A First Introduction to Torch.nn for Designing Deep Networks and to DLStudio for Experimenting with Them

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Monday 18th April, 2022 23:46

Preamble

The torch.nn module in PyTorch automates away for us several aspects of PyTorch programming.

As you already know from my Week 3 presentation, Autograd for automatic differentiation plays a central role in what PyTorch does. Ordinarily, in order to take advantage of Autograd, you must tell the system as to which tensors must be subject to the calculation of the partial derivatives by setting their requires_grad attribute to True. However, with the container classes of the torch.nn module, you can move up a notch on the level of automation used. The container classes can figure out on their own as to which tensors should be subject to automatic differentiation.

The torch.nn module is best appreciated through actual demonstrations of the code examples that use this module. So my first objective in this lecture is to give those demos in class. The slides to follow show some of the code snippets from those demos.

A second objective of this lecture is to introduce you to my Python module DLStudio.

Preamble (contd.)

With regard to the second objective mentioned at the bottom of the previous slide, the long-term goal for creating DLStudio is for it to serve as a framework that would make it easier to experiment with different networks, different training and testing protocols, and different ways for a user to interact with a network.

In its current state, DLStudio contains several inner classes, each devoted to a specific applications. These include classes for object detection and localization, semantic segmentation, text classification, etc.

During the last year, DLStudio has also acquired four co-classes. *A co-class resides at the same level of abstraction as the main DLStudio class in the DLStudio module.*

Here are the four co-classes: **(1)** **AdversarialLearning** for experimenting with adversarial learning; **(2)** **Seq2SeqLearning** for sequence-to-sequence learning as in automatic translation; **(3)** **DataPrediction** for making prediction from time-series data; and soon-to-be-released **(4)** **Transformers** for illustrating the basic architectural principles of purely attentional networks.

Preamble (contd.)

The focus of the **AdversarialLearning** co-class is solely on neural-network based probabilistic modeling of a given training dataset. A model thus learned can subsequently be used to create new instances that look surprisingly similar to those in the training dataset. This is what is referred to as "deep fakes" in the popular media.

The **Seq2SeqLearning** co-class is for demonstrating the concepts of sequence-to-sequence learning as needed for, say, automatic translation from one language to another. This implementation of seq2seq learning is based on using recurrence to model the sentences in the source and target languages and in using attention to learn how to map the source language sentences to target language sentences. I have used English-to-Spanish translation for the demonstrations.

The **DataPrediction** co-class is meant specifically for demonstrating what it takes to make predictions using time-series data. While the data prediction problem has much in common with the sequence-to-sequence learning problem, there are also significant differences between the two problems. The differences relate to data normalization, data chunking, datetime conditioning, etc.

Preamble (contd.)

Finally, the soon-to-be-released **Transformers** co-class is meant to convey the basic concepts in purely attention-based networks, that is, networks with no convolutions or recurrence. Unfortunately, such network are very difficult to train and the successful examples in the literature can all be characterized as "Big Data, Big Model, Big Hardware (BDBMBH)". That is, they are based on using large models that are trained on humongously large training datasets using industrial-strength hardware (that is, hardware with multiple high-performance GPUs). Such examples are not the best to use in an educational setting — especially if your teaching philosophy calls for the students to create their own networks. **What you will see in my co-class Transformers is an attempt to swim against the current.** It is an implementation that you could run on a GPU in a university laboratory. What I'll be releasing shortly will be a beta version of my Transformers code that shows decreasing loss with training iterations on a small English-to-Spanish dataset, but that does not yet produce meaningful translations. I am attributing that to insufficient hyperparameter tuning — but I could be wrong.

Outline

Outline

Possible Levels of Automation in DL Programming

You could say that there exist three levels of automation in DL programming:

1. At the lowest level, you manually construct each layer and also manually declare the interfaces between the successive layers. [The one-neuron and multi-neuron networks implemented in the ComputationalGraphPrimer that you used for Homework 3 are examples of manually constructed networks.]
2. You declare the different components of your network architecture in the constructor of a network class and, subsequently, in a method of the class, through explicit declarations you specify the order in which the data is supposed to flow through the different components. [The `torch.nn` based network you created for your Homework 2 solution would be an example such a network.]
3. You eliminate the need for a separate declarations of the components and the order in which the data is supposed to flow through them by using a special container that figures out the order just on the basis of the sequence in which you placed the components in the container.

