Evaluating the Impact of Hyperparameters on a Neural Network's Performance

Adrian Qin

Introduction

In the realm of machine learning, particularly within the context of neural networks, the optimization of hyperparameters emerges as a critical factor influencing model efficacy and accuracy. This study is anchored in the burgeoning field of neural network research, a domain where subtle adjustments in hyperparameters can yield profound impacts on performance outcomes.

My experiment is meticulously designed to investigate the influence of specific hyperparameters on the performance of a basic neural network, utilizing the widely recognized MNIST dataset of handwritten digits as a testing ground. MNIST is a classic dataset in machine learning, consisting of grayscale images of handwritten digits (0 through 9). Each image is 28x28 pixels, and the goal is often to train a neural network to recognize and classify these images into the correct digit.

I have chosen to focus on four critical factors: Learning Rate, Batch Size, Number of Layers, and Activation Function. Each of these factors plays a fundamental role in neural network training and optimization, and they are listed in the Table 1.

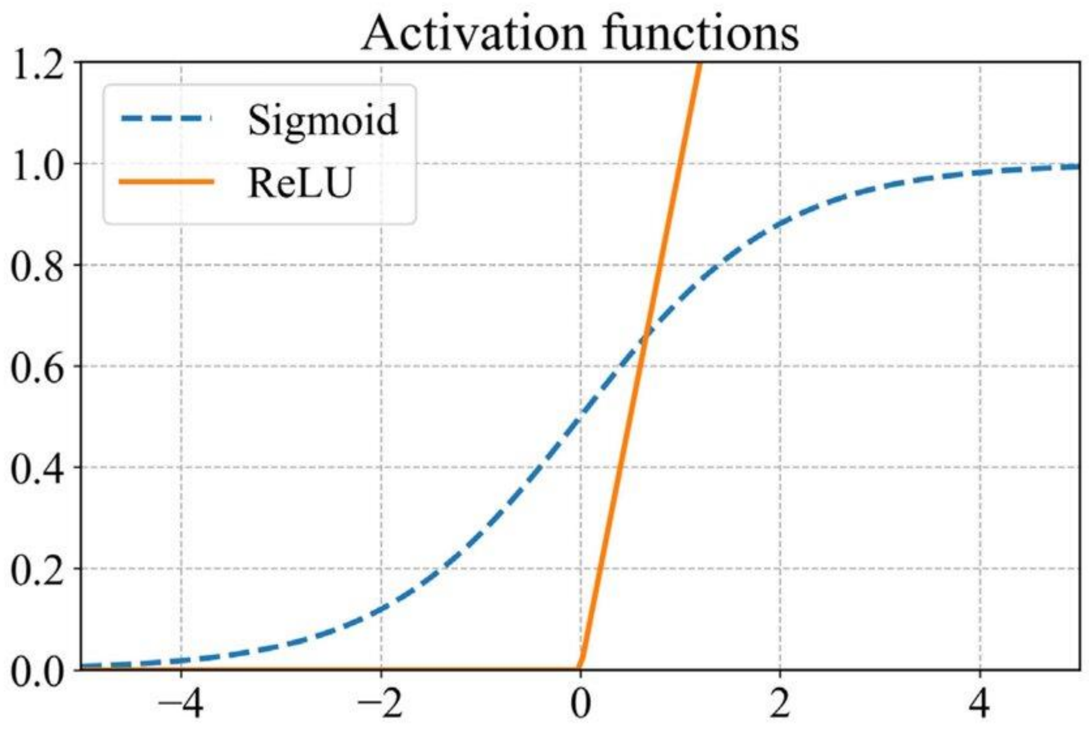
- The Learning Rate dictates the step size during the optimization process, influencing the convergence speed and stability of the training process. I choose the two standard learning rates 0.01 and 0.001 to see if higher learning rate result in better performance aligning with the straightforward nature of the MNIST dataset.

- Batch size refers to the number of training examples used in one iteration of model training. In other words, it's the number of data points that the neural network processes before updating the model parameters (weights and biases). The choice of batch size can affect the training dynamics. A smaller batch size often provides a regularizing effect and lower generalization error. However, too small a batch size may lead to instability in training, especially with noisy gradients. Conversely, a larger batch size allows for faster computation (thanks to better utilization of hardware optimizations) but might lead to a smoother optimization landscape, potentially leading to suboptimal solutions. So in this case, I select 32 and 128 which are commonly used as two levels of batch size.

- Layers are the basic units of computation. The number of layers in a network refers to how many of these computational units are stacked together. I use 2 layers as a level to represent shallow networks that only contains one hidden layer. The 6 layers level is used to represent deep networks which have multiple hidden layers and can model complex patterns in data. However, they may require more data and computational resources and can be harder to train.

- As for the activation functions, they are crucial for introducing non-linearity into the model, enabling it to learn and model complex data structures. I am going to focus on two specific activation functions, ReLU (Rectified Linear Unit) and sigmoid. The ReLU function is defined as $f(x) = max(0, x)$. It outputs the input directly if it is positive; otherwise, it outputs zero. The Sigmoid function is defined as $f(x) = \frac{1}{1+e^{-x}}$, and it maps the input values to a range between 0 and 1. Figure 1 illustrates the ReLU and Sigmoid activation functions.

