Recurrent Neural Networks for Text Classification and Data Prediction

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Tuesday 2nd April, 2024 07:10

Preamble

To TOC To HowTo

Recurrent neural networks (RNN) are neural networks with feedback. In such networks, each output of the neural network creates a context for the next input.

Such networks are important in applications in which each new data element must be "understood" in the context created by all previous data elements. At any given point in time, the feedback provided by the neural network summarizes all the previous data elements.

Text processing, natural language translation, and data prediction are three important applications for such networks.

Focusing first 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.

Preamble (contd.)

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.

The same thing happens in natural language translation: As a source-language sentence is being scanned word-by-word, how each word should be processed in order to yield a word in the target language depends on all the seen previously in the sentence.

Focusing now on data prediction, let's say you have a sequence of observations recorded at regular intervals. These could, for example, be the price of a stock share recorded every hour; the hourly recordings of electrical load at your local power utility company; the mean average temperature recorded on an annual basis; and so on. We want to use the past observations to predict the value of the next one. **An RNN allows us to do that because the feedback it provides at each time step is a summarization of all of the data seen up to that point.**

Preamble (contd.)

With regard to text classification, the goal of this lecture is to demonstrate how an RNN can be used to provide a context based on all previous words for the processing of each new word in a sequence of words.

And with regard to data prediction, the goal is to demonstrate how the feedback mechanism in an RNN can be used to give it the ability to predict the next value in a data sequence.

With regard to natural language translation, I'll take that up in the next lecture where I will introduce the notion of attention that is needed for that application.

In general, the feedback mechanism in an RNN creates chains of dependencies between the variables that are much longer than what you have seen so far in a typical neural network. As a result, the vanishing (or the exploding) gradient problem is even more acute for recurrent neural networks. **A very important goal of this lecture is to introduce you to the gating mechanisms that are used to get around those problems.**.

Preamble (contd.)

As far as gating is concerned, my focus will be on what is known as the **Gated Recurrent Unit (GRU)** — a specially designed RNN that has the ability to mitigate against vanishing and exploding gradients. To that end, I'll be introducing you to PyTorch's GRU network in my example on text processing for sentiment analysis. **This lecture also includes a separate section that talks about potential sources of confusion when using a GRU for the first time.** The original idea of GRU was presented by Cho et el. in the paper:

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

To enhance your understanding of how a GRU is actually implemented in code, I'll also be introducing you to my **pmGRU** for the example on data prediction. The name **pmGRU** stands for "poor man's GRU". The pmGRU network is based on the "Minimally Gated" version of a GRU presented by Heck and Salem in

https://arxiv.org/pdf/1701.03452.pdf

Preamble (contd.)

Before closing this Preamble, I should also mention that the text analysis example I'll be presenting in this lecture is based on what's known as Sentiment Analytics and I'll do so with the help of 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 positive-feedback comments and a file with 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.

Preamble – How to Learn from These Slides

Given the number of slides in this presentation, here's some help regarding how to wrap your head around it. The most important suggestion is that, at your first reading, you should focus on just the following three topics:

- Understanding the need to use RNNs for solving problems in machine learning and the unique challenges associated with the processing of text with neural networks. **These are explained on Slides 16 through 21, for a total of 5 slides.**

- Understanding thoroughly the basic functioning of a neural-network with feedback (RNN) as explained through the Sentiment Analysis network on Slides 32 through 38. **In particular you must understand how a sequence of words (in a sentence) is scanned one word at a time in the code lines labeled (B), (C), and (D) on Slide 37.** And how we examine the output of the neural network only after the entire sentence has been scanned. **This counts for a total of 6 slides.**

- Understanding the concept of gating in RNNs to mitigate what would otherwise be a severe problem of vanishing gradients. Vanishing gradients would be caused by long chains of dependencies in the computational graph as created by feedback. For this you must come to terms with the material on Slides 48 through 60, **for a total of 12 slides.**.

That makes for a total of just 23 slides you need to focus on at the beginning. Of the rest of the material, the most important is Section 9 entitled "Understanding the `torch.nn.GRU` API". Treat this section more as a reference that you may consult on as-needed basis.

Outline

1. Sentiment Analytics 9
2. A Sentiment Analysis Dataset 11
3. Challenges in Processing Text with Neural Networks 16
4. An RNN That Predicts the Ethnic Origin of a Last Name 22
5. Solving the Problem of Sentiment Analysis 31
6. Gating Mechanisms to Deal with the Problem of Vanishing Gradients in RNNs 47
7. Gated Recurrent Unit (GRU) 54
8. The GRU Based Classes in DLStudio 61
9. Understanding the `torch.nn.GRU` API 68
10. Data Prediction 89

Outline

1. Sentiment Analytics 9
2. A Sentiment Analysis Dataset 11
3. Challenges in Processing Text with Neural Networks 16
4. An RNN That Predicts the Ethnic Origin of a Last Name 22
5. Solving the Problem of Sentiment Analysis 31
6. Gating Mechanisms to Deal with the Problem of Vanishing Gradients in RNNs 47
7. Gated Recurrent Unit (GRU) 54
8. The GRU Based Classes in DLStudio 61
9. Understanding the `torch.nn.GRU` API 68
10. Data Prediction 89

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

1 Sentiment Analytics 9

2 A Sentiment Analysis Dataset 11

3 Challenges in Processing Text with Neural Networks 16

4 An RNN That Predicts the Ethnic Origin of a Last Name 22

5 Solving the Problem of Sentiment Analysis 31

6 Gating Mechanisms to Deal with the Problem of Vanishing Gradients in RNNs 47

7 Gated Recurrent Unit (GRU) 54

8 The GRU Based Classes in DLStudio 61

9 Understanding the `torch.nn.GRU` API 68

10 Data Prediction 89

The Amazon User Feedback Dataset

- The user feedback shown below is from the 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 there are 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. Here is a portion of that file:

```
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
```

A Sentiment Analysis Dataset

- DLStudio comes with the following text dataset archive:

```
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 three 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

1 Sentiment Analytics 9

2 A Sentiment Analysis Dataset 11

3 **Challenges in Processing Text with Neural Networks 16**

4 An RNN That Predicts the Ethnic Origin of a Last Name 22

5 Solving the Problem of Sentiment Analysis 31

6 Gating Mechanisms to Deal with the Problem of Vanishing Gradients in RNNs 47

7 Gated Recurrent Unit (GRU) 54

8 The GRU Based Classes in DLStudio 61

9 Understanding the `torch.nn.GRU` API 68

10 Data Prediction 89

Challenges in Processing Text with Neural Networks

- Neural networks obviously have a natural fit with images because both are numeric.
- But that's not the case with text and languages. Both represent purely symbolic domains and any attempt at representing the symbols with numbers just so that we can use neural networks seems highly contrived.
- Contrived or not, the fact today is that with appropriate numeric representations it is possible to construct powerful deep learning based solutions to problems in text and language processing.
- Traditionally, for the purpose of neural-network based processing, words have been represented by one-hot vectors, the size of the vectors being equal to the size of the vocabulary.

Processing Text with Neural Networks (contd.)

- Unfortunately, with one-hot vectors for representing the words, you run into the following vicious circle:

  The larger the vocabulary, the larger the one-hot vectors, which translates into networks with a significantly larger number of learnable parameters.

  And that, in turn, creates a need for larger training datasets.

  Unfortunately, as you saw with the sentiment analysis dataset in the previous section, increasing the size of the training dataset increases the size of the vocabulary that results in still larger one-hot vectors, *which jacks up the number of learnable parameters, and so on.*

  You get the picture.

- One of the goals of this lecture is to make you aware of this serious shortcoming of using one-hot vector based representation for words.

Processing Text with Neural Networks (contd.)

- **Modern system for processing text and language are all based on using what are called word embeddings.** Word embeddings are fixed-sized numerical representations for words that are learned on the basis of the similarity of word contexts. The most famous of these are the `word2vec` and `fastText`. I'll have more to say about these embeddings in the next lecture.

- The other major source of difficulty in processing text with neural networks is the variable length of the inputs to the networks. In sentiment analysis, length of the user feedback varies considerably from user to user. In automatic language translation, there really are no constraints on the length of a sentence. [**I used to be an avid reader of Joyce Carol Oats in my younger days. I recall that once I read a sentence by her that occupied one full page in one of her books. The amazing thing is that the unusual length of the sentence did not make it any harder to parse or to understand its meaning.**]

- One effective way to deal with variable length input is to use neural networks with feedback — the **Recurrent Neural Networks**.

Processing Text with Neural Networks (contd.)

- However, as you will see in this lecture, RNNs come with their own challenges: the long chains of dependencies created by feedback only exacerbate the problem of vanishing gradients. This has led to the development of gated RNNs, of which the two best-known examples are GRU (Gated Recurrent Unit) and LSTM (Long Short-Term Memory).
- Another consequence of variable-length inputs is that now it becomes a bit more difficult to do batching. As you'll recall, batching contributes significantly to the smooth convergence of SGD for the case of images.
- Other challenges that arise with text processing are more specific to the applications in which a sequence at the input goes into a sequence at the output. Automatic translation and automatic question-answer frameworks represent such applications. Such applications are generally referred to as sequence-to-sequence (seq2seq) learning.

Processing Text with Neural Networks (contd.)

- In seq2seq learning, an RNN may be used to encode a source sequence into a fixed-sized vector (that is typically referred to as a "hidden vector") and another RNN to decode the hidden vector into the output sequence. One of the challenges faced when using this approach is the "capacity" of a fixed-size hidden vector to encode input sequences that may be arbitrarily long.
- More recent solutions to the seq2seq learning problem do away with recurrence altogether and use what is known as the attention mechanism for solving the problem.

Outline

1 Sentiment Analytics 9

2 A Sentiment Analysis Dataset 11

3 Challenges in Processing Text with Neural Networks 16

4 **An RNN That Predicts the Ethnic Origin of a Last Name 22**

5 Solving the Problem of Sentiment Analysis 31

6 Gating Mechanisms to Deal with the Problem of Vanishing Gradients in RNNs 47

7 Gated Recurrent Unit (GRU) 54

8 The GRU Based Classes in DLStudio 61

9 Understanding the `torch.nn.GRU` API 68

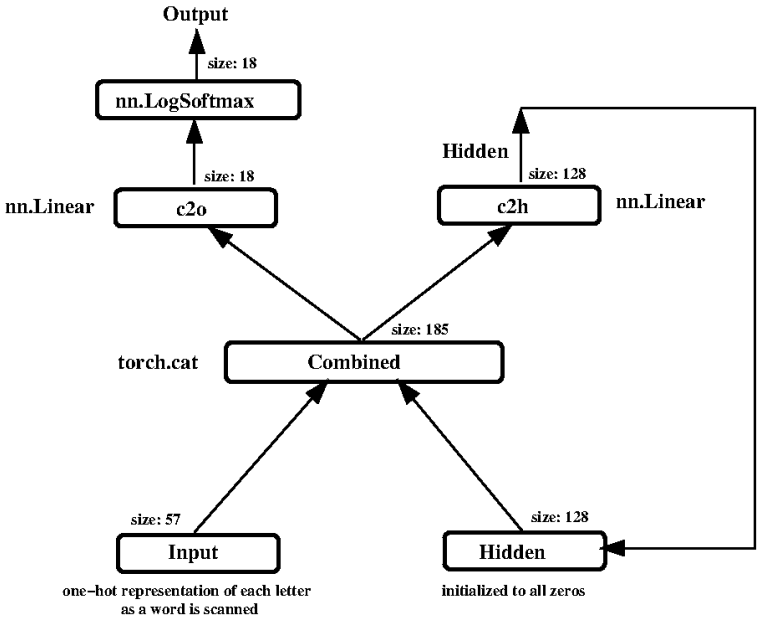
10 Data Prediction 89

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 cutest examples of how much fun you can have with a neural network that incorporates feedback. Here is a link to this example:

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

  The goal in this example 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 23.
- The size of the input, `input_size`, is dictated by the number of printable ascii characters (without including the digits, etc.). It is the size of the one-hot vector representation for each character.
- The size of the hidden state is 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 (See Slide 27)    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 four 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 the 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 `torch.max(output,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 = torch.max( output, 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():
        ## Add to the learnable weights their gradients times the learning rate:
        p.data.add_(p.grad.data, -learning_rate)
    return output, loss.item()
```

The train() Function (contd.)

