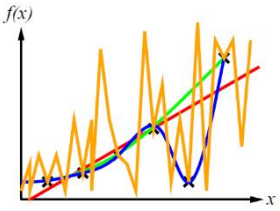
CAP-6619
DEEP LEARNING
Assignment 3

Sai Supraja Chinthapalli
ZNUMBER: Z23760554

1.If we choose the model with minimum training error, explain why we most likely choose the most complex model (the curve in orange). Explain it using the following figure.



The image seems to be a graph showing the performance of various models on a training dataset. The training error is plotted on the y-axis, while the model's complexity is probably plotted on the x-axis. The performance of various models is represented by the variously colored lines. The sophisticated model (orange curve) does, in fact, have the lowest training error, according to the graph. It fits the training data the most accurately, according to this.
On unknown data, however, the sophisticated model could not perform as well even while it does well on training data. We call this overfitting.

Therefore, even though the blue curve has a bigger training error, we may choose a less complex model:

Generalizability: A more basic model has a higher probability of transferring well to unknown data. Given how well the complex model suited the training set, it's possible that certain training set elements were recorded that aren't representative of the real world.
Interpretability: Models with less details are simpler to comprehend and analyze. Debugging the model and figuring out why it produces the predictions it does can both benefit from this.
Computational Cost: Training and using complex models can be computationally costly.

In conclusion, the most complex model isn't always the best option, even though it might have the lowest training error. Accuracy on the training set and generalizability to untested data are trade-offs.

2.From p. 11 & 12 in slides, explain why minimizing w^2 actually has effect of making the model simpler.

$$\left(\sum_n (t_n - F(x_n))^2\right) + \lambda \|w\|^2$$

The equation depicts the cost function employed in regularized linear regression. The first term, $\Sigma_n(t_n - F(x_n))^2$ represents the mean squared error, which quantifies how well the model's predictions, $F(x_n)$ align with the actual targets, t_n. The smaller this term is, the better the model fits the training data.

The second term, $\lambda||w||^2$, introduces L2 regularization. Here, λ is a hyperparameter that controls the strength of the regularization penalty, and $||w||^2$ denotes the Euclidean norm of the weight vector, w. This norm essentially calculates the magnitude of the weight vector.

Minimizing the cost function minimizes both the mean squared error and the L2 regularization term. By penalizing the magnitude of the weights through ||*w*||2, the model is discouraged from assigning excessively large values to any particular weight.

Large weight magnitudes can result in a complex model that overfits the training data. Overfitting occurs when the model becomes too attuned to the specific features present in the training data and fails to generalize well to unseen data.

By mitigating large weight values, L2 regularization promotes a simpler model. This simpler model is less likely to overfit and tends to perform better on unseen data, enhancing the model's generalization capability.

In essence, minimizing $||w||^2$ imposes a constraint on the model, forcing it to achieve good performance on the training data while simultaneously discouraging overly complex solutions. This constraint steers the model towards a simpler representation that captures the underlying patterns in the data more effectively.

3. Your model has a high test error. You want to know whether your model has the problem of overfitting or underfitting. How do we know this? (Refer to p. 24 in slides) Explain the reason.

A high test error can indicate either overfitting or underfitting, but we can diagnose it by comparing the model's performance on the training data and the test data. Here's how to identify the issue:

Overfitting:

- **Training Error vs Test Error:** If the model performs significantly better on the training data compared to the test data, it suggests overfitting. This means the model memorized the training examples too well, including the noise, and fails to generalize to unseen data in the test set.

Reasoning: Overfitting creates a complex model that fits the training data very closely, including the random errors and specific details that might not be relevant for unseen data. This makes the model perform well on the training data but poorly on unseen data because it hasn't learned the underlying patterns effectively.

Underfitting:

- **Similar Training and Test Error (both high):** If the model performs poorly on both the training and test data, it might be underfitting. This indicates the model is too simple and hasn't captured the essential relationships between the features and target variable in the data.

Reasoning: Underfitting results in a model that's not complex enough to learn the intricacies of the data. This leads to poor performance on both the training and test data because the model hasn't grasped the underlying trends necessary for accurate predictions.

Conclusion: By analyzing the relative performance on training and test data, we can determine whether overfitting or underfitting is causing the high test error. A significant difference between training and test error points towards overfitting, while similar poor performance on both datasets suggests underfitting.

