assn1

February 22, 2022

1 Intermediate Machine Learning: Assignment 1

Deadline

Assignment 1 is due Wednesday, February 23 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 on Gradescope. 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

- Lasso
- Bias-variance decomposition
- Mercer kernels
- Convolutional neural networks

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

1.1 Problem 1: A goat rodeo with the lasso (15 points)

In this problem, you will use the lasso to round up some predictor variables. (The term goat rodeo refers to "a chaotic event where many things must go right for the situation to work." With careful use of the lasso and least squares regression, you can avoid chaos.)

We have prepared a data set with $y = X\beta + \sigma\epsilon$ where β is a sparse vector and $\epsilon_i \sim N(0,1)$ is Gaussian noise. Your task is three-fold:

- Generate a plot of the lasso regularization paths;
- Determine which coefficients of β are nonzero;
- Give your best estimate of these nonzero coefficients.

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import pickle
from sklearn.linear_model import Lasso
%matplotlib inline
```

Just run the next cell to read in the data.

[ ]:
```
X, y = pd.read_pickle('https://raw.githubusercontent.com/YData123/sds365-sp22/
 ↪main/assignments/assn1/problem1_Xy.pkl')
n, p = X.shape
print("Number of rows: {}".format(n))
print("Number of columns: {}".format(p))
```

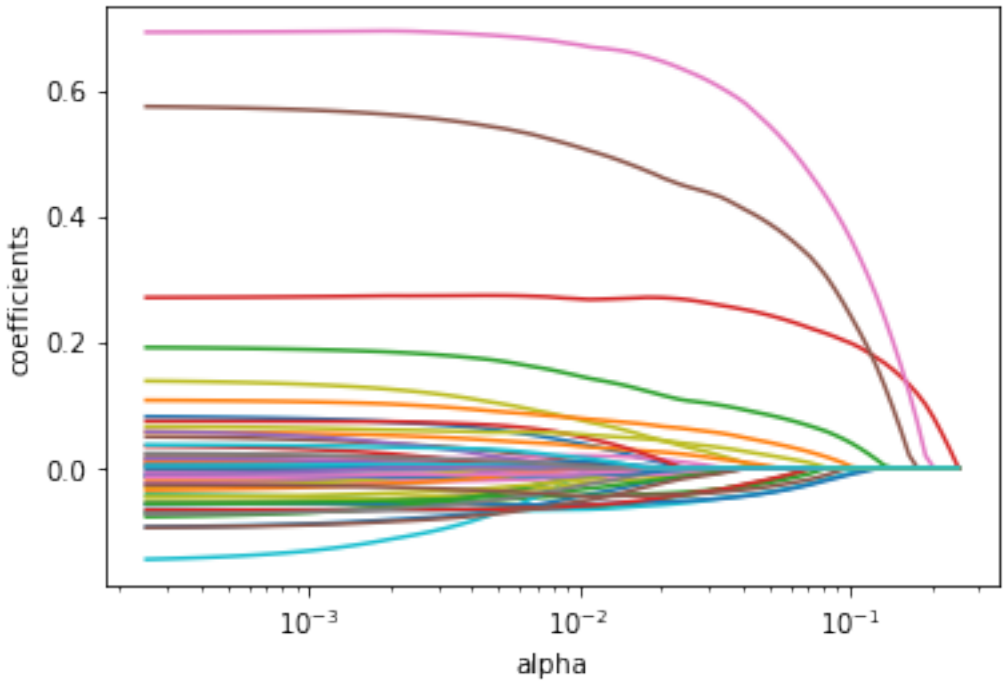
```
Number of rows: 100
Number of columns: 50
```

1.1.1 1.1: Lasso regularization paths

Run the lasso and plot the regularization paths. You can use the `Lasso` class from the `sklearn.linear_model` package. Plot the parameter paths with the regularization level λ (`alpha` in the code) on the log-scale, as done in the lasso demo code from class. (As always, be sure to label your axes.)

[ ]:
```
# your code here
from sklearn.linear_model import lasso_path

alphs, coefs, _ = lasso_path(X, y)

fig, axes = plt.subplots(1)
axes.plot(alphs, coefs.T)
axes.set_xscale('log')
axes.set_xlabel('alpha')
axes.set_ylabel('coefficients')
```

[ ]:
```
Text(0, 0.5, 'coefficients')
```



1.1.2 1.2: Select, estimate, and predict

The true model is linear, and only a subset $S \subset \{0, 1, \ldots, 49\}$ of the 50 variables have non-zero coefficients β_j. In this problem you should make three estimates:

1. An estimate $\hat{S}$ of S
2. An estimate $\hat{\beta}_j$ for each $j \in \hat{S}$
3. An estimate of the predictive risk $\mathbb{E}(Y - X\hat{\beta})$

We are not specifying how you should construct these estimates. You should use your judgement, taste, and the tools provided from class. However, you must clearly explain and justify whatever approach that you use.

[ ]:
```
# your code (and Markdown if needed) here
from sklearn.linear_model import LassoCV, LinearRegression
reg = LassoCV(cv=5, random_state=0).fit(X,y)
indices = np.where(reg.coef_!=0)[0]
print('The estimate subset indices', indices)

lnreg = LinearRegression().fit(X[:, indices], y)
print('The estimate beta hat', lnreg.coef_)

risk = np.mean(y - X[:, indices].dot(lnreg.coef_))
print('The estimated risk ', risk)
```

```
The estimate subset indices [ 2  3  5  6  8 11 12 19 20 21 23 26 32 42 45 48]
```

