Journal of Basic and Applied Engineering Research
p-ISSN: 2350-0077; e-ISSN: 2350-0255; Volume 11, Issue 2; April-June, 2024, pp. 37-44
© Krishi Sanskriti Publications
http://www.krishisanskriti.org/Publication.html

The Usage of Machine Learning to Predict Outcomes in Diverse Areas such as the Annual Water Consumption of a Country and the Coefficient of Drag of Vehicles

Aditya Ganesh

Dubai College, Dubai, UAE
E-mail: 001adityaganesh@gmail.com

Abstract—*Machine Learning is a fascinating and rapidly growing field that is projected to show an annual growth rate of 36.08% until 2030*[1]*. This paper introduces an outline of Machine Learning and explains the core mathematics behind one of its most common models. The paper then reviews its ability and efficacy in predicting outcomes when given certain factors, using very different data sets, evaluating the model's ability to adapt to the vastly different data, and comparing the outcomes and implementation of the model.*

INTRODUCTION

Machine Learning is a fascinating field that is a subset of artificial intelligence (AI) and focuses on allowing systems to learn, improve, and develop their 'understanding' from experience without being explicitly programmed[2]. In machine learning, algorithms are trained to recognize patterns, make predictions, or perform tasks without being explicitly programmed for each scenario[3].

There are several types of machine learning algorithms. One of them includes supervised learning; in this type of learning, the algorithm is trained on a labeled dataset, where each input is paired with the correct output[4]. The algorithm then trains itself on this data and learns to map inputs to outputs, enabling it to make predictions on new, unseen data[5]. Unsupervised learning - another machine learning algorithm - is where the algorithm is given unlabeled data and tasked with finding patterns or structures within it[6]. Semi-supervised learning - which this paper will be focusing on - is a combination of both aforementioned types of algorithms; the algorithm is trained on a dataset containing both labeled and unlabeled data[7]. It learns from the unlabeled data to improve its performance whilst also using the labeled data to make predictions[8].

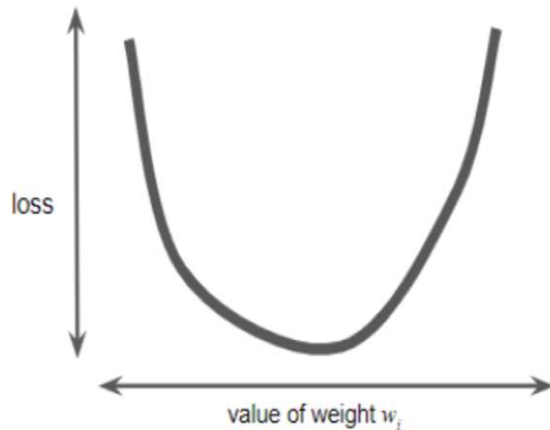
The mathematics behind the model

There are many approaches to machine learning, one of them being random initialization. Let's take the linear function $f(x) = Wx + b$[9]. To find the optimal parameters in this method, we randomly initialize or guess the values of W, the weights, and b, the bias[10]. This just means assigning random values to these parameters without any prior knowledge. Once we've used these guesses, we use the model to make predictions based on training, validation and testing data. We can then evaluate the performance of the model using metrics such as mean squared error in order to assess the accuracy of the model[11]. Based on the performance of the model, we can adjust the parameters W and b and repeat the process. We can then repeatedly make informed guesses based on the performance of the model in order to make it more accurate as needed. Even though this method could be a useful initial introduction into the field, it is not a scalable or efficient process.

The more accurate approach to machine learning is using techniques such as gradient descent in order to minimize a loss function, and this is the method that this paper will be utilizing. A loss function in machine learning quantifies how well the model's predictions match the real answers[12]. The primary objective of machine learning is to identify the parameters mentioned before that minimize the loss function. We aim to minimize the loss function as it makes the model's predictions as close as possible to the true values[13]. The

[1] Statista. *Machine Learning - Worldwide* (2024)
[2] Brown, S. *Machine learning, explained* (2021)
[3] *Undeartanding Machine Learning (ML)* (n.d.)
[4] *What is supervised learning?* (n.d.)
[5] ibid
[6] *What is unsupervised learning?* (n.d.).
[7] Bergmann, D. *What is semi-supervised learning?* (2023)
[8] ibid
[9] Kumar, A. *Neural Network* (2019)
[10] ibid
[11] ibid
[12] Arslan, E. *what does The Loss (Cost) Function mean in Deep Learning* (2023)
[13] ibid

process of finding the parameters that minimize the loss function is commonly referred to as optimization[14]. Various optimization algorithms can be used to reduce the loss function and find the best parameters, one of the most popular algorithms being gradient descent, which iteratively updates the parameters in the opposite direction of the gradient of the loss function, helping minimize the loss function and getting closer and closer to the true values[15]. The gradient of the loss function points in the direction of the steepest increase in the loss function[16]. By moving along the function in the opposite direction, we decrease the loss, as can be seen by the graph below that helps visualize this (this image has been taken from Google for Developers)[17]. The learning rate parameter controls the size of the steps taken during this process[18]. If the learning rate parameter is too large, it can cause the model to overshoot the true value and not minimize its loss function, causing the loss to start increasing as it goes past the minimum point of the curve. On the other hand, a parameter that is too small can make the model too slow. A typical loss function is mean squared error.



