AY19/20 Semester 2 : DSA4212 Python Tutorials

matplotlib

```
import matplotlib as plt

plt.plot(loss_history, "-.") # dotted-dash line
plt.plot(loss_history, "-x") # line with each point
    marked as 'x'
plt.plot(loss_history, "-s") # line with each point
    marked with a filled square
plt.plot(loss_history, "--x") # dotted lines with
    each point marked as 'x'

plt.plot(x_list, y_list, "r-", linewidth = 3) #
    thick red line
plt.plot(x_list, z_MLE, "g-", linewidth = 3,
    label="Reconstruction") # labelled thick green
    line
plt.plot(x_list, y_list, "bo", alpha = 0.5) # big
    blue filled dots, with some transparency
# black: 'k'

#  Axis scales
plt.yscale("log")

plt.grid(True)

# Axis labels
plt.xlabel("iteration")
plt.ylabel("loss")

# Plot 2 plots side by side
plt.figure(figsize = (10, 5))

plt.subplot(1, 2, 1)
plt.plot(hloss_history)
plt.xlabel("iter")
plt.ylabel("horizontal loss")
# Labels involving math symbols
plt.xlabel(r"$\mu$")
plt.ylabel(r"$\sigma$")
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(vloss_history)
plt.xlabel("iter")
plt.ylabel("vertical loss")
plt.grid(True)

# Plotting level lines
x = np.linspace(0, 2, num = 100)
y = np.linspace(0, 2.2, num = 100)
X, Y = np.meshgrid(x, y)
Z = func((X, Y))
plt.contour(X, Y, Z, colors='black')

# Scatterplot
plt.scatter(x_data, y_data)
plt.scatter(samples_gaussian[:,0],
    samples_gaussian[:,1], alpha=0.2, marker="+") #
    many points
plt.scatter(mu_MLE, sigma_MLE, c="red", marker="^",
    s=100, label="MLE") # one point (filled red
    triangle)
plt.scatter(mu_true, sigma_true, c="purple",
    marker="s", s=100, label="True") # one point
    (filled purple square)
```

```
# Plot multiple lines and points, label and with
    legend
plt.plot(x_list, z_true, "r--", label="Truth")
plt.plot(x_list, z_MLE, "g-", linewidth = 3,
    label="Reconstruction")
plt.plot(x_list, y_list, "bo", alpha=0.3,
    label="data")
plt.legend()
```

Reading in Data

```
# Read in csv file
path = "../datasets/"
file = "abalone.data"
pd.read_csv(os.path.join(path, file), sep = ",",
    header = None) # "../datasets/abalone.data"

# Read in image
M = imageio.imread("nus_logo.png").astype(float)
M = onp.mean(M, axis=2) / 255. # transform to
    greyscale
d = 100
M = resize(M, (d,d)) # resize
plt.imshow(M, cmap="gray") # display
```

Others

```
# Type of object
type(obj)

# Selecting rows and columns in pandas
attribute.iloc[:20000, [0,-1]] # first 20000 rows,
    first and last cols

# Similar to R which() for indices of True in
    boolean arrays
np.where(arr == 1)[0]

# Create a sequence of numbers
X = np.linspace(1.4, 1.9, num = 10) # R: seq(from,
    to, length)
Y = np.arange(2, 2.15, 0.02) # R: seq(from, to, by)

# Combine arrays pairwise
Z = np.array((X, Y)).T # X,Y same length

# Apply function over values in array
np.array(list(map(func, Z))) # over rows

# Sample random integers from a given range
randint(low, high, size) # inclusive end points
# Sample of random real numbers
np.random.rand(n)
# Sample values from given array with specified
    probabilities
np.random.choice(arr, 5, p = [0.5, 0.1, 0.1, 0.3],
    replace = False)

# Combine two arrays into one
np.array(arr1, arr2)

# Mean of values in array of arrays
np.mean(arr, axis = 0) # along cols
np.mean(arr, axis = 1) # along rows

# Mapping function over array(s)
def func(a, b, c):
    return a + b + c

A = np.arange(10) # [0, 1, 2, ..., 9]
B = np.arange(10)
C = np.arange(10)

mapped_func = jax.vmap(func, in_axes=(0, 0, 0)) #
    OR: in_axes(None, 0, 0)
mapped_func(A, B, C) # [ 0, 3, 6, 9, 12, 15, 18,
    21, 24, 27]

# Convert pandas Series object to NumPy array
obj.to_numpy(dtype="float64")
# Convert type of data in array of values
onp.array(x_data).astype(float)

# Concat NumPy arrays
A = np.array((1, 2, 3))
B = np.array((4, 5, 6))
C = np.array((7, 8, 9))

np.vstack((A, B, C)) # [ [1, 2, 3],
                     #   [4, 5, 6],
                     #   [7, 8, 9] ]
np.hstack((A, B, C)) # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Change map obj to array
np.array(list(map_obj)) # takes a while for large
    dim

# Set seed
import random
random.seed(n)

# Cumulative summation of an array of values
np.cumsum(arr)

# Mathematical functions
np.sin(theta)
np.cos(theta)

# Flatten a matrix (to a long vector)
x.flatten()
x.ravel() # faster than flatten()

# Distribution
onp.random.poisson(theta)
```

Mathematics

```
# Factorial, n!
math.factorial(n)
```

Logistic regression

Loss function: $\frac{1}{N}\sum_{i=1}^{N} log(1+exp(-y_i\langle\beta,x_i\rangle))$

.

Logistic probability: $P(y=1|x) = \frac{1}{1+exp(-\langle\beta,x\rangle)}$

.

Discussed in Tutorial 3 Exercise 4 - *fit logistic regression (i.e.find β) with basic gradient descent by minimising loss function*.

Download the fashion dataset and load them into a jupyter notebook. The labels $+1$ and -1 correspond to whether a piece of clothing has long or short sleeve. Implement a logistic regression on the training dataset and train it with a basic gradient descent algorithm. Evaluate the accuracy of your model on the test dataset. **Remark:** It may be useful to learn how to use the `vmap` function from JAX.

```
# load libraries
import jax.numpy as np
import numpy as onp
import os as os
import pylab as plt
import h5py # to read hdf5 files

# load datasets
path = "../datasets/"

filename = "fashion_img_training.h5" # 10 000
    samples, each of size 28x28 (train set)
fashion_img_training =
    h5py.File(os.path.join(path,
    filename),'r')['training_images'][:]

filename = "fashion_label_training.h5" # array of
    1, -1s
fashion_label_training =
    h5py.File(os.path.join(path,
    filename),'r')['training_labels'][:]

filename = "fashion_img_val.h5" # 2 000 samples
    (validation set)
fashion_img_val = h5py.File(os.path.join(path,
    filename),'r')['val_images'][:]

filename = "fashion_label_val.h5"
fashion_label_val = h5py.File(os.path.join(path,
    filename),'r')['val_labels'][:]

# plot a sample of images to view
plt.figure(figsize=(15,1))
for k in range(15):
    plt.subplot(1,15,k+1)
    plt.imshow(fashion_img_training[k],
        cmap="gray")
    plt.title(str(fashion_label_training[k]))
    plt.axis("off")
```

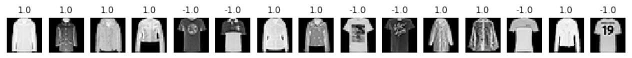


Figure 1: Long sleeve (1.0), short sleeve (-1.0)

```
# define probability and loss functions
def proba_single(beta, x):
    """ beta is a vector of dimension 784, and x
        as well """
    proba = 1. / (1. + np.exp(-np.dot(beta, x)))
    return proba

# returns probabilities of 1.0 on a whole dataset
prediction_data = jax.vmap(proba_single,
    in_axes=(None,0))

def loss_single(beta,x,y):
    """
    beta: vector of dimension 784
    x: vector of dimension 794
    y: a number that equals -1 or 1
    """
    return np.log(1. + np.exp(-y * np.dot(beta,
        x)))

# compute the loss on. whole dataset
loss_dataset = jax.vmap(loss_single,
    in_axes=(None,0,0))

def loss(beta, data, y):
    """ compute the mean of all the individual
        losses """
    list_of_all_losses = loss_dataset(beta, data,
        y)
    return np.mean(list_of_all_losses)

def accuracy(beta, data, y):
    pred = prediction_data(beta, data)
    threshold = 0.5
    prediction_binary = (pred > 0.5).astype(int)
        #equals 1 if pred > threshold and 0
        otherwise
    prediction_sign = 2*(prediction_binary - 0.5)
        #equals 1 if pred > threshold and -1
        otherwise
    return np.mean(prediction_sign == y)

# initialise values
beta_init = onp.random.normal(0, 0.1, size=784)
grad_loss = jax.grad(loss, argnums=0)
grad_loss = jax.jit(grad_loss) #compile it to make
    it faster
beta = onp.copy(beta_init)

n_iter = 500
learning_rate = 0.1
loss_history = []
acc_train_list = []
acc_val_list = []

# gradient descent to find optimal beta, min loss
for k in range(n_iter):
    gradient = grad_loss(beta, data_train, y_train)
    beta = beta - learning_rate * gradient # update
    current_loss = loss(beta, data_train, y_train)
        # value to min
    accuracy_train = accuracy(beta, data_train,
        y_train)
    accuracy_val = accuracy(beta, data_val, y_val)

    acc_train_list.append(accuracy_train)
    acc_val_list.append(accuracy_val)
    # prints e.g. "Loss:2.014  Accuracy: 50.000% /
        50.850%"
    if k % 50 == 0:
        print("Loss:{0:.3f} \t
        Accuracy: {1:.3f}% /
            {2:.3f}%".format(current_loss, \
        100*accuracy_train,\
        100*accuracy_val))
    loss_history.append(current_loss)

# plot loss and accuracy of train and val sets
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.plot(loss_history)
plt.grid(True)
plt.title("Loss")
plt.xlabel("Iteration")

plt.subplot(1,2,2)
plt.plot(acc_train_list)
plt.plot(acc_val_list)
plt.grid(True)
plt.title("Accuracy")
plt.xlabel("Iteration")
```

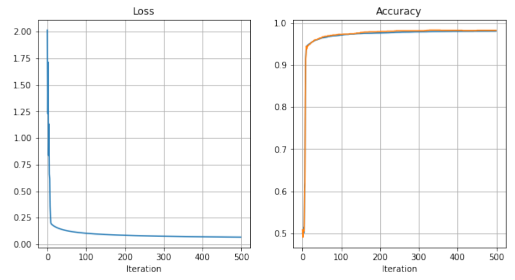


Figure 2: Loss and accuracy of basic GD

Highlight beta weights of pixels which are more than 0.2, notice that these are mostly focused on the sleeve area.

```
plt.figure(figsize=(15,1))
for k in range(15):
    plt.subplot(1,15,k+1)
    plt.imshow(fashion_img_training[k],
        cmap="gray")
    plt.imshow(beta.reshape((28,28)) > 0.2,
        alpha=0.7)
    plt.title(str(fashion_label_training[k]))
    plt.axis("off")
```

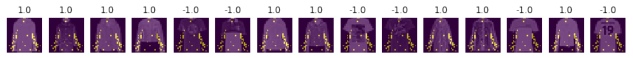


Figure 3: Beta weights with more than 0.2

It is good to try and intepret the model. Beta β can be randomly initialised with a small variance (e.g. 0.1) with mean 0. Logistic regression is convex optimisation, so its easy in this case even with high dimension.

Regularisation

→ Ridge regression

Minimise the function

$$L(\beta) = \frac{1}{N}\sum_{i=1}^{N} log[1 + e^{-y_i\langle x_i,\beta\rangle}] + \frac{1}{N}||\beta||^2$$

.

