Running AI Models

Samuel Yang-Zhao

National Computational Infrastructure, Australia

Acknowledgement of Country

The National Computational Infrastructure acknowledges, celebrates and pays our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

Outline

In this workshop we will introduce the basics of building and running Deep Learning AI models on the GPU using Python. Primarily, we will be using the Pytorch package which provides an easy-to-use library for building Deep Learning models.

- Introduction to PyTorch
  - Using Tensors
  - Using the GPU
- Introduction to Deep Learning
  - Neural Network Basics
  - Neural Network Training
- Deep Learning in PyTorch

Intro to PyTorch: Tensors

Tensors are one of the main data structures in PyTorch. They are very similar to NumPy's `ndarray`. The main difference is that Tensors can be run on the GPU.

Tensor initialization:

```
import torch
import numpy as np

# Directly from data
data = [[1,2],[3,4]]
X = torch.tensor(data)

# From numpy array
np_array = np.array(data)
X = torch.from_numpy(np_array)
```

```
# Direct constructors
shape = (100,100)

# Random tensor of dimension shape.
X = torch.rand(shape)

# Tensor of all ones.
Y = torch.ones(shape)

# Tensor of all zeros.
Z = torch.zeros(shape)
```

Tensor Attributes

Apart from holding the data, Tensors have many important attributes that you may need to access or modify.

```
# Tensor shape
>>> X = torch.rand(3,4)
>>> X.shape
torch.Size([3,4])

# Tensor data type
>>> X.dtype
torch.float32
```

```
# Modify data type (not in-place)
>>> X = X.type(torch.float16)
>>> X.dtype
torch.float16

# Tensor device
>>> X.device
device(type='cpu')
```

Tensor Operations

PyTorch provides over 100 tensor operations, including arithmetic, linear algebra, and matrix manipulations. Some example operations are listed here. When writing your own code, it is a good idea to quickly search PyTorch's documentation to see whether an operation you need is implemented.

```
# Indexing and slicing
>>> X = torch.tensor([[1,2],[3,4]]
>>> X[0]
tensor([1, 2])
>>> X[:,0]
tensor([1, 3])

# Tranpose
>>> X.T
tensor([[1, 3],
        [2, 4]])
```

```
# Matrix multiplication
>>> X.matmul(X.T)
tensor([[ 5, 11],
        [11, 25]])

# Element-wise multiplication
>>> X * X
tensor([[ 1,  4],
        [ 9, 16]])
```

Using GPU Devices

All operations on Tensors can be run on the GPU. By default, Tensors are created and run on the CPU (aka *host* in CUDA terms).

```
# Tensor device
>>> X.device
device(type='cpu')
```

We can use the `.to` method to send tensors to the GPU.

```
if torch.cuda.is_available():
    X = X.to("cuda")
```

Keep in mind that copying large tensors to GPU is expensive.

Using GPU Devices

Note that if we wish to perform tensor operations that require multiple tensors, all tensors must be on the same device.

```
X = torch.rand((10,10))
X = X.to("cuda")
Y = torch.rand((10,10))
X.matmul(Y)
# Runtime Error!
```

Selecting your GPU Device

On some systems with multiple GPUs you may want to use a specific GPU. We can run the command `nvidia-smi` display info about our GPUs:

```
[15]: !nvidia-smi

Thu Mar 23 15:14:28 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.60.13    Driver Version: 525.60.13    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000000:3E:00.0 Off |                    0 |
| N/A   33C    P0    42W / 300W |      0MiB / 32768MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

On this system we only have one GPU and its ID is 0. We can create a `torch.device` to specifically send our tensor to device 0.

