ECE 046211 - Technion - Deep Learning

HW1 - Optimization and Automatic Differentiation

Keyboard Shortcuts

- Run current cell: **Ctrl + Enter**
- Run current cell and move to the next: **Shift + Enter**
- Show lines in a code cell: **Esc + L**
- View function documentation: **Shift + Tab** inside the parenthesis or `help(name_of_module)`
- New cell below: **Esc + B**
- Delete cell: **Esc + D, D** (two D's)

Students Information

- Fill in

Name	Campus Email	ID
Student 1	student_1@campus.technion.ac.il	123456789
Student 2	student_2@campus.technion.ac.il	987654321

Submission Guidelines

- Maximal garde: 100.
- Submission only in **pairs**.
  - Please make sure you have registered your group in Moodle (there is a group creation component on the Moodle where you need to create your group and assign members).
- **No handwritten submissions.** You can choose whether to answer in a Markdown cell in this notebook or attach a PDF with your answers.
- SAVE THE NOTEBOOKS WITH THE OUTPUT, CODE CELLS THAT WERE NOT RUN WILL NOT GET ANY POINTS!
- What you have to submit:
  - If you have answered the questions in the notebook, you should submit this file only, with the name: `ece046211_hw1_id1_id2.ipynb` .
  - If you answered the questionss in a different file you should submit a `.zip` file with the name `ece046211_hw1_id1_id2.zip` with content:
    - `ece046211_hw1_id1_id2.ipynb` - the code tasks
    - `ece046211_hw1_id1_id2.pdf` - answers to questions.
  - No other file-types (`.py` , `.docx` ...) will be accepted.
- Submission on the course website (Moodle).
- **Latex in Colab** - in some cases, Latex equations may no be rendered. To avoid this, make sure to not use *bullets* in your answers ("* some text here with Latex equations" -> "some text here with Latex equations").

Working Online and Locally

- You can choose your working environment:

1. `Jupyter Notebook` , **locally** with Anaconda or **online** on Google Colab
   - Colab also supports running code on GPU, so if you don't have one, Colab is the way to go. To enable GPU on Colab, in the menu: `Runtime` $\rightarrow$ `Change Runtime Type` $\rightarrow$ `GPU` .
2. Python IDE such as PyCharm or Visual Studio Code.
   - Both allow editing and running Jupyter Notebooks.

- Please refer to `Setting Up the Working Environment.pdf` on the Moodle or our GitHub (https://github.com/taldatech/ee046211-deep-learning) to help you get everything installed.
- If you need any technical assistance, please go to our Piazza forum (`hw1` folder) and describe your problem (preferably with images).

Agenda

- Part 1 - Theory
  - Q1 - Convergence of Gradient Descent
  - Q2 - Optimization and Gradient Descent
  - Q3 - Efficient Differentiation
  - Q4 - Autodiff
- Part 2 - Code Assignments
  - Task 1 - The Beale Function
  - Task 2 - Building an Optimizer - Adam
  - Task 3 - PyTorch Autograd
  - Task 4 - Low Rank Matrix Factorization
- Credits

Part 1 - Theory

- You can choose whether to answser these straight in the notebook (Markdown + Latex) or use another editor (Word, LyX, Latex, Overleaf...) and submit an additional PDF file, **but no handwritten submissions**.
- You can attach additional figures (drawings, graphs,...) in a separate PDF file, just make sure to refer to them in your answers.
- $\LaTeX$ Cheat-Sheet (to write equations)
  - Another Cheat-Sheet

Question 1 - Convergence of Gradient Descent

Recall from the lecture notes:

- **Definition**: A function f is β-smooth if:

$$\forall w_1, w_2 \in \mathbb{R}^d : ||\nabla f(w_1) - \nabla f(w_2)|| \leq \beta ||w_1 - w_2||$$

- **Lemma**: If f is β-smooth then

$$f(w_1) - f(w_2) - \nabla f(w_2)^T (w_1 - w_2) \leq \frac{\beta}{2} ||w_1 - w_2||^2$$

Prove the lemma.

Hints:

- Represent f as an integral: $f(x) - f(y) = \int_0^1 \nabla f(y + t(x-y))^T (x-y) dt$
- Make use of Cauchy-Schwarz.

