Statistical Machine Learning: Exercise 4

TECHNISCHE UNIVERSITÄT DARMSTADT

Neural Networks, Support Vector Machines and Gaussian Processes
Group 82: Alper Gece, Jinyao Chen

Summer Term 2021

Task 1: Neural Networks

1a) Multi-layer Perceptron

We use mean cross-entropy loss because it is good for multi-category problems.

We use ReLu as an activation function due to its excellent adaptability to multi-class classification. We have also tested sigmoid, but there are problems with vanishing gradients.

The input layer has 784 neurons because the count of pixels is 28×28; The output layer has 10 neurons because we need to classify 10 categories.

The selection strategy of the learning rate is constantly changing in the process of network training. In the beginning, the parameters are relatively random, so we should choose a relatively large learning rate, so that loss will fall faster. When we train for a period of time, the parameter update should have a smaller range of change, and then we choose a smaller learning rate.

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


def initialize(hidden_dim, output_dim):
   # retrieve mnist data
   X_train = np.loadtxt('./dataSets/mnist_small_train_in.txt', delimiter=',')
   X_test = np.loadtxt('./dataSets/mnist_small_test_in.txt', delimiter=',')
   y_train = np.loadtxt('./dataSets/mnist_small_train_out.txt', delimiter=',', dtype="int32")
   y_test = np.loadtxt('./dataSets/mnist_small_test_out.txt', delimiter=',', dtype="int32")

   # normalize data
   X_train = (X_train - np.mean(X_train)) / np.std(X_train)
   X_test = (X_test - np.mean(X_test)) / np.std(X_test)

   # one hot encode labels
   temp_y_train = np.eye(10)[y_train]
   temp_y_test = np.eye(10)[y_test]
   y_train = temp_y_train.reshape(list(y_train.shape) + [10])
   y_test = temp_y_test.reshape(list(y_test.shape) + [10])

   # weights for single hidden layer
   W1 = np.random.randn(hidden_dim, X_train.shape[1]) * 0.01
   b1 = np.zeros((hidden_dim,))
   W2 = np.random.randn(output_dim, hidden_dim) * 0.01
   b2 = np.zeros((output_dim,))

   parameters = [W1, b1, W2, b2]
```

```
    # return to column vectors
    return parameters, X_train.T, X_test.T, y_train.T, y_test.T


def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))


def dsigmoid(x):
    # input x is already sigmoid, no need to recompute
    return x * (1.0 - x)


def ReLU(x):
    return x * (x > 0)


def dReLU(x):
    return 1. * (x > 0)

# Cross entropy loss
def closs(pred, y):
    return np.squeeze(-np.sum(np.multiply(np.log(pred), y)) / len(y))

def dcloss(pred, y):
    return pred - y

# Mean squared error loss
def loss(pred, y):
    return np.sum(.5 * np.sum((pred - y) ** 2, axis=0), axis=0) / y.shape[1]

def dloss(pred, y):
    return (pred - y) / y.shape[1]


class NeuralNet(object):

    def __init__(self, hidden_dim, output_dim):
        # size of layers
        self.hidden_dim_1 = hidden_dim
        self.output_dim = output_dim

        # weights and data
        parameters, self.x_train, self.x_test, self.y_train, self.y_test =
            initialize(hidden_dim, output_dim)
        self.W1, self.b1, self.W2, self.b2 = parameters

        # activations
        batch_size = self.x_train.shape[0]

        self.ai = np.ones((self.x_train.shape[1], batch_size))
        self.ah1 = np.ones((self.hidden_dim_1, batch_size))
        self.ao = np.ones((self.output_dim, batch_size))

        # classification output for transformed OHE
        self.classification = np.ones(self.ao.shape)

        # container for loss progress
        self.loss = None
        self.test_error = None
```

```
def forward_pass(self, x):
   # input activations
   self.ai = x

   # hidden_1 activations
   self.ah1 = sigmoid(self.W1 @ self.ai + self.b1[:, np.newaxis])

   # output activations
   self.ao = sigmoid(self.W2 @ self.ah1 + self.b2[:, np.newaxis])

   # transform to OHE for classification
   self.classification = (self.ao == self.ao.max(axis=0, keepdims=0)).astype(float)

def backward_pass(self, target):
   # calculate error for output
   out_error = dloss(self.ao, target)
   out_delta = out_error * dsigmoid(self.ao)

