ENGN6528/4528 / COMP6528/4528: Introduction to PyTorch

Jinguang Tong

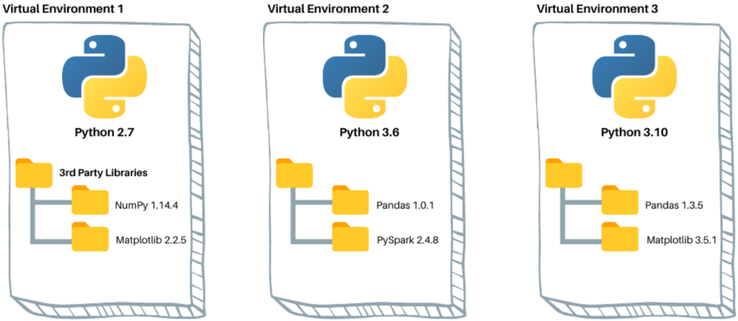
Australian National University

18/03/2024

Progression today

- Installing PyTorch
  - Create a virtual environment
  - Install PyTorch into the virtual environment
- A Classification Network with PyTorch
  - Basics
  - Prepare CIFAR-10 Dataset
  - Define a Model
  - Optimizer a Model
  - Visualize the process

Create a virtual environment



Create a virtual environment

Option 1: virtualenv

```
pip3 install virtualenv
```

Create a virtual environment "lab2"

```
cd $your_project_dir
virtualenv lab2
```

Activate the virtual environment

```
source lab2/bin/activate
```

Option 2: anaconda

Follow the instructions to install Anaconda
https://docs.anaconda.com/anaconda/install/linux/

Create a virtual environment "lab2"

```
conda create -n lab2
```

Activate the virtual environment

```
conda activate lab2
```

Installing PyTorch

- Install the latest stable version (2.0.0)

  https://pytorch.org/get-started/locally/

START LOCALLY

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, builds that are generated nightly. Please ensure that you have **met the prerequisites below (e.g., numpy)**, depending on your package manager. Anaconda is our recommended package manager since it installs all dependencies. You can also install previous versions of PyTorch. Note that LibTorch is only available for C++.

PyTorch Build	Stable (2.0.0)	Preview (Nightly)		
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python	C++ / Java		
Compute Platform	CUDA 11.7	CUDA 11.8	ROCm 5.4.2	CPU
Run this Command:	`pip3 install torch torchvision torchaudio`			

- Install previous PyTorch versions

  https://pytorch.org/get-started/previous-versions/

A Classification Network with PyTorch

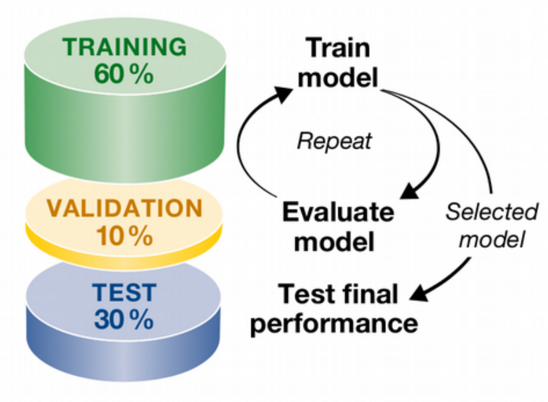
PyTorch: **NumPy** + **Auto differentiation** + **Utility Functions**

Basics: Tensor

Tensor is a specialized data structure very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters.

- Initialize a Tensor
- Attributes of a Tensor
- Operations on Tensors
- Bridge with NumPy

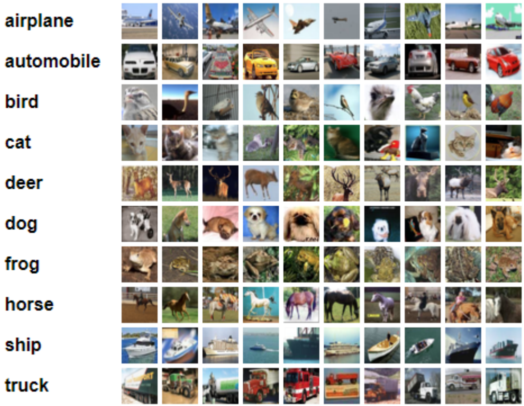
General Pipeline



- Train model on training set.
- Evaluate model on validation set and select the model with best performance.
- Test model on test set.

