Numpy & PyTorch Tutorial

Jacobs Undergraduate Mentoring Program

Allen Zeng

May 12, 2022

Overview

1. Installation
2. Numpy
3. PyTorch
4. Example

Installation

Installation
conda

- Conda is an open source package management system and environment management system that runs on Windows, macOS, Linux.
- Miniconda is a free minimal installer for conda.
- Installer: `https://docs.conda.io/en/latest/miniconda.html`
- Conda Cheatsheet (Google for latest version)

Installation
conda

- Making a new Conda environment and installing Numpy and PyTorch.
- In terminal, you start in the `base` environment.
- Create a new environment for this project: `conda create --name tutorial`
- Activate the new environment:
  - Windows: `activate tutorial`
  - Linux/Mac: `source activate tutorial`
- See a list of your environments: `conda env list`

Installation
Numpy

- Packages installed within this new environment will not affect other environments.
- For example, you can install different environments for different school classes.
- The conda installation command is
  - `conda install <list-of-packages> -c <channel>`
- For machine learning or data science, there are other recommend packages to install:
  - `numpy matplotlib seaborn scipy opencv-python pillow scikit-learn`
- Example installation:
  - `conda install numpy matplotlib -c conda-forge`

Installation
PyTorch

- In order to install PyTorch, go to `https://pytorch.org/`
  - PyTorch Build: Stable
  - Your OS: whichever is appropriate
  - Package: Conda
  - Language: Python
  - Compute Platform: highest CUDA[1] version, or CPU
- You should see a command like
  - `conda install pytorch torchvision torchaudio cudatoolkit=11.3 -c pytorch`

[1] CUDA version packaged with PyTorch is independent of other CUDA installations.

Numpy

Numpy
Overview

- Numpy is “The fundamental package for scientific computing with Python”
- Allows you to easily work with matrices and tensors
- Many other libraries use Numpy datatypes to hold their data
- Numpy is often abbreviated as “np” using
  - `import numpy as np`
- `np.ndarray` is a N-dimensional array with one associated datatype:
  - `np.single`, `np.double`, `np.uint8`, `np.int64`, etc.

Numpy

Creating simple matrices

- `np.zeros(shape)`, where `shape = (num_rows, num_cols)` [1]
- `np.zeros((3, 4))` gets
- `array([[0., 0., 0., 0.], [0., 0., 0., 0.], [0., 0., 0., 0.]])`
- Which is equivalent to:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

[1] The double parentheses are important! The `shape` argument should be a sequence, like list or tuple.

Numpy

Creating simple matrices

- Same thing for `np.ones((3, 4))`

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

- Create identity matrices with `np.eye(N)`, where `N` is the number of rows/cols
- For `N=3`

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Numpy

Creating simple matrices

- Create from python list, list of lists, or other iterables
- `np.array(iterable)`
- `np.array([[1, 2, 3], [4, 5, 6]])`

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- `np.array(range(9)).reshape(3, 3)`

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

- `np.array(range(9))` gets you a 1-D array with 9 elements
- `.reshape(3, 3)` formats it into a 2-D array with dimensions (3, 3)

Numpy
Creating simple matrices

- You can also create N-dimensional arrays
  - `np.zeros((3, 4, 5))`
- Very useful in image processing
- Say `image.png` is a color image with dimension width=1920, height=1080 pixels
  - `from PIL import Image`
  - `image = np.asarray(Image.open('image.png'))`
  - `image.shape`
  - Returns (1080, 1920, 3) corresponding to number of (height, width, channels)
- PyTorch uses (B, C, H, W) for training batches of images, where
  - `B=batch size, C=num channels, H=height, W=width`

Numpy
Matrix Operations

- Let
  - `a = np.array(range(12)).reshape(3, 4)`
  - `b = np.ones((3, 4))`

$$a = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}, b = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