In order to approximate a general 1-dimensional function f(x), we say that $f_{approx}(x) = \phi(W_l\, \phi\, (W_{l-1}\phi\, (\ldots W_1 x + b_1) + b_{l-1}) + b_l)$, where ϕ is a non-linear function like $\phi(x) = max(0, x)$. The input, x, is the independent variable representing the input data. This neural network has multiple layers (l layers). Within each layer, after performing the linear transformation $W_i x + b_i$, the result is passed through the non-linear function ϕ. This is a ReLU (rectified linear unit) function, where $\phi(x) = max(0, x)$. ReLU is a common activation function in neural networks, as it introduces a simple non-linearity to the model to help with the learning process[19]. Within each layer l,the expression represents a series of linear transformations followed by non-linear transformations. In a neural network, there are multiple hidden layers. Each neuron in a hidden layer receives inputs from all the neurons in the previous layers. W_i represents the weight matrix associated with the ith layer, and b_i is the bias vector for the ith layer[20]. The linear transformation $W_i x + b_i$ involves multiplying the input x with the weight matrix and then adding the bias vector to it. The final output is $f_{approx}(x)$, which is the approximation of the original function $f(x)$.

We then iteratively adjust the parameters W_i and b_i to better approximate the true function of $f(x)$: $W_i^{t+1} = W_i^t - \epsilon \frac{dl}{dW_i}|_{W_i^t}\, ; b_i^{t+1} = b_i^t - \epsilon \frac{dl}{db_i}|_{b_i^t}$. l is the loss function (such as $l = [f(x) - f_{approx}(x;\ W_i^t\, , b_i^t\,)]^2$) and ϵ is the learning rate. In this process, the parameters $W_i$and b_i of the neural network are adjusted iteratively to improve the approximation of the true value of the function $f(x)$. The parameters W_i^t and b_i^t represent the updated parameters at the iteration t, whilst the parameters W_i^{t+1} and b_i^{t+1} represent the updated parameters at the next iteration, $t + 1$. The updated parameters are determined, as mentioned before, by analyzing the gradient of the loss function with respect to the parameters ($\frac{dl}{dW_i}$ and $\frac{dl}{db_i}$), and moving in the opposite direction of the gradient in order to minimize the loss function l. The learning rate ϵ controls the size of the steps taken during the parameter 'updates'. The loss function l measures the difference between the true function $f(x)$ and the approximation of the function $f_{approx}(x;\ W_i\, , b_i)$. The loss function mentioned above ($l = [f(x) - f_{approx}(x;\ W_i^t\, , b_i^t\,)]^2$) is the squared difference between the approximated and true function. This entire process is repeated until the approximation of the function f_{approx} gets sufficiently close to the true function $f(x)$. Similar models can also be used for multi-dimensional data. When multiple dimensions make it very complex computationally to deal with high-dimensional data, networks can reduce the dimensionality in order to make all the data uniform and to make it easier to deal with it.

Now let us take a look at our first test that we will be doing with the model. We will be investigating the effectiveness of the model in predicting the drag coefficient of a vehicle after being given certain information about the vehicle, including height, width and frontal area. Usually, in order to calculate the drag coefficient of a vehicle, you would use the equation $C_D = \frac{F_D}{\rho A \frac{V^2}{2}}$, where C_D is the drag coefficient, F_D is the drag force, A is the reference area, ρ is the density of the fluid and V is the velocity of the object[21]. With the model that we have created, we are first going to see if the model can pick up any patterns or relationships between the height, width and frontal

[14] *Machine Learning Optimization - Why is it so Important?* (2021)
[15] Arslan, E. *what does The Loss (Cost) Function mean in Deep Learning* (2023)
[16] *Reducing Loss: Gradient Descent* (n.d.)
[17] ibid
[18] *Reducing Loss: Learning Rate* (n.d.)
[19] Krishnamurthy, B. *An Introduction to the ReLU Activation Function*; 26/02/2024
[20] Sarita. *Basic Understanding of Neural Network Structure*; 03/10/2023
[21] *Drag Coefficient*; n.d

area of many cars in relation to their drag coefficient and see if the model can predict the drag coefficient of other vehicles.