Prepare the dataset: CIFAR-10

CIFAR-10 dataset consists of 60000 32×32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.



Label	Description
0	airplane
1	automobile
2	bird
3	cat
4	deer
5	dog
6	frog
7	horse
8	ship
9	truck

For your assignment, you should download data from the given link.

Prepare the dataset: Dataset & DataLoader

PyTorch provides two data primitives: `torch.utils.data.Dataset` and `torch.utils.data.DataLoader` that allows you to use pre-loaded datasets as well as your own data.

- `Dataset` stores the samples and their corresponding labels.
- `DataLoader` wraps an iterable around the `Dataset` to enable easy access to the samples.

Prepare the dataset: Creating a Custom Dataset

A custom Dataset class should inherit `Dataset` class and implements three functions: `__init__`, `__len__`, and `__getitem__`.

- `__init__`: Initialize everything you will need for the dataset.
- `__len__`: Return the number of samples in the dataset.
- `__getitem__`: Load and return a sample from the dataset at the given `index`.

Prepare the dataset: Transforming and Augmenting

https://pytorch.org/vision/stable/transforms.html

Torchvision offers many common image transformations in the `torchvision.transform` module. Different transforms can be composed with `torchvision.transform.Compose`.

Pad

The `Pad` transform (see also `pad()`) fills image borders with some pixel values.

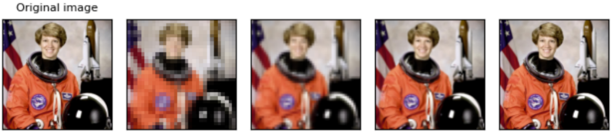
```
padded_imgs = [T.Pad(padding=padding)(orig_img) for padding in (3, 10, 30, 50)]
plot(padded_imgs)
```

Prepare the dataset: Transforming and Augmenting

Resize

The `Resize` transform (see also `resize()`) resizes an image.

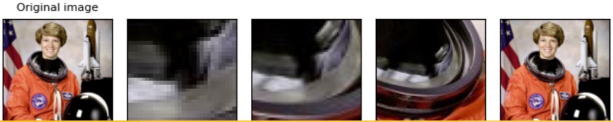
```
resized_imgs = [T.Resize(size=size)(orig_img) for size in (30, 50, 100, orig_img.size)]
plot(resized_imgs)
```



CenterCrop

The `CenterCrop` transform (see also `center_crop()`) crops the given image at the center.

```
center_crops = [T.CenterCrop(size=size)(orig_img) for size in (30, 50, 100, orig_img.size)]
plot(center_crops)
```



Prepare the dataset: Transforming and Augmenting

TOTENSOR

CLASS torchvision.transforms.ToTensor [SOURCE]

Convert a PIL Image or ndarray to tensor and scale the values accordingly.

This transform does not support torchscript.

Converts a PIL Image or numpy.ndarray (H x W x C) in the range [0, 255] to a torch.FloatTensor of shape (C x H x W) in the range [0.0, 1.0] if the PIL Image belongs to one of the modes (L, LA, P, I, F, RGB, YCbCr, RGBA, CMYK, 1) or if the numpy.ndarray has dtype = np.uint8

In the other cases, tensors are returned without scaling.

> • NOTE
>
> Because the input image is scaled to [0.0, 1.0], this transformation should not be used when transforming target image masks. See the references for implementing the transforms for image masks.

NORMALIZE

CLASS torchvision.transforms.Normalize(*mean*, *std*, *inplace=False*) [SOURCE]

Normalize a tensor image with mean and standard deviation. This transform does not support PIL Image. Given mean: `(mean[1],...,mean[n])` and std: `(std[1],..,std[n])` for `n` channels, this transform will normalize each channel of the input `torch.*Tensor` i.e., `output[channel] = (input[channel] - mean[channel]) / std[channel]`

Prepare the dataset: DataLoader

DataLoader used for loading data with **multi-process**, **batching** ....