Question 2 - Optimization and Gradient Descent

The function $f : \mathbb{R}^d \to \mathbb{R}$ is infinitely continuously differentiable, and satisfies $\min_{w\in\mathbb{R}^d} f(w) = f_* > -\infty$.

We wish to minimize this function using a version of Gradient Descent (GD) with step-size η, where in each iteration the gradients are multiplied by matrix A

$$(*)\; w(t+1) = w(t) - \eta A \nabla f\left(w(t)\right).$$

Matrix A is symmetric and strictly positive (positive definite with strictly positive eigenvalues), i.e., $\lambda_{min} \triangleq \lambda_{min}(A) > 0$, and denote $\lambda_{max} \triangleq \lambda_{max}(A)$.

1. In section only assume that $f(w) = \frac{1}{2} w^T H w$, where H is strictly positive (positive definite with strictly positive eigenvalues). Find/choose A and η such that the algorithm $(*)$ converges in minimal number of steps. Why is that choice is infeasible when d is large? What is a common applicable approximation?
2. Prove that Gradient Flow (i.e., GD in the limit $\eta \to 0$):

$$\dot{w}(t) = -A\nabla f\left(w(t)\right)$$

   converges to a critical point for all f and A that satisfy the conditions in the given question.
   - **Hint**: from the properties of eigenvalues it satifies that $\forall v \in \mathbb{R}^d : \lambda_{min}||v||^2 \le v^T A v \le \lambda_{max}||v||^2$.
3. Given that the function f is β-smooth, find a condition on the step-size η such that we get convergence to a critical point in algorithm $(*)$. Prove convergence under this condition.
   - **Hint**: for a β-smooth function, one can write:

$$f\left(w(t+1)\right) - f\left(w(t)\right) \le \left(w(t+1) - w(t)\right)^T \nabla f\left(w(t)\right) + \frac{\beta}{2}||w(t+1) - w(t)||^2$$

Question 3 - Efficient Differentiation

We wish to optimize a loss function $\mathcal{L}\left(\mathbf{w}\right)$ for $\mathbf{w} \in \mathbb{R}^d$ using Gradient Descent (GD) with some step size schedule η_t

$$(1)\;\; \forall t = 1, 2, ..: \mathbf{w}\left(t\right) = \mathbf{w}\left(t-1\right) - \eta_t \nabla \mathcal{L}\left(\mathbf{w}\left(t-1\right)\right) \tag{1}$$

initialized from some $\mathbf{w}\left(0\right)$. We would like to learn the best step size schedule using GD. **Hint**: throughout this question, you should use the *chain rule*.

1. Suppose we can consider each η_t as a separate parameter for each t. We initialize this parameter with η_0 and update η_{t-1} with a GD step on $\mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)$

$$(2)\;\; \eta_t = \eta_{t-1} - \alpha_t \frac{\partial \mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)}{\partial \eta_{t-1}} \tag{2}$$

   for every step of eq. (1), where α_t is the another step size. Calculate $\partial \mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)/\partial \eta_{t-1}$ as a function of the loss gradients $\nabla \mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)$ and $\nabla \mathcal{L}\left(\mathbf{w}\left(t-2\right)\right)$.
2. Now suppose we want to similarly update α_{t-1} using GD step on $\mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)$ every step of eq. (2) with update step κ_t

$$\alpha_t = \alpha_{t-1} - \kappa_t \frac{\partial \mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)}{\partial \alpha_{t-1}}\,. \tag{3}$$