- As mentioned in my Week 7 lecture, `nn.NLLLoss()` stands for "Negative Log Likelihood Loss". This is the loss function to use if the output of your network is produced by `nn.LogSoftmax` activation.
- `nn.NLLLoss()` expects at its input log-probabilities for the different classes — such as those produced by the `nn.LogSoftmax` activation.
- As mentioned on Slides 28 and 29 of my Week 7 lecture on multi-object detection, the combined effect of `nn.LogSoftmax` activation in the output of a network and `nn.NLLLoss()` for loss is exactly the same as using `nn.CrossEntropyLoss` for loss.
- For iterative training, we repeated call `train()` a large number of times over the training dataset as shown on the previous slide.
- The variable `line_tensor` on the previous slide 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

1. Sentiment Analytics 9
2. A Sentiment Analysis Dataset 11
3. Challenges in Processing Text with Neural Networks 16
4. An RNN That Predicts the Ethnic Origin of a Last Name 22
5. **Solving the Problem of Sentiment Analysis 31**
6. Gating Mechanisms to Deal with the Problem of Vanishing Gradients in RNNs 47
7. Gated Recurrent Unit (GRU) 54
8. The GRU Based Classes in DLStudio 61
9. Understanding the `torch.nn.GRU` API 68
10. Data Prediction 89

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. To that end, as shown by the function whose definition starts at line (A) on Slide 34, 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.

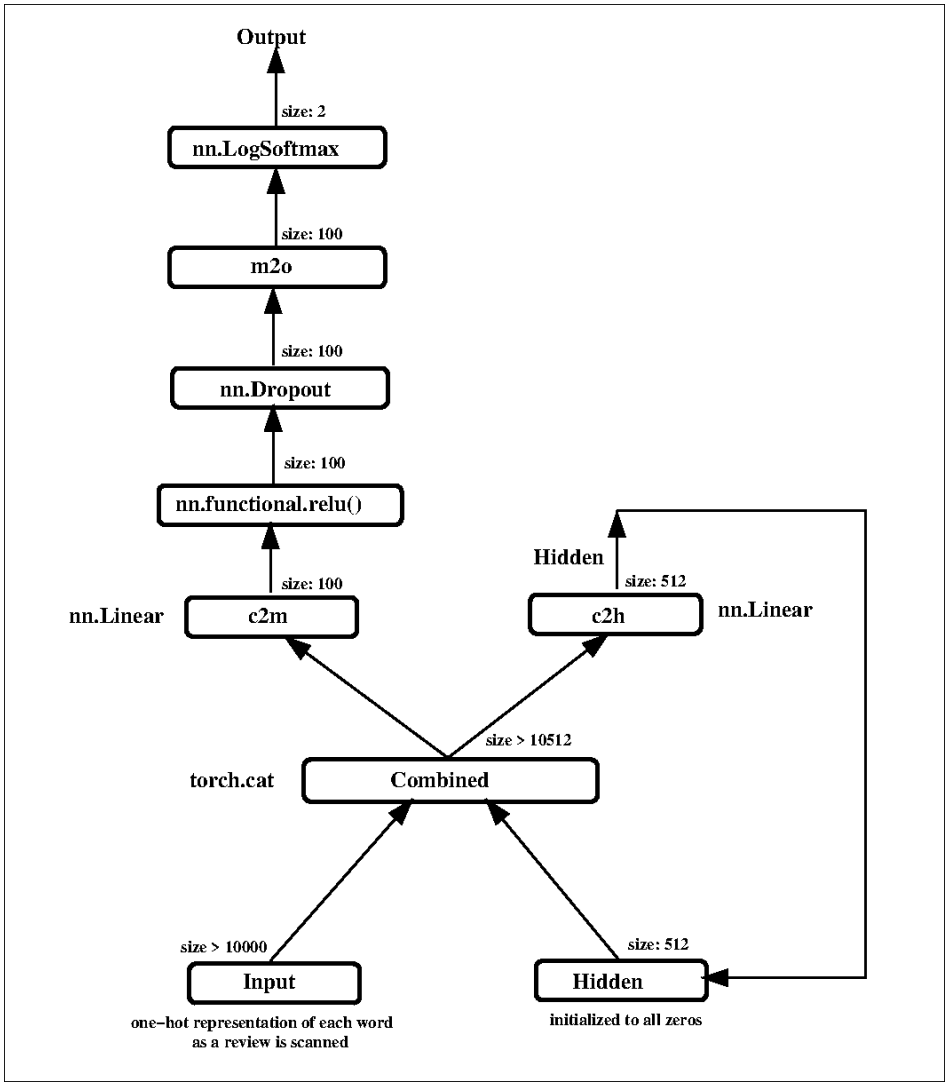
```
class TEXTnet(nn.Module):

    def __init__(self, input_size, hidden_size, output_size):
        super(DLStudio.TextClassification.TEXTnet, self).__init__()
        self.input_size = input_size           # size of the one-hot vector for each word
        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(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()
            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 previously in the review.

```
def run_code_for_testing_with_TEXTnet(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 15:

```
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.py` in the Examples directory of DLStudio.

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 15 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 13 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 43.

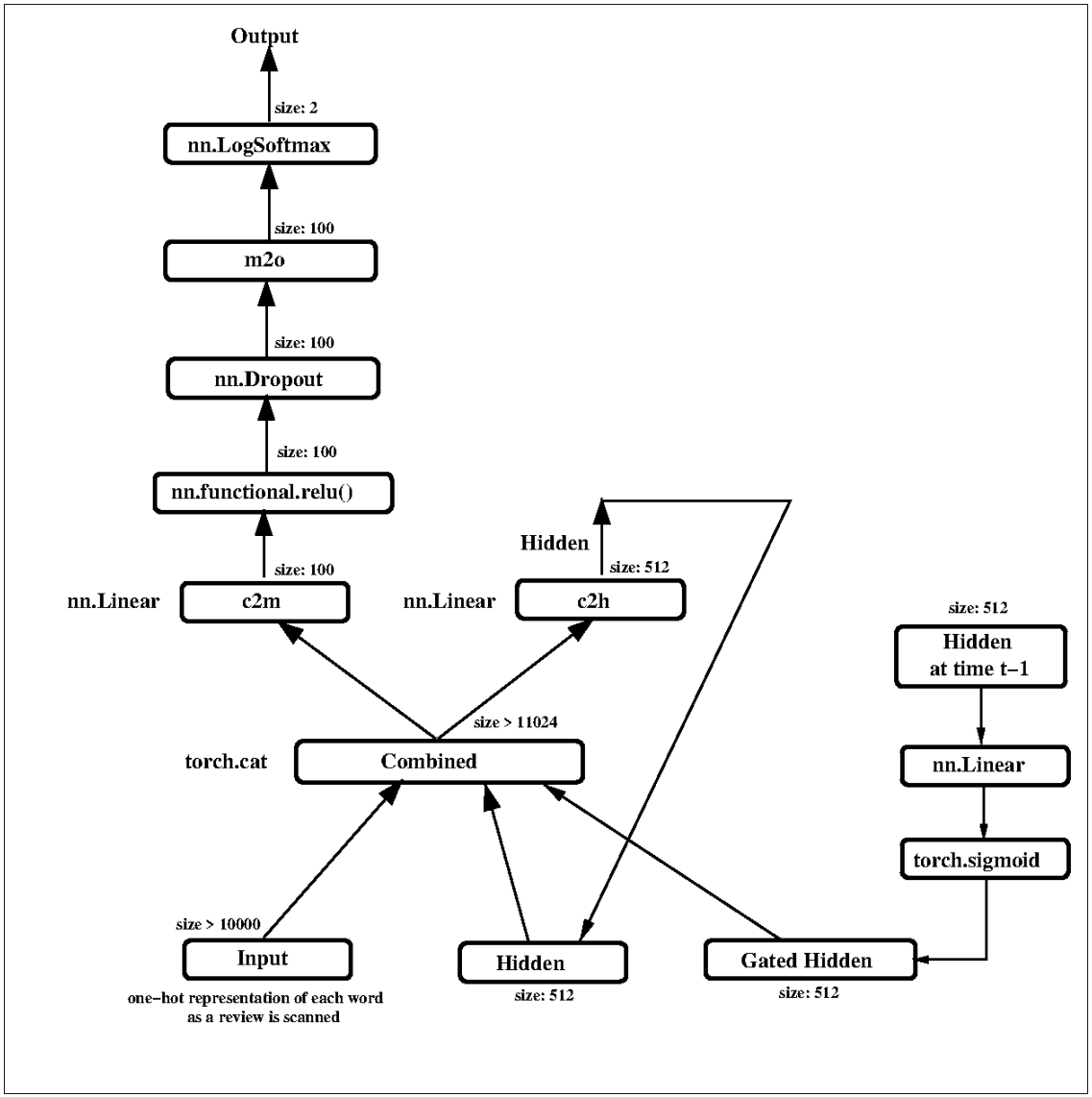
The TEXTnetOrder2 Network

```
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     # size of the one-hot vec for each word
        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(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(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, here is the confusion matrix 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.

- These results are still poor. Therefore, we have no choice but to dig into the material that follows in this lecture.

Outline

1. Sentiment Analytics 9
2. A Sentiment Analysis Dataset 11
3. Challenges in Processing Text with Neural Networks 16
4. An RNN That Predicts the Ethnic Origin of a Last Name 22
5. Solving the Problem of Sentiment Analysis 31
6. **Gating Mechanisms to Deal with the Problem of Vanishing Gradients in RNNs 47**
7. Gated Recurrent Unit (GRU) 54
8. The GRU Based Classes in DLStudio 61
9. Understanding the `torch.nn.GRU` API 68
10. Data Prediction 89

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 of more recent origin.

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 a network one element at a time by

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

- Assuming that t is the current time, **any** joint distribution over the obervations made up to the current time can be decomposed as:

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

- 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 (`nn.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 $\phi()$.

- In general, the input sequences that admit such $\mathbf{h}_t$'s lend themselves well to processing by RNNs. More particularly, such sequences can be processed by the gating mechanisms such as GRU and LSTM **to mitigate the vanishing gradients**. The variable $\mathbf{h}_t$ is referred to as the hidden state that characterizes the sequence up to the time step t.

- The question we next address is: **Is it possible to write down a model for the time evolution of the hidden state?** If we could conceive of such a model, we could try to think of a way to protect $\mathbf{h}_t$ from the ravages of vanishing gradients. We want to protect $\mathbf{h}_t$ because its final value will be our fixed-size representation of the entire input sequence regardless of its length.

The Hidden State and its Evolution (contd.)

- Regarding a possible model for the time evolution of the hidden state $\mathbf{h}_t$, we know that, in general, it must depend on (1) the previous value $\mathbf{h}_{t-1}$, which "summarizes" the previously seen input sequence elements, $(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{t-1})$, and (2) the new input element $\mathbf{x}_t$. In general, this dependence may be nonlinear. And, for the time evolution model of $\mathbf{h}_t$ to be useful, we must also ensure that it is learnable. Given these considerations, we write the following form for the time evolution model of the hidden state:

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

  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 is 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.)

- Now that we have a plausible looking and learnable model for the time evolution of the hidden state, our next mission is to figure out how this model can be learned **despite the presence of vanishing gradients in a RNN**.
- **In the next few slides, we will see how a GRU lets us do exactly that.**
- While a GRU (or an LSTM) is a tried and true approach that allows an RNN to learn the time evolution model for the hidden state in the presence of vanishing gradients, **what I showed in the network of Slide 43 is not an entirely laughable attempt at protecting the evolution of the hidden state**. In addition to the current value for the hidden state, that network has the freedom to directly access the previous value for the state to the extent allowed by the sigmoid. At the least, it provides a short circuit for refreshing the current state with its value at the previous time step. That can be important in the presence of vanishing gradients.

Outline

1. Sentiment Analytics 9
2. A Sentiment Analysis Dataset 11
3. Challenges in Processing Text with Neural Networks 16
4. An RNN That Predicts the Ethnic Origin of a Last Name 22
5. Solving the Problem of Sentiment Analysis 31
6. Gating Mechanisms to Deal with the Problem of Vanishing Gradients in RNNs 47
7. **Gated Recurrent Unit (GRU) 54**
8. The GRU Based Classes in DLStudio 61
9. Understanding the `torch.nn.GRU` API 68
10. Data Prediction 89

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.

