Training an MLP from scratch

Nom : Senhadji M Said

Necessary Packages

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
```

1- Initialization

Write the function `init_params(nx, nh, ny)`

```
def init_params(nx, nh, ny):
    W1 = np.random.normal(loc=0, scale=0.3, size=(nh,
nx+1)).astype(np.float32)
    W2 = np.random.normal(loc=0, scale=0.3, size=(ny,
nh+1)).astype(np.float32)
    return [W1, W2]
```

Forward propagation

Activation Functions:

- `tanh(z)` : Hyperbolic tangent activation function
- `sigmoid(z)` : Standard sigmoid function
- `softmax(z)` : Converts logits into probabilities, with a numerical stability trick `z.max(axis=0)` to prevent overflow.

```
def tanh(z):
    return np.tanh(z)

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def softmax(z):
    t = np.exp(z - np.max(z, axis=0, keepdims=True))
    return t / np.sum(t, axis=0, keepdims=True)
```

`forward` function

1. **inputs**:

- `params` : List of wight matrices for each layer.
   - `X`: Input data of shape `(n_batch, nx)`
   - `activation`: List of activation functions to be used for each layer
2. **Processing**:
   - The input `X` is transposed `(Y = X.T)` to match matrix multiplication dimensions.
   - The function loops through each weight matrix (`W`) and activation function (`activation`).
   - **Bias Handling**: Adds a row of ones to Y to account for bias terms.
   - Computes `Z = W @ Y` (weighted sum).
   - Applies the activation function : `Y = activation(Z)`.
   - Stores both `Z` (pre-activation) and `Y` (post-activation) in `outputs`.
3. **Returns**:
   - `outputs`: A list of intermediate values, as `[Z, Y]`.

```
def forward(params, X, activations):
    Y = X.T
    outputs = []

    for W, activation in zip(params, activations):
        Y = np.vstack([np.ones((1, Y.shape[1])), Y])
        Z = W @ Y  # Linear transformation
        Y = activation(Z)
        outputs.append([Z, Y])

    return outputs
```

Loss & Accuracy

Loss Calculation (Categorical Cross-Entropy)

For multi-class classification, we typically use categorical cross-entropy loss, which is defined as:

$$L=-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{i,j} log\left(\hat{y}_{i,j}\right)$$

where:

- N is the number of samples.
- C is the number of classes.
- $y_{i,j}$ is the actual label for sample i, class j.
- $\hat{y}_{i,j}$ is the predicted probability for sample i, class j.

Accuracy Calculation

$$\text{Accurancy}=\frac{1}{N}\sum_{i=1}^{N} 1\left(arg\,max\,\hat{y}_i = arg\,max\,y_i\right)$$

where:

- $arg\, max\, y_i$: finds the index of the correct class (ground truth label)
- $arg\, max\, \hat{y}_i$: finds the index of the predicted class
- $1(.)$: s an indicator function that returns 1 if the predicted class matches the true class, otherwise 0.

```
def loss_accuracy(y_hat, y):
    loss = -np.mean(np.log(np.sum(y_hat * y, axis=1)))
    accuracy = np.mean(np.argmax(y_hat, axis=1) == np.argmax(y,
axis=1))
    return loss,accuracy
```

Backward propagation

Inputs:

- `X`: Input data of shape (m, nx)
- `params`: List of weight matrices ``
- `outputs`: List of forward pass outputs `[[Z1, A1], [Z2, A2]]`
- `Y`: Ground truth labels

Step 1: Preparing Activations

- Adds a bias row (ones) to `A1` and `X` to match dimensions with weight matrices.
- `outputs[-2][1]` refers to `A1`, the activation from the hidden layer.

Step 2: Compute Gradients for Output Layer

- Computes the output layer error
$$dZ^{[2]} = \hat{Y} - Y$$
- Computes the gradient for `W2`:
$$dW^{[2]} = dZ^{[2]}.\, A^{[1]T}$$

Step 3: Compute Gradients for Hidden Layer

- `outputs[-2][0]` refers to Z1, the pre-activation of the hidden layer.
- Applies the tanh function to retrieve `A1` for the derivative.
$$dZ^{[1]} = \left(W^{[2]T} \cdot dZ^{[2]}\right) * \left(1 - A^{[1]2}\right)$$

Step 4: Compute Gradients for First Layer Computes

$$dW^{[1]} = dZ^{[1]} \cdot X^T$$

Final Return

```
return gradients
```

```
def backward(X, params, outputs, Y):

    # step 1
    outputs[-2][1] = np.vstack([np.ones(outputs[-2]
[1].shape[1]),outputs[-2][1]]) # dA
    X = np.vstack([np.ones(X.shape[0]),X.T])

    # step 2
    gradients = {}

    gradients["dZ2"] = outputs[-1][1] - Y.T # (ny,m) - (ny,m) = (ny,m)
    gradients["dW2"] = gradients["dZ2"] @ outputs[-2][1].T # (ny,m) *
(m,nh+1) = (ny,nh+1)