Setting up the model

A sample of some of the first data set given to the model is below:

Make	Model	Year	Height (In)	Width (In)	Frontal Area (ft^2)	Cd
Acura	Integra	1994 - 2001	51.9	66.7	19.5	0.32
Audi	A8	1994 - 2002	56.6	74	24.22	0.28
BMW	7-series	1994 - 2001	56.1	73.3	23.79	0.30
Chevrolet	Cavalier	1995 - 2005	53.2	67.4	20.2	0.36

All data has been taken from EcoModder[22].

This is a sample of 4 of the 307 cars' data that is used in the experiment. As can be seen, none of the actual data that is mathematically required to calculate the drag coefficient (the actual data being drag force, reference area, density of the fluid, and velocity of the object) is present. We want the model to find relationships between the drag coefficient and these characteristics of the vehicles in order to predict the drag coefficients of other vehicles without the model knowing what the answer is. The features that we use to predict it are height, width, frontal area, and the coefficient of drag. The make, model, and year of the vehicle are just to help our visualization of the data but will not be used by the program.

To start setting up the machine learning algorithm, we have to import packs such as NumPy[23] and PyTorch[24] onto Python[25]. The first thing we do is split the data up into three sections: training data, validation data and testing data. The purpose of the training data is for the model to learn any patterns and relationships present in the data. The model adjusts its parameters during the process using gradient descent in order to minimize the loss function. At the end of the training, the model is expected to have identified underlying patterns in order to make predictions. The purpose of the validation data is to evaluate the performance of the model during the training. After each training iteration, the model's performance is evaluated using the validation data, allowing for the evaluation of the model's performance on unseen data. The validation data helps the model choose the best parameters that produce the most accurate and precise outcome. The purpose of the testing data is to provide an unbiased evaluation of the final model. Its performance is tested using the testing data and is only used at the end of the development process of the model. The testing data will give an overall estimate of how well the model will perform on unseen, new data.

```python
train_data, temp_data = train_test_split(car_data_filtered, test_size=0.1, random_state=42)
val_data, test_data = train_test_split(temp_data, test_size=0.5, random_state=42)
```

As can be seen by the code, the data is split up into 90% training data, 5% validation data, and 5% testing data. There is no specific split for the amount of data for each subset, but this particular split is quite common for machine learning models. By allocating a large amount of data (90%) to the training subset, the model has a sufficient amount of data to learn the patterns and connections within the data, which can lead to better performance. The rest of the data is evenly split between validation and testing, which will ensure that the model can evaluate and test itself with sufficient data.

We then scale the data in order to standardize the data by calculating the mean of each column of training, validation, and testing data as well as the standard deviation of each column of training, validation, and testing data. The same process is done for each set of data. For example, we do the following to the training data:

```python
# Calculate the mean of each column for training data
mean_train = np.mean(train_data, axis=0)
print("Mean of each column for Training Data:")
for i, header in enumerate(entry_headers):
    print(f"{header}: {mean_train[i]}")
    print(mean_train.shape)

# Calculate the standard deviation of each column for training data
std_train = np.std(train_data, axis=0)
print("\nStandard Deviation of each column for Training Data:")
for i, header in enumerate(entry_headers):
    print(f"{header}: {std_train[i]}")
```

We then calculate the mean subtracted from the value divided by the standard deviation for each column ($\frac{value - \mu}{\sigma}$) - this is called z-score normalization, also known as standardization[26].

```python
mean_std_ratio_train = (train_data - mean_train) / std_train
mean_std_ratio_val = (val_data - mean_val) / std_train
mean_std_ratio_test = (test_data - mean_test) / std_train
```

This transformation results in a distribution of the data with a mean of 0 and a standard deviation of 1. It centers the data around 0 and also scales it to have a unit variance. This is useful when the features have different units or scales, as does

[22] EcoModder. *Vehicle Coefficient of Drag List*; n.d.

[23] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. *Array programming with NumPy*; 16/09/2020. Nature 585, 357–362

[24] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. *PyTorch: An Imperative Style, High-Performance Deep Learning*; 2019. p. 8024–35.

[25] Van Rossum G, Drake Jr FL. *Python reference manual*; 1995.

[26] Codecadamy Team. *Normalization*; n.d.

the height, width, frontal area and drag coefficient. It brings them onto a common scale and makes them easier to compare and so that each feature is of equal importance[27].