Discussed in Tutorial 5 - *minimise logistic function with ridge regularisation using stochastic gradient descent*.

→ LASSO

Minimise the function

$$L(\beta) = \sum_{i=1}^{N} log[1 + e^{-y_i\langle x_i,\beta\rangle}] + \lambda||\beta||$$

.

Gradient Descent

The optimisation problem and goal is to **minimise** or **maximise** a function.

Naive gradient descent

$$x_{n+1} = x_n - \eta\nabla f(x_n)$$

.

Discussed in Tutorial 3 Exercise 3 (Rosenbrock function) - *minimise given function with gradient descent*.

Consider the function

$$F(x, y) = (x-1)^2 + 10(y-x^2)^2$$

Context: The Rosenbrock function is **non-convex** and often used as test problem for optimisation algorithms to see if it works well. The general form of the function is

$$F(x, y) = (a-x)^2 + b(y-x^2)^2$$

with a global minimum of 0 at (a, a^2). Usually parameters are set such that $a = 1$ and $b = 100$. When $a = 0$, the function is symmetric and the minimum is at the origin.

1. Implement a basic gradient descent algorithm to minimise F.

```
# import packages
import jax.numpy as np
import jax
import matplotlib as plt

# define rosenbrock function to minimise
def func(x):
 """
 x = (x[0], x[1])
 """
 return (x[0]-1)**2 + 10*(x[1] - x[0]**2)**2

# initialise parameters
```

```
x_init = np.array([2., 2.]) # x0
grad_func = jax.grad(func)
n_iter = 100
learning_rate = 0.01

loss_history = [] # monitor changes in func(x)
    values
x1_history = [] # monitor changes in x values
x2_history = []
x = x_init[:]

for k in range(n_iter):
 gradient = grad_func(x)
 x = x - learning_rate * gradient
 x1_history.append(x[0])
 x2_history.append(x[1])
 loss_history.append(func(x))

# plot function values
plt.plot(loss_history, "-.") # dotted-dash line
plt.yscale("log")
plt.grid(True)
plt.xlabel("iteration")
plt.ylabel("Loss")
```

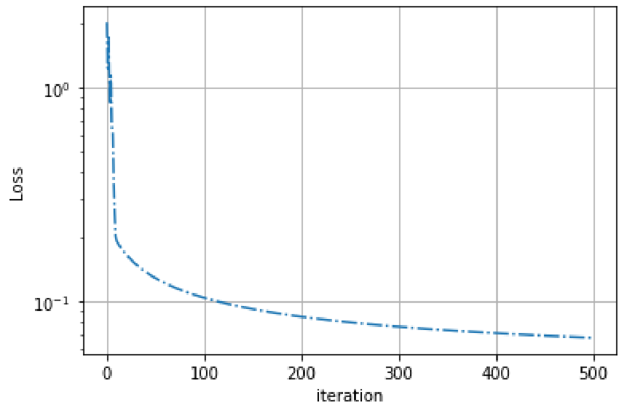


Figure 4: Plot of x_n across iterations

2. Find a way to visualise the trajectory of the algorithm, as well as the level lines of the function F.

```
x = np.linspace(0, 2, num = 100)
y = np.linspace(0, 2.2, num = 100)
X, Y = np.meshgrid(x, y)
Z = func((X, Y))

plt.plot(x1_history, x2_history) # trajectory of x
    values
plt.contour(X, Y, Z, colors='black') # level lines
plt.plot(1, 1, "ro") # true minimum
```

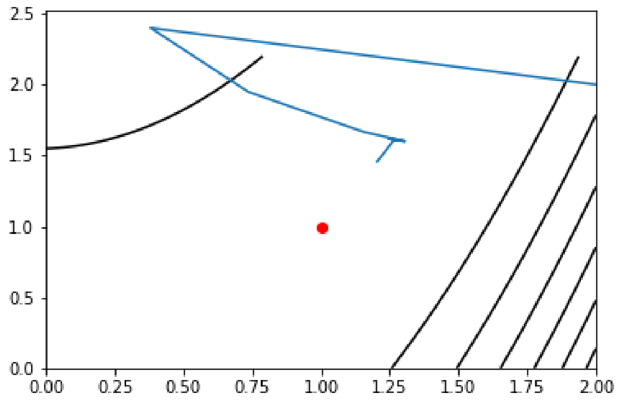


Figure 5: Trajectory of algorithm

Stochastic gradient descent

Refer to section *Momentum* for code example.

Descent algorithm

$$x_{n+1} = x_n + \eta_n d_n$$

.

where descent direction $d = -S\nabla F(x)$

s.t. $\langle d, \nabla F(x) \rangle < 0$

for any PD matrix S (where $S = I_d$ for naive GD)

$\rightarrow$ Backtracking

Set $\alpha \in (0, 1)$ (typically 0.5), find step size $\eta_n > 0$ such that

Backtracking cond: $F(x_n + \eta d_n) < F(x_n) + \alpha\eta\langle\nabla F(x_n), d_n\rangle$

.

Algo: Start from initial guess η, keep dividing it by 2 until condition satisfied (use this η to do update). .

A variant of Tutorial 3 Exercise 3 (Rosenbrock function) - *minimise given function with naive gradient descent **plus** backtracking*.

```
x_init = np.array([5.,5.])
grad_func = jax.jit( jax.grad(func))

init_learning_rate = 1.0 # too large may not
    decrease fn in each step
n_iter = 100
alpha = 0.5 # does not change
x = x_init
loss_history = []
learning_rate_history = []

for k in range(n_iter):
    # compute a descent direction
    gradient = grad_func(x)
    descent = -gradient
```

```
        # keep reducing the learning rate until
          backtracking condition is satisfied
        # for each iteration, learning rate starts
          again from init value
        learning_rate = init_learning_rate
        while func(x + learning_rate * descent) >
          func(x) +
          alpha*learning_rate*np.dot(gradient,
          descent):
            learning_rate = learning_rate / 2.

        # actually do the update
        x = x + learning_rate * descent

        # save the history of loss / learning-rates
        loss_history.append(func(x))
        learning_rate_history.append(learning_rate)

    # plot
    plt.figure(figsize=(10,5))

    plt.subplot(1,2,1)
    plt.plot(loss_history, "-s")
    plt.yscale("log")
    plt.title("Loss vs iteration")
    plt.grid(True)


    plt.subplot(1,2,2)
    plt.plot(learning_rate_history)
    plt.title("Learning Rate vs iterationr")
    plt.grid(True)
```

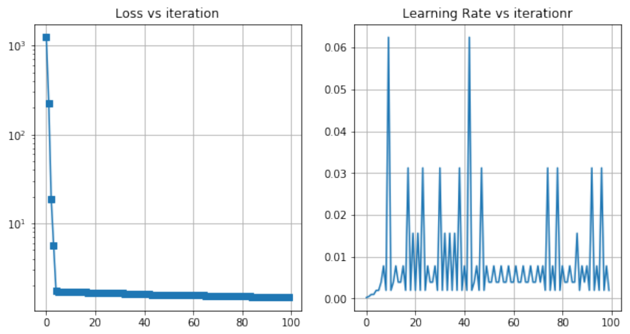


Figure 6: Loss and learning rate (Version 1)

OR: a slightly different implementation (con't from previous iteration)

```
for k in range(n_iter):
    # compute a descent direction
    descent = -grad_func(x)
    gradient = grad_func(x)

    # keep reducing the learning rate until
      backtracking condition is satisfied
    # *DIFFERENT*
    # learning rate continues from previous
      iteration
    learning_rate = learning_rate*3 # prevent
      value from becoming too small
    while func(x + learning_rate * descent) >
      func(x) +
      alpha*learning_rate*np.dot(gradient,
      descent):
        learning_rate = learning_rate / 2.

    # actually do the update
    x = x + learning_rate * descent

    loss_history.append(func(x))
    learning_rate_history.append(learning_rate)
```

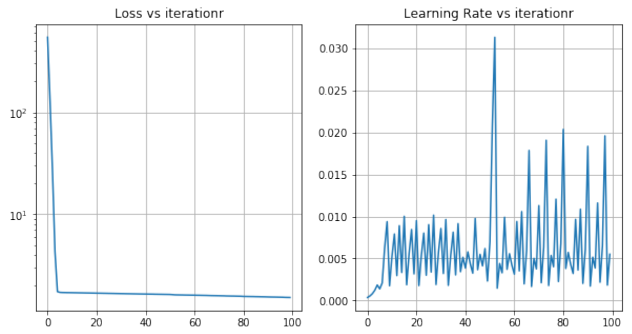


Figure 7: Loss and learning rate (Version 2)

OR: Automatic backtracking in logistic regression model

Discussed in Tutorial 4 Exercise 3 (Backtracking) - *logistic regression with backtracking*.

Consider the logistic regression model used in Tutorial 3 for differentiating between two types of clothes. Re-implement this algorithm using the automatic backtracking approach for choosing the gradient step-size.

```
n_iter = 100
learning_rate = 10.
loss_history = []
acc_train_list = []
acc_val_list = []
alpha = 0.5

for k in range(n_iter):
    # descent direction
    gradient = grad_loss(beta, data_train, y_train)
    descent_direction = -gradient

    # adaptive learning rate
    learning_rate = learning_rate * 3
    current_loss = loss(beta, data_train, y_train)
    while loss(beta + learning_rate *
        descent_direction, data_train, y_train) >
        current_loss +
        alpha*learning_rate*np.dot(descent_direction,
     gradient):
    learning_rate = learning_rate / 2.

#actually do the update
beta = beta + learning_rate * descent_direction

current_loss = loss(beta, data_train, y_train)
accuracy_train = accuracy(beta, data_train,
     y_train)
accuracy_val = accuracy(beta, data_val, y_val)

acc_train_list.append(accuracy_train)
acc_val_list.append(accuracy_val)
if k % 10 == 0:
    print("Loss:{0:.3f} \t Accuracy: {
    1:.3f}% / {2:.3f}%".format(current_loss, \
    100*accuracy_train,\
    100*accuracy_val))
loss_history.append(current_loss)
```

→ Exact line search

Find step-size η_n such that

$$F(x_n + \eta_n d_n) = min_{\eta>0} F(x_n + \eta d_n)$$

Learning rate

→ Fixed

In each update of the parameter (e.g. $\mathbf{x}, \beta$), learning rate is fixed and remains the same.

→ Adaptive

1. Backtracking (find suitable step-size with backtracking condition)

Momentum

$$x_{n+1} = x_n - \eta\nabla F(x_n) + \beta(x_n - x_{n-1})$$

for a fixed $\beta > 0$ (usually 0.9) where we go in roughly the same direction as before.

