Table of Contents

Deep Learning numerical problems

Feed forward neural network - vanishing gradient problem

You are training a large feedforward neural network (100 layers) on a binary classification task, using a sigmoid activation in the final layer, and a mixture of tanh and ReLU activations for all other layers. You notice your weights to your a subset of your layers stop updating after the first epoch of training, even though your network has not yet converged. Deeper analysis reveals the gradients to these layers completely, or almost completely, go to zero very early on in training. How can you fix this?

Solution:

This is a Classic vanishing gradient problem. Increasing size of the training set doesn't help as the issue lies with the learning dynamics of the network. Varying the learning rate might help the network learn faster, but as the problem states the gradients to specific layers almost completely go to zero, so the issue seems to be localized to specific layers. Switching the ReLU activations with leaky ReLUs everywhere solves the problem of dying relus by passing some gradient signal back through all relu layers. (iii) Adding BatchNorm prior to every

activation ensures the tanh layers have inputs distributed closer to the linear region of the activation, so the elementwise derivative across the layer evaluates closer to 1.

Alternate Solution

When training a very deep feedforward neural network, such as one with 100 layers, encountering the issue of gradients going to zero (often referred to as the "**vanishing gradients" problem**) is a common challenge. This issue typically arises because the gradients diminish as they are backpropagated through the layers of the network. Here are several strategies to address and potentially fix this problem:

1. Use Appropriate Initialization

- **Xavier/Glorot Initialization**: For networks with tanh activations, use Xavier initialization, which helps in maintaining the variance of activations and gradients across layers.
- **He Initialization**: For ReLU activations, He initialization is often preferred as it helps in preserving the variance of activations through layers.

2. Batch Normalization

- Batch normalization normalizes the output of each layer, which helps to mitigate issues related to vanishing and exploding gradients by keeping the activations in a reasonable range. It also helps in speeding up the training process.

3. Use Residual Connections

- Adding residual (skip) connections between layers can help gradients flow through the network without diminishing too much. Residual connections allow the gradients to be passed directly through the network, making it easier for the network to learn.

4. Gradient Clipping

- Gradient clipping involves setting a threshold to clip the gradients during backpropagation. This can prevent gradients from becoming too small or too large, thereby helping with the stability of training.

5. Use Different Activations

- **ReLU and Variants**: ReLU activations (and its variants like Leaky ReLU, Parametric ReLU) are less prone to vanishing gradients compared to tanh. Consider using ReLU or its variants in deeper layers of your network.

6. Use Better Optimizers

- **Adam** or other adaptive optimizers like RMSprop or AdaGrad might help as they adapt the learning rates during training and can handle varying gradient magnitudes more effectively.

7. Gradient Checking

- Ensure that your implementation of the backpropagation is correct. Sometimes bugs in the gradient computation can lead to issues where gradients do not update as expected.

8. Reduce Network Depth or Complexity

- If possible, consider reducing the depth of the network or using a different network architecture. Sometimes, a simpler model can achieve better results with more stable training.

9. Regularization

- Use techniques like dropout to prevent overfitting, which can sometimes exacerbate issues with vanishing gradients.

By combining these strategies, you can often mitigate the vanishing gradients problem and ensure that your network's layers continue to learn effectively throughout the training process.

Activation function

How will you decide if a given function is an activation function?

Solution

An activation function in neural networks typically should be:
Non-linear: It introduces non-linearity into the model, allowing the network to learn more complex patterns.
Differentiable: It should be differentiable almost everywhere, enabling gradient-based optimization techniques.

(i) $f(x) = -\min(2, x)$

- **Description**: This function returns the negative of the minimum of 2 and xxx. Essentially, for x≤2x, the function is −x and for x>2, it is −2.
- **Non-linearity**: The function is piecewise linear but introduces non-linearity due to the min function, although it is not smooth everywhere.
- **Differentiability**: The function is not differentiable at x=2 because there is a sharp corner where the function changes from −x to −2.

Conclusion: It is a non-linear function but not a smooth activation function due to non-differentiability at x=2. It might not be ideal for standard neural network use.

(ii) $f(x) = 0.9x + 1$

- **Description**: This is a linear function with a slope of 0.9 and a y-intercept of 1.
- **Non-linearity**: This function is linear, so it does not introduce any non-linearity into the model.
- **Differentiability**: It is differentiable everywhere, but since it is linear, it won't allow the network to learn complex patterns.

Conclusion: This function is not a valid activation function because it is linear, which limits the network's ability to model complex relationships.

