Intro to PyTorch and Neural Networks

PyTorch Library

PyTorch is a machine learning library for developing deep learning models in Python.

```
# import pytorch
import torch
```

Creating PyTorch Tensors

PyTorch tensors store numerical data. They can be created from NumPy arrays using the `torch.tensor()` method. All entries in a tensor have the same type, specified by the `dtype` parameter.

```
# NumPy array
np_array = np.array([100, 75.5])

# Convert to a tensor with float dtype
torch_tensor = torch.tensor(np_array,
dtype=torch.float)
```

Linear Regression Review

A **linear model** consists of:

- one or more input features
- a numeric weight corresponding to each input feature
- an additional numeric term called the **bias**

Starting with specific input values for the input features, the linear model

- multiplies each input by its weight
- adds up the weighted inputs
- adds on a bias term

Linear models are generalizations of the classic equation of a line:

$$y = mx + b$$

Here,

- `y` is the target output
- `x` is the input feature
- `m` is the weight for `x`
- `b` is the bias

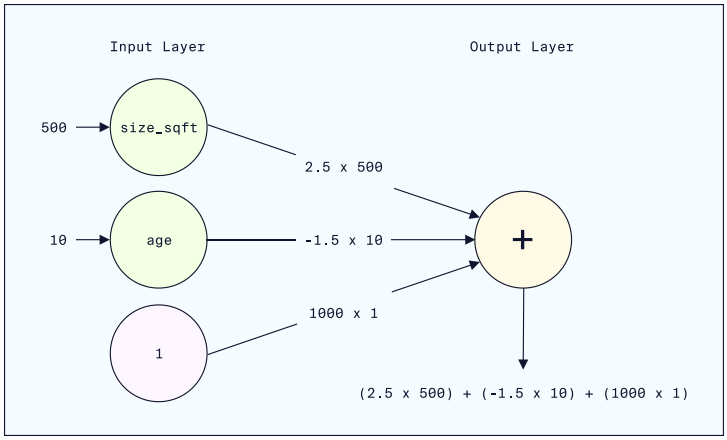
Linear Regression as Perceptrons

A linear equation can be modeled as a neural network structure called a **Perceptron** that consists of:

- an input layer with nodes for each input feature
- arrows containing weights corresponding to each input node
- an output layer with a single node that adds the weighted values together to produce an output

For example, the image attached to this review card demonstrates a Perceptron modelling

$$\text{rent} = 2.5\text{size\_sqft} - 1.5\text{age} + 1000$$



Activation Functions

Activation functions introduce non-linearities to a neural network. These allow the model to learn non-linear relationships in the dataset.

One way of thinking about activation functions is that they serve to "turn on" or "turn off" nodes, allowing the neural network to recognize specific properties of the training dataset (e.g. a particular node "turns on" under certain conditions.)

An activation function is applied by a node in a neural network *after* it computes the weighted sum of its inputs and bias.

PyTorch implements common activation functions like **ReLU** and **Sigmoid** in the `torch.nn` module.

```
# import nn module
from torch import nn
```

ReLU Activation Function

The **ReLU** activation function returns `0` for negative input values, otherwise it returns nonnegative values unchanged.
For example,

- `ReLU(-1)` returns `0` since `-1` is negative
- `ReLU(3)` returns `3` since `3` is nonnegative

ReLU can be implemented in PyTorch with `nn.ReLU` from the `torch.nn` module.

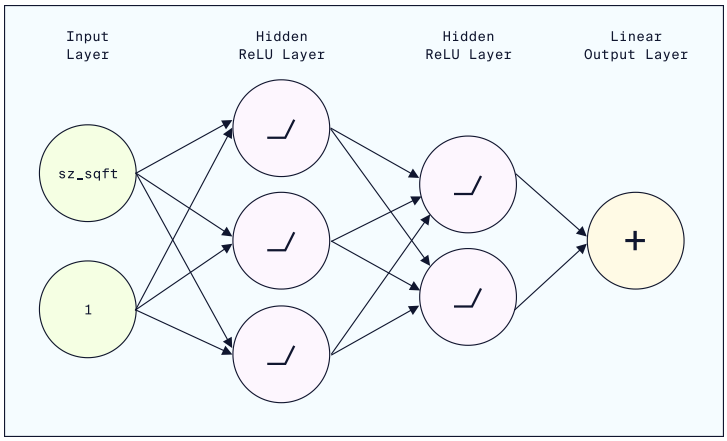
```
# by hand definition of ReLU
def ReLU(x):
    return max(0,x)

# ReLU in PyTorch
from torch import nn
ReLU = nn.ReLU()
```

Multi-Layer Neural Networks

Multi-layer Neural Networks consist of an input layer, several hidden layers, and an output layer.
Each node in a hidden layer is essentially a Perceptron.
Each one

- computes a weighted sum using the inputs and weights from the nodes in the prior layer
- (optionally) applies an activation function to the weighted sum
- sends the activated weighted sum as an input to the nodes of the next layer



Creating Sequential Networks in PyTorch

PyTorch's `nn.Sequential()` method builds neural networks by specifying layers and activation functions in sequence from input to output.
For example, the code attached to this review card defines a neural network with:

- 8 input nodes, feeding to
- a 16 node hidden layer with ReLU activation, and then
- a 10 node hidden layer with Sigmoid activation, and then
- a 1 node linear output layer

Data flows through this network in the order in which the layers and activation functions are specified.

