Homework 4: Machine learning

1 Part 1: A machine learning and SDE relation by a toy example

The purpose of the first part of the homework is to find a relation between the so called stochastic gradient descent method used in machine learning and the forward Euler method for a stochastic differential equation, by studying a special example. In particular the time step parameter, which is well understood for the Euler method, will provide understanding of the noise level in the stochastic gradient descent.

1.1 Machine learning background.

Here is first a short description of a machine learning problem to determine a neural network function from given data. For example, we are given data $\{(x_n, y_n)\}_{n=1}^N$, where $(x_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$ are independent samples drawn from an unknown probability density on $\mathbb{R}^d \times \mathbb{R}$. The objective is to train/learn a neural network function, e.g. $\alpha(x, \theta) := \sum_{k=1}^K \theta_k^1 \sigma(\theta_k^2 \cdot x + \theta_k^3)$, that solves the minimization problem

$$\min_{\theta \in \mathbb{R}^{(d+2)K}} \mathbb{E}[f\big(\alpha(x, \theta), y\big)]$$

with the activation function $\sigma(y) := 1/(1 + e^{-y})$, the loss function $f(\alpha, y) := |\alpha - y|^2$ and the neural network parameters $\theta = (\theta_k^1, \theta_k^2, \theta_k^3)_{k=1}^K$ where $\theta_k^1 \in \mathbb{R}, \theta_k^2 \in \mathbb{R}^d$ and $\theta_k^3 \in \mathbb{R}$. The stochastic gradient descent method for the iterations $\theta[n] \in \mathbb{R}^{(d+2)K}$, $n = 0, 1, 2, \ldots$ satisfying

$$\begin{aligned} \theta[0] &= \text{ some random guess in } \mathbb{R}^{(d+2)K}, \\ \theta[n+1] &= \theta[n] - \Delta t \nabla_\theta f\big(\alpha(x_n, \theta[n]), y_n\big), \ n = 0, 1, 2, \ldots \end{aligned} \tag{1}$$

is often used to approximately solve this minimization problem, based on a step size/learning rate $\Delta t > 0$. Chapter five in the book "Deep learning" by Ian Goodfellow, Yoshua Bengio and Aaron Courville, MIT Press (2016), http://www.deeplearningbook.org/ includes an introduction to machine learning.

1.2 A toy example.

The following toy example of a minimization problem to find a parameter $\theta \in \mathbb{R}$ uses similar iterations as (1). Assume we seek the minimal expected value

$$\min_{\theta \in \mathbb{R}} \mathbb{E}[f(\theta, Y)] \tag{2}$$

where Y is the stochastic variable with standard normal distribution (with mean zero and variance one) and $f(\theta, y) := |\theta - y|^2$ for $\theta \in \mathbb{R}$ and $y \in \mathbb{R}$.

1a. Consider the iterations

$$\begin{aligned} \theta_0 &= 1 \\ \theta_{n+1} &= \theta_n - \Delta t \frac{\partial f}{\partial \theta}(\theta_n, Y_n)\,, \ n = 0, 1, 2, \dots\,, \end{aligned} \tag{3}$$

and show that there is a constant C such that $\mathbb{E}[|\theta_n|^2] \leq C,\ n = 0, 1, 2, \dots$ for a suitable choice of $\Delta t \in \mathbb{R}$. Determine these Δt. Here, $Y_n,\ n = 0, 1, 2, \dots$ are independent stochastic variables which all are standard normal distributed and $\mathbb{E}[|\theta_n|^2]$ denotes the expected value of $|\theta_n|^2$. Determine also θ_* where

$$\mathbb{E}[f(\theta_*, Y)] = \min_{\theta \in \mathbb{R}} \mathbb{E}[f(\theta, Y)]$$

and establish the convergence rate of $\lim_{n\to\infty} \mathbb{E}[|\theta_n - \theta_*|^2]$.

1b. Write a Matlab program that performs the iterations in problem 1a and plots $\theta_n,\ n = 0, 1, 2, \dots N$.

