Assignment:

Part 1: Getting Started with PyTorch

PyTorch can run on your machine's CPU, but we need the servers to enable GPU usage. Think of PyTorch as a fancy version of NumPy (in fact, they're syntactically almost identical) that can also (1) run on GPUs, and (2) automatically compute gradients for you (more on this later).

No Python code needs to be submitted for this one, but you'll be writing and running short snippets of code for it. You'll be placing answers in the file `torch_snippets_and_errors_name_erp.txt`. The first five lines are already filled to get you started.

Instructions.

For each of the following bugs/errors/behaviors, write a snippet of code that does it, then write a line in `torch_snippets_and_errors.txt` saying what the error message is (if there is an error message) or what the behavior is (if there is no error message). There are essentially three possible outcomes of each snippet:

1. **It runs without error and produces the expected output**. In this case, you should write a line in `torch_snippets_and_errors.txt` saying what the correct and expected behavior was.
2. **It runs without error but produces unexpected output.** In this case, you should write a line in `torch_snippets_and_errors.txt` saying what you expected and what the unexpected outcome was.

3. **It produces an error message.** In this case, you should write a line in `torch_snippets_and_errors.txt` with the last line of the error message.

Snippets to Write

1. Add two PyTorch tensors of shape `(3, 4)` to a PyTorch tensor of shape `(4, 3)`. Example:

   torch.randn(3, 4) + torch.randn(4, 3)

2. Add a PyTorch tensor of shape `(3, 4)` to a PyTorch tensor of shape `(4,)`.

3. Add a PyTorch tensor of shape `(3, 4)` to a PyTorch tensor of shape `(4, 1)`.

4. Add a PyTorch tensor of dtype `torch.float32` to a PyTorch tensor of dtype `torch.float64`.

5. Create a tensor of all zeros with dtype `torch.uint8`, then try to assign it a value of `-3.14`.

6. Add two PyTorch tensors of the same shape and dtype but on different devices (e.g., CPU and GPU).

7. Generate random data for a scatter plot and plot it using `matplotlib.pyplot.scatter()`.

8. Repeat #7 but place the tensors on the GPU.

9. Repeat #7 but set the `requires_grad` attribute of the tensors to `True`.

10. Repeat #9 but use `plt.scatter(x.detach(), y.detach())`.

11. Create a PyTorch tensor and increment it in-place using `+=`.

12. Repeat #11 but with a tensor that has `requires_grad=True`.

13. Compute the derivative `dy/dx` where `y = x^2` at `x = 100` using `torch.autograd.grad()`.

14. Repeat #13 but with `x` as a tensor containing multiple elements.

15. Repeat #13 but with `y = torch.log(torch.exp(x))`.

16. Attempt to create a ~20GB array on CUDA using a tensor of shape `(5000, 1000, 1000, dtype=torch.float32, device='cuda')`.

17. Load an image from the CIFAR-10 dataset and attempt to display it using `plt.imshow(img)`. Record the error.

```
dataset = torchvision.datasets.CIFAR10(
   replace_with_directory_name,train=True,
   transform=transform,
   download=True
)
```

18. Fix the error from #17 by permuting the tensor's axes and displaying it correctly using `plt.imshow(img.permute(1, 2, 0))`.

Collaboration and Generative AI Disclosure

Did you collaborate with anyone? Did you use any Generative AI tools? Briefly explain what you did in the `collaboration-disclosure.txt` file.

Part 2: Building your own Pytorch Model

The goal of this assignment is to give you hands-on practice with the following concepts:

- Handling image datasets in PyTorch
- Training and evaluation of a simple image classification model

i) Datasets and DataLoaders

In this part of the assignment, you'll briefly poke around the MNIST and CIFAR-10 datasets using the torchvision library. You'll write two functions whose job it is to simply print out some basic information about a given dataset.

In PyTorch, a Dataset class is a simple way to represent a collection of data samples. Custom Datasets are super easy to implement; essentially, any object that subclasses

torch.utils.data.Dataset and provides an implementation of the __len__ and __getitem__ methods can be used as a dataset. For instance, you could write a custom Dataset class to load data from a CSV file, or from a directory of images, or from a database. As easy as that is, we're going to do something even easier and use some of the built-in datasets that PyTorch (specifically the torchvision library) provides.

