A deeper dive into loading data

```
# defining features

features = animals.iloc[:, 1:-1] # selects every column but first and last
X = features.to_numpy()

target = animals.iloc[:, -1] # select last column
y = target.to_numpy()

import torch
from torch.utils.data import TensorDataset

dataset = TensorDataset(torch.tensor(X), torch.tensor(y)) # instantiate

# access an individual sample
input_sample, label_sample = dataset[0]

from torch.utils.data import DataLoader
batch_size = 2 # how many samples in each interation
shuffle = True

dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)

for batch_inputs, batch_labels in dataloader:
    # print both for each iteration
```

Writing our first training loop

1. create a model
2. choose a loss function
3. define a dataset
4. set an optimizer
5. run a training loop (calculate loss, compute gradients, update model params)

We can use MSE loss for the regression loss function

```
dataset = TensorDataset(torch.tensor(features).float(),
torch.tensor(target).foat())

dataloader = DataLoader(dataset, batch_size=4, shuffle=True)

model = nn.Sequential(nn.Linear(4,2), nn.Linear(2,1))
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)
```

```
for epoch in range(num_epochs):
    for data in dataloader:
        # set gradients to zero
        optimizer.zero_grad()
        feature, target = data
        pred = model(feature)
        loss = criterion(pred, target)
        loss.backward()
        optimizer.step()
```

ReLU activation functions

ReLU = Rectified Linear Unit, it outputs the max value between its input and 0

For positive inputs, output=input
For negative inputs, output=0

```
relu = nn.ReLU()
```

Leaky ReLU is similar, but for negative inputs, it multiplies the input by a small coefficient (default 0.01)

```
leaky_relu = nn.LeakyReLU(negative_slope = 0.05)
```

Learning rate and momentum

We use SGD to train a neural network as an optimizer

```
sgd = optim.SGD(model.parameters(), lr=0.01, momentum=0.95)
# momentum adds inertia and avoids the model getting stuck, typically between
0.85 to 0.99
```

Momentum helps the optimizer move past a local minimum in a non-convex loss function