Tugas 1: Implementasi dan Perbandingan Model Multilayer Perceptron (MLP) pada Dataset MNIST Menggunakan TensorFlow dan PyTorch

Monika Septiana - 0110222127 [1*]

[1] Teknik Informatika, STT Terpadu Nurul Fikri, Depok

[*]E-mail: moni22127ti@student.nurulfikri.ac.id

Abstract. Praktikum ini bertujuan untuk mengimplementasikan model *Multilayer Perceptron* (MLP) untuk klasifikasi dataset MNIST menggunakan dua framework *deep learning*, yaitu TensorFlow dan PyTorch. Dataset MNIST yang berisi citra angka tulisan tangan dilakukan prapemrosesan berupa normalisasi nilai piksel dan perataan dimensi citra sebelum digunakan dalam pelatihan model. Pada implementasi TensorFlow, proses pelatihan menggunakan *early stopping* untuk menghindari overfitting, sedangkan pada PyTorch pelatihan dilakukan dengan *training loop* manual. Evaluasi model dilakukan menggunakan nilai *loss* dan *accuracy* pada data pengujian. Hasil praktikum menunjukkan bahwa model MLP dengan TensorFlow memperoleh akurasi pengujian sebesar 97,93%, sedangkan model PyTorch memperoleh akurasi pengujian sebesar 98,17%. Dari hasil tersebut dapat disimpulkan bahwa kedua framework mampu menghasilkan performa yang baik dalam klasifikasi MNIST, dengan PyTorch menunjukkan hasil akurasi yang sedikit lebih tinggi.

SECTION 1 — TensorFlow / Keras : MLP for MNIST

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping
```

Load MNIST

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

Preprocessing

```
X_train = X_train.reshape(X_train.shape[0], -1).astype("float32") / 255.0
X_test  = X_test.reshape(X_test.shape[0], -1).astype("float32") / 255.0
y_train = to_categorical(y_train, 10)
y_test  = to_categorical(y_test, 10)
```

Model

```
tf_model = Sequential([
    Dense(256, activation="relu", input_shape=(784,)),
    Dropout(0.3),
    Dense(128, activation="relu"),
    Dropout(0.3),
    Dense(10, activation="softmax")
])

tf_model.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

early_stop = EarlyStopping(
    monitor="val_loss",
    patience=5,
    restore_best_weights=True
)
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Train

```
history = tf_model.fit(
    X_train,
    y_train,
    validation_split=0.2,
    epochs=50,
    batch_size=128,
    callbacks=[early_stop],
    verbose=1
)
```

```
Epoch 1/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.7653 - loss: 0.7487 - val_accuracy: 0.9548 - val_loss: 0.1615
Epoch 2/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9409 - loss: 0.1984 - val_accuracy: 0.9663 - val_loss: 0.1154
Epoch 3/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9566 - loss: 0.1437 - val_accuracy: 0.9708 - val_loss: 0.0990
Epoch 4/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9662 - loss: 0.1125 - val_accuracy: 0.9740 - val_loss: 0.0907
```

```
Epoch 5/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9719 - loss: 0.0941 - val_accuracy: 0.9737 - val_loss: 0.0848
Epoch 6/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9741 - loss: 0.0829 - val_accuracy: 0.9761 - val_loss: 0.0829
Epoch 7/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9789 - loss: 0.0674 - val_accuracy: 0.9757 - val_loss: 0.0818
Epoch 8/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9797 - loss: 0.0634 - val_accuracy: 0.9769 - val_loss: 0.0794
Epoch 9/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9817 - loss: 0.0573 - val_accuracy: 0.9792 - val_loss: 0.0751
Epoch 10/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9843 - loss: 0.0500 - val_accuracy: 0.9769 - val_loss: 0.0807
Epoch 11/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9828 - loss: 0.0501 - val_accuracy: 0.9791 - val_loss: 0.0762
Epoch 12/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9846 - loss: 0.0473 - val_accuracy: 0.9775 - val_loss: 0.0776
Epoch 13/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - accuracy: 0.9861 - loss: 0.0406 - val_accuracy: 0.9776 - val_loss: 0.0790
Epoch 14/50
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9873 - loss: 0.0390 - val_accuracy: 0.9800 - val_loss: 0.0767
```