      # step 3
    t = tanh(outputs[-2][0]) # (nh,m)
    gradients["dZ1"] = (params[-1].T[1:] @ gradients["dZ2"]) * (1 - t
** 2)
    # (nh,ny) @ (ny,m) * (nh,m) = (nh,m)
    # step 4
    gradients["dW1"] = gradients["dZ1"] @ X.T
    # (nh,m) @ (m,nx+1) = (nh,nx+1)

    return gradients
```

Gradient Descent

Stochastic Gradient Descent (SGD) updates the parameters using the formula:

$$W = W - \eta \cdot \nabla W$$

where:

- $ W $ are the weight matrices (`params`)
- $ \eta $ is the learning rate (`eta`)
- $ \nabla W $ are the computed gradients (`grads`)

```
def sgd(params, grads, eta):
    params[0] = params[0] - eta * grads["dW1"]
    params[1] = params[1] - eta * grads["dW2"]
```

Train

utility functions

```
def one_hot(a):
    b = np.zeros((a.size, a.max() + 1))
    b[np.arange(a.size), a] = 1
    return b
```

```python
def predict(params, X):
    outputs = forward(params, X, [tanh,softmax])
    y_hat = outputs[-1][-1]
    y_hat = np.argmax(y_hat, axis=0)
    return y_hat

def add_eps(params, eps=10e-4):
    result = []

    for param in params:
        result.append(param + eps)

    return result
```

Train function

```python
from tqdm.notebook import trange

def train(X, Y, test_set=None,eta=0.01, epochs=50, batch_size=128,
nh=32):
    #  1 setup & initi
    m,n = X.shape

    ny = len(np.unique(Y))

    Y = one_hot(Y)

    if test_set is not None:
        test_set = (test_set[0], one_hot(test_set[1]))

    params = init_params(n,nh,ny)

    # 2 Initialize History
    history = {
        "accuracy": [],
        "loss":[],
        "test loss":[],
        "test accuracy":[]
    }

    real_grads = []
    approx_grads = []

      # 3 Training Loop
    for j in range(epochs):
        idx = np.arange(m)
        np.random.shuffle(idx)
        X = X[idx]
        Y = Y[idx]
```

```
        batches_count = int(np.floor(m / batch_size))
        # 4 mini-batch processing
        t = trange(batches_count, desc='Bar desc', leave=True)

        for i in t:

            X_batch = X[i * batch_size:(i+1) * batch_size,:]
            Y_batch = Y[i * batch_size:(i+1) * batch_size,:]

            outputs = forward(params, X_batch, [tanh, softmax])
            grads = backward(X_batch, params, outputs, Y_batch)

            # 5 gradient approximation
            outputs_p_eps = forward(add_eps(params, 10e-4), X_batch,
[tanh, softmax])[-1][-1]
            outputs_m_eps = forward(add_eps(params, -10e-4), X_batch,
[tanh, softmax])[-1][-1]
            approx_grad = (outputs_m_eps-outputs_p_eps) / (2 * 10e-4)

            # 6 update parametes using SGD
            real_grads.append(grads)
            approx_grads.append(approx_grad)

            sgd(params, grads,eta=eta)

            # 7 compute metrics
            if i % 50 == 0:
                Y_hat = outputs[-1][1].T
                loss, accuracy = loss_accuracy(Y_hat, Y_batch)

                msg = f"epoch = {j+1} | loss = {loss:.6f} | accuracy =
{100 * accuracy:.2f}%"
                test_loss,test_accuracy = None,None

                if test_set is not None:
                    X_test,y_test = test_set
                    y_test_hat = forward(params, X_test, [tanh,
softmax])[-1][1].T
                    test_loss, test_accuracy = loss_accuracy(y_test,
y_test_hat)
                    msg += f" | test loss = {test_loss:.6f} | test
accuracy = {100 * test_accuracy:.2f}%"

                if i % 50 == 0:
                    t.set_description(msg)
                    t.refresh()

