactions and relationships of people, topics, ideas and content. Interactions occur in workplace and external-facing communities.
- [IBM:] They talk about "Conducting social listening", "Enhancing customer service", "Integrating with Chatbots",
- [Accenture:] Natural Language Processing (NLP) is being integrated into our daily lives with virtual assistants like Siri, Alexa, or Google Home. In the enterprise world, NLP has become essential for businesses to gain a competitive edge. Consider the valuable insights hidden in your enterprise unstructured data: text, email, social media, videos, customer reviews, reports, etc. NLP applications are a game changer, helping enterprises analyze and extract value from this unstructured data.

Outline

The Amazon User Feedback Dataset

- The user feedback shown below is from product category "book" and is reproduced from the file "positive.review". What that means is that it is a user comment that was labeled by a human as being a positive comment. The part that we are interested in is between the tags `<review _text>` and `</review _text>`.

```
<review>
<unique_id>
188105201X:excellent_resource_for_principals!:onickre@mail.milwaukee.k12.wi.us
</unique_id>
<unique_id>
3294
</unique_id>
<asin>
188105201X
</asin>
<product_name>
Leadership and the New Science: Learning About Organization from an Orderly Universe: Books: Margaret J. Wheatley
</product_name>
<product_type>
books
</product_type>
<product_type>
books
</product_type>
<helpful>
3 of 3
</helpful>
<rating>
5.0
</rating>
<title>
Excellent resource for principals!
</title>
<date>
July 6, 1999
</date>
<reviewer>
ONICKRE@mail.milwaukee.k12.wi.us
</reviewer>
<reviewer_location>
Milwaukee, Wisconsin
</reviewer_location>
<review_text>
I am ordering copies for all 23 middle school principals and the two assistant principals leading two middle school programs
 in the Milwaukee Public Schools system. We will use Wheatley's book as the primary resource for our professional growth at
 our MPS Middle School Principals Collaborative institute August 9-11, 1999. We are not just concerned with reform; we seek
 renewal as well. Wheatley provides the basis. She notes that Einstein said that a problem cannot be solved from the same
 consciousness that created it. The entire book is a marvelous exploration of this philosophy
</review_text>
</review>
```

The Amazon User Feedback Dataset (contd.)

- The dataset consists of the following 25 merchandise categories:

```
apparel                  computer_&_video_games   kitchen_&_housewares     sports_&_outdoors
automotive               dvd                      magazines                tools_&_hardware
baby                     electronics              music                    toys_&_games
beauty                   gourmet_food             musical_instruments      video
books                    grocery                  office_products
camera_&_photo           health_&_personal_care   outdoor_living
cell_phones_&_service    jewelry_&_watches        software
```

- Each item in the listing shown above is a directory and each such directory contains the files listed below. Thought you might also like to see how large some of these files can be.

```
total 2925088
drwxr-xr-x  2 kak kak       4096 Apr 13 18:00 ./
drwxr-xr-x 27 kak kak       4096 Apr 11 14:05 ../
-rwxr-xr-x  1 kak kak 1413818930 May  6  2007 all.review*
-rwxr-xr-x  1 kak kak    1519069 Sep 10  2007 negative.review*
-rwxr-xr-x  1 kak kak    1423124 Apr 13 18:00 positive.review*
-rwxr-xr-x  1 kak kak  134420935 May  6  2007 processed.review*
-rwxr-xr-x  1 kak kak   33201221 May  6  2007 processed.review.balanced*
-rwxr-xr-x  1 kak kak 1410876740 May  4  2007 unlabeled.review*
```

The Amazon User Feedback Dataset (contd.)

- Our interest is primarily in the files `positive.reviews` and `negative.reviews`. If you'd like to know how many reviews are there in each of these two files, that information is provided in a file named `summary.txt` in the same directory that has all the product categories:

```
apparel/negative.review 1000
apparel/positive.review 1000
apparel/unlabeled.review 7252
automotive/negative.review 152
automotive/positive.review 584
baby/negative.review 900
baby/positive.review 1000
baby/unlabeled.review 2356
beauty/negative.review 493
beauty/positive.review 1000
beauty/unlabeled.review 1391
books/negative.review 1000
books/positive.review 1000
books/unlabeled.review 973194
camera & photo/negative.review 999
camera & photo/positive.review 1000
camera & photo/unlabeled.review 5409
cell phones & service/negative.review 384
cell phones & service/positive.review 639
computer & video games/negative.review 458
computer & video games/positive.review 1000
computer & video games/unlabeled.review 1313
dvd/negative.review 1000
dvd/positive.review 1000
dvd/unlabeled.review 122438
electronics/negative.review 1000
electronics/positive.review 1000
electronics/unlabeled.review 21009
gourmet food/negative.review 208
gourmet food/positive.review 1000
gourmet food/unlabeled.review 367
grocery/negative.review 352
grocery/positive.review 1000
grocery/unlabeled.review 1280
health & personal care/negative.review 1000
health & personal care/positive.review 1000
health & personal care/unlabeled.review 5225
jewelry & watches/negative.review 292
jewelry & watches/positive.review 1000
jewelry & watches/unlabeled.review 689
kitchen & housewares/negative.review 1000
kitchen & housewares/positive.review 1000
kitchen & housewares/unlabeled.review 17856
magazines/negative.review 970
magazines/positive.review 1000
magazines/unlabeled.review 2221
music/negative.review 1000
music/positive.review 1000
music/unlabeled.review 172180
musical instruments/negative.review 48
musical instruments/positive.review 284
office products/negative.review 64
office products/positive.review 367
outdoor living/negative.review 327
outdoor living/positive.review 1000
outdoor living/unlabeled.review 272
software/negative.review 915
```

Sentiments Analysis Dataset

- Version 1.1.4 of DLStudio comes with the following text dataset archive files:

```
sentiment_dataset_train_400.tar.gz          vocab_size = 64,350
sentiment_dataset_test_400.tar.gz

sentiment_dataset_train_200.tar.gz          vocab_size = 43,285
sentiment_dataset_test_200.tar.gz

sentiment_dataset_train_40.tar.gz           vocab_size = 17,001
sentiment_dataset_test_40.tar.gz
```

- The archive names with, say, '400' in them were made using the first 400 positive and 400 negative comments in each of the 25 product categories. The reviews thus collected are randomized and divided into the training and the testing datasets in 80:20 ratio.

