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}

Solution

Question 1

By applying the chain rule, we have, for the last layer:

$$
\delta_i^{(L)} = \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,in}^{(L)}} = \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,out}^{(L)}} \cdot \frac{\partial \mathbf{z}_{i,out}^{(L)}}{\partial \mathbf{z}_{i,in}^{(L)}} = \underbrace{\frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,out}^{(L)}}}_{\text{Loss-dependent}} \cdot {\sigma'}^{(L)}(\mathbf{z}_{i,in}^{(L)})
$$

Where the first term depends on the loss function. Using the chain rule again, the derivatives of the weights of a generic layer ℓ are:

$$
\begin{aligned}
\frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{W}_{ji}^{(\ell)}} &= \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,in}^{(\ell)}} \cdot \frac{\partial \mathbf{z}_{i,in}^{(\ell)}}{\partial \mathbf{W}_{ji}^{(\ell)}} \\
&= \delta_i^{(\ell)} \cdot \frac{\partial}{\partial \mathbf{W}_{ji}^{(\ell)}} \underbrace{\left(\sum_k \mathbf{W}_{ki}^{(\ell)} \mathbf{z}_{k,out}^{(\ell-1)} + \mathbf{b}_i^{(\ell)} \right)}_{\mathbf{z}_{i,in}^{(\ell)}} \\
&= \delta_i^{(\ell)} \cdot \mathbf{z}_{j,out}^{(\ell-1)}
\end{aligned}
$$

And, as for the bias:

$$
\begin{aligned}
\frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{b}_i^{(\ell)}} &= \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,in}^{(\ell)}} \cdot \frac{\partial \mathbf{z}_{i,in}^{(\ell)}}{\partial \mathbf{b}_i^{(\ell)}} \\
&= \delta_i^{(\ell)} \cdot \frac{\partial}{\partial \mathbf{b}_i^{(\ell)}} \underbrace{\left(\sum_k \mathbf{W}_{ki}^{(\ell)} \mathbf{z}_{k,out}^{(\ell-1)} + \mathbf{b}_i^{(\ell)} \right)}_{\mathbf{z}_{i,in}^{(\ell)}} \\
&= \delta_i^{(\ell)}
\end{aligned}
$$

Finally, the deltas of the previous layer are:

$$
\begin{aligned}
\delta_i^{(\ell-1)} &= \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,in}^{(\ell-1)}} \\
&= \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,out}^{(\ell-1)}} \cdot \frac{\partial \mathbf{z}_{i,out}^{(\ell-1)}}{\partial \mathbf{z}_{i,in}^{(\ell-1)}} \\
&= \left(\sum_k \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{k,in}^{(\ell)}} \cdot \frac{\partial \mathbf{z}_{k,in}^{(\ell)}}{\partial \mathbf{z}_{i,out}^{(\ell-1)}} \right) \cdot \sigma'^{(\ell-1)}(\mathbf{z}_{i,in}^{(\ell-1)}) \\
&= \left(\sum_k \delta_k^{(\ell)} \cdot \frac{\partial}{\partial \mathbf{z}_{i,out}^{(\ell-1)}} \left(\sum_l \mathbf{W}_{lk}^{(\ell)} \mathbf{z}_{l,out}^{(\ell-1)} + \mathbf{b}_k^{(\ell)} \right) \right) \cdot \sigma'^{(\ell-1)}(\mathbf{z}_{i,in}^{(\ell-1)}) \\
&= \left(\sum_k \delta_k^{(\ell)} \cdot \mathbf{W}_{ik}^{(\ell)} \right) \cdot \sigma'^{(\ell-1)}(\mathbf{z}_{i,in}^{(\ell-1)})
\end{aligned}
$$

Question 2

The trick to find vectorized formulas is to see how to compute the previous equations all at the same time via matrix multiplication. Always check the dimensionality of the matrices and vectors involved, to make sure the shape of the result matches what it should be.

For the last deltas, each neuron is treated independently from the others, therefore an element-wise multiplication between the two vectors does the job:

$$
\begin{aligned}
\delta^{(L)} &= \nabla_{\mathbf{z}_{\cdot,out}^{(L)}} \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)}) \odot \sigma'^{(L)}(\mathbf{z}_{\cdot,in}^{(L)}) \\
(N^{(L)} \times 1) &= (N^{(L)} \times 1) \odot (N^{(L)} \times 1)
\end{aligned}
$$

Where the second row indicates the dimensionality, rows times columns, of the elements involved.