To Elaborate on the Highest Level of Automation

- In the context of DL programming, at its highest level of automation, a container class is something in which you drop each layer of your network, without explicitly declaring the layer-to-layer interconnections. The container then makes two assumption:
  - The information will flow through the network in the order that is determined by the sequence of the layers you placed in the container.
  - And, that you did not make any errors in the sizes of input/output parameter tensors associated with the different layers.
- This represents the highest level automation — in the sense that you are saved from having to explicitly declare the learnable parameters that would correspond to the interconnections between the layers.
- With `torch.nn`, you achieve this level of automation with the `Sequential` container.

About the Possible Levels of Automation

- Of the three levels of automation listed on Slide 8, obviously, only the 2nd and the 3rd automation levels mentioned could be considered to be automated approaches to network construction. And `torch.nn` allows for both possibilities.

- Despite the fact that torch.nn makes it easy to write your code at the highest level of automation, **that does not make obsolete the practice of creating networks manually or through the second option on the previous slide.**

- Many aspects of research and development with neural networks require manually created networks or those that are created with the second approach.

The `Module` Class in `torch.nn`

- As you can see in the documentation page at

  https://pytorch.org/docs/stable/nn.html

  practically all of the structures in `torch.nn` are derived from the class `torch.nn.Module`

- Here is a typical example (from the doc page) of how you create a network using `torch.nn`:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)
    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

- As the example shows, you declare the individual layers of your network in the constructor initialization code of your own class. Subsequently, it is your declarations in the `forward()` method of this class that tell the system how to route the data through the network.

Outline

The Container Class `torch.nn.Sequential`

Here is the documentation page for this class:

```
class torch.nn.Sequential(*args)

    A sequential container. Modules will be added to it in the order they are passed
    in the constructor. Alternatively, an ordered dict of modules can also be passed in.

    To make it easier to understand, here is a small example:

    # Example of using Sequential
    model = nn.Sequential(
              nn.Conv2d(1,20,5),
              nn.ReLU(),
              nn.Conv2d(20,64,5),
              nn.ReLU()
            )

    # Example of using Sequential with OrderedDict
    model = nn.Sequential(OrderedDict([
              ('conv1', nn.Conv2d(1,20,5)),
              ('relu1', nn.ReLU()),
              ('conv2', nn.Conv2d(20,64,5)),
              ('relu2', nn.ReLU())
            ]))
```

A torch.nn.Sequential Based Demo in DLStudio

To see if the DLStudio class would work with any network that a user may want to experiment with, I copy-and-pasted the the network shown below from the following page by Zhenye at GitHub:

https://zhenye-na.github.io/2018/09/28/pytorch-cnn-cifar10.html

Here is the related code in DLStudio:

```
class Net(nn.Module):
    def __init__(self):
        super(DLStudio.ExperimentsWithSequential.Net, self).__init__()
        self.conv_seqn = nn.Sequential(
            # Conv Layer block 1:
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Conv Layer block 2:
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1),
```

Continued on the next slide ....

.... continued from the previous slide

```
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Dropout2d(p=0.05),
        # Conv Layer block 3:
        nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
        nn.BatchNorm2d(256),
        nn.ReLU(inplace=True),
        nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
    )
    self.fc_seqn = nn.Sequential(
        nn.Dropout(p=0.1),
        nn.Linear(4096, 1024),
        nn.ReLU(inplace=True),
        nn.Linear(1024, 512),
        nn.ReLU(inplace=True),
        nn.Dropout(p=0.1),
        nn.Linear(512, 10)
    )
def forward(self, x):
    x = self.conv_seqn(x)
    # flatten
    x = x.view(x.size(0), -1)
    x = self.fc_seqn(x)
    return x
```

Outline

The `DLStudio` Module

- Since there is never a unique DL solution to a problem, that raises the question of how to experiment with different possibilities without getting lost amongst all the alternatives available.
- `DLStudio` is an attempt by me to address the above problem. What I have released so far is still an early version, which will hopefully grow into a more useful tool down the road.
- The main idea in `DLStudio` is to place all of the common code that you'd need to experiment with the different alternatives in the main definition of the class itself. Subsequently, any alternative solutions to a problem that one would want to experiment with would be placed in the user-defined inner classes of `DLStudio`.