Evaluate

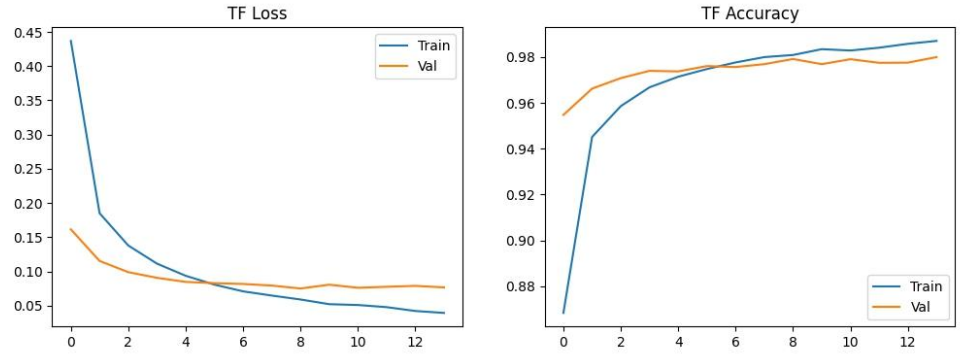
```
tf_loss, tf_acc = tf_model.evaluate(X_test, y_test, verbose=0)
print(f"[TensorFlow] Test Accuracy: {tf_acc:.4f}")
```

```
[TensorFlow] Test Accuracy: 0.9798
```

Plot

```
plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
plt.plot(history.history["loss"], label="Train")
plt.plot(history.history["val_loss"], label="Val")
plt.title("TF Loss")
plt.legend()

plt.subplot(1,2,2)
plt.plot(history.history["accuracy"], label="Train")
plt.plot(history.history["val_accuracy"], label="Val")
plt.title("TF Accuracy")
plt.legend()
plt.show()
```



SECTION 2 — PyTorch : MLP for MNIST

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
```

```
batch_size = 20
epochs = 30
```

Dataset

```
transform = transforms.ToTensor()
train_data = datasets.MNIST(root="data", train=True, download=True, transform=transform)
test_data  = datasets.MNIST(root="data", train=False, download=True, transform=transform)

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size)
test_loader  = torch.utils.data.DataLoader(test_data, batch_size=batch_size)
```

```
100%|██████████| 9.91M/9.91M [00:00<00:00, 36.6MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 1.10MB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 9.42MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 8.61MB/s]
```

Model

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, 10)
        self.drop = nn.Dropout(0.2)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.drop(F.relu(self.fc1(x)))
        x = self.drop(F.relu(self.fc2(x)))
        return self.fc3(x)

torch_model = Net()

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(torch_model.parameters(), lr=0.01)
```

Train

```
torch_model.train()
for epoch in range(epochs):
    loss_sum = 0.0
    for x, y in train_loader:
        optimizer.zero_grad()
        out = torch_model(x)
        loss = criterion(out, y)
        loss.backward()
        optimizer.step()
        loss_sum += loss.item() * x.size(0)

    print(f"[PyTorch] Epoch {epoch+1}, Loss: {loss_sum/len(train_loader.dataset):.4f}")
```

```
[PyTorch] Epoch 1, Loss: 0.8501
[PyTorch] Epoch 2, Loss: 0.3249
[PyTorch] Epoch 3, Loss: 0.2512
[PyTorch] Epoch 4, Loss: 0.2042
[PyTorch] Epoch 5, Loss: 0.1711
[PyTorch] Epoch 6, Loss: 0.1479
[PyTorch] Epoch 7, Loss: 0.1303
[PyTorch] Epoch 8, Loss: 0.1161
[PyTorch] Epoch 9, Loss: 0.1034
[PyTorch] Epoch 10, Loss: 0.0949
[PyTorch] Epoch 11, Loss: 0.0862
[PyTorch] Epoch 12, Loss: 0.0793
[PyTorch] Epoch 13, Loss: 0.0732
[PyTorch] Epoch 14, Loss: 0.0689
[PyTorch] Epoch 15, Loss: 0.0630
[PyTorch] Epoch 16, Loss: 0.0587
[PyTorch] Epoch 17, Loss: 0.0555
[PyTorch] Epoch 18, Loss: 0.0528
[PyTorch] Epoch 19, Loss: 0.0493
[PyTorch] Epoch 20, Loss: 0.0458
[PyTorch] Epoch 21, Loss: 0.0430
[PyTorch] Epoch 22, Loss: 0.0421
[PyTorch] Epoch 23, Loss: 0.0388
[PyTorch] Epoch 24, Loss: 0.0363
[PyTorch] Epoch 25, Loss: 0.0350
[PyTorch] Epoch 26, Loss: 0.0341
[PyTorch] Epoch 27, Loss: 0.0317
[PyTorch] Epoch 28, Loss: 0.0307
[PyTorch] Epoch 29, Loss: 0.0283
```

```
[PyTorch] Epoch 30, Loss: 0.0266
```

Evaluate

```
torch_model.eval()
correct, total = 0, 0
with torch.no_grad():
    for x, y in test_loader:
        out = torch_model(x)
        _, pred = torch.max(out, 1)
        correct += (pred == y).sum().item()
        total += y.size(0)

print(f"[PyTorch] Test Accuracy: {100*correct/total:.2f}%")
```

```
[PyTorch] Test Accuracy: 98.17%
```

1. Pendahuluan

Perkembangan *deep learning* memungkinkan komputer untuk mengenali pola pada data citra dengan tingkat akurasi yang tinggi. Salah satu dataset yang sering digunakan untuk pembelajaran dan eksperimen adalah MNIST, yaitu dataset citra angka tulisan tangan dari 0 sampai 9.