- When you download the above two datasets, you will also get the following two files:

```
sentiment_dataset_train_3.tar.gz            vocab_size = 3,402
sentiment_dataset_test_3.tar.gz
```

  These are to help you debug your code quickly.

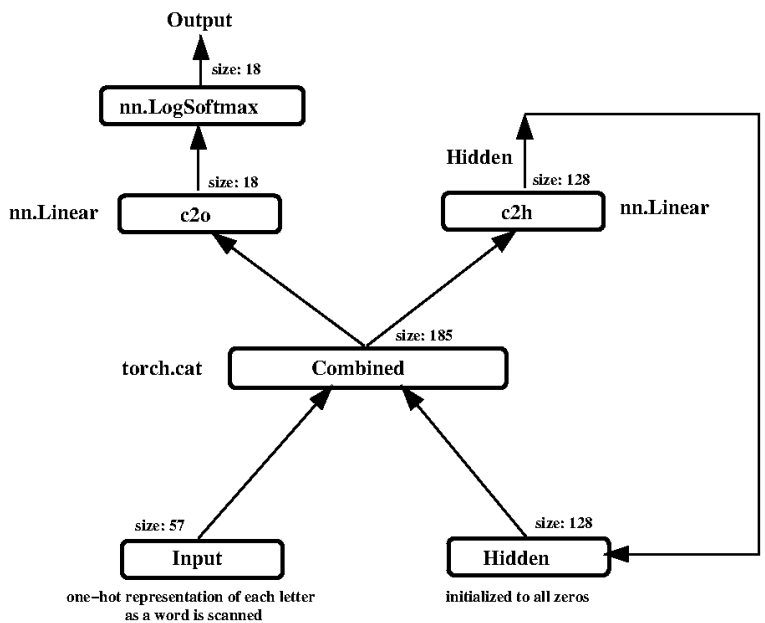
Outline

A Simple (But Fun) Example of a Neural Network with Feedback

- This example in a tutorial by Sean Robertson has got to be one of the most wonderful illustrations of how much fun you can have with a neural network when it is provided with feedback. Here is a link to this example:

  `https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html`

  The goal is to predict the ethnic origin of a last name.



A Simple But Fun Example of RNN (contd.)

Meanings associating with the legends in the figure shown on the previous slide:

- input: It is a one-hot vector representation of each character as a last name is scanned one character at a time.
- hidden: This is the recurrent hidden state that is updated at each step in in a character-by-character scan of a last name.
- combined: All that happens here is calling `torch.cat` to concatenate the input with the current value of the hidden state.
- c2o: stands for "combined-to-output", where "combined" means the concatenation of the input and the hidden state. This is neural layer of type `nn.Linear`.
- c2h: stands for "combined-to-hidden", which is also a neural layer of type `nn.Liner`. Its purpose is to generate the next value for the hidden state.
- output: This is an 18-element vector whose values shows the probabilities of the last name belonging to one of 18 different ethnicities.

Predicting the Ethnicity of a Last Name

- Each loop of training in the code for this example does the following:
  1. Create the input tensor for the last name and a tensor for the output ethnicity
  2. Create a zeroed initial hidden state
  3. Scan the last name one character at a time and feed its one-hot vector at the input
  4. Update the hidden state for the next character in the last name
  5. Compare the final output to the target ethnicity
  6. Backpropagate
  7. Return the output and loss

- The important thing to note that the hidden state is initialized to zeros for each new training sample, meaning for each new last name and its associated ethnic origin.

- All the last names for a given ethnicity are placed in a single file and the name of the file designates the ethnic origin of the names in that file

The Network

- Shown below is code that defines the network depicted on Slide 15.
- The size of the input is dictated by the number of printable ascii characters (without including the digits, etc.)
- The size of the hidden state was set experimentally.

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        # input_size=57  hidden_size=128
        self.c2h = nn.Linear(input_size + hidden_size, hidden_size)
        # output_size=18
        self.c2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)
    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.c2h(combined)
        output = self.c2o(combined)
        output = self.softmax(output)
        return output, hidden
```

Some Utility Functions

- The code for the RNN demo uses the utility functions defined on the next slide. We first define in line (A) the character set for the last names. This is done by calling `string.ascii_letters`, which returns the 52 letters a-z and A-Z. The statement in line (A) also appends to these 52 letter five additional punctuation characters and the blank space character, for a total of 57 characters.

- The function in line (B) on the next slide returns the index of a given letter in character set defined in line (A). This index is used in the function in line (C) to create a one-hot representation of a letter.

- In a one-hot vector representation, all the elements of the vector are zero except at the position that corresponds to the letter in question in a specified character set.

Some Utility Functions (contd.)

- The function in line (D) below is defined for interpreting the output of the neural network. The call `output.topk(1)` returns two values: one, the largest value, and, two, the index in the 18-element output that has the largest value. Both of these values are returned as one-element tensors.
- The function in line (E) returns one training sample — meaning one last name and its associated ethnicity —- chosen randomly from all the data.

```
all_letters = string.ascii_letters + " .,;'"                              ## (A)
n_letters = len(all_letters)                    ## 57 chars

def letterToIndex(letter):                                                ## (B)
    return all_letters.find(letter)

def letterToTensor(letter):                                               ## (C)
    tensor = torch.zeros(1, n_letters)
    tensor[0][letterToIndex(letter)] = 1
    return tensor

def categoryFromOutput(output):                                           ## (D)
    top_n, top_i = output.topk(1)
    category_i = top_i[0].item()
    return all_categories[category_i], category_i

def randomTrainingExample():                                              ## (E)
    category = randomChoice(all_categories)
    line = randomChoice(category_lines[category])
    category_tensor = torch.tensor([all_categories.index(category)], dtype=torch.long)
    line_tensor = lineToTensor(line)
    return category, line, category_tensor, line_tensor
```

The train() Function

- Note how the basic training function carries out the iterations involved in scanning the input last name one character at a time.
- Also, instead of using epochs, in this case, we will simply select randomly from all the training data and do so as many times as we wish for achieving basically the same effect that you get by training over several epochs.

