Linear Regression

Before studying deep neural networks, we need to cover the fundamental components of a simple (linear) neural network. We'll begin with the topic of linear regression. Since linear regression can be modeled as a linear neural network, it provides an excellent running example to introduce the essential components of neural networks. Regression is a form of supervised learning which aims to model the relationship between one or more input variables (features) and a continuous (target) variable. We assume that the relationship between the input variables xx and the target variable yy can be expressed as a weighted sum of the inputs (i.e., the model is linear in the parameters). In short, linear regression aims to learn a function that maps one or more input features to a single numerical target value.

We will begin by developing a mathematical model for linear regression and will demonstrate that it can be solved analytically using ordinary least squares (OLS), which has a closed-form solution called the normal equations. We will then show how this problem can be cast as a linear neural network in PyTorch using just a single neuron. Finally, we will demonstrate that linear regression can be used to fit a certain class of non-linear mathematical functions.

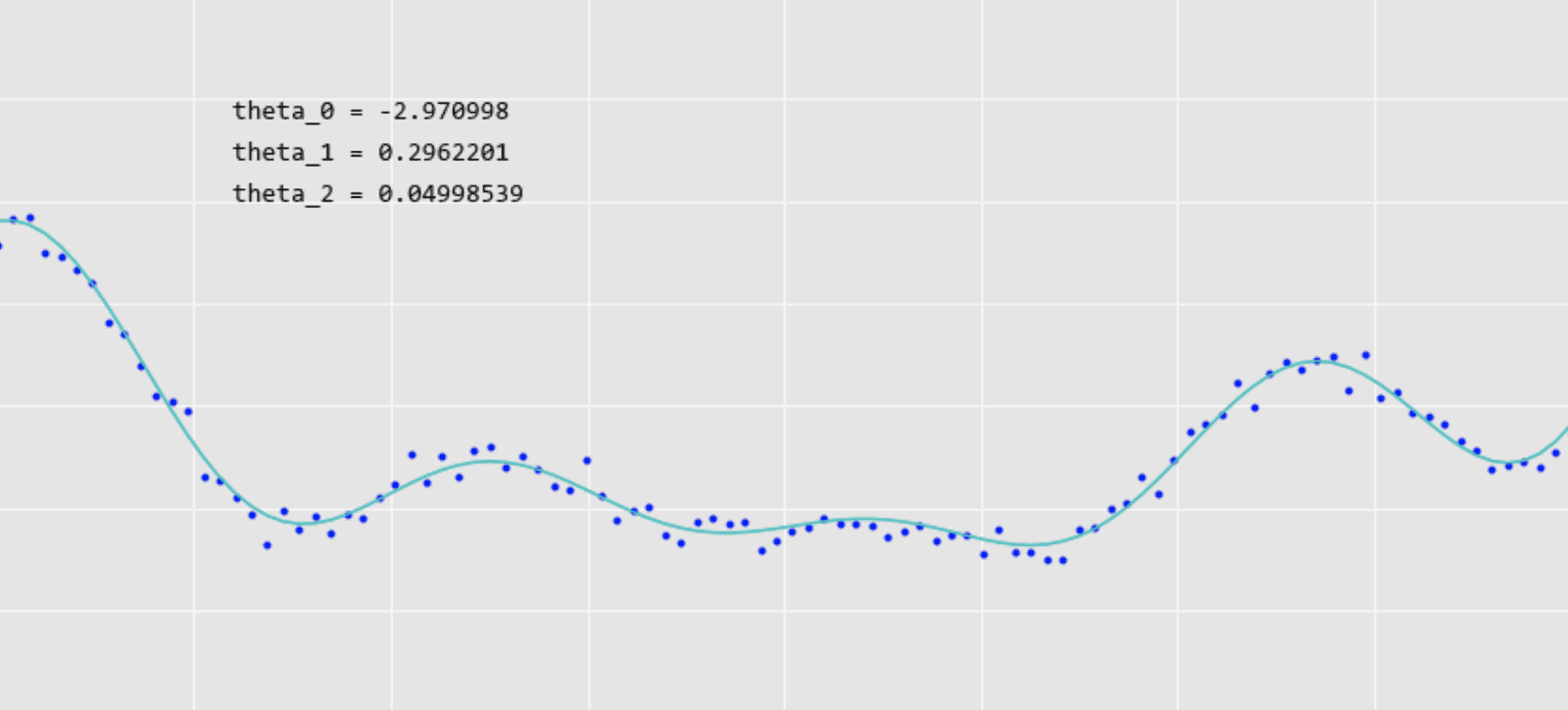


Table of Contents

* [1 Linear Regression Model](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/01_Introduction_to_Linear_Regression_PyTorch_v3.html#1-Linear-Regression-Model)
* [2 Ordinary Least Squares and the Normal Equations (Closed Form Solution)](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/01_Introduction_to_Linear_Regression_PyTorch_v3.html#2-Ordinary-Least-Squares-and-the-Normal-Equations-(Closed-Form-Solution))
* [3 Fitting a Straight Line using the Normal Equations](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/01_Introduction_to_Linear_Regression_PyTorch_v3.html#3-Fitting-a-Straight-Line-using-the-Normal-Equations)
* [4 Fitting a Straight Line using PyTorch](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/01_Introduction_to_Linear_Regression_PyTorch_v3.html#4-Fitting-a-Straight-Line-using-PyTorch)
* [5 Fitting Non-Linear Data using PyTorch](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/01_Introduction_to_Linear_Regression_PyTorch_v3.html#5-Fitting-Non-Linear-Data-using-PyTorch)
* [6 Conclusion](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/01_Introduction_to_Linear_Regression_PyTorch_v3.html#6-Conclusion)

1 Linear Regression Model

**Note**: The following model description is from Andrew Ng's machine learning class. This notation is commonly used in machine learning, and we are adopting it here to introduce the topic. As we transition to neural networks, some of the notation will change (for example, the parameters θθ will be referred to as weights ww); however, the discussion below and the use of subscripts and superscripts is helpful for communicating the concepts and dimensionality of the data.

Linear regression represents an important class of supervised learning problems in which one or more features are used to predict a scalar. More formally, the goal is to learn a function that maps input features to an output scalar value. The function is sometimes referred to as a **hypothesis** and for linear regression takes the general form below.

hθ(x)=θ0+θ1x1+θ2x2+...hθ(x)=θ0+θ1x1+θ2x2+...

We often simplify the notation as follows:

y′=θ0+θ1x1+θ2x2+...y′=θ0+θ1x1+θ2x2+...

Here, the θiθi's are the parameters (also called weights) parameterizing the space of linear functions mapping from xx to yy. The problem is called **regression** when the target value is a continuous variable. When the target value represents a small number of discrete values (e.g., classes), we call it a **classification** problem. We postulate a hypothesis space in both cases and use input training data (features) to learn the parameters that fit the data. In this notebook, we will restrict our attention to regression problems.

In the case of a single input feature, we have the familiar form for the equation of a line where y′y′ represents the estimated value of yy given an input value of xx and the parameters mm abd bb. Here, bb represents θ0θ0 and mm represents θ1θ1.

y′=mx+by′=mx+b

To simplify the general notation, we can introduce the convention of letting x0=1x0=1 (this is the intercept term), so that:

h(x)=n∑i=0θixi=θTxh(x)=∑i=0nθixi=θTx

where, on the right-hand side above, we are viewing θθ and xx both as vectors, and here nn is the number of input variables (not counting x0x0). This has no consequence on the results, but makes the mathematical notation more compact for the discussion that follows.

Now, given a training set, how do we pick, or learn, the parameters θθs A reasonable approach is to make h(x)h(x) close to yy, at least for the training examples we have. To formalize this, we will define a function that measures, for each value of the θθs, how close the h(x(i))h(x(i))’s are to the corresponding y(i)y(i)’s. We define the loss function:

J(θ)=12m∑i=1(hθ(x(i))−y(i))2J(θ)=12∑i=1m(hθ(x(i))−y(i))2

We have previously shown that the above loss function can be minimized using **gradient descent**. The next section below shows that linear regression can also be solved using a closed-form analytic solution referred to as the **normal equations**.

2 Ordinary Least Squares and the Normal Equations (Closed Form Solution)

In this section, we will develop an analytical solution to linear regression and will take the opportunity to also introduce linear algebra matrix notation to keep the equations more compact.

Given a training set, we define a **design matrix** XX that can be represented as an mm x nn matrix that contains mm training examples in its rows and, nn represents the number of features in each training example. Often times the bias (or intercept) term is added to each row in the matrix in which case the matrix, becomes as an mm x (n+1)(n+1).  
  
X=⎡⎢

⎢

⎢

⎢

⎢⎣−(x(1))T−−(x(2))T−⋮−(x(m))T−⎤⎥

⎥

⎥

⎥

⎥⎦X=[−(x(1))T−−(x(2))T−⋮−(x(m))T−]

Let →yy→ be the mm-dimensional vector that contains all the target values from the training set.

→y=⎡⎢

⎢

⎢

⎢

⎢⎣y(1)y(2)⋮y(m)⎤⎥

⎥

⎥

⎥

⎥⎦(1)(1)y→=[y(1)y(2)⋮y(m)]

Now, since h(x(i))=(x(i))Tθh(x(i))=(x(i))Tθ, we can easily verify that:

Xθ−→y=⎡⎢

⎢

⎢

⎢

⎢⎣−(x(1))Tθ−−(x(2))Tθ−⋮−(x(m))Tθ−⎤⎥

⎥

⎥

⎥

⎥⎦−⎡⎢

⎢

⎢

⎢

⎢⎣y(1)y(2)⋮y(m)⎤⎥

⎥

⎥

⎥

⎥⎦(2)(2)Xθ−y→=[−(x(1))Tθ−−(x(2))Tθ−⋮−(x(m))Tθ−]−[y(1)y(2)⋮y(m)]

=⎡⎢

⎢⎣hθ(x(i))−y(i)⋮hθ(x(m))−y(m)⎤⎥

⎥⎦(3)(3)=[hθ(x(i))−y(i)⋮hθ(x(m))−y(m)]

Using the fact that for a vector zz, we have that zTz=∑iz2izTz=∑izi2:

12(Xθ−→y)T(Xθ−→y)=12m∑i=1(hθ(x(i))−y(i))2(4)(4)12(Xθ−y→)T(Xθ−y→)=12∑i=1m(hθ(x(i))−y(i))2

To minimize JJ, we compute the gradient of JJ with respect to θθ as follows:

∇θJ(θ)=∇θ12(Xθ−→y)T(Xθ−→y)∇θJ(θ)=∇θ12(Xθ−y→)T(Xθ−y→)

Carrying out the above gradient requires some matrix calculus and some matrix properties, which we will not delve into, but is easily found on the internet. The above gradient simplifies to the following expression:

=XTXθ−XT→y=XTXθ−XTy→

To minimize JJ, we set its derivative to zero, and obtain the **normal equations**:

XTXθ=XT→yXTXθ=XTy→

Thus, the value of θθ that minimizes J(θ)J(θ) is given in closed-form by the following equation:

θ=(XTX)−1XT→yθ=(XTX)−1XTy→

In [ ]:

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** torch

**import** torch.nn **as** nn

**import** torch.optim **as** optim

plt**.**style**.**use("ggplot")

plt**.**rcParams["figure.figsize"] **=** (10, 5)

plt**.**rcParams["axes.titlesize"] **=** 18

plt**.**rcParams["axes.labelsize"] **=** 16

block\_plot **=** **False**

3 Fitting a Straight Line using the Normal Equations

We will begin by showing that the normal equations can be used to find the parameters of a straight line (slope and intercept) given a set of data points in two dimensions.

3.1 Create Convenience Functions

Let's first create a convenience function to generate some linear data with a small amount of added noise.

In [ ]:

**import** random

*# Random manual seed for consistency.*

seed **=** 42

random**.**seed(seed)

np**.**random**.**seed(seed)

torch**.**random**.**manual\_seed(seed)

torch**.**manual\_seed(seed)

torch**.**backends**.**deterministic **=** **True**

torch**.**backends**.**benchmark **=** **True**

In [ ]:

**def** create\_linear\_data(num\_data**=**100, y\_offset**=**0, slope**=**1, stddev**=**0.3):

*# Create some linear data with a small amount of noise.*

X **=** 10 **\*** torch**.**rand(size**=**[num\_data])

y **=** y\_offset **+** slope **\*** X **+** torch**.**normal(std**=**stddev, mean**=**0, size**=**[num\_data])

X **=** X**.**view((len(X), 1))

y **=** y**.**view((len(y), 1))

**return** X, y

In [ ]:

**def** plot\_data(x, y, xlim**=**(0, 10), ylim**=**(0, 10)):

plt**.**figure

plt**.**plot(x, y, "b.")

plt**.**xlabel("x")

plt**.**ylabel("y"),

plt**.**xlim(xlim)

plt**.**ylim(ylim)

plt**.**show(block**=**block\_plot)

3.2 Generate Linear Data

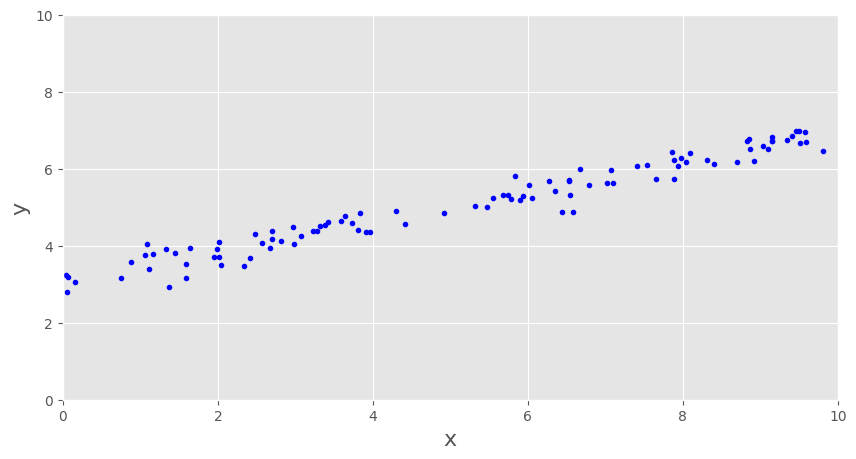
In [ ]:

y\_int **=** 3

slope **=** .4

X, y **=** create\_linear\_data(y\_offset **=** y\_int, slope **=** slope, stddev **=** 0.3)

plot\_data(X, y)



3.3 Implement Normal Equations

In [ ]:

**def** compute\_theta(X, y):

m **=** X**.**shape[0] *# Number of samples.*

*# Concatenate a 1 to the beginning of each feature vector.*

X **=** torch**.**cat((torch**.**ones((m, 1)), X), axis**=**1)

y **=** y**.**view((m, 1))

*# Solve for theta using the Normal Equations.*

X\_T **=** X**.**T

XT\_X **=** torch**.**matmul(X\_T, X)

XT\_X\_inv **=** torch**.**inverse(XT\_X)

XT\_y **=** torch**.**matmul(X\_T, y)

theta **=** torch**.**matmul(XT\_X\_inv, XT\_y)

**return** theta

3.4 Solve for the Model Parameters (slope and y-intercept)

In [ ]:

print("Actual Coefficients:\n")

print("Slope: ", slope)

print("Y-Int: ", y\_int)

print("\n")

*# Compute the parameters (theta) based on the closed-form solution using the Normal Equations.*

theta **=** compute\_theta(X, y)

y\_int **=** theta[0]**.**numpy()

slope **=** theta[1]**.**numpy()

print("Predicted Coefficients:\n")

print("Slope: ", slope[0])

print("Y-int: ", y\_int[0])

Actual Coefficients:

Slope: 0.4

Y-Int: 3

Predicted Coefficients:

Slope: 0.38614762

Y-int: 3.0692215

3.5 Display the Results

In [ ]:

**def** predict\_y(X, theta):

X **=** torch**.**cat((torch**.**ones((X**.**shape[0], 1)), X), axis**=**1)

pred\_y **=** torch**.**matmul(X, theta)

**return** pred\_y

In [ ]:

pred\_y **=** predict\_y(X, theta)

plt**.**plot(X, y, "b.")

plt**.**plot(X, pred\_y, "c-")

plt**.**xlim((0, 10))

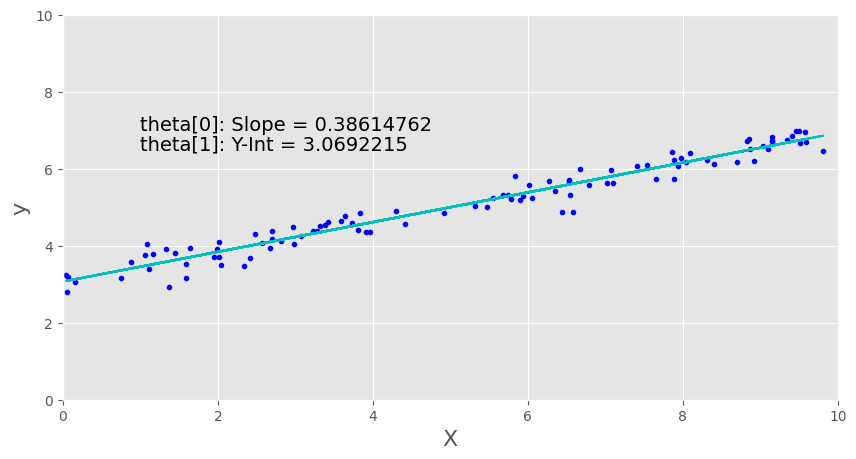
plt**.**ylim((0, 10))

plt**.**text(1, 7.0, "theta[0]: Slope = " **+** str(slope[0]), fontsize**=**14)

plt**.**text(1, 6.5, "theta[1]: Y-Int = " **+** str(y\_int[0]), fontsize**=**14)

plt**.**xlabel("X")

plt**.**ylabel("y")



Discussion

As you can see, using the normal equations to solve linear regression problems is very simple. However, there are several reasons why this approach can be problematic. Solving this equation requires inverting a matrix which can be computationally expensive, especially for very large problems which may include thousands of features. There is also an issue associated with its stability. Suppose that the matrix is not invertible due to numerical issues? There are methods that can mitigate some numerical problems, such as using what's known as the "pseudo" inverse, but this is not always ideal in practice, and these are some reasons why this approach is often not practical for large problems that we typically encounter in machine learning.

4 Fitting a Straight Line using PyTorch

Let's now take a look at solving this problem using a very simple linear neural network.

4.1 Mathematical Model

Create a simple linear model with one parameter (the slope) to model a straight line that passes through the origin.

The goal is to predict yy given some value of xx. To do this, we will fit a line that goes through the data points (xi,yi)(xi,yi). We will simplify the problem so that the line passes through the origin. The equation for such a line is:

y=mxy=mx

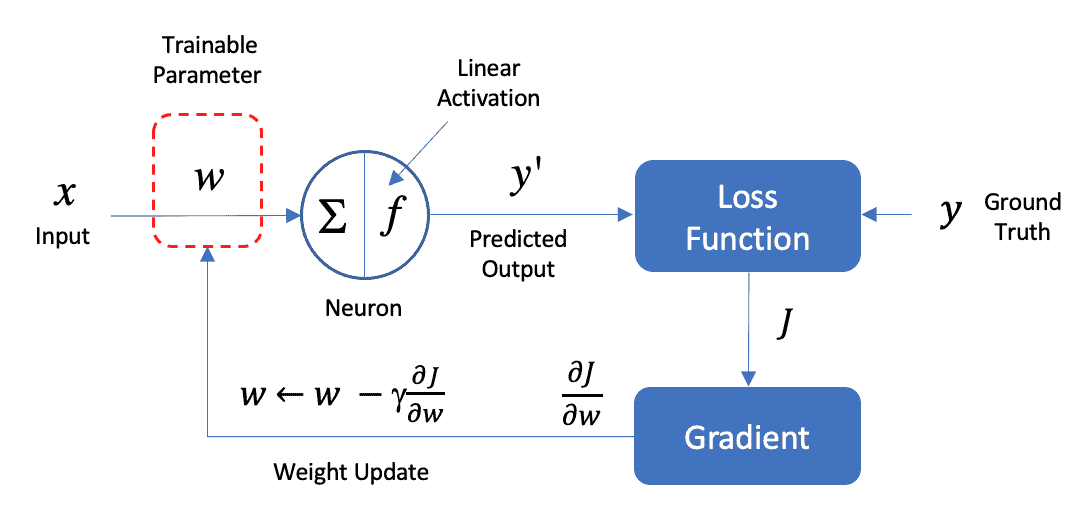
We have a set of data points (xi,yi)(xi,yi), and they should all satisfy the equation above. Therefore,

yi=mxiyi=mxi

The model has a single parameter mm (the slope of the line) that we wish to compute.

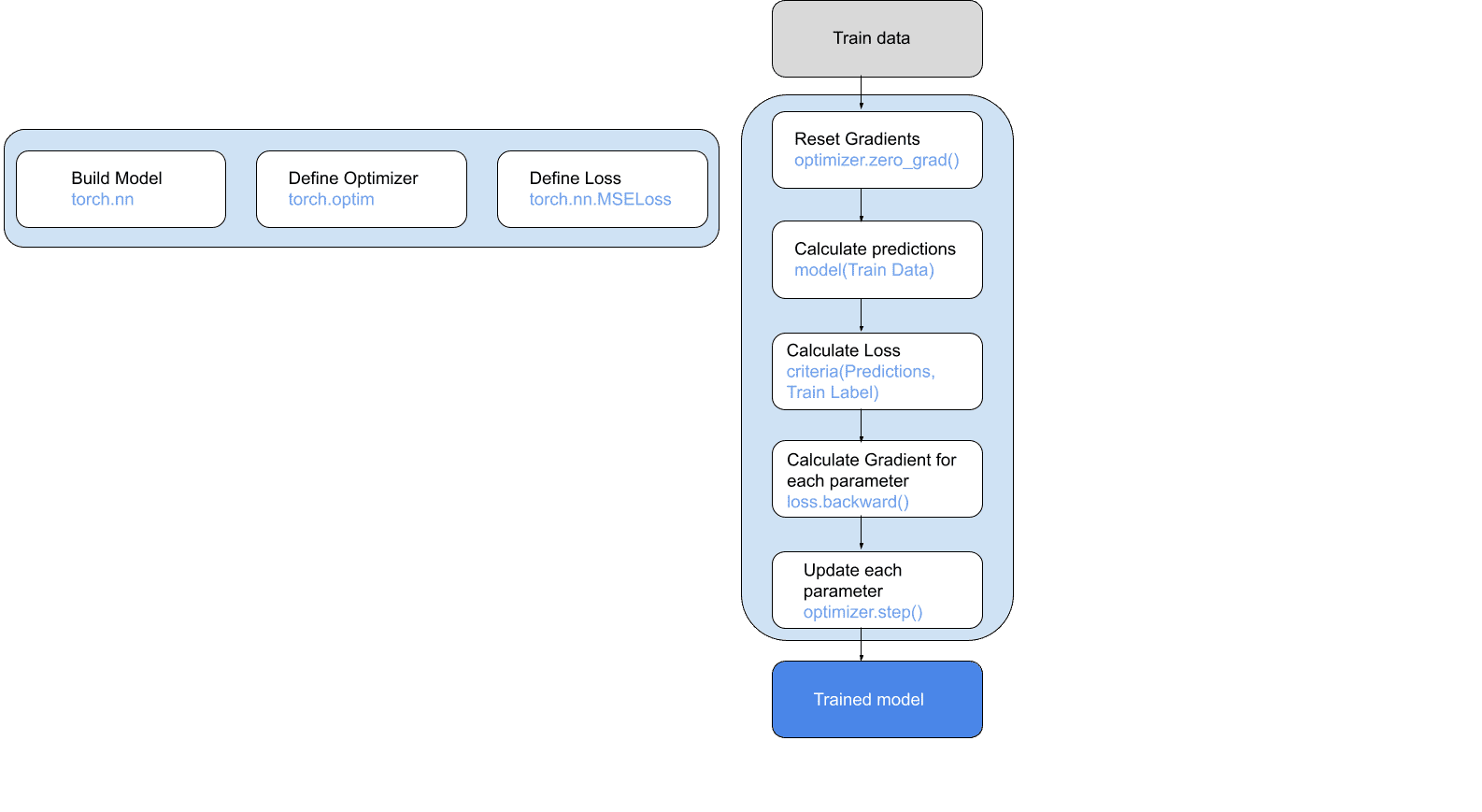
4.2 Modeling a Linear Neural Network in PyTorch

We restrict the model to a straight line that passes through the origin. Notice that this is very similar to the gradient descent notebook from the previous module, but rather than implementing all the details from scratch, we will use **PyTorch** to perform the same task. The network diagram below represents the simplest possible neural network. It has an input layer consisting of a single feature. Technically the input layer is not counted as a layer since there are no trainable parameters associated with it. The network has just a single layer consisting of a single neuron. The single-layer IS the output layer. The neuron has a linear activation function that simply multiplies the input feature xx by the weight ww. For every training sample, the predicted output y′y′ is compared to the actual value from the training data, and the loss is computed. This allows us to compute the gradient with respect to ww and update the weight (slope) according to an input learning rate. All of these details are taken care of by PyTorch once we define the network model and call a function to train the network.



The following steps summarize the workflow in PyTorch:

1. Build/Define a network model using PyTorch.
2. Define the optimizer optimizer = optim.Adam(model.parameters())
3. Define the loss to be used criterion = torch.nn.MSELoss()
4. Train the model with
   1. optimizer.zero\_grad()
   2. output = model(Train Data)
   3. loss = criterion(output, Train Label)
   4. loss.backward()
   5. optimizer.step()
5. Predict the output model(Test Data)



4.3 Create the PyTorch Model

The bias term for the neuron will be set to zero in this example using the option bias=False.

In [ ]:

model **=** nn**.**Linear(in\_features**=**1, out\_features**=**1, bias**=False**)

4.4 Compile the Model

Compiling the model requires you to specify the type of loss function (in this case, mean square error: 'mse') and to also specify the specific type of optimizer to use.

In [ ]:

optimizer **=** optim**.**Adam(model**.**parameters())

criterion **=** torch**.**nn**.**MSELoss()

4.5 Train the Model

Once the loss and optimizer are defined start training each epoch in a loop. At the begining of each loop use optimizer.zero\_grad to reset gradients from previous epoch. Calculate model's predictions, compute the loss between predicted labels and ground truth. Compute the gradients with for each parameter loss.backward update the parameters using the gradients optimizer.step.

In [ ]:

loss\_curve **=** []

**for** epoch **in** range(5000):

optimizer**.**zero\_grad()

output **=** model(X**.**float())

loss **=** criterion(output, y**.**float())

loss**.**backward()

optimizer**.**step()

loss\_curve**.**append(loss**.**detach()**.**numpy()**.**item())

In [ ]:

loss\_values **=** loss\_curve

epochs **=** range(1, len(loss\_values) **+** 1)

plt**.**figure(figsize**=**[15, 5])

plt**.**plot(epochs, loss\_values, "orange", label**=**"Training loss")

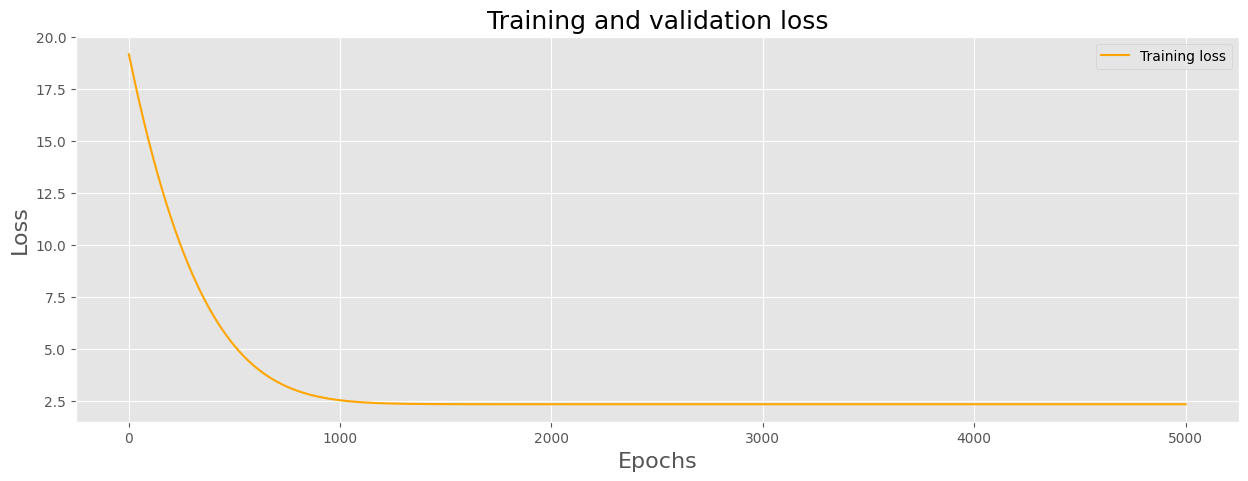
plt**.**title("Training and validation loss")

plt**.**xlabel("Epochs")

plt**.**ylabel("Loss")

plt**.**legend()

plt**.**show()



4.6 Predict Model Parameters and Display Results

After the model has been trained, we can access the trained parameters directly from the model object.

