Recurrent and Generative Artificial Neural Networks Assignment 1

Carla López Martínez (6637484) and Gaurav Niranjan (6599177)
carla.lopez-martinez@student.uni-tuebingen.de
gaurav.niranjan@student.uni-tuebingen.de

1 Exercise 1: Implementing Backpropagation

a) Backward Pass Implementation

These are the results of implementing backpropagation using the default parameters.

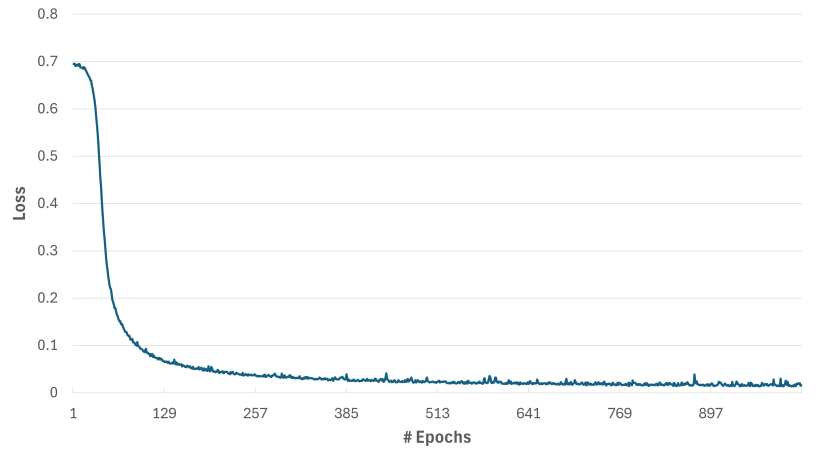

Figure 1: Loss over 1024 epochs

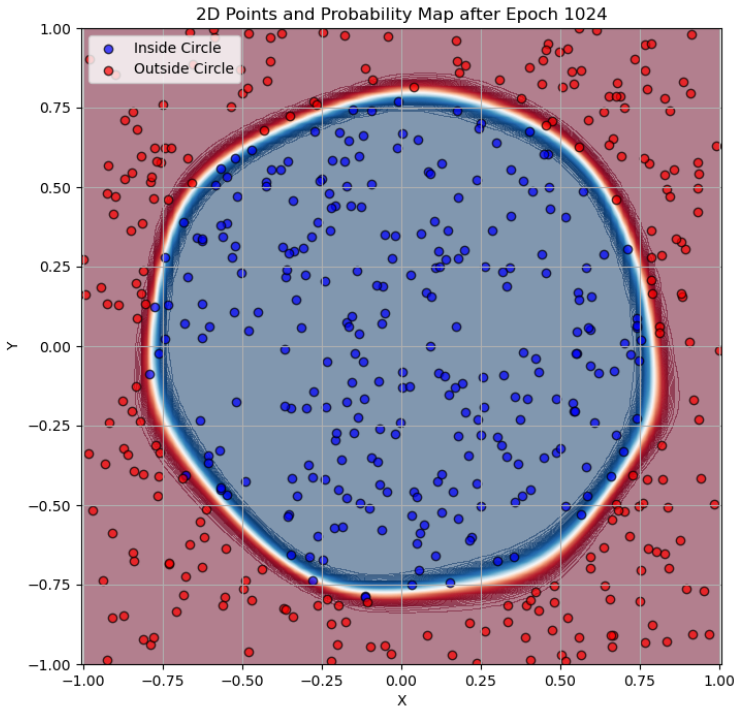

Figure 2: Final probability map

b) Cross-Entropy and Softmax Activation

1 Why is Cross-Entropy typically paired with the Softmax activation function?

They are typically paired because the softmax activation function gives values between 0 and 1 which add up to 1, which makes it possible to interpret its result as a probability distribution. This is useful in multi-class classification as we can use this probability distribution to decide to which class the input is more likely to belong to. Softmax activation also gives a smooth and differentiable result, which is important when using cross-entropy loss and backpropagation (which requires gradient computation and convergence) to train the neural network.

2 Derive the Gradient for Cross-Entropy with Softmax

The cross-entropy loss is:

$$L = -\sum_j y_j \log(\text{softmax}(z_j)) \tag{1}$$

The softmax function is:

$$\text{softmax}(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}} \tag{2}$$

Where k indexes all the classes and j refers to a specific output/target. The gradient of L with respect to z_j is:

$$\frac{\partial L}{\partial z_j} = -\sum_k y_k \frac{\partial}{\partial z_j} \log\left(\text{softmax}(z_k)\right) \tag{3}$$

$$\frac{\partial L}{\partial z_j} = -\sum_k y_k \left(\frac{1}{\text{softmax}(z_k)} \cdot \frac{\partial}{\partial z_j} \text{softmax}(z_k)\right) \tag{4}$$

The derivative of softmax is:

$$\frac{\partial \text{softmax}(z_k)}{\partial z_j} = \text{softmax}(z_k)\left(\delta_{kj} - \text{softmax}(z_j)\right) \tag{5}$$

Substituting into (4) we get:

$$\frac{\partial L}{\partial z_j} = -\sum_k y_k \left(\delta_{kj} - \text{softmax}(z_j)\right) \tag{6}$$

