L04 Assignment: AWS MLU Lab Reflection

During this assignment, you are expected to have completed, or will be completing, the following AWS Labs:

1. Lab 01: Getting Started with PyTorch
2. Lab 02: How Neural Networks Learn
3. Lab 03: First Example of Neural Networks

Upon completion of these labs, your task is to compose a reflective journal detailing your experiences and insights gained from each lab.

Overview: This will discuss my experiences with the three labs.

Purpose: To report on my findings

Background Information: These labs are the first three labs in “Module 1: Neural Networks” of the AWS Academy Curriculum course “AWS MLU Application of Deep Learning to Text and Image Data”.

Specific Details: The run results from the labs are given in the Appendices.

Personal Reflection: One criticism I have is how inordinately long it takes AWS to provision the labs. Also the could make it easier for neophytes to go diractly to the notebooks.

Lab 1:

Lab 1 introduced me to the PyTorch deep learning framework. It covered Data manipulation, Exploring tensors, Indexing and slicing tensors, Tensor operations, Conversion to other Python objects, and Automatic differentiation (CPU vs GPU).

Frnakly, most of this was old hat, as it had been covered in previous courses. I did enjoy seening how I could enjoy taking advantage of GPUs in the SageMaker environment.

Lab 2:

Lab 2 introduced me to Neural Network Architecture. Topics included Simulating and plotting a dataset, Neural network basics, Implementing a neural network with PyTorch, initializing PyTorch values, Loss functions, Optimization methods, neural network training, and plotting Loss vs. Epoch.

Again, I have covered most of this in previous courses. I did like the graphic that described neural network basics, but mostly the exercise consisted of running notebook cells.

Lab 3:

Lab 3 consisted of building an End-to-End Neural Network Solution. We used the Austin Animal Center Dataset as input. Tasks included reading and processing the data; EDA,;cleaning (Remove HTML tags, remove punctuation, remove extra white space); split the input into train, validation, and test datasets; using pipelines to ensure that all the data is handled correctly (SimpleImputer and MinMaxScaler for numerical values, OneHotEncoder for categorical values and three transformer pipelines for the three text features); neural network training and validation, defining our own multilayer perceptron; and the neural network.

I very much enjoyed this as it covered some things I hadn’t done before – especially the data prep pipelines. Was happy to see accuracy, precision and recall scores of 83%.

I had problems when I tried to set this up on my own – mainly issues with CPU/GPU conflicts. Once I was shown to append `.to(device)` for creating my own perceptron, all was well.

Appendix I
Lab 1

Application of Deep Learning to Text and Image Data

Module 1, Lab 1, Notebook 1: Getting Started with PyTorch

This notebook will introduce you to the PyTorch deep learning framework, which is the tool that you will use throughout this course to implement neural network models. For more information, see the PyTorch documentation.

This notebook has been divided into two parts. In the first part of the notebook, you will learn how to use PyTorch to manipulate data to be ready for use in model training. In the second part, you will practice a crucial step in nearly all deep learning optimization algorithms: *differentiation*.

To learn these topics, you will examine how PyTorch stores and manipulates data in *n*-dimensional arrays, which are also called *tensors*. To define the tensors and arrays, you will use *NumPy*, which is the most widely used scientific computing package in Python.

Other frameworks are available, but this lab focuses on PyTorch, which has two key features. First, GPU is well-supported to accelerate the computation, whereas NumPy supports only CPU computation. Second, the tensor class supports automatic differentiation. These properties make the tensor class suitable for deep learning.

You will learn the following:

- How to explore tensors
- Why you index and slice tensors
- How to index and slice tensors
- Common tensor operations
- How to perform tensor operations on data
- How to convert tensors to other Python objects

You will be presented with activities throughout the notebook:

No coding is needed for an activity. You try to understand a concept, answer questions, or run a code cell.

Index

Data manipulation

In this section, you will practice basic data manipulation.

In [1]:
```
!cat requirements.txt
```

```
scikit-learn
torch
matplotlib
seaborn
numpy
pandas
```

In [ ]:
```
!pip install numpy
```

In [ ]:
```
!pip install pandas
```

In [ ]:
```
!pip install seaborn
```

In [ ]:
```
!pip install matplotlib
```

In [ ]:
```
!pip install scikit-learn
```

In [ ]:
```
!pip install torch
```

In [ ]:
```
# Install libraries
#!pip install -U -q -r requirements.txt
```

In [2]:
```
!ls
```

```
data        MLUDTI-EN-M1-Lab1-1.ipynb            requirements.txt
images      MLUDTI-EN-M1-Lab1-2.ipynb
lost+found  MLUDTI_EN_M1_Lab1_quiz_questions.py
```

In [3]:
```
# Import basic libraries to work with data and tensors
import numpy as np
import pandas as pd
import torch
```

In [4]:
```
# Import utility functions for activities
from MLUDTI_EN_M1_Lab1_quiz_questions import *
```

Exploring tensors

A tensor represents an array of numerical values. A one-axis tensor corresponds to a one-dimensional vector in math, and a two-axis tensor corresponds to a matrix. Tensors with more than two axes don't have special mathematical names.

You can use the `arange` operation to create a row vector x that contains the first 12 integers, starting with 0. Unless otherwise specified, a new tensor will be stored in main memory and designated for CPU-based computation.

Note: Integers are created as floats by default.

In [5]:
```
# Create a tensor with values in the range 0-11
x = torch.arange(12)

# Print the tensor
x
```

Out[5]:
```
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

You can view a tensor's shape (the length along each axis) by reviewing its `shape` property.

In [6]:
```
# Check for tensor size
x.shape
```

Out[6]:
```
torch.Size([12])
```

You can use the `reshape` function to transform the tensor x from a row vector with shape (12) to a matrix with shape (3, 4).

In [8]:
```
# Reshape the tensor into a matrix with three rows and four columns
x_reshaped = x.reshape(3, 4)

# Print the reshaped tensor
x_reshaped, x.reshape(3, 4)
```

Out[8]:
```
(tensor([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]]),
 tensor([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]]))
```

You can also reshape a tensor into a matrix by choosing one dimension and letting the other dimension be calculated implicitly (automatically). This is done by using -1 for the dimension that you want to be determined automatically.

For example, instead of calling `x.reshape(3,4)`, you can call either `x.reshape(-1,4)` or `x.reshape(3,-1)` to get the same result.

It's important to initialize a tensor in memory with the desired shape. You can do this by initializing with zeros, ones, other constants, or numbers that are randomly sampled from a specific distribution.

You can use the `zeros()` function to create a tensor with all elements set to 0 and a shape of (2, 3, 4).

In [9]:
```
# Create two matrices with three rows and four columns respectively and fill
zeros_tensor = torch.zeros((2, 3, 4))

# Print the matrices
zeros_tensor
```

Out[9]:
```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

Similarly, you can use the `ones()` function to create tensors where each element is set to 1.

In [10]:
```
# Create two matrices with three rows and four columns respectively and fill
ones_tensor = torch.ones((2, 3, 4))

# Print the matrices
ones_tensor
```

Out[10]:
```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

In other situations, you will want to create a tensor with randomly sampled values from a probability distribution.

For example, when you construct arrays to serve as parameters in a neural network, you typically initialize their values randomly. To do this, you can use the `randn()` function

to create a tensor where each of its elements is randomly sampled from a standard Gaussian (normal) distribution with a mean of 0 and a standard deviation of 1.

In [11]:
```
# Create a 3x4 matrix where the fill values are randomly sampled from a norm
x = torch.randn(3, 4)

# Print the matrix
x
```

Out[11]:
```
tensor([[ 0.6808, -0.8495,  1.8730, -0.5076],
        [ 1.1172, -0.4965,  1.7714, -0.0404],
        [ 0.2926, -0.0351, -0.9554,  0.9136]])
```

As you work with tensors, you will need to be able to verify that they have the expected shape, size, and type of stored values:

- Use the `shape` attribute to determine the shape of the tensor.
- Use the `numel()` function to determine the number of elements in the tensor. This is equal to the product of the components of the shape.
- Use the `.dtype` attribute to determine the data type of the stored values.

In [12]:
```
# Check shape (rows and columns), total number of elements, and what data ty
x.shape, x.numel(), x.dtype
```

Out[12]:
```
(torch.Size([3, 4]), 12, torch.float32)
```

Try it yourself!

To test your understanding of basic tensor functionality, run the following cell.

In [13]:
```
# Run this cell to display the question and check your answer
question_1