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(8,16),
    nn.ReLU(),
    nn.Linear(16,10),
    nn.Sigmoid(),
    nn.Linear(10,1)
)
```

Loss Functions

The **loss function** is a mathematical formula used to measure the *error* (also known as loss values) between the model predictions and the actual target values.
The loss function is computed after feeding data through a neural network. Then, the loss is used to compute the gradients that tell the optimization algorithm how to adjust weights and biases to improve the neural network's performance.

Mean Squared Error Loss Function

The most common loss function for regression tasks is the **Mean Squared Error (MSE)** which is computed by

- computing the difference from each prediction to the actual value
- squaring each difference
- computing the mean of the squared differences

Notably, MSE emphasizes the largest differences which can be helpful but may lead to overfitting.

```
# Implement MSE in PyTorch
import torch.nn as nn
loss = nn.MSELoss()
MSE = loss(predictions, y)
```

The Optimizer

When training a neural network, **optimizers** are algorithms used to adjust the weights and biases of the neural network. The goal of an optimizer is to decrease the loss of the network.
PyTorch implements many common optimizer algorithms in its `optim` module.

```
from torch import optim
```

Gradient Descent Optimization

Gradient descent is an optimization algorithm that uses calculus to update the weights and biases of the network. We can think of gradient descent as being on top of a mountain where our objective is to descend down. To choose a direction, we look at the slope of the mountain and move towards where the mountain slopes down the most.
These slopes are called **gradients** in calculus. Gradient descent computes the gradients (slopes) of the loss function and then updates the weights and biases to move the neural network “downhill”, thereby decreasing loss.

Learning Rate

For optimization algorithms, the **learning rate** hyperparameter specifies how far we adjust each model parameter, like the weights and biases, at each training step.
Tradeoffs when choosing learning rate values:

- *high values* may cause the model to update too quickly and miss the lowest point, or even ricochet to high loss values
- *low values* may cause to model to learn too slowly or to get stuck in a low point, or valley, that isn’t the lowest possible

Adam Optimizer in PyTorch

The **Adam optimizer** is a popular variant of gradient descent that looks to iteratively minimize the loss function by adjusting the learning rate dynamically during training. The Adam optimizer in PyTorch takes two inputs:

- the model's current parameters, stored in `model.parameters()`
- the learning rate (usually fairly small)

```
import torch.optim as optim
optimizer = optim.Adam(model.parameters(),
lr=0.01)
```

The Backward Pass

To apply gradient descent optimizers in PyTorch, the following steps need to be taken:

- use the loss function to *compute the loss*
- apply `.backward()` to the loss to *calculate the gradients* of the loss function
- apply `.step()` to the optimizer to *update the weights and biases*

```
# Compute the loss
MSE = loss(predictions, y)

# Backward pass to calculate the gradient
MSE.backward()

# Use optimizer to update weights and
biases
optimizer.step()
```

Training Loop for Neural Networks

Training a neural network is an iterative process of:

- performing the forward pass to generate target predictions
- computing the loss between the predictions and the actual target values
- running a backward pass to compute gradients
- applying the optimizer to adjust the weights and biases
- resetting the gradients for the next iteration

Each iteration in this loop is called an **epoch**.

```
# Training loop with 400 iterations
num_epochs = 400
for epoch in range(num_epochs):
    predictions = model(X) # forward pass
    MSE = loss(predictions,y) # compute
loss
    MSE.backward() # compute gradients
    optimizer.step() # update weights and
biases
    optimizer.zero_grad() # reset the
gradients
```

Saving and Loading PyTorch Models

The `torch.save()` function saves an entire PyTorch model to a file that includes the model architecture and learned parameters.
The `torch.load()` function loads a saved PyTorch model.

```
# Saving PyTorch models
torch.save(model, 'model.pth')

# Loading PyTorch models
loaded_model = torch.load('model.pth')
```

Evaluation and Generating Predictions

To evaluate a PyTorch model on a testing dataset or generate predictions on a new dataset, we need to set the model to *evaluation mode* using `model.eval()` and *turn off gradient calculations* using `with torch.no_grad()`.

```
# Set model to evaluation mode
model.eval()

# Turn off gradient calculations
with torch.no_grad():
  # Generate predictions on testing dataset
  predictions = model(X_test)
  test_MSE = loss(predictions, y_test)
```

Build a Neural Network Class

Neural network classes can be constructed using *object-oriented programming (OOP)* with the PyTorch subclass `nn.Module`. This requires explicitly defining the `init` method to initialize the network components and the `forward` method to design the forward pass. Constructing networks as classes gives developers increased flexibility and control over pre-built sequential models using `nn.Sequential` while still inheriting PyTorch's built-in training and optimization libraries.

```
# Creating a Neural Network Class Using OOP
class NN_Regression(nn.Module):
  def __init__(self):
    super(NN_Regression, self).__init__()
    # Initialize layers and activation functions
    ..

  def forward(self, x):
    # Define forward pass
    ..
    return x

# Instantiate Neural Network Class
model = NN_Regression()
```