1c. Show that the stochastic gradient descent iterations (3) are in fact forward Euler steps for an Ornstein-Uhlenbeck process. Formulate also the gradient descent method, based on computing the expected value exactly, to solve the minimization problem (2), determine its convergence rate and compare with the result in problem 1a.

2 Part 2: Approximating a solution to a minimization problem using TensorFlow

2.1 Problem and algorithm descriptions

Consider the minimization problem

$$\begin{aligned} &\min_{\boldsymbol{\theta}\in\mathbb{R}^{3\times K}} \mathbb{E}[|\alpha_{\boldsymbol{\theta}}(X) - f(X)|^2] \\ &f(x) = |x - \frac{1}{2}|^2, \quad x \in \mathbb{R}, \end{aligned} \tag{4}$$

where $X \sim \mathcal{U}(-4, 4)$,

$$\alpha_{\boldsymbol{\theta}}(x) = \sum_{k=1}^{K} \theta_k^1 \sigma(\theta_k^2 x + \theta_k^3), \quad x \in \mathbb{R}, \quad \boldsymbol{\theta} = (\theta_k^i) \in \mathbb{R}^{3\times K}$$

and

$$\sigma(x) = \frac{1}{1 + \exp(-x)}, \quad x \in \mathbb{R}.$$

The term $E_{\mathrm{c}}(\boldsymbol{\theta}) := \mathbb{E}[|\alpha_{\boldsymbol{\theta}}(X) - f(X)|^2]$ is commonly called the *loss function* or *cost function*. A solution can be approximated using the following Stochastic Gradient Descent optimizer algorithm (hereafter also called SGD):

- Generate an initial guess $\boldsymbol{\theta}_0 \in \mathbb{R}^{3\times K}$ from some distribution.
- Generate N data points $x_1, x_2, ..., x_N$ independently from the uniform distribution $\mathcal{U}(-4, 4)$ and compute the corresponding function values $f_n = f(x_n), n = 1, 2, 3, ..., N$.
- Take $T \in (0, \infty)$ and denote by $\Delta t = \frac{T}{M}$ the step length for some positive integer M.
- for $m = 1, 2, ..., M$ do
  - Take a random $i \in \{1, 2, ..., N\}$.
  - $\boldsymbol{\theta}_{m+1} = \boldsymbol{\theta}_m - 2\Delta t \nabla_\theta |\alpha_{\boldsymbol{\theta}_m}(x_i) - f_i|^2$.

An approximate solution to the minimization problem (4) is then given by $\boldsymbol{\theta}_M$.

The Stochastich Gradient Descent optimizer algorithm tries to minimize the approximation $E_{\mathrm{t}}(\boldsymbol{\theta}) := \sum_{n=1}^{N} \frac{|f(x_n) - \alpha_{\boldsymbol{\theta}}(x_n)|^2}{N}$, of the loss function, for the set $\{(x_n, f(x_n))\}_{n=1}^{N}$ of training points. It is hence called *training error*.

It is important to know how the neural network generalizes to data outside of the set of training data. Thus one can compute the *generalization error* (also called *test error*) $E_{\mathrm{g}}(\boldsymbol{\theta}) := \sum_{n=1}^{\tilde{N}} \frac{|f(\tilde{x}_n)-\alpha_{\boldsymbol{\theta}}(\tilde{x}_n)|^2}{\tilde{N}}$, where $\{(\tilde{x}_n, f(\tilde{x}_n))\}_{n=1}^{\tilde{N}}$ is roughly disjoint from the set of training points.

A small training error but a large generalization error is a symptom of *overfitting*.

2.2 A guided implementation in Python using TensorFlow

This section gives a step by step implementation of a program approximating a solution to the minimization problem (4). With TensorFlow, in Python, a neural network model describing $\alpha_{\boldsymbol{\theta}}(\cdot)$ is set up. A detailed instruction for installation of TensorFlow 2 could be found in the course Canvas page.

More general and comprehensive information could also be found in the web-page `https://www.tensorflow.org/install/`.

