

Linear regression and MLPs in Pytorch





Population - profit dataset

Population (in 10000s)	Profit (in \$100000s)
6.1101	17.592
5.5277	9.1302
8.5186	13.662
5.2524	-1.22



Pytorch Dataset

- Class that holds the data in our dataset
- Initialization code:
`__init__`
- Get the n'th example:
`__getitem__`
- Size of the dataset:
`__len__`

```
class PopulationProfitDataset(Dataset):  
  
    def __init__(self, array):  
        self.array = array  
  
    def __getitem__(self, index):  
        population = self.array[index][0]  
        profit = self.array[index][1]  
        return torch.Tensor([population]), torch.Tensor([profit])  
  
    def __len__(self):  
        return len(self.array)
```

Init method, called once at the start, when we create the dataset

Return index'th element of the dataset
`dataset[6]` calls `__getitem__(6)`

Return the size of our dataset



Pytorch DataLoader

- To train a model, we feed it a **bunch of examples at once** = a **mini-batch**
- **DataLoader** takes a **Dataset** and splits it into mini-batches
- Optionally, a dataloader can shuffle the data in the dataset

this runs the `__init__`
method and stores
`data_array` in the object

Python array that holds our data

```
- dataset = PopulationProfitDataset(data_array)
- dataloader = DataLoader(dataset, batch_size = 10, shuffle =
  False)
```



Problem to solve: estimate profit for any population count

- program that given a population count returns a profit value
- must work for population values not in the dataset
- dataset may contain contradictory information: profit doesn't only depend on population size
- dataset may contain noise: someone miscounted how much money that made

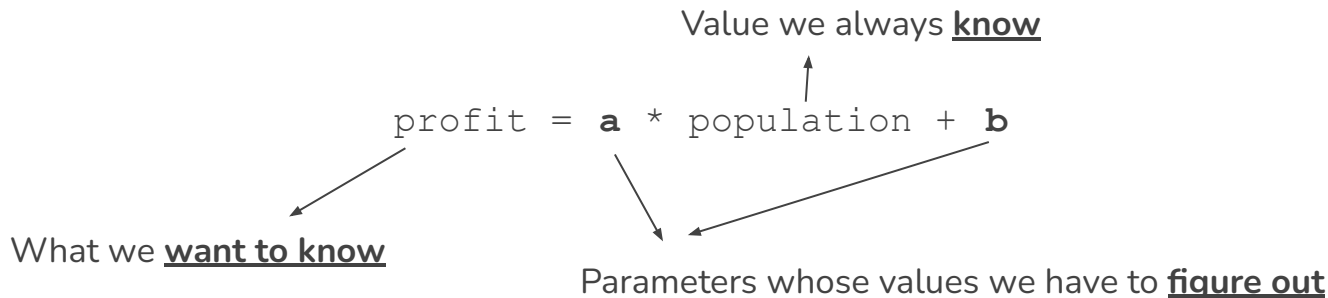
Learns the whole dataset perfectly, but won't work on anything else



```
def predict_profit(population):  
    if population == 6.1101:  
        return 17.592  
    elif population == 5.5277:  
        return 9.1302  
    elif population == 8.5186:  
        return 13.662:  
    elif population == 5.2524:  
        return -1.22  
    else:  
        return ???
```



Simplest model: linear regression





Linear regression in PyTorch

- `model = nn.Linear(1, 1)`
 $\text{profit} = a * \text{population} + b$
- `model.weight => tensor([[0.4872]], requires_grad=True)`
←
- `model.bias => tensor([0.9540], requires_grad=True)`
←
- notice: **weight** and **bias** have `requires_grad = True`
- that is because we will compute derivatives of the loss (`MSELoss()`) with respect to the weight and the bias



Judging how good a linear regression model is

- $\text{profit} = -0.1 * \text{population} + 0.5 \Rightarrow$ profit decreases when population increases ??
- $\text{profit} = \text{population} + 5 \Rightarrow$ profit of \$50.000 in an empty city ??
- $\text{profit} = 0.1 * \text{population} \Rightarrow$ profit is population divided by 10
- $\text{profit} = 0.1 * \text{population} - 5 \Rightarrow$ operate at a loss for small population