Discussed in Tutorial 5 (Stochastic GD with momentum) - *fit regularised logistic regression of fashion dataset with stochastic GD with momentum*

2. Compute the Hessian for an arbitrary vector β and check that it is positive definite.

```
# random initialization
n_training_data, D = data_train.shape
beta_init = onp.random.normal(0, scale =
     1./onp.sqrt(D), size=D)

# visualise the Hessian
H = hess_loss(beta_init, data_train, y_train)
eigenvalues, _ = onp.linalg.eigh(H) # all positive
plt.imshow(H)
```

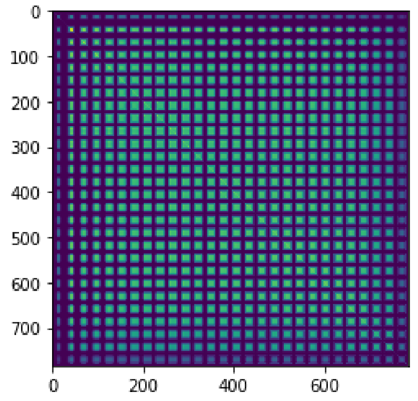


Figure 8: Hessian heatplot

Use a moving average of stochastic gradient estimates. For a moving average of $m \approx 0.9$, compute the approximated gradient to use in each update as

$$g_k = m g_{k-1} + (1-m)\hat{\nabla} L(\beta)$$

This is SGD of logistic regression with momentum and adaptive learning rate.

```
# initialise values
n_epoch = 50
batch_size=100 # size of the mini-batch (SGD)
beta = onp.copy(beta_init)
mov_avg_param = 0.95

learning_rate = 0.1
loss_train_history = []
loss_val_history = []

acc_train_history = [] # computed at the end of
     each epoch
acc_val_history = []

# initialize gradient
gradient = grad_loss(beta, data_train, y_train)

start = time.time()
for epoch in range(n_epoch):
    # divide learning rate by 2 every 10 epoch
    if epoch % 10 == 0 and epoch >= 1:
        learning_rate = learning_rate/2.

    for k in range(n_training_data // batch_size):
        # number of batch per epoch

        # 1st option
        #============
        # select at random "batch_size" random data
            point
        # index_ =
            onp.random.choice(train_data_size,
            batch_size, replace=False)

        # 2nd option
        #============
        # consider the batches sequentially
        index_ = onp.arange(k*batch_size,
             (k+1)*batch_size) % n_training_data
        gradient_local = grad_loss(beta,
             data_train[index_], y_train[index_])

        # moving average of the gradient ***
        gradient = gradient * mov_avg_param + (1 -
             mov_avg_param) * gradient_local

        # gradient descent update
        beta = beta - learning_rate * gradient

    # before moving on to next epoch
    accuracy_train = accuracy(beta, data_train,
         y_train)
    accuracy_val = accuracy(beta, data_val, y_val)
    acc_train_list.append(accuracy_train)
    acc_val_list.append(accuracy_val)

    loss_train = loss(beta, data_train, y_train)
    loss_val = loss(beta, data_val, y_val)
    loss_train_history.append(loss_train)
    loss_val_history.append(loss_val)

    timing = time.time() - start
    print("epoch:{0:.0f} \t time:{1:.0f} \t
      Loss(train):{2:.3f} \t Loss(val):{3:.3f}"
      .format(epoch, timing, loss_train, loss_val))

# plotting
plt.plot(loss_train_history, label="loss(train)")
plt.plot(loss_val_history, label="loss(val)")
plt.legend()
plt.grid(True)
plt.xlabel("epoch")
plt.title("Stochastic Gradient Descent")
```

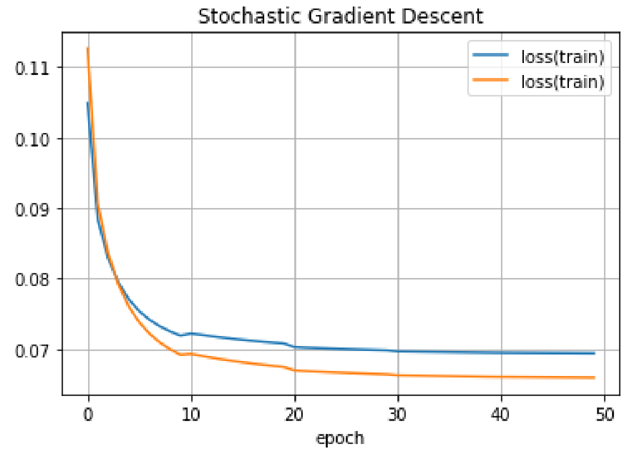


Figure 9: Higher curve (train), lower curve (val)

Pre-conditioning

$$x_{k+1} = x_k - \eta_k P_k \nabla F(x_k)$$

OR: $x_{k+1} = x_k - P_k d_k$

Descent direction: $d_n = P \nabla F(x_n)$

.

Discussed in Tutorial 4 Exercise 4 (Pre-conditioning) - *minimise given function with preconditioned gradient descent with backtracking*

Consider a positive definite symmetric matrix $S \in \mathbb{R}^d$, $d = 100$, and the quadratic function

$$F(x) = (1/2)\langle x, Sx \rangle$$

1. What is the minimum of the function F? *Ans: 0, when x is 0 and given $S > 0$*

2. In this exercise, we will take $S = C^{10}$ where C is the empirical covariance matrix of $N = 100$ random Gaussian vectors. Implement this in Python.

```
# Find cov mat C
N = 300
d = 100
gaussian_samples = onp.random.normal(0, 1, size =
     (N, d))
empirical_cov = onp.cov(gaussian_samples.T) # cols
     are obs
S = onp.copy(empirical_cov)
# for C^10
for k in range(9):
    S = S @ empirical_cov
```

3. Implement gradient descent with backtracking to minimise F. *Ans: refer to backtracking*

4. Implement the Preconditioned Gradient Descent (PGD) with backtracking.

```
def func(x):
    return 0.5 * np.dot(x, np.dot(S, x))
grad_func = jax.jit(jax.grad(func))

# PGD
def run_gradient_descent(P, n_iter,
     learning_rate_init, x_init):
    """
    P: preconditioning matrix
    n_iter: number of iteration
    learning_rate_init: initial learning rate
    x_init: initial guess
    """

    loss_history = []
    alpha = 0.5
    learning_rate = learning_rate_init

    x = onp.copy(x_init)
    for k in range(n_iter):
        # descent direction
        gradient = grad_func(x)
        descent_direction = -P @ gradient # NOTE:
             not gradient
        # adaptive learning rate (backtracking)
        learning_rate = learning_rate * 3
        current_loss = func(x)
        while func(x + learning_rate *
            descent_direction) > current_loss +
            alpha*learning_rate*np.dot(descent_direction,
            gradient):
            learning_rate = learning_rate / 2.

        # actually do the update
        x = x + learning_rate * descent_direction
        current_loss = func(x)
        loss_history.append(current_loss)

    return loss_history
```

5. Implement PGD with pre-conditioning matrices $P = S, P = I_d$ and $P = S^{-1}$. Which one converges faster?

```
n_iter = 100
learning_rate_init = 10.
x_init = onp.random.normal(0,1,size=d)

# 1) P = S
P = onp.copy(S)
loss_history_P = run_gradient_descent(P, n_iter,
    learning_rate_init, x_init)

# 2) P = Id (basic GD)
P = onp.eye(d) # identity matrix
loss_history_identity = run_gradient_descent(P,
    n_iter, learning_rate_init, x_init)

# 3) P = inverse S
eps = 0.001
P = onp.linalg.inv(S + eps*onp.eye(d)) # make sure
    matrix invertible, add to eigenvalues, else
    difficult to converge
loss_history_P_inv = run_gradient_descent(P,
    n_iter, learning_rate_init, x_init)
```

```
# plot
plt.plot(loss_history_identity, label="naive")
plt.plot(loss_history_P, label="C")
plt.plot(loss_history_P_inv, label="C_inv")
plt.legend()

plt.grid(True)
plt.yscale("log")

plt.title("Comparison Preconditionning Matrix")
plt.ylabel("log loss")
plt.xlabel("Iteration")
```

Choice of pre-conditioning matrix does matter, it can speed up the algorithm dramatically.

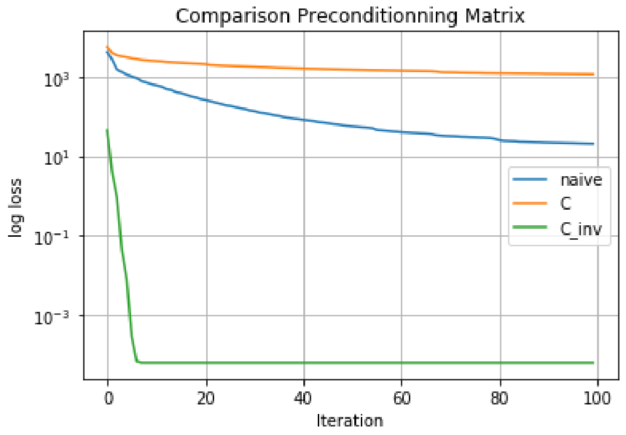


Figure 10: Comparison of pre-conditioning matrices

Newton method

$$x_{k+1} = x_k - \eta_k H_k^{-1} \nabla F(x_k)$$

.
An alternative to computing inverse of Hessian is to solve the equation to find the descent direction .

$$H_k d_k = -\nabla F(x_k)$$

Discussed in Tutorial 5 - *fit regularised logistic regression of fashion dataset with Newton- algorithm.*

4. Implement a Newton-algoritm (~~with backtracking~~) for fitting the model.

```
# functions
...
grad_loss = jax.jit(jax.grad(loss))
hess_loss = jax.jit(jax.hessian(loss))
...

# random initialization
n_training_data, D = data_train.shape
beta_init = onp.random.normal(0, scale =
    1./onp.sqrt(D), size=D)

# initialise values
n_iter = 20
learning_rate = 0.9
loss_history = []
beta = onp.copy(beta_init)

for k in range(n_iter):
    # compute the gradient
    gradient = grad_loss(beta, data_train, y_train)

    # compute the Hessian
    H = hess_loss(beta, data_train, y_train)

    # no need to compute the inverse of H
    # better is to solve the system ***
    # H*d = -gradient
    descent_direction = onp.linalg.solve(H,
        -gradient)
    # update estimate of the minimum
    beta = beta + learning_rate * descent_direction

    # save the loss for later plotting
    current_loss = loss(beta, data_train, y_train)
    loss_history.append(current_loss)

plt.plot(loss_history, "r-s")
plt.title("Newton Descent")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.grid(True)
```

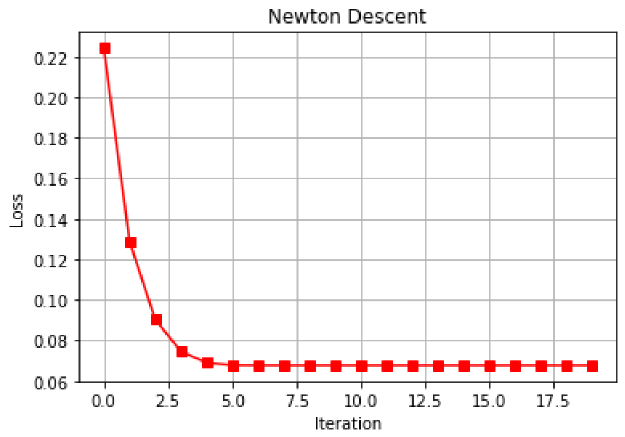


Figure 11: Trajectory of loss using Newton Descent across iterations

Quasi-Newton methods

Approximate Hessian $\mathbf{H}$ with $B_k \approx Hess(x_k)$. by solving for descent direction d_k

$$B_k d_k = -\nabla F(x_k)$$

.

Important properties of approximated Hessian B_k:

1. Quasi-Newton methods can be applied for any function .
2. B_k guaranteed to be PD (even if function not convex).
3. Finding the approximated inverse Hessian B_k^{-1} is fast (in $O(p^2)$ time).

For help in Python, look at `scipy.optimize.minimize`?

1. 'BFGS'
2. 'L-BFGS-B' : low memory item 'Nelder-Mead' : when you don't know how to compute the gradient

Gauss-Newton methods

Used on functions which can be expressed as

$$F(x) = \ell(G(x))$$

where

$G : \mathbb{R}^P \to \mathbb{R}$ is a complicated function where only gradient G' can be computed and

$\ell : \mathbb{R} \to \mathbb{R}$ is a simple scalar function where first and second derivatives ℓ', ℓ'' can be computed.

