Homework 2 Part 1

An Introduction to Convolutional Neural Networks

11-785: Introduction to Deep Learning (Fall 2020)

Out: **Sept 28th 2020 12:00:00am EST**
Due: **Oct 18th 2020 11:59:59pm EST**

(Last updated: 9/28/20 12:00AM EST)

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to talk with / work with other students on homework assignments
  - You can share ideas but not code, you must submit your own code. All submitted code will be compared against all code submitted this semester and in previous semesters using MOSS.

- **Overview:**
  - **Convolutional Network Components**: Using the Autograd framework to implement the forward and backward passes of a 1D convolutional layer and a flattening layer.
  - **CNNs as Scanning MLPs**: Two questions on converting a linear scanning MLP to a CNN
  - **Implementing a CNN Model**: You know the drill.

- **Directions:**
  - We estimate this assignment may take up to 5 hours; it is *significantly* shorter and simpler than not just HW1P1, but also the previous version of the assignment. We told you future homeworks would now be easier thanks to autograd, and we meant it.
    - If you're truly stuck, post on Piazza or come to OH; don't take more time than you need to.
  - You are required to do this assignment using Python3. Do not use any auto-differentiation toolboxes (PyTorch, TensorFlow, Keras, etc) - you are only permitted and recommended to vectorize your computation using the NumPy library.
  - If you haven't already been doing so, use `pdb` to debug your code and please PLEASE Google your error messages before posting on Piazza.
  - Note that Autolab uses numpy v1.18.1

Introduction

In this assignment, you will continue to develop your own version of PyTorch, which is of course called MyTorch (still a brilliant name; a master stroke. Well done!). In addition, you'll convert two scanning MLPs to CNNs and build a CNN model.

Homework Structure

Below is a list of files that are **directly relevant** to hw2.

IMPORTANT: First, copy the highlighted files/folders from the HW2P1 handout over to the corresponding folders that you used in hw1.

NOTE: We recommend you make a backup of your hw1 files before copying everything over, just in case you break code or want to revert back to an earlier version.

```
handout
├── autograder
│   └── hw2_autograder ..... Copy over this entire folder
│       ├── runner.py
│       ├── test_cnn.py
│       ├── test_conv.py
│       ├── test_scanning.py
│       ├── data
│       ├── weights
│       └── ref_result
├── mytorch ..... MyTorch library
│   ├── nn ..... Neural Net-related files
│   │   ├── activations.py
│   │   ├── conv.py ..... [Question 1] Convolutional layer objects
│   │   ├── functional.py
│   │   ├── linear.py
│   │   └── sequential.py
│   └── tensor.py
├── sandbox.py
├── create_tarball.sh
├── grade.sh
└── hw2 ..... Copy over this entire folder
    ├── hw2.py ..... [Question 3] Building a CNN model
    ├── mlp_scan.py ..... [Question 2] Converting Scanning MLPs to CNNs
    └── stubs.py ..... Contains new code for functional.py that you'll need to copy over
```

Next, copy and paste the following code stubs from hw2/stubs.py into the correct files.

1. Copy `Conv1d(Function)` into `nn/functional.py`.
2. Copy `Tanh(Function)` and `Sigmoid(Function)` into `nn/functional.py`.
3. Copy `Tanh(Module)` and `Sigmoid(Module)` into `nn/activations.py`.

Note that we're giving you the fully completed `Tanh` and `Sigmoid` code as a peace offering after the Great Battle of HW1P1.

0.1 Running/Submitting Code

This section covers how to test code locally and how to create the final submission.

0.1.1 Running Local Autograder

Run the command below to calculate scores and test your code locally.

```
./grade.sh 2
```

If this doesn't work, converting line-endings may help:

```
sudo apt install dos2unix
dos2unix grade.sh
./grade.sh 2
```

If all else fails, you can run the autograder manually with this:

```
python3 ./autograder/hw2_autograder/runner.py
```

0.1.2 Running the Sandbox

We've provided `sandbox.py`: a script to test and easily debug basic operations and autograd.

Note: We will not provide new sandbox methods for this homework. You are required to write your own from now onwards.

```
python3 sandbox.py
```

0.1.3 Submitting to Autolab

Note: You can submit to Autolab even if you're not finished yet. You should do this early and often, as it guarantees you a minimum grade and helps avoid last-minute problems with Autolab.