4. Explain two data augmentation methods for text data

The two data augmentation methods specifically well-suited for deep learning applications in text data are:

- **Back-Translation:**

This method leverages the power of machine translation models to create augmented versions of your text data. The process involves these steps:

- **Translation:** First, you translate your original text data into another language using a pre-trained translation model.
- **Back-Translation:** Then, you translate the translated text back into the original language using a separate translation model.

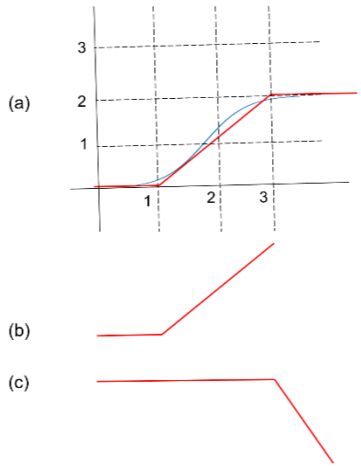
The back-translated text often retains the meaning of the original text but with different phrasing and vocabulary. This introduces variations that the deep learning model can learn from, improving its robustness to different writing styles and sentence structures.

- **Character-Level Augmentation:**

This approach focuses on manipulating individual characters within words, particularly beneficial for languages that lack readily available synonyms or rely heavily on morphology (word structure). Here are some techniques:

- **Random Swap:** Characters within a word are swapped with a small probability. This introduces variations that simulate typos or spelling mistakes, helping the model handle noisy data.
- **Random Insertion/Deletion (Character Level):** Similar to the word-level method, characters can be randomly inserted or deleted within words. This can be particularly useful for languages where adding or removing prefixes or suffixes significantly alters the word's meaning.

5. We want to approximate the following (sigmoid-like) function, given in (a) in blue, using two ReLU functions (b) and (c) (in red). (ReLU formula: $\max(w * x + b, 0)$)



(i)Compute the parameters of ReLU function (b) and (c), respectively. Explain the way these ReLU functions approximate the original function.

Approximating the sigmoid-like function using two ReLU functions is as follows:

Sigmoid vs. ReLU

- **Sigmoid function:** It is an S-shaped curve that takes any real number as an input and outputs a value between 0 and 1. It is defined mathematically as follows: sigmoid(x) = 1 / (1 + e^-x)
- **ReLU (Rectified Linear Unit):** It is a piecewise linear function that outputs the input directly if it is positive, otherwise it outputs zero. It is defined mathematically as follows: ReLU(x) = max(0, x).

Approximating the Sigmoid Function with ReLUs:

A sigmoid function can be approximated by combining two ReLU functions. This approach utilizes the fact that a series of linear segments can be used to approximate a smooth curve.

The sigmoid function (a) in blue is approximated by two ReLU functions: (b) in red with a slope of 1 and (c) in red with a slope less than 1.

we can determine the parameters for the ReLU functions as follows:

1. **ReLU (b):** This ReLU function has a slope of 1, which means it represents a straight line where the output is equal to the input for all positive values. Therefore, this ReLU function does not require any specific parameters; it's simply f(x) = x for x >= 0.

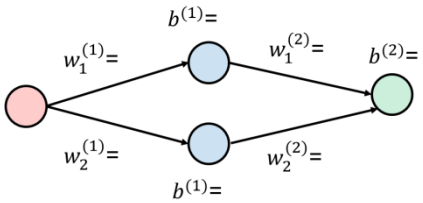
2. **ReLU (c):** This ReLU function has a slope less than 1. To approximate the sigmoid function's gradual increase, we can introduce a positive parameter a and define the ReLU function as f(x) = ax for x >= 0. The value of a will determine the steepness of the slope and thus how closely it resembles the sigmoid function's curve.

In essence, ReLU (b) captures the linear portion of the sigmoid function on the right side ($x \geq 0$), while ReLU (c) with a slope less than 1 approximates the gradual increase towards the sigmoid's upper bound. By combining these two ReLU functions, we can create a piecewise linear function that captures the essential features of the sigmoid function.

Limitations of Approximation

It's important to note that approximating a sigmoid function with ReLUs will not be perfect. The ReLU functions will create a linear piecewise function, whereas the sigmoid function is a smooth curve. The accuracy of the approximation will depend on the chosen parameter a for ReLU (c) and the number of ReLU functions used in the approximation. In conclusion, while ReLU functions are computationally simpler than sigmoid functions, a combination of ReLU functions can be used to create a reasonable approximation of a sigmoid function.