Figure 1: Activation Functions



My response variable, the accuracy of the neural network on the MNIST dataset, is a direct measure of model performance, the high the accuracy, the better the performance. if we have a dataset with N instances, and the model makes correct predictions for C of those instances, then the accuracy is calculated as:

$$Accuracy = \frac{Number\ of\ Correct\ Predictions\ (C)}{Total\ Number\ of\ Predictions\ (N)}$$

The objective of this study is to statistically find how to increase the accuracy of neural networks working on the MNIST dataset and reduce the variation of y (= accuracy) caused by the test loss of computer.

Table 1: Factors and Levels

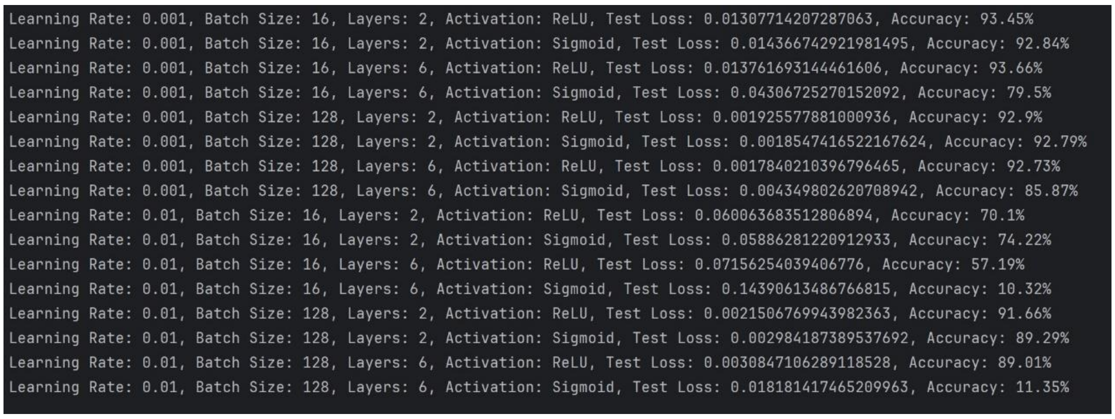
Factor	Level	
	-	+
Learning Rate	0.001	0.01
Batch Size	32	128
Number of Layers	2	8
Activation Function	Rectified Linear Unit	Sigmoid

Experimental design

Factorial effects on accuracy are analyzed by employing a Full Factorial Design at Two Levels, resulting in 16 experimental runs, allows for a comprehensive examination of the impact of these hyperparameters. The factors and their corresponding levels are detailed in Table 1. This approach facilitated the simultaneous analysis of two or more factor effects. With each factor set at two levels, the study focused on observing the linear influence of the response across the remaining levels of the factors. The results from this design yielded insights into the primary effects of each factor and the interaction effects among all possible factor combinations. Additionally, models for the response's location and dispersion provided further insights into the process

For data collection, I wrote the code in python to define the neural network class. After that the code trained each run in Table 2 in the training loop and test its accuracy. All the 16 runs are trained and tested in the same loop on one computer to mitigate any potential hidden effects. All 16 runs are printed together after the termination of the code, and there are four replicates in total. A sample output for the first replicate is showed in Figure 2.

Figure 2: Program Output



```
Learning Rate: 0.001, Batch Size: 16, Layers: 2, Activation: ReLU, Test Loss: 0.01307714207287063, Accuracy: 93.45%
Learning Rate: 0.001, Batch Size: 16, Layers: 2, Activation: Sigmoid, Test Loss: 0.014366742921981495, Accuracy: 92.84%
Learning Rate: 0.001, Batch Size: 16, Layers: 6, Activation: ReLU, Test Loss: 0.013761693144461606, Accuracy: 93.66%
Learning Rate: 0.001, Batch Size: 16, Layers: 6, Activation: Sigmoid, Test Loss: 0.04306725270152092, Accuracy: 79.5%
Learning Rate: 0.001, Batch Size: 128, Layers: 2, Activation: ReLU, Test Loss: 0.001925577881000936, Accuracy: 92.9%
Learning Rate: 0.001, Batch Size: 128, Layers: 2, Activation: Sigmoid, Test Loss: 0.0018547416522167624, Accuracy: 92.79%
Learning Rate: 0.001, Batch Size: 128, Layers: 6, Activation: ReLU, Test Loss: 0.0017840210396796465, Accuracy: 92.73%
Learning Rate: 0.001, Batch Size: 128, Layers: 6, Activation: Sigmoid, Test Loss: 0.004349802620708942, Accuracy: 85.87%
Learning Rate: 0.01, Batch Size: 16, Layers: 2, Activation: ReLU, Test Loss: 0.060063683512806894, Accuracy: 70.1%
Learning Rate: 0.01, Batch Size: 16, Layers: 2, Activation: Sigmoid, Test Loss: 0.05886281220912933, Accuracy: 74.22%
Learning Rate: 0.01, Batch Size: 16, Layers: 6, Activation: ReLU, Test Loss: 0.07156254039406776, Accuracy: 57.19%
Learning Rate: 0.01, Batch Size: 16, Layers: 6, Activation: Sigmoid, Test Loss: 0.14390613486766815, Accuracy: 10.32%
Learning Rate: 0.01, Batch Size: 128, Layers: 2, Activation: ReLU, Test Loss: 0.0021506769943982363, Accuracy: 91.66%
Learning Rate: 0.01, Batch Size: 128, Layers: 2, Activation: Sigmoid, Test Loss: 0.002984187389537692, Accuracy: 89.29%
Learning Rate: 0.01, Batch Size: 128, Layers: 6, Activation: ReLU, Test Loss: 0.0030847106289118528, Accuracy: 89.01%
Learning Rate: 0.01, Batch Size: 128, Layers: 6, Activation: Sigmoid, Test Loss: 0.018181417465209963, Accuracy: 11.35%
```