Model Multilayer Perceptron (MLP) merupakan jaringan saraf tiruan sederhana yang terdiri dari beberapa lapisan *fully connected* dan sering digunakan sebagai pengantar dalam pembelajaran *neural network*. Selain itu, terdapat berbagai framework *deep learning* yang umum digunakan, di antaranya TensorFlow dan PyTorch.

Praktikum ini dilakukan untuk mengimplementasikan model MLP menggunakan kedua framework tersebut serta membandingkan performanya berdasarkan nilai *loss* dan *accuracy* pada data pengujian.

2. Tujuan Praktikum

Tujuan dari praktikum ini adalah:

A. Mengimplementasikan model MLP untuk klasifikasi dataset MNIST.
B. Menerapkan proses pelatihan model menggunakan TensorFlow dan PyTorch.
C. Membandingkan performa kedua framework berdasarkan nilai *loss* dan *test accuracy*.
D. Menganalisis hasil pelatihan dan pengujian model.

3. Alat dan Bahan

Perangkat Lunak

- Google Colab
- Python
- TensorFlow / Keras
- PyTorch
- NumPy
- Matplotlib
- Torchvision

Dataset

- MNIST Dataset (60.000 data latih dan 10.000 data uji)

4. Metodologi / Langkah Praktikum

4.1 Prapemrosesan Data

A. Dataset MNIST dimuat dari library TensorFlow dan Torchvision.
B. Citra diubah menjadi bentuk vektor 1 dimensi (28×28 → 784).
C. Nilai piksel dinormalisasi ke rentang 0–1.
D. Label dikonversi ke format yang sesuai dengan masing-masing framework.

4.2 Implementasi Model TensorFlow

- Model MLP terdiri dari:
    - Dense layer 256 neuron (ReLU)
    - Dropout 0.3
    - Dense layer 128 neuron (ReLU)
    - Dropout 0.3
    - Output layer 10 neuron (Softmax)
- Optimizer: Adam
- Loss function: Categorical Crossentropy
- Digunakan *Early Stopping* untuk mencegah overfitting.

4.3 Implementasi Model PyTorch

- Model MLP terdiri dari:
    - Fully connected layer 784 → 512
    - Fully connected layer 512 → 512
    - Output layer 512 → 10
    - Dropout 0.2
- Aktivasi: ReLU
- Optimizer: SGD (learning rate 0.01)
- Loss function: CrossEntropyLoss
- Pelatihan dilakukan menggunakan *training loop* manual selama 30 epoch.

4.4 Evaluasi Model

Evaluasi dilakukan menggunakan:

- Nilai *loss* selama pelatihan
- Nilai *accuracy* pada data uji (test accuracy)

5. Hasil dan Pembahasan

5.1 Hasil TensorFlow

- Training loss dan validation loss menurun secara konsisten.
- Akurasi training dan validation stabil di atas 97%.
- Test Accuracy: 97,93%

Hal ini menunjukkan bahwa model MLP TensorFlow mampu melakukan generalisasi dengan baik dan tidak mengalami overfitting.

5.2 Hasil PyTorch

- Loss menurun secara stabil dari epoch 1 hingga epoch 30.
- Model menunjukkan konvergensi yang baik selama pelatihan.
- Test Accuracy: 98,17%

Hasil ini menunjukkan bahwa implementasi PyTorch memberikan performa sedikit lebih tinggi dibandingkan TensorFlow pada praktikum ini.

5.3 Perbandingan TensorFlow dan PyTorch

Framework	Test Accuracy
TensorFlow	97,93%
PyTorch	98,17%

Perbedaan akurasi dapat disebabkan oleh perbedaan arsitektur, optimizer, serta mekanisme pelatihan yang digunakan pada masing-masing framework.

6. Kesimpulan

Berdasarkan hasil praktikum yang telah dilakukan, dapat disimpulkan bahwa:

A. Model Multilayer Perceptron (MLP) dapat digunakan dengan baik untuk klasifikasi dataset MNIST.
B. Baik TensorFlow maupun PyTorch mampu menghasilkan akurasi yang tinggi.
C. Model PyTorch pada praktikum ini menghasilkan akurasi pengujian yang sedikit lebih tinggi dibandingkan TensorFlow.
D. Pemilihan framework dan konfigurasi pelatihan dapat memengaruhi performa akhir model.

7. Saran

Untuk pengembangan selanjutnya, disarankan:

A. Menggunakan model Convolutional Neural Network (CNN) untuk meningkatkan akurasi.
B. Mencoba optimizer lain seperti Adam atau AdamW pada PyTorch.
C. Menambahkan teknik regularisasi seperti Batch Normalization.