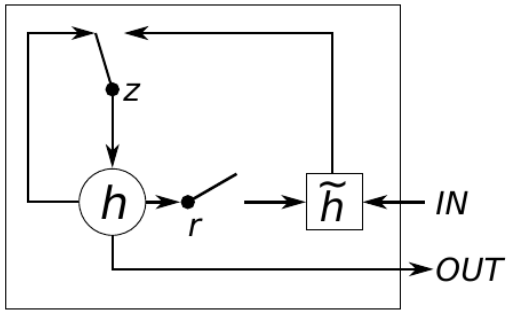


Figure:

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 $\mathbf{x}$ and $\mathbf{h}$ 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 jth 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 the sigmoid function (`nn.Sigmoid`) 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.

[**Showing the relationships element-wise on this slide is intentional to emphasize that the gating action is specific to each element of the hidden state.**]

Why Call it a Gate?

- With regard to the last equation shown on the previous slide, note the implications of the sigmoid nonlinearity with regard to how each component of $\mathbf{h}$ is updated.
- Just for illustration, assume that the argument to the $\sigma()$ is such that it is either sufficiently negative or sufficiently positive so that the output of $\sigma()$ is either 0 or 1.
- When $\sigma()$ returns 0, for that component of $\mathbf{h}$, 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* $\mathbf{h}$ *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 know already, $tanh(x)$ is another one of those famous activation functions for neural networks. In its appearance, it is very much like the sigmoid except that it saturates out at -1 at the low end, in addition to saturing out at $+1$ 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 56. 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 figure was produced by Constantine Roros of Purdue RVL. [**Constantine has created a new type of GRU, maskGRU, that is particularly effective in tracking small objects, like a volleyball, in a highly dynamic scene with large objects in motion (such as the players on a volleyball court).**]

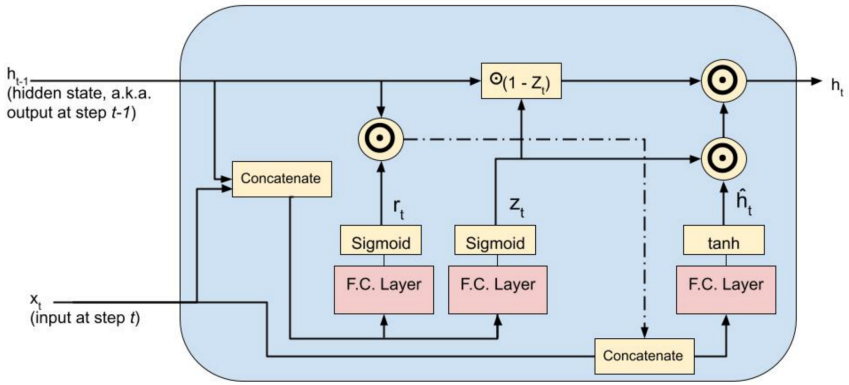


Figure: The network is a composite depiction created from the GRU update and reset equations presented.

GRU – Summary Presentation of the Equations

- For easy reference, in what you see below I have pulled together from Slides 56 and 58 the main equations of a GRU:

$$
\begin{aligned}
\mathbf{z}_t &= \sigma(W_z \mathbf{x}_t \;+\; U_z \mathbf{h}_{t-1}) \\
\mathbf{r}_t &= \sigma(W_r \mathbf{x}_t \;+\; U_r \mathbf{h}_{t-1}) \\
\tilde{\mathbf{h}}_t &= tanh(W_h \mathbf{x}_t \;+\; U_h(\mathbf{r}_t \odot \mathbf{h}_{t-1})) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} \;+\; \mathbf{z}_t \odot \tilde{\mathbf{h}}_t
\end{aligned}
$$

- The equations shown on Slides 56 and 58 were in terms of the individual elements of the vectors. Here they are shown as full vectors. [**Ask yourself if we could have switched the roles of σ and $tanh()$. The answer is: No!. But why?**]

- $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ are the input vector and the hidden state, respectively, at the iteration index t when the input $\mathbf{x}_t$ becomes available. The output is the hidden state $\mathbf{h}_t$ to be used at the next value of the iteration index. σ is the sigmoid function.

- z_t and r_t are the update and the reset gates at the iteration index t. Finally, $\tilde{h}_t$ is is the Candidate Hidden State at time t.

Outline

1 Sentiment Analytics 9

2 A Sentiment Analysis Dataset 11

3 Challenges in Processing Text with Neural Networks 16

4 An RNN That Predicts the Ethnic Origin of a Last Name 22

5 Solving the Problem of Sentiment Analysis 31

6 Gating Mechanisms to Deal with the Problem of Vanishing Gradients in RNNs 47

7 Gated Recurrent Unit (GRU) 54

8 **The GRU Based Classes in DLStudio** **61**

9 Understanding the `torch.nn.GRU` API 68

10 Data Prediction 89

GRU Based Classes in DLStudio

- DLStudio contains the following classes that use GRUs for RNN based modeling of the reviews for sentiment analysis:

  `GRUnet`

  `GRUnetWithEmbeddings`

- The first class is in the `TextClassification` section of DLStudio and the second in the `TextClassificationWithEmbeddings` section. Apart from that, the two GRU based classes possess identical definitions. The reason that they exist in two different sections of DLStudio are pedagogical.

- The first class is meant to be used with one-hot vector representations for the words in a review and the second with fixed-sized word embeddings that I discuss in my Week 13 lecture.

- In the rest of this section, I'll first talk about the `GRUnet` class and then show results with both this class and with the embeddings version of the class.

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 code at `https://blog.floydhub.com/gru-with-pytorch/` . Note that `input_size` is the size of the one-axis tensor representation for each word.

```
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 at all remarkable 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 GRUnet?

- Even with the smallest of the datasets listed on Slide 15, 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 15:

dataset	vocab size	Size of the GRU Based Model (number 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.

Results with GRUnetWithEmbeddings

- This network uses the word2vec embeddings presented in my Week 13 lecture. With the constructor options set as follows:

```
input_size      =   300          # this is the size of the one-axis tensor for each word

hidden_size     =   200          # size of the hidden state

num_layers      =   2            # this creates a stack of two GRU layers

dataset_used    =   dataset-40

learning_rate   =   1e-5

epochs          =   5
```

I get the following accuracy numbers for classification:

```
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%
```

Outline

1 Sentiment Analytics 9

2 A Sentiment Analysis Dataset 11

3 Challenges in Processing Text with Neural Networks 16

4 An RNN That Predicts the Ethnic Origin of a Last Name 22

5 Solving the Problem of Sentiment Analysis 31

6 Gating Mechanisms to Deal with the Problem of Vanishing Gradients in RNNs 47

7 Gated Recurrent Unit (GRU) 54

8 The GRU Based Classes in DLStudio 61

9 **Understanding the torch.nn.GRU API** **68**

10 Data Prediction 89

What Makes nn.GRU Frustrating to Learn

- Of all the different neural structures you can construct with PyTorch, the nn.GRU has got to be the most frustrating one to learn. This and the next slide enumerate the reasons for that.

- The main source of the frustration is that, depending on the constructor options chosen and the shape of the input, an nn.GRU can be made to operate in one of several different modes and that can be highly befuddling to a beginning programmer:

  - You can obviously use a GRU as a regular RNN that steps through each element of a sequence in a loop in your own code — just as the theory of RNN mentions.

  - But a GRU can also ingest an entire sequence all at once, in which case it steps through the sequence internally. On account of the rather poor documentation at the official website, this mode can be confusing to a novice programmer.

What Makes nn.GRU Frustrating (contd.)

- When a GRU ingests an entire sequence all at once, the shape of the output looks nothing like what you get if you stepped through a sequence on your own.
- A GRU can not only scan a sequence left-to-right, but also right-to-left, which only adds to the confusion for a beginning programmer.
- You can also end up with a stack of GRUs working together, which further enhances the confusion level for a novice.
- It is highly confusing for a beginning programmer that the shape of the sequence *elements* of what a GRU produces at the output is the same as the shape of the hidden state.
- Some of the constructor parameters for nn.GRU are relevant only in some usage modes for a GRU but you see them being specified in the code on the web even when those modes are not actually used in the rest of the implementation. Very confusing!

- The rest of this section is an attempt at clearing up these confusing aspects of nn.GRU.

Understanding the nn.GRU API

- In the GRU examples I have presented so far, we have stepped through an input sequence one word at a time. This I did primarily to connect my examples with the basic operation of an RNN presented at the beginning of this lecture. **Note, however, that an instance of nn.GRU is happy to process en entire sequence for you all at once.** That's because an instance of nn.GRU is a full-blown RNN unto itself.

- Obviously, whether you feed an entire sequence into a GRU all at once or do so one word at a time, your final answer for the output and the hidden state will be the same.

- When you construct an instance of nn.GRU (see the call to the constructor of nn.GRU in Slide 63), its only *required* parameters that you must supply values for are the input_size and hidden_size, the former standing for the size of the one-axis tensor for each word in your sequence and latter for the size for the hidden state in your RNN. It is best to think of these two required params as the sizes of two numerical vectors.

Understanding the nn.GRU API (contd.)

- **The bullet at the bottom of the previous slide means is that the GRU instance itself does not care whether you supply it with an entire sequence in one go or whether you use it to step through a sequence of words, one word at a time.**

- As to which behavior you elicit from the instance of nn.GRU depends entirely on the shape of the input you feed into the instance. In the following loop shown in Lines (B) and (C) on Slide 64, the call to torch.unsqueeze() will cause the dimensionality of the Axis 0 of the input tensor to be exactly 1. That works because in that example we want the GRU to step through a sequence of words one word at a time. That fact is signified by shape value of 1 for Axis 0 of the input tensor.

```
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)
```

Understanding the nn.GRU API (contd.)

- In general, the input to an instance of nn.GRU must be formatted as [This specification is obviously different from what you need for the constructor]:

```
( seqence_length,   batch_size,   input_size )
```

  where sequence_length is the number of words in the review you are feeding into the GRU. If this value is 1, then you are using the GRU to step through the review one word at a time. But, in keeping with what was said earlier, this value can also be the number of all the words in a review.

- The above bullet also says is that if I were to invoke shape on the tensor being fed into a GRU, it must return three values, the first of which is the number of elements in the sequence you are feeding into the GRU.

- The second parameter mentioned above, batch_size, is greater than 1 only if you are using batches in your training loop. More on batching on the next slide.

Understanding the nn.GRU API (contd.)

- Yes, an RNN like a GRU can do batch based processing of your input sequences. [**As to how that works for variable length sequences depends on you the programmer. You may have to resort to ploys such as sorting each input batch by length in decreasing order. Or you may yourself make all the sequences in a batch to be of the same length by using a filler character at the end of the shorter sequences. The filler character is often a special character that is referred to as the EOS ("End of Sentence") character. With batch based processing, you must also pay attention to** `batch_first` **constructor parameter of the class** `nn.GRU`**. Its default value is False. However, if you set it to True, that would change how the GRU would expect its input to be formatted from** `(sequence_length, batch_size, input_size)` **that was shown earlier to** `(batch_size, sequence_length, input_size)`**.**]

- The third parameter, `input_size`, in the shape of the input tensor to `nn.GRU`, as shown on the previous slide, is the size of the one-axis tensor for *each word* in the input sequence. **It must agree with the value you supplied for the first constructor parameter when you constructed your instance of** `nn.GRU` Recall the call to the constructor of `nn.GRU` on Slide 63.

- That brings us to the shape of the tensor to be used for the hidden state of the RNN — as presented on the next slide.

Understanding the nn.GRU API (contd.)

- Whatever tensor you specify for the hidden state must have its shape formatted as follows:

```
( num_layers,   batch_size,   hidden_size )
```

  where hidden_size must have the same value as for the second required parameter in the nn.GRU constructor call. For obvious reasons, the value for batch_size must be the same as in the second shape parameter for the input tensor.

- The new thing in the format specification for the hidden state is num_layers. Its default value is 1 and if you want its value to be greater than 1, you must do so in the constructor call to nn.GRU.

- **Setting num_layers to a value greater than 1 allows you to create stacked GRUs in which the output of one GRU is fed as input into the next GRU.**

Understanding the nn.GRU API (contd.)

- When num_layers is greater than 1, you will need to initialize the hidden state in each GRU layer separately. That's the reason for num_layers being a part of the shape specification of the hidden state. Additionally, when using num_layers > 1, as to how much of the output of one GRU is passed on to the input of the next is controlled by the parameter dropout in the constructor call for nn.GRU.
- **IN SUMMARY**, a GRU always expects to see a 3-Axis tensor for its input. It also expects a 3-Axis tensor for the initialization of the hidden state. As you would expect, the three axes carry very different meanings in the two cases.
- **In our work, apart from the example presented at the end of this section, I'll always be using a batch_size of 1 for variable-length sequence data. So the middle value in the shape of the 3-Axis tensors for both the input and the hidden will always be equal to 1 in our case.**

Understanding the nn.GRU API (contd.)

- In the rest of this section, I'll present the same example that's on the doc page for nn.GRU but with my annotations to highlight some key aspects of how a GRU operates.

- The input to the GRU in this example is specified by the tensor shown below:

```
##                          sequence length       batch_size        input_size
input  =  torch.randn(          5,                    3,                10      )
```

- The input to the GRU as shown above implies that it is a sequence of 5 elements, that I am using a batch size of 3, and that each element of each sequence in my batch is a tensor of 10 values. Since the sequence length exceeds 1, in this case I'll be asking the GRU to internally step through the sequence one step at a time. From the outside, it would seem like it was processing the input sequence all at once.

Understanding the nn.GRU API (contd.)

- For the example that follows, I'll specify the initial value for the hidden state through the following declaration:

```
##                                num_layers    batch_size     hidden_size
hidden_initial  =  torch.randn(      2,            3,            20      )
```

- This shape for the tensor supplied for the initial value of the hidden state of the RNN is predicated on us having specified `num_layers=2` when the `nn.GRU` instance was constructed. Also note that the size of the dimensionality of the third Axis of the tensor shown above must be equal to the second parameter value in the constructor call for `nn.GRU`.

- As you can see, the shape of the tensor you supply for the initial hidden state is fully constrained by how you called the `nn.GRU` constructor and the shape of what you supplied for the input to the GRU.

Understanding the nn.GRU API (contd.)

- Despite all of the discussion so far in this section, you may yet be surprised by the following aspects in the example script that follows:

  1. The first thing that will surprise you is that while the size of each element of the input sequence is 10, the size of each corresponding element of the output will be 20, which is the size of the initial hidden. So if you want your output to look like the input, you will have to run it through a Linear layer.

  2. The second thing that will surprise you is that when you run the entire sequence through the GRU in one fell swoop, you will get the entire time evolution of the *hidden* as the value for the *output*. However, for the *hidden* that is returned by the GRU, you will only get the final value of the *hidden*. What that means is that both the *output* and the *hidden* carry the same semantics at the output of the GRU, except that the former is the time-evolution whose final value is the latter.

  3. And the third thing that will surprise you is that the final value of the output for each of the three input sequences in a batch is the same as the final values for the hidden.

Understanding the nn.GRU API (contd.)

- In the example code starting with Slide 85, in Line (A) we call the GRU constructor and specify that each element of the input sequence will be a tensor of 10 values and that the hidden state for the sequence will be a tensor with 20 values. I have also specified the num_layers parameter in Line (A) and set it to 2. As mentioned earlier in this section, that will give me a stacked GRU, meaning an RNN that consists of two GRU layers, with the output of one GRU becoming the input to the next GRU.
- Line (B) of the code specifies the input tensor. As mentioned earlier, the shape of the input tensor plays a critical role in the behavior of the GRU instance created.
- In the input provided in line (B), the dimensionality of the first axis (Axis 0) is 5. That means that our intention is to supply a sequence of length 5 as the input. So we are asking the GRU to process an entire sequence all at once, implying that GRU will step through the sequence internally.

Understanding the nn.GRU API (contd.)

- The input declaration in Line (B) also sets the batch size to 3 besides stating that each element of the input will be a tensor of 10 values. Note that this 10-values specification for each element of the input sequence is the same as in the constructor call in Line (A).
- The input generated by the statement in Line (B) is shown in the commented out section after Line (D). My annotations in the print-out of the input should make it easier to see the data corresponding to each element of the input sequence for each member of the batch created in Line (B).
- The `saved_input` statement in Line (C) is to save a copy of the input so that I can use it later for showing that externally stepping through a sequence or letting a GRU do it internally produces the same final answer.
- Line (E) specifies the initial values to use for the hidden state.

Understanding the nn.GRU API (contd.)

- The dimensionality of its first axis (Axis 0) in Line (E) must be the same as the value of the num_layers parameter in the constructor call in Line (A). In Line (F), we again put away the initial hidden state so that I can use it later for manually stepping through the input sequence.
- Also, the batch_size specified for the initial hidden state in Line (E) must match the value for the same parameter in input tensor supplied through the statement in Line (B). And, the hidden_size in Line (E) must match the same in the constructor call in Line (A). The point to note here is that the shape of the values supplied for the hidden state are completely constrained by how your constructor call and your input tensor.
- My annotations in the commented-out section after Line (G) should help you see the purpose of the different parts of the tensor for the initial hidden state.

Understanding the nn.GRU API (contd.)

- Finally, in Line (H), we call on the GRU instance to do its thing. The output tensor produced by the GRU is shown in the commented-out section that follows Line (I).
- The annotations I have inserted in the commented-out section for output print-out after Line (I) should help you see that you are looking at is the entire time evolution of the output values for each member of the input batch.
- The commented-out section after Line (J) shows the final hidden for each of the two GRU's in our GRU stack.
- The rest of the script is devoted to showing that what the GRU produces by internally stepping through the input data is the same as what you would get if did the stepping in your code.
- In Lines (K) and (L) we recover the values we had used previously for the input and the hidden tensors.

Understanding the nn.GRU API (contd.)

- In Line (M), we specify the shape of the tensor we expect for the output values produced by the GRU.
- Subsequently, we step through the input sequence in our own loop in Lines (N) and (O).
- You will notice in the commented-out section that follows Line (P), the output values produced by the GRU are exactly the same as shown earlier in the section after Line (I). And the same is true for the final values for the hidden state shown after Line (Q). These are exactly the same as those shown in the print-out that follows line (J).
- In general, though, when you are stepping through a sequence on your own as in the loop in Lines (N) and (O), you are likely to retain only the final values for the output, as opposed to retaining the entire time-evolution of the output values.

Understanding the nn.GRU API (contd.)

```
## test_gru_api.py
import torch
import torch.nn as nn
##  For the constructor of GRU, the parameters are
##
##            input_size    hidden_size        num_layers
rnn = nn.GRU( 10,              20,                2      )                                                    ## (A)

##  for the input to the GRU, the parameters are
##                    sequence length       batch_size     input_size
input  =  torch.randn(      5,                  3,             10     )                                       ## (B)
saved_input = input.clone()                                                                                   ## (C)
print(input)                                                                                                  ## (D)
                ##           FIRST ELEMENT OF INPUT SEQUENCE:
                ## tensor([[[-2.7490e-01,  1.5572e-01, -4.5562e-01, -9.0869e-01, -1.3559e+00,   1st ele of batch
                ##           -1.1187e-01, -2.4815e-02,  1.2941e+00,  1.5490e+00,  1.6310e+00],
                ##          [-1.2259e+00, -6.7998e-01, -7.9810e-01,  1.3871e+00, -2.0728e-01,   2nd ele of batch
                ##           -2.5348e-01, -1.2607e+00,  1.5554e+00,  2.3675e-01,  1.1248e+00],
                ##          [ 1.1924e+00,  1.6073e+00, -1.4381e+00,  7.3911e-01,  2.4060e-01,   3rd ele of batch
                ##           -1.6051e-01, -2.6560e-01,  1.3879e+00, -9.0451e-01, -2.4080e-02]],
                ##
                ##           SECOND ELEMENT OF INPUT SEQUENCE:
                ##         [[-6.3043e-01,  6.1073e-01, -1.3459e+00,  8.2578e-01,  5.1279e-01,   1st ele of batch
                ##           -6.1258e-02, -1.3280e+00, -4.0958e-01, -1.0371e+00, -6.2618e-01],
                ##          [-7.6464e-02,  5.8006e-01,  1.9136e+00,  1.8935e-01, -4.6239e-01,   2nd ele of batch
                ##           -1.0933e+00,  6.8900e-01,  1.8229e+00,  3.2627e-01,  1.0502e-01],
                ##          [ 5.1518e-01, -9.5116e-01, -6.5869e-01, -2.0188e+00,  9.8163e-01,   3rd ele of batch
                ##           -4.0150e-01,  1.5605e+00, -1.2984e-01, -1.8764e+00,  5.1152e-01]],
                ##
                ##           THIRD ELEMENT OF INPUT SEQUENCE:
                ##         [[-1.8256e+00,  3.5896e-01, -8.0539e-01, -1.4091e+00, -1.9283e+00,   1st ele of batch
                ##            2.5797e-01,  2.8547e+00, -1.2005e+00, -9.1751e-02,  4.8838e-01],
                ##          [ 3.4067e-01, -2.4257e+00,  5.2876e-01,  1.6312e-01, -7.4958e-01,   2nd ele of batch
                ##            9.3663e-01,  1.6461e+00, -1.2853e+00, -1.8464e+00, -1.0850e+00],
                ##          [ 1.0250e+00,  2.5618e-01,  2.5213e+00,  1.8072e+00,  1.7868e+00,   3rd ele of batch
                ##            6.6243e-01, -6.9879e-01, -1.3618e+00,  4.6247e-01, -9.8518e-01]],
                ##
                ##           FOURTH ELEMENT OF INPUT SEQUENCE:
                ##         [[ 6.9661e-02,  3.3826e-03,  6.4682e-02, -1.7257e+00, -2.3697e-01,   1st ele of batch
                ##           -9.7295e-03,  1.1260e+00, -2.2301e-02,  6.4132e-01, -1.5874e-01],
                ##          [-7.9617e-01, -5.6042e-01,  1.0132e+00, -2.1136e+00,  6.8181e-01,   2nd ele of batch
                ##           -8.6248e-01,  6.6681e-01,  5.7546e-01, -5.6160e-01, -3.3636e-01],
                ##          [-6.4267e-01,  2.4836e-03,  1.8996e-01,  5.1884e-01,  1.1684e+00,   3rd ele of batch
                ##           -2.8622e-01,  4.3121e-01, -2.9535e-01, -3.5768e-01, -3.1172e-01]],
                ##
                ##           FIFTH ELEMENT OF INPUT SEQUENCE:
                ##         [[ 8.3951e-01, -6.5182e-01, -2.8393e-01,  4.3051e-01,  2.0633e-01,   1st ele of batch
                ##            7.6597e-01, -1.1447e+00, -5.0704e-01, -7.9532e-01,  3.3040e-01],
                ##          [ 9.2689e-01, -8.3979e-01,  6.9537e-01,  4.1327e-02, -1.9942e-01,   2nd ele of batch
                ##            1.1766e+00, -2.9096e-01,  8.6780e-01, -8.5436e-02, -1.5975e+00],
                ##          [ 6.9552e-01,  8.3874e-01,  2.2427e+00,  1.2162e-01, -5.7645e-01,   3rd ele of batch
                ##            8.3561e-01,  1.3859e+00,  5.1665e-02, -6.0968e-01, -3.0454e-01]]])
                ##

##  for the initial hidden state, the parameters are
##                             num_layers   batch_size    hidden_size
hidden_initial  =  torch.randn(    2,            3,          20   )                                           ## (E)
saved_hidden_initial = hidden.clone()                                                                         ## (F)
print(hidden_initial)                                                                                         ## (G)
                ## INITIAL HIDDEN_STATE (IHS) for the first of the two GRU layers:
                ## tensor([[[ 4.5998e-02, -2.3252e+00,  1.4819e+00, -1.1060e+00,  1.0274e+00,    IHS for 1st of
                ##           -2.3240e-01,  1.5145e+00, -5.5164e-01,  5.3943e-01, -1.3754e-01,     batch
                ##            1.9269e+00,  1.8606e+00, -7.1613e-01, -2.1408e+00, -6.5043e-01,
                ##           -1.3288e-02,  6.9259e-01, -7.9836e-01,  9.5284e-01,  4.3687e-01],
                ##          [-2.9889e-01,  1.4296e+00, -4.8474e-01,  8.3727e-01, -2.8728e-01,    IHS for 2nd of
                ##           -1.7080e+00, -7.9470e-01, -1.2507e+00,  7.6840e-01,  6.3289e-01,     batch
                ##           -2.1265e-01, -1.9708e+00, -9.4173e-01,  1.2335e-01,  1.0216e+00,
                ##            1.5063e+00, -8.0150e-01, -1.3001e+00, -6.4836e-01,  9.7768e-01],
```


(Continued on the next slide .....)

Understanding the nn.GRU API (contd.)

(..... continued from the previous slide)