Loss functions, a clear way to judge how good a model is

- **Loss function** = error function = objective function = **how bad is our model?**
- model says: profit for population = 61101 is \$1.000.000
- dataset says: profit for population = 61101 is \$175.000
- model is way off => record a penalty in the loss function

- $$\frac{1}{n} \sum_{k=1}^n (\text{model}(x_k) - y_k)^2$$


Number of examples we're training the model on

Population value

Ground truth: the real profit obtained for population x_k



Mean squared error loss in Pytorch

- `loss_function = torch.nn.MSELoss()`  Creates the loss function
- `loss = loss_function(prediction, ground_truth)`



The loss. One singular value, meaning we can run **.backward()** on it



What our model says

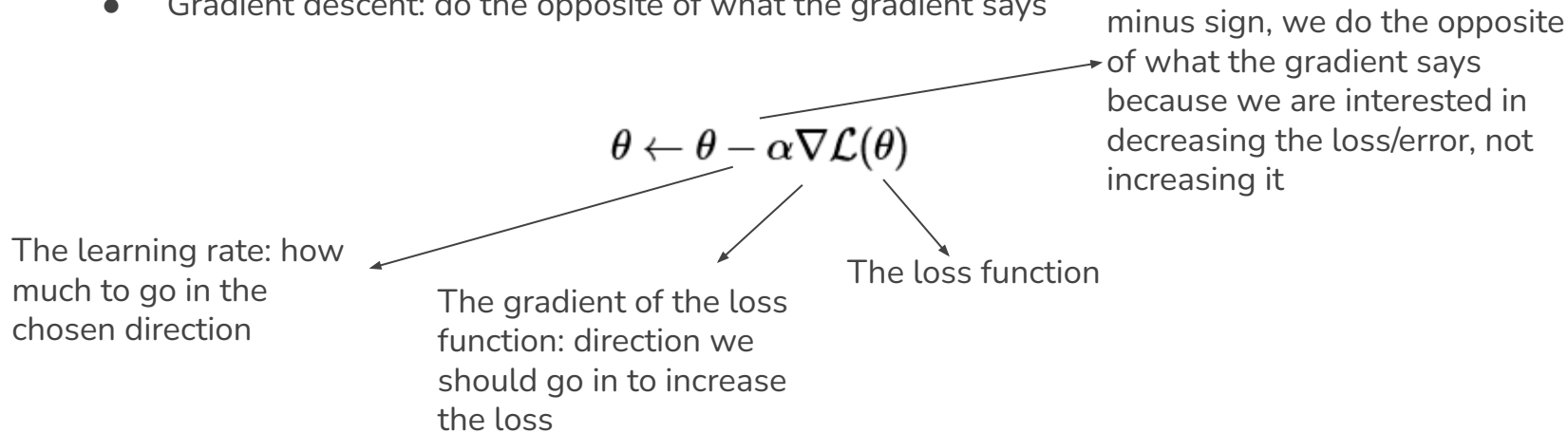


Reality: what we wish our model to say



Improving a model: gradient descent

- Loss function tells us how good / bad the model is at the moment
- Gradient of the loss function: what should we do to the parameters (increase / decrease them) to increase the loss
- Gradient descent: do the opposite of what the gradient says





Gradient descent in PyTorch

- `from torch.optim import SGD`
- `optimizer = SGD(model.parameters(), lr = 0.001)`



list of all the parameters of our model (`model.weight` and `model.bias` in our case)



Learning rate.
Rule of thumb: start small, increase if model improves too slowly, decrease if model goes nuts

