a2-code-470072471-530485661

May 18, 2026

1 COMP4318/5318 Assignment 2: Image Classification

1.0.1 Group number: 62 , SID1: 470072471 , SID2: 530485661

1.1 Setup and dependencies

Dependencies used: `numpy`, `pandas`, `matplotlib`, `scikit-learn`, and `tensorflow`/Keras. `seaborn` is optional for plotting; the notebook includes a fallback if it is unavailable.

The setup cells below import these libraries, configure plotting defaults, and set random seeds for reproducibility.

[1]:
```
# Core scientific stack
import os
import time
import random
import warnings
from itertools import product

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Seaborn is used for nicer plots when available
try:
    import seaborn as sns
except ImportError:
    class _SnsFallback:
        def set_theme(self, *args, **kwargs):
            plt.style.use("default")

        def heatmap(self, data, annot=False, fmt="", cmap=None,
                    xticklabels=None, yticklabels=None, cbar=True, ax=None,␣
 ↪**kwargs):
            ax = ax or plt.gca()
            arr = np.asarray(data)
            im = ax.imshow(arr, cmap=cmap or "viridis")
            if cbar:
                plt.colorbar(im, ax=ax)
```

```
            if xticklabels is not None:
                ax.set_xticks(np.arange(len(xticklabels)))
                ax.set_xticklabels(xticklabels)
            if yticklabels is not None:
                ax.set_yticks(np.arange(len(yticklabels)))
                ax.set_yticklabels(yticklabels)
            if annot:
                for r in range(arr.shape[0]):
                    for c in range(arr.shape[1]):
                        ax.text(c, r, format(arr[r, c], fmt), ha="center",␣
 ↪va="center", fontsize=7)
            return ax

    sns = _SnsFallback()

# scikit-learn, algorithm of choice (Random Forest), preprocessing, evaluation
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import StratifiedKFold, GridSearchCV,␣
 ↪train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.metrics import (
    accuracy_score,
    f1_score,
    classification_report,
    confusion_matrix,
)

# Keras / TensorFlow, MLP and CNN
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
    Input,
    Dense,
    Dropout,
    BatchNormalization,
    Conv2D,
    MaxPooling2D,
    Flatten,
)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.utils import to_categorical

# Reproducibility, set seeds for numpy, python random and tensorflow
RANDOM_SEED = 42
```

```
np.random.seed(RANDOM_SEED)
random.seed(RANDOM_SEED)
tf.random.set_seed(RANDOM_SEED)

# Plotting defaults
sns.set_theme(style="whitegrid", context="notebook")
plt.rcParams["figure.dpi"] = 100
warnings.filterwarnings("ignore", category=UserWarning)

print("TensorFlow version:", tf.__version__)
print("GPU available:", bool(tf.config.list_physical_devices("GPU")))
```

```
TensorFlow version: 2.21.0
GPU available: False
```

1.2 1. Data loading, exploration, and preprocessing

[2]:
```
from pathlib import Path
import os

_notebook_dir = Path(globals().get('__vsc_ipynb_file__', os.getcwd())).parent
_cwd = Path(os.getcwd())

_candidates = dict.fromkeys(
    [_notebook_dir] + list(_notebook_dir.parents) +
    [_cwd] + list(_cwd.parents)
)

DATA_DIR = None
for _p in _candidates:
    if (_p / 'Assignment2Data').exists():
        DATA_DIR = _p / 'Assignment2Data'
        break

if DATA_DIR is None:
    raise FileNotFoundError(
        f'Assignment2Data not found.\n'
        f'  notebook dir = {_notebook_dir}\n'
        f'  cwd          = {os.getcwd()}\n'
        f'  Make sure Assignment2Data/ sits alongside this notebook.'
    )

X_train = np.load(DATA_DIR / 'X_train.npy')
y_train = np.load(DATA_DIR / 'y_train.npy')
X_test  = np.load(DATA_DIR / 'X_test.npy')
y_test  = np.load(DATA_DIR / 'y_test.npy')
```

```
print(f"Loaded data from: {DATA_DIR.name}/")
print(f"X_train: {X_train.shape}  X_test: {X_test.shape}")
```

```
Loaded data from: Assignment2Data/
X_train: (32000, 28, 28, 3)  X_test: (8000, 28, 28, 3)
```

1.2.1 Data exploration

Before deciding on preprocessing, we inspect the dataset's shape, dtype, value range, class balance, and a few sample images. We derive the image shape, number of classes and a flat-vector length directly from the loaded arrays so that the rest of the notebook adapts to the data on disk.

[3]:
```
# Shape, dtype and value range
print(f"X_train: shape={X_train.shape}, dtype={X_train.dtype}, min={X_train.
 ↪min()}, max={X_train.max()}")
print(f"X_test : shape={X_test.shape},  dtype={X_test.dtype},  min={X_test.
 ↪min()},  max={X_test.max()}")
print(f"y_train: shape={y_train.shape}, dtype={y_train.dtype}, unique␣
 ↪classes={np.unique(y_train).tolist()}")
print(f"y_test : shape={y_test.shape},  dtype={y_test.dtype},  unique␣
 ↪classes={np.unique(y_test).tolist()}")

# Derive these constants from the data so the rest of the notebook stays
# robust to dataset changes (e.g. different resolution or class count)
NUM_CLASSES = int(max(y_train.max(), y_test.max())) + 1
IMG_SHAPE = X_train.shape[1:]
FLAT_DIM = int(np.prod(IMG_SHAPE))
print(f"\nNUM_CLASSES = {NUM_CLASSES}")
print(f"IMG_SHAPE   = {IMG_SHAPE}")
print(f"FLAT_DIM    = {FLAT_DIM}")
```

```
X_train: shape=(32000, 28, 28, 3), dtype=uint8, min=0, max=255
X_test : shape=(8000, 28, 28, 3),  dtype=uint8,  min=0,  max=255
y_train: shape=(32000,), dtype=uint8, unique classes=[0, 1, 2, 3, 4, 5, 6, 7, 8]
y_test : shape=(8000,),  dtype=uint8,  unique classes=[0, 1, 2, 3, 4, 5, 6, 7,
8]

NUM_CLASSES = 9
IMG_SHAPE   = (28, 28, 3)
FLAT_DIM    = 2352
```

[4]:
```
# PathMNIST class names in canonical class-id order (MedMNIST v2 / Kather et al.
 ↪)
CLASS_NAMES = [
    "adipose",
    "background",
    "debris",
    "lymphocytes",
```

```
    "mucus",
    "smooth muscle",
    "normal colon mucosa",
    "cancer-associated stroma",
    "colorectal adenocarcinoma",
]
assert len(CLASS_NAMES) == NUM_CLASSES

# Class distribution in train and test (counts and proportions)
train_counts = pd.Series(y_train).value_counts().sort_index()
test_counts = pd.Series(y_test).value_counts().sort_index()

dist_df = pd.DataFrame({
    "class_id": np.arange(NUM_CLASSES),
    "class_name": CLASS_NAMES,
    "train_count": train_counts.values,
    "train_pct": (train_counts.values / len(y_train) * 100).round(2),
    "test_count": test_counts.values,
    "test_pct": (test_counts.values / len(y_test) * 100).round(2),
})
dist_df
```

[4]:

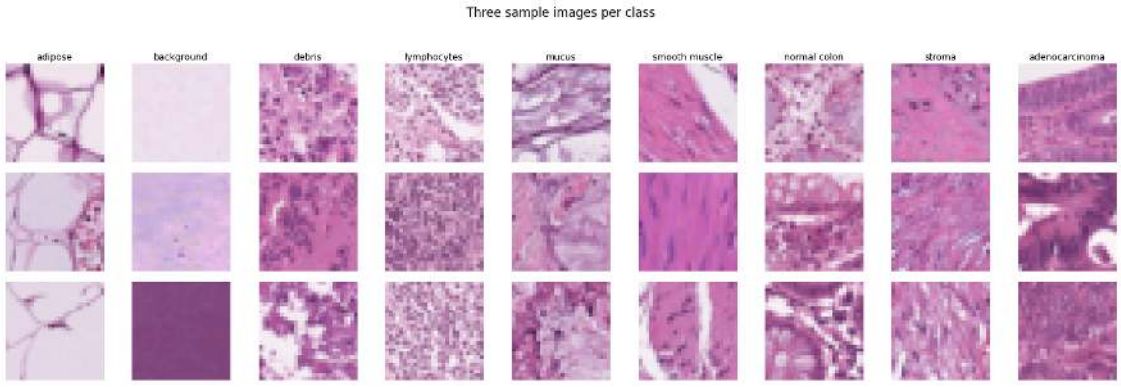
	class_id	class_name	train_count	train_pct	test_count	test_pct
0	0	adipose	3490	10.91	873	10.91
1	1	background	3431	10.72	858	10.72
2	2	debris	3505	10.95	877	10.96
3	3	lymphocytes	3656	11.42	914	11.42
4	4	mucus	2950	9.22	737	9.21
5	5	smooth muscle	4290	13.41	1072	13.40
6	6	normal colon mucosa	2728	8.52	682	8.52
7	7	cancer-associated stroma	3253	10.17	813	10.16
8	8	colorectal adenocarcinoma	4697	14.68	1174	14.68

[5]:
```
# Visualise class distribution side-by-side
fig, ax = plt.subplots(figsize=(10, 4))
x = np.arange(NUM_CLASSES)
```