```
The estimate beta hat [-0.0643364   0.29695741  0.52519071  0.71956429
0.06739147  0.08330628
 -0.05529041 -0.06542213 -0.07389919  0.03914796 -0.06664548 -0.02158837
 -0.04292513  0.11715993 -0.06012735  0.06022714]
The estimated risk  0.03770867616530804
```

Answer: We utilize `LassoCV` as part of the `Lasso` class in sklearn package to calculate the estimated parameter setting with coordinate descent. The `coef_` denotes the set of parameters selected after doing cross validation (5-fold) on a lambda grid search (100 lambda values by default). We then refit the model with selected subset of S to get $\hat{\beta}$.

1.2 Problem 2: Risky business (10 points)

In class (and in these notes) we sketched a proof that, when the regression function has two bounded derivatives, the bias and variance for kernel smoothing scale as

$$\text{bias}^2 = O\left(h^4\right)$$
$$\text{var} = O\left(\frac{1}{nh^p}\right).$$

Here h is the bandwidth parameter, n is the sample size, and p is the number of predictor variables. These expressions are asymptotic, meaning that they apply as n gets large and h gets small. In this problem your job is to reason about the implications of this bias-variance decomposition for prediction.

Note: For this problem, you may either enter your answers in Markdown using LaTeX, or you write them on paper and scan to insert as an image in the notebook; whichever you prefer.

1.2.1 2.1 Selecting the optimal bandwidth

Suppose that the bias and variance are such that

$$\text{bias}^2(\hat{m}(x)) \leq c_1 h^4$$
$$\text{var}(\hat{m}(x)) \leq c_2 \frac{1}{nh^p}.$$

for two constants c_1 and c_2. Using these expressions and a little calculus, determine the optimal bandwidth h to minimize the risk function

$$R(h) = \mathbb{E}\left(\hat{m}(x) - m(x)\right)^2.$$

Your answer should involve the constants c_1, c_2, and n and p. Give a bound on the resulting risk using this bandwidth.

1.2.2 2.2 Bandwith selection without tears

Now, going back to the expressions $\text{bias}^2 = O\left(h^4\right)$ and $\text{var} = O\left(\frac{1}{nh^p}\right)$, explain why the scaling of the optimal bandwidth (as a function of n and p), must satisfy $\text{bias}^2 \approx \text{var}$; that is, they must be of the same order as $h \to 0$. Then, without using any calculus, use this argument to determine the optimal scaling of the bandwidth and the fastest rate at which the risk $R(h) = \mathbb{E}\left(\hat{m}(x) - m(x)\right)^2$ can approach zero as the sample size increases.

1.2.3 2.3 The cursed COD

Using the risk bound you derive above, make a plot that demonstrates the curse of dimensionality by plotting the sample size required to achieve a given level of risk. Specifically, let the target risk R vary between 0.1 and 0.5, and let the dimension p vary between 1 and 20, and plot the sample size required to achieve that risk. Give a single plot that shows the collection of curves for each dimension.

1.3 Answer

1.3.1 2.1

The bias variance decomposition of the risk function can be derived as

$$
\begin{aligned}
R(h) &= \mathbb{E}\left[\left(\hat{m}(x) - \mathbb{E}[\hat{m}(x)] + \mathbb{E}[\hat{m}(x)] - m(x)\right)^2\right] && (1)\\
&= \left(\mathbb{E}[\hat{m}(x)] - m(x)\right)^2 + \left(\mathbb{E}\left[\hat{m}(x) - \mathbb{E}[\hat{m}(x)]\right]\right)^2 + \mathbb{E}\left[2(\hat{m}(x) - \mathbb{E}[\hat{m}(x)])(\mathbb{E}[\hat{m}(x)] - m(x))\right] && (2)\\
&= \text{bias}^2(\hat{m}(x)) + \text{var}(\hat{m}(x)) && (3)\\
&\leq c_1 h^4 + c_2 \frac{1}{nh^p}. && (4)
\end{aligned}
$$

Denote the RHS as $g(h)$, we have its derivate as

$$
\frac{dg(h)}{dh} = 4c_1 h^3 - \frac{c_2 p}{n} h^{-p-1} \tag{5}
$$

and the infimum is achieved when

$$
4c_1 h^3 - \frac{c_2 p}{n} h^{-p-1} = 0
$$

and thus the optimal bandwdith h is

$$
h = \left(\frac{c_2 p}{4c_1 n}\right)^{\frac{1}{4+p}}
$$

The risk function at the optimal bandwidth is

$$
R(h) \leq c_1 \left(\frac{c_2 p}{4c_1 n}\right)^{\frac{4}{4+p}} + c_2 \frac{1}{n\left(\frac{c_2 p}{4c_1 n}\right)^{\frac{p}{4+p}}}
$$

1.3.2 2.2

From 2.1, for the optimal bandwidth we have

$$\text{bias}^2 = \mathcal{O}(h^4) \approx \mathcal{O}(\left(\frac{1}{n}\right)^{\frac{4}{4+p}}))$$

and

$$\text{var} = \mathcal{O}\left(\frac{1}{nh^p}\right) \approx \mathcal{O}(\frac{1}{n\left(\frac{1}{n}\right)^{\frac{p}{4+p}}})$$

and therefore

$$\frac{\text{bias}^2}{\text{var}} \approx 1$$

Using such argument, we have the optimal bandwidth h is

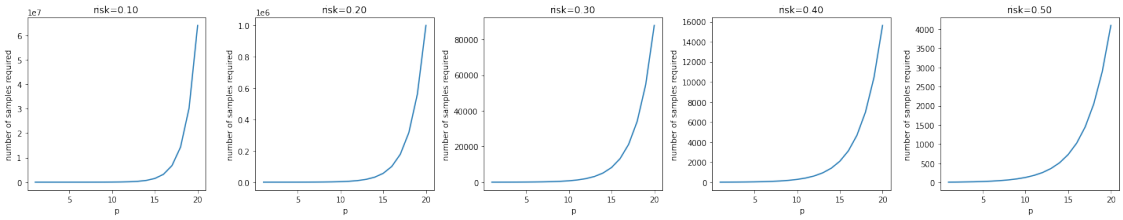
$$h \approx n^{-\frac{1}{4+p}}$$

1.3.3 2.3

Refer to the following code

[ ]:
```
# your code and markdown with derivations here
risk = np.linspace(0.1, 0.5, 5)
p = np.linspace(1, 20, 20)

def func_caln(risk_val, p_val):
    n = 1
    while 2*n**(-4/(4+p_val)) > risk_val:
        # print(risk_val)
        # print(2*n**(-4/(4+p_val)))
        n+=1
    return n

n_arr = np.zeros((risk.shape[0], p.shape[0]))
for i, risk_val in enumerate(risk):
    for j, p_val in enumerate(p):
        n_arr[i][j] = func_caln(risk_val, p_val)
```

[ ]:
```
fig, axes = plt.subplots(1, risk.shape[0])
fig.set_size_inches(20, 4)
for j in range(risk.shape[0]):
    axes[j].plot(p, n_arr[j,:])
    axes[j].set_title('risk={:.2f}'.format(risk[j]))
    axes[j].set_xlabel('p')
    axes[j].set_ylabel('number of samples required')
fig.tight_layout()
    # axes.legend()
```



1.4 Problem 3: A kernel of truth (15 points)

For problem you will implement nonparametric regression using Mercer kernels and penalization, in 1-dimension. This can be compared to regression using smoothing kernels.

As discussed in lecture, nonparametric regression with Mercer kernels is based on the infinite dimensional ridge regression

$$\hat{m} = \text{argmin} \|Y - m\|^2 + \lambda \|m\|_K^2$$

By the representer theorem, this is equivalent to setting $\hat{m}(x) = \sum_{i=1}^n \hat{\alpha}_i K(X_i, x)$ and using the finite dimensional optimization

$$\hat{\alpha} = \text{argmin} \|Y - \mathbb{K}\alpha\|^2 + \lambda \alpha^T \mathbb{K} \alpha$$

1.4.1 3.1 Solve

Derive a closed-form expression for the minimizer $\hat{\alpha}$. Show all of the steps in your derivation, and justify each step. (As above, you may either enter your answers in Markdown using LaTeX, or insert an image of your handwritten solution.)

1.4.2 Answer:

Denote

$$\begin{aligned} g(\alpha) &= \|Y - \mathbb{K}\alpha\|^2 + \lambda \alpha^T \mathbb{K} \alpha && (6) \\ &= Y^T Y - 2Y^T \mathbb{K}\alpha + \alpha^T \mathbb{K}^T \mathbb{K} \alpha + \lambda \alpha^T \mathbb{K} \alpha && (7) \end{aligned}$$

From the matrix calculus, we have

$$\frac{dg}{d\alpha} = -2\mathbb{K}^T Y + 2\mathbb{K}^T \mathbb{K}\alpha + 2\lambda \mathbb{K}\alpha = 0 \quad (8)$$

$$(9)$$

and thus

$$(\mathbb{K}^T \mathbb{K} + \lambda \mathbb{K})\alpha = \mathbb{K}^T Y$$

$$\begin{aligned} \alpha &= (\mathbb{K}^T \mathbb{K} + \lambda \mathbb{K})^{-1} \mathbb{K}^T Y && (10) \\ &= (\mathbb{K} + \lambda I)^{-1} Y && (11) \end{aligned}$$

1.4.3 3.2 Implement

Next you will use your solution above and implement Mercer kernel regression. We give some starter code.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import clear_output
from time import sleep
```

The following cell defines some "helper functions" for this exercise. You don't need to change any of this code. (If you do want to make changes, just describe what you did and why.)

