Final Assignment of Optimisic Machine Learning

23343630
Jingpeng Chen

2024.04.26

Introduction

In this report, the topic is the analysis of the effect of small batch stochastic gradient descent (SGD) on overfitting. The experimental study done is to predict the customer's regional or channel information based on the user's purchase behaviour. The experimental study is done from the following perspectives:
The purpose of the research is that traditional machine learning tasks combined with traditional methods can not meet the satisfaction of the results, turn to deep learning combined with neural network models, but due to the impact of computational resources and model complexity, there will be an overfitting phenomenon, resulting in the final model generalisation ability is weak, and the results are also unsatisfactory. Further, started to do some model optimization, but the traditional stochastic gradient descent (GD), each time before updating the parameters to load the entire dataset, which is not only the training time long and the model is weakly expressive, especially in terms of unknown datasets, even if the training set performs well. Therefore, this research is important to elaborate the optimization algorithm by reducing the sample size, i.e., small batch stochastic gradient descent, to study its necessity and how it prevents the overfitting phenomenon as a means of regularization.
The main methods revolve around the balance of the dataset, and the adjustment of the model structure, such as: increasing or decreasing the number of layers or adjusting the size of the learning rate, this report, focuses on the study of the batch size of the critical factor regulation brought about by the prevention of overfitting, which is one of the hyperparameter optimization.
Of course, I also read some literature reviews and found that the current strategy to improve the performance of the model is also data enhancement techniques, integrated learning methods or complex network structure adjustment. In the following, I will show the research methodology, performance evaluation, some more critical overfitting problems and a summary of future scalable methods.

Research Methods

1. Model Selection

This model uses the Multilayer Perceptron (MLP) from the basic deep learning model, which is a simple feed-forward neural network that is better suited to the dataset I chose and the classification prediction task and can draw good conclusions by modelling complex relationships.

At the same time, this base model, which will have a high chance of overfitting, is suitable for the model optimization theme of this research, which can be further investigated for learning rate and optimizer tuning, grid structure tuning, regularization tuning, and most importantly, batch size tuning.

2. Dataset Selection

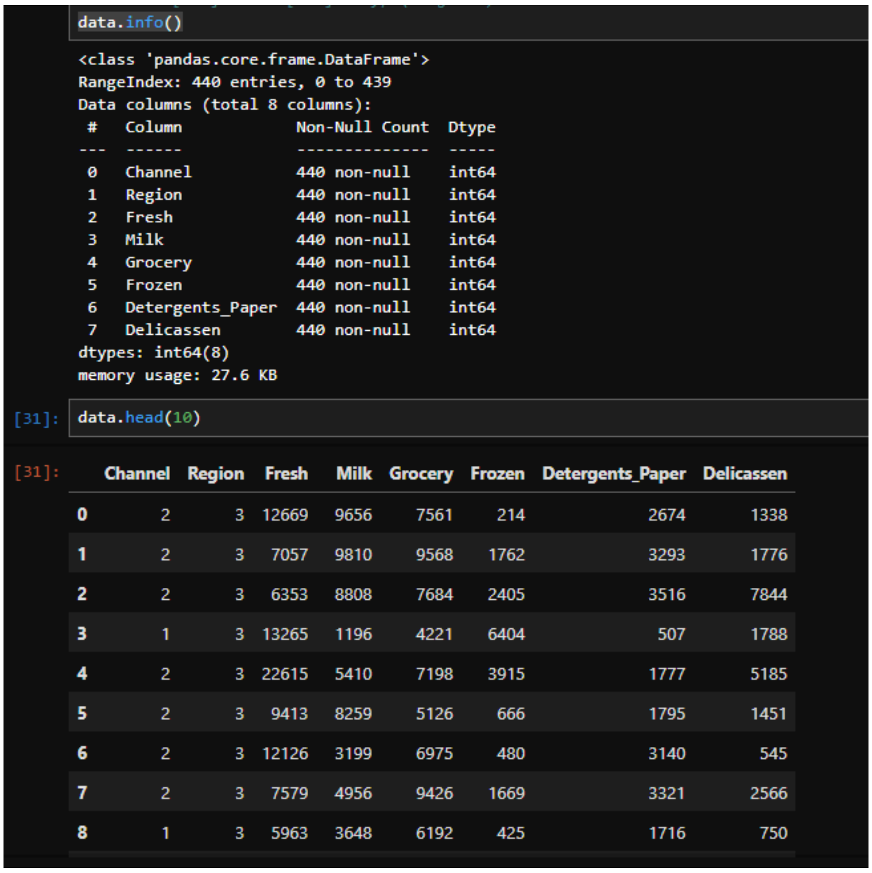
The selection of the dataset I considered according to the following aspects:

1) Firstly, it is based on the perspective of safe and authorizable problems, from the relevance of the project: it must be related to the research topic, I prefer classification prediction type tasks as well as adaptive recommender systems, and adaptive algorithms, and then, I combined with the frequent problems in retailing in the industrial world, how to predict the supply channel through the user's purchasing behaviour to achieve accurate recommendation marketing, I think this is more meaningful. So I retrieved this data source through a site search:
site:archive.ics.uci.edu/ml customer segmentation.

2) From the perspective of data size: I need to choose a small to medium data size data set, in this case, I can avoid too large a volume of data, easy to exist in the computational resource requirements of high, and long processing time of the interference factors, at the same time, but also to avoid too small a volume of data, can not illustrate the problem of overfitting.

3) Finally, I also considered the algorithms that I might need to use, such as datasets that support deep learning model algorithms, as well as datasets that evaluate optimization algorithms, and ruled out clustering tasks as well as datasets in the text labelling category, and then, chose this dataset in the numerical labelling category.