.

Gauss-Newton approximation to Hessian: .

$$\hat{H}(x) \approx \ell''(G(x))\nabla G(x)\nabla G^T(x)$$

.

Differences of Gauss-Newton methods with Quasi-Newton methods:

1. Can only work on composite functions (i.e. expressed in the form $\ell(G(x))$.
2. Faster (I think) at approximating Hessian of function $F(x)$.

$\rightarrow$ LBFGS

LBGFS is short for Low-memory Broyden-Fletcher-Goldfarb-Shanno algorithm developed in 1973. **Note:** Using L-BFGS in Python, you may need to transform back JAX gradients into usual numpy vectors before feeding it to the L-BFGS function. **Also, looks like L-BFGS can only optimise a vector so flatten your images (arr of arr) and reshape it back in the loss function.**

Discussed in Tutorial 5 Example - *how to use LBFGS with scipy*.

```
# define function to minimise
def func(x):
    """ F: R^p --> R"""
    return np.mean( np.sin(x)**2 )

# jax gradient
```

```
grad_func = jax.jit(jax.grad(func))

# create a wrapper, ensure the gradient output is
    standard numpy gradient
# should be faster than normal gradient descent
def gradient_wrapper(x):
    """ compute the gradient of func at x and make
        sure that the output is a numpy array"""
    return onp.array(grad_func(x))

# initialise values
dim = 100
x_init = onp.random.normal(0, 1 , size = dim)

# optimisation
loss_history = [] # save the loss trajectory
time_history = [] # save the compute time

def save_traj(beta):
    """ a function that saves a few statistics for
        later analysis"""
    loss_history.append(func(beta))
    time_history.append(time.time() - start)

start = time.time()
traj = scipy.optimize.minimize(
  fun = func, #function to minimize
  x0 = onp.array(x_init), #initial guess
  method = 'L-BFGS-B',
  jac = gradient_wrapper, # function that computes
      the gradient
  callback = save_traj, # function used to save
      some results for later plotting
  options = {"maxiter": 100}, # maximum number of
      iterations
  #tol=10**-8
)
traj.x # IMPORTANT (optimal value found)

# plot results
plt.plot(time_history, loss_history, "r-s")
plt.xlabel("time (sec)")
plt.ylabel("loss")
plt.title("L-BFGS")
plt.grid(True)
```

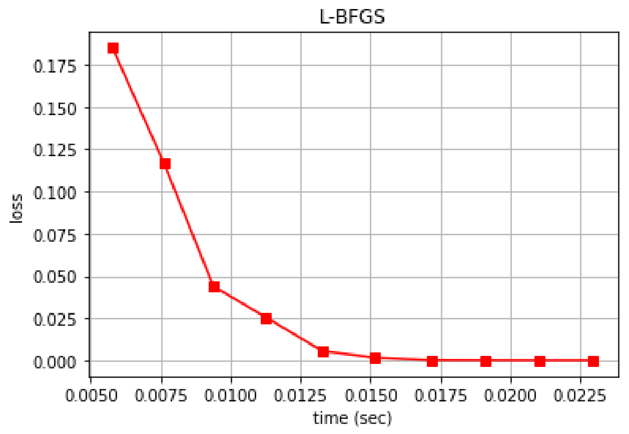
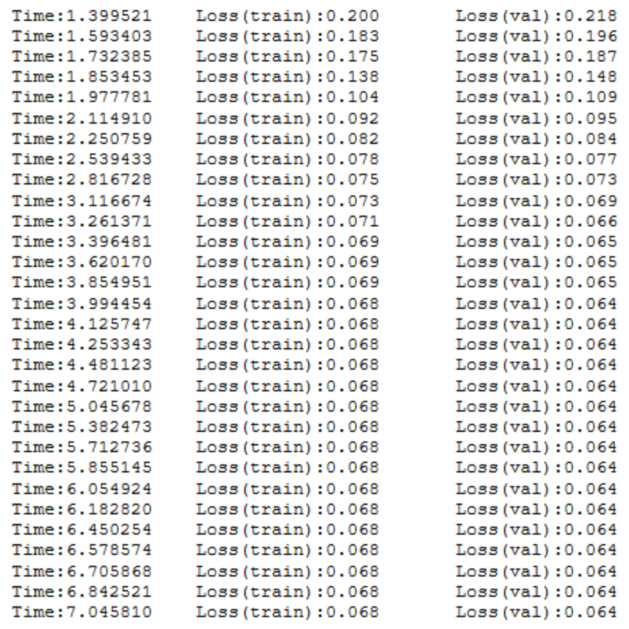


Figure 12: Trajectory of loss using L-BFGS across iterations (in terms of time)

OR: Discussed in Tutorial 5 - *fit regularised logistic regression of fashion dataset with LBFGS*.

5. Use the scipy implementation of L-BFGS for fitting the model.

```
# define function and gradient
def loss_wrapper(beta):
    """ only one argument """
    return loss(beta, data_train, y_train)

grad_loss_wrapper = jax.jit(jax.grad(loss_wrapper))

def gradient_wrapper(beta):
    """output is a numpy array"""
    return onp.array(grad_loss_wrapper(beta))

# optimise
loss_train_history = [] # save the loss trajectory
loss_val_history = []
time_history = [] # save the compute time

def save_traj(beta):
    """ a function that saves a few statistics for
        later analysis"""
    loss_train = loss(beta, data_train, y_train)
    loss_val = loss(beta, data_val, y_val)
    timing = time.time() - start
    print("Time:{0:3f} \t Loss(train):{1:.3f} \t
      Loss(val):{2:.3f}".format(timing, loss_train,
          loss_val))
    time_history.append(timing)
    loss_train_history.append(loss_train)
    loss_val_history.append(loss_val)

# run LBFGS
start = time.time()
traj = scipy.optimize.minimize(
 fun = loss_wrapper, # function to minimize
 x0 = onp.array(beta_init), # initial guess
 method = 'L-BFGS-B', # use L-BFGS
 jac = gradient_wrapper, # function that computes
     the gradient
 callback = save_traj, # function used to save
     some results for later plotting
 options={"maxiter":100}) # maximum number of
     iterations
```



```
Time:1.399521     Loss(train):0.200       Loss(val):0.218
Time:1.593403     Loss(train):0.183       Loss(val):0.196
Time:1.732385     Loss(train):0.175       Loss(val):0.187
Time:1.853453     Loss(train):0.138       Loss(val):0.148
Time:1.977781     Loss(train):0.104       Loss(val):0.109
Time:2.114910     Loss(train):0.092       Loss(val):0.095
Time:2.250759     Loss(train):0.082       Loss(val):0.084
Time:2.539433     Loss(train):0.078       Loss(val):0.077
Time:2.816728     Loss(train):0.075       Loss(val):0.073
Time:3.116674     Loss(train):0.073       Loss(val):0.069
Time:3.261371     Loss(train):0.071       Loss(val):0.066
Time:3.396481     Loss(train):0.069       Loss(val):0.065
Time:3.620170     Loss(train):0.069       Loss(val):0.065
Time:3.854951     Loss(train):0.069       Loss(val):0.065
Time:3.994454     Loss(train):0.068       Loss(val):0.064
Time:4.125747     Loss(train):0.068       Loss(val):0.064
Time:4.253343     Loss(train):0.068       Loss(val):0.064
Time:4.481123     Loss(train):0.068       Loss(val):0.064
Time:4.721010     Loss(train):0.068       Loss(val):0.064
Time:5.045678     Loss(train):0.068       Loss(val):0.064
Time:5.382473     Loss(train):0.068       Loss(val):0.064
Time:5.712736     Loss(train):0.068       Loss(val):0.064
Time:5.855145     Loss(train):0.068       Loss(val):0.064
Time:6.054924     Loss(train):0.068       Loss(val):0.064
Time:6.182820     Loss(train):0.068       Loss(val):0.064
Time:6.450254     Loss(train):0.068       Loss(val):0.064
Time:6.578574     Loss(train):0.068       Loss(val):0.064
Time:6.705868     Loss(train):0.068       Loss(val):0.064
Time:6.842521     Loss(train):0.068       Loss(val):0.064
Time:7.045810     Loss(train):0.068       Loss(val):0.064
```

Figure 13: Loss using L-BFGS across time

Projected subgradient descent (Constrained optimisation)

Where subgradient $g_n \in \partial f(x_n)$, .

$$\underline{x_{n+1} = x_n - \eta_n g_n}$$

$$\underline{x_{n+1} = \prod_{\omega}(x_n - \eta_n g_n)} \text{ (projected)}$$

.

For a **subgradient** of f at $x \in \mathbb{R}^p$,
for subgradient vector g, for any vector $\epsilon \in \mathbb{R}^p$,

$$\underline{f(x+\epsilon) \geq f(x) + \langle g, \epsilon \rangle}$$

.

For a **subdifferential** of f at $x \in \mathbb{R}^p$,

$$\underline{\partial f(x) = \{g \in \mathbb{R}^p : g \text{ is a subgradient to } f \text{ at } x\}}$$

.

Note:

1. There exists many subgradients to function f at $x \in \mathbb{R}^p$.
2. A **subdifferential** of f at x is the set of all possible subgradients to f at $x \in \mathbb{R}^p$.
3. For **convex function**, subdifferential at $x$is always a non-empty and bounded set.
4. For **convex and <u>differential at</u>** x function, subdifferential contains only a single vector (i.e. gradient of function at x, $\partial f(x) = \{\nabla f(x)\}$).
5. Subgradient descent even works for non-differentiable convex function.

→ L2 ball constraint

Discussed in an example after Tutorial 6.
Minimise the function

$$F(x_1, x_2) = (x_1 - 3)^2 + (x_2 - 3)^2$$

under the constraint that $(x_1, x_2) \in \omega$, where ω is a ball of radius $\mathbf{R} = 2$.

```python
def func(x):
    """ basic function """
    return (x[0]-3.)**2 + (x[1]-3.)**2

# compute a gradient
grad_func = jax.grad(func)

# IMPORTANT (constraint was norm of vector to be
     less than or equal to 2)
def proj_ball(x, R):
    """ projection on the disk on radius R"""
    x_norm = np.linalg.norm(x)
    if x_norm <= R:
        return x
    else:
        return R * (x / x_norm) # on circumference
             of ball, same direction with unit
             vector
```

Continue with projected gradient descent. Save values before (after update) and after projection.

```python
n_iter = 100
eta = 0.1

# store trajectory
trajectory_x = []
trajectory_y = []

# initialise values
R = 2
x_init = np.array([3., -3.])
x = x_init

# projected gradient descent
for k in range(n_iter):
    # gradient update
    g = grad_func(x)
    x = x - eta*g

    # save it for plotting later
    trajectory_x.append(x[0])
    trajectory_y.append(x[1])

    # do projection
    x = proj_ball(x, R)

    # save it for plotting later
    trajectory_x.append(x[0])
    trajectory_y.append(x[1])
```

Plot trajectory of values.

```
# plot circle (constraint space)
theta = np.linspace(0, 10, 1000)
plt.plot(R*np.sin(theta), R*np.cos(theta), "b-")

# plot the trajectory
plt.plot(trajectory_x, trajectory_y, "r-")

# plot the point (3,3) which is the unconstrained
    minimum (optimal point if no constraint)
plt.plot([3], [3], "gs")
plt.grid(True)
plt.title("Projected Gradient Descent")
```

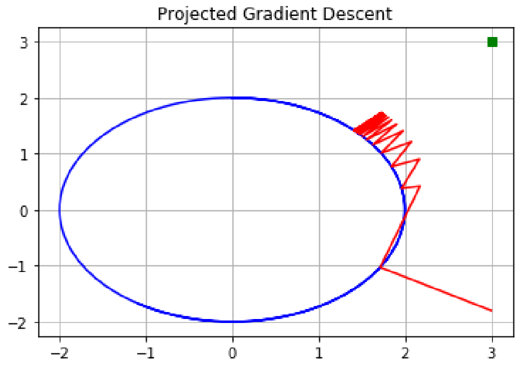


Figure 14: Trajectory of projected gradient descent with L2 ball as constrained subspace

$\rightarrow$ **Isotonic regression**

Discussed in Tutorial 6 Question 2.

