Double-click (or enter) to edit

You need to work on a popular Fashion MNIST dataset for this HW. The dataset includes tiny images of fashion pieces. The objective is to create a set of supervised learning models that can predict the type of item based on its image.

In order to load the dataset you need to have `tensorflow V2` on your computer. Use the following code to install the package

```
# !pip install --upgrade tensorflow
```

You can also check the version of it using the following code.

```
import tensorflow as tf
tf.__version__
```

```
'2.19.0'
```

Now, it's time to load the dataset

```
from tensorflow import keras
fashion_mnist = keras.datasets.fashion_mnist
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

As can be seen from the above code, the dataset was divided into train and test sets. Let's take a look at the X_train

```
X_train.shape
```

```
(60000, 28, 28)
```

As it is clear, the train dataset (`X_train`) contains 60,000 images of size 28 x 28. We can visualize one of the images using the following code:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
%matplotlib inline

sample_image = X_train[10]
plt.imshow(sample_image, cmap='binary')
plt.axis('off')
plt.show()
```

...

The `y_train` also includes values between 0 and 9. Each represents a particular category. For example, we can check the value of `y_train` for the above image.

```
y_train[10]
```

...

The above code shows that the image belongs to category 0. To get the associated label with each category, you can use the following code:

```
class_names = ['T-shirt/top','Trouser','Pullover','Dress','Coat','Sandal','Shirt','Sneaker','Bag','Ankle boot']
print(class_names[y_train[10]])
```

...

```
Start coding or generate with AI.
```

- **Task1:** Use the train set to train various supervised models and evaluate their performance using the test set.
  - Use different supervised learning models.
  - Use different metrics such as **accuracy**, **precision**, **AUC**, and ... in your model evaluation.
  - It is not enough to report the metrics. It is crucial that you interpret the metrics for each model and compare them across different models.
  - You may need to use the cross validation methods for hyperparameter selection.
  - Specify the model that outperforms the other models.
- **Task2:** Use the best model to predict your own fashion pieces.
  - Take a picture of ten fashion pieces of your own (take pictures in square format).
  - Resize images to the correct size (28,28).
  - Grayscale your images.
  - Visualize all the images side by side
  - Use the best model in Task 1 to predict the label of each of your own images.
  - How accurate is the final result?

Output

- Make sure to put descriptive comments on your code
- Use the markdown cell format in Jupiter to add your own interpretation to the result in each section.
- Make sure to keep the output of your runs when you want to save the final version of the file.
- The final work should be very well structured and should have a consistent flow of analysis.

0. Setup

```
# %pip -q install -U tensorflow scikit-learn matplotlib seaborn pillow

import sys, platform, numpy as np, matplotlib.pyplot as plt, seaborn as sns
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.metrics import accuracy_score, precision_score, classification_report, confusion_matrix, roc_auc_score
from sklearn.model_selection import train_test_split, GridSearchCV, StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from PIL import Image
import os, glob

print("Python:", sys.version)
print("Platform:", platform.platform())
print("TensorFlow:", tf.__version__)
```

...

1. Load Fashion-MNIST and Basic EDA

```
# Load Fashion-MNIST
(X_train, y_train), (X_test, y_test) = keras.datasets.fashion_mnist.load_data()

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

...

```
# Visualize a few samples
class_names = ['T-shirt/top','Trouser','Pullover','Dress','Coat',
               'Sandal','Shirt','Sneaker','Bag','Ankle boot']

plt.figure(figsize=(10,3))
for i in range(10):
    plt.subplot(2,5,i+1)
    plt.imshow(X_train[i], cmap='binary')
    plt.title(class_names[y_train[i]], fontsize=9)
    plt.axis('off')
plt.tight_layout()
plt.show()
```

...

2. Preprocessing

We'll prepare two pipelines:

- For classical ML (LogReg/SVM/RF): flatten 28x28 to 784 features; scale if needed.
- For CNN: keep 28x28x1, normalize to.