[ ]:
```
def plot_estimate(x, f, fhat, X, y, sigma, lmbda, sleeptime=.01):
    clear_output(wait=True)
    plt.figure(figsize=(10,6))
    plt.plot(x, f, color='red', linewidth=2, label='true function')
    plt.plot(x, fhat, color='blue', linewidth=2, label='estimated function')
    plt.scatter(X, y, color='black', alpha=.5, label='random sample')
    plt.ylim(np.min(f)-4*sigma, np.max(f)+4*sigma)
    plt.legend(loc='upper left')
    plt.title('lambda: %.3g' % lmbda)
    plt.xlabel('x')
    plt.ylabel('estimated m(x)')
    plt.show()
    sleep(sleeptime)

def true_fn(x):
    return 3*x**2

def run_simulation(kernel, lmbdas, show_bias_variance=True):
    min_x, max_x = -1, 1
    x = np.linspace(min_x, max_x, 100)
    f = true_fn(x)
    sigma = .25
    estimates = []
    trials = 500

    for lmbda in lmbdas:
        estimates_lambda = []
        for i in np.arange(trials):
            X = np.sort(np.random.uniform(low=min_x, high=max_x, size=50))
            fX = true_fn(X)
            y = fX + sigma*np.random.normal(size=len(X))
            fhat = mercer_kernel_regress(kernel, X, y, x, lmbda=lmbda)
            if i % 50 == 0:
                plot_estimate(x, f, fhat, X, y, sigma, lmbda)
```

```
            estimates_lambda.append(fhat)
        estimates.append(estimates_lambda)

    if show_bias_variance == False:
        return

    fhat = np.array(estimates)
    sq_bias = np.zeros(len(lmbdas))
    variance = np.zeros(len(lmbdas))

    for i in np.arange(len(lmbdas)):
        sq_bias[i] = np.mean((np.mean(fhat[i], axis=0) - f)**2)
        variance[i] = np.mean(np.var(fhat[i], axis=0))

    plt.figure(figsize=(10,6))
    plt.plot(lmbdas, sq_bias, label='squared bias', linewidth=2)
    plt.plot(lmbdas, variance, label='variance', linewidth=2)
    plt.plot(lmbdas, sq_bias + variance, label='risk')
    plt.legend()
    plt.show()
```

Your job is to implement Mercer kernel regression and run it on two different kernel functions. The two kernels could simply be the Gaussian kernel with two different bandwidths, or you might experiment with other kernels.

The function `mercer_kernel_regress` takes a kernel, training data `X` and `y`, an array of values `x` to evaluate the function on, and a regularization parameter. You'll use your derivation above to determine the coefficients α. For some clues and suggestions on how to do the implementation, see our demo code for smoothing kernels. You need to do something very similar.

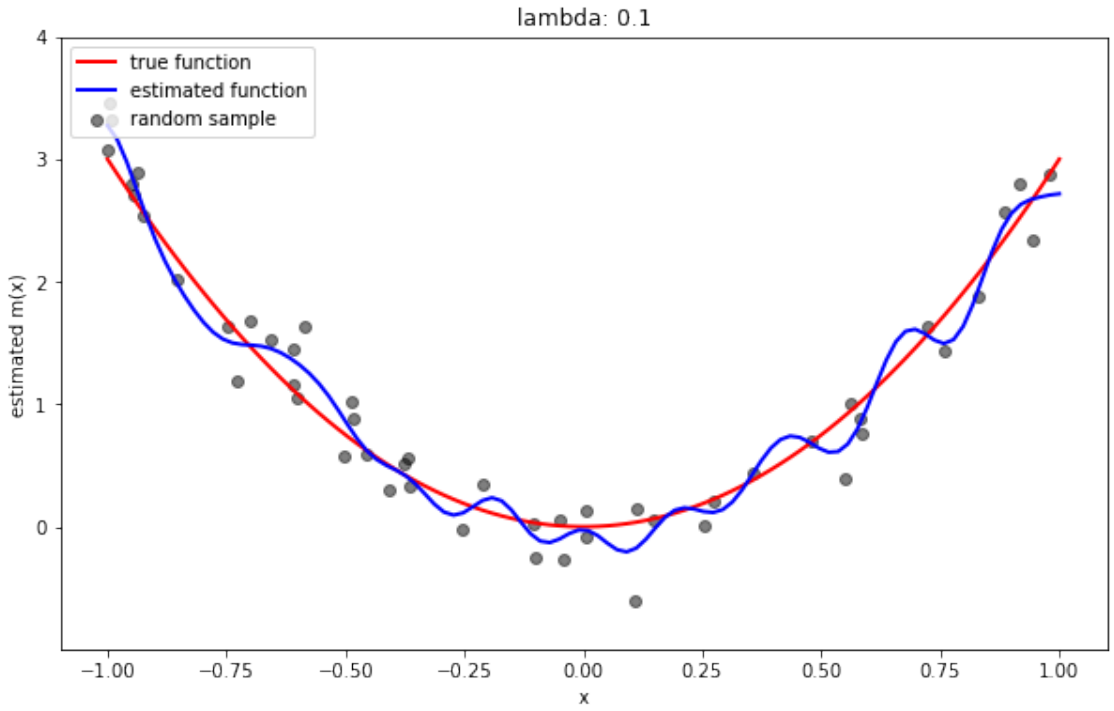
[ ]:
```
def mercer_kernel_regress(kernel, X, y, x, lmbda):
    # your implementation here
    K = np.zeros(len(X)*len(X)).reshape(len(X), len(X))
    for j in np.arange(K.shape[1]):
        K[:,j] = kernel(X, X[j])
    alpha = np.linalg.inv(K + lmbda*np.identity(len(X))).dot(y)

    Khat = np.zeros(len(x)*len(X)).reshape(len(x), len(X))
    for j in np.arange(Khat.shape[1]):
        Khat[:,j] = kernel(x, X[j])
    fhat = Khat @ alpha
    return fhat

def kernel1(x,y):
    # your implementation here
    h = 0.1
    return (1/h)*np.exp(-(x-y)**2/(2*h**2))
```

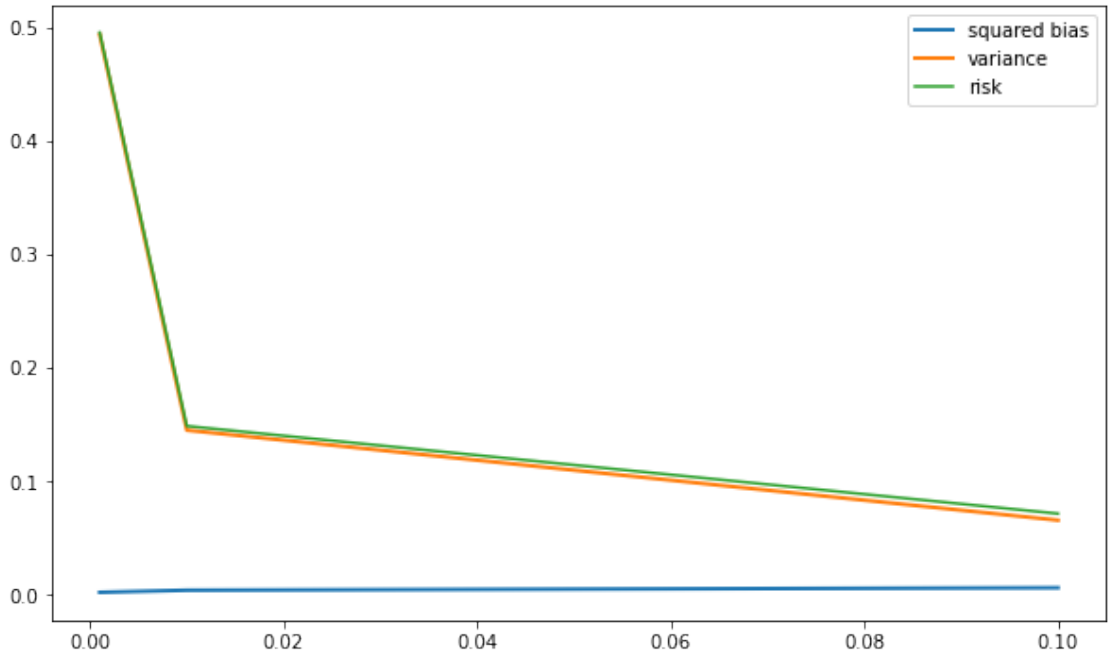
```
def kernel2(x,y):
    # your implementation here
    h = 1.5
    return (1/h)*np.exp(-(x-y)**2/(2*h**2))
```

