A brief introduction to PyTorch

IACV 2019
Prerequisites

Notion of multidimensional arrays (tensors)

NumPy basics

Perceptron (Fully-connected layer)

Convolution (Correlation filter)
PyTorch vs TensorFlow

- Both are powerful tools
- A matter of preference
- **PyTorch** is becoming a major tool in deep learning community
- PyTorch is easier for toy and research projects
- Clear “pythonic” API
- **TensorFlow** is great for deploying models in production
- TensorFlow graph definition API may incur engineering debt
- More than one way to do things
PyTorch tensor conventions

Images: 4D tensors with shape *(batch, channels, height, width)*

Labels: 1D tensors with shape *(batch, )*  

Calling conventions: `torch.sum(a)` vs `a.sum()`
torch.nn.Dataset

A base class to wrap any dataset.

Each split (train, test) must be wrapped separately.

Two methods to implement:

- __len__: return length of dataset
- __getitem__: return an item, in our case - a tuple (image, label)

class MyArrayDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return self.X.shape[0]

    def __getitem__(self, index):
        return self.X[index], self.y[index]
torch.nn.DataLoader

A wrapper around a class, derived from Dataset class

- Performs synchronous shuffling of the dataset samples
- Assembles samples into minibatches

```python
loader = DataLoader(dataset, batch_size, shuffle, drop_last)
```

- Batch size - how many samples will be returned in one iteration
- Shuffle - whether to randomize samples, or address dataset in order
- Drop last - whether to return incomplete batch upon reaching dataset end
torch.nn.Module

- A parent class to all building blocks:
  - torch.nn.Linear - a fully connected layer
  - torch.nn.Conv2d - convolutional filter for 2D images
  - torch.nn.MaxPool2d - maximum pooling for 2D images (no learnable parameters)
  - torch.nn.ReLU - elementwise activation function (no learnable parameters)

- Also a parent class to compositions:
  - torch.nn.Sequential(block1, block2, ...) - chain of modules

- Also a parent class to your network:
  - Any building block with trainable parameters (convolution weights, batchnorm statistics) must be defined in class construction: def __init__(self)
  - Data flow goes in def forward(self, *input)

- forward function must compute everything using torch.* tensor ops
- Backward derivatives are computed automagically (*)
torch.nn.Conv2d

nn.Conv2d: in_channels, out_channels, kernel_size, padding, bias

in_channels=1, out_channels=1, kernel_size=3, padding=0

in_channels=3, out_channels=10, kernel_size=5, padding=2

Images courtesy of Arden Dertat https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2
torch.nn.BatchNorm2d

**nn.BatchNorm2d**: `num_features`

- Mutually exclusive with bias in convolution
- Number of features must be same as the number of output features in preceding conv2d

```
Input: Values of x over a mini-batch: \( B = \{x_1...m\} \);
Parameters to be learned: \( \gamma, \beta \)
Output: \( \{y_i = \text{BN}_{\gamma,\beta}(x_i)\} \)
```

\[
\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \\
\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \\
\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\
y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)
\]
torch.nn.MaxPool2d

nn.MaxPool2d: kernel_size, stride

max pool with 2x2 window and stride 2

Images courtesy of Arden Dertat https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2
torch.nn.ReLU

**nn.ReLU**: no parameters

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>20</td>
<td>-10</td>
<td>35</td>
</tr>
<tr>
<td>18</td>
<td>-110</td>
<td>25</td>
<td>100</td>
</tr>
<tr>
<td>20</td>
<td>-15</td>
<td>25</td>
<td>-10</td>
</tr>
<tr>
<td>101</td>
<td>75</td>
<td>18</td>
<td>23</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>20</td>
<td>0</td>
<td>35</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>25</td>
<td>100</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>75</td>
<td>18</td>
<td>23</td>
</tr>
</tbody>
</table>
torch.nn.CrossEntropyLoss

**nn.CrossEntropyLoss**: no parameters; $\text{loss}(y\_hat, y)$
- Includes SoftMax activation for convenience
- Argument order matters!

$$D(\hat{y}, y) = - \sum_j y_j \ln \hat{y}_j$$

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad \text{for } j = 1, \ldots, K$$

$$CE = - \sum_x p(x) \log q(x)$$
torch.nn.Optimizer

**optim.SGD**: weights, lr
CNN Model with one convolutional layer

class CNN(nn.Module):
    def __init__(self, num_out_classes):
        super(CNN, self).__init__()
        self.op_conv = nn.Conv2d(
            in_channels=3, out_channels=20, \
            kernel_size=3, padding=1, bias=False)
        self.op_batchnorm = nn.BatchNorm2d(num_features=20)
        self.op_activation = nn.ReLU()

    def forward(self, x):
        x = self.op_conv(x)
        x = self.op_batchnorm(x)
        x = self.op_activation(x)
        return x
Typical PyTorch training loop

```python
for epoch in range(num_epochs):
    for batch in loader_train:
        X_batch, y_batch = batch
        optimizer.zero_grad()  # zero the gradients
        y_hat_batch = cnn(X_batch)  # forward pass
        loss = criterion(y_hat_batch, y_batch)
        loss.backward()  # backward pass
        optimizer.step()  # update parameters
```
Keep in mind

- Create separate cells to print tensors content (also from different cells)
- When in doubt, `print(tensor.shape)`
- Tensor operations can be invoked two ways: `tensor.sum()`
  - `torch.sum(tensor)`
- Construct network from building blocks:
  - `torch.nn.[Linear, Conv2d, MaxPool2d, ReLU]`
- Before running jupyter notebook, activate `iacv` conda environment:
  - `> source activate iacv`
Cconda environment

- Modern frameworks have hundreds of dependencies
- Different packages may depend on different versions of the same other packages
- An environment encapsulates all dependencies and allows to work on multiple projects from the same machine by just switching environments
- Lab computers have iacv environment pre-installed
- To work on a personal computer, install miniconda, download environment.yml from Extra material folder of the course site, and run shell command:

> conda env create -f environment.yml