```
# Normalization
X_train_cnn = (X_train.astype("float32") / 255.0)[..., np.newaxis]
X_test_cnn  = (X_test.astype("float32") / 255.0)[..., np.newaxis]

# Flatten for classical ML
X_train_flat = X_train.reshape(len(X_train), -1).astype("float32")/255.0
X_test_flat  = X_test.reshape(len(X_test), -1).astype("float32")/255.0

num_classes = 10
```

3. Baselines and Classical Models (Task 1)

We will evaluate:

- Logistic Regression (multinomial)
- Linear SVM and RBF SVM (using probabilities via probability=True)
- Random Forest

We'll compute accuracy, macro-precision, and OvR ROC-AUC (requires predict_proba). For SVM linear we'll do two runs: one with calibrated probs via probability=True (note the slower training) or use decision_function with OneVsRestClassifier+CalibratedClassifierCV if desired. To keep runtime sane, we'll limit grid sizes.

```
def evaluate_model(clf, X_tr, y_tr, X_te, y_te, model_name):
    y_proba = None
    if hasattr(clf, "predict_proba"):
        y_proba = clf.predict_proba(X_te)
    elif hasattr(clf, "decision_function"):
```

```
        # Convert decision scores to probabilities via softmax-like normalization per class
        # Note: not a calibrated probability; acceptable for a rough AUC comparison if needed
        scores = clf.decision_function(X_te)
        # If binary shape, convert to two-column
        if scores.ndim == 1:
            scores = np.vstack([-scores, scores]).T
        # Min-max per row to 0..1
        s_min = scores.min(axis=1, keepdims=True)
        s_max = scores.max(axis=1, keepdims=True)
        denom = np.where((s_max - s_min)==0, 1, (s_max - s_min))
        y_proba = (scores - s_min) / denom

    y_pred = clf.predict(X_te)

    acc = accuracy_score(y_te, y_pred)
    prec_macro = precision_score(y_te, y_pred, average='macro', zero_division=0)

    auc_ovr = None
    if y_proba is not None:
        # One-vs-Rest multiclass AUC with macro averaging
        y_true_bin = keras.utils.to_categorical(y_te, num_classes=num_classes)
        try:
            auc_ovr = roc_auc_score(y_true_bin, y_proba, average='macro', multi_class='ovr')
        except Exception:
            auc_ovr = None

    print(f"{model_name} — Accuracy: {acc:.4f} | Macro-Precision: {prec_macro:.4f} | OvR AUC: {auc_ovr if auc_ovr is not
    return {"model": model_name, "acc": acc, "prec_macro": prec_macro, "auc_ovr": auc_ovr}

results = []
```

```
results
```

...

3.1 Logistic Regression (with small grid search)

```
pipe_lr = Pipeline([
    ("scaler", StandardScaler(with_mean=False)),  # with_mean=False safer for sparse-like; here dense but fine
    ("clf", LogisticRegression(max_iter=200, multi_class='multinomial', solver='saga', n_jobs=-1))
])

param_lr = {
    "clf__C": [1.0, 0.5, 2.0]
}

cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
grid_lr = GridSearchCV(pipe_lr, param_lr, cv=cv, scoring='accuracy', n_jobs=-1, verbose=0)
grid_lr.fit(X_train_flat, y_train)
best_lr = grid_lr.best_estimator_
print("LR best params:", grid_lr.best_params_)

res_lr = evaluate_model(best_lr, X_train_flat, y_train, X_test_flat, y_test, "LogisticRegression")
results.append(res_lr)
```

...

3.2 Support Vector Machines

Linear SVM (probability=True uses Platt scaling internally; slower):

```
pipe_svm_lin = Pipeline([
    ("scaler", StandardScaler(with_mean=False)),
    ("clf", SVC(kernel='linear', probability=True))
])

param_svm_lin = {"clf__C": [0.5, 1.0]}
grid_svm_lin = GridSearchCV(pipe_svm_lin, param_svm_lin, cv=cv, scoring='accuracy', n_jobs=-1, verbose=0)
grid_svm_lin.fit(X_train_flat, y_train)
best_svm_lin = grid_svm_lin.best_estimator_
print("SVM-linear best params:", grid_svm_lin.best_params_)

```

