

# Intermediate Machine Learning: Assignment 2

## Deadline

Assignment 2 is due Thursday, October 9 by 11:59 pm. Late work will not be accepted as per the course policies (see the syllabus on Canvas).

Directly sharing answers is not okay, but discussing problems with the course staff or with other students is encouraged. Acknowledge any use of an AI system such as ChatGPT or Copilot.

You should start early so that you have time to get help if you're stuck. The drop-in office hours schedule can be found on Canvas. You can also post questions or start discussions on Ed Discussion. The assignment may look long at first glance, but the problems are broken up into steps that should help you to make steady progress.

## Submission

Submit your assignment as a .pdf on Gradescope. To convert your notebook to a .pdf while preserving the cell structure without truncating the output, you can convert to .html using [this notebook](#). Note: When submitting on Gradescope, please select the correct pages of your pdf that correspond to each problem. This will allow graders to more easily find your complete solution to each problem.

## Topics

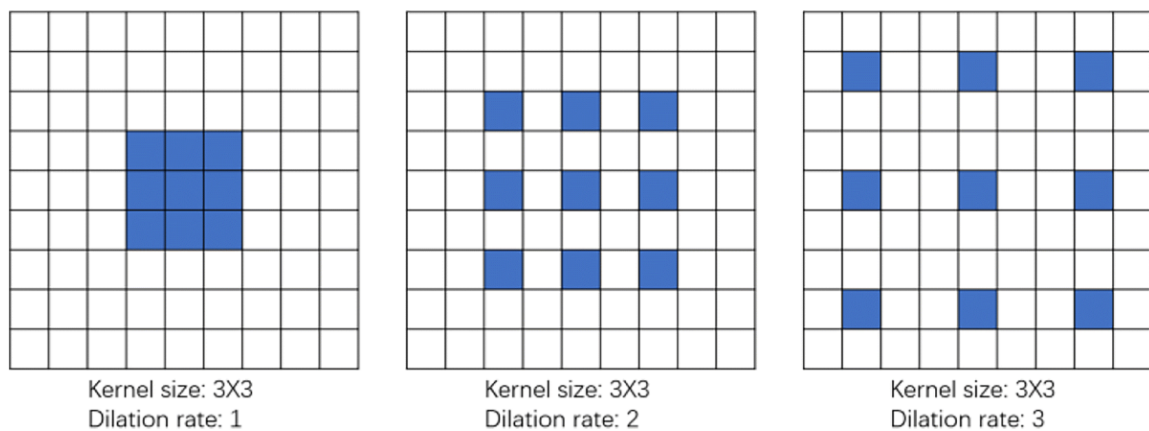
- Convolutional neural nets
- Gaussian processes
- Double descent

Please note that while the problems may look long, we give you substantial starter code to make the work manageable. The course staff members are available to help during office hours!

## Problem 1: Dilation elation (35 points)

### Overview

A dilated convolution (also known as atrous convolution) inserts gaps between kernel entries, so that a  $k \times k$  filter "jumps" by a dilation rate  $d$ . A dilation rate of  $d = 1$  is the regular convolution. CNN dilation can be adjusted using parameter  $d$  as shown in the figure below.



Dilated convolution captures a wider area of the input, allowing for potentially better context modeling without increased computational cost. It expands the effective "receptive field" (the area of the filter) without the need for larger kernels or more model parameters; dilated convolution can also reduce blurring. One of the most successful uses of dilated convolution was in the first AlphaFold paper, one of the earliest CNN-based attempts to predict the structure of proteins. For a glimpse of the full architecture and its details you may wish to take a look at <https://www.nature.com/articles/s41586-019-1923-7>

In this problem, we use dilated convolution to analyze MRI images to detect the presence of certain types of brain tumors. Cues about tumor type can include texture information (enhanced rim, necrotic core) and global characteristics (edema extent, mass effect, midline shift). Dilation expands the effective receptive field while keeping high-resolution features available for downstream neural network layers that make the multi-class decision. This approach is also helps address the problem of "anisotropy" where some characteristics are directionally dependent, by trying to incorporate a larger sub image in the kernel.

In the first part of this problem, we implement a regular CNN. Then, your job will be to implement a dilated CNN, and to choose the dilation parameters so that the model is as accurate as you can make it. Finally, we use Monte Carlo dropout as a way to estimate the uncertainty in the predictions.

## Dilated convolution in one and two dimensions

### Symbols (what each letter means):

- $x$ : the **input** (a 1D signal for the first formula; a 2D image for the second).
- $y$ : the **output feature map** after convolution (same shape type as  $x$ ).
- $w \in \mathbb{R}^k$ : a 1D **kernel/filter** of length  $k$ .
- $W \in \mathbb{R}^{k \times k}$ : a 2D **kernel/filter** of size  $k \times k$ .
- $k$ : **kernel size** (e.g.,  $k = 3$  for a  $3 \times 3$  kernel).
- $d$ : **dilation rate** (how far apart the kernel taps are;  $d = 1$  is a standard convolution).

- $i, j$ : spatial **indices** of the output  $y$  (row/column).
- $m, u, v$ : **indices inside the kernel** (1D:  $m$ , 2D:  $u$  for rows,  $v$  for cols).
- $u_c = \lfloor k/2 \rfloor, v_c = \lfloor k/2 \rfloor$ : the **kernel center** offsets (so the kernel is centered).
- We assume **stride** = 1 and appropriate padding so indices are valid.

**1D dilated convolution** of a signal  $x$  with kernel  $w \in \mathbb{R}^k$  and dilation  $d \in \mathbb{N}$ :

$$y[i] = \sum_{m=0}^{k-1} w[m] x(i + d m).$$

**2D dilated convolution** of an image  $x$  with kernel  $W \in \mathbb{R}^{k \times k}$  and dilation  $d$  (stride 1).

Let  $u_c = \lfloor k/2 \rfloor$  and  $v_c = \lfloor k/2 \rfloor$  denote the kernel center:

$$y[i, j] = \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} W[u, v] x(i + d(u - u_c), j + d(v - v_c)).$$

**Effective receptive field** of a  $k \times k$  kernel with dilation  $d$ :

$$k_{\text{eff}} = k + (k - 1)(d - 1).$$

For example, for  $k = 3$  and  $d = 3$ , we get  $k_{\text{eff}} = 7$ .

(When  $d = 1$ , this reduces to the standard convolution.)

## Loading the MRI data

The data used in this problem is cropped from this Kaggle repository <https://www.kaggle.com/datasets/masoudnickparvar/brain-tumor-mri-dataset>.

We only use the data corresponding to glioma tumors and non-tumor images. You can access the data in a zip file at <https://github.com/YData123/sds365-fa25/tree/main/assignments/assn2/>. You will need to upload the data to your Google Drive.

The cell below gives Colab access to your Google Drive.

```
In [ ]: # Please do not edit this cell.
# Imports
import os, random, time, math, keras, json
from glob import glob
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras import layers as L, models as M
```

```
# (Colab) Mount Google Drive
from google.colab import drive
if not os.path.ismount("/content/drive"):
    drive.mount("/content/drive")
```

```
In [ ]: !nvidia-smi
```

```
/bin/bash: line 1: nvidia-smi: command not found
```

```
In [ ]: import torch
if torch.cuda.is_available():
    print("GPU is available for PyTorch.")
else:
    print("GPU not available for PyTorch.")
```

GPU not available for PyTorch.

Now that we mounted Google Drive let's try visualizing images in the training data for the tumor and non-tumor classes.

```
In [ ]: # Do not edit this block.
# Point to the MRI/ folder that contains Training/ and Testing/
ROOT = "/content/drive/MyDrive/Assn2/kaggle-MRI"

TRAIN_DIR = os.path.join(ROOT, "Training")
TEST_DIR = os.path.join(ROOT, "Testing")
CLASS_NAMES = ["glioma", "notumor"] # expected two folders inside both train

# Validation of directory structure
def assert_dir(p):
    if not os.path.isdir(p):
        raise FileNotFoundError(f"Expected directory not found: {p}")

assert_dir(ROOT)
assert_dir(TRAIN_DIR)
assert_dir(TEST_DIR)

# Load image as grayscale for consistent viewing
def load_img_gray(path):
    img = Image.open(path).convert("L") # force grayscale
    return img

# List image files in a folder
def count_images(folder):
    exts = ("*.jpg", "*.jpeg")
    files = []
    for e in exts:
        files.extend(glob(os.path.join(folder, e)))
    return sorted(files)
```

The data we have requires pre-processing in order to be fed to any neural networks. The inconsistencies in the size as you see in the above output becomes problematic during feeding the data to the network. In this block, we strive to solve that issue.

```

In [ ]: # Helper functions: letterbox pad to square (no cropping) + resize to 160x160

IMG_SIZE = 160 # target size used throughout the assignment

def letterbox_to_square(img: Image.Image, fill=0) -> Image.Image:
    """
    Pads the image to a square using 'fill' (black), centered, without distortion.
    Keeps original content intact (no cropping).
    """
    if img.mode not in ("L", "RGB"):
        img = img.convert("L")
    w, h = img.size
    s = max(w, h)
    canvas = Image.new(img.mode, (s, s), color=fill)
    canvas.paste(img, ((s - w)//2, (s - h)//2))
    return canvas

def preprocess_image(img: Image.Image, size: int = IMG_SIZE) -> Image.Image:
    """Letterbox pad -> resize to (size, size)."""
    sq = letterbox_to_square(img, fill=0)
    return sq.resize((size, size), resample=Image.BILINEAR)

# Visualize ORIGINAL vs PREPROCESSED (letterboxed) for first training image
fig, axes = plt.subplots(len(CLASS_NAMES), 2, figsize=(6, 3*len(CLASS_NAMES)))
if len(CLASS_NAMES) == 1:
    axes = np.array([axes]) # ensure 2D array indexing

for i, c in enumerate(CLASS_NAMES):
    class_dir = os.path.join(TRAIN_DIR, c)
    files = count_images(class_dir)
    ax_orig, ax_proc = axes[i, 0], axes[i, 1]
    if not files:
        ax_orig.axis("off"); ax_proc.axis("off")
        ax_proc.set_title(f"No images found for {c}")
        continue

    p = files[0]
    img = load_img_gray(p) # original grayscale
    proc = preprocess_image(img) # padded + resized 160x160

    ax_orig.imshow(img, cmap="gray"); ax_orig.axis("off")
    ax_orig.set_title(f"{c} (original {img.size[0]}x{img.size[1]})", fontsize=12)

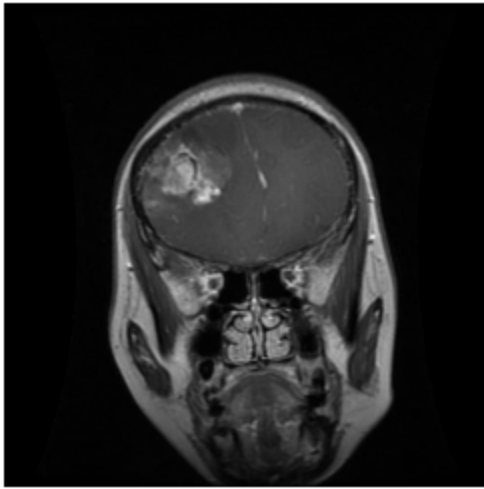
    ax_proc.imshow(proc, cmap="gray"); ax_proc.axis("off")
    ax_proc.set_title(f"{c} (letterbox -> {IMG_SIZE}x{IMG_SIZE})", fontsize=12)

plt.suptitle("Sanity check: letterbox padding avoids cropping", fontsize=12)
plt.tight_layout()
plt.show()

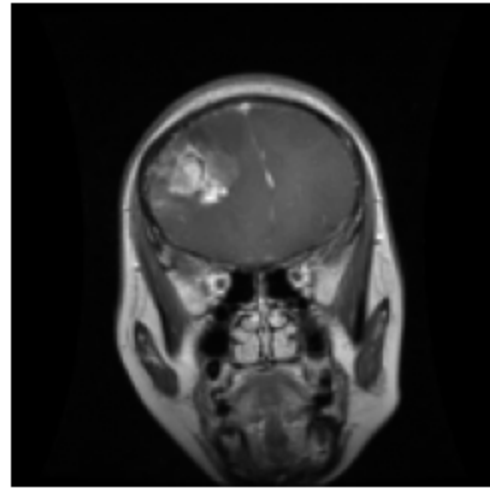
```

## Sanity check: letterbox padding avoids cropping

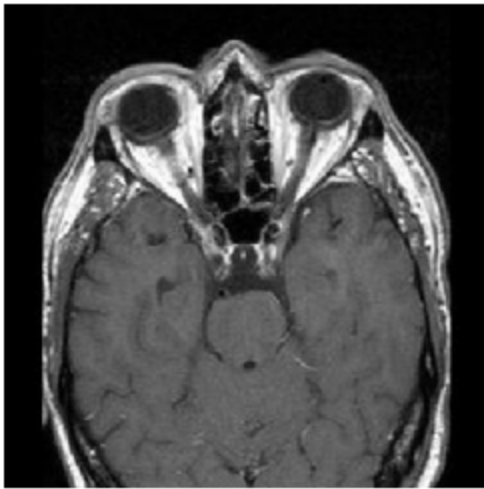
glioma (original 512×512)



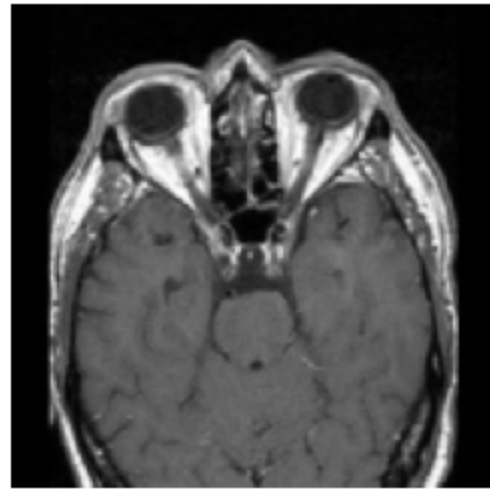
glioma (letterbox → 160×160)



notumor (original 350×350)



notumor (letterbox → 160×160)



In [ ]: # Gather file paths & labels; split TRAIN into train/validation (TEST untouched)

```
def list_images_with_labels(root_dir, classes):
    paths, labels = [], []
    for ci, cname in enumerate(classes):
        cdir = os.path.join(root_dir, cname)
        for p in count_images(cdir):
            paths.append(p); labels.append(ci)
    return np.array(paths), np.array(labels)

# load TRAIN and TEST file lists
train_paths, train_labels = list_images_with_labels(TRAIN_DIR, CLASS_NAMES)
test_paths, test_labels = list_images_with_labels(TEST_DIR, CLASS_NAMES)

# stratified split of TRAIN -> (train, val)
VAL_FRACTION = 0.20
tr_paths, val_paths, tr_labels, val_labels = train_test_split(
    train_paths, train_labels,
    test_size=VAL_FRACTION,
    random_state=42,
```

```

        stratify=train_labels
    )

    print(f"Counts -> train: {len(tr_paths)}, val: {len(val_paths)}, test: {len(
Counts -> train: 2332, val: 584, test: 705

