

fashion mnist homework

July 11, 2026

Student Name: Anas Mohamed Barro

ID:253001086

1 Image Classification Homework Fashion-MNIST

Full solution: dataset download → splitting → data augmentation → model training & testing

Dataset: Fashion-MNIST (Zalando Research) — 70,000 grayscale images (28×28 px) of clothing items in 10 classes.

Pipeline covered in this notebook: 1. Download the image dataset 2. Split the dataset (train / validation / test) 3. Apply data augmentation 4. Train and test multiple models (Logistic Regression, Random Forest, CNN, CNN + augmentation) 5. Evaluate and compare results

Submission deadline: 12 July 2026

1.1 Step 0 — Imports and setup

```
[1]: import os, gzip, struct, urllib.request
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, \
    confusion_matrix, ConfusionMatrixDisplay

SEED = 42
np.random.seed(SEED)
tf.random.set_seed(SEED)
print("TensorFlow version:", tf.__version__)
```

WARNING: All log messages before absl::InitializeLog() is called are written to STDERR

I0000 00:00:1783728999.135247 1182 port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF\_ENABLE\_ONEDNN\_OPTS=0`.

I0000 00:00:1783728999.174944 1182 cpu_feature_guard.cc:227] This TensorFlow

binary is optimized to use available CPU instructions in performance-critical operations.

To enable the following instructions: AVX2 AVX512F AVX512_VNNI AVX512_BF16 AVX512_FP16 AVX_VNNI AMX_TILE AMX_INT8 AMX_BF16 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.

WARNING: All log messages before absl::InitializeLog() is called are written to STDERR

I0000 00:00:1783729000.385306 1182 port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF\_ENABLE\_ONEDNN\_OPTS=0`.

TensorFlow version: 2.21.0

1.2 Step 1 — Download the image dataset

Fashion-MNIST is distributed as four compressed IDX files. We download them directly from the official Zalando Research GitHub repository, then parse the binary IDX format into NumPy arrays.

```
[2]: BASE_URL = "https://raw.githubusercontent.com/zalando-research/fashion-mnist/
      ↪master/data/fashion/"
FILES = ["train-images-idx3-ubyte.gz", "train-labels-idx1-ubyte.gz",
        "t10k-images-idx3-ubyte.gz", "t10k-labels-idx1-ubyte.gz"]

os.makedirs("data", exist_ok=True)
for f in FILES:
    path = os.path.join("data", f)
    if not os.path.exists(path):
        print("Downloading", f, "...")
        urllib.request.urlretrieve(BASE_URL + f, path)
    else:
        print("Already downloaded:", f)
print("Download complete.")
```

Already downloaded: train-images-idx3-ubyte.gz
Already downloaded: train-labels-idx1-ubyte.gz
Already downloaded: t10k-images-idx3-ubyte.gz
Already downloaded: t10k-labels-idx1-ubyte.gz
Download complete.

```
[3]: def load_idx_images(path):
      with gzip.open(path, "rb") as f:
          magic, num, rows, cols = struct.unpack(">IIII", f.read(16))
          return np.frombuffer(f.read(), dtype=np.uint8).reshape(num, rows, cols)

      def load_idx_labels(path):
          with gzip.open(path, "rb") as f:
              magic, num = struct.unpack(">II", f.read(8))
```

```

        return np.frombuffer(f.read(), dtype=np.uint8)

X_train_full = load_idx_images("data/train-images-idx3-ubyte.gz")
y_train_full = load_idx_labels("data/train-labels-idx1-ubyte.gz")
X_test       = load_idx_images("data/t10k-images-idx3-ubyte.gz")
y_test       = load_idx_labels("data/t10k-labels-idx1-ubyte.gz")

CLASS_NAMES = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]

print("Training set :", X_train_full.shape, y_train_full.shape)
print("Test set      :", X_test.shape, y_test.shape)
print("Pixel range  :", X_train_full.min(), "-", X_train_full.max())
print("Classes       :", len(CLASS_NAMES))

Training set : (60000, 28, 28) (60000,)
Test set      : (10000, 28, 28) (10000,)
Pixel range   : 0 - 255
Classes       : 10