# For help, read the section "Exploring Tensors".
```

Out[13]:

Which option would you use to create a 2x4 matrix that is filled with 0s?

- torch.ones(2,4)
- torch.zeros(4,2)
- torch.zeros(2,4)

Submit

Indexing and slicing tensors

You can access elements in a tensor by index the same way that you would access elements in a Python array.

In [14]:
```
# Create a 3x4 matrix where the fill values are randomly sampled from a norm
x = torch.randn(3, 4)
```

```
# Print the full tensor, the last row, and the last two rows
x, x[-1], x[1:3]
```

Out[14]:
```
(tensor([[-0.9841,  0.2894,  0.1126,  0.0540],
         [ 0.6062, -0.3289, -0.9334,  0.8345],
         [-0.1069,  0.3013,  0.4852, -0.3762]]),
 tensor([-0.1069,  0.3013,  0.4852, -0.3762]),
 tensor([[ 0.6062, -0.3289, -0.9334,  0.8345],
         [-0.1069,  0.3013,  0.4852, -0.3762]]))
```

You can access values for a specific location by specifying all of the location indices.

In [16]:
```
x[0, 0], x[2, 2]
```

Out[16]:
```
(tensor(-0.9841), tensor(0.4852))
```

You can also write elements of a matrix by specifying indices.

In [17]:
```
# Assign a specific value in a given row and column
x[1, 2] = 9

# Print the tensor with the newly assigned value
x
```

Out[17]:
```
tensor([[-0.9841,  0.2894,  0.1126,  0.0540],
        [ 0.6062, -0.3289,  9.0000,  0.8345],
        [-0.1069,  0.3013,  0.4852, -0.3762]])
```

You can also use multidimensional slicing to replace numbers inside the tensor.

In [18]:
```
# Assign the same value to a slice of rows and columns
x[0:2, :] = 12

# Print the tensor with the newly assigned values
x
```

Out[18]:
```
tensor([[12.0000, 12.0000, 12.0000, 12.0000],
        [12.0000, 12.0000, 12.0000, 12.0000],
        [-0.1069,  0.3013,  0.4852, -0.3762]])
```

Try it yourself!

To test your understanding of indexing and slicing tensors, run the following cell.

In [19]:
```
# Run this cell to display the question and check your answer
question_2

# x[0] would return the first row, not the very first element.
# x[1] would return the second row. Keep in mind, indexing starts with 0 in
# For help, read the section "Indexing and Slicing Tensors".
```

Out[19]:

Which option would you use to access the first element that is stored in tensor x with shape 4x6?

x[1]

x[0,0]

x[0]

Submit

Tensor operations

You can use common arithmetic operators (`+` , `-` , `*` , `/` , and `**`) for element-wise operations on any identically shaped tensors of arbitrary shape.

In the following example, a five-element tuple is created where each element is the result of an element-wise operation.

In [20]:
```
# Create two tensors: x and y filled with some values
x = torch.tensor([1.0, 2.0, 4.0, 8.0])
y = torch.tensor([2.0, 2.0, 2.0, 2.0])

# Sum, difference, element-wise multiplication, division, exponentiation (op
x + y, x - y, x * y, x / y, x**y
```

Out[20]:
```
(tensor([ 3.,  4.,  6., 10.]),
 tensor([-1.,  0.,  2.,  6.]),
 tensor([ 2.,  4.,  8., 16.]),
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
 tensor([ 1.,  4., 16., 64.]))
```

You can apply many more element-wise operations, including unary operators such as exponentiation.

In [21]:
```
# Calculate the exponential for each element in the tensor
torch.exp(x)
```

Out[21]:
```
tensor([2.7183e+00, 7.3891e+00, 5.4598e+01, 2.9810e+03])
```

In addition to element-wise computations, you can calculate linear algebra operations including vector dot products and matrix multiplication.

The dot product is an important concept in ML because it can be used to quantify similarity.

In [22]:
```
# Calculate the dot product between two tensors
torch.dot(x, y)
```

Out[22]:
```
tensor(30.)
```

You can also calculate the dot product of two vectors manually by performing an element-wise multiplication and then summing the result.

In [23]:
```
# Calculate the dot product and sum for two tensors
torch.sum(x * y)
```

Out[23]:
```
tensor(30.)
```

To perform matrix multiplication of tensors, PyTorch offers the `matmul()` function.

In [24]:
```
# Product of a 3x4 matrix and 4x1 vector
A = torch.arange(12, dtype=torch.float).reshape(3, 4)
print(A)

x = torch.arange(4, dtype=torch.float)
print(x)

# Perform the matrix multiplication
torch.matmul(A, x)
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
tensor([0., 1., 2., 3.])
```

Out[24]:
```
tensor([14., 38., 62.])
```

In [25]:
```
# The matmul can be calculated several ways
A.matmul(x)
```

Out[25]:
```
tensor([14., 38., 62.])
```

Note that matrix multiplication is not communicative. If you have matrices that aren't shaped properly for multiplication, you will receive a runtime error.

In [26]:
```
# Uncomment and run the following cell. You will get a runtime error.

torch.matmul(x, A)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[26], line 3
      1 # Uncomment and run the following cell. You will get a runtime erro
r.
----> 3 torch.matmul(x, A)

RuntimeError: mat1 and mat2 shapes cannot be multiplied (1x4 and 3x4)
```

PyTorch helps you do complex operations. You can create tensors, reshape them, and then multiply them to get the result with a few lines of code. These operations will make it easier for you to train and validate a model.

In [27]:
```
# Initialize matrix A
A = torch.arange(12).reshape(6, 2)

# Initialize matrix B
B = torch.arange(10).reshape(2, 5)

# Calculate the matrix multipliation between A and B and the resulting shape
C = torch.matmul(A, B)
print(C)
print(C.shape)
```

```
tensor([[  5,   6,   7,   8,   9],
        [ 15,  20,  25,  30,  35],
        [ 25,  34,  43,  52,  61],
        [ 35,  48,  61,  74,  87],
        [ 45,  62,  79,  96, 113],
        [ 55,  76,  97, 118, 139]])
torch.Size([6, 5])
```

For more information about linear algebra operations, see Linear Algebra on the Dive into Deep Learning site.

For more information about the PyTorch `matmul`, `dot`, and `mm` operations, see the PyTorch documentation: matmul, dot, and mm.

Try it yourself!

To test your understanding of tensor operations, run the following cell.

In [28]:
```
# Run this cell to display the question and check your answer
question_3
```

Out[28]:

True or false? Matrix multiplication is commutative, so torch.matmul(A, B) == torch.matmul(B, A).

True | False

Submit

Conversion to other Python objects

It's important to be able to convert between PyTorch and NumPy tensors. When you convert between different types, they don't share memory. This means that you need more memory resources; however, computations are not halted when different computations need to be performed on the CPU compared to the GPU. Because they don't share memory, no wait time occurs while deciding whether the NumPy package or PyTorch needs to perform an operation because they aren't using the same chunk of memory.

In [29]:
```
# Create a NumPy tensor
A = x.numpy()

# Convert the NumPy tensor to a PyTorch tensor
B = torch.tensor(A)

# Print the resulting types
type(A), type(B)
```

Out[29]:
```
(numpy.ndarray, torch.Tensor)
```

To convert a size-1 tensor to a Python scalar, you can use the `item` function or one of Python's built-in functions.

In [30]:
```
a = torch.tensor([3.5])

a, a.item(), float(a), int(a)
```

Out[30]:
```
(tensor([3.5000]), 3.5, 3.5, 3)
```

Automatic differentiation

In this section, you will practice automatic differentiation and see how to use PyTorch to take advantage of GPUs.

You can train a deep learning model on a CPU or GPU. The most computationally demanding piece in a neural network is multiple matrix multiplications. In general, when training on a CPU, each operation will be done sequentially. When using a GPU, all the operations will be done in parallel, which makes GPU faster than CPU.

CUDA is a parallel computing platform that focuses on general computing on GPUs. PyTorch natively supports CUDA, and you can access it with the `torch.cuda` library. To find out whether you have a GPU at your disposal and set your device accordingly, you can use `cuda.is_available()`.

In [31]:
```
# Set to GPU if GPU is available; otherwise, use CPU
device = "cuda" if torch.cuda.is_available() else "cpu"