4) Next, I initially analyze the data source, which provides information about customers' purchasing behaviour (e.g., annual expenditures on fresh food, milk, groceries, etc.), as well as customer channels and regions. This dataset is clearly more suitable for customer segmentation, classification or clustering tasks. The goal of the task is to predict a customer's region or channel based on purchase behaviour. In this case, the model may be more susceptible to overfitting because of the complex relationships that may exist between the characteristics (annual spending) and the categorization goal (channel or region).



```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 440 entries, 0 to 439
Data columns (total 8 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Channel           440 non-null    int64
 1   Region            440 non-null    int64
 2   Fresh             440 non-null    int64
 3   Milk              440 non-null    int64
 4   Grocery           440 non-null    int64
 5   Frozen            440 non-null    int64
 6   Detergents_Paper  440 non-null    int64
 7   Delicassen        440 non-null    int64
dtypes: int64(8)
memory usage: 27.6 KB
```

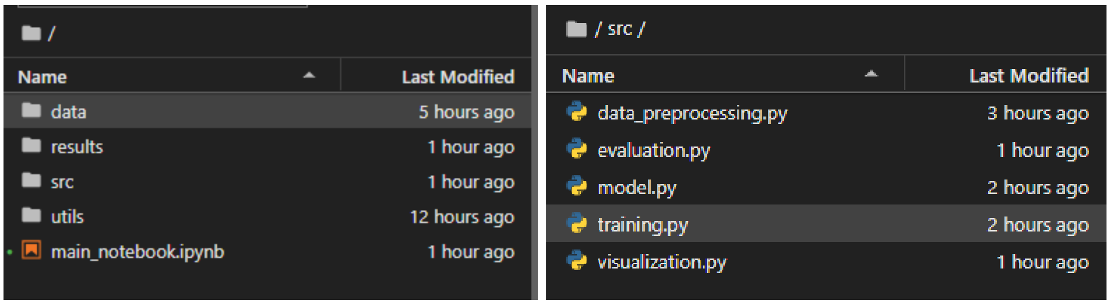
```
[31]: data.head(10)
```

[31]:

	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	2	3	12669	9656	7561	214	2674	1338
1	2	3	7057	9810	9568	1762	3293	1776
2	2	3	6353	8808	7684	2405	3516	7844
3	1	3	13265	1196	4221	6404	507	1788
4	2	3	22615	5410	7198	3915	1777	5185
5	2	3	9413	8259	5126	666	1795	1451
6	2	3	12126	3199	6975	480	3140	545
7	2	3	7579	4956	9426	1669	3321	2566
8	1	3	5963	3648	6192	425	1716	750

3. Experimental Design

The structure of the entire experimental program is shown:



/

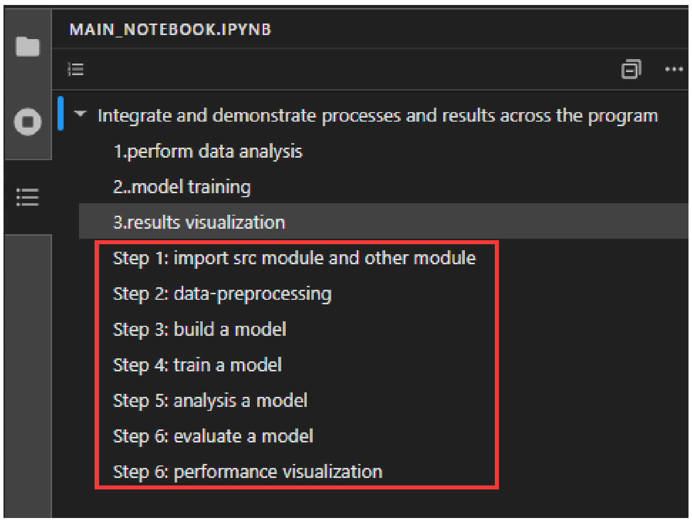
Name	Last Modified
data	5 hours ago
results	1 hour ago
src	1 hour ago
utils	12 hours ago
main_notebook.ipynb	1 hour ago

/ src /

Name	Last Modified
data_preprocessing.py	3 hours ago
evaluation.py	1 hour ago
model.py	2 hours ago
training.py	2 hours ago
visualization.py	1 hour ago

The main_notebook file is to integrate the process and results of the whole project, which mainly performs data analysis, model training and result visualization. src is the source file part, which has five main parts, the first one is the data_preprocessing part, which mainly does the data preprocessing, the second one is to do the definition and training of the model, the third one is the model training process, the fourth one is to evaluate the model, and finally the visualization and analysis of the model results. Then, the results file mainly contains the graphical results of the model running and the reports generated by the model evaluation, and the utils file mainly contains some optimization functions and auxiliary functions.

The flow of the entire project is as follows:



MAIN_NOTEBOOK.IPYNB

Integrate and demonstrate processes and results across the program

1.perform data analysis

2..model training

3.results visualization

Step 1: import src module and other module

Step 2: data-preprocessing

Step 3: build a model

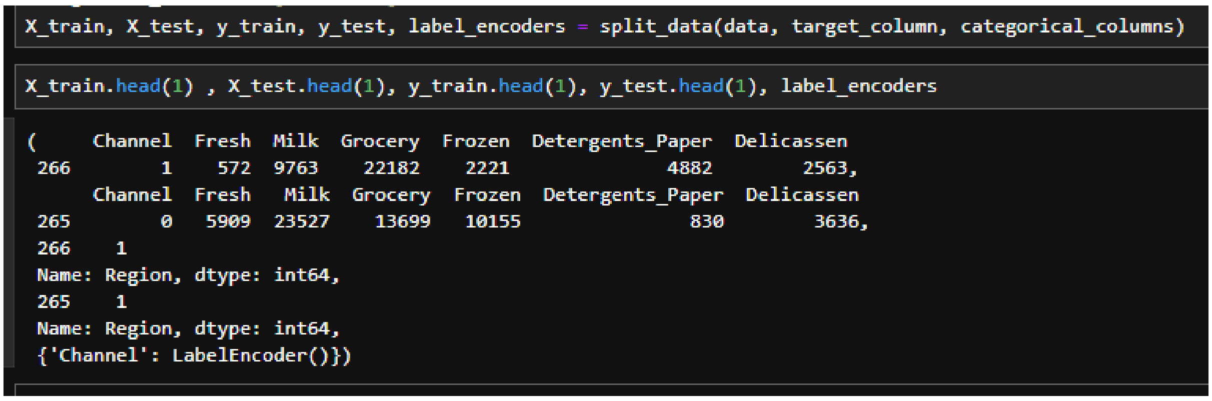
Step 4: train a model

Step 5: analysis a model

Step 6: evaluate a model

Step 6: performance visualization

In this case, the optimization of the model is mainly in the model and training files, and secondly, in the optimization file in the utils file, because, the model tuning may involve the tuning of the model creation itself, such as the tuning of the number of layers and neurons in the structure of the neural network, or the tuning of the learning rate and the optimizer, or the tuning of the batch size, and the regularization adjustments.

