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

Thursday 25th January, 2024 10:25

Preamble

To TOC

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`. [For example, see the code on the slides 98 and 99 of my Week 3 presentation where you'll see a "manual" configuration of a neural network with two linear layers, one denoted by the matrix `w1` and the other by the matrix `w2`. Note the `requires_grad` property of the learnable parameters in those two layers being set to True in Lines 7 and 8.] 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. This lecture includes such in-class demos.

Preamble (contd.)

A second objective of this lecture is to introduce you to my Python module DLStudio. Since DLStudio contains several classes that reside at the same level of software abstraction as the main DLStudio class, it is best to think of the software package as a platform.

As a learning platform, DLStudio is meant to serve as a sandbox for playing with several key architectural concepts in Deep Learning. Since DLStudio comes with its own easy-to-use datasets, you can quickly jump into its different Examples directories, vary the parameters of the code examples and try to gain key insights into how the implementations work. Subsequently, you can write your own implementation code that would work with a competition-grade dataset.

DLStudio incorporates different aspects of Deep Learning in two different ways: as inner classes of the main class file, which, as you would guess, is named DLStudio, and as co-classes of the main class file.

Whereas all the inner classes reside inside the main DLStudio class, *each co-class resides at the same level of software abstraction as the main DLStudio class.*

Preamble (contd.)

From the standpoint of overall code organization , each co-class is defined in a separate directory at the same level in the overall directory structure of overall DLStudio platform as the main class.

The inner classes are meant to serve as stepping stones to the more advanced topics addressed by the co-classes.

The topics addressed specifically by the inner classes can be considered to be generic — in the sense that the code presented gives you a most basic sense of how one solves a problem in deep learning.

The inner classes show what may be construed as the most basic solutions for problems such as **object detection and localization**, **semantic segmentation**, **text classification**, etc.

The inner classes also address basic architectural issues such as the skip connections that have become vitally important to how neural networks are designed today for real-world problems.

Preamble (contd.)

To learn from these inner classes, the best thing that a user can do is to play with the scripts in the Examples directory of the main distribution. That directory contains scripts for exercising the code in all of the inner classes of the main DLStudio class. **Go to the Examples directory, open up the script you are interested in in one of the windows of your computer, while you execute the script in another window. Next, make changes to the script and see what that does to the results.**

What facilitates learning in the manner described above is that DLStudio comes with its training datasets that you need to download and install separately from the main DLStudio webpage. Your initial experiments with the scripts in the Examples directory will be with these datasets.

The datasets that I supply consist of synthetically generated small sized images that should allow you to do most of your work with just CPU-based processing. You will typically be working with images of size 64×64. These datasets are meant to serve as stepping stones to working with the real-world datasets.

Preamble (contd.)

That brings me to the co-classes of DLStudio.

Here are DLStudio's five 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; **(4)** **Transformers** for illustrating the basic architectural principles of purely attentional networks; and **(5)** **MetricLearning** for how to create embedding vector representations for, say, images in such a way that the embeddings for what are supposed to be similiar images are pulled together and those for dissimilar images are pulled as far apart as possible.

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.

Preamble (contd.)

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.

The **Transformers** co-class is meant to convey the basic concepts in purely attention-based networks, that is, networks with no convolutions or recurrence. In general, such network can be 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 very large training datasets using industrial-strength hardware (that is, hardware with multiple high-performance GPUs).

Preamble (contd.)

Nonetheless, you should be able to use my implementation of Transformers for creating your own code for simple demonstrations like the one in my Week 14 lecture.

Finally, the **MetricLearning** co-class is meant to help you understand the role played by two very special loss functions that enable such learning: the Pairwise Contrastive Loss and the Triplet Loss. Since these losses involve comparing the labels associated with the batch instances, their estimation poses interesting programming challenges — for the simple reason that "for"-loops are an anathema to GPU based processing of data.

Each of the five co-classes comes with its own examples directory with names like `ExamplesTransfomers`, `ExamplesDataPrediction`, etc. As with the inner classes, go into these examples directories and start experimenting with the scripts presented there if you want to learn how to experiment with the co-classes.