Table 2: Planning Matrix

Run	Activation Function	Learning Rate	Batch Size	Number of Layers
1	ReLU	0.001	Small	Low
2	ReLU	0.001	Small	High
3	ReLU	0.001	Large	Low
4	ReLU	0.001	Large	High
5	ReLU	0.01	Small	Low
6	ReLU	0.01	Small	High
7	ReLU	0.01	Large	Low
8	ReLU	0.01	Large	High
9	Sigmoid	0.001	Small	Low
10	Sigmoid	0.001	Small	High
11	Sigmoid	0.001	Large	Low
12	Sigmoid	0.001	Large	High
13	Sigmoid	0.01	Small	Low
14	Sigmoid	0.01	Small	High
15	Sigmoid	0.01	Large	Low
16	Sigmoid	0.01	Large	High

In Table 2, I ensure that each factor level appears the same number of times in the design to keep the balance. For any pair of factors, each possible level combination appears the same number of times in the design aligning with its orthogonality.

Methodology

To achieve the two experimental objectives which are increasing the accuracy (y) reducing the variation of y, the mean (for location) and variance (for dispersion) need to be modeled separately as functions of the experimental factors, and they are shown in Equation 1.

$$\bar{y_i} = \frac{1}{n_i}\sum_{j=1}^{n_i} y_{ij}, \qquad s_i^2 = \frac{1}{n_i - 1}\sum_{j=1}^{n_i}(y_{ij} - \bar{y_i})^2, \qquad and \qquad n_i = 4. \quad (1)$$

In this research, the main effects and interaction effects between 2 and more factors are needed for $\bar{y}$ and logarithmic s^2. With respect to factors A and B, Equation 2 and 3 shows the equation used for calculating the main effects and interaction effects for individual factor and between two factors.

$$ME(A) = \bar{z}(A+) - \bar{z}(A-). \quad (2)$$

$$INT(A,B) = \frac{1}{2}\{ME(A|B+) - ME(A|B-)\}. \quad (3)$$

Table 3: Design Matrix and Accuracy Data

Run	Factor				Accuracy				$\bar{y}$	s^2	$\ln s^2$
	A	B	C	D							
1	-	-	-	-	93.45	93.74	94.49	94.77	94.11	0.38	-0.96
2	-	-	-	+	92.84	94.43	94.48	94.39	94.04	0.64	-0.45
3	-	-	+	-	93.66	94.28	93.83	94.08	93.96	0.07	-2.60
4	-	-	+	+	79.50	89.36	88.36	88.81	86.51	21.99	3.09
5	-	+	-	-	92.90	94.05	93.29	93.74	93.50	0.25	-1.37
6	-	+	-	+	92.79	93.16	92.78	91.89	92.66	0.29	-1.23
7	-	+	+	-	92.73	92.04	91.89	92.25	92.23	0.13	-2.01
8	-	+	+	+	85.87	57.20	60.41	85.59	72.27	243.38	5.49
9	+	-	-	-	70.10	80.97	88.02	74.06	78.29	62.27	4.13
10	+	-	-	+	74.22	74.53	70.06	82.02	75.21	24.78	3.21
11	+	-	+	-	57.19	54.88	62.00	54.62	57.17	11.69	2.45
12	+	-	+	+	10.32	9.74	10.28	10.10	10.11	0.07	-2.66
13	+	+	-	-	91.66	87.27	90.50	90.83	90.07	3.71	1.31
14	+	+	-	+	89.29	89.37	88.99	88.16	88.95	0.31	-1.18
15	+	+	+	-	89.01	92.31	88.81	89.20	89.83	2.75	1.01
16	+	+	+	+	11.35	10.09	10.10	11.35	10.72	0.53	-0.64

Analysis

After getting all the four columns of accuracy with four replicates, the $\bar{y}$, s^2, and $ln\, s^2$ are calculated with R language for each run. With the equations provided for calculating main effects and interaction effects, the $\bar{y}$ and $ln\, s^2$ for different factors are also calculated. These operations make me able to get the two significant tables --- Table 3 and 4.