With the help of a high-level API named Keras in Tensorflow, the defining and training tasks for constructing a neural network model can be simplified. The neural network model can then be trained with a Stochastic Gradient Descent optimizer to learn the target function f in minimization problem (4). After the network has been trained, an approximation to $\alpha_{\boldsymbol{\theta}^*}(x),\ x \in [-4, 4]$, where $\boldsymbol{\theta}^*$ solves the minimization problem (4), is plotted.

At the end of this section the code in its whole is given.

Packages and variables

The implementation given uses the following packages:

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
```

Let

```
N = 100
```

denote the number of data points. The neural network model will consist of one hidden layer with

```
K = 10
```

nodes. The number of steps for the SGD is set to

```
M = 40000
```

and the step length to

```
dt = 0.005
```

. Thus

```
T = M*dt
```

is the end time of the SGD.

In the context of machine learning, the term '**minibatch**' refers to the subset of the dataset utilized for one iteration step, and 'batch size' defines the number of training examples in this subset. For instance, following our SGD algorithm introduced on Page 3, for each iteration step m, we take one random index i and use the corresponding data point (x_i, f_i) to compute the gradient $\nabla_\theta |\alpha_{\boldsymbol{\theta}_m}(x_i) - f_i|^2$, so as to instruct the searching direction of the next step $m+1$. This corresponds to the choice of batch size as 1. Instead, if we use 10 random data points $\{(x_j, y_j)\}_{j=1}^{10}$ for each SGD step and compute their average effect in the gradient $\frac{1}{10}\sum_{j=1}^{10} \nabla_\theta \big((\alpha_{\boldsymbol{\theta}_m}(x_j) - f_j)^2\big)$ to determine the next searching direction, then we are using a minibatch of size 10. For simplicity we shall use

```
Nbatch = 1
```

in our program, that is, our batch size for each SGD step is simply 1. Besides, in the training of neural network model, we can also split the whole training dateset into two parts. One majority part of these data points will be used for training, and the rest part will not be used for training, but only for validation of the model. The ratio between the validation set and the entire training dataset is called 'validation_split_ratio' in our program, and is specifically defined as

```
validation_split_ratio = 0.2
```

Also, in machine learning, the term **epoch** is a hyperparameter that refers to the number of times that the learning algorithm will work through the entire training dataset. For example, if we take batch size Nbatch = 10, and use EPOCH = 1 in a program with a dataset of size $N = 200$, then the whole algorithm will need to perform stochastic gradient descent steps for $M = (N* (1 - \text{validation\_split\_ratio})/\text{Nbatch}) \cdot \text{EPOCH} = (200 \times (1-0.2)/10) \times 1 = 16$ times to traverse the entire training dataset for once. Correspondingly, with the parameter setting in the code above, we shall have number of epochs

```
EPOCHS =
np.int64(M / (N * (1 - validation_split_ratio) / Nbatch))
```

where we use 'numpy.int64' function to convert the fraction from float to int type. Let

```
x_training = np.random.uniform(-4,4,size=(N,1))
```

be the uniformly distributed points in $[-4, 4]$ that will be fed through the neural network. For computing the generalization error, another set of uniformly distributed points

```
x_test = np.random.uniform(-4,4,size=(100,1))
```

in $[-4, 4]$ is needed. The target function f in minimization problem (4) is defined with

```
def f(x):
    return np.square(np.abs(x - 0.5))
```

in the code. Then the function values for these training set and testing set points can be computed simply with

