DEEP LEARNING

HANDOUT: CLASSIFICATION

Classification: Hand Written Digital Picture Dataset, Build a model, Error Calculation, Non-Linear model, model complexity, Optimization Method, Hands-On Hand-Written Digital Image Recognition

MODULE 1: Understanding the MNIST Dataset

Key Concepts:

- **MNIST** is a dataset of handwritten digits (0-9)
- Each image is grayscale, 28x28 pixels (784 total pixels)
- Total: 70,000 images (60,000 train, 10,000 test)
- Pixel values range from 0 (black) to 255 (white)

Q1: Why is MNIST popular for deep learning beginners?

A: It is small, clean, well-labeled, and easy to visualize. Perfect for practicing image classification.

Q2: What is the format of the dataset?

A: Images as 2D arrays (28x28), labels as integers from 0 to 9.

Task: Load MNIST using TensorFlow and print the shape of training and test data.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
print(x_train.shape, y_train.shape)
```

MODULE 2: Preprocessing the Data

Raw image data needs to be prepared to improve training performance.

Key Concepts:

a) **Normalization**

- Converts pixel values from 0–255 to [0, 1] or [-1, 1]
- Ensures model converges faster with more stable gradients

b) **One-hot Encoding**

- Converts labels like 3 → [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
- Essential for multi-class classification using softmax or cross-entropy

Q1: Why normalize the input data?

A: Helps speed up training and ensures stable gradients during backpropagation.

Q2: What is one-hot encoding and why is it useful?

A: It converts class labels into binary vectors for compatibility with categorical loss functions.

Task: Normalize the data and apply one-hot encoding.

```
x_train = 2 * tf.cast(x_train, tf.float32)/255. - 1
y_train = tf.one_hot(y_train, depth=10)
```

MODULE 3: Dataset Management & Batching

Key Concepts:

- Training is faster and more efficient when done in **batches**.
- **Batch size** impacts speed and memory usage.

Q1: What is batching and why is it important? A: Divides data into manageable groups for faster, vectorized computation.

Q2: What happens if we don't batch the data? A: Training will be slow and inefficient, especially on large datasets.

Task: Create and batch a TensorFlow dataset:

```
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train)).batch(512)
```

MODULE 4: Linear vs Non-Linear Models

A linear model cannot capture complex image patterns. We introduce non-linearity using activation functions.

Key Concepts:

- **Linear models** cannot capture complex patterns in data
- **Non-linear models** (with activation functions) can
- Introduce non-linearity using **ReLU**, **Sigmoid**, or **Tanh**

Common Activation Functions

Activation	Graph Shape	Output Range	Good For	Limitations
ReLU (Rectified Linear Unit)	f(x) = max(0, x)	[0, ∞)	Hidden layers	Can "die" (output 0 for all inputs if stuck in negative zone)
Sigmoid	f(x) = 1 / (1 + e^-x)	(0, 1)	Binary classification output	Saturates, vanishing gradients for large values
Tanh	f(x) = (e^x - e^-x) / (e^x + e^-x)	(-1, 1)	Centered data; better than sigmoid	Also suffers from vanishing gradients
Softmax	Converts logits to probabilities	(0, 1) for each class	Multi-class classification output	Only for final layer, not hidden layers

When to Use What?

Layer Type	Suggested Activation
Hidden Layer	**ReLU** (most common), or Tanh
Binary Output	**Sigmoid**
Multi-class Output	**Softmax**
Regression Output	**None** or Linear

```
# Hidden Layer using ReLU
tf.keras.layers.Dense(64, activation='relu')
# Binary classification output
tf.keras.layers.Dense(1, activation='sigmoid')
# Multi-class classification output (digits 0–9)
tf.keras.layers.Dense(10, activation='softmax')
```

Q1: What is the limitation of a linear model? A: It cannot model the curved or complex decision boundaries needed for image recognition.

Q2: How does ReLU work? A: ReLU(x) = max(0, x). It passes positive values and blocks negatives.

Task: Try modeling with a single Dense layer (linear), then add ReLU and compare results.