For Table 3, the four vectors learning rate, batch size, number of layers and activation functions that we are interested are represented respectfully by letter A, B, C and D, which make it easier for the following analysis. Observing the $\bar{y}$ column, the highest average accuracy is produced by all the factors with negative signs, which means A, B and C are at lower level and D is the ReLU function. Similarly, the 8 runs with negative factor A have significantly higher $\bar{y}$ compared to the other situation. At the same time, runs #12 and #16 contributes the lowest average accuracy which are much lower than the rest runs. A and B are negative for run #12, and all the four factors are positive for run #16, which is a quite interesting outcome.

Table 4: Factorial Effects

Effect	$\bar{y}$	$\ln s^2$
A	-27.37	0.96
B	5.10	-0.60
C	-24.25	0.08
D	-19.84	0.46
AB	-22.27	0.36
AC	-51.62	1.04
AD	-47.20	1.42
BC	-19.15	-0.52
BD	-14.73	-0.15
CD	-44.09	0.54
ABC	-43.80	0.89
ABD	-44.20	0.87
ACD	-83.39	-0.49
BCD	-44.71	0.84
ABCD	-83.39	0.32

The factorial effects analysis indicates that the learning rate (A) has the most substantial impact on the mean accuracy. The learning rate (A) was anticipated to exert a larger influence on the accuracy, and it aligns with my hypothesis. So, it is the most significant main effect. However, although a learning rate of 0.001 has a larger absolute value of $\bar{y}$, the stability which is represented by $ln\, s^2$ is not ideal. However, the effect C is a relatively stable choice with relatively smaller influence on the mean accuracy. The batch size is the least significant, and it is not surprising since the batch size is actually considered not determinant, simultaneously, it is within the commonly used range. Among all the interaction effects, effects ACD and ABCD have the same impact on $\bar{y}$, and they are nearly twice as the third one, so we can say that both ACD and ABCD are significant, at the same time, the logarithmic variance is relatively small which brings confidence to declare the significance of the two effects. BD in all the instruction effects has the smallest impact on accuracy.

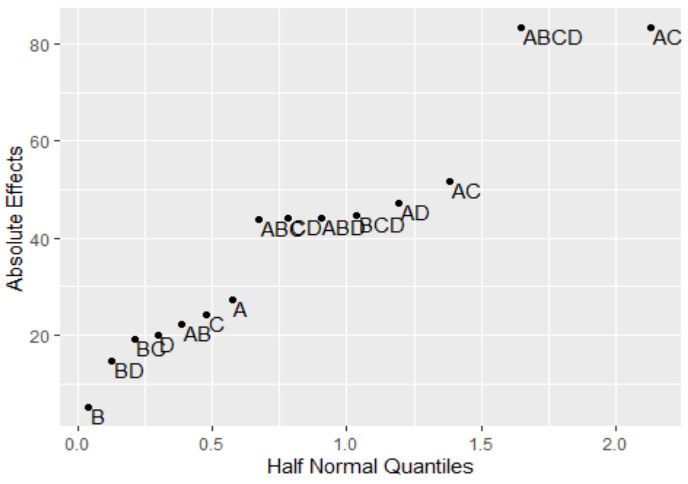


Figure 3: Location Effects

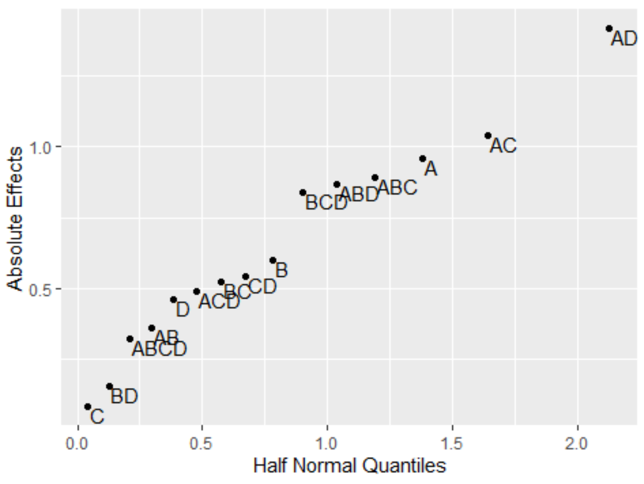
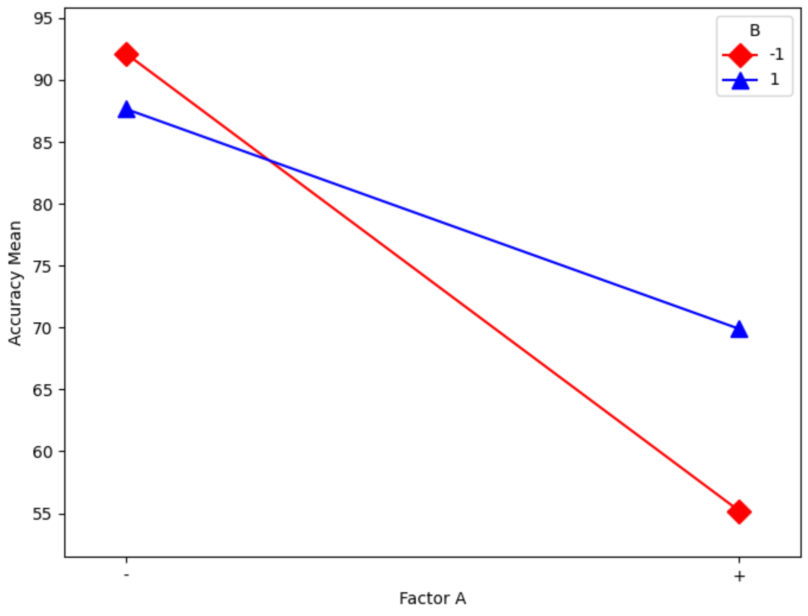