In [ ]:

slope **=** model**.**state\_dict()["weight"][0]

print('Slope: ', slope[0]**.**numpy())

pred\_y **=** model(X**.**float())**.**detach()**.**numpy()

Slope: 0.8359515

In [ ]:

plt**.**plot(X, y, 'b.')

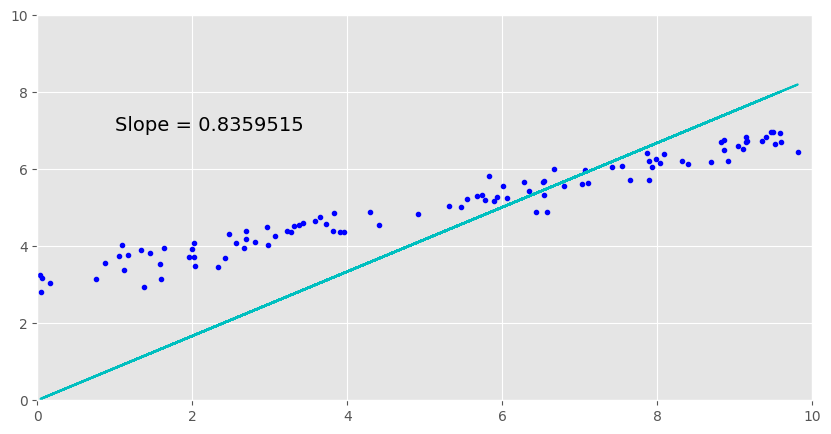
plt**.**plot(X, pred\_y, 'c-')

plt**.**text(1, 7.0, 'Slope = ' **+** str(slope[0]**.**numpy()), fontsize **=** 14)

plt**.**xlim(0, 10)

plt**.**ylim(0, 10)

plt**.**show()

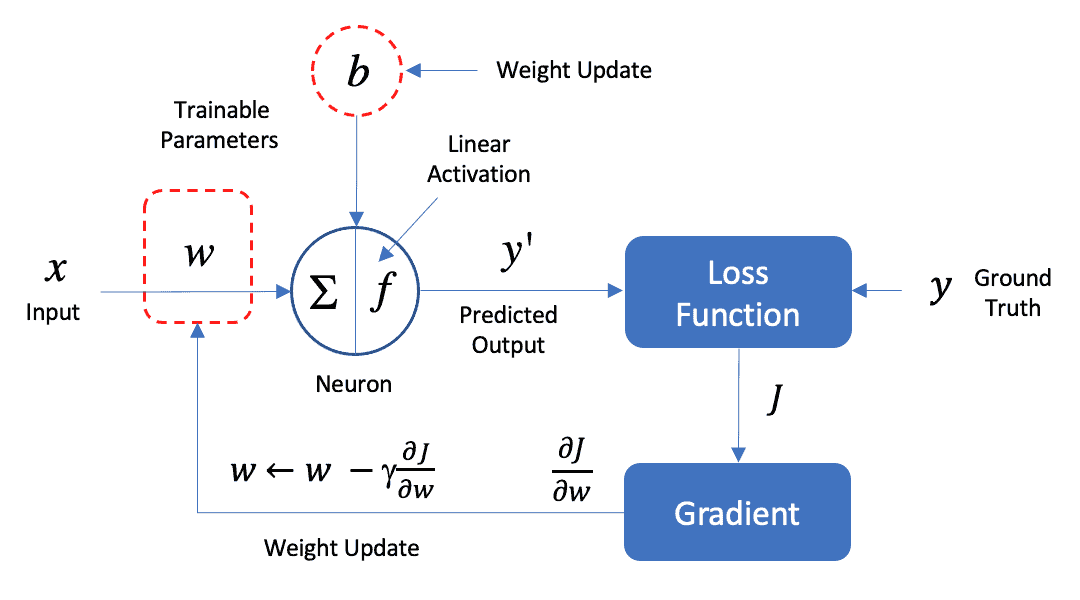


Discussion

The fitted line must pass through the origin since our model for the line only includes the slope, so it cannot fit the data very well, as shown in the plot above. Next, let's add the bias term to the network model and see how much better the fitted model performs. This is easily done in PyTorch by passing an input flag bias=True when we define the model.

4.7 Create the PyTorch Model with a Bias Term

In this section, we will model the bias term as shown in the network diagram below.



In [ ]:

model **=** nn**.**Linear(in\_features **=** 1, out\_features **=** 1, bias **=** **True**)

4.8 Compile the Model

In [ ]:

optimizer **=** optim**.**Adam(model**.**parameters())

criterion **=** torch**.**nn**.**MSELoss()

4.9 Train the Model

In [ ]:

loss\_curve **=** []

**for** epoch **in** range(10000):

optimizer**.**zero\_grad()

output **=** model(X**.**float())

loss **=** criterion(output, y**.**float())

loss**.**backward()

optimizer**.**step()

loss\_curve**.**append(loss**.**detach()**.**numpy()**.**item())

In [ ]:

loss\_values **=** loss\_curve

epochs **=** range(1, len(loss\_values) **+** 1)

plt**.**figure(figsize **=** [15, 5])

plt**.**plot(epochs, loss\_values, "orange", label **=** "Training loss")

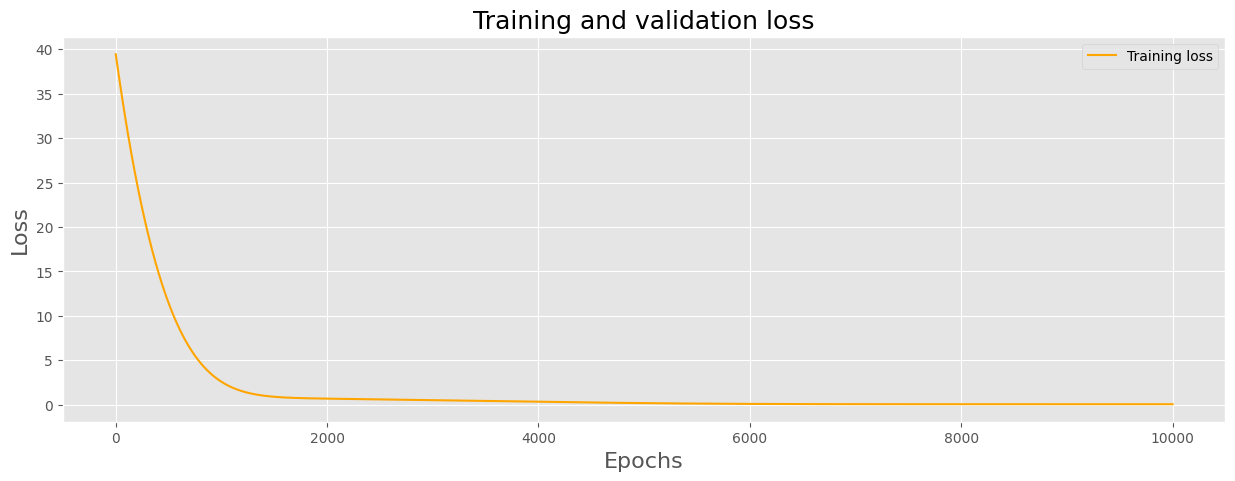
plt**.**title("Training and validation loss")

plt**.**xlabel("Epochs")

plt**.**ylabel("Loss")

plt**.**legend()

plt**.**show()



4.10 Predict Model Parameters and Display Results

In [ ]:

slope **=** model**.**state\_dict()['weight'][0]

y\_int **=** model**.**state\_dict()['bias'][0]

print('Slope: ', slope[0]**.**numpy())

print('Y-Int: ', y\_int**.**numpy())

pred\_y **=** model(X**.**float())**.**detach()**.**numpy()

Slope: 0.38615018

Y-Int: 3.0692043

In [ ]:

plt**.**plot(X, y, 'b.')

plt**.**plot(X, pred\_y, 'c-')

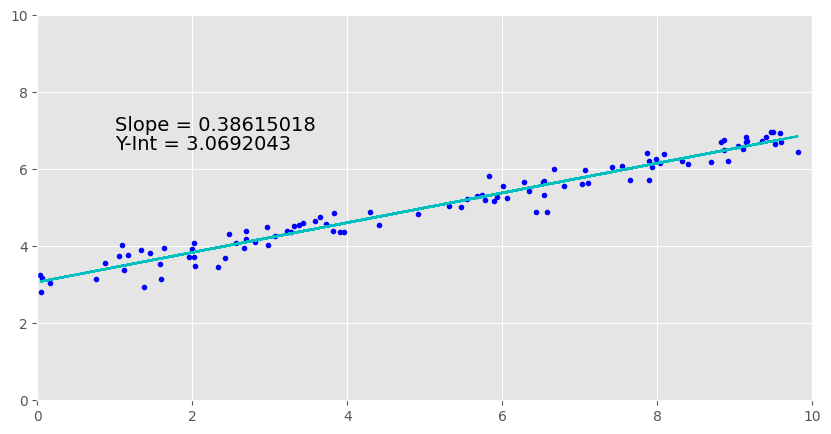
plt**.**text(1, 7.0, 'Slope = ' **+** str(slope[0]**.**numpy()), fontsize **=** 14)

plt**.**text(1, 6.5, 'Y-Int = ' **+** str(y\_int**.**numpy()), fontsize **=** 14)

plt**.**xlim(0, 10)

plt**.**ylim(0, 10)

plt**.**show()



5 Fitting Non-Linear Data using PyTorch

In this section, we will use a single neuron to fit a non-linear mathematical function of the form below:

y=θ0+θ1xcos(x)+θ2x2+noisey=θ0+θ1xcos(x)+θ2x2+noise

Notice this is a non-linear function but it is *linear* in the parameters (θ0,θ1,θ2θ0,θ1,θ2). In other words, the independent variables can be non-linear, as in the example above. The terms xcos(x)xcos(x) and x2x2 are the input features we will use to train the model.

An example of a functional form that is **not** linear in the parameters is given below since it cannot be expressed as the weighted sum of input features.

y=θ0+Xθ1cos(X+θ2)y=θ0+Xθ1cos(X+θ2)

5.1 Create Convenience Function

In [ ]:

**def** create\_nonlinear\_data(xmin **=** **-**10, xmax **=** 10, num\_data **=** 100, theta\_0 **=** 0, theta\_1 **=** .3, theta\_2 **=** .05, noise **=** .1):

X **=** np**.**linspace(xmin, xmax, num**=**num\_data)

y **=** theta\_0 **+** theta\_1**\***X**\***np**.**cos(X) **+** theta\_2**\***X**\*\***2 **+** noise**\***np**.**random**.**normal(size**=**num\_data)

X **=** torch**.**from\_numpy(X)

y **=** torch**.**from\_numpy(y)

X **=** X**.**view((len(X), 1))

y **=** y**.**view((len(y), 1))

**return** X, y

5.2 Generate Non-linear Data

In [ ]:

theta\_0 **=** **-**3

theta\_1 **=** .3

theta\_2 **=** .05

X, y **=** create\_nonlinear\_data(theta\_0 **=** theta\_0, theta\_1 **=** theta\_1, theta\_2 **=** theta\_2, noise **=** .3)

*# Create two features from the input data that match the functional form of the data we generated above.*

Xf **=** torch**.**cat((X**\***torch**.**cos(X), X**\***X), axis **=** 1)**.**float()

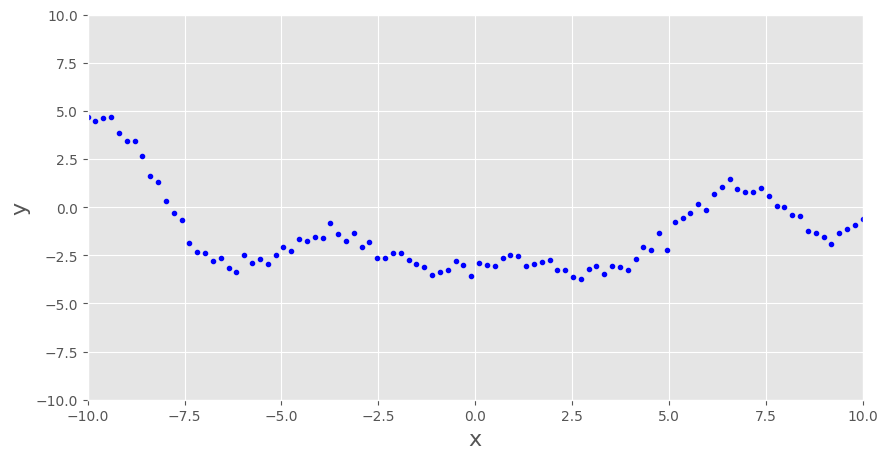
temp **=** Xf**.**numpy()

print('Xf contains two features based on X: ', temp**.**shape)

Xf contains two features based on X: (100, 2)

In [ ]:

plot\_data(X, y, (**-**10, 10), (**-**10, 10))



5.3 Create the PyTorch Model

This model has two inputs that represent xcos(x)xcos(x) and x2x2 with corresponding weights θ1θ1 ad θ2θ2 respectively, and also a bias term to represent θ0θ0. Notice that the model summary listed below has three trainable parameters (θ0,θ1,θ2θ0,θ1,θ2).

In [ ]:

**import** torch.optim **as** optim

model **=** nn**.**Linear(in\_features**=**2, out\_features**=**1)

5.4 Compile the Model

In [ ]:

optimizer **=** optim**.**Adam(model**.**parameters())

criterion **=** torch**.**nn**.**MSELoss()

5.5 Train the Model

In [ ]:

loss\_curve**=**[]

**for** epoch **in** range(10000):

optimizer**.**zero\_grad()

output **=** model(Xf)

loss **=** criterion(output, y**.**float())

loss**.**backward()

optimizer**.**step()

loss\_curve**.**append(loss**.**detach()**.**numpy()**.**item())

In [ ]:

loss\_values **=** loss\_curve

epochs **=** range(1, len(loss\_values) **+** 1)

plt**.**figure(figsize **=** [15, 5])

plt**.**plot(epochs, loss\_values, "orange", label**=**"Training loss")

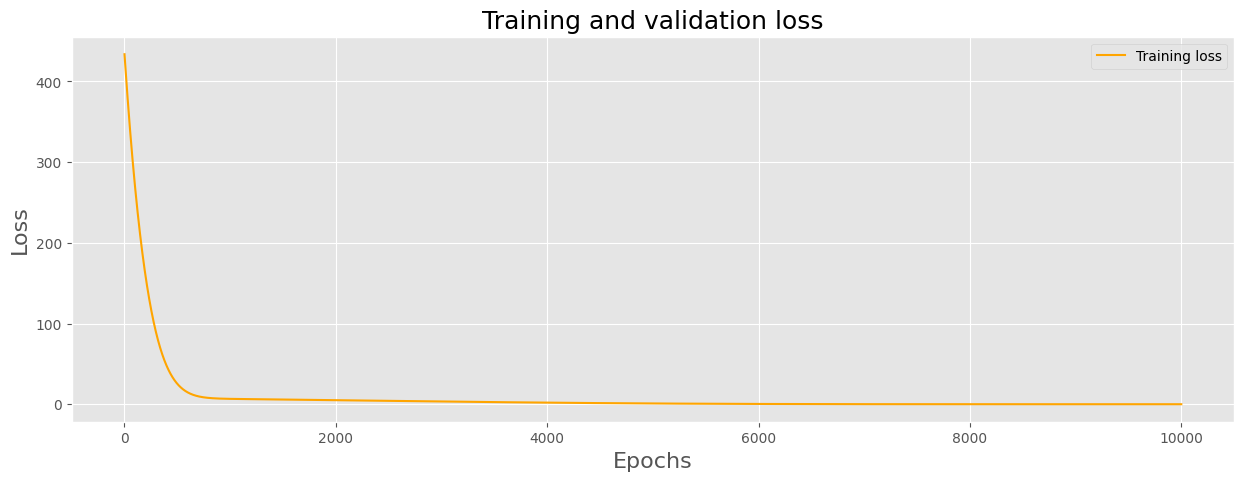
plt**.**title("Training and validation loss")

plt**.**xlabel("Epochs")

plt**.**ylabel("Loss")

plt**.**legend()

plt**.**show()



5.6 Predict Model Parameters and Display Results

In [ ]:

pred\_theta\_0 **=** model**.**state\_dict()['bias'][0]**.**numpy();

temp **=** model**.**state\_dict()['weight'][0]**.**numpy();

pred\_theta\_1 **=** temp[0]

pred\_theta\_2 **=** temp[1]

print('Actual Coefficients:\n')

print('theta\_0 = ', theta\_0)

print('theta\_1 = ', theta\_1)

print('theta\_2 = ', theta\_2)

print('\n')

print('Predicted Coefficients:\n')

print('theta\_0 = ', pred\_theta\_0)

print('theta\_1 = ', pred\_theta\_1)

print('theta\_2 = ', pred\_theta\_2)

pred\_y **=** model(Xf)**.**detach()**.**numpy()

Actual Coefficients:

theta\_0 = -3

theta\_1 = 0.3

theta\_2 = 0.05

Predicted Coefficients:

theta\_0 = -3.0488756

theta\_1 = 0.30952707

theta\_2 = 0.050533775

Notice that the parameters learned by the network are very close to the actual values that were used to generate the training data. As an exercise, you could try setting the noise to zero in the create\_nonlinear\_data() function call and confirm that the learned parameters match the parameters that were used to generate the training data.

In [ ]:

plt**.**plot(X, y, 'b.')

plt**.**plot(X, pred\_y, 'c-')

plt**.**text(**-**7, 7.0, 'theta\_0 = ' **+** str(pred\_theta\_0), fontsize**=**14)

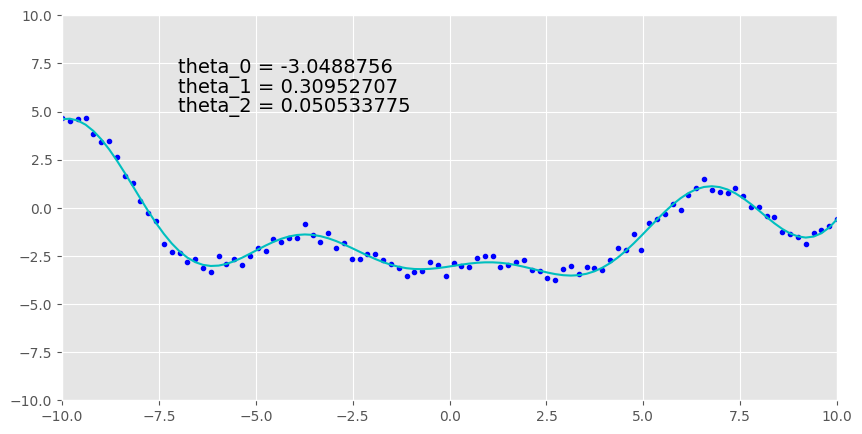
plt**.**text(**-**7, 6.0, 'theta\_1 = ' **+** str(pred\_theta\_1), fontsize**=**14)

plt**.**text(**-**7, 5.0, 'theta\_2 = ' **+** str(pred\_theta\_2), fontsize**=**14)

plt**.**xlim(**-**10, 10)

plt**.**ylim(**-**10, 10)

plt**.**show()



6. Conclusion

In this notebook, we introduced the concept of linear regression in a machine learning context in which one or more features are used to predict a single scalar value. We covered the mathematical model and notation on which linear regression is based.

We demonstrated that linear regression has a closed-form analytic solution derived from ordinary least squares, known as the normal equations but also highlighted that it’s often not a practical approach for many real-world problems and that gradient descent is most often the preferred method.

We showed that a solution to linear regression can also be viewed as a linear neural network, and we used PyTorch to help us model such as case.

Finally, We also showed that some non-linear mathematical functions can be solved using linear regression as long as the model is linear in the parameters (i.e., the target value can be expressed as the weighted sum of the input features).

Data Processing

In this notebook, we will cover the topic of data processing which includes exploring and cleaning datasets, as well as splitting the data between train and test, in preparation for model training and testing. We will also cover the topic of data normalization, which is a preprocessing technique that is often used to scale feature data prior to training models. We will learn how to create a normalization layer in PyTorch that automatically normalizes feature data for this purpose.

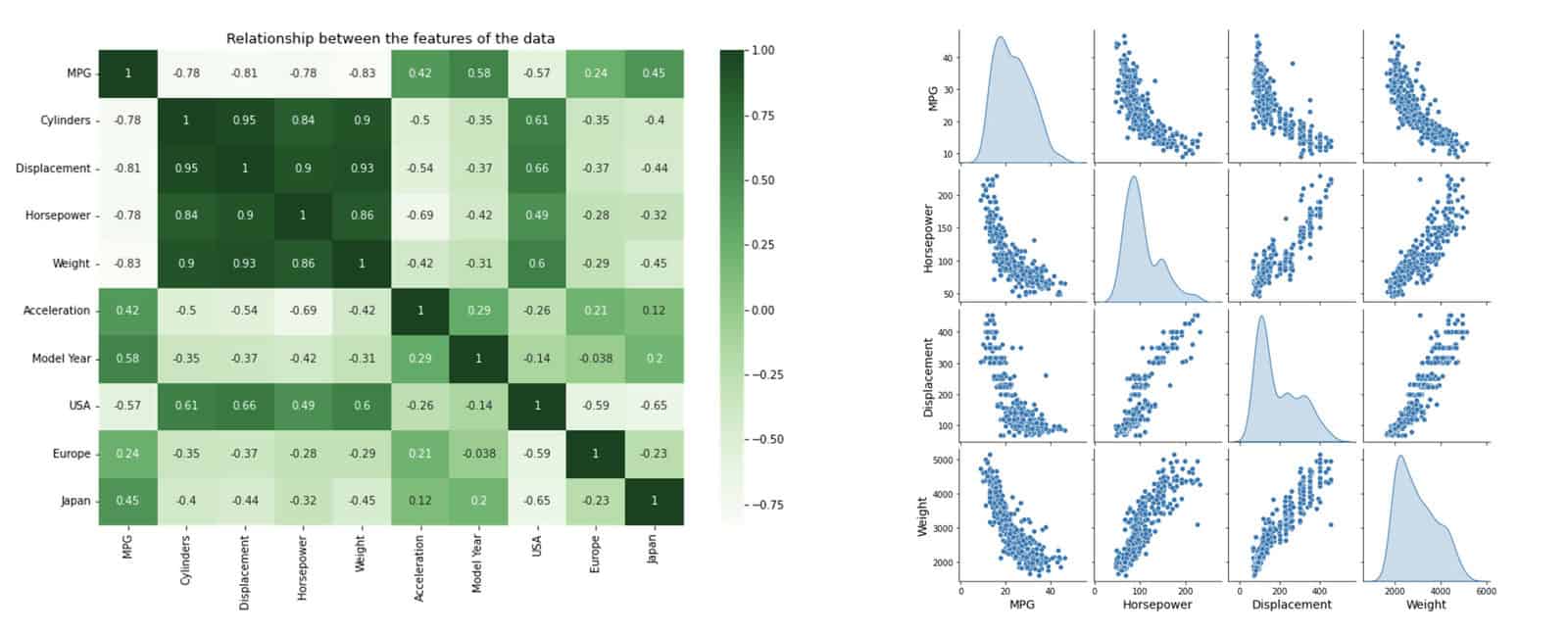


Table of Contents

* [1 Load the Auto MPG Dataset](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/02_Auto-MPG_Data_Processing_v2.html#1-Load-the-Auto-MPG-Dataset)
* [2 Clean the Dataset](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/02_Auto-MPG_Data_Processing_v2.html#2-Clean-the-Dataset)
* [3 Explore the Dataset](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/02_Auto-MPG_Data_Processing_v2.html#3-Explore-the-Dataset)
* [4 Split the Dataset into Train and Test](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/02_Auto-MPG_Data_Processing_v2.html#4-Split-the-Dataset-into-Train-and-Test)
* [5 Check Dataset Statistics](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/02_Auto-MPG_Data_Processing_v2.html#5-Check-Dataset-Statistics)
* [6 Split the Features from Target Values](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/02_Auto-MPG_Data_Processing_v2.html#6-Split-the-Features-from-Target-Values)
* [7 Normalize the Feature Data](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/02_Auto-MPG_Data_Processing_v2.html#7-Normalize-the-Feature-Data)
* [8 Conclusion](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/02_Auto-MPG_Data_Processing_v2.html#8-Conclusion)

In [ ]:

**import** os

**import** math

**import** numpy **as** np

**from** numpy **import** loadtxt

**import** matplotlib.pyplot **as** plt

**import** seaborn **as** sns

**import** pandas **as** pd

**from** zipfile **import** ZipFile, BadZipFile

**import** requests

plt**.**rcParams["figure.figsize"] **=** (15, 8)

plt**.**rcParams['axes.titlesize'] **=** 16

plt**.**rcParams['axes.labelsize'] **=** 14

1 Load the Auto MPG Dataset

In this notebook, we will be working with the Auto MPG dataset from the UC Irvine machine learning repository [here](https://archive.ics.uci.edu/dataset/9/auto+mpg). This data set contains nearly 400 samples of automobile data from the 1970s. There are eight data fields in the dataset consisting of various attributes such as vehicle weight and horsepower, and the goal is to use these features to predict the vehicle MPG.

In [ ]:

**def** download\_file(url, save\_name):

response **=** requests**.**get(url, stream**=True**)

**with** open(save\_name, 'wb') **as** file:

**for** chunk **in** response**.**iter\_content(chunk\_size**=**1024):

**if** chunk:

file**.**write(chunk)

print(f"Downloaded: {save\_name}")

In [ ]:

**def** unzip(zip\_file\_path):

**try**:

**with** ZipFile(zip\_file\_path, 'r') **as** z:

z**.**extractall("./")

print(f"Extracted: {os**.**path**.**splitext(zip\_file\_path)[0]}\n")

**except** FileNotFoundError:

print("File not found")

**except** BadZipFile:

print("Invalid or corrupt zip file")

**except** Exception **as** e:

print(f"Error occurred: {e}")

**return**

In [ ]:

URL **=** r"https://www.dropbox.com/scl/fi/dthi5o1822ayfb6wkqhnz/data\_auto\_mpg.zip?rlkey=hbyo3ndrdbissft57yt89wmak&dl=1"

archive\_name **=** "data\_auto\_mpg"

zip\_name **=** f"./{archive\_name}.zip"

**if** **not** os**.**path**.**exists(archive\_name):

download\_file(URL, zip\_name)

unzip(zip\_name)

In [ ]:

column\_names **=** [

"MPG",

"Cylinders",

"Displacement",

"Horsepower",

"Weight",

"Acceleration",

"Model Year",

"Origin",

]

*# Load the dataset into a Pandas data frame.*

raw\_dataset **=** pd**.**read\_csv(

os**.**path**.**join(archive\_name,"./auto-mpg.data"),

names**=**column\_names,

na\_values**=**"?",

comment**=**"\t",

sep**=**" ",

skipinitialspace**=True**,

)

dataset **=** raw\_dataset**.**copy()

dataset**.**tail()

Out[ ]:

|  | **MPG** | **Cylinders** | **Displacement** | **Horsepower** | **Weight** | **Acceleration** | **Model Year** | **Origin** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **393** | 27.0 | 4 | 140.0 | 86.0 | 2790.0 | 15.6 | 82 | 1 |
| **394** | 44.0 | 4 | 97.0 | 52.0 | 2130.0 | 24.6 | 82 | 2 |
| **395** | 32.0 | 4 | 135.0 | 84.0 | 2295.0 | 11.6 | 82 | 1 |
| **396** | 28.0 | 4 | 120.0 | 79.0 | 2625.0 | 18.6 | 82 | 1 |
| **397** | 31.0 | 4 | 119.0 | 82.0 | 2720.0 | 19.4 | 82 | 1 |

2 Clean the Dataset

Most datasets require some level of pre-processing, often referred to as "cleaning." For example, some fields may be missing numeric fields labelled in the dataset with various markers ('?', 'N/A', 'NaN', etc.). To check for such a condition, we can use the following command.