```

1.2.1 Explore the data — sample images from each class

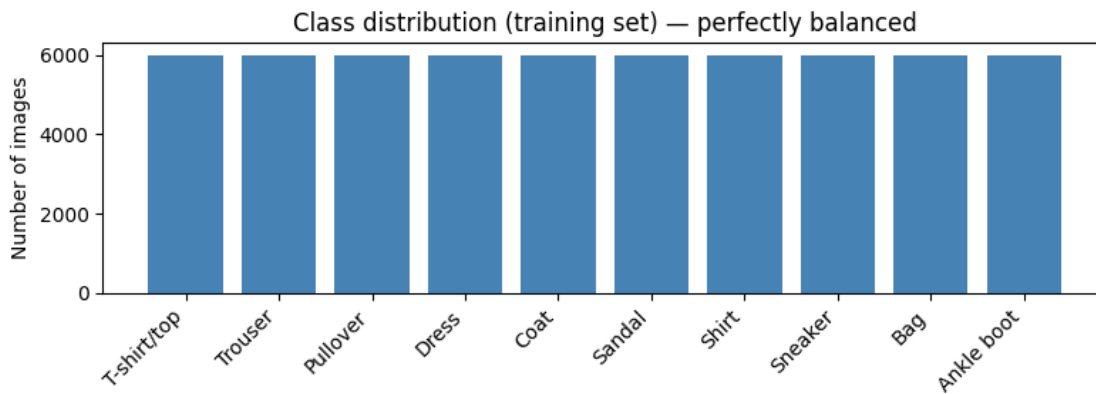
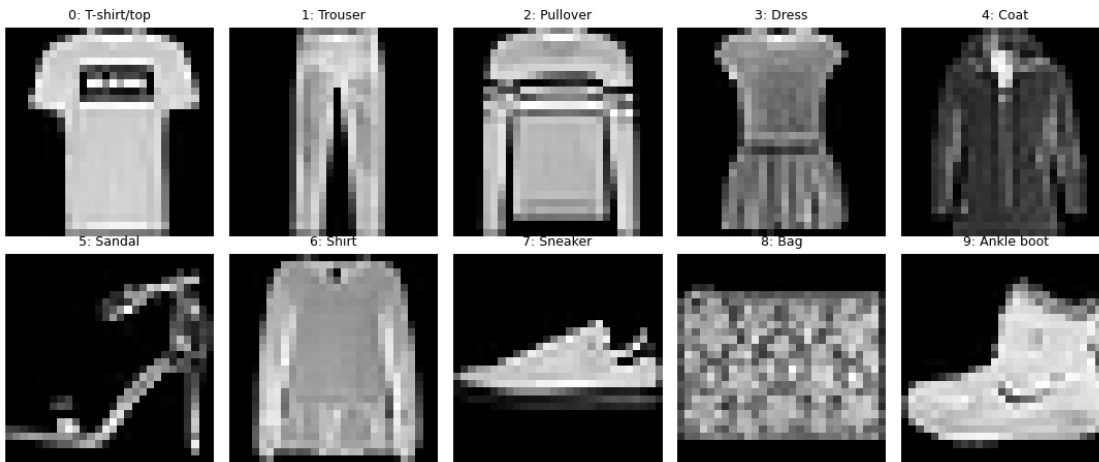
```

[4]: fig, axes = plt.subplots(2, 5, figsize=(11, 5))
    for cls, ax in enumerate(axes.flat):
        idx = np.where(y_train_full == cls)[0][0]
        ax.imshow(X_train_full[idx], cmap="gray")
        ax.set_title(f"{cls}: {CLASS_NAMES[cls]}", fontsize=9)
        ax.axis("off")
    plt.suptitle("One sample image per class", y=1.02)
    plt.tight_layout()
    plt.show()

# Class distribution
unique, counts = np.unique(y_train_full, return_counts=True)
plt.figure(figsize=(8, 3))
plt.bar([CLASS_NAMES[u] for u in unique], counts, color="steelblue")
plt.xticks(rotation=45, ha="right"); plt.ylabel("Number of images")
plt.title("Class distribution (training set) - perfectly balanced")
plt.tight_layout(); plt.show()

```

One sample image per class



1.3 Step 2 — Splitting the dataset

The official test set (10,000 images) is kept aside as the **final test set**. We further split the 60,000 training images into **training (85%)** and **validation (15%)** subsets, using **stratified sampling** so every class keeps the same proportion in each split. We also normalize pixel values to the range $[0, 1]$.