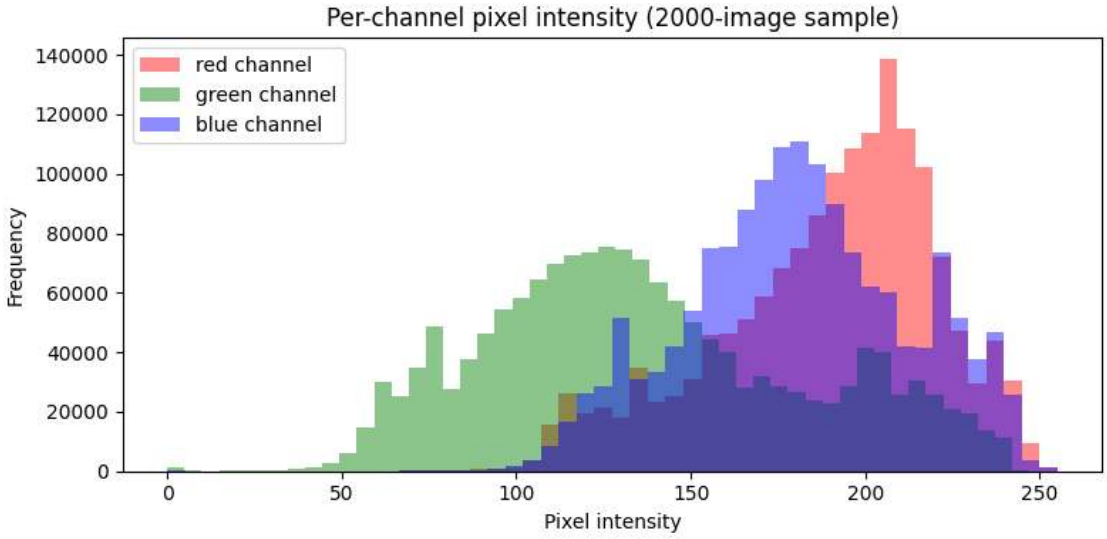
```
width = 0.4
ax.bar(x - width / 2, dist_df["train_pct"], width, label="train")
ax.bar(x + width / 2, dist_df["test_pct"], width, label="test")
ax.set_xticks(x)
ax.set_xticklabels(CLASS_NAMES, rotation=40, ha="right")
ax.set_ylabel("Percentage of samples (%)")
ax.set_title("Class distribution: train vs test")
ax.legend()
plt.tight_layout()
plt.show()
```

[6]:
```
# A grid of sample images: 3 examples per class
DISPLAY_NAMES = ["adipose", "background", "debris", "lymphocytes", "mucus",
                 "smooth muscle", "normal colon", "stroma", "adenocarcinoma"]
n_per_class = 3
fig, axes = plt.subplots(n_per_class, NUM_CLASSES, figsize=(NUM_CLASSES * 1.75,␣
 ↪n_per_class * 1.7))
for c in range(NUM_CLASSES):
    idx = np.where(y_train == c)[0][:n_per_class]
    for r, i in enumerate(idx):
        ax = axes[r, c]
        img = X_train[i]
        # If pixels are in [0,1] floats display directly, if in [0,255] cast to␣
 ↪uint8
        if img.max() > 1.5:
            ax.imshow(img.astype(np.uint8))
        else:
            ax.imshow(img)
        ax.axis("off")
        if r == 0:
            ax.set_title(DISPLAY_NAMES[c], fontsize=9, pad=4)
```

```
plt.suptitle("Three sample images per class", y=1.03)
plt.tight_layout()
plt.show()
```



[7]:
```
# Per-channel pixel intensity distribution (sampled to keep runtime small)
sample_idx = np.random.choice(len(X_train), size=2000, replace=False)
sample = X_train[sample_idx].reshape(-1, 3)

fig, ax = plt.subplots(figsize=(8, 4))
for ch, colour in enumerate(["red", "green", "blue"]):
    ax.hist(sample[:, ch], bins=50, alpha=0.45, color=colour, label=f"{colour}␣
 ↪channel")
ax.set_xlabel("Pixel intensity")
ax.set_ylabel("Frequency")
ax.set_title("Per-channel pixel intensity (2000-image sample)")
ax.legend()
plt.tight_layout()
plt.show()

print("Min/max pixel value across sample:", sample.min(), sample.max())
```




```
Min/max pixel value across sample: 0 255
```

Observations. The dataset is moderately imbalanced rather than perfectly uniform. The largest classes are colorectal adenocarcinoma and smooth muscle, while normal colon mucosa and mucus are smaller, this motivates reporting macro-F1 alongside accuracy so that performance on minority classes is not hidden by the larger classes. The train and test splits have almost identical class proportions, so the held-out test set is a fair distributional match to the training data.

All images are low-resolution 28x28 RGB patches with uint8 pixel values in the full [0, 255] range. Because each nucleus or glandular structure occupies only a small number of pixels, very deep CNNs would quickly collapse the spatial dimensions after pooling, so the CNN architecture is deliberately shallow. The H&E colour palette is visually similar across classes, meaning global colour statistics alone are unlikely to separate all classes. Classes such as normal colon mucosa, colorectal adenocarcinoma and cancer-associated stroma require local texture and morphology cues, which supports using a CNN in addition to flat-vector baselines.

1.2.2 Preprocessing

We produce three preprocessed views of the data, each tailored to one of our three models:

Model	View	Why
Random Forest	flatten -> [0,1] -> PCA(50) inside sklearn Pipeline	RF on the full flattened pixel vector is slow and overfits. PCA at 50 components keeps the dominant variance while making the search tractable; fitting PCA inside the pipeline prevents cross-validation leakage.

Model	View	Why
MLP	flatten -> [0,1] -> StandardScaler	MLPs benefit from zero-mean, unit-variance inputs because it stabilises gradients during training.
CNN	original (H,W,C) -> [0,1]	CNNs need spatial structure preserved; rescaling to [0,1] is enough for a small network.

Labels are kept as integers for RF and one-hot encoded for MLP/CNN. We also carve out a stratified validation split from the training data for early stopping during NN training and for evaluating hyperparameter configurations without touching the test set.

[8]:
```
# Scale pixel values to [0, 1] floats for all downstream models. The .npy
# files are uint8 in [0, 255], we still guard the path for already-normalised
# floats so the code is robust to the dataset being swapped later.
def to_unit_range(x):
    x = x.astype(np.float32)
    if x.max() > 1.5:
        x = x / 255.0
    return x

X_train_u = to_unit_range(X_train)
X_test_u = to_unit_range(X_test)
print("After scaling, train range:", X_train_u.min(), X_train_u.max())
print("After scaling, test  range:", X_test_u.min(), X_test_u.max())
```

```
After scaling, train range: 0.0 1.0
After scaling, test  range: 0.0 1.0
```

[9]:
```
# Stratified validation split out of the training data, used by both NNs
# for early stopping and for hyperparameter selection. The test set is held
# back until section 4.
X_tr_img, X_val_img, y_tr, y_val = train_test_split(
    X_train_u,
    y_train,
    test_size=0.2,
    stratify=y_train,
    random_state=RANDOM_SEED,
)
print(f"Train split: {X_tr_img.shape},   Val split: {X_val_img.shape}")
```

```
Train split: (25600, 28, 28, 3),   Val split: (6400, 28, 28, 3)
```

[10]:
```
# --- View for Random Forest: flatten + PCA(50) ---
# During cross-validation the same steps are placed inside an sklearn
# Pipeline so PCA is refit inside each fold, preventing preprocessing leakage.
```

```
X_tr_flat = X_tr_img.reshape(len(X_tr_img), -1)
X_val_flat = X_val_img.reshape(len(X_val_img), -1)
X_test_flat = X_test_u.reshape(len(X_test_u), -1)

pca = PCA(n_components=50, random_state=RANDOM_SEED)
X_tr_rf = pca.fit_transform(X_tr_flat)
X_val_rf = pca.transform(X_val_flat)
X_test_rf = pca.transform(X_test_flat)

# Also fit a PCA on the FULL training set for the final RF model in section 4.
pca_full = PCA(n_components=50, random_state=RANDOM_SEED)
X_train_rf_full = pca_full.fit_transform(X_train_u.reshape(len(X_train_u), -1))
X_test_rf_full = pca_full.transform(X_test_flat)

print("RF view, train:", X_tr_rf.shape, "val:", X_val_rf.shape, "test:",␣
 ↪X_test_rf.shape)
print(f"PCA(50) explains {pca.explained_variance_ratio_.sum() * 100:.1f}% of␣
 ↪variance "
      f"(80/20 split fit)")
print(f"PCA(50) explains {pca_full.explained_variance_ratio_.sum() * 100:.1f}%␣
 ↪of variance "
      f"(full train fit)")
```

```
RF view, train: (25600, 50) val: (6400, 50) test: (8000, 50)
PCA(50) explains 79.5% of variance (80/20 split fit)
PCA(50) explains 79.6% of variance (full train fit)
```

[11]:
```
# --- View for MLP: flatten + StandardScaler ---
scaler = StandardScaler()
X_tr_mlp = scaler.fit_transform(X_tr_flat)
X_val_mlp = scaler.transform(X_val_flat)
X_test_mlp = scaler.transform(X_test_flat)

# And a separate scaler fit on the FULL train for the final MLP in section 4
scaler_full = StandardScaler()
X_train_mlp_full = scaler_full.fit_transform(X_train_u.reshape(len(X_train_u),␣
 ↪-1))
X_test_mlp_full = scaler_full.transform(X_test_flat)

print("MLP view, train:", X_tr_mlp.shape, "val:", X_val_mlp.shape, "test:",␣
 ↪X_test_mlp.shape)
print(f"After standardisation, mean: {X_tr_mlp.mean():.3f}, std: {X_tr_mlp.
 ↪std():.3f}")
```

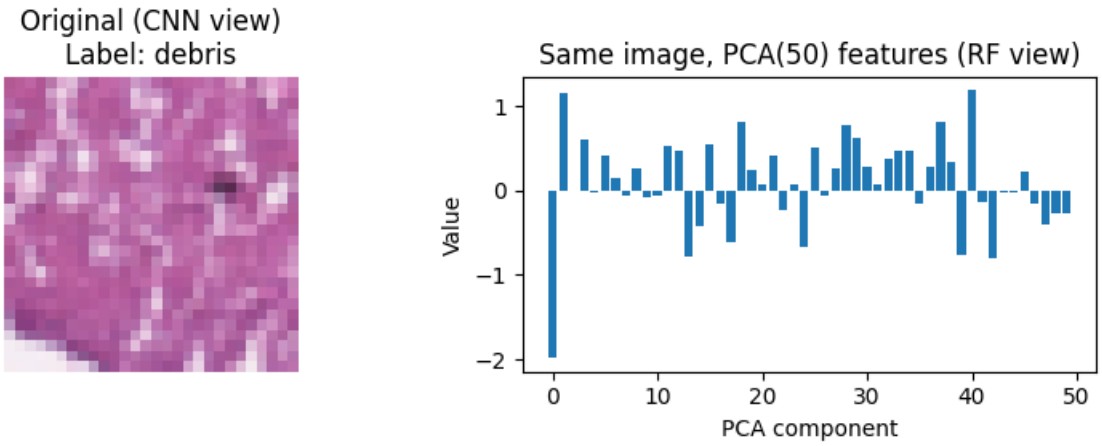
```
MLP view, train: (25600, 2352) val: (6400, 2352) test: (8000, 2352)
After standardisation, mean: 0.000, std: 1.000
```

