Lab: Computer Vision

ACTL3143 & ACTL5111 Deep Learning for Actuaries

Learning Goal

This lab uses one connected problem throughout: an insurer wants to classify vehicle inspection photos as either `damage` or `whole`.

The lab is grouped by topic. Each topic has numbered problems and a short concept explanation. Your job is to **work out the answers yourself** — by hand for the small calculations, and by completing the code stubs marked `# TODO` for the modelling parts.

> ❗ How to use this worksheet
>
> - Each problem has a **Concept** box explaining the key idea. Read it if you are stuck, and revisit the Week 3 lecture notes for more detail.
> - Code cells are intentionally left incomplete. Fill in the `# TODO` parts and run them in Jupyter.

Dataset used in the real-data section:

https://www.kaggle.com/datasets/anujms/car-damage-detection

Please download and unzip it into a `data/` folder beside this notebook, so the images live under `data/data1a/training/` and `data/data1a/validation/`.

```
import os
# Keras runs on the PyTorch backend in this course. This must be set before
# importing keras (otherwise Keras looks for TensorFlow and errors out).
os.environ["KERAS_BACKEND"] = "torch"

from pathlib import Path

import numpy as np
import matplotlib.pyplot as plt
```

```python
from PIL import Image

import keras
from keras import layers

SEED = 3143
keras.utils.set_random_seed(SEED)

IMG_SIZE = (96, 128)      # height, width
BATCH_SIZE = 32
INPUT_SHAPE = IMG_SIZE + (3,)

# We force this order later so label 0 = whole and label 1 = damage.
DESIRED_CLASS_NAMES = ["01-whole", "00-damage"]

print("Keras:", keras.__version__, "| backend:", keras.config.backend())
print("Input shape:", INPUT_SHAPE)
```

```
Keras: 3.15.0 | backend: torch
Input shape: (96, 128, 3)
```

Topic 1: Images as Tensors

Problem 1.1: Pixel Encoding and Rescaling

An RGB pixel from one inspection photo has channel values written in hexadecimal as:

```
(0xB4, 0x98, 0x7E)
```

Convert it to decimal RGB values. Then divide each channel by 255, as a `Rescaling(1./255)` layer would do.

> **Concept**
>
> Each colour channel is stored as an integer from 0 to 255. Hexadecimal is just a compact way to write the same integer. CNNs usually train on rescaled channel values because inputs around `[0, 1]` are numerically easier for gradient descent than raw values around `[0, 255]`.

```
hex_pixel = ("B4", "98", "7E")

# TODO: convert each hex string to a decimal integer (hint: int(channel, 16))
# TODO: build rgb_decimal as a numpy array, then rgb_rescaled = rgb_decimal / 255
# TODO: print the decimal RGB and the rescaled RGB (rounded to 3 dp)
```

Problem 1.2: Image and Batch Shapes

All images are resized to height `96`, width `128`, and `3` colour channels. A mini-batch contains `32` photos.

1. What is the shape of one image?
2. What is the shape of one mini-batch?
3. How many numeric channel values describe one image?
4. If you predict on one RGB image, what shape should be passed to Keras?

> **Concept**
>
> Keras `Conv2D` layers expect rank-4 input: `(batch, height, width, channels)`. A single image still needs a batch axis, so a one-image prediction is a batch of length one.

```
# TODO: write expressions for each of the four answers using IMG_SIZE / INPUT_SHAPE / BATCH_
# one_image_shape = ...
# batch_shape = ...
# n_values = ...                  # hint: np.prod(...)
# one_prediction_shape = ...      # hint: a leading axis of length 1
# print them all
```

Topic 2: A Local Convolution Filter

Problem 2.1: Compute One Filter Response

A small greyscale crop from a bumper photo is:

```
[[35, 42, 48],
 [40, 210, 52],
 [38, 45, 50]]
```

Use this local contrast filter with bias `0`:

```
[[-1, -1, -1],
 [-1,  8, -1],
 [-1, -1, -1]]
```

Compute the convolution output at the centre of the patch. Then apply ReLU.