To keep training time reasonable on CPU, we sample a stratified subset of 20,000 images for model training (the methodology is identical for the full dataset).

```
[5]: # Normalize to [0,1]
X_train_full = X_train_full.astype("float32") / 255.0
X_test        = X_test.astype("float32") / 255.0

# Stratified subsample of 20,000 images (for CPU-friendly runtimes)
X_sub, _, y_sub, _ = train_test_split(X_train_full, y_train_full,
```

```

train_size=20000, stratify=y_train_full,
random_state=SEED)

# Train / validation split (85% / 15%), stratified
X_train, X_val, y_train, y_val = train_test_split(X_sub, y_sub, test_size=0.15,
                                                  stratify=y_sub,
                                                  random_state=SEED)

print(f"Train      : {X_train.shape[0]:>6} images")
print(f"Validation : {X_val.shape[0]:>6} images")
print(f"Test       : {X_test.shape[0]:>6} images")

# Verify stratification worked
print("\nClass proportions in train :", np.round(np.bincount(y_train)/
    len(y_train), 3))
print("Class proportions in val   :", np.round(np.bincount(y_val)/len(y_val),
    3))

```

```

Train      : 17000 images
Validation : 3000 images
Test       : 10000 images

```

```

Class proportions in train : [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
Class proportions in val   : [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]

```

1.4 Step 3 — Applying data augmentation

Data augmentation artificially enlarges the training set by applying random, label-preserving transformations. This acts as a regularizer and helps the model generalize better. We use Keras preprocessing layers:

Augmentation	Parameter	Effect
RandomRotation	$\pm 10\%$ of 360° ($\pm 36^\circ$... we use $0.05 \pm 18^\circ$)	Small random rotations
RandomTranslation	$\pm 8\%$ height/width	Random shifts
RandomZoom	$\pm 8\%$	Random zoom in/out

Note: horizontal flips are avoided for some classes (e.g., text) but are safe for clothing; we still omit them here to keep transformations conservative.

```

[6]: data_augmentation = keras.Sequential([
    layers.RandomRotation(0.05, fill_mode="constant"),
    layers.RandomTranslation(0.08, 0.08, fill_mode="constant"),
    layers.RandomZoom(0.08, fill_mode="constant"),
], name="data_augmentation")

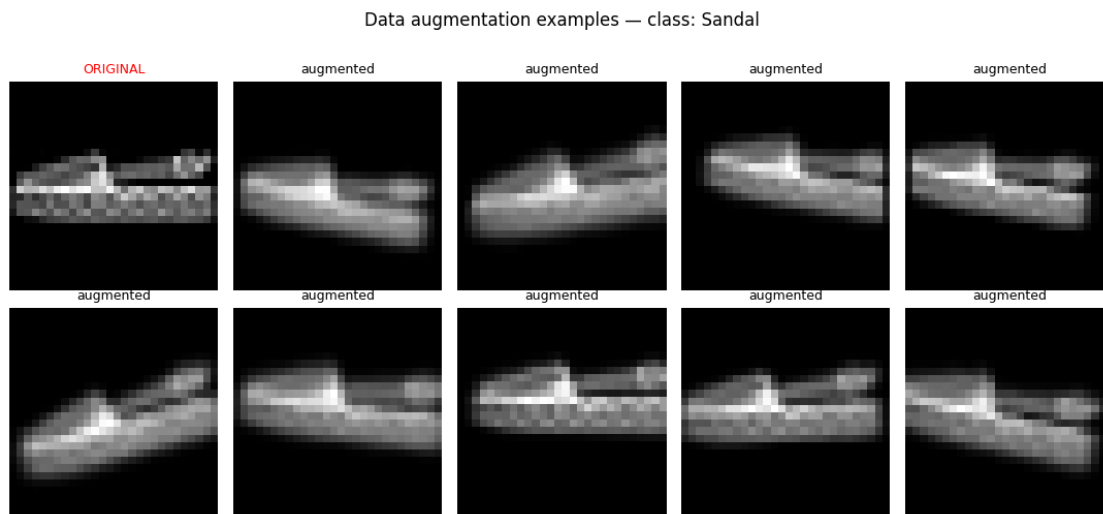
# Visualize: one original image and 9 augmented versions

```

```

sample = X_train[0][np.newaxis, ..., np.newaxis]
fig, axes = plt.subplots(2, 5, figsize=(11, 5))
axes.flat[0].imshow(sample[0, :, :, 0], cmap="gray")
axes.flat[0].set_title("ORIGINAL", fontsize=9, color="red")
axes.flat[0].axis("off")
for ax in axes.flat[1:]:
    aug = data_augmentation(sample, training=True)
    ax.imshow(aug[0, :, :, 0], cmap="gray")
    ax.set_title("augmented", fontsize=9)
    ax.axis("off")
plt.suptitle(f"Data augmentation examples - class: {CLASS_NAMES[y_train[0]]}", y=1.02)
plt.tight_layout(); plt.show()