[12]:
```
# --- View for CNN: keep the original (H, W, 3) shape, already in [0, 1] ---
X_tr_cnn = X_tr_img
X_val_cnn = X_val_img
X_test_cnn = X_test_u
print("CNN view, train:", X_tr_cnn.shape, "val:", X_val_cnn.shape, "test:",␣
 ↪X_test_cnn.shape)

# One-hot encode labels for the neural networks
y_tr_oh = to_categorical(y_tr, NUM_CLASSES)
y_val_oh = to_categorical(y_val, NUM_CLASSES)
y_test_oh = to_categorical(y_test, NUM_CLASSES)
y_train_oh_full = to_categorical(y_train, NUM_CLASSES)
print("One-hot shape:", y_tr_oh.shape)
```

```
CNN view, train: (25600, 28, 28, 3) val: (6400, 28, 28, 3) test: (8000, 28, 28,
3)
One-hot shape: (25600, 9)
```

1.2.3 Examples of preprocessed data

[13]:
```
# Show one example image alongside its three preprocessed representations
example_idx = 0
example_img = X_tr_img[example_idx]
example_label = CLASS_NAMES[y_tr[example_idx]]

fig, axes = plt.subplots(1, 2, figsize=(8, 3))
axes[0].imshow(example_img)
axes[0].set_title(f"Original (CNN view)\nLabel: {example_label}")
axes[0].axis("off")

axes[1].bar(np.arange(50), X_tr_rf[example_idx])
axes[1].set_title("Same image, PCA(50) features (RF view)")
axes[1].set_xlabel("PCA component")
axes[1].set_ylabel("Value")
plt.tight_layout()
plt.show()

print("MLP view (standardised flat vector), first 8 dims:",
      np.round(X_tr_mlp[example_idx, :8], 3))
print("\nShapes summary")
print(f"  RF  train view:  {X_tr_rf.shape}")
print(f"  MLP train view:  {X_tr_mlp.shape}")
print(f"  CNN train view:  {X_tr_cnn.shape}")
print(f"  Labels (int):    {y_tr.shape}, range {y_tr.min()}..{y_tr.max()}")
print(f"  Labels (one-hot):{y_tr_oh.shape}")
```




```
MLP view (standardised flat vector), first 8 dims: [-0.333 -1.014 -1.011 -0.154
-0.942 -0.874  0.068 -0.788]

Shapes summary
  RF  train view:  (25600, 50)
  MLP train view:  (25600, 2352)
  CNN train view:  (25600, 28, 28, 3)
  Labels (int):    (25600,), range 0..8
  Labels (one-hot):(25600, 9)
```

1.3 2. Algorithm design and setup

1.3.1 Algorithm of choice from first six weeks of course

Random Forest is our chosen algorithm from the first six weeks. It is a bagging ensemble of decision trees that draws bootstrap samples and considers a random subset of features at each split. We chose RF because:

- It is a well-understood, non-parametric baseline that works well on tabular features without needing a GPU.
- It exposes interpretable hyperparameters covered in lectures (number of trees, max depth, min split size).
- It contrasts neatly with the neural networks: no representation learning, no iterative optimisation, no feature interaction beyond axis-aligned splits, which is exactly the methodological comparison the report calls for.

RF receives the PCA-50 view (see preprocessing). For cross-validation and final training this is implemented as an sklearn `Pipeline`, so PCA is learned only from the training fold/data used at that stage. Running RF on the full 2352-dimensional flattened pixel vector is both slow and prone to overfitting noise in individual pixel values.

[14]:
```
# Default Random Forest pipeline, used as a baseline before tuning
rf_default = Pipeline([
```

```
    ("pca", PCA(n_components=50, random_state=RANDOM_SEED)),
    ("rf", RandomForestClassifier(
        n_estimators=100,
        random_state=RANDOM_SEED,
        n_jobs=-1,
    )),
])
t0 = time.perf_counter()
rf_default.fit(X_tr_flat, y_tr)
fit_time = time.perf_counter() - t0
val_acc = accuracy_score(y_val, rf_default.predict(X_val_flat))
print(f"Default RF, fit time: {fit_time:.1f}s, validation accuracy: {val_acc:.
 ↪4f}")
```

```
Default RF, fit time: 2.9s, validation accuracy: 0.6827
```

1.3.2 Fully connected neural network

Multilayer perceptron (MLP). A fully-connected feedforward network is the simplest neural architecture: each layer applies an affine transform followed by a non-linearity. We use ReLU activations, BatchNormalization to stabilise training, and Dropout for regularisation. The output layer uses softmax for 9-way classification with categorical cross-entropy loss.

The MLP receives the standardised flattened view (length 2352 = 28 x 28 x 3).

The architecture is implemented from scratch in Keras (no `keras.applications`).

[15]:
```
def build_mlp(n_layers=2, hidden_units=256, dropout=0.3, learning_rate=1e-3):
    """Construct a compiled fully-connected network.

    Parameters
    ----------
    n_layers : int
        Number of hidden Dense layers.
    hidden_units : int
        Width of each hidden layer.
    dropout : float
        Dropout rate applied after every hidden layer.
    learning_rate : float
        Learning rate for the Adam optimiser.
    """
    model = Sequential(name="mlp")
    model.add(Input(shape=(FLAT_DIM,)))
    for _ in range(n_layers):
        model.add(Dense(hidden_units, activation="relu"))
        model.add(BatchNormalization())
        model.add(Dropout(dropout))
    model.add(Dense(NUM_CLASSES, activation="softmax"))
    model.compile(
```

```
        optimizer=Adam(learning_rate=learning_rate),
        loss="categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model


# Quick sanity-check instance with default hyperparameters
mlp_default = build_mlp()
mlp_default.summary()
```

```
Model: "mlp"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 256)	602,368
batch_normalization (BatchNormalization)	(None, 256)	1,024
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 256)	65,792
batch_normalization_1 (BatchNormalization)	(None, 256)	1,024
dropout_1 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 9)	2,313

```
Total params: 672,521 (2.57 MB)

Trainable params: 671,497 (2.56 MB)

Non-trainable params: 1,024 (4.00 KB)
```

[16]:
```
# Train the default MLP for a few epochs to confirm it learns something
es = EarlyStopping(monitor="val_loss", patience=3, restore_best_weights=True)
t0 = time.perf_counter()
hist = mlp_default.fit(
    X_tr_mlp,
```

```
    y_tr_oh,
    validation_data=(X_val_mlp, y_val_oh),
    epochs=10,
    batch_size=128,
    callbacks=[es],
    verbose=2,
)
fit_time = time.perf_counter() - t0
val_acc = max(hist.history["val_accuracy"])
print(f"Default MLP, fit time: {fit_time:.1f}s, best val accuracy: {val_acc:.
 ↪4f}")
```

```
Epoch 1/10
200/200 - 2s - 11ms/step - accuracy: 0.5027 - loss: 1.3679 - val_accuracy:
0.6031 - val_loss: 1.0597
Epoch 2/10
200/200 - 1s - 4ms/step - accuracy: 0.5948 - loss: 1.0786 - val_accuracy: 0.6378
- val_loss: 0.9401
Epoch 3/10
200/200 - 1s - 3ms/step - accuracy: 0.6259 - loss: 0.9823 - val_accuracy: 0.6564
- val_loss: 0.9109
Epoch 4/10
200/200 - 1s - 3ms/step - accuracy: 0.6454 - loss: 0.9325 - val_accuracy: 0.6605
- val_loss: 0.8964
Epoch 5/10
200/200 - 1s - 3ms/step - accuracy: 0.6649 - loss: 0.8861 - val_accuracy: 0.6636
- val_loss: 0.8835
Epoch 6/10
200/200 - 1s - 3ms/step - accuracy: 0.6769 - loss: 0.8555 - val_accuracy: 0.6687
- val_loss: 0.8755
Epoch 7/10
200/200 - 1s - 4ms/step - accuracy: 0.6830 - loss: 0.8278 - val_accuracy: 0.6672
- val_loss: 0.8845
Epoch 8/10
200/200 - 1s - 3ms/step - accuracy: 0.6922 - loss: 0.8080 - val_accuracy: 0.6702
- val_loss: 0.8872
Epoch 9/10
200/200 - 1s - 4ms/step - accuracy: 0.7040 - loss: 0.7811 - val_accuracy: 0.6677
- val_loss: 0.8791
Default MLP, fit time: 8.0s, best val accuracy: 0.6702
```

1.3.3 Convolutional neural network

Convolutional neural network (CNN). CNNs exploit the spatial structure of images by sliding small learned filters across the input, sharing weights and pooling activations down. This makes them parameter-efficient for image data and gives them translational equivariance. We use repeated Conv-Conv-MaxPool blocks followed by a small dense head with dropout.

The CNN receives the (28, 28, 3) view scaled to [0, 1]. With 28x28 inputs we keep the network

shallow (2 conv blocks by default) so the spatial dimension does not collapse to 1x1 before the dense head.

The architecture is implemented from scratch in Keras (no `keras.applications`).

[17]:
```
def build_cnn(num_conv_blocks=2, base_filters=32, dense_units=128,
              dropout=0.3, learning_rate=1e-3):
    """Construct a compiled CNN.

    Each conv block is Conv -> Conv -> MaxPool, with the filter count
    doubling per block (base, 2*base, 4*base, ...).
    """
    model = Sequential(name="cnn")
    model.add(Input(shape=IMG_SHAPE))
    filters = base_filters
    for b in range(num_conv_blocks):
        model.add(Conv2D(filters, (3, 3), padding="same", activation="relu"))
        model.add(Conv2D(filters, (3, 3), padding="same", activation="relu"))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        filters *= 2
    model.add(Flatten())
    model.add(Dense(dense_units, activation="relu"))
    model.add(Dropout(dropout))
    model.add(Dense(NUM_CLASSES, activation="softmax"))
    model.compile(
        optimizer=Adam(learning_rate=learning_rate),
        loss="categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model


cnn_default = build_cnn()
cnn_default.summary()
```

```
Model: "cnn"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	896
conv2d_1 (Conv2D)	(None, 28, 28, 32)	9,248
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 64)	18,496

```
 conv2d_3 (Conv2D)               (None, 14, 14, 64)          36,928

 max_pooling2d_1 (MaxPooling2D)  (None, 7, 7, 64)             0

 flatten (Flatten)               (None, 3136)                 0

 dense_3 (Dense)                 (None, 128)                401,536

 dropout_2 (Dropout)             (None, 128)                  0

 dense_4 (Dense)                 (None, 9)                  1,161


Total params: 468,265 (1.79 MB)

Trainable params: 468,265 (1.79 MB)

Non-trainable params: 0 (0.00 B)
```