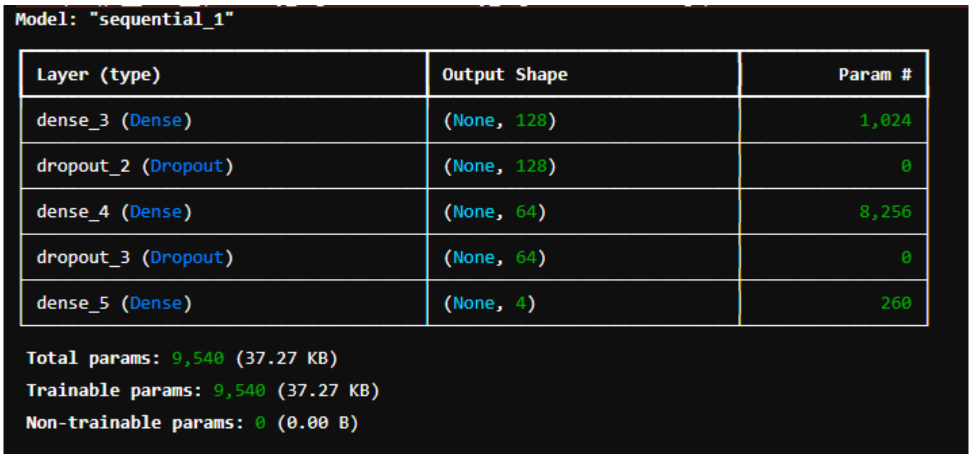


```
X_train, X_test, y_train, y_test, label_encoders = split_data(data, target_column, categorical_columns)
```

```
X_train.head(1) , X_test.head(1), y_train.head(1), y_test.head(1), label_encoders
```

```
(     Channel  Fresh  Milk  Grocery  Frozen  Detergents_Paper  Delicassen
 266        1    572  9763    22182    2221              4882        2563,
      Channel  Fresh   Milk  Grocery  Frozen  Detergents_Paper  Delicassen
 265        0   5909  23527    13699   10155               830        3636,
 266    1
 Name: Region, dtype: int64,
 265    1
 Name: Region, dtype: int64,
 {'Channel': LabelEncoder()})
```

In this case, the dataset splitting is mainly performed according to: 0.2, while in the data preprocessing section, converters for numerical and categorical features are defined, which can be used to fill in the missing values with the median as well as the unique heat coding. Of course, splitting the dataset is done to separate features and target variables.

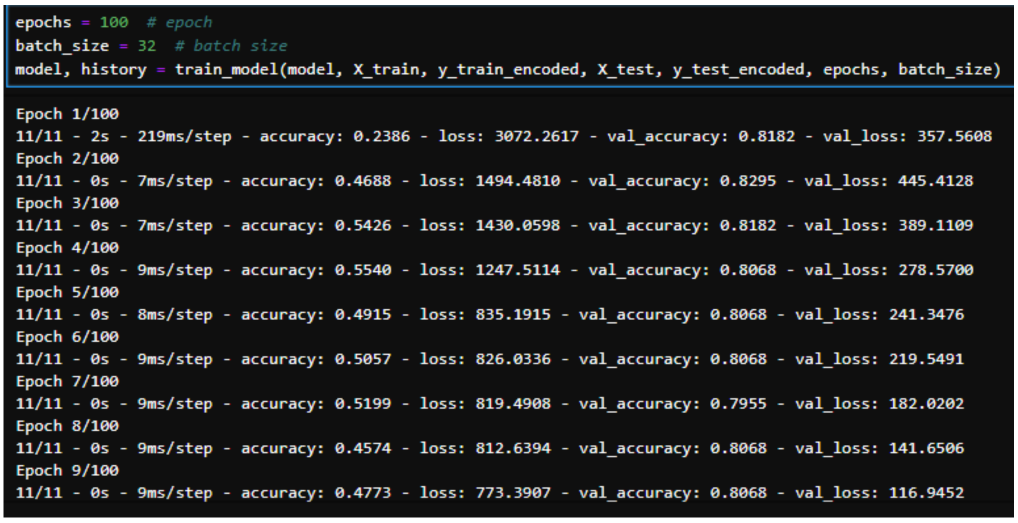


Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(None, 128)	1,024
dropout_2 (Dropout)	(None, 128)	0
dense_4 (Dense)	(None, 64)	8,256
dropout_3 (Dropout)	(None, 64)	0
dense_5 (Dense)	(None, 4)	260

```
Total params: 9,540 (37.27 KB)
Trainable params: 9,540 (37.27 KB)
Non-trainable params: 0 (0.00 B)
```

After the creation of the model used here, the structure of the model was examined, and the ‘Keras’ model of ‘TensorFlow’ was used here.



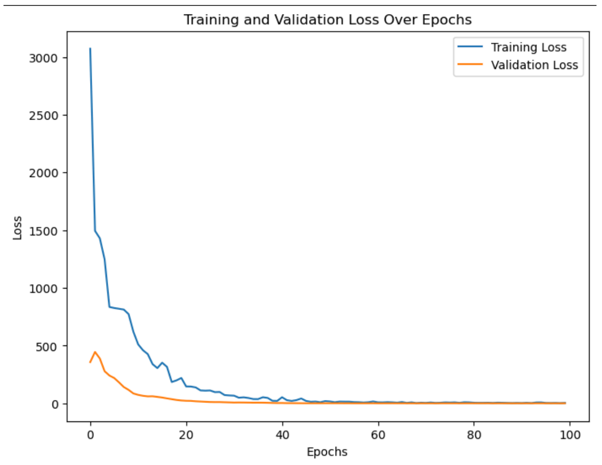
```
epochs = 100  # epoch
batch_size = 32  # batch size
model, history = train_model(model, X_train, y_train_encoded, X_test, y_test_encoded, epochs, batch_size)
```