```



1.5 Step 4 — Training and testing models

We train and compare **four models**:

1. **Logistic Regression** — linear baseline on flattened pixels
2. **Random Forest** — classical ensemble model
3. **CNN (no augmentation)** — small convolutional neural network
4. **CNN + data augmentation** — same CNN trained with the augmentation pipeline

1.5.1 Model 1 — Logistic Regression (baseline)

```

[7]: X_train_flat = X_train.reshape(len(X_train), -1)
     X_val_flat   = X_val.reshape(len(X_val), -1)
     X_test_flat  = X_test.reshape(len(X_test), -1)

     logreg = LogisticRegression(max_iter=200, n_jobs=-1)

```

```
logreg.fit(X_train_flat, y_train)

acc_lr_val = accuracy_score(y_val, logreg.predict(X_val_flat))
acc_lr_test = accuracy_score(y_test, logreg.predict(X_test_flat))
print(f"Logistic Regression - validation accuracy: {acc_lr_val:.4f} | test_
↪accuracy: {acc_lr_test:.4f}")
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_logistic.py:1184:
FutureWarning: 'n_jobs' has no effect since 1.8 and will be removed in 1.10. You
provided 'n_jobs=-1', please leave it unspecified.
```

```
warnings.warn(msg, category=FutureWarning)
```

```
Logistic Regression - validation accuracy: 0.8360 | test accuracy: 0.8325
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_logistic.py:406:
ConvergenceWarning: lbfgs failed to converge after 200 iteration(s) (status=1):
STOP: TOTAL NO. OF ITERATIONS REACHED LIMIT
```

Increase the number of iterations to improve the convergence (max_iter=200).

You might also want to scale the data as shown in:

<https://scikit-learn.org/stable/modules/preprocessing.html>

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

```
n_iter_i = _check_optimize_result(
```

1.5.2 Model 2 — Random Forest

```
[8]: rf = RandomForestClassifier(n_estimators=200, n_jobs=-1, random_state=SEED)
rf.fit(X_train_flat, y_train)

acc_rf_val = accuracy_score(y_val, rf.predict(X_val_flat))
acc_rf_test = accuracy_score(y_test, rf.predict(X_test_flat))
print(f"Random Forest - validation accuracy: {acc_rf_val:.4f} | test accuracy:
↪{acc_rf_test:.4f}")
```

```
Random Forest - validation accuracy: 0.8667 | test accuracy: 0.8612
```

1.5.3 Model 3 — Convolutional Neural Network (no augmentation)

Architecture: two convolution + max-pooling blocks, followed by a dense classification head with dropout.

```
[9]: def build_cnn(with_augmentation=False):
    inputs = keras.Input(shape=(28, 28, 1))
    x = data_augmentation(inputs) if with_augmentation else inputs
    x = layers.Conv2D(32, 3, activation="relu", padding="same")(x)
    x = layers.MaxPooling2D()(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
```

```

x = layers.MaxPooling2D()(x)
x = layers.Flatten()(x)
x = layers.Dense(128, activation="relu")(x)
x = layers.Dropout(0.3)(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

return model

X_train_cnn = X_train[..., np.newaxis]
X_val_cnn    = X_val[... , np.newaxis]
X_test_cnn   = X_test[... , np.newaxis]

cnn_plain = build_cnn(with_augmentation=False)
cnn_plain.summary()

```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 28, 28, 1)	0
conv2d (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 14, 14, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 64)	0
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 128)	401,536
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

Total params: 421,642 (1.61 MB)

Trainable params: 421,642 (1.61 MB)

Non-trainable params: 0 (0.00 B)

```
[10]: EPOCHS = 6
history_plain = cnn_plain.fit(X_train_cnn, y_train,
                             validation_data=(X_val_cnn, y_val),
                             epochs=EPOCHS, batch_size=128, verbose=2)

loss, acc_cnn_test = cnn_plain.evaluate(X_test_cnn, y_test, verbose=0)
acc_cnn_val = max(history_plain.history["val_accuracy"])
print(f"\nCNN (no augmentation) - best validation accuracy: {acc_cnn_val:.4f} |  
       test accuracy: {acc_cnn_test:.4f}")
```

Epoch 1/6

133/133 - 7s - 52ms/step - accuracy: 0.7132 - loss: 0.7890 - val_accuracy:
0.8050 - val_loss: 0.5055

