Homework 1

Xiaoyan Ouyang

1.(a)

A box function with height h and width δ is defined as:

$$f(x) = \begin{cases} h, & 0 < x < \delta \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

To approximate this using a neural network with two hidden neurons and step activation $\sigma(u)$:

- The first neuron detects when $x > 0$.
- The second neuron detects when $x > \delta$.
- The output neuron computes the difference of these activations, scaled by h.

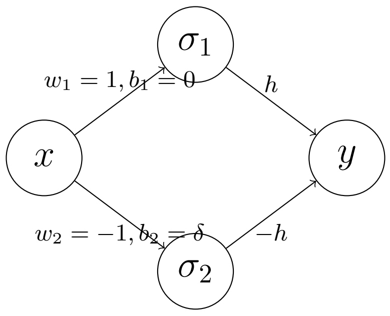
Define the neurons as:

$$\begin{aligned} z_1 &= \sigma(x - 0) = \sigma(x), \\ z_2 &= \sigma(x - \delta). \end{aligned}$$

Then, the output neuron computes:

$$y = h(z_1 - z_2) = h(\sigma(x) - \sigma(x - \delta)). \tag{2}$$

This correctly produces h for $0 < x < \delta$ and 0 otherwise. The network in the following shows all the weights and biases.



1.(b)

For any smooth, bounded function $f(x)$ over $[-B, B]$, we approximate it by summing multiple box functions. We partition $[-B, B]$ into small intervals $[x_i, x_{i+1}]$ of width δ and set the height of each box as $f(x_i)$.

The neural network consists of many neurons, each representing a box function:

$$y = \sum_i h_i(\sigma(x - x_i) - \sigma(x - x_{i+1})). \tag{3}$$

For small δ, the sum of these box functions approximates $f(x)$ closely.

1.(c)

This argument can be extended to d-dimensional inputs by constructing small box functions in $\mathbb{R}^d$ using multiple neurons per dimension. However, practical issues arise:

- The number of neurons grows exponentially with dimension d (curse of dimensionality).
- Efficiently selecting neurons and weights becomes computationally difficult.

In the 1-dimensional case, we only need 2 neurons to construct a Box function. However, in the d-dimensional case, if we want to model a high-dimensional hyperrectangle $[0, \delta_1] \times [0, \delta_2] \times \cdots \times [0, \delta_d]$, each dimension requires 2 neurons (similar to the "start detection" and "end detection" in the 1-dimensional case). Therefore, the total number of neurons is approximately 2^d (as each dimension requires 2 neurons). In the 1-dimensional case, we only need to manually adjust a few weights, but in higher-dimensional space, we need to precisely set weights and biases for each dimension to ensure that the Box function is correctly activated. The total number of connections (weights) also grows exponentially, leading to a significant computational overhead.

2.(a)

Given a fully-connected layer with n_{in} inputs and n_{out} outputs, let the weights be initialized as

$$W_{ij} \sim \mathcal{N}(0, \sigma^2) \tag{4}$$

and assume the inputs x_i are independent, zero-mean random variables with variance

$$\operatorname{Var}(x_i) = v^2. \tag{5}$$

The pre-activation a_j for neuron j is given by:

$$a_j = \sum_{i=1}^{n_{in}} W_{ij} x_i. \tag{6}$$

Since W_{ij} and x_i are independent, so:

$$\operatorname{Var}(a_j) = \sum_{i=1}^{n_{in}} \operatorname{Var}(W_{ij} x_i). \tag{7}$$

Using the fact that $\operatorname{Var}(AB) = \operatorname{Var}(A)\operatorname{Var}(B)$ for independent A and B, we get:

$$\operatorname{Var}(a_j) = \sum_{i=1}^{n_{in}} \sigma^2 v^2 = n_{in} \sigma^2 v^2. \tag{8}$$

2.(b)

For stability, we require that $\operatorname{Var}(a_j)$ remains of the same order as v^2. That is,

$$n_{in} \sigma^2 v^2 \approx v^2. \tag{9}$$

Solving for σ^2:

$$\sigma^2 \approx \frac{1}{n_{in}}. \tag{10}$$

If σ^2 is too small, the variance of activations will shrink layer by layer, leading to vanishing activations. If σ^2 is too large, the activations will explode, leading to unstable gradients. Thus, we would better to set:

$$\sigma^2 = \frac{1}{n_{in}}. \tag{11}$$

2.(c)

In the case of the ReLU activation function ($\text{ReLU}(x) = \max(0, x)$), the output distribution of the network changes. A key property of ReLU is that it outputs zero for negative inputs while retaining positive inputs unchanged. When the input follows a symmetric distribution (such as a zero-mean Gaussian distribution), approximately 50% of the neurons' outputs become zero, thereby reducing the variance of the entire layer by half. To maintain the variance of activation values during forward propagation, we have:

$$\sigma^2_{\text{ReLU}} = \frac{2}{n_{\text{in}}}$$

Compared to the case without an activation function (where fully connected layers typically use $\frac{1}{n_{\text{in}}}$), we need to multiply by 2 to compensate for the loss of half of the activation values due to ReLU. Otherwise, the variance signal would decay layer by layer.

2.(d)

The gradients are multiplied by the transpose of W:

$$g_i = \sum_{j=1}^{n_{\text{out}}} W_{ij} g_j$$

Since g_j are independent and $W_{ij} \sim \mathcal{N}(0, \sigma^2)$, using the variance addition formula:

$$\text{Var}(g_i) = \sum_{j=1}^{n_{\text{out}}} \text{Var}(W_{ij} g_j)$$

Since weights and gradients are independent:

$$\text{Var}(W_{ij} g_j) = \text{Var}(W_{ij}) \, \text{Var}(g_j) = \sigma^2 g^2$$

Thus, we obtain:

$$\text{Var}(g_i) = n_{\text{out}} \sigma^2 g^2$$

To prevent gradient vanishing or explosion, we require:

$$\text{Var}(g_i) \approx g^2$$

Substituting into the equation:

$$n_{\text{out}} \sigma^2 g^2 \approx g^2$$

Canceling g^2 (assuming it is nonzero):

$$\sigma^2 \approx \frac{1}{n_{\text{out}}}$$

Therefore, a reasonable initialization strategy is:

$$\sigma^2 = \frac{1}{n_{\text{out}}}$$

This ensures that the variance of gradients remains stable during propagation.

2.(e)

To balance both forward and backward signal propagation, we take the average of the constraints found in parts (b) and (d):

$$\sigma^2 = \frac{2}{n_{in} + n_{out}}. \tag{12}$$

This is the Xavier (Glorot) initialization, ensuring neither vanishing nor exploding signals during training.