DLStudio Even Allows for a String Based Description for a Network

- For networks that are not too complex, with DLStudio you could define them with a string like

```
convo_layers_config = "2x[128,7,7,1]-MaxPool(2) 1x[16,3,3,1]-MaxPool(2)"
fc_layers_config = [-1,1024,10]
```

- In the configuration string shown above, the basic component of a convolutional network is expressed as

```
nx[a,b,c,d]-MaxPool(k)
```

  where

```
n   =  num of this type of convo layer
a   =  number of out_channels                     [in_channels determined by prev layer]
b,c =  kernel for this layer is of size (b,c)     [b along height, c along width]
d   =  stride for convolutions
k   =  maxpooling over kxk patches with stride of k
```

Parsing the Configuration String and Building the Network

- Given a config string based description of a network, `DLStudio` calls on the following method:

  ```
  parse_config_string_for_convo_layers()
  ```

  to parse the string.

- The output of the parser is supplied to the following method

  ```
  build_convo_layers()
  ```

  to actually build the network using the facilities provided by `torch.nn`

The Network Class for String Based Configs

- The call to `build_convo_layers()` only specifies the individual components of the network you have specified with your config string and the data-flow order for the components.

- The network itself is created by the piece of code shown below:

```
class Net(nn.Module):
    def __init__(self, convo_layers, fc_layers):
        super(DLStudio.Net, self).__init__()
        self.my_modules_convo = convo_layers
        self.my_modules_fc = fc_layers
    def forward(self, x):
        for m in self.my_modules_convo:
            x = m(x)
        x = x.view(x.size(0), -1)
        for m in self.my_modules_fc:
            x = m(x)
        return x
```

Outline

The Inner Classes of DLStudio

To make it easy to experiment with different applications of deep learning, DLStudio contains a set of application-specific inner classes that are listed below:

For Experimenting with Skip Connections : Deep networks suffer from the problem of vanishing gradients that degrades their performance. The `SkipConnections` inner class allows you to experiment with different mitigation strategies for addressing this problem.

For Experimenting with Object Detection and Localization: You can experiment with networks for object detection and localization using the inner class `DetectAndLocalize`. In addition to classification, such networks must also localize them. The localization part requires regression for the coordinates of the bounding box that localize the object.

The Inner Classes of DLStudio (contd.)

For Experimenting with Semantic Segmentation: DLStudio comes with the inner class `SemanticSegmentation` for experimenting with semantic segmentation. Given a scene with multiple objects in it, the purpose of semantic segmentation is to assign correct labels to the pixels in the image, while, at the same time, grouping the pixels together that belong to the same object.

For Experimenting with Text Classification :

You can use the inner class `TextClassification` to experiment with recurrent neural networks (RNN) for analyzing user-feedback text for sentiment analysis.

Outline

Co-Class on Adversarial Learning

- More than anything else, it is what can be accomplished with Adversarial Learning that has fired up popular imagination about the powers of AI. **People are fascinated and terrified by the deep-fakes that can be produced with deep learning algorithms.** If a neural network can transform a pure noise vector into the likeness of a *particular* human face, is there anything it cannot do? — that's what people wonder.

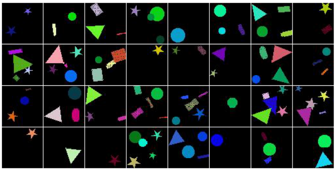
- The purpose of the **AdversarialLearning** co-class is to simply teach you the basic architectural principles that underlie the design of networks that can generate the so-called deep fakes.

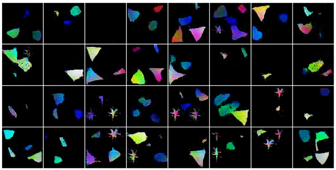
- The two basic components of such networks are the Generator, whose job is transform a pure noise vector (also known as a *latent vector*) into an image that looks like those in the training dataset, and the Discriminator (also called Critic) whose job is to keep on getting better at recognizing the training images as the training proceeds but, at the same time, to disbelieve whatever the Generator is producing

Co-Class on Adversarial Learning (contd.)