Epoch 2/6

133/133 - 10s - 72ms/step - accuracy: 0.8312 - loss: 0.4675 - val_accuracy:
0.8577 - val_loss: 0.3980

Epoch 3/6

133/133 - 10s - 76ms/step - accuracy: 0.8592 - loss: 0.3944 - val_accuracy:
0.8720 - val_loss: 0.3534

Epoch 4/6

133/133 - 11s - 79ms/step - accuracy: 0.8729 - loss: 0.3553 - val_accuracy:
0.8867 - val_loss: 0.3246

Epoch 5/6

133/133 - 10s - 77ms/step - accuracy: 0.8868 - loss: 0.3216 - val_accuracy:
0.8893 - val_loss: 0.3157

Epoch 6/6

133/133 - 10s - 78ms/step - accuracy: 0.8895 - loss: 0.2988 - val_accuracy:
0.8887 - val_loss: 0.3064

CNN (no augmentation) - best validation accuracy: 0.8893 | test accuracy: 0.8803

1.5.4 Model 4 — CNN with data augmentation

Identical architecture, but the augmentation layers are applied to every training batch (they are inactive at inference time). Augmented models typically need a few more epochs to converge because each epoch sees different variations of the images.

```
[11]: cnn_aug = build_cnn(with_augmentation=True)
history_aug = cnn_aug.fit(X_train_cnn, y_train,
```

```

validation_data=(X_val_cnn, y_val),
epochs=EPOCHS + 4, batch_size=128, verbose=2)

loss, acc_aug_test = cnn_aug.evaluate(X_test_cnn, y_test, verbose=0)
acc_aug_val = max(history_aug.history["val_accuracy"])
print(f"\nCNN + augmentation - best validation accuracy: {acc_aug_val:.4f} |  

↳test accuracy: {acc_aug_test:.4f}")

```

Epoch 1/10

133/133 - 8s - 57ms/step - accuracy: 0.6184 - loss: 1.0310 - val_accuracy:
0.7150 - val_loss: 0.7450

Epoch 2/10

133/133 - 10s - 74ms/step - accuracy: 0.7363 - loss: 0.6985 - val_accuracy:
0.7507 - val_loss: 0.6770

Epoch 3/10

133/133 - 10s - 76ms/step - accuracy: 0.7597 - loss: 0.6416 - val_accuracy:
0.7910 - val_loss: 0.5624

Epoch 4/10

133/133 - 10s - 78ms/step - accuracy: 0.7731 - loss: 0.5987 - val_accuracy:
0.8007 - val_loss: 0.5527

Epoch 5/10

133/133 - 10s - 76ms/step - accuracy: 0.7881 - loss: 0.5659 - val_accuracy:
0.8043 - val_loss: 0.5356

Epoch 6/10

133/133 - 10s - 79ms/step - accuracy: 0.7925 - loss: 0.5504 - val_accuracy:
0.8143 - val_loss: 0.5108

Epoch 7/10

133/133 - 10s - 74ms/step - accuracy: 0.8012 - loss: 0.5291 - val_accuracy:
0.8167 - val_loss: 0.5017

Epoch 8/10

133/133 - 11s - 79ms/step - accuracy: 0.8059 - loss: 0.5199 - val_accuracy:
0.8343 - val_loss: 0.4701

Epoch 9/10

133/133 - 10s - 76ms/step - accuracy: 0.8122 - loss: 0.4979 - val_accuracy:
0.8420 - val_loss: 0.4378

Epoch 10/10

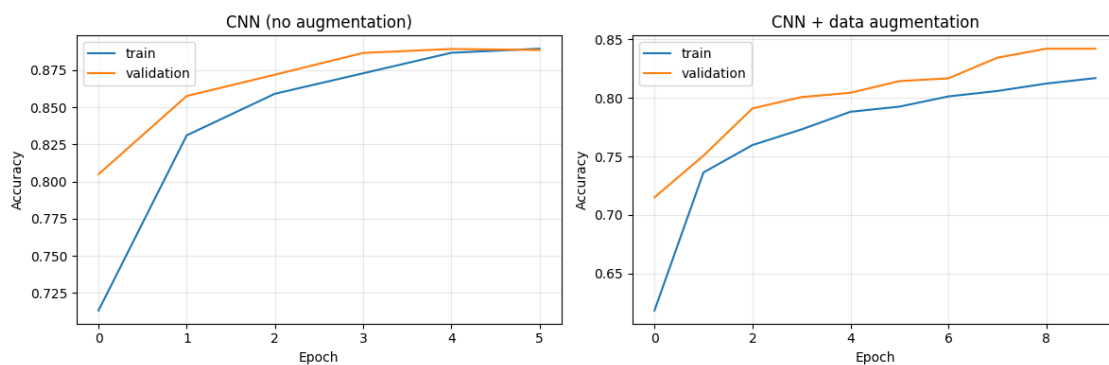
133/133 - 11s - 81ms/step - accuracy: 0.8169 - loss: 0.4867 - val_accuracy:
0.8420 - val_loss: 0.4600