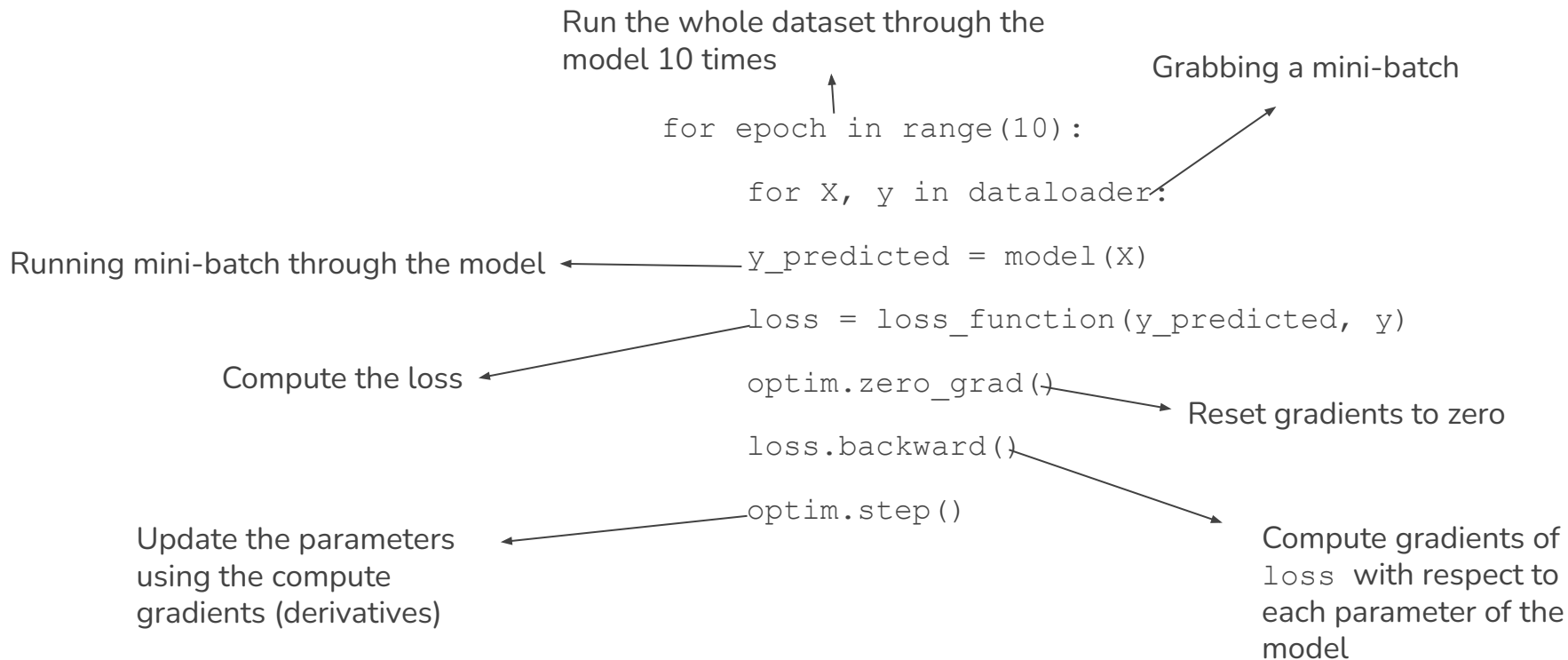


PyTorch training loop

- Step 1: obtain a mini-batch from the `DataLoader`
- Step 2: pass the mini-batch through the network, obtaining a prediction
- Step 3: compute the loss, by passing the prediction and the ground-truth to the loss function
- Step 4: reset gradients to 0 using `optimizer.zero_grad()`
- Step 5: compute the gradients of the parameters using `loss.backward()`
- Step 6: update the parameters using `optimizer.step()`
- Step 7: repeat!



PyTorch training loop





Final notes

- Our data tensors don't have **requires_grad = True**, because we don't need to compute derivatives / gradients with respect to them
- Usually only compute gradients with respect to something we can change (the parameters)
- Before using a PyTorch model (**nn.Linear**), optimizer (**optim.SGD**), loss function (**nn.MSELoss**) we first have to create them!
- **nn.MSELoss(y_predicted, y_real)** => incorrect
- **nn.MSELoss()(y_predicted, y_real)** => correct
- We always need: **a dataset, a dataloader, a model, an optimizer and a loss function!**