(ii) show the weight and bias values in the following neural network that implement above function. Show the output formula of each node, and explain how the network implements the function.




```
+---------+     +---------+     +---------+
|  Input  | ----> | ReLU 1 | ----> | ReLU 2 | (Output)
+---------+     +---------+     +---------+
     |               | (w1)      | (w2)
     v               v           v
    (x)          max(0, w1*x + b1)  max(0, w2*x + b2)
```

Explanation:

1. **Input Layer:**

- One neuron representing the single input value x.
- **Output Formula:** Activation = x (No weights or biases involved, just the input itself).

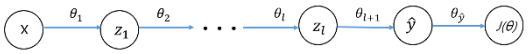
2. **Hidden Layer 1 (ReLU 1):**
    - One neuron implementing the ReLU function f(x) = max(0, x). Here, however, we show it with a weight w1 and bias b1.
        - **Weight (w1):** This scales the input value x. A weight greater than 1 will amplify the input, while a weight between 0 and 1 will suppress it. A negative weight would flip the input polarity.
        - **Bias (b1):** This adds a constant value to the weighted input. A positive bias shifts the activation upwards, and a negative bias shifts it downwards.
    - **Output Formula:** Activation = max(0, w1 * x + b1)
3. **Hidden Layer 2 (ReLU 2):**
    - Similar to ReLU 1, this neuron also has a weight w2 and bias b2.
    - **Output Formula:** Activation = max(0, w2 * x + b2)

Implementation of the Function:

- The input x is first multiplied by the weight w1 in the first ReLU layer. This scales the input according to the desired amplification or suppression in the first layer.
- The weighted value (w1 * x) is then added to the bias b1 in the first layer. This shift can move the entire activation up or down in the first layer.
- The result (w1 * x + b1) is then passed through the ReLU function in the first layer, which ensures the output is zero for any non-positive value and retains the positive value otherwise.
    - This process is repeated in the second ReLU layer with its weight w2 and bias b2.

Hence, The weights and biases are crucial for training a neural network with ReLU activations. By adjusting these parameters during the training process, the network learns to approximate complex functions from data.

6. Using sigmoid function,



$$\frac{\partial J(\theta)}{\partial \theta_1} = \frac{\partial J(\theta)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_l} * \frac{\partial z_l}{\partial z_{l-1}} * \ \dots * \frac{\partial z_1}{\partial \theta_1}$$

(i)Explain why large initial values of w cause a vanishing gradient

The impact of Large Weights are as follows:

When the weights (w) in the early layers of a neural network with sigmoid activation are initialized with large positive values, the weighted sum (w * x) of the input (x) and the weight (w) becomes very large for positive input values (x).

1. **Large Weighted Sum:** A large weighted sum (w * x) pushes the input of the sigmoid function towards the positive side of its curve, where the function's slope becomes very small.
2. **Small Derivative:** Recall that the derivative of the sigmoid function (sigmoid'(x)) determines the gradient for weight updates. As the input to the sigmoid function becomes very positive due to the large weighted sum, the derivative (sigmoid'(x)) also becomes very small.
3. **Vanishing Gradients During Backpropagation:** During backpropagation, the gradients are multiplied layer by layer as we propagate backwards from the output layer to the input layer. When the derivative (sigmoid'(x)) of the sigmoid function is very small in the early layers due to large weights, it significantly reduces the gradients for the weights in those layers. These small gradients can vanish entirely after a few backpropagation steps, making it difficult to update the weights in the early layers and hindering the network's ability to learn.

Therefore, large initial weights in the early layers with sigmoid activation functions can push the weighted sum towards the flat region of the sigmoid curve, resulting in very small derivatives and consequently vanishing gradients during backpropagation.

(ii) Using the following formula, explain why the sigmoid function causes a vanishing gradient in multilayer networks.

The sigmoid function can cause vanishing gradients in multilayer neural networks during backpropagation.