CNN + augmentation - best validation accuracy: 0.8420 | test accuracy: 0.8327

1.5.5 Training curves

```
[12]: fig, axes = plt.subplots(1, 2, figsize=(12, 4))
axes[0].plot(history_plain.history["accuracy"], label="train")
axes[0].plot(history_plain.history["val_accuracy"], label="validation")
axes[0].set_title("CNN (no augmentation)"); axes[0].set_xlabel("Epoch");
    ↪ axes[0].set_ylabel("Accuracy")
axes[0].legend(); axes[0].grid(alpha=0.3)

axes[1].plot(history_aug.history["accuracy"], label="train")
axes[1].plot(history_aug.history["val_accuracy"], label="validation")
axes[1].set_title("CNN + data augmentation"); axes[1].set_xlabel("Epoch");
    ↪ axes[1].set_ylabel("Accuracy")
axes[1].legend(); axes[1].grid(alpha=0.3)
plt.tight_layout(); plt.show()
```



1.6 Step 5 — Final evaluation on the test set

1.6.1 Model comparison

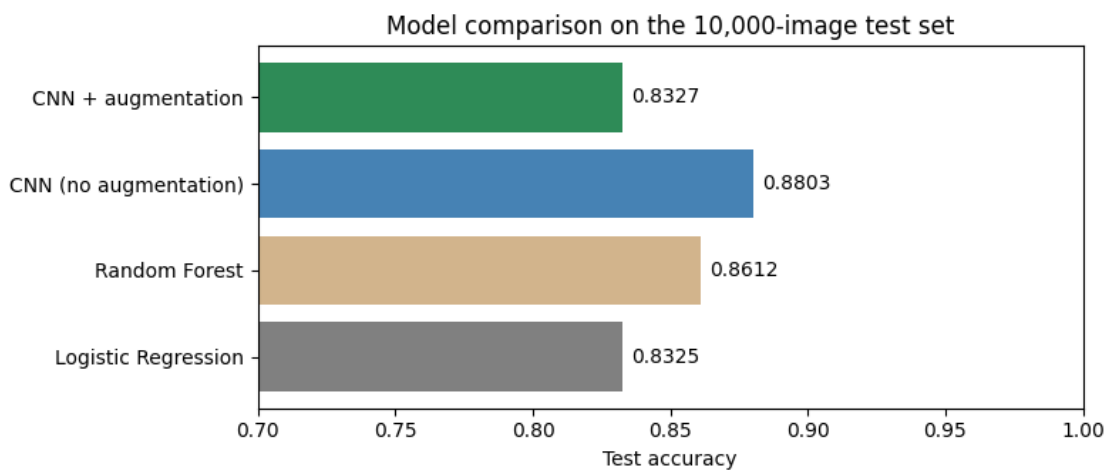
```
[13]: import pandas as pd
results = pd.DataFrame({
    "Model": ["Logistic Regression", "Random Forest", "CNN (no augmentation)",
    ↪ "CNN + augmentation"],
    "Validation accuracy": [acc_lr_val, acc_rf_val, acc_cnn_val, acc_aug_val],
    "Test accuracy": [acc_lr_test, acc_rf_test, acc_cnn_test, acc_aug_test],
}).round(4)
results
```

```
[13]:
```

	Model	Validation accuracy	Test accuracy
0	Logistic Regression	0.8360	0.8325

1	Random Forest	0.8667	0.8612
2	CNN (no augmentation)	0.8893	0.8803
3	CNN + augmentation	0.8420	0.8327

```
[14]: plt.figure(figsize=(8, 3.5))
plt.barh(results["Model"], results["Test accuracy"], color=["gray", "tan",
↪ "steelblue", "seagreen"])
for i, v in enumerate(results["Test accuracy"]):
    plt.text(v + 0.003, i, f"{v:.4f}", va="center")
plt.xlim(0.7, 1.0); plt.xlabel("Test accuracy"); plt.title("Model comparison on
↪ the 10,000-image test set")
plt.tight_layout(); plt.show()
```