To compute the gradients for the weights in Eq. 2, we multiply every activation of the previous layer by every delta of the current layer, resulting into a matrix which contains all combinations of $\mathbf{z}_{i,out}^{(\ell)}$ times $\delta_j^{(\ell)}$. This is computed as an "outer product":

$$
\begin{aligned}
\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)}) &= \mathbf{z}_{\cdot,out}^{(\ell-1)} \cdot \delta^{(\ell)T} \\
(N^{(\ell-1)} \times N^{(\ell)}) &= (N^{(\ell-1)} \times 1) \cdot (N^{(\ell)} \times 1)^T
\end{aligned}
$$

The gradient for the biases is easy to compute:

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

Finally, the deltas for the previous layer:

$$
\begin{aligned}
\delta^{(\ell-1)} &= \left(\mathbf{W}^{(\ell)} \cdot \delta^{(\ell)} \right) \odot \sigma'^{(\ell-1)}(\mathbf{z}_{\cdot,in}^{(\ell-1)}) \\
(N^{(\ell-1)} \times 1) &= \left((N^{(\ell-1)} \times N^{(\ell)}) \cdot (N^{(\ell)} \times 1) \right) \odot (N^{(\ell-1)} \times 1)
\end{aligned}
$$

Which follows because the sum in Eq. 4 is the dot-product of the i-th row of $\mathbf{W}^{(\ell)}$ with $\delta^{(\ell)}$. Doing this separately for each row results in the matrix-vector multiplication $\mathbf{W}^{(\ell)} \cdot \delta^{(\ell)}$.

Question 3

We now extend these formulas to handle batched data. Vectors become matrices where each row contains the vector for the corresponding sample in the batch:

- The sample labels become a matrix $\mathbf{Y}$, with $\mathbf{Y}_{ij}$ the label for the j-th output of the i-th sample;
- The hidden activations become $\mathbf{Z}^{(\ell)}_{out}$, with $\mathbf{Z}^{(\ell)}_{ij,out}$ the activation of the j-th unit in the ℓ-th layer for the i-th sample;
- The deltas become a matrix $\Delta^{(\ell)}$, where row i contains $\delta^{(\ell)}$ for the i-th example in the batch.

Remember, the first thing you should do to understand these formulas is to think at the dimensionality of the vectors and matrices involved and make sure they match.

The delta for the output layer is:

$$\Delta^{(L)} = \nabla_{\mathbf{Z}^{(L)}_{out}} \mathcal{L}(\mathbf{Y}, \mathbf{Z}^{(L)}_{out}) \odot \sigma'^{(L)}(\mathbf{Z}^{(L)}_{in})$$

which looks the same as Eq. 21 above, except that now we are using matrices instead of vectors. But the operation is the same: element-wise multiplication.

The gradient with respect to $\mathbf{W}^{(\ell)}$ is a bit more involved to compute, as it includes a three-dimensional tensor: the first dimension is for the samples, the second dimension is for the neurons of the $(\ell - 1)$-th layer, and the third dimension for the neurons of the ℓ-th layer. In other words, we are taking the gradients in Eq. 22, which are matrices, for each sample, and "stacking" them one on top of each other to get a "cube" of gradients. The element indexed by i, j, k is the derivative of the loss of the i-th sample in the batch with respect to $\mathbf{W}^{(\ell)}_{jk}$.

$$\left(\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}(\mathbf{Y}, \mathbf{Z}^{(L)}_{out})\right)_{ijk} = \left(\frac{\partial \mathcal{L}(\mathbf{Y}, \mathbf{Z}^{(L)}_{out})}{\partial \mathbf{W}^{(\ell)}_{jk}}\right)_i = \mathbf{Z}^{(\ell-1)}_{ij,out} \cdot \Delta^{(\ell)}_{ik}$$

To find the gradient of the weights with respect to the whole batch, we need to average this on the first dimension (the samples in the batch) to get the gradient:

$$\frac{\partial \mathcal{L}(\mathbf{Y}, \mathbf{Z}^{(L)}_{out})}{\partial \mathbf{W}^{(\ell)}_{jk}} = \sum_i \mathbf{Z}^{(\ell-1)}_{ij,out} \cdot \Delta^{(\ell)}_{ik}$$