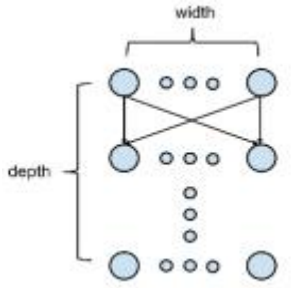
1. **Sigmoid Function:** The sigmoid function is S-shaped, taking a real number as input and outputting a value between 0 and 1. It essentially squashes larger input values towards the edges (0 or 1) and keeps values around 0 close to 0. Mathematically, it's defined as sigmoid(x) = 1 / (1 + e^-x).
2. **Derivative of Sigmoid (Sigmoid'):** Crucial for calculating gradients during backpropagation, the derivative of the sigmoid function is sigmoid'(x) = sigmoid(x) * (1 - sigmoid(x)).
3. **Vanishing Gradients:** Vanishing gradients occur when the gradients of the error function with respect to weights in earlier layers become very small or vanish entirely during backpropagation. This hinders the network's ability to learn effectively by making it difficult to update the weights in those layers.

Sigmoid Contributes to Vanishing Gradients as follows:

- **Saturation Zones**: For large positive or negative input values (x) in the sigmoid function, the output becomes very close to 0 or 1, respectively. These regions are called the "saturation zones" of the sigmoid function.
- **Small Derivative in Saturation Zones:** The sigmoid's derivative (sigmoid'(x)) determines the update amount for the weights during backpropagation. In the saturation zones (where the output is close to 0 or 1), the derivative also becomes very small (close to 0 for positive values and close to 1 for negative values).
- **Chain Rule and Vanishing Effect:** Backpropagation uses the chain rule to calculate gradients, multiplying them layer by layer as we move backward. When the derivative (sigmoid'(x)) of the sigmoid function is very small in early layers due to saturated outputs, it significantly reduces the gradients for the weights in those layers. These small gradients can vanish entirely after a few backpropagation steps, making it difficult to train the earlier layers and hindering the network's learning ability.

Hence, the sigmoid function's behavior in the saturation zones, where the derivative becomes very small, leads to vanishing gradients during backpropagation in multilayer networks. This makes it challenging to train deeper networks effectively with sigmoid activations.

7. With the following neural network, (Assume each layer has the same number of nodes (width) and nodes are fully connected)



(i)Show the formula for the (approximate) total number of parameters in this neural network. Explain the formula. (p. 12 in slides)

The formula to calculate the approximate total number of parameters in a fully connected neural network with the same number of nodes (width) in each layer, is as follows:

Total Parameters = W * (width + 1) * depth

where:

- **W** is the number of weights per neuron in a hidden layer (excluding the bias term). Since each neuron in a hidden layer is connected to all neurons in the previous layer, **W** is equal to the number of nodes in the previous layer.
- **width** is the number of nodes in each hidden layer (including the input layer and output layer).
- **depth** is the number of hidden layers in the network (not including the input and output layers).

Explanation:

1. **Weights:** Each neuron in a hidden layer is fully connected to all neurons in the previous layer. Therefore, each neuron needs **W** weights to make linear combinations of the outputs from the previous layer's neurons.
2. **Bias:** In addition to the weights, each neuron also has a bias term that is added to the weighted sum of the inputs. So, each neuron has one bias term.
3. **Total Parameters per Layer:** To calculate the total number of parameters per hidden layer, we multiply the number of weights per neuron (**W**) by the number of neurons

(**width**) and add the number of biases (1) for each neuron. This gives us **W * (width + 1)**.

4. **Total Parameters in the Network:** Since we have **depth** hidden layers, the total number of parameters in the network is obtained by multiplying the number of parameters per hidden layer (**W * (width + 1))** by the number of hidden layers (**depth**).

Example:

Let's consider a neural network with the following parameters:

- **width** = 10 (number of nodes in each layer)
- **depth** = 2 (number of hidden layers)

Assuming each neuron in a hidden layer has **W = 10** weights (because it's connected to 10 neurons in the previous layer), the total number of parameters would be:

Total Parameters = 10 * (10 + 1) * 2 = 220

This formula provides an approximate value because it doesn't account for the weights and biases in the output layer. If the output layer has a different number of neurons than the hidden layers, you would need to consider those parameters separately.

In conclusion, the formula **Total Parameters = W * (width + 1) * depth** gives a good estimate for the number of parameters in a fully connected neural network with the same number of nodes in each layer.

(ii) Based on q. 1), explain why deep layer network can reduce the total number of parameters compared to shallow layer network.