CLASS torch.utils.data.DataLoader(*dataset*, *batch_size=1*, *shuffle=None*, *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*, *pin_memory_device=''*) [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.

Define a Model

Our model should inherit the `torch.nn.Module` class. We define our model in `__init__` and `forward` functions.

CLASS torch.nn.Module [SOURCE]

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

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

class Model(nn.Module):
    def __init__(self):
        super().__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))
```

TORCH.NN

These are the basic building blocks for graphs:

torch.nn

- Containers
- Convolution Layers
- Pooling layers
- Padding Layers
- Non-linear Activations (weighted sum, nonlinearity)
- Non-linear Activations (other)
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers
- Distance Functions
- Loss Functions
- Vision Layers
- Shuffle Layers
- DataParallel Layers (multi-GPU, distributed)
- Utilities
- Quantized Functions
- Lazy Modules Initialization

https://pytorch.org/docs/stable/nn.html

Define a Model: 2d Convolution Layer

2d Convolution Layer.

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.

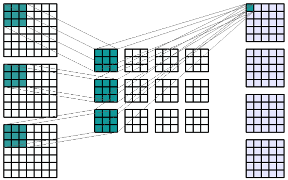
This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

- `stride` controls the stride for the cross-correlation, a single number or a tuple.
- `padding` controls the amount of padding applied to the input. It can be either a string {'valid', 'same'} or an int / a tuple of ints giving the amount of implicit padding applied on both sides.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - At groups=1, all inputs are convolved to all outputs.
  - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels and producing half the output channels, and both subsequently concatenated.
  - At groups= `in_channels`, each input channel is convolved with its own set of filters (of size $\frac{\text{out\_channels}}{\text{in\_channels}}$).

The parameters `kernel_size`, `stride`, `padding`, `dilation` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension



in_channels: 3
out_channels: 4
kernel_size: $(3, 3)$
padding: 0

2d maxpooling

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:

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

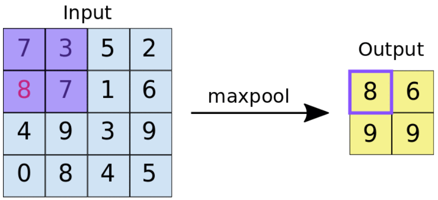
If `padding` is non-zero, then the input is implicitly padded with negative infinity on both sides for `padding` number of points. `dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

• NOTE

When ceil_mode=True, sliding windows are allowed to go off-bounds if they start within the left padding or the input. Sliding windows that would start in the right padded region are ignored.

The parameters `kernel_size`, `stride`, `padding`, `dilation` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension



kernel_size: $(2, 2)$
stride: 2

Define a Model: Non-linear Activation Layer

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

Applies the rectified linear unit function element-wise:

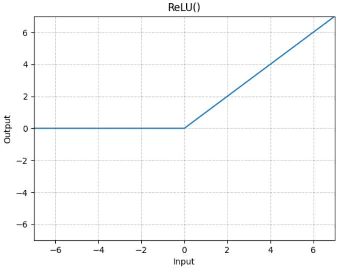
$$\text{ReLU}(x) = (x)^+ = \max(0, x)$$

Parameters:

inplace (*bool*) – can optionally do the operation in-place. Default: `False`

Shape:

- Input: $(*)$, where $*$ means any number of dimensions.
- Output: $(*)$, same shape as the input.



As a module:

```
relu = nn.ReLU()
x = relu(x)
```

As a function:

```
x = torch.nn.functional.relu(x)
```

Train a Model: Optimization Loop

We can train and optimize our model with an optimization loop. Each iteration of the optimization loop is called an **epoch**.

Each epoch consists of two main parts:

- **The Train Loop** - iterate over the training dataset and try to converge to optimal parameters.
- **The Evaluation Loop** - iterate over the test dataset to check if model performance is improving.

Train a Model: Loss Function

https://pytorch.org/docs/stable/nn.html#loss-functions

Loss Functions

nn.L1Loss	Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.
nn.MSELoss	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.
nn.CrossEntropyLoss	This criterion computes the cross entropy loss between input logits and target.
nn.CTCLoss	The Connectionist Temporal Classification loss.
nn.NLLLoss	The negative log likelihood loss.
nn.PoissonNLLLoss	Negative log likelihood loss with Poisson distribution of target.
nn.GaussianNLLLoss	Gaussian negative log likelihood loss.
nn.KLDivLoss	The Kullback-Leibler divergence loss.
nn.BCELoss	Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:
nn.BCEWithLogitsLoss	This loss combines a *Sigmoid* layer and the *BCELoss* in one single class.
nn.MarginRankingLoss	Creates a criterion that measures the loss given inputs $x1$, $x2$, two 1D mini-batch or 0D *Tensors*, and a label 1D mini-batch or 0D *Tensor* y (containing 1 or -1).
nn.HingeEmbeddingLoss	Measures the loss given an input tensor x and a labels tensor y (containing 1 or -1).

nn.CrossEntropy

Shape:

- Input: Shape (C), (N, C) or $(N, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of *K*-dimensional loss.
- Target: If containing class indices, shape $()$, (N) or $(N, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss where each value should be between $[0, C)$. If containing class probabilities, same shape as the input and each value should be between $[0, 1]$.
- Output: If reduction is 'none', shape $()$, (N) or $(N, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss, depending on the shape of the input. Otherwise, scalar.

where:

$$C = \text{number of classes}$$
$$N = \text{batch size}$$

Examples:

```
>>> # Example of target with class indices
>>> loss = nn.CrossEntropyLoss()
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.empty(3, dtype=torch.long).random_(5)
>>> output = loss(input, target)
>>> output.backward()
>>>
>>> # Example of target with class probabilities
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randn(3, 5).softmax(dim=1)
>>> output = loss(input, target)
>>> output.backward()
```

We pass model's output logits to `nn.CrossEntropyLoss`, which will normalize the logits and compute the prediction error.

Train a Model: Optimizer

https://pytorch.org/docs/1.13/optim.html

CLASS torch.optim.Optimizer(*params*, *defaults*) [SOURCE]

Base class for all optimizers.

• WARNING

Parameters need to be specified as collections that have a deterministic ordering that is consistent between runs. Examples of objects that don't satisfy those properties are sets and iterators over values of dictionaries.

Parameters:

- **params** (*iterable*) – an iterable of `torch.Tensor` s or `dict` s. Specifies what Tensors should be optimized.
- **defaults** – (dict): a dict containing default values of optimization options (used when a parameter group doesn't specify them).

`Optimizer.add_param_group`	Add a param group to the `Optimizer` s *param_groups*.
`Optimizer.load_state_dict`	Loads the optimizer state.
`Optimizer.state_dict`	Returns the state of the optimizer as a `dict`.
`Optimizer.step`	Performs a single optimization step (parameter update).
`Optimizer.zero_grad`	Sets the gradients of all optimized `torch.Tensor` s to zero.

Optimization is the process of adjusting model parameters to reduce model error in each training step. **Optimization algorithms** define how this process is performed. There are many different optimizers available in PyTorch such as SGD, ADAM and RMSProp.

We initialize the optimizer by registering the model's parameters that need to be trained, and passing in the learning rate hyperparameter.

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

Train a Model: Train Loop

```python
#@title Train Loop
def train_loop(dataloader, model, loss_fn, optimizer, device, epoch):
    running_loss = 0.0
    total_loss = 0.0
    # Get a batch of training data from the DataLoader
    for batch, data in enumerate(dataloader):
        # Every data instance is an image + label pair
        img, label = data

        # Transfer data to target device
        img = img.to(device)
        label = label.to(device)

        # Zero your gradients for every batch
        optimizer.zero_grad()

        # Compute prediction for this batch
        logit = model(img)

        # compute the loss and its gradients
        loss = loss_fn(logit, label)
        # Backpropagation
        loss.backward()