In [ ]:

dataset**.**isna()**.**sum()

Out[ ]:

MPG 0

Cylinders 0

Displacement 0

Horsepower 6

Weight 0

Acceleration 0

Model Year 0

Origin 0

dtype: int64

In [ ]:

dataset**.**shape

Out[ ]:

(398, 8)

In [ ]:

*# Use the dropna() method to remove data samples that are not fully populated.*

dataset **=** dataset**.**dropna()

dataset**.**shape

Out[ ]:

(392, 8)

In [ ]:

dataset**.**isna()**.**sum()

Out[ ]:

MPG 0

Cylinders 0

Displacement 0

Horsepower 0

Weight 0

Acceleration 0

Model Year 0

Origin 0

dtype: int64

In [ ]:

dataset**.**info()

<class 'pandas.core.frame.DataFrame'>

Int64Index: 392 entries, 0 to 397

Data columns (total 8 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 MPG 392 non-null float64

1 Cylinders 392 non-null int64

2 Displacement 392 non-null float64

3 Horsepower 392 non-null float64

4 Weight 392 non-null float64

5 Acceleration 392 non-null float64

6 Model Year 392 non-null int64

7 Origin 392 non-null int64

dtypes: float64(5), int64(3)

memory usage: 27.6 KB

3 Explore the Dataset

It's essential to understand the data you are working with. We can learn quite a bit about the various features in a dataset with two simple functions, as shown below. The first figure displays a correlation matrix that quantifies how correlated the various features are.

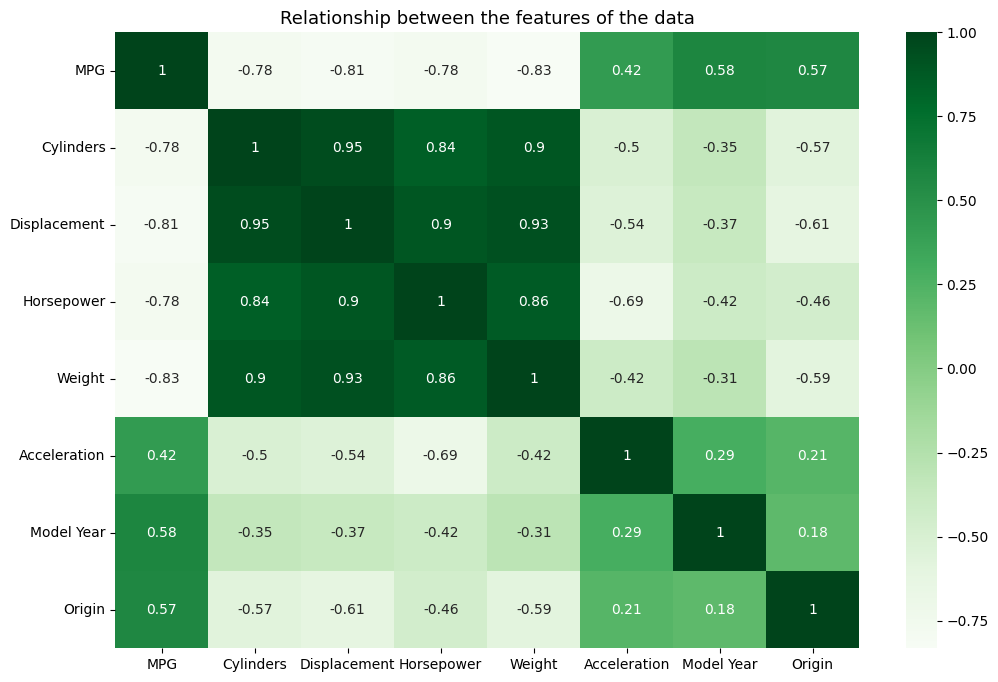
In [ ]:

plt**.**figure(figsize**=**(12, 8))

sns**.**heatmap(dataset**.**corr(), cmap**=**plt**.**cm**.**Greens, annot**=True**)

plt**.**title("Relationship between the features of the data", fontsize**=**13)

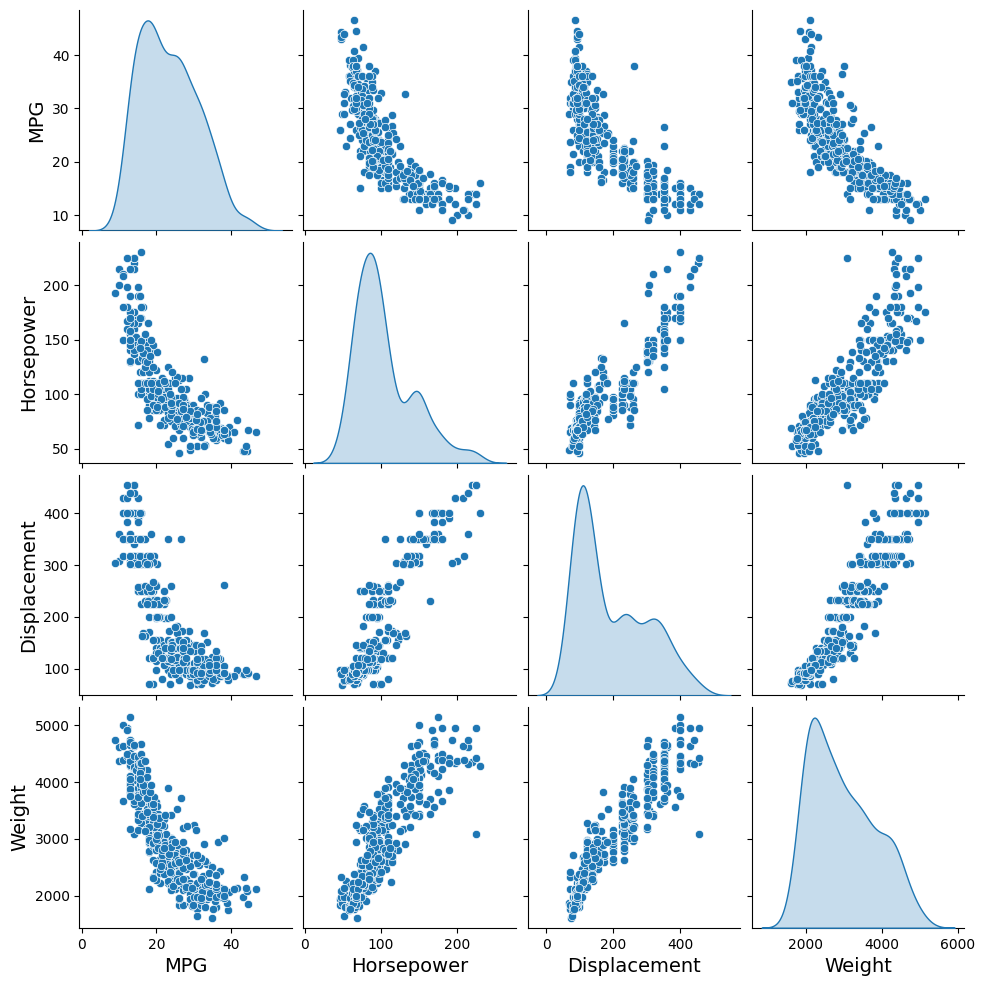
plt**.**show()



Seaborn has an excellent function called pairplot() that produces a grid of plots so you can better visualize the correlation between features. The distributions for each feature are shown along the diagonal. The type of distributions we're using, in this case, is called kde for kernel density estimate. You can think of these plots as smoothed histograms based on the sample data.

In [ ]:

sns**.**pairplot(dataset[['MPG', 'Horsepower', 'Displacement', 'Weight']], diag\_kind**=**'kde');



4 Split the Dataset into Train and Test

Let's now split the dataset into test and train components which is required in order to properly train and test models.

In [ ]:

train\_dataset **=** dataset**.**sample(frac**=**0.8, random\_state**=**42)

test\_dataset **=** dataset**.**drop(train\_dataset**.**index)

print(train\_dataset**.**shape)

print(test\_dataset**.**shape)

train\_dataset

(314, 8)

(78, 8)

Out[ ]:

|  | **MPG** | **Cylinders** | **Displacement** | **Horsepower** | **Weight** | **Acceleration** | **Model Year** | **Origin** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **79** | 26.0 | 4 | 96.0 | 69.0 | 2189.0 | 18.0 | 72 | 2 |
| **276** | 21.6 | 4 | 121.0 | 115.0 | 2795.0 | 15.7 | 78 | 2 |
| **248** | 36.1 | 4 | 91.0 | 60.0 | 1800.0 | 16.4 | 78 | 3 |
| **56** | 26.0 | 4 | 91.0 | 70.0 | 1955.0 | 20.5 | 71 | 1 |
| **393** | 27.0 | 4 | 140.0 | 86.0 | 2790.0 | 15.6 | 82 | 1 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **218** | 36.0 | 4 | 79.0 | 58.0 | 1825.0 | 18.6 | 77 | 2 |
| **281** | 19.8 | 6 | 200.0 | 85.0 | 2990.0 | 18.2 | 79 | 1 |
| **342** | 30.0 | 4 | 135.0 | 84.0 | 2385.0 | 12.9 | 81 | 1 |
| **238** | 33.5 | 4 | 98.0 | 83.0 | 2075.0 | 15.9 | 77 | 1 |
| **209** | 19.0 | 4 | 120.0 | 88.0 | 3270.0 | 21.9 | 76 | 2 |

314 rows × 8 columns

5 Check Dataset Statistics

Let's take a look at the magnitude of the various features. As you can see from the table below, the various features have a wide range of values spanning three orders of magnitude. When the feature data varies so widley, it is generally advised to scale the feature data as a pre-processing step before training a model. Scaling of the input features will be discussed further below.

In [ ]:

dataset**.**describe()**.**transpose()[['mean', 'std']]

Out[ ]:

|  | **mean** | **std** |
| --- | --- | --- |
| **MPG** | 23.445918 | 7.805007 |
| **Cylinders** | 5.471939 | 1.705783 |
| **Displacement** | 194.411990 | 104.644004 |
| **Horsepower** | 104.469388 | 38.491160 |
| **Weight** | 2977.584184 | 849.402560 |
| **Acceleration** | 15.541327 | 2.758864 |
| **Model Year** | 75.979592 | 3.683737 |
| **Origin** | 1.576531 | 0.805518 |

6 Split the Features from Target Values

Since the features and the target value are contained in the same dataframe we will separate them into two dataframes to keep them isolated. This also makes it easier to manage the data.

In [ ]:

X\_train **=** train\_dataset**.**copy()

X\_test **=** test\_dataset**.**copy()

*# Separate target values from features.*

y\_train **=** X\_train**.**pop('MPG')

y\_test **=** X\_test**.**pop('MPG')

In [ ]:

X\_train\_stats **=** X\_train**.**describe()**.**transpose()[['mean', 'std']]

X\_train\_stats

Out[ ]:

|  | **mean** | **std** |
| --- | --- | --- |
| **Cylinders** | 5.531847 | 1.729449 |
| **Displacement** | 197.855096 | 106.501896 |
| **Horsepower** | 105.971338 | 39.636557 |
| **Weight** | 3005.745223 | 859.060925 |
| **Acceleration** | 15.510828 | 2.803560 |
| **Model Year** | 75.910828 | 3.688989 |
| **Origin** | 1.579618 | 0.808322 |

7 Normalize the Feature Data

As mentioned above, this dataset contains a wide range of feature values, and it is often recommended to scale features so that they span a similar value range. One reason this is important is that the features are multiplied by the model weights. So, the scale of the outputs and the scale of the gradients are affected by the scale of the inputs. Although a model might converge without feature scaling, scaling features makes training much more stable and also facilitates the optimization process by allowing gradient descent to converge much faster.

Various methods can be used to scale features, but normalization and standardization are the most commonly used methods. Normalization usually refers to min/max scaling, where each feature is scaled to the range [0, 1] as shown below for each feature (xixi):

xi=xi−minximaxxi−minxixi=xi−minximaxxi−minxi

Standardization (also referred to as z-score scaling) assume the original data is normally distributed and scales the feature to have zero mean and a standard deviation of 1. This is accomplished for each feature (xixi) by subtracting the mean of the feature data from each data point (referred to as mean subtraction) and then dividing that result by the standard deviation for the feature data as shown below:

xi=xi−μiσixi=xi−μiσi

In the code cell below, we are using numpy to calculate the mean and standard deviation for feature Horsepower of training dataset which are then used to normalize the data.

**Note**: The normalization parameters (mean and standard deviation) are derived only from the **training** dataset, but will be applied to all of the data (train, validation and test).

In [ ]:

*# Convert the Horsepower feature in the dataframe to a NumPy array.*

hp **=** np**.**array(X\_train['Horsepower'])

*# Create the normalization layer (for Horsepower).*

hp **=** np**.**array(X\_train['Horsepower'])

mean **=** np**.**mean(np**.**array(X\_train['Horsepower']))

std **=** np**.**std(np**.**array(X\_train['Horsepower']))

print('Mean: ', mean)

print('Std: ', std)*#degree of freedom(ddof=1) cannot be changed*

print('Count: ', hp**.**shape[0])

Mean: 105.97133757961784

Std: 39.57339154776915

Count: 314

8 Conclusion

In this notebook we introduced two python packages (Pandas and Seaborn) that are often used to manage and explore datasets. In the next notebook we will use this dataset to perform both linear and non-linear regression using PyTorch.

Linear Regression to Deep Learning using PyTorch

In this notebook, we will continue with the topic of linear regression, and we will work with the Auto-MPG dataset that contains multiple input features. The Auto-MPG dataset contains 398 sample instances and 7 features that can help us analyze and predict continuous value outcomes.

First, we will construct and train a single-layer neural network that incorporates a single input feature. From there, we will advance to performing a multi-variate linear regression, considering multiple input features. Next, we will explore how adding hidden layers in the network with non-linear activation functions will produce a non-linear response.

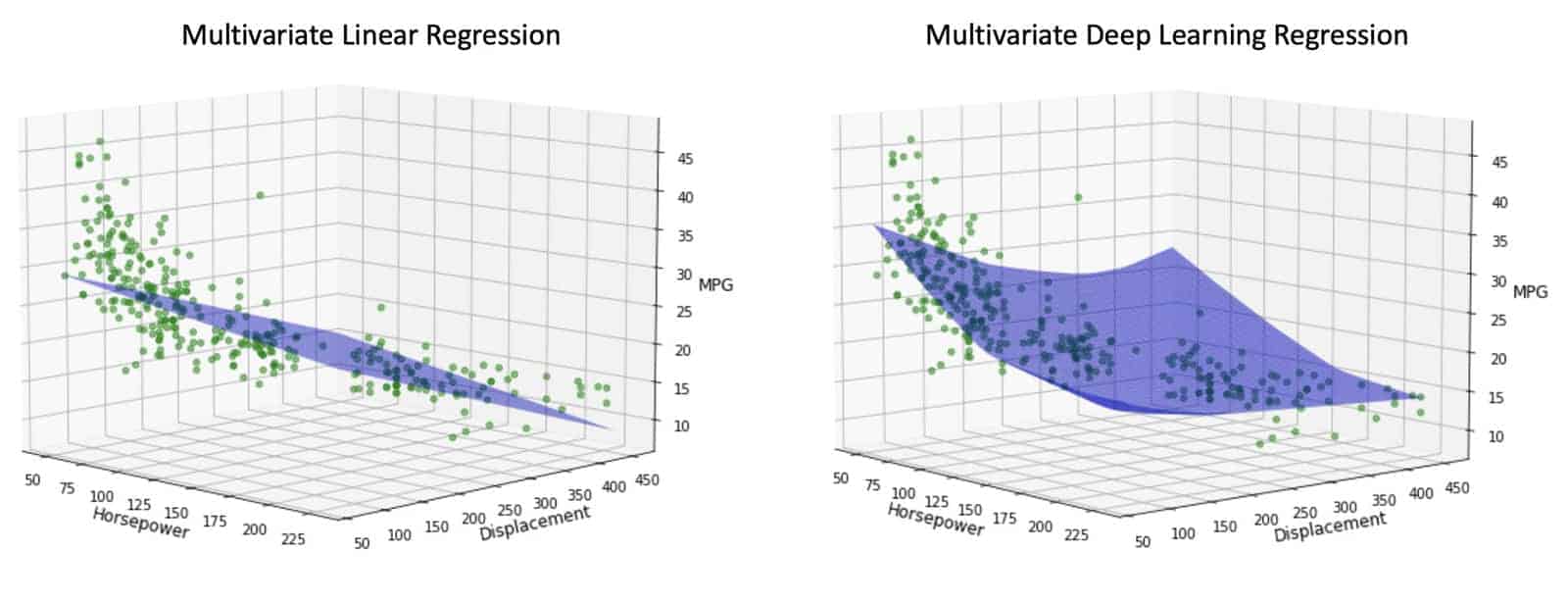


Table of Contents

* [1 Load and Pre-Process the Auto MPG Dataset](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/03_Linear_Regression_Auto_MPG_v3.html#1-Load-and-Explore-the-Auto-MPG-Dataset)
* [2 PyTorch Training Workflow](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/03_Linear_Regression_Auto_MPG_v3.html#2-Keras-Training-Workflow)
* [3 Linear Regression](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/03_Linear_Regression_Auto_MPG_v3.html#3-Linear-Regression)
* [4 Multivariate Linear Regression](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/03_Linear_Regression_Auto_MPG_v3.html#4-Multivariate-Linear-Regression)
* [5 Deep Learning with a Single Feature](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/03_Linear_Regression_Auto_MPG_v3.html#5-Deep-Learning-with-a-Single-Feature)
* [6 Deep Learning with Multiple Features](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/03_Linear_Regression_Auto_MPG_v3.html#6-Deep-Learning-with-Multiple-Features)
* [7 Comparison of Test Results](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/03_Linear_Regression_Auto_MPG_v3.html#7-Comparison-of-Test-Results)
* [8 Conclusion](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/03_Linear_Regression_Auto_MPG_v3.html#8-Conclusion)

Let's start by importing all the functions and classes we will use in this notebook.

In [ ]:

*# Import necesary support libraries.*

**import** os

**import** numpy **as** np

**import** random

**import** requests

**import** math

**import** warnings

**from** IPython.display **import** clear\_output

**from** zipfile **import** ZipFile, BadZipFile

**import** numpy **as** np

**import** pandas **as** pd

**import** matplotlib.pyplot **as** plt

**from** sklearn.model\_selection **import** train\_test\_split

*# To download the dataset.*

**from** urllib.request **import** urlretrieve

*# Necessary PyTorch imports.*

**import** torch

**import** torch.nn **as** nn

**from** torch **import** optim

**import** torch.nn.functional **as** F

*# from torchinfo import summary*

*# Set plotting related parameters.*

plt**.**rcParams["figure.figsize"] **=** (15, 6)

plt**.**rcParams["axes.titlesize"] **=** 16

plt**.**rcParams["axes.labelsize"] **=** 12

warnings**.**filterwarnings(action**=**'ignore', category**=**UserWarning)

*# Text formatting*

BOLD **=** "\033[1m"

END **=** "\033[0m"

In [ ]:

**def** system\_config(SEED\_VALUE**=**42, package\_list**=None**):

"""

Configures the system environment for PyTorch-based operations.

Args:

SEED\_VALUE (int): Seed value for random number generation. Default is 42.

package\_list (str): String containing a list of additional packages to install

for Google Colab or Kaggle. Default is None.

Returns:

tuple: A tuple containing the device name as a string and a boolean indicating GPU availability.

"""

random**.**seed(SEED\_VALUE)

np**.**random**.**seed(SEED\_VALUE)

torch**.**manual\_seed(SEED\_VALUE)

**def** is\_running\_in\_colab():

**return** 'COLAB\_GPU' **in** os**.**environ

**def** is\_running\_in\_kaggle():

**return** 'KAGGLE\_KERNEL\_RUN\_TYPE' **in** os**.**environ

*#--------------------------------*

*# Check for the availability GPUs.*

*#--------------------------------*

**if** torch**.**cuda**.**is\_available():

print('Using CUDA GPU')

*# This section for installing packages required by Colab.*

**if** is\_running\_in\_colab() **or** is\_running\_in\_kaggle():

print('Installing required packages...')

**!**pip install **{**package\_list**}**

*# Set the device to the first CUDA device.*

DEVICE **=** torch**.**device('cuda')

print("Device: ", DEVICE)

GPU\_AVAILABLE **=** **True**

torch**.**cuda**.**manual\_seed(SEED\_VALUE)

torch**.**cuda**.**manual\_seed\_all(SEED\_VALUE)

*# Performance and deterministic behavior.*

torch**.**backends**.**cudnn**.**enabled **=** **True** *# Provides highly optimized primitives for DL operations.*

torch**.**backends**.**cudnn**.**deterministic **=** **True** *# Insures deterministic even when above cudnn is enabled.*

torch**.**backends**.**cudnn**.**benchmark **=** **False** *# Setting to True can cause non-deterministic behavior.*

**elif** torch**.**backends**.**mps**.**is\_available() **and** torch**.**backends**.**mps**.**is\_built():

print('Using Apple Silicon GPU')

*# Set the device to the Apple Silicon GPU Metal Performance Shader (MPS).*

DEVICE **=** torch**.**device("mps")

print("Device: ", DEVICE)

*# Environment variable that allows PyTorch to fall back to CPU execution*

*# when encountering operations that are not currently supported by MPS.*

os**.**environ['PYTORCH\_ENABLE\_MPS\_FALLBACK'] **=** '1'

GPU\_AVAILABLE **=** **True**

torch**.**mps**.**manual\_seed(SEED\_VALUE)

torch**.**use\_deterministic\_algorithms(**True**)

**else**:

print('Using CPU')

DEVICE **=** torch**.**device('cpu')

print("Device: ", DEVICE)

GPU\_AVAILABLE **=** **False**

**if** is\_running\_in\_colab() **or** is\_running\_in\_kaggle():

print('Installing required packages...')

**!**pip install **{**package\_list**}**

print('Note: Change runtime type to GPU for better performance.')

torch**.**use\_deterministic\_algorithms(**True**)

**return** str(DEVICE), GPU\_AVAILABLE

In [ ]:

*# Additional packages required for Google Colab or Kaggle.*

package\_list **=** "torchinfo"

DEVICE, GPU\_AVAILABLE **=** system\_config(package\_list**=**package\_list)

Using CPU

Device: cpu

Installing required packages...

Collecting torchinfo

Downloading torchinfo-1.8.0-py3-none-any.whl (23 kB)

Installing collected packages: torchinfo

Successfully installed torchinfo-1.8.0

Note: Change runtime type to GPU for better performance.

In [ ]:

**from** torchinfo **import** summary

1 Load and Pre-Process the Auto MPG Dataset

In this notebook, we will be working with the Auto MPG dataset from the UC Irvine machine learning repository [here](https://archive.ics.uci.edu/ml/datasets.php). This data set contains nearly 400 samples of automobile data from the 1970s. There are eight data fields in the dataset consisting of various attributes such as vehicle weight and horsepower, and the goal is to use these features to *predict the vehicle MPG.*

In [ ]:

**def** download\_file(url, save\_name):

response **=** requests**.**get(url, stream**=True**)

**with** open(save\_name, 'wb') **as** file:

**for** chunk **in** response**.**iter\_content(chunk\_size**=**1024):

**if** chunk:

file**.**write(chunk)

print(f"Downloaded: {save\_name}")

In [ ]:

**def** unzip(zip\_file\_path**=None**):

**try**:

**with** ZipFile(zip\_file\_path, 'r') **as** z:

z**.**extractall("./")

print(f"Extracted: {os**.**path**.**splitext(zip\_file\_path)[0]}\n")

**except** FileNotFoundError:

print("File not found")

**except** BadZipFile:

print("Invalid or corrupt zip file")

**except** Exception **as** e:

print(f"Error occurred: {e}")

**return**

In [ ]:

URL **=** r"https://www.dropbox.com/scl/fi/dthi5o1822ayfb6wkqhnz/data\_auto\_mpg.zip?rlkey=hbyo3ndrdbissft57yt89wmak&dl=1"

archive\_name **=** "data\_auto\_mpg"

zip\_name **=** f"./{archive\_name}.zip"

**if** **not** os**.**path**.**exists(archive\_name):

download\_file(URL, zip\_name)

unzip(zip\_name)

Downloaded: ./data\_auto\_mpg.zip

Extracted: ./data\_auto\_mpg

In [ ]:

column\_names **=** ["MPG", "Cylinders", "Displacement", "Horsepower", "Weight", "Acceleration", "Model Year", "Origin"]

*# Load the dataset into a Pandas data frame.*

raw\_dataset **=** pd**.**read\_csv(

os**.**path**.**join(archive\_name,"./auto-mpg.data"),

names**=**column\_names,

na\_values**=**"?",

comment**=**"\t",

sep**=**" ",

skipinitialspace**=True**,

)

dataset **=** raw\_dataset**.**copy()

print(dataset**.**tail())

dataset**.**columns

MPG Cylinders Displacement Horsepower Weight Acceleration \

393 27.0 4 140.0 86.0 2790.0 15.6

394 44.0 4 97.0 52.0 2130.0 24.6

395 32.0 4 135.0 84.0 2295.0 11.6

396 28.0 4 120.0 79.0 2625.0 18.6

397 31.0 4 119.0 82.0 2720.0 19.4

Model Year Origin

393 82 1

394 82 2

395 82 1

396 82 1

397 82 1

Out[ ]:

Index(['MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight',

'Acceleration', 'Model Year', 'Origin'],

dtype='object')

1.1 Clean the Dataset

In [ ]:

dataset**.**isna()**.**sum()

Out[ ]:

MPG 0

Cylinders 0

Displacement 0

Horsepower 6

Weight 0

Acceleration 0

Model Year 0

Origin 0

dtype: int64

In [ ]:

*# Use the dropna() method to remove data samples that are not fully populated.*

dataset **=** dataset**.**dropna()

dataset**.**shape

Out[ ]:

(392, 8)

1.2 Split the Dataset into Train and Test

Let's now split the dataset into test and train components so we can train some models and evaluate them.