```

```

In [ ]: # TensorFlow input pipeline with the SAME preprocessing:
# grayscale -> letterbox to square (no crop) -> resize to IMG_SIZE -> per-im

def tf_letterbox_resize(img, target=IMG_SIZE):
    shape = tf.shape(img)
    h, w = shape[0], shape[1]
    s = tf.maximum(h, w)
    pad_top = (s - h) // 2
    pad_bottom = s - h - pad_top
    pad_left = (s - w) // 2
    pad_right = s - w - pad_left
    img_sq = tf.pad(img, [[pad_top, pad_bottom], [pad_left, pad_right], [0, 0]])
    img_rs = tf.image.resize(img_sq, (target, target), method="bilinear")
    return img_rs

def tf_per_image_zscore(img):
    mean = tf.reduce_mean(img)
    std = tf.math.reduce_std(img)
    return (img - mean) / tf.maximum(std, 1e-6)

def load_and_preprocess(path, label, augment=False):
    img_bytes = tf.io.read_file(path)
    img = tf.io.decode_image(img_bytes, channels=1, expand_animations=False)
    img = tf.image.convert_image_dtype(img, tf.float32)
    img = tf_letterbox_resize(img, target=IMG_SIZE)

    if augment:
        img = tf.image.random_flip_left_right(img)
        img = tf.image.random_brightness(img, max_delta=0.10)
        img = tf.image.random_contrast(img, lower=0.9, upper=1.1)

    img = tf_per_image_zscore(img)
    label = tf.cast(label, tf.float32)
    label = tf.expand_dims(label, axis=-1) # (1,)
    return img, label

def make_dataset(paths, labels, batch_size=32, shuffle=False, augment=False):
    ds = tf.data.Dataset.from_tensor_slices((paths, labels))
    if shuffle:
        ds = ds.shuffle(buffer_size=len(paths), seed=42, reshuffle_each_iter
    ds = ds.map(lambda p, y: load_and_preprocess(p, y, augment=augment),
               num_parallel_calls=tf.data.AUTOTUNE)
    ds = ds.batch(batch_size).prefetch(tf.data.AUTOTUNE)
    return ds

BATCH_SIZE = 16
ds_train = make_dataset(tr_paths, tr_labels, batch_size=BATCH_SIZE, shuffle=True)
ds_val = make_dataset(val_paths, val_labels, batch_size=BATCH_SIZE, shuffle=False)

```



```
ds_test = make_dataset(test_paths, test_labels, batch_size=BATCH_SIZE, shuf

for xb, yb in ds_train.take(1):
    print("Train batch:", xb.shape, yb.shape)
```

Train batch: (16, 160, 160, 1) (16, 1)

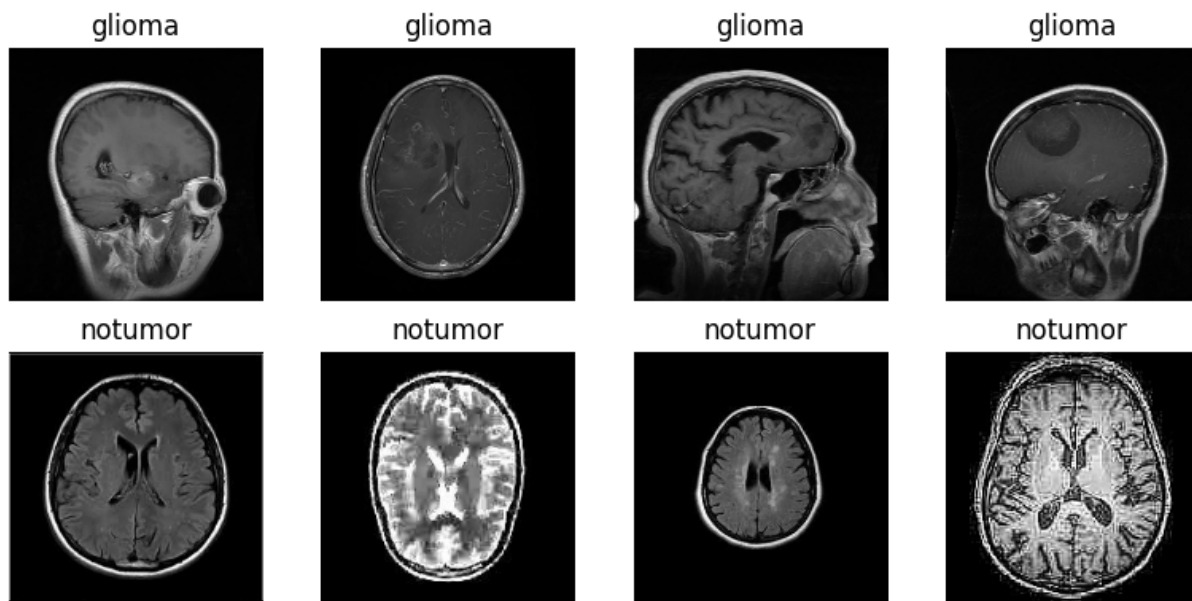
```
In [ ]: # Quick visual sanity check: Show 4 preprocessed samples per class from the
class_names = CLASS_NAMES

# collect a small batch
xb, yb = next(iter(ds_train))
xb = xb.numpy(); yb = yb.numpy()

fig, axes = plt.subplots(2, 4, figsize=(8,4))
axes = axes.reshape(-1)
shown = {i:0 for i in range(len(class_names))}
for i in range(len(xb)):
    c = int(yb[i])
    if shown[c] >= 4:
        continue
    axes[c*4 + shown[c]].imshow(xb[i,...,0], cmap="gray")
    axes[c*4 + shown[c]].axis("off")
    axes[c*4 + shown[c]].set_title(class_names[c])
    shown[c] += 1
    if all(v==4 for v in shown.values()):
        break
plt.tight_layout(); plt.show()
```

/tmp/ipython-input-1886849759.py:12: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)

```
c = int(yb[i])
```



```
In [ ]: """
You'll learn two new layers in this assignment. MCSpatialDropout2D is a smart
networks. Instead of randomly turning off individual pixels, it turns off en
```



once. This works better for images because it prevents the network from getting detectors working together, which helps it generalize better to new data. By internal signals stable during training. It takes the outputs from each layer (data), then applies two learnable parameters to scale and shift the results. This often lets you use higher learning rates, leading to faster and more reliable training.

```

# Keras model factory: Build a baseline CNN
# Force-dropout layers that stay active at inference for MC sampling (Batch Normalization)
@keras.saving.register_keras_serializable(package="mc")
class MCDropout(tf.keras.layers.Dropout):
    def call(self, inputs, training=None):
        # Force sampling masks even at inference for MC passes
        return super().call(inputs, training=True)

@keras.saving.register_keras_serializable(package="mc")
class MCSpatialDropout2D(tf.keras.layers.SpatialDropout2D):
    def call(self, inputs, training=None):
        # Force sampling masks even at inference for MC passes
        return super().call(inputs, training=True)

def build_cnn(
    dilated=False,
    input_shape=(IMG_SIZE, IMG_SIZE, 1),
    num_classes=1,  # 1 sigmoid unit (binary)
    drop_rate=0.20,
    use_mc_dropout=True,  # if True, use MC* layers; else standard Dropout
    use_spatial=True  # SpatialDropout2D is usually better for convs
):
    """
    Tiny CNN; toggle dilation in block-2 convs. Adds dropout so MC sampling
    - dilated=False: Conv3x3 (rate=1)
    - dilated=True : dilation rates (2,3,5) in the 2nd conv of each block
    - Dropout: after each block; optionally before Dense head
    """
    rates = (1,1,1)

    Drop = (MCSpatialDropout2D if (use_mc_dropout and use_spatial) else
            L.SpatialDropout2D if use_spatial else
            MCDropout if use_mc_dropout else
            L.Dropout)

    x_in = L.Input(shape=input_shape)
    x = x_in

    # Block A
    x = L.Conv2D(32, 3, padding="same", activation="relu")(x)
    x = L.BatchNormalization()(x)
    x = L.Conv2D(32, 3, padding="same", activation="relu", dilation_rate=rates[0])(x)
    x = L.BatchNormalization()(x)
    x = L.MaxPool2D(pool_size=2)(x)
    x = Drop(drop_rate)(x)  # <-- dropout for MC

    # Block B
    x = L.Conv2D(64, 3, padding="same", activation="relu")(x)
    x = L.BatchNormalization()(x)
    x = L.Conv2D(64, 3, padding="same", activation="relu", dilation_rate=rates[1])(x)

```

```
x = L.BatchNormalization()(x)
x = L.MaxPool2D(pool_size=2)(x)
x = Drop(drop_rate)(x) # <-- dropout for MC

# Block C
x = L.Conv2D(128, 3, padding="same", activation="relu")(x)
x = L.BatchNormalization()(x)
x = L.Conv2D(128, 3, padding="same", activation="relu", dilation_rate=ra
x = L.BatchNormalization()(x)
x = Drop(drop_rate)(x) # <-- dropout for MC

x = tf.keras.layers.GlobalAveragePooling2D()(x)
x = L.Dropout(drop_rate)(x) # a small dense-level dropout is fine (stan

out = tf.keras.layers.Dense(1, activation="sigmoid")(x) # binary head
model = tf.keras.Model(x_in, out)
return model

# Rebuild models (students then train as in Step 14)
model_base = build_cnn(dilated=False)
model_base.summary()
```

**Model: "functional"**

Layer (type)	Output Shape	Par
input_layer ( <a href="#">InputLayer</a> )	( <a href="#">None</a> , 160, 160, 1)	
conv2d ( <a href="#">Conv2D</a> )	( <a href="#">None</a> , 160, 160, 32)	
batch_normalization ( <a href="#">BatchNormalization</a> )	( <a href="#">None</a> , 160, 160, 32)	
conv2d_1 ( <a href="#">Conv2D</a> )	( <a href="#">None</a> , 160, 160, 32)	9
batch_normalization_1 ( <a href="#">BatchNormalization</a> )	( <a href="#">None</a> , 160, 160, 32)	
max_pooling2d ( <a href="#">MaxPooling2D</a> )	( <a href="#">None</a> , 80, 80, 32)	
mc_spatial_dropout2d ( <a href="#">MCSpatialDropout2D</a> )	( <a href="#">None</a> , 80, 80, 32)	
conv2d_2 ( <a href="#">Conv2D</a> )	( <a href="#">None</a> , 80, 80, 64)	18
batch_normalization_2 ( <a href="#">BatchNormalization</a> )	( <a href="#">None</a> , 80, 80, 64)	
conv2d_3 ( <a href="#">Conv2D</a> )	( <a href="#">None</a> , 80, 80, 64)	36
batch_normalization_3 ( <a href="#">BatchNormalization</a> )	( <a href="#">None</a> , 80, 80, 64)	
max_pooling2d_1 ( <a href="#">MaxPooling2D</a> )	( <a href="#">None</a> , 40, 40, 64)	
mc_spatial_dropout2d_1 ( <a href="#">MCSpatialDropout2D</a> )	( <a href="#">None</a> , 40, 40, 64)	
conv2d_4 ( <a href="#">Conv2D</a> )	( <a href="#">None</a> , 40, 40, 128)	73
batch_normalization_4 ( <a href="#">BatchNormalization</a> )	( <a href="#">None</a> , 40, 40, 128)	
conv2d_5 ( <a href="#">Conv2D</a> )	( <a href="#">None</a> , 40, 40, 128)	147
batch_normalization_5 ( <a href="#">BatchNormalization</a> )	( <a href="#">None</a> , 40, 40, 128)	
mc_spatial_dropout2d_2 ( <a href="#">MCSpatialDropout2D</a> )	( <a href="#">None</a> , 40, 40, 128)	
global_average_pooling2d ( <a href="#">GlobalAveragePooling2D</a> )	( <a href="#">None</a> , 128)	
dropout ( <a href="#">Dropout</a> )	( <a href="#">None</a> , 128)	
dense ( <a href="#">Dense</a> )	( <a href="#">None</a> , 1)	

Total params: 288,353 (1.10 MB)

**Trainable params:** 287,457 (1.10 MB)

**Non-trainable params:** 896 (3.50 KB)

## Problem 1.1: Explain the CNN

The cell above shows layer of the CNN, with the output tensor shape and the number of trainable parameters. For each layer, explain in a couple of sentences (1) what the purpose of the layer is (2) why the output shape and number of parameters are as shown.

input\_1 (InputLayer)

(1) Input

(2) fixable batch size, width, height=160, channel=1; no param for layer 1

conv2d (Conv2D)

(1) Convolution layer 1, learn 32 features

(2) output shape = (batch size, output\_h, output\_w, n\_filter), output\_h and \_w param

calculation:  $\frac{160+2*2-3}{1} + 1 = 160$ ; total params calculation:  $(3 * 3 + 1) * 32 = 320$

batch\_normalization

(1) scaling all parameters to make a more stable training.

(2) no shape change, just normalization with two trainable param  $\gamma$  and  $\beta$ , two no-trainable  $mean$ ,  $Var$ ; total params calculation:

$32 * (train + test + mean + var) = 32 * 4 = 128$

conv2d\_1 (Conv2D)

(1) Convolution layer, learn 32 features.

(2) output shape = (batch size, output\_h, output\_w, n\_filter), output\_h and \_w param

calculation:  $\frac{160+2*2-3}{1} + 1 = 160$ ; total params calculation:

$(3 * 3 * 32 + 1) * 32 = 9248$

batch\_normalization\_1

(1) scaling all parameters to make a more stable training.

(2) no shape change, just normalization with two trainable param  $\gamma$  and  $\beta$ , two no-trainable  $mean$ ,  $Var$ ; total params calculation:

$32 * (train + test + mean + var) = 32 * 4 = 128$

max\_pooling2d

(1) find the maximum params in each  $2 * 2$  grid, and remember location to better learn, and also for backpropagate to relocate them.

(2) output shape = (batch size, output\_h, output\_w, n\_filter), output\_h and \_w param

calculation:  $\frac{160}{2} = 80$ , n\_filter = 32; No total params in pooling process.

mc\_spatial\_dropout2d

(1) randomly let some numbers be zero, and the probability is 20%.

(2)no shape change, no params in dropout layer.

conv2d\_2 (Conv2D)

(1)in block B, the filter size is 64, meaning learning and extracting 64 features.

(2)output shape = (batch size, output\_h, output\_w, n\_filter), output\_h and \_w param calculation:  $\frac{160+2*2-3}{1} + 1 = 160$ ; total params calculation:

$(3 * 3 * 32 + 1) * 64 = 18496$ , mention that the input filter is 32.

batch\_normalization\_2

(1)scaling all parameters to make a more stable training again, since the pooling and other Conv2D messed up the learned results.

(2)no shape change, just normalization with two trainable param  $\gamma$  and  $\beta$ , two no-trainable  $mean$ ,  $Var$ ; total params calculation:

$32 * (train + test + mean + var) = 64 * 4 = 256$

conv2d\_3 (Conv2D)

(1)convolution layer in block B.

(2)output shape = (batch size, output\_h, output\_w, n\_filter), output\_h and \_w param calculation:  $\frac{80+2*2-3}{1} + 1 = 80$ ; total params calculation:

$(3 * 3 * 64 + 1) * 64 = 36928$

batch\_normalization\_3

(1)scaling all parameters to make a more stable training again, since the pooling and other Conv2D messed up the learned results.

(2)no shape change, just normalization with two trainable param  $\gamma$  and  $\beta$ , two no-trainable  $mean$ ,  $Var$ ; total params calculation:

$64 * (train + test + mean + var) = 64 * 4 = 256$

max\_pooling2d\_1

(1)find the maximum params in each  $2 * 2$  grid, and remember location to better learn, and also for backpropagate to relocate them.

(2)output shape = (batch size, output\_h, output\_w, n\_filter), output\_h and \_w param calculation:  $\frac{80}{2} = 40$ ,  $n\_filter = 64$ ; No total params in pooling process.

mc\_spatial\_dropout2d\_1

(1)randomly let some numbers be zero, and the probability is 20%.

(2)no shape change, no params in dropout layer.

conv2d\_4 (Conv2D)

(1)convolution layer in block C, the filter size is 128, meaning learning and extracting 128 features.

(2)output shape = (batch size, output\_h, output\_w, n\_filter), output\_h and \_w param calculation:  $\frac{80+2*2-3}{1} + 1 = 80$ ; total params calculation:

$(3 * 3 * 64 + 1) * 128 = 73856$

## batch\_normalization\_4

(1)scaling all parameters to make a more stable training again, since the pooling and other Conv2D messed up the learned results.

(2)no shape change, just normalization with two trainable param  $\gamma$  and  $\beta$ , two no-trainable  $mean$ ,  $Var$ ; total params calculation:

$$128 * (train + test + mean + var) = 128 * 4 = 512$$

## conv2d\_5

(1)convolution layer in block C, the filter size is 128, meaning learning and extracting 128 features.