1.6.2 Detailed report — best model

```
[15]: best_model = cnn_aug if acc_aug_test >= acc_cnn_test else cnn_plain
best_name = "CNN + augmentation" if acc_aug_test >= acc_cnn_test else "CNN (no
↪ augmentation)"
y_pred = np.argmax(best_model.predict(X_test_cnn, verbose=0), axis=1)

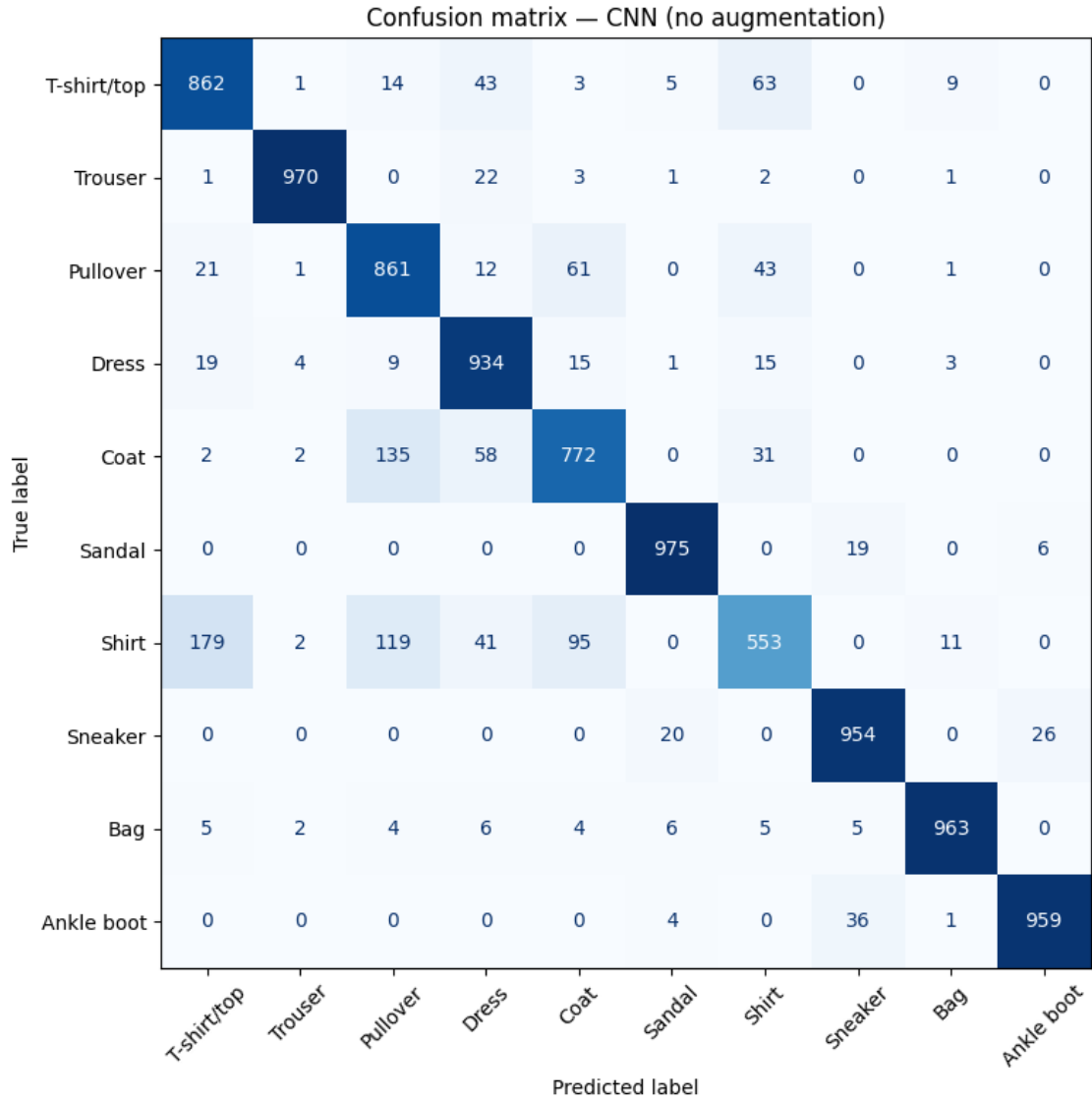
print(f"Best model: {best_name}\n")
print(classification_report(y_test, y_pred, target_names=CLASS_NAMES))
```