FashionMNIST dataset

- 60000 training images, 10000 test images
 - each image belongs to some category(**shirt, trouser etc.**)
 - **Problem to solve:** for any image, model must output its class
-
- Load this dataset in PyTorch: `mnist_train_dataset =
train_dataset = FashionMNIST(root = "/data", train =
True, transform = ToTensor(), download = True)`

t-shirt	0
trouser	1
pullover	2
dress	3
coat	4
sandal	5
shirt	6
sneaker	7
bag	8
ankle boot	9



Model input

- An image is represented as a matrix that contains all of the image's pixels
 - Our previous linear regression model doesn't seem to take in a matrix as input!
 - We **nn.Flatten()** the matrix into a vector of pixels!
-
- Our profit from population model outputs a **real value**: it can be -1, -1.54, 2.5, any other value => this is called **regression**
 - FashionMNIST => every image has one of 10 **classes** => we have a **finite count of reasonable outputs for our model** => this is called **classification**



Model output

- We have 10 classes, so our model will output 10 values!
- To output 10 values, we do 10 linear regressions “in parallel”

Model parameters: one for each pixel in the image

- $p_1 = a_{11} * p_1 + a_{12} * p_2 + \dots + a_{1n} * p_n + a_{10}$
- $p_2 = a_{21} * p_1 + a_{22} * p_2 + \dots + a_{2n} * p_n + a_{20}$
- ...

Probability that image
contains a shirt

Image pixels

Bias parameter



Our model in Pytorch

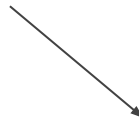
- `model = nn.Sequential(nn.Flatten(), nn.Linear(28*28, 10))`



Applies layers one after another



Image matrix to vector



28 by 28 images means $28 \times 28 = 784$ inputs to the model
This layer runs 10 linear regressions, obtaining 10 values
We interpret the values as probabilities: probability the image contains a shirt, probability the image contains trousers, etc.

- Access the linear layer: `model[1]`
- `model[1].weight.shape` => 10 x 784 matrix: 10 linear regressions, each with 784 inputs, therefore also 784 parameters
- `Model[1].bias.shape` => vector of 10 elements: 10 linear regression, each has one bias parameter



Our model output is not yet useable!

- Linear regressions: multiply some stuff and add the results => we can expect negative values and values greater than 1
- We want probabilities: our values must be positive, smaller than 1, and must add up to 1
- The **softmax** function:

$$\text{softmax}(o_1, o_2, \dots, o_{10}) = \left(\frac{e^{o_1}}{e^{o_1} + e^{o_2} + \dots + e^{o_{10}}}, \dots, \frac{e^{o_{10}}}{e^{o_1} + e^{o_2} + \dots + e^{o_{10}}} \right)$$

- In PyTorch: `model = nn.Sequential(nn.Flatten(), nn.Linear(28*28, 10), nn.Softmax())`



Cross-entropy loss function

- Our model outputs a **probability distribution**.
- If we train the model on an image of a **t-shirt**, we expect the model to output a probability of **1** for the **t-shirt**
- Say the 5th output of the model (**p_5**) corresponds to the **t-shirt** class
- We penalize the model by the quantity **-ln(p_5)**
- Let's analyze the penalty:
 - If the model says **p_5=1**, then the penalty is **-ln(p_5)=-ln(1)=-0=0** => no penalty for the model
 - If the model says **p_5=0.8**, then the penalty is **-ln(p_5)=-ln(0.8)=0.09** => some penalty, not too much, because model still says it's 80% sure we have a t-shirt
 - If the model says **p_5=0.1**, then the penalty is **-ln(p_5)=-ln(0.1)=1** => model gives a pretty bad answer, it's only 10% sure we have a t-shirt, so we give a much larger penalty

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{k=1}^n -\ln(p_{c_k})$$

p_{c_k} is our model's output for the class that the image belongs to

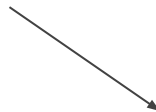


Cross-Entropy in Pytorch

- Initialize: `loss_function = nn.CrossEntropyLoss()`
- Usage: `loss_function([[1, 2, 3]], [1])`



The output of our model, **before Softmax**: `CrossEntropyLoss` applies **Softmax** itself



The expected class is **1**, which means our loss will only care about the value **2**

Question: our loss only cares about the probability the model gives to the correct class. Does that mean the model can output whatever it wants for the other classes??