   # calculate error for hidden_1
   hidden_1_error = self.W2.T @ out_delta
   hidden_1_delta = hidden_1_error * dReLU(self.ah1)

   # derivative for W2/b2 (hidden_1 --> out)
   w2_deriv = out_delta @ self.ah1.T
   b2_deriv = np.sum(out_delta, axis=1)

   # derivative for W1/b1 (input --> hidden_1)
   w1_deriv = hidden_1_delta @ self.ai.T
   b1_deriv = np.sum(hidden_1_delta, axis=1)

   return [w1_deriv, b1_deriv, w2_deriv, b2_deriv]

def train(self, epochs, lr, batch_size=128):

   self.loss = np.zeros((epochs,))
   self.test_error = np.zeros((epochs,))

   for epoch in range(epochs):

      indices = np.arange(self.x_train.shape[1])
      np.random.shuffle(indices)

      for i in range(0, self.x_train.shape[1] - batch_size + 1, batch_size):
         excerpt = indices[i:i + batch_size]

         batch_x, batch_y = self.x_train[:, excerpt], self.y_train[:, excerpt]

         # compute output of forward pass
         self.forward_pass(batch_x)

         # back prop error
         w1_deriv, b1_deriv, w2_deriv, b2_deriv = self.backward_pass(batch_y)

         # adjust weights with simple SGD
         self.W1 -= lr * w1_deriv
         self.b1 -= lr * b1_deriv
         self.W2 -= lr * w2_deriv
         self.b2 -= lr * b2_deriv
```

```
            # compute error
            self.forward_pass(self.x_test)
            error = loss(self.ao, self.y_test)
            self.loss[epoch] = error

            t = 0
            for i, pred in enumerate(self.classification.T):
                if np.argmax(self.y_test.T[i]) == np.argmax(pred):
                    t += 1

            acc = t / self.classification.shape[1]
            self.test_error[epoch] = 1 - acc

            # print progress
            if (epoch + 1) % 50 == 0:
                print("Epoch {}/{} -> loss: {:.5f} -> val_acc: {:.5f}".format(epoch + 1, epochs,
                    error, acc))

    def predict(self, x):
        self.forward_pass(x)
        return self.classification

    def visualize_prediction(self, expected, predicted):
        # plot loss progress
        plt.figure(2)
        plt.plot(self.loss)
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
        plt.show()

        # plot loss progress
        plt.figure(3)
        plt.plot(self.test_error)
        plt.xlabel("Epochs")
        plt.ylabel("Test Error")
        plt.show()


nn = NeuralNet(hidden_dim=784, output_dim=10)
nn.train(epochs=500, lr=.05)

y_hat = nn.predict(nn.x_test)
t = 0
for i, pred in enumerate(y_hat.T):
    if np.argmax(nn.y_test.T[i]) == np.argmax(pred):
        t += 1