                      # 8 Store History
                history["loss"].append(loss)
                history["accuracy"].append(accuracy)
```

```
                if test_set is not None:
                    history["test loss"].append(test_loss)
                    history["test accuracy"].append(test_accuracy)
    return params,history,real_grads,approx_grads
```

Quick Breakdown of the Code

1. **Setup & Initialization**
   - Converts labels to one-hot encoding.
   - Intializes parameters (`W1, W2`)
   - Stores test data (if provided).
2. **Initialize History for Tracking Metrics**
   - `history`: Tracks loss and accuracy for training & test sets
   - `real_grads`: Stores actual gradients from backpropagation.
   - `approx_grads`: Stores numerical gradient approximations.
3. **Training Loop**
   - Shuffles data each epoch for better generalization.
   - Divides data into mini-batches.
4. **Mini-Batch Processing**
   - Iterates over mini-batches.
   - Performs forward pass to compute predictions.
   - Computes gradients using backpropagation.
5. **Gradient Approximation for Debugging**
   - Computes numerical gradients using **finite difference approximation**:
     $$\frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$$
   - Compares backpropagation gradients vs. numerical gradients.
6. **Update Parameters Using SGD**
   - Stores real gradients.
   - Stores approximated gradients for debugging.
7. **Compute Metrics**
   - Computes **loss & accuracy** for training batch.
   - If test data is available, computes test loss & accuracy.
   - Displays progress bar updates in real time.
8. **Store History**
   - Saves training & test metrics for plotting later.

Graphing Accuracy & Loss

```
import matplotlib.pyplot as plt

def plot_metrics(history):
    plt.figure(figsize=(12, 5))
```

```
    # plot loss
    plt.subplot(1, 2, 1)
    plt.plot(history["loss"], label="Train Loss")
    if "test loss" in history:
        plt.plot(history["test loss"], label="Test Loss")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.legend()
    plt.title("Loss Over Time")

    # plot accuracy
    plt.subplot(1, 2, 2)
    plt.plot(history["accuracy"], label="Train Accuracy")
    if "test accuracy" in history:
        plt.plot(history["test accuracy"], label="Test Accuracy")
    plt.xlabel("Iterations")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.title("Accuracy Over Time")

    plt.show()

# Call function after training
```

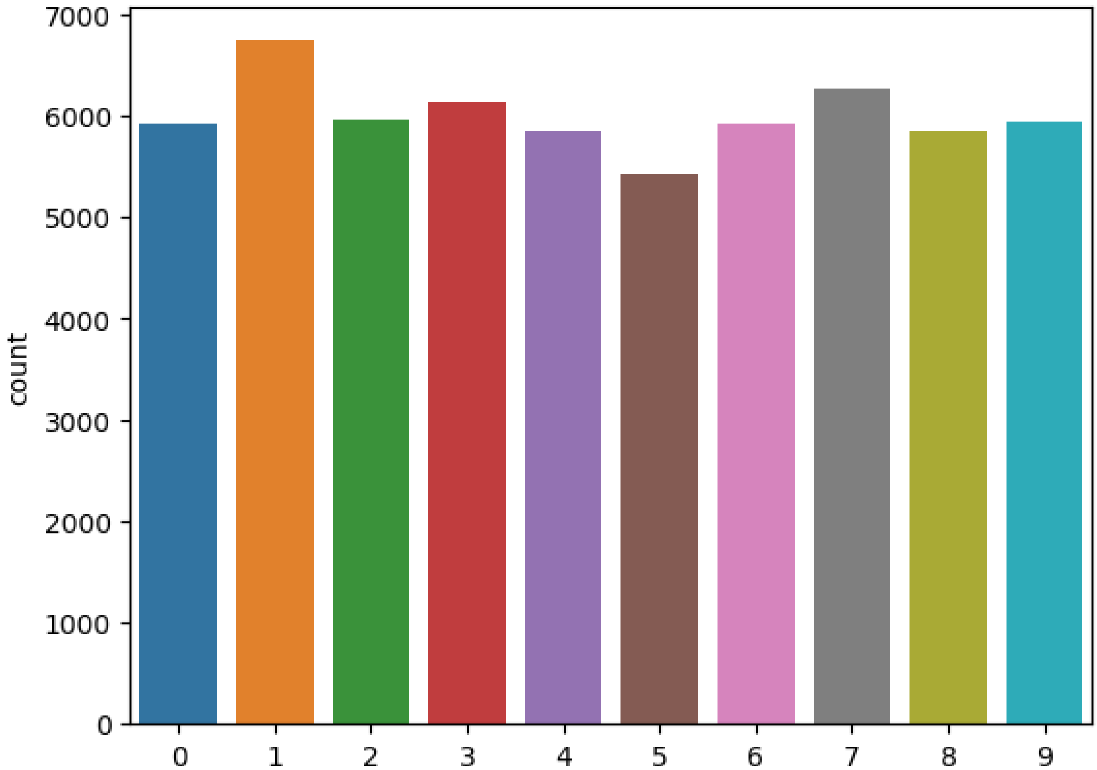
Train on mnist handwritten digits

Load the dataset

- **Dataset link** : https://www.kaggle.com/datasets/senhadjimohamedsaid/mnist-dataset/data

```
# Install dependencies as needed:
# pip install kagglehub[pandas-datasets]
import kagglehub
from kagglehub import KaggleDatasetAdapter