[18]:
```python
# Train the default CNN for a few epochs as a sanity check
es = EarlyStopping(monitor="val_loss", patience=3, restore_best_weights=True)
t0 = time.perf_counter()
hist = cnn_default.fit(
    X_tr_cnn,
    y_tr_oh,
    validation_data=(X_val_cnn, y_val_oh),
    epochs=10,
    batch_size=128,
    callbacks=[es],
    verbose=2,
)
fit_time = time.perf_counter() - t0
val_acc = max(hist.history["val_accuracy"])
print(f"Default CNN, fit time: {fit_time:.1f}s, best val accuracy: {val_acc:.
  ↪4f}")
```

```
Epoch 1/10
200/200 - 13s - 64ms/step - accuracy: 0.3546 - loss: 1.6873 - val_accuracy:
0.5208 - val_loss: 1.2247
Epoch 2/10
200/200 - 12s - 58ms/step - accuracy: 0.5667 - loss: 1.1396 - val_accuracy:
0.5969 - val_loss: 1.0570
Epoch 3/10
200/200 - 11s - 57ms/step - accuracy: 0.6502 - loss: 0.9375 - val_accuracy:
0.6902 - val_loss: 0.8699
Epoch 4/10
```

```
200/200 - 11s - 53ms/step - accuracy: 0.7057 - loss: 0.8031 - val_accuracy:
0.7173 - val_loss: 0.7493
Epoch 5/10
200/200 - 10s - 48ms/step - accuracy: 0.7286 - loss: 0.7413 - val_accuracy:
0.7294 - val_loss: 0.7243
Epoch 6/10
200/200 - 10s - 48ms/step - accuracy: 0.7523 - loss: 0.6822 - val_accuracy:
0.7770 - val_loss: 0.6082
Epoch 7/10
200/200 - 11s - 55ms/step - accuracy: 0.7703 - loss: 0.6286 - val_accuracy:
0.7558 - val_loss: 0.6542
Epoch 8/10
200/200 - 11s - 54ms/step - accuracy: 0.7871 - loss: 0.5910 - val_accuracy:
0.7977 - val_loss: 0.5460
Epoch 9/10
200/200 - 11s - 53ms/step - accuracy: 0.8023 - loss: 0.5374 - val_accuracy:
0.8130 - val_loss: 0.5085
Epoch 10/10
200/200 - 11s - 55ms/step - accuracy: 0.8163 - loss: 0.5017 - val_accuracy:
0.8217 - val_loss: 0.4803
Default CNN, fit time: 109.1s, best val accuracy: 0.8217
```

1.4 3. Hyperparameter tuning

Search strategy. We use a small grid search for all three models. Random search or Bayesian optimisation would be appropriate for larger search spaces; here the spaces are small and exhaustive grid search makes the comparison between configurations easier to interpret.

Validation strategy. - Random Forest uses 3-fold stratified cross-validation via `GridSearchCV` on the 80% training split, with PCA inside an sklearn `Pipeline` so it is refit for each fold. - The neural networks are evaluated on the held-out validation split with early stopping (the validation set acts as our held-out estimate; touching the test set during HP search would leak information).

1.4.1 Algorithm of choice from first six weeks of course

[19]:
```python
# Random Forest hyperparameter search
# 3 hyperparameters (n_estimators, max_depth, min_samples_split), 8 combinations
rf_grid = {
    "rf__n_estimators": [100, 300],
    "rf__max_depth": [None, 20],
    "rf__min_samples_split": [2, 5],
}
rf_pipeline = Pipeline([
    ("pca", PCA(n_components=50, random_state=RANDOM_SEED)),
    ("rf", RandomForestClassifier(random_state=RANDOM_SEED, n_jobs=-1)),
])

cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=RANDOM_SEED)
```

```
gs = GridSearchCV(
    rf_pipeline,
    rf_grid,
    cv=cv,
    scoring="accuracy",
    n_jobs=1,  # RF already parallelises internally
    return_train_score=False,
    verbose=1,
)
t0 = time.perf_counter()
gs.fit(X_tr_flat, y_tr)
rf_search_time = time.perf_counter() - t0

rf_results = pd.DataFrame(gs.cv_results_)[
    ["param_rf__n_estimators", "param_rf__max_depth",␣
 ↪"param_rf__min_samples_split",
     "mean_test_score", "std_test_score", "mean_fit_time"]
].rename(columns={
    "param_rf__n_estimators": "param_n_estimators",
    "param_rf__max_depth": "param_max_depth",
    "param_rf__min_samples_split": "param_min_samples_split",
}).sort_values("mean_test_score", ascending=False).reset_index(drop=True)
print(f"\nTotal RF search time: {rf_search_time:.1f}s")
best_rf_clean = {k.replace("rf__", ""): v for k, v in gs.best_params_.items()}
print(f"Best params: {best_rf_clean}")
print(f"Best CV accuracy: {gs.best_score_:.4f}")
rf_results
```

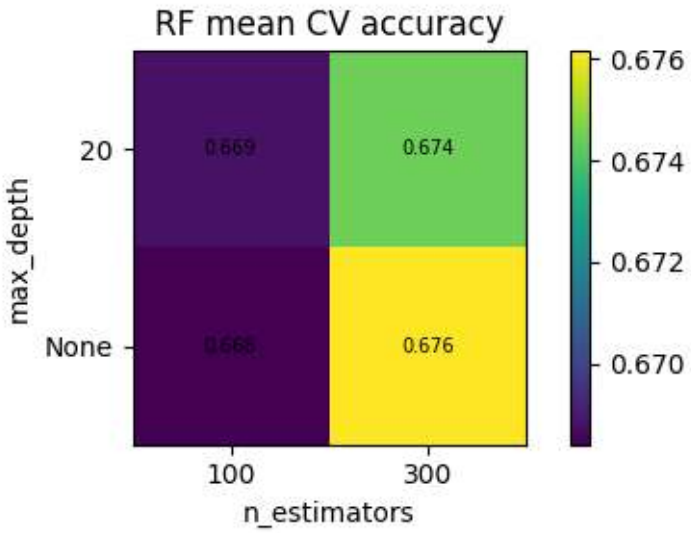
```
Fitting 3 folds for each of 8 candidates, totalling 24 fits

Total RF search time: 80.8s
Best params: {'max_depth': None, 'min_samples_split': 5, 'n_estimators': 300}
Best CV accuracy: 0.6772
```

[19]:

	param_n_estimators	param_max_depth	param_min_samples_split	mean_test_score	std_test_score	mean_fit_time
0	300	None	5	0.677187	0.006835	4.105490
1	300	20	5	0.675351	0.006462	4.627174
2	300	None	2			
3	300	20	2			
4	100	20	5			
5	100	None	5			
6	100	20	2			
7	100	None	2			

```
2          0.675117          0.007420          4.574369
3          0.673516          0.006679          3.943926
4          0.669258          0.006020          1.667764
5          0.668867          0.005746          1.632749
6          0.668359          0.008629          1.503824
7          0.667968          0.008127          1.781022
```

[20]:
```python
# Heatmap of mean CV accuracy across (n_estimators, max_depth), averaged
# over min_samples_split for compact visualisation
heat = rf_results.copy()
heat["max_depth"] = heat["param_max_depth"].astype(str)
heat["n_estimators"] = heat["param_n_estimators"].astype(int)
pivot = heat.groupby(["max_depth", "n_estimators"])["mean_test_score"].mean().
 ↪unstack()

plt.figure(figsize=(5, 3))
sns.heatmap(
    pivot, annot=True, fmt=".3f", cmap="viridis",
    xticklabels=pivot.columns.astype(str),
    yticklabels=pivot.index.astype(str),
)
plt.title("RF mean CV accuracy")
plt.xlabel("n_estimators")
plt.ylabel("max_depth")
plt.tight_layout()
plt.show()
```



1.4.2 Fully connected neural network

[21]:
```
# MLP hyperparameter search, 3 hyperparameters
mlp_grid = {
    "n_layers": [1, 2, 3],
    "hidden_units": [128, 256],
    "learning_rate": [1e-3, 1e-4],
}
mlp_combos = list(product(
    mlp_grid["n_layers"], mlp_grid["hidden_units"], mlp_grid["learning_rate"]
))
print(f"MLP search: {len(mlp_combos)} combinations\n")

EPOCHS_SEARCH = 15
BATCH_SIZE = 128
mlp_records = []

for i, (n_layers, hidden_units, lr) in enumerate(mlp_combos, 1):
    tf.keras.backend.clear_session()
    tf.random.set_seed(RANDOM_SEED)
    model = build_mlp(n_layers=n_layers, hidden_units=hidden_units,
                      dropout=0.3, learning_rate=lr)
    es = EarlyStopping(monitor="val_loss", patience=3,␣
 ↪restore_best_weights=True)
    t0 = time.perf_counter()
    hist = model.fit(
        X_tr_mlp, y_tr_oh,
        validation_data=(X_val_mlp, y_val_oh),
        epochs=EPOCHS_SEARCH, batch_size=BATCH_SIZE,
        callbacks=[es], verbose=0,
    )
    elapsed = time.perf_counter() - t0
    val_acc = max(hist.history["val_accuracy"])
    val_loss = min(hist.history["val_loss"])
    epochs_run = len(hist.history["val_loss"])
    mlp_records.append({
        "n_layers": n_layers,
        "hidden_units": hidden_units,
        "learning_rate": lr,
        "val_accuracy": val_acc,
        "val_loss": val_loss,
        "epochs_run": epochs_run,
        "fit_time_s": elapsed,
    })
    print(f"[{i:2d}/{len(mlp_combos)}] layers={n_layers}, units={hidden_units},␣
 ↪"
          f"lr={lr:.0e} -> val_acc={val_acc:.4f} ({elapsed:.1f}s)")
```