(2)output shape = (batch size, output\_h, output\_w, n\_filter), output\_h and \_w param calculation:  $\frac{80+2*2-3}{1} + 1 = 80$ ; total params calculation:

$$(3 * 3 * 128 + 1) * 128 = 147,584$$

## batch\_normalization\_5

(1)scaling all parameters to make a more stable training again, since the pooling and other Conv2D messed up the learned results.

(2)no shape change, just normalization with two trainable param  $\gamma$  and  $\beta$ , two no-trainable  $mean$ ,  $Var$ ; total params calculation:

$$128 * (train + test + mean + var) = 128 * 4 = 512$$

## mc\_spatial\_dropout2d\_2

(1)randomly let some numbers be zero, and the probability is 20%.

(2)no shape change, no params in dropout layer.

## global\_average\_pooling2d

(1)calculates the average of the feature plots for each channel, the out put dimension is very similar to flatten.

(2)no shape change, no params in dropout layer.

## dropout (Dropout)

(1)force model not rely heavily on any few features, making the model of network learning more robust and generalization. (2)no shape change, no params in dropout layer.

## dense (Dense)

(1)final output layer, means all features in a vector. (2)output shape = (batch size, dense=1); total params =  $(128 + 1) * 1 = 129$

The following cell now trains the baseline CNN, and evaluates it on test data.

```
In [ ]: # Train the baseline model (with validation) + evaluate on TEST + plots

EPOCHS = 10
OUTDIR = "/content/outputs_cnn_dilation"
os.makedirs(OUTDIR, exist_ok=True)
```

```

def compile_model(m):
    m.compile(optimizer=tf.keras.optimizers.Adam(3e-5),
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
              metrics=[tf.keras.metrics.BinaryAccuracy(name="accuracy", threshold=0.5)])
    return m

def train_model(model, name, train_ds, val_ds, epochs=EPOCHS):
    ckpt = os.path.join(OUTDIR, f"{name}.keras") # best checkpoint
    cbs = [ModelCheckpoint(ckpt, monitor="val_accuracy", save_best_only=True)]
    t0 = time.time()
    hist = model.fit(train_ds, validation_data=val_ds, epochs=epochs, callbacks=cbs)
    train_seconds = time.time() - t0
    print(f"[TIMER] {name}: training wall-clock = {train_seconds:.2f} sec "
          f"({train_seconds/max(1, len(hist.history['loss'])):.2f} sec/epoch)")
    # OPTIONAL: save the final weights separately (do NOT overwrite ckpt)
    final_path = os.path.join(OUTDIR, f"{name}_final.keras")
    model.save(final_path)
    return ckpt, hist, train_seconds

def evaluate_and_report(model, ds, class_names):
    y_true, y_pred = [], []
    out_last_dim = model.output_shape[-1] # 1 for sigmoid, 2 for softmax
    for xb, yb in ds:
        probs = model.predict(xb, verbose=0)
        probs = np.array(probs).reshape((probs.shape[0], -1))
        preds = (probs[:,0] >= 0.5).astype(int) if out_last_dim == 1 else np.argmax(probs, axis=-1)
        yt = yb.numpy().reshape(-1).astype(int)
        y_true.append(yt); y_pred.append(preds)
    y_true = np.concatenate(y_true); y_pred = np.concatenate(y_pred)
    print(classification_report(y_true, y_pred, target_names=class_names, digits=2))
    print("Confusion matrix:\n", confusion_matrix(y_true, y_pred))
    return y_true, y_pred

def measure_eval_time(model, ds, n_samples):
    t0 = time.time()
    loss, acc = model.evaluate(ds, verbose=0)
    eval_seconds = time.time() - t0
    print(f"[TIMER] evaluate: {eval_seconds:.2f} sec")
    t1 = time.time()
    _ = model.predict(ds, verbose=0)
    pred_seconds = time.time() - t1
    ips = n_samples / pred_seconds if pred_seconds > 0 else float("inf")
    print(f"[TIMER] predict : {pred_seconds:.2f} sec (~{ips:.1f} images/sec)")
    return loss, acc, eval_seconds, pred_seconds, ips

# --- compile
model_base = compile_model(model_base)

# --- train with validation (no test leakage)
ckpt_base, hist_base, train_time_base = train_model(model_base, "baseline_cn



















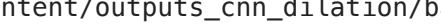
# --- reload best and evaluate on TEST (timed)
model_base = tf.keras.models.load_model(ckpt_base)

print("\n[TEST] Baseline:")

```



```
y_true_base, y_pred_base = evaluate_and_report(model_base, ds_test, CLASS_NAMES)
loss_b, acc_b, eval_t_b, pred_t_b, ips_b = measure_eval_time(model_base, ds_test)
print(f"\n[TEST] Baseline -> loss: {loss_b:.4f} | acc: {acc_b:.4f}")
```

Epoch 1/10  
146/146  0s 2s/step - accuracy: 0.5967 - loss: 0.6548  
Epoch 1: val\_accuracy improved from -inf to 0.45377, saving model to /content/outputs\_cnn\_dilation/baseline\_cnn.keras  
146/146  413s 3s/step - accuracy: 0.5971 - loss: 0.6546  
- val\_accuracy: 0.4538 - val\_loss: 1.3767  
Epoch 2/10  
146/146  0s 66ms/step - accuracy: 0.7282 - loss: 0.5551  
Epoch 2: val\_accuracy improved from 0.45377 to 0.46062, saving model to /content/outputs\_cnn\_dilation/baseline\_cnn.keras  
146/146  12s 78ms/step - accuracy: 0.7284 - loss: 0.5550  
- val\_accuracy: 0.4606 - val\_loss: 1.0715  
Epoch 3/10  
145/146  0s 67ms/step - accuracy: 0.7696 - loss: 0.5041  
Epoch 3: val\_accuracy improved from 0.46062 to 0.57363, saving model to /content/outputs\_cnn\_dilation/baseline\_cnn.keras  
146/146  12s 79ms/step - accuracy: 0.7696 - loss: 0.5038  
- val\_accuracy: 0.5736 - val\_loss: 0.7609  
Epoch 4/10  
145/146  0s 59ms/step - accuracy: 0.8186 - loss: 0.4351  
Epoch 4: val\_accuracy improved from 0.57363 to 0.77055, saving model to /content/outputs\_cnn\_dilation/baseline\_cnn.keras  
146/146  10s 71ms/step - accuracy: 0.8187 - loss: 0.4350  
- val\_accuracy: 0.7705 - val\_loss: 0.4670  
Epoch 5/10  
145/146  0s 49ms/step - accuracy: 0.8284 - loss: 0.4173  
Epoch 5: val\_accuracy improved from 0.77055 to 0.85274, saving model to /content/outputs\_cnn\_dilation/baseline\_cnn.keras  
146/146  10s 65ms/step - accuracy: 0.8285 - loss: 0.4170  
- val\_accuracy: 0.8527 - val\_loss: 0.3704  
Epoch 6/10  
145/146  0s 53ms/step - accuracy: 0.8521 - loss: 0.3629  
Epoch 6: val\_accuracy improved from 0.85274 to 0.86986, saving model to /content/outputs\_cnn\_dilation/baseline\_cnn.keras  
146/146  10s 66ms/step - accuracy: 0.8521 - loss: 0.3628  
- val\_accuracy: 0.8699 - val\_loss: 0.3267  
Epoch 7/10  
145/146  0s 58ms/step - accuracy: 0.8743 - loss: 0.3192  
Epoch 7: val\_accuracy did not improve from 0.86986  
146/146  10s 69ms/step - accuracy: 0.8742 - loss: 0.3194  
- val\_accuracy: 0.8408 - val\_loss: 0.3531  
Epoch 8/10  
145/146  0s 59ms/step - accuracy: 0.8722 - loss: 0.3048  
Epoch 8: val\_accuracy improved from 0.86986 to 0.88870, saving model to /content/outputs\_cnn\_dilation/baseline\_cnn.keras  
146/146  10s 71ms/step - accuracy: 0.8723 - loss: 0.3048  
- val\_accuracy: 0.8887 - val\_loss: 0.2894  
Epoch 9/10  
145/146  0s 50ms/step - accuracy: 0.8988 - loss: 0.2843  
Epoch 9: val\_accuracy did not improve from 0.88870  
146/146  9s 63ms/step - accuracy: 0.8988 - loss: 0.2842  
- val\_accuracy: 0.8836 - val\_loss: 0.2701  
Epoch 10/10  
145/146  0s 58ms/step - accuracy: 0.8872 - loss: 0.2859  
Epoch 10: val\_accuracy improved from 0.88870 to 0.90753, saving model to /content/outputs\_cnn\_dilation/baseline\_cnn.keras

146/146 ————— 10s 70ms/step – accuracy: 0.8873 – loss: 0.2857  
 – val\_accuracy: 0.9075 – val\_loss: 0.2405  
 [TIMER] baseline\_cnn: training wall-clock = 506.12 sec (50.61 sec/epoch) (monitoring val\_accuracy)

[TEST] Baseline:

	precision	recall	f1-score	support
glioma	0.8090	0.9600	0.8780	300
notumor	0.9656	0.8321	0.8939	405
accuracy			0.8865	705
macro avg	0.8873	0.8960	0.8860	705
weighted avg	0.8990	0.8865	0.8872	705

Confusion matrix:

```
[[288  12]
 [ 68 337]]
```

[TIMER] evaluate: 10.58 sec

[TIMER] predict : 8.01 sec (~88.0 images/sec)

[TEST] Baseline -> loss: 0.2814 | acc: 0.8879

## Problem 1.2: Improve on the baseline with a dilated CNN

Your job is now to use the above code, appropriately modified as needed, to build and train a dilated CNN that has better accuracy than the baseline.

Note that the keras Conv2D method can handle dilation. You may experiment with any architecture (combination of layers) and dilation parameters that you wish.

To keep the notebook clear, only show your best model, not intermediate, experimental models. Explain your choice of model, and why your best dilation CNN model does or does not improve on the baseline. Note that you should use a runtime environment that uses a GPU for faster processing speed.

```
In [ ]: def build_dilated_cnn(
    dilated=True, # use dilated strategy
    input_shape=(IMG_SIZE, IMG_SIZE, 1),
    num_classes=1,
    drop_rate=0.40,
    use_mc_dropout=True,
    use_spatial=True
):

    if dilated:
        rates = (2, 4, 8) # (2, 4, 8) is basic, also (1, 2, 4) or (4, 8, 16)
        print(f"Building DILATED CNN with rates: {rates}")
    else:
        rates = (1, 1, 1)
        print(f"Building BASELINE CNN with rates: {rates}")

    Drop = (MCSpatialDropout2D if (use_mc_dropout and use_spatial) else
```

```

        L.SpatialDropout2D if use_spatial else
        MCDropout if use_mc_dropout else
        L.Dropout)

x_in = L.Input(shape=input_shape)
x = x_in
x_shortcut = x

# Block A
x = L.Conv2D(32, 3, padding="same", activation="relu")(x)
x = L.BatchNormalization()(x)
x = L.Conv2D(32, 3, padding="same", activation="relu", dilation_rate=rat
x = L.BatchNormalization()(x)
x = L.Add()([x, x_shortcut])
x = L.MaxPool2D(pool_size=2)(x)
x = Drop(drop_rate)(x)

# Block B
x = L.Conv2D(64, 3, padding="same", activation="relu")(x)
x = L.BatchNormalization()(x)
x = L.Conv2D(64, 3, padding="same", activation="relu", dilation_rate=rat
x = L.BatchNormalization()(x)
x = L.MaxPool2D(pool_size=2)(x)
x = Drop(drop_rate)(x)

# Block C
x = L.Conv2D(128, 3, padding="same", activation="relu")(x)
x = L.BatchNormalization()(x)
x = L.Conv2D(128, 3, padding="same", activation="relu", dilation_rate=ra
x = L.BatchNormalization()(x)
x = Drop(drop_rate)(x)

# x = tf.keras.layers.GlobalAveragePooling2D()(x)
# x = L.Dropout(drop_rate)(x)

x = L.Flatten()(x) # Replace GlobalAveragePooling2D
x = L.Dense(128, activation="relu")(x)
x = L.BatchNormalization()(x)
x = L.Dropout(drop_rate)(x)

# out = tf.keras.layers.Dense(1, activation="sigmoid")(x)
out = L.Dense(num_classes, activation="sigmoid")(x)
model = tf.keras.Model(x_in, out)
return model

model_dilated = build_dilated_cnn(dilated=False)
model_dilated.summary()

# --- compile
model_dilated = compile_model(model_dilated)

# --- train with validation (no test leakage)
ckpt_base, hist_base, train_time_base = train_model(model_dilated, "dilated_

# --- reload best and evaluate on TEST (timed)
model_dilated = tf.keras.models.load_model(ckpt_base)

```

```
print("\n[TEST] dilated:")
y_true_base, y_pred_base = evaluate_and_report(model_dilated, ds_test, CLASS
loss_b, acc_b, eval_t_b, pred_t_b, ips_b = measure_eval_time(model_dilated,

print(f"\n[TEST] dilated -> loss: {loss_b:.4f} | acc: {acc_b:.4f}")
```

Building BASELINE CNN with rates: (1, 1, 1)

**Model: "functional\_3"**

Layer (type)	Output Shape	Param #	Connected to
input_layer_3 (InputLayer)	(None, 160, 160, 1)	0	–
conv2d_18 (Conv2D)	(None, 160, 160, 32)	320	input_layer_3
batch_normalizatio... (BatchNormalizatio...)	(None, 160, 160, 32)	128	conv2d_18[0] [
conv2d_19 (Conv2D)	(None, 160, 160, 32)	9,248	batch_normali
batch_normalizatio... (BatchNormalizatio...)	(None, 160, 160, 32)	128	conv2d_19[0] [
add (Add)	(None, 160, 160, 32)	0	batch_normali input_layer_3
max_pooling2d_6 (MaxPooling2D)	(None, 80, 80, 32)	0	add[0][0]
mc_spatial_dropout... (MCSpatialDropout2...)	(None, 80, 80, 32)	0	max_pooling2d
conv2d_20 (Conv2D)	(None, 80, 80, 64)	18,496	mc_spatial_dr
batch_normalizatio... (BatchNormalizatio...)	(None, 80, 80, 64)	256	conv2d_20[0] [
conv2d_21 (Conv2D)	(None, 80, 80, 64)	36,928	batch_normali
batch_normalizatio... (BatchNormalizatio...)	(None, 80, 80, 64)	256	conv2d_21[0] [
max_pooling2d_7 (MaxPooling2D)	(None, 40, 40, 64)	0	batch_normali
mc_spatial_dropout... (MCSpatialDropout2...)	(None, 40, 40, 64)	0	max_pooling2d
conv2d_22 (Conv2D)	(None, 40, 40, 128)	73,856	mc_spatial_dr
batch_normalizatio... (BatchNormalizatio...)	(None, 40, 40, 128)	512	conv2d_22[0] [
conv2d_23 (Conv2D)	(None, 40, 40, 128)	147,584	batch_normali
batch_normalizatio... (BatchNormalizatio...)	(None, 40, 40, 128)	512	conv2d_23[0] [



mc_spatial_dropout... (MCSpatialDropout2...	(None, 40, 40, 128)	0	batch_normali
flatten (Flatten)	(None, 204800)	0	mc_spatial_dr
dense_3 (Dense)	(None, 128)	26,214,528	flatten[0][0]
batch_normalizatio... (BatchNormalizatio...	(None, 128)	512	dense_3[0][0]
dropout_3 (Dropout)	(None, 128)	0	batch_normali
dense_4 (Dense)	(None, 1)	129	dropout_3[0][



**Total params:** 26,503,393 (101.10 MB)



**Trainable params:** 26,502,241 (101.10 MB)



**Non-trainable params:** 1,152 (4.50 KB)







Epoch 1/10  
146/146  0s 92ms/step - accuracy: 0.7783 - loss: 0.4978  
Epoch 1: val\_accuracy improved from -inf to 0.86130, saving model to /content/outputs\_cnn\_dilation/dilated\_cnn.keras  
146/146  31s 140ms/step - accuracy: 0.7788 - loss: 0.4970 - val\_accuracy: 0.8613 - val\_loss: 0.3054