```
y_training = f(x_training)
y_test = f(x_test)
```

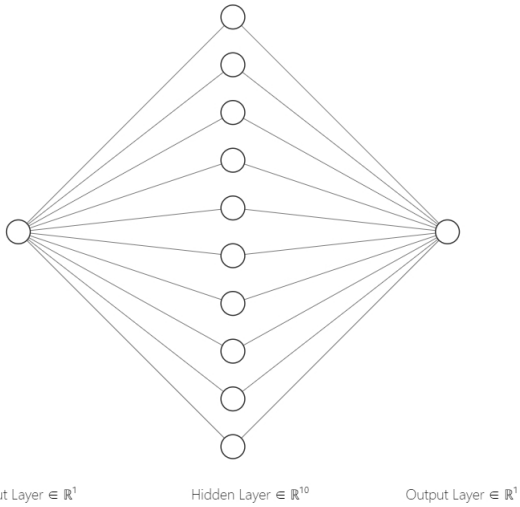
in the program.

Neural network model function

The function $\alpha_{\boldsymbol{\theta}}(\cdot) = \sum_{k=1}^{K} \theta_k^1 \sigma(\theta_k^2 \cdot + \theta_k^3)$ can be represented as a neural network model. In such a model each of the K elements of the sum can be represented as a node in a hidden layer. The "$\cdot$" can be represented as the one node in the input layer and $\alpha_{\boldsymbol{\theta}}$ as the one node in the output layer. Each $\theta_k^1, \theta_k^2, k = 1, 2, ..., K$ is called a weight and each $\theta_k^3, k = 1, 2, ..., K$ called a bias. A simple diagram illustrating the structure of this neural network model could be found in figure (1).
By applying the convenient Keras interface of Tensorflow, corresponding to the input data $x \in \mathbb{R}^1$, the first input layer can be simply defined with the code

```
Input_layer = tf.keras.Input(shape=(1,))
```

In order to construct the hidden-layer with $K = 10$ nodes, we should first specify the choice of the activation function $\sigma(\cdot)$. In practice, we can use a



Figur 1: A simple diagram for the structure of the one-hidden-layer neural network in the program.

function

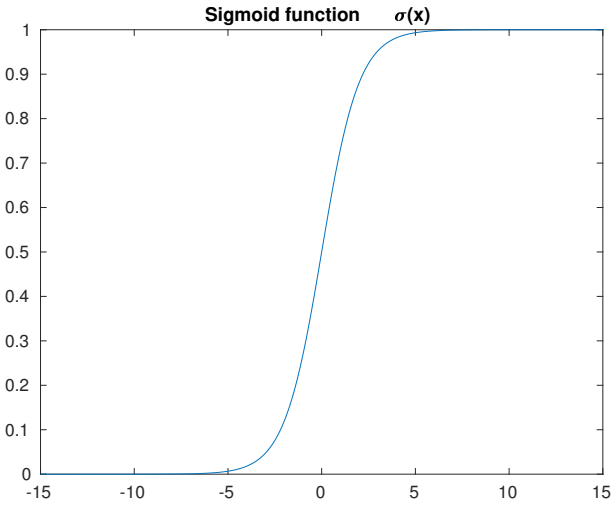
$$\sigma(x) = \frac{1}{1 + \exp(-x)},$$

which is known as the sigmoid function, and this is the name of it in TensorFlow as well. In neural network models it is a type of an activation function which loosely speaking is 1 if the neuron triggers and 0 otherwise. Since the sigmoid function is non-flat in a quite small interval, as seen in figure (2), its derivative is non-zero in a quite small interval. The Stochastic Gradient Descent optimization algorithm requires the computations of gradients of $\alpha_{\boldsymbol{\theta}}(x)$ which risk to be zero if the initial values $\boldsymbol{\theta}_0$ are not chosen carefully.

A practical choice for initial values of weight parameters θ_k^1 and θ_k^2, $k = 1, \cdots, K$ is to generate them from Normal distribution, and the initial values of bias parameters θ_k^3 can be simply set to zeros.

Then the definition of the hidden-layer can be conducted with

```
Hidden_layer =
keras.layers.Dense(units=K,
                   activation="sigmoid",
                   use_bias=True,
                   kernel_initializer='random_normal',
                   bias_initializer='zeros',
                   name="hidden_layer_1")
```



Figur 2: The activation function $\sigma(x)$ is non-flat in a quite small interval.

The output layer corresponding to the approximate function value $\alpha_{\boldsymbol{\theta}}(x) \in \mathbb{R}^1$, which is taken as the sum $\sum_{k=1}^{K} \theta_k^1 \sigma(\theta_k^2 \cdot x + \theta_k^3)$ on the $K = 10$ nodes of the hidden layer, can be defined with