We then have to extract certain features from our data in order to make it easier for the model to train and use the data.

```python
train_features = mean_std_ratio_train[:, 1:]
val_features = mean_std_ratio_val[:, 1:]
test_features = mean_std_ratio_test[:, 1:]

train_output = mean_std_ratio_train[:, 0]
val_output = mean_std_ratio_val[:, 0]
test_output = mean_std_ratio_test[:, 0]
```

For the features, the code slices the arrays with [: , 1 :], which extracts the features from all the rows and columns starting from index 1. It considers all the data except for that from the first column, which contains the coefficient of drag for every vehicle, the outcome we are trying to predict. For the output, the code slices the arrays with [: , 0], which only extracts data from the first column, containing the target output (the drag coefficient).

Next, we have to determine the dimensions of our data and set up the model.

```python
# Determine the dimensions
n_features = train_features.shape[1]
n_hidden = 10
```

```
n_embd = 1
# Define the model
model = nn.Sequential(
    nn.Linear(n_features, n_hidden, bias=True),#Line 1
    nn.ReLU(), #Line 2
    nn.Linear(n_hidden, n_embd, bias=True), #Line 3
```

The line defining n_features calculates the number of features in the training dataset. 'n_hidden=20' specifies the number of neurons in the hidden layer of an MLP (multi-layer perceptron) neural network. In this type of neural network, the hidden layers are intermediary layers between the input and output layers[28]. They are responsible for the model learning the different patterns and connections in the data. The number of hidden layers we have defined here is 10. 'n_embd = 1' specifies the dimensionality of the output embedding. It is defined as 1, setting the dimensionality of the output embedding to 1, meaning that the output will be a scalar value. We then come to the next part of the code where we define the code using PyTorch's 'nn.Sequential' module. It is a module that is used to sequentially stack layers or modules one after the other, allowing us to create a neural network model[29]. In line 1, we are creating a linear transformation layer that maps the input features to the hidden layer. 'N_features' represents the number of input features and 'n_hidden' represents the number of neurons in the hidden layer. This layer applies a linear transformation to the input features, where each of the features is multiplied by a weight and then summed together, and biases are also included here. Line 2 runs the data through the ReLU activation function, and finally, line 3 creates another linear transformation layer that maps the output of the hidden layer to the output layer. It also contains bias terms, producing the final output of the model.

The next steps in setting up the model include looking at the actual iteration of the parameters and introducing our loss functions where the model can go through the process of gradient descent.

```python
#Creating Loop
for p in parameters:
    p.requires_grad = True

#Training Loss and Optimizer
loss_function = nn.MSELoss() #Line 1
optimizer = optim.SGD(model.parameters(), lr=0.01) #Line 2
```

We first create a loop that iterates over the parameters of the model. 'Parameters' contains a list of all the parameters in the model, including weights and biases of the neural network layers. By setting 'requires_grad' to true, we enable the model to track the gradients for these parameters, allowing them to be continually updated throughout the training process. Next, we define the training loss and the optimizer. Line 1 introduces the mean squared error (MSE) loss function, which is commonly used in machine learning to compute the mean squared difference between the target and the predicted values[30]. During the training process of this model, the goal will be to minimize this loss function in order to get as close as possible to the target value. Line 2 initializes the optimizer in order to train the model. 'optim.SGD' introduces the actual gradient descent segment of this mode, introducing the stochastic gradient descent (SGD) optimizer[31]. Line 2 also provides the parameters of the model that need to be optimized, and it also provides the learning rate. The learning rate, as mentioned before, determines the size of the steps to be taken during the optimization process. The SGD optimizer updates the parameters of the model based on the gradients calculated - as well as based on the learning rate - in order to minimize the loss function.

The next step in the model is to set up the model for training.

[27] Codecadamy Team. *Normalization*; n.d.
[28] *Hidden Layer*; n.d.
[29] *Sequential*; n.d.
[30] Alake, R. *Loss Functions in Machine Learning ExplainedI*; 11/2023
[31] *SGD*; n.d.

```
Python
# Step 1
batch_size = 32

# Step 2
random_indices = torch.randint(train_features.shape[0], (batch_size,))
batch_data = torch.tensor(train_features[random_indices], dtype=torch.float32)

# Step 3
first_layer_output = model[0](batch_data)
first_layer_activation = torch.relu(first_layer_output)
```

First, the code sets the batch size to 32, which dictates how many samples are processed together in one iteration of training, influencing both the speed and the stability of learning[32]. The number 32 is commonly used as it balances computational efficiency and the accuracy of the gradient descent process[33]. The next step generates random indices to select a subset of training data ('batch_data'), ensuring the input is of the type 'float32'. This creates variability in the training process to help the model generalize better. Finally, in step 3, the batch data is fed through the model's first layer, undergoing a transformation by the layer's weights. The output is then passed through a ReLU activation function, introducing non-linearity by setting negative values to zero. This step prepares the data for further processing by adding the capability to capture complex patterns, with first_layer_activation holding the resulting activated output.