```
criterion = nn.NLLLoss()
learning_rate = 0.005

def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()
    rnn.zero_grad()          ## zeros out the gradients for learnable params
    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)
    loss = criterion(output, category_tensor)
    loss.backward()
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)
    return output, loss.item()
```

The train() Function (contd.)

- About the loss function `nn.NLLLoss()`, it stands for "Negative Log Likelihood Loss". This is the loss function to use if the output of your network is produced by a `nn.LogSoftmax` layer.
- The `nn.NLLLoss()` loss function expects at its input log-probabilities for the different classes — such as those produced by `nn.LogSoftmax`.
- One can show theoretically that the combined effect of `nn.LogSoftmax` in the output of a network and `nn.NLLLoss()` for loss is exactly the same as using `CrossEntropyLoss` for loss.
- For iterative training, we repeated call `train()` a large number of times over the training dataset as shown below
- The variable `line_tensor` is the tensor representation of the last name in a training file. If a last name has, say, 5 characters, this tensor has shape $(5, 57)$.

Outline

A Recurrent Network for Sentiment Analysis

- This section presents an attempt at solving the sentiment analysis problem. Shown on the next slide are some of the utility functions for this exercise.
- First we must define the functions that convert the text into tensor representations. Toward that end, as shown by the function whose definition starts at line (A) on Slide 26, each word is given a one-hot representation that, as you would expect, depends on the size of the vocabulary (which, by the way, comes with the datasets that I provide).
- The size of the vocabulary is 17,001 if you only scrape 40 positive reviews and an equal number of negative reviews from each product category. However, the size of the vocabulary goes up to 43,285 when the number of reviews collected for each product type is 400. For this sized vocabulary, the one-hot representation for a word will involve tensors that are as large as 43,285.

A Recurrent Network for Sentiment Analysis (contd.)

- To continue from the previous slide, and if a review has a couple of hundred words in it (not uncommon), you are looking at the tensor representation of a review whose shape could be $(200, 43285)$
- The tensor representation of a review is generated by the function whose definition starts in line (B) on the next slide.
- We also need to convert the sentiment associated with a review into a tensor. In the dataset that I provide, the negative sentiment is represented by the integer 0 and the positive by the integer 1. The function whose definition starts in line (C) on the next slide does the job of converting these numbers into tensors.
- The functions in lines (D) and (E) are the required functions for the custom dataloader.

The Utility Functions

```
def one_hotvec_for_word(self, word):                                          ## (A)
    word_index =  self.vocab.index(word)
    hotvec = torch.zeros(1, len(self.vocab))
    hotvec[0, word_index] = 1
    return hotvec

def review_to_tensor(self, review):                                           ## (B)
    review_tensor = torch.zeros(len(review), len(self.vocab))
    for i,word in enumerate(review):
        review_tensor[i,:] = self.one_hotvec_for_word(word)
    return review_tensor

def sentiment_to_tensor(self, sentiment):                                     ## (C)
    """
    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 is 1:
        sentiment_tensor[1] = 1
    elif sentiment is 0:
        sentiment_tensor[0] = 1
    sentiment_tensor = sentiment_tensor.type(torch.long)
    return sentiment_tensor

def __len__(self):                                                            ## (D)
    if self.train_or_test is 'train':
        return len(self.indexed_dataset_train)
    elif self.train_or_test is 'test':
        return len(self.indexed_dataset_test)

def __getitem__(self, idx):                                                   ## (E)
    sample = self.indexed_dataset_train[idx] if self.train_or_test is '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
```

Sentiment Analysis Network

- Shown below is a vanilla implementation of a network for sentiment analysis — it does not lend itself to the use of any gates for protecting against the vanishing or exploding gradients.

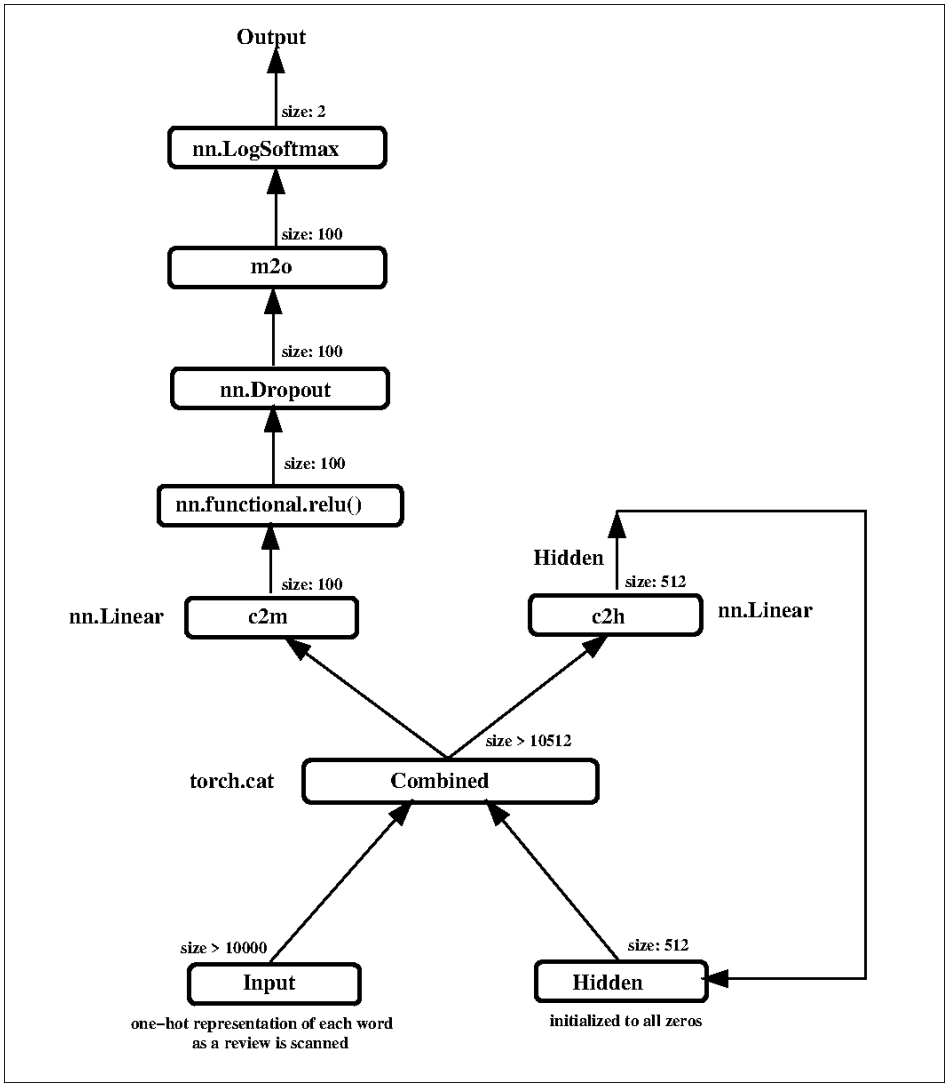
```
class TEXTnet(nn.Module):