# Print device type for reference
device
```

Out[31]:
```
'cuda'
```

PyTorch can allocate the tensors to the GPU on object creation by specifying the `device` parameter.

In [32]:
```
# Create a tensor and allocate memory with 'requires_grad', store on GPU
a = torch.arange(4, requires_grad=True, dtype=torch.float, device=device)
a
```

Out[32]:
```
tensor([0., 1., 2., 3.], device='cuda:0', requires_grad=True)
```

Differentiation is a crucial step in nearly all deep learning optimization algorithms. In this section, you will examine how PyTorch's automatic differentiation expedites this work by automatically calculating derivatives, which enables the system to backpropagate gradients.

Consider an example where you want to differentiate a function $f(\mathbf{x}) = 0.6x^2$ with respect to parameter x. Start by assigning an initial value of x.

In [33]:
```
# Print tensor that was created in the previous section
x
```

Out[33]:
```
tensor([0., 1., 2., 3.])
```

Before you calculate the gradient of $f(x)$ with respect to x, you need a place to store it.

It's important not to allocate new memory every time you take a derivative with respect to a parameter because the same parameters might be updated thousands or millions of times. This will cause memory to run out.

Note: A gradient of a scalar-valued function with respect to a vector x is itself vector valued and has the same shape as x.

In [34]:
```
# Allocate memory for a tensor's gradient by invoking .requires_grad
# Note that the tensor can be created already with an attached gradient by w
x.requires_grad_(True)

# After calculating the gradient taken with respect to x, you can access it
# The grad attribute's values are initialized with None
x.grad
```

Now, calculate $f(x)$.

In [35]:
```
# Calculate the dot product and multiply with .6 (as in the toy function exa
y = 0.6 * torch.dot(x, x)
```

```
# Print new tensor
y
```

Out[35]:
```
tensor(8.4000, grad_fn=<MulBackward0>)
```

Next, you can automatically calculate the gradient of $f(x)$ with respect to each component of x by calling the function for backpropagation and printing the gradient.

In [36]:
```
# Calculate the gradient
y.backward()

# Print the gradient values
x.grad
```

Out[36]:
```
tensor([0.0000, 1.2000, 2.4000, 3.6000])
```

Now, determine if this is the expected output. The gradient of the function $f(x) = 0.6x^2$ with respect to x should be $1.2x$.

Verify that the desired gradient was calculated correctly.

In [37]:
```
# Check if the calculated gradient matches the manual calculation
x.grad == 1.2 * x
```

Out[37]:
```
tensor([True, True, True, True])
```

Conclusion

In this notebook, you practiced using PyTorch to perform different mathematical calculations.

Next lab

In the next lab, you will learn the basics of neural networks and train your first one.

Appendix II
Lab 2

Application of Deep Learning to Text and Image Data

Module 1, Lab 1, Notebook 2: Examining a Neural Network Architecture

In this notebook, you will implement a minimum viable neural network to see the different architecture components.

The simplest possible neural network architecture is logistic regression. This lab will cover data ingestion, how to define the model, loss function, and the optimization algorithm. Although modern deep learning frameworks can automate nearly all of this work, you will only using matrix multiplications and automatic differentiation to build, train, and test a logistic regression model.

You will do the following:

- Generate a simulated dataset
- Use basic components of a neural network
- Implement a neural network by using PyTorch
- Train a neural network

You will be presented with activities throughout the notebook:

No coding is needed for an activity. You try to understand a concept, answer questions, or run a code cell.

Index

Simulated dataset

In this example, you will train a neural network on a dataset that is randomly generated. The dataset will have two classes, and you will train the neural network to classify them.

In [ ]:
```
!cat requirements.txt
```

In [ ]:
```
!pip install numpy
```

In [ ]:
```
!pip install pandas
```

In [ ]:
```
!pip install scikit-learn
```

In [ ]:
```
!pip install matplotlib
```

In [ ]:
```
!pip install seaborn
```

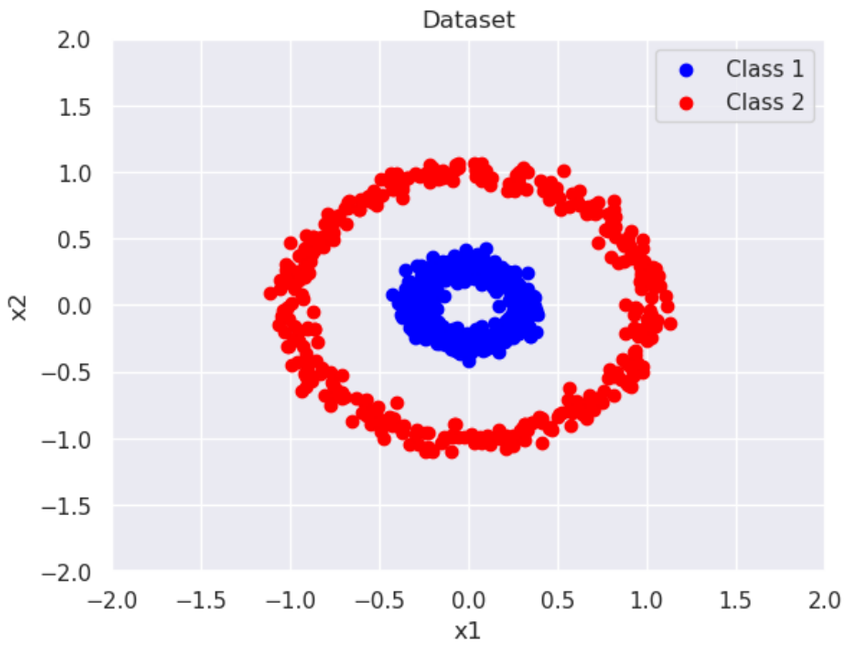
In [ ]:
```
!pip install torch
```

In [ ]:
```
# Install libraries
#!pip install -U -q -r requirements.txt
```

In [1]:
```
# Load the sample data
from sklearn.datasets import make_circles

# Specify settings, including how many examples to extract
X, y = make_circles(
    n_samples=750, shuffle=True, random_state=42, noise=0.05, factor=0.3
)
```

Plot the simulated dataset.

In [2]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns


def plot_dataset(X, y, title):

    # Activate the Seaborn visualization
    sns.set()

    # Plot both classes: Class 1 is blue, Class 2 is red
    plt.scatter(X[y == 1, 0], X[y == 1, 1], c="blue", label="Class 1")
    plt.scatter(X[y == 0, 0], X[y == 0, 1], c="red", label="Class 2")
    plt.legend(loc="upper right")
    plt.xlabel("x1")
    plt.ylabel("x2")
    plt.xlim(-2, 2)
    plt.ylim(-2, 2)
    plt.title(title)
    plt.show()


plot_dataset(X, y, title="Dataset")
```

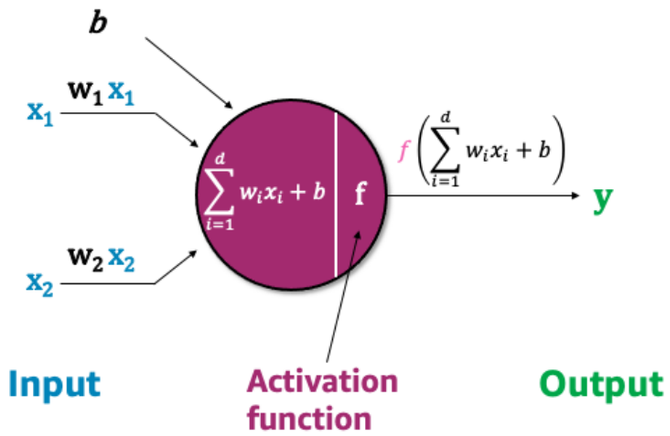


The goal is to build a neural network that can differentiate between the two classes in the dataset. The simplest neural network that can tackle this problem is a single-layer neural network, basically a logistic regression.

In [3]:
```
import torch

# Use a GPU resource if available; otherwise, use CPU
device = "cuda" if torch.cuda.is_available() else "cpu"

# To work with PyTorch, you need to convert the dataset to tensors first
X = torch.tensor(X, dtype=torch.float32).to(device)
y = torch.tensor(y, dtype=torch.float32).reshape((-1, 1)).to(device)
```

Neural network basics

The fundamental building blocks of neural networks are *neurons*, which are functions that are followed by an activation function. It's common for the initial function to be a linear regression and the activation function to be a sigmoid.



