Intermediate Machine Learning: Assignment 2

Deadline

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

Directly sharing answers is not okay, but discussing problems with the course staff or with other students is encouraged.

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 file on Gradescope, and as a notebook (.ipynb) on Canvas. You can access Gradescope through Canvas on the left-side of the class home page. The problems in each homework assignment are numbered. 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.

To produce the .pdf, please do the following in order to preserve the cell structure of the notebook:

Go to "File" at the top-left of your Jupyter Notebook Under "Download as", select "HTML (.html)" After the .html has downloaded, open it and then select "File" and "Print" (note you will not actually be printing) From the print window, select the option to save as a .pdf

Topics

- Convolutional neural networks
- Gaussian processes
- Double descent

This assignment will also help to solidify your Python and Jupyter notebook skills.

Problem 1: It's not a bug, it's a feature! (20 points)

In this problem, we will "open the black box" and inspect the filters and feature maps learned by a convolutional neural network trained to classify handwritten digits, using the MNIST

database.

In [ ]:
```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import random
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

1.1 Visualizing the filters

To begin, we load the dataset with 60000 training images and 10000 test images.

In [ ]:
```
num_classes = 10
input_shape = (28, 28, 1)

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")


# convert class vectors to binary class matrices
y_train_binary = keras.utils.to_categorical(y_train, num_classes)
y_test_binary = keras.utils.to_categorical(y_test, num_classes)
```

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

Next, we initialize our convolutional neural network.

In [ ]:
```
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(5, 5), activation="relu", name='conv1'),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(32, kernel_size=(5, 5), activation="relu", name='conv2'),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)
```

```
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	
conv1 (Conv2D)	(None, 24, 24, 32)	
max_pooling2d_2 (MaxPooling2D)	(None, 12, 12, 32)	
conv2 (Conv2D)	(None, 8, 8, 32)	
max_pooling2d_3 (MaxPooling2D)	(None, 4, 4, 32)	
flatten_1 (Flatten)	(None, 512)	
dropout_1 (Dropout)	(None, 512)	
dense_1 (Dense)	(None, 10)	

Total params: 31,594 (123.41 KB)

Trainable params: 31,594 (123.41 KB)

Non-trainable params: 0 (0.00 B)

In [ ]:
```
batch_size = 128
epochs = 1

model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy

model.fit(x_train, y_train_binary, batch_size=batch_size, epochs=epochs, validation
```

```
422/422 ━━━━━━━━━━━━━━━━━━━━ 49s 112ms/step - accuracy: 0.7384 - loss: 0.8324 - val_
accuracy: 0.9785 - val_loss: 0.0794
```

Out[ ]:
```
<keras.src.callbacks.history.History at 0x7e8269876770>
```

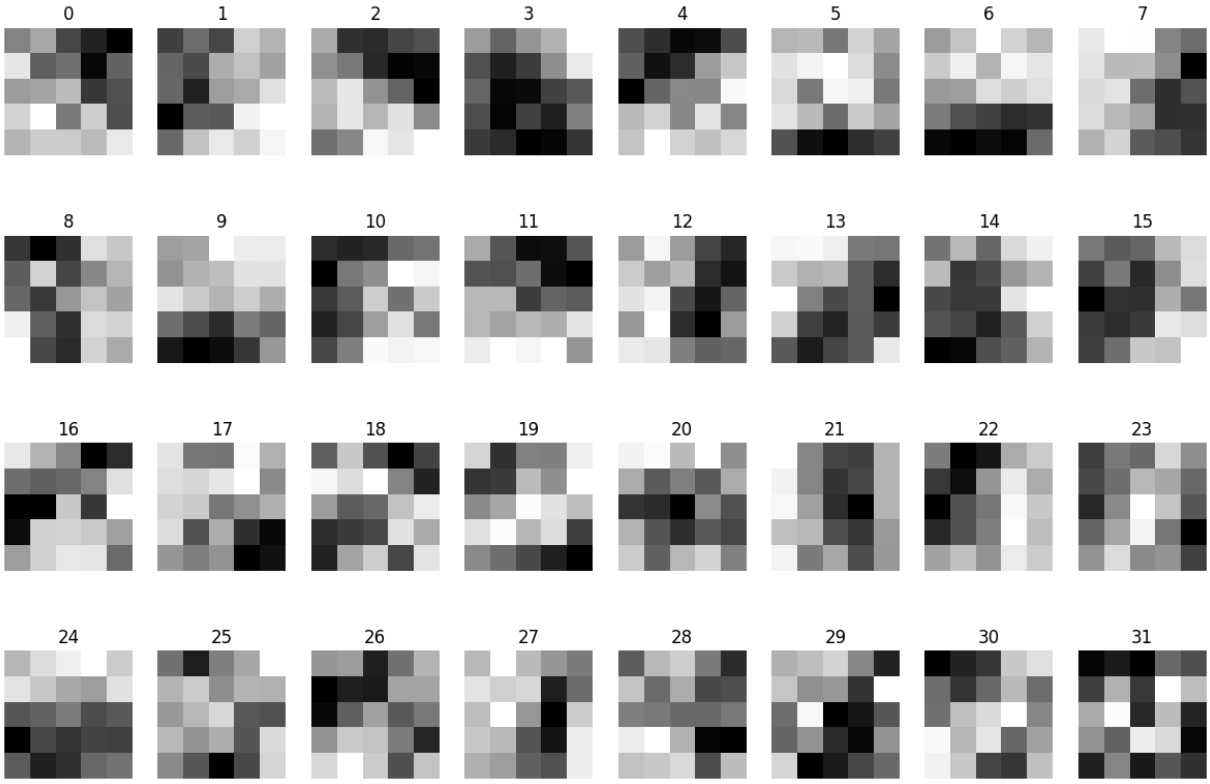
In [ ]:
```
score = model.evaluate(x_test, y_test_binary, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
Test loss: 0.08754748851060867
Test accuracy: 0.9747999906539917
```

Now that we've trained and tested the model, let's look at the filters learned in the first convolutional layer.

In [ ]:
```
filters_conv1 = model.get_layer(name='conv1').get_weights()[0]

fig, axs = plt.subplots(4, 8)
fig.set_figheight(10)
fig.set_figwidth(15)

for i in range(4):
    for j in range(8):
        f = filters_conv1[:, :, 0, 8*i+j]
        axs[i, j].imshow(f[:, :], cmap='gray')
```

```
        axs[i, j].axis('off')
        axs[i, j].set_title(8*i+j)
```



Describe what you see. Do (some of) the learned filters make sense to you?

Hint: Many filters have been designed and widely applied in image processing. Here are some examples of edge detection filters and their effect on the image. You can find the details about each filter by clicking the links at the bottom.

Almost all of the filters appear to be split between a contiguous region of high parameter value (dark), and a contiguous region of low parameter value (light). In many filters, the boundary between light and dark is linear, showing they act to detect straight edges of various angles. This makes sense, because the location of edges contains significant information about the digit depicted.

1.2 Visualizing the feature maps

We can also look at the corresponding feature map for each filter. There are 32 kernels at the first convolutional layer, so there are 32 feature maps for each sample. feature_map_conv1 is a 4D matrix where the first dimension is the index of the sample and the last dimension is the index of the correpsonding filter.

In [ ]:
```
conv1_layer_model = keras.Model(inputs=model.inputs, outputs=model.get_layer('conv1
feature_map_conv1 = conv1_layer_model(x_test)
```

Randomly draw 16 samples for visualization.

In [ ]:
```
sample_index = random.sample(range(1, len(x_test)), 16)
```

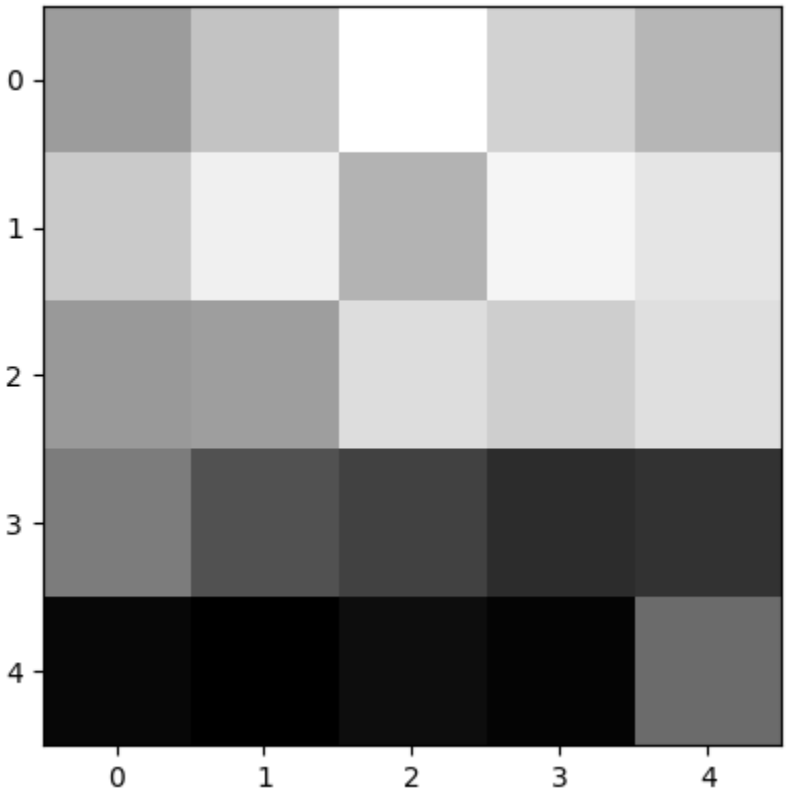
Choose two filters among all 32 filters from 2.1, and visualize their feature maps.