> **Concept**
>
> A convolution neuron takes a local patch, multiplies each pixel by the corresponding filter weight, adds the products, and adds a bias. The same filter is then reused across the image. This reuse is called weight sharing.

```
patch = np.array([
    [35, 42, 48],
    [40, 210, 52],
    [38, 45, 50],
])

contrast_filter = np.array([
    [-1, -1, -1],
    [-1,  8, -1],
    [-1, -1, -1],
])

# TODO: raw_response = elementwise product of patch and filter, summed (hint: np.sum(patch *
# TODO: relu_response = max(raw_response, 0)
# TODO: print both
```

Problem 2.2: Compare With a Flat Patch

Repeat the same calculation for a flat patch where all nine values are `60`. What does this filter respond to? What kinds of damage cues would it pick up, and what could fool it?

> **Concept**
>
> The same filter slides across the whole image (weight sharing). A filter that sums to zero responds to *local contrast* rather than overall brightness.

```
# TODO: build a 3x3 patch of all 60s (hint: np.full((3, 3), 60))
# TODO: compute and print its response with contrast_filter
# Then answer in a sentence: what does this filter respond to, and what could fool it?
```

Topic 3: CNN Shapes and Parameters

Problem 3.1: Trace a Scratch CNN

Consider this scratch CNN for `96x128` RGB photos:

```
Input((96, 128, 3))
Rescaling(1./255)
Conv2D(12, 3, padding="same", activation="relu")
MaxPooling2D()
Conv2D(24, 3, padding="same", activation="relu")
MaxPooling2D()
Conv2D(48, 3, padding="same", activation="relu")
MaxPooling2D()
Flatten()
Dense(32, activation="relu")
Dense(1)
```

Fill in the output shape and trainable parameter count after every trainable layer. Which single layer holds most of the parameters, and why?

> **Concept**
>
> For a convolution layer:
>
> ```
> parameters = (kernel_height * kernel_width * input_channels + bias) * filters
> ```
>
> For a dense layer:
>
> ```
> parameters = (number_of_inputs + bias) * number_of_outputs
> ```
>
> With `padding="same"` and stride 1, convolution keeps height and width unchanged. The default `MaxPooling2D()` halves each spatial dimension.

```python
# The two formulas from the Concept box are provided as tools - use them to check your hand
def conv_params(kernel_size, in_channels, filters):
    return (kernel_size * kernel_size * in_channels + 1) * filters


def dense_params(n_in, n_out):
    return (n_in + 1) * n_out
```

```
# TODO: fill in (output_shape, params) for each row by hand, then verify with the helpers ab
layer_table = [
    ("Input",                (96, 128, 3), 0),
    ("Rescaling",            (96, 128, 3), 0),
    ("Conv2D(12, 3, same)",  None,         None),
    ("MaxPooling2D",         None,         0),
    ("Conv2D(24, 3, same)",  None,         None),
    ("MaxPooling2D",         None,         0),
    ("Conv2D(48, 3, same)",  None,         None),
    ("MaxPooling2D",         None,         0),
    ("Flatten",              None,         0),
    ("Dense(32)",            (32,),        None),
    ("Dense(1)",             (1,),         None),
]

# TODO: print the table and the total parameter count.
```

Problem 3.2: Replace Flatten With Global Average Pooling

Replace `Flatten()` by `GlobalAveragePooling2D()`, keeping `Dense(32) -> Dense(1)`. Recompute the trainable parameter count. Why does this reduce overfitting risk on a small image dataset?

> **Concept**
>
> `GlobalAveragePooling2D()` averages each feature map across height and width. A tensor of shape `(12, 16, 48)` becomes a vector of length `48`, one summary per channel.

```
# TODO: keep the three conv layers, but feed Dense(32) from 48 channel means instead of 9,21
# gap_total = conv_params(...) + conv_params(...) + conv_params(...) + dense_params(48, 32)
# TODO: print gap_total and compare it with the Flatten total from Problem 3.1.
```

Problem 3.3: Build the Scratch Model in Keras

Implement the architecture above as a Keras functional model, so you can check your hand calculations against `model.summary()`. The same builder is reused in later topics, so support optional augmentation, global average pooling, and dropout via arguments.