Your first goal is to implement a simple neural network with one neuron that uses the sigmoid function as the activation function to predict class 1 or class 2 from the sample dataset.

As an equation, this would look like the following: For some input $\mathbf{X}$ and output $\mathbf{y}$, the logistic regression model is defined as:

$$\hat{\mathbf{y}} = \sigma(\mathbf{X}\mathbf{w} + \mathbf{b}),$$

with some initial choices for the parameters, $\mathbf{w}$ weights matrix and bias $\mathbf{b}$.

You don't know what the best values for $\mathbf{w}$ and $\mathbf{b}$ are, so initialize the weights matrix $\mathbf{w}$ at random with zero mean and standard deviation 1, and let the bias take the initial value 0.

In [4]:
```
# Define the logistic regression
def log_reg(X, w, b):
    return torch.sigmoid(torch.matmul(X, w) + b)


# Define the basic neural network
net = log_reg

# Initialize for w and b
w = torch.normal(0, 1, size=(2, 1), requires_grad=True).to(device)
b = torch.zeros(1, requires_grad=True).to(device)

# To test the neural net, pass in an example data point
print(
    f"For datapoint 0, the probability of being class 1 is {float(net(X[0], w, b).item()
)
```

```
For datapoint 0, the probability of being class 1 is 0.44.
```

This is a basic single-layer neural network that hasn't been trained yet. Ultimately, the goal is to find the best possible values for $\mathbf{w}$ and $\mathbf{b}$. You need to choose a loss function that systematically evaluates how good the predictions are and an optimization method that updates the weight and bias values. You can do this manually, but PyTorch has built-in features to do this automatically.

In [5]:
```
# Import system library and append path
import sys

sys.path.insert(1, "..")

# Import utility functions that provide answers to challenges
from MLUDTI_EN_M1_Lab1_quiz_questions import *
```

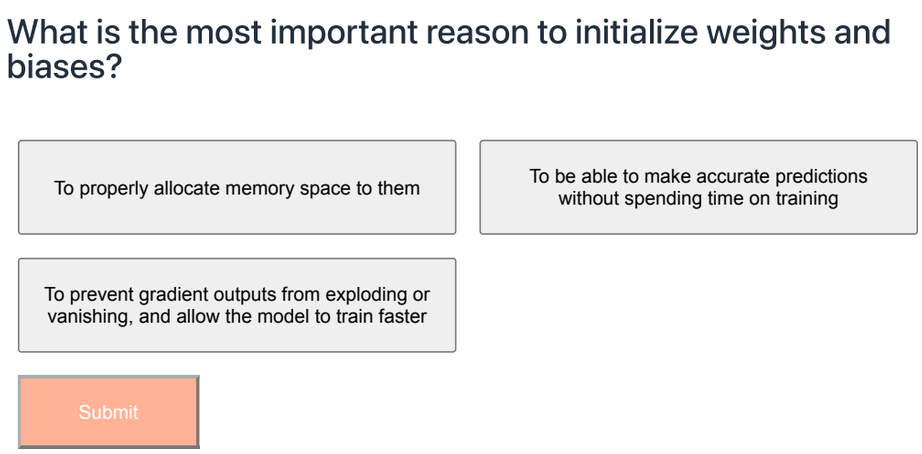
Try it yourself!

Activity

It's time to check your knowledge. To load the question, run the following cell.

In [6]:
```
# Run this cell to display the question and check your answer
question_4
```

Out[6]:



Implementing a neural network with PyTorch

Now you need to build, train, and validate a neural network in PyTorch. With PyTorch, you can list the different layers and activation functions that you want to use, in the sequence that you want them to run. For more information, see the PyTorch documentation.

You can use the `Sequential()` function to define the functions in the order that you want them to run. The first row refers to the first function to be used, followed by an activation function. You can chain together as many functions as you want to create your final model output. For more information, see torch.nn Sequential in the PyTorch documentation.

In [7]:
```
from torch import nn

# Create a sequential container that chains a linear regression function with a sigmoid
net = nn.Sequential(
    nn.Linear(2, 1),  # Linear layer-1 with 1 out_features and input size 2
    nn.Sigmoid(),  # Sigmoid activation function
)
```

Initialization

Before you continue, you need to initialize PyTorch values. This is important for the same reason that you initialized values for $\mathbf{w}$ and $\mathbf{b}$.

Picking the starting point is critical. Researchers have developed several initialization strategies that you can use. The *Xavier initialization* is commonly used because it can keep the scale of gradients roughly the same in all the layers, which helps to keep the gradient from vanishing or exploding.

For a list of initializers, see the PyTorch documentation. For information about Xavier initialization, see Xavier Initialization on the Dive into Deep Learning site.

In [8]:
```
# Initialize the weights with an Xavier initializer
def xavier_init_weights(m):
    if type(m) == nn.Linear:
        torch.nn.init.xavier_uniform_(m.weight)
        torch.nn.init.zeros_(m.bias)

# Apply the initialization to the sequential network that you created earlier
net.apply(xavier_init_weights)
```

Out[8]:
```
Sequential(
  (0): Linear(in_features=2, out_features=1, bias=True)
  (1): Sigmoid()
)
```

Loss function

Now that you have set up a neural network, you need to select a loss function to quantify how good a given selection of parameters are. Many loss functions exist, and you need to select one that suits the prediction problem, which is classificiation in this case. For information about loss functions, see the Dive into Deep Learning site.

Binary cross-entropy loss (log loss) is a loss function that is commonly used for binary classification:

```
loss = nn.BCELoss()
```

During the training of the neural network, the initial model parameters will be updated until model predictions fit the data sufficiently well. One way to control for this is by setting the number of epochs that the model trains for. If large changes occur in the output quality, you can increase the number of epochs and train the model for longer.

For a full list of supported loss functions, see torch.nn Loss Functions in the PyTorch documentation.

Optimization method

Each update requires taking the gradient of the loss function with respect to the parameters. Automatic differentiation is used to compute the gradient, and given this gradient, each parameter is updated in the direction that might reduce the loss.

The `torch.optim` module provides necessary optimization algorithms for neural networks. You can use an optimizer to train a network by using the stochastic gradient descent (SGD) method and setting the learning rate at 0.001. For more information, see Stochastic Gradient Descent on the Dive into Deep Learning site.

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

Three lines of code are required to perform a gradient descent update:

```
loss.backward() # Compute updates for each parameter
optimizer.step() # Make the updates for each parameter
optimizer.zero_grad() # Clean-up step for PyTorch
```

Training of the neural network

Now that you have all the components, you can create a loop that takes a given set of parameter values, creates outputs, evaluates the performance, and updates the parameters accordingly until it has completed a set number of iterations (epochs). Repeating the training for more epochs should further improve the model. However, some additional considerations impact how much the model will improve. One example is the number of data points that you want to evaluate per iteration.

Start by looking at the basic network again.

In [9]:
```
# Print network
net.to(device)
```

Out[9]:
```
Sequential(
  (0): Linear(in_features=2, out_features=1, bias=True)
  (1): Sigmoid()
)
```

Next, specify the loss function, optimization method, and how many epochs (loops) are run to update the parameters.

In [10]:
```
num_epochs = 50  # Total number of epochs (loops)

# Define the loss. Because you used sigmoid in the last layer, use nn.BCELoss.
# Otherwise, you could have used nn.BCEWithLogitsLoss. This loss combines a sigmoid laye
loss = nn.BCELoss(reduction="none")

# Define the optimizer, SGD with learning rate
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
```

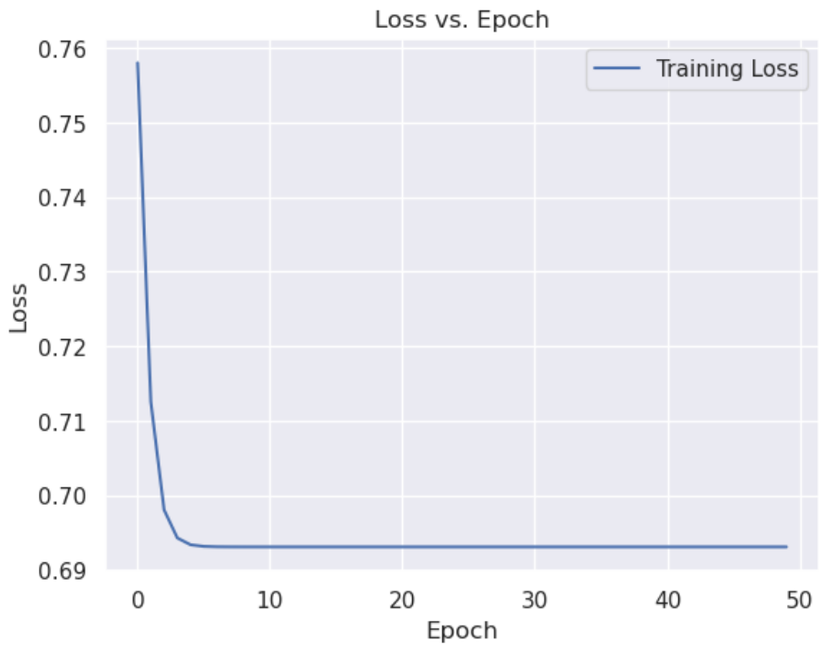
Finally, it is time for training!