Outline

Outline

Possible Levels of Automation in DL Programming

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

- **Level 1:** At the lowest level, you manually construct each layer and also manually declare the connections between the nodes in the successive layers. In addition, you also manually specify a data structure for storing the partial derivatives during the forward propagation of the data through the network. [The one-neuron and multi-neuron networks implemented in the ComputationalGraphPrimer that you used for Homework 3 are examples of manually constructed networks.]

- **Level 2:** You want to take advantage of the Computational Graph that Autograd creates for storing the partial derivatives during the forward propagation of data through the network, but you still want to specify the network topology manually. [The code you see on Slides 98 and 99 of my Week 3 lecture represents this level of automation.]

Possible Levels of Automation in DL Programming (contd.)

- **Level 3:** You declare the different components of your network architecture in the constructor of a network class and, subsequently, in a method of the class typically called `forward()`, through explicit declarations you specify the order in which the data is supposed to flow through the different components. [The code you see on Slide 87 of my Week 3 lecture is probably the smallest possible example of such a network.]

- **Level 4:** 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. [The most commonly used such a container class is `torch.nn.Sequential`.]

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 four levels of automation listed on Slides 11 and 12, obviously, only the 3rd and the 4th levels could be considered to be truly 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 and the third options on Slides 11 and 12.**
- Many aspects of education, research, and development with neural networks require manually created networks or those that are created with the second and the third approaches.

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:

https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html#torch.nn.Sequential

****

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

    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 are a couple of small examples:

```
    # 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 More Elaborate torch.nn.Sequential Example

- What follows is a more elaborate example of the use of the torch.nn.Sequential container. This code is a part of the code library at

  https://github.com/Zhenye-Na/blog

- Just to see if my general-purpose DLStudio platform would work with Zhenye's code, I simply copy-and-pasted the code shown below in an inner class of the DLStudio main class:

```
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
```

- As I mentioned on the previous slide, using this code in one of the inner classes of DLStudio is to demonstrate the fact that you may be able to drop other people's networks into DLStudio and use the functionality built into the platform for experimenting with that code.

Outline

Intro to DLStudio

- The idea of DLStudio is to offer an integrated software platform for teaching (and learning) a wide range of basic architectural aspects of deep-learning neural networks.
- **But why create a separate platform?**
- Most instructors who teach deep learning ask their students to download the so-called famous networks from, say, GitHub and become familiar with them by running them on the datasets used by the authors of those networks. This approach is akin to teaching automobile engineering by asking the students to take the high-powered cars of the day out for a test drive and to become familiar with their handling.
- In my opinion, this rather commonly used approach does not work for instilling in the student a deep understanding of the issues related to network architectures.

Intro to DLStudio (contd.)

- DLStudio offers its own implementations for a variety of key elements of neural network architectures. These implementations, along with their explanations through detailed slides at our Deep Learning class website at Purdue, **result in an educational framework that is much more efficient in what it can deliver within the time constraints of a single semester**.
- DLStudio facilitates learning through a combination of **inner classes** of the main module class — called DLStudio naturally — and several **co-classes** of the main class.
- For the most part, the common code that you'd need in different scenarios for using neural networks has been placed inside the definition of the main DLStudio class in a file named `DLStudio.py` in the distribution. That makes more compact the definition of the other inner classes within DLStudio. And, to a certain extent, that also results in a bit more compact code in the co-classes of DLStudio.

Intro to DLStudio (contd.)

- The inner-classes and the co-classes listed in the next two sections of this lecture are meant to give you a well-rounded exposure to the following fundamental architectural elements of deep-learning networks:

  1. Linear Layers
  2. Convolutional Layers
  3. Recurrent Layers
  4. Attention Layers

- DLStudio will also expose you to higher-level architectural organizations of the elements listed above in different kinds of Encoder-Decoder networks.

- And, in addition to the above, DLStudio would make you thoroughly aware of some fundamental problems in neural learning that are caused by vanishing or exploding gradients and how one mitigates against them with skip connections between the layers.

Outline

The Inner Classes of DLStudio

Each inner class is designed to illustrate either a fundamental aspect of neural learning or something that is foundational to an application based on neural learning. These 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

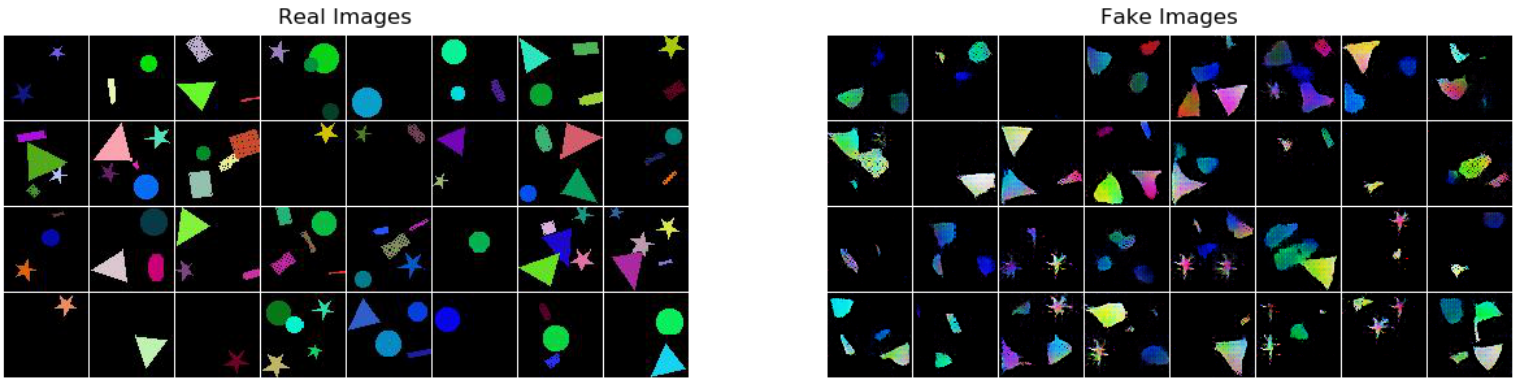
- Adversarial Learning first seized public imagination when people started generating fake images with this approach. That AI could be used to create "deep fakes" both fascinated and terrified folks at large.
- As it turns out, now we have two radically different methods for creating deep fakes: Adversarial Learning and Diffusion. **A soon-to-be-released version of DLStudio will include Diffusion as a new co-class.** With both approaches, you can have a neural network transform a pure noise vector into the likeness of, say, a *particular* human face.
- 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 called the Generator and the Discriminator.

Co-Class on Adversarial Learning (contd.)

- The Generator's 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 at its output.
- In the latest research literature, it is possible to associate specific elements of the latent vector with specific features in the output of the Generator.
- 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).

Co-Class on Adversarial Learning (contd.)

- 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.

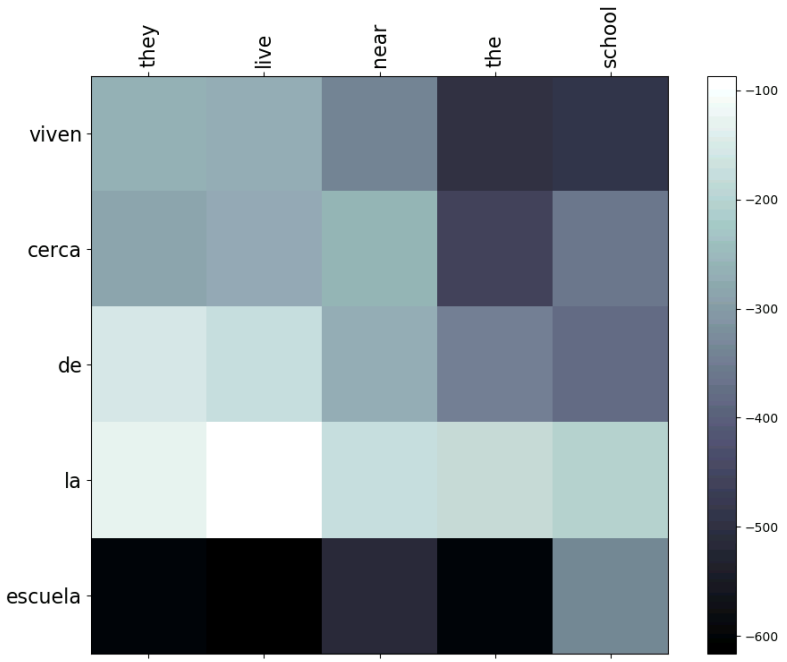
An Example Result with the Seq2Seq Network You'll See in Week 13