Let's say you call ds = MyDataset(). Then, you could access the i-th sample in the dataset by

calling ds[i], which would call the __getitem__ method of the MyDataset class. However, this isn't how we use datasets in practice, for a few reasons:

1. we'll want to get an entire batch of samples at a time,
2. it's useful to shuffle the dataset so that we don't always see the samples in the same order, and
3. if the __getitem__ method is slow (e.g., if it's reading from disk), we'd ideally run the slow i/o operations for the next batch in the background while the model is training on the current batch.

All of this is handled by the DataLoaderclass in PyTorch. The DataLoaderclass wraps a Dataset object and provides an iterator that returns batches of samples. It also handles shuffling, batching, and (if you set num_workers to a value greater than 0) running the slow __getitem__ method in parallel with the model training. It looks like this:

dl = DataLoader(ds, batch_size=32, shuffle=True, num_workers=4,

pin_memory=True)

The pin_memory argument is useful if you're using a GPU and can speed up data transfer to the GPU.

Instructions

Download the Datasets

in config.py you have:

```
DATA_ROOT = Path("/path/to/your/datasets")
```

Unfortunately, You have to download manually the MNIST and CIFAR-10 datasets into sub-folders under DATA_ROOT (IBA servers cannot upload that much for assignments :( )

Below are commands for downloading the datasets manually:

For CIFAR-10:

```
# Create the directory for CIFAR-10 (if not already existing)

mkdir -p /path/to/your/datasets/cifar10


# Download the CIFAR-10 tarball into the cifar10 folder

wget https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz -P
/path/to/your/datasets/cifar10


# Extract the contents

tar -xvzf /path/to/your/datasets/cifar10/cifar-10-python.tar.gz -C
/path/to/your/datasets/cifar10


# (Optionally) Rename the extracted folder to simply "cifar10" if needed.

mv /path/to/your/datasets/cifar10/cifar-10-batches-py /path/to/your/datasets/cifar10
```

For MNIST:

```
# Create the directory for MNIST

mkdir -p /path/to/your/datasets/mnist


# Download the MNIST dataset files into the mnist folder

wget http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz -P
/path/to/your/datasets/mnist

wget http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz -P
/path/to/your/datasets/mnist

wget http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz -P
/path/to/your/datasets/mnist

wget http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz -P
/path/to/your/datasets/mnist


# Unzip the files

gunzip /path/to/your/datasets/mnist/*.gz
```

Instructions:

Fill in the missing code in datasets.py to print out useful diagnostic information about a dataset or data loader. The string formatting is done for you, you just need to inspect the properties of the

Datasetand DataLoaderobject.

Run the datasets.py script with --dataset mnist or --dataset cifar10 to see some information about each one.

Example output

Output from running python datasets.py --dataset mnist:

Dataset with 60000 samples, 10 classes, image shape torch.Size([1, 28, 28]) and dtype torch.float32.

DataLoader with 60000 total samples split across 1875 batches of size 32. Batch shape is torch.Size([32,

1, 28, 28]).

ii) Logistic Regression

In this part of the assignment, you'll train a simple logistic regression model to classify small hand-written digits (MNIST dataset) and small color images (CIFAR-10 dataset). This is one of the simplest

model types we can use for image classification. We're using it here so that

1. You can get experience with PyTorch's image handling and training process without the added complexity of big fancy models, and
2. You get a good sense of the baseline performance of a simple model on these datasets.

Instructions

1. Fill in the missing code in models.py. You must implement the __init__ and forward methods of the LogisticRegression class. The __init__ method should initialize the

weights and biases of the model, and the forward method should compute the output of the model given an input tensor. Logistic regression from data X to labels y is given by the equation

$p(y) = \sigma(z)$, where $z = WX + b$ and σ is the softmax function. This maps from an input vector X to a probability distribution over class labels. Importantly, a single nn.Linear layer accomplishes the $WX + b$ part of the equation, and we're actually going to return z from the forward method, so the model itself is a bit simpler than the equation might suggest.

2. Fill in the missing code in train.py, and study the code that is already provided for you there.

Your primary task is to implement the train_single_epoch function, which is called by the train function. The train_single_epoch function should train the model for one epoch (i.e., one loop over the data loader). It should log the accuracy and loss of the model

on the training set in the Tensorboard SummaryWriter. Use the evaluate() function as a reference. The primary differences between train_single_epoch and evaluate are that

- train_single_epoch should call model.train() at the start rather than model.eval()
- it should call optimizer.zero_grad() at the start of each batch

- it should call loss.backward() and optimizer.step() at the end of each batch
- it should log the loss and accuracy to the SummaryWriter every batch
- it should increment step each batch
- it should not have a torch.no_grad() block (we do want grads while training!)