In [ ]:
```
filter_n1 = 6
filter_n2 = 13
```

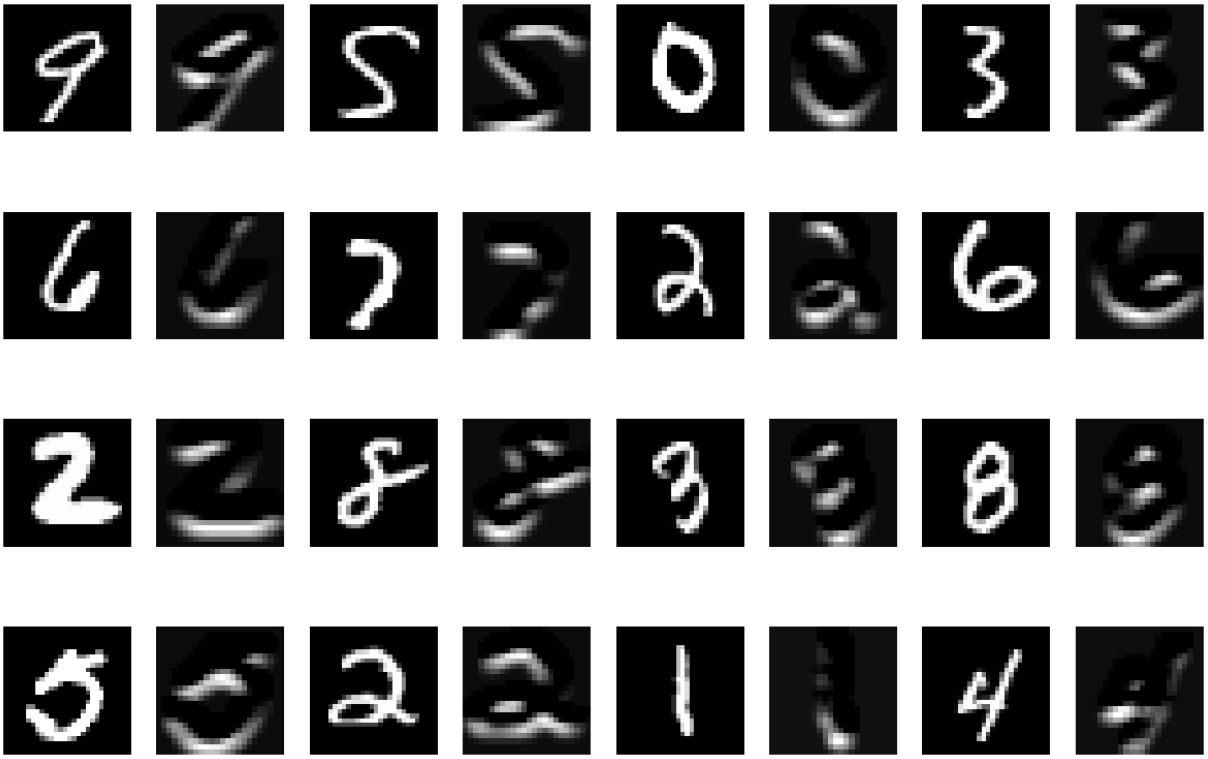
There is no need to modify the next code cells, just run the four cells below.

In [ ]:
```
plt.imshow(filters_conv1[:, :, 0, filter_n1], cmap='gray')
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7e82697bf250>
```

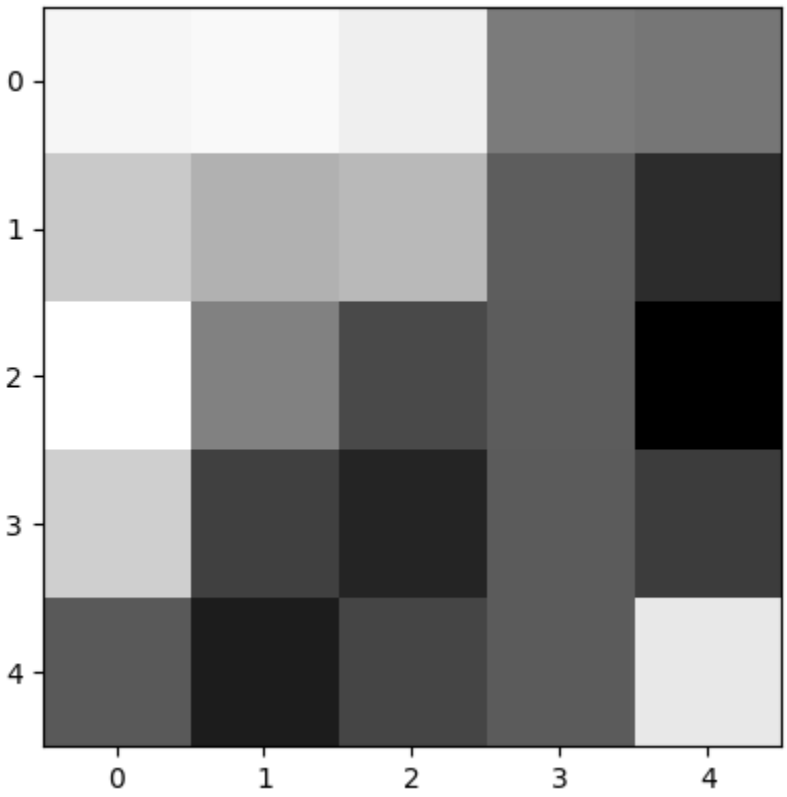


In [ ]:
```
fig, axs = plt.subplots(4, 8)
fig.set_figheight(10)
fig.set_figwidth(15)

ix=0
for i in range(4):
    for j in range(4):
        axs[i, 2*j].imshow(x_test[sample_index[4*i+j], :, :, 0], cmap='gray')
        axs[i, 2*j].axis('off')
        axs[i, 2*j+1].imshow(feature_map_conv1[sample_index[4*i+j], :, :, filter_n1
        axs[i, 2*j+1].axis('off')
```



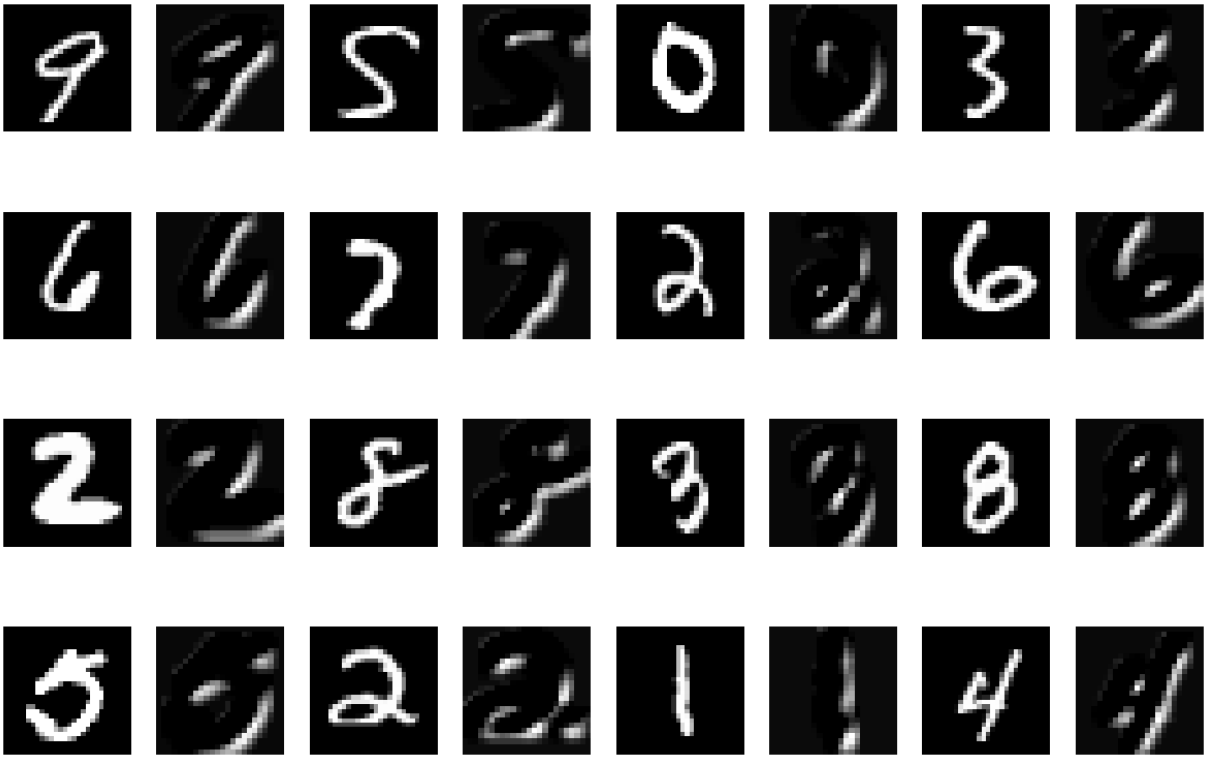
In [ ]:
```
plt.imshow(filters_conv1[:, :, 0, filter_n2], cmap='gray')
```

Out[ ]: <matplotlib.image.AxesImage at 0x7e82685ff790>



In [ ]:
```
fig, axs = plt.subplots(4, 8)
fig.set_figheight(10)
fig.set_figwidth(15)
```

```
ix=0
for i in range(4):
    for j in range(4):
        axs[i, 2*j].imshow(x_test[sample_index[4*i+j], :, :, 0], cmap='gray')
        axs[i, 2*j].axis('off')
        axs[i, 2*j+1].imshow(feature_map_conv1[sample_index[4*i+j], :, :, filter_n2
        axs[i, 2*j+1].axis('off')
```



Comment on what you see in the feature maps.