1.4.4 3.3 Run two simulations and select regularization parameters

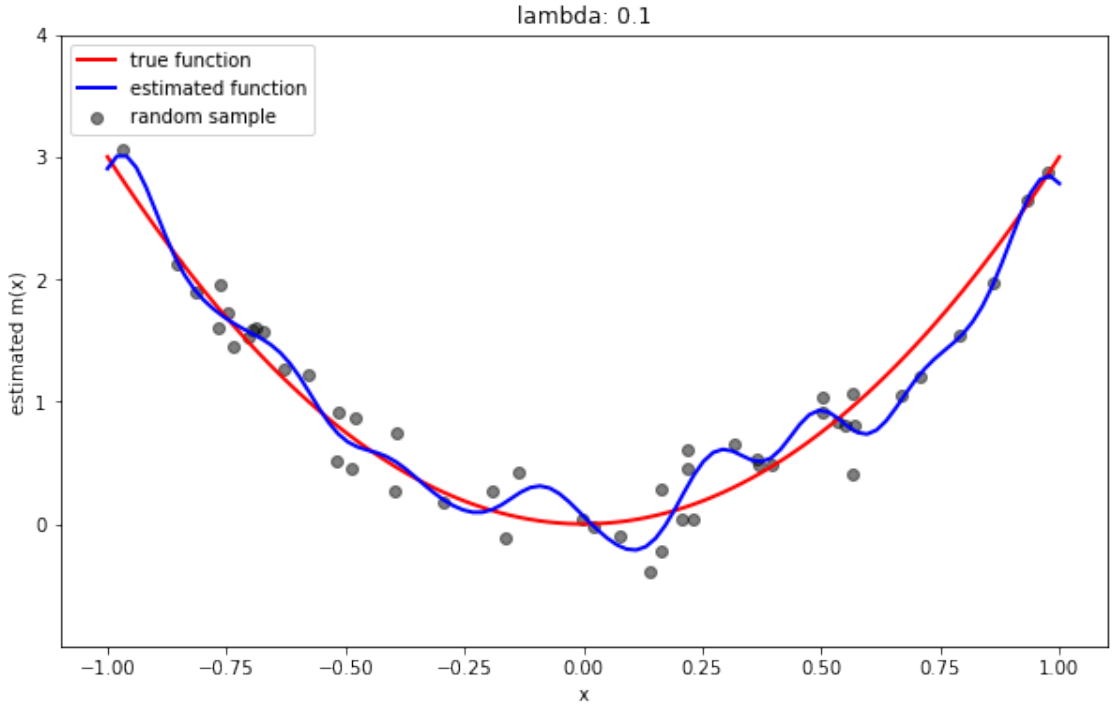
Finally, using our starter code and your own implementation above, run two simulations, one using `kernel1` and the other using `kernel2`. After each simulation, select a regularization level from the bias-variance tradeoff, and then run a final simulation with that regularization level. In the following starter code, you only need to specify the sequence of regularization parameters.

[ ]:
```
lmbdas = [0.001, 0.01, 0.1]# define your sequence of lambdas
run_simulation(kernel1, lmbdas)
```

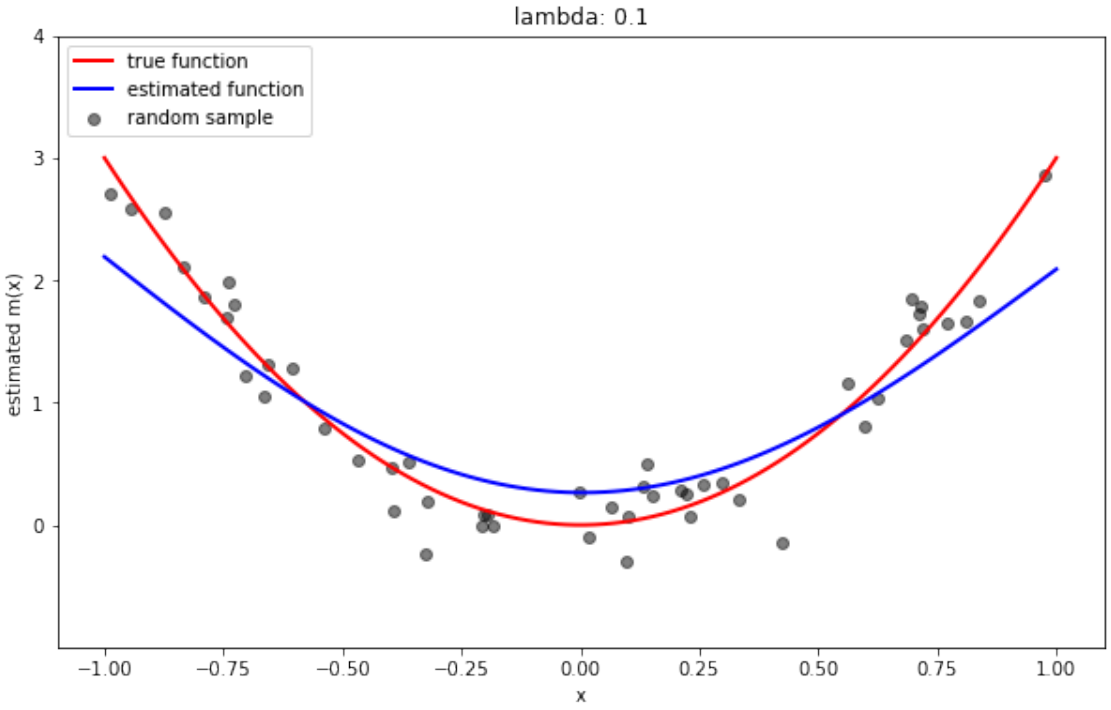


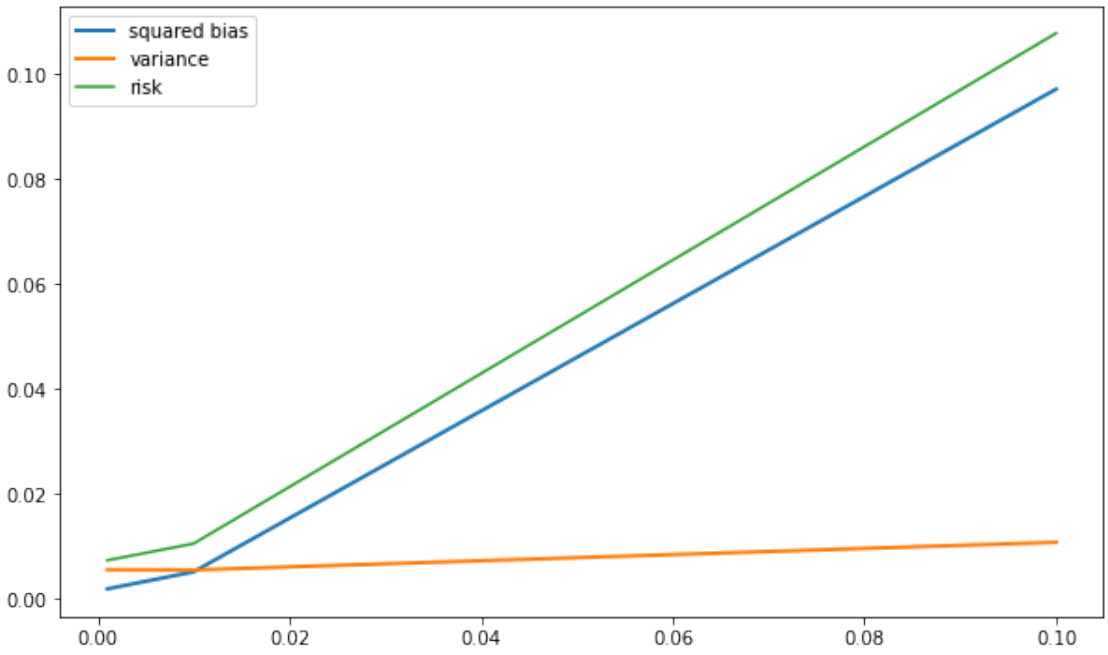


[ ]:
```
lambda_hat = 0.1# set the optimal lambda
run_simulation(kernel1, [lambda_hat], show_bias_variance=False)
```