3. Run the training script at least four times: twice for MNIST and twice for CIFAR-10, using at least two different learning rates. You can use the --dataset and --lr flags to specify the dataset and learning rate, respectively. For instance, to train on CIFAR-10 with a learning rate of 0.01, you would run python train.py --dataset cifar10 --lr 0.01. You can also specify the number of epochs to train for with the --epochs flag. Remember, the goal at this stage is not to make the best or fanciest model. It's to get comfortable with the training process and to get a sense of how well a simple model can do on these datasets.

iii) Tensorboard (Bonus points)

Tensorboard is a great tool for visualizing how your model training is progressing. It works by writing logs to a specified directory, which you can then view as interactive graphs in a browser window once you start the tensorboard server.

You can install TensorBoard using pip. For example, run the following command in your virtual environment:

```
pip install tensorboard
```

For PyTorch, you might also consider installing the torch-tensorboard package:

```
pip install torch-tensorboard
```

Once installed, you can log training information and go where its being logged and then launch TensorBoard with a command like:

```
tensorboard --bind_all --logdir=logs --port=6006
```

Then open a browser at http://127.0.0.1:6006 to see the Tensorboard UI.

Once you've successfully trained at least 2 models on each dataset with different hyperparameters, take a screenshot of the Tensorboard UI and include it in your submission in the images directory.

Expected output

Here's what a functioning tensorboard UI might look like after training two MNIST models with different learning rates:

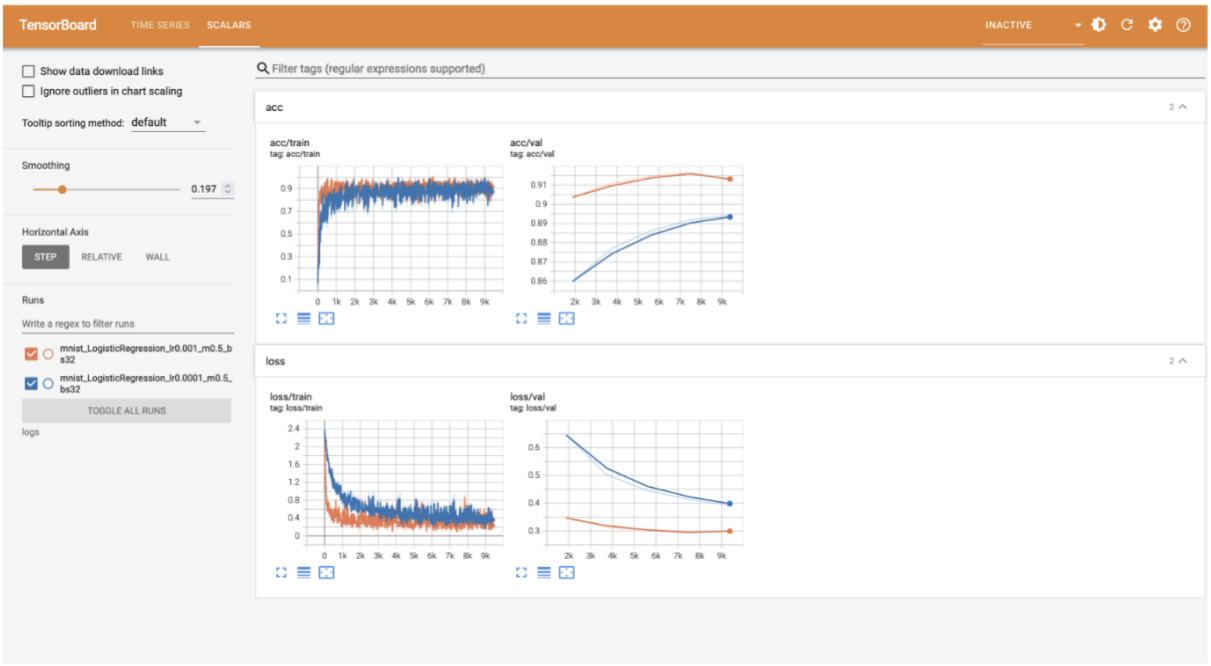

Figure 1: Example tensorboard screenshot

Your screenshot(s) must also include loss and accuracy plots for logistic regression trained on the CIFAR-10 dataset.

Collaboration and Generative AI disclosure

Did you collaborate with anyone? Did you use any Generative AI tools? Briefly explain what you did in the collaboration-disclosure.txt file.