Epoch 2/10  
146/146  0s 58ms/step - accuracy: 0.8998 - loss: 0.2428  
Epoch 2: val\_accuracy improved from 0.86130 to 0.92123, saving model to /content/outputs\_cnn\_dilation/dilated\_cnn.keras  
146/146  13s 87ms/step - accuracy: 0.8998 - loss: 0.2426 - val\_accuracy: 0.9212 - val\_loss: 0.2015



Epoch 3/10  
145/146  0s 63ms/step - accuracy: 0.9326 - loss: 0.1807  
Epoch 3: val\_accuracy did not improve from 0.92123  
146/146  11s 74ms/step - accuracy: 0.9326 - loss: 0.1808 - val\_accuracy: 0.9195 - val\_loss: 0.2246



Epoch 4/10  
145/146  0s 57ms/step - accuracy: 0.9394 - loss: 0.1559  
Epoch 4: val\_accuracy improved from 0.92123 to 0.95377, saving model to /content/outputs\_cnn\_dilation/dilated\_cnn.keras  
146/146  22s 85ms/step - accuracy: 0.9393 - loss: 0.1562 - val\_accuracy: 0.9538 - val\_loss: 0.1744



Epoch 5/10  
146/146  0s 58ms/step - accuracy: 0.9480 - loss: 0.1420  
Epoch 5: val\_accuracy did not improve from 0.95377  
146/146  12s 68ms/step - accuracy: 0.9480 - loss: 0.1420 - val\_accuracy: 0.9538 - val\_loss: 0.1774

Epoch 6/10  
145/146  0s 59ms/step - accuracy: 0.9497 - loss: 0.1398  
Epoch 6: val\_accuracy improved from 0.95377 to 0.96062, saving model to /content/outputs\_cnn\_dilation/dilated\_cnn.keras  
146/146  17s 114ms/step - accuracy: 0.9497 - loss: 0.1398 - val\_accuracy: 0.9606 - val\_loss: 0.1595

Epoch 7/10  
145/146  0s 63ms/step - accuracy: 0.9607 - loss: 0.1107  
Epoch 7: val\_accuracy did not improve from 0.96062  
146/146  11s 75ms/step - accuracy: 0.9606 - loss: 0.1109 - val\_accuracy: 0.9572 - val\_loss: 0.1352

Epoch 8/10  
145/146  0s 57ms/step - accuracy: 0.9512 - loss: 0.1149  
Epoch 8: val\_accuracy improved from 0.96062 to 0.96404, saving model to /content/outputs\_cnn\_dilation/dilated\_cnn.keras  
146/146  13s 90ms/step - accuracy: 0.9513 - loss: 0.1148 - val\_accuracy: 0.9640 - val\_loss: 0.1277

Epoch 9/10  
145/146  0s 61ms/step - accuracy: 0.9676 - loss: 0.0957  
Epoch 9: val\_accuracy did not improve from 0.96404  
146/146  11s 73ms/step - accuracy: 0.9675 - loss: 0.0959 - val\_accuracy: 0.9589 - val\_loss: 0.1286

Epoch 10/10  
145/146  0s 50ms/step - accuracy: 0.9559 - loss: 0.1055  
Epoch 10: val\_accuracy did not improve from 0.96404  
146/146  9s 62ms/step - accuracy: 0.9559 - loss: 0.1056 - val\_accuracy: 0.9572 - val\_loss: 0.1209  
[TIMER] dilated\_cnn: training wall-clock = 149.86 sec (14.99 sec/epoch) (mon

```
itoring val_accuracy)
```

```
[TEST] dilated:
```

	precision	recall	f1-score	support
glioma	0.9538	0.9633	0.9585	300
notumor	0.9726	0.9654	0.9690	405
accuracy			0.9645	705
macro avg	0.9632	0.9644	0.9638	705
weighted avg	0.9646	0.9645	0.9646	705

```
Confusion matrix:
```

```
[[289  11]
 [ 14 391]]
```

```
[TIMER] evaluate: 5.75 sec
```

```
[TIMER] predict : 6.45 sec (~109.3 images/sec)
```

```
[TEST] dilated -> loss: 0.1253 | acc: 0.9603
```

## Problem 1.3 Monte Carlo dropout to assess model uncertainty

Your CNN should use Monte Carlo (MC) dropout to compute the risk at test time. For a fixed MRI image scan  $x$ , you obtain  $T$  stochastic predictions  $\{\hat{y}_t\}_{t=1}^T$ .

**(a) Mean and variance from MC passes.** Give mathematical expressions for the MC estimate of the predictive mean  $\mu_T$  and its variance  $s_T^2$  from  $\{\hat{y}_t\}$ .

### 1.3 Monte Carlo dropout to assess model uncertainty

(a) Mean and variance from MC passes. Given a set of  $T$  stochastic predictions  $\{\hat{y}_t\}_{t=1}^T$  for a single input image  $x$ , the predictive mean is:

$$\mu_T = \frac{1}{T} \sum_{t=1}^T \hat{y}_t$$

Where:

- $T$  is the total number of Monte Carlo forward passes.
- $\hat{y}_t$  is the model's prediction on the  $t$ -th forward pass with a unique dropout mask.

The predictive variance,  $s_T^2$  is:

$$s_T^2 = \frac{1}{T} \sum_{t=1}^T (\hat{y}_t - \mu_T)^2$$

It calculates the average of the squared differences between each individual prediction  $\hat{y}_t$  and the overall predictive mean  $\mu_T$ .

**(b) Accuracy guarantee.** Using Chebyshev's inequality, derive a lower bound on  $T$  ensuring  $\Pr(|\mu_T - \mu_{\text{true}}| \geq \varepsilon) \leq \delta$ . Express the minimal  $T$  in terms of  $\text{Var}(\hat{y}_t)$  (or a plug-in estimate). Clearly state any assumptions you need to make, such as Gaussianity.

(b) Accuracy guarantee.

By using Chebyshev's inequality:

$$\Pr(|X - \mu| \geq k) \leq \frac{\sigma^2}{k^2}$$

In our case, the random variable  $k$  is the sample mean from our Monte Carlo passes,  $X = \mu_T$ .

1. Assume that each stochastic prediction  $\hat{y}_t$  is i.i.d. random variable drawn from Gaussian distribution  $p(y|x, \mathcal{D})$ .
2. Assume the sample mean  $\mu_T$  is an unbiased estimator of the true predictive mean  $\mu_{\text{true}}$ ,  $E[\mu_T] = \mu_{\text{true}}$ .

then, we get:

$$\Pr(|\mu_T - \mu_{\text{true}}| \geq \varepsilon) \leq \frac{\text{Var}(\mu_T)}{\varepsilon^2} = \frac{\text{Var}(\hat{y}_t)}{T\varepsilon^2}$$

bounded by  $\delta$ :

$$\frac{\text{Var}(\hat{y}_t)}{T\varepsilon^2} \leq \delta$$

Solving for  $T$  gives us the required lower bound:

$$T \geq \frac{\text{Var}(\hat{y}_t)}{\delta\varepsilon^2}$$

the minimal  $T$  in terms of  $\text{Var}(\hat{y}_t)$

**(c) Numerical budget.** With target  $\varepsilon = 0.05$ , confidence  $1 - \delta = 0.95$  ( $\delta = 0.05$ ), a pilot of  $T_0 = 50$  passes giving standard deviation  $s_{T_0}^2 = 0.08$ , compute the required  $T$  from part (b). Describe what this means--what guarantee does this give about the accuracy of the prediction?

(c) Numerical budget. We are given the following values:

- Desired precision,  $\varepsilon = 0.05$
- Error probability,  $\delta = 0.05$  (from confidence  $1 - \delta = 0.95$ )
- A plug-in estimate for the variance from a pilot run,  $s_{T_0}^2 = 0.08$ .

Using the formula derived in part (b), we can compute the required number of MC passes,  $T$ :

$$T \geq \frac{s_{T_0}^2}{\delta \varepsilon^2}$$

$$T \geq \frac{0.08}{(0.05) \cdot (0.05)^2} = \frac{0.08}{(0.05) \cdot (0.0025)} = \frac{0.08}{0.000125}$$

$$T \geq 640$$

Therefore, the required number of Monte Carlo passes is  $T = 640$ .

It means that if we perform  $T = 640$  stochastic forward passes for a given input image, we can be at least 95% confident that our calculated predictive mean  $\mu_{640}$  will be within  $\varepsilon = 0.05$  of the "true" predictive mean  $\mu_{\text{true}}$ .

## Problem 1.4 Monte Carlo (MC) Dropout on test images

You will implement MC Dropout to quantify predictive uncertainty at test time. By keeping Dropout active during inference and sampling the model multiple times on the same inputs, you will approximate the posterior predictive distribution and summarize it with a mean prediction and a dispersion measure (standard deviation). Use MC Dropout to approximate the posterior predictive and quantify epistemic uncertainty at test time. Keep dropout on during inference, sample the model multiple times on the same inputs, and summarize the resulting distribution. Set the number of stochastic forward passes to a reasonable value based on the outputs you obtain. Remember by default `model.predict` does not keep dropout active at inference.

In this part you will:

1. Build A sampler function that deterministically picks the first 5 images per class from the test set folder for each label (10 images in total)
2. A stochastic inference predictive route that runs the model with dropout active and returns one probability per image in each round. Your CNN models is preferably defined as a function so that you can feed the baseline and dilated CNNs to compare the values here.
3. Calculate the mean and standard deviation of the stochastic forward passes in step 2 to generate a table or grid that depicts indices, `y_true`, `y_hat`, `p_mean`, `p_std`. The two latter values should be obtained with your stochastic forward passes with dropout on.
4. See if there is a mispredicted label based on your `y_hat`, then discuss the correlation between standard deviation of the stochastic runs and the final `y_hat`.

```
In [ ]: import numpy as np
import pandas as pd
```

```
import tensorflow as tf
from tensorflow.keras import layers as L, models as M, Model
from sklearn.model_selection import train_test_split
import os
```

```
In [ ]: def sample_test_images(paths, labels, images_per_class=5):
    df = pd.DataFrame({'path': paths, 'label': labels})
    sampled_paths = []
    sampled_labels = []

    unique_labels = sorted(np.unique(labels))

    for label in unique_labels:
        class_samples = df[df['label'] == label].iloc[:images_per_class]
        sampled_paths.extend(class_samples['path'].tolist())
        sampled_labels.extend(class_samples['label'].tolist())

    return sampled_paths, np.array(sampled_labels)

# 1. Build A sampler function that deterministically picks the first 5 image

sampled_paths, y_true_sampled = sample_test_images(test_paths, test_labels,
print(f"\nSuccessfully sampled {len(sampled_paths)} images from the test set

X_sampled = tf.stack([load_and_preprocess(p, y)[0] for p, y in zip(sampled_p

MC_SAMPLES = 50
```

Successfully sampled 10 images from the test set (5 per class).

```
In [ ]: # 2. A stochastic inference predictive route that runs the model with dropout
# Your CNN models is preferably defined as a function so that you can feed t

@tf.function(experimental_relax_shapes=True)
def run_stochastic_forward_pass(model, inputs):
    """Runs a single stochastic forward pass with dropout active."""
    return model(inputs, training=True)

def run_mc_dropout(model, X_data, T=MC_SAMPLES, model_name="Model"):
    """Executes T stochastic forward passes and calculates metrics."""
    all_predictions = []
    print(f"\nRunning {T} MC Dropout passes for {model_name}...")
    for t in range(T):
        pred = run_stochastic_forward_pass(model, X_data)
        all_predictions.append(pred)
    mc_predictions = np.array(all_predictions).squeeze(axis=-1).T

    p_mean = np.mean(mc_predictions, axis=1)
    p_std = np.std(mc_predictions, axis=1)

    return p_mean, p_std

p_mean_base, p_std_base = run_mc_dropout(model_base, X_sampled, model_name="
p_mean_dilated, p_std_dilated = run_mc_dropout(model_dilated, X_sampled, mod
```

Running 50 MC Dropout passes for Baseline CNN...

Running 50 MC Dropout passes for Dilated CNN...

```
In [ ]: # 3. Calculate the mean and standard deviation of the stochastic forward pass

y_hat_base = (p_mean_base >= 0.5).astype(int)
y_hat_dilated = (p_mean_dilated >= 0.5).astype(int)

results_df = pd.DataFrame({
    'Index': np.arange(len(y_true_sampled)),
    'y_true': y_true_sampled,
    'y_hat_base': y_hat_base,
    'y_hat_dilated': y_hat_dilated,
    'p_mean_base': p_mean_base,
    'p_std_base': p_std_base,
    'p_mean_dilated': p_mean_dilated,
    'p_std_dilated': p_std_dilated,
})

results_df['Correct_base'] = results_df['y_true'] == results_df['y_hat_base']
results_df['Correct_dilated'] = results_df['y_true'] == results_df['y_hat_dilated']

pd.options.display.float_format = '{:,.4f}'.format
print("\n" + "="*80)
print("---- MC Dropout Uncertainty Comparison (T=50) ----")
print("p_std measures Epistemic Uncertainty (model's lack of knowledge)")
print(results_df)
print("="*80)
```

```
=====
====
---- MC Dropout Uncertainty Comparison (T=50) ----
p_std measures Epistemic Uncertainty (model's lack of knowledge)
  Index  y_true  y_hat_base  y_hat_dilated  p_mean_base  p_std_base  \
0      0      0          0          1          0.3193    0.1272
1      1      0          0          0          0.3548    0.1524
2      2      0          0          0          0.0661    0.0568
3      3      0          0          0          0.2499    0.1205
4      4      0          0          0          0.3983    0.1587
5      5      1          0          1          0.3225    0.1547
6      6      1          1          1          0.9602    0.0385
7      7      1          1          1          0.8855    0.0909
8      8      1          1          1          0.7773    0.1363
9      9      1          1          1          0.8013    0.1497

  p_mean_dilated  p_std_dilated  Correct_base  Correct_dilated
0          0.7256          0.2235          True          False
1          0.1837          0.1999          True          True
2          0.0001          0.0001          True          True
3          0.1055          0.1452          True          True
4          0.0353          0.0718          True          True
5          0.9019          0.1166          False          True
6          0.9963          0.0093          True          True
7          0.8570          0.1780          True          True
8          0.9650          0.0517          True          True
9          0.9286          0.1067          True          True
=====
=====
```

```
In [ ]: # 4. See if there is a mispredicted label based on your y_hat, then discuss t

print("\n" + "="*80)
print("Correlation Analysis: Standard Deviation (p_std) vs. Misprediction")
print("="*80)

mispredicted_df = results_df[results_df['Correct_base'] == False]

if not mispredicted_df.empty:
    print(f"Found {len(mispredicted_df)} Mispredicted Sample(s):")
    print(mispredicted_df[['Index', 'y_true', 'y_hat_base', 'p_mean_base', '
    mean_std_mispredicted = mispredicted_df['p_std_base'].mean()
    mean_std_correct = results_df[results_df['Correct_base'] == True]['p_std

    print(f"\nMean Std Dev (p_std) (Mispredicted Samples): {mean_std_mispred
    print(f"Mean Std Dev (p_std) (Correctly Predicted Samples): {mean_std_co

else:
    print("No mispredicted samples found in the sampled data.")
```



```
=====
====
Correlation Analysis: Standard Deviation (p_std) vs. Misprediction
=====
====
```

Found 1 Mispredicted Sample(s):

	Index	y_true	y_hat_base	p_mean_base	p_std_base
5	5	1	0	0.3225	0.1547

Mean Std Dev (p\_std) (Mispredicted Samples): 0.1547

Mean Std Dev (p\_std) (Correctly Predicted Samples): 0.1146

### Observation

1. The average standard deviation of misclassified samples was 0.1547: this means that the model has a higher uncertainty in predicting this sample because the results of multiple random forward propagations fluctuate more.
2. The average standard deviation for correctly classified samples was 0.1146: this means that the model has lower predictive uncertainty for these samples, as it gives relatively consistent results in multiple random forward propagations.

