Intro to PyTorch and Neural Networks

Intro To Tensors

- Tensors: Storage containers for Numerical Data
- `torch.tensor()` has 2 arguments
    - what you want to convert
    - and TO what
    -

    ```
    rent = 2500
    rent_tensor = torch.tensor(rent, dtype=torch.int)
    ```

```
arr = np.array([2500,750, 3.5])
arrTotensor = torch.tensor(arr, dtype = torch.float)
```

This is for a dataframe:

```
torch.tensor(df.values, dtype=torch.float)
```

Note: working with individual columns in DF can cause issues because torch assumes dimensions

Therefore, make sure that the column is also a DF

```
torch.tensor(df[['column1']].values, dtype=torch.float)
#OR
torch.tensor(df['column1'].values, dtype=torch.float).view(-1,1)
```

💡 `torch.tensor(numerical_data, dtype = desired_datatype)`

💡 Binary cross entropy loss expects a 2d tensor (y)
Cross entropy loss expects a 1d tensor (y) and (long)

Linear Regression Review

- use linear eqn to make predictions

$$y = mx + b$$

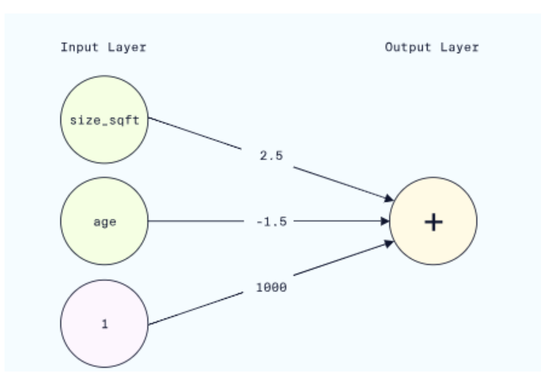
$$rent = 2.5sz_sqft + 1000$$

rent = o/p; 2.5 = weight; sz_feat = input or feature; 1000 = bias

Linear Regression With Perceptrons

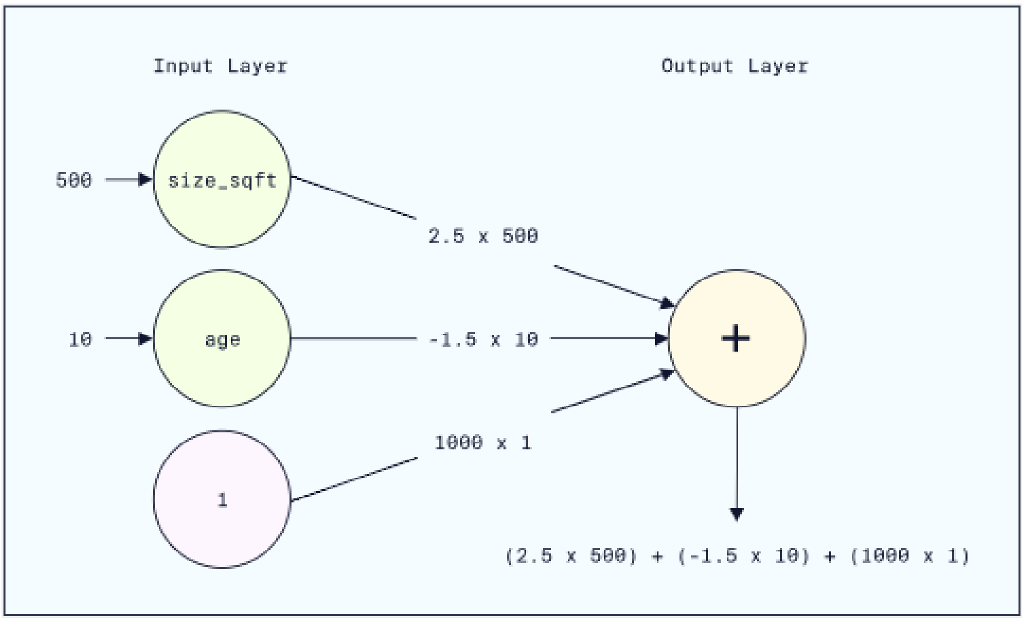
Perceptron:

- A Neural Network Structure



- Has : Nodes, Edges
- One set of input leads to ONE single output node
-

$$rent = 2.5 \times 500 - 1.5 \times 10 + 1000 \times 1$$
$$rent = 2.5 \times 500 - 1.5 \times 10 + 1000 \times 1$$



Activation Functions

> An **activation function** is the function used by a node in a neural network to take the summed weighted input to the node and transform it into the output value.