Finally, we test the model with the training and the validation data.

```
Python
import torch
import torch.nn as nn
import torch.optim as optim

n_iterations = 100000
batch_size = 32

train_loss_list = []
val_loss_list = []

for i in range(n_iterations):
    batch_ix = torch.randint(0, train_features.shape[0], (batch_size,))
            batch_x,    batch_y    =    torch.tensor(train_features[batch_ix]).float(),
torch.tensor(train_output[batch_ix]).float()

    optimizer.zero_grad()
    output = model(batch_x)
    loss = loss_function(output, batch_y)
    loss.backward()
    optimizer.step()

    with torch.no_grad():
                                                    testingvalidation_output    =
model(torch.tensor(val_features).float()).squeeze()
                              val_loss   =   loss_function(torch.tensor(val_output),
torch.tensor(testingvalidation_output).float())

    train_loss_list.append(loss.item())
    val_loss_list.append(val_loss.item())

    if i % 25 == 0:
        print(f"Batch: {i} Train Loss: {loss.item()}, Validation Loss:
```

[32] Brownlee, J. *How to Control the Stability of Training Neural Networks With the Batch Size*; 28/08/2020

[33] ibid

In this cell, PyTorch modules for core functionalities such as for the neural network layers and the optimization algorithms are imported. The number of iterations is then set to 100,000 and the batch size is set to 32. Additionally, two lists, 'train_loss_list' and 'val_loss_list', are initialized to keep track of the training and validation loss at each iteration. The code then moves onto the main part of the machine learning process: the training loop. It iterates 100,000 times, first selecting a random batch from the training set and then performing a forward pass to compute predictions (a forward pass is just the computation of the output from the input through all layers, applying weights and activation functions). Next, it calculates the loss by comparing the predictions against the true labels and conducts a backward pass to compute the gradients (a backward pass calculates and propagates gradients from the output back to the input, updating model weights based on loss differentiation). It then updates the model weights using the optimizer and periodically evaluates the model on a validation set to calculate the validation loss, without computing gradients. Finally, it prints the training and validation loss, enabling us to find trends.

Evaluating the model

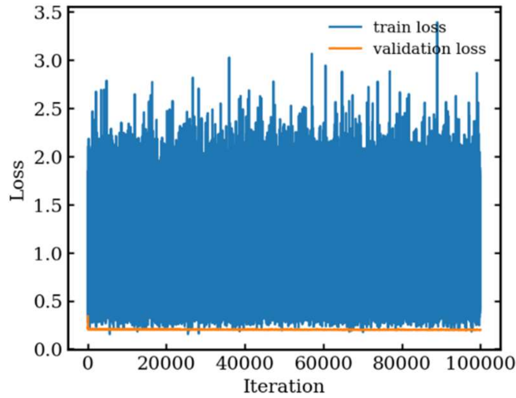
Upon the first review of the results, the training data does not seem to have much consistency. There does not seem to be much stabilization and there is a lot of noise and variance as the loss keeps fluctuating with no general rate. The batch size was then switched to 64, and with this the model worked much better. With the learning rate set to 0.01, the batch size set to 64, and the number of iterations set to 100,000, the training loss seems to vary between 0.3 and 2.3, with great fluctuations in the training loss and with no trend coming down to its minimum loss. Contrastingly, the validation loss is consistently lower, between 0.198 and 0.200. We want this loss to be as minimal as possible.

To visualize the training and the validation loss, we can write a short program to plot this information on a graph:

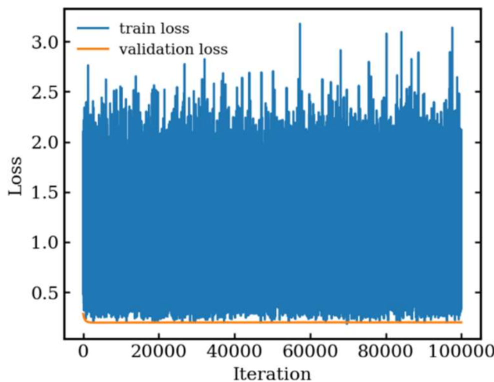
```
Python
plt.figure()

plt.plot(train_loss_list,label='train loss')
plt.plot(val_loss_list,label='validation loss')

plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.legend()
```

With the current settings of a learning rate of 0.01, a batch size of 64 and the number of iterations set to 100000, the model produces a graph like the one attached below. It shows that the training loss is not consistent whatsoever, and has major fluctuations as the number of iterations progresses. The training loss does not become consistently lower, highlighting