The fact that  $0.1547 > 0.1146$  suggests that when the model predicts an error, its own uncertainty (measured by p\_std) is higher. This strongly demonstrates that standard deviation can be used as an effective indicator of the model's prediction of confidence.

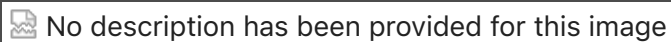
## Problem 2: All that glitters (20 points)

In this problem you will use Gaussian process regression to model the trends in gold medal performances of selected events in the summer Olympics. The objectives of this problem are for you to:

- Gain experience with Gaussian processes, to better understand how they work
- Explore how posterior inference depends on the properties of the prior mean and kernel
- Use Bayesian inference to identify unusual events
- Practice making your Python code modular and reusable

For this problem, the only starter code we provide is to read in the data and extract one event. You may write any GP code that you choose to, but please do not use any package for Gaussian processes; your code should be "np-complete" (using only basic numpy methods). You are encouraged to start from the [GP demo code](#) used in class.

When we ran the GP demo code from class on the marathon data, it generated the following plot:



Note several properties of this plot:

- It shows the Bayesian confidence of the regression, as a shaded area. This is a 95% confidence band because it has width given by  $\pm 2\sqrt{V}$ , where  $V$  is the estimated variance. The variance increases at the right side, for future years.
- The gold medal time for the 1904 marathon is outside of this confidence band. In fact, the 1904 marathon was an [unusual event](#), and this is apparent from the model.
- The plot shows the posterior mean, and also shows one random sample from the posterior distribution.

Your task in this problem is generate such a plot for six different Olympic events by writing a function

```
def gp_olympic_event(year, result, kernel, mean, noise,
event_name):    ...
```

where the input variables are the following:

- `year` : a numpy array of years (integers)
- `result` : a numpy array of numerical results, for the gold medal performances in that event
- `kernel` : a kernel function
- `mean` : a mean function
- `noise` : a single float for the variance of the noise,  $\sigma^2$
- `event_name` : a string used to label the y-axis, for example "marathon min/mile (men's event)"

Your function should compute the Gaussian process regression, and then display the resulting plot, analogous to the plot above for the men's marathon event.

You will then process **six** of the events, three men's events and three women's events, and call your function to generate the corresponding six plots.

For each event, you should create a markdown cell that describes the resulting model. Comment on such things as:

- How you chose the kernel, mean, and noise.
- Why the plot does or doesn't look satisfactory to you
- If there are any events such as the 1904 marathon that are notable.
- What happens to the posterior mean (for example during WWII) if there are gaps in the data

Use your best judgement to describe your findings; post questions to EdD if things are unclear. And have fun!

---

In the remainder of this problem description, we recall how we processed the marathon data, as an example. The following cell reads in the data and displays the collection of events that are included in the dataset.

```
In [ ]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

dat = pd.read_csv('/content/drive/MyDrive/Assn2/olympic_results.csv')
events = set(np.array(dat['Event']))
print(events)

marathon = dat[dat['Event'] == 'Triple Jump Women']
marathon = marathon[marathon['Medal']=='G']
marathon = marathon.sort_values('Year')
# time = np.array(marathon['Result'])
# mpm = []
# for tm in time:
```

```
# t = np.array(tm.split(':'), dtype=float)
# print(t)
# # minutes_per_mile = (t[0]*60*60 + t[1]*60 + t[2])/(60*26.2)
# # mpm.append(minutes_per_mile)
# mpm.append(t)

# marathon['Minutes per Mile'] = np.round(mpm,2)
# marathon = marathon.drop(columns=['Gender', 'Event'], axis=1)
# marathon.reset_index(drop=True, inplace=True)
# year = np.array(marathon['Year'])
# result = np.array(marathon['Minutes per Mile'])
marathon
```

```
{'800M Men', '5000M Men', 'Triple Jump Women', 'Long Jump Women', '10000M Men', '400M Hurdles Men', '4X400M Relay Women', 'Pole Vault Men', 'Javelin Throw Women', '100M Men', '4X100M Relay Women', 'Long Jump Men', 'High Jump Men', '4X100M Relay Men', 'Heptathlon Women', '50Km Race Walk Men', 'Discus Throw Women', 'Shot Put Men', 'Shot Put Women', '3000M Steeplechase Women', '10000M Women', '1500M Men', 'Triple Jump Men', '110M Hurdles Men', '100M Hurdles Women', '400M Women', '200M Women', '800M Women', '400M Hurdles Women', 'Javelin Throw Men', '20Km Race Walk Women', '200M Men', 'Hammer Throw Women', '20Km Race Walk Men', 'Hammer Throw Men', '1500M Women', '100M Women', 'Pole Vault Women', 'Decathlon Men', '4X400M Relay Men', 'Marathon Women', '400M Men', 'High Jump Women', 'Marathon Men', '5000M Women', '3000M Steeplechase Men', 'Discus Throw Men'}
```

Out[1]:

	Gender	Event	Location	Year	Medal	Name	Nationality	Result
2391	W	Triple Jump Women	Atlanta	1996	G	Inessa KRAVETS	UKR	15.33
2382	W	Triple Jump Women	Sydney	2000	G	Tereza MARINOVA	BUL	15.2
2388	W	Triple Jump Women	Athens	2004	G	Francoise MBANGO ETONE	CMR	15.3
2381	W	Triple Jump Women	Beijing	2008	G	Francoise MBANGO ETONE	CMR	15.39
2385	W	Triple Jump Women	London	2012	G	Olga RYPAKOVA	KAZ	14.98
2378	W	Triple Jump Women	Rio	2016	G	Caterine IBARGUEN	COL	15.17

We then process the time to compute the minutes per mile (without checking that the race was actually 26.2 miles!)

```
In [ ]: def rbf_kernel(x1, x2, length_scale=10.0, sigma_f=1.0):
        .....
```

```

    Computes the radial basis function (RBF) kernel.
    This is our "engine" for modeling smooth trends.
    """
    sqdist = np.subtract.outer(x1, x2)**2
    return sigma_f**2 * np.exp(-0.5 / length_scale**2 * sqdist)

def constant_mean(x, intercept=0.0):
    """
    A simple constant mean function.
    This is our "transmission" for when we expect no overall trend.
    """
    return np.full_like(x, fill_value=intercept, dtype=float)

def linear_mean(x, slope, intercept):
    """
    Computes a linear mean function for an input array x.

    Args:
        x (np.ndarray): The input years.
        slope (float): The slope of the linear trend.
        intercept (float): The intercept of the linear trend.

    Returns:
        np.ndarray: The mean values for each input x.
    """
    return slope * x + intercept

```

```

In [ ]: def mpm_converter(time_str, distance_miles=26.2):
    t = np.array(time_str.split(':'), dtype=float)
    total_seconds = t[0] * 3600 + t[1] * 60 + t[2]
    minutes_per_mile = (total_seconds / 60.0) / distance_miles

    return minutes_per_mile

def preprocess_event_data(df, event_name, unit_conversion=None, unit_label='
    """
    Filters and processes Olympic data for a specific event.
    """
    event_df = df[(df['Event'] == event_name) & (df['Medal'] == 'G')].copy()
    event_df = event_df.sort_values('Year').reset_index(drop=True)

    if unit_conversion:
        event_df['Result_Processed'] = event_df['Result'].apply(unit_conversion)
    else:
        event_df['Result_Processed'] = pd.to_numeric(event_df['Result'], errors='coerce')

    event_df.dropna(subset=['Result_Processed'], inplace=True)
    y_axis_label = f"{event_name} ({unit_label})"
    return np.array(event_df['Year']), np.array(event_df['Result_Processed'])

```

```

In [ ]: def gp_olympic_event(year, result, kernel, mean, noise_var, event_name,
                             use_noisy_samples):

    year_mean = np.mean(year)
    year_centered = year - year_mean

```

```

xs_years = np.linspace(min(year) - 1, max(year) + 10, 500)
xs_centered = xs_years - year_mean

K = kernel(year_centered, year_centered)
Ks = kernel(year_centered, xs_centered)
Kss = kernel(xs_centered, xs_centered)

Ki = np.linalg.inv(K + noise_var * np.eye(len(year)))

post_mean = mean(xs_centered) + Ks.T @ Ki @ (result - mean(year_centered))

if use_noisy_samples:
    Kss_noisy = Kss + noise_var * np.eye(len(xs_years))
    post_cov = Kss_noisy - Ks.T @ Ki @ Ks
else:
    post_cov = Kss - Ks.T @ Ki @ Ks

plt.style.use('seaborn-v0_8-whitegrid')
fig, ax = plt.subplots(1, 1, figsize=(12, 7))

S2_smooth = np.diag(Kss - Ks.T @ Ki @ Ks).copy()
S2_smooth[S2_smooth < 0] = 0

ax.fill_between(xs_years, post_mean - 2*np.sqrt(S2_smooth), post_mean +
                step="pre", alpha=0.2, label='posterior 95% confidence',

sample = np.random.multivariate_normal(post_mean, post_cov + 1e-6 * np.e

ax.plot(xs_years, sample, c='r', linestyle='--', linewidth=.2, label='sam
ax.plot(xs_years, post_mean, c='black', linestyle='--', linewidth=.5, lab
ax.scatter(year, result, color='red', marker='o', linewidth=.4, label='d

ax.set_xlabel('year', fontsize=14)
ax.set_ylabel(event_name, fontsize=14)
ax.set_title(f"Gaussian Process Regression for {event_name.split(' ')[0]
ax.legend()
ax.set_xlim(min(xs_years), max(xs_years))
plt.show()

```

```

In [ ]: # for process six of the events, three men's events and three women's events
# How you chose the kernel, mean, and noise.
# Why the plot does or does not look satisfactory to you
# If there are any events such as the 1904 marathon that are notable.
# What happens to the posterior mean (for example during WWII) if there are

# Adjust these parameters for each event

event_name = 'Marathon Men'

year, result, y_label = preprocess_event_data(dat, event_name, unit_conversion

kernel_choice = lambda x1, x2: rbf_kernel(x1, x2, length_scale=7.0, sigma_f=
mean_choice = lambda x: constant_mean(x, intercept=np.mean(result))
# slope, intercept = np.polyfit(year, result, 1)
# mean_choice = lambda x: linear_mean(x, slope, intercept)

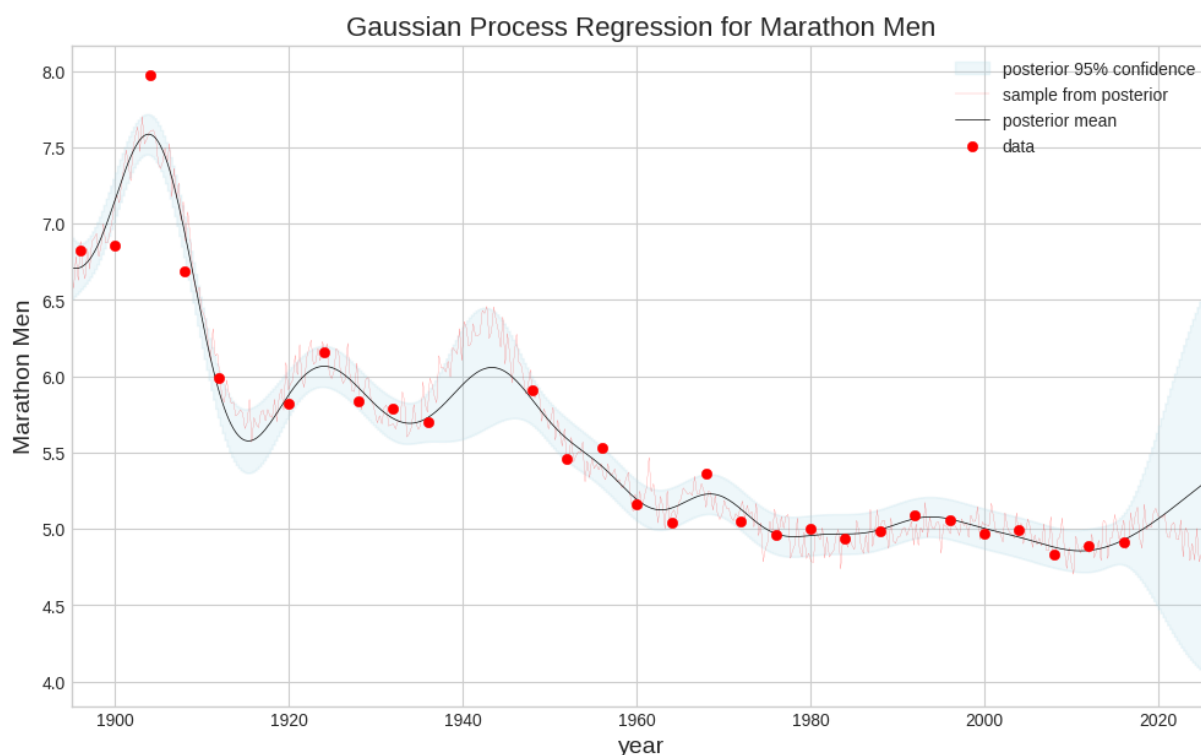
```

```

noise_choice = np.square(5/60)

gp_olympic_event(
    year=year,
    result=result,
    kernel=kernel_choice,
    mean=mean_choice,
    noise_var=noise_choice,
    event_name=event_name,
    use_noisy_samples=True
)

```



### Model Setup: How you chose the kernel, mean, and noise.

The Gaussian Process model was configured with these key components:

1. **Kernel:** An RBF kernel was selected with `length_scale=7.0` to capture trends over a span of about seven years and `sigma_f` set to the standard deviation of the results.
2. **Mean Function:** A constant mean was applied, assuming a flat baseline for performance and allowing the kernel to model all the trends and fluctuations.
3. **Noise:** The noise was set to `np.square(5/60)`, which represents a slight variation of about 5 seconds in the gold medal times.

### Why the plot does or doesn't look satisfactory to you

The posterior mean smoothly follows the observed gold medal times, which indicates improving performance over time.

The 95% confidence is narrow where data points exist, meaning a high certainty. The



band widens significantly for years after data disappear, correctly showing the increasing uncertainty.

The dataset doesn't contain any major outliers, and the model captures the data smoothly.

### If there are any events such as the 1904 marathon that are notable

Yes, the 1904 marathon data point is a notable event, as it shows a significant spike that deviates from the overall trend.