```
Output_layer = keras.layers.Dense(units=1,
                                  use_bias=False,
                                  name="output_layer")
```

Here in the definition of the Hidden_layer and the Output_layer, we use the function keras.layers.Dense, where the term 'Dense' means to configure the connection between consecutive layers of the neural network in a fully-connected way, as illustrated in the above figure (1).

Finally, the neural network model is set up by assembling these three layers together with the Sequential function of Keras

```
model =
keras.Sequential([Input_layer, Hidden_layer, Output_layer])
```

Training of neural network function

This far the neural network model has been defined but nothing have been executed. Training the neural network model to learn the target function f means taking steps with the Stochastic Gradient Descent optimizer algorithm. Thus in the training process, actual computations will be done.

Corresponding to the stochastic gradient descent method, we can first start by identifying the SGD optimizer from Keras

```
optimizer = tf.keras.optimizers.SGD(learning_rate=dt)
```

where we also specify the learning rate with the step size $dt = 0.005$ as defined before. Then we can configure the model for training with the following code

```
model.compile(optimizer=optimizer, loss='mean_squared_error')
```

where we also specify the loss function defined with the form of mean squared error:

$$E_{\text{train}}(\boldsymbol{\theta}) := \frac{1}{N}\sum_{n=1}^{N} |f(x_n) - \alpha_{\boldsymbol{\theta}}(x_n)|^2,$$

for the set of training points $\{(x_n, f(x_n))\}_{n=1}^{N}$. After these preparations, we can train the neural network function on the training dataset with the following command

```
history = model.fit(x=x_training, y=y_training,
                    batch_size=Nbatch,
                    epochs=EPOCHS,
                    validation_split=validation_split_ratio,
                    verbose=1)
```

Here the 'fit'-function will conduct training process where for every SGD iteration step a minibatch of the specified batch size is utilized, and the entire dataset is traversed for the a fixed number of epochs, where the value of epochs is predetermined by our considerations with dataset size N, total number of iterations steps M and batch size Nbatch as EPOCH $= M/(N/\text{Nbatch})$.

The parameter 'validation_split' of the 'fit'-function determines the fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss function on this data at the end of each epoch.

Evaluate neural network model and plot the result

After training the neural network function with the data set, we can evaluate the quality of the neural network model by performing it on the test dataset $\{(x_n, y_n)\}_{n=1}^{\tilde{N}}$, and particularly we can compute the corresponding generalization error (also called test error)

$$E_{\text{g}}(\boldsymbol{\theta}) := \sum_{n=1}^{\tilde{N}} \frac{|f(\tilde{x}_n) - \alpha_{\boldsymbol{\theta}}(\tilde{x}_n)|^2}{\tilde{N}},$$

where $\{(\tilde{x}_n, f(\tilde{x}_n))\}_{n=1}^{\tilde{N}}$ is roughly disjoint from the set of training points. In the program we can use the code

```
model.evaluate(x=x_test, y=y_test, verbose=1)
```

where the 'evaluate'-function returns the loss function value for the model on the test dataset.

Suppose $\boldsymbol{\theta}^*$ solves the minimization problem (4). Then after the neural network model has been trained, an approximation to $\alpha_{\boldsymbol{\theta}^*}(x_m), m = 1, 2, 3, ..., 300$ where $x_1 < x_2 < x_3 < ... < x_{300}$ are equally spaced in the interval $[-4, 4]$ is to be plotted. First the 300 equally spaced points are generated and stored in the array

```
pts = np.linspace(-4, 4, 300).reshape(-1, 1)
```

The target function values corresponding to the `pts` values are computed with

```
target_fcn_vals = f(pts)
```

And the approximated values for $\alpha_{\boldsymbol{\theta}^*}(x_m)$ are computed with

```
alpha_vals = model(pts)
```

All data is now ready and only visualisation of the results remain. With