- For classroom instruction and demonstration, I'll be using images not as complex as, say, the face images — since my goal in this class is that you should be able to run my demos on ordinary hardware (as opposed to industrial strength hardware).

- Shown below is a comparison of a batch of real images on the left and a batch of fake images on the right.





At the end of 30 epochs of training, shown at left is a batch of real images and, at right, the images produced by the Generator from noise vectors

- The following animated GIF shows how the Generator's output evolves over 30 epochs using the same set of noise vectors.

https://engineering.purdue.edu/DeepLearn/pdf-kak/DG1_generation_animation.gif

Co-Class on Sequence-to-Sequence Learning

- Although, in the world of research, the excitement has shifted from recurrence-based sequence-to-sequence learning to using purely attention-based networks (known as Transformers), it is still important to learn the old-style recurrence based methods. [**Just because airplanes are much faster than, say, automobiles, it does not mean that automobiles have ceased to be important.**]

- Recurrence for Seq2Seq learning means using a neural network with feedback. When you scan a sequence of data, one element at a time, with such a network, the output of the neural network for each input element creates a context for processing the next element.

- **In language translation, recurrence allows for each word to be understood in the context of all the words seen previously.** And with a bidirectional scan of a sentence, such contexts can be incorporated in both directions.

- A concept that is central to using recurrence is the notion of **hidden state**. More on that on the next slide.

Co-Class on Sequence-to-Sequence Learning (contd.)

- After you have finished scanning a sequence of data, the **hidden state** is a compact representation of the entire sequence. In language translation using recurrence, the goal would be to use the hidden state for a source-language sentence to generate its translation in the target language. During training, you would produce a target sentence, again one word at a time, starting with the final hidden state for the source sentence and the evolving hidden state for the target sentence as it is produced one word at a time again through recurrence.

- In the more modern versions of the above, the evolving hidden state in the forward scan of a source language sentence is combined with the the same for the backward scan to generate what are known as the **hidden units** that are then put to use during translation.

- Shown on the next slide is an example of a translation produced by the Seq2SeqLearning co-class of DLStudio.

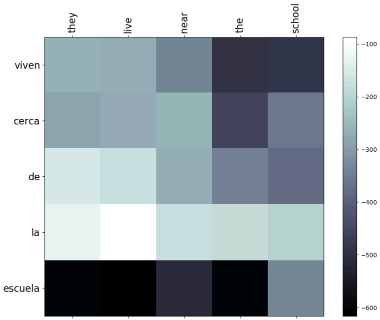
An Example Result with the Seq2Seq Network You'll See in Week 13

```
Original sentence in English:                         SOS they live near the school EOS
Spanish Translation in the "manythings" Corpus:       SOS viven cerca del colegio EOS

Translation Generated by the seq2seq network:         SOS viven cerca de la escuela EOS

Comment:    Easy case. Seq2Seq translation is excellent. "Colegio" and "escuela" are generally
            considered to be synonyms, although in some Spanish speaking countries one may stand
            for the elementary school and the other for the high school.

Attention depiction:
```



Co-Class on Time-Series Data Prediction

- Predicting the next value in a time-series sequence is of great importance in many applications including weather forecasting, investing in stocks and bonds, balancing electrical power grids, and so on.

- It would seem that time-series data prediction has much in common with the sort of sequence-based learning one needs for language translation, text classification, etc. **As it turns out, that's only partly true.**

- Time-series data prediction presents some unique challenges from the standpoint of machine learning: **(1)** Datetime conditioning; **(2)** Data chunking; and **(3)** Data normalization.

- Regarding **datetime conditioning**, each data entry in a data file is typically time-stamped with date-time string following by the actual value of the datum at that datetime.

The DataPrediction Co-Class (contd.)