- NN not just a fancy way of doing Linear Regression
- Non Linear Activation Functions
    - Can model Non linear relationships within the data
- Earlier:

    - get weights
    - weighted output given
- Now:
    - get weights
    - weighted output (gives same o/p as before)
    - introduce non linearity

ReLU Activation Function

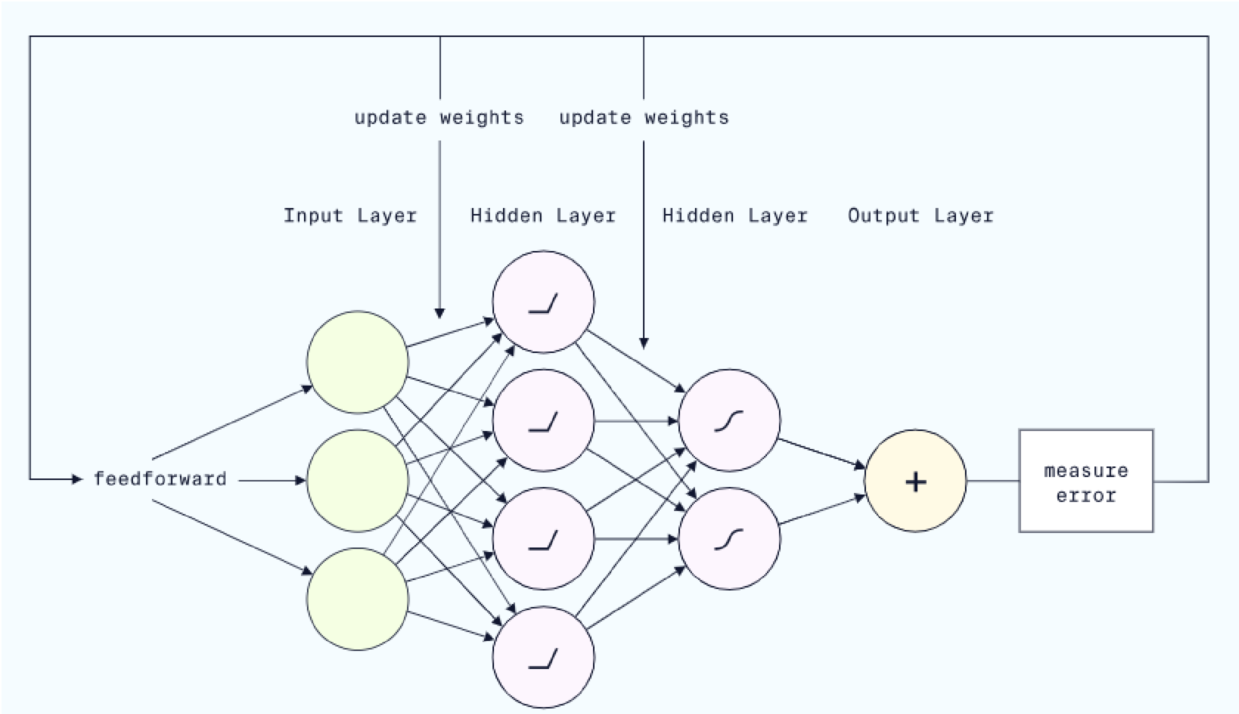
- If a number is negative ⇒ 0 else no change
- Can turn a node off (?)
    - I get weights 3 and -4
    - 3+(-4) = -1
    - relu(-1) = 0
    - That node is now, OFF

Other Activation Functions

- Linear: Just returns the input (also identity function)
- Binary Step: Node gives an output or no depends on a certain threshold amount
- Sigmoid: (more -ve) 0 → 1 (more +ve) [cant be 0 or 1, just btw them]
- tanh : -1 → 1
- Gaussian: a bell curve 0 → 1