Run this script to gather the needed files into a `handin.tar` file:

```
./create_tarball.sh
```

If this crashes (with some message about a hw4 folder) use `dos2unix` on this file too.

You can now upload `handin.tar` to Autolab.

---

1 Convolutional Network Components [Total: 60 points]

🎵 Let's get down to business 🎶

🎶 To write a CNN 🎵

1.1 `Conv1d` [Total: 50 points]

If you have not already, make sure to copy over files and code as instructed on the previous page.

In this problem, you will be implementing the forward and backward of a 1-dimensional convolutional layer.

You will only need to implement the 1D version for this course; the 2D version will be part of the optional hw2 bonus, which will be released at the same time as hw3.

Important note: You will be **required** to implement **Conv1d** using a subclass of Function. To repeat, you may not automate backprop for **Conv1d**.

We are requiring this over automating backprop for these reasons:

1. Implementing backprop for CNNs teaches valuable lessons about convolutions and isolating influences of their weights.
2. We hope it'll cement your understanding of autograd, as you do what it would do, in tracing paths and accumulating gradients.
3. The actual Torch implements a subclass of `Function`. It's almost certainly faster and takes MUCH less memory than maintaining dozens of tensors and instructions for each individual operation. You'll see what we mean; picture storing every 2 tensors for every operation of a 100 Conv layer network.
4. Finally, the lec 10 slides contain detailed pseudocode for the forward/backward to make things easier.

Again, if you automate backprop, you will not receive credit for this problem, even if you pass the test on autolab.

1.1.1 `Conv1d.forward()` [20 points]

First, in `nn/conv.py`, complete the `get_conv1d_output_size()` function.

You'll need this function when writing the `Conv1d` forward pass in order to calculate the size of the output data (`output_size`). Because we're not implementing padding or dilation, we can implement this simplified formula[1]:

$$[(\text{input\_size} - \text{kernel\_size})//\text{stride}] + 1$$

Next, in `nn/functional.py`, complete `Conv1d.forward()`. This is the 1D convolutional layer's forward behavior in the computational graph.

The pseudocode for this is in the slides on CNNs.

For explanations of each variable in `Conv1d`, read the comments in both `functional.Conv1d` and `nn.conv.Conv1d`. Also, read the excellent Torch documentation for it.

Finally, note that we've already provided the complete user-facing `Conv1d(Module)` object in `nn/conv.py`.

1.1.2 `Conv1d.backward()` [30 points]

In `nn/functional.py`, complete `Conv1d.backward()`.

This will be uncannily similar to the forward's code (I wonder why... 🤔).

Again, pseudocode in the slides.

1.2 `Flatten.forward()` [10 points]

In `nn/conv.py`, complete `Flatten.forward()`. Note that this is in the `conv.py` file, not in `functional.py`.

This layer is often used between `Conv` and `Linear` layers, in order to squish the high-dim convolutional outputs into a lower-dim shape for the linear layer. For more info, see the torch documentation and the example we provided in the code comments.

Hint: This can be done in one line of code, with no new operations or (horrible, evil) broadcasting needed.

Bigger Hint: Flattening is a subcase of reshaping. `np.prod()` may be useful.

[1] If you want to reuse this function in hw2p2, see the `Conv1d` torch docs for the full formula that includes padding, dilation, etc.

2 Converting Scanning MLPs to CNNs [Total: 20 points]

In these next two problems, you will be converting the weights of a 3-layer scanning MLP into the weights of a 3-layer CNN. The goal is to demonstrate the equivalence between scanning MLPs and CNNs.

Below is a description of the MLP you'll be converting.

Background

Note that you won't need to program or train this MLP anywhere; this is just context and background info you need to solve the upcoming problems. Also, We highly recommend you draw a simple visualization of this. See the lecture slides for examples. We left this exercise to you so you'd fully understand this problem.

The MLP is scanning a single observation of time series data, sized `(1, 128, 24)` (note: batch size of 1). In other words, this observation has 128 time instants, which are each a 24-dimensional vector.