```
res_svm_lin = evaluate_model(best_svm_lin, X_train_flat, y_train, X_test_flat, y_test, "SVM-Linear")
results.append(res_svm_lin)

```

...

RBF SVM:

```
pipe_svm_rbf = Pipeline([
    ("scaler", StandardScaler(with_mean=False)),
    ("clf", SVC(kernel='rbf', probability=True))
])

param_svm_rbf = {"clf__C": [1.0], "clf__gamma": ['scale']}
grid_svm_rbf = GridSearchCV(pipe_svm_rbf, param_svm_rbf, cv=cv, scoring='accuracy', n_jobs=-1, verbose=0)
grid_svm_rbf.fit(X_train_flat, y_train)
best_svm_rbf = grid_svm_rbf.best_estimator_
print("SVM-RBF best params:", grid_svm_rbf.best_params_)

res_svm_rbf = evaluate_model(best_svm_rbf, X_train_flat, y_train, X_test_flat, y_test, "SVM-RBF")
results.append(res_svm_rbf)

```

...

3.3 Random Forest

```
rf = RandomForestClassifier(
    n_estimators=300, max_depth=None, n_jobs=-1, random_state=42, class_weight=None
)
rf.fit(X_train_flat, y_train)
res_rf = evaluate_model(rf, X_train_flat, y_train, X_test_flat, y_test, "RandomForest")
results.append(res_rf)

```

...

3.4 Compare Classical Models

```
import pandas as pd
df_results = pd.DataFrame(results).sort_values(by="acc", ascending=False)
df_results

```

...

add confusion matrices:

```
best_name = df_results.iloc[0]["model"]
best_clf = {
    "LogisticRegression": best_lr,
    "SVM-Linear": best_svm_lin,
    "SVM-RBF": best_svm_rbf,
    "RandomForest": rf
}[best_name]

y_pred_best = best_clf.predict(X_test_flat)
cm = confusion_matrix(y_test, y_pred_best)
plt.figure(figsize=(7,6))
sns.heatmap(cm, annot=False, cmap='Blues', xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted"); plt.ylabel("True"); plt.title(f"Confusion Matrix — {best_name}")
plt.tight_layout(); plt.show()

```

```
print(classification_report(y_test, y_pred_best, target_names=class_names))
```

...

4. CNN Model (Task 1, Deep Learning)

A compact CNN often outperforms classical models on Fashion-MNIST.

```
def make_cnn():
    inputs = keras.Input(shape=(28,28,1))
    x = layers.Conv2D(32, (3,3), activation='relu')(inputs)
    x = layers.MaxPooling2D()(x)
    x = layers.Conv2D(64, (3,3), activation='relu')(x)
    x = layers.MaxPooling2D()(x)
    x = layers.Flatten()(x)
    x = layers.Dropout(0.3)(x)
    x = layers.Dense(128, activation='relu')(x)
    outputs = layers.Dense(num_classes, activation='softmax')(x)
    model = keras.Model(inputs, outputs)
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

cnn = make_cnn()
history = cnn.fit(
    X_train_cnn, y_train,
    validation_split=0.1,
    epochs=8,
    batch_size=128,
    verbose=1
)

test_loss, test_acc = cnn.evaluate(X_test_cnn, y_test, verbose=0)
print(f"CNN — Accuracy: {test_acc:.4f}")

```

```
y_proba_cnn = cnn.predict(X_test_cnn, verbose=0)
y_pred_cnn = np.argmax(y_proba_cnn, axis=1)

prec_macro_cnn = precision_score(y_test, y_pred_cnn, average='macro', zero_division=0)
y_true_bin = keras.utils.to_categorical(y_test, num_classes=num_classes)
auc_ovr_cnn = roc_auc_score(y_true_bin, y_proba_cnn, average='macro', multi_class='ovr')

print(f"CNN — Macro-Precision: {prec_macro_cnn:.4f} | OvR AUC: {auc_ovr_cnn:.4f}")
```

Add a markdown interpretation comparing CNN vs the best classical model:

Accuracy and macro-precision differences

CNN's ability to capture spatial structure vs flattened features

Optional: confusion matrix for CNN.