Training will run through the dataset 50 times, and print training and validation losses after each epoch.

In [11]:
```
train_losses = []
for epoch in range(num_epochs):
    training_loss = 0
    # Zero the parameter gradients
    optimizer.zero_grad()
    output = net(X)
    L = loss(output, y).sum()
    training_loss += L.item()
    L.backward()
    optimizer.step()
    training_loss = training_loss / len(y)
    train_losses.append(training_loss)
```

Now that training has completed, you can plot the training and validation loss plots.

In [12]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns

plt.plot(train_losses, label="Training Loss")
plt.title("Loss vs. Epoch")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Notice that the loss (errors) decreases as the training process continues, as expected.

One final step is to compare this plot to the validation loss to see if it is overfitting the model.

Conclusion

In this notebook, you learned about training a neural network. You should now understand the basic steps of building a neural network and how to evaluate its performance.

Next lab

In the next lab, you will learn about the multilayer perceptron, which is the simplest feed-forward neural network architecture, and how to use dropout layers to prevent overfitting.

Appendix II
Lab 3

Application of Deep Learning to Text and Image Data

Module 1, Lab 3: Building an End-to-End Neural Network Solution

In the previous lab, you used a neural network with image data to predict the category that an item belonged to. In this lab, you will process text data by building an end-to-end neural network solution. The solution will incorporate all the data processing techniques that you have learned so far.

You will learn how to do the following:

- Import and preprocess data.
- Create a neural network with multiple layers.
- Train text data with your neural network.
- Validate your model as you train.
- Change different parameters to improve your neural network.

Austin Animal Center Dataset

In this lab, you will work with historical pet adoption data in the Austin Animal Center Shelter Intakes and Outcomes dataset. The target field of the dataset (**Outcome Type**) is the outcome of adoption: 1 for adopted and 0 for not adopted. Multiple features are used in the dataset.

Dataset schema:

- **Pet ID:** Unique ID of the pet
- **Outcome Type:** State of pet at the time of recording the outcome (0 = not placed, 1 = placed). This is the field to predict.
- **Sex upon Outcome:** Sex of pet at outcome
- **Name:** Name of pet
- **Found Location:** Found location of pet before it entered the shelter
- **Intake Type:** Circumstances that brought the pet to the shelter
- **Intake Condition:** Health condition of the pet when it entered the shelter

- **Pet Type:** Type of pet
- **Sex upon Intake:** Sex of pet when it entered the shelter
- **Breed:** Breed of pet
- **Color:** Color of pet
- **Age upon Intake Days:** Age (days) of pet when it entered the shelter
- **Age upon Outcome Days:** Age (days) of pet at outcome

You will be presented with two kinds of exercises throughout the notebook: activities and challenges.

Activity	**Challenge**
No coding is needed for an activity. You try to understand a concept, answer questions, or run a code cell.	Challenges are where you can practice your coding skills.

Index

Data processing

The first step is to process the dataset.

In [1]:
```
!cat requirements.txt
```

```
scikit-learn==1.3.2
torch==2.0.1
torchvision==0.15.2
matplotlib==3.7.2
seaborn==0.13.1
numpy==1.23.5
pandas==2.0.3
nltk==3.8.1
```

In [2]:
```
!pip install nltk
```

```
Requirement already satisfied: nltk in /home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packages (3.8.1)
Requirement already satisfied: click in /home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packages (from nltk) (8.1.7)
Requirement already satisfied: joblib in /home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packages (from nltk) (1.3.2)
Requirement already satisfied: regex>=2021.8.3 in /home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packages (from nltk) (2023.12.25)
Requirement already satisfied: tqdm in /home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packages (from nltk) (4.66.1)
```

In [3]:
```
!pip install boto3
```

```
Requirement already satisfied: boto3 in /home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packages (1.34.3)
Requirement already satisfied: botocore<1.35.0,>=1.34.3 in /home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packages (from boto3) (1.34.3)
Requirement already satisfied: jmespath<2.0.0,>=0.7.1 in /home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packages (from boto3) (1.0.1)
Requirement already satisfied: s3transfer<0.10.0,>=0.9.0 in /home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packages (from boto3) (0.9.0)
Requirement already satisfied: python-dateutil<3.0.0,>=2.1 in /home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packages (from botocore<1.35.0,
>=1.34.3->boto3) (2.8.2)
Requirement already satisfied: urllib3<2.1,>=1.25.4 in /home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packages (from botocore<1.35.0,>=1.34.
3->boto3) (1.26.18)
Requirement already satisfied: six>=1.5 in /home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packages (from python-dateutil<3.0.0,>=2.1->botoco
re<1.35.0,>=1.34.3->boto3) (1.16.0)
```

In [4]:
```
# Install libraries
#!pip install -U -q -r requirements.txt
```

In [5]:
```
# Import the dependencies
import boto3
import os
from os import path
import pandas as pd

%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import re, string
import nltk
from nltk.stem import SnowballStemmer
from sklearn.model_selection import train_test_split
```

```
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, MinMaxScaler
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.utils import shuffle
import torch
from torch import nn

from MLUDTI_M1_Lab3_neural_network import NeuralNetwork
```

First, read the dataset into a DataFrame and look at it. The data might look familiar because it was used in the labs of the Machine Learning through Application course.

In [6]:
```
df = pd.read_csv("data/austin-animal-center-dataset.csv")

print("The shape of the dataset is:", df.shape)
```

```
The shape of the dataset is: (95485, 13)
```

In [7]:
```
# Print the first and last five rows of the dataset
df
```

Out[7]:

	Pet ID	Outcome Type	Sex upon Outcome	Name	Found Location	Intake Type	Intake Condition	Pet Type	Sex upon Intake	Breed	Color	Age upon Intake Days	Age upon Outcome Days
0	A794011	1.0	Neutered Male	Chunk	Austin (TX)	Owner Surrender	Normal	Cat	Neutered Male	Domestic Shorthair Mix	Brown Tabby/White	730	730
1	A776359	1.0	Neutered Male	Gizmo	7201 Levander Loop in Austin (TX)	Stray	Normal	Dog	Intact Male	Chihuahua Shorthair Mix	White/Brown	365	365
2	A674754	0.0	Intact Male	NaN	12034 Research in Austin (TX)	Stray	Nursing	Cat	Intact Male	Domestic Shorthair Mix	Orange Tabby	6	6
3	A689724	1.0	Neutered Male	*Donatello	2300 Waterway Bnd in Austin (TX)	Stray	Normal	Cat	Intact Male	Domestic Shorthair Mix	Black	60	60
4	A680969	1.0	Neutered Male	*Zeus	4701 Staggerbrush Rd in Austin (TX)	Stray	Nursing	Cat	Intact Male	Domestic Shorthair Mix	White/Orange Tabby	7	60
...	...	...	...	...	...	...	...	...	...	...	...	...	...
95480	A811886	1.0	Neutered Male	Kane	Pflugerville (TX)	Owner Surrender	Normal	Dog	Neutered Male	American Staffordshire Terrier Mix	Blue/White	1095	1095
95481	A817331	1.0	Spayed Female	*Squanchy	1601 E Slaughter Lane in Austin (TX)	Public Assist	Normal	Cat	Intact Female	Domestic Shorthair	Calico/White	7	60
95482	A818067	1.0	Intact Male	Bluto	Austin (TX)	Owner Surrender	Normal	Dog	Intact Male	Pit Bull Mix	Blue/White	4745	4745
95483	A815418	0.0	Intact Female	Laverne	2211 Santa Rita Street in Austin (TX)	Stray	Normal	Dog	Intact Female	Chihuahua Shorthair	Tan/White	365	365
95484	A815419	0.0	Intact Female	Shirley	2211 Santa Rita Street in Austin (TX)	Stray	Normal	Dog	Intact Female	Chihuahua Longhair	Tan	365	365

95485 rows × 13 columns

EDA

Now, perform the basic steps of exploratory data analysis (EDA) and look for insights to inform later ML modeling choices.

