CS 197 Pset 3

Christy Jestin

October 12, 2022

1 Part 1: PyTorch Lightning

1. We can split first by writing a new, simple Dataset class:

```
# use identity lambda function to prevent conversion to tensor
train_dataset = CIFAR10(root = DATASET_PATH, train = True, transform = lambda x: x, download = True)
test_set = CIFAR10(root = DATASET_PATH, train = False, transform = test_transform, download = True)
```

✓ 26.6s

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to data/cifar-10-python.tar.gz


Extracting data/cifar-10-python.tar.gz to data/
Files already downloaded and verified
```

```
split = np.concatenate((np.zeros(45000), np.ones(5000)))
np.random.shuffle(split)
train_indices = np.where(split == 0)[0]
val_indices = np.where(split == 1)[0]

class SplitDataset(torch_data.Dataset):
    def __init__(self, dataset, indices, transform):
        tuple_transform = lambda x, y: (transform(x), y)
        self.data = [tuple_transform(*dataset[i]) for i in indices]

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        return self.data[index]

train_set = SplitDataset(train_dataset, train_indices, train_transform)
val_set = SplitDataset(train_dataset, val_indices, test_transform)
```

2. The code errors without Line 6 and says that (3, 8, 386) is an invalid shape for image data. This is because PyPlot expects the three color channels to be in the third dimension which is why line 6 is needed to permute the dimensions.

3. a. Linear is a matrix multiplication or linear transformation layer.
   b. GELU is a nonlinear activation function that is similar to RELU but doesn't have a corner.
   c. Dropout is a layer that sets values to 0 with some probability and scales the remaining values such that the sum of values is maintained. Dropout is meant to encourage learning stronger features.
   d. LayerNorm normalizes the inputs over some number of dimensions and then scales the values by a learned weight before adding a learned bias. It builds on batch normalization and is meant to reduce training time.

e. Multihead Attention is a technique to extract different information about the relationship between inputs in a sequence. In this case, we are attending the image to itself, and each head theoretically represents a different aspect of how parts of the image relate to each other.

4. Line 20 passes the input into an attention mechanism. Attention makes use of three vectors representing a query, a key, and a value. These vectors are computed by multiplying separate query, key, and value weight matrices with the input vector to produce three separate vectors. For a specific token, we compute the dot product of its query vector with the key vector of every token and apply a softmax to these dot products to yield weights that sum to 1. These weights are then used to sum up the value vectors of all tokens and produce a single attention output for that token. With multihead attention, we simply compute these outputs multiple times with the same input embeddings but different query, key, and value weight matrices. In this case, the image is attending to itself, so we use the same input for all three pieces: query, key, and value. Additionally, we only want to use the output of attention (and we don't need the weight matrices), so we index into the first element of the output from self.attn. Finally, we want to hold onto information from the original input, so we simply add to x instead of replacing x.

5. Lines 30-33 are randomly initializing a learnable class token and a learnable set of positional embeddings. Because the transformer deals with all inputs at once instead of in sequence like an RNN, the positional embeddings help provide needed context about where a patch is within the image. The class token is shared amongst all inputs: it is prepended to each sequence of patches and serves as a place to "write the answer down" i.e. only the value in this first position is kept after the input is passed through the attention blocks, so this index is meant to be a malleable placeholder where the network can fill in a representation of the image. This setup is meant to mimic language transformers.

6. In line 46, we utilize the fact that we prepended every sequence with the class token and read the value of this first position to get a representation of the image as a whole.

7. The fast dev run argument sets the number of batches that the trainer should run to check for errors. In this case, it'll run on only 5 batches while both fitting and testing. The default root dir argument tells the trainer where to save logs and checkpoints (unless told otherwise by other arguments).

2 Part 2: PyTorch

1. a. The output will be Weight: True, Gradient: False since the weights haven't been modified or used in any calculations yet.

   b. The output will be Weight: True, Gradient: False since zeroing does nothing when the gradients don't exist yet.

   c. The output will be Weight: True, Gradient: False since there has been no back-propagation or stepping from the optimizer despite the loss being computed.

   d. The output will be Weight: True, Gradient: True because the gradient has now been back-propagated.

   e. The output will be Weight: False, Gradient: True because the gradient has been used to update the value of the weights.

2. My implementation is:

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

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.layer1 = nn.Linear(1024, 100)
        self.layer2 = nn.Linear(100, 10)

    def forward(self, x):
        return self.layer2(F.relu(self.layer1(torch.flatten(x))))
```