   Calculate $\partial \mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)/\partial \alpha_{t-1}$ as a function of $\left\{\nabla \mathcal{L}\left(\mathbf{w}\left(t-k\right)\right)\right\}_{k=1}^{3}$.
3. Now we wish to update $\left(\eta_{t-1}, \eta_{t-2}\right)$ by doing a GD step on $\mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)$

$$(3)\;\; \left(\eta_{t+1}, \eta_t\right) = \left(\eta_{t-1}, \eta_{t-2}\right) - \alpha_t \left(\frac{\partial \mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)}{\partial \eta_{t-1}}, \frac{\partial \mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)}{\partial \eta_{t-2}}\right) \tag{4}$$

   every two steps of eq. (1). Calculate the derivative $\frac{\partial \mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)}{\partial \eta_{t-2}}$ as a function of η_{t-1}, $\left\{\nabla \mathcal{L}\left(\mathbf{w}\left(t-k\right)\right)\right\}_{k=1}^{3}$, and $\nabla^2 \mathcal{L}\left(\mathbf{w}\left(t-2\right)\right)$.
4. Now we wish again to update $\left(\eta_t, \eta_{t+1}, \dots, \eta_{t+T}\right)$ by doing a GD step on $\mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)$ every T steps of eq. (1)

$$(4)\;\; \left(\eta_{t+T}, \dots, \eta_t\right) = \left(\eta_{t-1}, \dots, \eta_{t-1-T}\right) - \alpha_t \left(\frac{\partial \mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)}{\partial \eta_{t-1}}, \dots, \frac{\partial \mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)}{\partial \eta_{t-1-T}}\right) \tag{5}$$

Calculate the derivative $\frac{\partial \mathcal{L}(\mathbf{w}(t-1))}{\partial \eta_{t-\tau}}$ as a function of $\left\{\eta_{t-k}, \nabla^2 \mathcal{L}\left(\mathbf{w}\left(t-k-1\right)\right)\right\}_{k=1}^{\tau-1}, \nabla \mathcal{L}\left(\mathbf{w}\left(t-1\right)\right)$ and $\nabla \mathcal{L}\left(\mathbf{w}\left(t-\tau-1\right)\right)$.

5. Compare this approach (eq. (4) with $T > 1$) to the first one (eq. (2)). Name one advantage for each approach. Hints: Think of computional complexity, ease of optimization, suitability of the objective.

Question 4 - Automatic Differentiation

Consider the scalar function:

$$f = \exp(\exp(x) + \exp(x)^2) + \sin(\exp(x) + \exp(x)^2)$$

1. Write down the derivative w.r.t. x explicitly, i.e., $\frac{df}{dx}$
2. We define the following intermediate variables:

$$a = \exp(x)$$
$$b = a^2$$
$$c = a + b$$
$$d = \exp(c)$$
$$e = \sin(c)$$
$$f = d + e$$

   Draw a graph picturing the relationship between all variables (called the **computation graph**).
3. Using the graph, write down the derivatives of the individual terms, working backwards to compute the derivative of f (i.e., write down the derivatives $\frac{df}{dd}, \frac{df}{de}, \ldots, \frac{df}{dx}$)

Part 2 - Code Assignments

- You must write your code in this notebook and save it with the output of aall of the code cells.
- Additional text can be added in Markdown cells.
- You can use any other IDE you like (PyCharm, VSCode...) to write/debug your code, but for the submission you must copy it to this notebook, run the code and save the notebook with the output.

In [ ]:
```
# imports for the practice (you can add more if you need)
import os
import numpy as np
import pandas as pd
import torch
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import LogNorm
from sklearn.datasets import load_iris
seed = 211
np.random.seed(seed)
torch.manual_seed(seed)
# %matplotlib notebook
%matplotlib inline
```

Task 1 - The Beale Function

The Beale function is defined as follows:

$$f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$$

1. What is the global minima of this function?
2. Implement the Beale function: `beale_f(x,y)`.
3. Implement a function, `beale_grads(x,y)` that returns the gradients of the Beale function.

4. 3D plot the Beale function wit the global minima you found. Use Matplotlib's `ax.plot_surface(x_mesh, y_mesh, z, norm=LogNorm(), rstride=1, cstride=1, edgecolor='none', alpha=.8, cmap=plt.cm.jet)` for the function, and `ax.plot(x, y, f(x, y), 'r*', markersize=20)` for the minima.
5. 2D plot the contours with `ax.contour(x_mesh, y_mesh, z, levels=np.logspace(-.5, 5, 35), norm=LogNorm(), cmap=plt.cm.jet)` and the minima with `ax.plot(x, y, 'r*', markersize=20)`.