- In other words, a data file is likely to consist of a continuously running timestamp in one column and the corresponding data observations in a second column. Let's now say that the data you are observing is such that the time of the day, the day of the week, the week in a month, the month in a year, etc., strongly influence the data. **To account for such multi-dimensional effects, the predictor that you are designing will also have to learn a multi-dimensional encoding of what's initially a one-dimensional representation of time.**

- **Data chunking** refers to the fact that, unlike natural language sentences, time-series data has no particular end. Collecting the data can be a continuous process that can go on and on indefinitely with no particular punctuations. From a machine-learning perspective, that raises the issue how to best segment the data for training a next-value predictor? Additionally, should the data segments be formed on a running basis or on a non-overlapping basis?

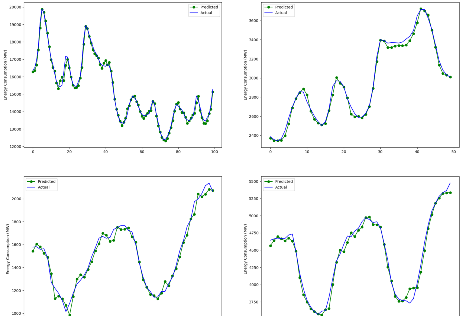
The DataPrediction Co-Class (contd.)

- Finally, the issue of **data normalization** in the context of data prediction refers to the fact that any data scaling and other transformations you carry out on the time-series observation in order to make them suitable for neural-network based processing must be remembered so that you reverse-apply them to the predictions coming out of the network.
- In my Week 12 lecture, I'll demonstrate a data prediction network based on the DataPrediction co-class and show the results obtained with it on the Kaggle electrical utility power-load dataset that was collected over a span of over 10 years.
- Shown on the next slide are some sample prediction results on the unseen test data portion of the overall dataset: The vertical axis for the "Hourly Energy Consumption" in Megawatts and the horizontal axis the hourly tick marks. The curve passing through the small green dots represents the predicted value at that hour based on the N prior data observations with N set to 90.

The DataPrediction Co-Class (contd.)

- Shown below are the prediction results produced by the time-series data prediction network I'll be presenting in my Week 12 lecture.

- As stated earlier, these prediction results are on the **unseen test dataset**, meaning that this data played no role in training the prediction network. Each prdiction represented by a green dot was made on the basis of 90 preceding values in the test dataset. **The ground-truth is shown by the continuous curve in the same plots.**



Co-Class on Transformers

- **Transformers** is a work-in-progress co-class of DLStudio. Its main purpose is to teach the fundamental notions of **self-attention** and **cross-attention**.

- Transformers is still a work-in-progress project because, as is now widely known, **it is extremely difficult to train a small enough model that would fit in the memory of a typical lab-based GPU and do so with a dataset that is not, say, all of Wikipedia).** As I mentioned in my Preamble to this lecture, all of the successful examples of Transformers in the literature are best described as BdBmBh examples where the acronym stands for "Big Data, Big Model, Big Hardware".

- What you will see in the Transformers co-class is an attempt at seq2seq learning, of the same sort as illustrated by the Seq2SeqLearning co-class, but with no recurrence or convolution.

- Overall, the logic used my implementation of Transformers is the same as proposed originally in the now seminal paper by Vaswani et

Outline

Examples Directory for the Main DLStudio Class

The **Examples** subdirectory in the DLStudio distribution contains the following scripts based on the code in the main DLStudio class:

playing_with_reconfig.py Shows how you can specify a convolution network with a configuration string. The DLStudio module parses the string constructs the network.

playing_with_sequential.py Shows you how you can call on a custom inner class of the 'DLStudio' module that is meant to experiment with your own network. The name of the inner class in this example script is ExperimentsWithSequential

playing_with_cifar10.py This is similar to the previous example script but is based on the inner class ExperimentsWithCIFAR that uses more common examples of networks for playing with the CIFAR-10 dataset.

playing_with_skip_connections.py This script illustrates how to use the inner class BMEnet of the module for experimenting with skip connections in a CNN. the constructor of the BMEnet class comes with two options: "skip_connections" and "depth". By turning the first on and off, you can see the improvement you can get with skip connections. And by giving an appropriate value to the "depth" option, you can see the results for networks of different depths.

custom_data_loading.py This script shows how to use the custom dataloader in the inner class CustomDataLoading of the DLStudio module. That custom dataloader is meant specifically for the PurdueShapes5 dataset that is used in object detection and localization experiments in DLStudio.

object_detection_and_localization.py This script shows how you can use the functionality provided by the inner class DetectAndLocalize of the DLStudio module for experimenting with object detection and localization. Detecting and localizing (D&L) objects in images is a more difficult problem than just classifying the objects. D&L requires that your CNN make two different types of inferences simultaneously, one for classification and the other for localization. For the localization part, the CNN must carry out what is known as regression.