```
# Without activation
Dense(10)
# With ReLU activation
Dense(128, activation='relu')
```

MODULE 5: Building a Deep Neural Network

Key Concepts:

- Deep networks have **input**, **hidden**, and **output layers**
- More layers and neurons = more capacity to learn complex patterns

Q1: What is a hidden layer? A: A layer between input and output that helps the network learn abstract features.

Q2: Why use multiple layers? A: To allow hierarchical learning — each layer captures increasing complexity.

Task: Build a 3-layer network using Keras:

```
model = tf.keras.Sequential([
  tf.keras.layers.Flatten(input_shape=(28,28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(64, activation='relu'),
  tf.keras.layers.Dense(10)
])
```

MODULE 6: Loss Function & Evaluation

Key Concepts:

- **Cross-entropy loss** is ideal for classification
- Accuracy is used to **evaluate**, not to train

Questions & Answers:

Q1: Why not use accuracy as a loss function? A: It is not differentiable, so gradient descent can't optimize it.

Q2: What does cross-entropy do? A: Measures how close predicted probabilities are to actual labels.

Task: Compile model with correct loss function:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

MODULE 7: Overfitting vs Underfitting

Key Concepts:

- **Underfitting**: Model is too simple; fails on training data
- **Overfitting**: Model learns training data too well, fails on new data

Q1: How to detect overfitting?

A: High training accuracy but low validation accuracy.

Q2: How to prevent it?

A: Use regularization, dropout, or more training data.

Task: Add dropout to the model:

```
model = tf.keras.Sequential([
  tf.keras.layers.Flatten(input_shape=(28,28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.3),
  tf.keras.layers.Dense(64, activation='relu'),
  tf.keras.layers.Dense(10)
])
```

MODULE 8: Training & Optimization

Optimizers are algorithms used to update model parameters (weights) during training. They aim to minimize the loss function by moving the weights in the direction of the steepest descent (negative gradient).

Key Concepts:

- Optimizers like **Adam** use gradients to update weights
- TensorFlow handles **backpropagation** automatically

Q1: What is gradient descent?

A: An algorithm that adjusts weights to minimize loss.

Q2: What is auto-differentiation?

A: TensorFlow tracks computations and calculates gradients for us.

Popular Optimizers

Optimizer	Characteristics	Pros	Cons
SGD (Stochastic Gradient Descent)	Basic gradient update	Simple and memory-efficient	Slower convergence, sensitive to learning rate
Momentum	SGD + velocity (past gradient)	Faster than plain SGD	Still requires tuning
RMSProp	Uses moving average of squared gradients	Good for RNNs	Sensitive to learning rate
Adam (Adaptive Moment Estimation)	Combines momentum and RMSProp	Fast convergence, works well in practice	May generalize worse in some tasks

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

```
# SGD

optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)

# RMSProp

optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.001)

# Momentum

optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
```

Task: Train your model and evaluate it:

```
model.fit(train_dataset, epochs=5)

model.evaluate(x_test, tf.one_hot(y_test, depth=10))
```

Module	Topic	Key Concepts	Core Task / Output
1	Handwritten Digital Dataset (MNIST)	28×28 grayscale images, 0–9 digits, 60K train, 10K test, pixel range 0–255	Load dataset using TensorFlow; print data shapes
2	Preprocessing	Normalize pixels to [-1, 1] or [0, 1]; One-hot encode labels	Normalize x_train, x_test; apply one-hot encoding to y_train, y_test
3	Dataset Batching	Group samples for efficient training (e.g., batch size 512)	Use tf.data.Dataset.from_tensor_slices(...).batch(512)
4	Linear vs Non-Linear Models	Linear = simple, weak; Non-linear = powerful (via activation functions like ReLU)	Implement layers with and without activation='relu'; compare learning

Module	Topic	Key Concepts	Core Task / Output
5	Deep Neural Network Architecture	Input → Hidden → Output layers; more neurons = more complexity	Build model with Flatten → Dense → Dropout → Dense → Dense (output)
6	Error Calculation	Accuracy = evaluation metric; Cross-Entropy = loss function for classification	Compile model using SparseCategoricalCrossentropy loss and 'accuracy' as metric
7	Model Complexity / Overfitting Handling	Underfitting: too simple; Overfitting: too complex; Dropout helps generalization	Add Dropout layers; monitor train vs test performance
8	Optimization Methods	Use optimizers like Adam; auto-diff handles gradients	Compile model with optimizer='adam'; train using model.fit()
9	Classification Results & Visualization	Use confusion matrix, classification report, sample predictions	classification_report(), confusion_matrix(), plt.imshow() for true vs predicted image comparison

Hands-On: Hand-Written Digital Image Recognition (MNIST)

```
import tensorflow as tf
import numpy as np
from tensorflow.keras.datasets import mnist
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
```