```
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.**

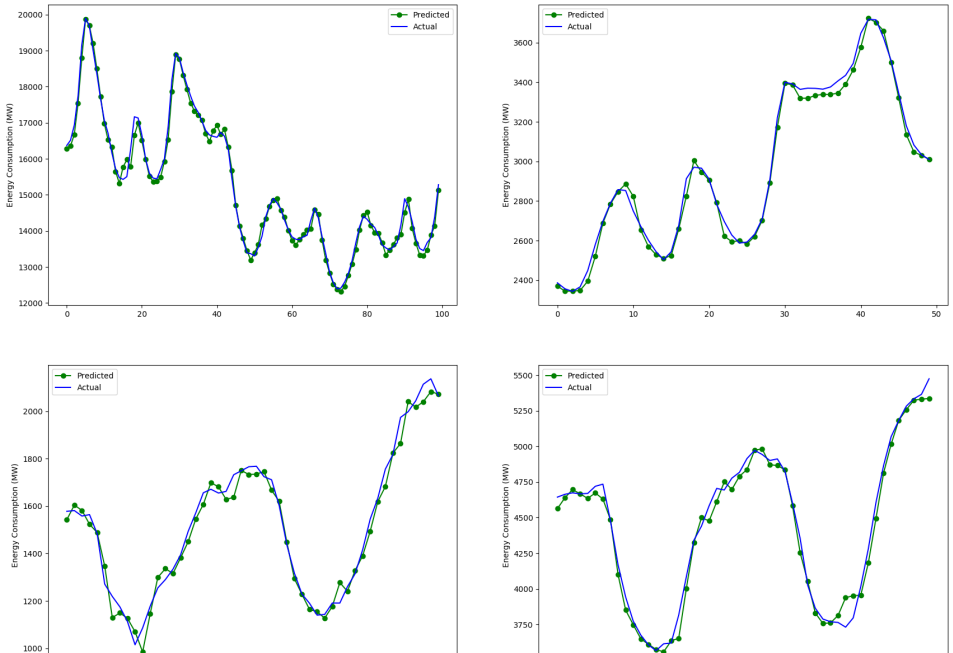
- **Data chunking** refers to the fact that, unlike natural language sentences, time-series data has no particular end. Collecting the data can be a continuous process that can go on and on indefinitely with no particular punctuations. From a machine-learning perspective, that raises the issue how to 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.**



The Transformers Co-Class

- The main purpose of the co-class **Transformers** is to teach the fundamental notions of **self-attention** and **cross-attention**.

- What makes this co-class special — if you don't mind my referring to my own code in that manner — is the conventional wisdom related to this topic in DL: **That it is extremely difficult to train a small enough Transformer based 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 the Preamble, 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 my Transformers co-class is an excercise in seq2seq learning, of the same sort as illustrated by the Seq2SeqLearning co-class, but with no recurrences and no convolutions.

The Transformers Co-Class (contd.)

- The code in Transformers (note the terminating "s") contains the two slightly different classes, one called **TransformerFG** and the other called **TransformerPreLN**. The suffix "FG" stands for "First Generation" and the suffix "PreLN" stands for "Pre Layer Norm". Since some people in the research community claim that they get superior results with the latter, you need to know about it.
- The "FG" implementation of transformers is the same as proposed originally in the now seminal paper by Vaswani et el. "Attention is All You Need". However, the architecture parameters are different in order to create smaller models. And the "PreLN" version incorporates the modifications proposed by Xiong et el.
- You should feel free to compare the result produced by either of the transformer based implementations with those produced by the Seq2SeqLearning based implementation.