In [8]:
```
# Print the data types and nonnull values for each column
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 95485 entries, 0 to 95484
Data columns (total 13 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Pet ID                 95485 non-null  object
 1   Outcome Type           95485 non-null  float64
 2   Sex upon Outcome       95484 non-null  object
 3   Name                   59138 non-null  object
 4   Found Location         95485 non-null  object
 5   Intake Type            95485 non-null  object
 6   Intake Condition       95485 non-null  object
 7   Pet Type               95485 non-null  object
 8   Sex upon Intake        95484 non-null  object
 9   Breed                  95485 non-null  object
 10  Color                  95485 non-null  object
 11  Age upon Intake Days   95485 non-null  int64
 12  Age upon Outcome Days  95485 non-null  int64
dtypes: float64(1), int64(2), object(10)
memory usage: 9.5+ MB
```

In [9]:
```
# Print the column names
df.columns
```

Out[9]:
```
Index(['Pet ID', 'Outcome Type', 'Sex upon Outcome', 'Name', 'Found Location',
       'Intake Type', 'Intake Condition', 'Pet Type', 'Sex upon Intake',
       'Breed', 'Color', 'Age upon Intake Days', 'Age upon Outcome Days'],
      dtype='object')
```

In [10]:
```
# Create lists that identify the numerical, categorical, and text features, and the target/label
numerical_features = ["Age upon Intake Days", "Age upon Outcome Days"]

categorical_features = [
    "Sex upon Outcome",
    "Intake Type",
    "Intake Condition",
    "Pet Type",
    "Sex upon Intake",
]

text_features = ["Found Location", "Breed", "Color"]
```

```
model_features = numerical_features + categorical_features + text_features
model_target = "Outcome Type"
```

Note: The **Pet ID** and **Name** features were omitted because they are irrelevant to the outcome.

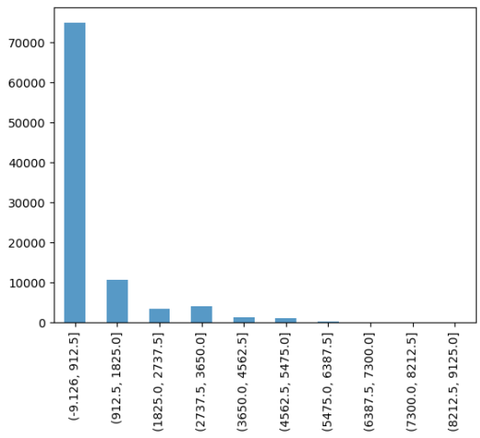
Cleaning the data

Cleaning numerical features

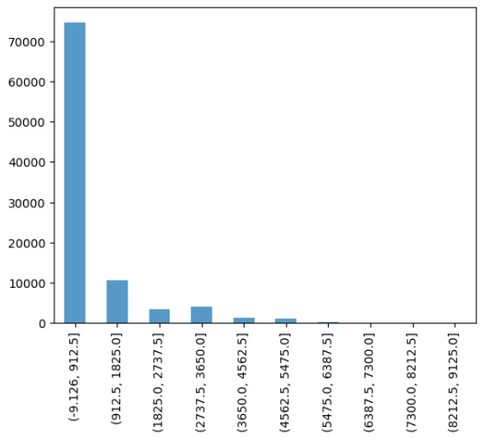
Take a moment to examine the numerical features. Remember that the `value_counts()` function can give a view of the numerical features by placing feature values in respective bins. The function can also be used for plotting.

In [11]:
```
for c in numerical_features:
    print(c)
    print(df[c].value_counts(bins=10, sort=False))
    df[c].value_counts(bins=10, sort=False).plot(kind="bar", alpha=0.75, rot=90)
    plt.show()
```

```
Age upon Intake Days
(-9.126, 912.5]     74835
(912.5, 1825.0]     10647
(1825.0, 2737.5]     3471
(2737.5, 3650.0]     3998
(3650.0, 4562.5]     1234
(4562.5, 5475.0]     1031
(5475.0, 6387.5]      183
(6387.5, 7300.0]       79
(7300.0, 8212.5]        5
(8212.5, 9125.0]        2
Name: Age upon Intake Days, dtype: int64
```




```
Age upon Outcome Days
(-9.126, 912.5]     74642
(912.5, 1825.0]     10699
(1825.0, 2737.5]     3465
(2737.5, 3650.0]     4080
(3650.0, 4562.5]     1263
(4562.5, 5475.0]     1061
(5475.0, 6387.5]      187
(6387.5, 7300.0]       81
(7300.0, 8212.5]        5
(8212.5, 9125.0]        2
Name: Age upon Outcome Days, dtype: int64
```



If any outliers are identified as likely wrong values, dropping them could improve the histograms for the numerical values and could later improve overall model performance.

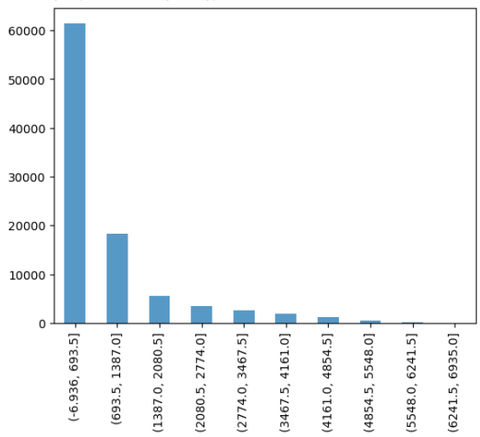
Remove any values in the upper 10 percent for the feature, and then plot the features.

In [12]:
```
for c in numerical_features:
    # Drop values beyond 90% of max()
    dropIndexes = df[df[c] > df[c].max() * 9 / 10].index
    df.drop(dropIndexes, inplace=True)
```

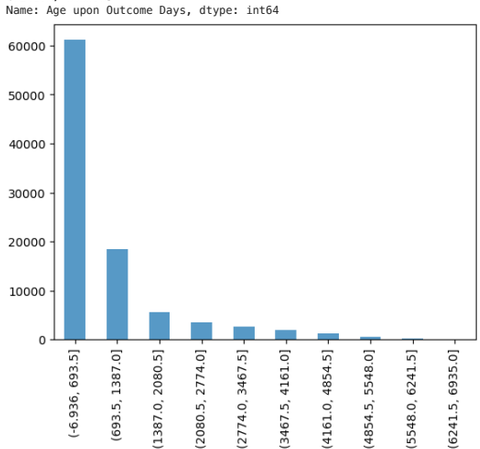
In [13]:
```
for c in numerical_features:
    print(c)
    print(df[c].value_counts(bins=10, sort=False))
```

```
    df[c].value_counts(bins=10, sort=False).plot(kind="bar", alpha=0.75, rot=90)
    plt.show()
```

```
Age upon Intake Days
(-6.936, 693.5]     61425
(693.5, 1387.0]     18400
(1387.0, 2080.5]     5657
(2080.5, 2774.0]     3471
(2774.0, 3467.5]     2557
(3467.5, 4161.0]     1962
(4161.0, 4854.5]     1148
(4854.5, 5548.0]      596
(5548.0, 6241.5]      183
(6241.5, 6935.0]       63
Name: Age upon Intake Days, dtype: int64
```




```
Age upon Outcome Days
(-6.936, 693.5]     61208
(693.5, 1387.0]     18490
(1387.0, 2080.5]     5643
(2080.5, 2774.0]     3465
(2774.0, 3467.5]     2600
(3467.5, 4161.0]     2004
(4161.0, 4854.5]     1196
(4854.5, 5548.0]      604
(5548.0, 6241.5]      187
(6241.5, 6935.0]       65
Name: Age upon Outcome Days, dtype: int64
```



Cleaning text features

Take a moment to examine the text features.