# Set the path to the file you'd like to load
file_path = "mnist_train.csv"
print(file_path)

# Load the latest version
df = kagglehub.load_dataset(
  KaggleDatasetAdapter.PANDAS,
  "senhadjimohamedsaid/mnist-dataset",
  file_path,
  # Provide any additional arguments like
  # sql_query or pandas_kwargs. See the
  # documenation for more information:
  #
https://github.com/Kaggle/kagglehub/blob/main/README.md#kaggledataseta
```

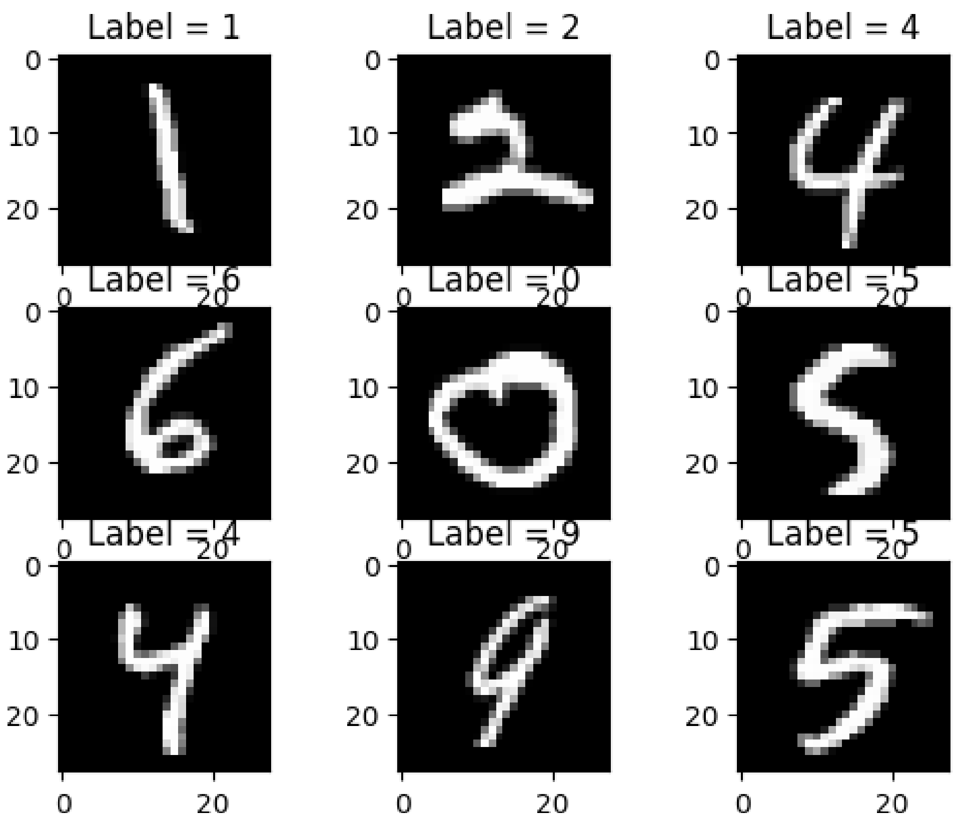
```
dapterpandas
)

print("First 5 records:", df.head())

mnist_train.csv
First 5 records:    label  1x1  1x2  1x3  1x4  1x5  1x6  1x7  1x8  1x9
...  28x19  28x20  \
0      5    0    0    0    0    0    0    0    0    0  ...      0
0
1      0    0    0    0    0    0    0    0    0    0  ...      0
0
2      4    0    0    0    0    0    0    0    0    0  ...      0
0
3      1    0    0    0    0    0    0    0    0    0  ...      0
0
4      9    0    0    0    0    0    0    0    0    0  ...      0
0

   28x21  28x22  28x23  28x24  28x25  28x26  28x27  28x28
0      0      0      0      0      0      0      0      0
1      0      0      0      0      0      0      0      0
2      0      0      0      0      0      0      0      0
3      0      0      0      0      0      0      0      0
4      0      0      0      0      0      0      0      0

[5 rows x 785 columns]

Y = df["label"].values
X = df[df.columns[1:]].values

sns.countplot(x=Y)

<Axes: ylabel='count'>
```



Show some images

```
def plot_random_images(X,Y, nrows=3, ncols=3, real_lables = None):

    fig, axes = plt.subplots(nrows=nrows, ncols=ncols)
    n = nrows * ncols

    idx = np.arange(X.shape[0])
    np.random.shuffle(idx)
    idx = idx[:n]

    X = X[idx]
    Y = Y[idx]

    if real_lables is not None:
        real_lables = real_lables[idx]

    i = 0
    for row in axes:
        for cell in row:
            img = X[i].reshape(28,28)
            cell.imshow(img,cmap="gray")
            if real_lables is None:
                cell.set_title(f"Label = {Y[i]}")
            else:
                cell.set_title(f"Label = {Y[i]} \n Real Label =
```