print("Accuracy:", t / y_hat.shape[1])
nn.visualize_prediction(nn.y_test.T, y_hat.T)
```

The misclassification error is %8.266932270916338 on the testing set during learning.

Show how the misclassification error (in percent) on the testing set evolves during learning:

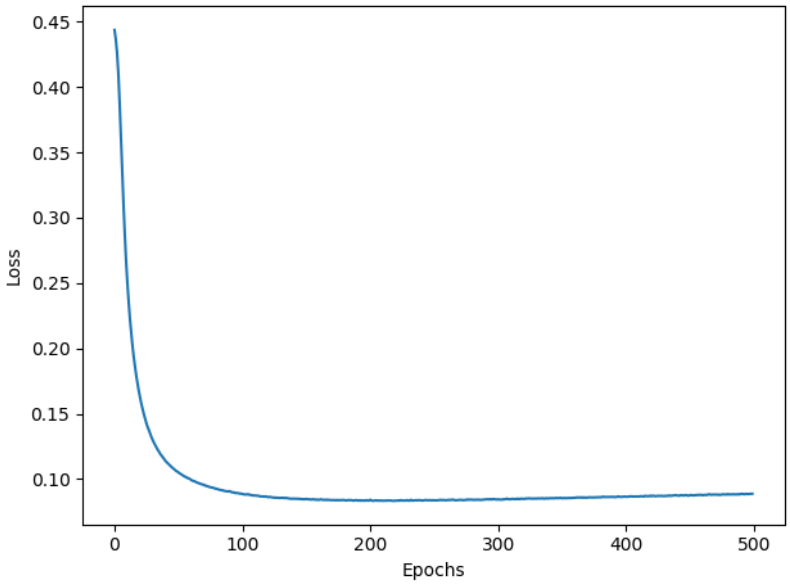

Figure 1: Loss

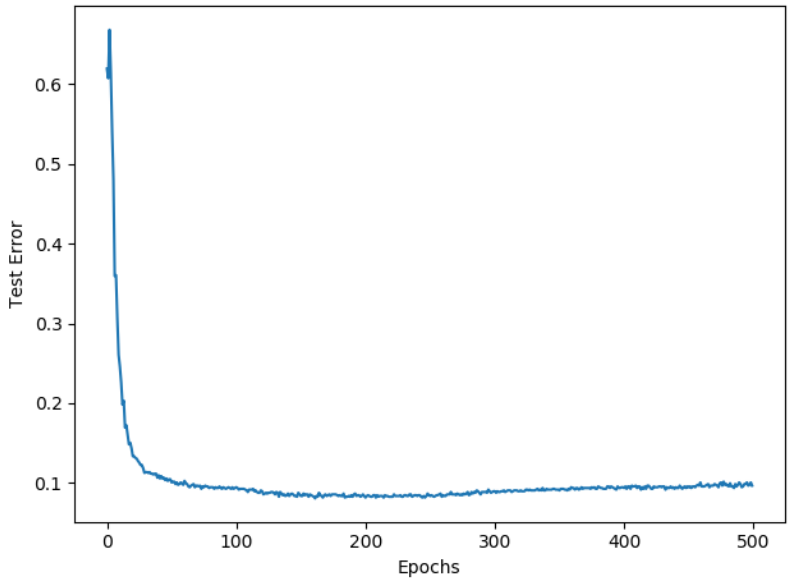

Figure 2: Test Error

Task 2: Support Vector Machines

2a) Definition

Support Vector Machines are algorithms that are used to distinguish two classes by calculating a hyperplane with a maximized margin. The margin to the closest data point is maximal on the training data points.

Unlike other machine learning methods that are discussed in the lecture, SVM can handle non-linear data efficiently using the Kernel trick. Thanks to the closed-form solution of the SVMs, a hyperplane could be found with the maximized margin so the resulting model is reduce over-fitting. Also, the closed-form solution provides us with no random italicization. The next advantage is that there is no need full dataset since the support vectors are enough for the final SVM model.

2b) Quadratic Programming

We have a linearly separable data x_i and N training points. The conditions and the constraints are given below.

$X_i \in \mathrm{R}^d$ and $\{x_i, \mathrm{t}_i\}_i^N = 1 t_i \in \{-1, 1\}$

The Hyperplane between the data points can be described as: $y(x) = w^T x + b$

We can define the closest data points as below.

$$t_i.y(x) = t_i.(w^T x_i + b) = 1 \tag{1}$$

And the other points can be described as below.

$$t_i.y(x_i) = t_i.(W^T x_i + b) \geq 1 \tag{2}$$

After writing these equations, the distance of the closest point to the hyperplane is stated as below.

$$\frac{y(x_i)}{|| w ||} = \frac{W^T x_i + b}{|| w ||} = \frac{1}{|| w ||} \tag{3}$$

Instead of maximizing the margin $\frac{1}{||w||}$ we can minimize the term $|| w^2 ||$.

$$\underset{w,b}{argmin} \frac{1}{2} || w ||^2 \tag{4}$$

$$s.t. t_i.(w^T x_i + b) - 1 \geq 0 \tag{5}$$

2c) Slack Variables

Slack variables are used for datasets whose classes can not be separated linearly by using a hyperplane. ξ is used to define slack variable and it is equal to zero for all data points. Data points on the boundary margin for the all other points ξ can be defined as $\xi = | t_n - y(x_n) |$. Points that stays between decision boundary and margin are $0 < \xi < 1$ and points for the data which is wrong are $\xi > 1$. The slack variation could be use in the optimization problem and C is a trade-off parameter the penalization of misclassification.

$$argmin \frac{1}{2} || W ||^2 + C. \sum_{N}^{i=1} \xi_i \tag{6}$$

$$s.t. t_i.(w^T x_i + b) - 1 + \xi_i \geq 0 \tag{7}$$

$$s.t. \xi_i \geq 0 \tag{8}$$

2d) Optimization with Slack Variables