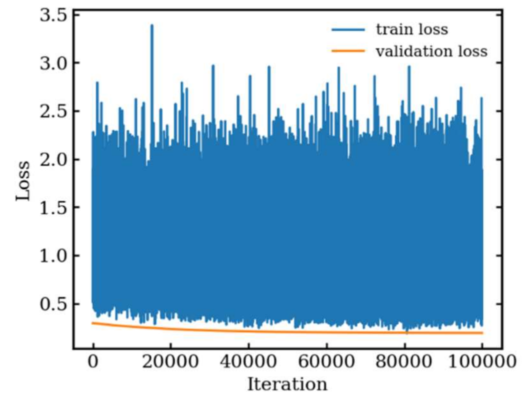
that the model cannot find many links or patterns between the provided data for predicting the drag coefficient. The validation loss, however, starts off positively. It initially starts to decrease as the number of iterations increases, but then the validation loss decreases at an extremely low rate as the number of iterations increases. The validation loss decreases minimally over time and reaches a minimum of 0.198.



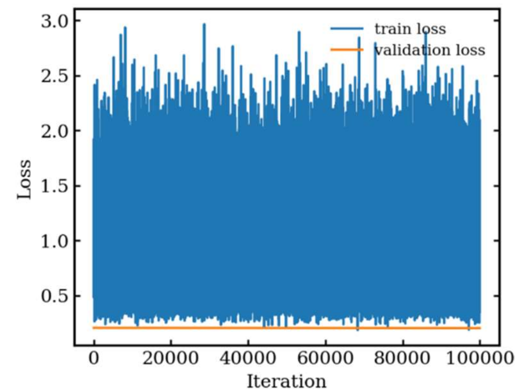
However, we can vary factors like the learning rate and see what effect it has on the outcome. Changing the learning rate from 0.01 to 0.001 makes the optimization process more gradual and precise, theoretically leading to improved training stability and accuracy but at the cost of longer training time as well as potentially overshooting the minimum loss. Doing, this, we still receive similar results - the graph looks very similar to the previous graph. The training data loss still has very large fluctuations within a similar range as before; however, the validation loss is marginally less accurate as it is slightly higher compared to when the learning rate was 0.01 (the lowest loss it outputs is 0.199). The range of the validation loss is lower as the initial loss is lower compared to the previous graph.



When further reducing the learning rate to 0.0001, the training loss as well as the validation loss both have a similar pattern, and the validation loss falls to the lowest of 0.197. When decreasing the learning rate even further to 0.00001, the training and validation loss, when plotted, produces a graph below. The training loss decreases as the number of iterations increases as well as is a bit more consistent. The validation loss, however, sees the biggest change; the validation loss reduces the most over time, having a much more prominent gradient function compared to the gradient functions of the other graphs. The validation loss decreases to the lowest of 0.193.



Decreasing the learning rate further to 0.000001 produces a graph below, where the training and validation loss are both relatively constant and do not change. The validation loss line does not change and stays constant at 0.202. This tells us that for this model with this specific data, a learning rate of 0.00001 is the most ideal learning rate as it produces the lowest loss in predicting the coefficient of drag, which is our target.



Now, we can adjust the number of iterations and keep everything else constant (batch size 64 and learning rate 0.00001). When decreasing the number of iterations from 100,000 to 10,000, the training loss is more consistent, however, there is still a lot of noise and variance in the training loss. The validation loss is much higher, with the lowest validation loss at 0.336. When further decreasing the number of iterations to 1000, the trends stay the same but the lowest validation loss comes to 0.224. There is no maximum number of iterations, but 100,000 is pretty high, and an iteration size above this would be too time-consuming (~10 minutes). Testing at 1,000,000 iterations, the validation loss only comes down to a minimum of 0.199 which is quite inefficient.

After testing the model with various hyperparameters, we can conclude that for predicting the drag coefficient of vehicles,

the optimal hyperparameters that produce the lowest loss of 0.193 are a learning rate of 0.00001, a batch size of 64, and a number of iterations of 100000. This tells us that the model is fairly accurate in predicting the drag coefficient of vehicles. By being able to predict the coefficient of drag with just height, width, and frontal area, the model can be used by manufacturers and aerodynamicists during the early design stages of the car to "model all of the complex dependencies of drag on shape, inclination, and some flow conditions"[34].

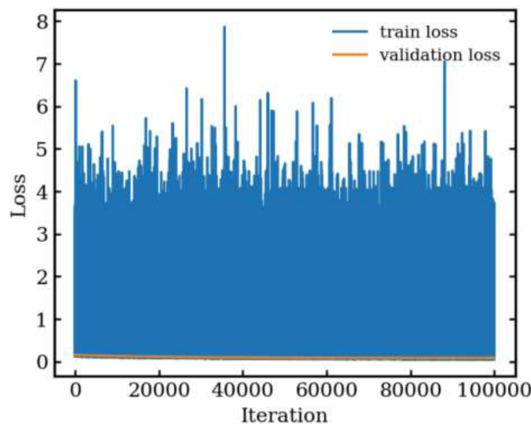
Second model

We can also test this model on completely different data; we can try to use population, GDP, and inflation to predict the yearly water used by a country. Below is a sample of the data for 4 out of 186 countries that will be used by the model.