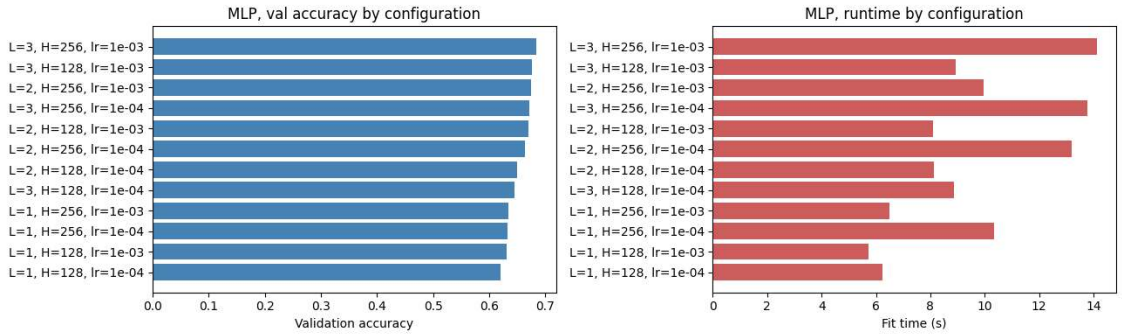
```
mlp_results = pd.DataFrame(mlp_records).sort_values("val_accuracy",␣
  ↪ascending=False).reset_index(drop=True)
mlp_results
```

```
MLP search: 12 combinations

[ 1/12] layers=1, units=128, lr=1e-03 -> val_acc=0.6309 (5.7s)
[ 2/12] layers=1, units=128, lr=1e-04 -> val_acc=0.6202 (6.3s)
[ 3/12] layers=1, units=256, lr=1e-03 -> val_acc=0.6336 (6.5s)
[ 4/12] layers=1, units=256, lr=1e-04 -> val_acc=0.6330 (10.4s)
[ 5/12] layers=2, units=128, lr=1e-03 -> val_acc=0.6706 (8.1s)
[ 6/12] layers=2, units=128, lr=1e-04 -> val_acc=0.6495 (8.1s)
[ 7/12] layers=2, units=256, lr=1e-03 -> val_acc=0.6750 (10.0s)
[ 8/12] layers=2, units=256, lr=1e-04 -> val_acc=0.6642 (13.2s)
[ 9/12] layers=3, units=128, lr=1e-03 -> val_acc=0.6756 (8.9s)
[10/12] layers=3, units=128, lr=1e-04 -> val_acc=0.6455 (8.9s)
[11/12] layers=3, units=256, lr=1e-03 -> val_acc=0.6848 (14.1s)
[12/12] layers=3, units=256, lr=1e-04 -> val_acc=0.6717 (13.8s)
```

[21]:

	n_layers	hidden_units	learning_rate	val_accuracy	val_loss	epochs_run	fit_time_s
0	3	256	0.0010	0.684844	0.856346	15	14.127074
1	3	128	0.0010	0.675625	0.866620	15	8.935274
2	2	256	0.0010	0.675000	0.876587	12	9.971508
3	3	256	0.0001	0.671719	0.882915	15	13.790058
4	2	128	0.0010	0.670625	0.875213	13	8.082743
5	2	256	0.0001	0.664219	0.906143	15	13.192636
6	2	128	0.0001	0.649531	0.940858	15	8.145820
7	3	128	0.0001	0.645469	0.929092	15	8.876338
8	1	256	0.0010	0.633594	0.981652	10	6.488078
9	1	256	0.0001	0.632969	1.016400	15	10.358307
10	1	128	0.0010	0.630938	0.980814	10	5.716144
11	1	128	0.0001	0.620156	1.033230	15	6.251781

[22]:
```
# Visualise the MLP search
fig, axes = plt.subplots(1, 2, figsize=(13, 4))
labels = [f"L={r.n_layers}, H={r.hidden_units}, lr={r.learning_rate:.0e}"
          for r in mlp_results.itertuples()]
axes[0].barh(labels, mlp_results["val_accuracy"], color="steelblue")
axes[0].set_xlabel("Validation accuracy")
axes[0].set_title("MLP, val accuracy by configuration")
axes[0].invert_yaxis()

axes[1].barh(labels, mlp_results["fit_time_s"], color="indianred")
axes[1].set_xlabel("Fit time (s)")
axes[1].set_title("MLP, runtime by configuration")
axes[1].invert_yaxis()
plt.tight_layout()
plt.show()

best_mlp = mlp_results.iloc[0]
print(f"\nBest MLP: layers={int(best_mlp.n_layers)}, units={int(best_mlp.
  ↪hidden_units)}, "
      f"lr={best_mlp.learning_rate:.0e}, val_acc={best_mlp.val_accuracy:.4f}")
```




```
Best MLP: layers=3, units=256, lr=1e-03, val_acc=0.6848
```

1.4.3 Convolutional neural network

[23]:
```
# CNN hyperparameter search, 3 hyperparameters
cnn_grid = {
    "num_conv_blocks": [2, 3],
    "base_filters": [16, 32],
    "dropout": [0.25, 0.5],
}
cnn_combos = list(product(
    cnn_grid["num_conv_blocks"], cnn_grid["base_filters"], cnn_grid["dropout"]
))
```

```
print(f"CNN search: {len(cnn_combos)} combinations\n")

cnn_records = []

for i, (n_blocks, base_f, drop) in enumerate(cnn_combos, 1):
    tf.keras.backend.clear_session()
    tf.random.set_seed(RANDOM_SEED)
    model = build_cnn(num_conv_blocks=n_blocks, base_filters=base_f,
                      dense_units=128, dropout=drop, learning_rate=1e-3)
    es = EarlyStopping(monitor="val_loss", patience=3,␣
 ↪restore_best_weights=True)
    t0 = time.perf_counter()
    hist = model.fit(
        X_tr_cnn, y_tr_oh,
        validation_data=(X_val_cnn, y_val_oh),
        epochs=EPOCHS_SEARCH, batch_size=BATCH_SIZE,
        callbacks=[es], verbose=0,
    )
    elapsed = time.perf_counter() - t0
    val_acc = max(hist.history["val_accuracy"])
    val_loss = min(hist.history["val_loss"])
    epochs_run = len(hist.history["val_loss"])
    cnn_records.append({
        "num_conv_blocks": n_blocks,
        "base_filters": base_f,
        "dropout": drop,
        "val_accuracy": val_acc,
        "val_loss": val_loss,
        "epochs_run": epochs_run,
        "fit_time_s": elapsed,
    })
    print(f"[{i}/{len(cnn_combos)}] blocks={n_blocks}, filters={base_f}, "
          f"dropout={drop} -> val_acc={val_acc:.4f} ({elapsed:.1f}s)")

cnn_results = pd.DataFrame(cnn_records).sort_values("val_accuracy",␣
 ↪ascending=False).reset_index(drop=True)
cnn_results
```

```
CNN search: 8 combinations

[1/8] blocks=2, filters=16, dropout=0.25 -> val_acc=0.8031 (78.1s)
[2/8] blocks=2, filters=16, dropout=0.5 -> val_acc=0.7914 (67.1s)
[3/8] blocks=2, filters=32, dropout=0.25 -> val_acc=0.8512 (140.4s)
[4/8] blocks=2, filters=32, dropout=0.5 -> val_acc=0.8477 (137.1s)
[5/8] blocks=3, filters=16, dropout=0.25 -> val_acc=0.8078 (83.7s)
[6/8] blocks=3, filters=16, dropout=0.5 -> val_acc=0.8066 (73.8s)
[7/8] blocks=3, filters=32, dropout=0.25 -> val_acc=0.8630 (219.2s)
```

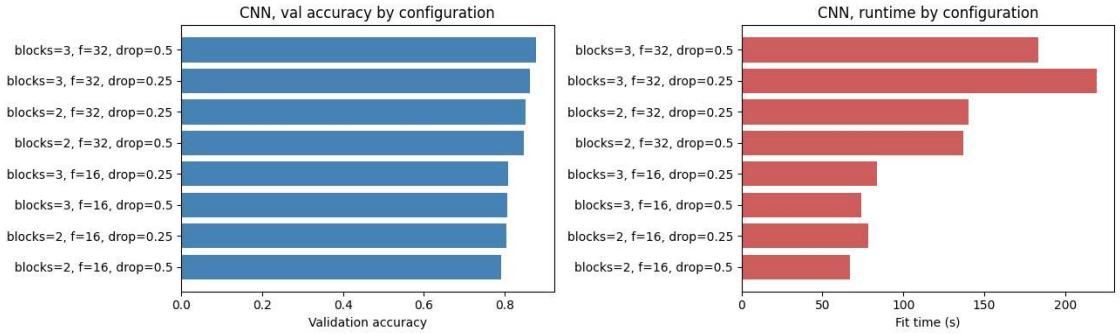
```
[8/8] blocks=3, filters=32, dropout=0.5 -> val_acc=0.8777 (183.3s)
```

[23]:

	num_conv_blocks	base_filters	dropout	val_accuracy	val_loss	epochs_run	fit_time_s
0	3	32	0.50	0.877656	0.333920	15	183.327921
1	3	32	0.25	0.862969	0.384821	15	219.231464
2	2	32	0.25	0.851250	0.405578	15	140.359234
3	2	32	0.50	0.847656	0.419767	15	137.081493
4	3	16	0.25	0.807813	0.504013	15	83.670105
5	3	16	0.50	0.806562	0.530043	13	73.839129
6	2	16	0.25	0.803125	0.531144	12	78.086348
7	2	16	0.50	0.791406	0.569519	15	67.111012

[24]:

```python
# Visualise the CNN search
fig, axes = plt.subplots(1, 2, figsize=(13, 4))
labels = [f"blocks={r.num_conv_blocks}, f={r.base_filters}, drop={r.dropout}"
          for r in cnn_results.itertuples()]
axes[0].barh(labels, cnn_results["val_accuracy"], color="steelblue")
axes[0].set_xlabel("Validation accuracy")
axes[0].set_title("CNN, val accuracy by configuration")
axes[0].invert_yaxis()

axes[1].barh(labels, cnn_results["fit_time_s"], color="indianred")
axes[1].set_xlabel("Fit time (s)")
axes[1].set_title("CNN, runtime by configuration")
axes[1].invert_yaxis()
plt.tight_layout()
plt.show()

best_cnn = cnn_results.iloc[0]
print(f"\nBest CNN: blocks={int(best_cnn.num_conv_blocks)}, "
      f"filters={int(best_cnn.base_filters)}, dropout={best_cnn.dropout}, "
      f"val_acc={best_cnn.val_accuracy:.4f}")
```



`Best CNN: blocks=3, filters=32, dropout=0.5, val_acc=0.8777`

Best hyperparameters chosen (the cells in section 4 hardcode these values; if you re-run the searches above and a different combination wins, update the constants at the top of section 4 accordingly):

- **Random Forest:** `n_estimators=300, max_depth=None, min_samples_split=5`
- **MLP:** `n_layers=3, hidden_units=256, learning_rate=1e-3`, dropout 0.3
- **CNN:** `num_conv_blocks=3, base_filters=32, dropout=0.50`, lr 1e-3