In [ ]:

train\_dataset **=** dataset**.**sample(frac**=**0.8, random\_state**=**42)

test\_dataset **=** dataset**.**drop(train\_dataset**.**index)

print(train\_dataset**.**shape)

print(test\_dataset**.**shape)

(314, 8)

(78, 8)

In [ ]:

train\_dataset

Out[ ]:

|  | **MPG** | **Cylinders** | **Displacement** | **Horsepower** | **Weight** | **Acceleration** | **Model Year** | **Origin** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **79** | 26.0 | 4 | 96.0 | 69.0 | 2189.0 | 18.0 | 72 | 2 |
| **276** | 21.6 | 4 | 121.0 | 115.0 | 2795.0 | 15.7 | 78 | 2 |
| **248** | 36.1 | 4 | 91.0 | 60.0 | 1800.0 | 16.4 | 78 | 3 |
| **56** | 26.0 | 4 | 91.0 | 70.0 | 1955.0 | 20.5 | 71 | 1 |
| **393** | 27.0 | 4 | 140.0 | 86.0 | 2790.0 | 15.6 | 82 | 1 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **218** | 36.0 | 4 | 79.0 | 58.0 | 1825.0 | 18.6 | 77 | 2 |
| **281** | 19.8 | 6 | 200.0 | 85.0 | 2990.0 | 18.2 | 79 | 1 |
| **342** | 30.0 | 4 | 135.0 | 84.0 | 2385.0 | 12.9 | 81 | 1 |
| **238** | 33.5 | 4 | 98.0 | 83.0 | 2075.0 | 15.9 | 77 | 1 |
| **209** | 19.0 | 4 | 120.0 | 88.0 | 3270.0 | 21.9 | 76 | 2 |

314 rows × 8 columns

In [ ]:

test\_dataset

Out[ ]:

|  | **MPG** | **Cylinders** | **Displacement** | **Horsepower** | **Weight** | **Acceleration** | **Model Year** | **Origin** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | 15.0 | 8 | 350.0 | 165.0 | 3693.0 | 11.5 | 70 | 1 |
| **13** | 14.0 | 8 | 455.0 | 225.0 | 3086.0 | 10.0 | 70 | 1 |
| **20** | 25.0 | 4 | 110.0 | 87.0 | 2672.0 | 17.5 | 70 | 2 |
| **21** | 24.0 | 4 | 107.0 | 90.0 | 2430.0 | 14.5 | 70 | 2 |
| **35** | 17.0 | 6 | 250.0 | 100.0 | 3329.0 | 15.5 | 71 | 1 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **381** | 36.0 | 4 | 107.0 | 75.0 | 2205.0 | 14.5 | 82 | 3 |
| **388** | 26.0 | 4 | 156.0 | 92.0 | 2585.0 | 14.5 | 82 | 1 |
| **391** | 36.0 | 4 | 135.0 | 84.0 | 2370.0 | 13.0 | 82 | 1 |
| **395** | 32.0 | 4 | 135.0 | 84.0 | 2295.0 | 11.6 | 82 | 1 |
| **397** | 31.0 | 4 | 119.0 | 82.0 | 2720.0 | 19.4 | 82 | 1 |

78 rows × 8 columns

1.3 Split the Features from Target Values

Since the features and the target value are contained in the same dataframe we will separate them into two dataframes to keep them isolated. This also makes it easier to manage the data.

In [ ]:

X\_train **=** train\_dataset**.**copy()

X\_test **=** test\_dataset**.**copy()

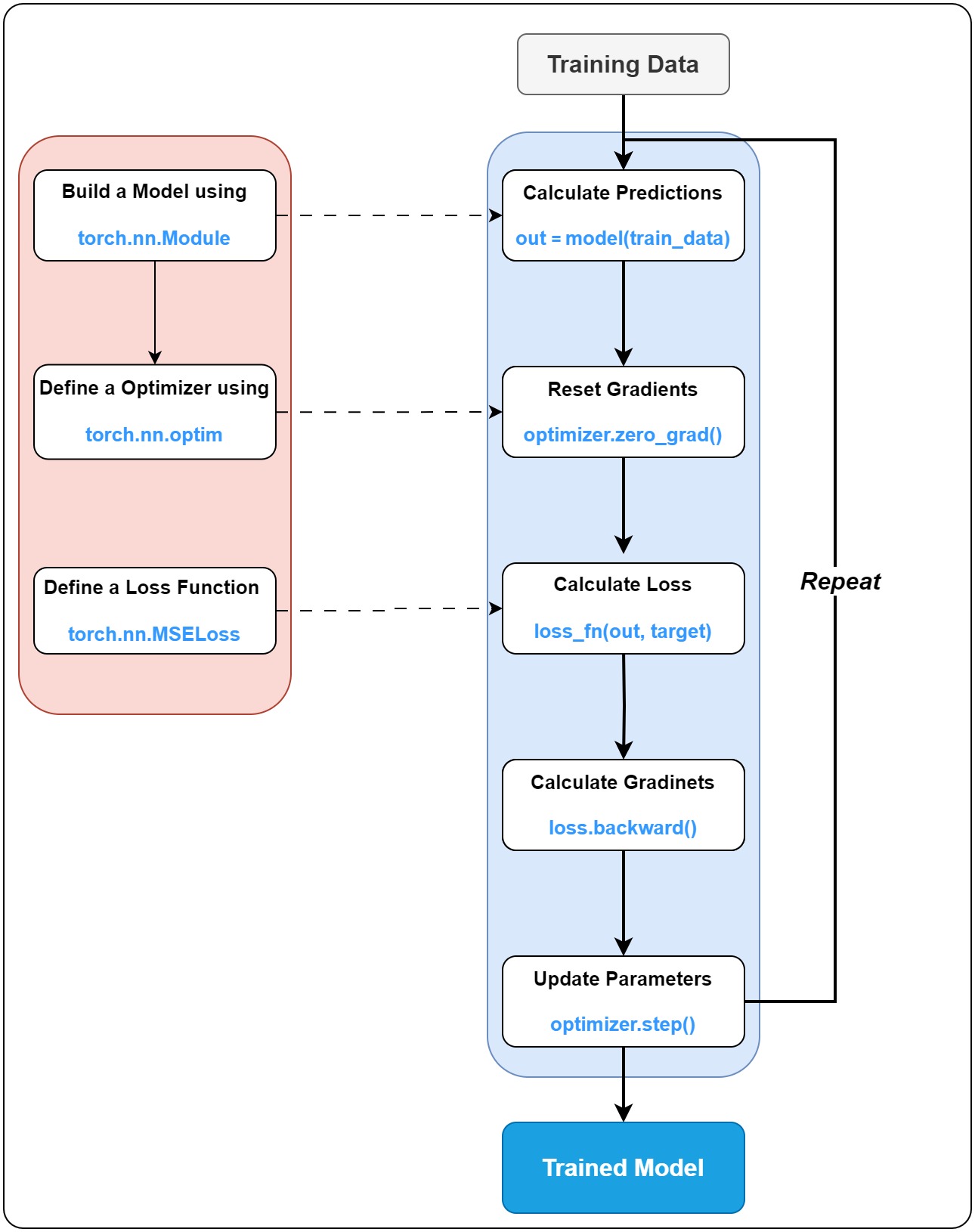
*# Separate target values from features.*

y\_train **=** X\_train**.**pop('MPG')

y\_test **=** X\_test**.**pop('MPG')

2 PyTorch Training Workflow

The diagram shown below summarizes the training workflow in PyTorch that is used to create a trained model.



1. We begin by defining a PyTorch model using torch.nn.Module.
2. Next, we define the optimizer and loss criterion to use on the dataset.
3. The training loop consits of 5 main lines of code which you'll find in any PyTorch based project.
   1. Generate predictions using the model: out = model(input\_data)
   2. Clear any previously calculated gradients (is effective starting from the second pass through the model).
   3. Calculate the loss: loss = loss\_fn(out, target)
   4. Calculate gradients of the parameter based on the loss: loss.backward()
   5. Finally, we update the model parameters: optimizer.step().

3 Linear Regression

**Linear regression** is a statistical method and a type of predictive modelling technique. It investigates the relationship between a dependent (target) variable and one (simple linear regression) or more (multiple linear regression) independent variables. The relationships are modeled using linear predictor functions where the unknown parameters are estimated from the data. The linear model is used to predict output based on the weighted sum of the input features. The output in this case, is a real-valued number:

hθ(x)=y′=θ0+θ1x1+θ2x2+...hθ(x)=y′=θ0+θ1x1+θ2x2+...

The general notation can be simplified to the following by letting x0=1x0=1 which means that θ0θ0 becomes the bias term, and therefore the above expression can be simplified as follows:

h(x)=y′=n∑i=0θixi=θTxh(x)=y′=∑i=0nθixi=θTx

Where,

* x0...xnx0...xn are input features or independent variables.
* θ0...θnθ0...θn are the parameters/weights of the linear model.
* y′y′ is a real-valued predicted target variable in a regression problem.

In this section, we will create a PyTorch model with a single output neuron (using a linear activation) to perform linear regression. We will only use a single input variable from the dataset.

Specifically, we'll use Horsepower as the input feature to predict the MPG of the vehicle. Thus, the model will be a straight line with two unknown coefficients (the slope and the intercept). We will train the model to determine the coefficients.

hθ(x)=y′=θ0+θ1x1hθ(x)=y′=θ0+θ1x1

When working with neural networks, it is more common to use the notation below:

y′=b+w1x1y′=b+w1x1

3.1 Normalize the Feature Data

As mentioned above, this dataset contains a wide range of feature values, and often in machine learning and deep learning it is recommended to scale features so that they span a similar value range. One reason this is important is that the features are multiplied by the model weights. So, the scale of the outputs and the scale of the gradients are affected by the scale of the inputs. Although a model might converge without feature scaling, scaling features makes training much more stable and also facilitates the optimization process by allowing gradient descent to converge much faster.

Various methods can be used to scale features, but normalization and standardization are the most commonly used methods. Normalization usually refers to min/max scaling, where each feature is scaled to the range [0, 1] as shown below for each feature (xixi):

xi=xi−minximaxxi−minxixi=xi−minximaxxi−minxi

Standardization (also referred to as z-score scaling) assume the original data is normally distributed and scales the feature to have zero mean and a standard deviation of 1. This is accomplished for each feature (xixi) by subtracting the mean of the feature data from each data point (referred to as mean subtraction) and then dividing that result by the standard deviation for the feature data as shown below:

xi=xi−μiσixi=xi−μiσi

In the code cell below, we are using numpy to calculate the mean and standard deviation for the Horsepower and Displacement features of the training dataset which are then used to normalize the data.

**Note**: The normalization parameters (mean and standard deviation) are derived only from the **training** dataset, but will be applied to all of the data (train, validation and test).

In [ ]:

*# Calculate mean and standard deviation for Horsepower*

mean\_hp **=** np**.**mean(X\_train['Horsepower'])

std\_hp **=** np**.**std(X\_train['Horsepower'])

print("\nFor 'Horsepower' column:")

print("Mean: ", mean\_hp)

print("Std: ", std\_hp)

print("Count: ", X\_train**.**shape[0])

*# Scale Horsepower feature*

X\_train["Horsepower\_scaled"] **=** (X\_train["Horsepower"] **-** mean\_hp) **/** std\_hp

X\_test["Horsepower\_scaled"] **=** (X\_test["Horsepower"] **-** mean\_hp) **/** std\_hp

*# Calculate mean and standard deviation for Displacement*

mean\_dis **=** np**.**mean(X\_train["Displacement"])

std\_dis **=** np**.**std(X\_train["Displacement"])

print("\nFor 'Displacement' column:")

print("Mean: ", mean\_dis)

print("Std: ", std\_dis)

print("Count: ", X\_train**.**shape[0])

*# Scale Displacement feature*

X\_train["Displacement\_scaled"] **=** (X\_train["Displacement"] **-** mean\_dis) **/** std\_dis

X\_test["Displacement\_scaled"] **=** (X\_test["Displacement"] **-** mean\_dis) **/** std\_dis

For 'Horsepower' column:

Mean: 105.97133757961784

Std: 39.57339154776915

Count: 314

For 'Displacement' column:

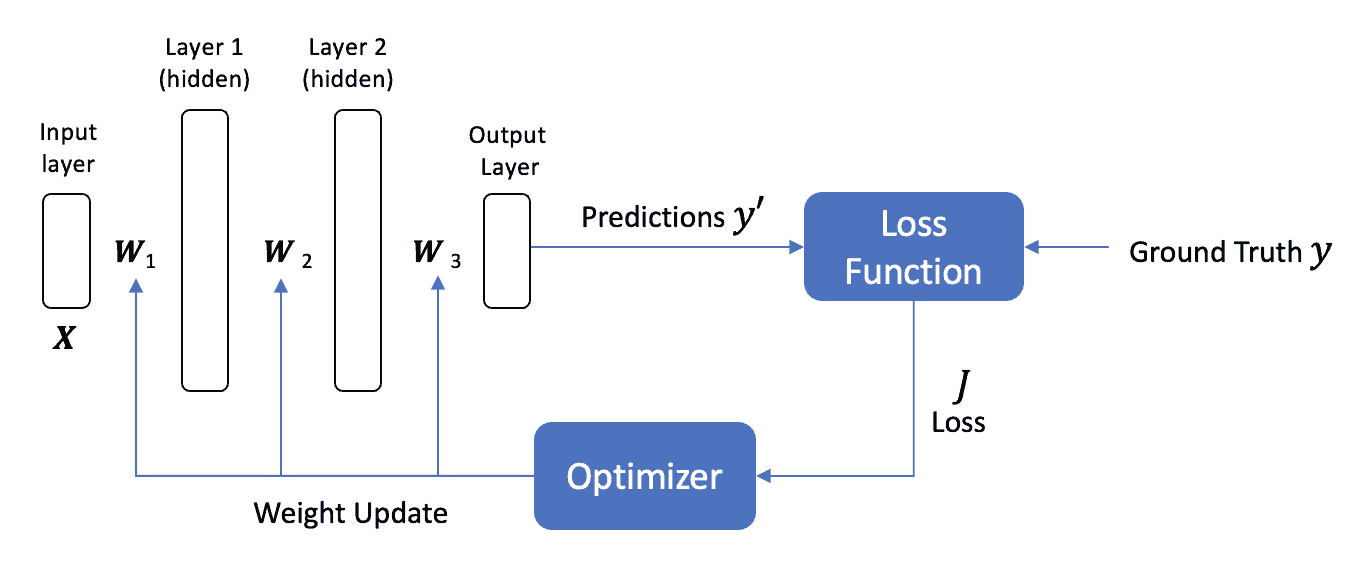
Mean: 197.85509554140128

Std: 106.33217177453977

Count: 314

3.2 Create PyTorch Model

Here we define the PyTorch model. The figure below shows the model architecture.



Note: The weight update process includes the bias term even though an update equations is not explciity shown in the figure.

In [ ]:

**class** Regressor\_1(nn**.**Module):

*# Initialize the parameter.*

**def** \_\_init\_\_(self, in\_features**=**1, out\_features**=**1):

super()**.**\_\_init\_\_()

*# Define a single Linear layer.*

self**.**linear\_1 **=** nn**.**Linear(in\_features**=**in\_features, out\_features**=**out\_features)

*# Forward pass*

**def** forward(self, x):

**return** self**.**linear\_1(x)

In [ ]:

*# Intialize an instance of the PyTorch model class.*

linear\_1d\_model **=** Regressor\_1(in\_features**=**1, out\_features**=**1)

batch\_size **=** 1

summary(linear\_1d\_model, input\_size**=**(batch\_size, 1,), device**=**"cpu", col\_names**=**("input\_size", "output\_size", "num\_params"))

Out[ ]:

===================================================================================================================

Layer (type:depth-idx) Input Shape Output Shape Param #

===================================================================================================================

Regressor\_1 [1, 1] [1, 1] --

├─Linear: 1-1 [1, 1] [1, 1] 2

===================================================================================================================

Total params: 2

Trainable params: 2

Non-trainable params: 0

Total mult-adds (M): 0.00

===================================================================================================================

Input size (MB): 0.00

Forward/backward pass size (MB): 0.00

Params size (MB): 0.00

Estimated Total Size (MB): 0.00

===================================================================================================================

In [ ]:

*# Initialize Optimizer by passing the parameters of the model and the learning rate to use.*

*# The model parameters passed to the optimizer will be updated during training.*

optimizer **=** optim**.**Adam(linear\_1d\_model**.**parameters(), lr**=**0.1)

*# Intialize a Loss function.*

*# Here we are using L1Loss that measures the mean absolute error (MAE) between the input x and target y.*

criterion **=** torch**.**nn**.**L1Loss()

3.3 Train the Model

It's now time to train the model using the single input feature. The training data is split into 70% training and 30% validation, to evaluate the model's performance during the training process.

In [ ]:

X\_train\_split, X\_val\_split, y\_train\_split, y\_val\_split **=** train\_test\_split(X\_train, y\_train, test\_size**=**0.3, random\_state**=**0)

In [ ]:

*# To record the traning and validation loss metrics.*

loss\_curve\_train\_linear\_1d **=** []

loss\_curve\_eval\_linear\_1d **=** []

X\_train\_hp **=** torch**.**from\_numpy(X\_train\_split["Horsepower\_scaled"]**.**values)**.**reshape(**-**1, 1)**.**to(torch**.**float32)

y\_train\_t **=** torch**.**from\_numpy(y\_train\_split**.**values)**.**reshape(**-**1, 1)**.**to(torch**.**float32) *# Train target converted to pytorch tensor.*

X\_val\_hp **=** torch**.**from\_numpy(X\_val\_split["Horsepower\_scaled"]**.**values)**.**reshape(**-**1, 1)**.**to(torch**.**float32)

y\_val\_t **=** torch**.**from\_numpy(y\_val\_split**.**values)**.**reshape(**-**1, 1)**.**to(torch**.**float32) *# Validation target converted to pytorch tensor.*

*# This is the training loop. We will go through the dataset 500 times.*

*# During each pass the model will be updated and get closer to the true target values.*

**for** epoch **in** range(500):

linear\_1d\_model**.**train() *# Set model in training mode.*

*# Not required here as the layers used behave the same way during trainig and evaluation*

output **=** linear\_1d\_model(X\_train\_hp) *# Perform forward pass through the model.*

loss **=** criterion(output, y\_train\_t) *# Calculate L1 loss on the model predictions.*

optimizer**.**zero\_grad() *# Reset gradients.*

loss**.**backward() *# Calcualte gradients based on the loss.*

optimizer**.**step() *# Update parameters.*

loss\_curve\_train\_linear\_1d**.**append(loss**.**detach()**.**item()) *# Record training loss.*

linear\_1d\_model**.**eval() *# Set model in evaluation mode.*

**with** torch**.**no\_grad():

output **=** linear\_1d\_model(X\_val\_hp) *# Perform forward pass through the validation set.*

loss **=** criterion(output, y\_val\_t) *# Calculate the loss on the validation set.*

loss\_curve\_eval\_linear\_1d**.**append(loss**.**item()) *# Record validation loss.*

*# # Print per epoch log table.*

*# # Slows down speed because we are printing and erasing the table continuously.*

*# clear\_output(wait=True)*

*# print(f"{BOLD}{'Epoch':^10}{END}|{BOLD}{'Loss':^15}{END}|{BOLD}{'Val\_loss':^10}{END}")*

*# print(f"{'':=^35}")*

*# print(f"{epoch+1:^10}|{loss\_curve\_train\_linear\_1d[-1]:^15.8f}|{loss\_curve\_eval\_linear\_1d[-1]:^10.3f}")*

In [ ]:

**def** plot\_loss(loss\_curve\_train, loss\_curve\_eval):

plt**.**figure(figsize**=**(15, 5))

plt**.**plot(loss\_curve\_train, label**=**"Loss")

plt**.**plot(loss\_curve\_eval, label**=**"Val Loss")

plt**.**ylim([0, 30])

plt**.**xlabel("Epoch")

plt**.**ylabel("Error [MPG]")

plt**.**legend()

plt**.**grid(**True**)

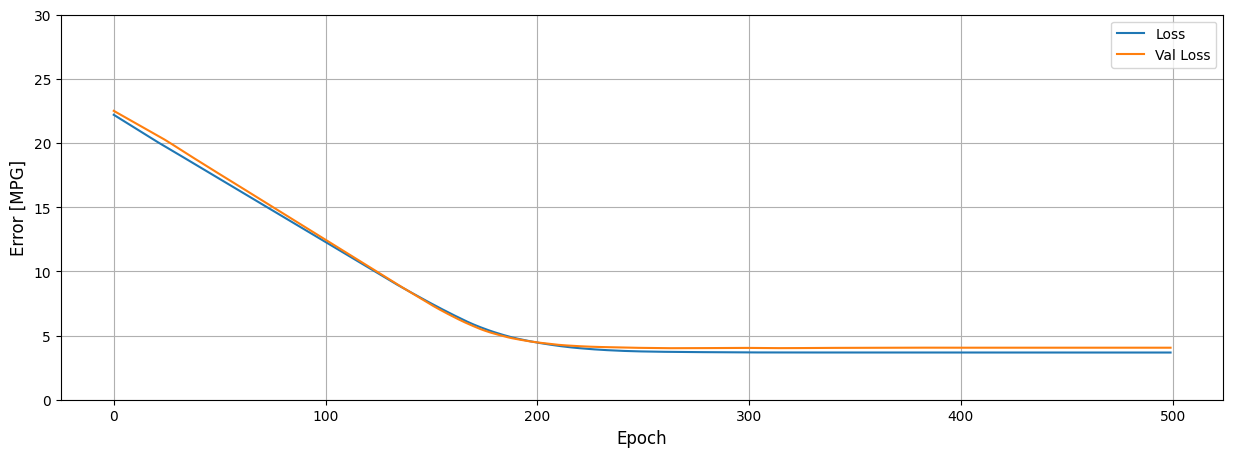
plt**.**show()

plt**.**close()

**return**

In [ ]:

plot\_loss(loss\_curve\_train\_linear\_1d, loss\_curve\_eval\_linear\_1d)



In [ ]:

hist **=** pd**.**DataFrame(

{

"loss": loss\_curve\_train\_linear\_1d,

"val\_loss": loss\_curve\_eval\_linear\_1d,

}

)

hist**.**describe()

Out[ ]:

|  | **loss** | **val\_loss** |
| --- | --- | --- |
| **count** | 500.000000 | 500.000000 |
| **mean** | 7.256331 | 7.519203 |
| **std** | 5.508316 | 5.528823 |
| **min** | 3.677526 | 4.015281 |
| **25%** | 3.677597 | 4.051866 |
| **50%** | 3.762090 | 4.057088 |
| **75%** | 9.841221 | 9.917352 |
| **max** | 22.202372 | 22.510668 |

Training and validation statistics are very close to each other which is a good sign because it tells us that the model can generalize to unseen data as well.

3.4 Model Prediction

Once the model has been trained, we can now use it to predict MPG given a range of values for Horsepower.

In [ ]:

*# # Generate feature data for Horsepower.*

x **=** torch**.**linspace(X\_train["Horsepower"]**.**min(), X\_train["Horsepower"]**.**max(), len(X\_train["Horsepower"]))

*# Use the model to predict MPG.*

linear\_1d\_model**.**eval()

**with** torch**.**no\_grad():

y **=** linear\_1d\_model((x**.**view(**-**1, 1) **-** mean\_hp) **/** std\_hp)

In [ ]:

**def** plot\_horsepower(x, y):

plt**.**figure(figsize**=**(15, 5))

plt**.**scatter(

(X\_train["Horsepower\_scaled"] **\*** std\_hp) **+** mean\_hp,

y\_train,

label**=**"Data",

color**=**"green",

alpha**=**0.5,

)

plt**.**plot(x, y, color**=**"k", label**=**"Predictions")

plt**.**xlabel("Horsepower")

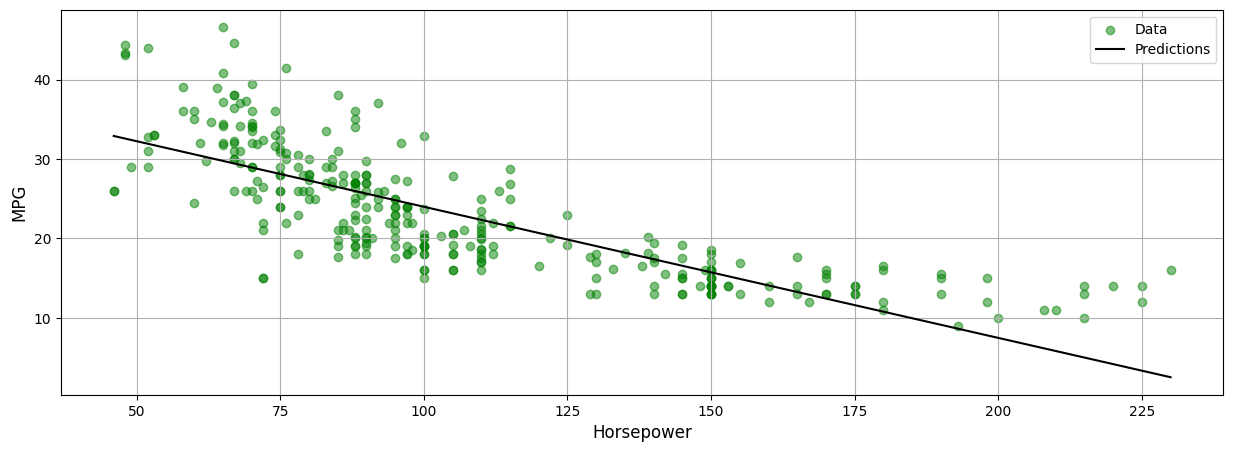
plt**.**ylabel("MPG")

plt**.**grid(**True**)

plt**.**legend()

In [ ]:

plot\_horsepower(x**.**view(**-**1, 1)**.**numpy(), y**.**numpy())



4 Multivariate Linear Regression

In this section, we will extend the previous example to now use two input features. All of the processing steps will remain the same. The only differences are those related to the input shape. With two input features, our model for the response variable is now a plane rather than a line. Using the general form that we introduced in Module 1, the hypothesis (function) takes the following form with two input features:

hθ(x)=θ0+θ1x1+θ2x2hθ(x)=θ0+θ1x1+θ2x2

When working with neural networks, it is more common to use the notation below:

y′=w1x1+w2x2+by′=w1x1+w2x2+b

Since we have two inputs, we have two weights (one for each input) and then a single bias term associated with the single neuron. The equations above represent the equation of a plane in 3D space.

4.1 Intialize Model, Optimizer & Loss function

In [ ]:

linear\_2d\_model **=** Regressor\_1(in\_features**=**2, out\_features**=**1)

batch\_size **=** 1

summary(linear\_2d\_model, input\_size**=**(batch\_size, 2,), device**=**"cpu", col\_names**=**("input\_size", "output\_size", "num\_params"))

Out[ ]:

===================================================================================================================

Layer (type:depth-idx) Input Shape Output Shape Param #

===================================================================================================================

Regressor\_1 [1, 2] [1, 1] --

├─Linear: 1-1 [1, 2] [1, 1] 3

===================================================================================================================

Total params: 3

Trainable params: 3

Non-trainable params: 0

Total mult-adds (M): 0.00

===================================================================================================================

Input size (MB): 0.00

Forward/backward pass size (MB): 0.00

Params size (MB): 0.00

Estimated Total Size (MB): 0.00

===================================================================================================================

In [ ]:

*# Initialize Optimizer and loss function.*

optimizer **=** optim**.**Adam(linear\_2d\_model**.**parameters(), lr**=**0.1)

criterion **=** torch**.**nn**.**L1Loss()

4.2 Train the Model

In this section, we will first define two small helper functions train\_one\_epoch(...) and evaluate(...) to perform training and evaluate the model during each training loop. By creating these function, it's help reduce redundant code and help avoid making mistakes that can happen during code updation.

We also define one more function generate\_predictions(...) that ensures the model runs in evaluation mode.

The evaluate(..) and generate\_predictions(...) make use of [torch.no\_grad()](https://pytorch.org/docs/stable/generated/torch.no_grad.html) and [torch.inference\_mode()](https://pytorch.org/docs/stable/generated/torch.inference_mode.html) to ensure no gradients are tracked or calculated during inference.

In [ ]:

**def** train\_one\_epoch(model: torch**.**nn**.**Module, optimizer: torch**.**optim**.**Optimizer, loss\_fn: torch**.**nn**.**Module, dataset: tuple):

data, target **=** dataset

model**.**train() *# Set model in training mode.*

outputs **=** model(data) *# Perform forward pass through the model.*

loss **=** loss\_fn(outputs, target) *# Calculate L1 loss on the model predictions.*

optimizer**.**zero\_grad() *# Reset gradients.*

loss**.**backward() *# Calcualte gradients based on the loss.*

optimizer**.**step() *# Update parameters.*

**return** loss**.**detach()**.**item()

**def** evaluate(model: torch**.**nn**.**Module, loss\_fn: torch**.**nn**.**Module, dataset: tuple):

data, target **=** dataset

model**.**eval() *# Set model in evaluation mode.*

**with** torch**.**no\_grad():

outputs **=** model(data) *# Perform forward pass through the validation set.*

loss **=** loss\_fn(outputs, target) *# Calculate the loss on the validation set.*

**return** loss**.**item()

@torch**.**inference\_mode()

**def** generate\_predictions(model, data):

model**.**eval()

outputs **=** model(data)

**return** outputs

In [ ]:

*# Initialize the 2-feature dataset. The targets remain the same.*

X\_train\_hp\_dp **=** torch**.**from\_numpy(X\_train\_split[['Horsepower\_scaled','Displacement\_scaled']]**.**values)**.**reshape(**-**1, 2)**.**to(torch**.**float32)

X\_val\_hp\_dp **=** torch**.**from\_numpy(X\_val\_split[['Horsepower\_scaled', 'Displacement\_scaled']]**.**values)**.**reshape(**-**1, 2)**.**to(torch**.**float32)

We also use the tqdm package to display a progress bar indicating the number of epochs completed along with the time required to finish.