Feedforward neural network Problem 2

To find the network output, we need to calculate the activations through each layer of the network. Here, we have a feedforward neural network with one hidden layer. Let's go through the steps to calculate the network's output for the given input.

Network Structure

- **Input layer**: 2 neurons
- **Hidden layer**: 3 neurons
- **Output layer**: 1 neuron

Given Parameters

- **Weights for hidden layer**:

$$W^1 = \begin{matrix} 0.4 & 0.6 & 0.2 \\ 0.3 & 0.9 & 0.5 \end{matrix}$$

- **Bias for hidden layer**: $b_1 = \begin{matrix}1\\1\\1\end{matrix}$
- **Weights for output layer**: $W^2 = \begin{matrix}0.2\\0.2\\0.8\end{matrix}$
- **Bias for output layer**: $b^2 = 0.5$
- **Input**: $\begin{bmatrix}1\\1\end{bmatrix}$
- Step 1: Compute Hidden Layer Activations

1. **Compute the weighted sum for hidden layer neurons**:

   $z^{(1)}=W^{(1)}\cdot x+b^{(1)}$

   First, calculate the matrix-vector product $W^1\cdot x$:

   $W1.x = \begin{bmatrix}0.4 & 0.6 & 0.2\\0.3 & 0.9 & 0.5\end{bmatrix} x \begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}0.4x.1 + 0.6x1\\0.3x1 + 0.9x1\end{bmatrix} = \begin{bmatrix}1.0\\1.2\end{bmatrix}$

   Add the bias vector b^1

$$z^{(1)} = \begin{bmatrix}1.0\\1.2\end{bmatrix} + \begin{bmatrix}1\\1\\1\end{bmatrix} = \begin{bmatrix}2.0\\2.2\\2.0\end{bmatrix}$$

2. **Apply the activation function (assuming we use ReLU here)**:

For ReLU activation function $\mathrm{ReLU}(x) = \max(0, x)$:

$$a^{(1)} = \mathrm{ReLU}(z^{(1)}) = \begin{bmatrix}\mathrm{ReLU}(2.0)\\\mathrm{ReLU}(2.2)\\\mathrm{ReLU}(2.0)\end{bmatrix} = \begin{bmatrix}2.0\\2.2\\2.0\end{bmatrix}$$

Step 2: Compute Output Layer Activation

1. **Compute the weighted sum for output layer neuron**:

1. **Compute the weighted sum for output layer neuron:**

$$z^{(2)} = W^{(2)} \cdot a^{(1)} + b^{(2)}$$

First, calculate the dot product $W^{(2)} \cdot a^{(1)}$:

$$W^{(2)} \cdot a^{(1)} = \begin{bmatrix} 0.2 \\ 0.2 \\ 0.8 \end{bmatrix} \cdot \begin{bmatrix} 2.0 \\ 2.2 \\ 2.0 \end{bmatrix} = (0.2 \cdot 2.0) + (0.2 \cdot 2.2) + (0.8 \cdot 2.0) =$$

Add the bias $b^{(2)}$:

$$z^{(2)} = 2.44 + 0.5 = 2.94$$

2. **Apply the activation function (using sigmoid here)**:

For sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$:

$$a^{(2)} = \sigma(z^{(2)}) = \frac{1}{1 + e^{-2.94}}$$

Calculate $e^{-2.94}$:

$$e^{-2.94} \approx 0.0530$$

Thus,

$$a^{(2)} = \frac{1}{1 + 0.0530} = \frac{1}{1.0530} \approx 0.9496$$

Final Output

The final output of the network for the input $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ is approximately 0.9496

CNN Algorithm selection

You are training a convolutional neural network on the ImageNet dataset, and you are thinking of using gradient descent as your optimization function. Which is the fastest algorithm and why ? Compare with others and give me a solution

Solution

(i) It is possible for Stochastic Gradient Descent to converge faster than Batch Gradient Descent

True. Stochastic Gradient Descent (SGD) can converge faster than Batch Gradient Descent in practice because it updates the model parameters more frequently. Batch Gradient Descent uses the entire dataset to compute the gradient before making an update, which can be very slow for large datasets. SGD, on the other hand, updates the model parameters after each individual training example or mini-batch, which allows for more frequent updates and can help the optimization process escape local minima and potentially converge faster. However, it may exhibit higher variance in updates.