```
plt.figure('Learned function', figsize=(15, 10))
plt.plot(x_training, y_training, 'o', label='Training data')
plt.plot(pts, target_fcn_vals, label='Target function')
plt.plot(pts, alpha_vals, label='alpha')
plt.legend()
plt.show()
```

the training data points, the target function and $\alpha_{\boldsymbol{\theta}^*}$ are plotted. With

```
plt.figure('Error', figsize=(15, 10))
plt.semilogy(history.history['loss'], label='Training error')
plt.semilogy(history.history['val_loss'], label='Validation error')
plt.xlabel('Epoch')
plt.ylabel('Mean squared error')
plt.legend()
plt.grid(True)
plt.show()
```

the training error and validation error in each epoch is plotted.

2.3 Problems

1. Gradually first increase and then gradually decrease `dt`. Explain what you see. A suitable range to test is $0.05 >$ `dt`, cf. Part 1.

2. Plot the training error.

3. Set `K = 1`. What do you see?

4. For `K = 10`. What is a suitable choice of `N`? Motivate your answer.

5. Give an example of parameter settings that gives a small value of the training error but a large value of the generalization error.

6. This problem aims to show how using the sinus function instead of the sigmoid function as activation function can affect the outcome for a given bias initialization.

   Add the parameter

   ```
   bias_initializer
   =bias_init_unif(minval=20*np.pi,maxval=22*np.pi)
   ```

   as an argument to `keras.layers.Dense` when defining `Hidden_layer` and don't forget to define

   ```
   bias_init_unif = tf.initializers.random_uniform
   ```

   first. Does the SGD converge? Another activation function that can be used instead of the sigmoid function is the sinus function. Change the activation function in `Hidden_layer` from `'sigmoid'` to `tf.sin`. Does the SGD converge now? Explain the result.

7. Problem (4) is a problem in one dimension. Reformulate problem (4) to be in dimension d where d is a positive integer. Also extend the program so that it approximates a solution for a general positive integer d.

2.4 Code

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

N = 100
```

```
K = 10
dt = 0.005
M = 40000
T = M * dt
Nbatch = 1
validation_split_ratio = 0.2
EPOCHS = np.int64(M / (N * (1 - validation_split_ratio) / Nbatch))

np.random.seed(123)
tf.random.set_seed(124)
x_training = np.random.uniform(-4, 4, size=(N, 1))
x_test = np.random.uniform(-4, 4, size=(100, 1))

def f(x):
    return np.square(np.abs(x - 0.5))

y_training = f(x_training)
y_test = f(x_test)

Input_layer = tf.keras.Input(shape=(1,))
Hidden_layer =
keras.layers.Dense(units=K,
                   activation='sigmoid',
                   use_bias=True,
                   kernel_initializer='random_normal',
                   bias_initializer='zeros',
                   name="hidden_layer_1")
Output_layer = keras.layers.Dense(units=1,
                                  use_bias=False,
                                  name="output_layer")

model = keras.Sequential([Input_layer, Hidden_layer, Output_layer])

optimizer = tf.keras.optimizers.SGD(learning_rate=dt)
model.compile(optimizer=optimizer, loss='mean_squared_error')

history = model.fit(x=x_training, y=y_training,
                    batch_size=Nbatch,
                    epochs=EPOCHS,
                    validation_split=validation_split_ratio,
                    verbose=1)
```

```
model.evaluate(x=x_test, y=y_test, verbose=1)

pts = np.linspace(-4, 4, 300).reshape(-1, 1)
target_fcn_vals = f(pts)
alpha_vals = model(pts)

# figure 1
plt.figure('Learned function', figsize=(15, 10))
plt.plot(x_training, y_training, 'o', label='Training data')
plt.plot(pts, target_fcn_vals, label='Target function')
plt.plot(pts, alpha_vals, label='alpha')
plt.legend()
plt.show()

# figure 2
plt.figure('Error', figsize=(15, 10))
plt.semilogy(history.history['loss'], label='Training error')
plt.semilogy(history.history['val_loss'], label='Validation error')
plt.xlabel('Epoch')
plt.ylabel('Mean squared error')
plt.legend()
plt.grid(True)
plt.show()
```