Introduction to PyTorch

Computer Vision
Assignment 1

Ingredients for training a neural network

Architecture
Loss

Datasets

Optimizers
Schedulers

Hyperparameters

Architecture and Loss

1. Architecture
    a. Multi-layer perceptron
    b. Convolutional network
    c. *Capsule networks*
    d. *Transformers, graph neural networks*
2. Losses
    a. Traditional regression / classification losses
    b. Task-specific losses
    c. *Find a differentiable proxy to a non-differentiable objective (e.g., softening)*

Binary 0/1 -> Use a sigmoid function to make it differentiable

Architecture - nn.Module

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

```
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))
```

- Define the layers of the network
- Each layer is another module

- Define the forward pass
- As long as it uses only torch operations with predefined gradients, no need for backward

predefined modules and operations

Basic Modules - Linear Layer

https://pytorch.org/docs/stable/generated/torch.nn.Linear

CLASS `torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

Basic Modules - Non-linearity (ReLU)

https://pytorch.org/docs/stable/generated/torch.nn.ReLU

CLASS `torch.nn.ReLU(inplace=False)` [SOURCE]

Applies the rectified linear unit function element-wise:

$$\mathrm{ReLU}(x) = (x)^+ = \max(0, x)$$

Basic Modules - 2D Convolutional Layer

https://pytorch.org/docs/stable/generated/torch.nn.Conv2d

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

Basic Modules - 2D Max Pooling Layer

https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d

CLASS `torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)` [SOURCE]

Applies a 2D max pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points. `dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

Basic Modules - Sequential

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

```
# Using Sequential to create a small model. When `model` is run,
# input will first be passed to `Conv2d(1,20,5)`. The output of
# `Conv2d(1,20,5)` will be used as the input to the first
# `ReLU`; the output of the first `ReLU` will become the input
# for `Conv2d(20,64,5)`. Finally, the output of
# `Conv2d(20,64,5)` will be used as input to the second `ReLU`
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )
```

Loss

https://pytorch.org/docs/stable/generated/torch.nn.functional.binary_cross_entropy.html

```
torch.nn.functional.binary_cross_entropy(input, target, weight=None,
size_average=None, reduce=None, reduction='mean')
```
[SOURCE]

Function that measures the Binary Cross Entropy between the target and the output.

See `BCELoss` for details.

https://pytorch.org/docs/stable/generated/torch.nn.functional.cross_entropy.html

```
torch.nn.functional.cross_entropy(input, target, weight=None,
size_average=None, ignore_index=-100, reduce=None, reduction='mean')
```
[SOURCE]

This criterion combines *log_softmax* and *nll_loss* in a single function.

See `CrossEntropyLoss` for details.

Datasets

- Training / validation / test samples with annotations
- Preprocessing / augmentation technique
- *Improve / accelerate ground truth generation*
- *Automatic data cleaning*
- *Reduce quantity of data required*

Datasets

https://pytorch.org/docs/stable/data.html?highlight=dataset#torch.utils.data.Dataset

```python
class Simple2DDataset(Dataset):
    def __init__(self, split='train'):
        super().__init__()
        assert split in ['train', 'valid'], f'Split parameters "{split}" must be either "train" or "valid"'
        # Read either train or validation data from disk based on split parameter using np.load.
        # Data is located in the folder "data".
        # Hint: you can use os.path.join to obtain a path in a subfolder.
        # Save samples and annotations to class members self.samples and self.annotations respectively.
        # Samples should be an Nx2 numpy array. Annotations should be Nx1.
        raise NotImplementedError()

    def __len__(self):
        # Returns the number of samples in the dataset.
        return self.samples.shape[0]

    def __getitem__(self, idx):
        # Returns the sample and annotation with index idx.
        raise NotImplementedError()
        sample = None
        annotation = None

        # Convert to tensor.
        return {
            'input': torch.from_numpy(sample).float(),
            'annotation': torch.from_numpy(annotation[np.newaxis]).float()
        }
```

- Stores parameters relevant to dataset generation
- Can be used to read dataset or at least index from disk

- Returns the number of samples in the dataset

- Returns one sample from the dataset (and its annotation if available)
- Can be used for on-the-fly data augmentation

Datasets - Dataloader

https://pytorch.org/docs/stable/data.html?highlight=dataloader#torch.utils.data.DataLoader

- Launches multiple dataset instances in parallel (based on num_workers)
- Combines the data from the parallel dataset into batches of size batch_size
- This also results in parallel data augmentation

CLASS `torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, sampler=None, batch_sampler=None, num_workers=0, collate_fn=None, pin_memory=False, drop_last=False, timeout=0, worker_init_fn=None, multiprocessing_context=None, generator=None, *, prefetch_factor=2, persistent_workers=False)` [SOURCE]

Data loader. Combines a dataset and a sampler, and provides an iterable over the given dataset.

The `DataLoader` supports both map-style and iterable-style datasets with single- or multi-process loading, customizing loading order and optional automatic batching (collation) and memory pinning.

See `torch.utils.data` documentation page for more details.

Optimizers and learning rate schedulers

- Stochastic gradient descent (SGD)
- Adam and variants (default choice for most recent SOTA results)
- *New optimizers with better convergence properties*
- *Provide guarantees on existing ones*

Optimizers

https://pytorch.org/docs/stable/generated/torch.optim.SGD.html?highlight=sgd#torch.optim.SGD

CLASS torch.optim.SGD(*params, lr=<required parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False*) [SOURCE]

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

https://pytorch.org/docs/stable/generated/torch.optim.Adam.html?highlight=adam#torch.optim.Adam

CLASS torch.optim.Adam(*params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False*) [SOURCE]

Implements Adam algorithm.

It has been proposed in Adam: A Method for Stochastic Optimization. The implementation of the L2 penalty follows changes proposed in Decoupled Weight Decay Regularization.

Hyperparameter Tuning

- Number of epochs / gradient descent steps
- Preprocessing / augmentation parameters
- Optimizer learning rate and other parameters
- *Study of variability / reproducibility based on parameters*
- *Genetic algorithm for finding the best hyperparameters*

`torch.manual_seed(3407)` is all you need: On the influence of random seeds in deep learning architectures for computer vision

David Picard DAVID.PICARD@ENPC.FR
LIGM, École des Ponts, 77455 Marnes la vallée, France