```
##          [ 5.5195e-01,  1.2437e+00, -1.7453e+00, -4.2713e-01, -1.8701e+00,    IHS for 3rd of
##           -1.1637e-01,  1.0971e+00,  3.8930e-02,  9.8278e-01, -4.6390e-02,     batch
##           -1.9551e-01, -9.9347e-01, -2.7240e-01, -9.1676e-01, -4.8000e-01,
##           -1.4576e+00,  2.8401e-01,  5.5791e-01,  4.9224e-01, -1.9487e-01]],
##
## INITIAL HIDDEN_STATE (IHS) for the second of the two GRU layers:
##         [[-9.7618e-01,  9.9822e-01, -2.5622e-01,  1.3409e+00, -8.4607e-01,   IHS for 1st of
##           -9.6225e-01, -1.0095e+00,  9.2252e-02, -3.7621e-01,  2.0340e+00,    batch
##            1.2597e+00, -9.5884e-01, -8.4218e-02, -4.4557e-01, -9.0376e-01,
##            1.4264e-01,  4.3378e-01,  5.3742e-01,  1.0015e+00,  6.4354e-01],
##          [-8.6194e-01,  1.7280e+00, -1.2023e+00,  1.0128e+00,  1.3570e-01,   IHS for 2nd of
##            7.2283e-01,  6.4411e-01,  9.9038e-01,  6.2875e-01,  1.2034e+00,    batch
##            5.6466e-01,  2.4503e-01,  1.1460e+00,  1.5808e+00, -6.6498e-01,
##            2.5036e-01,  2.8193e+00, -1.4831e+00, -1.7151e-01, -7.5258e-01],
##          [-9.5357e-01, -1.3288e-01,  5.4279e-01,  1.2880e+00, -9.9795e-01,   IHS for 3rd of
##            3.6519e-01, -1.1906e+00, -8.3167e-01, -2.2317e-01, -5.5570e-01,    batch
##           -5.5081e-02,  2.9953e-04,  3.5435e-01, -2.0565e-01,  1.7733e-01,
##           -1.2533e+00,  6.8476e-01,  9.1666e-01,  1.9814e-02,  1.7234e-02]]])
```

```
output, hidden   =  rnn(input, hidden_initial)                                                    ## (H)
print(output)                                                                                     ## (I)
```

```
##  OUTPUT 20 ELEMENTS AT THE FIRST TIME STEP:
## tensor([[[-0.6145,  0.7062,  0.2062,  0.8882, -0.5146, -0.8570, -0.7565,   for batch ele 1
##            0.0102, -0.1173,  0.0984,  0.6949, -0.4685,  0.6003, -0.3886,
##           -0.2867,  0.1176,  0.3916,  0.4601,  0.6118, -0.0940],
##          [-0.0267,  1.4833, -0.9404,  0.7825,  0.2087,  0.2681,  0.5103,   for batch ele 2
##            0.8062,  0.4681,  0.1633,  0.0023,  0.2424,  0.1165,  1.0775,
##           -0.3483,  0.3051,  1.9098, -1.3842, -0.2044, -0.2472],
##          [-0.4676, -0.1418,  0.5482,  0.8199, -0.5905,  0.0650, -0.6915,   for batch ele 3
##           -0.0192, -0.1617, -0.4756,  0.1122, -0.0111,  0.5523, -0.0151,
##           -0.0049, -0.9140,  0.3833,  0.6870, -0.1147, -0.0878]],
##
##  OUTPUT 20 ELEMENTS AT THE SECOND TIME STEP:
##         [[-0.5166,  0.4246,  0.3781,  0.5633, -0.2118, -0.6227, -0.4139,   for batch ele 1
##           -0.0219, -0.0266, -0.2870,  0.4384, -0.1863,  0.6857, -0.3368,
##            0.0123,  0.0056,  0.2856,  0.3321,  0.3360, -0.1385],
##          [ 0.0901,  1.0802, -0.7471,  0.6326,  0.1298, -0.0344,  0.3584,   for batch ele 2
##            0.5079,  0.3032,  0.1163, -0.1710,  0.2856, -0.0677,  0.5892,
##           -0.1919,  0.3106,  1.0635, -1.1712, -0.1801, -0.1011],
##          [-0.2911, -0.1248,  0.5043,  0.3676, -0.2735, -0.0434, -0.4250,   for batch ele 3
##            0.2247, -0.1488, -0.2199,  0.1505,  0.1404,  0.4654,  0.0302,
##           -0.1745, -0.6136,  0.2471,  0.6004, -0.0972, -0.1815]],
##
##  OUTPUT 20 ELEMENTS AT THE THIRD TIME STEP:
##         [[-0.3052,  0.1871,  0.3690,  0.3464, -0.0971, -0.3828, -0.1962,   for batch ele 1
##            0.0825, -0.0161, -0.2086,  0.2517,  0.1071,  0.5743, -0.3282,
##            0.2200, -0.0317,  0.1650,  0.2292,  0.1956, -0.1170],
##          [-0.0207,  0.7124, -0.5238,  0.3642,  0.1458, -0.1568,  0.0559,   for batch ele 2
##            0.3810,  0.1652,  0.2176, -0.2217,  0.3645, -0.0561,  0.1549,
##           -0.1161,  0.3089,  0.5574, -0.8695, -0.1236, -0.0485],
##          [-0.1570, -0.1034,  0.3929,  0.2215, -0.0594, -0.1542, -0.3284,   for batch ele 3
##            0.1930, -0.1608, -0.0676,  0.0049,  0.2860,  0.3306, -0.0330,
##           -0.2524, -0.4262,  0.1278,  0.4482,  0.0565, -0.2077]],
##
##  OUTPUT 20 ELEMENTS AT THE FOURTH TIME STEP:
##         [[-0.1740,  0.0372,  0.3267,  0.1715, -0.0175, -0.2233, -0.0906,   for batch ele 1
##            0.1270, -0.0995, -0.1259,  0.1093,  0.2768,  0.3745, -0.3385,
##            0.2305, -0.0711,  0.0838,  0.1851,  0.1480, -0.0888],
##          [-0.0654,  0.3784, -0.3303,  0.2705,  0.1459, -0.2213, -0.1266,   for batch ele 2
##            0.2145, -0.0288,  0.2305, -0.2224,  0.4744, -0.0907, -0.0436,
##           -0.0787,  0.3041,  0.2788, -0.5141, -0.0366, -0.0631],
##          [-0.1080, -0.1076,  0.2962,  0.1110,  0.0511, -0.2023, -0.2667,   for batch ele 3
##            0.1459, -0.1909,  0.0347, -0.0414,  0.3788,  0.2649, -0.0826,
##           -0.1675, -0.2601,  0.0454,  0.3471,  0.0703, -0.1937]],
```


(Continued on the next slide .....)

Understanding the nn.GRU API (contd.)

(..... continued from the previous slide)

```
                ##  OUTPUT 20 ELEMENTS AT THE FIFTH TIME STEP:
                ##         [[-0.1946, -0.0495,  0.3520, -0.0102,  0.0236, -0.1250, -0.0506,   for batch ele 1
                ##            0.1075, -0.1467, -0.0938,  0.0665,  0.3229,  0.3234, -0.3478,
                ##            0.1698, -0.1750,  0.0682,  0.1237,  0.0957, -0.0100],
                ##          [-0.1428,  0.1841, -0.1660,  0.1063,  0.1473, -0.1861, -0.1587,   for batch ele 2
                ##            0.1402, -0.1128,  0.1849, -0.2049,  0.4618, -0.0279, -0.2384,
                ##           -0.0194,  0.1890,  0.1170, -0.2755, -0.0067, -0.0164],
                ##          [-0.0579, -0.1543,  0.1382,  0.0576,  0.1095, -0.1854, -0.2986,   for batch ele 3
                ##            0.1280, -0.2162,  0.1274, -0.1198,  0.4345,  0.1287, -0.1186,
                ##           -0.2176, -0.1788, -0.0157,  0.2952,  0.1191, -0.2073]]],
                ##        grad_fn=<StackBackward>)

print(hidden)                                                                                                                    ## (J)
                ##    FINAL value of hidden in Layer 1:
                ## tensor([[[-0.0819, -0.1318, -0.0695, -0.1417,  0.0672, -0.2632,  0.1564,   for batch ele 1
                ##           -0.1025,  0.1888, -0.0014, -0.2386, -0.0271,  0.1169, -0.0621,
                ##            0.3172,  0.3049,  0.3771,  0.2958,  0.1154,  0.3129],
                ##          [-0.1143, -0.1312, -0.2996,  0.0806,  0.3955,  0.0143,  0.2247,   for batch ele 2
                ##           -0.1451,  0.2493, -0.2628, -0.2526, -0.3790,  0.3634, -0.4030,
                ##            0.0979,  0.4831, -0.1072, -0.2282, -0.1177, -0.0334],
                ##          [-0.0130,  0.0913, -0.1344,  0.2251, -0.3673,  0.3260,  0.2173,   for batch ele 3
                ##            0.1227, -0.1169, -0.2013,  0.2644, -0.3908,  0.2944, -0.2770,
                ##            0.0619,  0.2666, -0.1293,  0.0663, -0.2971, -0.3164]],
                ##
                ##    FINAL value of hidden in Layer 2:
                ##         [[-0.1946, -0.0495,  0.3520, -0.0102,  0.0236, -0.1250, -0.0506,   for batch ele 1
                ##            0.1075, -0.1467, -0.0938,  0.0665,  0.3229,  0.3234, -0.3478,
                ##            0.1698, -0.1750,  0.0682,  0.1237,  0.0957, -0.0100],
                ##          [-0.1428,  0.1841, -0.1660,  0.1063,  0.1473, -0.1861, -0.1587,   for batch ele 2
                ##            0.1402, -0.1128,  0.1849, -0.2049,  0.4618, -0.0279, -0.2384,
                ##           -0.0194,  0.1890,  0.1170, -0.2755, -0.0067, -0.0164],
                ##          [-0.0579, -0.1543,  0.1382,  0.0576,  0.1095, -0.1854, -0.2986,   for batch ele 3
                ##            0.1280, -0.2162,  0.1274, -0.1198,  0.4345,  0.1287, -0.1186,
                ##           -0.2176, -0.1788, -0.0157,  0.2952,  0.1191, -0.2073]]],
                ##        grad_fn=<StackBackward>)

##########################################################################################
#                       Stepping through the sequence one element at a time
##########################################################################################

print("\n\n\nExperiments with overtly stepping through the input sequence:\n\n")

input = saved_input                                                                                                              ## (K)
hidden = saved_hidden_initial                                                                                                    ## (L)
##                    seq length   batch_size   hidden_size
output = torch.zeros(     5,           3,          20,       dtype=float  )                                                    ## (M)

##  We can use the same GRU instance as constructed previously because it does not care about
##  the sequence length.

for i in range(input.shape[0]):                                                                                                  ## (N)
    output[i], hidden   =  rnn(torch.unsqueeze(input[i],0), hidden)                                                              ## (O)

##  You will notice that the following output is exactly the same as shown previously:
print(output)                                                                                                                    ## (P)
                ## tensor([[[-0.6145,  0.7062,  0.2062,  0.8882, -0.5146, -0.8570, -0.7565,
                ##            0.0102, -0.1173,  0.0984,  0.6949, -0.4685,  0.6003, -0.3886,
                ##           -0.2867,  0.1176,  0.3916,  0.4601,  0.6118, -0.0940],
                ##          [-0.0267,  1.4833, -0.9404,  0.7825,  0.2087,  0.2681,  0.5103,
                ##            0.8062,  0.4681,  0.1633,  0.0023,  0.2424,  0.1165,  1.0775,
                ##           -0.3483,  0.3051,  1.9098, -1.3842, -0.2044, -0.2472],
                ##          [-0.4676, -0.1418,  0.5482,  0.8199, -0.5905,  0.0650, -0.6915,
                ##           -0.0192, -0.1617, -0.4756,  0.1122, -0.0111,  0.5523, -0.0151,
                ##           -0.0049, -0.9140,  0.3833,  0.6870, -0.1147, -0.0878]],
```


(Continued on the next slide .....)

Understanding the nn.GRU API (contd.)

(..... continued from the previous slide)