Figure 4: Location Effect

The half-normal plots of factorial effects estimated from $\bar{y}$; and $\ln s^2$ can be used to identify what factors influence location as measured by $\bar{y}$; and what factors influence dispersion as measured by s^2. The half-normal plots for $\bar{y}$; and $\ln s^2$ are displayed in Figures 3 and 4, respectively. It is clear that the interaction AD (learning rate and activation function) is the single most influential factor for $\ln s^2$ from Figure 3. As for Figure 4, it displays that both interaction ABCD and ACD are influential for $\bar{y}$. Based on the sign of the AD effect, the learning rate needs to be changed from 0.01 (level +) to 0.001 (level -) to reduce dispersion, and we also need to change the activation function from Sigmoid to ReLU at the same time.

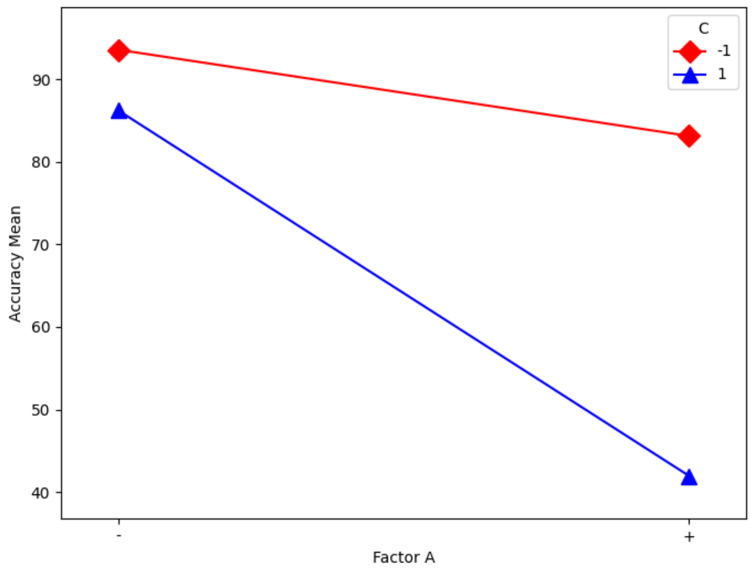
Figure 5: Interaction Plot of A and B



For the interaction plots, the blue line always represents a positive sign, and a red line represents a negative sign. In Figure 5, when the learning rate changes from 0.001 to 0.01, accuracy drops significantly no matter what the batch size is. This phenomenon has a theoretical support that for a higher rate the model will converge quickly, and a small learning rate can stick in local minima especially for a dataset like MNIST. For MNIST, the advantage of 0.001 learning rate is prominent.

When A is negative, smaller batch size is at first advantageous, however, as A goes to the positive side, larger batch size starts to have higher accuracy.

Figure 6: Interaction Plot of A and C



In Figure 6, the overall trend for learning rate lowering is similar to the Figure 5, but this time there is a difference in the slope of the line. When number of layers is large, accuracy drops quickly when switching from a learning rate of 0.001 to 0.01. When number of layers is 2, the accuracy does not drop much from 0.001 to 0.01. As for factor C itself, the difference between more and less layers is enlarged as the learning rate lowers, and less layers have a better performance on accuracy. This is a unexpected result since deep networks theoretically have a better capability handling complex patterns in data. However, a neural network does not require an excessively deep architecture to achieve high accuracy for MNIST, so there is an exemption.

Figure 7: Interaction Plot of A and D

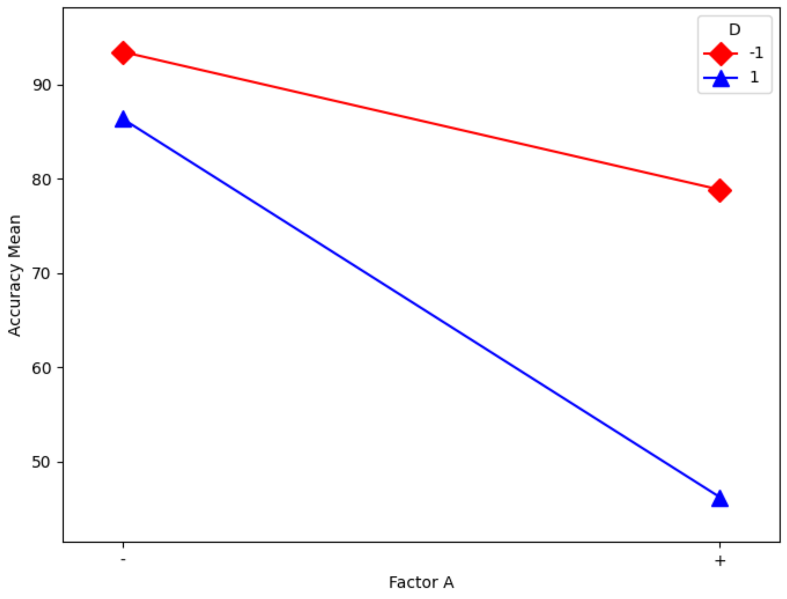


Figure 7 has a similar trend and analysis with Figure 6, ReLU function performs absolutely better than Sigmoid function, and the gap is exaggerated by the lowering of learning rate according to Figure 7.