- How do they correspond to the original images?
- How do they correspond to the filters?
- Why might the feature maps be helpful for classifying digits?

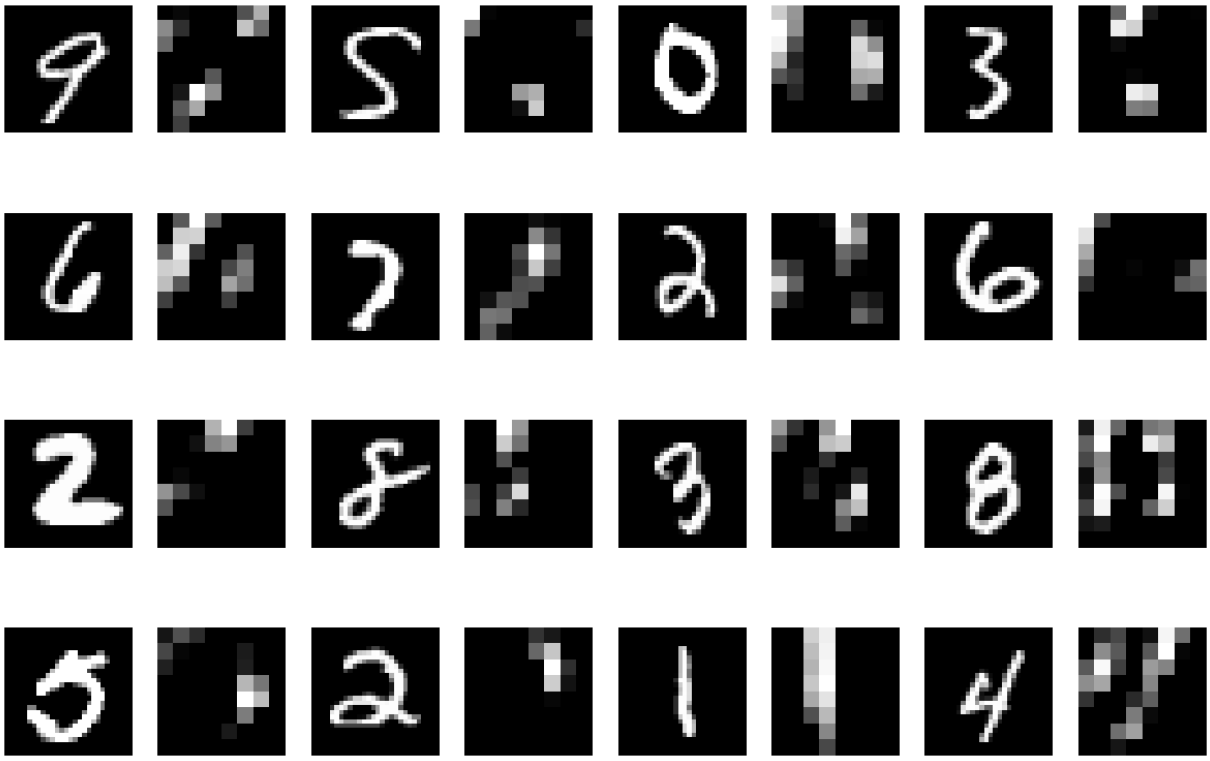
Each filter acts as an edge detector. Filter 1 acts to detect horizontal edges, and filter 2 acts to detect slanted up (/) edges. This causes the observed affect where the feature maps pronounces the part of the digit that is that edge type. This helps classify images, because "1" for example has very few horizontal edges, so the horizontal edge detector applied to it produces a distinguishably sparse feature map.

1.3 Fitting a logistic regression model on feature maps

The features of the images are further summarized after the second convolutional layer.

In [ ]:
```
conv2_layer_model = keras.Model(inputs=model.inputs, outputs=model.get_layer('conv2
feature_map_conv2 = conv2_layer_model(x_test)

fig, axs = plt.subplots(4, 8)
fig.set_figheight(10)
```

```
fig.set_figwidth(15)

ix=0
for i in range(4):
    for j in range(4):
        axs[i, 2*j].imshow(x_test[sample_index[4*i+j], :, :, 0], cmap='gray')
        axs[i, 2*j].axis('off')
        axs[i, 2*j+1].imshow(feature_map_conv2[sample_index[4*i+j], :, :, 0], cmap=
        axs[i, 2*j+1].axis('off')
```



Build and test a logistic regression model to classify two digits of your choice (i.e. a binary classification) using the features maps at the second convolutional layer as the input. You may use logistic regression functions such as LogisticRegression in sklearn. Use 80% of the data for training and 20% for test.

- How many features are there in your input X? Show the derivation of this number based on the architecture of the convolutional neural network.
- How is your logistic regression model related to the fully connected layer and softmax layer in the convolutional neural network?
- What is the accuracy of your model? Is this expected, or surprising?
- Comment on any other aspects of your findings that are interesting to you.

In [ ]:
```
X_lr = np.reshape(feature_map_conv2,(np.shape(feature_map_conv2)[0],-1))
y_lr = y_test
```

In [ ]:
```
# I choose 6 and 7 as the digits to classify.
```

```
digit_categories = [6, 7]

# pull out images of 6s and 7s
indicies = np.where((y_lr==digit_categories[0]) | (y_lr==digit_categories[1]))
X_lr_subset = X_lr[indicies]
y_lr_subset = y_lr[indicies]

# coding labels
y_lr_subset[y_lr_subset==digit_categories[0]] = 0
y_lr_subset[y_lr_subset==digit_categories[1]] = 1

# split data into test and train
X_lr_train, X_lr_test, y_lr_train, y_lr_test = train_test_split(X_lr_subset, y_lr_s
```

In [ ]:
```
lr_model = LogisticRegression()
lr_model.fit(X_lr_train, y_lr_train)
y_lr_pred = lr_model.predict(X_lr_test)
```

In [ ]:
```
np.mean(y_lr_pred == y_lr_test)
```

Out[ ]: 1.0

- There are 2048 features in each x. Input data to the CNN has dimension 28x28x1. After a 32 5x5x1 filters, this is 24x24x32. After another 32 5x5x32 filters, this is 8x8x32. That, flattened is 2048 parameters.
- In general, the mathematical form of the logistic regression model is the same that induced by a fully connected plus softmax layer, which is to say the probability of a class label is modeled to be the softmax of logits. In the case of this specific example, the full CNN has an additional pooling layer and classifies into 10 categories vs. the logistic regression model which does not and only classifies into 2 categories.
- The model differentiates between the digits 6 and 7 with 100% accuracy. Its a bit suprising to me that it didn't make a single mistake, but not super suprising because 6 and 7 are very different looking digits (that's why I choose those two in particular as categories!)
- When I tried training on the digits 1 and 7 there was a lower prediction accuracy of 99.77%, which makes sense because those digits are much easier to confuse

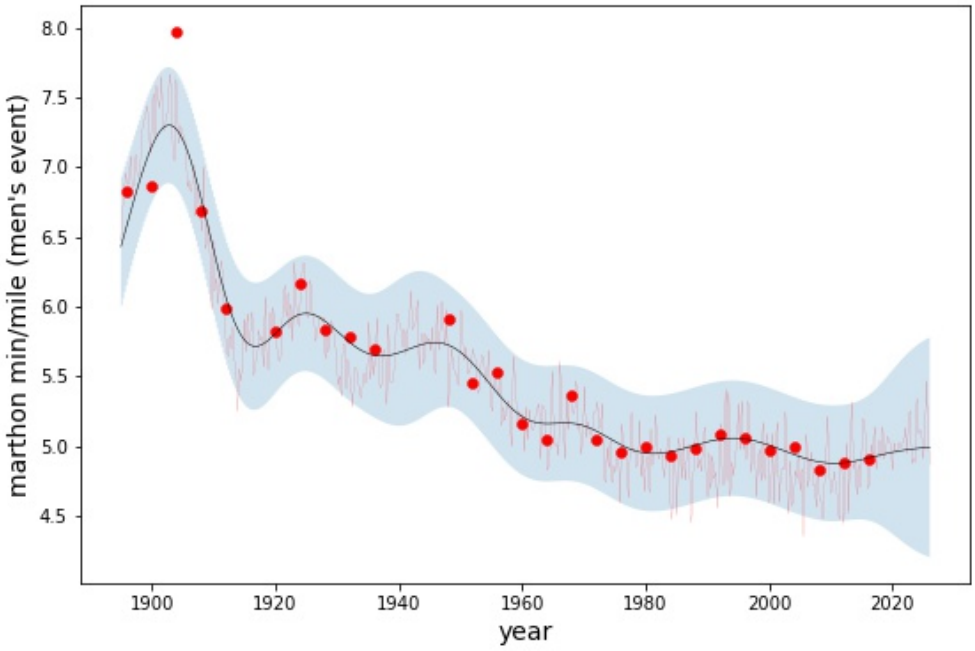
Problem 2: All that glitters (20 points)