    def __init__(self, input_size, hidden_size, output_size):
        super(DLStudio.TextClassification.TEXTnet, 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 + hidden_size, hidden_size)
        self.combined_to_middle = nn.Linear(input_size + 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)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.combined_to_hidden(combined)
        out = self.combined_to_middle(combined)
        out = torch.nn.functional.relu(out)
        out = self.dropout(out)
        out = self.middle_to_out(out)
        out = self.logsoftmax(out)
        return out,hidden
```

The TEXTnet Network

- The network defined on the previous slide is shown below:



The Training Function Used for the TEXTnet Network

- In the training code that follows note how in line (A) we reinitialize the hidden state to all zeros for each review.
- Note how a review, which may consist of any number of words, is scanned one word at a time in lines (B), (C), and (D) and how the one-hot vector for each new word is combined with the value of the hidden that summarizes all the words seen previously in that review.

```
def run_code_for_training_with_TEXTnet_no_gru(self, net, hidden_size):
    net = net.to(self.dl_studio.device)
    criterion = nn.NLLLoss()
    optimizer = optim.SGD(net.parameters(), lr=self.dl_studio.learning_rate, momentum=self.dl_studio.momentum)
    start_time = time.clock()
    for epoch in range(self.dl_studio.epochs):
        running_loss = 0.0
        for i, data in enumerate(self.train_dataloader):
            hidden = torch.zeros(1, hidden_size)                                    ## (A)
            hidden = hidden.to(self.dl_studio.device)
            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)
            optimizer.zero_grad()
            input = torch.zeros(1,review_tensor.shape[2])
            input = input.to(self.dl_studio.device)
            for k in range(review_tensor.shape[1]):                                 ## (B)
                input[0,:] = review_tensor[0,k]                                     ## (C)
                output, hidden = net(input, hidden)                                 ## (D)
            loss = criterion(output, torch.argmax(sentiment,1))
            running_loss += loss.item()
            loss.backward(retain_graph=True)
            optimizer.step()
            if i % 100 == 99:
                avg_loss = running_loss / float(100)
                current_time = time.clock()
                time_elapsed = current_time-start_time
                print("[epoch:%d  iter:%4d  elapsed_time: %4d secs]     loss: %.3f" % (epoch+1,i+1, time_elapsed,avg_loss))
                running_loss = 0.0
    self.save_model(net)
```

The Testing Function Used for the TEXTnet Network

- As in the training function, in the testing function also we reinitialize in line (A) the hidden state to all zeros for each new unseen review.
- In lines (B), (C), and (D), the review is scanned one word at a time and, for each new word, its one-hot vector concatenated with the hidden state that represents all the words seen preiously in the review.

```
def run_code_for_testing_with_TEXTnet_no_gru(self, net, hidden_size):
    net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
    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']
            input = torch.zeros(1,review_tensor.shape[2])
            hidden = torch.zeros(1, hidden_size)                                    ## (A)
            for k in range(review_tensor.shape[1]):                                 ## (B)
                input[0,:] = review_tensor[0,k]                                     ## (C)
                output, hidden = net(input, hidden)                                 ## (D)
            predicted_idx = torch.argmax(output).item()
            gt_idx = torch.argmax(sentiment).item()
            if i % 100 == 99:
                print("   [i=%4d]    predicted_label=%d       gt_label=%d" % (i+1, predicted_idx,gt_idx))
            if gt_idx is 0:
                negative_total += 1
            elif gt_idx is 1:
                positive_total += 1
            confusion_matrix[gt_idx,predicted_idx] += 1
    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\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)
```

Results Obtained with the TEXTnet Network

- The results shown on this slide are based on the following dataset that was described earlier on Slide 13:

```
sentiment_dataset_train_40.tar.gz
sentiment_dataset_test_40.tar.gz
```

- Using one epoch of training, shown below is the confusion matrix that is produced by the network presented on the previous two slides with the learning rate set to 10^{-5}:

```
Number of unseen positive reviews tested: 200
Number of unseen negative reviews tested: 195

Displaying the confusion matrix:

                    predicted negative    predicted positive

true negative:              0.0                 100.0

true positive:              0.0                 100.0
```

- These results were produced by Python 3 execution of the script `text_classification_with_TEXTnet_no_gru.py` in the Examples directory of DLStudio, Version 1.1.4.

Results with the TEXTnet Network (contd.)

- The dismal results shown on the previous slide are, in all likelihood, a result of a combination of the vanishing gradients and the very small size of the training data.
- These poor results are in keeping with the fact that, as shown below, the loss does not exhibit any decrease with training:

```
[epoch:1  iter: 100  elapsed_time:   26 secs]     loss: 0.693
[epoch:1  iter: 200  elapsed_time:   61 secs]     loss: 0.694
[epoch:1  iter: 300  elapsed_time:   94 secs]     loss: 0.694
[epoch:1  iter: 400  elapsed_time:  133 secs]     loss: 0.692
[epoch:1  iter: 500  elapsed_time:  164 secs]     loss: 0.696
[epoch:1  iter: 600  elapsed_time:  197 secs]     loss: 0.693
[epoch:1  iter: 700  elapsed_time:  228 secs]     loss: 0.692
[epoch:1  iter: 800  elapsed_time:  254 secs]     loss: 0.693
[epoch:1  iter: 900  elapsed_time:  288 secs]     loss: 0.690
[epoch:1  iter:1000  elapsed_time:  322 secs]     loss: 0.694
[epoch:1  iter:1100  elapsed_time:  358 secs]     loss: 0.694
[epoch:1  iter:1200  elapsed_time:  394 secs]     loss: 0.694
[epoch:1  iter:1300  elapsed_time:  427 secs]     loss: 0.694
[epoch:1  iter:1400  elapsed_time:  466 secs]     loss: 0.694
[epoch:1  iter:1500  elapsed_time:  501 secs]     loss: 0.690
```

- **AN IMPORTANT NOTE:** About the role played by the small size of the training dataset (which has only 40 positive and 40 negative reviews for each product category) in the results presented on the previous slide, note that its vocabulary size as shown in Slide 13 is 17,001. This would also be the size of the one-hot vectors for the words. If we were to use the larger datasets mentioned on that slide, the size of the one-hot vectors would go up also, which would increase the number of learnable parameters, and that, in turn, would create a need for a still larger dataset. This is a catch-22 situation that can only be solved by using, say, the fixed-size word embeddings for the words (see the Week 14 slides) as opposed to the one-hot vectors.

A Stepping Stone to Gating — The TEXTnetOrder2 Network

- Shown on the next slide is an attempt at making stronger the feedback mechanism in TEXTnet by incorporating in it the value of the hidden state at the previous time step.
- The third parameter '`cell`' in the definition of `forward()`' plays an important role in how `TEXTnetOrder2` lends itself to some rudimentary gating action. See the explanation for the training function for why that is the case.
- As shown in line (C), the current value of hidden is processed by a linear layer, followed by the sigmoid nonlinearity, for its storage in the outgoing value of `cell`.
- The network that results from the definition on the next slide is shown in Slide 35.

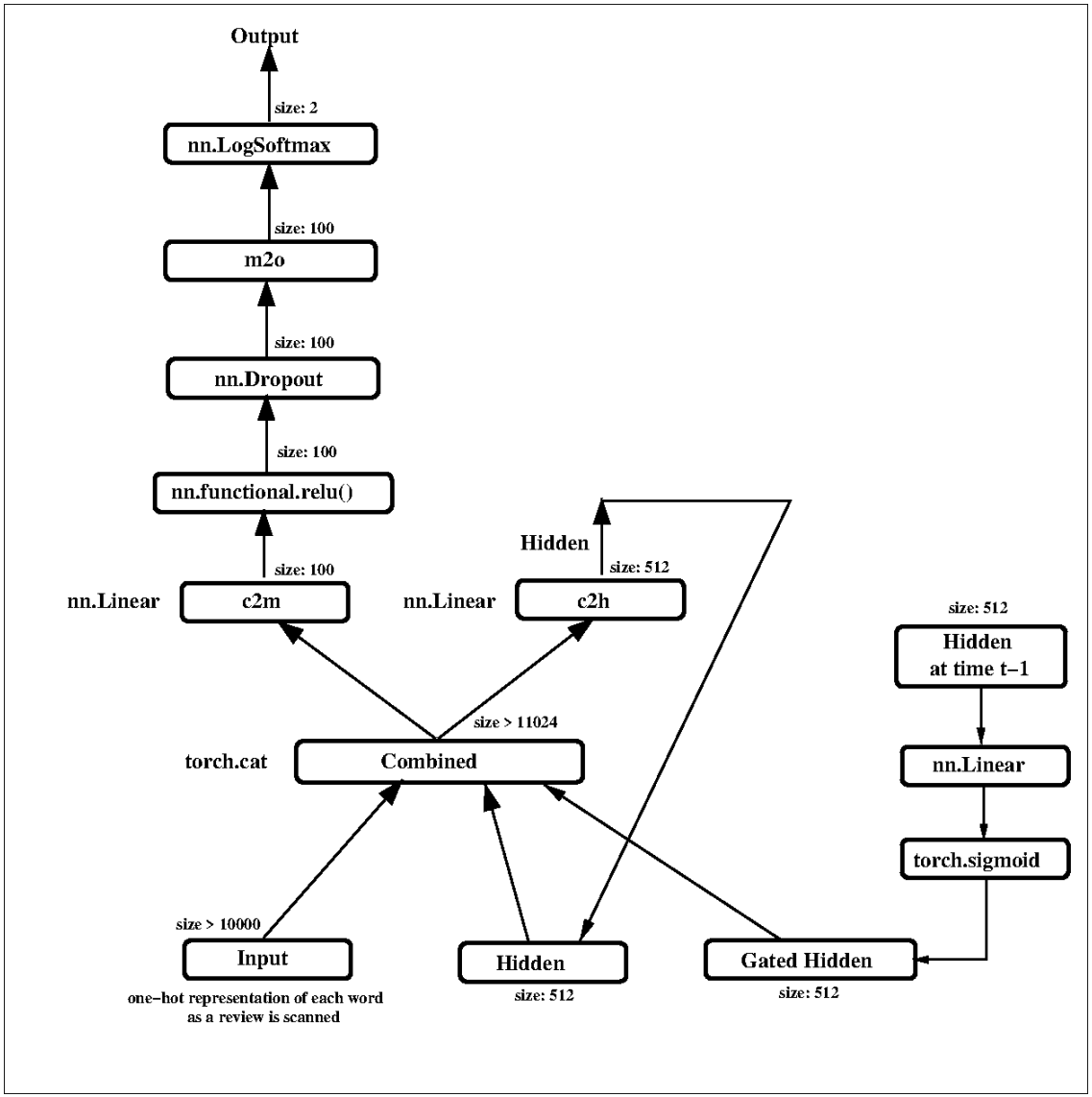
The TEXTnetOrder2 Network

```
class TEXTnetOrder2(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, dls):
        super(DLStudio.TextClassification.TEXTnetOrder2, 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)                  ## (A)

    def forward(self, input, hidden, cell):                                         ## (B)
        combined = torch.cat((input, hidden, cell), 1)
        hidden = self.combined_to_hidden(combined)
        out = self.combined_to_middle(combined)
        out = torch.nn.functional.relu(out)
        out = self.dropout(out)
        out = self.middle_to_out(out)
        out = self.logsoftmax(out)
        hidden_clone = hidden.clone()
        cell = torch.sigmoid(self.linear_for_cell(hidden_clone))                    ## (C)
        return out,hidden,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
```

The TEXTnetOrder2 Network (contd.)



The Training Function Used for the TEXTnetOrder2 Network

- The local variables `cell_prev` and `cell_prev_2_prev` defined in lines (A) and (B) allow for the value of the hidden state at the previous time step to be factored into the logic at the current time-step in line (F). As to how a review is scanned word by word is the same as for the `TEXTnet` network. Also note how the hidden state is initialized to all zeros in line (C) for each product review.