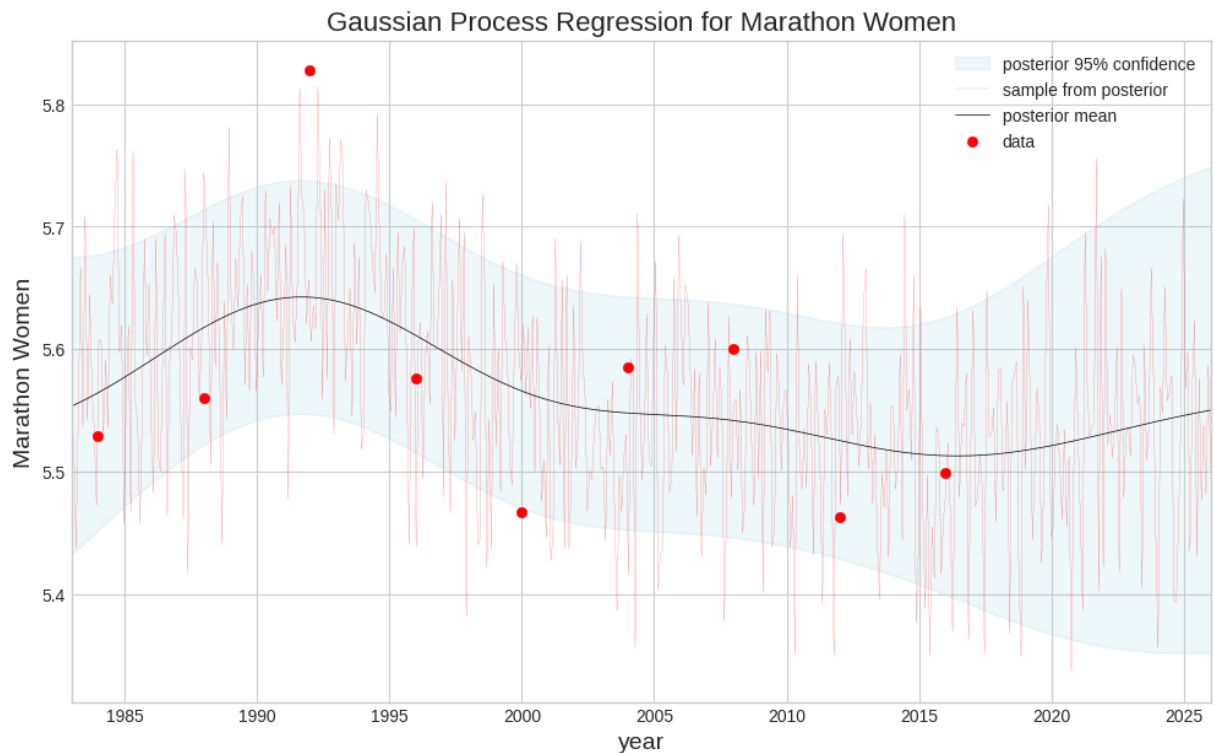
### What happens to the posterior mean (for example during WWII) if there are gaps in the data

During the periods with gaps in the data, such as during WWII (1940 and 1944) and after 2016, the posterior mean smoothly interpolates between the available data points. The confidence band widens in these regions, reflecting the increased uncertainty caused by the lack of data.

```
In [ ]: event_name = 'Marathon Women'
year, result, y_label = preprocess_event_data(dat, event_name, unit_conversion='m')

kernel_choice = lambda x1, x2: rbf_kernel(x1, x2, length_scale=7.0, sigma_f=1)
mean_choice = lambda x: constant_mean(x, intercept=np.mean(result))
# slope, intercept = np.polyfit(year, result, 1)
# mean_choice = lambda x: linear_mean(x, slope, intercept)
noise_choice = np.square(5 / 60)

gp_olympic_event(
    year=year,
    result=result,
    kernel=kernel_choice,
    mean=mean_choice,
    noise_var=noise_choice,
    event_name=event_name,
    use_noisy_samples=True
)
```



### Model Setup: How you chose the kernel, mean, and noise.

The Gaussian Process model was configured with these key components:

1. **Kernel:** An RBF kernel was selected with `length_scale=7.0` to capture trends over a span of about seven years and `sigma_f` set to the standard deviation of the results.
2. **Mean Function:** A constant mean was applied, assuming a flat baseline for performance and allowing the kernel to model all the trends and fluctuations.
3. **Noise:** The noise was set to `np.square(5/60)`, which represents a slight variation of about 5 seconds in the gold medal times.

### Why the plot does or doesn't look satisfactory to you

The posterior mean smoothly follows the observed gold medal times, which indicates improving performance over time.

The 95% confidence is narrow where data points exist, meaning a high certainty. The band widens significantly for years after data disappear, correctly showing the increasing uncertainty.

The dataset doesn't contain any major outliers, and the model captures the data smoothly.

### If there are any events such as the 1904 marathon that are notable

Yes, the 1904 marathon data point is a notable event, as it shows a significant spike that deviates from the overall trend.

## What happens to the posterior mean (for example during WWII) if there are gaps in the data

During the periods with gaps in the data, such as during WWII (1940 and 1944) and after 2016, the posterior mean smoothly interpolates between the available data points. The confidence band widens in these regions, reflecting the increased uncertainty caused by the lack of data.

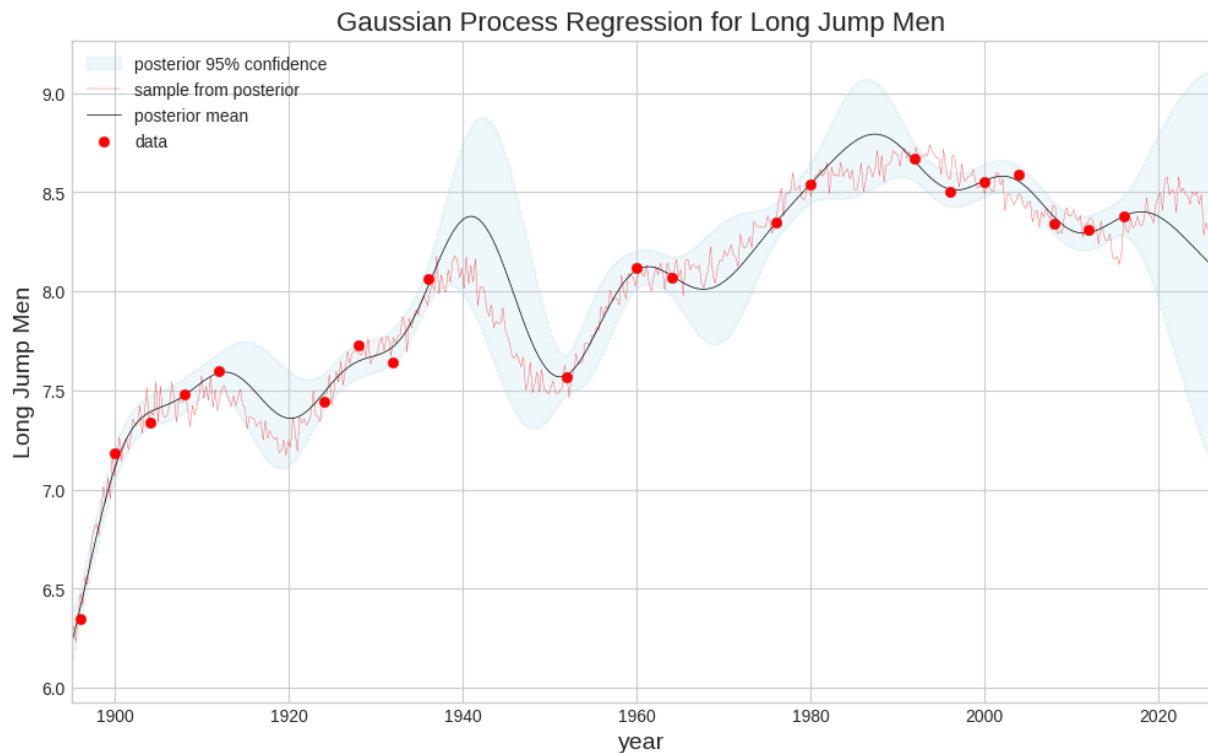
```
In [ ]: def meters_converter(result_str, unit_label="meters (m)"):
        """
        Converts a string result (in meters, e.g., '6.24' or '7.') to a float.
        For long jump, the unit is already meters, so this function primarily cl
        """
        try:
            # Handle the incomplete entry '7.' by adding '0' and converting to f
            # This assumes incomplete entries are meant to be '7.00' or similar.
            if result_str.endswith('.'):
                return float(result_str + '00')
            return float(result_str.strip())
        except ValueError:
            print(f"Error converting result string: {result_str}. Returning NaN.")
            return np.nan
```

```
In [ ]: event_name="Long Jump Men"

year, result, y_label = preprocess_event_data(dat, event_name, unit_conversion

kernel_choice = lambda x1, x2: rbf_kernel(x1, x2, length_scale=7.0, sigma_f=
mean_choice = lambda x: constant_mean(x, intercept=np.mean(result))
# slope, intercept = np.polyfit(year, result, 1)
# mean_choice = lambda x: linear_mean(x, slope, intercept)
noise_choice = np.square(0.05)

gp_olympic_event(
    year=year,
    result=result,
    kernel=kernel_choice,
    mean=mean_choice,
    noise_var=noise_choice,
    event_name=event_name,
    use_noisy_samples=True
)
```



### Model Setup: How you chose the kernel, mean, and noise.

The Gaussian Process model was configured with these key components:

1. **Kernel:** An RBF kernel was selected with `length_scale=7.0`.
2. **Mean Function:** A constant mean was applied.
3. **Noise:** The noise was set to `np.square(0.05)`.

### Why the plot does or doesn't look satisfactory to you

The plot is highly satisfactory. The posterior mean is smooth and indicates an overall improving performance.

The 95% confidence is narrow where data points exist. The band widens significantly for years after data disappear, showing an uncertainty.

### If there are any events such as the 1904 marathon that are notable

No any data point seems to be an outlier.

### What happens to the posterior mean (for example during WWII) if there are gaps in the data

During the periods with gaps in the data during WWII (1940 and 1944), the posterior mean smoothly interpolates between the data points. The confidence band widens in these gaps, reflecting the increased uncertainty. After 2020, the mean shows a slight dip, and the confidence band expands dramatically, indicating high uncertainty about future gold medal results.

```

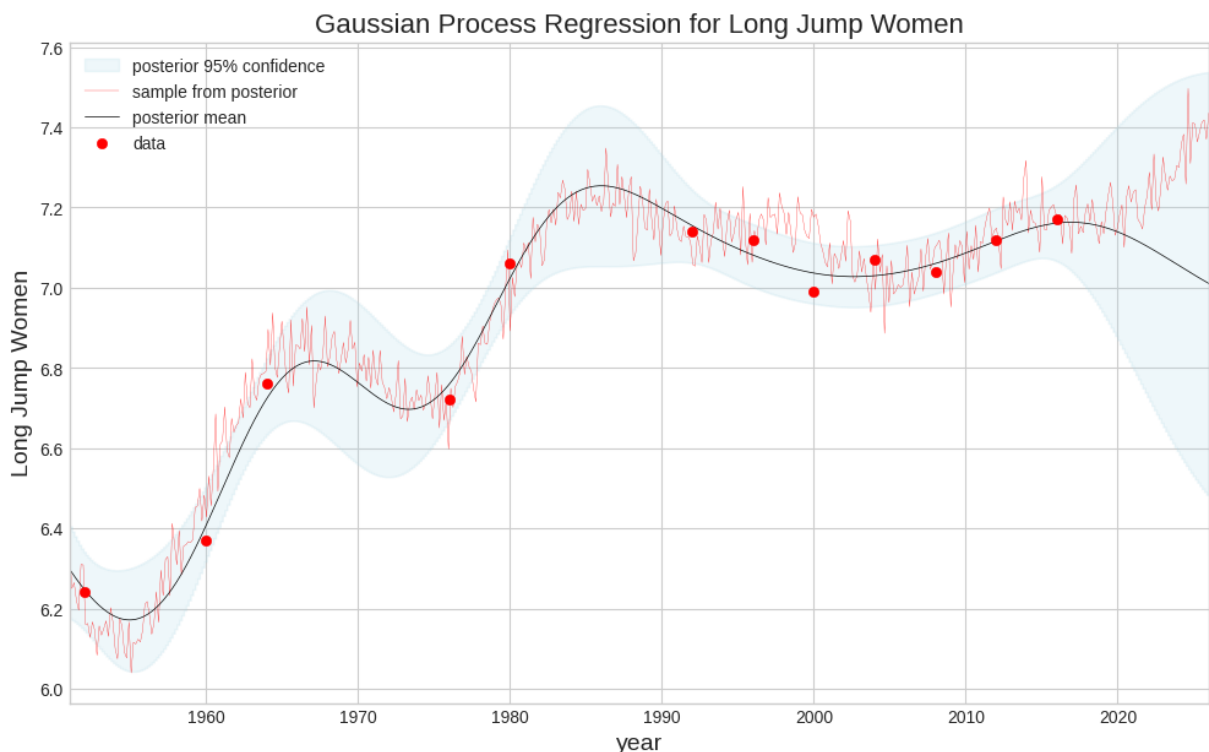
In [ ]: event_name="Long Jump Women"

year, result, y_label = preprocess_event_data(dat, event_name, unit_conversion)

kernel_choice = lambda x1, x2: rbf_kernel(x1, x2, length_scale=7.0, sigma_f=1)
mean_choice = lambda x: constant_mean(x, intercept=np.mean(result))
# slope, intercept = np.polyfit(year, result, 1)
# mean_choice = lambda x: linear_mean(x, slope, intercept)
noise_choice = np.square(0.05)

gp_olympic_event(
    year=year,
    result=result,
    kernel=kernel_choice,
    mean=mean_choice,
    noise_var=noise_choice,
    event_name=event_name,
    use_noisy_samples=True
)

```



### Model Setup: How you chose the kernel, mean, and noise.

The Gaussian Process model was configured with these key components:

1. **Kernel**: An RBF kernel was selected with `length_scale=7.0`.
2. **Mean Function**: A constant mean was applied.
3. **Noise**: The noise was set to `np.square(0.05)`.

### Why the plot does or doesn't look satisfactory to you

The plot is highly satisfactory. The posterior mean is smooth and indicates an overall

improving performance.

The 95% confidence is narrow where data points exist. The band widens significantly for years after data disappear, showing an uncertainty.

**If there are any events such as the 1904 marathon that are notable**

No any data point seems to be an outlier. The event was first introduced in 1952.

**What happens to the posterior mean (for example during WWII) if there are gaps in the data**

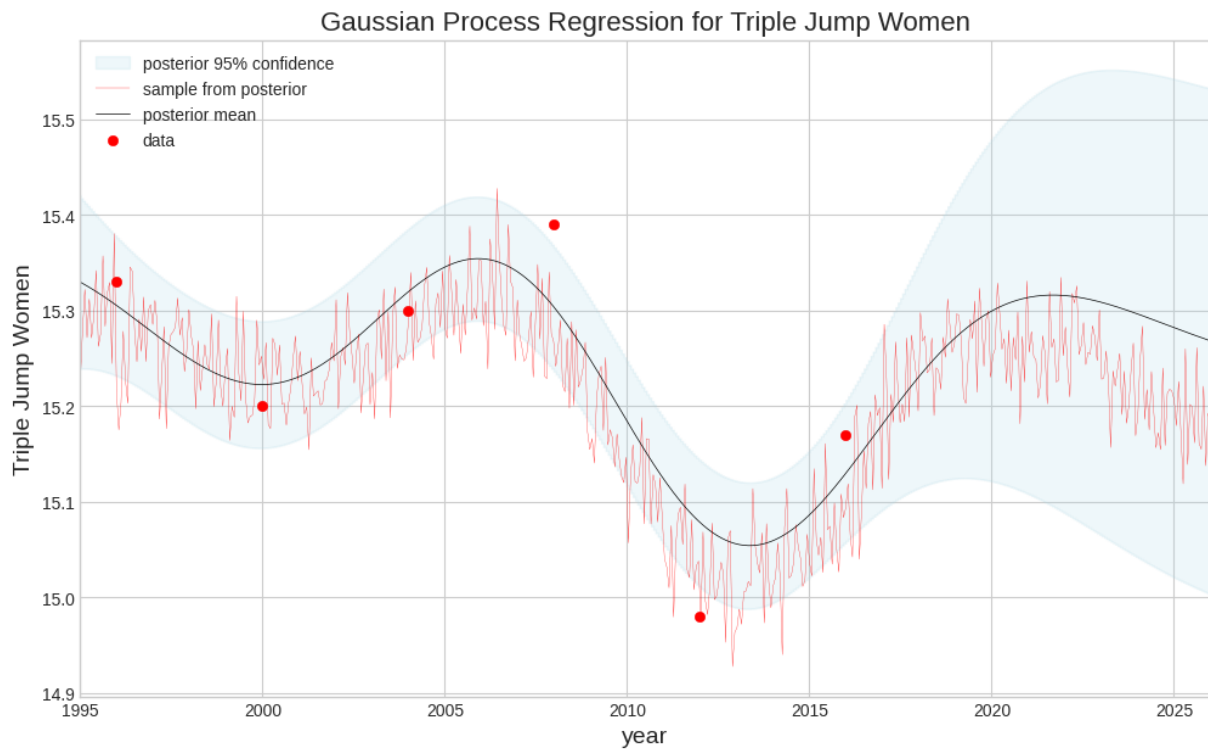
No events during WWII.

```
In [ ]: event_name="Triple Jump Women"

year, result, y_label = preprocess_event_data(dat, event_name, unit_conversion)

kernel_choice = lambda x1, x2: rbf_kernel(x1, x2, length_scale=5.0, sigma_f=
mean_choice = lambda x: constant_mean(x, intercept=np.mean(result))
# slope, intercept = np.polyfit(year, result, 1)
# mean_choice = lambda x: linear_mean(x, slope, intercept)
noise_choice = np.square(0.04)

gp_olympic_event(
    year=year,
    result=result,
    kernel=kernel_choice,
    mean=mean_choice,
    noise_var=noise_choice,
    event_name=event_name,
    use_noisy_samples=True
)
```



### Model Setup: How you chose the kernel, mean, and noise.

1. **Kernel:** An RBF kernel with `length_scale=5.0`.
2. **Mean Function:** A constant mean was applied, assuming a flat baseline for performance and allowing the kernel to model all the fluctuations.
3. **Noise:** The noise was set to `np.square(0.04)`.