```
{real_lables[i]}")
            i += 1

plot_random_images(X,Y)
```




```
X = X / 255.0

from sklearn.model_selection import train_test_split

X_train,X_test,y_train,y_test = train_test_split(X, Y,
test_size=0.2,stratify=Y)

params, history, real_grads, approx_grads = train(
    X_train,
    y_train,
    test_set=(X_test,y_test),
    eta=0.01,
    epochs=50,
    batch_size=64,
    nh=64
)
```

{"model_id":"76c567cc2ae844459fcc2128e3a4b716","version_major":2,"vers
ion_minor":0}

```
{"model_id":"eba621d88b394571b5c8b9bcef26a327","version_major":2,"version_minor":0}

{"model_id":"5d962ed9f09841eb80aae1f34873659c","version_major":2,"version_minor":0}

{"model_id":"b2e42f1252e24cde9db49ec15cf0ae22","version_major":2,"version_minor":0}

{"model_id":"a7490fe964bc46459f882f4432a73f5a","version_major":2,"version_minor":0}

{"model_id":"f29c3410338844c3aed9b4d30f0b366f","version_major":2,"version_minor":0}

{"model_id":"367e028ecc00468a941b06ddb32194f1","version_major":2,"version_minor":0}

{"model_id":"e234b8a00a234d2386b8d72bcf3950f9","version_major":2,"version_minor":0}

{"model_id":"d7bf2a50ed8b483ab5f37b995b2f9a5b","version_major":2,"version_minor":0}

{"model_id":"808309ad0fd7469493a2328299397279","version_major":2,"version_minor":0}

{"model_id":"fda22fa68bf94ba89f8b69100f379704","version_major":2,"version_minor":0}

{"model_id":"368e033ffb884ef88b11d086cef02ba2","version_major":2,"version_minor":0}

{"model_id":"303b0a8e52b3441f90df6a0dc8d6a93b","version_major":2,"version_minor":0}

{"model_id":"274bda126f5d487483b1bb6d7f5e2c0c","version_major":2,"version_minor":0}

{"model_id":"702bc6f963f942e2b9a546ee0948e525","version_major":2,"version_minor":0}

{"model_id":"4fbfd12db70a49a4af1c6444d8b97218","version_major":2,"version_minor":0}

{"model_id":"123e782881bf4878863f9365339fdd67","version_major":2,"version_minor":0}

{"model_id":"897c2abfec5a4b25abf0a2959992fa22","version_major":2,"version_minor":0}

{"model_id":"0072d587d95f448c91ef761d0fe25512","version_major":2,"version_minor":0}
```

{"model_id":"99234aef4d0344b4b4bb2a993dbee712","version_major":2,"version_minor":0}

{"model_id":"9c9fb2ec0b704163b50bbf466789a3bb","version_major":2,"version_minor":0}

{"model_id":"b290dc998c724c2f9a8f16692ff73021","version_major":2,"version_minor":0}

{"model_id":"bd6f19c34371464b8efc551d4e3c06d7","version_major":2,"version_minor":0}

{"model_id":"a3f7dc4439ca45ecbc6f0c633e242b6a","version_major":2,"version_minor":0}

{"model_id":"a705cbfb5c054344a06ff72920496505","version_major":2,"version_minor":0}

{"model_id":"fc8bfbcc6b884769b03734381577c36b","version_major":2,"version_minor":0}

{"model_id":"def41e45fd3b4d4580b40a031bb7d6ce","version_major":2,"version_minor":0}

{"model_id":"929775f5b1cc4a69a5e9618b4c7abf3f","version_major":2,"version_minor":0}

{"model_id":"b233da73a9614e1ea47b26b0f46c7920","version_major":2,"version_minor":0}

{"model_id":"1f6c3b5e326949aeb5c8c050cca48cb1","version_major":2,"version_minor":0}

{"model_id":"1cc1b17d84d04d60bc944d8a4a84ade0","version_major":2,"version_minor":0}

{"model_id":"09b76df6fd114af0a855cd26c992af99","version_major":2,"version_minor":0}

{"model_id":"54e263cbc0964d8fa4ffcb3c2229377c","version_major":2,"version_minor":0}

{"model_id":"c5ba6e41f4394cb2a82885003d0bed52","version_major":2,"version_minor":0}

{"model_id":"c4df2e9a9d8248ea83471f77f4b4c49a","version_major":2,"version_minor":0}

{"model_id":"7b7c9b0f161b4d5c917f70e1026c4955","version_major":2,"version_minor":0}

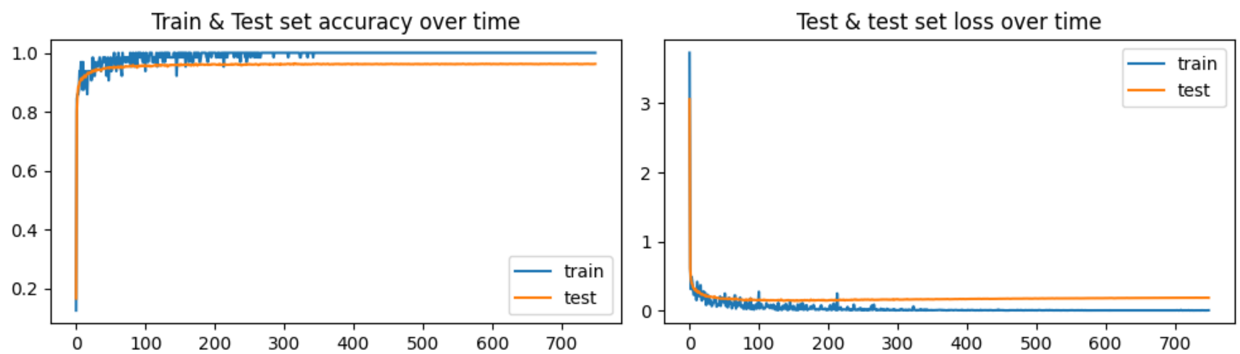
{"model_id":"68edea7227924750a4355062f9124b5b","version_major":2,"version_minor":0}