If you look closely, you should realize that this is just a matrix product. Let's use a simpler notation to make it clear:

$$A_{jk} = \sum_i B_{ij} \cdot C_{ik} = \sum_i \left(B^T\right)_{ji} \cdot C_{ik}$$

Therefore, after much pain:

$$\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}(\mathbf{Y}, \mathbf{Z}^{(\ell)}_{out}) = \mathbf{Z}^{(\ell-1)^T}_{out} \cdot \Delta^{(\ell)}$$

The biases are straightforward, we just have to sum over the deltas of each sample:

$$\nabla_{\mathbf{b}^{(\ell)}} \mathcal{L}(\mathbf{Y}, \mathbf{Z}^{(L)}_{out}) = \sum_i \Delta^{(\ell)^T}_i$$

Finally, the deltas of the previous layer. From Eq. 4, each element is:

$$\Delta^{(\ell-1)}_{ij} = \left(\sum_k \Delta^{(\ell)}_{ik} \cdot \mathbf{W}^{(\ell)}_{jk}\right) \cdot \sigma'^{(\ell-1)}(\mathbf{Z}^{(\ell-1)}_{ij,in})$$

The sum is again a matrix product, therefore:

$$\Delta^{(\ell-1)} = \Delta^{(\ell)} \mathbf{W}^{(\ell)^T} \odot \sigma'^{(\ell-1)}(\mathbf{Z}^{(\ell-1)}_{in})$$

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:
        for layer in self.layers:
            x = layer.forward(x)
        return x.squeeze(-1)

    def get_loss(self, x: Tensor, y: Tensor) -> Tensor:
        """Compute the loss for a dataset and given labels."""
        return self.loss.forward(y, self.forward(x))

    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)


        # Compute the gradient of the loss.
        loss_grad = self.loss.get_gradient(y, y_pred)
```

```
        # Compute deltas for the output layer.
        z_in = self.layers[-1].last_zin
        act = self.layers[-1].act
        deltas = loss_grad.unsqueeze(-1) * act.get_gradient(z_in)

        # Traverse the whole network backwards and compute gradients on the way.
        # The gradients for the weights/biases are directly stashed within the laye
        for i in range(1, len(self.layers)):
            previous_z_out = self.layers[-(i + 1)].last_zout
            previous_z_in = self.layers[-(i + 1)].last_zin
            previous_act_func = self.layers[-(i + 1)].act

            current_layer = self.layers[-i]

            weight_grad = previous_z_out.T @ deltas
            bias_grad = torch.sum(deltas, dim=0)

            current_layer.set_weight_gradient(weight_grad)
            current_layer.set_bias_gradient(bias_grad)

            deltas = deltas @ current_layer.get_weight().T * previous_act_func.get_

        # Compute gradients for the input layer.
        self.layers[0].set_weight_gradient(x.T @ deltas)
        self.layers[0].set_bias_gradient(torch.sum(deltas, dim=0))


        # 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."""
        for layer in self.layers:
            if layer.linear.weight_grad is not None:
                layer.linear.weight -= learning_rate * layer.linear.weight_grad
            if layer.linear.bias_grad is not None:
                layer.linear.bias -= learning_rate * layer.linear.bias_grad
```

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

In [3]:

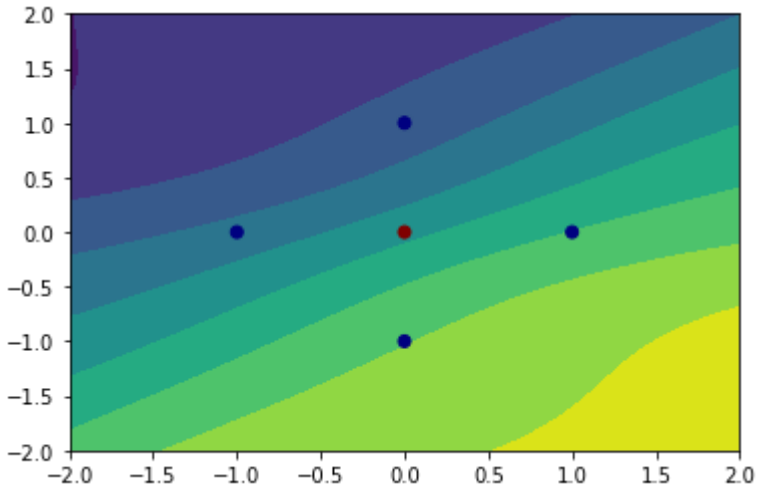
```
x = torch.tensor([
    [0, 0],
    [1, 0],
    [0, -1],
    [-1, 0],
    [0, 1]
], dtype=torch.float)
y = torch.tensor([1, 0, 0, 0, 0])