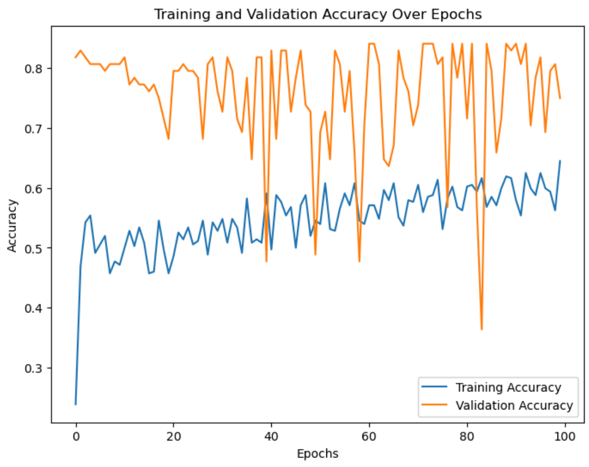
```
Epoch 1/100
11/11 - 2s - 219ms/step - accuracy: 0.2386 - loss: 3072.2617 - val_accuracy: 0.8182 - val_loss: 357.5608
Epoch 2/100
11/11 - 0s - 7ms/step - accuracy: 0.4688 - loss: 1494.4810 - val_accuracy: 0.8295 - val_loss: 445.4128
Epoch 3/100
11/11 - 0s - 7ms/step - accuracy: 0.5426 - loss: 1430.0598 - val_accuracy: 0.8182 - val_loss: 389.1109
Epoch 4/100
11/11 - 0s - 9ms/step - accuracy: 0.5540 - loss: 1247.5114 - val_accuracy: 0.8068 - val_loss: 278.5700
Epoch 5/100
11/11 - 0s - 8ms/step - accuracy: 0.4915 - loss: 835.1915 - val_accuracy: 0.8068 - val_loss: 241.3476
Epoch 6/100
11/11 - 0s - 9ms/step - accuracy: 0.5057 - loss: 826.0336 - val_accuracy: 0.8068 - val_loss: 219.5491
Epoch 7/100
11/11 - 0s - 9ms/step - accuracy: 0.5199 - loss: 819.4908 - val_accuracy: 0.7955 - val_loss: 182.0202
Epoch 8/100
11/11 - 0s - 9ms/step - accuracy: 0.4574 - loss: 812.6394 - val_accuracy: 0.8068 - val_loss: 141.6506
Epoch 9/100
11/11 - 0s - 9ms/step - accuracy: 0.4773 - loss: 773.3907 - val_accuracy: 0.8068 - val_loss: 116.9452
```

The initial batch size defined here is 32 and the initial value of epochs is 100.

Evaluation and Presentation

1. Model Performance Analysis

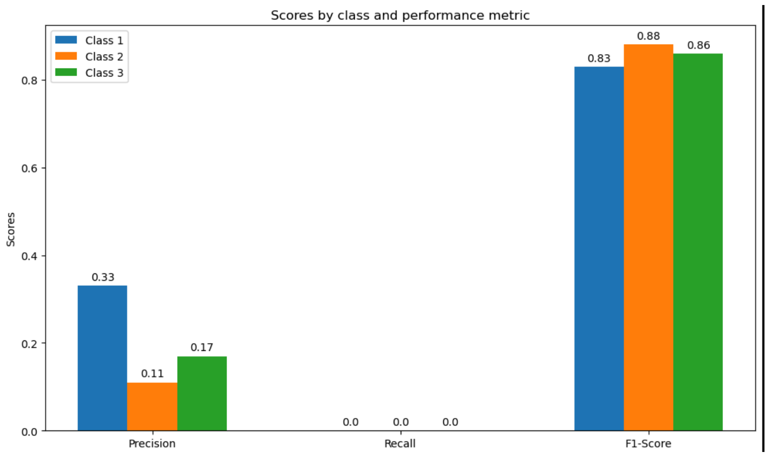


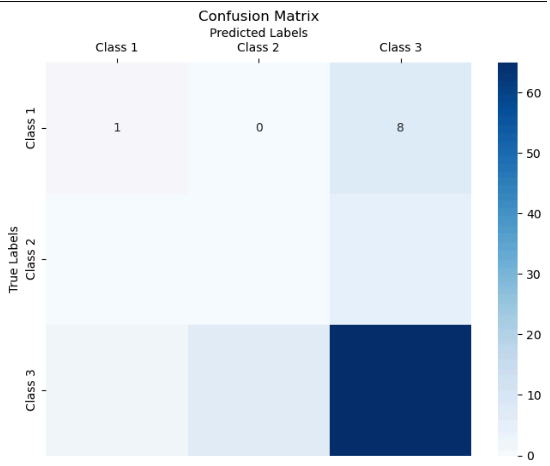


1) Analyze based on the loss curve:
Training loss decreases: training loss decreases as the ‘epoch’ increases, which is a good sign that the model is learning.
Validation loss decreases: validation loss also decreases, indicating that the model has some predictive ability on unseen data.
The difference in loss: the difference between training loss and validation loss is small, meaning that the overfitting is not serious.

2) Analyzed based on the accuracy curve:
Accuracy fluctuations: large fluctuations in validation accuracy may be due to the model's erratic performance on certain batches of data.
Signs of overfitting: accuracy fluctuations may also indicate that the model is overfitting a specific portion of the training data.

2. Assessment of Indicators

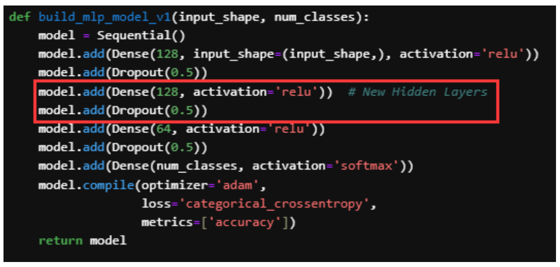




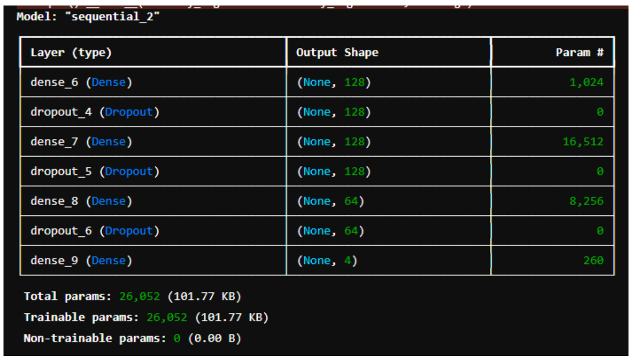
As can be seen from both graphs: the precision, recall and F1 scores for category 2 are all 0. This means that the model did not correctly predict any of the samples from Category 2, or all the samples predicted as Category 2 were wrong.
The precision, recall and F1 scores for Categories 1 and 3 are higher than those for Category 2, with Category 3 being the best performer.
Improvements in model performance are needed from network structure tuning (in the model creation file), learning rate, optimizer, and batch size tuning (in the model training file), and regularization tuning (in the model training file), respectively.