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, σ^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

dat = pd.read_csv('https://raw.githubusercontent.com/YData123/sds365-sp22/main/demo
events = set(np.array(dat['Event']))
print(events)
```

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

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

In [ ]:
```
marathon = dat[dat['Event'] == 'Marathon Men']
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)
    minutes_per_mile = (t[0]*60*60 + t[1]*60 + t[2])/(60*26.2)
    mpm.append(minutes_per_mile)

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
```

Out[ ]:

	Location	Year	Medal	Name	Nationality	Result	Minutes per Mile
0	Athens	1896	G	Spyridon LOUIS	GRE	2:58:50	6.83
1	Paris	1900	G	Michel THÃATO	FRA	2:59:45.0	6.86
2	St Louis	1904	G	Thomas HICKS	USA	3:28:53.0	7.97
3	London	1908	G	John HAYES	USA	2:55:18.4	6.69
4	Stockholm	1912	G	Kennedy Kane MCARTHUR	RSA	2:36:54.8	5.99
5	Antwerp	1920	G	Hannes KOLEHMAINEN	FIN	2:32:35.8	5.82
6	Paris	1924	G	Albin STENROOS	FIN	2:41:22.6	6.16
7	Amsterdam	1928	G	BoughÃ¨ra EL OUAFI	FRA	2:32:57	5.84
8	Los Angeles	1932	G	Juan Carlos ZABALA	ARG	2:31:36	5.79
9	Berlin	1936	G	Kitei SON	JPN	2:29:19.2	5.70
10	London	1948	G	Delfo CABRERA	ARG	2:34:51.6	5.91
11	Helsinki	1952	G	Emil ZÃTOPEK	TCH	2:23:03.2	5.46
12	Melbourne / Stockholm	1956	G	Alain MIMOUN	FRA	2:25:00	5.53
13	Rome	1960	G	Abebe BIKILA	ETH	2:15:16.2	5.16
14	Tokyo	1964	G	Abebe BIKILA	ETH	2:12:11.2	5.05
15	Mexico	1968	G	Mamo WOLDE	ETH	2:20:26.4	5.36
16	Munich	1972	G	Frank Charles SHORTER	USA	2:12:19.8	5.05
17	Montreal	1976	G	Waldemar CIERPINSKI	GDR	2:09:55.0	4.96
18	Moscow	1980	G	Waldemar CIERPINSKI	GDR	2:11:03.0	5.00
19	Los Angeles	1984	G	Carlos LOPES	POR	2:09:21	4.94
20	Seoul	1988	G	Gelindo BORDIN	ITA	2:10:32	4.98
21	Barcelona	1992	G	Young-Cho HWANG	KOR	2:13:23	5.09
22	Atlanta	1996	G	Josia THUGWANE	RSA	2:12:36	5.06
23	Sydney	2000	G	Gezahegne ABERA	ETH	2:10:11	4.97

	Location	Year	Medal	Name	Nationality	Result	Minutes per Mile
24	Athens	2004	G	Stefano BALDINI	ITA	2:10:55	5.00
25	Beijing	2008	G	Samuel Kamau WANJIRU	KEN	2:06:32	4.83
26	London	2012	G	Stephen KIPROTICH	UGA	2:08:01	4.89
27	Rio	2016	G	Eliud Kipchoge ROTICH	KEN	02:08:44	4.91

Enter your code and markdown following this cell.

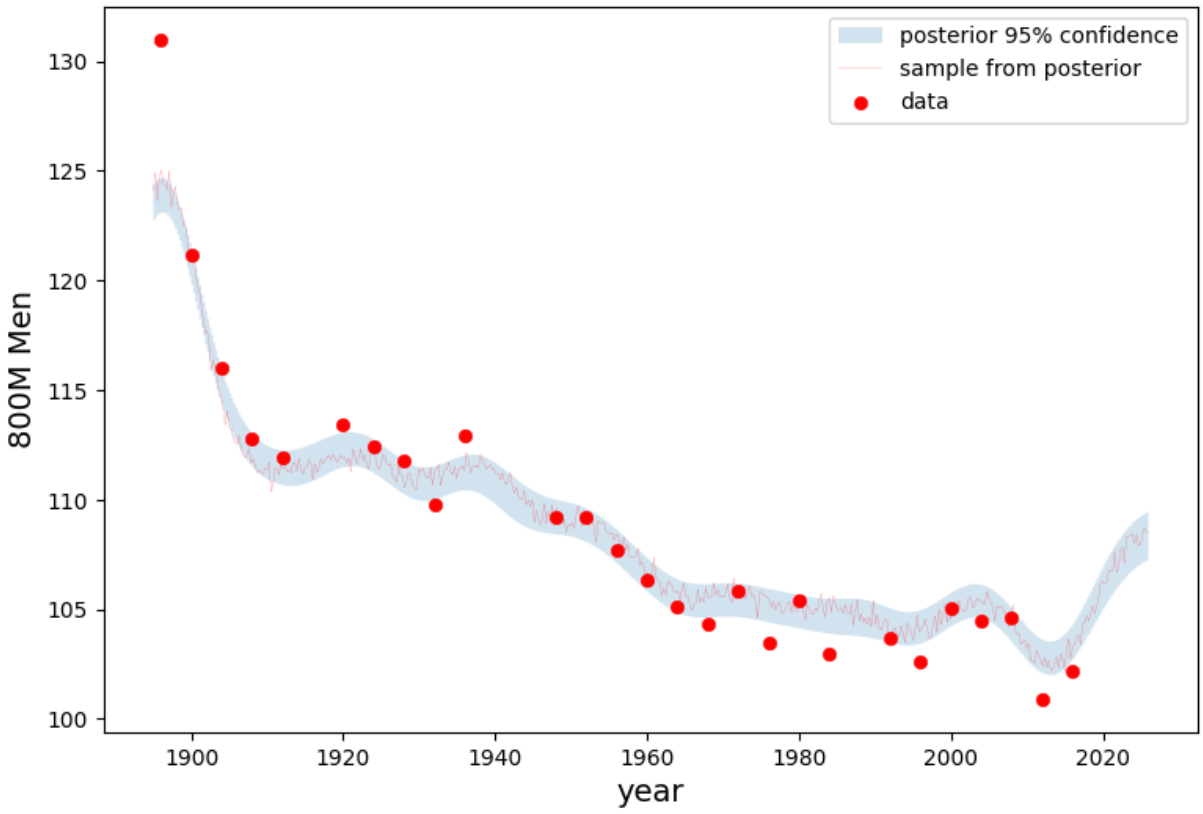
In [ ]:
```
def gaussian_sample(mu, Sigma):
  rng = np.random.default_rng()
  return rng.multivariate_normal(mu, Sigma, method='cholesky')

def mu_fun(x, mu=0):
  return mu * np.ones(len(x))

def K_fun(x, z, h=1):
  K = np.zeros(len(x)*len(z)).reshape(len(x), len(z))
  for j in np.arange(K.shape[1]):
    K[:, j] = (1/h) * np.exp(-(x-z[j])**2/(2*h**2))
  return K

