Task 1 — Install TensorFlow and Keras

Run in terminal / command prompt:

```
python -m pip install --upgrade pip
pip install tensorflow     # installs TF (includes Keras API)
```

Notes:

tensorflow package includes Keras as tf.keras.

If you have a GPU and want GPU support, install proper CUDA/cuDNN and use a compatible TF version (check TensorFlow docs for matching versions).

Task 2 — Perceptron model for binary classification (simple example)

This trains a single-layer perceptron (logistic regression style) on a synthetic binary dataset.

```
# perceptron_binary.py
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from tensorflow.keras import layers, models

# 1) Create synthetic binary classification data
X, y = make_classification(n_samples=2000, n_features=10, n_informative=6,
                    n_redundant=2, n_clusters_per_class=2, random_state=42)

# 2) Train-test split and scaling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test  = scaler.transform(X_test)

# 3) Build simple perceptron model
model = models.Sequential([
  layers.Input(shape=(X_train.shape[1],)),
  layers.Dense(1, activation='sigmoid')   # single neuron -> perceptron (logistic)
])

model.compile(optimizer='adam',
       loss='binary_crossentropy',
       metrics=['accuracy'])

# 4) Train
```

```
history = model.fit(X_train, y_train, epochs=30, batch_size=32, validation_split=0.1, verbose=2)

# 5) Evaluate
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print(f"Test loss: {loss:.4f}, Test accuracy: {acc:.4f}")
```

Explanation: A perceptron with sigmoid outputs probability for class 1; trained with binary cross-entropy.

Task 3 — Multi-Layer Perceptron (MLP) for MNIST

Simple MLP that classifies MNIST handwritten digits using tf.keras.datasets.

```
# mlp_mnist.py
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
import numpy as np

# 1) Load data
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# 2) Preprocess: normalize and flatten
```

```
x_train = x_train.astype('float32') / 255.0
x_test  = x_test.astype('float32') / 255.0
x_train = x_train.reshape((-1, 28*28))
x_test  = x_test.reshape((-1, 28*28))


# 3) Build MLP
model = models.Sequential([
  layers.Input(shape=(28*28,)),
  layers.Dense(128, activation='relu'),
  layers.Dense(64, activation='relu'),
  layers.Dense(10, activation='softmax')
])


model.compile(optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])


# 4) Train
history = model.fit(x_train, y_train, epochs=10, batch_size=128, validation_split=0.1,
verbose=2)


# 5) Evaluate
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"MNIST test acc: {test_acc:.4f}")
```

Tip: You can increase epochs or use dropout/regularization to improve generalization.

Task 4 — Visualize model accuracy and loss graphs using Matplotlib

Use the history object returned by model.fit() to plot.

```
# visualize_history.py  (use after training to plot the history returned by fit)
import matplotlib.pyplot as plt

def plot_history(history):
    # loss
    plt.figure(figsize=(10,4))
    plt.subplot(1,2,1)
    plt.plot(history.history['loss'], label='train_loss')
    plt.plot(history.history.get('val_loss', []), label='val_loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Loss vs Epoch')

    # accuracy
    plt.subplot(1,2,2)
    plt.plot(history.history.get('accuracy', history.history.get('acc')), label='train_acc')
    plt.plot(history.history.get('val_accuracy', history.history.get('val_acc', [])), label='val_acc')
```

```
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.title('Accuracy vs Epoch')

    plt.tight_layout()
    plt.show()

# Example usage (after training): plot_history(history)
```

If using this in a notebook, plt.show() will render plots inline.

Task 5 — Perform hyperparameter tuning (change epochs, batch size, activation functions)

Simple manual grid search example (train small model multiple times and compare). For quick experiments, use a small subset or fewer epochs. For production-scale tuning, use keras-tuner.

```
# simple_hyperparam_search.py
import itertools
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
import numpy as np
```

```
# use a small subset to speed up grid search in examples
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test  = x_test.astype('float32') / 255.0
x_train = x_train.reshape((-1, 28*28))
x_test  = x_test.reshape((-1, 28*28))

# Use a small subset for quick experiments
x_train_small = x_train[:20000]
y_train_small = y_train[:20000]
x_val = x_train[20000:25000]
y_val = y_train[20000:25000]

def build_model(activation='relu'):
    model = models.Sequential([
        layers.Input(shape=(28*28,)),
        layers.Dense(128, activation=activation),
        layers.Dense(64, activation=activation),
        layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

```python
# hyperparameters to try
param_grid = {
    'epochs': [5, 10],
    'batch_size': [64, 128],
    'activation': ['relu', 'tanh']
}

results = []
for epochs, batch_size, activation in itertools.product(param_grid['epochs'],
                                                        param_grid['batch_size'],
                                                        param_grid['activation']):
    print(f"Training: epochs={epochs}, batch_size={batch_size}, activation={activation}")
    model = build_model(activation=activation)
    hist = model.fit(x_train_small, y_train_small, epochs=epochs, batch_size=batch_size,
                     validation_data=(x_val, y_val), verbose=0)
    val_acc = hist.history['val_accuracy'][-1]
    results.append({
        'epochs': epochs, 'batch_size': batch_size, 'activation': activation,
        'val_acc': val_acc
    })
    print(f" -> val_acc={val_acc:.4f}")

# show best
best = max(results, key=lambda r: r['val_acc'])
print("Best config:", best)
```

Alternative: Use keras-tuner for a more principled search (random search / Bayesian optimization).

Task 6 — Short blog/note explaining how neural networks learn

English (short):

Neural networks learn by adjusting weights to reduce the difference between predicted outputs and target outputs. During training:

1. A forward pass computes predictions from inputs through layers (matrices multiply and nonlinear activations).

2. A loss (error) function quantifies how far predictions are from true labels (e.g., cross-entropy).

3. Backpropagation computes gradients of the loss w.r.t. each weight using the chain rule.

4. An optimizer (e.g., SGD, Adam) updates weights in the direction that reduces loss using those gradients.

5. Repeat over many examples (epochs) until the network generalizes well.

Key concepts:

Activation functions (ReLU, sigmoid, tanh) add nonlinearity so networks can learn complex mappings.

Learning rate controls step size of updates; too large → divergence; too small → slow training.

Overfitting vs generalization: networks can memorize training data — regularization (dropout, weight decay), early stopping, and more data help generalize.

Batch size affects gradient estimate quality: small batches are noisy but can generalize well; large batches give stable gradients.