### Why the plot does or doesn't look satisfactory to you

The plot is highly satisfactory because it correctly reflects the inherent instability of a relatively new event with few data points. It shows a clear periodic-like fluctuation, with an initial peak (around 2005) followed by a sharp dip (around 2012) and then a recovery.

The 95% confidence band is wide across the entire plot, especially compared to events with longer histories. This means the model has low certainty about the true underlying function, which is appropriate for a data series with only seven points since its introduction. The band widens dramatically after 2020, showing high uncertainty.

The model is sensitive due to the small dataset, but the curve successfully interpolates the points.

### If there are any events such as the 1904 marathon that are notable

The event was only introduced to the Olympics in **1996**, so there are no historical data points from previous eras. The most notable data point is the very low result around **2012**, which forces the posterior mean to dip significantly.

### What happens to the posterior mean (for example during WWII) if there are gaps in

## the data

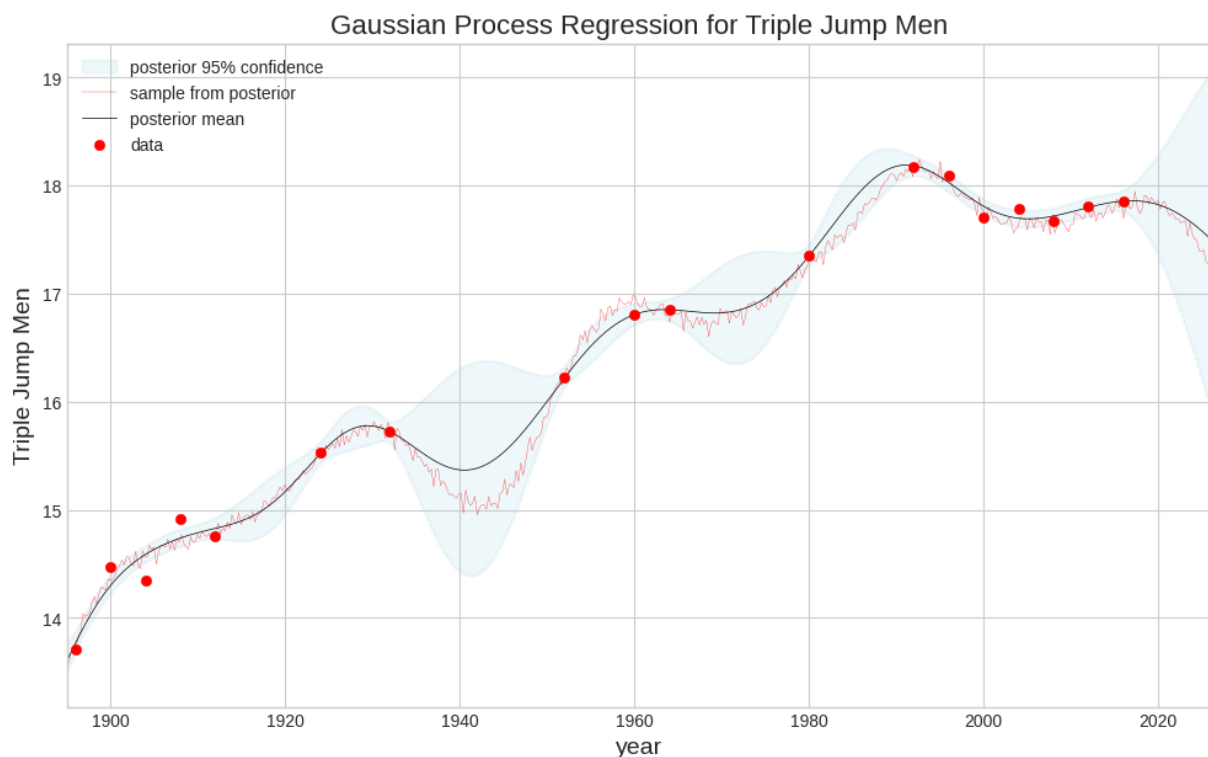
No events during WWII.

```
In [ ]: event_name="Triple Jump Men"

year, result, y_label = preprocess_event_data(dat, event_name, unit_conversion='m')

kernel_choice = lambda x1, x2: rbf_kernel(x1, x2, length_scale=10.0, sigma_f=1)
mean_choice = lambda x: constant_mean(x, intercept=np.mean(result))
# slope, intercept = np.polyfit(year, result, 1)
# mean_choice = lambda x: linear_mean(x, slope, intercept)
noise_choice = np.square(0.05)

gp_olympic_event(
    year=year,
    result=result,
    kernel=kernel_choice,
    mean=mean_choice,
    noise_var=noise_choice,
    event_name=event_name,
    use_noisy_samples=True
)
```



## Model Setup: How you chose the kernel, mean, and noise.

The Gaussian Process model was configured with these key components:

1. **Kernel:** An RBF kernel with an intentionally long `length_scale=10.0`.
2. **Mean Function:** A constant mean was applied, assuming a flat baseline for performance and allowing the kernel to model all the trends and fluctuations.



3. **Noise:** The noise was set to `np.square(0.05)`, meaning a slight variation of about 5 centimeters (0.05 meters) in the gold medal results.

### Why the plot does or doesn't look satisfactory to you

The plot is highly satisfactory. The posterior mean shows a very clear, smooth progression from around 13.5m at the start of the 20th century to over 17.5m now, with two distinct periods of rapid improvement (1950s-1970s and 1980s-1990s).

The 95% confidence is narrow where the data is dense, reflecting high certainty in the interpolation. The wider `length_scale` successfully smooths out minor fluctuations, making the confidence band relatively tight across the entire history. The band correctly widens significantly for years after the data disappear, showing an uncertainty.

### If there are any events such as the 1904 marathon that are notable

The period of rapid ascent from the 1950s to the 1990s is the most notable structural feature, showing a massive shift in performance over a few decades.

The data point in 1936 is also notable, as it is relatively low and forces the posterior mean to dip sharply before the gap of WWII.

### What happens to the posterior mean (for example during WWII) if there are gaps in the data

During the periods with gaps in the data, such as during WWII (1940 and 1944), the posterior mean smoothly interpolates between the data points. The confidence band widens in these gaps, reflecting the increased uncertainty.

After 2020, the mean shows a slight continued decline, and the confidence band expands dramatically, indicating high uncertainty about future gold medal results.

## Problem 3: Double descent! (20 points)



In this problem you will explore the "double descent" phenomenon that was recently discovered as a key principle underlying the performance of deep neural networks. The problem setup is a "random features" version of a 2-layer neural network. The weights in the first layer are random and fixed, and the weights in the second layer are estimated from data. As we increase the number of neurons in the hidden layer, the dimension  $p$  of

model increases. It's helpful to define the ratio  $\gamma = p/n$  of variables to sample points. If  $\gamma < 1$  then we want to use the OLS estimator, and if  $\gamma > 1$  we want to use the minimum

norm estimator.

Your mission (should you choose to accept it), is

1. Implement a function `OLS_or_minimum_norm` that computes the least squares solution when  $\gamma < 1$ , and the minimum norm solution when  $\gamma > 1$ . (When  $\gamma = 1$  the estimator does not, in general, exist.)
2. Run the main code we give you to average over many trials, and to compute and plot the estimated risk for a range of values of  $\gamma$ .
3. Next, extend the starter code so that you compute (estimates of) the squared-bias and variance of the models. To do this, note that you'll need access to the true regression function, which is provided. You may want to refer to the demo code for smoothing kernels as an example.
4. Using your new code, extend the plotting function we provide so that you plot the squared-bias, variance, and risk together on the same plot.
5. Finally, comment on the results, describing why it might make sense that the squared bias, variance, and risk have the given shapes that they do.
6. Show that in the overparameterized regime  $\gamma > 1$ , as  $\lambda \rightarrow 0$ , the ridge regression estimator converges to the minimum norm estimator.

By doing this exercise you will solidify your understanding of the meaning of bias and variance, and also gain a better understanding of the "double descent" phenomenon for overparameterized neural networks, and their striking resistance to overfitting.

We're available in OH to help with any issues you run into!

If you have any interest in background reading on this topic (not expected or required), we recommend Hastie et al., ["Surprises in high-dimensional ridgeless least squares regression"](#).

```
In [1]: import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline
```

## Problem 3.1

Implement the function `OLS_or_minimum_norm` that computes the OLS solution for  $\gamma < 1$ , and the minimum norm solution for  $\gamma > 1$ .

```
In [2]: def OLS_or_minimum_norm(X, y):
    ## Your code here
    ....

    Computes the OLS solution for an overdetermined system (n > d)
    and the minimum norm solution for an underdetermined system (n < d).

    The behavior is determined by gamma = d / n.
```

```

- If  $\gamma < 1$ , it computes the OLS solution:  $\beta = (X^T X)^{-1} X^T y$ 
- If  $\gamma > 1$ , it computes the minimum norm solution:  $\beta = X^T (X X^T)^{-1} y$ 

Args:
    X (np.ndarray): The design matrix of shape (n_samples, n_features).
    y (np.ndarray): The target vector of shape (n_samples,).

Returns:
    np.ndarray: The estimated coefficient vector  $\beta$ .
    """
    n_samples, n_features = X.shape
    gamma = n_features / n_samples

    if gamma < 1:
        beta = np.linalg.inv(X.T @ X) @ X.T @ y

    elif gamma > 1:
        beta = X.T @ np.linalg.inv(X @ X.T) @ y

    else:
        try:
            beta = np.linalg.solve(X, y)
        except np.linalg.LinAlgError:
            raise np.linalg.LinAlgError("For gamma = 1, X must be an invertible matrix")

    return beta

```

In [31]: # A plotting function we provide. No need to change this, although you can i

```

def plot_double_descent_risk(gammas, risk, sigma):
    gammas = np.round(gammas, 2)
    fig, ax = plt.subplots(figsize=(10,6))
    tick_pos = np.zeros(len(gammas))
    for i in np.arange(len(gammas)):
        if gammas[i] <= 1:
            tick_pos[i] = gammas[i] * 10
        else:
            tick_pos[i] = gammas[i] + 9
    ax.axvline(x=tick_pos[np.array(gammas)==1][0], linestyle='dashed', color='gray')
    ax.axhline(y=sigma**2, linestyle='dashed', color='gray')
    ax.scatter(tick_pos, risk, color='salmon')
    ax.plot(tick_pos, risk, color='gray', linewidth=.5)

    tickgam = [gam for gam in gammas if (gam > .05 and gam <= .9) or gam >= 1]
    ticks = [tick_pos[j] for j in np.arange(len(tick_pos)) if gammas[j] in tickgam]
    ax.xaxis.set_ticks(ticks)
    ax.xaxis.set_ticklabels(tickgam)

    plt.xlabel(r'$\gamma = \frac{p}{n}$', fontsize=18)
    _ = plt.ylabel('Risk', fontsize=18)

```

## Data setup

The following cell sets up our data. The inputs  $X$  are random Gaussian vectors of

dimension  $d = 10$ . Then, we map these using a neural network with fixed, Gaussian weights, to get random features corresponding to  $p^* = 150$  hidden neurons. The second layer coefficients are  $\beta^* \in \mathbb{R}^{p^*}$ , which are fixed. This defines the true model.

```
In [4]: # just execute this cell, after you define the function above.

np.random.seed(123456)

sigma = 1
d = 10
p_star = 150
signal_size = 5

W_star = (1/np.sqrt(d)) * np.random.randn(d, p_star)
beta_star = np.arange(p_star)
beta_star = signal_size * beta_star / np.sqrt(np.sum(beta_star**2))

N = 10000
X = np.random.randn(N, d)

# f_star is the true regression function, for computing the squared bias
f_star = np.dot(np.tanh(np.dot(X, W_star)), beta_star)
noise = sigma * np.random.randn(N)
y = f_star + noise
yf = np.concatenate((y.reshape(N,1), f_star.reshape(N,1)), axis=1)
```

## Train a sequence of models for different values of $\gamma$

Next, we train a sequence of models for different values of  $\gamma$ , always fixing the sample size at  $n = 200$ , but varying the dimension  $p = \gamma n$ . When  $p < p^*$  we just take the first  $p$  features in the true model. When  $p > p^*$  we add  $p - p^*$  neurons to the hidden layer, with their own random weights.

In the code below, we loop over the different values of  $\gamma$ , and for each  $\gamma$  we run 100 trials, each time generating a new training set of size  $n = 200$ . The model (either OLS or minimum norm) is then computed, the MSE is computed, and finally the risk is estimated by averaging over all 100 trials.

```
In [5]: trials = 100
n = 200

gammas = list(np.arange(.1, 1, .1)) + [.92, .94, 1, 1.1, 1.2, 1.4, 1.6] + li
gammas = [.01, .05] + gammas
risk = []
for gamma in gammas:
    err = []
    p = int(n * gamma)
    if gamma == 1:
        risk.append(np.inf)
        continue
    W = (1/np.sqrt(d)) * np.random.randn(d, p)
```

```

W[:, :min(p, p_star)] = W_star[:, :min(p, p_star)]
for i in np.arange(trials):
    X_train, X_test, yf_train, yf_test = train_test_split(X, yf, train_s
    H_train = np.tanh(np.dot(X_train, W))
    H_test = np.tanh(np.dot(X_test, W))
    beta_hat = OLS_or_minimum_norm(H_train, yf_train[:,0])
    yhat_test = H_test @ beta_hat
    err.append(np.mean((yhat_test - yf_test[:,0])**2))
print('gamma=%.2f  p=%d  n=%d  risk=%.3f' % (gamma, p, n, np.mean(err)))
risk.append(np.mean(err))

```

```

gamma=0.01  p=2  n=200  risk=7.373
gamma=0.05  p=10  n=200  risk=4.146
gamma=0.10  p=20  n=200  risk=2.294
gamma=0.20  p=40  n=200  risk=2.099
gamma=0.30  p=60  n=200  risk=2.267
gamma=0.40  p=80  n=200  risk=2.534
gamma=0.50  p=100  n=200  risk=3.011
gamma=0.60  p=120  n=200  risk=3.573
gamma=0.70  p=140  n=200  risk=4.443
gamma=0.80  p=160  n=200  risk=6.885
gamma=0.90  p=180  n=200  risk=15.773
gamma=0.92  p=184  n=200  risk=20.500
gamma=0.94  p=188  n=200  risk=31.661
gamma=1.10  p=220  n=200  risk=24.179
gamma=1.20  p=240  n=200  risk=14.097
gamma=1.40  p=280  n=200  risk=8.089
gamma=1.60  p=320  n=200  risk=6.543
gamma=2.00  p=400  n=200  risk=5.301
gamma=3.00  p=600  n=200  risk=4.073
gamma=4.00  p=800  n=200  risk=3.658
gamma=5.00  p=1000  n=200  risk=3.430
gamma=6.00  p=1200  n=200  risk=3.309
gamma=7.00  p=1400  n=200  risk=3.135
gamma=8.00  p=1600  n=200  risk=3.190
gamma=9.00  p=1800  n=200  risk=3.056
gamma=10.00  p=2000  n=200  risk=3.062