```
{"model_id":"3ba1dff23a3644f4adeac95952f8d598","version_major":2,"version_minor":0}
{"model_id":"9e79474b231042cc8a064c7e6140c28f","version_major":2,"version_minor":0}
{"model_id":"d5765c81509849f8baced735993c7cba","version_major":2,"version_minor":0}
{"model_id":"02b41fbe3af748b5aca39957fe44cd04","version_major":2,"version_minor":0}
{"model_id":"a6d50b7c8cb54185b5b3d8c02627f4e4","version_major":2,"version_minor":0}
{"model_id":"55d6d896a24e47a192e0a1693f5c2ed0","version_major":2,"version_minor":0}
{"model_id":"971668be80b84792bfe8ccf24c4212ef","version_major":2,"version_minor":0}
{"model_id":"1437734a7e0b4547bb3f18887f0e3d3a","version_major":2,"version_minor":0}
{"model_id":"051503eb1c5d48fab1270aaf157f18cf","version_major":2,"version_minor":0}
{"model_id":"948cb615f2ca4845985d467f1d3198d4","version_major":2,"version_minor":0}
{"model_id":"0dee5eb9926f4fa49d33bd03aca24693","version_major":2,"version_minor":0}
{"model_id":"5da7a905f9c8437a91e02e8a4160ed83","version_major":2,"version_minor":0}
{"model_id":"86d3609fa79c496bbfd9a40b12819d08","version_major":2,"version_minor":0}
```


Learning graph

```
fig, (ax1,ax2) = plt.subplots(nrows=1,ncols=2)

fig.set_size_inches(10,3)

ax1.plot(np.arange(len(history['accuracy'])),history['accuracy'],
label="train")
ax2.plot(np.arange(len(history['loss'])),history['loss'],label="train"
)

ax1.set_title("Train & Test set accuracy over time")
```

```
ax1.plot(np.arange(len(history['test accuracy'])),history['test
accuracy'], label='test')
ax2.plot(np.arange(len(history['test loss'])),history['test loss'],
label="test")

ax2.set_title("Test & test set loss over time")

ax1.legend()
ax2.legend()