Model Optimization

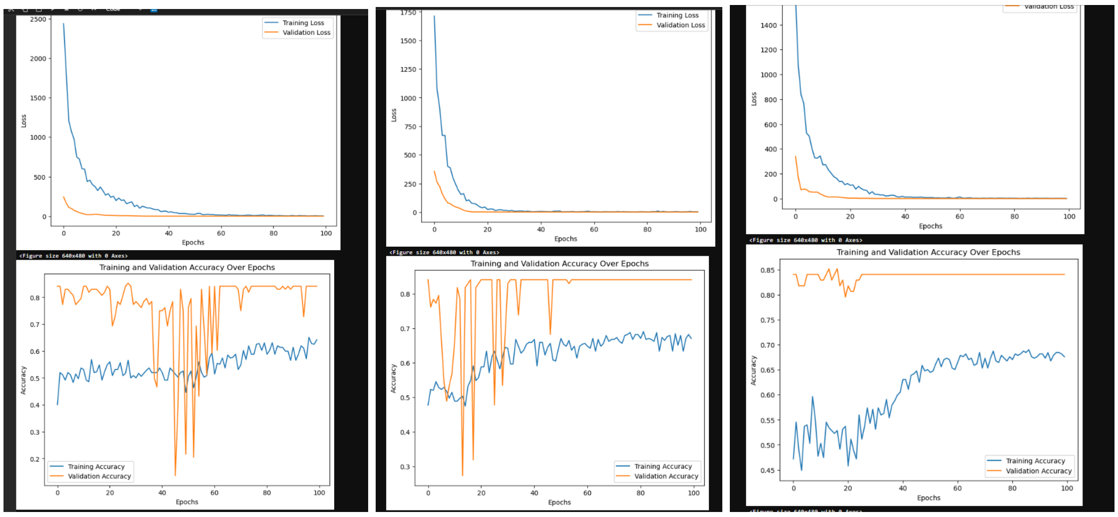
1. network layer adjustment



```python
def build_mlp_model_v1(input_shape, num_classes):
    model = Sequential()
    model.add(Dense(128, input_shape=(input_shape,), activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(128, activation='relu'))  # New Hidden Layers
    model.add(Dropout(0.5))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```



```
Model: "sequential_2"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                       ┃ Output Shape                ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ dense_6 (Dense)                    │ (None, 128)                 │         1,024 │
├────────────────────────────────────┼─────────────────────────────┼───────────────┤
│ dropout_4 (Dropout)                │ (None, 128)                 │             0 │
├────────────────────────────────────┼─────────────────────────────┼───────────────┤
│ dense_7 (Dense)                    │ (None, 128)                 │        16,512 │
├────────────────────────────────────┼─────────────────────────────┼───────────────┤
│ dropout_5 (Dropout)                │ (None, 128)                 │             0 │
├────────────────────────────────────┼─────────────────────────────┼───────────────┤
│ dense_8 (Dense)                    │ (None, 64)                  │         8,256 │
├────────────────────────────────────┼─────────────────────────────┼───────────────┤
│ dropout_6 (Dropout)                │ (None, 64)                  │             0 │
├────────────────────────────────────┼─────────────────────────────┼───────────────┤
│ dense_9 (Dense)                    │ (None, 4)                   │           260 │
└────────────────────────────────────┴─────────────────────────────┴───────────────┘
 Total params: 26,052 (101.77 KB)
 Trainable params: 26,052 (101.77 KB)
 Non-trainable params: 0 (0.00 B)
```