```
device = torch.device("cuda:0")
X = torch.rand((100,100))
X = X.to(device)
```

Selecting your GPU Device

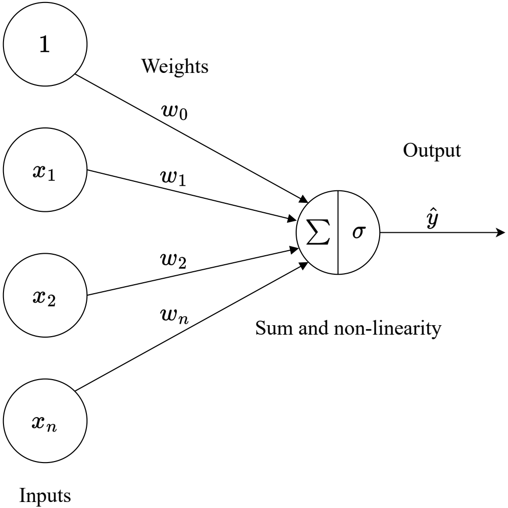
After sending the tensor to our GPU, running `nvidia-smi` again shows a process occupying memory on our GPU:

```
[24]: !nvidia-smi

Thu Mar 23 15:15:28 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.60.13    Driver Version: 525.60.13    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000000:3E:00.0 Off |                    0 |
| N/A   34C    P0    57W / 300W |   1109MiB / 32768MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A   2394296      C   ...thon_hpc_venv/bin/python3     1106MiB |
+-----------------------------------------------------------------------------+
```

Summary

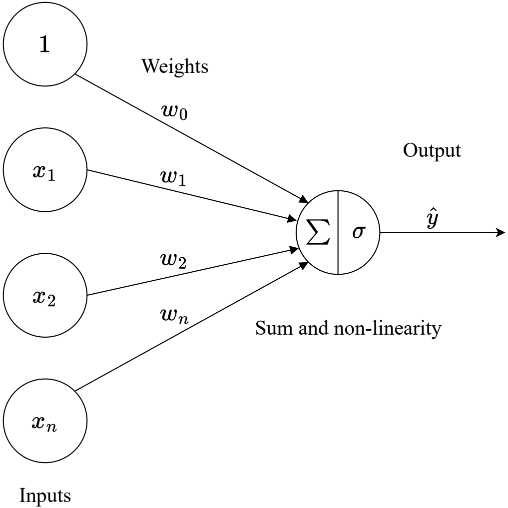
- Operating with PyTorch tensors is very similar to NumPy's ndarray.
- Main difference is PyTorch tensors can run on the GPU.
- Use the `.to` method to send tensors to GPU.
- Tensor operations on multiple tensors can only run if all tensors are located on the same device.

Neural Network Basics: The Perceptron



$$\hat{y} = \sigma\left(w_0 + \sum_{i=1}^{n} x_i w_i\right)$$

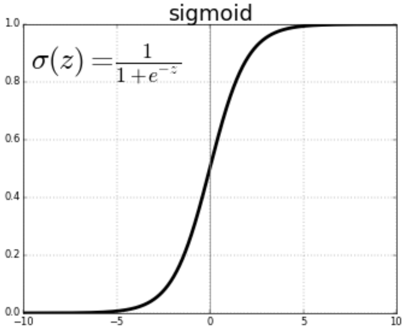
Neural Network Basics: The Perceptron

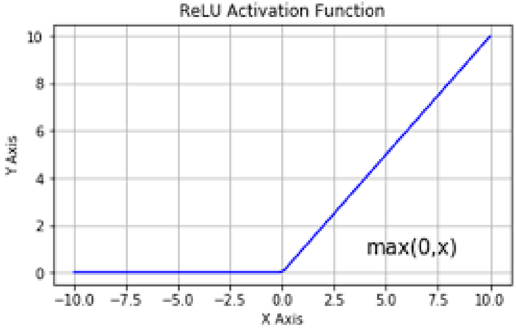


$$\hat{y} = \sigma\left(w_0 + X^\top W\right)$$

where $X^\top = [x_1, \ldots, x_n]$ and $W = [w_1, \ldots, w_n]^\top$

Activation Functions

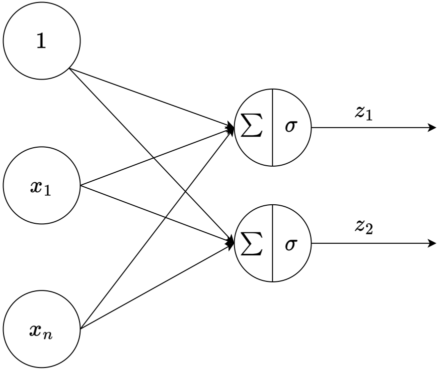




$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

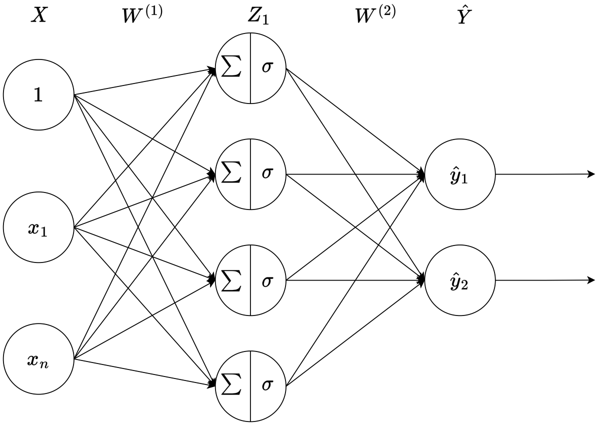
$$\sigma(z) = \max(0, z)$$

Multi-output Perceptron



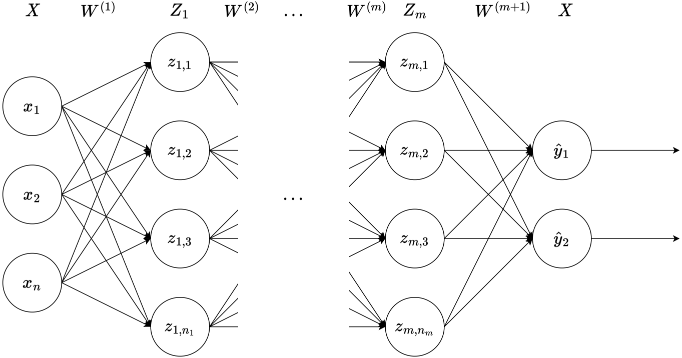
$$z_i = \sigma\left(w_0 + X^\top W\right)$$

Single Layer Neural Network



$$z_i^{(1)} = \sigma\left(w_{0,i}^{(1)} + X^\top W^{(1)}\right) \qquad \hat{y}_i = w_{0,i}^{(2)} + Z_1^\top W^{(2)}$$

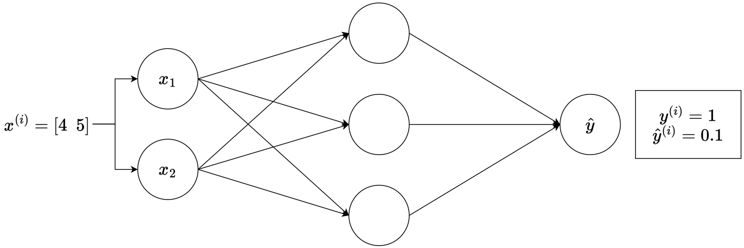