```
##         [[-0.5166,  0.4246,  0.3781,  0.5633, -0.2118, -0.6227, -0.4139,
##           -0.0219, -0.0266, -0.2870,  0.4384, -0.1863,  0.6857, -0.3368,
##            0.0123,  0.0056,  0.2856,  0.3321,  0.3360, -0.1385],
##          [ 0.0901,  1.0802, -0.7471,  0.6326,  0.1298, -0.0344,  0.3584,
##            0.5079,  0.3032,  0.1163, -0.1710,  0.2856, -0.0677,  0.5892,
##           -0.1919,  0.3106,  1.0635, -1.1712, -0.1801, -0.1011],
##          [-0.2911, -0.1248,  0.5043,  0.3676, -0.2735, -0.0434, -0.4250,
##            0.2247, -0.1488, -0.2199,  0.1505,  0.1404,  0.4654,  0.0302,
##           -0.1745, -0.6136,  0.2471,  0.6004, -0.0972, -0.1815]],
##
##         [[-0.3052,  0.1871,  0.3690,  0.3464, -0.0971, -0.3828, -0.1962,
##            0.0825, -0.0161, -0.2086,  0.2517,  0.1071,  0.5743, -0.3282,
##            0.2200, -0.0317,  0.1650,  0.2292,  0.1956, -0.1170],
##          [-0.0207,  0.7124, -0.5238,  0.3642,  0.1458, -0.1568,  0.0559,
##            0.3810,  0.1652,  0.2176, -0.2217,  0.3645, -0.0561,  0.1549,
##           -0.1161,  0.3089,  0.5574, -0.8695, -0.1236, -0.0485],
##          [-0.1570, -0.1034,  0.3929,  0.2215, -0.0594, -0.1542, -0.3284,
##            0.1930, -0.1608, -0.0676,  0.0049,  0.2860,  0.3306, -0.0330,
##           -0.2524, -0.4262,  0.1278,  0.4482,  0.0565, -0.2077]],
##
##         [[-0.1740,  0.0372,  0.3267,  0.1715, -0.0175, -0.2233, -0.0906,
##            0.1270, -0.0995, -0.1259,  0.1093,  0.2768,  0.3745, -0.3385,
##            0.2305, -0.0711,  0.0838,  0.1851,  0.1480, -0.0888],
##          [-0.0654,  0.3784, -0.3303,  0.2705,  0.1459, -0.2213, -0.1266,
##            0.2145, -0.0288,  0.2305, -0.2224,  0.4744, -0.0907, -0.0436,
##           -0.0787,  0.3041,  0.2788, -0.5141, -0.0366, -0.0631],
##          [-0.1080, -0.1076,  0.2962,  0.1110,  0.0511, -0.2023, -0.2667,
##            0.1459, -0.1909,  0.0347, -0.0414,  0.3788,  0.2649, -0.0826,
##           -0.1675, -0.2601,  0.0454,  0.3471,  0.0703, -0.1937]],
##
##         [[-0.1946, -0.0495,  0.3520, -0.0102,  0.0236, -0.1250, -0.0506,
##            0.1075, -0.1467, -0.0938,  0.0665,  0.3229,  0.3234, -0.3478,
##            0.1698, -0.1750,  0.0682,  0.1237,  0.0957, -0.0100],
##          [-0.1428,  0.1841, -0.1660,  0.1063,  0.1473, -0.1861, -0.1587,
##            0.1402, -0.1128,  0.1849, -0.2049,  0.4618, -0.0279, -0.2384,
##           -0.0194,  0.1890,  0.1170, -0.2755, -0.0067, -0.0164],
##          [-0.0579, -0.1543,  0.1382,  0.0576,  0.1095, -0.1854, -0.2986,
##            0.1280, -0.2162,  0.1274, -0.1198,  0.4345,  0.1287, -0.1186,
##           -0.2176, -0.1788, -0.0157,  0.2952,  0.1191, -0.2073]]],
##        dtype=torch.float64, grad_fn=<CopySlices>)
```

```
## The following values shown for the final hidden are also exactly the same as before:
print(hidden)
```

```
## tensor([[[-0.0819, -0.1318, -0.0695, -0.1417,  0.0672, -0.2632,  0.1564,
##           -0.1025,  0.1888, -0.0014, -0.2386, -0.0271,  0.1169, -0.0621,
##            0.3172,  0.3049,  0.3771,  0.2958,  0.1154,  0.3129],
##          [-0.1143, -0.1312, -0.2996,  0.0806,  0.3955,  0.0143,  0.2247,
##           -0.1451,  0.2493, -0.2628, -0.2526, -0.3790,  0.3634, -0.4030,
##            0.0979,  0.4831, -0.1072, -0.2282, -0.1177, -0.0334],
##          [-0.0130,  0.0913, -0.1344,  0.2251, -0.3673,  0.3260,  0.2173,
##            0.1227, -0.1169, -0.2013,  0.2644, -0.3908,  0.2944, -0.2770,
##            0.0619,  0.2666, -0.1293,  0.0663, -0.2971, -0.3164]],
##
##         [[-0.1946, -0.0495,  0.3520, -0.0102,  0.0236, -0.1250, -0.0506,
##            0.1075, -0.1467, -0.0938,  0.0665,  0.3229,  0.3234, -0.3478,
##            0.1698, -0.1750,  0.0682,  0.1237,  0.0957, -0.0100],
##          [-0.1428,  0.1841, -0.1660,  0.1063,  0.1473, -0.1861, -0.1587,
##            0.1402, -0.1128,  0.1849, -0.2049,  0.4618, -0.0279, -0.2384,
##           -0.0194,  0.1890,  0.1170, -0.2755, -0.0067, -0.0164],
##          [-0.0579, -0.1543,  0.1382,  0.0576,  0.1095, -0.1854, -0.2986,
##            0.1280, -0.2162,  0.1274, -0.1198,  0.4345,  0.1287, -0.1186,
##           -0.2176, -0.1788, -0.0157,  0.2952,  0.1191, -0.2073]]],
```

```
## (Q)
```

Outline

1 Sentiment Analytics 9

2 A Sentiment Analysis Dataset 11

3 Challenges in Processing Text with Neural Networks 16

4 An RNN That Predicts the Ethnic Origin of a Last Name 22

5 Solving the Problem of Sentiment Analysis 31

6 Gating Mechanisms to Deal with the Problem of Vanishing Gradients in RNNs 47

7 Gated Recurrent Unit (GRU) 54

8 The GRU Based Classes in DLStudio 61

9 Understanding the `torch.nn.GRU` API 68

10 Data Prediction 89

Time Series Data

- A time-series consists of a sequence of observations recorded at regular intervals. These could, for example, be:
  - the price of a stock share recorded every hour
  - the hourly recordings of electrical load at your local power utility company
  - the mean average temperature recorded on an annual basis
  - and so on.
- Using past observations, we want to predict the value of the next one.
- While there is much in common between data prediction and other forms of sequence based learning and inference, the data prediction problem presents interesting challenges of its own that are stated on the next three slides.

What Distinguishes Data Prediction Problems — Datetime Conditioning

- Time-series data typically comes with a "datetime" stamp for each observation, an in the following example that shows the first few entries in one of the data files for the utility power-load prediction problem made available by Kaggle:

```
2013-12-31 01:00:00,  1861.0
2013-12-31 02:00:00,  1835.0
2013-12-31 03:00:00,  1841.0
2013-12-31 04:00:00,  1872.0
...                   ...
```

- As you can see, the datafile has two comma separated columns: the left column is the datetime and the right the electrical load recorded in megawatts at that time.

- Representing datetime as a one-dimensional ever-increasing time value does not work for data prediction if the observations depend on the time of the day, the day of the week, the season of the year, etc. **What you need is a multi-dimensional encoding of datetime.**

What Distinguishes Data Prediction Problems — Input Data Chunking

- The notion of a sentence that is important in text analytics does **not** carry over to the data prediction problem.
- In general, you would want a prediction to be made using all the past observations. You are also likely to believe that the longer the time span over which the past observations are available, the greater the likelihood that the prediction made for the next time instant will be accurate.
- When the sequential data available for training a predictor is arbitrarily long, as is the case with numerical data in general, you would need to decide how to "chunk" the data — that is, **how to extract sub-sequences from the data for the purpose of training a neural network.**

What Distinguishes Data Prediction Problems — Data Normalization

- As you know by this time, neural networks require that your input data be normalized to the [0,1] interval, assuming it consists of non-negative numbers, or the [-1,1] interval otherwise. For multi-channel inputs, with each channel being represented by a different axis of the input tensor, such normalization is generally carried out separately for each channel.
- In most neural-network based solutions, after you have normalized the input data, you can forget about it. **You don't have that luxury when solving a data prediction problem.**
- As you would expect, the next value predicted by an algorithm must be at the same scale as the original input input data. This requires that the output of a neural-network-based prediction algorithm must be "inverse-normalized". And that, in turn, requires remembering the normalization parameters used in each channel of the input data.

The Main Goals of This Section on Data Prediction

- Now that you understand how the data prediction problem differs from the other problems you have looked at so far, it is time for me to state my main goals in this section of the lecture:

  1. To further deepen your understanding of a GRU. At this point, your understanding of a GRU is likely to be based on calling PyTorch's GRU in your own code. [**Using a pre-programmed implementation for a GRU makes life easy and you also get a piece of highly optimized code that you can just call in your own code. However, with a pre-programmed GRU, you are unlikely to get insights into how such an RNN is actually implemented.**]
     To give you further insights into how the gating action in a GRU is actually implemented, my data prediction code that I'll be presenting in this section is based on `pmGRU` (for Poor Man's GRU), which is my implementation of a light-weight GRU proposed by Heck and Salem.

  2. To demonstrate how you can use a Recurrent Neural Network (RNN) for data prediction taking into account the datetime conditioning, input data chunking, and data normalization issues mentioned earlier. This I will do with the help of DLStudio's co-class named **DataPrediction**.

Minimally Gated GRU Proposed by Heck and Salem

- As mentioned already, `pmGRU` in DLStudio's co-class DataPrediction is my implementation of the "Minimally Gated Unit" GRU variant that was presented by Joel Heck and Fathi Salem in their paper "*Simplified Minimal Gated Unit Variations for Recurrent Neural Networks*". The following equations describe the gating action in this GRU:

$$
\begin{aligned}
\mathbf{f}_t &= \sigma(W_f \mathbf{x}_t \;+\; U_f \mathbf{h}_{t-1}) \\
\tilde{\mathbf{h}}_t &= tanh(W_h \mathbf{x}_t \;+\; U_h(\mathbf{f}_t \odot \mathbf{h}_{t-1})) \\
\mathbf{h}_t &= (1 - \mathbf{f}_t) \odot \mathbf{h}_{t-1} \;+\; \mathbf{f}_t \odot \tilde{\mathbf{h}}_t
\end{aligned}
$$

- Compare these equations with those shown on Slide 60 for the regular GRU. As you can see, we have replaced the *update* and the *reset* gates, z and r of a regular GRU with the *forget* gate f shown above.

- As on Slide 60, σ is the Sigmoid function, and W_f and U_f the matrices of the learnable parameters for the *forget* gate, and W_h and U_h the learnable matrices for the Candidate Hidden State $\tilde{\mathbf{h}}_t$.

Minimally Gated GRU (contd.)

- Again comparing the pmGRU on the previous slide with the regular GRU on Slide 60, two things are obvious:
  - From the third equation on the previous slide, the forget gate in pmGRU plays the same role as the update gate in a regular GRU — in sense that it decides in what ratio to combine the previous value of the hidden state at time $t-1$ and the candidate hidden state at time t for the production of the latest hidden state;
  - From the second equation on the previous slide, the forget gate in pmGRU is also pressed into service for immediate modulation of the elements of the previous hidden state before they can be used for constructing the candidate hidden state at time t.
- Therefore, the forget gate in pmGRU serves both as the update gate and the reset gate in the regular GRU.

The pmGRU Implementation in DLStudio

- Shown below is my implementation of `pmGRU` based on the equations shown on the previous slide. The statement in Line (A) is an implementation of the equation for f_t on the previous slide. I have combined the learnable matrices W_f and U_f in that equation into the weights in the `nn.Linear` layer of the network.
- I have also combined the learnable matrices W_h and U_h into the weights in the `nn.Linear` layer in Line (B). The `nn.Linear` layer you see in Line (C) is for shaping the final output of the network.