```
<Figure size 640x480 with 0 Axes>
```

The three curves above are the results of tuning and training the model by increasing the number of hidden layers, increasing the number of neurons in the hidden layers, and increasing both the number of hidden layers and the number of neurons. In comparison, the first one shows that the loss curve stabilizes after a rapid decline, while the validation accuracy fluctuates and then stabilizes, but with large fluctuations, which may indicate that the model's ability to generalize to certain data is insufficient or that there is noise in the training.

The second one has a similar loss profile to the first one, but the stability of the accuracy is improved. The fluctuations in accuracy are reduced and the gap between training and validation accuracies is reduced, suggesting that this model may be able to generalize better than the first.

The results for the third show a smoother downward trend in the training loss curve, while the accuracy curve also reaches a relatively high and stable level after initial fluctuations. Despite the obvious gap between training and validation accuracies, they both remained stable in the later stages of training.

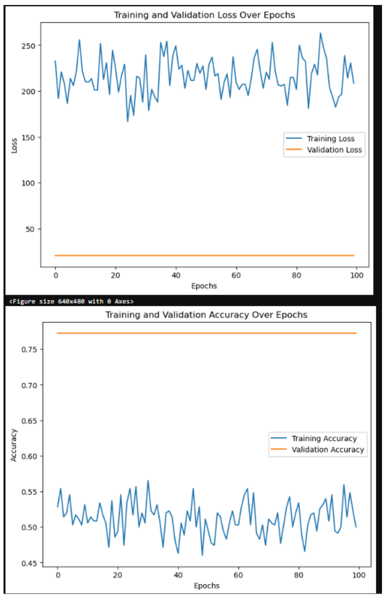
To summarize, the third tweak works best, although it has a gap between training and validation accuracies; however, we can improve it with further regularization or parameter tuning.

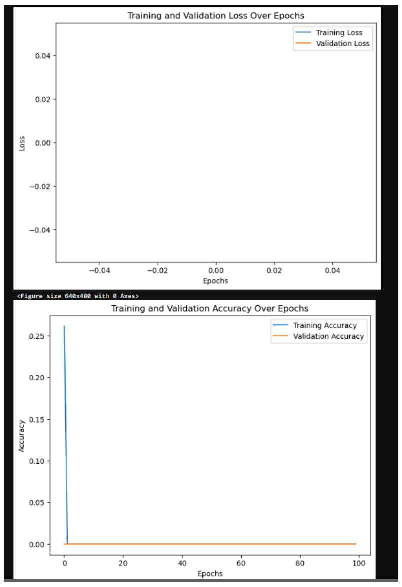
2. Learning rate, SGD and RMSprop optimizer tuning

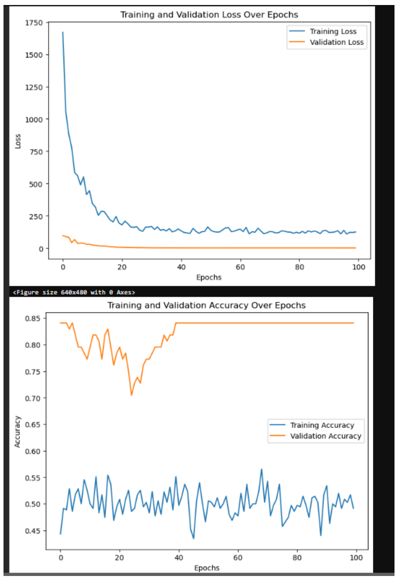
The following three curves are all viewed by adjusting the optimizer, mainly from the step of model interpretation to do the optimization. The first of them is the adjustment of learning rate, the second is the adjustment of SGD, and the third is the adjustment of RMSprop. In comparison, the first loss curve and accuracy curve fluctuate a lot, especially the training accuracy, which indicates that the model may not have fully converged yet, and we need to try to continue to reduce the learning rate, or increase the number of Epochs or introduce learning rate decay.

The second SGD component, and the core optimization we will explore. The display is that the loss curve declines very rapidly, but is almost straight at low loss values, showing that the model may have reached the limits of optimization. However, the

accuracy curve shows large fluctuations, which is usually a sign of excessive learning rates or model instability.
The first shows that both the loss and accuracy curves are relatively smooth and the gap between the training and validation loss curves is small, indicating that the model generalizes well. The final validation accuracy stabilizes at a high level, which is a positive sign.
To summarize, the third adjustment is the best, meanwhile, although the SGD is not the best, probably because the learning rate is too high, it needs to be combined with the learning rate reduction, and then further the batch size optimization to improve it, our final study is the minimum batch SGD optimization.

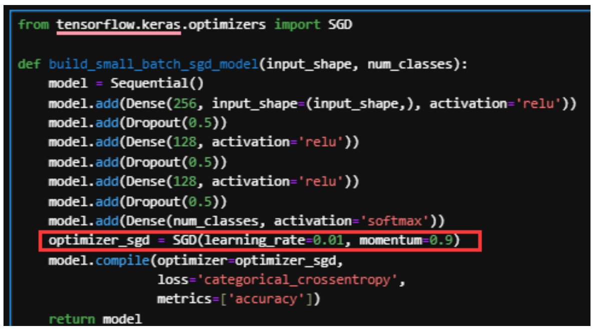






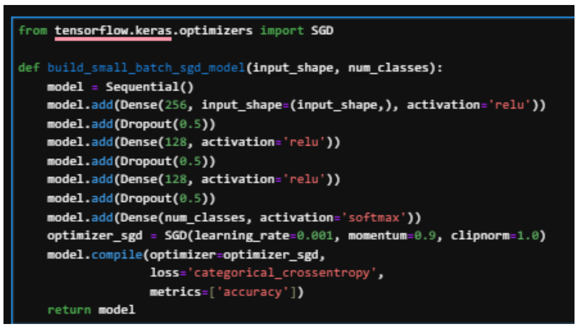
3. the minimum batch SGD optimization

Adjusting the batch size usually affects the training speed and convergence of the model, the smaller the batch size, the more it increases the noise, which helps to improve the model generalization and prevent the overfitting phenomenon. However, it can also lead to slower training and poor convergence, and larger batch sizes require more memory resources. Several different batch sizes were adapted here, the batch size is set to 1.
On the left is the minimum batch SGD with gradient explosion, and on the right is the optimized minimum batch SGD by reducing the learning rate and using gradient trimming in the optimizer.



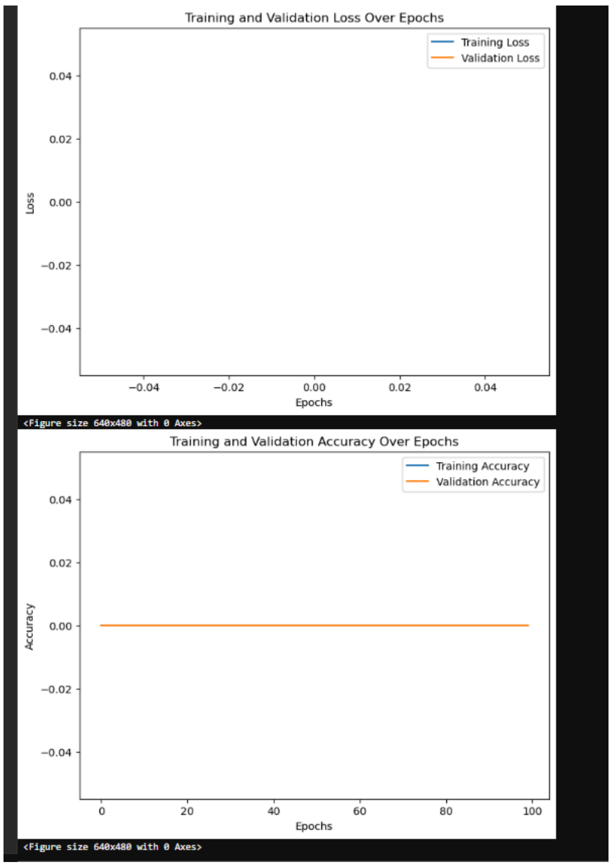
```
from tensorflow.keras.optimizers import SGD

def build_small_batch_sgd_model(input_shape, num_classes):
    model = Sequential()
    model.add(Dense(256, input_shape=(input_shape,), activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    optimizer_sgd = SGD(learning_rate=0.01, momentum=0.9)
    model.compile(optimizer=optimizer_sgd,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