- `(a + b)` is element-wise addition, and `(a - b)` is element-wise subtraction

$$a + b = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

- `(a * b)` and `(a / b)` are also **element-wise** multiplication and division
  - In this case since `b` is all 1's, `(a * b) = a` and `(a / b) = a`

Numpy
Matrix Operations

- You can also perform scalar operations on these matrices:
  - `a + 2`
  - `a - 2`
  - `a * 2`
  - `a / 2`
- For example, `a / 2` returns

```
array([[0.,  0.5,  1.,  1.5],
       [2.,  2.5,  3.,  3.5],
       [4.,  4.5,  5.,  5.5]])
```

Numpy
Matrix Operations

- Dot product (matrix multiplication)
- `a @ b` or `np.matmul(a, b)`
- As before, let
  - `a = np.array(range(12)).reshape(3, 4)`
  - `b = np.ones((3, 4))`
- `a @ b` $\rightarrow$ `ValueError: matmul: Input operand 1 has a mismatch...`
- Number of columns in `a` must match number of rows in `b`

Numpy
Matrix Operations

- Now let
  - `a = np.array(range(12)).reshape(3, 4)`
  - `b = np.ones((4, 3))`

$$a = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}, \; b = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$a \mathbin{@} b = \begin{bmatrix} 6 & 6 & 6 \\ 22 & 22 & 22 \\ 38 & 38 & 38 \end{bmatrix}, \; b \mathbin{@} a = \begin{bmatrix} 12 & 15 & 18 & 21 \\ 12 & 15 & 18 & 21 \\ 12 & 15 & 18 & 21 \\ 12 & 15 & 18 & 21 \end{bmatrix}$$

Numpy
Matrix Elements

- Access individual matrix elements with `[]`
- Let `a = np.array(range(12)).reshape(3, 4)`

$$a = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}$$

- `a[0, 0]` $\rightarrow 0$
- `a[1, 1]` $\rightarrow 5$
- `a[2, 3]` $\rightarrow 11$
- Number of indexing values should be the same as the number of matrix dimensions

Numpy
Matrix Elements

- Access “slices” of a matrix elements with `[:]`
- Let `a = np.array(range(12)).reshape(3, 4)`

$$a = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}$$

- `a[1,:]` $\rightarrow$ `array([4, 5, 6, 7])`
- `a[:,1]` $\rightarrow$ `array([1, 5, 9])`

Numpy
Matrix Elements

- Access “slices” of a matrix elements with `[:]`
- Let `c = np.ones((3, 4, 5, 6))`
- `c[0, 0, 0, 0]` $\rightarrow$ 1
- `c[0, 0, 0, :]` $\rightarrow$ matrix with shape (6,)
- `c[0, 0, :, :]` $\rightarrow$ matrix with shape (5, 6)
- `c[:, :, :, 0]` $\rightarrow$ matrix with shape (3, 4, 5)
- `c[0, :, :, 0]` $\rightarrow$ matrix with shape (4, 5)
- What happens if there are less indexing values than number of dimensions?
  - `c[0]` returns a matrix with shape (4, 5, 6), but this could be confusing to readers

Numpy
Matrix Utilities

- `a.shape`, returns the dimensions of the matrix, e.g. (3, 4, 5)
- `a.ndim`, returns the number of dimensions of the matrix, e.g. 3
- `a.size`, returns the total number of elements in the matrix, e.g. 60
- `np.transpose(a)` or `a.T`
- `np.mean(a)` or `a.mean()`
- `np.nanmean(a)` or `a.nanmean()`, computes mean while ignoring NaNs
- `np.std(a)` or `a.std()`
- `np.linalg.norm(a)`
- The `a.func()` format allows you to chain operations together, e.g.
  - `a.T.mean(axis=0).shape`