> **Concept**
>
> The final output has one unit and **no sigmoid**, because the model will be trained with `BinaryCrossentropy(from_logits=True)`.

```
def build_scratch_cnn(input_shape=INPUT_SHAPE, use_gap=False, use_augmentation=False, dropout
    inputs = keras.Input(shape=input_shape)
    x = inputs

    # TODO (optional, used later): if use_augmentation, add a small augmentation block
    #   (RandomFlip("horizontal"), RandomRotation(0.04), RandomZoom(0.08)).

    # TODO: Rescaling(1/255)
    # TODO: Conv2D(12, 3, padding="same", activation="relu") then MaxPooling2D()
    # TODO: Conv2D(24, 3, padding="same", activation="relu") then MaxPooling2D()
    # TODO: Conv2D(48, 3, padding="same", activation="relu") then MaxPooling2D()
    # TODO: if use_gap -> GlobalAveragePooling2D(), else -> Flatten()
    # TODO: if dropout > 0 -> Dropout(dropout)
    # TODO: Dense(32, activation="relu")
    # TODO: outputs = Dense(1)  (a single logit, no activation)

    outputs = x  # replace with your Dense(1) output
    return keras.Model(inputs, outputs, name="scratch_vehicle_cnn")


# scratch_shape_model = build_scratch_cnn(use_gap=False)
# scratch_shape_model.summary()   # compare the Param # column against Problems 3.1 and 3.2
```

Topic 4: Binary Logits and Evaluation

Problem 4.1: Convert Logits to Probabilities

The final `Dense(1)` returns one logit `z`. A positive logit favours `damage`; a negative logit favours `whole`.

The probability of damage is:

```
sigmoid(z) = 1 / (1 + exp(-z))
```

Convert `z = 1.2` and `z = -0.8` to probabilities. Which class does each predict? If accuracy is computed directly on logits, what is the decision threshold — `0.0` or `0.5`?

> **Concept**
>
> `sigmoid(0) = 0.5`. So a probability threshold of `0.5` corresponds to a logit threshold of `0.0`.

```
logits = np.array([1.2, -0.8])

# TODO: prob_damage = 1 / (1 + np.exp(-logits))
# TODO: for each logit, decide the predicted class (damage if z >= 0 else whole) and print z
```

Problem 4.2: Compile a Binary-Logit Model

Write a helper that compiles any of these single-logit models with the right loss and metrics: `BinaryCrossentropy(from_logits=True)`, an accuracy that thresholds the logit at `0.0`, and `AUC(from_logits=True)`.

> **Concept**
>
> `BinaryCrossentropy(from_logits=True)` tells Keras to apply the sigmoid inside the loss in a numerically stable way. For accuracy computed directly on logits, the threshold is `0.0`.
> Some Keras versions require `BinaryAccuracy.threshold` to be strictly between `0` and `1`, so it is cleaner to define a small custom logit-threshold metric.

```
def binary_logit_accuracy(y_true, y_pred):
    # TODO: cast y_true to float32
    # TODO: predicted class = 1.0 where y_pred (a logit) >= 0.0, else 0.0
    # TODO: return 1.0 where predicted class == y_true, else 0.0 (use keras.ops)
    raise NotImplementedError


def compile_binary_logit_model(model, lr=1e-3):
    # TODO: model.compile(
    #     optimizer = Adam(learning_rate=lr),
    #     loss      = BinaryCrossentropy(from_logits=True),
    #     metrics   = [binary_logit_accuracy, AUC(from_logits=True, name="auc")],
    # )
    return model
```

```
# demo_model = build_scratch_cnn(use_gap=True)
# compile_binary_logit_model(demo_model, lr=1e-3)
```

Topic 5: Real Kaggle Data

The remaining topics give you a working data pipeline so you can focus on understanding and running it. Read each cell, run it, and answer the questions in the markdown.

Problem 5.1: Download or Locate the Dataset

Use KaggleHub to download the dataset if it is not already present.