1.5 4. Final models

[25]:
```
# Best hyperparameters from section 3, hardcoded so this section runs
# independently of the hyperparameter search cells.
BEST_RF = {"n_estimators": 300, "max_depth": None, "min_samples_split": 5}
BEST_MLP = {"n_layers": 3, "hidden_units": 256, "dropout": 0.3, "learning_rate":
 ↪ 1e-3}
BEST_CNN = {"num_conv_blocks": 3, "base_filters": 32, "dense_units": 128,
            "dropout": 0.50, "learning_rate": 1e-3}

FINAL_EPOCHS = 25
FINAL_BATCH = 128

# Container for the comparison table at the end
final_results = []
```

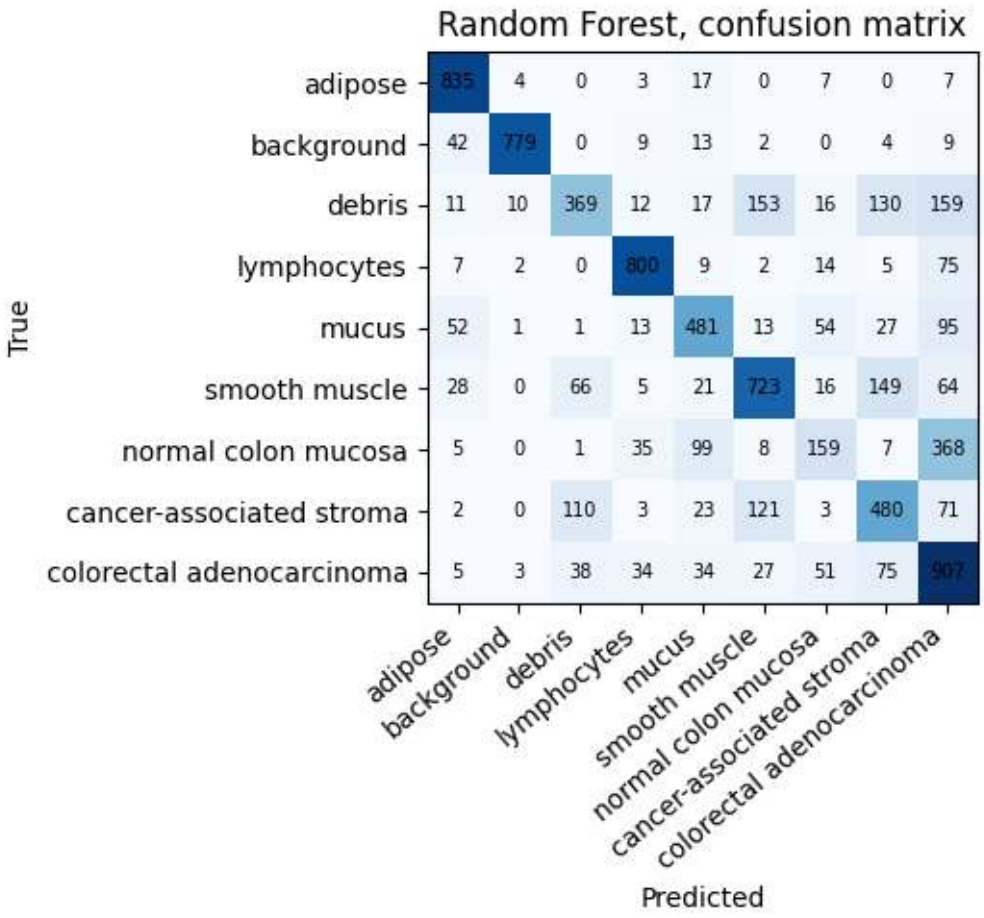
1.5.1 Algorithm of choice from first six weeks of course

[26]:
```
# Final Random Forest pipeline, trained on the FULL training set with PCA(50)
rf_final = Pipeline([
    ("pca", PCA(n_components=50, random_state=RANDOM_SEED)),
    ("rf", RandomForestClassifier(
        n_estimators=BEST_RF["n_estimators"],
        max_depth=BEST_RF["max_depth"],
        min_samples_split=BEST_RF["min_samples_split"],
```

```
        random_state=RANDOM_SEED,
        n_jobs=-1,
    )),
])
t0 = time.perf_counter()
rf_final.fit(X_train_u.reshape(len(X_train_u), -1), y_train)
rf_train_time = time.perf_counter() - t0

t0 = time.perf_counter()
y_pred_rf = rf_final.predict(X_test_flat)
rf_pred_time = time.perf_counter() - t0

rf_acc = accuracy_score(y_test, y_pred_rf)
rf_f1 = f1_score(y_test, y_pred_rf, average="macro")
print(f"Random Forest -- test accuracy: {rf_acc:.4f}, macro-F1: {rf_f1:.4f}")
print(f"Train time: {rf_train_time:.1f}s, prediction time: {rf_pred_time:.
 ↪2f}s\n")
print(classification_report(y_test, y_pred_rf, target_names=CLASS_NAMES,␣
 ↪digits=3))

cm = confusion_matrix(y_test, y_pred_rf)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=CLASS_NAMES, yticklabels=CLASS_NAMES, cbar=False)
plt.xticks(rotation=40, ha="right")
plt.yticks(rotation=0)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Random Forest, confusion matrix")
plt.tight_layout()
plt.show()

final_results.append({
    "model": "Random Forest",
    "test_accuracy": rf_acc,
    "macro_f1": rf_f1,
    "train_time_s": rf_train_time,
    "predict_time_s": rf_pred_time,
    "params": "n_est=300, max_depth=None, min_split=5 (PCA=50)",
})
```

```
Random Forest -- test accuracy: 0.6916, macro-F1: 0.6742
Train time: 7.7s, prediction time: 0.10s

                    precision    recall  f1-score   support

           adipose      0.846     0.956     0.898       873
```

background	0.975	0.908	0.940	858
debris	0.631	0.421	0.505	877
lymphocytes	0.875	0.875	0.875	914
mucus	0.674	0.653	0.663	737
smooth muscle	0.689	0.674	0.682	1072
normal colon mucosa	0.497	0.233	0.317	682
cancer-associated stroma	0.547	0.590	0.568	813
colorectal adenocarcinoma	0.517	0.773	0.619	1174
accuracy			0.692	8000
macro avg	0.695	0.676	0.674	8000
weighted avg	0.694	0.692	0.682	8000



1.5.2 Fully connected neural network

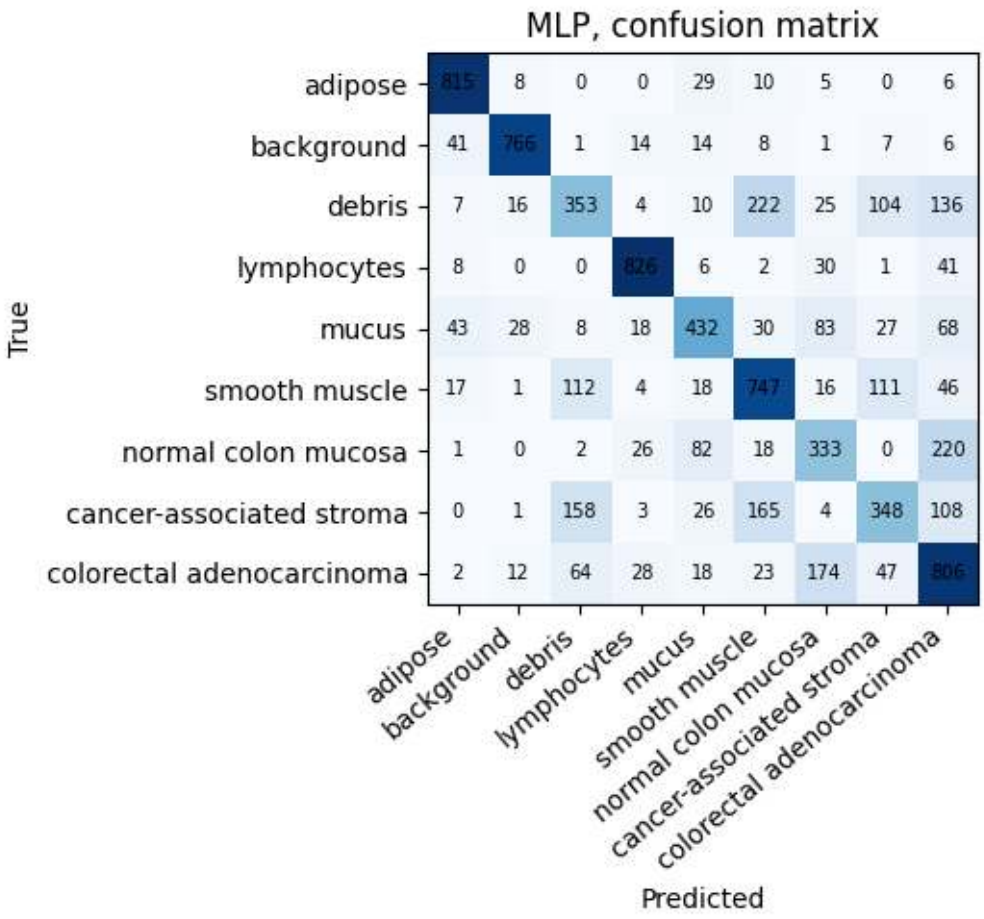
[27]:
```
# Final MLP, trained on the FULL training set, small slice held back
# only for early stopping
tf.keras.backend.clear_session()
tf.random.set_seed(RANDOM_SEED)
mlp_final = build_mlp(
    n_layers=BEST_MLP["n_layers"],
    hidden_units=BEST_MLP["hidden_units"],
    dropout=BEST_MLP["dropout"],
    learning_rate=BEST_MLP["learning_rate"],
)

# Re-do a small stratified split on the full training data for early stopping.
# We use the previously-fit scaler_full so test-set processing matches.
X_tr_mlp_f, X_val_mlp_f, y_tr_oh_f, y_val_oh_f = train_test_split(
    X_train_mlp_full, y_train_oh_full,
    test_size=0.1, stratify=y_train, random_state=RANDOM_SEED,
)

es = EarlyStopping(monitor="val_loss", patience=4, restore_best_weights=True)
t0 = time.perf_counter()
hist = mlp_final.fit(
    X_tr_mlp_f, y_tr_oh_f,
    validation_data=(X_val_mlp_f, y_val_oh_f),
    epochs=FINAL_EPOCHS, batch_size=FINAL_BATCH,
    callbacks=[es], verbose=2,
)
mlp_train_time = time.perf_counter() - t0

t0 = time.perf_counter()
y_pred_mlp_proba = mlp_final.predict(X_test_mlp_full, batch_size=FINAL_BATCH,␣
 ↪verbose=0)
mlp_pred_time = time.perf_counter() - t0
y_pred_mlp = y_pred_mlp_proba.argmax(axis=1)

mlp_acc = accuracy_score(y_test, y_pred_mlp)
mlp_f1 = f1_score(y_test, y_pred_mlp, average="macro")
print(f"\nMLP -- test accuracy: {mlp_acc:.4f}, macro-F1: {mlp_f1:.4f}")
print(f"Train time: {mlp_train_time:.1f}s, prediction time: {mlp_pred_time:.
 ↪2f}s\n")
print(classification_report(y_test, y_pred_mlp, target_names=CLASS_NAMES,␣
 ↪digits=3))

cm = confusion_matrix(y_test, y_pred_mlp)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
```