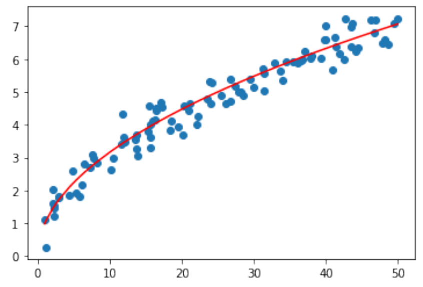
Consider the following synthetic dataset. Generate some random x_i and define

$$y_i = F(x_i) + (noise)$$

for an unknown increasing function $\mathbf{F}$. The goal of this exercise is to find an approximation of the (unknown) increasing function $\mathbf{F}$ from a set of samples $\{x_i, y_i\}_{i=1}^{N}$. In other words, this is again a regression problem, but this time with a constraint on the regression function.

```
n_data = 100
sd_noise = 0.4
x_data = onp.sort( 50 * onp.random.random(n_data)
    ) #generate and sort

def func(x):
 return onp.sqrt(x)

y_data = onp.sqrt(x_data) +
    onp.random.normal(loc=0, scale=sd_noise,
    size=n_data)
plt.scatter(x_data, y_data)
plt.plot(x_data, func(x_data), "r-")
```

First plot data with noise.



Our model again will be very simple. We will be looking for $z_1 \ldots, z_N$ such that

$$z_i \approx y_i$$

and such that $z_1 < z_2 < \ldots < z_N$. This means that z_i will represent the value of the unknown function $\mathbf{F}$ at location x_i,

$$z_i = F(x_i)$$

In order to simplify things, we will instead be looking for non-negative coefficients $\alpha_1, \ldots \alpha_N \geq 0$ and set

$$z_i = \alpha_1 + \ldots + \alpha_i$$

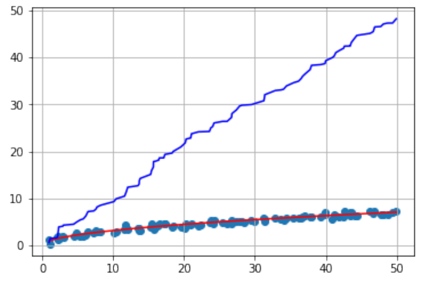
Putting this all together, it means that we will be minimising the function

$$L(\alpha) \equiv \sum_{i=1}^{N} (y_i - [\alpha_1 + \ldots + \alpha_i])^2$$

under the constraint that $\alpha_1, \ldots, \alpha_N \geq 0$. Carry out this minimisation and plot the resultant function.

Remark: You might want to use the function `jax.numpy.cumsum`

```
# Introduce all increments
alpha = onp.random.rand(n_data)
f = np.cumsum(alpha)

plt.scatter(x_data, y_data)
plt.plot(x_data, func(x_data), "r-")
plt.plot(x_data, f, "b-")
plt.grid(True)
```



```
def loss(alpha):
    """
    alpha: set of increments
    """
    # compute function itself
    f = np.cumsum(alpha)
    # compute MSE
    return np.mean((y_data - f)**2)

grad_loss = jax.jit(jax.grad(loss))

def projection_positive(alpha):
    """ projection on the set of vectors with all
        coordinates postive """
    alpha_projected = onp.copy(alpha)
    # set to zero all coordinates that are negative
    alpha_projected[alpha_projected < 0] = 0
    return np.array(alpha_projected) # make sure
        array is JAX array

# Implement projected gradient descent algorithm
n_iter = 5000
loss_history = []
learning_rate = 0.001

for k in range(n_iter):
    gradient = grad_loss(alpha)

    # 1) gradient descent step
    alpha = alpha - learning_rate * gradient

    # 2) projection step
    alpha = projection_positive(alpha)

    # save loss
    loss_history.append(loss(alpha))

plt.plot(loss_history)
plt.yscale("log")
plt.grid(True)
```

With optimised values of $alpha$ (i.e. increments), plot out fitted values with true values.

```
f_isotonic = np.cumsum(alpha)

plt.scatter(x_data, y_data, alpha = 0.5) # given
    data points
plt.plot(x_data, func(x_data), "r-") # true
    underlying function
plt.plot(x_data, f_isotonic, "b-", linewidth = 3)
```

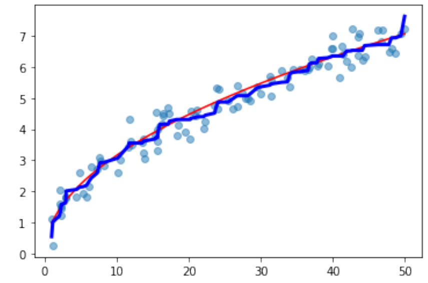


Figure 15: Isotonic fitted regression (blue line) with true function (red line) and given data points (scatterplot)

The goal was to recover the blue line given the data points.

Function Approximation

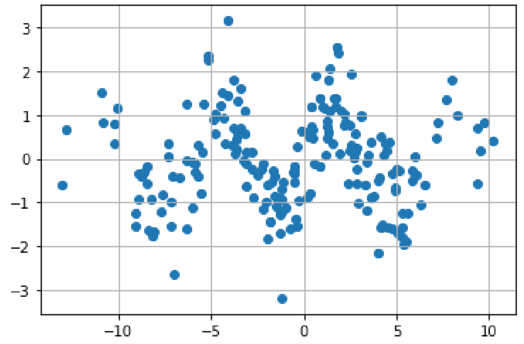
Discussed in Tutorial 6 Question 1 - given **mathematical formula** of function, approximate **parameter values of function**.

Consider the following synthetic dataset. Some x_i have been generated at random and defined

$$y_i = F(x_i) + (noise)$$

The goal of this exercise is to find an approximation of the (unknown) function $\mathbf{F}$ from a set of samples $\{x_i, y_i\}_{i=1}^{N}$. In other words, this is a regression problem.

```
# Initialise data
n_data = 200
sd_noise = 0.7
x_data = onp.random.normal(loc=0, scale=5,
    size=n_data)
# sine curve with normal noise
y_data = onp.sin(x_data) + onp.random.normal(loc=0,
    scale=sd_noise, size=n_data)
plt.scatter(x_data, y_data)
plt.grid(True)
```



The model used will be very simple. We would like to find a function of the type

$$f(x; c) = \sum_{i=1}^{N} c_i \exp\left\{-\frac{(x - x_i)^2}{2\ell^2}\right\}$$

for unknown coefficients $c = (c_1, c_2, \ldots c_N)$ For this exercise, you can take $\ell = 0.2$ for example. In other words you will be minimising the function

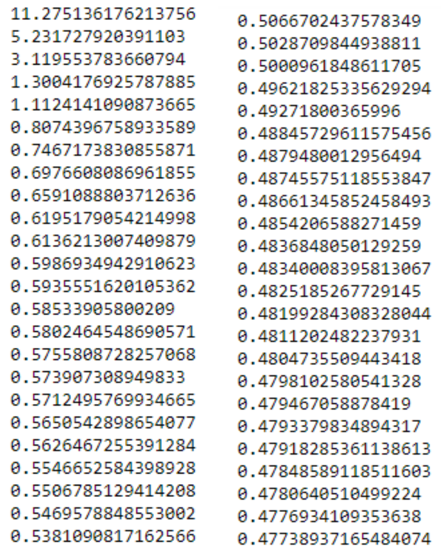
$$f(x; c) = \sum_{i=1}^{N} (y_i - f(x_i; c))^2$$

Carry out this minimisation and plot the resulting function $x \mapsto f(x; c\star)$.

```
#  Store unknown parameters in a vector of "c"of
    length n_data (i.e. c is like beta)
c = onp.random.normal(0, 1, size = n_data)

# Fix the lengthscale parameter "L"
LL = 1.

# Define function (NOT the function to minimise)
def func(x, c):
 return np.sum(c * np.exp( -(x - x_data)**2 /
     (2*LL) ))

# Mean squared loss function
def mse(c):
 # compute func(x_i) for each x_i and call the
     output z
 z = np.array([fun(x, c) for x in x_data])
 return np.mean((y_data - z)**2)

grad_mse = jax.grad(mse) # unfortunately, not
    "that" easy to JIT this function

# Wrapper to ensure output is the standard numpy
    gradient/array
def gradient_wrapper(c):
 return onp.array(grad_mse(c))
```

Minimise loss function with L-BGFS.

```
# Initialise guess
c_init = onp.random.normal(0, 1, size = n_data)

# Optimise with LBFGS
loss_history = [] # to save loss trajectory
def save_traj(c):
 curr_loss = mse(c)
 loss_history.append(curr_loss)
 print(curr_loss)

traj = scipy.optimize.minimize(
 fun = mse, # function to minimise
 x0 = onp.array(c_init) # initial guess
 method = 'L-BFGS-B',
 jac = gradient_wrapper, # fn to compute gradient
 callback = save_traj, # fn used to save some
     results for later plotting
 options = {"maxiter": 50}
)
```



```
11.275136176213756      0.5066702437578349
5.231727920391103       0.5028709844938811
3.119553783660794       0.5000961848611705
1.3004176925787885      0.49621825335629294
1.1124141090873665      0.49271800365996
0.8074396758933589      0.48845729611575456
0.7467173830855871      0.4879480012956494
0.6976608086961855      0.48745575118553847
0.6591088803712636      0.48661345852458493
0.6195179054214998      0.4854206588271459
0.6136213007409879      0.4836848050129259
0.598693494291O623      0.48340008395813067
0.593551620105362       0.4825185267729145
0.58533905800209        0.48199284308328044
0.580246454869O571      0.4811202482237931
0.5755808728257068      0.4804735509443418
0.573907308949833       0.4798102580541328
0.5712495769934665      0.479467058878419
0.5650542898654077      0.4793379834894317
0.5626467255391284      0.47918285361138613
0.5546652584398928      0.47848589118511603
0.5506785129414208      0.4780640510499224
0.5469578848553002      0.477693410935 3638
0.5381090817162566      0.47738937165484074
```

```
# final estimate
c_lbfgs = traj["x"]

# draw estimated function
x_list = onp.linspace(-15, 15, 1000)
plt.plot(x_list, [func(x, c_lbfgs) for x in
    x_list], "-r", linewidth = 3)

# superimpose given data
plt.scatter(x_data, y_data)
plt.grid(True)
plt.title("Function Fitting")
```

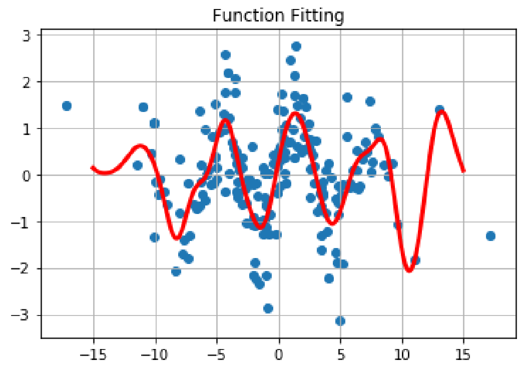


Figure 16: Approximated function with given data

Support Vector Machine (SVM)

Discussed in Tutorial 6 Question 3.