Figure 8: Interaction Plot of B and C

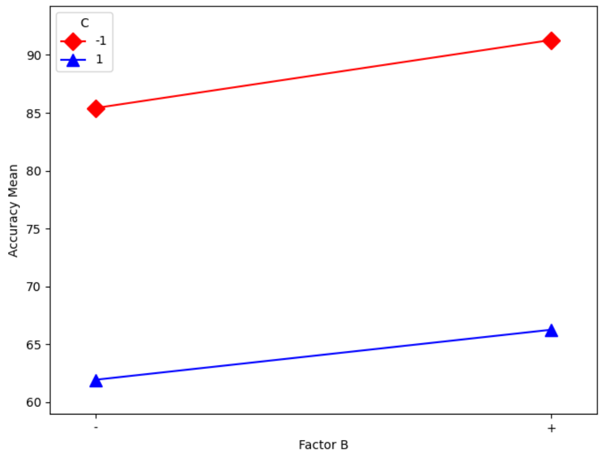
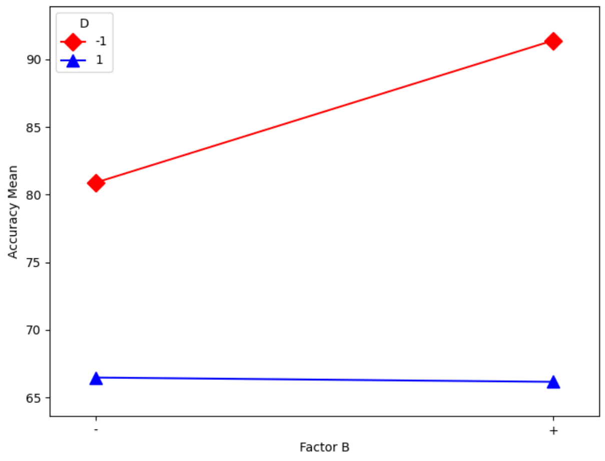
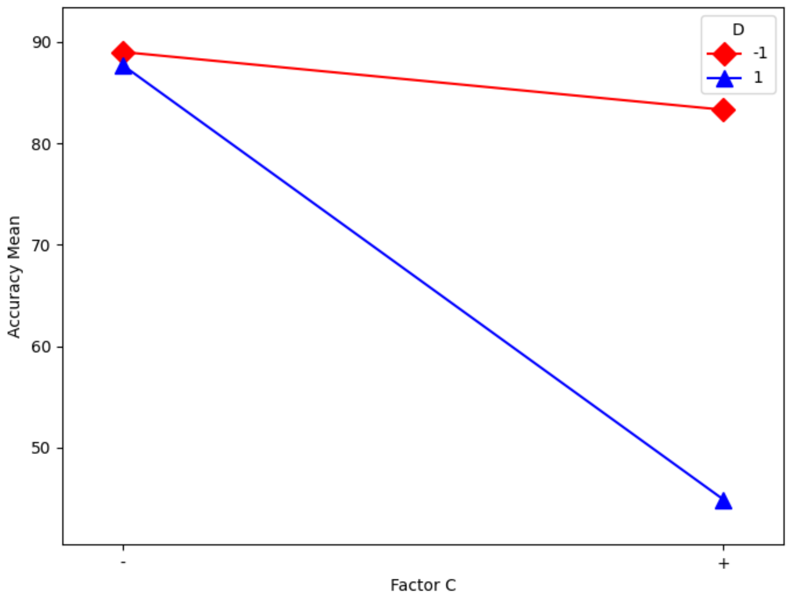


Figure 9: Interaction Plot of B and D



In Figure 8 and 9, the larger batch size generally increases the accuracy. The increase of number of layers and the switching from ReLU to Sigmoid clearly lose accuracy.

Figure 10: Interaction Plot of C and D



In Figure 10, the accuracy is pretty much the same at a high level when the number of layers is small, as the increase of number of layers, the difference between the performance of ReLU and Sigmoid enlarges. For ReLU, the accuracy is still retained at a relatively higher level, but the situation of Sigmoid function is not that satisfying. This result aligns with the property of Rectified Linear Unit function and Sigmoid function. ReLU is particularly beneficial for deeper networks due to its properties that help in mitigating vanishing gradients. Sigmoid can lead to the vanishing gradient problem, which can hinder the training of the model in deeper networks.