```
# 1. Load dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
# 2. Normalize data
x_train = x_train / 255.0
x_test = x_test / 255.0
```

```
# 3. Build model
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10)
])
```

```
# 4. Compile model
```

```
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
            metrics=['accuracy'])
```

```
# 5. Train model

model.fit(x_train, y_train, epochs=5, batch_size=512)
```

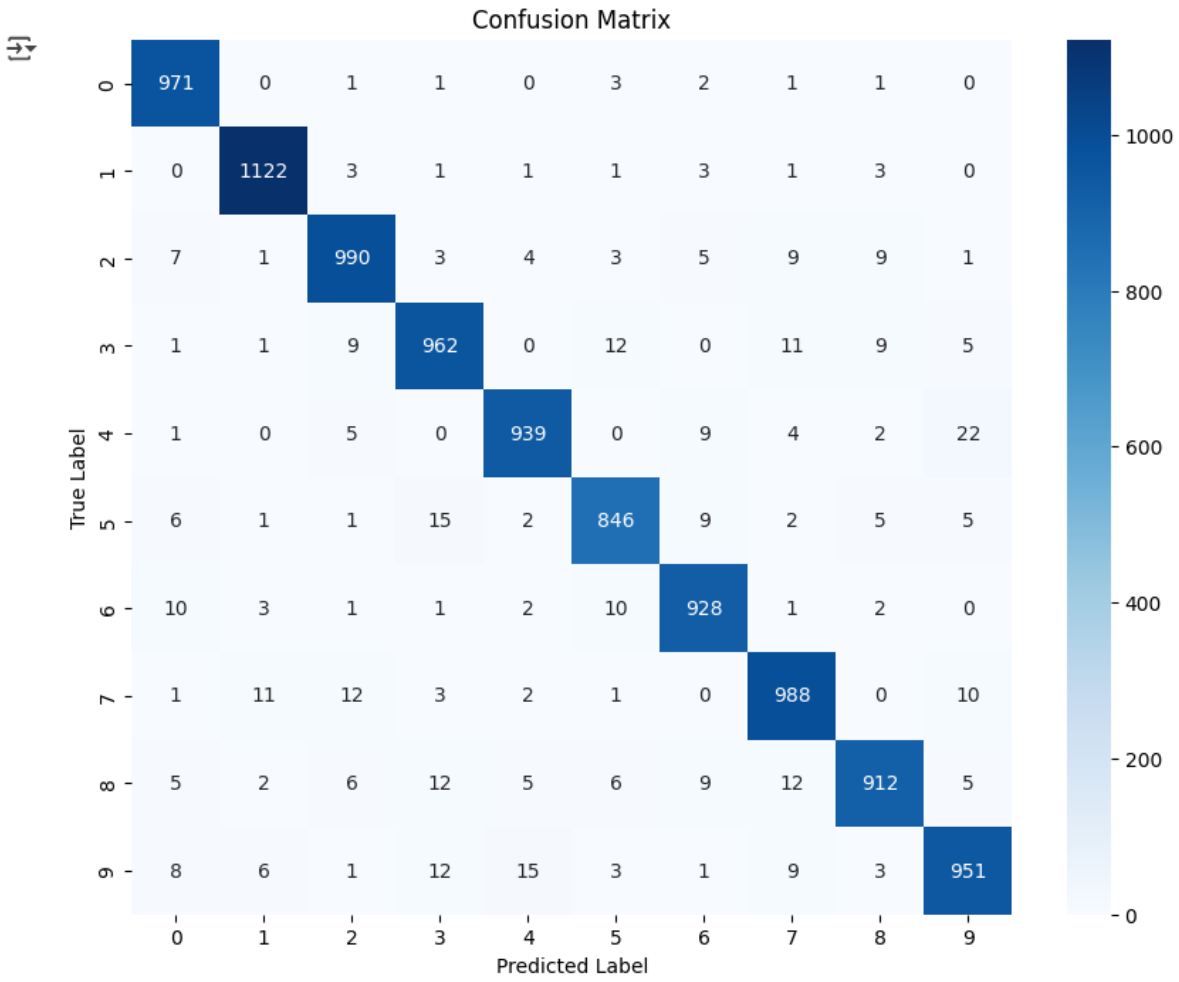
```
# 6. Evaluate model

test_loss, test_acc = model.evaluate(x_test, y_test)

print("Test Accuracy:", test_acc)
```

```
# Get logits and convert to predicted labels

logits = model.predict(x_test)