def min_time_to_sec(time):
  t = np.array(time.split(':'), dtype=float)
  return (t[0]*60 + t[1])
```

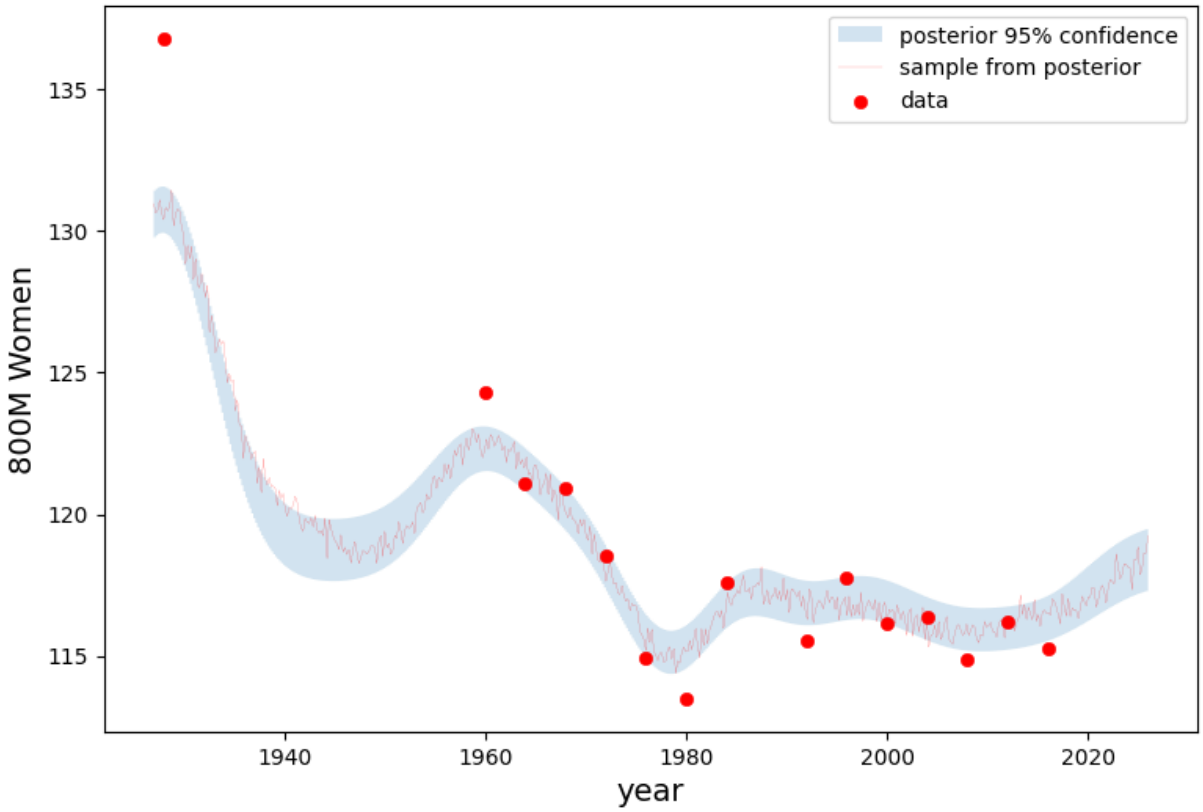
In [ ]:
```
def gp_olympic_event(year, result, kernel, mean, noise, event_name):
  Xtrain = year
  ytrain = result

  # calculate posterior
  xs = np.linspace(min(year)-1, max(year)+10, 500)

  K = kernel(Xtrain, Xtrain)
  Ks = kernel(Xtrain, xs)
  Kss = kernel(xs, xs) + noise * np.eye(len(xs))
  Ki = np.linalg.inv(K + noise * np.eye(len(Xtrain)))

  postMu = mean(xs) + Ks.T @ Ki @ (ytrain - mean(Xtrain))
  postCov = Kss - Ks.T @ Ki @ Ks

  # plot posterior
  fig, ax = plt.subplots(1, 1, figsize=(9, 6))
  S2 = np.diag(postCov)
  ax.fill_between(xs, postMu - 2*np.sqrt(S2), postMu + 2*np.sqrt(S2),
                  step="pre", alpha=0.2, label="posterior 95% confidence")
  ys = gaussian_sample(postMu, postCov)
```

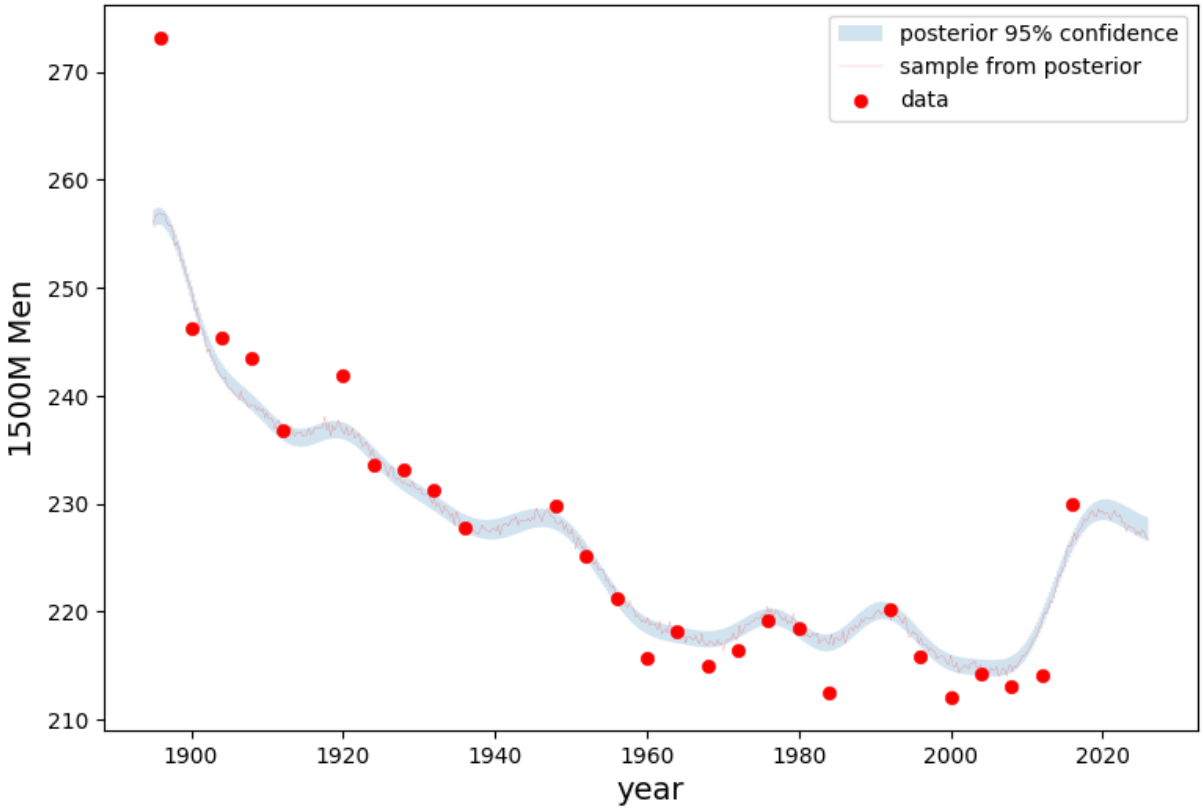
```
  ax.plot(xs, ys, c='r', linestyle='-', linewidth=0.1, label='sample from posterior
  ax.scatter(Xtrain, ytrain, c='red', marker='o', linewidth=0.4, label='data')
  ax.set_xlabel('year', fontsize=14)
  ax.set_ylabel(event_name, fontsize=14)
  ax.legend()
```

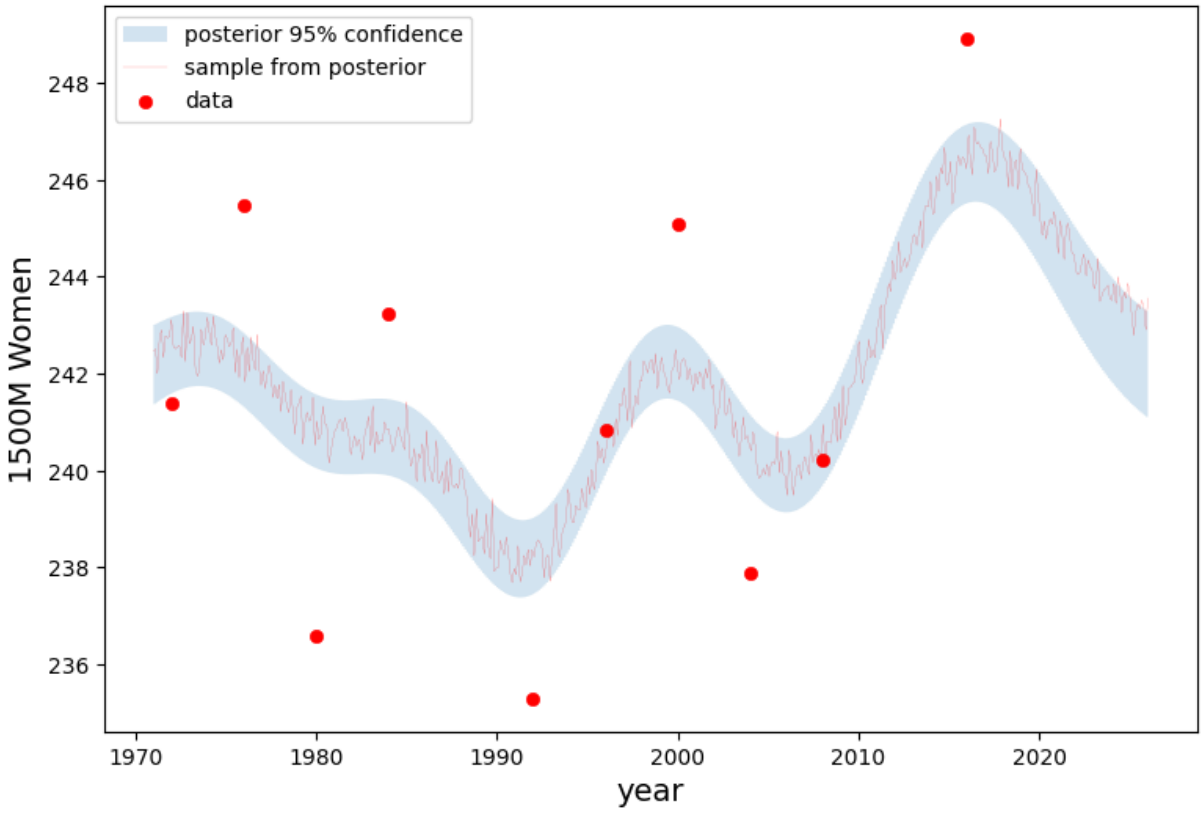
In [ ]:

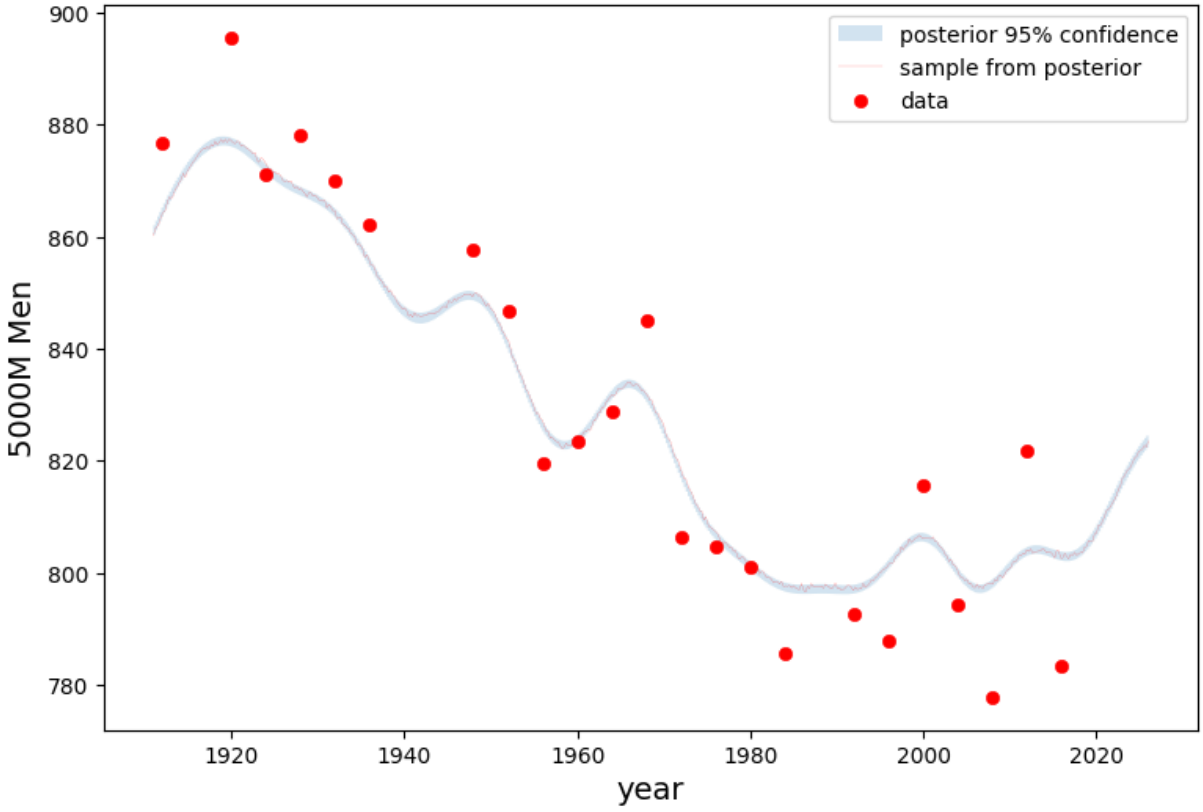
```
event_names = ['800M Men', '800M Women', '1500M Men', '1500M Women', '5000M Men', '
h =  5          # bandwidth