Your Answers Here

In [ ]:

```python
# Set the manually calculated minima
min_x = None
min_y = None

def beale_f(x, y):
    value = None
    """
    Your Code Here
    """
    return value

def beale_grads(x, y):
    dx, dy = None, None
    """
    Your Code Here
    """

    grads = np.array([dx, dy])
    return grads
```

In [ ]:

```python
minima = np.array([min_x, min_y])
beale_res = beale_f(*minima)
grads_res = beale_grads(*minima)
print(f"minima (1x2 row vector shape): {minima}")
print(f"beale_f output: {beale_res}")
print(f"beale_grad output: {grads_res}")
```

Task 2 - Building an Optimizer - Adam

In this task, you are going to implement the Adam optimizer. We are giving the skeleton of the code and the description of the methods, and you need to implement the optimizer.

Recall the Adam update rule:

$$m_{k+1} = \beta_1 m_k + (1-\beta_1)\nabla f(w^k) = \beta_1 m_k + (1-\beta_1) g_k$$

$$v_{k+1} = \beta_2 v_k + (1-\beta_2)(\nabla f(w^k))^2 = \beta_2 v_k + (1-\beta_2) g_k^2$$

Then, they use an **unbiased** estimation:

$$\hat{m}_{k+1} = \frac{m_{k+1}}{1-\beta_1^{k+1}}$$

$$\hat{v}_{k+1} = \frac{v_{k+1}}{1-\beta_2^{k+1}}$$

(the β's are taken with the power of the current iteration)

$$w_{k+1} = w_k - \frac{\alpha}{\sqrt{\hat{v}_{k+1}} + \epsilon}\hat{m}_{k+1}$$

- ϵ deafult's is 10^{-8}

1. Implement `class AdamOptimizer()`.
   - `function` is the Python function you want to optimize.
   - `gradients` is the Python function that returns the gradients of `function`.
   - `x_init` and `y_init` are the initialization points for the optimizer.
   - Save the `path` of the optimizer (the minima points the optimizer visits during the optimization).
   - Stopping criterion: change in minima `<1e-7`.
   - **You can change the class however you wish, you can remove/add variables and methods as you wish**

2. For `x_init=0.7, y_init=1.4, learning_rate=0.1, beta1=0.9, beta2=0.999`, optimize the Beale function. Plot the results **with the path taken** (better do it on the 2D contour plot).
3. Choose different initialization and learning rate and show the results as in 2.

In [ ]:
```python
class AdamOptimizer():
    def __init__(self, function, gradients, x_init=None, y_init=None,
                 learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.f = function
        self.g = gradients
        scale = 3.0
        self.current_val = np.zeros([2])
        if x_init is not None:
            self.current_val[0] = x_init
        else:
            self.current_val[0] = np.random.uniform(low=-scale, high=scale)
        if y_init is not None:
            self.current_val[1] = y_init
        else:
            self.current_val[1] = np.random.uniform(low=-scale, high=scale)
        print("x_init: {:.3f}".format(self.current_val[0]))
        print("y_init: {:.3f}".format(self.current_val[1]))

        self.lr = learning_rate
        self.grads_first_moment = np.zeros([2])
        self.grads_second_moment = np.zeros([2])
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon

        # for accumulation of loss and path (w, b)
        self.z_history = []
        self.x_history = []
        self.y_history = []


    def func(self, variables):
        """Beale function.

        Args:
          variables: input data, shape: 1-rank Tensor (vector) np.array
            x: x-dimension of inputs
            y: y-dimension of inputs

        Returns:
          z: Beale function value at (x, y)
        """

    def gradients(self, variables):
        """Gradient of Beale function.

        Args:
          variables: input data, shape: 1-rank Tensor (vector) np.array
            x: x-dimension of inputs
            y: y-dimension of inputs

        Returns:
          grads: [dx, dy], shape: 1-rank Tensor (vector) np.array
            dx: gradient of Beale function with respect to x-dimension of inputs
            dy: gradient of Beale function with respect to y-dimension of inputs
        """


    def weights_update(self, grads, time):
        """Weights update using Adam.

          g1 = beta1 * g1 + (1 - beta1) * grads
          g2 = beta2 * g2 + (1 - beta2) * grads ** 2
          g1_unbiased = g1 / (1 - beta1**time)
          g2_unbiased = g2 / (1 - beta2**time)
          w = w - lr * g1_unbiased / (sqrt(g2_unbiased) + epsilon)
        """

    def history_update(self, z, x, y):
        """Accumulate all interesting variables
        """


    def train(self, max_steps):
```

