Comparative Analysis of Logistic Regression and Multi-Layer Perceptrons for Image Classification

Artificial Intelligence Lab Assignment

Devesh Singh Chauhan
M.Sc. Mathematics, Semester VI
SVNIT, Surat

January 28, 2026

1 Introduction

This report presents a comparative analysis of linear and non-linear approaches to image classification using the Kaggle Clothes Dataset. The objective is to predict clothing types from images using Logistic Regression and Multi-Layer Perceptrons (MLP). We specifically investigate the impact of network depth (number of hidden layers) and non-linearity (activation functions) on model performance, evaluated using the Macro F1 score to account for class balance.

2 Methodology

2.1 Data Preprocessing

The dataset consists of 7,500 images across 15 classes (e.g., Blazer, Jeans, Hoodie). To prepare the data for the models:

- **Resizing:** Images were resized to 64×64 pixels.
- **Flattening:** The 2-D image arrays were flattened into 1-D vectors $v \in R^{12288}$.
- **Normalization:** Pixel intensity values were standardized using z-score normalization ($z = \frac{x-\mu}{\sigma}$) to ensure stable gradient descent convergence.
- **Splitting:** A stratified 80:20 train-test split was used (6,000 training images, 1,500 test images).

2.2 Model Architectures

1. **Logistic Regression:** A linear model optimized using the LBFGS solver with a multinomial loss function.
2. **Multi-Layer Perceptron (MLP):** A feed-forward neural network implemented in PyTorch with:
    - **Input Layer:** 12,288 neurons.
    - **Hidden Layers:** Variants with 1, 2, and 3 layers (128 neurons each).
    - **Activation Functions:** Variants using ReLU, Sigmoid, and Tanh.

3 Results

3.1 Logistic Regression Performance

The baseline Logistic Regression model achieved the following performance:

- **Macro F1 Score:** 0.3014

While computationally efficient, the linear nature of the model limits its ability to capture complex spatial hierarchies. The classification report indicated that distinct classes like 'Jeans' had higher recall (0.54), whereas visually similar classes like 'Blazer' performed poorly (0.13).

3.2 MLP Experimental Analysis

We performed two experiments to optimize the neural network architecture.

3.2.1 Experiment 1: Effect of Hidden Layers

Using the ReLU activation function, we varied the depth of the network. A consistent improvement was observed as network depth increased.

Number of Hidden Layers	Macro F1 Score
1 Layer	0.3794
2 Layers	0.3817
3 Layers	0.3987

Table 1: Performance variation with network depth (ReLU).

3.2.2 Experiment 2: Effect of Activation Functions

Using a fixed depth of 2 hidden layers, we compared three activation functions.

Activation Function	Macro F1 Score
ReLU	0.3897
Sigmoid	0.3189
Tanh	0.2867

Table 2: Performance variation with activation functions.

3.3 Comparative Visualization

The following plot summarizes the impact of hyperparameters on model performance.