In this question, we will consider the dataset `abalone.data` available at https://archive.ics.uci.edu/ml/machine-learning-databases/abalone/.

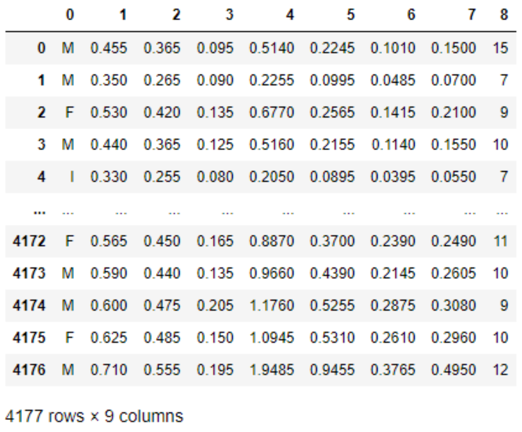
A SVM algorithm is a binary classifier similar to logistic regression and that can be described as follows: given training examples $\{x_i, y_i\}_{i=1}^N$ with covariate $x_i \in \mathbb{R}^p$ and binary response $y_i \in \{-1, +1\}$; the SVM minimises with loss function

$$L(\beta) = \sum_{i=1}^{N} max(0, 1 - y_i \langle \beta, x_i \rangle)$$

Once the SVM is fitted and the optimal parameter $\beta\star$ is found, the predicted value associated to a vector of covariates x simply equals the sign of the dot product $\langle \beta\star, x_i \rangle$.

1. Download and load the dataset.
2. Use a SVM classifier to distinguish male from female abalone. To do so, first get rid of the training examples that correspond to infants (I).
3. Use 70% of the dataset to train a SVM classifier and estimate the accuracy on the remaining 30%. (**Remark:** It is normal if the accuracy is not great)

```
# Load dataset
path = "../datasets/"
file = "abalone.data"
data = pd.read_csv(os.path.join(path, file), sep =
    ",", header = None)

data
```



	0	1	2	3	4	5	6	7	8
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.1500	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.0700	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.2100	9
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.1550	10
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.0550	7
...	...	...	...	...	...	...	...	...	...
4172	F	0.565	0.450	0.165	0.8870	0.3700	0.2390	0.2490	11
4173	M	0.590	0.440	0.135	0.9660	0.4390	0.2145	0.2605	10
4174	M	0.600	0.475	0.205	1.1760	0.5255	0.2875	0.3080	9
4175	F	0.625	0.485	0.150	1.0945	0.5310	0.2610	0.2960	10
4176	M	0.710	0.555	0.195	1.9485	0.9455	0.3765	0.4950	12

4177 rows × 9 columns

Figure 17: Quick look at `abalone` data

Form training and validation sets from given data.

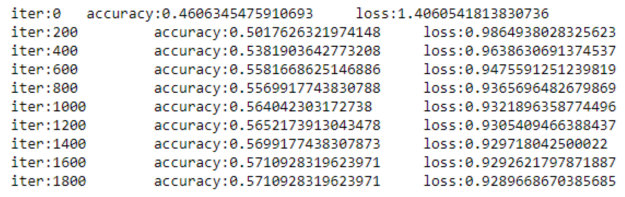
```
sex = data.values[:, 0] # rename column name
x_data = data.values[ (sex == "M") | (sex == "F"),
    1:] # select rows matching predicate, exclude
    first column (labels)
n, p = x_data.shape

# recale covariates x_i s
for k in range(p):
 # select a column with x_data[:, k]
 x_data[:, k] = (x_data[:, k] - onp.mean(x_data[:,
     k])) onp.std(x_data[:, k])

x_data = onp.array(x_data).astype(float)
y_data = sex.[(sex == "M") | (sex == "F")]
# Change labels to -1, +1
y_data[y_data == "M"] = 1.
y_data[y_data == "F"] = -1.
y_data = onp.array(y_data).astype(float)
```

Define loss functions, gradient and accuracy functions.

```
# Implement loss function for SVM
def loss_svm_single(beta, x, y):
 return np.maximum(0., 1. - y*np.dot(beta, x))

# Loss for entire dataset
loss_svm_all = jax.vmap(loss_svm_single, in_axes =
    (None, 0, 0))

# Global loss
def loss_svm(beta, x_batch, y_batch):
 return np.mean(loss_svm_all(beta, x_batch,
     y_batch))

# Gradient of loss
grad_svm = jax.jit(jax.grad(loss_svm))

def accuracy(beta, x_batch, y_batch):
 """
 proportion of correct predictions where prediction
     is 1 if the dot product is positive
 """
 return np.mean( (x_batch @ beta > 0) == (y_batch >
     0) )
```

Minimise loss function with gradient descent. Here, `x_data` is an array of arrays of size 2835 by 8.

```
# Random initialisation
beta_init = onp.random.normal(0, 1, size = p)

# Number of training data (70%)
n_train = int(len(x_data) * 0.7)

# Basic gradient descent (by default JAX computes a
    subgradient)
beta = onp.copy(beta_init)
learning_rate = 0.1
n_iter = 2000
accuracy_history = []
loss_history = []

for k in range(n_iter):
 gradient = grad_svm(beta, x_data[:n_train],
     y_data[:n_train]) # on train data
    beta = beta - learning_rate * gradient

        acc = accuracy(beta, x_data[n_train:],
              y_data[n_train:]) # on validation data
        loss = loss_svm(beta, x_data[:n_train],
              y_data[:n_train])
        loss_history.append(loss)
        accuracy_history.append(acc)
        if k % 200 == 0:
            print("iter:{} \t accuracy:{} \t
                loss:{}".format(k, acc, loss))
```



```
iter:0   accuracy:0.4606345475910693    loss:1.4060541813830736
iter:200         accuracy:0.5017626321974148     loss:0.9864938028325623
iter:400         accuracy:0.5381903642773208     loss:0.9638630691374537
iter:600         accuracy:0.5581668625146886     loss:0.9475591251239819
iter:800         accuracy:0.5569917743830788     loss:0.9365696482679869
iter:1000        accuracy:0.564042303172738      loss:0.9321896358774496
iter:1200        accuracy:0.5652173913043478     loss:0.9305409466388437
iter:1400        accuracy:0.5699177438307873     loss:0.929718042500022
iter:1600        accuracy:0.5710928319623971     loss:0.9292621797871887
iter:1800        accuracy:0.5710928319623971     loss:0.9289668670385685
```

Figure 18: Accuracy and loss values in each iteration of GD

```
plt.figure(figsize=(10,3))
plt.subplot(1,2,1)
plt.plot(loss_history)
plt.title("Loss")
plt.grid(True)

plt.subplot(1,2,2)
plt.plot(accuracy_history)
plt.title("Test Accuracy")
plt.grid(True)
```

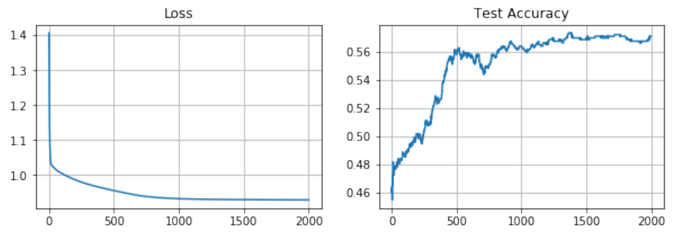


Figure 19: Accuracy and loss plots for SVM algorithm

Low Rank Approximation

Discussed in Tutorial 7 Question 3.

Load the image `nus_logo.png` and transform it into greyscale and consider the associated matrix $\mathbf{M} \in \mathbf{R}^{d,d}$. Let $r \geq 1$ be an integer. Find two matrices $\mathbf{U} \in d, r$ such that $\mathbf{M} \approx \mathbf{UV}$. To do so, one can try to minimise the function

$$L(\mathbf{U}, \mathbf{V}) = \frac{1}{d^2} ||\mathbf{M} - \mathbf{UV}||^2$$

Implement this optimisation with:

1. Gradient descent
2. L-BFGS
3. Alternate minimisation strategy

First read in the image.

```
import imageio
from skimage.transform import resize

# load in file
M = imageio.imread("nus_logo.png").astype(float)
# transform to grayscale
M = onp.mean(M, axis = 2) / 255.
# resize
d = 100
M = resize(M, (d, d))
```

```
#display
plt.imshow(M, cmap="gray")
```

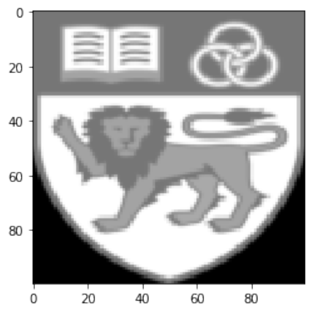


Figure 20: Read in image of NUS logo represented by a matrix of floats

Note that the larger the rank, the better the approximation while the lower the rank the lower the quality of the estimate since the image is being compressed.

→ Initialise values and setting up functions

Note that we have: $\mathbf{U} \in \mathbf{R}^{d,r}$ and $\mathbf{V} \in \mathbf{R}^{r,d}$

```
# Function and gradients
def loss(U,V):
    return np.mean( (M - U @ V)**2 )

# NOTE: how to do optimisation with 2 parameters
# differentiate gradient only with respect to
    parameter in question
grad_loss_u = jax.jit(jax.grad(loss, argnums = 0))
grad_loss_v = jax.jit(jax.grad(loss, argnums = 1))

# Init values
r = 20
U_init = onp.random.normal(0, 1, size = (d, r)) /
    onp.sqrt(d*r)
```

```
V_init = onp.random.normal(0, 1, size = (r, d)) /
     onp.sqrt(d*r)

plt.imshow(U_init @ V_init)
```

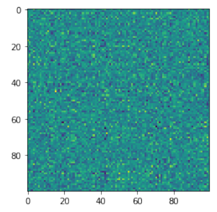


Figure 21: Initial matrix M

→ Gradient Descent

```
U = onp.copy(U_init)
V = onp.copy(V_init)

learning_rate = 10.
n_iter = 500
loss_history = []

for k in range(n_iter):
 gradient_u = grad_loss_u(U,V)
    gradient_v = grad_loss_v(U,V)

    # Update BOTH parameters in each iteration
    U = U - learning_rate * gradient_u
     V = V - learning_rate * gradient_v

    current_loss = loss(U,V)
    loss_history.append(current_loss)

plt.plot( loss_history )
plt.grid(True)
```

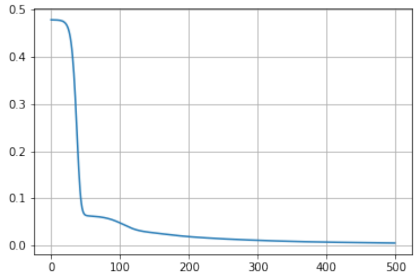


Figure 22: Loss plot (gradient descent)

```
plt.subplot(1,2,1)
plt.imshow(M, cmap="gray")

plt.subplot(1,2,2)
plt.imshow(U @ V, cmap="gray")
```

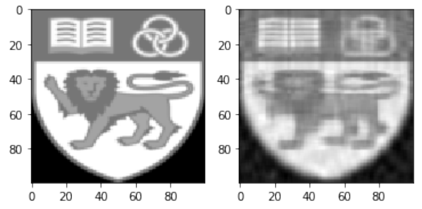


Figure 23: (Left) True image, (Right) Optimised image approx - gradient descent

→ Alternate Minimisation

Do alternate minimisation (i.e. minimise loss w.r.t. $\mathbf{U}$, then w.r.t. $\mathbf{V}$) with L-BFGS. The problem is non-convex in terms of being a joint function of U, V. Alternate minimisation is a faster alternative approach since the function is marginally convex.