> **Concept**
>
> Kaggle datasets usually do not have a stable unauthenticated direct ZIP URL. KaggleHub is the clean Python interface:
>
> ```
> import kagglehub
> path = kagglehub.dataset_download("anujms/car-damage-detection")
> ```
>
> Here we can add `output_dir` so the files stay inside the local folder.

```
DATASET_HANDLE = "anujms/car-damage-detection"

# The dataset has been downloaded and unzipped into ./data/data1a.
DATA_DIR = Path("data") / "data1a"
TRAIN_DIR = DATA_DIR / "training"
VAL_DIR = DATA_DIR / "validation"

# If you do not have the data yet, download it from Kaggle with KaggleHub:
#     import kagglehub
#     kagglehub.dataset_download(DATASET_HANDLE)
# then move the data1a folder to ./data/data1a.

print("DATA_DIR:", DATA_DIR)
```

Problem 5.2: Count the Images

Run the cell and write down the class balance. Is the dataset balanced? How would your choice of evaluation metric change if it were heavily imbalanced?

> **Concept**
>
> Before modelling, check the dataset layout and class balance. A balanced dataset makes accuracy easier to interpret; an imbalanced dataset requires more care with AUC, recall, precision, and class weights.

```
IMAGE_EXTENSIONS = {".jpg", ".jpeg", ".png", ".bmp", ".gif"}


def count_images(folder):
    return sum(1 for p in folder.rglob("*") if p.suffix.lower() in IMAGE_EXTENSIONS)


for split_dir in [TRAIN_DIR, VAL_DIR]:
    print(f"\n{split_dir.name}")
    for class_dir in sorted(p for p in split_dir.iterdir() if p.is_dir()):
        print(f"  {class_dir.name:10s} {count_images(class_dir):5d} images")
```

Problem 5.3: Load and Inspect Batches

Load the images straight into NumPy arrays with a small helper. We pass the class order `["01-whole", "00-damage"]` so that the label is the *position* in that list:

```
whole  -> 0
damage -> 1
```

Question: what label would `damage` get if you instead ordered the class folders *alphabetically* (`00-damage`, `01-whole`)? Why might that be confusing given the usual positive-class convention?

> **Concept**
>
> We load each image with Pillow, resize it to `IMG_SIZE`, and stack the results into an array of shape `(n, height, width, 3)`. The label is the index of the class in the list we pass in, so choosing the order explicitly keeps `damage` as the positive class (label 1). We keep the raw `0-255` pixel values because the model's `Rescaling(1./255)` layer does the scaling.

```
def load_images_from_directory(directory, class_names, img_size=IMG_SIZE):
    """Load every image under directory/<class>/ into NumPy arrays.
```

```
    Returns X of shape (n, height, width, 3) as uint8, and y of shape (n, 1)
    as float32, where each label is the index of its class in ``class_names``
    (so whole -> 0 and damage -> 1).
    """
    directory = Path(directory)
    height, width = img_size
    paths, labels = [], []
    for index, class_name in enumerate(class_names):
        for path in sorted((directory / class_name).iterdir()):
            if path.suffix.lower() in IMAGE_EXTENSIONS:
                paths.append(path)
                labels.append(index)

    X = np.empty((len(paths), height, width, 3), dtype=np.uint8)
    for i, path in enumerate(paths):
        img = Image.open(path).convert("RGB").resize((width, height), Image.BILINEAR)
        X[i] = np.asarray(img, dtype=np.uint8)
    y = np.array(labels, dtype=np.float32).reshape(-1, 1)
    return X, y


class_names = DESIRED_CLASS_NAMES
X_train, y_train = load_images_from_directory(TRAIN_DIR, class_names)
X_val, y_val = load_images_from_directory(VAL_DIR, class_names)

# Shuffle once so batches (and the small samples below) mix the two classes.
rng = np.random.default_rng(SEED)
train_perm = rng.permutation(len(X_train))
X_train, y_train = X_train[train_perm], y_train[train_perm]
val_perm = rng.permutation(len(X_val))
X_val, y_val = X_val[val_perm], y_val[val_perm]

print("class_names:", class_names)
print("X_train:", X_train.shape, "y_train:", y_train.shape)
print("X_val:", X_val.shape, "y_val:", y_val.shape)
```

```
plt.figure(figsize=(9, 5))
for i in range(8):
    ax = plt.subplot(2, 4, i + 1)
    ax.imshow(X_train[i])
    label_index = int(y_train[i, 0])
```

```
    ax.set_title(class_names[label_index])
    ax.axis("off")
plt.tight_layout()
```