(ii) It is possible for Mini Batch Gradient Descent to converge faster than Stochastic Gradient Descent

False. Mini-Batch Gradient Descent is a compromise between Batch Gradient Descent and Stochastic Gradient Descent. It typically converges faster than Batch Gradient Descent because it uses smaller subsets (mini-batches) of the data, which allows for more frequent updates. However, it generally converges slower than pure Stochastic Gradient Descent because the updates in Mini-Batch Gradient Descent are based on averages of mini-batches rather than individual examples, which reduces the noisiness of updates but can also slow down the convergence compared to the frequent updates of SGD.

(iii) It is possible for Mini Batch Gradient Descent to converge faster than Batch Gradient Descent

True. Mini-Batch Gradient Descent can converge faster than Batch Gradient Descent because it strikes a balance between the computational efficiency and the frequency of parameter updates. By using mini-batches, it can provide more frequent updates than Batch Gradient Descent while being more computationally feasible than Stochastic Gradient Descent, especially on large datasets. This often leads to faster convergence in practice.

(iv) It is possible for Batch Gradient Descent to converge faster than Stochastic Gradient Descent

False. Batch Gradient Descent typically converges slower compared to Stochastic Gradient Descent due to the computational cost of using the entire dataset to compute gradients before updating the model. While Batch Gradient Descent provides more accurate gradient estimates, it is less frequent in updates, which can lead to slower convergence, especially on large datasets like ImageNet. Stochastic Gradient Descent updates more frequently, which often helps in faster convergence despite the noisiness of the updates.

Summary

The true statements are:

- (i) It is possible for Stochastic Gradient Descent to converge faster than Batch Gradient Descent.
- (iii) It is possible for Mini Batch Gradient Descent to converge faster than Batch Gradient Descent.

Mini-Batch Gradient Descent generally provides a good compromise between the two extremes and is widely used in practice due to its balance of convergence speed and computational efficiency.

Dropout

Which of the following is true about dropout?
(i) Dropout leads to sparsity in the trained weights
(ii) At test time, dropout is applied with inverted keep probability
(iii) The larger the keep probability of a layer, the stronger the regularization of the weights in that layer
(iv) None of the above

Solution

(i) Dropout leads to sparsity in the trained weights

False. Dropout is a regularization technique that helps prevent overfitting by randomly setting a fraction of neurons to zero during training. However, dropout does not directly induce sparsity in the trained weights themselves; rather, it helps prevent the network from becoming overly reliant on specific neurons, which can lead to better generalization. The weights of the network are still present and non-zero; dropout affects only the activations during training.

(ii) At test time, dropout is applied with inverted keep probability

False. At test time, dropout is not applied at all. During training, dropout randomly sets a fraction of the activations to zero (based on the keep probability) to prevent overfitting. However, during inference or test time, dropout is turned off, and the full network is used with all neurons active. Instead, the activations are scaled by the keep probability to account for the dropout that was applied during training. This scaling ensures that the expected value of the activations remains consistent with what was learned during training.

(iii) The larger the keep probability of a layer, the stronger the regularization of the weights in that layer

False. The keep probability (the probability of retaining a neuron) and regularization are inversely related. A larger keep probability means that fewer neurons are dropped out during training, which results in weaker regularization. Conversely, a smaller keep probability means more neurons are dropped out, which provides stronger regularization. Dropout strengthens regularization by increasing the amount of noise during training and making the network more robust.

(iv) None of the above

True. Given that statements (i), (ii), and (iii) are false, the correct answer is (iv) None of the above.

Summary

- (i) Dropout does not lead to sparsity in trained weights.
- (ii) Dropout is not applied at test time; instead, activations are scaled by the keep probability.
- (iii) A larger keep probability results in weaker regularization.

Thus, the correct statement is (iv) None of the above.

You have been tasked by the Supreme Leader of the Republic of Wadiya to build a deep learning model to help him decide what Wadiyan movie to watch. Specifically you're asked to build a classifier that takes in an image of a movie poster and classifies it into one of four genres: comedy, horror, action, and romance. You have been provided with a large dataset of movie posters where each movie poster corresponds to a move with exactly one of these genres. (i) (2 points) You want to get a sense of what the human level performance is for the task. How would you go about getting an estimate of human level performance?

Solution

To estimate human-level performance for the task of classifying movie posters into genres, you can follow these steps:

1. Human Annotation or Evaluation

Objective: Obtain a benchmark for the best possible performance by humans.