Numpy
Matrix Utilities

- For some operations, you can specify an `axis` argument
- Specifies which axis to perform the operation along, and "deleting" that axis
- Let `a = np.array(range(12)).reshape(3, 4)`

$$a = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}$$

  - `np.mean(a, axis=0)` $\rightarrow$ `array([4., 5., 6., 7.])`, 4 elements left
  - `np.mean(a, axis=1)` $\rightarrow$ `array([1.5, 5.5, 9.5])`, 3 elements left

Numpy
Matrix Utilities

- From previous slide, let `a = np.array(range(12)).reshape(3, 4)`
  - `np.mean(a, axis=0)` $\rightarrow$ `array([4., 5., 6., 7.])`, output shape is (4,)
  - `np.mean(a, axis=1)` $\rightarrow$ `array([1.5, 5.5, 9.5])`, output shape is (3,)
- Notice the above returns 1-D arrays, this may be inconvenient if you are using 2-D arrays
- You can reshape the output, or use the `keepdims` argument
  - `np.mean(a, axis=0, keepdims=True)` $\rightarrow$ `array([[4., 5., 6., 7.]])`
    - Output shape is (1, 4)
  - `np.mean(a, axis=1, keepdims=True)` $\rightarrow$ `array([[1.5], [5.5], [9.5]])`
    - Output shape is (3, 1)

PyTorch

PyTorch
Overview

- “An open source machine learning framework that accelerates the path from research prototyping to production deployment.”
- Commonly used for deep learning
  - Competes with TensorFlow
- Although both libraries can do many of the same things, PyTorch is more commonly used in research and TensorFlow is more commonly used in production
- Can convert deep learning models trained in PyTorch/TensorFlow to others using open formats like ONNX
- PyTorch is easier to do step-through debugging/inspecting like with normal python

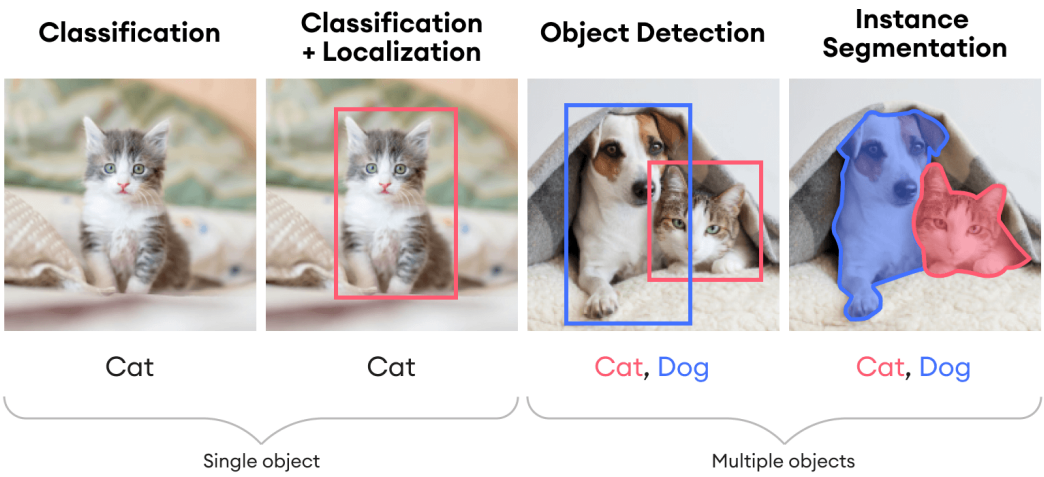
PyTorch
From Numpy

- PyTorch “tensors” are functionally the same as Numpy “ndarrays”
- Converting to and from Numpy
  - `t = torch.from_numpy(a)`
  - `a = t.numpy()`
- The tensor and the ndarray share the same underlying storage in memory.
- Tensors have a lot of the same basic operations as ndarrays

PyTorch
Deep Neural Networks

- Deep neural networks (DNNs) can be thought of as universal function approximators
- Given an input and output pair x, y, we can approximate function $y = f(x)$ using a DNN
  - Input x is usually a complicated signal like text, audio, or images
  - Output y, also called "labels" or "ground truth", are values that you want to extract with the input signal
- Neural Network Playground `https://playground.tensorflow.org/`