In [ ]:

**from** tqdm **import** trange

*# To record the traning and validation loss metrics.*

loss\_curve\_train\_linear\_2d **=** []

loss\_curve\_eval\_linear\_2d **=** []

**for** epoch **in** trange(500):

*# Perform one epoch of training and then evaluate on the validation set.*

train\_loss **=** train\_one\_epoch(linear\_2d\_model, optimizer, criterion, (X\_train\_hp\_dp, y\_train\_t))

val\_loss **=** evaluate(linear\_2d\_model, criterion, (X\_val\_hp\_dp, y\_val\_t))

*# Record training and validation loss*

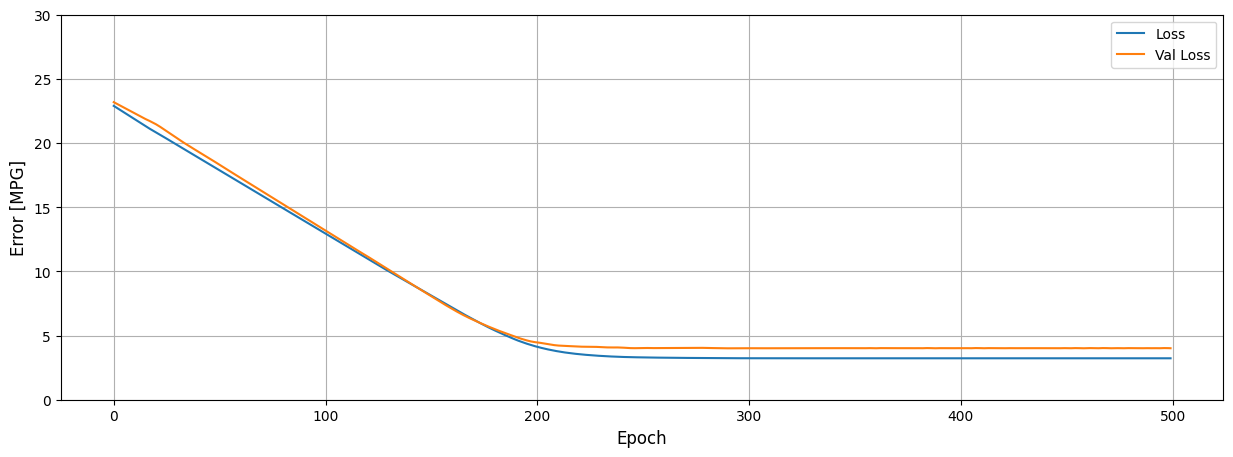
loss\_curve\_train\_linear\_2d**.**append(train\_loss)

loss\_curve\_eval\_linear\_2d**.**append(val\_loss)

100%|██████████| 500/500 [00:01<00:00, 397.93it/s]

In [ ]:

plot\_loss(loss\_curve\_train\_linear\_2d, loss\_curve\_eval\_linear\_2d)



4.3 Visualize the Trained Model

In [ ]:

hp\_min **=** (X\_train**.**Horsepower\_scaled**.**min() **\*** std\_hp) **+** mean\_hp

hp\_max **=** (X\_train**.**Horsepower\_scaled**.**max() **\*** std\_hp) **+** mean\_hp

dp\_min **=** (X\_train**.**Displacement\_scaled**.**min() **\*** std\_dis) **+** mean\_dis

dp\_max **=** (X\_train**.**Displacement\_scaled**.**max() **\*** std\_dis) **+** mean\_dis

x\_surf, y\_surf **=** np**.**meshgrid(np**.**linspace(hp\_min, hp\_max, 100), np**.**linspace(dp\_min, dp\_max, 100))

x\_grid **=** pd**.**DataFrame(

{

'Horsepower': (x\_surf**.**ravel() **-** mean\_hp) **/** std\_hp,

'Displacement': (y\_surf**.**ravel() **-**mean\_dis) **/** std\_dis,

}

)

pred\_y **=** generate\_predictions(linear\_2d\_model, data**=**torch**.**from\_numpy(x\_grid**.**values)**.**view(**-**1,2)**.**float())**.**numpy()

In [ ]:

fig **=** plt**.**figure(figsize**=**(20, 10))

ax **=** fig**.**add\_subplot(121, projection**=**'3d')

ax**.**scatter((X\_train['Horsepower\_scaled'] **\*** std\_hp) **+** mean\_hp, (X\_train['Displacement\_scaled'] **\*** std\_dis) **+** mean\_dis, y\_train, c**=**'green', marker**=**'o', alpha**=**0.5)

ax**.**plot\_surface(x\_surf, y\_surf, pred\_y**.**reshape(x\_surf**.**shape), color**=**'blue', alpha**=**0.5)

ax**.**set\_xlabel('Horsepower')

ax**.**set\_ylabel('Displacement')

ax**.**set\_zlabel('MPG')

ax**.**view\_init(9, **-**40)

ax **=** fig**.**add\_subplot(122, projection**=**'3d')

ax**.**scatter((X\_train['Horsepower\_scaled'] **\*** std\_hp) **+** mean\_hp, (X\_train['Displacement\_scaled'] **\*** std\_dis) **+** mean\_dis, y\_train, c**=**'green', marker**=**'o', alpha**=**0.5)

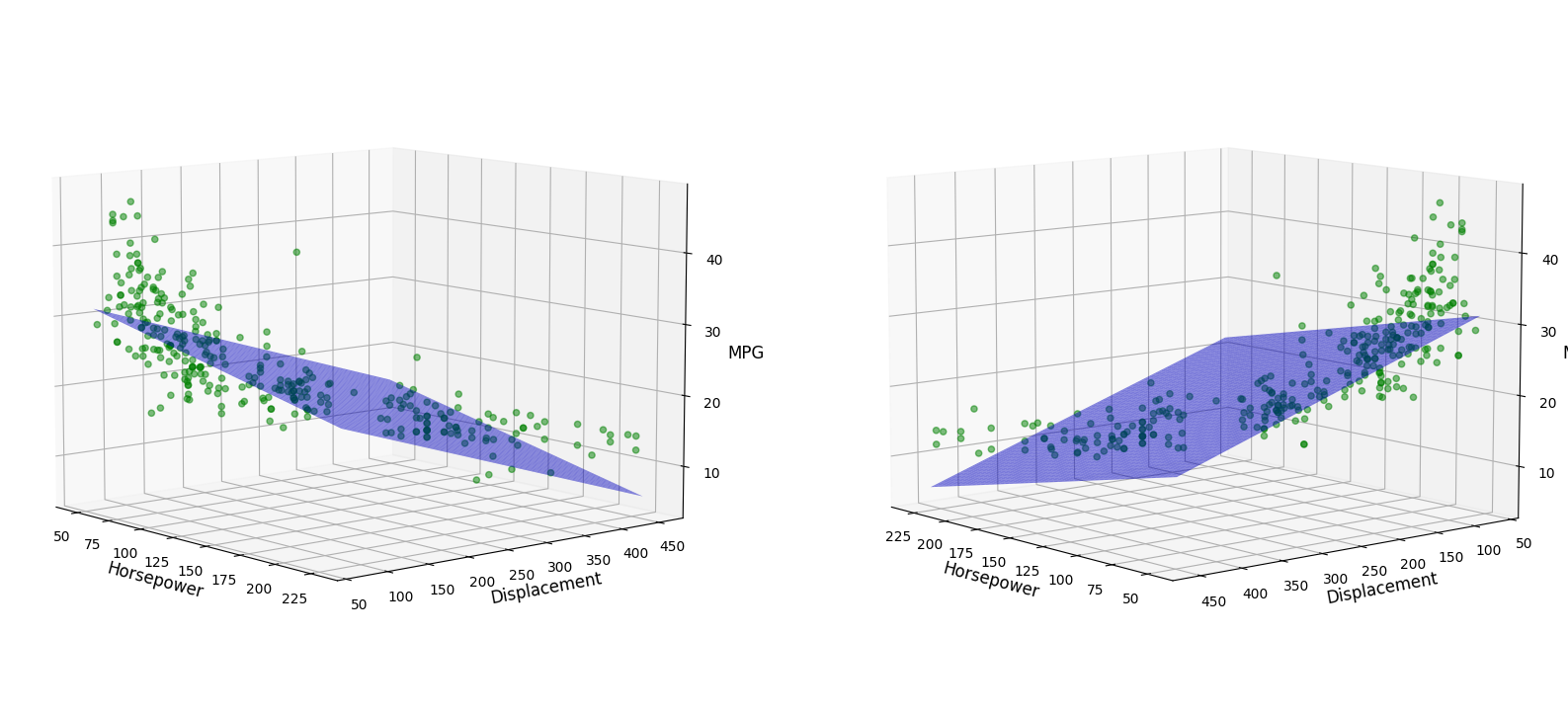
ax**.**plot\_surface(x\_surf, y\_surf, pred\_y**.**reshape(x\_surf**.**shape), color**=**'blue', alpha**=**0.5)

ax**.**set\_xlabel('Horsepower')

ax**.**set\_ylabel('Displacement')

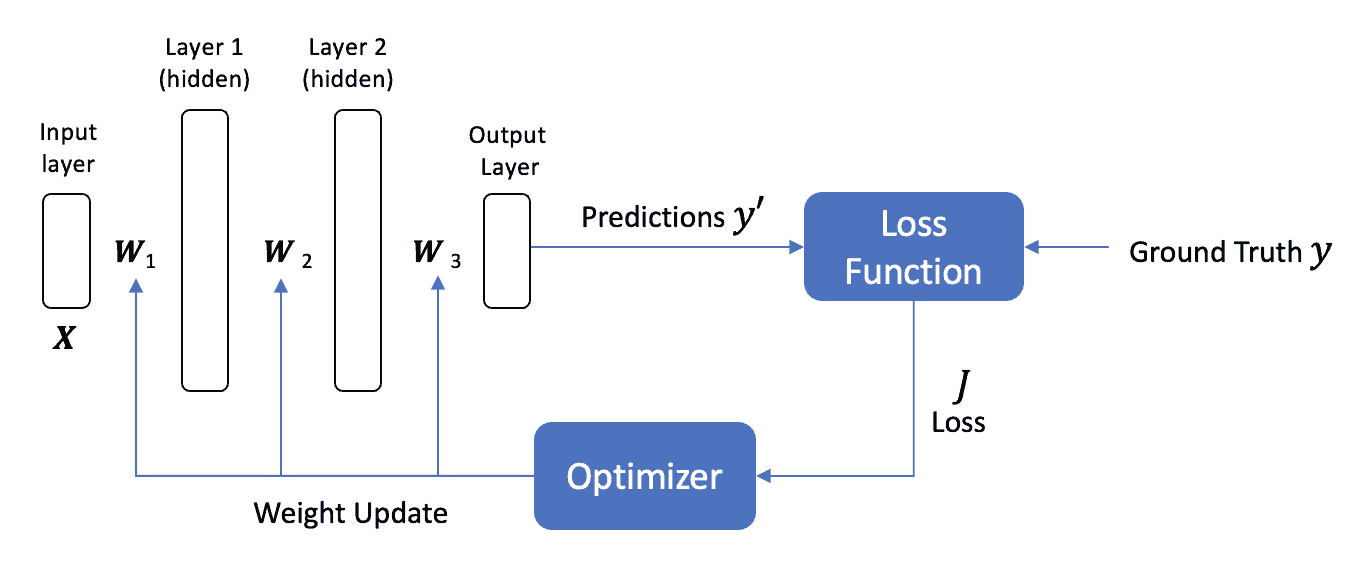
ax**.**set\_zlabel('MPG')

ax**.**view\_init(9, 140)



5 Deep Learning with a Single Feature

We are now ready to model hidden layers in our network. Adding hidden layers with non-linear activation functions is what makes them "Deep" and capable of learning general non-linear functional mappings between inputs and outputs. The simplified network diagram below shows the general architecture, which consists of an input layer, two hidden layers, and the output layer. In this section, we are going to model a network like this using single input feature. We will use a non-linear (relu) activation function in the hidden layers that will allow the network to model non-linear functions.



5.1 Build the PyTorch Model

Here we are going to build a simple deep neural network with two hidden layers (each with 32 neurons). Each of the neurons in the hidden layers will use relu activation functions. The choice of two hidden layers and 32 neurons for each layer is rather arbitrary, but we will need at least one hidden layer to model a non-linear response.

In [ ]:

**class** Regressor\_2(nn**.**Module):

*# Initialize the parameter*

**def** \_\_init\_\_(self, in\_features**=**2, out\_features**=**1, intermediate**=**10):

super()**.**\_\_init\_\_()

self**.**linear\_1 **=** nn**.**Linear(in\_features**=**in\_features, out\_features**=**intermediate)

self**.**linear\_2 **=** nn**.**Linear(in\_features**=**intermediate, out\_features**=**intermediate)

self**.**linear\_3 **=** nn**.**Linear(in\_features**=**intermediate, out\_features**=**out\_features)

*# Forward pass*

**def** forward(self, x):

*# First Linear layer --=> ReLU activation*

pred **=** F**.**relu(self**.**linear\_1(x))

*# Second Linear layer --=> ReLU activation*

pred **=** F**.**relu(self**.**linear\_2(pred))

*# Third Linear layer.*

pred **=** self**.**linear\_3(pred)

**return** pred

Display the initial values for the weights and biases in the first hidden layer.

In [ ]:

dnn\_1d\_model **=** Regressor\_2(in\_features**=**1, out\_features**=**1, intermediate**=**32)

batch\_size **=** 1

summary(dnn\_1d\_model, input\_size**=**(batch\_size, 1,), device**=**"cpu", col\_names**=**("input\_size", "output\_size", "num\_params"))

Out[ ]:

===================================================================================================================

Layer (type:depth-idx) Input Shape Output Shape Param #

===================================================================================================================

Regressor\_2 [1, 1] [1, 1] --

├─Linear: 1-1 [1, 1] [1, 32] 64

├─Linear: 1-2 [1, 32] [1, 32] 1,056

├─Linear: 1-3 [1, 32] [1, 1] 33

===================================================================================================================

Total params: 1,153

Trainable params: 1,153

Non-trainable params: 0

Total mult-adds (M): 0.00

===================================================================================================================

Input size (MB): 0.00

Forward/backward pass size (MB): 0.00

Params size (MB): 0.00

Estimated Total Size (MB): 0.01

===================================================================================================================

In [ ]:

*# Initialize Optimizer and loss function.*

optimizer **=** optim**.**Adam(dnn\_1d\_model**.**parameters(), lr**=**0.01)

criterion **=** torch**.**nn**.**L1Loss()

5.2 Train the Model

In [ ]:

*# To record the traning and validation loss metrics.*

loss\_curve\_train\_dnn\_1d **=** []

loss\_curve\_eval\_dnn\_1d **=** []

**for** epoch **in** trange(500):

*# Perform one epoch of training and then evaluate on the validation set.*

train\_loss **=** train\_one\_epoch(dnn\_1d\_model, optimizer, criterion, (X\_train\_hp, y\_train\_t))

val\_loss **=** evaluate(dnn\_1d\_model, criterion, (X\_val\_hp, y\_val\_t))

*# Record training and validation loss*

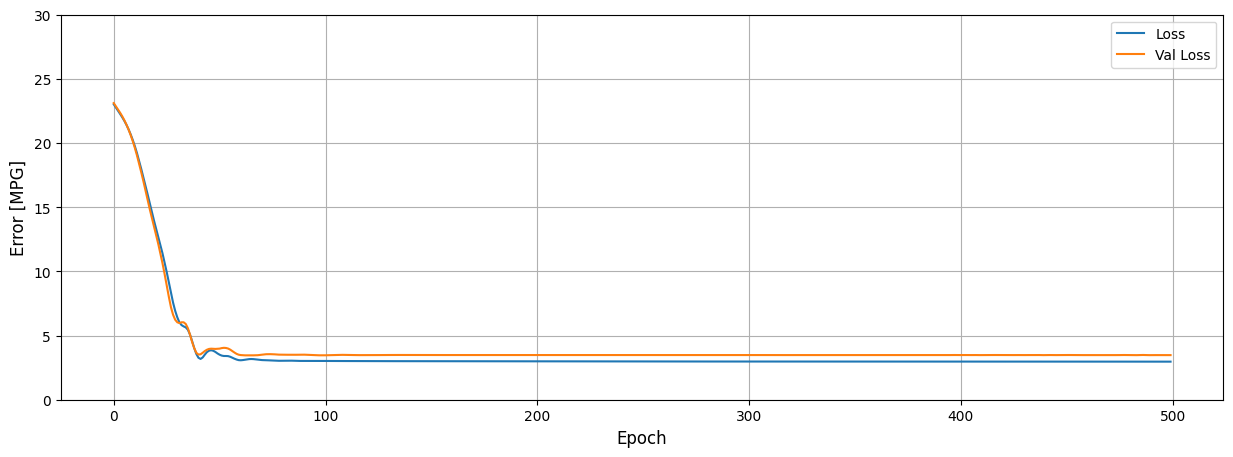
loss\_curve\_train\_dnn\_1d**.**append(train\_loss)

loss\_curve\_eval\_dnn\_1d**.**append(val\_loss)

100%|██████████| 500/500 [00:01<00:00, 347.99it/s]

In [ ]:

plot\_loss(loss\_curve\_train\_dnn\_1d, loss\_curve\_eval\_dnn\_1d)



5.4 Visualize the Trained Model

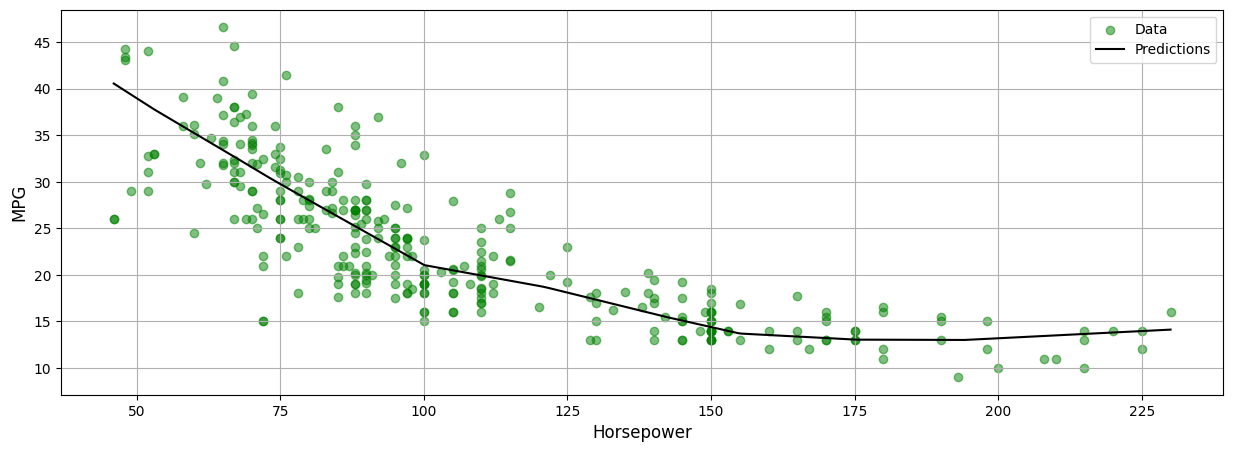
As you can see in the plot below, we now have a non-linear response from the neural network. In the next section we are going to add a second input feature so we an visualize the non-linear response in more than two dimensions.

In [ ]:

x **=** torch**.**linspace(X\_train["Horsepower"]**.**min(), X\_train["Horsepower"]**.**max(), len(X\_train["Horsepower"]))

y **=** generate\_predictions(dnn\_1d\_model, data**=**((x**.**view(**-**1, 1) **-** mean\_hp) **/** std\_hp))**.**numpy()

plot\_horsepower(x, y)



6 Deep Learning with Multiple Features

We will now introduce a second feature as we did in the Linear Regression section. This time we'll be using two input features with a deep neural network so that we can demonstrate the non-linear behavior of a neural work in more than two dimensions. When we used two input variables with just a single neuron (linear regression) the response function was a 2D plane in 3D space. This time we expect a non-linear surface in 3D space that better fits the data. Let's continue on and see!

6.1 Build the PyTorch Model

The model below is the same as the one in the previous section with the exception that we are using two input features instead of one.

In [ ]:

dnn\_2d\_model **=** Regressor\_2(in\_features**=**2, out\_features**=**1, intermediate**=**32)

batch\_size **=** 1

summary(dnn\_2d\_model, input\_size**=**(batch\_size, 2,), device**=**"cpu", col\_names**=**("input\_size", "output\_size", "num\_params"))

Out[ ]:

===================================================================================================================

Layer (type:depth-idx) Input Shape Output Shape Param #

===================================================================================================================

Regressor\_2 [1, 2] [1, 1] --

├─Linear: 1-1 [1, 2] [1, 32] 96

├─Linear: 1-2 [1, 32] [1, 32] 1,056

├─Linear: 1-3 [1, 32] [1, 1] 33

===================================================================================================================

Total params: 1,185

Trainable params: 1,185

Non-trainable params: 0

Total mult-adds (M): 0.00

===================================================================================================================

Input size (MB): 0.00

Forward/backward pass size (MB): 0.00

Params size (MB): 0.00

Estimated Total Size (MB): 0.01

===================================================================================================================

In [ ]:

*# Initialize Optimizer and loss function.*

optimizer **=** optim**.**Adam(dnn\_2d\_model**.**parameters(), lr**=**0.01)

criterion **=** torch**.**nn**.**L1Loss()

In [ ]:

**from** tqdm **import** trange

*# To record the traning and validation loss metrics.*

loss\_curve\_train\_dnn\_2d **=** []

loss\_curve\_eval\_dnn\_2d **=** []

**for** epoch **in** trange(500):

*# Perform one epoch of training and then evaluate on the validation set.*

train\_loss **=** train\_one\_epoch(dnn\_2d\_model, optimizer, criterion, (X\_train\_hp\_dp, y\_train\_t))

val\_loss **=** evaluate(dnn\_2d\_model, criterion, (X\_val\_hp\_dp, y\_val\_t))

*# Record training and validation loss*

loss\_curve\_train\_dnn\_2d**.**append(train\_loss)

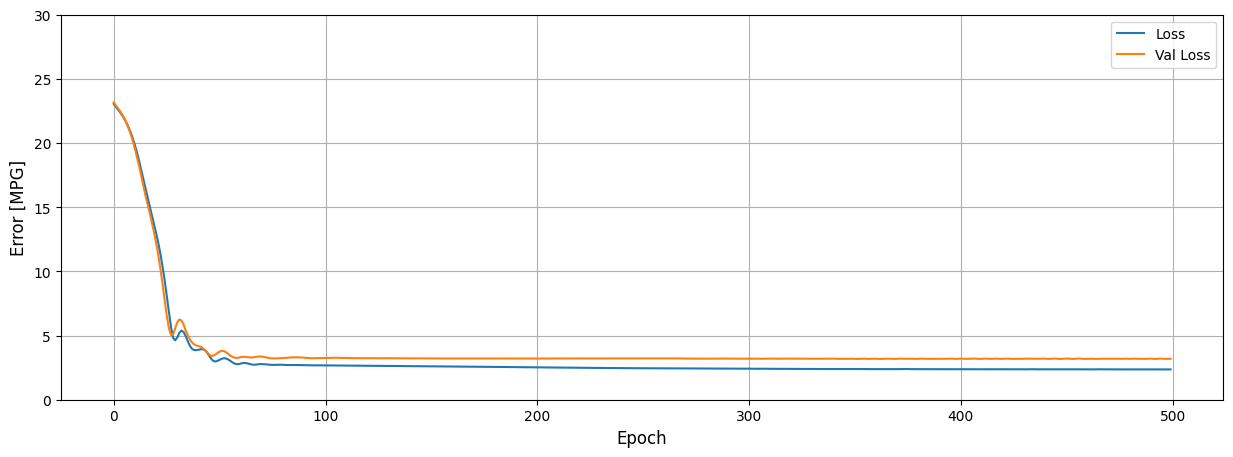
loss\_curve\_eval\_dnn\_2d**.**append(val\_loss)

100%|██████████| 500/500 [00:00<00:00, 535.25it/s]

6.3 Train the Model

In [ ]:

plot\_loss(loss\_curve\_train\_dnn\_2d, loss\_curve\_eval\_dnn\_2d)



6.4 Visualize the Trained Model

As shown in the plots below the response function from the network is a non-linear surface which does a better job of fitting the data.

In [ ]:

hp\_min **=** (X\_train**.**Horsepower\_scaled**.**min() **\*** std\_hp) **+** mean\_hp

hp\_max **=** (X\_train**.**Horsepower\_scaled**.**max() **\*** std\_hp) **+** mean\_hp

dp\_min **=** (X\_train**.**Displacement\_scaled**.**min() **\*** std\_dis) **+** mean\_dis

dp\_max **=** (X\_train**.**Displacement\_scaled**.**max() **\*** std\_dis) **+** mean\_dis

x\_surf, y\_surf **=** np**.**meshgrid(np**.**linspace(hp\_min, hp\_max, 100), np**.**linspace(dp\_min, dp\_max, 100))

x\_grid **=** pd**.**DataFrame(

{

'Horsepower': (x\_surf**.**ravel() **-** mean\_hp) **/** std\_hp,

'Displacement': (y\_surf**.**ravel() **-** mean\_dis) **/** std\_dis,

}

)

pred\_y **=** generate\_predictions(dnn\_2d\_model, data**=**torch**.**from\_numpy(x\_grid**.**values)**.**view(**-**1,2)**.**float())**.**numpy()

In [ ]:

fig **=** plt**.**figure(figsize**=**(20, 10))

ax **=** fig**.**add\_subplot(121, projection**=**'3d')

ax**.**scatter(

(X\_train['Horsepower\_scaled'] **\*** std\_hp) **+** mean\_hp, (X\_train['Displacement\_scaled'] **\*** std\_dis) **+** mean\_dis,

y\_train,

c**=**'green',

marker**=**'o',

alpha**=**0.5

)

ax**.**plot\_surface(x\_surf, y\_surf, pred\_y**.**reshape(x\_surf**.**shape), color**=**'blue', alpha**=**0.5)

ax**.**set\_xlabel('Horsepower')

ax**.**set\_ylabel('Displacement')

ax**.**set\_zlabel('MPG')

ax**.**view\_init(8, **-**40)

ax **=** fig**.**add\_subplot(122, projection**=**'3d')

ax**.**scatter(

(X\_train['Horsepower\_scaled'] **\*** std\_hp) **+** mean\_hp, (X\_train['Displacement\_scaled'] **\*** std\_dis) **+** mean\_dis,

y\_train,

c**=**'green',

marker**=**'o',

alpha**=**0.5

)

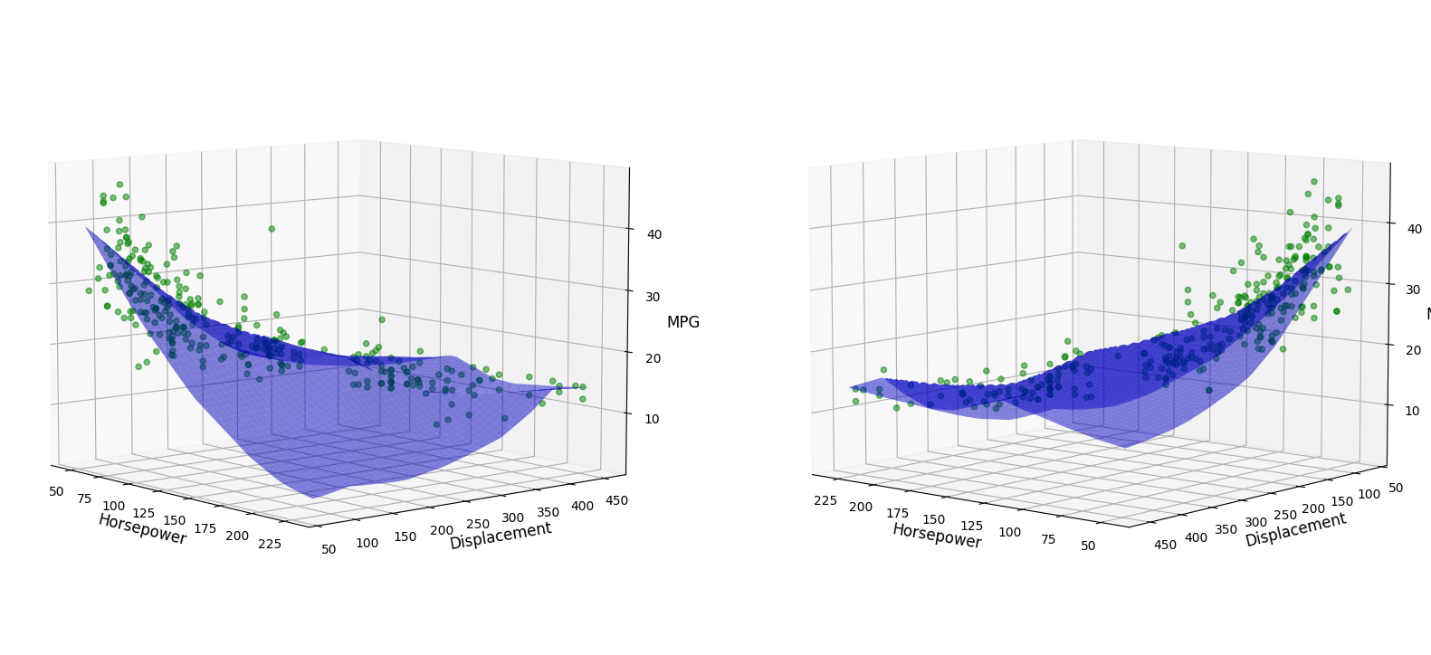
ax**.**plot\_surface(x\_surf, y\_surf, pred\_y**.**reshape(x\_surf**.**shape), color**=**'blue', alpha**=**0.5)

ax**.**set\_xlabel('Horsepower')

ax**.**set\_ylabel('Displacement')

ax**.**set\_zlabel('MPG')

ax**.**view\_init(8, 130)



7 Model Comparison

Let's now compare each of the four models that we developed. We can quickly evaluate the models on the test data using model.evaluate(). This function will return the mean absolute error associated with the test data for each model. As expected, the test error for each model is progressively lower as we introduce additional features and non-linearity.