In [ ]:
```python
"""
Your Code Here
"""
```

In [ ]:
```python
opt = AdamOptimizer(beale_f, beale_grads, x_init=0.7, y_init=1.4, learning_rate=0.1, beta1=0.9, beta2=0.999, eps
```

In [ ]:
```python
%time
opt.train(1000)
print("Global minima")
print("x*: {:.2f}  y*: {:.2f}".format(minima[0], minima[1]))
print("Solution using the gradient descent")
print("x: {:.4f}  y: {:.4f}".format(opt.x, opt.y))
```

In [ ]:
```python
# plot the Beale function values during the optimization
```

In [ ]:
```python
# plot the optimization path
path = opt.path
```

Task 3 - PyTorch Autograd

For the function from the theory practice:

$$f = \exp(\exp(x) + \exp(x)^2) + \sin(\exp(x) + \exp(x)^2)$$

1. Implement it and its dervative (explicitly) using `torch`.
2. Define a scalar tensor `x` and use `autograd` to calculate the derivative w.r.t x. Does the result correspond to the output of the function the calculates the derivative explicitly?

In [ ]:
```python
def f(x):
    f_val = None
    """
    Your Code Here
    """
    return f_val

def derv_f(x):
    derv_val = None
    """
    Your Code Here
    """
    return derv_val
```

In [ ]:
```python
x = torch.tensor(0.5, requires_grad=True)
print(x)
f_res = f(x)
f_manual_grad = derv_f(x.detach())

"""
Your Code Here
"""
# Calculate with torch autograd
f_autograd = None


print(f_manual_grad)
print(f_autograd)
```

Task 4 - Low Rank Matrix Factorization

Consider the following optimization problem:

$$\min_{\hat{U},\hat{V}} ||A - \hat{U}\hat{V}||_F^2$$

Where $A \in \mathcal{R}^{m \times n}, \hat{U} \in \mathcal{R}^{m \times r}, \hat{V} \in \mathcal{R}^{r \times n}$ and $r < min(m, n)$ (r is the rank of the matrix). $|| \cdot ||_F^2$ denotes the Frobenius norm.

1. Implement a function, `gd_factorize_ad(A, rank, num_epochs=1000, lr=0.01)`, that given a 2D tensor `A` and a `rank`, will calculate the low-rank factorization of `A` using **gradient decsent**. Compute and apply all the gradients of $\hat{U}$ and of $\hat{V}$ once per epoch. $\hat{U}$ and $\hat{V}$ should be initially created with uniform random values. Use PyTorch's `autograd` for the gradients.
    * To compute the squared Frobenius norm loss (reconstruction loss), use `torch.nn.functional.mse loss with reduction='sum'`.
2. Use the provided `data` of the Iris dataset of 150 instances and 4 features. Apply `gd_factorize_ad` to compute the 2-rank matrix factorization of `data`. What is the reconstruction loss?

In [ ]:
```
df = load_iris(as_frame=True).data # option 1
# df = pd.read_csv('./iris.data', header=None) # option 2
data = torch.tensor(df.iloc[:, [0, 1, 2, 3]].values)
data = data - data.mean(dim=0)
```

In [ ]:
```
def gd_factorize_ad(A, rank, num_epochs=1000, lr=0.01):
    # initialize
    U = None
    V = None

    """
    Your Code Here
    """

    # implement gradient descent
    for epoch in range(num_epochs):

        """
        Your Code Here
        """
        loss = None
        if epoch % 5 == 0:
            print(f'epoch: {epoch}, loss: {loss}')
    return U, V
```

In [ ]:
```
U, V = gd_factorize_ad(data.float(), rank=2, num_epochs=1000, lr=0.01)
```

Credits

---

* Icons made by Becris from www.flaticon.com
* Icons from Icons8.com - https://icons8.com
* Datasets from Kaggle - https://www.kaggle.com/