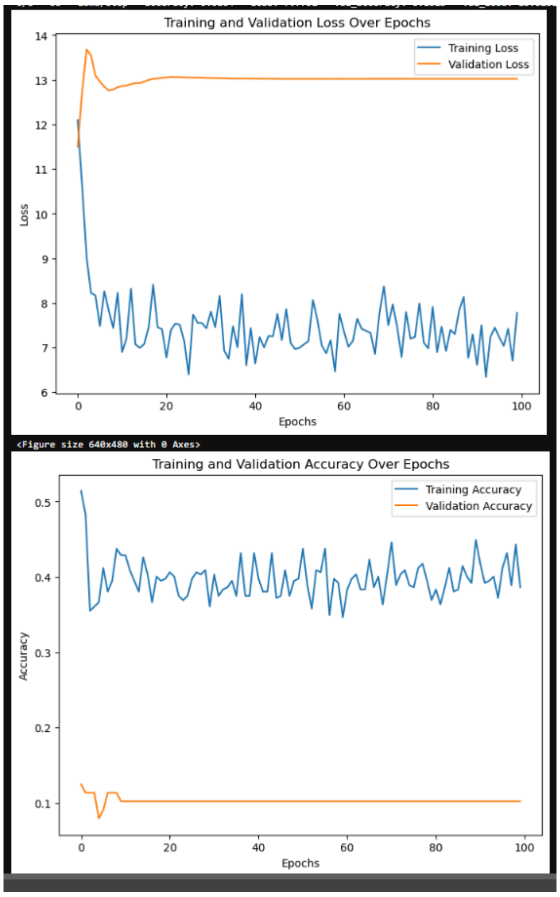


```
from tensorflow.keras.optimizers import SGD

def build_small_batch_sgd_model(input_shape, num_classes):
    model = Sequential()
    model.add(Dense(256, input_shape=(input_shape,), activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    optimizer_sgd = SGD(learning_rate=0.001, momentum=0.9, clipnorm=1.0)
    model.compile(optimizer=optimizer_sgd,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```



```
<Figure size 640x480 with 0 Axes>
<Figure size 640x480 with 0 Axes>
```



```
<Figure size 640x480 with 0 Axes>
```

In the image on the left, both loss and accuracy are NAN, which indicates a numerical instability problem with model training. In general, this can be caused by factors such as an unsuitable loss function, data defects, too high a learning rate, or improper model initialization.
In the image on the right, the training loss fluctuates after a drop, which is a sign that the model is learning but has not yet fully stabilized. The validation loss is slightly higher than the training loss, which may indicate slight overfitting. Meanwhile, the training accuracy fluctuates but is generally stable in the image on the right. This means that the model is learning, but may have limited performance gains due to data or model architecture limitations. The validation accuracy is relatively low, which could be overfitting or the model is not complex enough to capture the complexity of the data.

Question Disscions

1. Adding Gaussian noise and changing batch size

Adding noise to the gradient can mimic the effect of smaller batches because the noise introduces additional randomness, which helps to explore different regions of the loss function and thus potentially reduce overfitting. In contrast, actually reducing the batch size not only adds randomness but also changes the amount of data used in each update. Both of these strategies help to improve the generalization of the model, but adding noise may have different effects on the stability of the model and the speed of convergence.

2. Use of adaptive methods

Using an adaptive method like Adam's can often help speed up convergence in the early stages of training and eliminate the need to manually adjust the step size. Adaptive methods automatically adjust the step size based on the historical gradient of each parameter. This may or may not increase overfitting

3. Using line search (line search) in gradient descent:

Line search is a method of determining the step size with the goal of finding a step size in gradient descent that causes the objective function (cost function) to decrease along the direction of the gradient. In each iteration, line search tries to find a step size that satisfies certain conditions (such as Wolfe's condition or Armijo's rule) to ensure that the new point improves the value of the objective function.

1) Advantages:

Precise control: provides precise control of the step size, which can avoid instability due to too large a step size or slow convergence due to too small a step size.

Adaptive: adaptively adjusts the step size to be able to cope with different regions of the objective function, which is especially useful in complex optimization problems.

Theoretical support: there is theoretical support that ensures that convergence is obtained under certain conditions.

2) Disadvantages:

Computational cost: multiple function evaluations may be required to determine the appropriate step size, which increases the computational cost in each iteration.

4. Add penalties to the cost function to impose constraints

Adding penalties to the cost function is a soft constraint approach where the goal is to steer the optimization process without violating the constraints.

Summary

Through the research of this assignment, it was found that if the optimization of the model depends on many factors, firstly, the selection of the type of task, then the selection of the dataset, and then the selection of the base model, if it is a deep learning model, it will involve the phenomenon of model overfitting, then the optimization of the model needs to be viewed from the following points, firstly, the optimization of the base model itself, that is to say, the adjustment of the structure of the network, which was tested and evaluated, the At the same time increase the number of hidden layers and the number of neural networks, the model generalization effect will be better. Then the optimizer adjustment, this part is mainly for the model interpreter part of the optimization, after testing and evaluation, the RMSprop effect is the best, the SGD effect is second, and Adam's learning rate is the lowest. Finally, the batch size adjustment, after testing and evaluation, the smaller the batch size, the better the effect of SGD.

Appendix

```
import pandas as pd
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split



# Load Dataset
def load_data(file_path):
    data = pd.read_csv(file_path)
    return data

# Cleaning data and pre-processing
def clean_data(data):
    # Remove duplicate values
    data = data.drop_duplicates()

    # Converting Label Types
    cols_to_convert = ['Channel', 'Region']
    for col in cols_to_convert:
        data[col] = data[col].astype('object')

    # Define the processing of numerical and categorical features
    numeric_features = data.select_dtypes(include=['int64', 'float64']).columns
    categorical_features = data.select_dtypes(include=['object']).columns

    # Create converters for numerical and categorical features
    numeric_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='median')),  # Median Fill Missing Values
        ('scaler', StandardScaler())])  # standardization

    categorical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),  # Filling in
missing values
        ('onehot', OneHotEncoder(handle_unknown='ignore'))])  # unique thermal code

    # Merge converter
    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numeric_transformer, numeric_features),
            ('cat', categorical_transformer, categorical_features)])

    # Applying Converters to Data Sets
    data = preprocessor.fit_transform(data)

    return data


# Divide the dataset
def split_data(data, target_column, categorical_columns):
    # 1)the classification features are encoded as values
    label_encoders = {}
    for col in categorical_columns:
        if col != target_column:
            label_encoder = LabelEncoder()
            data[col] = label_encoder.fit_transform(data[col])
            label_encoders[col] = label_encoder

    # 2)Separation of characteristics and target variables
    X = data.drop(target_column, axis=1)
    y = data[target_column]