plt.tight_layout()
```



Explanation of the Learning Curves

1. **Left Plot (Accuracy Over Time)**
   - The blue curve represents training accuracy.
   - The orange curve represents test accuracy.
   - Both curves increase rapidly at the beginning and then stabilize close to 1.0.
   - This indicates that your model is learning well and generalizing effectively.
2. **Right Plot (Loss Over Time)**
   - The blue curve represents training loss.
   - The orange curve represents test loss.
   - Both losses drop sharply in the early epochs and then stabilize at low values.
   - This suggests the model is minimizing the error successfully.

Observations

- **Good Convergence**: The training and test accuracy stabilize near 1.0, showing that the MLP has effectively learned the digit recognition task.
- **No Overfitting**: The test accuracy remains close to training accuracy, indicating that the model generalizes well.
- **Smooth Loss Reduction**: The loss curves decline steadily without major fluctuations, suggesting stable optimization.

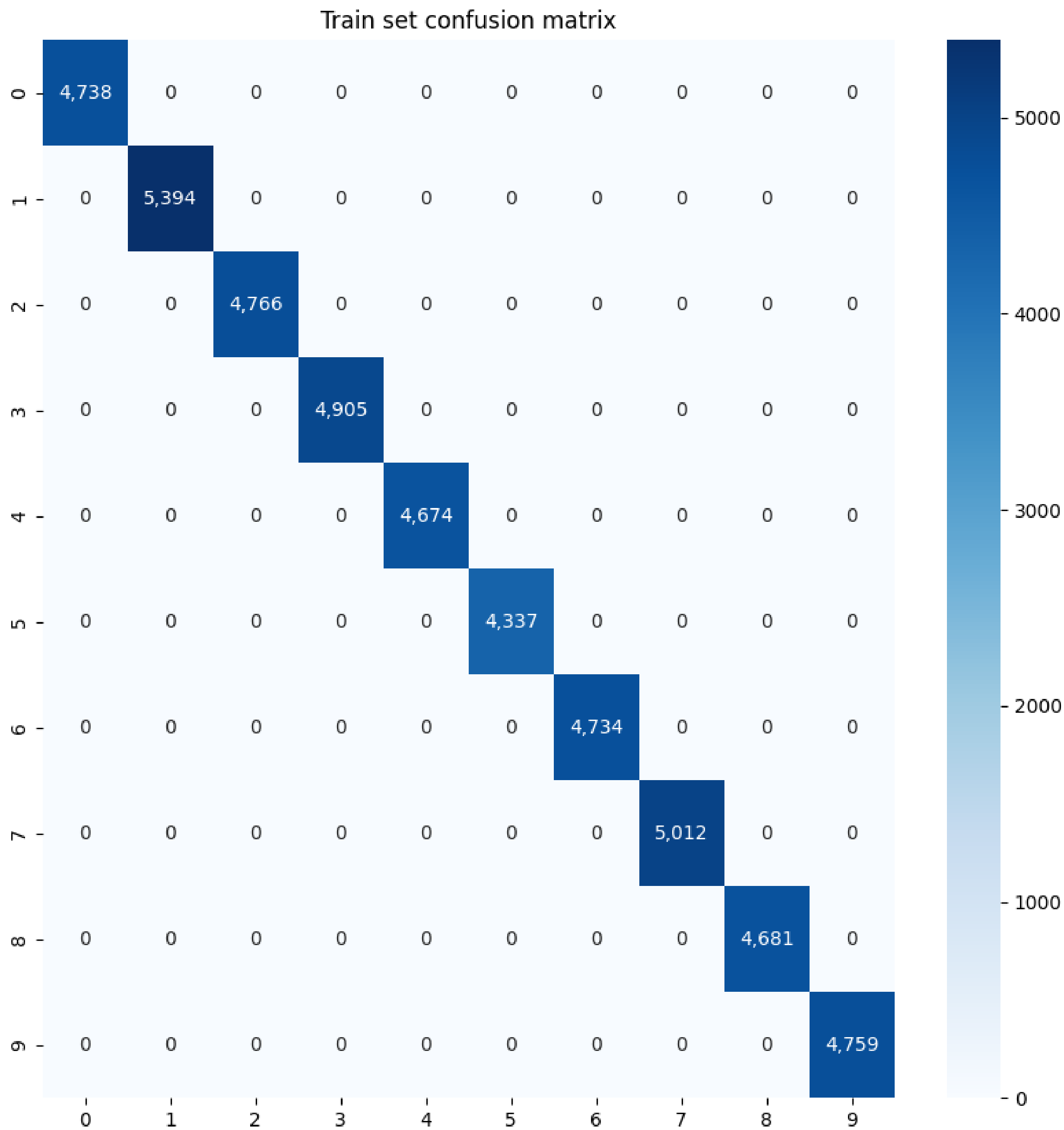
Evaluation

```
from sklearn.metrics import
confusion_matrix,accuracy_score,precision_score,recall_score,f1_score

y_train_hat = predict(params, X_train)
y_test_hat = predict(params, X_test)

def get_matrics(y, y_hat):

    accuracy = accuracy_score(y, y_hat)
    f1 = f1_score(y, y_hat, average="macro")
    precision = precision_score(y, y_hat, average="macro")
    recall = recall_score(y, y_hat, average="macro")

    return pd.Series({
        "accuracy":accuracy,
        "f1_score":f1,
        "precision":precision,
        "recall":recall
    })

train_metrics = get_matrics(y_train, y_train_hat)
test_metrics = get_matrics(y_test, y_test_hat)
metrics = pd.DataFrame(data={
    "train": train_metrics,
    "test":test_metrics
})