7.1 Compare Training Loss Curves

In [ ]:

plt**.**figure(figsize**=**(15, 10))

plt**.**subplot(2, 1, 1)

plt**.**plot(loss\_curve\_train\_linear\_1d, label**=**"linear\_1d")

plt**.**plot(loss\_curve\_train\_linear\_2d, label**=**"linear\_2d")

plt**.**plot(loss\_curve\_train\_dnn\_1d, label**=**"dnn\_1d")

plt**.**plot(loss\_curve\_train\_dnn\_2d, label**=**"dnn\_2d")

plt**.**ylim([0, 30])

plt**.**ylabel("Error [MPG]")

plt**.**title("Loss per Epoch")

plt**.**legend()

plt**.**grid(**True**)

plt**.**figure(figsize**=**(15, 10))

plt**.**subplot(2, 1, 2)

plt**.**plot(loss\_curve\_eval\_linear\_1d, label**=**"linear\_1d")

plt**.**plot(loss\_curve\_eval\_linear\_2d, label**=**"linear\_2d")

plt**.**plot(loss\_curve\_eval\_dnn\_1d, label**=**"dnn\_1d")

plt**.**plot(loss\_curve\_eval\_dnn\_2d, label**=**"dnn\_2d")

plt**.**ylim([0, 30])

plt**.**xlabel("Epoch")

plt**.**ylabel("Error [MPG]")

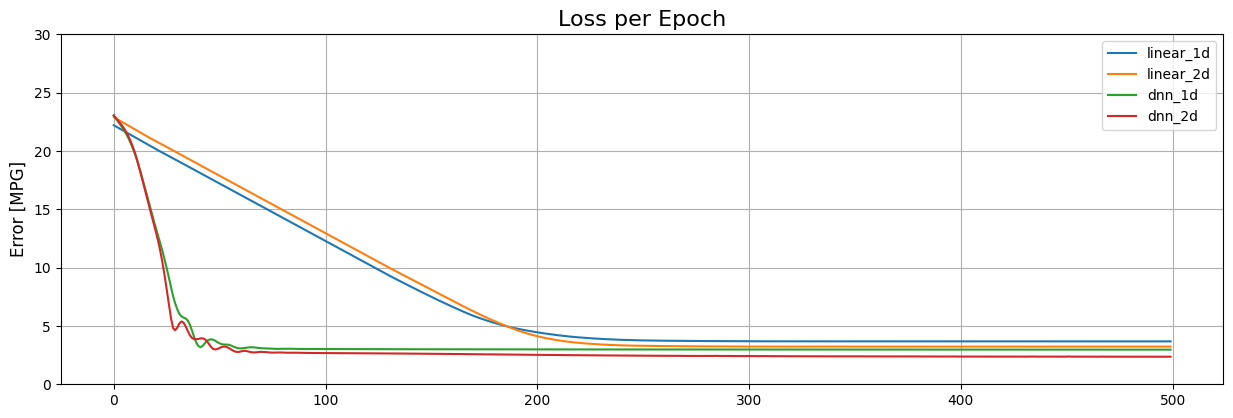
plt**.**title("Val loss per Epoch")

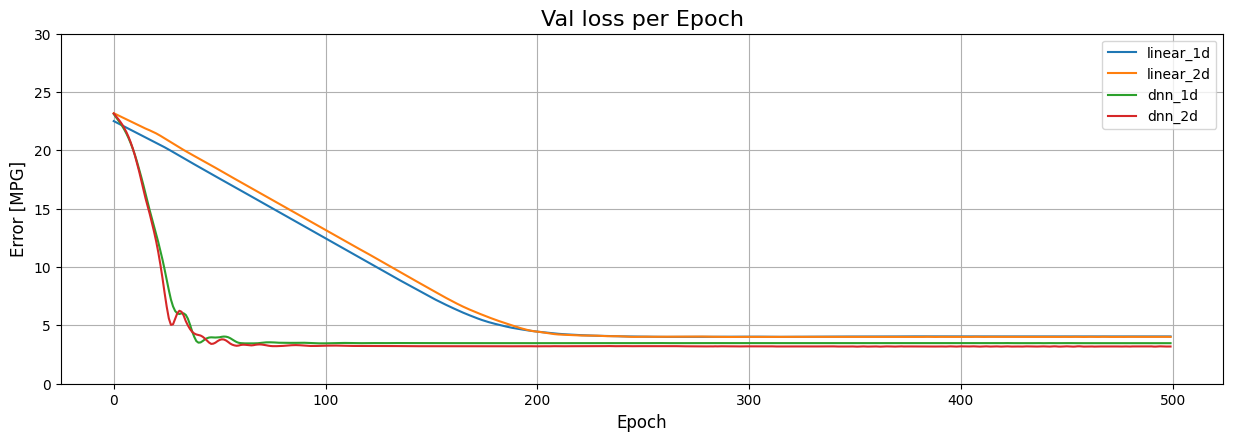
plt**.**legend()

plt**.**grid(**True**)

plt**.**show()

plt**.**close()





7.2 Model Evaluation Comparison

In [ ]:

*# Prepare test set.*

X\_test\_hp **=** torch**.**from\_numpy(X\_test["Horsepower\_scaled"]**.**values)**.**reshape(**-**1, 1)**.**to(torch**.**float32)

X\_test\_hp\_dp **=** torch**.**from\_numpy(X\_test[["Horsepower\_scaled", "Displacement\_scaled"]]**.**values)**.**reshape(**-**1, 2)**.**to(torch**.**float32)

y\_test **=** torch**.**from\_numpy(y\_test**.**values)**.**reshape(**-**1, 1)**.**to(torch**.**float32)

In [ ]:

*# Create a dictionary to store test results.*

test\_results **=** {}

*# Linear regression test results.*

**with** torch**.**no\_grad():

test\_results['linear\_1d\_model'] **=** criterion(linear\_1d\_model(X\_test\_hp), y\_test)**.**numpy()

test\_results['linear\_2d\_model'] **=** criterion(linear\_2d\_model(X\_test\_hp\_dp), y\_test)**.**numpy()

*# Deep learning regression test results.*

test\_results['dnn\_1d\_model'] **=** criterion(dnn\_1d\_model(X\_test\_hp), y\_test)**.**numpy()

test\_results['dnn\_2d\_model'] **=** criterion(dnn\_2d\_model(X\_test\_hp\_dp), y\_test)**.**numpy()

In [ ]:

pd**.**DataFrame(test\_results, index**=**['Mean Absolute Error [MPG]'])**.**T

Out[ ]:

|  | **Mean Absolute Error [MPG]** |
| --- | --- |
| **linear\_1d\_model** | 3.874628 |
| **linear\_2d\_model** | 3.369334 |
| **dnn\_1d\_model** | 3.398708 |
| **dnn\_2d\_model** | 2.759579 |

7.3 Visualizing Model Predictions

Let's now take a look at how well the final model (dnn\_2d\_model) predicts vehicle MPG based on two input features.

In [ ]:

**with** torch**.**no\_grad():

train\_predictions **=** dnn\_2d\_model(X\_train\_hp\_dp)**.**squeeze()

test\_predictions **=** dnn\_2d\_model(X\_test\_hp\_dp)**.**squeeze()

plt**.**scatter(y\_train\_t**.**numpy(), train\_predictions**.**numpy(), c**=**'lightgray')

plt**.**scatter(y\_test**.**numpy(), test\_predictions**.**numpy(), c**=**'green', alpha**=**.5)

plt**.**title('Comparison of Test Predictions (green) to Fitted Model and Training Points (grey)')

plt**.**xlabel('True Values [MPG]')

plt**.**ylabel('Predictions [MPG]')

plt**.**legend(['Train', 'Test'])

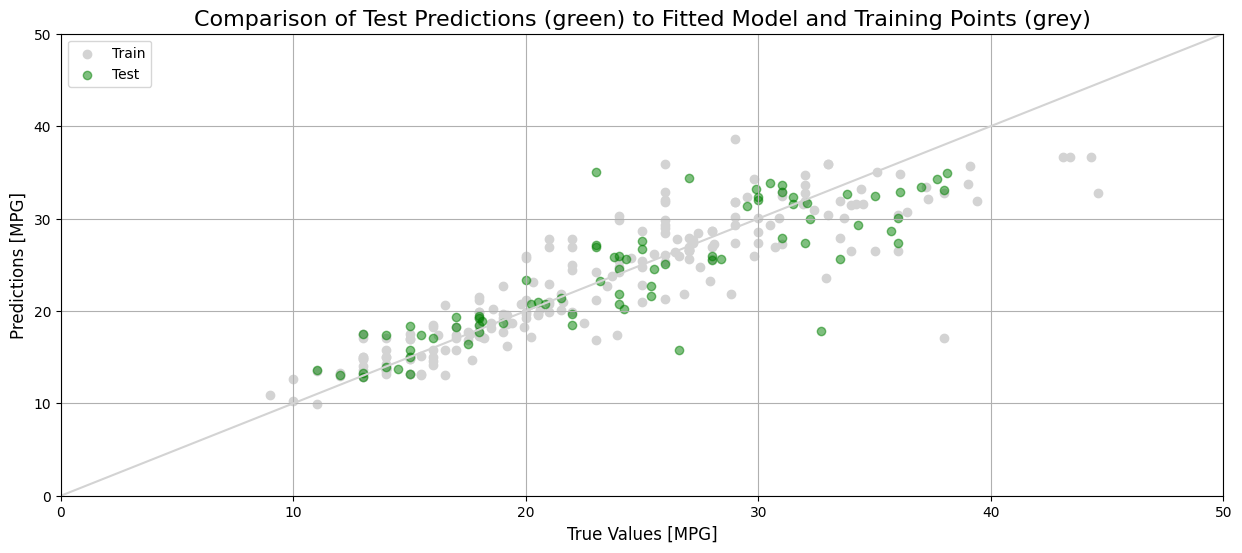
lims **=** [0, 50]

plt**.**xlim(lims)

plt**.**ylim(lims)

plt**.**grid('on')

plt**.**plot(lims, lims, c**=**'lightgray');



We can also compute the Mean Absolute Error (MAE) manually from the test data.

In [ ]:

test\_error **=** (test\_predictions **-** y\_test**.**squeeze())

mae **=** torch**.**mean(torch**.**abs(test\_error))

print('MAE for dnn\_2d\_model: ', int(mae **\*** 10000) **/** 10000)

MAE for dnn\_2d\_model: 2.7595

7.4 Test Error Distribution

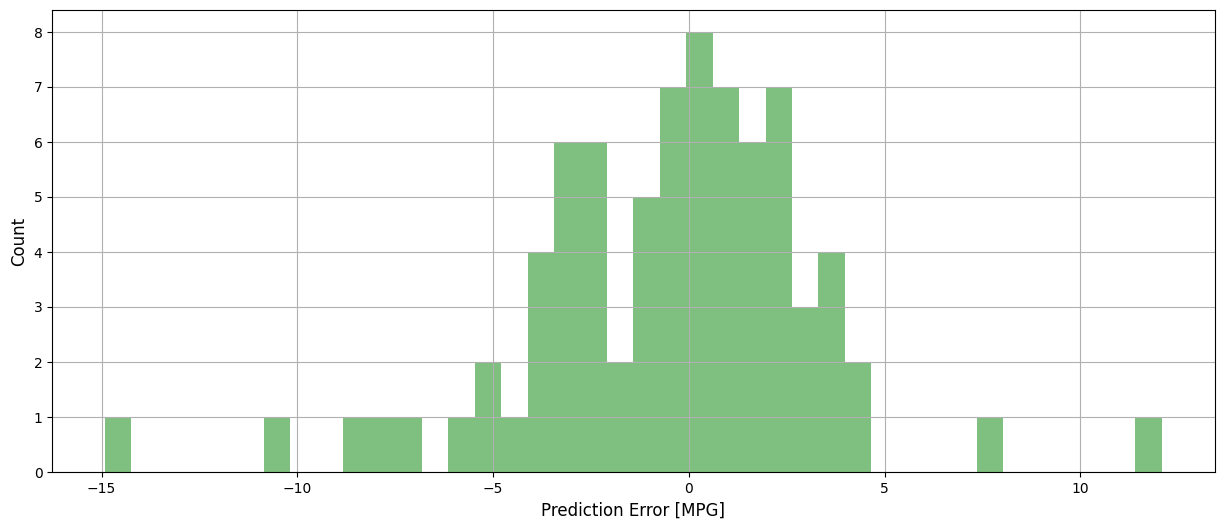
In [ ]:

plt**.**hist(test\_error, bins**=**40, color**=**'green', alpha**=**.5)

plt**.**xlabel('Prediction Error [MPG]')

plt**.**grid('on')

plt**.**ylabel('Count');



8 Conclusion

In this first part of this notebook, we learned how to model linear regression using a neuron (with a linear activation function), which can be thought of as a simple linear neural network with a single layer (the output layer). We showed that the response from the network is a linear function. We looked at two examples with a single input feature and two input features and showed that these correspond to response functions of a straight line and 2D plane, respectively. In higher dimensions (when the number of features exceeds 2), the response from the linear regression model becomes a hyperplane that cannot be visualized.

Next, we explored the use of hidden layers and non-linear activation functions in a neural network. We showed that this type of architecture allows the network to learn non-linear mappings between the inputs and the output. We showed examples for both a single input feature and two input features to visualize the non-linear response functions. This can easily be extended to using all the features in the dataset. Although it would not be possible to visualize the response function in higher dimensions, you should expect slightly better predictions when using more features. This would be a good exercise to experiment with on your own.

Binary Classification using PyTorch

In this notebook, we will introduce the topic of **Binary classification**. In machine learning, classification refers to the predictive modeling problem of identifying which of a set of categories an observation belongs to. The categories are referred to as classes. In the figure below, we are showing a hypothetical dataset that represents two classes. We have color coded the classes as red and blue, but this is only for visualization purposes. The classes themselves are characterized by two features (x1x1 and x2x2). Our task is to define a model that will predict the correct class based on the value of two input features. This is the definition of binary classification. The model that we develop will lead to a linear decision boundary, as shown in the figure. Later on, in this module we will cover the topic of multinomial classification, which involves three or more classes.

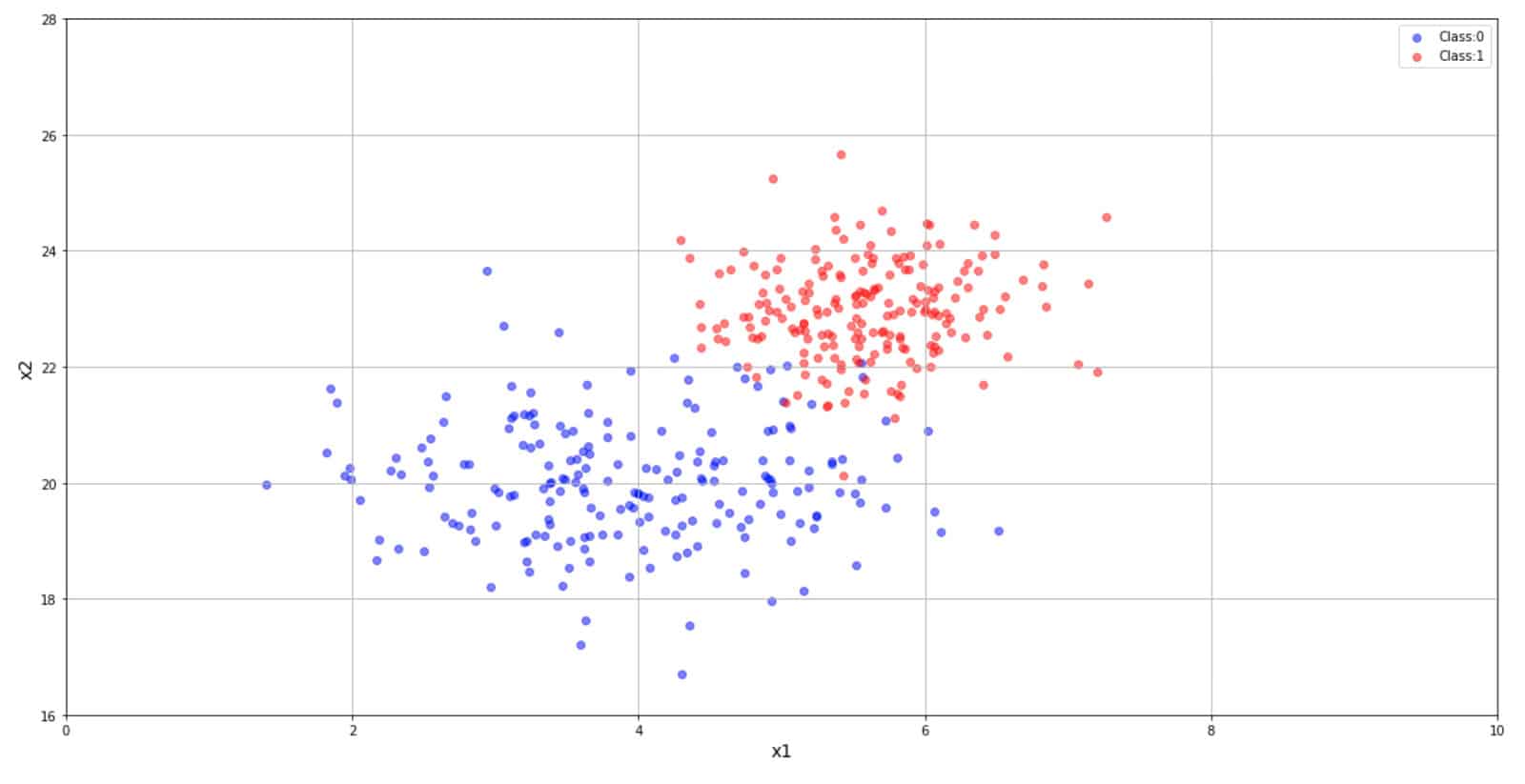


Table of Contents

* [1 Binary Classification](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/04_Binary_Classification_with_PyTorch_v4.html#1-Binary-Classification)
* [2 Mathematical Foundation](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/04_Binary_Classification_with_PyTorch_v4.html#2-Mathematical-Foundation)
* [3 Model Architecture](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/04_Binary_Classification_with_PyTorch_v4.html#3-Model-Architecture)
* [4 Data Generation](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/04_Binary_Classification_with_PyTorch_v4.html#4-Data-Generation)
* [5 Data Preparation](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/04_Binary_Classification_with_PyTorch_v4.html#5-Data-Preparation)
* [6 Perform Training](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/04_Binary_Classification_with_PyTorch_v4.html#6-Perform-Training)
* [7 Model Evaluation](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/04_Binary_Classification_with_PyTorch_v4.html#7-Model-Evaluation)
* [8 Conclusion](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/04_Binary_Classification_with_PyTorch_v4.html#8-Conclusion)

1 Binary Classification

Before we begin, let's set the stage with a simple binary classification problem to introduce the objective and the terminology. Suppose, we have some data that contains two features that are known to be good discriminators for whether a tumor is either benign or malignant. In this example, we have two classes (benign and malignant), and we have two features (say, x1𝑥1 and x2x2). For this discussion, it's not important what these features represent at this point, but let's assume they are real-valued quantities that can be measured. The two classes, by definition, are categorical, and we assign each class a numerical label such "benign" = 0 and "malignant" = 1.

Our task is to define a model that will predict the correct class based on the value of two input features. So, rather than predicting a real-valued number as we did with regression, our goal is to predict the correct class according to its class label. Note that we can use any number of features available. The reason this is called binary classification is that we only have two classes.

In the next section, we will develop the mathematical foundation for binary classification, which is known as Logistic Regression, and in the subsequent sections, we will demonstrate how to use PyTorch to model binary classification using a single perception with a sigmoid activation function. Much of the material covered in this notebook has been introduced in the two previous notebooks on regression, but there are a few key differences that allow us to transform the problem from regression (predicting a real-valued number) to the problem of classification (predicting a class label).

2 Mathematical Foundation

In Module 1, we introduced the mathematical model for **linear regression** in which we developed a linear model for the predicted output based on the weighted sum of the input features. The output in this case, is a real-valued number. To recap, we developed the following model:

hθ(x)=y′=θ0+θ1x1+θ2x2+...hθ(x)=y′=θ0+θ1x1+θ2x2+...

As before, the general notation can be simplified to the following by letting x0=1x0=1 which means that θ0θ0 becomes the bias term, and therefore the above expression can be simplified as follows:

h(x)=y′=n∑i=0θixi=θTxh(x)=y′=∑i=0nθixi=θTx

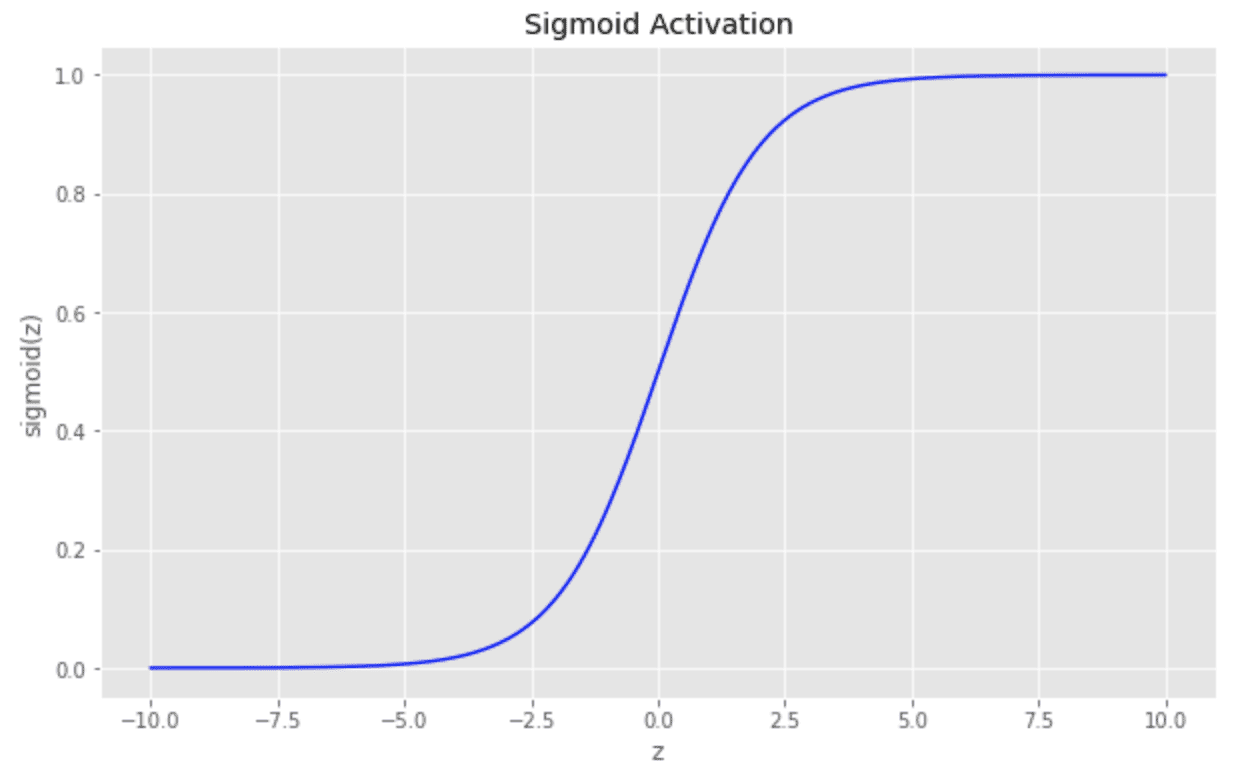
Where y′y′ is a real-valued predicted target variable in a regression problem. Let's now see how this can be extended to model a classification problem.

2.1 Classification and the Logistic Function

In a **classification** problem, the predicted output is a categorical label that specifies which class the input sample belongs to. But how do we accomplish this? This is where the **logistic function** comes to the rescue as defined below:

y′=σ(z)=11+e−zy′=σ(z)=11+e−z

This function is also commonly referred to as a **sigmoid** function because the shape resembles an 's' as shown below. Here, z=θTxz=θTx, and the output of this function is our new prediction y′y′ for classification. Notice that σ(z)σ(z) tends towards 1 as z→+∞z→+∞, and σ(z)σ(z) tends towards 0 as z→−∞z→−∞. And further, the function is bounded between 0 and 1.



Using this function, we can now transform our weighted inputs to produce a real-valued number bounded between 0 and 1 (where 0 represents one class and 1 represents the other class). In a binary classifcation problem, these two classes are sometimes referred to as the negative and positive classes, respectively.

2.2 Probabilistic Interpretation

Since this function is bounded between 0 and 1, we can now think of the output from this function as the probability that an input sample belongs to class 1(or the positive class).

P(y = 1|x;θ)=y′P(y = 1|x;θ)=y′

And therefore, the probability that the input sample belongs class 0 (or the negative class) is:

P(y = 0|x;θ)=1−y′P(y = 0|x;θ)=1−y′

These two expressions can be conveniently combined into a single equation as shown below, where yy represents the ground truth (or label) for the class.

p(y | x;θ)=(y′) y (1−y′)1−yp(y | x;θ)=(y′) y (1−y′)1−y

The above expression is read as: The probability of yy given xx, parameterized by θθ. For example, an input of z=0z=0, would correspond to y′=0.5y′=0.5 indicating that the input sample is equally likely to be from either class. Furthermore, an input value of z=2.5z=2.5, would correspond to a greater than 90% probability of the input sample belonging to class 1. So we can now see that since this function is monotonically increasing, we can use a single threshold to determine if the input sample corresponds to class 0 or to class 1.