In [14]:

```
# Prepare cleaning functions
import re, string
import nltk
from nltk.stem import SnowballStemmer

stop_words = ["a", "an", "the", "this", "that", "is", "it", "to", "and", "in"]

stemmer = SnowballStemmer("english")


def preProcessText(text):
    # Lowercase text, and strip leading and trailing white space
    text = text.lower().strip()

    # Remove HTML tags
    text = re.compile("<.*?>").sub("", text)

    # Remove punctuation
    text = re.compile("[%s]" % re.escape(string.punctuation)).sub(" ", text)

    # Remove extra white space
    text = re.sub("\s+", " ", text)

    return text


def lexiconProcess(text, stop_words, stemmer):
    filtered_sentence = []
    words = text.split(" ")
    for w in words:
        if w not in stop_words:
            filtered_sentence.append(stemmer.stem(w))
    text = " ".join(filtered_sentence)

    return text


def cleanSentence(text, stop_words, stemmer):
    return lexiconProcess(preProcessText(text), stop_words, stemmer)
```

Note: The text cleaning process can take a while to complete, depending on the size of the text data.

In [15]:
```
# Clean the text features
for c in text_features:
    print("Text cleaning: ", c)
    df[c] = [cleanSentence(item, stop_words, stemmer) for item in df[c].values]
```

```
Text cleaning:  Found Location
Text cleaning:  Breed
Text cleaning:  Color
```

Train, validation, and test datasets

Now that the data has been cleaned, you need to split the full dataset into training and test subsets by using sklearn's train_test_split() function. With this function, you can specify the following:

- The proportion of the dataset to include in the test split as a number between 0.0-1.0 with a default of 0.25.
- An integer that controls the shuffling that is applied to the data before the split. Passing an integer allows for reproducible output across multiple function calls.

To help reduce sampling bias, the original dataset is shuffled before the split. After the initial split, the training data is further split into training and validation subsets.

In [16]:
```
from sklearn.model_selection import train_test_split

train_data, test_data = train_test_split(
    df, test_size=0.15, shuffle=True, random_state=23
)

train_data, val_data = train_test_split(
    train_data, test_size=0.15, shuffle=True, random_state=23
)

# Print the shapes of the training, validation, and test datasets
print(
    "Train - Validation - Test dataset shapes: ",
    train_data.shape,
    val_data.shape,
    test_data.shape,
)
```

```
Train - Validation - Test dataset shapes:  (68970, 13) (12172, 13) (14320, 13)
```

Data processing with a pipeline and ColumnTransformer

In a typical ML workflow, you need to apply data transformations, such as imputation and scaling, at least twice: first on the training dataset by using `.fit()` and `.transform()` when preparing the data to train the model, and then by using `.transform()` on any new data that you want to predict on (validation or test). Sklearn's Pipeline is a tool that simplifies this process by enforcing the implementation and order of data processing steps.

In this section, you will build separate pipelines to handle the numerical, categorical, and text features. Then, you will combine them into a composite pipeline along with an estimator. To do this, you will use a LogisticRegression classifier.

You will need multiple pipelines to ensure that all the data is handled correctly:

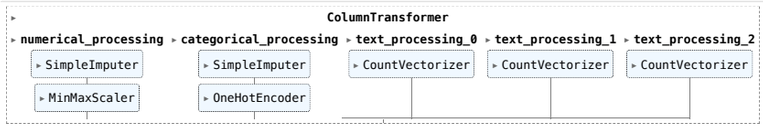
- **Numerical features pipeline:** Impute missing values with the mean by using sklearn's `SimpleImputer`, followed by `MinMaxScaler`. If different processing is desired for different numerical features, different pipelines should be built as described for the text features pipeline.
- **Categoricals pipeline:** Impute with a placeholder value (this won't have an effect because you already encoded the `nan` values), and encode with sklearn's `OneHotEncoder`. If computing memory is an issue, it is a good idea to check the number of unique values for the categoricals to get an estimate of how many dummy features one-hot encoding will create. Note the `handle_unknown` parameter, which tells the encoder to ignore (rather than throw an error for) any unique value that might show in the validation or test set that was not present in the initial training set.
- **Text features pipeline:** With memory usage in mind, build three more pipelines, one for each of the text features. The current sklearn implementation requires a separate transformer for each text feature (unlike the numericals and categoricals).

Finally, the selective preparations of the dataset features are then put together into a collective ColumnTransformer, which is used in a pipeline along with an estimator. This ensures that the transforms are performed automatically in all situations. This includes on the raw data when fitting the model, when making predictions, when evaluating the model on a validation dataset through cross-validation, or when making predictions on a test dataset in the future.

In [17]:
```
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, MinMaxScaler
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

### COLUMN_TRANSFORMER ###
##########################

# Preprocess the numerical features
numerical_processor = Pipeline(
    [
        ("num_imputer", SimpleImputer(strategy="mean")),
        (
            "num_scaler",
            MinMaxScaler(),
        ),
    ]
)

# Preprocess the categorical features
categorical_processor = Pipeline(
    [
        (
            "cat_imputer",
            SimpleImputer(strategy="constant", fill_value="missing"),
        ),  # Shown in case it is needed. No effect here because you already imputed with 'nan' strings.
        (
            "cat_encoder",
            OneHotEncoder(handle_unknown="ignore"),
        ),  # handle_unknown tells it to ignore (rather than throw an error for) any value that was not present in the initial training set.
    ]
)

# Preprocess first text feature
text_processor_0 = Pipeline(
    [("text_vectorizer_0", CountVectorizer(binary=True, max_features=50))]
)

# Preprocess second text feature
text_processor_1 = Pipeline(
    [("text_vectorizer_1", CountVectorizer(binary=True, max_features=50))]
)

# Preprocess third text feature
text_processor_2 = Pipeline(
    [("text_vectorizer_2", CountVectorizer(binary=True, max_features=50))]
)

# Combine all data preprocessors (add more if you choose to define more)
# For each processor/step, specify: a name, the actual process, and the features to be processed.
data_processor = ColumnTransformer(
    [
        ("numerical_processing", numerical_processor, numerical_features),
        ("categorical_processing", categorical_processor, categorical_features),
        ("text_processing_0", text_processor_0, text_features[0]),
        ("text_processing_1", text_processor_1, text_features[1]),
        ("text_processing_2", text_processor_2, text_features[2]),
    ]
)

# Visualize the data processing pipeline
```

```
from sklearn import set_config

set_config(display="diagram")
data_processor
```

Out[17]:



In [18]:

```
# Prepare data for training
X_train = train_data[model_features]
y_train = train_data[model_target].values

# Get validation data to validate the network
X_val = val_data[model_features]
y_val = val_data[model_target].values

# Get test data to test the network for submission to the leaderboard
X_test = test_data[model_features]
y_test = test_data[model_target].values

print("Dataset shapes before processing: ", X_train.shape, X_val.shape, X_test.shape)

X_train = data_processor.fit_transform(X_train).toarray()
X_val = data_processor.transform(X_val).toarray()
X_test = data_processor.transform(X_test).toarray()

print("Dataset shapes after processing: ", X_train.shape, X_val.shape, X_test.shape)
```

```
Dataset shapes before processing:  (68970, 10) (12172, 10) (14320, 10)
Dataset shapes after processing:  (68970, 171) (12172, 171) (14320, 171)
```

Training and validation of a neural network

Now, run the following code cell to interact with the neural network to gain insight into how neural networks train.

Architect the neural network by updating the number of layers (maximum of 4) and the number of neurons per layer (maximum of 3) to solve the classification problem that displays when you run the following cell. The background colors show the neural network's predicted classification regions for the true data (circles).

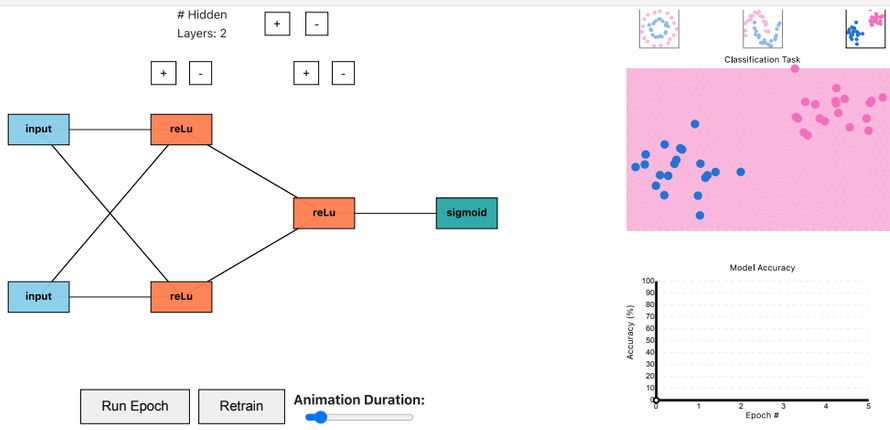
Note that upon retraining the network, the weights are randomly initialized, and the gradients are reset to 0. In the visual representation, each green circle corresponds to an epoch. Each red circle corresponds to that layer's weight update gradient (from backpropagation).