Country	Yearly Water Used (m³, thousands of liters)	Population	GDP (nominal)	Inflation
Argentina	37780000000	45510318	632770000000	94.8
Belgium	6005000000	11655930	578604000000	9.6
India	761000000000	1417173173	3385090000000	6.7
New Zealand	5201000000	5185288	247234000000	7.2

All data is from 2022. Yearly Water Used (m³, thousands of liters) taken from Worldometer[35], Population and GDP (nominal) taken from Worldometer[36], and Inflation taken from WorldData[37].

After setting up the model in the same way, we can analyze the results. After testing for the optimal hyperparameters by using the same process when finding them for the coefficient of drag prediction, the optimal hyperparameters found are a learning rate of 0.0001, a batch size of 64, and a number of iterations of 100000, we receive the results below.



[34] Benson, T. *The Drag Coefficient*; n.d.
[35] Worldometer. *Water Use Statistics*; n.d.
[36] Worldometer. *GDP by Country*; n.d.
[37] WorldData. *Inflation rates in a global comparison*; n.d.

The training data for predicting the annual water consumption has much less consistency as compared to the training data for predicting the coefficient of drag. The range of results is much larger, generally ranging from 0.07 - 4.00. However, there are some very large spikes going all the way up to 6.5 and there does not seem to be much stabilization, and there is much more noise and variance as the loss keeps fluctuating. However, both the training loss and the validation loss decrease over time. Unlike the other model, the validation loss is higher than the minimum training loss, but the validation loss is much lower than that of the validation loss with the drag coefficient. There, the lowest validation loss was 0.193, however here the lowest validation loss is 0.0391. This is significantly lower, which tells us that the model can effectively predict with very little uncertainty the annual water consumption of a country given the factors of population, GDP, and inflation. The high accuracy in these predictions can help economists predict water consumption before official statistics are released, and help governments prepare policies and responses to the statistics.

CONCLUSION

The results show that the model can successfully predict the outcomes in two vastly different fields given factors not directly correlated to the outcome, although at different levels: the model can predict water consumption given some factors (population, GDP (nominal), and inflation) at a much higher accuracy and precision than predicting the coefficient of drag given some factors (height, weight, and frontal area). This could be due to various reasons, including a possibly more accurate data set (sources such as Worldmeter and WorldData are more accurate than sources such as EcoModder) and a greater feature relevance (features such as population are more directly linked to water consumption compared to features such as height to the coefficient of drag). The minimum loss when predicting the coefficient of drag is 0.193 which is nearly 5 times larger than the minimum loss when predicting the annual water consumption.

ACKNOWLEDGMENTS

Throughout the composition of this research review, I am extremely grateful to Mr Sandip Roy, who is currently pursuing a PhD in Physics from Princeton University, Jadwin Hall, Princeton NJ, USA. I am very thankful for his continued mentorship and support as I pursued this research from the Summer of 2023 until the Spring of 2024.

BIBLIOGRAPHY

1) Alake, R. (2023) *Loss Functions in Machine Learning Explained | DataCamp*. Available at: https://www.datacamp.com/tutorial/loss-function-in-machine-learning. (Accessed: 31 March 2024).
2) Arslan, E. (2023) *what does The Loss (Cost) Function mean in Deep Learning | Medium*. Available at: https://medium.com/@erhan_arslan/what-does-the-loss-cost-function-mean-in-deep-learning-71911d14f7a2#:~:text=A%20loss%20function%20quantifies%2

0how,predicted%20values%20and%20actual%20values. (Accessed: 31 March 2024).