```
def run_code_for_training_with_TEXTnetOrder2_no_gru(self, net, hidden_size):
    net = net.to(self.dl_studio.device)
    criterion = nn.NLLLoss()
    optimizer = optim.SGD(net.parameters(), lr=self.dl_studio.learning_rate, momentum=self.dl_studio.momentum)
    start_time = time.clock()
    for epoch in range(self.dl_studio.epochs):
        running_loss = 0.0
        for i, data in enumerate(self.train_dataloader):
            cell_prev = net.initialize_cell(1).to(self.dl_studio.device)                ## (A)
            cell_prev_2_prev = net.initialize_cell(1).to(self.dl_studio.device)         ## (B)
            hidden = torch.zeros(1, hidden_size)                                        ## (C)
            hidden = hidden.to(self.dl_studio.device)
            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)
            optimizer.zero_grad()
            input = torch.zeros(1,review_tensor.shape[2])
            input = input.to(self.dl_studio.device)
            for k in range(review_tensor.shape[1]):                                     ## (D)
                input[0,:] = review_tensor[0,k]                                         ## (E)
                output, hidden, cell = net(input, hidden, cell_prev_2_prev)             ## (F)
                if k == 0:                                                              ## (G)
                    cell_prev = cell                                                    ## (H)
                else:                                                                   ## (I)
                    cell_prev_2_prev = cell_prev                                        ## (J)
                    cell_prev = cell                                                    ## (K)
            loss = criterion(output, torch.argmax(sentiment,1))
            running_loss += loss.item()
            loss.backward()
            optimizer.step()
            if i % 100 == 99:
                avg_loss = running_loss / float(100)
                current_time = time.clock()
                time_elapsed = current_time-start_time
                print("[epoch:%d  iter:%4d  elapsed_time: %4d secs]     loss: %.3f" % (epoch+1,i+1, time_elapsed,avg_loss))
                running_loss = 0.0
    self.save_model(net)
```

The Testing Function Used for the TEXTnetOrder2 Network

- Like the training function shown on the previous slide, the testing function also uses the local variables `cell_prev` and `cell_prev_2_prev` defined in lines (A) and (B) to allow for the value of the hidden state at the previous time step to be factored into the logic at the current time-step in line (D).

```
def run_code_for_testing_with_TEXTnetOrder2_no_gru(self, net, hidden_size):
    net.load_state_dict(torch.load(self.dl_studio.path_saved_model))
    negative_total = 0
    positive_total = 0
    confusion_matrix = torch.zeros(2,2)
    with torch.no_grad():
        for i, data in enumerate(self.test_dataloader):
            cell_prev = net.initialize_cell(1)                                          ## (A)
            cell_prev_2_prev = net.initialize_cell(1)                                   ## (B)
            review_tensor,category,sentiment = data['review'], data['category'], data['sentiment']
            input = torch.zeros(1,review_tensor.shape[2])
            hidden = torch.zeros(1, hidden_size)                                        ## (C)
            for k in range(review_tensor.shape[1]):
                input[0,:] = review_tensor[0,k]
                output, hidden, cell = net(input, hidden, cell_prev_2_prev)             ## (D)
                if k == 0:                                                              ## (E)
                    cell_prev = cell                                                    ## (F)
                else:                                                                   ## (G)
                    cell_prev_2_prev = cell_prev                                        ## (H)
                    cell_prev = cell                                                    ## (I)
            predicted_idx = torch.argmax(output).item()
            gt_idx = torch.argmax(sentiment).item()
            if gt_idx is 0:
                negative_total += 1
            elif gt_idx is 1:
                positive_total += 1
            confusion_matrix[gt_idx,predicted_idx] += 1
    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))
    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)
```

Results Obtained with TEXTnetOrder2 Network

- The results shown on this slide are with the same dataset that was used earlier for the `TEXTnet` network:

```
sentiment_dataset_train_40.tar.gz
sentiment_dataset_test_40.tar.gz
```

- Using one epoch of training, shown below is the confusion matrix that is produced by the `TEXTnetOrder2` network with the learning rate set to 10^{-5}:

```
Number of unseen positive reviews tested: 200
Number of unseen negative reviews tested: 195

Displaying the confusion matrix:

                     predicted negative    predicted positive

true negative:             33.84                 66.15

true positive:             30.0                  70.0
```

- These results were produced by Python 3 execution of the script `text_classification_with_TEXTnetOrder2_no_gru.py` in the Examples directory of DLStudio, Version 1.1.4.

Outline

Why We Need the Gating Mechanisms

- For non-trivial problems, the backpropagation of loss in an RNN involves long chains of dependencies because it must span all previous values of the hidden state that contributed to the present value at the output.
- This leads to an even more challenging version of the vanishing gradients problem that you saw earlier in deep neural networks (with no feedback).
- In the chains of dependencies created by feedback, the short-term dependencies can completely dominate the long-term dependencies, which basically negates what you had hoped to achieve with feedback.
- In the literature you will find two commonly used gating mechanisms to deal with the vanishing gradients problem in RNNs: LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit), with the latter being the more recent.

The Basic Idea of Gating

- The basic idea of a gating mechanism is that you designate a special variable (usually referred to as the cell) that is the keeper of information from the past. What *was* placed in the cell is subject to being forgotten if it is not so relevant to the current state of the input/output relationship. At the same time, the cell can be updated based on the current input/output relationship if that is deemed to be important for future characterizations of the input.
- The previous slide mentioned the two most commonly used gated RNNs as GRUs and LSTMs. Here is a wonderful paper by Chung et al. that has carried out a comparative evaluation of the two:

  `https://arxiv.org/abs/1412.3555`

- In the rest of this section, I'll first present some preliminary concepts drawn from the above paper, which will be followed by an explanation of the structure of a GRU.

The Hidden State and its Evolution

- Let's denote the input sequence that is scanned by the network one element at a time by

$$\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$$

- In general, any joint distribution over these T variables can be decomposed in the following manner:

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_T) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_3|\mathbf{x}_1, \mathbf{x}_2)\ldots p(\mathbf{x}_T|\mathbf{x}_1, \ldots, \mathbf{x}_{T-1})$$

- Let's say that we can postulate the existence of a variable, denoted $\mathbf{h}_t$, that allows us to write the following form for any of the component distributions shown at right above:

$$p(\mathbf{x}_t|\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}) = \phi(\mathbf{h}_t)$$

where $\phi()$ is a bounded and differentiable function like certain activation functions such as the sigmoid and the hyperbolic tangent function (`nn.tanh()`).

The Hidden State and its Evolution (contd.)

- The relationship at the bottom of the previous slide says that our *variable length* input sequence $\mathbf{x}_i$ is such that the probabilistic dependence of each sample on all the previous samples is dictated by the time evolution of a specific *fixed-sized entity* $\mathbf{h}_t$ through a bounded and differentiable $\phi()$.
- Input sequences that admit $\mathbf{h}_t$'s that obey the condition presented on the previous slide can be processed by the gating mechanisms such as GRU and LSTM for the remediation of the vanishing gradients problem. The function $\mathbf{h}_t$ is referred to as the hidden state of the network at time step t.
- The explanation so far has focused on how an input sequence may be generated from the time evolution of a hidden state. However, that implies the assumption that we are already in possession of the hidden state at time steps t. [By the way, that's exactly what happens in the encoder-decoder frameworks used in neural-network based approaches to machine translation: The encoder learns the fixed-sized hidden state for a given input sentence in one language and the decoder maps that hidden state to a sentence in the target language.]

The Hidden State and its Evolution (contd.)

- The last bullet on the previous slide begs the question: How to generate the hidden state in the first place? Since the "job" of the hidden state is to generate the input sequence, and considering the differentiability and the boundedness assumptions we made in the relationship between $\mathbf{x}_i$ and $\mathbf{h}_t$, the following form for learning the hidden state seems reasonable:

$$\mathbf{h}_t = \begin{cases} 0, & t = 0 \\ g(W\mathbf{x}_t + U\mathbf{h}_{t-1}), & \textit{otherwise} \end{cases}$$

  where $g()$ is again a bounded and differentiable function as before and where W and U are the matrices of learnable parameters.

- The important thing to focus on in the equation shown above that the value of the hidden state at time-step t depends nonlinearly on its value at the previous time-step $t-1$.

The Hidden State and its Evolution (contd.)

- If you couple the insight presented at the bottom of the previous slide with the feedback diagram presented on Slide 15, you can see that with our new model for the input sequence and for the time evolution of the hidden state, each element of the output sequence in that diagram will acquire a longer-range dependence on the current *and* the previous samples of the input sequence. That's one element of what powers gated RNNs: A gated RNN can choose to forget the past (and to possibly give greater emphasis to the present) over a longer range of dependencies into the past. The act of forgetting and resetting can be implemented with the help of gates as you will see in the next section.
- In the next few slides, let's see how the GRU exploits the model presented here for its gating action.

Outline

GRU — Reset and Update Gates

- I'll now focus on GRUs (Gated Recurrent Unit) in the rest of the discussion here. GRUs were first proposed in the paper by Cho et el. that you can access through the following link:

  `https://arxiv.org/abs/1409.1259`

- Shown in the figure below is a high-level diagrammatic representation of a GRU. Basically, it has two "gates", denoted r and z that stand for the reset gate and the update gate. As to why they are referred to as "gates" will become clear shortly.

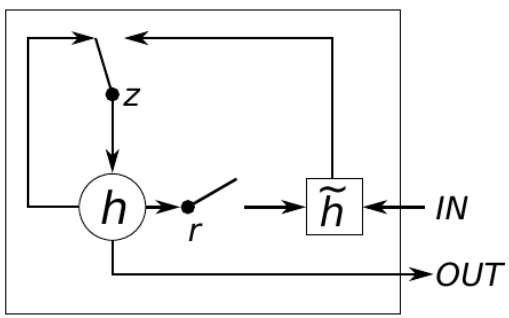


Figure: This figure, taken from the GRU vs. LSTM comparative evaluation paper cited earlier, is a pictorial depiction of a GRU. The reset and the update gates of a GRU are show as r and z in the figure, whereas h and $\tilde{h}$ are the current value for the hidden state and a candidate value for the same.

GRU – Updating the Hidden State

- The input into a GRU consists of the ongoing values for the sequences **x** and **h** where the latter represents the hidden state for the former.
- The figure shown on the previous slide seeks to depict the following relationship between the value of the hidden state at time-step t and the same at time-step $t-1$:

$$h_t^j \;=\; (1 - z_t^j) h_{t-1}^j \;+\; z_t^j \tilde{h}_t^j$$

where h_t^j is the j^{th} element of the vector $\mathbf{h}_t$ and where the *update gate* z_t^j is given by

$$z_t^j \;=\; \sigma(W_z \mathbf{x}_t \;+\; U_z \mathbf{h}_{t-1})^j$$

where σ is a logistic sigmoid function given by $\sigma(x) = \frac{1}{1+\exp^{-x}}$. It smoothly rises from 0 to 1 over its entire domain. Its value at $x = 0$ is 0.5. W_z and U_z are matrices of learnable parameters for the update gate.

Why Call it a Gate?

- With regard to the last equation shown on the previous slide, note the implications of the logistic sigmoid nonlinearity with regard to how each component of **h** gets updated.
- Just for illustration, assume that the argument to the $\sigma()$ is such that it is either sufficient negative or sufficiently positive so that the output of $\sigma()$ is either 0 or 1.
- When $\sigma()$ returns 0, for that component of **h**, the update gate will evaluate to 0 and, therefore, the previous value of the hidden state will dominate its current value.
- On the other hand, should $\sigma()$ return 1, the previous value for the hidden state will be ignored and the new value for the hidden state will be dominated by $\tilde{\mathbf{h}}_t$. That should explain why we may refer to $\mathbf{z}_t$ as a gate that decides the fate of each component of **h** separately. But what is $\tilde{\mathbf{h}}_t$?

GRU – Equation for the Reset Gate

- The new thing $\tilde{\mathbf{h}}_t$ that you see in the update equation for z_t^j depends on the current value for the sequence $\mathbf{x}$ and the previous value for the hidden state $\mathbf{h}_{t-1}$ but as modulated by another gate known as the reset gate:

$$\tilde{h}_t^j \;=\; tanh(W_h \mathbf{x}_t \;+\; U_h(\mathbf{r}_t \odot \mathbf{h}_{t-1}))^j$$

  where $\odot$ represents an element-wise multiplication. As you surely know already, $tanh(x)$ is another one of those famous activation functions for neural networks. In its appearance, it is very much like the logistic sigmoid except that it saturates out at -1 at the low end end +1 at the high end. As was the case earlier with similar entities, W_h and U_h are matrices of learnable parameters.

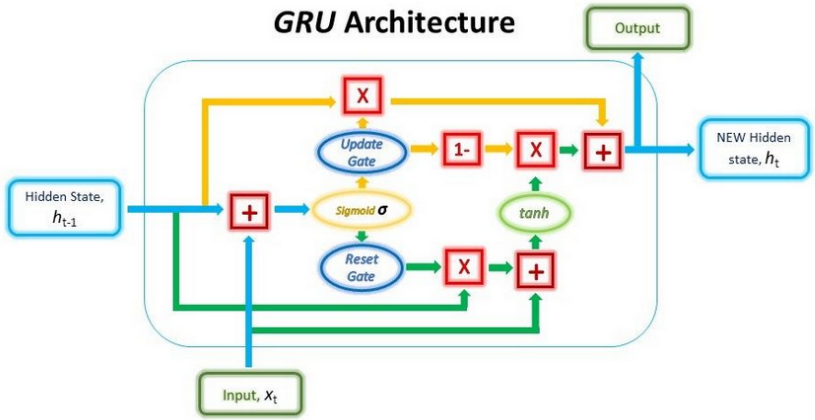
- About the reset gate $\mathbf{r}$ shown above, it is given by

$$r_t^j \;=\; \sigma(W_r \mathbf{x}_t \;+\; U_r \mathbf{h}_{t-1})^j$$

GRU – The Gate Semantics

- In the description of a GRU on the previous slides, notice how the mathematical forms shown force the *update* and the *reset* semantics on the values calculated as **z** and **r**. The fact that **z**'s role is that of updating the hidden state should be obvious from the first equation on Slide 42. The role of the **r** gate as that of resetting the hidden state should also be obvious.
- Shown below is a pictorial depiction of the network that incorporates all of the GRU related equations shown so far. This picture is from:

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



The GRU (contd.)

- The blog cited in the previous slide contains an interesting example on the modeling of sequential data: the example is based on the sequences of energy consumption data at several power distribution centers in the US. The goal is to create a "language model" for the sequences and then use the model to predict the energy consumption at the next time step.
- Using RNNs to create a language model is a big part of the modern neural-network based algorithms for automatic translation from one language to another. I believe the more recent versions of the Google Translate app are powered by such algorithms.

Outline

The GRUnet class in DLStudio

- Shown below is the `GRUnet` class in the inner class `TextClassification` of the DLStudio module. Except for the fact that the output in `forward()` is routed through a LogSoftmax activation, it is the same as what you'll find in Gabriel Loye's GitHub code for the example I mentioned on Slides 51 and 52.

