Lab 3

Hüseyin Anil Gündüz

Imports

In [1]:

```
from typing import Any

import torch
from torch.autograd import Function
from torch import Tensor

import matplotlib.pyplot as plt
from  matplotlib_inline.backend_inline import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')
```

Welcome to the third lab. The first exercise is an implementation of gradient descent on a bivariate function. The second exercise is about computing derivatives of the weights of a neural network, and the third exercise combines the previous two.

Exercise 1

This exercise is about gradient descent. We will use the function $f(x_1, x_2) = (x_1 - 6)^2 + x_2^2 - x_1 x_2$ as a running example:

1. Use pen and paper to do three iterations of gradient descent:
   - Find the gradient of f;
   - Start from the point $x_1 = x_2 = 6$ and use a step size of $1/2$ for the first step, $1/3$ for the second step and $1/4$ for the third step;
   - What will happen if you keep going?
2. Write a function that performs gradient descent:
   - For simplicity, we use a constant learning rate.
   - Can you find a way to prematurely stop the optimization when you are close to the optimum?

In [2]:

```
# Note: Defining a custom autograd function is not a necessity for this small task,
# but it is a good place to showcase some capabilities of PyTorch.

class MyFunction(Function):
    @staticmethod
    def forward(ctx: Any, x: Tensor) -> Tensor:
        # The "ctx" object serves to stash information for the backward pass
        ctx.save_for_backward(x)
        func_value = (
# TODO compute the value of f at x.
        )
        return func_value

    @staticmethod
    def backward(ctx: Any, grad_output: Tensor):
        # The "grad_output" parameter is the backpropagated gradient from subsequen
        # operations w.r.t. to the output of this function.
        x = ctx.saved_tensors[0]

        grad_x = torch.tensor([
# TODO compute the gradient of f at x.
        ])
        return grad_output * grad_x
```

In [3]:

```

func = MyFunction()
# The "required_grad" argument needs to be True.
# Otherwise no gradients will be computed.
x = torch.tensor([6., 6.], requires_grad=True)

# Custom functions are applied over the "apply" method.
y = func.apply(x)
print('Function output: {}'.format(y))

# Gradients for every operation in this chain are computed
# by calling the "backward" method on the output tensor.
y.backward()

# The x tensor now has a grad attribute with the gradients.
print('Gradients: {}'.format(x.grad))

# Note: No usage of auto differentiation was done in this example.
```

Does it match what you computed?

In the next step we define a small gradient descent optimizer.

In [4]:

```
class GradientDescentOptimizer:
    def __init__(self,
                 func: Function,
                 max_steps: int,
                 alpha: float):
        """
        Init an Optimizer for performing GD.

        :param func: Function to apply.
        :param max_steps: Maximum number of GD steps.
        :param alpha: Learning Rate.
        """
        self.func = func
        self.max_steps = max_steps
        self.alpha = alpha

    def __call__(self, x: Tensor) -> Tensor:
        """
        Apply GD on a tensor.

        :param x: Input tensor.
        """
        # Usually you would apply the gradients inplace on the input tensor,
        # but for the sake of the example we keep the input tensor consistent and
        # work on a copy.
        x_cp = x.detach().clone()
        x_cp.requires_grad = True

# TODO use a for loop to do gradient descent.
# HINT When applying gradients you will need an "torch.no_grad()" context
# manager. To modify the content of the tensor you will need its ".data"
# attribute. Don't forget to erase the gradients after each iteration or
# or they will accumulate.
        return x_cp
```

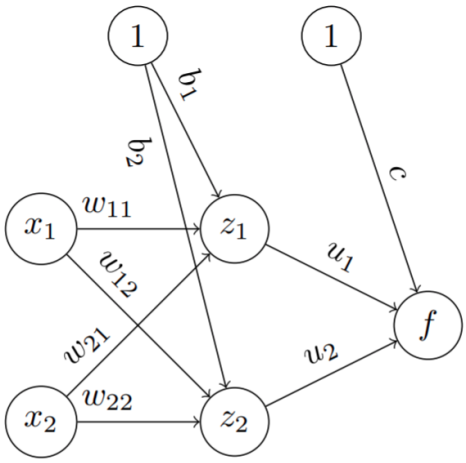
In [5]:

```
x = torch.tensor([6., 6.], requires_grad=True)
gd_optimizer = GradientDescentOptimizer(func=MyFunction(), max_steps=10, alpha=0.1)
x_new = gd_optimizer(x)
print(x_new)
```

Play a bit with the starting point and learning rate to get a feel for its behavior. How close can you get to the minimum?

Exercise 2

This exercise is about computing gradients with the chain rule, with pen and paper. We will work with a neural network with a single hidden layer with two neurons and an output layer with one neuron.



The neurons in the hidden layer use the $\tanh$ activation, while the output neuron uses the sigmoid. The loss used in binary classification is the *binary cross-entropy*:

$$\mathcal{L}(y, f_{out}) = -y \log f_{out} - (1 - y) \log(1 - f_{out})$$

where $y \in \{0, 1\}$ is the true label and $f_{out} \in (0, 1)$ is the predicted probability that $y = 1$.