Topic 6: Scratch CNN on Real Images

Problem 6.1: Train a Small Sanity-Check Model

Train the global-average-pooling version on a few batches only. This is a **pipeline check**, not a final performance estimate. (Requires your completed `build_scratch_cnn` and `compile_binary_logit_model` from Topics 3–4.)

> Concept
>
> A short sanity run answers: "Does the data pipeline, model, loss, and metric setup work?" It does not answer: "What is the best achievable validation AUC?"

```
# Use a few batches' worth of images as a quick pipeline check.
X_train_small, y_train_small = X_train[:6 * BATCH_SIZE], y_train[:6 * BATCH_SIZE]
X_val_small, y_val_small = X_val[:3 * BATCH_SIZE], y_val[:3 * BATCH_SIZE]

sanity_model = build_scratch_cnn(use_gap=True, use_augmentation=True, dropout=0.25)
compile_binary_logit_model(sanity_model, lr=5e-4)

sanity_history = sanity_model.fit(
    X_train_small,
    y_train_small,
    validation_data=(X_val_small, y_val_small),
    batch_size=BATCH_SIZE,
    epochs=2,
    verbose=2,
)
```

Problem 6.2: Confusion Matrix From Logits

Collect logits over the validation sample, threshold at `0.0`, and build a confusion matrix with precision and recall for the `damage` class. Why do we use logits for the loss but convert to probabilities for interpretation?

Concept

Use logits for the loss, but convert to probabilities for interpretation. A logit threshold of `0.0` is equivalent to a probability threshold of `0.5`.

```
def collect_logits_and_labels(model, X, y):
    logits = model.predict(X, verbose=0).reshape(-1)
    labels = y.reshape(-1).astype(int)
    return logits, labels


def binary_report_from_logits(logits, labels):
    probs = 1 / (1 + np.exp(-logits))
    preds = (logits >= 0).astype(int)
    cm = np.zeros((2, 2), dtype=int)
    for y_true, y_pred in zip(labels, preds):
        cm[y_true, y_pred] += 1

    accuracy_value = np.mean(preds == labels)
    precision_damage = cm[1, 1] / max(cm[0, 1] + cm[1, 1], 1)
    recall_damage = cm[1, 1] / max(cm[1, 0] + cm[1, 1], 1)

    print("confusion matrix rows=true [whole, damage], columns=predicted [whole, damage]")
    print(cm)
    print(f"accuracy: {accuracy_value:.3f}")
    print(f"damage precision: {precision_damage:.3f}")
    print(f"damage recall: {recall_damage:.3f}")
    print("first five probabilities:", np.round(probs[:5], 3).tolist())
    return probs, preds, cm


sanity_logits, sanity_labels = collect_logits_and_labels(sanity_model, X_val_small, y_val_sma
sanity_probs, sanity_preds, sanity_cm = binary_report_from_logits(sanity_logits, sanity_label
```

Problem 6.3: What Mistakes Should We Expect?

Before looking at the wrong-prediction plot below, **predict** what kinds of images this model is likely to misclassify, and write down at least four. Then run the cell and compare your list against what you actually see.

```
logits = sanity_model.predict(X_val_small, verbose=0).reshape(-1)
preds = (logits >= 0).astype(int)
labels = y_val_small.reshape(-1).astype(int)

shown = 0
plt.figure(figsize=(9, 5))
for image, y_true, y_pred, z in zip(X_val_small, labels, preds, logits):
    if y_true != y_pred and shown < 8:
        ax = plt.subplot(2, 4, shown + 1)
        ax.imshow(image)
        ax.set_title(f"true={class_names[y_true]}\npred={class_names[y_pred]}\nz={z:.2f}")
        ax.axis("off")
        shown += 1
    if shown >= 8:
        break
plt.tight_layout()
if shown == 0:
    print("No wrong predictions found in this small validation sample.")
```

Topic 7: Stretch Refinements and Transfer Learning

Problem 7.1: Which Refinement Changes the Parameter Count Most?