Multi-Layered Networks

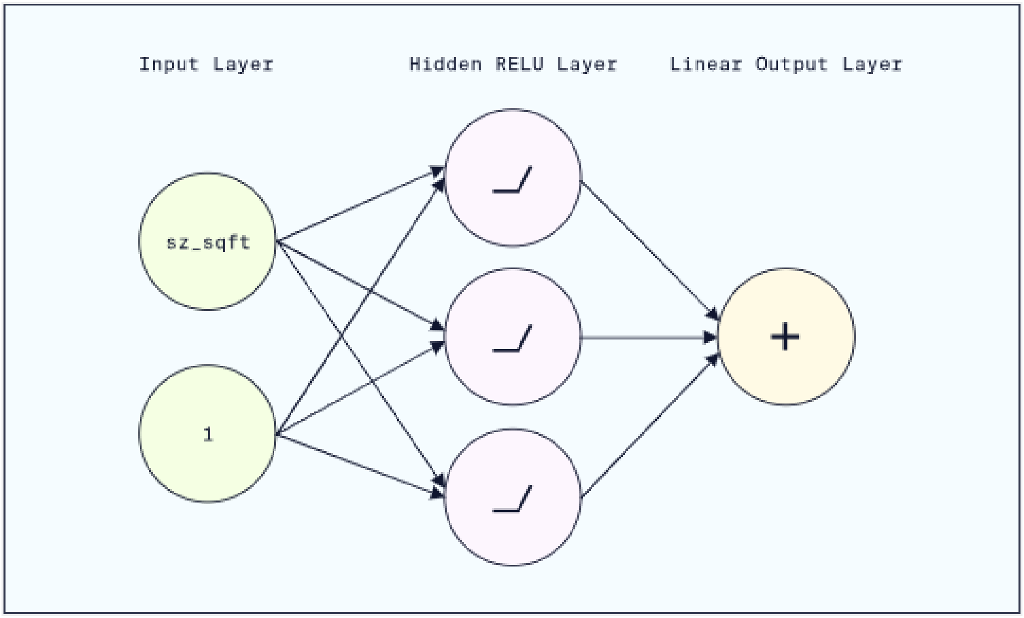
Hidden Layers:



- Inputs fed into input layer.
- The output from the InpurLayer goes to Hidden layer. Hidden Layer Nodes take the weighted sum. The ReLU is applied.
- It's output is sent to Hidden Layer2. Sigmoid function is applied
- The output of this goes to the OutputLayer where each of them are multiplied with a different weight
- Measure the error
- re-adjust the weights

Build a Sequential Neural Network

- `sequential`
    -




```
model = nn.Sequential(
                nn.Linear(2,3),
                nn.ReLU(),
                nn.Linear(3,1)
)
```

Note that the connections between layers must be properly aligned. Once we have defined `nn.Linear(2,3)` as the first layer, the next `nn.Linear()` must start with 3 nodes.

- `nn.Linear()` will use random weights and biases

FeedForward

- Just going through all layers once
-

```
# create apartment data
apts = np.array(
    [[100,3], # 100 years old, 3 bedrooms
    [50,4]]) # 50 years old, 4 bedrooms

# convert to a tensor
```

```
X = torch.tensor(apts, dtype=torch.float)

# run feedforward
model(X)

# output: the result of a feedforward through the network layers
>>>tensor([[-23.0715],
        [-11.8710]], grad_fn=<AddmmBackward0>)
```

Build A NN Class

- Why OOPS?

⇒ Helps to give me freedom of skipping a layer r Loop a layer

  - In Sequential i can only give input from one layer to another

  `model = nn.Sequential(<inputs>)`

  Sequential : type of NN (types of things == Classes)

  model is a specific *sequential* network (Instance of class)

1. Create the NN_Regression Class

```
class NN_Regression(nn.Module):
```

2. Initialize Network Components

- Define all the we are going to use ~ “Gather your ingredients”

> 💡 `__init__()` gathers different ingredient of neural network (layers and activation function)

```
def __init__(self):
    super(NN_Regression,self).__init__()
    self.layer1 = nn.Linear(3,16)
    self.layer2 = nn.Linear(16,8)
    self.layer3 = nn.Linear(8, 4)
  self.layer4 = nn.Linear(4, 1)
  self.relu = nn.ReLU()
```

3. Define the Forward Pass

💡 `forward()` defines the order in which input data flows through network

```
def forward(self, x):
    # define the forward pass
    x = self.layer1(x)
    x = self.relu(x)
    x = self.layer2(x)
    x = self.relu(x)
    x = self.layer3(x)
    x = self.relu(x)
    x = self.layer4(x)
    return x
```