```

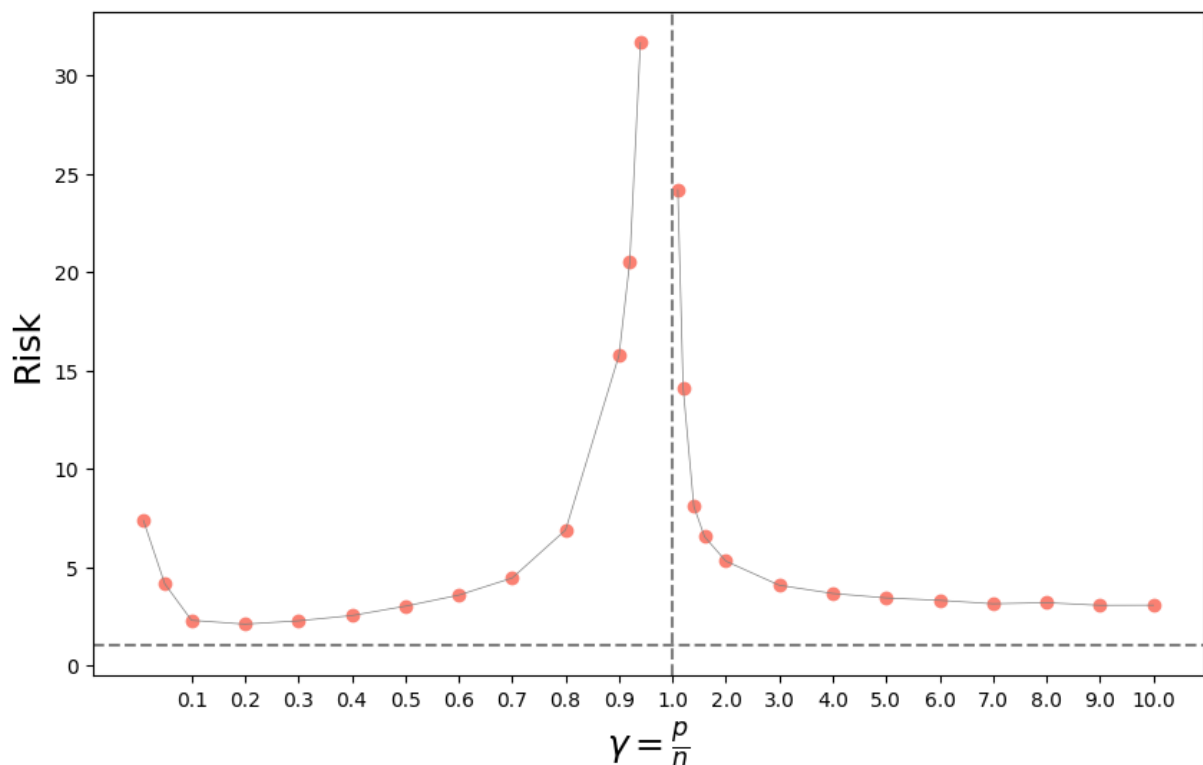
## Plot the risk

At this point, you can plot the risk by just evaluating the cell below. This should reveal the "double descent" behavior.

```

In [6]: # Just evaluate the next line
plot_double_descent_risk(gammas, risk, sigma)

```



### Problem 3.2

Comment on the results. Explain why the risk plot does or does not make sense in each regime: The underparameterized regime  $\gamma < 1$ , and the overparameterized regime  $\gamma > 1$ . Is the curve "U-shaped" in the underparameterized regime? Why or why not? What about in the overparameterized regime? You will be able to give better answers to these questions when you estimate the bias and variance below.

Answer:

The provided risk plot shows the "double descent" phenomenon, which extends the classical U-shaped bias-variance curve.

1. In the underparameterized regime ( $\gamma < 1$ ), the curve is U-shaped due to the standard bias-variance tradeoff: as model complexity increases, risk first falls by reducing bias, then rises due to high variance from overfitting.
2. The risk unexpectedly peaks at the interpolation threshold ( $\gamma = 1$ ), where the model solution is unique but extremely unstable.
3. In the overparameterized regime ( $\gamma > 1$ ), the risk descends again because the model can now fit the training data perfectly in many ways; it defaults to the minimum norm solution, which acts as implicit regularization, finding smoother and better-generalizing solutions as more features are added.

### Problem 3.3

Now, modify the above code so that you can estimate both the squared bias and the variance of the estimator. Before you do this, you may want to revisit the kernel smoothing demo from class, where we computed the squared bias, variance, and risk. You'll need the true function, which is provided in the variable `yf`. You should not have to write a lot of code, but can compute the bias and variance after you store the predicted values on the test data for each trial.

Plot the results, by plotting both the squared bias, the variance, and the risk for the sequence of gammas. To do this you will have to modify the plotting function appropriately, but this again involves minimal changes. When you obtain your final plot, comment on the shape of the bias and variance curves, as above for Problem 3.2.

```
In [7]: def plot_bias_variance_decomposition(gammas, risks, biases_squared, variance
        """
        Plots the bias-variance decomposition.

        Args:
            gammas (list): The gamma values (p/n).
            risks (list): The calculated risks for each gamma.
            biases_squared (list): The calculated squared biases.
            variances (list): The calculated variances.
        """
        plt.figure(figsize=(10, 6))
        plt.plot(gammas, biases_squared, label='Squared Bias', linewidth=2)
        plt.plot(gammas, variances, label='Variance', linewidth=2)
        plt.plot(gammas, risks, label='Risk', linewidth=2)
        plt.xlabel(r'$\gamma = p/n$')
        plt.ylabel('Risk, Bias2, Variance')
        plt.title('Bias-Variance Tradeoff for Random Features Model')
        plt.legend(loc='upper left')
        plt.grid(True, linestyle='--', alpha=0.6)
        plt.show()
```

```
In [8]: trials = 100
        n = 200
        gammas = [.01, .05] + list(np.arange(.1, 1, .1)) + [.92, .94, 1, 1.1, 1.2, 1.5]

        X_main, X_test, yf_main, yf_test = train_test_split(X, yf, test_size=1000)
        f_test_true = yf_test[:, 1]

        risks = []
        biases_squared = []
        variances = []

        for gamma in gammas:
            p = int(n * gamma)

            if gamma == 1:
                risks.append(np.inf)
                biases_squared.append(np.inf)
                variances.append(np.inf)
                continue
```

```

y_preds_across_trials = []

W = (1/np.sqrt(d)) * np.random.randn(d, p)
W[:, :min(p, p_star)] = W_star[:, :min(p, p_star)]
H_test = np.tanh(np.dot(X_test, W)) # Transform the FIXED test set featur

for i in np.arange(trials):
    X_train, _, yf_train, _ = train_test_split(X_main, yf_main, train_si

    H_train = np.tanh(np.dot(X_train, W))

    beta_hat = OLS_or_minimum_norm(H_train, yf_train[:,0])
    yhat_test = H_test @ beta_hat

    y_preds_across_trials.append(yhat_test)

y_preds_matrix = np.array(y_preds_across_trials)
avg_y_pred = np.mean(y_preds_matrix, axis=0)
bias_sq = np.mean((avg_y_pred - f_test_true)**2)
variance = np.mean(np.var(y_preds_matrix, axis=0))
risk = bias_sq + variance

print(f'gamma={gamma:.2f}  p={p:d}  n={n:d}  Risk={risk:.3f} (Bias²={bia

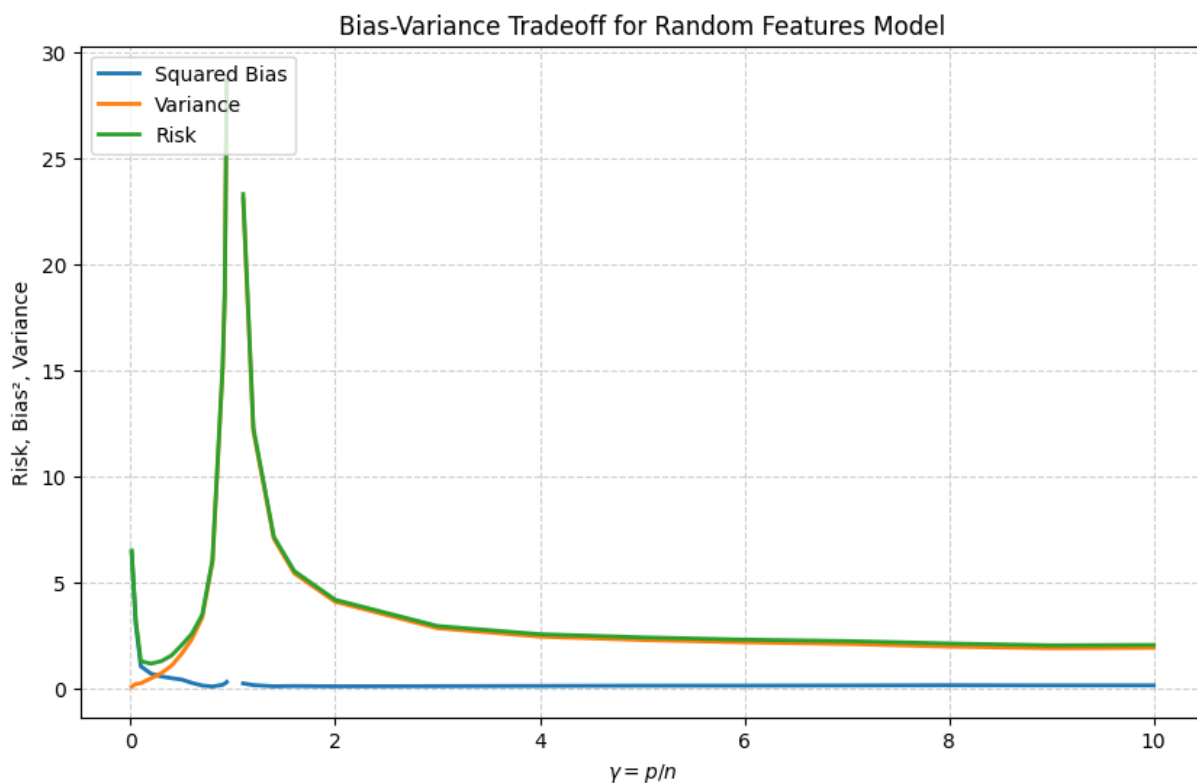
risks.append(risk)
biases_squared.append(bias_sq)
variances.append(variance)

plot_bias_variance_decomposition(gammas, risks, biases_squared, variances)

```



gamma=0.01	p=2	n=200	Risk=6.499 (Bias <sup>2</sup> =6.423, Var=0.076)
gamma=0.05	p=10	n=200	Risk=3.236 (Bias <sup>2</sup> =3.033, Var=0.203)
gamma=0.10	p=20	n=200	Risk=1.277 (Bias <sup>2</sup> =1.037, Var=0.240)
gamma=0.20	p=40	n=200	Risk=1.164 (Bias <sup>2</sup> =0.690, Var=0.474)
gamma=0.30	p=60	n=200	Risk=1.282 (Bias <sup>2</sup> =0.563, Var=0.719)
gamma=0.40	p=80	n=200	Risk=1.551 (Bias <sup>2</sup> =0.481, Var=1.070)
gamma=0.50	p=100	n=200	Risk=2.045 (Bias <sup>2</sup> =0.412, Var=1.633)
gamma=0.60	p=120	n=200	Risk=2.585 (Bias <sup>2</sup> =0.255, Var=2.330)
gamma=0.70	p=140	n=200	Risk=3.475 (Bias <sup>2</sup> =0.133, Var=3.342)
gamma=0.80	p=160	n=200	Risk=6.039 (Bias <sup>2</sup> =0.086, Var=5.953)
gamma=0.90	p=180	n=200	Risk=15.230 (Bias <sup>2</sup> =0.167, Var=15.062)
gamma=0.92	p=184	n=200	Risk=18.615 (Bias <sup>2</sup> =0.223, Var=18.392)
gamma=0.94	p=188	n=200	Risk=28.843 (Bias <sup>2</sup> =0.301, Var=28.542)
gamma=1.10	p=220	n=200	Risk=23.301 (Bias <sup>2</sup> =0.233, Var=23.068)
gamma=1.20	p=240	n=200	Risk=12.276 (Bias <sup>2</sup> =0.153, Var=12.123)
gamma=1.40	p=280	n=200	Risk=7.148 (Bias <sup>2</sup> =0.095, Var=7.053)
gamma=1.60	p=320	n=200	Risk=5.528 (Bias <sup>2</sup> =0.109, Var=5.418)
gamma=2.00	p=400	n=200	Risk=4.175 (Bias <sup>2</sup> =0.095, Var=4.080)
gamma=3.00	p=600	n=200	Risk=2.938 (Bias <sup>2</sup> =0.104, Var=2.835)
gamma=4.00	p=800	n=200	Risk=2.553 (Bias <sup>2</sup> =0.114, Var=2.439)
gamma=5.00	p=1000	n=200	Risk=2.404 (Bias <sup>2</sup> =0.125, Var=2.279)
gamma=6.00	p=1200	n=200	Risk=2.295 (Bias <sup>2</sup> =0.127, Var=2.168)
gamma=7.00	p=1400	n=200	Risk=2.230 (Bias <sup>2</sup> =0.136, Var=2.094)
gamma=8.00	p=1600	n=200	Risk=2.114 (Bias <sup>2</sup> =0.146, Var=1.969)
gamma=9.00	p=1800	n=200	Risk=2.027 (Bias <sup>2</sup> =0.139, Var=1.888)
gamma=10.00	p=2000	n=200	Risk=2.046 (Bias <sup>2</sup> =0.139, Var=1.906)



### Problem 3.4

In class, we discussed the interpretation of the minimum-norm estimator  $\hat{\beta}_{mn}$ .

Geometrically, we can describe  $\hat{\beta}_{mn}$  as the orthogonal projection of the zero vector in

$\mathbb{R}^p$  onto the  $(p - 1)$ -dimensional hyperplane  $\{\beta : X\beta = Y\}$ .

This can also be viewed as "ridgeless" regression. In ridge regression, we minimize the objective function

$$\|Y - X\beta\|_2^2 + \lambda\|\beta\|_2^2,$$

which has the closed-form solution

$$\hat{\beta}_\lambda = (X^T X + \lambda I)^{-1} X^T Y.$$

In the overparameterized regime where  $p > n$ , it can be shown that as  $\lambda \rightarrow 0$ ,  $\hat{\beta}_\lambda$  converges to  $\hat{\beta}_{\text{mn}}$ .

Your task is to show that as  $\lambda \rightarrow 0$ , the limit of the ridge regression estimator  $\hat{\beta}_\lambda$ , in the overparameterized regime where  $\gamma > 1$ , is the minimum-norm estimator  $\hat{\beta}_{\text{mn}}$ . You may want to use the Woodbury formula for this derivation.

*Hint:*

1. Applying the simplified version of Woodbury formula

$$(I + UV^T)^{-1} = I - U(I + V^T U)^{-1} V^T.$$

we can derive the identity:

$$(X^T X + \lambda I_p)^{-1} X^T = X^T (X X^T + \lambda I_n)^{-1},$$

2. You might consider using the Woodbury formula twice.

Answer:

the woodbury let:

$$(X^T X + \lambda I_p)^{-1} X^T = X^T (X X^T + \lambda I_n)^{-1}$$

according to the condition,  $\hat{\beta}_\lambda$  can be:

$$\hat{\beta}_\lambda = X^T (X X^T + \lambda I_n)^{-1} Y$$

when  $\lambda \rightarrow 0$ :

$$\lim_{\lambda \rightarrow 0} \hat{\beta}_\lambda = \lim_{\lambda \rightarrow 0} [X^T (X X^T + \lambda I_n)^{-1} Y]$$

rewrite:

$$\lim_{\lambda \rightarrow 0} \hat{\beta}_\lambda = X^T \left( \lim_{\lambda \rightarrow 0} (X X^T + \lambda I_n)^{-1} \right) Y$$

$$\lim_{\lambda \rightarrow 0} (XX^T + \lambda I_n)^{-1} = (XX^T + 0 \cdot I_n)^{-1} = (XX^T)^{-1}$$

so:

$$\lim_{\lambda \rightarrow 0} \hat{\beta}_\lambda = X^T (XX^T)^{-1} Y$$

and this is the definition of MNS:

$$\lim_{\lambda \rightarrow 0} \hat{\beta}_\lambda = \hat{\beta}_{\text{mn}}$$

proof complete