1. **Select a Sample Set**:
    - Randomly select a subset of movie posters from your dataset. This subset should be representative of the entire dataset to ensure that the performance estimate is accurate.
2. **Prepare the Annotation Task**:
    - Create a simple annotation interface where human annotators can label the genre of each movie poster. This could be a web-based tool or a straightforward labeling sheet.
3. **Recruit Annotators**:
    - Use multiple human annotators to label the posters. Ideally, recruit people who are familiar with the genres or have an understanding of movie genres.
4. **Annotate the Data**:
    - Ask the annotators to classify each movie poster into one of the four genres: comedy, horror, action, or romance.
5. **Aggregate Annotations**:
    - Collect the annotations from all annotators. If you use multiple annotators per poster, you can use a majority voting system to determine the final label for each poster.
6. **Evaluate Human Performance**:
    - Compare the aggregated human annotations with a pre-defined set of ground truth labels for the subset. Calculate metrics such as accuracy, precision, recall, and F1-score to gauge the human-level performance.

2. Calculate Metrics

Objective: Quantify how well humans perform in terms of classification accuracy.

1. **Accuracy**:

- Measure the proportion of correctly classified movie posters by humans compared to the ground truth. This provides a direct estimate of how often human annotators correctly identify the genre.

$$\text{Accuracy}=\frac{Number\ of\ Correct\ Predictions}{Total\ Number\ of\ predictions}$$

2. **Confusion Matrix**:
   - Generate a confusion matrix to understand where human annotators might make errors, such as confusing one genre with another. This matrix shows the number of correct and incorrect classifications for each genre.
3. **Other Metrics**:
   - Optionally, calculate precision, recall, and F1-score for each genre to get a more detailed view of performance. This can be particularly useful if the dataset is imbalanced.
   - **Precision**: $\frac{True\ Positives}{True\ Positives+False\ Positives}$
   - **Recall**: $\frac{True\ Positives}{True\ Positives+False\ Negatives}$
   - **F1-score**:2. $\frac{Precision \cdot Recall}{Precision+Recall}$

Summary

To get an estimate of human-level performance:

1. Annotate a representative sample of the movie posters with the help of multiple human annotators.
2. Use these annotations to calculate performance metrics such as accuracy, precision, recall, and F1-score.

This approach provides a benchmark for what the best possible performance could be for human annotators, which you can then compare against the performance of your deep learning model.

MLP Perceptron Problem

You have a simple MLP with the following architecture:

- **Input Layer**: 2 neurons (features)
- **Hidden Layer**: 2 neurons with ReLU activation
- **Output Layer**: 1 neuron with a sigmoid activation (binary classification)

You are given the following:

- **Input vector** $\mathbf{x} = [0.5, 0.8]$
- **Weights and biases** for the hidden layer:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.4 & 0.6 \\ 0.3 & 0.9 \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

- **Weights and biases** for the output layer:

$$\mathbf{W}^{(2)} = \begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix}, \quad \mathbf{b}^{(2)} = 0.5$$

Calculate the output of the MLP for the given input vector.

Solution

Step 1: Compute the hidden layer activations

1. **Calculate the weighted sum for the hidden layer:**

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)} \cdot \mathbf{x} + \mathbf{b}^{(1)}$$

Perform the matrix multiplication and addition:

$$\mathbf{z}^{(1)} = \begin{bmatrix} 0.4 & 0.6 \\ 0.3 & 0.9 \end{bmatrix} \cdot \begin{bmatrix} 0.5 \\ 0.8 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

$$\mathbf{z}^{(1)} = \begin{bmatrix} (0.4 \cdot 0.5 + 0.6 \cdot 0.8) \\ (0.3 \cdot 0.5 + 0.9 \cdot 0.8) \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

↓

$$\mathbf{z}^{(1)} = \begin{bmatrix} (0.2 + 0.48) \\ (0.15 + 0.72) \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

$$\mathbf{z}^{(1)} = \begin{bmatrix} 0.68 \\ 0.87 \end{bmatrix}$$

2. **Apply the ReLU activation function:**

$$\mathbf{a}^{(1)} = \text{ReLU}(\mathbf{z}^{(1)})$$

ReLU is defined as $\text{ReLU}(x) = \max(0, x)$, so:

$$\mathbf{a}^{(1)} = \begin{bmatrix} \max(0, 0.68) \\ \max(0, 0.87) \end{bmatrix} = \begin{bmatrix} 0.68 \\ 0.87 \end{bmatrix}$$