4. Instantiate the Model

```
model = NN_Regression()
```

OR

```
class OneHidden(nn.Module):
    # add a new numHiddenNodes input
    def __init__(self, numHiddenNodes):
        super(OneHidden, self).__init__()
        # initialize layers
        # 3 input features, variable output features
        self.layer1 = nn.Linear(2, numHiddenNodes)
        # variable input features, 8 output features
        self.layer2 = nn.Linear(numHiddenNodes, 1)

        # initialize activation functions
        self.relu = nn.ReLU()

    def forward(self, x):
        ## YOUR SOLUTION HERE ##
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        return x

## YOUR SOLUTION HERE ##
model = OneHidden(4)

## do not modify below this comment

# create an input tensor
input_tensor = torch.tensor([3,4.5], dtype=torch.float32)

# run feedforward
predictions = model(input_tensor)

# show output
predictions
```

The Loss Function

- To track how bad our prediction was

- ⇒ 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 (sometimes called labels) the model is trying to predict

> 💡 Loss = (Actual - Pred)^2

Why squared? So i dont cancel out +ves and -ves

mean squared error

One data point

$$500 - 1000 = -500$$

For another data point

$$2000 - 1500 = 5002000 - 1500 = 500$$

$$500 + (-500) = 0$$

How can i get 0 loss?

hence MSE

$$((500 - 1000)^2 + (1500 - 1000)^2)/2 = 250,000$$

$$sqrt(25000) = 500$$

Selection of Loss Function

MSE: gives largest difference (*Can lead to overfitting)

MAE: if mse gived overfitting, you can use this.

```
loss = nn.MSELoss()
MSE = loss(predictions,y)
RMSE = MSE**(.5)
```

The Optimizer

- Loss only tells how good or bad our model does
- Optimizer **ADJUSTS** the weights to *improve* the model performance

Gradient descent

- I am on top of mountain and i need to go down. (making the loss function as small as possible)
- Its very dark and i cant see anything more than a meter
- So, I'll ASSESS the one meter around me, and walk in the direction that is "steepest"
- We reach that point and re-evaluate the situation the same manner

Learning Rate

- How far do I need to walk?
- Learning Rate High: If I walk too fast then I might miss the point
- Learning Rate Low: If I walk too slow then I might get stuck.

This is one of the **Hyper Parameter**

Hyper-parameter Tuning is adjusting these parameters to get best results

Optimizer in Pytorch

ADAM: uses gradient descent with few other things like (adjusting the LR while dynamically training)

```
import torch.optim as optim
optimizer = optim.Adam(model.parameters(), lr=0.01)
#model.parameters() => tells Adam about current weights and biases
```

1. Calculate the gradients of loss function (backwards pass) (determine the "downward" direction)
2. Step : update the weights and biases

```
# compute the loss
MSE = loss(predictions,y)
# backward pass to determine "downward" direction
MSE.backward()
# apply the optimizer to update weights and biases
optimizer.step()
```

Note that `backward` is applied to the computed loss, not the loss function

Training

```
predictions = model(x) #forward pass
loss = nn.MSELoss()
MSE = loss(predictions, y) #compute loss

optimizer = optim.Adam(model.parameters(), lr=0.01)

MSE.backward() #compute gradient
optimizer.step() #update weights and biases
```

Loop so that we can reduce loss as much as possible

- `optimizer.zero_grad()` resets the grad. At a new pt we want to find new direction to walk in
    - The gradients determine the direction to move the weights and biases, and we want to pick a brand new direction each time.
- Iteration in training loop == **Epoch**

```
loss = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

```
epochs = 1000
for i in range(epochs):
    predictions = model(x) #forward pass
    MSE = loss(predictions, y) #compute loss
    MSE.backward() #compute gradient
    optimizer.step() #update weights and biases
    optimizer.zero_grad() #reset grad for next iteration

    if (epoch + 1) % 100 == 0:   #as 0 indexing 100th => 99th
        print(f'Epoch [{epoch + 1}/{num_epochs}], MSE Loss: {MSE.it
        #MSE.item() so it will only print MSE
```

Testing and Evaluation

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
    train_size=0.80,
    test_size=0.20,
    random_state=2)
```

```
model.eval() #evaluation mode
with torch.no_grad(): #switch off grad calc, we dont need it in eva
    predictions = model(X_text)
    test_MSE = loss(predictions, y_test)
```

Saving and loading

```
# save the neural network to a specified path
torch.save(model, 'model.pth')
```

```
# load the saved neural network from the specified path
loaded_model = torch.load('model.pth')
```