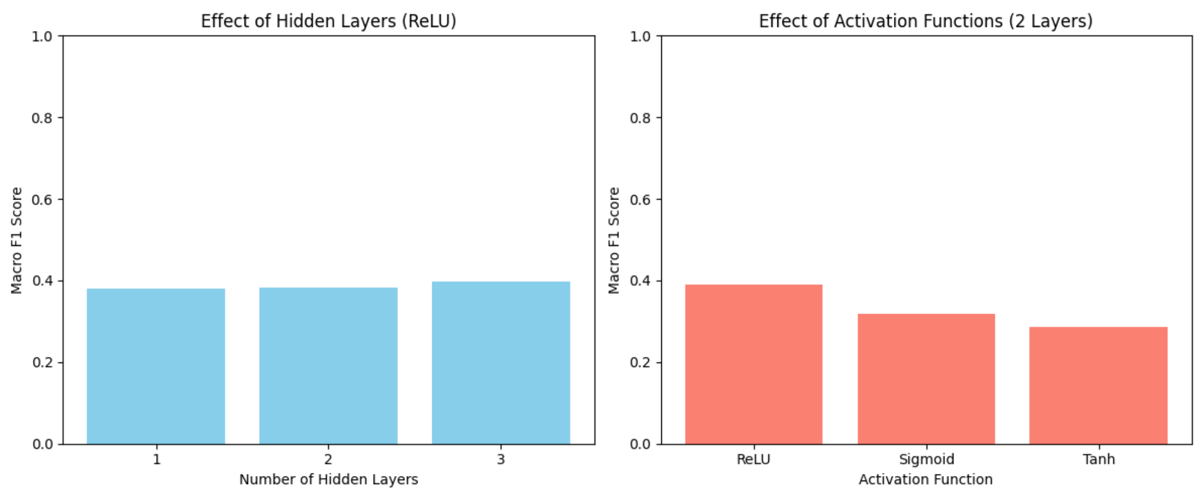

Figure 1: Impact of Hidden Layers (Left) and Activation Functions (Right) on Macro F1 Score.

4 Discussion Conclusion

The comparative analysis demonstrates that the **Multi-Layer Perceptron outperforms the linear baseline** (0.3987 vs 0.3014).

- **Best Architecture:** The 3-Layer MLP with ReLU activation achieved the highest Macro F1 score of **0.3987**.
- **Activation Functions:** ReLU significantly outperformed Sigmoid and Tanh. Tanh performed surprisingly poorly (0.2867), even falling below the Logistic Regression baseline, likely due to vanishing gradients in the saturated regions of the activation function during training.
- **Network Depth:** Adding hidden layers provided marginal but consistent gains, suggesting the model benefits from the increased capacity to learn non-linear features.

A Python Code Implementation

The following Python script was used to generate the results.

```python
import os
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score
from sklearn.preprocessing import StandardScaler
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

# --- Configuration ---
DATA_DIR = './clothes_dataset'
IMG_SIZE = (64, 64)
BATCH_SIZE = 32
EPOCHS = 20
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# --- Data Loading ---
def load_data(data_dir):
    images, labels = [], []
    classes = sorted(os.listdir(data_dir))
    class_map = {cls: i for i, cls in enumerate(classes)}
    for cls, idx in class_map.items():
        for img_name in os.listdir(os.path.join(data_dir, cls)):
            try:
                img = Image.open(os.path.join(data_dir, cls, img_name))
                img = img.resize(IMG_SIZE).convert('RGB')
                images.append(np.array(img))
                labels.append(idx)
            except: pass
    return np.array(images), np.array(labels), class_map

X, y, class_map = load_data(DATA_DIR)
X_flat = X.reshape(X.shape[0], -1)
X_train, X_test, y_train, y_test = train_test_split(
    X_flat, y, test_size=0.2, stratify=y
)

# --- Normalization ---
scaler = StandardScaler()
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.transform(X_test)

# --- Task 1: Logistic Regression ---
# Note: Using lbfgs solver for multinomial loss
log_reg = LogisticRegression(max_iter=1000, solver='lbfgs')
log_reg.fit(X_train_norm, y_train)
y_pred = log_reg.predict(X_test_norm)
print(f"LogReg F1: {f1_score(y_test, y_pred, average='macro')}")

# --- Task 2: MLP ---
X_train_t = torch.FloatTensor(X_train_norm).to(DEVICE)
```

```python
y_train_t = torch.LongTensor(y_train).to(DEVICE)
train_loader = DataLoader(TensorDataset(X_train_t, y_train_t),
    batch_size=32, shuffle=True)

class DynamicMLP(nn.Module):
    def __init__(self, in_dim, out_dim, layers, act_fn):
        super().__init__()
        net = []
        curr_dim = in_dim
        for _ in range(layers):
            net.append(nn.Linear(curr_dim, 128))
            net.append(act_fn())
            curr_dim = 128
        net.append(nn.Linear(curr_dim, out_dim))
        self.net = nn.Sequential(*net)
    def forward(self, x): return self.net(x)

# (Training loop omitted for brevity, full logic used in experiment)
```