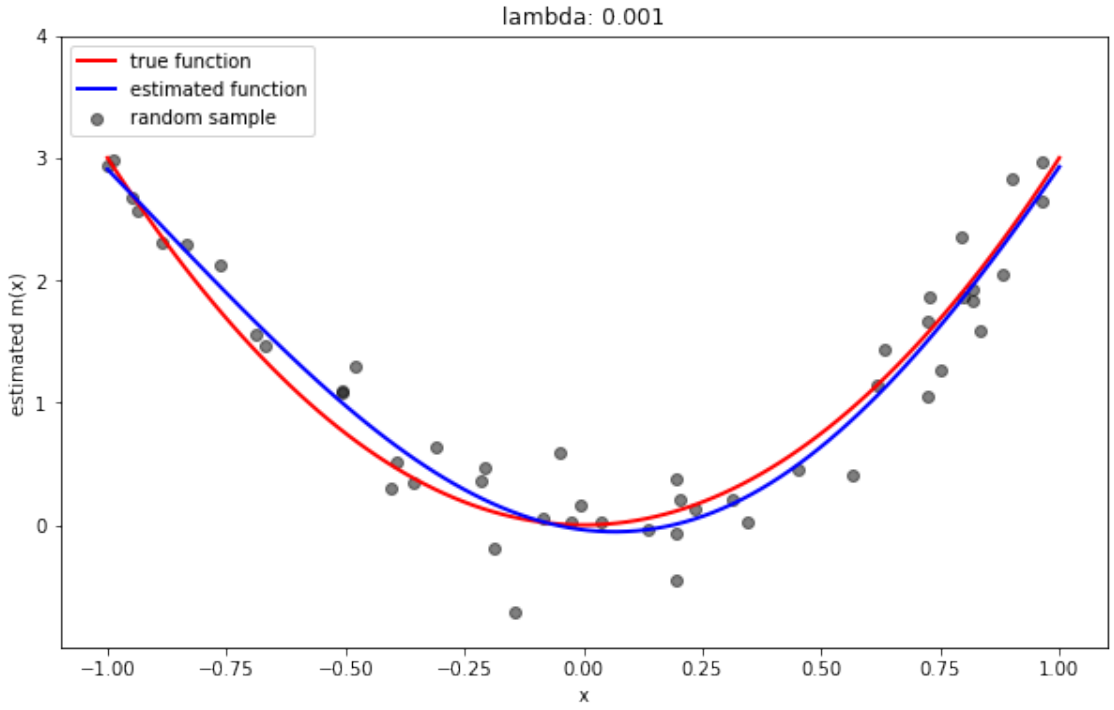


```
[ ]: lmbdas = [0.001, 0.01, 0.1]# define your sequence of lambdas
     run_simulation(kernel2, lmbdas)
```





```
[ ]: lambda_hat = 0.001# set the optimal lambda
     run_simulation(kernel2, [lambda_hat], show_bias_variance=False)
```



1.5 Problem 4: Brain food (20 points)

This problem gives you some experience with convolutional neural networks for image classification using TensorFlow.

The classification task is to discriminate real optical images of brain activity in mice from fake images that were constructed using a generative adversarial network (GAN). A paper on the underlying imaging technologies developed by Yale researchers is here.

For this problem we'll walk you through the following steps: * Downloading the data * Loading the data * Displaying some sample images * Building a classification model using a simple CNN

After this, your task will be to improve upon this baseline model by building, training, and evaluating two more CNNs.

1.5.1 Downloading the data

The data are contained in a group of compressed files on AWS. There are 10 files of real images, and 10 files of fake images; each file is roughly 100 MB in size; so the entire dataset is about 2 GB. You should download the data to the computer you are running on, and place the in a folder named "data".

Important note: If you do not have enough space to download all of the data, just download what you can; there will be no penalty for running on less data. If you want assistance running in Google

Colab, please let us know. You'll need to download segments 7,8,9 (real and fake) for testing, and at least one other segment for training.

Here are URLs to access the 20 data files:

https://sds365.s3.amazonaws.com/calcium/real_0.gz https://sds365.s3.amazonaws.com/calcium/real_1.gz
https://sds365.s3.amazonaws.com/calcium/real_2.gz https://sds365.s3.amazonaws.com/calcium/real_3.gz
https://sds365.s3.amazonaws.com/calcium/real_4.gz https://sds365.s3.amazonaws.com/calcium/real_5.gz
https://sds365.s3.amazonaws.com/calcium/real_6.gz https://sds365.s3.amazonaws.com/calcium/real_7.gz
https://sds365.s3.amazonaws.com/calcium/real_8.gz https://sds365.s3.amazonaws.com/calcium/real_9.gz

https://sds365.s3.amazonaws.com/calcium/fake_0.gz https://sds365.s3.amazonaws.com/calcium/fake_1.gz
https://sds365.s3.amazonaws.com/calcium/fake_2.gz https://sds365.s3.amazonaws.com/calcium/fake_3.gz
https://sds365.s3.amazonaws.com/calcium/fake_4.gz https://sds365.s3.amazonaws.com/calcium/fake_5.gz
https://sds365.s3.amazonaws.com/calcium/fake_6.gz https://sds365.s3.amazonaws.com/calcium/fake_7.gz
https://sds365.s3.amazonaws.com/calcium/fake_8.gz https://sds365.s3.amazonaws.com/calcium/fake_9.gz

We import some Python packages from TensorFlow and Keras.

[ ]:
```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import numpy as np
import gzip
import matplotlib.pyplot as plt
```

[ ]:
```
tf.__version__
```

[ ]:
```
'1.15.5'
```

Some helper functions for reading the data and plotting images.

[ ]:
```
def plot_images(imgs, title):
    plt.figure(figsize=(10,10))
    for i in range(9):
        plt.subplot(3,3,i+1)
        plt.imshow(imgs[i], cmap='rainbow')
        plt.axis('off')
    plt.suptitle(title)

def read_gz(filedir, shape=[-1,128,128]):
    print('reading %s' % filedir)
    with gzip.open(filedir, 'rb') as f:
        content = f.read()
    imgs = np.frombuffer(content, dtype='float32').reshape(shape)
    return imgs

def load_data(pieces):
    data = []
    label = []
    print('Loading data:\n-------------')
```

```
    for i in pieces:
        real_img = read_gz('data/real_{:d}.gz'.format(i), shape=[-1,128,128, 1])
        fake_img = read_gz('data/fake_{:d}.gz'.format(i), shape=[-1,128,128, 1])
        real_label = np.zeros((real_img.shape[0],1))
        fake_label = np.zeros((fake_img.shape[0],1))
        real_label[:,0] = 0
        fake_label[:,0] = 1
        data.append(real_img)
        data.append(fake_img)
        label.append(real_label)
        label.append(fake_label)
    print()
    data_all = np.concatenate(data, axis=0)
    label_all = np.concatenate(label, axis=0)
    return data_all, label_all
```

1.5.2 Loading the data

Let's look at some images.