Deep Neural Network



$$z_{k,i} = \sigma\left(w_{0,i}^{(k)} + Z_{k-1}^{\top} W^{(k)}\right)$$

Supervised Learning

In supervised learning, we are given data $X = (x^{(1)}, \ldots, x^{(N)})$ and the associated labels $Y = (y^{(1)}, \ldots, y^{(N)})$, e.g. $x^{(i)}$ is an image of an animal and $y^{(i)}$ is the label ‘dog’. Our model is a function $f(\cdot; W)$ with parameters W. We pass $x^{(i)}$ through our neural network and observe its prediction $\hat{y}^{(i)} = f(x^{(i)}; W)$.



Loss functions

A loss function $\mathcal{L}$ allows us to compare the model's predictions to the true label. This provides a cost to the model for incorrect predictions. e.g. when $\mathcal{L}$ is the squared error:

$$\mathcal{L}(f(x^{(i)}; W), y^{(i)}) = (f(x^{(i)}; W) - y^{(i)})^2$$

For a given set of parameters W, the empirical loss (also known as empirical risk) is the average cost incurred over the training data.

$$\widehat{\mathcal{J}}(W) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f(x^{(i)}; W), y^{(i)})\ .$$

Optimizing the Loss

We wish to find a configuration of the network weights that minimizes the empirical risk.

$$W^* = \arg\min_W \widehat{\mathcal{J}}(W)$$

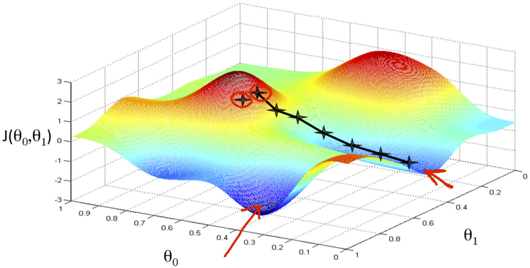
This is now an optimization problem and we can use optimization algorithms to try find a solution.

Gradient Descent

Gradient descent is a first-order iterative optimization algorithm that can find local minima of functions.

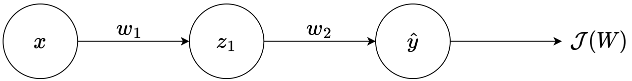
Algorithm

1. Initialize weights randomly
2. Loop until convergence:
3. Compute gradient $\frac{\partial \widehat{\mathcal{J}}(W)}{\partial W}$
4. Update $W \leftarrow W - \eta \frac{\partial \widehat{\mathcal{J}}(W)}{\partial W}$.
5. Return W



Backpropogation

Backpropogation is the algorithm used to compute the gradients used in gradient descent. Consider the following simple neural network.