for event_name in event_names:
# event_name = event_names[0]

  event_data = dat[dat['Event'] == event_name]
  event_data = event_data[event_data['Medal']=='G']

  years = np.array(event_data['Year'])
  results = np.vectorize(min_time_to_sec)(np.array(event_data['Result']))
  def kernel(x, z): return K_fun(x, z, h=h)
  def mean(x): return mu_fun(x, mu=np.mean(results))
  noise = 0.1

  gp_olympic_event(years, results, kernel, mean, noise, event_name)
```

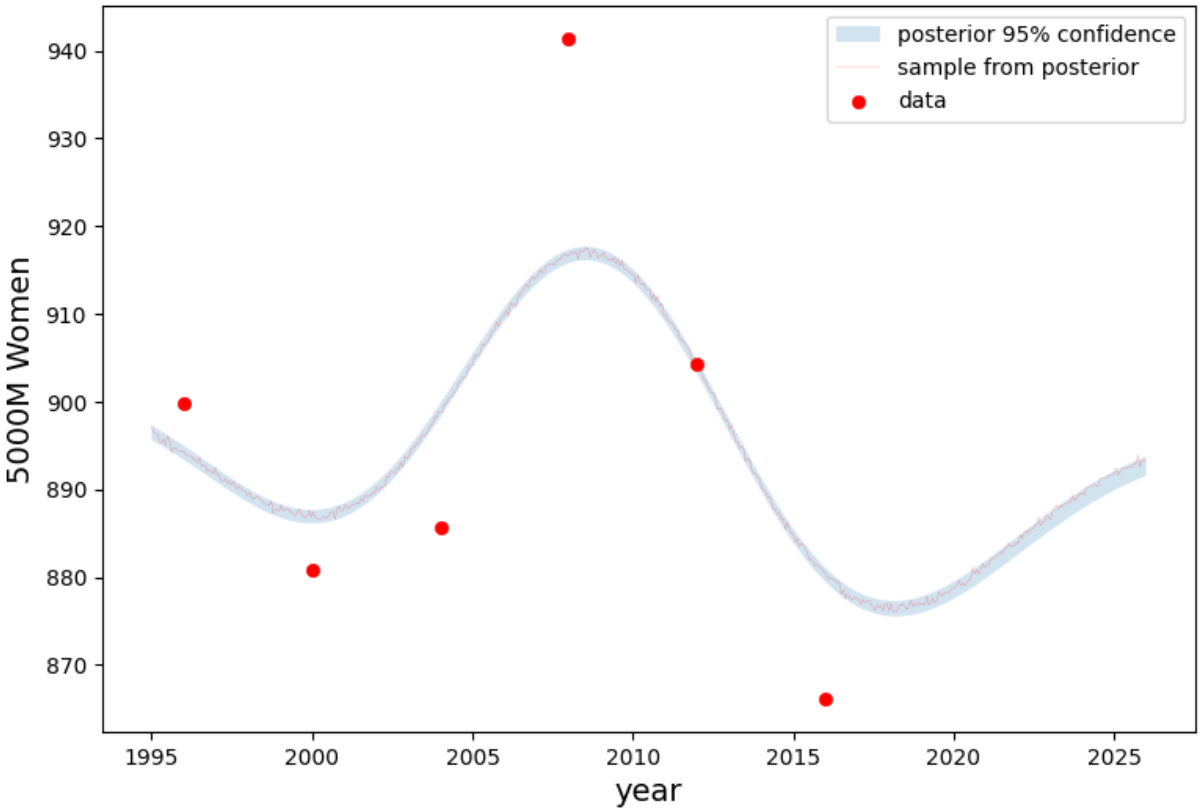



posterior 95% confidence
sample from posterior
data
800M Women
135
130
125
120
115
1940
1960
1980
2000
2020
year


posterior 95% confidence
sample from posterior
data
1500M Men
270
260
250
240
230
220
210
1900
1920
1940
1960
1980
2000
2020
year


posterior 95% confidence
sample from posterior
data
1500M Women
248
246
244
242
240
238
236
1970
1980
1990
2000
2010
2020
year


posterior 95% confidence
sample from posterior
data
5000M Men
900
880
860
840
820
800
780
1920
1940
1960
1980
2000
2020
year



Models Overall:

Because of key similarities across all 6 running events I employed the same method for choosing the prior kernel, mean, and error for all of them.

- I choose a gaussian kernel (due to its computational simplicity) with a bandwidth of 5 (somewhat arbitrarily, but after sampling the posterior, I'm satisfied with the general level of smoothness it induces). I'm using the same bandwidth for all events because I have no reason to believe any one event has a less smooth function model than any other.
- For the measurement error I choose 0.1s. This results from timing error which I expect to be the same across all races. Races 50+ years ago probably have a timing error of an order of magnitude higher and current races probably have a timing error of an order of magnitude less, so if I'm going to pick a uniform timing error hyperparameter, 0.1s feels reasonable.
- For the mean, I supposed that for all races, there hasn't been any tendency for faster times over time, and thus I set the "prior" mean to be the mean of the data for each race. Technically this isn't actually a prior, because I used the data to determine the prior, but I think this is reasonable in that it captures a reasonable mean.

Other general observations:

- Across the models, when data is sparse (like during WWII or in making future predictions), there is a strong tendency to revert to the prior mean

Model 1: 800M Men

The model appears to fit the data relatively well. The race times over the years seem to be decreasing, so maybe the prior assumption of constant mean was not fair

Model 2: 800M Women

This model has a very significant outlier in the 1930s. This seems to be because there is no data between the 1930s and the 1960s, and during that data less stretch, the model has a strong tendency to rever to the mean which is much lower than the 1930s data point.

Model 3: 1500M Men

The model seems to fit the data relatively well. There are extreme outliers. One in 1896 and another in 2016. The 1896 games were the first modern games, so its not shocking to me that they'd be an outlier.

Model 4: 1500M Women

This model fit is interesting, because almost every data point is outside the confidence interval. There seems to be more significant variation in times than for any other race. Additionaly there are significantly few data points, I'm assuming because the women's 1500M event was more recently added to the olympics

Model 5: 5000M Men

The model here does not seem to be a good fit for the data. The model shows a tendency towards the prior constant mean, but the data seems to show a significant decrease in times over the years. This suggests the prior mean assumption is wrong, and causing poor model fit to the data.

Model 6: 5000M Women

There is very little data for this event, as it seems to have been introduced as recently at 1996. Additionally, the data shows significant variation, much like the 1500M women's times. Despite all of the data points lying outside the confidence interval, I think the model itself is reasonable, besided a major source of variance that's not being account for.

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 γ.
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 $\lambda \to 0$, 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 [9]:
```
def OLS_or_minimum_norm(X, y):
  n = X.shape[0]
  p = X.shape[1]
  gamma = p / n

  if gamma < 1:
    # OLS solution
    beta_hat = np.linalg.inv(X.T @ X) @ X.T @ y
  else:
    # minimum norm solution
    beta_hat = X.T @ np.linalg.inv(X @ X.T) @ y

  return beta_hat
```

In [10]:
```
# A plotting function we provide. No need to change this, although you can if you'd

