Lab 4

Hüseyin Anil Gündüz

Welcome to the fourth lab. In this lab, we will derive the backpropagation equations, code the training procedure, and test it on our beloved dataset with five points.

Imports

In [1]:

```
from typing import Optional, List, Tuple

import matplotlib.pyplot as plt
import torch
from matplotlib_inline.backend_inline import set_matplotlib_formats
from torch import Tensor

set_matplotlib_formats('png', 'pdf')
```

Exercise 1

Consider a neural network with L layers and a loss function $\mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out})$. Call the output of the i-th unit of the ℓ-th layer $\mathbf{z}^{(\ell)}_{i,out} = \sigma^{(\ell)}(\mathbf{z}^{(\ell)}_{i,in})$ with $\mathbf{z}^{(\ell)}_{i,in} = \sum_j \mathbf{W}^{(\ell)}_{ji}\mathbf{z}^{(\ell-1)}_{j,out} + \mathbf{b}^{(\ell)}_i$ its pre-activation output. Finally, consider $\delta^{(\ell)}_i = \partial\mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out})/\partial\mathbf{z}^{(\ell)}_{i,in}$ the gradient of the loss with respect to the pre-activation outputs of layer ℓ.

Derive the back-propagation algorithm for a network with arbitrary architecture. You might find the results of the previous lab a useful reference, as well as chapter 5 of the book *Mathematics for Machine Learning* (https://mml-book.github.io (https://mml-book.github.io)).

1. Show that

$$\delta^{(L)}_i = \frac{\partial\mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out})}{\partial\mathbf{z}^{(L)}_{i,out}} \cdot \sigma'^{(L)}(\mathbf{z}^{(L)}_{i,in})$$

$$\frac{\partial\mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out})}{\partial\mathbf{W}^{(\ell)}_{ji}} = \delta^{(\ell)}_i \cdot \mathbf{z}^{(\ell-1)}_{j,out}$$

$$\frac{\partial\mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out})}{\partial\mathbf{b}^{(\ell)}_i} = \delta^{(\ell)}_i$$

$$\delta^{(\ell-1)}_i = \left(\sum_k \delta^{(\ell)}_k \cdot \mathbf{W}^{(\ell)}_{ik}\right) \cdot \sigma'^{(\ell-1)}(\mathbf{z}^{(\ell-1)}_{i,in})$$

2. Use vectorized operations (i.e., operations with vectors and matrices) to compute the gradients with respect to a single sample.
3. Extend the vectorized operations to handle data in batches, and show that:

$$
\begin{aligned}
\Delta^{(L)} &= \nabla_{\mathbf{Z}_{out}^{(L)}} \mathcal{L}(\mathbf{Y}, \mathbf{Z}_{out}^{(L)}) \odot \sigma'^{(L)}(\mathbf{Z}_{in}^{(L)}) \\
\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}(\mathbf{Y}, \mathbf{Z}_{out}^{(\ell)}) &= {\mathbf{Z}_{out}^{(\ell-1)}}^T \cdot \Delta^{(\ell)} \\
\nabla_{\mathbf{b}^{(\ell)}} \mathcal{L}(\mathbf{Y}, \mathbf{Z}_{out}^{(L)}) &= \sum_i {\Delta_i^{(\ell)}}^T \\
\Delta^{(\ell-1)} &= \Delta^{(\ell)} {\mathbf{W}^{(\ell)}}^T \odot \sigma'^{(\ell-1)}(\mathbf{Z}_{in}^{(\ell-1)})
\end{aligned}
$$

where $\Delta^{(\ell)}$, $\mathbf{Y}$ and $\mathbf{Z}_{out}^{(\ell)}$ are matrices whose i-th row contain the respective vectors δ, $\mathbf{y}$ and $\mathbf{z}_{\cdot,out}^{(\ell)}$ for the i-th sample in the batch, and $\odot$ is the element-wise product. \end{enumerate}

Exercise 2

In this exercise, we will code the backpropagation algorithm and apply it to our five-points dataset.

First, let's define some structures to quickly create a neural network with layers of given size. It will use tanh activation in the hidden layers and sigmoid for the output layer. Although we will use it for classification, we use the mean squared error loss for a change.

NOTE: We use PyTorch only as computation engine. To showcase how backpropagation works under the hood, we do not utilize auto diff or other structures like modules or autograd functions in this example (just basic OOP). However, we still use some conventions like forward/backward notation.