```
class pmGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, batch_size):
        self.input_size  = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.batch_size  = batch_size
        self.project1 = nn.Sequential( nn.Linear(self.input_size + self.hidden_size, self.hidden_size), nn.Sigmoid() )    ## (A)
        self.project2 = nn.Sequential( nn.Linear( self.input_size + self.hidden_size, self.hidden_size), nn.Tanh() )      ## (B)
        self.project3 = nn.Sequential( nn.Linear( self.hidden_size, self.output_size ), nn.Tanh() )                       ## (C)

    def forward(self, x, h, sequence_end=False):
        combined1 = torch.cat((x, h), 1)
        forget_gate = self.project1(combined1)
        interim =  forget_gate * h
        combined2  = torch.cat((x, interim), 1)
        output_interim =  self.project2( combined2 )
        output = (1 - forget_gate) * h  +  forget_gate * output_interim
        if sequence_end == False:
            return output, output
        else:
            final_out = self.project3(output)
            return final_out, final_out

   def init_hidden(self):
        weight = next(self.parameters()).data
        hidden = weight.new(self.batch_size, self.hidden_size).zero_()
        return hidden
```

DLStudio's Co-Class DataPrediction

- The `pmGRU` class shown on the previous slide is actually an inner class of the DataPrediction co-class of DLStudio.
- As you would imagine, the main job of DataPrediction is to illustrate how to use an RNN (like, say, `pmGRU`) for predicting the next value in a time-series while taking into account the following issues mentioned earlier: (1) datetime conditioning; (2) input data chunking; and (3) input data normalization that can subsequently be reversed for the final predictions.
- The next few slides show the functions in DataPrediction that are used for each of the three issues listed above.
- For a more global understanding how the code in DataPrediction is organized, click on the link "View the DataPrediction Code in your browser" at the top of the main doc page for DLStudio.

Datetime Conditioning for Data Prediction

- As mentioned at the beginning of this section, time-series data typically comes with a "datetime" stamp for each observation. DataPrediction assumes that the training datafiles contain two comma-separated columns, with the first entry for datetime and the second for the actual observations, as shown below:

```
2013-12-31 01:00:00,  1861.0
2013-12-31 02:00:00,  1835.0
...                   ...
```

- We also make the assumption that our predictions need to be sensitive to the hour of the day, the day of the week, the day of the year, and the month of the year (to capture the seasonal effects). The very popular Pandas module makes it easy to extract these temporal parameters from the datetime entry. [**These assumptions are the same as in the previously cited Gabriel Loye's implementation for a data predictor for the electric load at the power utilities.**]

- However, in order to use Pandas for the purpose indicated above, you must first ask Pandas to represent the data in a training file as a `DataFrame` object, as explained on the next slide.

Datetime Conditioning for Data Prediction (contd.)

- In Line (A) in the code shown below, we first construct a Pandas `DataFrame` object for the content of the training file. Setting `parse_dates` to '[0]' in that statement tells Pandas that datetime is in the column indexed 0 (meaning the first column) of the file.

- A `DataFrame` instance is a two-dimensional data structure with rows and columns, like an Excel spreadsheet. [**Each column is represented by a separate Pandas Series object. A DataFrame object has two axes: "Axis 0" and "Axis 1". "Axis 0" points in the direction of increasing row-index values and "Axis 1" points in the direction of increasing column index.**]

- In Lines (B) through (E), we then extract the temporal parameters we need. Next, in Line (F), we sort the rows based on the values in the 'Datetime' column and then drop the 'Datetime' column.

```
def construct_dataframes_from_datafiles(self):
    dataframes = {}
    datafiles = os.listdir( self.dls.dataroot )
    for file in datafiles:
        ## The following code block borrowed from Gabriel Loye's implementation:
        df = pd.read_csv(self.dls.dataroot + file, parse_dates=[0], encoding='utf-8')                    ## (A)
        df['hour'] = df.apply(lambda x: x['Datetime'].hour, axis=1)                                      ## (B)
        df['dayofweek'] = df.apply(lambda x: x['Datetime'].dayofweek, axis=1)                            ## (C)
        df['month'] = df.apply(lambda x: x['Datetime'].month, axis=1)                                    ## (D)
        df['dayofyear'] = df.apply(lambda x: x['Datetime'].dayofyear, axis=1)                            ## (E)
        df = df.sort_values("Datetime").drop("Datetime", axis=1)                                         ## (F)
        dataframes[file] = df
    return dataframes
```

Data Normalization for Data Prediction

- For a data prediction network based on neural networks, we must normalize (meaning, scale) each column of the data in a `DataFrame` object to the [0,1] interval. That sounds like a straightforward thing do do, provided you also keep in mind the fact any normalization carried out on the observations column (the same thing as channel) in the `DataFrame` objects would need to be reversed on the final predictions for the observations.

- You would also need to remember that the normalization parameters you would need to recall at the time of inference may need to be keyed to the individual training files separately.

- In what follows, I'll first clarify what I mean by scaling each column of the `DataFrame` object for a file to the [0,1] interval.

Data Normalization for Data Prediction (contd.)

- The data normalization step changes the value x in each channel of the input data into

$$\frac{x - x_{min}}{x_{max} - x_{min}}$$

where x_{min} is the minimum and x_{max} the maximum of all the values in that channel. So, logically speaking, the data normalization is applied separately to each channel. In our case, each channel of the input corresponds to each column of the `DataFrame` object.

- To illustrate the normalization step with an example, consider the following 3-column (the same thing as 3-channel) input data:

```
X   =    [ [1.0    -1.0    2.0]
           [2.0     0.     0. ]
           [0.      1.0    -1.] ]
```

The min values in each column are given by

```
X.min(axis=0)    =    [0.0   -1.0   -1.]
```

Data Normalization for Data Prediction (contd.)

- Subtracting the values in each column by its min will turn all values into positive numbers:

```
X  -  X.min(axis=0)    =   [ [1.   0.0   3.0]
                             [2.   1.0   1.0]
                             [0.   2.0   0.0] ]
```

- Next, we need to apply the scaling multiplier to these numbers. For each column, the scaling multiplier is

```
                                  1.0
X_std        =    ---------------------------------
                  X.max(axis=0)   -   X.min(axis=0)

             =   [ 0.5       , 0.5       , 0.33333333]
```

- The multiplication

```
( X  -  X.min(axis=0) )  *  X_std
```

gives a normalized version of the data with each of its columns separately mapped to the [0,1] interval.

Data Normalization for Data Prediction (contd.)

- As was shown by Gabriel Loye in his implementation of a predictor, this can be easily done by first constructing an instance of `MinMaxScaler` from the "sklearn.preprocessing" module as shown in Line (A) on the next slide and calling on its `fit_transform()` method as shown in Line (C).
- The explanation given so far applies to the role of the `data_scaler` object defined Line (A) in the code shown on the next slide.
- About the role of a similar data scaler object, `gt_predictions_scaler`, in Line (B), that is for the purpose of remembering the x_{min} and x_{max} values for that column of the input data that is subject to predictions. As already stated, we need those values for inverting the data scaling step at the output of the prediction network.

Data Normalization for Data Prediction (contd.)

- The x_{min} and x_{max} values are returned by the `fit()` method of a `MinMaxScaler` instance, as shown in Line (D). The `MinMaxScaler` instance constructed in Line (B) is for extracting the x_{min} and the x_{max} values for the main observation column that is focus of predictions.

```
def data_normalizer(self, dataframes):

     dataframes_normalized = {}
     for file in dataframes:
         df = dataframes[file]
         data_scaler           =   MinMaxScaler()                                          ## (A)
         gt_predictions_scaler =   MinMaxScaler()                                          ## (B)
         data = data_scaler.fit_transform(df.values)                                       ## (C)
         gt_predictions_scaler.fit(df.iloc[:,0].values.reshape(-1,1))                      ## (D)
         ##  save the MinMaxScaler instance
         self.predictions_scaling_params[file] = gt_predictions_scaler
         dataframes_normalized[file] = data
     return dataframes_normalized
```

Input Data Chunking for Data Prediction

- Since time-series data available for training a predictor can be arbitrarily long, one must decide how to chunk the data — meaning, how to extract sequences from the data for training a neural network.
- The main design parameter that is relevant to this issue is "sequence length", the length of the sequences to be extracted from the datafiles.
- The code shown on the next slide extracts a sequence of length `sequence_length` from each DataFrame object in your dataset and that is done on a running basis. By "running basis" I mean the following: If a datafile has N timestamped observations in it, then its DataFrame representation consists of N rows. Extraction on a running basis will yield $(N - sequence\_length)$ number of sequences. For each sequence thus extracted, the predicted value to be used in training is the next value after the last value in the sequence. This is achieved by lines (B) and (C) in the code.

Input Data Chunking for Data Prediction (contd.)

- From all the sequences thus extracted, we divide them in a prescribed ratio into a training dataset and a testing dataset. We pool all training sequences from all the datafiles in the list `self.training_sequences`, which is initialized in Line (D) and then added to in Line (F). The corresponding predictions are stored in the list `self.training_predictions`, as shown in Lines (E) and (F).
- The testing sequences from each datafile are stored in dict that is keyed to the name of the file, as you can see in Lines (H) and (I).

```
def construct_sequences_from_data(self, dataframes):
    ##  This code is patterned after Gabriel Loye's code for extracting sequences from training data
    for file in dataframes:
        num_records_in_file = len(dataframes[file])
        sequence_length = self.sequence_length
        sequences_from_file = np.zeros((num_records_in_file - sequence_length, sequence_length, dataframes[file].shape[1]))
        predictions_from_file = np.zeros(num_records_in_file - sequence_length)
        predictions_from_file = predictions_from_file.reshape(-1, 1)
        for i in range(sequence_length, num_records_in_file):                                                    ## (A)
            sequences_from_file[i - sequence_length] = dataframes[file][i-sequence_length : i]                   ## (B)
            predictions_from_file[i - sequence_length] = dataframes[file][i,0]                                   ## (C)
        ## We save 10% of the sequences for extracted from each data file for the final evaluation of the
        ## prediction model:
        test_portion = int(0.1*len(sequences_from_file))
        if len(self.training_sequences) == 0:
            self.training_sequences = sequences_from_file[:-test_portion]                                        ## (D)
            self.training_predictions = predictions_from_file[:-test_portion]                                    ## (E)
        else:
            self.training_sequences = np.concatenate((self.training_sequences, sequences_from_file[:-test_portion]))       ## (F)
            self.training_predictions = np.concatenate((self.training_predictions, predictions_from_file[:-test_portion])) ## (G)
        ##  We store the test sequences and their associated labels in the two dicts shown below that are keyed
        ##  to the file names.
        self.test_sequences[file] = (sequences_from_file[-test_portion:])                                        ## (H)
        self.test_gt_predictions[file] = (predictions_from_file[-test_portion:])                                 ## (I)
```

The Kaggle Power-Load Dataset for Training a Data Predictor

- I have tested the `pmGRU` based data prediction network presented in this section on a Kaggle dataset that I first ran into in Gabriel Loye's blog on data prediction with GRUs and LSTMs:

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

- Through the following archive, I make available a subset of this dataset in which all the files use the same 2-column comma-separated format:

  `dataset_for_DataPrediction.tar.gz`

  that you can download directly from the main doc page for the DLStudio module.

- The dataset that Gabriel Loye's blog points to is available at

  `https://www.kaggle.com/robikscube/hourly-energy-consumption`

The Power-Load Dataset (contd.)

- The dataset consists of over 10-years worth of hourly electric load recordings made available by several utilities in the United States. Shown below is the start of a typical datafile:

```
            Datetime,  AEP_MW
2013-12-31 01:00:00,  1861.0
2013-12-31 02:00:00,  1835.0
2013-12-31 03:00:00,  1841.0
2013-12-31 04:00:00,  1872.0
...                   ...
...                   ...
```

  The whitespaces you see in the entries shown above are mine so that you can see better the separation between the two columns.

- As I have already mentioned, given that power load observations depend on the time of the day, the day of the week, the season, etc., it is necessary to create a multi-dimensional encoding of the entries in the datetime column for a prediction framework.

Training the Data Predictor

- Training a predictor for time-series data begins with first constructing an instance of the `pmGRU` RNN that we refer to as `model` in the last statement of the code block shown below. However, before you can do that, you have to first construct an instance of `DLStudio` and also an instance of `DataPrediction` as shown below. [**As you should know by this time, using the DLStudio module requires that, at the least, you would need to construct an instance of the `DLStudio` class and supply basic parameter values, such as those for `batch_size`, `learning_rate`, etc., through that instance, as shown below. And, if you are using the functionality in any of the co-classes of DLStudio, as we are doing with the DataPrediction co-class below, you would need to construct an instance of that class also. You would use the latter instance for supplying the values for the parameters that relate to the functionality of the co-class.**]

```
dls = DLStudio(
                  dataroot = dataroot,
                  path_saved_model = "./saved_PredModel",
                  learning_rate = .001,
                  epochs = 5,
                  batch_size = 1024,
                  use_gpu = True,
              )
predictor = DataPrediction(
                  dlstudio = dls,
                  input_size = 5,      # means that each input into the network consists of one obse
                                       # and 4 values for encoding datetime
                  hidden_size = 256,
                  output_size = 1,     # for the prediction
                  sequence_length = 90,
              )
model = DataPrediction.pmGRU(predictor)
```

Training the Data Predictor (contd.)

- Shown on the next slide is the implementation for the training loop. The parameter `model` in Line (A) is set as shown at the bottom of the previous slide.

- In Line (B), the dataloader is asked for the next batch of data. The shape of the tensor delivered by the dataloader is $(batch\_size, sequence\_length, input\_size)$. For the constructor options shown on the previous slide, this shape is $(1024, 90, 5)$.

- Subsequently, after initializing the hidden state in Line (C), in the loop that starts at Line (D) we step through a sequence, one element at a time, and do so for all the sequences in the batch in parallel. For the dataset in question, each element of a sequence is a vector of 5 values.

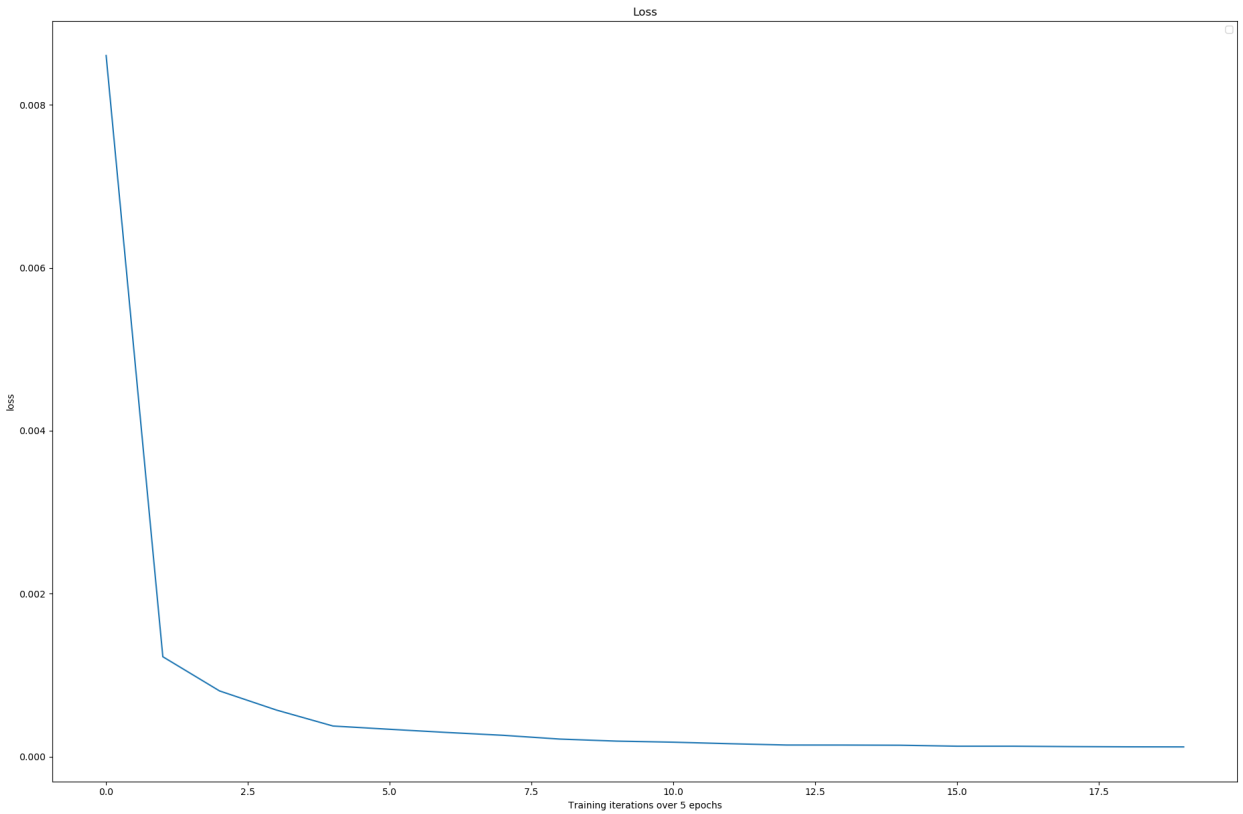
Training the Data Predictor (contd.)

- The conditional logic that you see in Lines (E) through (H) is just to ensure that when the stepping action reaches the end of of sequence, the final output produced has the correct shape — we want it to be a one-element tensor that is the prediction for the sequence.

```
def run_code_for_training_data_predictor(self, train_loader, model):                          ## (A)
    device = self.dls.device
    model.to(device)
    learning_rate = self.dls.learning_rate
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    print("\n\nStarting Training")
    start_time = time.perf_counter()
    epochs = self.dls.epochs
    loss_tally = []
    for epoch in range(epochs):
        avg_loss = 0.
        running_loss = 0.0
        for counter,data in enumerate(train_loader):                                          ## (B)
            x,pred = data
            optimizer.zero_grad()
            ##  Shape of x:   (batch_size, sequence_length, input_size).  For the power-load
            ##  dataset and for the constructor params chosen, the shape of x is  (1024,90,5)
            pred = pred.to(device).float()
            h = model.init_hidden().data.to(device)                                           ## (C)
            ##  The following loop steps through each element of the input sequence, one element at a
            ##  time.  Note that x.shape[1] is the sequence_length, which is 90 as set by the constructor
            ##  in the script in the ExamplesDataPrediction directory.
            for load_sample_index in range(x.shape[1]):                                       ## (D)
                input_sample = x[:,load_sample_index,:]
                input_sample = torch.squeeze(input_sample)
                input_sample = input_sample.to(device).float()
                if load_sample_index < x.shape[1] - 1:                                        ## (E)
                    out, h = model(input_sample, h)                                           ## (F)
                elif load_sample_index == x.shape[1] - 1:                                     ## (G)
                    out, h = model(input_sample, h, sequence_end=True)                        ## (H)
            loss = criterion(out, pred)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            ...
            ...
```

Losses vs. Iterations for the Data Prediction Network

- For the choice of the constructor parameters presented earlier on Slide 110, shown below the loss vs. iterations graph. This plot was produced by running training over 5 epochs.



Evaluating the Prediction Network on Unseen Data

- Following Gabriel Loye, I have evaluated the `pmGRU` based data predictor using the metric "Symmetric Mean Absolute Percentage Error" (sMAPE) that is the sum of the absolute difference between the predicted and the actual values divided by the average of the predicted and the actual values:

$$sMAPE = \frac{100}{N} \sum_{k=1}^{N} \frac{|P_k - GT_k|}{|P_k + GT_k|/2.0}$$

  where N is the number of predictions being used for evaluation, the P_k the kth prediction and GT_k its ground-truth (meaning the actual) value.

- Shown on the slide after the next is the code used for evaluating the data prediction network. The goal is to run the unseen sequences (meaning, the sequences there were set aside at the time the training data was created for the purpose of evaluation) through the trained prediction network.

Evaluating the Prediction Network (contd.)

- As you know well by this time, a time-series data predictor must reverse the data normalization that is applied at the input to the prediction network so that the predicted values are at the original scale. In the code on the next slide, we retrieve the `MinMaxScaler` object in line (C) because we previously stored the x_{min} and x_{max} for the observed data channel in a dictionary keyed to the file names.

- Subsequently, in Line (F), we construct a dataloader for the unseen test data in each file. For at least the educational examples of neural networks, a common practice for evaluation is to process one test data element at a time in an iterative loop. If you did that in our case here, you could be waiting for several minutes for all of the test cases to be processed in a single file. With the help of a dataloader as shown and by exploiting the parallelism made possible by batch-based processing of the data in a GPU, you can reduce the overall evaluation time to just a couple of seconds for all the files.

Evaluating the Prediction Network (contd.)

- In Line (G), we run one batch at a time of the evaluation sequences through our prediction network. [**The shape of x that is returned by the dataloader in Line (G) is ($batch\_size$, $sequence\_length$, $input\_size$). And the variable $gt\_predictions$ holds the actually predicted values for the sequences in x. Line (H) applies the inverse scaling step to the ground-truth values.**]

- The logic in the code in the inference loop that starts in Line (K) is the same as for training. Line (L) inverse-scales the inference for the predictions. From all the calculated predictions and their ground-truth values, we then compute the sMAPE metric in Line (M).

```
def run_code_for_evaluation_on_unseen_data(self, model):
     hidden_size = self.hidden_size
     with torch.no_grad():
         model.eval()
         model.to(device)
         all_predictions = []                                                                    ## (A)
         all_gt_predictions = []                                                                 ## (B)
         for file in sorted(self.test_sequences):
             scaler = self.predictions_scaling_params[file]                                      ## (C)
             minval, maxval = torch.zeros(1,1), torch.zeros(1,1)
             minval[0,0], maxval[0,0] = scaler.data_min_[0], scaler.data_max_[0]                 ## (D)
             minval, maxval = minval.to(device), maxval.to(device)                               ## (E)
             testdata = TensorDataset(torch.from_numpy(self.test_sequences[file]), torch.from_numpy(self.test_gt_predictions[file]))
             testdata_loader = DataLoader(testdata, batch_size=self.dls.batch_size, drop_last=True)    ## (F)
             for x, gt_predictions in testdata_loader:                                           ## (G)
                 gt_predictions = gt_predictions.to(device)
                 gt_predictions_unscaled = ( gt_predictions*(maxval - minval) + minval ).reshape(-1)   ## (H)
                 all_gt_predictions +=  gt_predictions_unscaled.tolist()                         ## (I)
                 h = model.init_hidden().data.to(device)                                         ## (J)
                 for load_sample_index in range(x.shape[1]):                                     ## (K)
                     input_sample = x[:,load_sample_index,:]
                     input_sample = torch.squeeze(input_sample)
                     if load_sample_index < x.shape[1] - 1:
                         out, h = model(input_sample.to(device).float(), h)
                     elif load_sample_index == x.shape[1] - 1:
                         out, h = model(input_sample.to(device).float(), h, sequence_end=True)
                 out_unscaled = ( out*(maxval - minval) + minval ).reshape(-1)                   ## (L)
                 all_predictions +=  out_unscaled.tolist()
         sMAPE = 0
         for i in range(len(all_predictions)):
         sMAPE += np.mean( abs(all_predictions[i] - all_gt_predictions[i]) / (all_gt_predictions[i] + all_predictions[i])/2)/len(all_predictions)  ## (M)
         print("\n\nsMAPE: %.8f" % (sMAPE*100))
```

Evaluation Results for the Data Prediction Network

- For the choice of the constructor parameters presented earlier on Slide 109, shown below are some sample prediction results on the unseen test data.
- The value of the sMAPE metric for all of the test data was around 0.2606. This value will vary a bit on account of the randomization carried out during training.