[ ]:
```
real_images = read_gz('data/real_0.gz')
fake_images = read_gz('data/fake_0.gz')

# real_img are original data, fake_img are synthetic data generated using a GAN␣
 ↪model
```

```
reading data/real_0.gz
reading data/fake_0.gz
```

Each of the images is 128x128 pixels, and there are 2048 images in each file:

[ ]:
```
real_images.shape, fake_images.shape
```

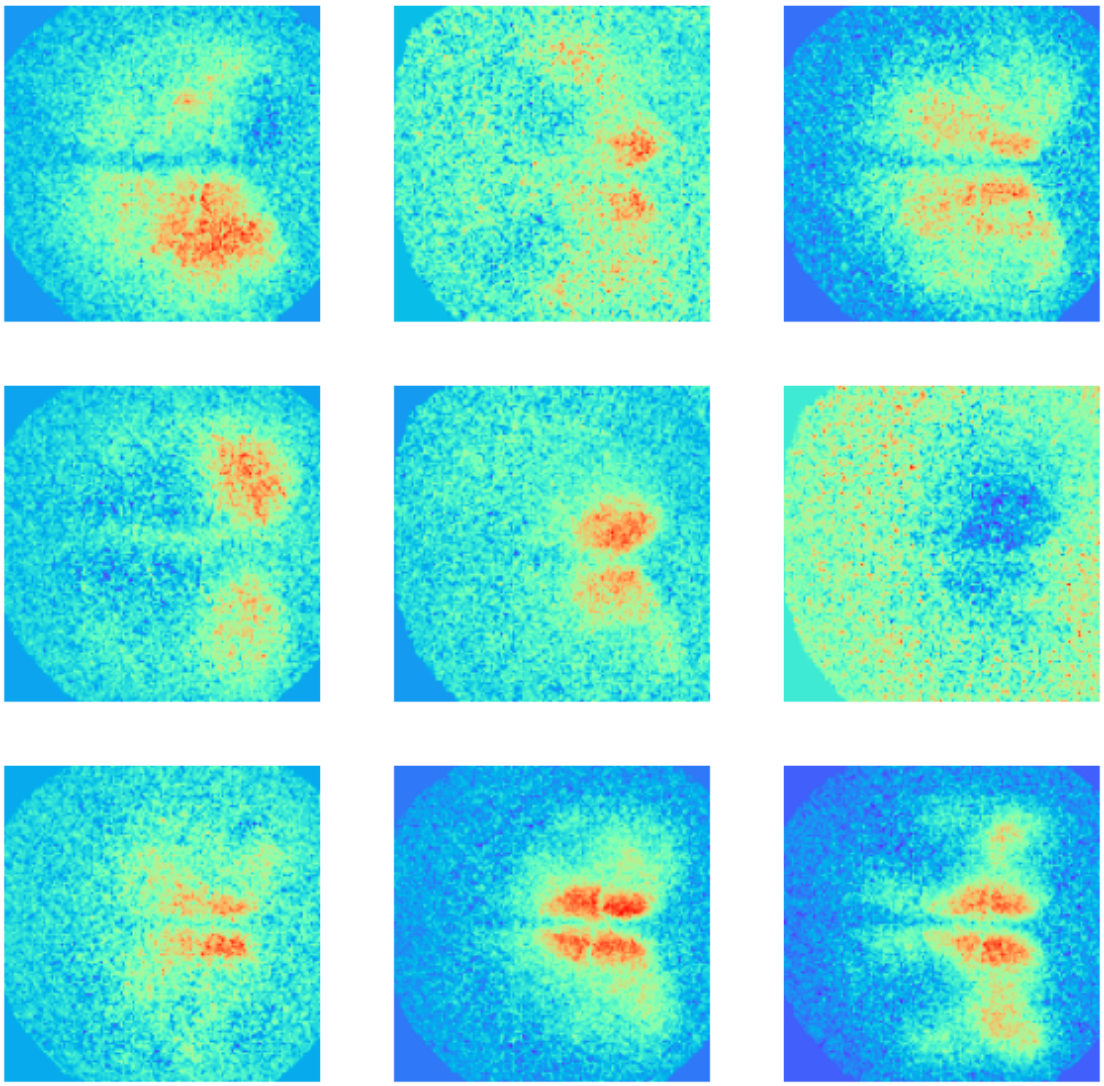
[ ]:
```
((2048, 128, 128), (2048, 128, 128))
```

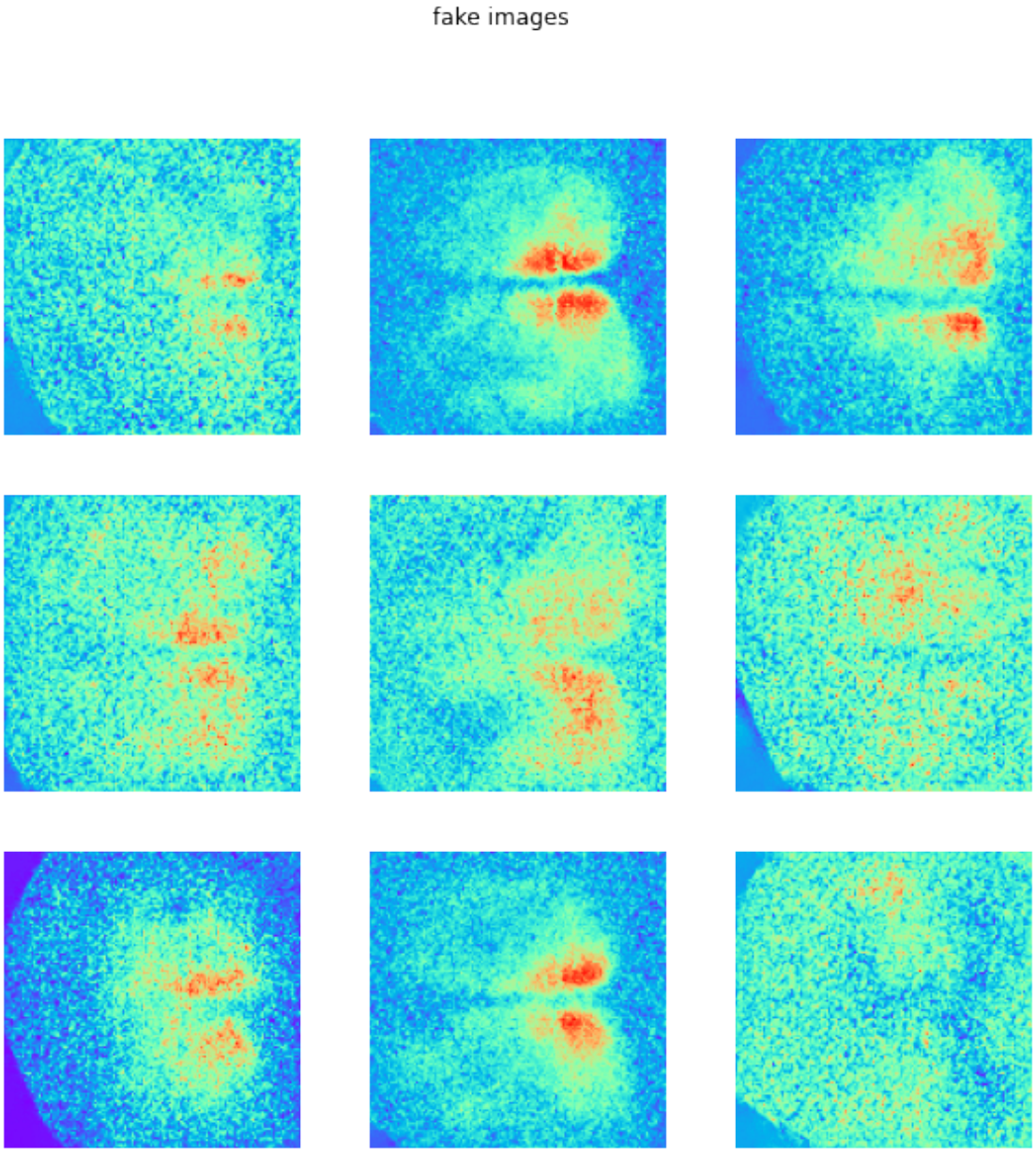
1.5.3 Displaying some sample images

Now we'll display some real and fake images. Can you spot any differences between the two. Do you think that you could learn to tell them apart? Please comment below.