In [2]:

```
class Linear:
    def __init__(self, in_features: int, out_features: int):
        self.weight = self._init_glorot(in_features, out_features)
        self.bias = torch.zeros(out_features)

        self.weight_grad: Optional[Tensor] = None
        self.bias_grad: Optional[Tensor] = None

    @staticmethod
    def _init_glorot(in_features: int, out_features: int) -> Tensor:
        """Init a weight matrix with glorot initialization."""
        b = torch.sqrt(torch.tensor([6. / (in_features + out_features)]))
        return (2 * b) * torch.rand(in_features, out_features) - b

    def forward(self, x: Tensor) -> Tensor:
        return x @ self.weight + self.bias

class Sigmoid:
    def __init__(self):
        self.func = lambda x: 1 / (1 + torch.exp(-x))

    def forward(self, x: Tensor) -> Tensor:
        return self.func(x)

    def get_gradient(self, x: Tensor) -> Tensor:
        return self.func(x) * (1 - self.func(x))


class TanH:
    @staticmethod
    def forward(x: Tensor) -> Tensor:
        return torch.tanh(x)

    @staticmethod
    def get_gradient(x: Tensor) -> Tensor:
        return  1 - torch.tanh(x)**2

class MSELoss:
    @staticmethod
    def forward(y_true: Tensor, y_pred: Tensor) -> Tensor:
        return torch.mean((y_true - y_pred)**2)

    @staticmethod
    def get_gradient(y_true: Tensor, y_pred: Tensor) -> Tensor:
        return 2 * (y_pred - y_true) / len(y_true)


# Now we bring everything together and create our neural network.
class NeuralLayer:
    def __init__(self, in_features: int, out_features: int, activation: str):
        self.linear = Linear(in_features, out_features)

        if activation == 'sigmoid':
            self.act = Sigmoid()
        elif activation == 'tanh':
            self.act = TanH()
        else:
            raise ValueError('{} activation is unknown'.format(activation))
```

```
        # We save the last computation as we'll need it for the backward pass.
        self.last_input: Optional[None] = None
        self.last_zin: Optional[None] = None
        self.last_zout: Optional[None] = None

    def forward(self, x: Tensor) -> Tensor:
        self.last_input = x
        self.last_zin = self.linear.forward(x)
        self.last_zout = self.act.forward(self.last_zin)
        return self.last_zout

    def get_weight(self) -> Tensor:
        """Get the weight matrix in the linear layer."""
        return self.linear.weight

    def get_bias(self) -> Tensor:
        """Get the weight matrix in the linear layer."""
        return self.linear.bias

    def set_weight_gradient(self, grad: Tensor) -> None:
        """Set a tensor as gradient for the weight in the linear layer."""
        self.linear.weight_grad = grad

    def set_bias_gradient(self, grad: Tensor) -> None:
        """Set a tensor as gradient for the bias in the linear layer."""
        self.linear.bias_grad = grad


class NeuralNetwork:
    def __init__(self, input_size, output_size, hidden_sizes: List[int]):
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_sizes = hidden_sizes

        self.layers: List[NeuralLayer] = []
        layer_sizes = [self.input_size] + self.hidden_sizes
        for i in range(1, len(layer_sizes)):
            self.layers.append(NeuralLayer(layer_sizes[i-1], layer_sizes[i], 'tanh'
        self.layers.append(NeuralLayer(hidden_sizes[-1], self.output_size, 'sigmoid

        self.loss = MSELoss()

    def forward(self, x: Tensor) -> Tensor:
# TODO perform the forward pass and return the predictions.

    def get_loss(self, x: Tensor, y: Tensor) -> Tensor:
        """Compute the loss for a dataset and given labels."""
# TODO: Use the loss function and the forward method to compute the loss on the dat

    def backward(self, x: Tensor, y: Tensor) -> None:
        """Compute all gradients over backpropagation."""
        # Perform forward pass.
        # The z's are automatically saved by our NeuralLayer object.
        y_pred = self.forward(x)

# TODO: Compute the gradients.
# Hint: Rely on the objects and structures we defined above. (Especially NeuralLaye
# Also remember that the `z_in` and `z_out`s are saved in the the `linear` object o

        # Check if gradients have the right size.
        for i, layer in enumerate(self.layers):
```

```
            if layer.linear.weight_grad.shape != layer.linear.weight.shape \
                or layer.linear.bias_grad.shape != layer.linear.bias.shape:
                raise ValueError('Gradients in layer with index {} have a wrong sha
                                 .format(i))


    def apply_gradients(self, learning_rate: float) -> None:
        """Update weights with the computed gradients."""
# TODO: Apply the gradients that are stashed in NeuralLayer/Linear to perform gradi
```

After we have defined our network, we can create it and test if the passes work without errors on our small dataset.

In [3]:

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

network = NeuralNetwork(
    input_size=2,
    hidden_sizes=[5, 3],
    output_size=1
)

print(network.forward(x))
network.backward(x, y)
```

We can inspect the decision boundary as in the previous exercises for the randomly initialized network:

In [4]:

```
def plot_decision_boundary(x: Tensor, y: Tensor, net: NeuralNetwork) -> None:
    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

    predictions = net.forward(grid_data)

    plt.contourf(grid_x, grid_y, predictions.view(grid_x.shape))
    plt.scatter(x[:, 0], x[:, 1], c=y, cmap='jet')
    plt.show()

plot_decision_boundary(x, y, network)
```

We can now finally train our network!

In [5]:

```
def train(
        x: Tensor,
        y: Tensor,
        net: NeuralNetwork,
        epochs: int,
        lr: float
) -> Tuple[NeuralNetwork, Tensor]:
    """
    Train a neural network.
    :param x: Training dataset.
    :param y: Training labels.
    :param net: Neural network to train.
    :param epochs: Number of training epochs.
    :param lr: Learning rate for gradient descent.
    :return: Trained network and losses over course of training.
    """

# TODO iterate over the dataset for the given number of epochs and modify the weigh
# Use all the data to compute the gradients.
# Also calculate the loss, so that we can track how the training is going.
    return net, torch.stack(losses)

network, losses = train(x, y, network, 2500, 0.25)
```

By plotting the loss after each parameter update, we can be sure that the network converged:

In [6]:

```
plt.plot(losses)
plt.show()
```

And the decision boundary of the network is:

In [7]:

```
plot_decision_boundary(x, y, network)
```

Try to train a few randomly initialized networks and vary depth and hidden sizes to discover different decision boundaries. Try to modify the learning rate and see how it affects the convergence speed. Finally, try different ways to initialize the weights and note how the trainability of the network is affected.