Examples Directory (contd.)

noisy_object_detection_and_localization.py This script in the Examples directory is exactly the same as the previous one, the only difference is that it calls on the noise-corrupted training and testing dataset files.

semantic_segmentation.py This script is based on my mUnet neural network for semantic segmentation of images. The mUnet network assigns an output channel to each different type of object that you wish to segment out from an image.

text_classification_with_TEXTnet.py This script is your first introduction to recurrent neural networks, meaning neural-networks with feedback. This particular example is for text classification.

text_classification_with_TEXTnet_word2vec.py Instead of using one-hot vectors for representing the words in the previous script, this script uses pre-trained word2vec embeddings

text_classification_with_TEXTnetOrder2.py This scrpt uses a "poor man's solution" to "gated recurrence" that you need for dealing with the problem of vanishing gradients in RNN.

text_classification_with_TEXTnetOrder2_word2vec.py This script uses the same network as the previous script, but now we use the word2vec embeddings for representing the words.

text_classification_with_GRU.py This script demonstrates how one can use a GRU (Gated Recurrent Unit) to remediate one of the main problems associated with recurrence – vanishing gradients in the long chains of dependencies created by feedback.

text_classification_with_GRU_word2vec.py While this script uses the same learning network as the previous one, the words are now represented by fixed-sized word2vec embeddings.

ExamplesAdversarialLearning Directory

The **ExamplesAdversarialLearning** subdirectory in the DLStudio distribution contains the following scripts based on the code in the AdversarialLearning class:

dcgan_DG1.py Demonstrates the DCGAN logic for probabilistic data modeling. Uses the PurdueShapes5GAN dataset for the domonstration.

dcgan_DG2.py Shows the sensitivity of the basic DCGAN logic to any variations in the network or in how the weights are initialized.

wgan_CG1.py This script is a demonstration of using the Wasserstein distance for data modeling through adversarial learning. The fourth script adds a Gradient Penalty term to the Wasserstein Distance based logic of the third script. The PurdueShapes5GAN dataset consists of 64x64 images with randomly shaped, randomly positioned, and randomly colored shapes.

wgan_with_gp_CG2.py This script adds a Gradient Penalty term to the Wasserstein Distance based logic of the third script. The PurdueShapes5GAN dataset consists of 64×64 images with randomly shaped, randomly positioned, and randomly colored shapes.

ExamplesSeq2SeqLearning Directory

The **ExamplesSeq2SeqLearning** subdirectory in the DLStudio distribution contains the following scripts based on the code in the Seq2SeqLearning co-class:

seq2seq_with_learnable_embeddings.py This script demonstrates the basic PyTorch structures and idioms to use for seq2seq learning. The application example addressed in the script is English-to-Spanish translation. And the attention mechanism used for seq2seq is the one proposed by Bahdanau, Cho, and Bengio. This network used in this example calls on the nn.Embeddings layer in the encoder to learn the embeddings for the words in the source language and a similar layer in the decoder to learn the embeddings to use for the target language.

seq2seq_with_pretrained_embeddings.py This script, also for seq2seq learning, differs from the previous one in only one respect: it uses Google's word2vec embeddings for representing the words in the source-language sentences (English). As to why I have not used at this time the pre-trained embeddings for the target language is explained in the main comment doc associated with the class Seq2SeqWithPretrainedEmbeddings.

ExamplesDataPrediction Directory

The **ExamplesDataPrediction** subdirectory in the DLStudio distribution contains the following demo script based on the code in the DataPrediction co-class:

power_load_prediction_with_pmGRU.py For making predictions from time-series data, this script uses a subset of the dataset provided by Kaggle for one of their machine learning competitions. The dataset consists of over 10-years worth of hourly electric load recordings made available by several utilities in the east and the midwest of the United States.