metrics
```

	train	test
accuracy	1.0	0.962167
f1_score	1.0	0.961739
precision	1.0	0.961787
recall	1.0	0.961758

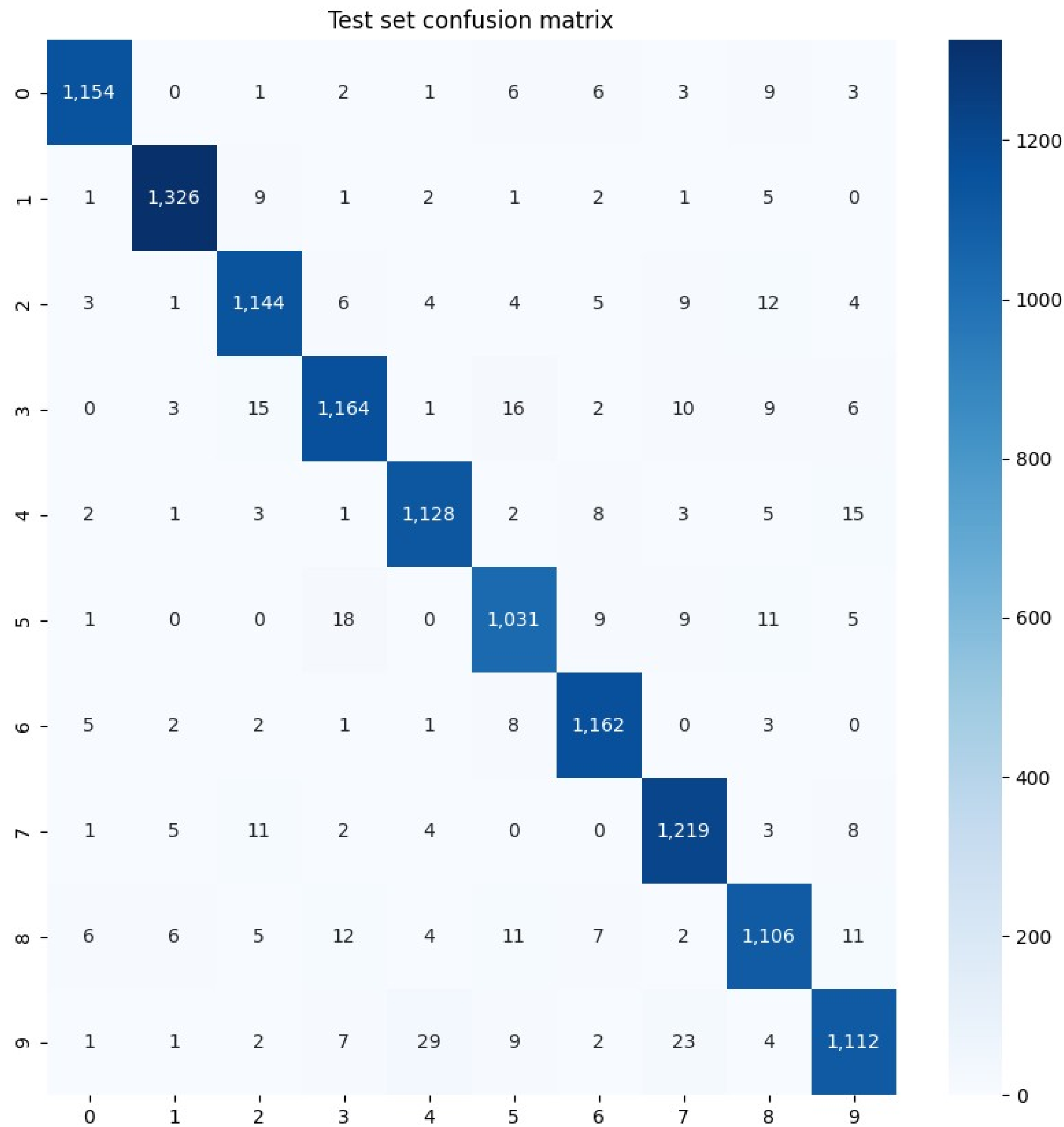
```
def plot_confusion_matrix(y, y_hat):
    cm = confusion_matrix(y, y_hat)
    ax = sns.heatmap(data=cm, annot=True,cmap='Blues', fmt=',d')
    ax.get_figure().set_size_inches(10,10)
    return ax

ax = plot_confusion_matrix(y_train, y_train_hat)
ax.set_title("Train set confusion matrix")
```

```
Text(0.5, 1.0, 'Train set confusion matrix')
```




```
ax = plot_confusion_matrix(y_test, y_test_hat)
ax.set_title("Test set confusion matrix")
```

```
Text(0.5, 1.0, 'Test set confusion matrix')
```



Explanation of the Train and test Set Confusion Matrix

Observations:

- The matrix is nearly diagonal, meaning almost all predictions are correct.
- The numbers on the diagonal indicate that the model correctly classified almost every sample for each digit.
- There are **zero misclassifications**, meaning the model has 100% **accuracy on the training set**.

- Unlike the training set, this matrix is not perfectly diagonal—some misclassifications are present.
- The diagonal elements still contain high values, meaning the majority of the test samples were classified correctly.

Common Misclassifications: Digit 9 → 4, 5 Digit 5 → 3, 8 Digit 3 → 2, 5 Digit 8 → 3, 5 These errors are expected since some handwritten numbers look similar.

Error analysis

```
X_test_ = X_test[y_test_hat != y_test]
y_test_ = y_test[y_test_hat != y_test]
y_test_hat_ = y_test_hat[y_test_hat != y_test]

plot_random_images(X_test_,y_test_hat_,real_lables=y_test_)
plt.tight_layout(pad=0.25)
```