2.3 Binary Cross Entropy Loss

The question now becomes, how do we solve for the parameters of the model (θθ)? This can be accomplished using the technique of maximum likelihood estimation (MLE) which is beyond the scope of this discussion, but we will briefly provide the intuition for how this is done by developing an appropriate loss functon for binary classification. Let's start with the idea that we would like to learn the weights that maximize the probabilty of the correct label given by:

P(y | x;θ)=(y′)y (1−y′)1−yP(y | x;θ)=(y′)y (1−y′)1−y

If we take the log of both sides of the above equation, we get a very convenient mathematical expression that will not affect the optimization (maximizing the probability will also maximize the log of the probability).

ln [p(y | x;θ) ]=y ln(y′)+(1−y) ln(1−y′)ln [p(y | x;θ) ]=y ln(y′)+(1−y) ln(1−y′)

In order to turn this into a loss function that we can minimize, we can take the negative log of the above probability that leads us to the **Binary Cross Entropy Loss Function** shown below:

J(y′)=−y ln(y′) − (1−y) ln(1−y′)J(y′)=−y ln(y′) − (1−y) ln(1−y′)

And recall that with z=θTxz=θTx, the predicted value for a given input sample is:

y′=σ(z)=11+e−zy′=σ(z)=11+e−z

And therefore, if

* σ(z)>0.5σ(z)>0.5 then input belongs to the positive class or class 1
* σ(z)<0.5σ(z)<0.5 then input belongs to the negative class or class 0

A few numerical examples are shown below that indicate the loss based on the true class yy and the predicted value y′y′. Notice that when the activation function output (y′y′) is close to the true label the loss is very small.

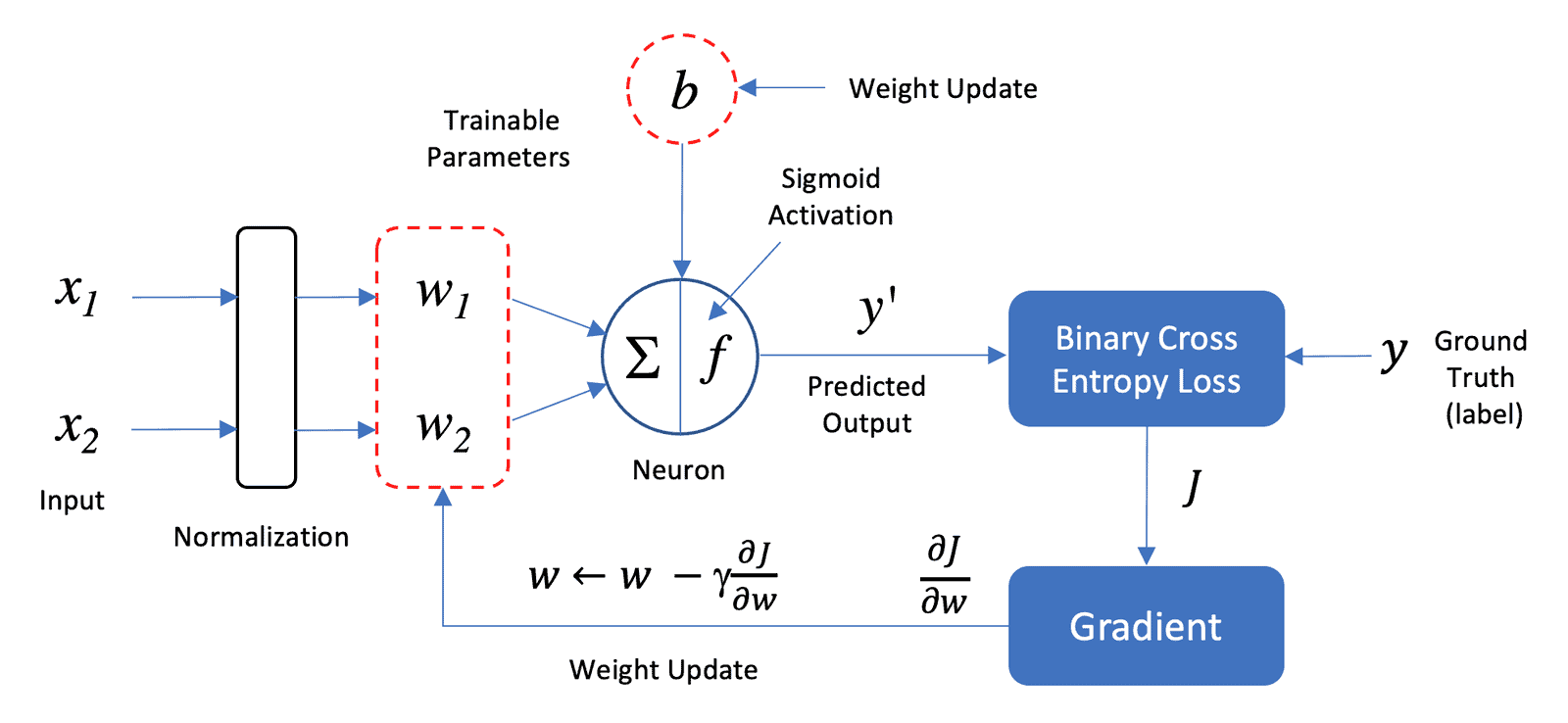
| **y** | **y'** | **Loss** | **Pred. Class** | **Notes** |
| --- | --- | --- | --- | --- |
| 1 | .90 | 0.046 | 1 | y' > 0.5, assigned to class 1 |
| 1 | .10 | 1.000 | 0 | y' < 0.5, assigned to class 0 |
| 0 | .01 | 0.004 | 0 | y' < 0.5, assigned to class 0 |
| 0 | .99 | 1.301 | 1 | y' > 0.5, assigned to class 1 |

**Note**: This binary classification model is often referred to as **Logistic Regression** because it is based on the *Logistic* function, but this has no relation to the regression problems we previously discussed. It is purely categorical and refers to classification, not regression. The terminology is unfortunate, but it is long-standing.

As we will see further below, we can specify Binary Cross Entropy as the loss function in PyTorch, and select an optimizer to train a binary classification model.

3 Model Architecture

The model architecture below for binary classification is similar to regression, but let's highlight the differences. First, notice that we use a sigmoid activation function and that the predicted output is a label rather than a real valued number. The label indcates which class is predicted. In this example, we are assuming binary classification, so we have two labels (say, 0 and 1). Note that binary classification can use any number of input features. In this notebook, we will use two input features, but this has nothing to do with the fact that we are performing binary classification. As we will see later in the course, binary classification can be performed using literally thousands of input features.



In [ ]:

**import** os

**import** random

**import** math

**import** time

**from** IPython.display **import** clear\_output

**import** torch

**import** torch.nn **as** nn

**import** torch.nn.functional **as** F

**import** torch.distributions **as** dist

**import** torch.optim **as** optim

**import** numpy **as** np

**import** seaborn **as** sn

**from** sklearn.metrics **import** confusion\_matrix

**import** matplotlib.pyplot **as** plt *# one of the best graphics library for python*

**import** matplotlib.animation **as** animation

plt**.**rcParams["figure.figsize"] **=** (15, 6)

plt**.**rcParams['axes.titlesize'] **=** 16

plt**.**rcParams['axes.labelsize'] **=** 14

block\_plot **=** **False**

In [ ]:

bold **=** f"\033[1m"

reset **=** f"\033[0m"

Set random number seeds to maintain reproducability.

In [ ]:

**def** system\_config(SEED\_VALUE**=**42, package\_list**=None**):

"""

Configures the system environment for PyTorch-based operations.

Args:

SEED\_VALUE (int): Seed value for random number generation. Default is 42.

package\_list (str): String containing a list of additional packages to install

for Google Colab or Kaggle. Default is None.

Returns:

tuple: A tuple containing the device name as a string and a boolean indicating GPU availability.

"""

random**.**seed(SEED\_VALUE)

np**.**random**.**seed(SEED\_VALUE)

torch**.**manual\_seed(SEED\_VALUE)

**def** is\_running\_in\_colab():

**return** 'COLAB\_GPU' **in** os**.**environ

**def** is\_running\_in\_kaggle():

**return** 'KAGGLE\_KERNEL\_RUN\_TYPE' **in** os**.**environ

*#--------------------------------*

*# Check for the availability GPUs.*

*#--------------------------------*

**if** torch**.**cuda**.**is\_available():

print('Using CUDA GPU')

*# This section for installing packages required by Colab.*

**if** is\_running\_in\_colab() **or** is\_running\_in\_kaggle():

print('Installing required packages...')

**!**pip install **{**package\_list**}**

*# Set the device to the first CUDA device.*

DEVICE **=** torch**.**device('cuda')

print("Device: ", DEVICE)

GPU\_AVAILABLE **=** **True**

torch**.**cuda**.**manual\_seed(SEED\_VALUE)

torch**.**cuda**.**manual\_seed\_all(SEED\_VALUE)

*# Performance and deterministic behavior.*

torch**.**backends**.**cudnn**.**enabled **=** **True** *# Provides highly optimized primitives for DL operations.*

torch**.**backends**.**cudnn**.**deterministic **=** **True** *# Insures deterministic even when above cudnn is enabled.*

torch**.**backends**.**cudnn**.**benchmark **=** **False** *# Setting to True can cause non-deterministic behavior.*

**elif** torch**.**backends**.**mps**.**is\_available() **and** torch**.**backends**.**mps**.**is\_built():

print('Using Apple Silicon GPU')

*# Set the device to the Apple Silicon GPU Metal Performance Shader (MPS).*

DEVICE **=** torch**.**device("mps")

print("Device: ", DEVICE)

*# Environment variable that allows PyTorch to fall back to CPU execution*

*# when encountering operations that are not currently supported by MPS.*

os**.**environ['PYTORCH\_ENABLE\_MPS\_FALLBACK'] **=** '1'

GPU\_AVAILABLE **=** **True**

torch**.**mps**.**manual\_seed(SEED\_VALUE)

torch**.**use\_deterministic\_algorithms(**True**)

**else**:

print('Using CPU')

DEVICE **=** torch**.**device('cpu')

print("Device: ", DEVICE)

GPU\_AVAILABLE **=** **False**

**if** is\_running\_in\_colab() **or** is\_running\_in\_kaggle():

print('Installing required packages...')

**!**pip install **{**package\_list**}**

print('Note: Change runtime type to GPU for better performance.')

torch**.**use\_deterministic\_algorithms(**True**)

**return** str(DEVICE), GPU\_AVAILABLE

In [ ]:

*# Additional packages required for Google Colab or Kaggle.*

package\_list **=** "torchinfo"

DEVICE, GPU\_AVAILABLE **=** system\_config(package\_list**=**package\_list)

Using CUDA GPU

Installing required packages...

Collecting torchinfo

Downloading torchinfo-1.8.0-py3-none-any.whl.metadata (21 kB)

Downloading torchinfo-1.8.0-py3-none-any.whl (23 kB)

Installing collected packages: torchinfo

Successfully installed torchinfo-1.8.0

Device: cuda

In [ ]:

**from** torchinfo **import** summary

4 Data Generation

Here, we are going to generate some synthetic data to represent two features from each of the two classes.

In [ ]:

**def** generate\_data(mean\_0**=**[4.0, 20.0], stddev\_0**=**[1.0, 1.0],

mean\_1**=**[5.5, 23.0], stddev\_1**=**[0.6, 0.8],

num\_points\_0**=**200, num\_points\_1**=**200):

class\_0\_dist **=** dist**.**Normal(loc**=**torch**.**tensor(mean\_0), scale**=**torch**.**tensor(stddev\_0))

class\_1\_dist **=** dist**.**Normal(loc**=**torch**.**tensor(mean\_1), scale**=**torch**.**tensor(stddev\_1))

class\_0\_points **=** class\_0\_dist**.**sample((num\_points\_0,))

class\_1\_points **=** class\_1\_dist**.**sample((num\_points\_1,))

**return** class\_0\_points, class\_1\_points

4.1 Visualize the Dataset

In [ ]:

class\_0\_points, class\_1\_points **=** generate\_data()

plt**.**figure(figsize**=**(20, 10))

plt**.**scatter(class\_0\_points[:, 0], class\_0\_points[:, 1], color**=**"b", alpha**=**0.5, label**=**"Class:0")

plt**.**scatter(class\_1\_points[:, 0], class\_1\_points[:, 1], color**=**"r", alpha**=**0.5, label**=**"Class:1")

plt**.**legend()

plt**.**xlabel("x1")

plt**.**ylabel("x2")

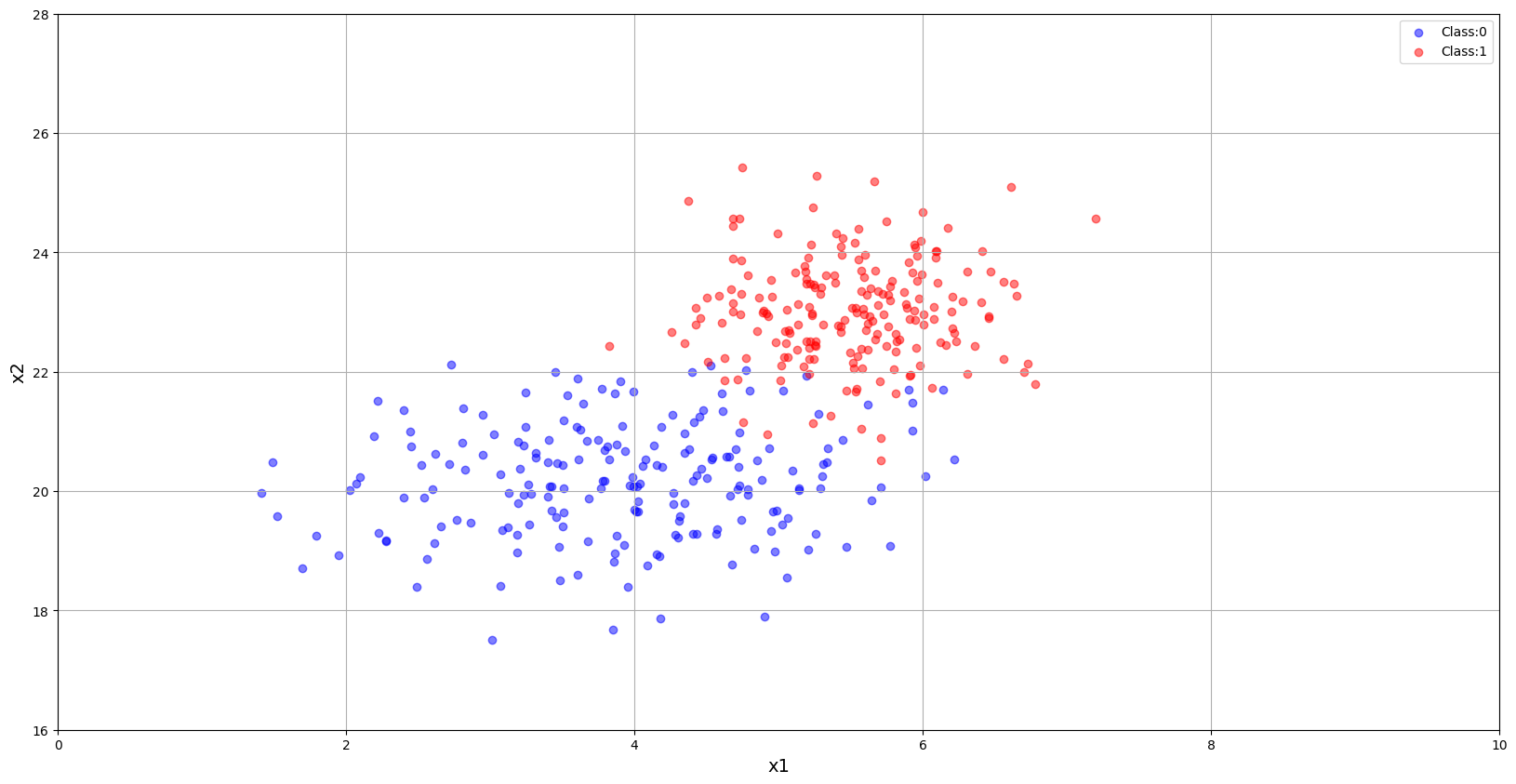
plt**.**xlim([0, 10])

plt**.**ylim([16, 28])

plt**.**grid(**True**)

plt**.**show(block**=**block\_plot)

plt**.**close()



5 Data Preparation

The following function will combine the features and labels from each class into a single data array and label array.

In [ ]:

**def** prepare\_data(class\_0\_points, class\_1\_points):

label\_zero **=** torch**.**zeros(class\_0\_points**.**shape[0], dtype**=**torch**.**float)

label\_one **=** torch**.**ones(class\_1\_points**.**shape[0], dtype**=**torch**.**float)

labels **=** torch**.**cat([label\_zero, label\_one], dim**=**0)**.**unsqueeze(dim**=**1)

data\_points **=** torch**.**cat([class\_0\_points, class\_1\_points], dim**=**0)

print(f"Data points size: {data\_points**.**shape}")

print(f"Label size: {labels**.**shape}")

**return** data\_points, labels

In [ ]:

*# Combine class and generate class labels.*

X\_train, y\_train **=** prepare\_data(class\_0\_points, class\_1\_points)

Data points size: torch.Size([400, 2])

Label size: torch.Size([400, 1])

6 Perform Training

6.1 Perform Data Normalization

Before we start with our actual training, we need to normalize our data using its mean and standard deviation.

In [ ]:

**def** normalize\_data(data, mean, std):

data\_norm **=** (data **-** mean)**/**std

**return** data\_norm

In [ ]:

print(f"Train Data shape: {X\_train**.**shape}")

mean **=** X\_train**.**mean(0)

std **=** X\_train**.**std(0)

print(f"Mean: {mean}")

print(f"Std: {std}")

X\_train **=** normalize\_data(X\_train, mean**=**mean, std**=**std)

Train Data shape: torch.Size([400, 2])

Mean: tensor([ 4.7079, 21.5937])

Std: tensor([1.1404, 1.6858])

6.2 Model Architecture Definition

In [ ]:

model **=** nn**.**Linear(in\_features**=**2, out\_features**=**1)

print(summary(model, input\_size**=**(1,2)))

==========================================================================================

Layer (type:depth-idx) Output Shape Param #

==========================================================================================

Linear [1, 1] 3

==========================================================================================

Total params: 3

Trainable params: 3

Non-trainable params: 0

Total mult-adds (M): 0.00

==========================================================================================

Input size (MB): 0.00

Forward/backward pass size (MB): 0.00

Params size (MB): 0.00

Estimated Total Size (MB): 0.00

==========================================================================================

6.3 Train the Model

In [ ]:

**def** train(model, data\_points, labels, epochs, optimizer, batch\_size**=**10):

num\_batches **=** math**.**ceil(len(labels)**/** batch\_size)

loss\_history **=** []

acc\_history **=** []

model **=** model**.**to(DEVICE)

model**.**train()

*# Trainig time measurement.*

t\_begin **=** time**.**time()

**for** epoch\_idx **in** range(epochs):

*# Clear cell outputs at the start of each epoch.*

clear\_output()

*# Shuffle data at the start of each epochs.*

shuffled\_indices **=** torch**.**randperm(len(labels))

shuffled\_data **=** data\_points[shuffled\_indices]

shuffled\_labels **=** labels[shuffled\_indices]

step\_loss **=** 0

step\_accuracy **=** 0

**for** batch\_idx **in** range(num\_batches):

start\_ind **=** batch\_idx**\***batch\_size

end\_ind **=** (batch\_idx **+** 1)**\***batch\_size

batched\_data **=** shuffled\_data[start\_ind:end\_ind]**.**to(DEVICE)

batched\_targets **=** shuffled\_labels[start\_ind:end\_ind]**.**to(DEVICE)

*# Set the weight gradients to zero for every min-batch to avoid*

*# gradient accumulation.*

optimizer**.**zero\_grad()

*# Forward pass through the model.*

logits **=** model(batched\_data)

*# Compute Loss.*

loss **=** F**.**binary\_cross\_entropy\_with\_logits(logits, batched\_targets)

*# Compute gradients using backpropagation.*

loss**.**backward()

*# Update model weights.*

optimizer**.**step()

*# Convert the output logits to probabilities.*

predictions **=** logits**.**sigmoid()

*# Batch Loss.*

step\_loss**+=** loss**.**item()**\*** batched\_data**.**shape[0]

*# Batch Acuuracy.*

step\_accuracy**+=** ((predictions **>** 0.5)**.**int()**.**cpu() **==** batched\_targets**.**cpu())**.**sum()

epoch\_loss **=** float(step\_loss **/** len(labels))

epoch\_acc **=** float(step\_accuracy**/** len(labels))

print(f"{f'{bold}[ Epoch: {epoch\_idx**+**1} ]{reset}':=^80}")

train\_loss\_stat **=** f"{bold}Loss: {epoch\_loss:.4f}{reset}"

train\_acc\_stat **=** f"{bold}Accuracy: {epoch\_acc:.4f}{reset}"

print(f"\n{train\_loss\_stat:<30}{train\_acc\_stat}")

print(f"{'='**\***72}\n")

loss\_history**.**append(epoch\_loss)

acc\_history**.**append(epoch\_acc)

print(f"Total time taken: {(time**.**time() **-** t\_begin):.2f}s")

**return** model, loss\_history, acc\_history

In [ ]:

optimizer\_sgd **=** optim**.**SGD(model**.**parameters(), lr**=**0.01)

trained\_model, loss\_history, acc\_history **=** train(

model**=**model,

data\_points**=**X\_train,

labels**=**y\_train,

epochs**=**100,

optimizer**=**optimizer\_sgd,

batch\_size**=**16)

=============================**[ Epoch: 100 ]**=============================

**Loss: 0.1341** **Accuracy: 0.9550**

========================================================================

Total time taken: 10.11s

7 Model Evaluation

7.1 Plot Training Loss

In [ ]:

plt**.**figure(figsize**=**(20, 6))

plt**.**plot(range(len(loss\_history)), loss\_history)

plt**.**xlabel("Epoch")

plt**.**ylabel("Loss")

plt**.**title("Training Loss")

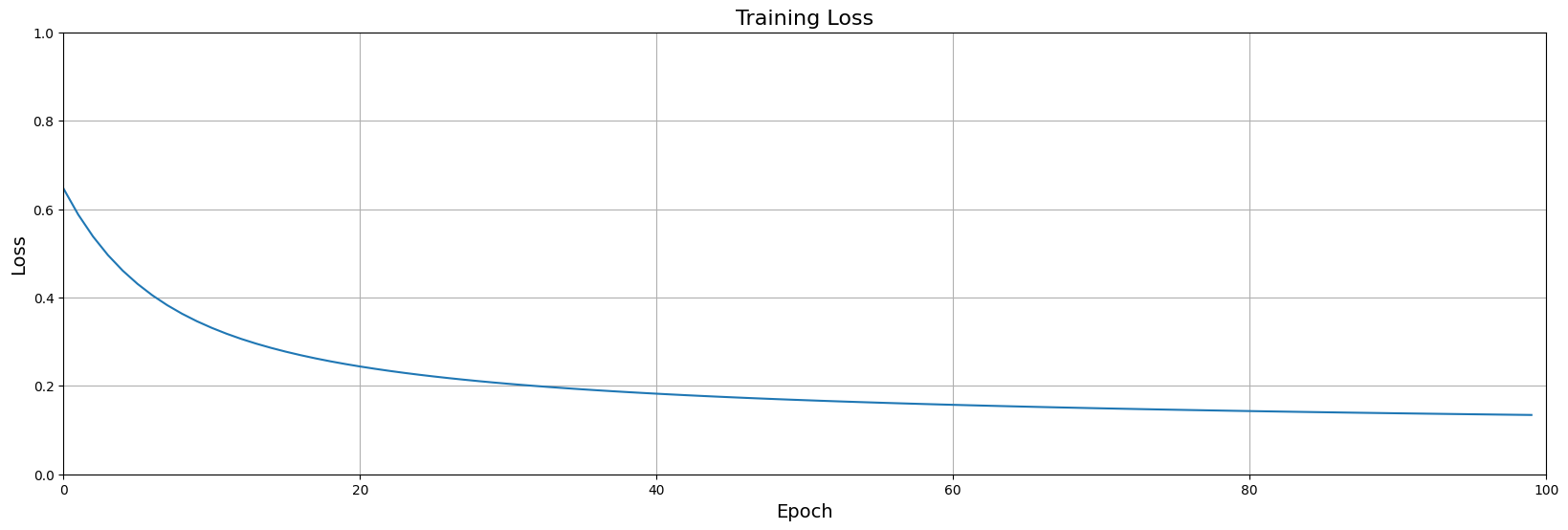
plt**.**xlim([0, 100])

plt**.**ylim([0, 1])

plt**.**grid(**True**)

plt**.**show(block**=**block\_plot)

plt**.**close()



7.2 Plot Training Accuracy

In [ ]:

plt**.**figure(figsize**=**(20, 6))

plt**.**plot(range(len(acc\_history)), acc\_history)

plt**.**xlabel("Epoch")

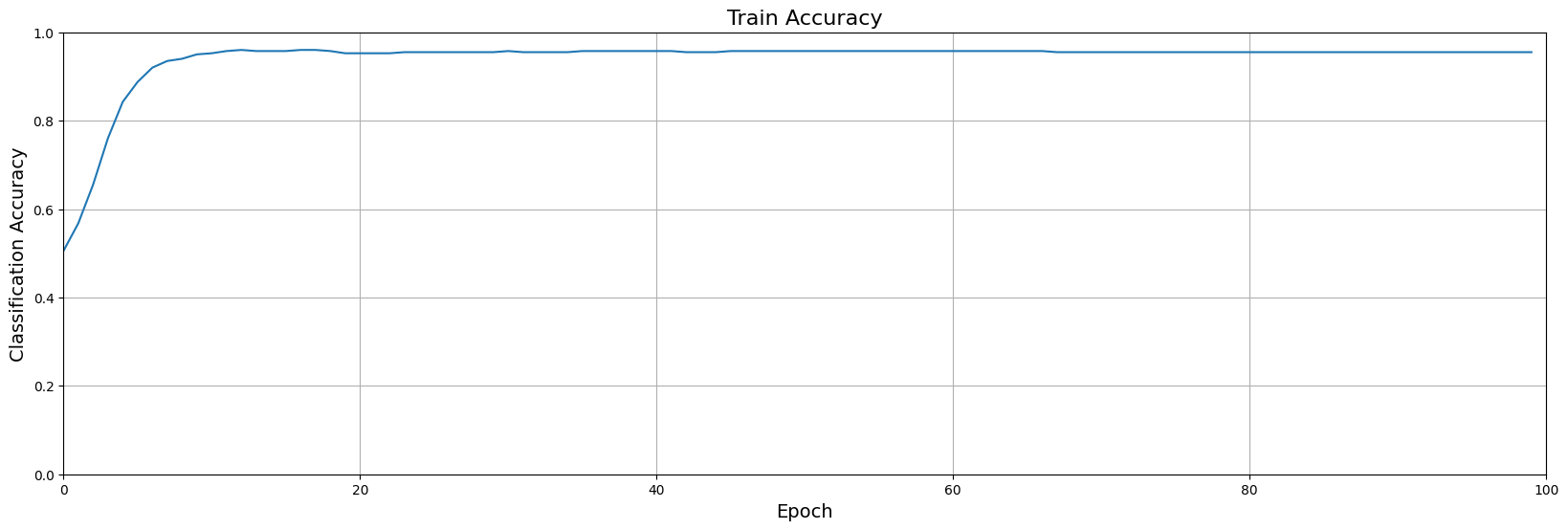
plt**.**ylabel("Classification Accuracy")

plt**.**title("Train Accuracy")

plt**.**grid(**True**)

plt**.**xlim([0, 100]);

plt**.**ylim([0, 1]);



7.3 Model Prediction

In [ ]:

*# Define a function that maps the predicted score to the appropriate class name.*

**def** pred\_class(pred):

pred **=** pred**.**squeeze()**.**numpy()

**if** pred **>** 0.5:

class\_label **=** 'Malignant'

**else**:

class\_label **=** 'Benign'

**return** class\_label

In [ ]:

*# Define a function to convert output logits to probability scores.*

**def** predict(model, inputs, device**=**"cpu"):

model**.**eval()

**with** torch**.**no\_grad():

logits **=** model(inputs**.**to(device))**.**cpu()

preds **=** logits**.**sigmoid()

**return** preds

Let's see how the model performs on a couple of example data points. Feel free to edit the code below and try other values for the features.