Conclusion

Since the first day I learned about the importance of hyperparameters on machine learning, I am curious about whether there is an optimistic solution for the setting of hyperparameters for all the environments. However, when the situation changed, the settings of hyperparameters are needed to be altered to accommodate new models and data. There seems always to be trade-off between efficiency and accuracy, precision and complexity. What this paper mainly talked about is to evaluate the impact of four hyperparameters on neural networks with MNIST dataset through a statistical modelling process. By doing the Full Factorial Design at Two Levels with detailed analysis, the property of neural networks and MNIST dataset is examined. Learning rate and number of layers are taken to be the two significant main effects on accuracy, with the instability of learning rate during training process, the number of layers may be the more influential one. The batch size is relatively not important, and the other three hyperparameters are more determinant in the case of MNIST dataset. In this experiment, I also found the interactions between different hyperparameters can be very interesting, and their inter-changing reveals some of the thorough properties of neural networks.

This experiment is a complete study in the sense of Full Factorial Design. However, there are always more things to do. When evaluating the location and dispersion effects, the experiments refer to a visual approach for better intuitionistic understanding, but I did not employ the numerical approach to calculate the exact number following regression equations. In the future, I will conduct more experiments for more activation functions and other factors related to get a better sense of fine – tuned hyperparameters in the field of machine learning.

Appendix

Neural Network Code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

class NeuralNetwork(nn.Module):
    def __init__(self, num_layers, activation_func):
        super(NeuralNetwork, self).__init__()
        layers = []

        layers.append(nn.Linear(784, 128))

        for _ in range(num_layers - 1):
            layers.append(activation_func())
            layers.append(nn.Linear(128, 128))

        layers.append(nn.Linear(128, 10))

        self.layers = nn.Sequential(*layers)

    def forward(self, x):
        x = x.view(x.size(0), -1)    # Flatten the input
        return self.layers(x)

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

learning_rates = [0.001, 0.01]
batch_sizes = [16, 128]
num_layers = [2, 6]
activations = [nn.ReLU, nn.Sigmoid]

results = []

for lr in learning_rates:
    for batch_size in batch_sizes:
        for layers in num_layers:
            for activation in activations:
                model = NeuralNetwork(layers, activation)

                criterion = nn.CrossEntropyLoss()
                optimizer = optim.Adam(model.parameters(), lr=lr)

                train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)

                model.train()
                for batch_idx, (data, target) in enumerate(train_loader):
                    optimizer.zero_grad()
                    output = model(data)
                    loss = criterion(output, target)
                    loss.backward()
                    optimizer.step()

                    if batch_idx % 100 == 0:
                        print(f"Train Epoch: {lr}-{batch_size}-{layers}-
{activation.__name__} [{batch_idx * len(data)}/{len(train_loader.dataset)} ({100. *
batch_idx / len(train_loader):.0f}%)]\tLoss: {loss.item():.6f}")

                test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)
                model.eval()
                test_loss = 0
                correct = 0
                with torch.no_grad():
                    for data, target in test_loader:
                        output = model(data)
                        test_loss += criterion(output, target).item()
                        pred = output.argmax(dim=1, keepdim=True)
                        correct += pred.eq(target.view_as(pred)).sum().item()

                test_loss /= len(test_loader.dataset)
                test_accuracy = 100. * correct / len(test_loader.dataset)

                results.append((lr, batch_size, layers, activation.__name__,
test_loss, test_accuracy))
for result in results:
    print(f"Learning Rate: {result[0]}, Batch Size: {result[1]}, Layers: {result[2]},
Activation: {result[3]}, Test Loss: {result[4]}, Accuracy: {result[5]}%")
```

R for factorial effects

```
factor_design <- data.frame(
    A = c(-1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1),
    B = c(-1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1),
    C = c(-1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1 ,1 ,1),
    D = c(-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1) )

factor_design$Mean_Accuracy <- c(-0.96, -0.45, -2.60, 3.09, -1.37, -
1.23, -2.01, 5.49, 4.13, 3.21, 2.45, -2.66, 1.31, -1.18
, 1.01, -0.64)

effects <- with(factor_design, c(
    A = mean(Mean_Accuracy[A == 1]) - mean(Mean_Accuracy[A == -1]),
    B = mean(Mean_Accuracy[B == 1]) - mean(Mean_Accuracy[B == -1]),
    C = mean(Mean_Accuracy[C == 1]) - mean(Mean_Accuracy[C == -1]),
    D = mean(Mean_Accuracy[D == 1]) - mean(Mean_Accuracy[D == -1]),
    AB = mean(Mean_Accuracy[A == 1 & B == 1]) - mean(Mean_Accuracy[A=
= -1 & B == -1]),
    AC = mean(Mean_Accuracy[A == 1 & C == 1]) - mean(Mean_Accuracy[A
== -1 & C == -1]),
    AD = mean(Mean_Accuracy[A == 1 & D == 1]) - mean(Mean_Accuracy[A
== -1 & D == -1]),
    BC = mean(Mean_Accuracy[B == 1 & C == 1]) - mean(Mean_Accuracy[B
== -1 & C == -1]),
    BD = mean(Mean_Accuracy[B == 1 & D == 1]) - mean(Mean_Accuracy[B
== -1 & D == -1]),
    CD = mean(Mean_Accuracy[C == 1 & D == 1]) - mean(Mean_Accuracy[C
== -1 & D == -1]),
    ABC = mean(Mean_Accuracy[A == 1 & B == 1 & C == 1]) - mean(Mean_A
ccuracy[A == -1 & B == -1 & C == -1]),
    ABD = mean(Mean_Accuracy[A == 1 & B == 1 & D == 1]) - mean(Mean_A
ccuracy[A == -1 & B == -1 & D == -1]),
    ACD = mean(Mean_Accuracy[A == 1 & C == 1 & D == 1]) - mean(Mean_A
ccuracy[A == -1 & C == -1 & D == -1]),
    BCD = mean(Mean_Accuracy[B == 1 & C == 1 & D == 1]) - mean(Mean_A
ccuracy[B == -1 & C == -1 & D == -1]),
    ABCD = mean(Mean_Accuracy[A == 1 & B == 1 & C == 1 & D == 1]) - m
ean(Mean_Accuracy[A == -1 & B == -1 & C == -1 & D == -1])
))