PyTorch
Deep Neural Networks

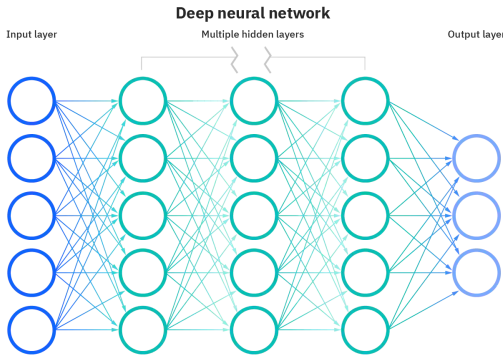


Credit: https://blog.superannotate.com/image-segmentation-for-machine-learning/

PyTorch
Deep Neural Networks

- There are many types of neural networks, each structured for different domains
- Two common types:
  - Convolutional Neural Networks (CNNs) are useful for working with images
  - Recurrent Neural Networks (RNNs) are useful for working with temporal sequences, e.g. text and audio
- Many more specializations: graph neural networks, long short-term memory (LSTM), gated recurrent units (GRU), U-Net, transformers, spiking neural networks, etc.

PyTorch
Deep Neural Networks



Credit: `https://www.ibm.com/cloud/learn/neural-networks`

PyTorch
Training Neural Networks

- Neural networks are trained through a “forward pass” and a “backward pass”
  - **Forward pass:** Neural networks learn by taking an input, passing it through the network (lots of linear and non-linear matrix operations), and getting an output.
  - **Backward pass:** A loss is then calculated between the network output (“prediction”) and the ground truth “label”), and gradients at each network layer is calculated with respect to this loss. The layers’ weights are updated using these gradients so that the network prediction should be closer to the labels.
- Good news: PyTorch handles most of this for you.[1]

[1] Calculating these gradients is actually a complicated topic, and I recommend taking an actual deep learning class to learn more.

PyTorch
Training Neural Networks

- Main ingredients to train a DNN
  - Good dataset[1]
  - DNN model, e.g. a CNN
  - Training loop
  - Good GPU[2]

[1] In practice, this is often the bottleneck for good results and around 80% of the work.

[2] For learning, you can use Google Colab (`https://colab.research.google.com/`). For industry, you can build your own servers or rent some from a cloud company.

PyTorch
Dataloader

- In order to train a DNN as fast as possible, you want to parallelize operations
- For custom datasets, PyTorch's `torch.utils.data.Dataset` abstract class handles data parallelization for most use cases
- You would need to inherit `Dataset` and override two methods:
  1. `__len__(self)`, which returns the size of the dataset
  2. `__getitem__(self, idx)`, which returns an input-output (x, y) pair of the dataset
  3. (Optional) `__init__(self, ...)`, to initialize the dataset

PyTorch
Dataloader

- For example, if your dataset is small enough to fit into memory, load it inside `__init__` and `__getitem__` could consist of a simple array access `data[idx]`
- If your dataset is too big to fit into memory, such as thousands of images, you could read image `#idx` inside `__getitem__`
- Further tutorial:
  `https://pytorch.org/tutorials/beginner/data_loading_tutorial.html`

PyTorch
Training Loop Pseudocode

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Initialize neural network
cnn = MyCNN().to(device)
cnn.train()

# Initialize loss criterion and optimizers
criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.Adam(cnn.parameters(), lr=learning_rate, weight_decay=weight_decay)
scheduler = lr_scheduler.StepLR(optimizer, step_size=step_size, gamma=gamma)

# Initialize dataset
train_dataset = MyDataset()
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, num_workers=num_workers, shuffle=True)

```

PyTorch
Training Loop Pseudocode

```
# Training loop
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Clear gradients for this training iteration
        optimizer.zero_grad()