3) Benson, T. (no date) *The Drag Coefficient | Glenn Research Center, Nasa.* Available at: https://www.grc.nasa.gov/www/k-12/VirtualAero/BottleRocket/airplane/dragco.html. (Accessed: 31 March 2024).
4) Bergmann, D. (2023) *What is semi-supervised learning? | IBM*. Available at: https://www.ibm.com/topics/semi-supervised-learning#:~:text=Semi%2Dsupervised%20learning%20is%20a,for%20classification%20and%20regression%20tasks. (Accessed: 31 March 2024).
5) Brown, S. (2021) *Machine Learning, explained | MIT Sloan*. Available at: https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained. (Accessed: 31 March 2024).
6) Brownlee, J. (2020) *How to Control the Stability of Training Neural Networks With the Batch Size | Machine Learning Mastery*. Available at: https://machinelearningmastery.com/how-to-control-the-speed-and-stability-of-training-neural-networks-with-gradient-descent-batch-size/. (Accessed: 31 March 2024).
7) Codecadamy Team. (no date) *Normalization | Codecademy*. Available at: https://www.codecademy.com/article/normalization. (Accessed: 31 March 2024).
8) *Drag Coefficient* (no date) | *Glenn Research Center, Nasa*. Available at: https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/drag-coefficient-2/#determining-value-for-drag-coefficient. (Accessed: 31 March 2024)
9) Harris, C.R., Millman, K.J., van der Walt, S.J. et al. (2020) *Array programming with NumPy | Nature 585, 357–362. DOI: 10.1038/s41586-020-2649-2.* Available at: https://www.nature.com/articles/s41586-020-2649-2. (Accessed: 31 March 2024).
10) Krishnamurthy, B. (2024) *An Introduction to the ReLU Activation Function | BuiltIn*. Available at: https://builtin.com/machine-learning/relu-activation-function. (Accessed: 31 March 2024).
11) Kumar, A. (2019) *Neural network | Medium*. Available at: https://towardsdatascience.com/neural-network-74f53424ba82. (Accessed: 31 March 2024).
12) Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. (2019) *PyTorch: An Imperative Style, High-Performance Deep Learning Library | Advances in Neural Information Processing Systems 32. Curran Associates, Inc.; 2019. p. 8024–35*. Available at: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf. (Accessed: 31 March 2024).
13) *Reducing loss: Gradient descent, Machine Learning* (no date) | *Google for Developers*, *Google*. Available at: https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent#:~:text=The%20gradient%20always%20points%20in,descent%20relies%20on%20negative%20gradients. (Accessed: 31 March 2024).
14) *Reducing Loss: Learning Rate, Machine Learning* (no date) | *Google for Developers, Google*. Available at: https://developers.google.com/machine-learning/crash-course/reducing-loss/learning-rate. (Accessed: 31 March 2024).
15) *ReLU activation function explained* (no date) | *Built In*. Available at: https://builtin.com/machine-learning/relu-activation-function#:~:text=The%20rectified%20linear%20unit%20(ReLU)%20or%20rectifier%20activation%20function%20introduces,activation%20functions%20in%20deep%20learning. (Accessed: 31 March 2024).
16) Seldon. (2023) *Machine learning optimization - why is it so important? | Seldon*. Available at: https://www.seldon.io/machine-learning-optimisation. (Accessed: 31 March 2024).
17) *Sequential* (no date) | *PyTorch*. Available at: https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html. (Accessed: 31 March 2024).
18) Statista. (2024) *Machine Learning - Worldwide | Statista.* Available at: https://www.statista.com/outlook/tmo/artificial-intelligence/machine-learning/worldwide (Accessed: 31 March 2024).
19) *SGD* (no date) | *PyTorch*. Available at: https://pytorch.org/docs/stable/generated/torch.optim.SGD.html. (Accessed: 31 March 2024).
20) Understanding Machine Learning (ML) (no date) | Alpine AI. Available at: https://alpineai.swiss/en/glossary/machine-learning-ml/. (Accessed: 31 March 2024).
21) Van Rossum G, Drake Jr FL. (1995) *Python reference manual | Centrum voor Wiskunde en Informatica Amsterdam*. Available at: https://ir.cwi.nl/pub/5008/05008D.pdf. (Accessed: 31 March 2024).
22) Vehicle Coefficient of Drag List (no date) | EcoModder. Available at: https://ecomodder.com/wiki/Vehicle_Coefficient_of_Drag_List. (Accessed: 31 March 2024).
23) What is supervised learning? (no date) | Google Cloud, Google. Available at: https://cloud.google.com/discover/what-is-supervised-learning#:~:text=Supervised%20learning%20is%20a%20category,the%20input%20and%20the%20outputs. (Accessed: 31 March 2024).
24) What is unsupervised learning? (no date) | Google Cloud, Google. Available at: https://cloud.google.com/discover/what-is-unsupervised-learning#:~:text=Unsupervised%20learning%20in%20artificial%20intelligence,any%20explicit%20guidance%20or%20instruction. (Accessed: 31 March 2024).
25) WorldData. (no date) *Inflation rates in a global comparison | WorldData*. Available at: https://www.worlddata.info/inflation.php. (Accessed: 31 March 2024).
26) Worldometer. (no date) *GDP by Country | Worldometer*. Available at: https://www.worldometers.info/gdp/gdp-by-country/. (Accessed: 31 March 2024).
27) Worldometer. (no date) *Water Use Statistics | Worldometer*. Available at: https://www.worldometers.info/water/ (Accessed: 31 March 2024).