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

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

```
tensor([0.5000, 0.5401, 0.5680, 0.4599, 0.4320])
```

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]:

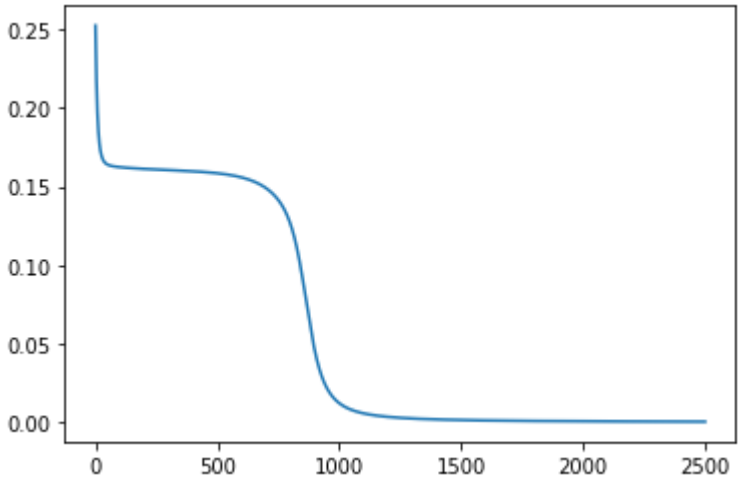
```
def train(
        x: Tensor,
        y: Tensor,
        net: NeuralNetwork,
        epochs: int,
        lr: float
) -> Tuple[NeuralNetwork, Tensor]:
    """
    Train a neural network.
    :param x: Training dataset.
    :param y: Training labels.
    :param net: Neural network to train.
    :param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


    losses: List = []
    for ep in range(1, epochs + 1):
        # Compute the loss (for tracking).
        loss = net.get_loss(x, y)
        losses.append(loss)

        # Backpropagate gradients.
        net.backward(x, y)

        # Apply gradients.
        net.apply_gradients(learning_rate=lr)

        print('EPOCH: \t {:5} \t LOSS: \t {:.5f}'.format(ep, float(loss)), end='\r'

    return net, torch.stack(losses)

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

```
EPOCH:     511   LOSS:   0.15836

EPOCH:    1014   LOSS:   0.01079

EPOCH:    1497   LOSS:   0.00152

EPOCH:    1990   LOSS:   0.00074

EPOCH:    2456   LOSS:   0.00048

EPOCH:    2500   LOSS:   0.00047
```

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

In [6]:

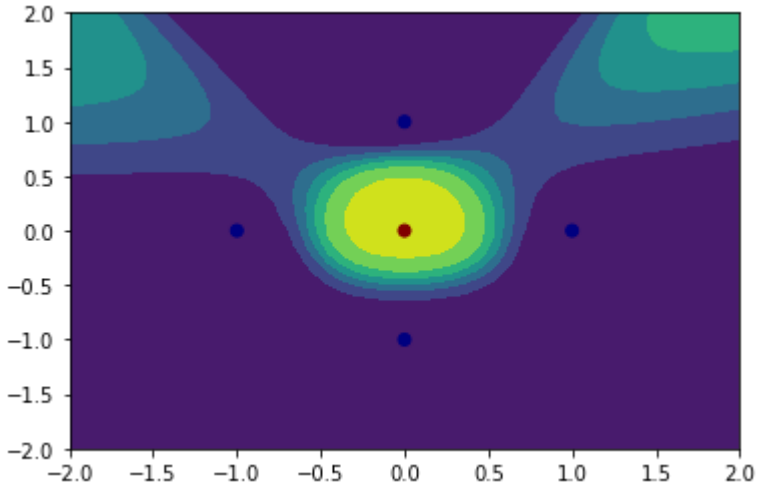
```
plt.plot(losses)
plt.show()
```



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.