[ ]:
```
plot_images(real_images, 'real images')
plot_images(fake_images, 'fake images')
```

real images





1.6 Answer:

It is quite challenging to tell the differences without having too much prior knowledge on the details of the images. I think there are still some differences such as the coloring of the real images is slightly stronger than fake images. The cortex region with red in the fake images also seems to be more diffused than real images.

1.6.1 4.1 Building a baseline model

The code below trains and evaluates a baseline model. The model is trained on six of the data files (3 real, and 3 fake, about 12,000 images total) and is tested on six of the data files. We begin by loading in the data.

[ ]:
```
train_images, train_labels = load_data([0,1,2])
test_images, test_labels = load_data([7,8,9])
```

```
Loading data:
-------------
reading data/real_0.gz
reading data/fake_0.gz
reading data/real_1.gz
reading data/fake_1.gz
reading data/real_2.gz
reading data/fake_2.gz

Loading data:
-------------
reading data/real_7.gz
reading data/fake_7.gz
reading data/real_8.gz
reading data/fake_8.gz
reading data/real_9.gz
reading data/fake_9.gz
```

Next, we initialize our convolutional neural network. This particular network has four layers: A convolutional layer, a max pooling latter, a flattened layer, and a dense layer with two terminal neurons and no activation function. The total number of parameters in this network is 62,338.

[ ]:
```
model = models.Sequential()
model.add(layers.Conv2D(32, (5, 5), activation='relu', input_shape=(128, 128,␣
↪1)))
model.add(layers.MaxPooling2D((4, 4)))
model.add(layers.Flatten())
model.add(layers.Dense(2))
model.summary()
```

```
WARNING:tensorflow:From
/home/xiaoranzhang/anaconda3/envs/tf_directml/lib/python3.6/site-
packages/tensorflow_core/python/ops/resource_variable_ops.py:1630: calling
BaseResourceVariable.__init__ (from tensorflow.python.ops.resource_variable_ops)
with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
Model: "sequential"
_________________________________________________________________
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 124, 124, 32)      832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 31, 31, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 30752)             0
_________________________________________________________________
dense (Dense)                (None, 2)                 61506
=================================================================
Total params: 62,338
Trainable params: 62,338
Non-trainable params: 0
_________________________________________________________________
```

Next, we train the model. Here we just train for two "epochs", where each epoch scans through the data in random order, processing a batch of images in each stochastic gradient descent step.

[ ]:
```
model.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=2, validation_split=0.2,␣
 ↪shuffle=True)
```

```
Train on 9830 samples, validate on 2458 samples
Epoch 1/2
9830/9830 [==============================] - 10s 1ms/sample - loss: 0.2821 -
acc: 0.8938 - val_loss: 0.0714 - val_acc: 0.9882
Epoch 2/2
9830/9830 [==============================] - 4s 435us/sample - loss: 0.0183 -
acc: 0.9992 - val_loss: 0.0179 - val_acc: 0.9988
```

Finally, we evaluate the model on the test data (the last three segments of images: 7,8,9). Your accuracy may vary for this identical configuration due to the randomness in SGD.

[ ]:
```
est_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print('test accuracy: {:.4f}'.format(test_acc))
```

```
12288/12288 - 3s - loss: 0.1955 - acc: 0.8808
test accuracy: 0.8808
```

1.6.2 4.2 and 4.3: Improving upon the baseline

Your job is to improve upon this baseline model. *You should always test on the same segments 7,8,9 as above.* You can extend this model in various ways:

- Train on more data (but always test on the same test set)
- Add more convolutional layers
- Add more dense layers, using an activation function
- Regularize using a dropout layer

- Use a different mini-batch size

For each model that you decide to train, describe how and why you built this model. For example, you can:

- Give a diagram showing the sequence of layers used;
- Explain your rationale for using each of the layers;
- Comment on the number of parameters used by the model, and which layers have the most parameters;
- Describe your findings on number of epochs and training data size;
- Comment on models you experimented with but did not include;
- Describe the increase or decrease in accuracy that resulted.

When we evaluate your notebook, we will look for (1) significant improvements in test accuracy (2) descriptions of your models that show understanding of how they work and (3) why you chose a given architecture.

1.6.3 Experiment 1: train the model with more data with longer epoch

We aim to first experiment with logner epoch and larger dataset without changing the model architecture to evaluate the impact of the size of training data on model's performance.

[ ]:
```
train_images, train_labels = load_data([0,1,2,3,4,5,6])
test_images, test_labels = load_data([7,8,9])
```

```
Loading data:
-------------
reading data/real_0.gz
reading data/fake_0.gz
reading data/real_1.gz
reading data/fake_1.gz
reading data/real_2.gz
reading data/fake_2.gz
reading data/real_3.gz
reading data/fake_3.gz
reading data/real_4.gz
reading data/fake_4.gz
reading data/real_5.gz
reading data/fake_5.gz
reading data/real_6.gz
reading data/fake_6.gz

Loading data:
-------------
reading data/real_7.gz
reading data/fake_7.gz
reading data/real_8.gz
reading data/fake_8.gz
reading data/real_9.gz
```

```
reading data/fake_9.gz
```

[ ]:
```
model = models.Sequential()
model.add(layers.Conv2D(32, (5, 5), activation='relu', input_shape=(128, 128,␣
 ↪1)))
model.add(layers.MaxPooling2D((4, 4)))
model.add(layers.Flatten())
model.add(layers.Dense(2))
model.summary()

model.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=10, validation_split=0.
 ↪2, shuffle=True)

est_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print('test accuracy: {:.4f}'.format(test_acc))
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 124, 124, 32)      832
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 31, 31, 32)        0
_________________________________________________________________
flatten_3 (Flatten)          (None, 30752)             0
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 61506
=================================================================
Total params: 62,338
Trainable params: 62,338
Non-trainable params: 0
_________________________________________________________________
Train on 22937 samples, validate on 5735 samples
Epoch 1/10
22937/22937 [==============================] - 10s 444us/sample - loss: 0.1014 -
acc: 0.9680 - val_loss: 0.0102 - val_acc: 0.9998
Epoch 2/10
22937/22937 [==============================] - 12s 508us/sample - loss: 0.0031 -
acc: 1.0000 - val_loss: 0.0021 - val_acc: 1.0000
Epoch 3/10
22937/22937 [==============================] - 12s 509us/sample - loss:
7.5405e-04 - acc: 1.0000 - val_loss: 8.9080e-04 - val_acc: 1.0000
Epoch 4/10
22937/22937 [==============================] - 12s 521us/sample - loss:
```

```
3.2931e-04 - acc: 1.0000 - val_loss: 5.5835e-04 - val_acc: 1.0000
Epoch 5/10
22937/22937 [==============================] - 12s 524us/sample - loss:
1.6504e-04 - acc: 1.0000 - val_loss: 2.6272e-04 - val_acc: 1.0000
Epoch 6/10
22937/22937 [==============================] - 11s 481us/sample - loss:
9.9351e-05 - acc: 1.0000 - val_loss: 1.7169e-04 - val_acc: 1.0000
Epoch 7/10
22937/22937 [==============================] - 10s 443us/sample - loss:
6.7046e-05 - acc: 1.0000 - val_loss: 1.0814e-04 - val_acc: 1.0000
Epoch 8/10
22937/22937 [==============================] - 11s 485us/sample - loss:
3.6514e-05 - acc: 1.0000 - val_loss: 6.8222e-05 - val_acc: 1.0000
Epoch 9/10
22937/22937 [==============================] - 12s 543us/sample - loss:
2.3229e-05 - acc: 1.0000 - val_loss: 4.3395e-05 - val_acc: 1.0000
Epoch 10/10
22937/22937 [==============================] - 12s 508us/sample - loss:
1.6099e-05 - acc: 1.0000 - val_loss: 3.3958e-05 - val_acc: 1.0000
12288/12288 - 3s - loss: 5.5824e-05 - acc: 1.0000
test accuracy: 1.0000
```