print(effects)

##         A         B         C         D        AB        AC        AD        BC
##  0.95875 -0.60375  0.08375  0.45875  0.35500  1.04250  1.41750 -0.
52000
##       BD       CD      ABC      ABD      ACD      BCD     ABCD
## -0.14500  0.54250  0.89000  0.87000 -0.48500  0.84000  0.32000
```

R for Half Normal Plot

```
library(ggplot2)

## Warning: package 'ggplot2' was built under R version 4.3.2
```

```
effects <- c('A', 'B', 'C', 'D', 'AB', 'AC', 'AD', 'BC', 'BD', 'CD',
'ABC', 'ABD', 'ACD', 'BCD', 'ABCD')
y_bar <- c(-27.37, 5.10, -24.25, -19.84, -22.27, -51.62, -47.20, -
19.15, -14.73, -44.09, -43.80, -44.20, -83.39, -44.71, -83.39)
ln_s_squared <- c(0.96, -0.60, 0.08, 0.46, 0.36, 1.04, 1.42, -0.52, -
0.15, 0.54, 0.89, 0.87, -0.49, 0.84, 0.32)

df  <-  data.frame(Effect  =  effects,  abs_y_bar  =  abs(y_bar),
abs_ln_s_squared = abs(ln_s_squared))

create_half_normal_plot <- function(df, value_column, label_column,
title) {
  df <- df[order(-df[[value_column]]), ]
  df$theoretical <- abs(qnorm(ppoints(nrow(df)) / 2))
  ggplot(df, aes(x = theoretical, y = df[[value_column]])) +
    geom_point() +
    geom_text(aes(label = df[[label_column]]), vjust = 1, hjust = 0,
nudge_x = 0.01, check_overlap = FALSE) +
    ggtitle(title) +
    xlab("Half Normal Quantiles") +
    ylab("Absolute Effects")
}

effect_plot <- create_half_normal_plot(df, 'abs_y_bar', 'Effect', ' ')
ln_s_squared_plot <- create_half_normal_plot(df, 'abs_ln_s_squared',
'Effect', ' ')
```

Interaction plot with Python

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.graphics.factorplots import interaction_plot
from statsmodels.formula.api import ols

df = pd.DataFrame({
    'A': [-1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, 1, 1],
    'B': [-1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1],
    'C': [-1, -1, 1, 1, -1, -1, 1, 1, -1, -1, 1, 1, -1, -1 ,1 ,1],
    'D': [-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1],
    'Accuracy': [94.11, 94.04, 93.96, 86.51, 93.50, 92.66, 92.23, 72.27,
                 78.29, 75.21, 57.17, 10.11, 90.07, 88.95, 89.83, 10.72]
})

model = ols('Accuracy ~ A * B * C * D', data=df).fit()

def create_and_save_interaction_plot(df, factor_x, factor_trace, response, file_name):
    fig, ax = plt.subplots(figsize=(8, 6))
    interaction_plot(x=df[factor_x], trace=df[factor_trace], response=df[response],
ax=ax,
                     colors=['red', 'blue'], markers=['D', '^'], ms=10)
    ax.set_xlabel(f'Factor {factor_x}')
    ax.set_ylabel('Accuracy Mean')
    ax.set_xticks([-1, 1])
    ax.set_xticklabels(['-', '+'])
    plt.savefig(file_name)
    plt.close(fig)

create_and_save_interaction_plot(df, 'A', 'B', 'Accuracy', 'interaction_AB.png')
create_and_save_interaction_plot(df, 'A', 'C', 'Accuracy', 'interaction_AC.png')
create_and_save_interaction_plot(df, 'A', 'D', 'Accuracy', 'interaction_AD.png')
create_and_save_interaction_plot(df, 'B', 'C', 'Accuracy', 'interaction_BC.png')
create_and_save_interaction_plot(df, 'B', 'D', 'Accuracy', 'interaction_BD.png')
create_and_save_interaction_plot(df, 'C', 'D', 'Accuracy', 'interaction_CD.png')
```