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 >= 2 or ga
    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 [11]:
```
# 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 γ

Next, we train a sequence of models for different values of γ, 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 γ, and for each γ 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.

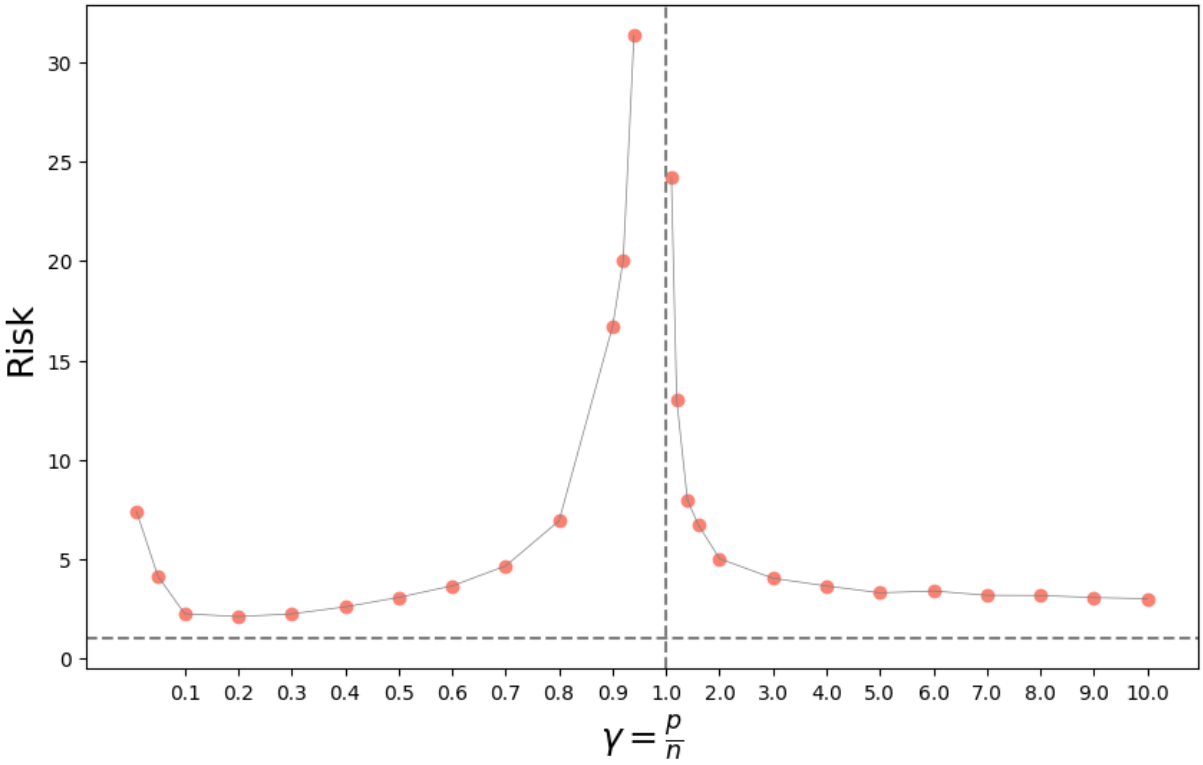
In [20]:
```
trials = 100
n = 200

gammas = list(np.arange(.1, 1, .1)) + [.92, .94, 1, 1.1, 1.2, 1.4, 1.6] + list(np.a
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_size=n,
        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=7.402
gamma=0.05  p=10  n=200  risk=4.145
gamma=0.10  p=20  n=200  risk=2.259
gamma=0.20  p=40  n=200  risk=2.114
gamma=0.30  p=60  n=200  risk=2.251
gamma=0.40  p=80  n=200  risk=2.602
gamma=0.50  p=100  n=200  risk=3.080
gamma=0.60  p=120  n=200  risk=3.658
gamma=0.70  p=140  n=200  risk=4.659
gamma=0.80  p=160  n=200  risk=6.934
gamma=0.90  p=180  n=200  risk=16.697
gamma=0.92  p=184  n=200  risk=20.048
gamma=0.94  p=188  n=200  risk=31.335
gamma=1.10  p=220  n=200  risk=24.229
gamma=1.20  p=240  n=200  risk=13.009
gamma=1.40  p=280  n=200  risk=7.983
gamma=1.60  p=320  n=200  risk=6.746
gamma=2.00  p=400  n=200  risk=5.028
gamma=3.00  p=600  n=200  risk=4.042
gamma=4.00  p=800  n=200  risk=3.656
gamma=5.00  p=1000  n=200  risk=3.310
gamma=6.00  p=1200  n=200  risk=3.402
gamma=7.00  p=1400  n=200  risk=3.187
gamma=8.00  p=1600  n=200  risk=3.175
gamma=9.00  p=1800  n=200  risk=3.074
gamma=10.00  p=2000  n=200  risk=3.013
```

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 [21]:
```
# 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.

The risk for a squared error loss function is bias-squared plus variance. In the underparameterized regime, the risk curve is the u shape I've come to expect. When there are too few parameters (gamma < 0.1) to accurately model the true function, the bias is high, and when there are too many parameters (1 > gamma > 0.3), estimating each parameter introduces more variance. The competition between these factors manifests in a u-shaped risk curve in the underparameterized regime. In the overparamterized regime, the risk appears to decrease monotonically. This is a suprise because I'd assume that with many parameters, the model would overfit leading to high variance. We learned in class that the random parameters cause the input features to the final layer to also have significant randomness, and that randomness activates some type of central limit theorm affect which prevents overfitting.