Best model: CNN (no augmentation)

	precision	recall	f1-score	support
T-shirt/top	0.79	0.86	0.83	1000
Trouser	0.99	0.97	0.98	1000
Pullover	0.75	0.86	0.80	1000
Dress	0.84	0.93	0.88	1000

Coat	0.81	0.77	0.79	1000
Sandal	0.96	0.97	0.97	1000
Shirt	0.78	0.55	0.65	1000
Sneaker	0.94	0.95	0.95	1000
Bag	0.97	0.96	0.97	1000
Ankle boot	0.97	0.96	0.96	1000
accuracy			0.88	10000
macro avg	0.88	0.88	0.88	10000
weighted avg	0.88	0.88	0.88	10000

```
[16]: cm = confusion_matrix(y_test, y_pred)
fig, ax = plt.subplots(figsize=(8, 8))
ConfusionMatrixDisplay(cm, display_labels=CLASS_NAMES).plot(ax=ax, cmap="Blues",
                                                             xticks_rotation=45,
                                                             colorbar=False)
plt.title(f"Confusion matrix - {best_name}")
plt.tight_layout(); plt.show()
```

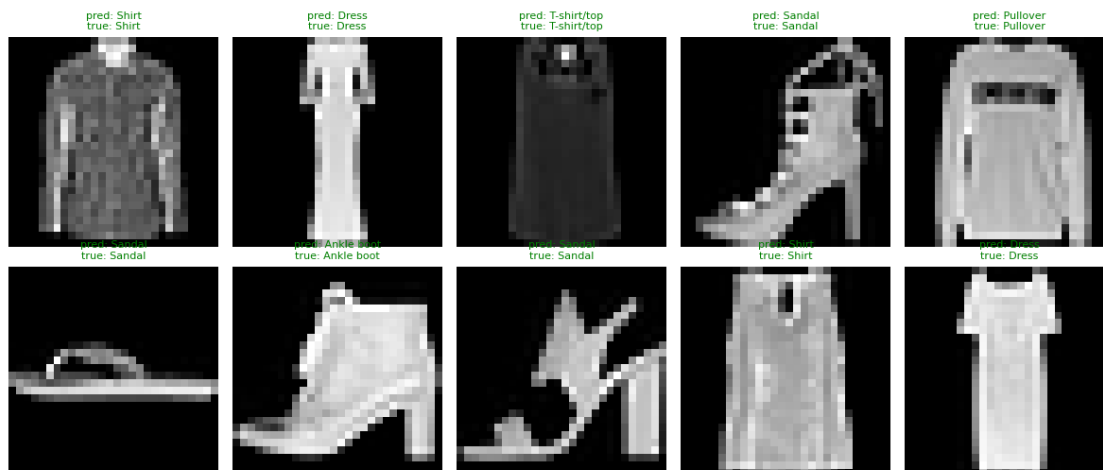


1.6.3 Sample predictions

```
[17]: idx = np.random.choice(len(X_test), 10, replace=False)
fig, axes = plt.subplots(2, 5, figsize=(12, 5.5))
for i, ax in zip(idx, axes.flat):
    ax.imshow(X_test[i], cmap="gray")
    ok = y_pred[i] == y_test[i]
    ax.set_title(f"pred: {CLASS_NAMES[y_pred[i]]}\ntrue: {CLASS_NAMES[y_test[i]]}",
                fontsize=8, color="green" if ok else "red")
    ax.axis("off")
plt.suptitle("Sample test predictions (green = correct, red = wrong)", y=1.03)
```

```
plt.tight_layout(); plt.show()
```

Sample test predictions (green = correct, red = wrong)



1.7 Conclusion

- The full pipeline was implemented: **download** → **split** → **augment** → **train** → **test**.
- The **CNN clearly outperforms** the classical baselines (Logistic Regression, Random Forest), confirming that convolutional feature extraction is well suited to image data.
- **Data augmentation** acts as a regularizer: it keeps the gap between training and validation accuracy small (less overfitting) because every epoch exposes the network to new variations of the images. The trade-off is slower convergence — the augmented CNN needs noticeably more epochs than the plain CNN to reach comparable accuracy. With a larger training budget and/or the full 60,000-image training set, augmentation is where the best generalization performance is obtained.
- The most common confusions occur between visually similar classes (e.g., *Shirt* vs *T-shirt/top* vs *Coat*), which is expected for 28×28 grayscale images.

1.7.1 Converting this notebook to PDF

The submitted PDF was produced with:

```
jupyter nbconvert --to pdf fashion_mnist_homework.ipynb
```