        # Move data to the GPU is you have one
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        preds = cnn(images)

        # Backward pass
        loss = criterion(preds, labels)
        loss.backward()  # Calculates gradients with respect to the loss
        optimizer.step()  # Updates the network weights
        scheduler.step()  # Updates the step size

```

Example

Example
MNIST

- MNIST handwritten digit database of 60K training images and 10K testing images
  - Training images are for training the machine learning model
  - Testing is for after training the model, and seeing how it would perform on never-before-seen data
- Handwritten digits here are 28x28 grayscale images, i.e. each image has shape (28, 28)
- Inspired by this tutorial[1], which is good but has some out-of-date code patterns

[1] https://medium.com/@nutanbhogendrasharma/pytorch-convolutional-neural-network-with-mnist-dataset-4e8a4265e118

- Downloading MNIST data

```
from torchvision import datasets
from torchvision.transforms import ToTensor

train_data = datasets.MNIST(
    root='data', train=True, transform=ToTensor(), download=True)
test_data = datasets.MNIST(
    root='data', train=False, transform=ToTensor(), download=True)
```

- Visualizing MNIST data

```
import matplotlib.pyplot as plt

plt.imshow(train_data.data[0], cmap='gray')
plt.show()
```

- Setting up dataloaders

```
from torch.utils.data import DataLoader

train_loader = torch.utils.data.DataLoader(
    train_data, batch_size=100, shuffle=True, num_workers=1)
test_loader = torch.utils.data.DataLoader(
    test_data, batch_size=100, shuffle=True, num_workers=1)
```

CNN module

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=1,
                      out_channels=16,
                      kernel_size=(5, 5),
                      stride=(1, 1),
                      padding=2,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels=16,
                      out_channels=32,
                      kernel_size=(5, 5),
                      stride=(1, 1),
                      padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.out = nn.Linear(32 * 7 * 7, 10)  # 32 channels * (5 + 2) * (5 + 2) feature map size

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        # Flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        x = x.view(x.size(0), -1)
        return self.out(x)
```

Training Loop

```
if __name__ == '__main__':
    # Initialize neural network
    cnn = CNN().to(device)
    cnn.train()

    # Initialize loss criterion and optimizers
    criterion = nn.CrossEntropyLoss().to(device)
    optimizer = optim.Adam(cnn.parameters(), lr=0.01, weight_decay=0.001)

    # Training loop
    for epoch in range(5):
        for i, (images, labels) in tqdm(enumerate(train_loader), total=600):
            # Clear gradients for this training iteration
            optimizer.zero_grad()

            # Move data to the GPU is you have one
            images = images.to(device)
            labels = labels.to(device)

            # Forward pass
            preds = cnn(images)

            # Backward pass
            loss = criterion(preds, labels)
            loss.backward()  # Calculates gradients with respect to the loss
            optimizer.step()  # Updates the network weights
```

Testing Loop

```
# Testing loop
cnn.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for i, (images, labels) in tqdm(enumerate(test_loader), total=100):
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        preds = cnn(images)
        preds = torch.max(preds, 1)[1].data.squeeze()
        correct += (preds == labels).sum().item()
        total += labels.shape[0]  # Get batch dimension
    print('Test accuracy', correct / total)
```

- Performance

```
Training loop:
100%|xxxxxxxxxx| 600/600 [00:07<00:00, 76.63it/s]
100%|xxxxxxxxxx| 600/600 [00:04<00:00, 120.58it/s]
100%|xxxxxxxxxx| 600/600 [00:04<00:00, 124.95it/s]
100%|xxxxxxxxxx| 600/600 [00:04<00:00, 126.53it/s]
100%|xxxxxxxxxx| 600/600 [00:04<00:00, 123.15it/s]
Testing loop:
100%|xxxxxxxxxx| 100/100 [00:01<00:00, 53.43it/s]
Test accuracy 0.9855
```

- Performance without training is about 10%, which is random guessing