To develop a better understanding, train the model for different architectures. Note that the model gets stuck sometimes—initialization is important!

In [19]:

```
NeuralNetwork()
```

Out[19]:



Now you need to build a PyTorch neural network and use it to fit to the training data. As part of the training, you need to use the validation data to check performance at the end of each training iteration.

Try it yourself!

To define the hyperparameters of the algorithm, run the following cell. Note how the data is loaded into PyTorch tensors. Observe how the DataLoader is defined. The DataLoader is used to load the data in batches during training.

In [20]:

```
# Define the hyperparamaters
batch_size = 16
num_epochs = 15
learning_rate = 0.001
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Convert the data into PyTorch tensors

X_train = torch.tensor(X_train, dtype=torch.float32).to(device)
X_val = torch.tensor(X_val, dtype=torch.float32).to(device)
X_test = torch.tensor(X_test, dtype=torch.float32).to(device)

y_train = torch.tensor(y_train, dtype=torch.long).to(device)
y_val = torch.tensor(y_val, dtype=torch.long).to(device)
y_test = torch.tensor(y_test, dtype=torch.long).to(device)

# Use PyTorch DataLoaders to load the data in batches
train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
train_loader = torch.utils.data.DataLoader(train_dataset,
                                           batch_size=batch_size,
                                           drop_last=True)

val_dataset = torch.utils.data.TensorDataset(X_val, y_val)
val_loader = torch.utils.data.DataLoader(val_dataset,
                                         batch_size=batch_size,
                                         drop_last=True)
device
```

Out[20]:

```
device(type='cuda', index=0)
```

Try it yourself!

Create a multilayer perceptron by using the `Sequential` module with the following attributes:

1. Use two hidden layers, both of size 64.
2. Attach a dropout layer to each hidden layer.
3. Use a ReLU activation for each hidden layer.
4. Create one output layer.

In [25]:
```
# Create a multilayer perceptron by using the Sequential module. Add the following in sequence:
# Two hidden layers of size 64
# Dropout layers attached to the hidden layers
# ReLU activation functions
# One output layer

############### CODE HERE ###############

input_size = X_train.shape[1]
#num_classes = 2
output_size = 2

# Calculate the size of the flattened layer
# The formula depends on the input size and the parameters of the Conv2D layer
# Assuming a stride of 1 and no padding, the output size of each dimension after the Conv2D layer is (input_size - kernel_size + 1)
#conv_output_size = (input_size - 3 + 1) ** 2 * 32  # 32 is the number of output channels

"""
net = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(conv_output_size, 128),
    nn.ReLU(),
    nn.Linear(128, num_classes),
    nn.Softmax()
)
"""
net = nn.Sequential(
    # First hidden layer
    nn.Linear(in_features=input_size, out_features=64),  # Replace `input_size` with your input size
    nn.ReLU(),
    #nn.Dropout(0.5),
    nn.Dropout(p = 0.4),    # 40 percent
    # Second hidden layer
```

```
    nn.Linear(64, 64),
    nn.ReLU(),
    #nn.Dropout(0.5),
    nn.Dropout(p = 0.3),    # 30 percent
    # Output layer
    nn.Linear(64, output_size)  # Replace `output_size` with the number of your output classes
).to(device)


############## END OF CODE ##############


def xavier_init_weights(m):
    if type(m) == nn.Linear:
        torch.nn.init.xavier_uniform_(m.weight)


net.apply(xavier_init_weights)
```

Out[25]:
```
Sequential(
  (0): Linear(in_features=171, out_features=64, bias=True)
  (1): ReLU()
  (2): Dropout(p=0.4, inplace=False)
  (3): Linear(in_features=64, out_features=64, bias=True)
  (4): ReLU()
  (5): Dropout(p=0.3, inplace=False)
  (6): Linear(in_features=64, out_features=2, bias=True)
)
```

In [26]:
```
# Define the loss function and the optimizer
# Choose cross-entropy loss for this classification problem
loss = nn.CrossEntropyLoss()

# Optimize with stochastic gradient descent. You can experiment with other optimizers.
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
```

Try it yourself!

To train the network, run the following cell. Watch how the training and validation loss change for each epoch.

In [27]:
```
import time

######################
# Network training and validation

# Start the outer epoch loop (epoch = full pass through the dataset)
for epoch in range(num_epochs):
    start = time.time()

    training_loss, validation_loss = 0.0, 0.0

    # Training loop (with autograd and trainer steps)
    # This loop trains the neural network
    # Weights are updated here
    net.train()  # Activate training mode (dropouts and so on)
    for data, target in train_loader:
        # Zero the parameter gradients
        optimizer.zero_grad()
        data = data.to(device)
        target = target.to(device)
        # Forward + backward + optimize
        output = net(data)
        L = loss(output, target)
        L.backward()
        optimizer.step()
        # Add batch loss
        training_loss += L.item()

    net.eval()  # Activate eval mode (don't use dropouts and so on)
    for data, target in val_loader:
        data = data.to(device)
        target = target.to(device)
        output = net(data)
        L = loss(output, target)
        # Add batch loss
        validation_loss += L.item()

    # Take the average losses
    training_loss = training_loss / len(train_loader)
    val_loss = validation_loss / len(val_loader)

    end = time.time()
    print(
        "Epoch %s. Train_loss %f Validation_loss %f Seconds %f"
```

```
        % (epoch, training_loss, val_loss, end - start)
    )
```

```
Epoch 0. Train_loss 0.642918 Validation_loss 0.565680 Seconds 16.062835
Epoch 1. Train_loss 0.552202 Validation_loss 0.474820 Seconds 4.509893
Epoch 2. Train_loss 0.497483 Validation_loss 0.443819 Seconds 4.649142
Epoch 3. Train_loss 0.473968 Validation_loss 0.434936 Seconds 4.620003
Epoch 4. Train_loss 0.462879 Validation_loss 0.430313 Seconds 4.589993
Epoch 5. Train_loss 0.454005 Validation_loss 0.427804 Seconds 4.592375
Epoch 6. Train_loss 0.449191 Validation_loss 0.425981 Seconds 4.610003
Epoch 7. Train_loss 0.446426 Validation_loss 0.424917 Seconds 4.586667
Epoch 8. Train_loss 0.442533 Validation_loss 0.423767 Seconds 4.633833
Epoch 9. Train_loss 0.440957 Validation_loss 0.423062 Seconds 4.863455
Epoch 10. Train_loss 0.438682 Validation_loss 0.422318 Seconds 4.771560
Epoch 11. Train_loss 0.434879 Validation_loss 0.421614 Seconds 4.571795
Epoch 12. Train_loss 0.434225 Validation_loss 0.420950 Seconds 4.610446
Epoch 13. Train_loss 0.432753 Validation_loss 0.420441 Seconds 4.551708
Epoch 14. Train_loss 0.431406 Validation_loss 0.419900 Seconds 4.554403
```

Testing the neural network

Now you can evaluate the performance of the trained network on the test set.

In [28]:
```
from sklearn.metrics import classification_report

# Activate eval mode (don't use dropouts and so on)
net.eval()

# Get test predictions
predictions = net(X_test)

# Print performance of the test data
print(
    classification_report(
        y_test.cpu().numpy(), predictions.argmax(axis=1).cpu().detach().numpy()
    )
)
```

```
              precision    recall  f1-score   support

           0       0.87      0.71      0.78      6267
           1       0.80      0.91      0.86      8053

    accuracy                           0.83     14320
   macro avg       0.83      0.81      0.82     14320
weighted avg       0.83      0.83      0.82     14320
```

Improvement ideas

You can improve this neural network by tuning network parameters such as the following:

- Architecture
- Number of layers
- Number of hidden neurons
- Choice of activation function
- Weight initialization
- Dropout
- Choice of optimizer function
- Learning rate
- Batch size
- Number of epochs

As you make changes, closely monitor the loss function and the accuracy on both training and validation to identify what changes improve your model.

Conclusion

In this notebook, you built a basic neural network to process text data.

Next Lab: Introducing CNNs