Step 2: Compute the output layer activation

1. **Calculate the weighted sum for the output layer:**

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)} \cdot \mathbf{a}^{(1)} + \mathbf{b}^{(2)}$$

$$\mathbf{z}^{(2)} = \begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix}^T \cdot \begin{bmatrix} 0.68 \\ 0.87 \end{bmatrix} + 0.5$$

$$\mathbf{z}^{(2)} = (0.2 \cdot 0.68 + 0.4 \cdot 0.87) + 0.5$$

$$\mathbf{z}^{(2)} = (0.136 + 0.348) + 0.5$$

$$\mathbf{z}^{(2)} = 0.984 + 0.5$$

$$\mathbf{z}^{(2)} = 1.484$$

2. **Apply the sigmoid activation function:**

$$\text{Output} = \sigma(\mathbf{z}^{(2)}) = \frac{1}{1 + e^{-\mathbf{z}^{(2)}}}$$

$$\text{Output} = \frac{1}{1 + e^{-1.484}}$$

$$e^{-1.484} \approx 0.227$$

$$\text{Output} = \frac{1}{1 + 0.227} = \frac{1}{1.227} \approx 0.814$$

Summary

For the given input vector $[0.5, 0.8]$, the output of the MLP is approximately **0.814**. This value represents the probability assigned to one of the binary classes, depending on your specific classification task.

Perceptron Usecase – real life scenario

Let’s consider a real-life problem where a perceptron can be used to classify emails into two categories: "Spam" or "Not Spam." We'll use a simple example with two features:

- **Feature 1**: Frequency of certain keywords (e.g., "free")
- **Feature 2**: Frequency of other keywords (e.g., "win")

The goal is to classify an email as "Spam" or "Not Spam" based on these features.

Problem Statement

Suppose we have a dataset of emails with two features:

- **Feature 1**: Number of times the keyword "free" appears in the email.
- **Feature 2**: Number of times the keyword "win" appears in the email.

We will train a perceptron to classify emails into "Spam" (label = 1) or "Not Spam" (label = 0) based on these features.

Dataset

Let’s use the following training data:

Email	Feature 1 (free)	Feature 2 (win)	Label (Spam=1, Not Spam=0)
1	2	1	1
2	1	0	0
3	0	2	0

Email	Feature 1 (free)	Feature 2 (win)	Label (Spam=1, Not Spam=0)
4	3	2	1

Perceptron Model

A perceptron model has the following form:

y=step(w1·x1+w2·x2+b)y = \text{step}(w_1 \cdot x_1 + w_2 \cdot x_2 + b)y=step(w1·x1 +w2·x2+b)

Where:

- x1 and x2 are the features (Feature 1 and Feature 2).
- w1 and w2 are the weights.
- b is the bias.
- step is the step function that outputs 1 if the input is greater than or equal to 0, otherwise 0.

Training the Perceptron

Step 1: Initialize the weights and bias

Assume initial weights w1=0 w2=0, and bias b= 0

Step 2: Update rules

The perceptron learning rule is:

$$w_i = w_i + \eta \cdot (y_{true} - y_{pred}) \cdot x_i$$

$$b = b + \eta \cdot (y_{true} - y_{pred})$$

where η is the learning rate. Let's use η=1.

Step 3: Training Iterations

Let's train for one epoch with the given dataset.

1. **Email 1**: x1=2 , x2=1, ytrue=1

- Update weights and bias:

$$w_1 = 0 + 1 \cdot (1 - 0) \cdot 2 = 2$$

$$w_2 = 0 + 1 \cdot (1 - 0) \cdot 1 = 1$$

$$b = 0 + 1 \cdot (1 - 0) = 1$$

2. **Email 2**: $x_1 = 1$, $x_2 = 0$, $y_{\text{true}} = 0$
   - Compute output:

$$y_{\text{pred}} = \text{step}(2 \cdot 1 + 1 \cdot 0 + 1) = \text{step}(3) = 1$$

   - Update weights and bias:

$$w_1 = 2 - 1 \cdot (1 - 0) \cdot 1 = 1$$

$$w_2 = 1 - 1 \cdot (1 - 0) \cdot 0 = 1$$

$$b = 1 - 1 \cdot (1 - 0) = 0$$

3. **Email 3**: $x_1 = 0$, $x_2 = 2$, $y_{\text{true}} = 0$
   - Compute output:

$$y_{\text{pred}} = \text{step}(1 \cdot 0 + 1 \cdot 2 + 0) = \text{step}(2) = 1$$