To update our weights according to gradient descent, we need to compute $\frac{\partial \mathcal{J}(W)}{\partial w_1}$ and $\frac{\partial \mathcal{J}(W)}{\partial w_2}$.

Backpropogation

We first look to compute $\frac{\partial \mathcal{J}(W)}{\partial w_2}$.

$$x \xrightarrow{w_1} z_1 \xrightarrow{w_2} \hat{y} \longrightarrow \mathcal{J}(W)$$

Using the chain rule, we have

$$\frac{\partial \mathcal{J}(W)}{\partial w_2} = \frac{\partial \mathcal{J}(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2}$$

Backpropogation

We now look to compute $\frac{\partial \mathcal{J}(W)}{\partial w_1}$.

$$x \xrightarrow{w_1} z_1 \xrightarrow{w_2} \hat{y} \longrightarrow \mathcal{J}(W)$$

By iterating the chain rule, we have

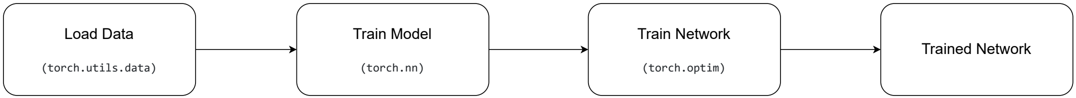
$$\begin{aligned}\frac{\partial \mathcal{J}(W)}{\partial w_1} &= \frac{\partial \mathcal{J}(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_1} \\ &= \frac{\partial \mathcal{J}(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2} \frac{\partial z_1}{\partial w_1}\end{aligned}$$

Gradient Descent in practice

- PyTorch contains adaptive learning rate schedulers to automatically adjust the learning rate η.
- The gradient $\frac{\partial \mathcal{J}(W)}{\partial W}$ is extremely expensive to compute in practice as it is a summation over all N data points.
- In practice, the gradient is computed with respect to B samples and is known as batch gradient descent.

Neural Networks in PyTorch

Training neural networks in PyTorch follows the following process.



- Data loading utilities are provided in `torch.utils.data`
- The `torch.nn` module provides all the building blocks to build neural networks.
- `torch.optim` provides the utilities to optimize the parameters of your neural network.

Loading Data

The canonical way to load data is to create a `Dataset` object and pass it to a `Dataloader` to sample batches.