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 [48]:
```
trials = 100
n = 200

gammas = list(np.arange(.1, 1, .1)) + [.92, .94, 1, 1.1, 1.2, 1.4, 1.6] + list(np.a
gammas = [.01, .05] + gammas

risk = []
bias = []
variance = []
irreducible_error = sigma**2

# seperate a fixed test set
test_size = 1000
X_train_pool, X_test, yf_train_pool, yf_test = train_test_split(X, yf, test_size=te


for gamma in gammas:
    trial_yminusfhat2 = np.empty((trials, test_size))      # (y-fhat)^2  evaluated
    trial_fhatminusf =  np.empty((trials, test_size))      # (fhat-f)    evaluated
    trial_fhat =        np.empty((trials, test_size))      # fhat        evaluated

    p = int(n * gamma)
    if gamma == 1:
        risk.append(np.inf)
        bias.append(np.inf)
        variance.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, _, yf_train, _ = train_test_split(X_train_pool, yf_train_pool, tra
        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

        trial_yminusfhat2[i, :] = (yf_test[:,0] - yhat_test)**2
        trial_fhatminusf[i, :]  = (yhat_test - yf_test[:, 1])
        trial_fhat[i, :] = yhat_test


    gamma_risk = np.mean(np.mean(trial_yminusfhat2, axis=0))
    gamma_bias = np.mean(np.mean(trial_fhatminusf, axis=0)**2)
```

```
    gamma_variance = np.mean(np.mean((trial_fhat - np.mean(trial_fhat, axis=0))**2,

    print('gamma=%.2f  p=%d  n=%d  risk=%.3f  bias=%.3f  var=%.3f  irreducible_erro
          % (gamma, p, n, gamma_risk, gamma_bias, gamma_variance, irreducible_error

    risk.append(gamma_risk)
    bias.append(gamma_bias)
    variance.append(gamma_variance)
```

```
gamma=0.01  p=2  n=200  risk=7.305  bias=6.305  var=0.084  irreducible_error=1.000
gamma=0.05  p=10  n=200  risk=3.799  bias=2.692  var=0.186  irreducible_error=1.000
gamma=0.10  p=20  n=200  risk=2.173  bias=0.970  var=0.254  irreducible_error=1.000
gamma=0.20  p=40  n=200  risk=2.033  bias=0.593  var=0.460  irreducible_error=1.000
gamma=0.30  p=60  n=200  risk=2.214  bias=0.509  var=0.754  irreducible_error=1.000
gamma=0.40  p=80  n=200  risk=2.512  bias=0.413  var=1.101  irreducible_error=1.000
gamma=0.50  p=100  n=200  risk=2.973  bias=0.338  var=1.647  irreducible_error=1.000
gamma=0.60  p=120  n=200  risk=3.586  bias=0.235  var=2.348  irreducible_error=1.000
gamma=0.70  p=140  n=200  risk=4.495  bias=0.118  var=3.374  irreducible_error=1.000
gamma=0.80  p=160  n=200  risk=6.710  bias=0.095  var=5.615  irreducible_error=1.000
gamma=0.90  p=180  n=200  risk=18.746  bias=0.223  var=17.488  irreducible_error=1.0
00
gamma=0.92  p=184  n=200  risk=20.617  bias=0.277  var=19.349  irreducible_error=1.0
00
gamma=0.94  p=188  n=200  risk=29.719  bias=0.328  var=28.427  irreducible_error=1.0
00
gamma=1.10  p=220  n=200  risk=25.128  bias=0.284  var=23.853  irreducible_error=1.0
00
gamma=1.20  p=240  n=200  risk=13.322  bias=0.157  var=12.186  irreducible_error=1.0
00
gamma=1.40  p=280  n=200  risk=8.557  bias=0.110  var=7.453  irreducible_error=1.000
gamma=1.60  p=320  n=200  risk=6.544  bias=0.108  var=5.489  irreducible_error=1.000
gamma=2.00  p=400  n=200  risk=5.071  bias=0.092  var=3.993  irreducible_error=1.000
gamma=3.00  p=600  n=200  risk=4.072  bias=0.104  var=3.025  irreducible_error=1.000
gamma=4.00  p=800  n=200  risk=3.626  bias=0.108  var=2.575  irreducible_error=1.000
gamma=5.00  p=1000  n=200  risk=3.282  bias=0.119  var=2.212  irreducible_error=1.00
0
gamma=6.00  p=1200  n=200  risk=3.223  bias=0.110  var=2.155  irreducible_error=1.00
0
gamma=7.00  p=1400  n=200  risk=3.143  bias=0.137  var=2.070  irreducible_error=1.00
0
gamma=8.00  p=1600  n=200  risk=3.104  bias=0.134  var=2.025  irreducible_error=1.00
0
gamma=9.00  p=1800  n=200  risk=2.999  bias=0.128  var=1.947  irreducible_error=1.00
0
gamma=10.00  p=2000  n=200  risk=3.057  bias=0.128  var=1.976  irreducible_error=1.0
00
```

In [49]:
```
# A plotting function we provide. No need to change this, although you can if you'd

def plot_double_descent_risk_decomp(gammas, risk, bias, variance, 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
```

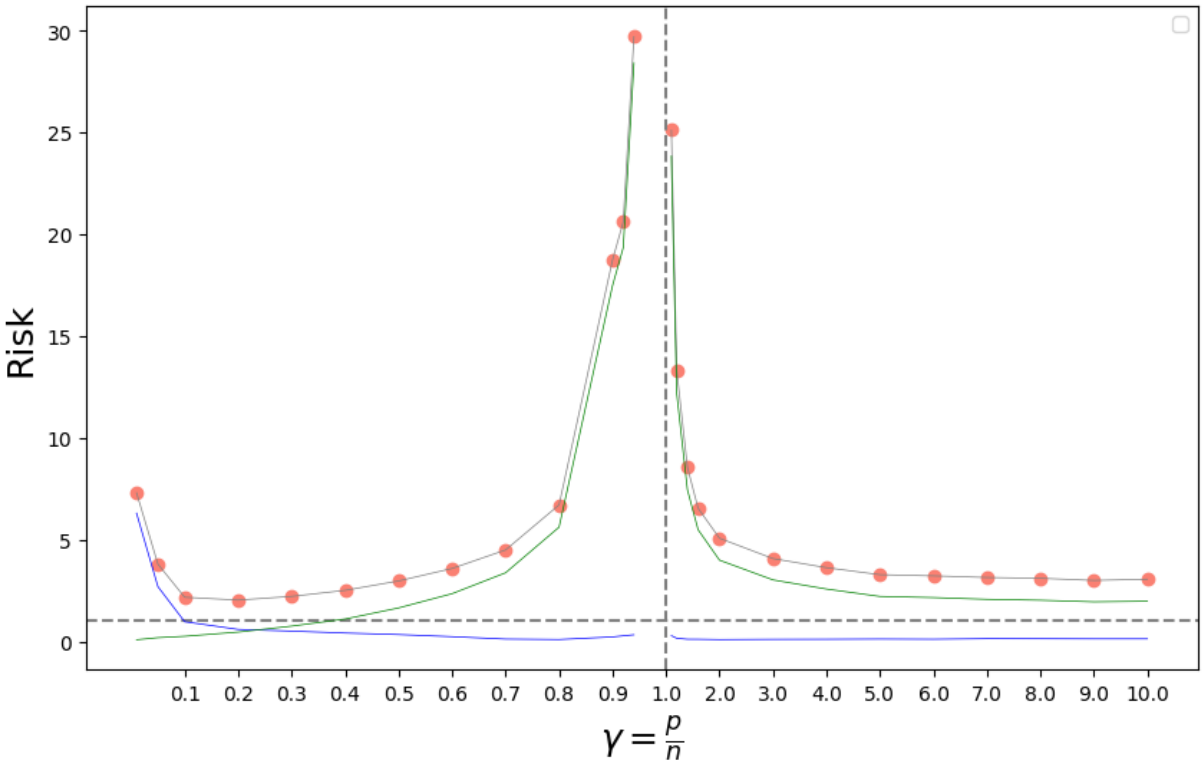
```
        else:
            tick_pos[i] = gammas[i] + 9
    ax.axvline(x=tick_pos[np.array(gammas)==1][0], linestyle='dashed', color='gray'
    ax.axhline(y=sigma**2, linestyle='dashed', color='gray')
    ax.scatter(tick_pos, risk, color='salmon')
    ax.plot(tick_pos, risk, color='gray', linewidth=.5)
    ax.plot(tick_pos, bias, color='blue', linewidth=.5)
    ax.plot(tick_pos, variance, color='green', linewidth=.5)
    ax.legend(['risk', 'bias', 'variance'])

    tickgam = [gam for gam in gammas if (gam > .05 and gam <= .9) or gam >= 2 or ga
    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)
    plt.legend(loc='upper right')
```

In [50]:
```
len(variance)
```

Out[50]: 27

In [51]:
```
plot_double_descent_risk_decomp(gammas, risk, bias, variance, sigma)
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that
artists whose label start with an underscore are ignored when legend() is called wit
h no argument.
```



The bias decreases initially as gamma increases, but then it flattens at about 0.1. The initial decrease occurs as the model gains enough parameters to accurately fit the data. Once it does, there's an irreducible bias due to the sampling noise sigma. In the underparameterized

domain, the variance increases with gamma, because each parameter is adding its own variance. In the overparameterized region, the variance goes down, and thats due to central limit affects induced by the randomness of the features, as discussed in class. As described by the risk decomposition, the bias^2 plus the variance appears to about equal the risk. Technically the bias variance decomposition is a formula conditioned on x; I averaged it across all x which is why the breakdown is not exact equality.

Problem 3.4

In class, we discussed the interpretation of the minimum-norm estimator $\hat{\beta}_{\text{mn}}$. Geometrically, we can describe $\hat{\beta}_{\text{mn}}$ 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 $\lambda \to 0$, $\hat{\beta}_\lambda$ converges to $\hat{\beta}_{\text{mn}}$.

Your task is to show that as $\lambda \to 0$, 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.

IML PSET Problem 3.4

ridge regression: $\hat{B}_\lambda = (X^TX + \lambda I)^{-1}X^Ty$

min. norm solution: $\hat{B}_{mn} = X^T(XX^T)^{-1}y$

I WTS $\hat{B}_\lambda \to \hat{B}_{mn}$ as $\lambda \to 0$ in our parametrized region

$$\left((X^TX + \lambda I)^{-1}X^T\right)y \qquad (\hat{B}_\lambda \text{ ridge})$$

$$= \left(X^T(XX^T + \lambda I)^{-1}\right)y \qquad (\text{identity})$$

$$\Rightarrow X^T(XX^T)y \qquad (\text{as } \lambda \to 0)$$

which is $\hat{B}_{mn}$ ∎

Proof of Hint Identity

$$(X^TX + \lambda I)^{-1}X^T$$

$$= \left(\frac{1}{\lambda}\right)\left(\frac{1}{\lambda}X^TX + I\right)^{-1}X^T \qquad (\text{pull out } 1/\lambda)$$

$$= \left(\frac{1}{\lambda}\right)\left[X^T\left(\frac{1}{\lambda}XX^T + I\right)^{-1}\right] \qquad (\text{push through identity})$$

$$= X^T(XX^T + \lambda I)^{-1} \qquad (\text{push in } 1/\lambda)$$

∎

* Used Wikipedia page for Woodbury as reference *

Proof of Push Through Identity

$$U + UVU = U + UVU$$

$$U(I + VU) = (I + UV)U$$

$$(I + UV)^{-1}U = U(I + VU)^{-1}$$

∎