This is the architecture for the MLP:

```
[Flatten(), Linear(8 * 24, 8), ReLU(), Linear(8, 16), ReLU(), Linear(16, 4)]

[Flatten()] # after scanning is completed, all outputs are concatenated then flattened
```

Some architecture notes:

- Assume all bias values are 0. Don't worry about converting or setting bias vectors for this homework; focus on the weights.
- This architecture will be (nominally) identical in both Problem 2.1 and 2.2, although there is a key difference in their weight values, which we'll explain in Problem 2.2.

For each forward pass, the network will receive 8 adjacent vectors at a time: a tensor sized `(1, 8, 24)`. So the network's first scan will be over `training_data[:, 0:8, :]`. The network will then flatten this input into a size `(1, 192)` tensor[2] and then pass it into the first linear layer. Because the final layer has 4 neurons, the network will produce a `(1, 4)` output for each forward pass.

After each forward pass, the MLP will "stride" forward 4 time instants (i.e. the second scan is over `training_data[:, 4:12, :]`), until it scans the last possible vectors given its stride and input layer size. Note that the network will not pad the data anywhere.

This means the network will stride 31 times and produce a concatenated output of size `(1, 4, 31)`[3]. After flattening, the output size will be `(1, 124)` (because $4 \times 31 = 124$).

Summary:

- 3 layer MLP with # neurons `[8, 16, 4]`.
- Data we are scanning is size `(1, 128, 24)`
- Network scans 8 inputs at a time (`(1, 8, 24)` input tensor), producing an output of `(1, 4)`.
- After each forward pass, strides 4 time instants.
- Outputs are concatenated together into tensor sized `(1, 4, 31)`, and after flattening will be size `(1, 124)`

Goal: convert this scanning MLP to a CNN.

[2] `batch_size` first

[3] This is tricky; why 31? You may expect the output size to be 32, because $128/4 = 32$. Try this simple example: draw out an MLP that sees 3 observations at a time, and strides by 2, over an observation of 16 time instants. How many forward passes will it perform? Try to come up with a formula that would tell you the number of outputs. Hint: remember that CNNs and scanning MLPs are equivalent.

2.1 Converting a Simple Scanning MLP [10 Points]

In `hw2/mlp_scan.py`, complete `CNN_SimpleScanningMLP`.

There's only two methods you need to complete: `__init__()` and `init_weights()`. You'll need to finish both before you can run the autograder. Feel free to do them in either order; they're both pretty short (but can be conceptually difficult, as is normal for IDL).

The comments in the code will tell you what to do and what each object does.

As you'll quickly see, the challenge will be in determining what numbers to use when initializing the `Conv1d` layers.

Here are a lot of hints.

General Tips:

- Make sure to not add nodes to the comp graph. You should know what this means by now.
- Tensor settings matter; autograder will check for them.
- You're allowed to read the autograder files (obviously we haven't given away answers, but it may help you get unstuck).

Pseudocode for `init_weights()`:

1. Reshape each Linear weight matrix into `(out_channel, kernel_size, in_channel)`.
2. Transpose the axes back into these shapes: `(out_channel, in_channel, kernel_size)`. (Note: `.T` will not work for this step)
3. Set each array as the params of each conv layer.

Again, don't worry about biases for this problem AND the next one. If you ask about biases at OH or Piazza, we'll know you didn't read the writeup properly and you'll instantly lose a lot of street cred[4].

[4]The cool kids will hear about it and won't let you sit at their table anymore.

2.2 Converting a Distributed Scanning MLP [10 points]

This problem is very similar to the previous, except the MLP is now a **shared-parameter network** that captures a distributed representation of the input.

All background/context is still the same (data size, stride size, network components, etc). Even the overall architecture of the MLP is identical. This includes the number of neurons in each layer (still `[8, 16, 4]`).

But the only difference is that **many of the neurons within each layer share parameters with each other**.

We've illustrated the parameter-sharing pattern below.