   - Update weights and bias:

$$w_1 = 1 - 1 \cdot (1 - 0) \cdot 0 = 1$$

$$w_2 = 1 - 1 \cdot (1 - 0) \cdot 2 = -1$$

$$b = 0 - 1 \cdot (1 - 0) = -1$$

4. **Email 4**: $x_1 = 3$, $x_2 = 2$, $y_{\text{true}} = 1$
   - Compute output:
     $y_{\text{pred}} = \text{step}(1 \cdot 3 + (-1) \cdot 2 - 1) = \text{step}(3 - 2 - 1) = \text{step}(0) =$

- Update weights and bias:

$$w_1 = 1 + 1 \cdot (1 - 0) \cdot 3 = 4$$

$$w_2 = -1 + 1 \cdot (1 - 0) \cdot 2 = 1$$

$$b = -1 + 1 \cdot (1 - 0) = 0$$

Final Weights and Bias after one epoch of training:

- $w_1 = 4$
- $w_2 = 1$
- $b = 0$

Classification

To classify a new email with $x_1 = 2$ and $x_2 = 1$:

$$y_{\text{pred}} = \text{step}(4 \cdot 2 + 1 \cdot 1 + 0) = \text{step}(8 + 1) = \text{step}(9) = 1$$

The perceptron classifies this new email as "Spam".

Summary

This simple perceptron model classifies emails into "Spam" or "Not Spam" based on the frequency of keywords. The calculations involved adjusting the weights and bias using the perceptron learning rule, which allowed the model to learn and make predictions based on the provided training data.

Delta rule

The Delta Rule (or Widrow-Hoff rule) is a method used for training a single-layer perceptron. It adjusts the weights to minimize the error between the predicted output and the actual target output. Here's a step-by-step guide to training a single perceptron using the Delta Rule, along with a numerical example.

Delta Rule Overview

The Delta Rule is used to update the weights w_i and bias b of a perceptron according to the following equations:

1. **Update Rule for Weights**:

$$w_i = w_i + \eta \cdot (y_{\text{true}} - y_{\text{pred}}) \cdot x_i$$

2. **Update Rule for Bias**:

$$b = b + \eta \cdot (y_{\text{true}} - y_{\text{pred}})$$

where:

- η is the learning rate (a small positive number, e.g., 0.1).
- y_{true} is the actual target output.
- y_{pred} is the predicted output.
- x_i are the input features.

Example Problem

Let's train a single perceptron to solve a binary classification problem.

Problem Statement:

You have a simple dataset with two features and a binary target. You want to classify whether an input vector belongs to Class 1 (label = 1) or Class 0 (label = 0).

Dataset:

Feature 1 (x_1)	Feature 2 (x_2)	Target (label)
2	1	1
1	0	0
0	2	0
3	2	1

Initial Weights and Bias:

Let's start with:

- $w_1 = 0$
- $w_2 = 0$
- $b = 0$
- Learning rate $\eta = 0.1$

Training Procedure:

1. **Compute the output** for each training example using the step function:

$$y_{\text{pred}} = \text{step}(w_1 \cdot x_1 + w_2 \cdot x_2 + b)$$

   where $\text{step}(z) = 1$ if $z \geq 0$ else 0.

2. **Update weights and bias** according to the Delta Rule if the predicted output does not match the target.

Step-by-Step Solution:

Iteration 1

Training Example 1:

- $x_1 = 2, x_2 = 1, y_{\text{true}} = 1$

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + b = 0 \cdot 2 + 0 \cdot 1 + 0 = 0$$

$$y_{\text{pred}} = \text{step}(0) = 1$$

$$\text{No update needed as } y_{\text{pred}} = y_{\text{true}}.$$

Training Example 2:

- $x_1 = 1, x_2 = 0, y_{\text{true}} = 0$

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + b = 0 \cdot 1 + 0 \cdot 0 + 0 = 0$$

$$y_{\text{pred}} = \text{step}(0) = 1$$

$$\text{Error} = y_{\text{true}} - y_{\text{pred}} = 0 - 1 = -1$$

$$w_1 = w_1 + \eta \cdot \text{Erro}\downarrow x_1 = 0 + 0.1 \cdot (-1) \cdot 1 = -0.1$$

$$w_2 = w_2 + \eta \cdot \text{Error} \cdot x_2 = 0 + 0.1 \cdot (-1) \cdot 0 = 0$$

$$b = b + \eta \cdot \text{Error} = 0 + 0.1 \cdot (-1) = -0.1$$

Updated Parameters:

- $w_1 = -0.1$
- $w_2 = 0$
- $b = -0.1$

Iteration 2

Training Example 3:

- $x_1 = 0, x_2 = 2, y_{\text{true}} = 0$

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + b = -0.1 \cdot 0 + 0 \cdot 2 - 0.1 = -0.1$$

$$y_{\text{pred}} = \text{step}(-0.1) = 0$$

$$\text{No update needed as } y_{\text{pred}} = y_{\text{true}}.$$

Training Example 4:

- $x_1 = 3, x_2 = 2, y_{\text{true}} = 1$

  $z = w_1 \cdot x_1 + w_2 \cdot x_2 + b = -0.1 \cdot 3 + 0 \cdot 2 - 0.1 = -0.3 - 0.1 = -($

$$y_{\text{pred}} = \text{step}(-0.4) = 0$$

$$\text{Error} = y_{\text{true}} - y_{\text{pred}} = 1 - 0 = 1$$

$$w_1 = w_1 + \eta \cdot \text{Error} \cdot x_1 = -0.1 + 0.1 \cdot 1 \cdot 3 = -0.1 + 0.3 = 0.2$$

$$w_2 = w_2 + \eta \cdot \text{Error} \cdot x_2 = 0 + 0.1 \cdot 1 \cdot 2 = 0 + 0.2 = 0.2$$

$$b = b + \eta \cdot \text{Error} = -0.1 + 0.1 \cdot 1 = -0.1 + 0.1 = 0$$

Updated Parameters:

- $w_1 = 0.2$
- $w_2 = 0.2$
- $b = 0$

Summary

After iterating through the dataset, the final weights and bias of the perceptron are:

- $w_1 = 0.2$
- $w_2 = 0.2$
- $b = 0$

The perceptron model has been trained using the Delta Rule to classify emails based on the given features. By applying the Delta Rule, the weights and bias are adjusted to minimize the prediction error for the training examples.

Cross entropy Problem

You decide to use cross entropy loss to train your network. Recall that the cross-entropy loss for a single example is defined as follows: LCE(ˆy, y) = − Pny i=1 yi log(ˆyi). where ˆy = (ˆy1, yˆ2...yˆny) T represents the predicted probability distribution over the classes and y = (y1, y2...yny) T represents the ground truth vector, which is zero everywhere except for the correct class (eg y = (1, 0, 0, 0)T for comedy, and y = (0, 0, 1, 0)T for action). (iii) (2 points) Suppose you're given an example poster of a horror movie. If the model correctly predicts the resulting probability distribution as yˆ= (0.1, 0.4, 0.3, 0.2), what is the value of the cross-entropy loss? You can give an answer in terms of logarithms.

Given Data

- **Ground Truth Vector** y: For a horror movie, let's assume the correct class is the second class. Thus, the ground truth vector y is:

$$y = (0, 1, 0, 0)^T$$

- **Predicted Probability Distribution** $\hat{y}$: The predicted probability distribution is:

$$\hat{y} = (0.1, 0.4, 0.3, 0.2)$$

Cross-Entropy Loss Formula

The cross-entropy loss for a single example is given by:

$$L_{\text{CE}}(\hat{y}, y) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

Here, y_i is the ground truth label for class i, and $\hat{y}_i$ is the predicted probability for class i.

Calculation

For the given example:

- The ground truth vector y indicates that the correct class is the second class (index 2).
- Therefore, only the term corresponding to y_2 (which is 1) will contribute to the loss calculation.

The cross-entropy loss simplifies to:

$$L_{\text{CE}}(\hat{y}, y) = -\left(y_2 \log(\hat{y}_2)\right)$$

Substituting the values from the vectors:

- $y_2 = 1$
- $\hat{y}_2 = 0.4$

So,

$$L_{\text{CE}}(\hat{y}, y) = -(1 \cdot \log(0.4)) = -\log(0.4)$$

Answer

The value of the cross-entropy loss for the given example is:

$$L_{\text{CE}}(\hat{y}, y) = -\log(0.4)$$

This formula shows the loss in terms of logarithms, which you can use directly or evaluate using a calculator or logarithm table if needed.

Problem cross entropy 2

After some training, the model now incorrectly predicts romance with distribution ⟨0, 0.4, 0, 0.6) for the same poster. What is the new value of the crossentropy loss for this example?