1. Compute $\partial \mathcal{L}(y, f_{out})/\partial f_{out}$
2. Compute $\partial f_{out}/\partial f_{in}$
3. Show that $\partial \sigma(x)/\partial x = \sigma(x)(1 - \sigma(x))$
4. Show that $\partial \tanh(x)/\partial x = 1 - \tanh(x)^2$ (Hint: $\tanh(x) = (e^x - e^{-x})(e^x + e^{-x})^{-1}$)
5. Compute $\partial f_{in}/\partial c$
6. Compute $\partial f_{in}/\partial u_1$
7. Compute $\partial \mathcal{L}(y, f_{out})/\partial c$
8. Compute $\partial \mathcal{L}(y, f_{out})/\partial u_1$
9. Compute $\partial f_{in}/\partial z_{2,out}$
10. Compute $\partial z_{2,out}/\partial z_{2,in}$
11. Compute $\partial z_{2,in}/\partial b_2$
12. Compute $\partial z_{2,in}/\partial w_{12}$
13. Compute $\partial z_{2,in}/\partial x_1$
14. Compute $\partial \mathcal{L}(y, f_{out})/\partial b_2$
15. Compute $\partial \mathcal{L}(y, f_{out})/\partial w_{12}$
16. Compute $\partial \mathcal{L}(y, f_{out})/\partial x_1$

You will notice that there are lots of redundancies. We will see how to improve these computations in the lecture and in the next lab. Luckily, modern deep learning software computes gradients automatically for you.

Exercise 3

Now that we know how to do gradient descent and how to compute the derivatives of the weights of a simple network, we can try to do these steps together and train our first neural network! We will use the small dataset with five points we studied in the first lab.

First, let's define the dataset:

In [6]:

```
x = torch.tensor([
    [0, 0],
    [1, 0],
    [0, -1],
    [-1, 0],
    [0, 1]
], dtype=torch.float)
y = torch.tensor([1, 0, 0, 0, 0])
```

Next, a function to compute the output of the network:

In [7]:

```
def sigmoid(x: Tensor) -> Tensor:
# TODO compute the sigmoid on x and return.

def predict(x: Tensor, b1: float, b2: float,
            w11: float, w12: float, w21: float, w22: float,
            c: float, u1: float, u2:float) -> Tensor:
# TODO compute and return the output of the network.

# This should return the predictions for the five points in the datasets
# We can unpack the param vector for the positional params of the function so that
# need to enter every single entry.
params = torch.randn(9)
predictions = predict(x, *params)
```

Since gradient descent is done on the loss function, we need a function to compute it:

In [8]:

```
def get_loss(target: Tensor, pred: Tensor) -> Tensor:
# TODO return the average loss.

loss = get_loss(y, predictions)
print(loss)
```

Now, we need to compute the gradient of each parameter:

In [9]:

```
def get_gradients(x: Tensor, target: Tensor,
            b1: float, b2: float,
            w11: float, w12: float, w21: float, w22: float,
            c: float, u1: float, u2:float) -> Tensor:
    # First, we perform the forward pass.
    z1in = b1 + x[:, 0] * w11 + x[:, 1] * w21
    z1out = torch.tanh(z1in)

    z2in = b2 + x[:, 0] * w12 + x[:, 1] * w22
    z2out = torch.tanh(z2in)

    fin = c + u1 * z1out + u2 * z2out
    fout = sigmoid(fin)

# TODO compute all the partial derivatives.

    # Return the derivatives in the same order as the parameters vector
    return torch.stack([
        dL_db1, dL_db2, dL_dw11, dL_dw12, dL_dw21, dL_dw22, dL_dc, dL_du1, dL_du2
    ])

print(get_gradients(x, y, *params))
```

Finite differences are a useful way to check that the gradients are computed correctly:

In [10]:

```
# First, compute the analytical gradient of the parameters.
gradient = get_gradients(x, y, *params)
eps = 1e-9
for i in range(9):
    # Compute loss when subtracting eps to parameter i.
    neg_params = params.clone()
    neg_params[i] = neg_params[i] - eps
    neg_value = get_loss(y, predict(x, *neg_params))

    # Compute loss when adding eps to parameter i.
    pos_params = params.clone()
    pos_params[i] = pos_params[i] + eps
    pos_value = get_loss(y, predict(x, *pos_params))

    # Compute the "empirical" gradient of parameter i
    fdiff_gradient = torch.mean((pos_value - neg_value) / (2 * eps))

    # Error if difference is too large
    if torch.abs(gradient[i] - fdiff_gradient) < 1e-5:
        raise ValueError('Gradients are probably wrong!')

print("Gradients are correct!")
```

We can finally train our network. Since the network is so small compared to the dataset, the training procedure is very sensitive to the way the weights are initialized and the step size used in gradient descent.

Try to play around with the learning rate and the random initialization of the weights and find reliable values that make training successful in most cases.

In [11]:

```
min_loss = 10
alpha = 1.
steps = 100
best_params = None

for i in range(10):
    params = torch.randn(9)

    # Do GD
    for _ in range(steps):
        gradients = get_gradients(x, y, *params)
        params -= alpha * gradients

    final_loss = get_loss(y, predict(x, *params))
    print('RUN {} \t LOSS {:.4f}'.format(i + 1, float(final_loss)))

    if final_loss < min_loss:
        best_params = params
        min_loss = final_loss
```

We can use the function in the previous lab to visualize the decision boundary of the best network:

In [12]:

```
def plot_decision_boundary(
        x: Tensor, y: Tensor, grid_x: Tensor, grid_y, pred: Tensor) -> None:
    """Plot the estimated decision boundary for a 2D grid with predictions."""
    plt.contourf(grid_x, grid_y, pred.view(grid_x.shape))
    plt.scatter(x[:, 0], x[:, 1], c=y, cmap='jet')
    plt.show()
```

In [13]:

```
grid_range = torch.linspace(-2, 2, 50)
grid_x, grid_y = torch.meshgrid(grid_range, grid_range)
grid_data = torch.stack([grid_x.flatten(), grid_y.flatten()]).T
pred = predict(grid_data, *best_params)

plot_decision_boundary(x, y, grid_x, grid_y, pred)
```

Also try to visualize the decision boundary of network with random parameters:

In [14]:

```
pred = predict(grid_data, *torch.randn(9))
plot_decision_boundary(x, y, grid_x, grid_y, pred)
```