First define functions that does a few steps of LBFGS with U and R fixed respectively. **Note:** LBFGS expects *vectors*, not matrices so everything has to be flattened.

```
# Optimise V, done wrt V
def minimize_U_fixed(U, V, LBFGS_iter):
 """
 Given current U, V - fix U and optimise V with
     LBFGS
 Output optimised V (unflattened)
 """

 # 1) Function to unflatten V and output loss
     gradient
 def grad_v_wrapper(V_flat):
  V = V_flat.reshape(r, d)
  return onp.array(grad_loss_v(U, V)).ravel()

 # 2) Function to unflatten V and output loss
 def loss_wrapper(V_flat):
  V = V_flat.reshape(r, d)
  return loss(U, V)

 # 3) L-BFGS-B
 traj = scipy.optimize.minimize(
  fun = loss_wrapper, # function to minimise
  x0 = onp.array(V.ravel()), # initial guess
  method = 'L-BFGS-B',
  jac = grad_v_wrapper, # fn that computes gradient
  options = {"maxiter": LBFGS_iter} # max number of
      iterations
 )

 V_final = traj["x"].reshape(r, d)
 return V_final

# Optimise U, done wrt U
def minimize_V_fixed(U, V, LBFGS_iter):
 """
 Given current U, V - fix V and optimise U with
     LBFGS
 Output optimised U (unflattened)
 """

 # 1) Function to unflatten U and output loss
```

```
     gradient
def grad_u_wrapper(U_flat):
 U = U_flat.reshape(d, r)
 return onp.array(grad_loss_u(U, V)).ravel()

# 2) Function to unflatten U and output loss
def loss_wrapper(U_flat):
 U = U_flat.reshape(d, r)
 return loss(U, V)

# 3) L-BFGS-B
traj = scipy.optimize.minimize(
 fun = loss_wrapper, # function to minimise
 x0 = onp.array(U.ravel()), # initial guess
 method = 'L-BFGS-B',
 jac = grad_u_wrapper, # fn that computes gradient
 options = {"maxiter": LBFGS_iter} # max number of
      iterations
)

U_final = traj["x"].reshape(d, r)
return U_final
```

Start optimisation by looping through iterations - each iteration consists of optimising U and V (separately). They are optimised by further iterations of LBFGS.

```
U, V = onp.copy(U_init), onp.copy(V_init)

LBFGS_iter = 50
n_iter = 30
loss_history = []

for _ in range(n_iter):
 V = minimize_U_fixed(U, V, LBFGS_iter)
 U = minimize_V_fixed(U, V, LBFGS_iter)
 loss_history.append(loss(U, V))

plt.plot(loss_history, "-s")
```

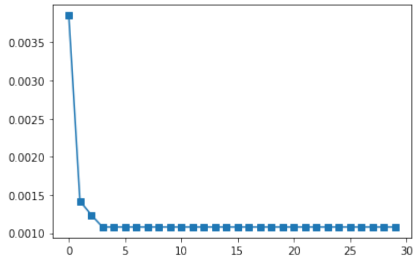


Figure 24: Loss plot (alternate minimisation with LBFGS)

Plot out true image with approximated image.

```
plt.subplot(1,2,1)
plt.imshow(M, cmap="gray")

plt.subplot(1,2,2)
plt.imshow(U @ V, cmap="gray")
```

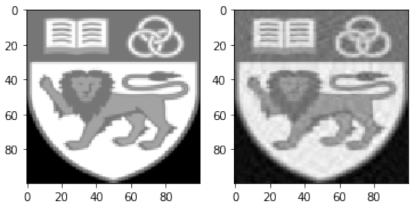


Figure 25: (Left) True image, (Right) Optimised image approx - alt min, LBFGS

Function Reconstruction

In this type of optimisation problems, the goal is to recover the true y-values of the function. We are given data where the y-values modified.

Possible modifications:

1. y-values added with random noise
2. y-values go through another function (e.g. poisson)

→ Sine observations

Discussed in an example, likely in the realm of Tutorial 8 - regularised function reconstruction (given **only data points, approx function**). This is different from the previous example of "*Function Approximation*" since the parameters of a function are not found here. Instead, given $\{x, z\}$ pairs, where $z = y + noise$, the goal is to **recover the y values**.

First, create samples drawn from the sine function with random noise added.

```
def func(x):
 """ one arbitrary function """
 return np.sin(x)

# Generate noise samples
n_samples = 100
sigma_noise = 0.3
x_list = onp.linspace(0, 10, n_samples)
z_true = funct(x_list)
y_list = z_true + onp.random.normal(0,
    sigma_noise, n_samples)

plt.plot(x_list, y_list, "x")
plt.grid(True)
```

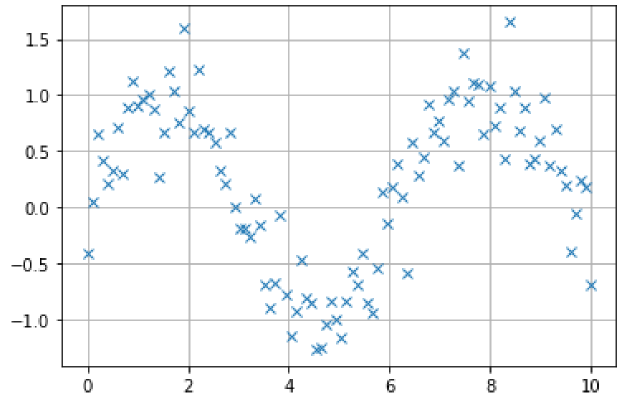


Figure 26: Scatterplot of data points (sine + noise)

```
reg = 5.
def loss(z):
 """
 loss function for regularised function
     reconstruction
 """
 difference = z[1:] - z[:(-1)]
 return np.mean( (z - y_list)**2 ) + reg *
     np.mean(difference**2)

grad_loss = jax.jit(jax.grad(loss))

# gradient descent
z = onp.copy(z_init)
learning_rate = 0.5
n_iter = 1000
loss_history = []

for k in range(n_iter):
 z = z - learning_rate*grad_loss(z)
 curr_loss = loss(z)
 loss_history.append(curr_loss)

plt.plot(loss_history)
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.grid(True)
```

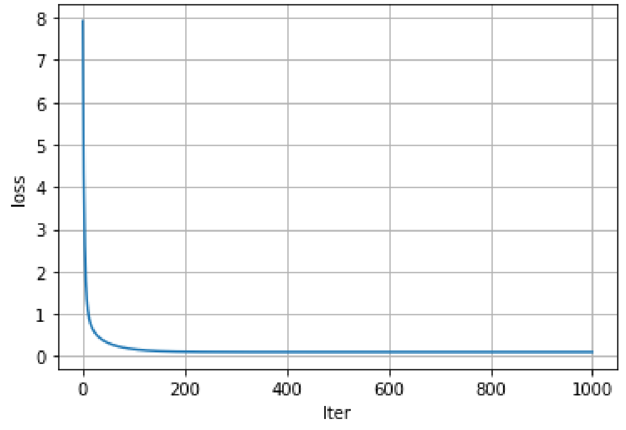


Figure 27: Loss history (sine reconstruction)

```
plt.plot(x_list, y_list, "x")
plt.plot(x_list, z_true, "r-")
plt.plot(x_list,z, "b--")
plt.title("Function Reconstruction")
plt.grid(True)
```

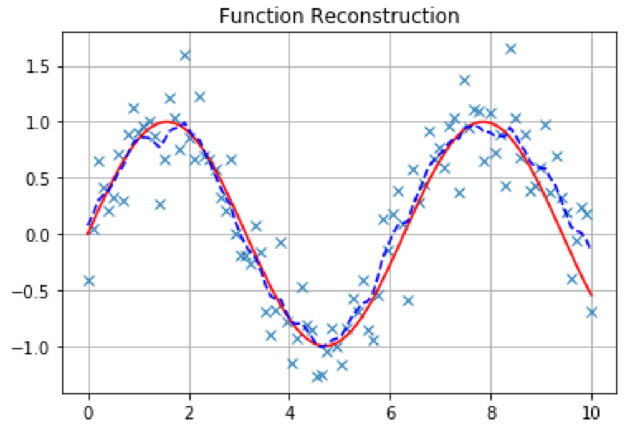


Figure 28: Recovered sine values

→ Possion observations

This is discussed likely for Tutorial 8. The true labels are now processed not by added noise but are inputs to a poisson distribution which outputs a random integer.

```
%matplotlib inline
from jax.config import config
config.update("jax_enable_x64", True) #important
    for L-BFGS

import jax
import jax.numpy as np

import pylab as plt
import imageio
import os
import numpy as onp
from skimage.transform import rescale, resize,
    downscale_local_mean
import pandas as pd
```

First, generate the samples.

```
# list of coordinates
x_list = onp.linspace(0, 10, 100)

# arbitrary function
def func(x):
 return 10 + 7*np.sin(3*x)
z_true = func(x_list)

# list of poisson observations
y_list = []
for z in z_true:
 y_list.append(onp.random.poisson(z))
y_list = onp.array(y_list)

plt.plot(x_list, z_true, "r--")
plt.plot(x_list, y_list, "bo", alpha=0.5)
plt.grid(True)
```

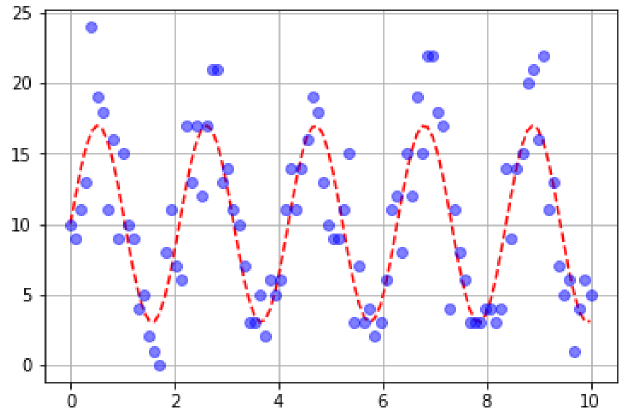


Figure 29: Scatterplot of given values (blue dots) and actual function (red dotted line)

Define functions, negative loglikelihood. Note that output needs to be reparameterised (like in projected gradient descent), since poisson function can only take in positive values.

```
def neg_loglik(z):
    """
    using it directly during optimisation may
        through errors because z may become
        negative
    """
    regularization = np.mean( (z[1:] - z[:99])**2 )
    log_lik = np.mean(-z + y_list * np.log(z))
    return -log_lik + 0.1 * regularization

def neg_loglik_reparam(u):
    """
    u = log(z) <---> z = exp(u)
    """
    z = np.exp(u) # ensure input is positive
    return neg_loglik(z)

grad_neg_loglik_reparam =
     jax.jit(jax.grad(neg_loglik_reparam))
```

Optimise with basic gradient descent.

```
u = onp.zeros(100) # 100 data points given
n_iter = 1000
learning_rate = 0.1
loss_history = []
for k in range(n_iter):
    u -= learning_rate *
        grad_neg_loglik_reparam(u) # if just
        log-likelihood, need to maximise - so do
        the update with a plus (i.e. u = u + ...)
    loss_history.append(neg_loglik_reparam(u))

plt.plot(loss_history)
plt.grid(True)
```