The Lagrangian multiplier should be used in order to solve this problem.

The Langragian multipliers are a=$\{\alpha, \mu\}$:

$$L(w,b,a) = \frac{1}{2} \mid\mid w \mid\mid^2 + C.\sum_{n=1}^{N} \xi_n - \sum_{n=1}^{N} \alpha_n [t_n y(x_n) - 1 + \xi_n] - \sum_{n=1}^{N} \mu_n \xi_n \tag{9}$$

The following KKT conditions are defined.

$$\alpha_n \geq 0 \tag{10}$$

$$t_n y(x_n) - 1 + \xi_n \geq 0 \tag{11}$$

$$\alpha_n (t_n y(x_n) - 1 + \xi_n) = 0 \tag{12}$$

$$\mu_n \geq 0 \tag{13}$$

$$\xi_n \geq 0 \tag{14}$$

$$\mu_n \xi_n = 0 \tag{15}$$

Optimization of the w,b and ξ_n parameters;

$$\frac{\partial L}{\partial w} = 0 \Rightarrow w = \sum_{n=1}^{N} \alpha_n t_n x_n \tag{16}$$

$$\frac{\partial L}{\partial b} = 0 \Rightarrow \sum_{n=1}^{N} \alpha_n t_n = 0 \tag{17}$$

$$\frac{\partial L}{\partial \xi_n} = 0 \Rightarrow 0 = C - \alpha_n - \mu_n \Rightarrow \alpha_n = C - \mu_n \tag{18}$$

With the usage of the above equations, we can find the below equation.

$$\widetilde{L}(\alpha) = \frac{1}{2} \sum_{i=1}^{N} \alpha_i t_i x_i \sum_{j=1}^{N} \alpha_j t_j x_+ C \sum_{i=1}^{N} \xi_i - \sum_{i=1}^{N} \alpha_i t_i x_i \sum_{j=1}^{N} \alpha_j t_j x_j - \sum_{i=1}^{N} \alpha_i t_i b + \sum_{i=1}^{N} \alpha_i - \sum_{i=1}^{N} \alpha_i \xi_i - \sum_{i=1}^{N} (C - \alpha_i) \xi_i \tag{19}$$

$$= \sum_{i=1}^{N} \alpha_i + \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j t_i t_j x_i^T x_j - \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j t_i t_j x_i^T x_j - b \sum^{N_i=1} \alpha_i t_i + \sum_{i=1}^{N} \xi_i C - \sum_{i=1}^{N} \alpha_{ii} - \sum_{i=1}^{N} (C - \alpha_i) \xi_1$$

(20)

$$= \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j t_i t_j x_i^T x_j \tag{21}$$

Finally, the dual optimization problem shows up.

$$G(\alpha) = max\widetilde{L}(\alpha)$$
$$s.t. 0 \leq \alpha_i \leq C$$
$$s.t. \sum_{i=1}^{N} t_i \alpha_i = 0$$

2e) The Dual Problem

Data should be linearly separable in the SVM case. If the data is not linear separable, we can apply a nonlinear transformation $\xi(x)$. The nonlinear transformation on the data vectors can be easily applicable to the dual form of Lagrangian. Accordingly, the nonlinear SVM in dual form is given below.

$$\widetilde{L}(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j t_i t_j [\sigma(x_i)^T (x_j)] \tag{22}$$

Compared to the primal form, there are fewer unknown variables in the dual Lagrangian so that it is a bit easier to solve. Since we use few data points in the SVM, the computing will be more efficient by using only the data points where $\alpha_i \neq 0$.

2f) Kernel Trick