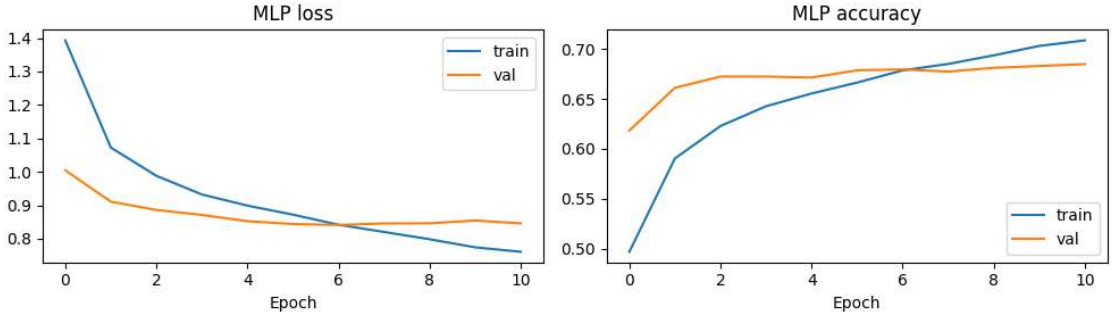
```
            xticklabels=CLASS_NAMES, yticklabels=CLASS_NAMES, cbar=False)
plt.xticks(rotation=40, ha="right")
plt.yticks(rotation=0)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("MLP, confusion matrix")
plt.tight_layout()
plt.show()

# Training curve
fig, ax = plt.subplots(1, 2, figsize=(10, 3))
ax[0].plot(hist.history["loss"], label="train")
ax[0].plot(hist.history["val_loss"], label="val")
ax[0].set_title("MLP loss"), ax[0].set_xlabel("Epoch"), ax[0].legend()
ax[1].plot(hist.history["accuracy"], label="train")
ax[1].plot(hist.history["val_accuracy"], label="val")
ax[1].set_title("MLP accuracy"), ax[1].set_xlabel("Epoch"), ax[1].legend()
plt.tight_layout()
plt.show()

final_results.append({
    "model": "MLP",
    "test_accuracy": mlp_acc,
    "macro_f1": mlp_f1,
    "train_time_s": mlp_train_time,
    "predict_time_s": mlp_pred_time,
    "params": f"layers={BEST_MLP['n_layers']},␣
 ↪units={BEST_MLP['hidden_units']}, "
              f"lr={BEST_MLP['learning_rate']:.0e},␣
 ↪dropout={BEST_MLP['dropout']}",
})
```

```
Epoch 1/25
225/225 - 2s - 10ms/step - accuracy: 0.4968 - loss: 1.3931 - val_accuracy:
0.6184 - val_loss: 1.0048
Epoch 2/25
225/225 - 1s - 4ms/step - accuracy: 0.5904 - loss: 1.0726 - val_accuracy: 0.6612
- val_loss: 0.9109
Epoch 3/25
225/225 - 1s - 4ms/step - accuracy: 0.6229 - loss: 0.9882 - val_accuracy: 0.6725
- val_loss: 0.8861
Epoch 4/25
225/225 - 1s - 4ms/step - accuracy: 0.6427 - loss: 0.9325 - val_accuracy: 0.6725
- val_loss: 0.8712
Epoch 5/25
225/225 - 1s - 4ms/step - accuracy: 0.6555 - loss: 0.8991 - val_accuracy: 0.6716
- val_loss: 0.8527
```

```
Epoch 6/25
225/225 - 1s - 4ms/step - accuracy: 0.6665 - loss: 0.8721 - val_accuracy: 0.6787
- val_loss: 0.8440
Epoch 7/25
225/225 - 1s - 4ms/step - accuracy: 0.6787 - loss: 0.8419 - val_accuracy: 0.6797
- val_loss: 0.8413
Epoch 8/25
225/225 - 1s - 4ms/step - accuracy: 0.6852 - loss: 0.8204 - val_accuracy: 0.6775
- val_loss: 0.8459
Epoch 9/25
225/225 - 1s - 4ms/step - accuracy: 0.6938 - loss: 0.7986 - val_accuracy: 0.6812
- val_loss: 0.8463
Epoch 10/25
225/225 - 1s - 4ms/step - accuracy: 0.7032 - loss: 0.7744 - val_accuracy: 0.6831
- val_loss: 0.8546
Epoch 11/25
225/225 - 1s - 4ms/step - accuracy: 0.7088 - loss: 0.7611 - val_accuracy: 0.6850
- val_loss: 0.8461

MLP -- test accuracy: 0.6783, macro-F1: 0.6692
Train time: 10.3s, prediction time: 0.20s
```

	precision	recall	f1-score	support
adipose	0.873	0.934	0.902	873
background	0.921	0.893	0.907	858
debris	0.506	0.403	0.448	877
lymphocytes	0.895	0.904	0.899	914
mucus	0.680	0.586	0.630	737
smooth muscle	0.610	0.697	0.650	1072
normal colon mucosa	0.496	0.488	0.492	682
cancer-associated stroma	0.540	0.428	0.477	813
colorectal adenocarcinoma	0.561	0.687	0.617	1174
accuracy			0.678	8000
macro avg	0.676	0.669	0.669	8000
weighted avg	0.675	0.678	0.674	8000





1.5.3 Convolutional neural network

[28]:
```
# Final CNN, trained on the FULL training set
tf.keras.backend.clear_session()
tf.random.set_seed(RANDOM_SEED)
cnn_final = build_cnn(
```

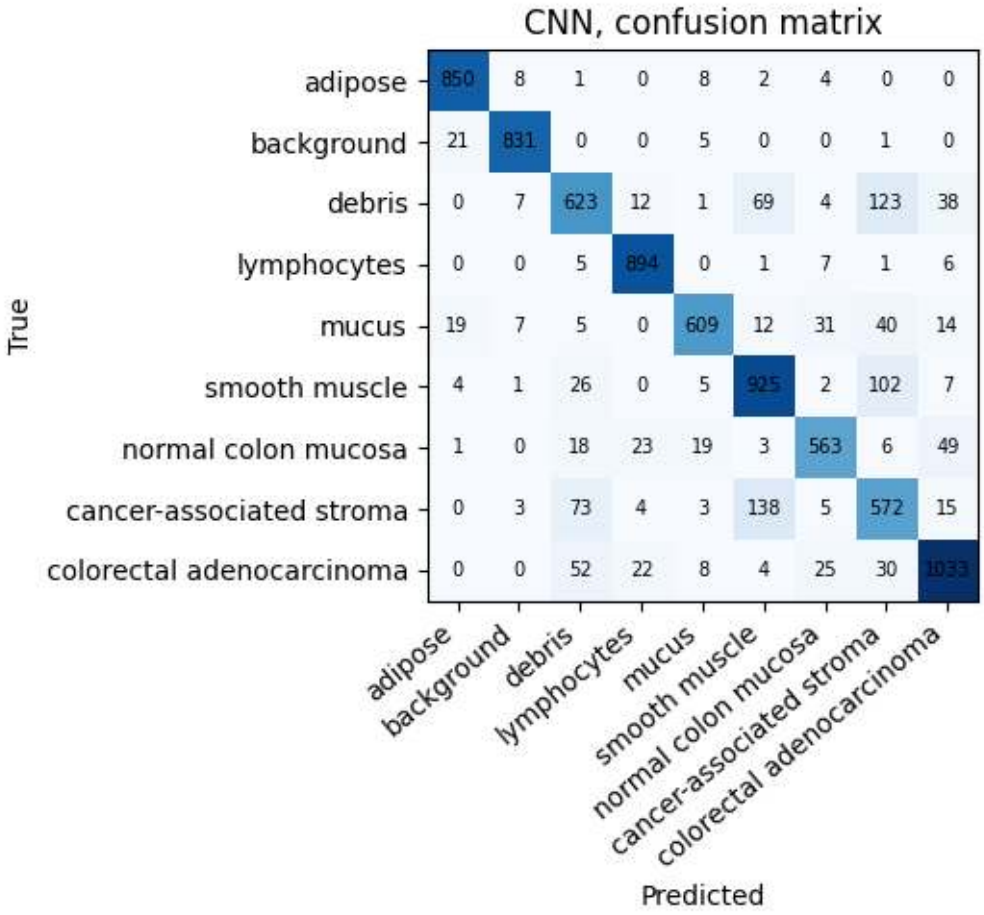
```
    num_conv_blocks=BEST_CNN["num_conv_blocks"],
    base_filters=BEST_CNN["base_filters"],
    dense_units=BEST_CNN["dense_units"],
    dropout=BEST_CNN["dropout"],
    learning_rate=BEST_CNN["learning_rate"],
)

# Re-do a small stratified split for early stopping
X_tr_cnn_f, X_val_cnn_f, y_tr_oh_cf, y_val_oh_cf = train_test_split(
    X_train_u, y_train_oh_full,
    test_size=0.1, stratify=y_train, random_state=RANDOM_SEED,
)

es = EarlyStopping(monitor="val_loss", patience=4, restore_best_weights=True)
t0 = time.perf_counter()
hist = cnn_final.fit(
    X_tr_cnn_f, y_tr_oh_cf,
    validation_data=(X_val_cnn_f, y_val_oh_cf),
    epochs=FINAL_EPOCHS, batch_size=FINAL_BATCH,
    callbacks=[es], verbose=2,
)
cnn_train_time = time.perf_counter() - t0

t0 = time.perf_counter()
y_pred_cnn_proba = cnn_final.predict(X_test_u, batch_size=FINAL_BATCH,␣
 ↪verbose=0)
cnn_pred_time = time.perf_counter() - t0
y_pred_cnn = y_pred_cnn_proba.argmax(axis=1)

cnn_acc = accuracy_score(y_test, y_pred_cnn)
cnn_f1 = f1_score(y_test, y_pred_cnn, average="macro")
print(f"\nCNN -- test accuracy: {cnn_acc:.4f}, macro-F1: {cnn_f1:.4f}")
print(f"Train time: {cnn_train_time:.1f}s, prediction time: {cnn_pred_time:.
 ↪2f}s\n")
print(classification_report(y_test, y_pred_cnn, target_names=CLASS_NAMES,␣
 ↪digits=3))

cm = confusion_matrix(y_test, y_pred_cnn)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=CLASS_NAMES, yticklabels=CLASS_NAMES, cbar=False)
plt.xticks(rotation=40, ha="right")
plt.yticks(rotation=0)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("CNN, confusion matrix")
plt.tight_layout()
```