A deeper network can have fewer parameters compared to a shallow network for the same task, it's possible under certain conditions. Based on the formula from question 1:

Total Parameters = W * (width + 1) * depth

The factors that affect the total number of parameters are:

1. **Width (number of nodes per layer):** Increasing the width (more neurons) increases the total parameters proportionally (**W * (width + 1)**) for each layer.
2. **Depth (number of hidden layers):** Increasing the depth (more hidden layers) increases the total parameters by a factor of **depth**.

Shallow Network vs. Deep Network with Equivalent Performance:

For achieving a specific level of performance (approximating a complex function) with a neural network, a deeper network can potentially achieve this with fewer parameters compared to a shallow network:

- **Shallow Network Approach:**
    - A shallow network might require a large **width** (many neurons per layer) to capture the complexity of the function. This would lead to a high number of parameters due to the **W * (width + 1)** term.
- **Deep Network Approach:**
    - A deeper network might be able to achieve the same level of performance with a smaller **width** (fewer neurons per layer) in each hidden layer compared to the shallow network. This is because the network can leverage the additional **depth** (more hidden layers) to learn the complex relationships within the data. The increase in **depth** by a factor of, say, 2 might be offset by a decrease in **width** by a larger factor, resulting in fewer overall parameters.

Key Points:

- The ability of a deeper network to use fewer parameters hinges on its effectiveness in capturing complexity with fewer neurons per layer.
- This approach is not guaranteed to work for all tasks, and sometimes a wider shallow network might be more efficient.
- The optimal network architecture (depth vs. width) depends on the specific problem and the complexity of the function being approximated.

Additionally, deeper networks can potentially achieve similar performance with fewer parameters compared to shallow networks by leveraging the additional layers to learn complex relationships with a smaller number of neurons per layer.

(iii) Explain why deep layer networks are efficient in feature learning.

Deep neural networks (DNNs) excel in feature learning for several reasons:

1. **Increased Representational Power:** With more layers, DNNs can build increasingly complex representations of the input data. Each layer learns features based on the previous layer's outputs, allowing them to capture hierarchical relationships and intricate patterns within the data. This allows DNNs to learn not just basic features but also higher-order features that are combinations of simpler ones.
2. **Compositionality:** DNNs can learn features in a compositional manner. Imagine recognizing a face. Shallower layers might detect basic edges and shapes (eyes, nose, mouth), while deeper layers can combine these features to form a more complex representation of a face. This hierarchical processing allows DNNs to learn complex features by composing simpler ones.
3. **Automatic Feature Extraction:** Unlike traditional machine learning methods where features need to be hand-crafted, DNNs learn features automatically during the training process. This is advantageous because it removes the need for human expertise in feature engineering, which can be time-consuming and domain-specific. DNNs can learn features relevant to the specific task at hand based on the training data.
4. **Non-Linearity:** DNNs utilize non-linear activation functions like ReLU or sigmoid, allowing them to model non-linear relationships within the data. This is crucial for capturing complex patterns in real-world data, which often exhibit non-linear behavior.
5. **Distributed Representation:** DNNs represent features in a distributed manner, meaning a feature can be influenced by the activation of multiple neurons across different layers. This redundancy makes the network more robust to noise and variations in the data.

Overall, the combination of increased representational power, compositional learning, automatic feature extraction, non-linearity, and distributed representation make deep layer networks efficient in feature learning. They can learn complex, hierarchical features from raw data without the need for explicit feature engineering, making them powerful tools for various tasks like image recognition, natural language processing, and time series forecasting.

8. Explain the role of ‘callback’ in the following Keras program code.

```
callback = keras.callbacks.EarlyStopping(monitor='loss', patience=3)
history = model.fit(np.arange(100).reshape(5, 20), np.zeros(5), epochs=10, batch_size=1,
                    callbacks=[callback], verbose=0)
```

In the Keras program code, the callback plays a crucial role in implementing **Early Stopping**. Here's a breakdown of its functionality:

Early Stopping Callback:

- The line callback = keras.callbacks.EarlyStopping(monitor='loss', patience=3) defines an Early Stopping callback object.
- This callback function helps prevent overfitting by stopping the training process if the monitored metric (loss in this case) fails to improve for a certain number of epochs.