Of the common refinements — data augmentation, dropout, and replacing `Flatten()` with `GlobalAveragePooling2D()` — which one changes the **trainable parameter count** the most, and which mainly help generalisation without adding parameters? Justify your answer using the counts from Topic 3, then build the refined model and confirm with `summary()`.

> **Concept**
>
> Augmentation and dropout act on activations, not weights, so they add (almost) no trainable parameters. Structural changes to the head are what move the parameter count.

```
refined_model = build_scratch_cnn(use_gap=True, use_augmentation=True, dropout=0.25)
compile_binary_logit_model(refined_model, lr=5e-4)
refined_model.summary()
```

Problem 7.2: Full Training Code

The following is the full training version. Run it when you have time and the dataset available. Compare its validation AUC/loss with the sanity run.

```
callbacks = [
    keras.callbacks.EarlyStopping(
        monitor="val_loss",
        patience=3,
        restore_best_weights=True,
    )
]

history = refined_model.fit(
    X_train,
    y_train,
    validation_data=(X_val, y_val),
    batch_size=BATCH_SIZE,
    epochs=12,
    callbacks=callbacks,
)
```

Problem 7.3: Optional Transfer Learning With MobileNetV2

Build a transfer-learning model on a frozen MobileNetV2 base with a small binary head. **Question:** why is transfer learning often more reliable than training from scratch on a small specialised dataset, and why should fine-tuning use a very small learning rate (e.g. `1e-5`) on only the last block?

> Concept
>
> Transfer learning reuses a CNN trained on a large general image dataset. The early and middle layers already know visual features such as edges, textures, corners, and object parts. On a small specialised dataset, training only a small head is often more reliable than training every convolutional weight from scratch.

```
def build_mobilenet_transfer_model(input_shape=INPUT_SHAPE, dropout=0.25):
    base = keras.applications.MobileNetV2(
        input_shape=input_shape,
        include_top=False,
        weights="imagenet",
```

```
    )
    base.trainable = False

    inputs = keras.Input(shape=input_shape)
    x = keras.Sequential(
        [
            layers.RandomFlip("horizontal"),
            layers.RandomRotation(0.04),
            layers.RandomZoom(0.08),
        ],
        name="transfer_augmentation",
    )(inputs)
    x = keras.applications.mobilenet_v2.preprocess_input(x)
    x = base(x, training=False)
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dropout(dropout)(x)
    outputs = layers.Dense(1, name="damage_logit")(x)
    model = keras.Model(inputs, outputs, name="mobilenetv2_vehicle_damage")
    return model, base


transfer_model, transfer_base = build_mobilenet_transfer_model()
compile_binary_logit_model(transfer_model, lr=1e-3)

transfer_history = transfer_model.fit(
    X_train,
    y_train,
    validation_data=(X_val, y_val),
    batch_size=BATCH_SIZE,
    epochs=8,
    callbacks=callbacks,
)

# Optional fine-tuning.
transfer_base.trainable = True
for layer in transfer_base.layers[:-20]:
    layer.trainable = False

compile_binary_logit_model(transfer_model, lr=1e-5)

fine_tune_history = transfer_model.fit(
    X_train,
```

```
    y_train,
    validation_data=(X_val, y_val),
    batch_size=BATCH_SIZE,
    epochs=5,
    callbacks=callbacks,
)
```

Closing Checklist

By the end of this lab, you should be able to:

1. Read RGB image tensors and Keras batch shapes.
2. Compute one convolution response by hand.
3. Trace CNN output shapes and parameter counts.
4. Explain why `Flatten()` can create many parameters.
5. Interpret binary logits, sigmoid probabilities, and thresholding.
6. Load a real image-folder dataset with explicit class ordering.
7. Train and evaluate a scratch CNN pipeline.
8. Explain why data augmentation, dropout, global average pooling, and transfer learning can matter.