```
class GRUnet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, n_layers, drop_prob=0.2):
        super(DLStudio.TextClassification.GRUnet, self).__init__()
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        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
```

Training with the GRUnet Class

- Shown below is the code for training the GRU based RNN. If there is anything remarkable at all about the code shown, it would be in the statements in lines (A), (B), and (C). Line (A) causes the hidden to be re-initialized to all zeros for each new review being processed. Then, the loop in lines (B) and (C) scans the review one word at a time as the hidden state for the review is continually updated. Finally, after a review has all been processed, we compare the output with the ground-truth sentiment for the review to calculate the loss.

```
def run_code_for_training_for_text_classification_with_gru(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)
    criterion = nn.NLLLoss()
    optimizer = optim.SGD(net.parameters(), lr=self.dl_studio.learning_rate, momentum=self.dl_studio.momentum)
    for epoch in range(self.dl_studio.epochs):
        print("")
        running_loss = 0.0
        start_time = time.clock()
        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)
            optimizer.zero_grad()
            hidden = net.init_hidden(1).to(self.dl_studio.device)                                                    ## (A)
            for k in range(review_tensor.shape[1]):                                                                  ## (B)
                output, hidden = net(torch.unsqueeze(torch.unsqueeze(review_tensor[0,k],0),0), hidden)     ## (C)
            loss = criterion(output, torch.argmax(sentiment, 1))
            running_loss += loss.item()
            loss.backward()
            optimizer.step()
            if i % 100 == 99:
                avg_loss = running_loss / float(99)
                current_time = time.clock()
                time_elapsed = current_time-start_time
                print("[epoch:%d  iter:%3d  elapsed_time: %d secs]     loss: %.3f" % (epoch+1,i+1, time_elapsed,avg_loss))
                FILE.write("%.3f\n" % avg_loss)
                FILE.flush()
                running_loss = 0.0
    self.save_model(net)
```

Testing with the GRUnet Class

- As with the training script shown on the previous slide, we reinitialize the hidden state to all zeros in line (A) for each new unseen review used for testing. Each review is scanned word by word in the loop in lines (B) and (C) where the one-hot vector for each new word is combined with the hidden state that represents all the previous words in the review.

```
def run_code_for_testing_text_classification_with_gru(self, net, hidden_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(1)                                                                    ## (A)
            for k in range(review_tensor.shape[1]):                                                        ## (B)
                output, hidden = net(torch.unsqueeze(torch.unsqueeze(review_tensor[0,k],0),0), hidden)     ## (C)
            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 is 0:
                negative_total += 1
            elif gt_idx is 1:
                positive_total += 1
            confusion_matrix[gt_idx,predicted_idx] += 1
    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)
```

How Come No Results with GRU?

- Even with the smallest of the datasets listed on Slide 13, it takes 10 times longer to train with the `GRUnet` than with `TEXTnetOrder2`. That is because of the size of the model created when using GRU. Shown below are the sizes of the GRU based models for the three different datasets listed on Slide 13:

dataset	vocab size	Size of GRU Based Model (in # of learnable params)
40-dataset	17,001	28,480,002
200-dataset	43,285	68,852,226
400-dataset	64,350	101,208,066

  This display nicely summarizes why one-hot vectors is not the way to go for the numerical representations for words in a text corpus if the end-goal is to classify variable-length text.

- As you increase the size of the dataset, the vocabulary size goes up, which makes the one-hot representations larger, and that, in turn, increases the size of the model, which creates a need for a still larger dataset. This is the catch-22 situation mentioned previously in this lecture that can only be remedied by using, say, word embeddings.