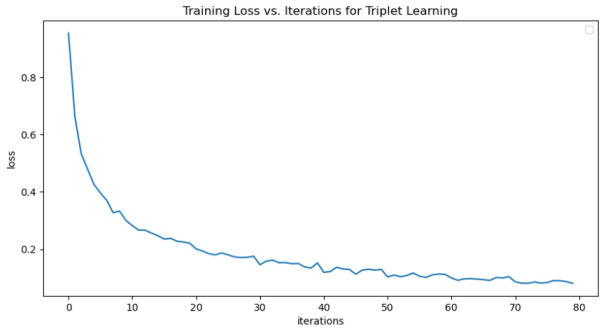
The MetricLearning Co-Class

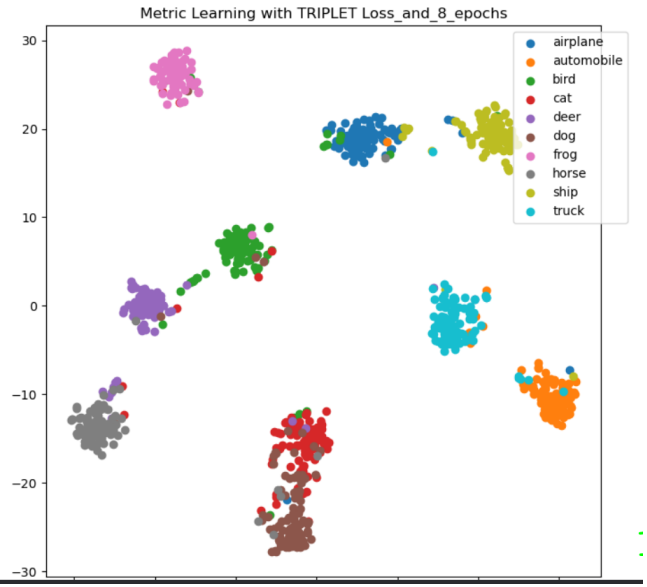
- Metric Learning gives us an entirely different approach to solving image classification problems. Instead of feeding an image directly into the input layer of a CNN, we have a neural network learn what are known as embedding vector representations for each of the training images in such a way that the embeddings for the images that carry the same label are pulled together and the embeddings for the images that carry dissimilar labels are pulled apart to the largest extent possible.
- Two loss functions have emerged as arguably the most popular for the sort of learning mentioned above: Pairwise Contrastive Loss and Triplet Loss. Calculating these losses requires that you reorganizwe your batch instance in the form of Positive Pairs and Negative Pairs.
- A Positive Pair consists of a pair of training images that that carry the same classification label and a Negative Pair of a pair of images whose labels are different.

The MetricLearning Co-Class (contd.)

- Loosely speaking, the Pairwise Contrastive Loss is a measure of the distances between the two emedding vectors in Positive Pairs and negative of the distance between the two embedding vectors in Negative Pairs.
- The Triplet Loss is a bit more sophisticated and its formulation begins with treating one of the two members of a positive pair as the anchor for that pair, thus forming an $(anchor, pos)$ combination. Subsequently, for a given such pair, it seeks all the negative elements in a batch that are within a certain distance of the anchor in relation to the distance between the anchor and the positive. The loss is based on all the triplets thus formed.
- Show below is a 2D visualization of the clustering result produced by the MetricLearning class in DLStudio on the CIFAR-10 dataset. This dataset of 50,0000 small-sized images consists of 10 classes whose labels on the right. More on this result in my Week 9 lecture.

The MetricLearning Co-Class (contd.)





A 2D visualization of the result of Triplet Loss based metric learning applied to the CIFAR-10 dataset. The dataset consists of 50,000 training and 10,000 testing images that belong to 10 different classes whose names are listed at the right edge of the display.

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` co-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 $\times$ 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 two demo scripts based on the code in the `Transformers` co-class:

seq2seq_with_transformerFG.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 but without using recurrences or convolutions.

seq2seq_with_transformerPreLN.py This example script seeks to do the same as the one mentioned above, but with my other implementation of transformers — this one is based on the TransformerPreLN inner class of the Transformers class. In TransformerPreLN, LayerNorm is applied to the input to the self-attention layer and residual connection wraps around both. Similarly, LayerNorm is applied to the input to FFN and the residual connection wraps around both. Similar considerations applied to the decoder side, except we now also have a layer of cross-attention interposed between the self-attention and FFN.

ExamplesMetricLearning

The **ExamplesMetricLearning** subdirectory in the DLStudio distribution contains the following two demo scripts based on the code in the `MetricLearning` co-class:

example_for_pairwise_contrastive_loss.py This script demonstrates Metric Learning using the Pairwise Contrastive Loss. The results are shown on the CIFAR-10 dataset. This script can work with either the pretrained ResNet-50 trunk model or the homebrewed network supplied with the MetricLearning module.

example_for_triplet_loss.py This script demonstrates Metric Learning using the Triplet Loss. As with the previous script, the results are shown on the CIFAR-10 dataset. Again, as with the previous script, the code in the script can work with either the pretrained ResNet-50 trunk model or the homebrewed network supplied with the MetricLearning module.

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-classes.
- 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".