As increasing the dataset with longer epoch already achieves 100% test accuracy, we reduce the dataset with only 3 files for training to evaluate the impact of changing the architectures in the next two experiments.

[ ]:
```
train_images, train_labels = load_data([0,1,2])
test_images, test_labels = load_data([7,8,9])
```

```
Loading data:
-------------
reading data/real_0.gz
reading data/fake_0.gz
reading data/real_1.gz
reading data/fake_1.gz
reading data/real_2.gz
reading data/fake_2.gz

Loading data:
-------------
reading data/real_7.gz
reading data/fake_7.gz
reading data/real_8.gz
reading data/fake_8.gz
reading data/real_9.gz
reading data/fake_9.gz
```

1.6.4 Experiment 2: add more dense layer

Add more dense layer will increase the number of parameters in after flattening, which increases the network's capacity to learn more complicated latent representation. The result shows that adding additional Dense layer significantly improves the test accuracy from 0.881 to 0.996.

[ ]:
```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128,␣
 ↪1), padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(32))
model.add(layers.Dense(2))
model.summary()

model.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=5, validation_split=0.2,␣
 ↪shuffle=True)

est_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print('test accuracy: {:.4f}'.format(test_acc))
```

```
Model: "sequential_22"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_73 (Conv2D)           (None, 128, 128, 32)      320
_________________________________________________________________
max_pooling2d_55 (MaxPooling (None, 64, 64, 32)        0
_________________________________________________________________
flatten_20 (Flatten)         (None, 131072)            0
_________________________________________________________________
dense_23 (Dense)             (None, 32)                4194336
_________________________________________________________________
dense_24 (Dense)             (None, 2)                 66
=================================================================
Total params: 4,194,722
Trainable params: 4,194,722
Non-trainable params: 0
_________________________________________________________________
Train on 9830 samples, validate on 2458 samples
Epoch 1/5
9830/9830 [==============================] - 6s 633us/sample - loss: 0.4305 -
acc: 0.9507 - val_loss: 8.9766e-04 - val_acc: 0.9996
Epoch 2/5
9830/9830 [==============================] - 6s 609us/sample - loss: 1.5883e-04
```

```
- acc: 1.0000 - val_loss: 3.9961e-04 - val_acc: 1.0000
Epoch 3/5
9830/9830 [==============================] - 6s 630us/sample - loss: 6.7346e-05
- acc: 1.0000 - val_loss: 2.5271e-04 - val_acc: 1.0000
Epoch 4/5
9830/9830 [==============================] - 6s 585us/sample - loss: 3.7114e-05
- acc: 1.0000 - val_loss: 1.9328e-04 - val_acc: 1.0000
Epoch 5/5
9830/9830 [==============================] - 7s 720us/sample - loss: 2.4002e-05
- acc: 1.0000 - val_loss: 1.8491e-04 - val_acc: 1.0000
12288/12288 - 4s - loss: 0.0249 - acc: 0.9961
test accuracy: 0.9961
```

1.6.5 Experiment 3: evaluate the effectiveness of dropout and batchnorm

Having dropout and batchnorm layers reduces the chances of overfitting in the network. The results in this experiment show the effectiveness of the two layers as the test accuracy improves from 0.881 to 0.888.

[ ]:
```
model = models.Sequential()
model.add(layers.Conv2D(32, (5, 5), activation='relu', input_shape=(128, 128,␣
↪1)))
model.add(layers.MaxPooling2D((4, 4)))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.5))
model.add(layers.Flatten())
model.add(layers.Dense(2))
model.summary()

model.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=10, validation_split=0.
↪2, shuffle=True)

est_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print('test accuracy: {:.4f}'.format(test_acc))
```

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_38 (Conv2D)           (None, 124, 124, 32)      832
_________________________________________________________________
max_pooling2d_25 (MaxPooling (None, 31, 31, 32)        0
_________________________________________________________________
batch_normalization (BatchNo (None, 31, 31, 32)        128
_________________________________________________________________
```

```
dropout (Dropout)            (None, 31, 31, 32)        0
_________________________________________________________________
flatten_5 (Flatten)          (None, 30752)             0
_________________________________________________________________
dense_7 (Dense)              (None, 2)                 61506
=================================================================
Total params: 62,466
Trainable params: 62,402
Non-trainable params: 64
_________________________________________________________________
Train on 9830 samples, validate on 2458 samples
Epoch 1/10
9830/9830 [==============================] - 8s 787us/sample - loss: 0.0837 -
acc: 0.9826 - val_loss: 0.0731 - val_acc: 0.9992
Epoch 2/10
9830/9830 [==============================] - 6s 655us/sample - loss: 0.0123 -
acc: 0.9965 - val_loss: 6.6533e-04 - val_acc: 1.0000
Epoch 3/10
9830/9830 [==============================] - 7s 668us/sample - loss: 4.8391e-05
- acc: 1.0000 - val_loss: 3.9306e-05 - val_acc: 1.0000
Epoch 4/10
9830/9830 [==============================] - 7s 680us/sample - loss: 1.2855e-05
- acc: 1.0000 - val_loss: 1.9165e-05 - val_acc: 1.0000
Epoch 5/10
9830/9830 [==============================] - 7s 708us/sample - loss: 6.0477e-05
- acc: 1.0000 - val_loss: 2.7309e-06 - val_acc: 1.0000
Epoch 6/10
9830/9830 [==============================] - 7s 689us/sample - loss: 3.6059e-04
- acc: 0.9998 - val_loss: 9.1362e-06 - val_acc: 1.0000
Epoch 7/10
9830/9830 [==============================] - 6s 638us/sample - loss: 3.4653e-05
- acc: 1.0000 - val_loss: 9.3589e-06 - val_acc: 1.0000
Epoch 8/10
9830/9830 [==============================] - 7s 665us/sample - loss: 1.9094e-05
- acc: 1.0000 - val_loss: 5.0512e-07 - val_acc: 1.0000
Epoch 9/10
9830/9830 [==============================] - 7s 675us/sample - loss: 3.7043e-06
- acc: 1.0000 - val_loss: 1.5131e-06 - val_acc: 1.0000
Epoch 10/10
9830/9830 [==============================] - 7s 732us/sample - loss: 2.8497e-06
- acc: 1.0000 - val_loss: 1.1919e-06 - val_acc: 1.0000
12288/12288 - 4s - loss: 0.5391 - acc: 0.8884
test accuracy: 0.8884
```