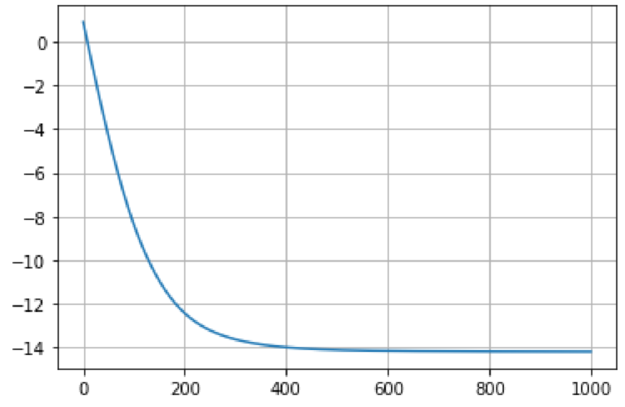


Figure 30: Loss history (poisson)

To find the optimal set of values:

```
# compute z_MLE out of the u_MLE
z_MLE = np.exp(u)

plt.plot(x_list, z_true, "r--", label="Truth")
plt.plot(x_list, z_MLE, "g-", linewidth = 3,
     label="Reconstruction")
plt.plot(x_list, y_list, "bo", alpha=0.3,
     label="data")
plt.legend()
plt.grid(True)
```

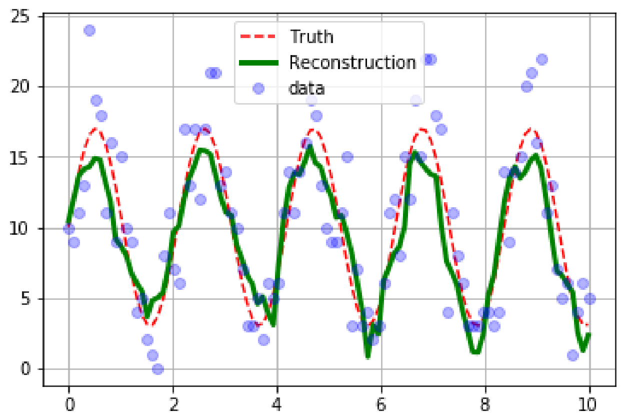


Figure 31: Recovered function values

Confidence Intervals

Discussed about the time of Tutorial 8, do so using Hessian at minimum (i.e. MLE).

Given some samples $(y_1 \dots y_N)$, fit a Gaussian $N(\mu, \sigma^2)$ to it. We do not know the mean μ and we do not know the standard deviation $\sigma > 0$.

Complete the following:

1. Write down the likelihood
2. Minimise the negative log-likelihood
3. Take care of the constraint $\sigma > 0$ by re-parameterisation
4. Get the confidence intervals using the Hessian at the minimum

Remark: The density of a Gaussian distribution with mean μ and standard deviation $\sigma > 0$ is

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right)$$

First, generate some data.

```
mu_true = 1.
sigma_true = 2.
N = 100
y = onp.random.normal(mu_true, sigma_true, size =
    N) # NOTE: takes in s.d., not variance
H = plt.hist(y, bins = 20)
```

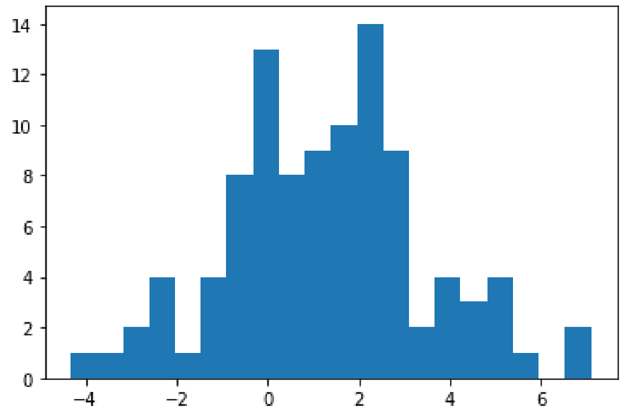


Figure 32: Histogram of true data (N = 100)

Question 1. Write down the likelihood.
Question 2. Minimise the negative log-likelihood.
Question 3. Take care of the constraint $\sigma > 0$ by re-parameterisation.

```
# IMPORTANT: know how to derive this
def mean_negloglik(mu, sigma):
 return np.mean(np.log(sigma) +
     (y-mu)**2/(2*sigma**2))
```

In order to deal with the constraint $\sigma > 0$. Introduce instead $u = \log \sigma$. In other words, we are looking for the parameter $x = (\mu, u) \in \mathbf{R}^2$.

```
# Define functions, gradient
def loss(x):
    """ loss == mean negative log likelihood """
    mu = x[0]
    u = x[1]
    sigma = np.exp(u)
    return mean_negloglik(mu, sigma)

grad_loss = jax.jit(jax.grad(loss))

# NECESSARY FOR LBFGS
```

```
def gradient_wrapper(x):
    """ computes the gradient of function at x,
        makes sure that output is a numpy array"""
    return onp.array(grad_loss(x))

loss_history = [] # save the loss trajectory

def save_traj(x):
    """ function that saves a few statistics for
        later analysis"""
    loss_history.append(loss(x))

# Initialise values
x_init = onp.array([0.,0.])
grad_loss(x_init) # test: [-1.29710711, -5.3042877 ]

# Optimise
traj = scipy.optimize.minimize(
 fun = loss, # function to minimise
 x0 = onp.array(x_init), # initial guess
 method = 'L-BFGS-B',
 callback = save_traj, # function used to save some
     results for later plotting
 tol = 10**-10, # NOTE: important to set tolerance,
     esp for images
 options={"maxiter":100} # maximum number of
     iterations
)

# Plot loss
plt.plot(loss_history, "-s")
plt.title("mean negative log likelihood")
plt.grid(True)

# Optimal values
mu_MLE, u_MLE = traj["x"]
sigma_MLE = onp.exp(u_MLE)
mu_MLE, sigma_MLE # (1.2971070934835425,
    2.1498373989917465)
```

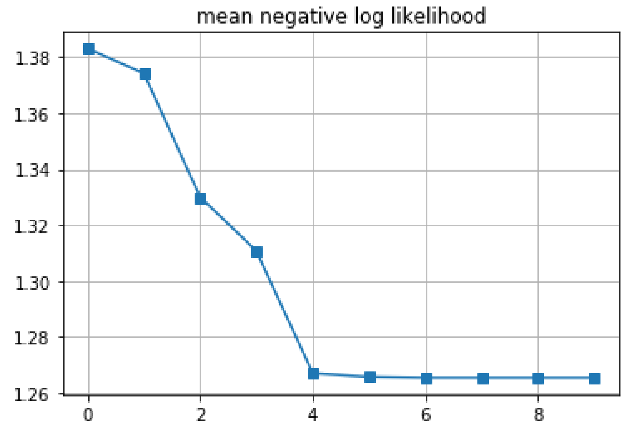


Figure 33: Loss plot

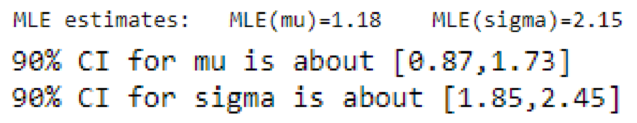
Question 4. Get the (90%) confidence intervals using the Hessian at the minimum.

The confidence interval for the estimated parameters can be estimated by computing the Hessian $\mathbf{H}$ at the *MLE*. The **inverse** Hessian is an approximation of the covariance structure of the parameter estimates.

Remark: Get $\underline{90\%}$ C.I. using $CI = estimate \pm \underline{\mathbf{2}} * s.d.$.

The approximate covariance structure is the inverse of the hessian of the negative log-likelihood (not the mean log-likelihood), hence the use of sum over the individual likelihoods for each observation. A good/intuituve way to understand is that the more data you have, the lower the confidence interval should it. Using the average log-likelihood will not shrink the confidence interval as more data is obtained.

```
def true_negloglik(z):
    mu = z[0]
    sigma = z[1]
    return np.sum(np.log(sigma) +
        (y-mu)**2/(2*sigma**2))

# Compute the Hessian
hess_neg_loglik = jax.hessian(true_negloglik)
# ---------------

# Compute the Hessian at the MLE
param_MLE = onp.array([mu_MLE, sigma_MLE])
hessian = hess_neg_loglik(param_MLE) # a 2x2 array

# Compute the inverse of Hessian to approximate
    the covariance
covariance_MLE = onp.linalg.inv(hessian) # a 2x2
    array

# ---------------

# Compute marginal standard deviation
# NOTE: standard deviation of parameters are roots
    of diagonal elements (i.e. variances)
std_mu = onp.sqrt(covariance_MLE[0, 0])
std_sigma = onp.sqrt(covariance_MLE[1, 1])

print("MLE estimates: \t MLE(mu)={0:2.2f} \t
    MLE(sigma)={1:2.2f}".format(std_mu, std_sigma))
print("90% CI for mu is about
    [{0:2.2f},{1:2.2f}]".format(mu_MLE - 2*std_mu,
    mu_MLE + 2*std_mu))
print("90% CI for sigma is about
    [{0:2.2f},{1:2.2f}]".format(sigma_MLE -
    2*std_sigma, sigma_MLE + 2*std_sigma))
```



```
MLE estimates:    MLE(mu)=1.18    MLE(sigma)=2.15
90% CI for mu is about [0.87,1.73]
90% CI for sigma is about [1.85,2.45]
```

Plot the joint distribution of the Gaussian approximation of the parameters at the *MLE*. i.e. Using the MLE estimates, sample data from Gaussian distribution. Indicate the true and MLE pair of (μ, σ) using coordinates.

```
samples_gaussian =
    onp.random.multivariate_normal(param_MLE,
    covariance_MLE, size=10000)
```

```
plt.scatter(samples_gaussian[:,0],
    samples_gaussian[:,1], alpha=0.2, marker="+")
plt.scatter(mu_MLE, sigma_MLE, c="red",
    marker="^", s=100, label="MLE")
plt.scatter(mu_true, sigma_true, c="purple",
    marker="s", s=100, label="True")
plt.legend()
plt.grid(True)
plt.xlabel(r"$\mu$")
plt.ylabel(r"$\sigma$")
```

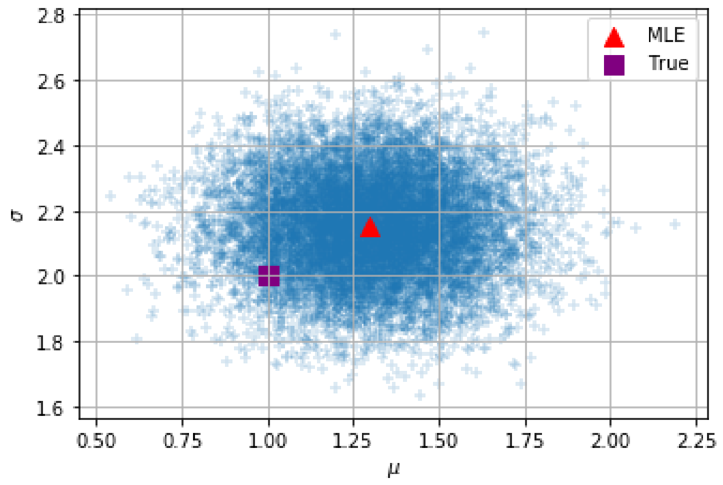


Figure 34: MLE estimate of μ and σ compared to true values