```
cm_cnn = confusion_matrix(y_test, y_pred_cnn)
plt.figure(figsize=(7,6))
sns.heatmap(cm_cnn, annot=False, cmap='Greens', xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted"); plt.ylabel("True"); plt.title("Confusion Matrix — CNN")
plt.tight_layout(); plt.show()

print(classification_report(y_test, y_pred_cnn, target_names=class_names))
```

5. Pick Best Model

```
# Consolidate with CNN metrics
row_cnn = {"model":"CNN", "acc": float(test_acc), "prec_macro": float(prec_macro_cnn), "auc_ovr": float(auc_ovr_cnn)}
df_all = pd.concat([df_results, pd.DataFrame([row_cnn])]).sort_values(by="acc", ascending=False)
df_all
```

Add markdown:

State the best model (likely CNN by accuracy and AUC).

Briefly justify selection and note any trade-offs (training time vs performance).

If the classical model somehow wins (rare), proceed with that instead for Task 2.

6. Task 2: Predict On Your Own 10 Images

Instructions:

Take 10 square photos (one per class if possible).

Save as PNG/JPG in a folder.

The pipeline: load → convert to grayscale → resize to 28x28 → normalize → shape to (28,28,1) → batch predict.

```
# Path with 10 images
user_img_dir = "YOUR_10_IMAGES_FOLDER"  # e.g., "my_fashion_photos"
paths = sorted(glob.glob(os.path.join(user_img_dir, "*.*")))
assert len(paths) >= 10, f"Found {len(paths)} images, need at least 10."

def load_and_prepare_image(p, target=(28,28)):
    img = Image.open(p).convert("L")   # grayscale
    img = img.resize(target)           # ensure (28,28)
    arr = np.array(img, dtype="float32")/255.0
    return arr

user_imgs = [load_and_prepare_image(p) for p in paths[:10]]
user_stack_cnn = np.stack(user_imgs)[..., np.newaxis]  # (N,28,28,1)
```

Visualize side-by-side:

```
plt.figure(figsize=(10,3))
for i, (p, arr) in enumerate(zip(paths[:10], user_imgs)):
    plt.subplot(2,5,i+1)
    plt.imshow(arr, cmap='binary')
    plt.title(os.path.basename(p)[:12], fontsize=8)
```

```
    plt.axis('off')
plt.tight_layout()
plt.show()

```

Predict with the best model (choose CNN if it won):

```
use_cnn = df_all.iloc[0]["model"] == "CNN"

if use_cnn:
    probs = cnn.predict(user_stack_cnn, verbose=0)
    preds = np.argmax(probs, axis=1)
else:
    # If a classical model won:
    user_stack_flat = user_stack_cnn.reshape(len(user_imgs), -1)
    best_name = df_all.iloc[0]["model"]
    best_clf = {
        "LogisticRegression": best_lr,
        "SVM-Linear": best_svm_lin,
        "SVM-RBF": best_svm_rbf,
        "RandomForest": rf
    }[best_name]
    preds = best_clf.predict(user_stack_flat)

pred_labels = [class_names[i] for i in preds]
pred_labels

```

If you have ground-truth labels for your images (optional), compute accuracy:

```
# Optional: set your true labels if known (list of 10 strings matching class_names)
# Example: y_true_names = ['Sneaker','Bag',...]
y_true_names = None  # replace with actual list

if y_true_names is not None:
    true_idx = [class_names.index(n) for n in y_true_names]
    pred_acc = accuracy_score(true_idx, preds)
    print("User images accuracy:", pred_acc)

```

7. Short Discussion and Conclusions

Use markdown to:

Summarize which model outperformed and by how much.

Interpret metric differences (accuracy vs macro-precision vs AUC).

Mention practical considerations (speed, simplicity, deployment).

Note improvements: data augmentation, deeper CNN (BatchNorm, more epochs), regularization, or transfer learning variants.