y_pred = np.argmax(logits, axis=1)
```

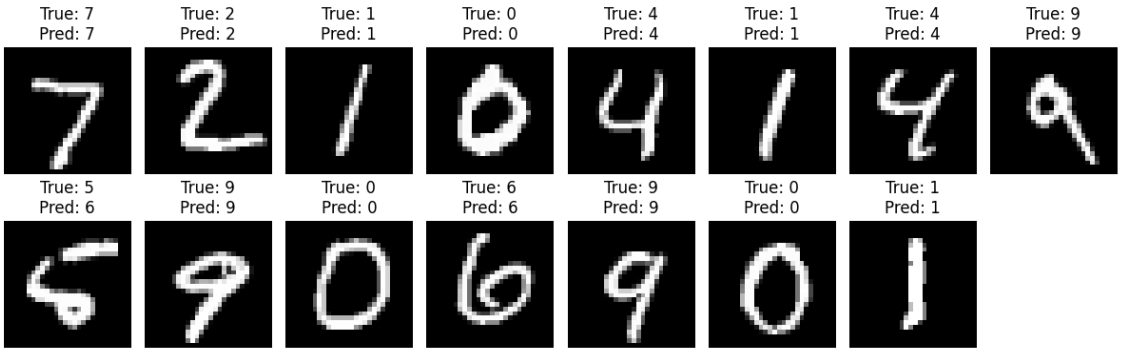
```
Classification Report:
              precision    recall  f1-score   support

           0     0.9614    0.9908    0.9759       980
           1     0.9782    0.9885    0.9833      1135
           2     0.9621    0.9593    0.9607      1032
           3     0.9525    0.9525    0.9525      1010
           4     0.9680    0.9562    0.9621       982
           5     0.9559    0.9484    0.9522       892
           6     0.9607    0.9687    0.9647       958
           7     0.9518    0.9611    0.9564      1028
           8     0.9641    0.9363    0.9500       974
           9     0.9520    0.9425    0.9472      1009

    accuracy                         0.9609     10000
   macro avg     0.9607    0.9604    0.9605     10000
weighted avg     0.9609    0.9609    0.9608     10000
```

```
#7 Classification report

print("Classification Report:")

print(classification_report(y_test, y_pred, digits=4))
```

```
#8. Confusion Matrix Visualization

cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(10, 8))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=range(10),
yticklabels=range(10))

plt.xlabel('Predicted Label')

plt.ylabel('True Label')

plt.title('Confusion Matrix')

plt.show()
```




```
#9. Visualize Sample Predictions

# Visualizing first 15 test images with predictions

plt.figure(figsize=(12, 4))

for i in range(15):

    plt.subplot(2, 8, i+1)

    plt.imshow(x_test[i], cmap='gray')

    plt.title(f"True: {y_test[i]}\nPred: {y_pred[i]}")

    plt.axis('off')

plt.tight_layout()

plt.show()
```



Explore Activations & Optimizers

1. Activation Function Impact

Try replacing ReLU with Tanh or Sigmoid in all hidden layers. How does this affect the training speed and model accuracy?

- Observe:
    - Does training take longer?
    - Do gradients vanish?
    - Is accuracy better or worse?

Note down observations for each activation in a table.

2. Optimizer Comparison

Train the same model using Adam, SGD, and RMSprop separately. Plot and compare accuracy and loss graphs. Which optimizer converges fastest?

- Record:
    - Training loss vs epochs
    - Validation accuracy vs epochs

Which one works best with MNIST? Why might that change for other datasets?

3. Output Layer Activation

Try removing softmax from the final layer and observe the predictions. Then add softmax and compare the predictions. How do the outputs differ in format and interpretability?

- Before softmax: raw logits (e.g., [1.2, 2.1, 0.3,...])
- After softmax: probabilities (e.g., [0.01, 0.85, 0.04,...])

Why is softmax necessary for multi-class classification?

4. Activation Behavior on Edge Cases

What happens when all inputs to ReLU are negative? Replace ReLU with LeakyReLU and check the model's behavior. Do you see any improvement?

- Try:

tf.keras.layers.LeakyReLU(alpha=0.1)

Document whether the network performs better and why LeakyReLU helps.

5. Learning Rate Sensitivity

Choose one optimizer (e.g., SGD) and try changing the learning rate from 0.01 → 0.1 → 1. What happens to loss, accuracy, and convergence behavior?

- Tip: High learning rate may cause the model to diverge!

Create a chart showing learning rate vs accuracy. Which rate worked best?