ExamplesTransformers Directory

The **ExamplesTransformers** subdirectory in the DLStudio distribution contains the following demo script based on the code in the Transformers co-class:

seq2seq_with_transformer.py This script is for experimenting with the code in the Transformers co-class of the DLStudio module. The goal here is to carry out English-to-Spanish translation in a manner similar to what's demonstrated by the code in the Seq2SeqLearning co-class and the example scripts associated with that co-class. When you run this script, you will see the training loss decrease with the iterations. Unfortunately, at this time, you are not going to be able to see successful translations in the same manner I have demonstrated with my Seq2SeqLearning class. As I have said a couple of time already in this lecture, there do not exist in the literature any successful examples (to the best of what I know) of small-model, small-data, and regular-hardware implementations of Transformers. Despite that, I do hope that with sufficient hyperparameter tuning it might be possible to create such an example. That's the reason for providing the co-class Transformers and this example script.

Outline

DLStudio's Image Datasets for Deep Learning

- DLStudio comes with multiple small-sized training and testing image datasets for experimenting with its inner classes and with the co-class.
- The images for object detection and localization work are 32×32 and the images for semantic segmentation 64×64.
- As to the reason for small-sized images: I want the students to be able to experiment with the basic idioms of the programming needed for deep learning using their personal laptops that may only come with a rudimentary GPU if any at all.
- All of the datasets that you can use with the image-related inner classes are packaged in the following compressed tar archive:

```
datasets_for_DLStudio.tar.gz
```

Image Datasets (contd.)

- When you uncompress the archive mentioned at the bottom of the previous slide, you will gain access to the following PurdueShapes5 datasets:

```
PurdueShapes5-10000-train.gz                  ##  meant for object detection and localization
PurdueShapes5-1000-test.gz

PurdueShapes5-10000-train-noise-20.gz         ##  meant for detection and localization under noisy
PurdueShapes5-1000-test-noise-20.gz           ##    conditions

[there are two additional versions of the above dataset
 for different levels of noise, 50% and 80%]

PurdueShapes5MultiObject-10000-train.gz       ##  meant for semantic segmentation
PurdueShapes5MultiObject-1000-test.gz
```

- The large integer you see in the name of each archive is the number of images it contains. So each of the training datasets mentioned above contains 10,000 training images and the corresponding testing dataset 1000 images.

DLStudio's Dataset for Adversarial Learning

- DLStudio provides the following dataset archive for adversarial learning:

```
datasets_for_AdversarialNetworks.tar.gz
```

- Unpacking the main adversarial learning archive mentioned above gives you access to the following more specific archive:

```
PurdueShapes5GAN-20000.tar.gz
```

- The above mentioned archive consists of 20,000 images of size 64×64 for experimenting with the logic of adversarial learning.

DLStudio's Text Datasets for RNN-Based Learning

- For experimenting with text processing for, say, sentiment analysis, you would need to download the following archive:

```
text_datasets_for_DLStudio.tar.gz
```

- Unpacking this archive will give you the following datasets:

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

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

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

- I extracted these datasets from the publicly available Amazon user-feedback archive for the year 2007.

- The integer number in the name of each dataset is the number of the positive and the number of the negative reviews I extracted from the Amazon archive. In general, the larger the number of reviews, the larger the vocabulary you have to contend with in your solution.

DLStudio's Datasets for Seq2Seq Learning

- For sequence-to-sequence learning with DLStudio, you can download an English-Spanish translation corpus through the folloiwng archive:

  `en_es_corpus_for_seq2sq_learning_with_DLStudio.tar.gz`

- This data archive is a lighly curated version of the main dataset posted at `http://www.manythings.org/anki/` by the folks at "tatoeba.org". My alterations to the original dataset consist mainly of expanding the contractions like "it's", "I'm", "don't", "didn't", "you'll", etc., into their "it is", "i am", "do not", "did not", "you will", etc. The original form of the dataset contains 417 such unique contractions. Another alteration I made to the original data archive is to surround each sentence in both English and Spanish by the "SOS" and "EOS" tokens, with the former standing for "Start of Sentence" and the latter for "End of Sentence".