    # 3)Divide the dataset into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)


    return X_train, X_test, y_train, y_test

from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

def build_mlp_model(input_shape, num_classes):
    # 1)
    model = Sequential()
    model.add(Dense(128, input_shape=(input_shape,), activation='relu'))
    model.add(Dropout(0.5))                    # Dropout
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    # softmax
    model.add(Dense(num_classes, activation='softmax'))
    # 2）
    model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

    return model

from tensorflow.keras.callbacks import ModelCheckpoint

def train_model(model, X_train, y_train_encoded, X_val, y_val_encoded, epochs,
batch_size):

    # checkpoint = ModelCheckpoint('results/best_model.h5', monitor='val_loss',
save_best_only=True)
    checkpoint = ModelCheckpoint('results/best_model.keras', monitor='val_loss',
save_best_only=True)

    history = model.fit(X_train, y_train_encoded,
                        epochs=epochs,
                        batch_size=batch_size,
                        validation_data=(X_val, y_val_encoded),
                        callbacks=[checkpoint],
                        verbose=2)
    return model, history

import matplotlib.pyplot as plt



def plot_loss(history):
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Training and Validation Loss Over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
    plt.savefig('results/figures/training_validation_loss.png')


def plot_accuracy(history):
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Training and Validation Accuracy Over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
    plt.savefig('results/figures/training_validation_accuracy.png')

import numpy as np
from sklearn.metrics import classification_report, confusion_matrix
from tensorflow.keras.utils import to_categorical

def evaluate_model(model, X_test, y_test_encoded):

    probabilities = model.predict(X_test)

    predictions = np.argmax(probabilities, axis=1)

    y_test = np.argmax(y_test_encoded, axis=1)

    report = classification_report(y_test, predictions)
    print(report)

    cm = confusion_matrix(y_test, predictions)
    print('Confusion Matrix:')
    print(cm)

from src.data_preprocessing import load_data, clean_data, split_data
from src.model import build_model
from src.training import train_model
from src.evaluation import evaluate_model
from src.visualization import plot_performance

import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, classification_report
from tensorflow.keras.utils import to_categorical

try:
    import keras
    print("Keras is installed and ready to use.")
except ImportError:
    print("Keras is not installed.")
```

```
file_path = 'data/dataset.csv'
data = load_data(file_path)

data.info()

data = clean_data(data)

target_column = 'Region'  # predict Region
categorical_columns = ['Channel']
X_train, X_test, y_train, y_test, label_encoders = split_data(data, target_column,
categorical_columns)

X_train , X_test, y_train, y_test, label_encoders


input_shape = X_train.shape[1]


num_classes = y_train.nunique()


y_train_encoded = to_categorical(y_train)
y_test_encoded = to_categorical(y_test)


model = build_mlp_model(input_shape, num_classes=4)


model.summary()

epochs = 100  # epoch
batch_size = 32  #  batch
model, history = train_model(model, X_train, y_train_encoded, X_test,
y_test_encoded, epochs, batch_size)

plot_loss(history)
plot_accuracy(history)

evaluate_model(model, X_test, y_test_encoded)


def build_mlp_model_v1(input_shape, num_classes):
    model = Sequential()
    model.add(Dense(128, input_shape=(input_shape,), activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(128, activation='relu'))  # New Hidden Layers
    model.add(Dropout(0.5))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model


input_shape = X_train.shape[1]


num_classes = y_train.nunique()


y_train_encoded = to_categorical(y_train)
y_test_encoded = to_categorical(y_test)


model = build_mlp_model_v1(input_shape, num_classes=4)


model.summary()


def build_mlp_model_v2(input_shape, num_classes):
    model = Sequential()
    model.add(Dense(256, input_shape=(input_shape,), activation='relu'))  #
model.add(Dropout(0.5))
    model.add(Dense(64, activation='relu'))  #
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model


input_shape = X_train.shape[1]
```

```
num_classes = y_train.nunique()


y_train_encoded = to_categorical(y_train)
y_test_encoded = to_categorical(y_test)

model = build_mlp_model_v2(input_shape, num_classes=4)

model.summary()


from tensorflow.keras.optimizers import Adam
def build_mlp_model_v3(input_shape, num_classes):
    model = Sequential()
    model.add(Dense(256, input_shape=(input_shape,), activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    optimizer = Adam(learning_rate=0.001)
    model.compile(optimizer=optimizer,
loss='categorical_crossentropy',
                 metrics=['accuracy'])
    return model

from tensorflow.keras.optimizers import SGD

def build_mlp_model_v3(input_shape, num_classes):
    model = Sequential()
    model.add(Dense(256, input_shape=(input_shape,), activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    optimizer_sgd = SGD(learning_rate=0.01, momentum=0.9)
model.compile(optimizer=optimizer_sgd,
loss='categorical_crossentropy',
                 metrics=['accuracy'])
    return model
```

```
from tensorflow.keras.optimizers import RMSprop

def build_mlp_model_v3(input_shape, num_classes):
   model = Sequential()
   model.add(Dense(256, input_shape=(input_shape,), activation='relu'))
   model.add(Dropout(0.5))
   model.add(Dense(128, activation='relu'))
   model.add(Dropout(0.5))
   model.add(Dense(128, activation='relu'))
   model.add(Dropout(0.5))
   model.add(Dense(num_classes, activation='softmax'))
   optimizer_rmsprop = RMSprop(learning_rate=0.001)
model.compile(optimizer=optimizer_rmsprop,
loss='categorical_crossentropy',
              metrics=['accuracy'])
   return model

from tensorflow.keras.optimizers import SGD

def build_small_batch_sgd_model(input_shape, num_classes):
   model = Sequential()
   model.add(Dense(256, input_shape=(input_shape,), activation='relu'))
   model.add(Dropout(0.5))
   model.add(Dense(128, activation='relu'))
   model.add(Dropout(0.5))
   model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
   model.add(Dense(num_classes, activation='softmax'))
   optimizer_sgd = SGD(learning_rate=0.001, momentum=0.9, clipnorm=1.0)
model.compile(optimizer=optimizer_sgd,                              #
loss='categorical_crossentropy',
              metrics=['accuracy'])
   return model
batch_size = 1
model = build_small_batch_sgd_model(input_shape=X_train.shape[1],
num_classes=4)
model.summary()
history = model.fit(X_train, y_train_encoded, epochs=100, batch_size=batch_size,
               validation_data=(X_test, y_test_encoded), verbose=2)
model, history = train_model(model, X_train, y_train_encoded, X_test,
y_test_encoded, epochs, batch_size)

plot_loss(history)
plot_accuracy(history)
```