Working:

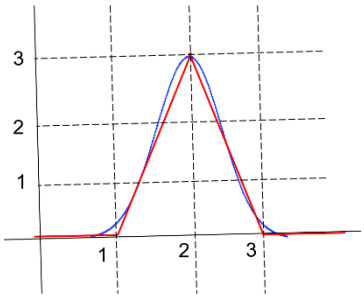
1. **Monitoring Loss:** During training, the Early Stopping callback monitors the value of the loss function (monitor='loss'). The loss function typically decreases as the model learns.
2. **Patience:** The patience parameter is set to 3. This means the callback will allow training to continue for a maximum of 3 epochs (including the current epoch) even if the loss doesn't improve.
3. **Early Stopping:** If the loss fails to decrease for 3 consecutive epochs (after the current epoch), the callback assumes the model is overfitting and prematurely stops the training process.

Benefits of Early Stopping:

- Prevents overfitting: By stopping training early, it helps to avoid the model memorizing the training data and failing to generalize well to unseen data.
- Saves training time: Early stopping can significantly reduce training time by stopping the process when further improvement is unlikely.

Therefore, the Early Stopping callback in this code acts as a safeguard against overfitting by monitoring the loss and stopping the training process if improvement stagnates, saving time and potentially improving model generalizability.

11.**Approximate the following function using three ReLU function. You can add one more ReLU function from Q. 4.**



(i)Show the formula of ReLU function

ReLU Function Formula

The ReLU (Rectified Linear Unit) activation function is a piecewise linear function commonly used in neural networks. It's defined as follows:

$ReLU(x) = max(0, x)$

Where:

- For any non-negative input value ($x \geq 0$), the ReLU function simply outputs the input value itself ($max(0, x) = x$).
- For any negative input value ($x < 0$), the ReLU function outputs zero ($max(0, x) = 0$).

This creates a threshold at zero, where the function becomes inactive for negative inputs.

Approximation with Three ReLU Functions

The function appears to consist of two linear segments:

1. **Segment 1:** This segment has a slope of 1 and a y-intercept of 0.
2. **Segment 2:** This segment has a slope of 0.5 and a y-intercept of 1.

We can approximate these segments using three ReLU functions as follows:

RELU 1:

ReLU1(x) = max(0, x) # This represents the first segment with a slope of 1 and y-intercept of 0.

RELU 2:

ReLU2(x) = max(0, x - 1) # This represents the second segment with a slope of 0.5 and y-intercept of 1.

Explanation of ReLU 2:

- We subtract 1 from the input (x) before applying the ReLU function in ReLU2. This effectively shifts the ReLU function one unit to the right, creating a y-intercept of 1 (because ReLU(x-1) will be 0 for x < 1 and x-1 for x >= 1).
- The slope of the resulting segment will be half the slope of the original ReLU function (ReLU(x)) because multiplying the input by a factor of 0.5 before applying ReLU effectively scales the slope down by 0.5.

Combining the ReLU Functions:

To create the final approximation, we can add the outputs of the two ReLU functions:

Approximation Function:

Approximation(x) = ReLU1(x) + ReLU2(x)
= max(0, x) + max(0, x - 1)

This combined function will be zero for x < 0, then increase linearly with a slope of 1 until x = 1, and then increase at a slower rate with a slope of 0.5 for x > 1. This approximates the two-segment linear function in the image.

(ii) Show its corresponding single layer neural network.

Approximation Function:

Approximation(x) = ReLU1(x) + ReLU2(x) + ReLU3(x)
= max(0, x) + max(0, b - x) + max(0, x - 1)

This combined function will be zero for $x < 0$, then increase linearly until $x = 1$, then remain constant at 1. By adjusting the constant 'b' in ReLU2, we can control the y-intercept of the second segment and achieve a better approximation of the original function.

Corresponding Single Layer Neural Network:

The corresponding single-layer neural network would have:

- **Input Layer:** One neuron that takes the input value (x).
- **Hidden Layer:** Three neurons, each representing one of the ReLU functions (ReLU1, ReLU2, and ReLU3). Each hidden neuron would have weights and biases associated with it.
- **Output Layer:** One neuron that sums the outputs from the three hidden layer neurons.

Weights and Biases:

- The weights associated with the connections from the input layer to the hidden layer neurons determine the slopes of the linear segments in the ReLU functions.
- The biases act as thresholds or y-intercepts for the ReLU functions.

By adjusting the weights and biases in this network during training, we can attempt to optimize the approximation of the piecewise linear function.

In conclusion, by combining three ReLU functions as described above, we can create a reasonable approximation of the three-segment linear function in the image. The corresponding single-layer neural network can be used to learn the weights and biases to optimize the approximation further.