When the data is not linearly separable in the SVM case, the Kernel trick should be used. The kernel trick is a technique to perform non-linear feature mappings implicitly by replacing all scalar products with a kernel function $k(x, x)$.The kernel function has non-mapped original data as an input and computes mapping the data to a higher-dimensional space and calculating a scalar product. Therefore, complex spaces could be used without having a very costly feature mapping. Kernel function is given below.

K(x_i, x_j) $= \sigma(x_i)^T \sigma(X_j)$

By using a Kernel function, we make the scalar product directly with a non-linear function instead of mapping the data to higher-dimensional space.

Task 3: Gaussian Processes

3a) GP Regression

In this task, Gaussian Process is implemented to fit the target function y. Codes and GP plots are given below.

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

def target(x):
   s = np.sin(x)
   return s + np.square(s)


def kernel(x, z):
   return np.exp(-(x - z) ** 2)


def compute_c(x, noise):
   return kernel(x, x) + noise


def predict(x, X, Y, C_inv, noise):
   k = np.array([kernel(x, val) for val in X])
   mean = k.T.dot(C_inv).dot(Y)
   std = np.sqrt(compute_c(x, noise) - k.T.dot(C_inv).dot(k))
   return mean, std


def plot_gp(X, mean, std, iteration):
   y1 = mean - 2 * std
   y2 = mean + 2 * std

   plt.plot(X, target(X), "-", color="red", label="$sin(x) + sin^2(x)$")
   plt.plot(X, mean, color="black", label="$\mu$")
   plt.fill_between(X, y1, y2, facecolor='blue', interpolate=True, alpha=.5,
       label="$2\sigma$")
   plt.title("$\mu$ and $\sigma$ for iteration={}".format(iteration))
   plt.xlabel("x")
   plt.ylabel("y")


def gpr():
   noise = 0.001
   step_size = 0.005
   X = np.arange(0, 2 * np.pi + step_size, step_size)
   iterations = 15

   std = np.array([np.sqrt(compute_c(x, noise)) for x in X])
   j = np.argmax(std)

   Xn = np.array([X[j]])
   Yn = np.array([target(Xn)])

   C = np.array(compute_c(X[j], noise)).reshape((1, 1))
   C_inv = np.linalg.solve(C, np.identity(C.shape[0]))

   for iteration in range(0, iterations):
      mean = np.zeros(X.shape[0])
```

```
        std = np.zeros(X.shape[0])

        for i, x in enumerate(X):
            mean[i], std[i] = predict(x, Xn, Yn, C_inv, noise)

        if iteration + 1 in [1, 2, 5, 10, 15]:
            plot_gp(X, mean, std, iteration + 1)
            plt.plot(Xn, Yn, "o", c="blue", label="sampled")
            plt.legend()
            plt.show()

        j = np.argmax(std)
        Xn = np.append(Xn, X[j])
        Yn = np.append(Yn, target(Xn[-1]))

        # update C matrix
        x_new = Xn[-1]
        k = np.array([kernel(x_new, val) for val in Xn])
        c = kernel(x_new, x_new) + noise

        dim = C.shape[0]
        C_new = np.empty((dim + 1, dim + 1))
        C_new[:-1, :-1] = C
        C_new[-1, -1] = c
        C_new[:, -1] = k
        C_new[-1:] = k.T

        C = C_new
        C_inv = np.linalg.solve(C, np.identity(C.shape[0]))

gpr()
```