δ_{kj} is the Kronecker delta, which is 1 if $k = j$ and 0 otherwise. So we can simplify (6) as:

$$\frac{\partial L}{\partial z_j} = \text{softmax}(z_j) - y_j \tag{7}$$

3 Alternative Normalization Without Exponentiation

We define the cross-entropy as:

$$L = -\sum_j y_j \log(\frac{z_j}{\sum_k z_k}) \tag{8}$$

$$L = -\sum_j y_j \log(z_j) + \sum_j y_j \log(\sum_k z_k) \tag{9}$$

Deriving we get:

$$\frac{\partial L}{\partial z_j} = -\sum_k y_k (\frac{1}{z_j} - \frac{1}{\sum_k z_k}) \tag{10}$$

Only when k = j will $y_k = 1$, so:

$$\frac{\partial L}{\partial z_j} = -\frac{1}{z_j} + \frac{1}{\sum_k z_k} \tag{11}$$

In this case, there isn't a non-linear component in the gradient as we removed the exponentiation. This means changes in z_j will linearly affect the result instead of small changes potentially causing exponential variations in the result as with softmax. This means less sensitivity to differences, which may result in slower convergence and less steep gradients.

2 Exercise 2: Getting familiar with PyTorch

2.1 Backward Pass Implementation

Given:
grad_output: the gradient of the loss with respect to the output of this layer,
weight: the weight matrix of this layer,
input: the input to this layer,
bias: the bias of this layer.

The gradients are calculated as follows:

1. Gradient with respect to the input (grad_input):
   grad_input = grad_output × weight

2. Gradient with respect to the weight matrix (grad_weight):
   grad_weight = $\text{grad\_output}^T \times \text{input}$

3. Gradient with respect to the bias (grad_bias):
   grad_bias = $\sum \text{grad\_output}_{(\text{dim}=0)}$

2.2 Gradient Test Case Implementation

The random tensors required are created by torch.randn or torch.rand and some further manipulation for specific cases, as done in the code.

2.3 Implementing the Adam Optimizer

Given:

g : gradient of the parameter p

b_1, b_2 : exponential decay rates for the moment estimates

ϵ : a small constant to avoid division by zero

α : learning rate (lr)

state['m'] : first moment (moving average of the gradient)

state['v'] : second moment (moving average of the squared gradient)

state['step'] : current timestep (incremented with each step)

1. Update biased first moment estimate:
   $m_t = b_1 \cdot m_{t-1} + (1 - b_1) \cdot g$

2. Update biased second moment estimate:
   $v_t = b_2 \cdot v_{t-1} + (1 - b_2) \cdot g^2$

3. Compute bias-corrected first moment estimate:
   $\hat{m}_t = \dfrac{m_t}{1 - b_1^{\text{state['step']}}}$

4. Compute bias-corrected second moment estimate:
   $\hat{v}_t = \dfrac{v_t}{1 - b_2^{\text{state['step']}}}$

5. Update parameters:
   $p = p - \alpha \cdot \dfrac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$

Implementation in optimization.py file.

3 Exercise 3: Exploring the Decision Boundary

3.1 Describe the Network Configuration

1. Input Layer: The input layer has 2 neurons since our input has 2 features.
2. Hidden Layer: We use 1 hidden layer with 4 neurons and ReLU activation.
3. Output Layer: The output layer has 1 neuron and linear activation.

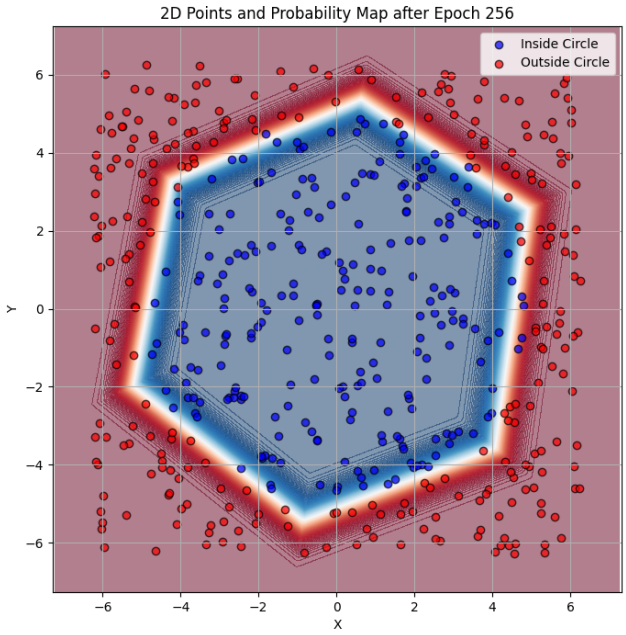

Figure 3: Hexagonal decision boundary

3.2 Explain the Hexagonal Boundary

The hexagonal decision boundary is obtained primarily because of ReLU activation function used in the hidden layer. ReLU inherently introduces sharp angles in decision boundaries as it is linear for positive inputs and zero for negative inputs and this tends to partition the feature space with many piecewise linear functions. While some other activation like tanh, which is smooth and differentiable across the entire input range, tend to approximate boundaries that are more curved.