✓ 0.3s

3. I ran it on a randomly generated input here:

```python
n = Net()
a = torch.rand(32, 32)
n(a).shape
```

✓ 0.6s

```
torch.Size([10])
```

4. The length should be 2 since RELU doesn't have any parameters.
5. This should be referring to the second linear layer and have size 100 x 10. It turns out I forgot that the linear layers can have bias parameters which is also why the answer to the last question was 4 instead of 2. Also the Linear module implementation does not transpose the weight matrix before doing multiplication with vectors, so the size should've been 10 x 100 even if it was a weight matrix.
6. My implementation is:

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

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.linear1 = nn.Linear(400, 120)
        self.linear2 = nn.Linear(120, 84)
        # think this would normally be handcrafted
        self.rbf = nn.Parameter(torch.rand(10, 84))


    def forward(self, x):
        # add dimension for single color channel -- needed for Conv2d
        x = torch.unsqueeze(x, 0)
        x = F.max_pool2d(self.conv1(x), 2)
        x = F.max_pool2d(self.conv2(x), 2)
        # fully connected layers
        x = self.linear2(F.relu(self.linear1(torch.flatten(x))))
        # gaussian connections
        return ((self.rbf - x) ** 2).sum(dim = 1)
```

✓ 0.4s

7. My modified implementation is:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(1, 6, 5),
            nn.MaxPool2d(2),
            nn.Conv2d(6, 16, 5),
            nn.MaxPool2d(2),
            nn.Flatten(0, 2),
            nn.Linear(400, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
        )
        # think this would normally be handcrafted
        self.rbf = nn.Parameter(torch.rand(10, 84))


    def forward(self, x):
        # add dimension for single color channel -- needed for Conv2d
        x = self.model(torch.unsqueeze(x, 0))
        # gaussian connections
        return ((self.rbf - x) ** 2).sum(dim = 1)
```

✓ 0.4s

```
n = Net()
a = torch.rand(32, 32)
n(a).shape
```

✓ 0.8s

```
torch.Size([10])
```

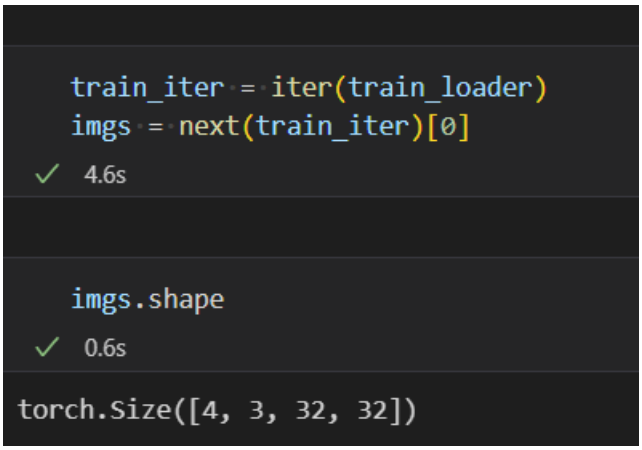
8. You need to increase the number of input channels in the second convolutional layer to match the

number of output channels in the first convolutional layer.

3 Part 3: Classifier Training

1. The train and test datasets and dataloaders are being set up improperly. The train argument for the CIFAR10 class should be True for the train dataset and False for test instead of the other way around. As is, the wrong data will be used for testing and training, which is a problem because the training data is intentionally much larger to allow the model to learn while the testing dataset is smaller since it is just meant for evaluation. In addition, we can see that neither the train nor test dataloader has the shuffle argument set to True. This should actually be True for train and False for test. Without this being set to True, the model may adjust a certain way each time it sees the same batch in the same order during training. Shuffling is unnecessary for testing since we're only considering cumulative results (and not back-propagating in batches), so shuffling will have no affect.

2. This retrieves 4 random images:



3. This training loop never calls the zero grad method on the optimizer which means the gradients will accumulate instead of resetting for each batch. The solution is adding in a line with zero grad either at the beginning of the loop or after we call the step method on the optimizer. The variable loss is a zero dimensional torch tensor, but it is being added to the scalar running loss variable. This will turn running loss into a zero dimensional torch tensor as well. We can fix this by calling the item method on loss to retrieve just the scalar value. Finally the break keyword is used within the if loop which actually breaks the parent for loop. This means that we'll only go through 2000 mini-batches a single time before the code stops running. We can fix this by simply removing that line.

4. Again we are adding a zero dimensional torch tensor to a scalar in the line that beings with correct plus equals. This turns correct into a tensor as well, and we can fix it by calling the item method on the tensor sum. This won't change the outputs at all, but it is inefficient to compute the test performance this way: we're not updating the model at all during these computations, so we have no use for gradients, and we can fix this inefficiency by wrapping the whole loop in a "with torch.no_grad():" call.

4 Part 4: Project Groups

I'll be working with Kostas Tingos.