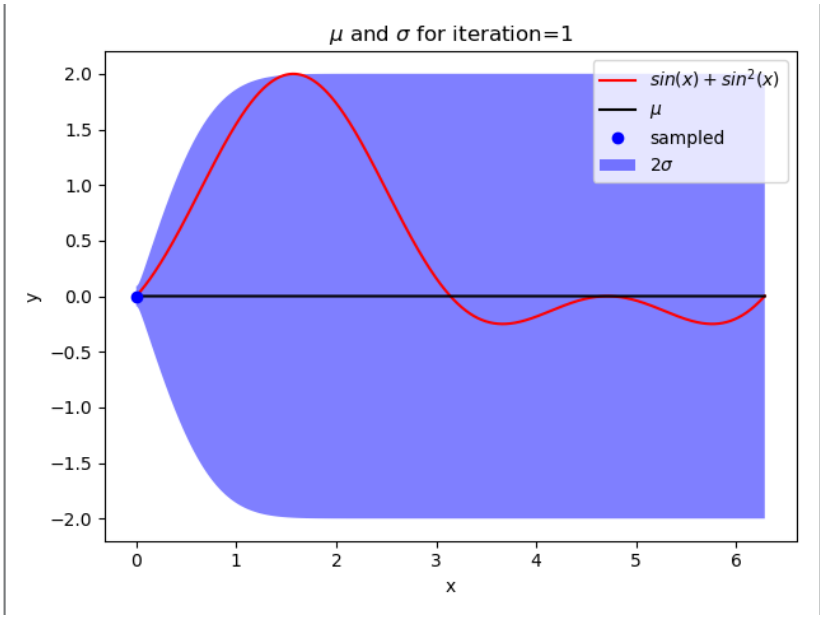

Figure 3: μ and σ for iteration = 1

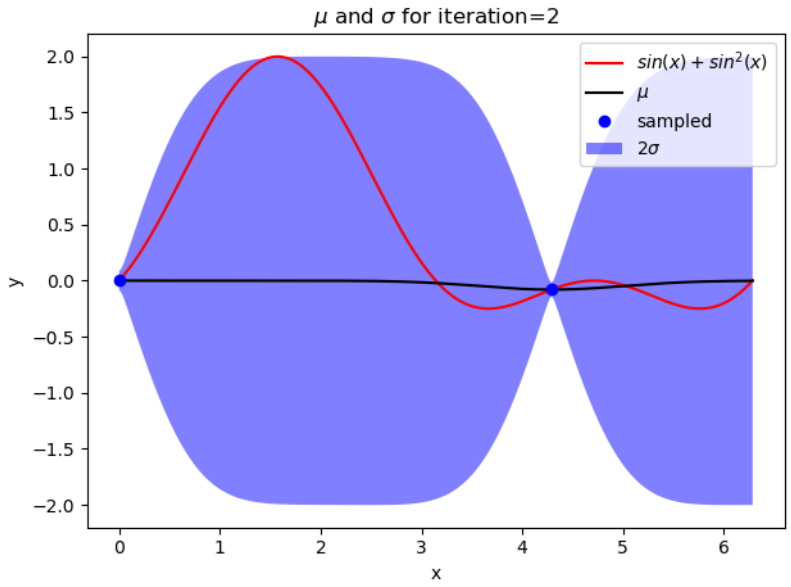

Figure 4: μ and σ for iteration = 2

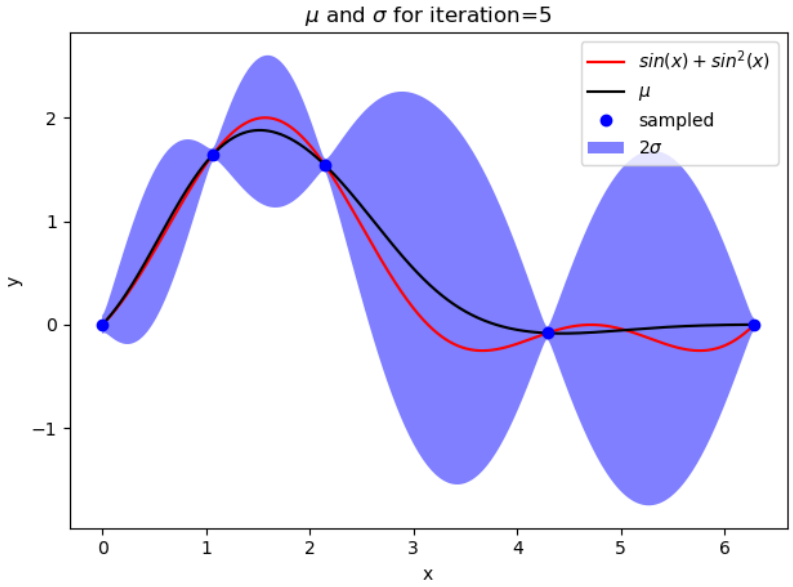

Figure 5: μ and σ for iteration = 5

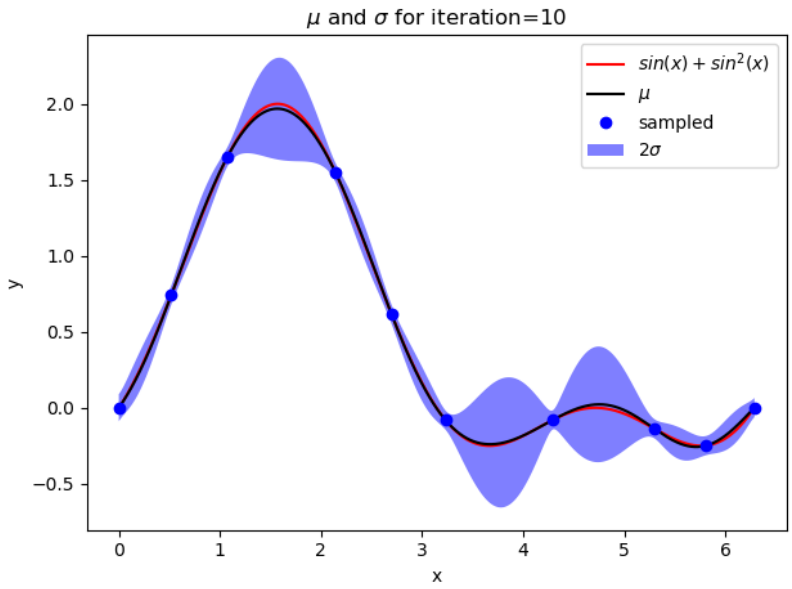

Figure 6: μ and σ for iteration = 10

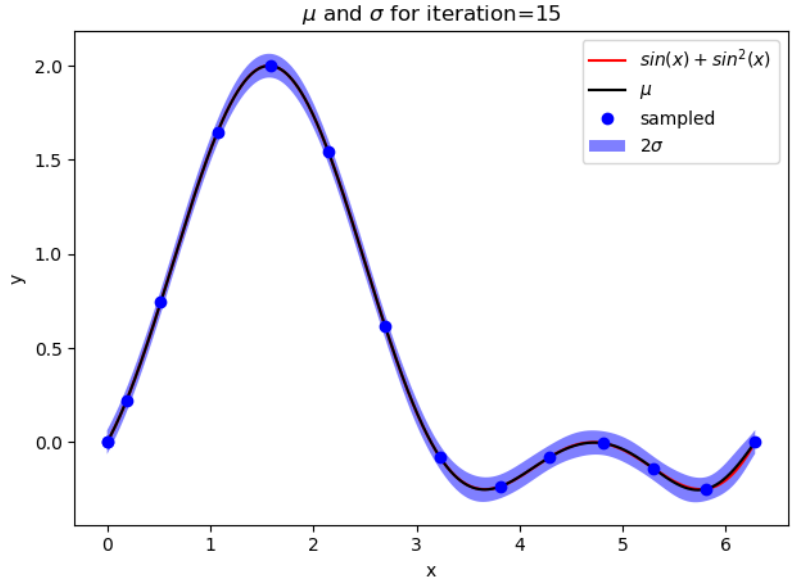

Figure 7: μ and σ for iteration = 15

References

[1] Multilayer Perceptron,
https://github.com/JGuymont/numpy-multilayer-perceptron

[2] Vector Support Machine,
https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-al

[3] Kernel Trick,
https://towardsdatascience.com/the-kernel-trick-c98cdbcaeb3f

[4] 2015 Aaron Hertzmann, David J. Fleet and Marcus Brubaker: Support Vector Machines,
http://www.cs.toronto.edu/ mbrubake/teaching/C11/Handouts/SupportVectorMachines.pdf