```
# Create an example dataset
from sklearn.datasets import make_classification
X,y = make_classification()

# Load necessary Pytorch packages
from torch.utils.data import DataLoader, TensorDataset
from torch import Tensor

# Create basic Dataset using TensorDataset
dataset = TensorDataset( Tensor(X), Tensor(y) )

# Create a data loader
loader = DataLoader(dataset, batch_size=64)

# Iterate over all batches
for batch_idx, (data, target) in enumerate(loader):
    ...
```

Loading Data

For more custom datasets, a custom class that inherits from `Dataset` will need to be created. The custom dataset must implement `__init__`, `__len__`, `__getitem__`.

```
from torch.utils.data import Dataset
class CustomDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.tensor(X)
        self.y = torch.tensor(y)

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

    def __getitem__(self, idx):
        data = self.X[idx]
        label = self.y[idx]
        return x, y
```

Building Neural Networks

A neural network is built using the functionality provided in `torch.nn`. A neural network class in PyTorch subclasses `nn.Module` and holds a sequence of layers.

```
from torch import nn
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
                    nn.Linear(2, 100),
                    nn.ReLU(),
                    nn.Linear(100, 3)
                )
    def forward(self, x):
        output = self.layers(x)
        return output
```

Subclasses of `nn.Module` can be moved to the GPU.

```
import torch
device = torch.device("cuda")
model = NeuralNetwork()
model = model.to(device)
```

Training the Model

Now that we can create datasets and our NN model we can now look to optimize our parameters. The optimization process requires three key choices:

- Setting hyperparameters e.g. number of epochs, batch size, learning rate etc.
- Choice of loss function
- Choice of optimizer

Hyperparameter Tuning

Using the loss function

The specific loss function to choose will vary from task to task. Some popular choices are:

- `nn.MSELoss` (mean-squared error) for regression
- `nn.NLLLoss` (negative log-likelihood) for classification
- `nn.CrossEntropyLoss` for classification.

After computing the loss, the `.backward` method can be called to compute backpropogation calculations and get the gradient for our model parameters.

```
loss = nn.CrossEntropyLoss(y_hat, y)
loss.backward()
```

Optimizers

The choice of optimization algorithm you choose will define how model parameters are adjusted at each step. The `torch.optim` package provides implementations for many different optimizers. All require passing in your NN's parameters. For example stochastic gradient descent can be initialized as follows

```
optimizer = torch.optim.SGD(model.parameters(), lr = 1e-3)
```

Calling the optimizer's `.step` method then updates the model's parameters

```
optimizer.step()
```

Note that in training, `optimizer.zero_grad()` should be called in each loop to reset the gradients on the model parameters. This is because gradients add up by default.

Putting it all together

Assuming we've initialized our NN model and our dataset into a Dataloader we can now optimize our model in a loop. There are three key steps:

- Call `optimizer.zero_grad()` to reset the gradients on the model parameters and pevent double-counting. This is because gradients add up by default.
- Backpropagate the prediction loss with a call to `loss.backward()`. PyTorch deposits the gradients of the loss w.r.t. each parameter.
- Once we have our gradients, call `optimizer.step()` to adjust the parameters by the gradients collected in the backward pass.

Putting it all together

Assuming we've got the NN model and Dataloader from before, the training loop may look like:

```
# hyperparams
nepochs = 100
batch_size = 256
learning_rate = 1e-3

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(
              model.parameters(),
              lr = learning_rate)
```

```
for epoch in range(nepochs):
    for batch_idx, (x, y) in enumerate(loader):
        # send to device
        x, y = x.to(device), target.to(y)

        # Optimize
        optimizer.zero_grad()
        output = model(x)
        loss = criteria(output, target)
        loss.backward()
        optimizer.step()
```