Training the model!

```
for epoch in range(10):  
    for pixels_batch, classes_batch in mnist_train_loader:  
        predicted_probabilities = model(pixels_batch)  
        loss = loss_function(predicted_probabilities, classes_batch)  
        optimizer.zero_grad()  
        loss.backward()  
        optimizer.step()
```



Using the trained model

- `model(image) => pass an image to the model`
- `tensor([[-8.4489, -9.3227, -2.7202, -3.7881, -6.5726, 7.9927,
 -2.2151, 7.5391, 6.5984, 10.6699]])`



Index of the largest value is index of
the predicted class (t-shirt, for 9)



Deeper models: multilayer perceptrons

- Our classification model has 784 inputs (the pixels of the image) and 10 outputs
- We can take the 10 outputs and feed them to a similar model, that has 10 inputs and 10 outputs!
- `nn.Sequential(nn.Flatten(), nn.Linear(784, 100), nn.Linear(100, 10))`
- Good news: Pytorch code for training is the exact same!
- **Question:** this model has more parameters, because we have 2 linear layers (a linear layer essentially doing multiple linear regressions). But is our model stronger?



Representation power of a model

- How complex of a relationship can a model represent?
- A basic linear regression model can represent relationships of the form $a * \text{feature} + b$
- Chaining 2 linear regression models together: $a_1 (a_2 * \text{feature} + b_2) + b_1$
- Simplifying: $a_1 * a_2 * \text{feature} + a_1 * b_2 + b_1$
- Denoting: $a_3 = a_1 * a_2$, $b_3 = a_1 * b_2 + b_1$
- Chaining 2 linear regression models together leads us to: $a_3 * \text{feature} + b_3$
- **Chaining 2 linear regression models together produces a model of the same shape as a single linear regression: same representation power!**



Activation functions

- Need something in between 2 linear layers to allow the model to represent more complex relationships
- Can't be a linear function (like linear regression is)
- Must be a **non-linear** function
- Most common activation function: Rectified Linear Unit = **ReLU**
- **ReLU**: if the value is negative, replace it with 0, otherwise leave it unchanged



MLP with a ReLU function

- `nn.ReLU()`
- `nn.Sequential(nn.Flatten(), nn.Linear(784, 100), nn.ReLU(), nn.Linear(100, 10))`
- Can go deeper: `nn.Sequential(nn.Flatten(), nn.Linear(784, 100), nn.ReLU(), nn.Linear(100, 200), nn.ReLU(), nn.Linear(200, 100))`
- **Question:** should we place a **ReLU** function after the last layer of the model?
- **Extra question:** notice that when placing 2 linear layers one after the other, the second layer must have as many inputs as the first layer has outputs. In other words, we must have `nn.Linear(m, n)` followed by `nn.Linear(n, p)`. Does this remind you of anything?



Training a network on the GPU

- move the model itself to the GPU: `model.to("cuda")`
- this moves the model's parameters on the GPU
- the logic of the model will now run on the GPU
- our input must also be on the GPU
- in the training loop: `pixels_batch = pixels_batch.to("cuda")` and `classes_batch = classes_batch.to("cuda")`
- everything else remains unchanged



Conclusions

- Dataset: a bunch of examples that we want to learn from. Usually contains inputs (what we have to give the model, also called features; the population / image pixels in our case) and outputs (what we want the model to compute for us: profit / image class in our case)
- DataLoader: grab **n** examples at once
- Models: `nn.Sequential(nn.Linear(n, m), nn.ReLU(), nn.Linear(m, p))`
- Optimizer: `optim.SGD(model.parameters(), lr = 0.1)`
- Loss function: `nn.MSELoss()` for regression problems (model output is a real number: person's height, someone's age, a company's profit) and `nn.CrossEntropyLoss()` for classification problems (model output tells us that its input represents a dog or a cat: a finite amount of classes)
- Training loop: for however many epochs, grab all mini-batches using the dataloader. Pass them through the model. Use model output and correct output to compute loss. Reset gradients to 0 with `optimizer.zero_grad()`. Compute gradients using `loss.backward()`. Use gradients to make a step towards a better model: `optimizer.step()`