In [ ]:

test\_data\_1 **=** torch**.**tensor([ 5.0, 21.0 ])**.**unsqueeze(0)

test\_data\_2 **=** torch**.**tensor([ 5.0, 22.0 ])**.**unsqueeze(0)

test\_data\_1 **=** normalize\_data(test\_data\_1, mean, std)

test\_data\_2 **=** normalize\_data(test\_data\_2, mean, std)

pred\_score\_1 **=** predict(trained\_model, test\_data\_1, device**=**DEVICE)

pred\_score\_2 **=** predict(trained\_model, test\_data\_2, device**=**DEVICE)

class\_label\_1 **=** pred\_class(pred\_score\_1)

class\_label\_2 **=** pred\_class(pred\_score\_2)

print('Tumor-1: ', class\_label\_1)

print('Tumor-2: ', class\_label\_2)

Tumor-1: Benign

Tumor-2: Malignant

7.4 Model Diagnostics: The Role of the Activation Function

In this section we will create some diagnostics plots to better illustrate how binary classificaton actually works. In order to create these diagnostics

Define two convenience functions

In [ ]:

*# Neuron: WX + B*

**def** wx\_plus\_b(W, X, B):

**return** torch**.**matmul(X, W) **+** B

*# Sigmoid activation.*

**def** sigmoid(z):

**return** 1 **/** (1 **+** torch**.**exp(**-**z))

Retrieve weights from the trained model

In [ ]:

*# Retrieve weights from the trained model.*

weights **=** trained\_model**.**weight**.**detach()**.**cpu()

bias **=** trained\_model**.**bias**.**detach()**.**cpu()

w1 **=** weights[0][0]

w2 **=** weights[0][1]

b **=** bias[0]

print('Weights associated with normalized data')

print('b: ',b**.**numpy())

print('w1: ',w1**.**numpy())

print('w2: ',w2**.**numpy())

print('\n')

*# Unnormalize the weights for use in diagnostics.*

w1 **=** w1**/**std[0]

w2 **=** w2**/**std[1]

b **=** b **-** w1**\***mean[0] **-** w2**\***mean[1]

print('Weights associated with unnormalized data')

print('b: ',b**.**numpy())

print('w1: ',w1**.**numpy())

print('w2: ',w2**.**numpy())

Weights associated with normalized data

b: -0.17992271

w1: 1.4262156

w2: 2.4768257

Weights associated with unnormalized data

b: -37.793068

w1: 1.2506005

w2: 1.4691973

Plot the sigmoid activations

In [ ]:

W **=** torch**.**zeros((2, 1))

W[0][0] **=** w1

W[1][0] **=** w2

*# Compute sigmoid activations for class 0.*

z\_0 **=** wx\_plus\_b(W, class\_0\_points, b)

y\_pred\_0 **=** sigmoid(z\_0)

*# Compute sigmoid activations for class 1.*

z\_1 **=** wx\_plus\_b(W, class\_1\_points, b)

y\_pred\_1 **=** sigmoid(z\_1)

plt**.**figure(figsize**=**(20, 7))

plt**.**scatter(z\_0, y\_pred\_0, s**=**20, color**=**"b", alpha**=**0.5, label**=**"Class:0")

plt**.**scatter(z\_1, y\_pred\_1, s**=**20, color**=**"r", alpha**=**0.5, label**=**"Class:1")

plt**.**plot([0, 0], [0, 1], color**=**"darkgray")

plt**.**plot([**-**10, 10], [0.5, 0.5], color**=**"darkgray")

plt**.**xlabel('z')

plt**.**ylabel('Sigmoid(z)')

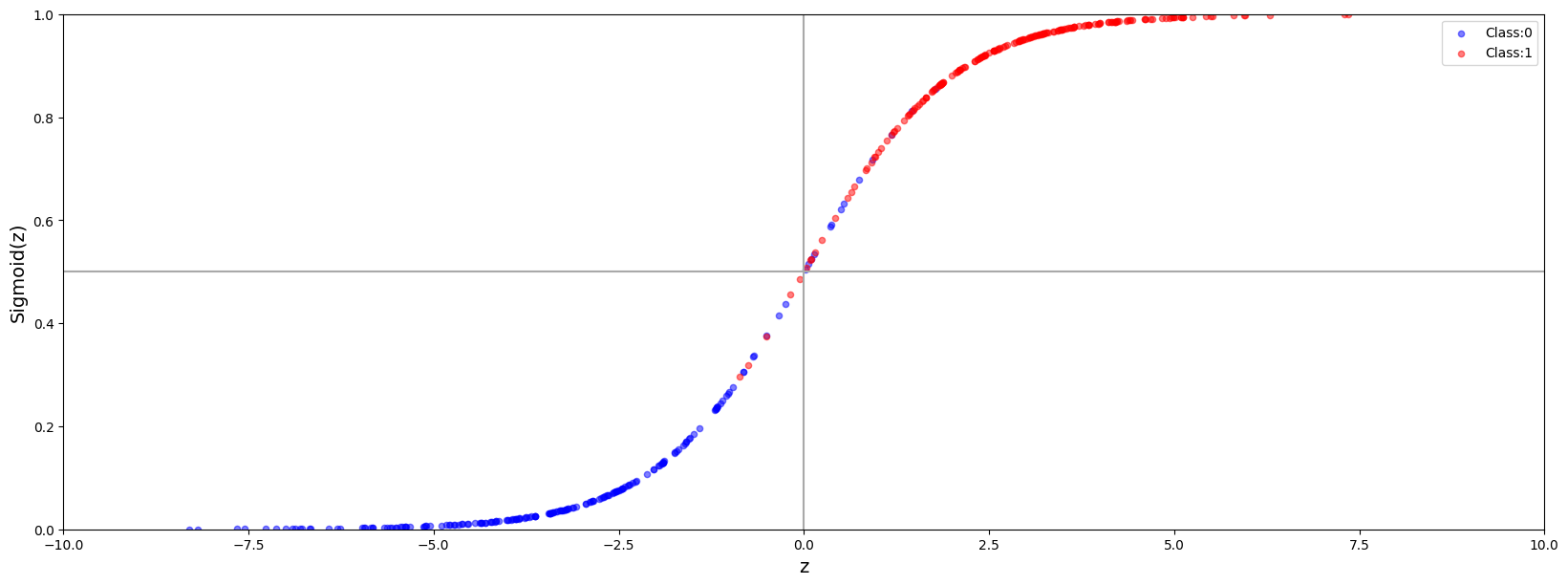
plt**.**grid(**False**)

plt**.**xlim([**-**10, 10])

plt**.**ylim([0, 1]);

plt**.**legend()

plt**.**show()



7.5 Model Diagnostics: Plot the Decision Boundary

We will plot the decision boundary using the weight and biases obtained from training the network.

In [ ]:

**def** plot\_decision\_boundary(b, w1, w2, class\_0\_points, class\_1\_points):

plt**.**figure(figsize**=**(20, 8))

plt**.**scatter(class\_0\_points[:, 0], class\_0\_points[:, 1], color**=**"b", alpha**=**0.5, label**=**"Class:0")

plt**.**scatter(class\_1\_points[:, 0], class\_1\_points[:, 1], color**=**"r", alpha**=**0.5, label**=**"Class:1")

x1 **=** torch**.**linspace(0.0, 10.0, 1000)

x2 **=** **-**(w1**/**w2)**\***x1 **-** b**/**w2

plt**.**plot(x1, x2, c**=**"black", alpha**=**.5)

plt**.**xlabel("x1")

plt**.**ylabel("x2")

plt**.**xlim([0, 10])

plt**.**ylim([16, 28])

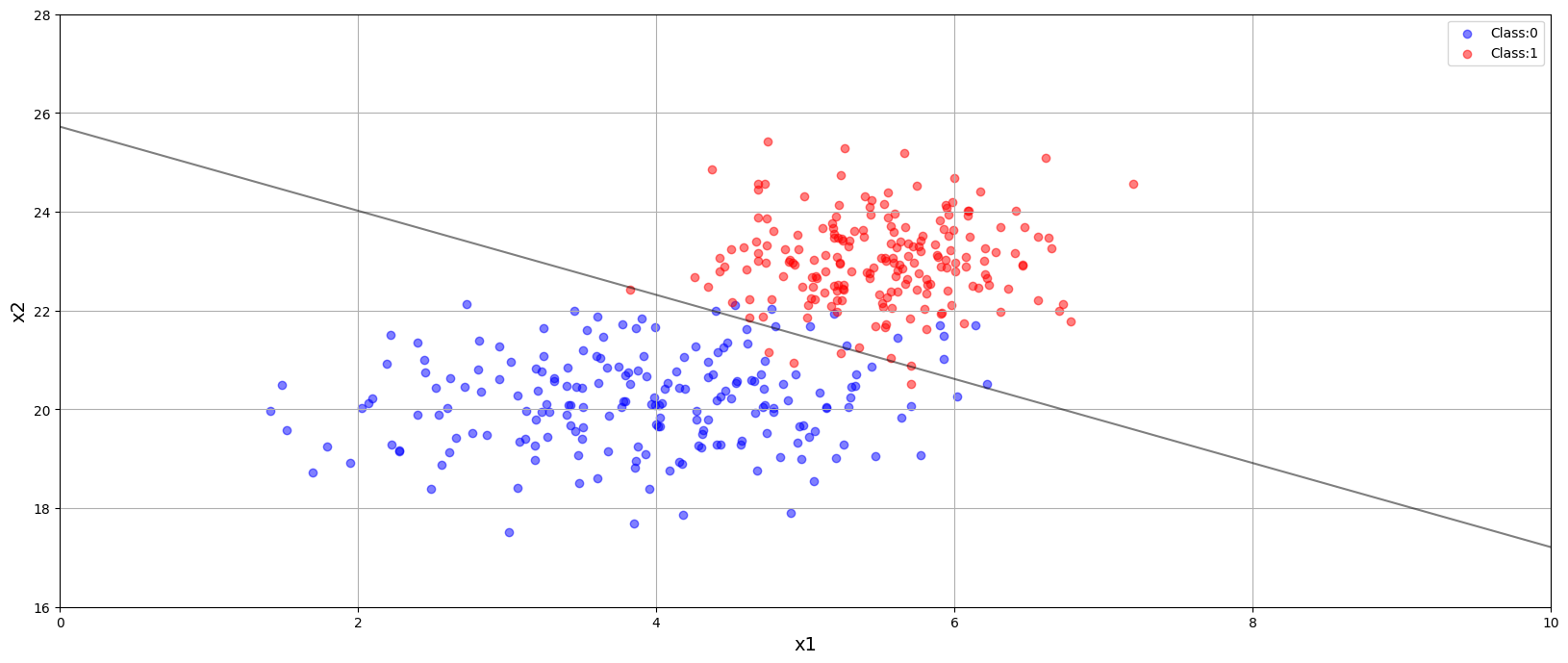
plt**.**grid(**True**)

plt**.**legend()

plt**.**show(block**=**block\_plot)

In [ ]:

plot\_decision\_boundary(b, w1, w2, class\_0\_points, class\_1\_points)



7.6 Model Diagnostics: Confusion Matrix

A confusion matrix is a very standard way to summarize the performance of a classification. In a simple visual table it provides the number (or percentage) of the following classifications. Class 0 is typically refered to as the negative class and class 1 as the positive class. In many applications these have definite meanings. For example, a malignant tumor would be considered as the positive class. So a false positive would be classifying a benign tumor (0) as malignant (1), which is an error, but not nearly as grave an error as a false negative, classifying a malignant tumor (1) as benign (0).

- True Positive (class 1 is classified as class 1)  
- False Positive (class 0 is classified as class 1)  
- True Negative (class 0 is classified as class 0)  
- False Negative (class 1 is classified as class 0)

In [ ]:

predictions **=** predict(trained\_model, X\_train, device**=**DEVICE)**.**squeeze()

predicted\_labels **=** torch**.**where(predictions**>**0.5, 1, 0)

cm **=** confusion\_matrix(y\_train**.**squeeze()**.**numpy(), predicted\_labels**.**numpy())

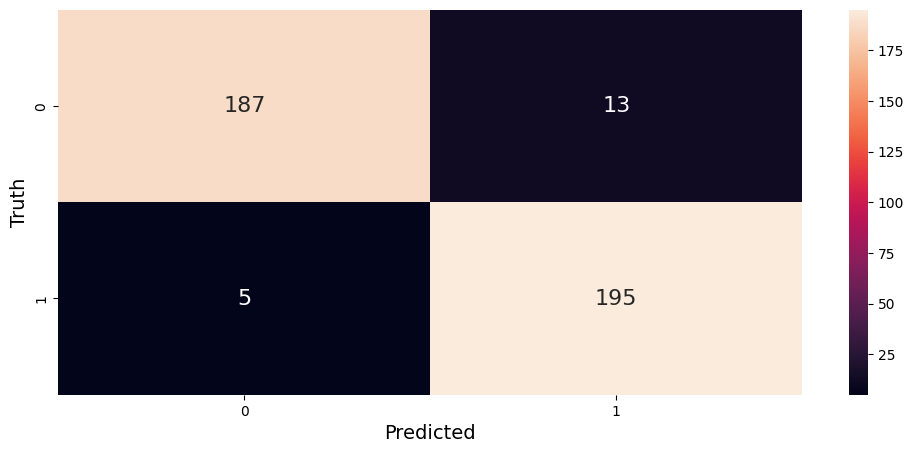
plt**.**figure(figsize**=**[12, 5])

sn**.**heatmap(cm, annot**=True**, fmt**=**'d', annot\_kws**=**{"size": 16})

plt**.**xlabel('Predicted')

plt**.**ylabel('Truth')

plt**.**show()



8 Conclusion

In this notebook, we introduced the mathematical foundation for binary classification and showed that the logistic function (or sigmoid) forms the basis for predicting categorical outputs. We showed that the loss function for binary classification is Binary Cross Entropy Loss and that this can be derived from a probabilistic interpretation of the sigmoid function. We also demonstrated how this problem can easily be formulated as a single layer neural network and implemented in PyTorch. Finally, we demonstrated how the model weights can be helpful in creating various diagnostic plots that provide additional insight.

In [ ]:

MLP with Single Hidden Layer using PyTorch

1. Define an MLP with variable number of inputs (num\_inputs), outputs (num\_outputs), and nodes in hidden layer (num\_hidden\_layer\_nodes).
2. Use ReLU activation for each node
3. Use MSE loss
4. Use SGD optimizer

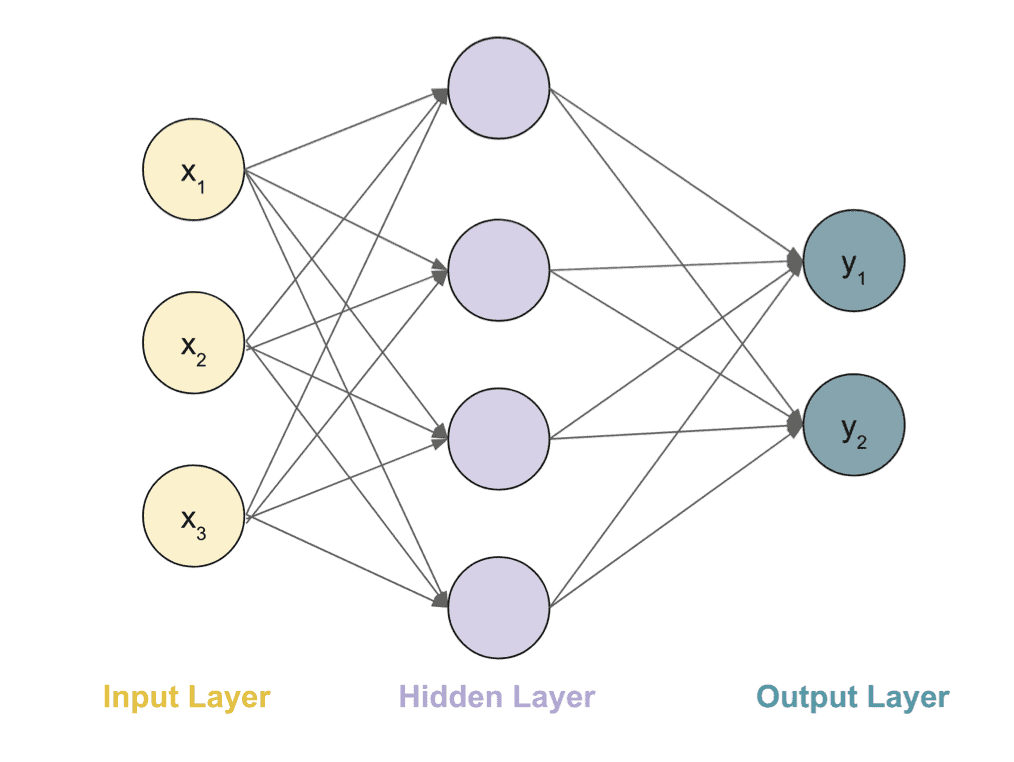


Table of Contents

* [1 Define MLP using NN Module](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/05_MLP_using_NN_Module_v3.html#1-Define-MLP-using-NN-Module)
* [2 Generate Data](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/05_MLP_using_NN_Module_v3.html#2-Generate-Data)
* [3 Perform Training](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/05_MLP_using_NN_Module_v3.html#3-Perform-Training)
* [4 MLP with Sequential Module](https://courses.opencv.org/asset-v1:deep+learning+pytorch+type@asset+block/05_MLP_using_NN_Module_v3.html#4-MLP-with-Sequential-Module)

In [ ]:

**import** numpy **as** np

**import** os

**import** random

**import** torch

**import** torch.nn **as** nn

**import** torch.optim **as** optim

**import** torch.nn.functional **as** F

**from** IPython.display **import** clear\_output

In [ ]:

**def** system\_config(SEED\_VALUE**=**42, package\_list**=None**):

"""

Configures the system environment for PyTorch-based operations.

Args:

SEED\_VALUE (int): Seed value for random number generation. Default is 42.

package\_list (str): String containing a list of additional packages to install

for Google Colab or Kaggle. Default is None.

Returns:

tuple: A tuple containing the device name as a string and a boolean indicating GPU availability.

"""

random**.**seed(SEED\_VALUE)

np**.**random**.**seed(SEED\_VALUE)

torch**.**manual\_seed(SEED\_VALUE)

**def** is\_running\_in\_colab():

**return** 'COLAB\_GPU' **in** os**.**environ

**def** is\_running\_in\_kaggle():

**return** 'KAGGLE\_KERNEL\_RUN\_TYPE' **in** os**.**environ

*#--------------------------------*

*# Check for the availability GPUs.*

*#--------------------------------*

**if** torch**.**cuda**.**is\_available():

print('Using CUDA GPU')

*# This section for installing packages required by Colab.*

**if** is\_running\_in\_colab() **or** is\_running\_in\_kaggle():

print('Installing required packages...')

**!**pip install **{**package\_list**}**

*# Set the device to the first CUDA device.*

DEVICE **=** torch**.**device('cuda')

print("Device: ", DEVICE)

GPU\_AVAILABLE **=** **True**

torch**.**cuda**.**manual\_seed(SEED\_VALUE)

torch**.**cuda**.**manual\_seed\_all(SEED\_VALUE)

*# Performance and deterministic behavior.*

torch**.**backends**.**cudnn**.**enabled **=** **True** *# Provides highly optimized primitives for DL operations.*

torch**.**backends**.**cudnn**.**deterministic **=** **True** *# Insures deterministic even when above cudnn is enabled.*

torch**.**backends**.**cudnn**.**benchmark **=** **False** *# Setting to True can cause non-deterministic behavior.*

**elif** torch**.**backends**.**mps**.**is\_available() **and** torch**.**backends**.**mps**.**is\_built():

print('Using Apple Silicon GPU')

*# Set the device to the Apple Silicon GPU Metal Performance Shader (MPS).*

DEVICE **=** torch**.**device("mps")

print("Device: ", DEVICE)

*# Environment variable that allows PyTorch to fall back to CPU execution*

*# when encountering operations that are not currently supported by MPS.*

os**.**environ['PYTORCH\_ENABLE\_MPS\_FALLBACK'] **=** '1'

GPU\_AVAILABLE **=** **True**

torch**.**mps**.**manual\_seed(SEED\_VALUE)

torch**.**use\_deterministic\_algorithms(**True**)

**else**:

print('Using CPU')

DEVICE **=** torch**.**device('cpu')

print("Device: ", DEVICE)

GPU\_AVAILABLE **=** **False**

**if** is\_running\_in\_colab() **or** is\_running\_in\_kaggle():

print('Installing required packages...')

**!**pip install **{**package\_list**}**

print('Note: Change runtime type to GPU for better performance.')

torch**.**use\_deterministic\_algorithms(**True**)

**return** str(DEVICE), GPU\_AVAILABLE

In [ ]:

*# Additional packages required for Google Colab or Kaggle.*

package\_list **=** "torchinfo"

DEVICE, GPU\_AVAILABLE **=** system\_config(package\_list**=**package\_list)

Using CUDA GPU

Installing required packages...

Collecting torchinfo

Downloading torchinfo-1.8.0-py3-none-any.whl.metadata (21 kB)

Downloading torchinfo-1.8.0-py3-none-any.whl (23 kB)

Installing collected packages: torchinfo

Successfully installed torchinfo-1.8.0

Device: cuda

In [ ]:

**from** torchinfo **import** summary

**from** IPython.display **import** clear\_output

In [ ]:

bold **=** f"\033[1m"

reset **=** f"\033[0m"

1 Define MLP using NN Module

In [ ]:

*# Define the model*

**class** MLP(nn**.**Module):

**def** \_\_init\_\_(self, num\_inputs, num\_hidden\_layer\_nodes, num\_outputs):

*# Initialize super class.*

super()**.**\_\_init\_\_()

*# Add hidden layer.*

self**.**linear1 **=** nn**.**Linear(num\_inputs, num\_hidden\_layer\_nodes)

*# Add output layer.*

self**.**linear2 **=** nn**.**Linear(num\_hidden\_layer\_nodes, num\_outputs)

**def** forward(self, x):

*# Forward pass through hidden layer with*

x **=** F**.**relu(self**.**linear1(x))

*# Foward pass to output layer*

**return** self**.**linear2(x)

2 Generate Data

In [ ]:

*# Num data points.*

num\_data **=** 1000

*# Data parameters.*

num\_inputs **=** 1000

num\_outputs **=** 10

*# Create random Tensors to hold inputs and outputs.*

X **=** torch**.**randn(num\_data, num\_inputs)

Y **=** torch**.**randn(num\_data, num\_outputs)

3 Perform Training

In [ ]:

**def** train(model, criterion, optimizer, data, targets, num\_epochs):

model**.**train()

**for** epoch\_idx **in** range(num\_epochs):

*# Clear cell outputs at the start of each epoch.*

clear\_output()

*# Zero gradients, perform a backward pass, and update the weights.*

optimizer**.**zero\_grad()

*# Forward pass: Compute predicted y by passing data to the model.*

y\_pred **=** model(data)

*# Compute and print loss*

loss **=** loss\_function(y\_pred, targets)

*# Calculate gradient using backward pass.*

loss**.**backward()

*# Update model parameters (weights).*

optimizer**.**step()

print(f"{f'{bold}[ Epoch: {epoch\_idx**+**1} ]{reset}':=^80}")

train\_loss\_stat **=** f"{bold}Loss: {loss:.4f}{reset}"

print(f"\n{train\_loss\_stat}")

print(f"{'='**\***72}\n")

**return**

3.1 Define Model Parameters, Loss Function and Optimizer

In [ ]:

*# Training parameters.*

num\_epochs **=** 100

*# Network parameters.*

num\_hidden\_layer\_nodes **=** 100

*# Get reproducible results*

torch**.**manual\_seed(42);

*# Construct our model by instantiating the class defined above*

model **=** MLP(num\_inputs, num\_hidden\_layer\_nodes, num\_outputs)

print(summary(model, input\_size**=**(1,num\_inputs), device**=**"cpu", row\_settings**=**["var\_names"]))

*# Define loss function*

loss\_function **=** nn**.**MSELoss(reduction**=**'sum')

*# Define optimizer*

optimizer **=** optim**.**SGD(model**.**parameters(), lr**=**1e-4)

==========================================================================================

Layer (type (var\_name)) Output Shape Param #

==========================================================================================

MLP (MLP) [1, 10] --

├─Linear (linear1) [1, 100] 100,100

├─Linear (linear2) [1, 10] 1,010

==========================================================================================

Total params: 101,110

Trainable params: 101,110

Non-trainable params: 0

Total mult-adds (M): 0.10

==========================================================================================

Input size (MB): 0.00

Forward/backward pass size (MB): 0.00

Params size (MB): 0.40

Estimated Total Size (MB): 0.41

==========================================================================================

In [ ]:

train(model, loss\_function, optimizer, data**=**X, targets**=**Y, num\_epochs**=**num\_epochs)

=============================**[ Epoch: 100 ]**=============================

**Loss: 0.9612**

========================================================================

4 MLP with Sequential Module

Observe that in the section above, we had defined the Linear and the ReLU modules individually.

The value a Sequential module provides over manually calling a sequence of modules is that it allows treating the whole container as a single module, such that performing a transformation on the Sequential applies to each of the modules it stores.

In [ ]:

*# Define the model*

**class** MLP\_Sequential(torch**.**nn**.**Module):

**def** \_\_init\_\_(self, num\_inputs, num\_hidden\_layer\_nodes, num\_outputs):

*# Initialize super class*

super()**.**\_\_init\_\_()

*# Build model using Sequential container.*

self**.**model **=** nn**.**Sequential(

*# Add hidden layer.*

nn**.**Linear(num\_inputs, num\_hidden\_layer\_nodes),

*# Add ReLU activation.*

nn**.**ReLU(),

*# Add output layer.*

nn**.**Linear(num\_hidden\_layer\_nodes, num\_outputs)

)

**def** forward(self, x):

*# Forward pass.*

**return** self**.**model(x)

We are going to use the same training parameters that we have defined in the previous sections.

In [ ]:

*# Training parameters.*

num\_epochs **=** 100

*# Network parameters.*

num\_hidden\_layer\_nodes **=** 100

*# Get reproducible results*

torch**.**manual\_seed(42);

*# Construct our model by instantiating the class defined above*

model\_seq **=** MLP\_Sequential(num\_inputs, num\_hidden\_layer\_nodes, num\_outputs)

print(summary(model\_seq, input\_size**=**(1,num\_inputs), device**=**"cpu", row\_settings**=**["var\_names"]))

*# Define loss function*

loss\_function **=** nn**.**MSELoss(reduction**=**'sum')

*# Define optimizer*

optimizer **=** optim**.**SGD(model\_seq**.**parameters(), lr**=**1e-4)

==========================================================================================

Layer (type (var\_name)) Output Shape Param #

==========================================================================================

MLP\_Sequential (MLP\_Sequential) [1, 10] --

├─Sequential (model) [1, 10] --

│ └─Linear (0) [1, 100] 100,100

│ └─ReLU (1) [1, 100] --

│ └─Linear (2) [1, 10] 1,010

==========================================================================================

Total params: 101,110

Trainable params: 101,110

Non-trainable params: 0

Total mult-adds (M): 0.10

==========================================================================================

Input size (MB): 0.00

Forward/backward pass size (MB): 0.00

Params size (MB): 0.40

Estimated Total Size (MB): 0.41

==========================================================================================

In [ ]:

train(model\_seq, loss\_function, optimizer, data**=**X, targets**=**Y, num\_epochs**=**num\_epochs)

=============================**[ Epoch: 100 ]**=============================

**Loss: 0.9612**

========================================================================