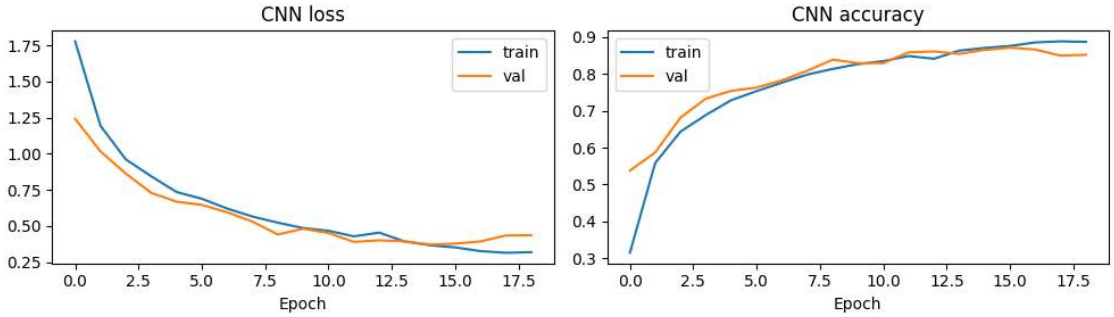
```
plt.show()

fig, ax = plt.subplots(1, 2, figsize=(10, 3))
ax[0].plot(hist.history["loss"], label="train")
ax[0].plot(hist.history["val_loss"], label="val")
ax[0].set_title("CNN loss"), ax[0].set_xlabel("Epoch"), ax[0].legend()
ax[1].plot(hist.history["accuracy"], label="train")
ax[1].plot(hist.history["val_accuracy"], label="val")
ax[1].set_title("CNN accuracy"), ax[1].set_xlabel("Epoch"), ax[1].legend()
plt.tight_layout()
plt.show()

final_results.append({
    "model": "CNN",
    "test_accuracy": cnn_acc,
    "macro_f1": cnn_f1,
    "train_time_s": cnn_train_time,
    "predict_time_s": cnn_pred_time,
    "params": f"blocks={BEST_CNN['num_conv_blocks']},␣
 ↪filters={BEST_CNN['base_filters']}, "
              f"dropout={BEST_CNN['dropout']}, lr={BEST_CNN['learning_rate']:.
 ↪0e}",
})
```

```
Epoch 1/25
225/225 - 14s - 61ms/step - accuracy: 0.3145 - loss: 1.7768 - val_accuracy:
0.5375 - val_loss: 1.2411
Epoch 2/25
225/225 - 12s - 54ms/step - accuracy: 0.5587 - loss: 1.1928 - val_accuracy:
0.5872 - val_loss: 1.0152
Epoch 3/25
225/225 - 12s - 54ms/step - accuracy: 0.6436 - loss: 0.9601 - val_accuracy:
0.6816 - val_loss: 0.8628
Epoch 4/25
225/225 - 13s - 56ms/step - accuracy: 0.6888 - loss: 0.8436 - val_accuracy:
0.7331 - val_loss: 0.7286
Epoch 5/25
225/225 - 12s - 54ms/step - accuracy: 0.7288 - loss: 0.7349 - val_accuracy:
0.7541 - val_loss: 0.6684
Epoch 6/25
225/225 - 12s - 55ms/step - accuracy: 0.7534 - loss: 0.6880 - val_accuracy:
0.7631 - val_loss: 0.6455
Epoch 7/25
225/225 - 12s - 54ms/step - accuracy: 0.7764 - loss: 0.6203 - val_accuracy:
0.7828 - val_loss: 0.5949
Epoch 8/25
225/225 - 12s - 54ms/step - accuracy: 0.7983 - loss: 0.5650 - val_accuracy:
```

```
0.8087 - val_loss: 0.5298
Epoch 9/25
225/225 - 12s - 54ms/step - accuracy: 0.8136 - loss: 0.5226 - val_accuracy:
0.8388 - val_loss: 0.4410
Epoch 10/25
225/225 - 12s - 54ms/step - accuracy: 0.8267 - loss: 0.4855 - val_accuracy:
0.8294 - val_loss: 0.4805
Epoch 11/25
225/225 - 14s - 62ms/step - accuracy: 0.8349 - loss: 0.4663 - val_accuracy:
0.8294 - val_loss: 0.4510
Epoch 12/25
225/225 - 13s - 57ms/step - accuracy: 0.8486 - loss: 0.4282 - val_accuracy:
0.8587 - val_loss: 0.3897
Epoch 13/25
225/225 - 12s - 55ms/step - accuracy: 0.8413 - loss: 0.4537 - val_accuracy:
0.8609 - val_loss: 0.4006
Epoch 14/25
225/225 - 12s - 55ms/step - accuracy: 0.8632 - loss: 0.3930 - val_accuracy:
0.8544 - val_loss: 0.3941
Epoch 15/25
225/225 - 13s - 59ms/step - accuracy: 0.8706 - loss: 0.3665 - val_accuracy:
0.8653 - val_loss: 0.3714
Epoch 16/25
225/225 - 14s - 62ms/step - accuracy: 0.8759 - loss: 0.3516 - val_accuracy:
0.8712 - val_loss: 0.3776
Epoch 17/25
225/225 - 12s - 55ms/step - accuracy: 0.8856 - loss: 0.3258 - val_accuracy:
0.8662 - val_loss: 0.3930
Epoch 18/25
225/225 - 12s - 55ms/step - accuracy: 0.8885 - loss: 0.3143 - val_accuracy:
0.8497 - val_loss: 0.4340
Epoch 19/25
225/225 - 13s - 57ms/step - accuracy: 0.8870 - loss: 0.3189 - val_accuracy:
0.8522 - val_loss: 0.4363

CNN -- test accuracy: 0.8625, macro-F1: 0.8607
Train time: 240.4s, prediction time: 1.14s

                         precision    recall  f1-score   support

                adipose      0.950     0.974     0.962       873
             background      0.970     0.969     0.969       858
                 debris      0.776     0.710     0.742       877
            lymphocytes      0.936     0.978     0.957       914
                  mucus      0.926     0.826     0.873       737
          smooth muscle      0.802     0.863     0.831      1072
     normal colon mucosa      0.878     0.826     0.851       682
 cancer-associated stroma      0.654     0.704     0.678       813
```

colorectal adenocarcinoma	0.889	0.880	0.884	1174
accuracy			0.863	8000
macro avg	0.864	0.859	0.861	8000
weighted avg	0.864	0.863	0.862	8000





1.5.4 Final comparison

Side-by-side comparison of the three models on the held-out test set, including the best hyperparameter combination, accuracy, macro-F1, and wall-clock training/inference times.

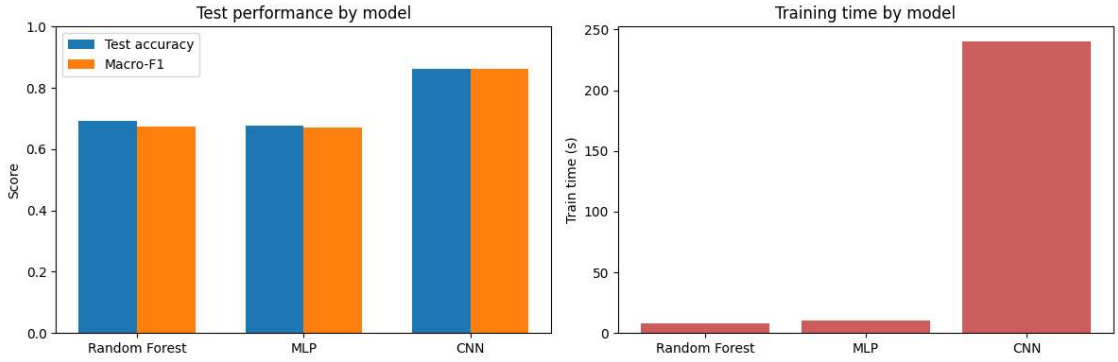
[29]:
```
summary = pd.DataFrame(final_results)
summary["test_accuracy"] = summary["test_accuracy"].map(lambda x: float(f"{x:.
↪4f}"))
summary["macro_f1"] = summary["macro_f1"].map(lambda x: float(f"{x:.4f}"))
summary["train_time_s"] = summary["train_time_s"].round(1)
summary["predict_time_s"] = summary["predict_time_s"].round(2)
summary
```

[29]:

	model	test_accuracy	macro_f1	train_time_s	predict_time_s	\
0	Random Forest	0.6916	0.6742	7.7	0.10	
1	MLP	0.6783	0.6692	10.3	0.20	
2	CNN	0.8625	0.8607	240.4	1.14	

	params
0	n_est=300, max_depth=None, min_split=5 (PCA=50)
1	layers=3, units=256, lr=1e-03, dropout=0.3
2	blocks=3, filters=32, dropout=0.5, lr=1e-03

[30]:
```
# Visual comparison: accuracy + macro-F1 side by side, and a runtime bar chart
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

x = np.arange(len(summary))
width = 0.35
axes[0].bar(x - width / 2, summary["test_accuracy"], width, label="Test␣
↪accuracy")
axes[0].bar(x + width / 2, summary["macro_f1"], width, label="Macro-F1")
axes[0].set_xticks(x)
axes[0].set_xticklabels(summary["model"])
axes[0].set_ylim(0, 1)
axes[0].set_ylabel("Score")
axes[0].set_title("Test performance by model")
axes[0].legend()

axes[1].bar(summary["model"], summary["train_time_s"], color="indianred")
axes[1].set_ylabel("Train time (s)")
axes[1].set_title("Training time by model")
plt.tight_layout()
plt.show()
```



1.6 5. AI Acknowledgement

What generative AI tools have you used to complete A2?

OpenAI Codex.

How have you used generative AI tools in A2?

Codex was used as an assistant to review the notebook and report against the assignment specification, suggest wording improvements, and identify formatting/compliance issues such as dependency listing and acknowledgement completeness. The group reviewed the generated suggestions and remained responsible for the final code, results, analysis, and submission.