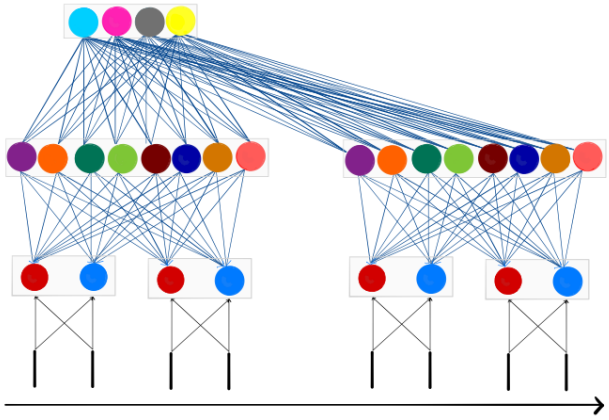

Figure 1: The Distributed Scanning MLP network architecture

Each time instant t is a single line on the bottom. Each circle is a neuron, and each edge is a connection. So "Layer 1" would be the edges between the inputs and the neurons just above the inputs.

Key point: within each layer, the neurons with the same color share the same weights. Note that this coloring scheme only applies within each layer; neurons with the same/similar colors in other layers do **not** share weights.

If this illustration is unfamiliar or confusing, please rewatch the explanations of distributed scanning MLPs in the lectures. Verbal explanations make this much clearer.

Try to picture what the shape of each weight matrix would look like, especially in comparison to those in the previous problem.

In `hw2/mlp_scan.py`, complete the class `CNN_DistributedScanningMLP`.

Most of the code will be identical; you need to complete the same two methods as the previous problem. But there are a few differences, described below:

1. In `init_weights()`, you'll need to first somehow slice the weight matrix(s) to account for the shared weights. Afterwards, the rest of the code will be the same.
2. In `__init__()`, the model components will be identical, but the params of each `Conv1d` (i.e. kernel_size, stride, etc) will not be.

3 Build a CNN model [Total: 15 points]

Finally, in `hw2/hw2.py`, implement a CNN model.

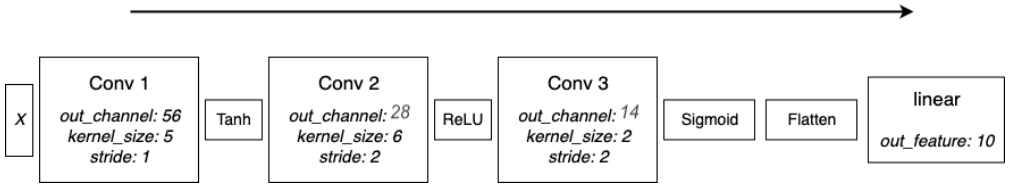

Figure 2: CNN Architecture to implement.

Remember that we provided you the `Tanh` and `Sigmoid` code; if you haven't already, see earlier instructions for copying and pasting them in.

Previously for this problem, students would have to implement step, forward, backward, etc. But thanks to your hard work in this homework and especially in HW1P1, all you have to do is initialize the layer objects like in the architecture pictured above, and you're done.

The only tedious thing is calculating the input size of the final `Linear` layer.

Why is this tedious? Notice that the input to the final `Linear` layer is flattened into size `(batch_size, conv3_num_channels * conv3_output_size)`. The batch size and the number of channels for conv3 are obvious, but what's that second term in the second dimension: the output size?

The network input `x` will be shaped `(batch_size, num_input_channels, input_width)`, where `input_width=60`. But remember from your Conv implementation that each Conv forward will change the size of the data's final dimension (`input_width -> output_width`). So the size of this final dimension will change every time it passes through a Conv layer. Here, it will change 3 times. So you'll need to calculate this in order to initialize the final linear layer properly.

On the bottom of the same file, complete `get_final_conv_output_size()`. Given the initial `input_size`, this method should iterate through the layers, check if the layer is a convolutional layer, and update the current size of the final dimension using the `get_conv1d_output_size()` function you implemented earlier. This method will take the current `output_size` and the params of the layer to calculate the next `output_size`. After iterating through all the layers, you'll have the `output_size` for the final `Conv1d` layer.

(Above is essentially pseudocode for this method; don't miss it.)

We ask you to implement this because you may want to modify it and use it for HW2P2.

Given any CNN-based architecture, this function should return the correct output size of the final layer before flattening. While you can assume only `Conv1d` layers will matter for now, if you do reuse this you may need to modify it to account for other layer types.

Great work as usual 💃🕺, and all the best on HW2P2 🙌!