        # update the parameters according to gradients
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()     # ! Don't forget to use .item() to retri
        total_loss += loss.item()
        # report every 100 iterations
        if batch % 100 == 99:
            print(' epoch {} loss: {:.4f}'.format(epoch+1, running_loss / 100))
            running_loss = 0.0
    return total_loss / (batch+1)
```

1. Get a batch of training data from DataLoader
2. Zero the optimizer's gradients
3. Performs an inference
4. Calculate the loss for that set of predictions vs. the labels
5. Calculate the backward gradients over the learning weights
6. Tells the optimizer to perform one learning step
7. Report training statistics
8. Finally, return the average loss for camparison with a validation run

Train a Model: Evaluation Loop

```python
#@title Evaluation Loop
def evaluate_loop(dataloader, model, loss_fn, device):

    # Get number of batches
    num_batches = len(dataloader)

    test_loss, correct, total = 0, 0, 0

    # Context-manager that disabled gradient calculation.
    with torch.no_grad():
        for data in dataloader:
            # Every data instance is an image + label pair
            img, label = data
            # Transfer data to target device
            img = img.to(device)
            label = label.to(device)

            # Compute prediction for this batch
            logit = model(img)

            # compute the loss
            test_loss += loss_fn(logit, label).item()    # ! Don't forget .item() again!!!

            # Calculate t argmax: Any aximum logit as the predicted label
            pred = logit.argmax(dim=1)
            # record correct predictions
            correct += (pred == label).type(torch.float).sum().item()
            total += label.size(0)
    # Gather data and report
    test_loss /= num_batches
    accuracy = correct / total
    print("Test Error: \n   Accuracy: {:.2f}, Avg loss: {:.4f} \n".format(100*accuracy, test_loss))

    return test_loss, accuracy
```

1. Disable gradient calculation
2. Get a batch of testing data from DataLoader
3. Performs an inference
4. Calculate the loss for that set of predictions vs. the labels
5. Calculate metrics
6. Report testing statistics
7. Finally, return the metrics for comparison

Visualization with TensorBoard: Setup

https://pytorch.org/docs/master/tensorboard.html

TensorBoard is a tool for providing the measurements and visualizations needed during the machine learning workflow. It enables tracking experiment metrics like **loss** and **accuracy**, **visualizing the model graph** and much more.

Install TensorBoard (in your virtual environment)

```
pip3 install tensorboard
```

Import and create tensorboard instance

```python
from torch.utils.tensorboard import SummaryWriter

# default `log_dir` is "runs" - we'll be more specific here
writer = SummaryWriter('runs/fashion_mnist_experiment_1')
```

Visualization with TensorBoard: Write to TensorBoard

- Inspect the model using TensorBoard

```
add_graph(model, input_to_model=None, verbose=False, use_strict_trace=True)
```

- Add image to TensorBoard

```
add_image(tag, img_tensor, global_step=None, walltime=None, dataformats='CHW')
```

- Tracking model training with TensorBoard

```
add_scalar(tag, scalar_value, global_step=None, walltime=None, new_style=False,
double_precision=False)
```
[SOURCE]

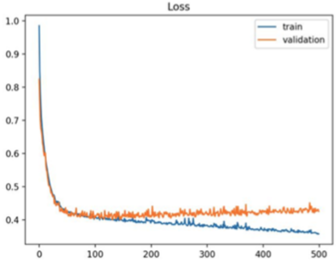
Visualization: Manually plot using matplotlib

Import matplotlib

```
import matplotlib.pyplot as plt
```

Plot the training curves

```
plt.plot(losses)
```



Note

Using tensorboard is a better way to monitor the training process. You will not lose marks if you plot it manually, but we recommend using TensorBoard.

Train a Model: Save and Load Model Weights

PyTorch models store the learned parameters in an internal state dictionary, called `state_dict`. These can be persisted via the `torch.save` method:

```
model = MyModel()
torch.save(model.state_dict(), 'model_weights.pth')
```

To load model weights, you need to create an instance of the same model first, and then load the parameters using `load_state_dict()` method:

```
model = MyModel()
model.load_state_dict(torch.load('model_weights.pth'))
```

Good Luck!