Given Data

- **Ground Truth Vector** y: For a horror movie, the ground truth vector is:

$$y = (0, 1, 0, 0)^T$$

- **New Predicted Probability Distribution** $\hat{y}$: The predicted probability distribution is:

$$\hat{y} = (0, 0.4, 0, 0.6)$$

Cross-Entropy Loss Formula

The cross-entropy loss for a single example is given by:

$$L_{\text{CE}}(\hat{y}, y) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

Calculation

For the new example:

- The ground truth vector y indicates that the correct class is the second class (index 2).
- Therefore, only the term corresponding to y_2 (which is 1) will contribute to the loss calculation.

The cross-entropy loss simplifies to:

$$L_{\text{CE}}(\hat{y}, y) = -\left(y_2 \log(\hat{y}_2)\right)$$

Substituting the values from the vectors:

- $y_2 = 1$
- $\hat{y}_2 = 0.4$

So,

$$L_{\text{CE}}(\hat{y}, y) = -\left(1 \cdot \log(0.4)\right) = -\log(0.4)$$

Answer

The value of the cross-entropy loss for the new prediction is:

$$L_{\text{CE}}(\hat{y}, y) = -\log(0.4)$$

Note: Even though the model has predicted the wrong class (romance with distribution $\hat{y} = (0, 0.4, 0, 0.6)$), the cross-entropy loss calculation remains the same as long as we are using the correct ground truth class.

L1 and L2 regularization

While the model does well on the training set, it only achieves an accuracy of 85% on the dev set. You conclude that the model is overfitting, and plan to use L1 or L2 regularization to fix the issue. However, before you can do so, you learn from your lab mate that some of the examples in the data may be incorrectly labelled. Which form of regularisation would you prefer to use and why?

Given the situation where some examples in the data may be incorrectly labeled, choosing the appropriate form of regularization is crucial. Here's a detailed comparison between **L1** and **L2 regularization** and recommendations based on the issue of potentially incorrect labels:

L1 vs. L2 Regularization

1. **L1 Regularization (Lasso Regularization)**:
   - **Form**: Adds a penalty equal to the absolute value of the magnitude of coefficients to the loss function.

$$L_1 = \lambda \sum_i |w_i|$$

   - **Effect**: Promotes sparsity in the model parameters, effectively driving some weights to exactly zero. This can lead to simpler models with fewer features being used.
   - **Advantages**: Useful when you suspect that many features are irrelevant or when you want a feature selection effect.
2. **L2 Regularization (Ridge Regularization)**:
   - **Form**: Adds a penalty equal to the square of the magnitude of coefficients to the loss function.

$$L_2 = \lambda \sum_i w_i^2$$

   - **Effect**: Encourages small weights rather than zero weights, leading to a model where all features contribute, albeit with smaller weights.
   - **Advantages**: Generally leads to better performance in scenarios where all features are relevant. It tends to handle noisy data and outliers better.

Addressing Incorrectly Labeled Data

Incorrectly labeled data can introduce noise into the training process, potentially affecting model performance. Here's how each type of regularization handles this:

- **L1 Regularization**:

    - May not be as effective in the presence of noisy labels because it tries to force many weights to zero, potentially removing useful features.
    - Less robust to noisy data as it can discard features that might be informative if their weights are close to zero.
- **L2 Regularization**:
    - More robust to noisy data compared to L1 regularization. It discourages large weights but doesn't necessarily set them to zero, making it less likely to discard useful features due to noise.
    - Helps in stabilizing the training process and can better handle outliers, which might be beneficial if there are incorrectly labeled examples in the data.

Recommendation

L2 Regularization is generally preferred in the presence of potentially incorrect labels for the following reasons:

1. **Robustness to Noise**: L2 regularization is more robust to noisy data and outliers, which can be advantageous when some examples are incorrectly labeled. It helps in smoothing the model weights without forcing any weights to zero, thus preserving the potential usefulness of features.
2. **Stability**: It provides a more stable training process by discouraging large weights, which can help in managing the impact of noisy or incorrect labels.
3. **Feature Utilization**: It avoids discarding potentially useful features entirely, which could happen with L1 regularization if the incorrect labels lead to weights being driven to zero.

Summary

L2 Regularization is generally the better choice when dealing with noisy or incorrectly labeled data due to its robustness and ability to handle outliers. It helps in improving generalization and stabilizing the learning process, making it more suitable in scenarios where data quality might be compromised.

Activation function

Propose a different activation function for the last layer and a loss function that are better suited for this multi-class labeling task.

For a multi-class classification task, it's crucial to choose an appropriate activation function and loss function that align with the nature of the problem. Here's a proposed solution for the last layer activation function and the loss function that are well-suited for multi-class classification:

1. Activation Function for the Last Layer

Softmax Activation Function

Mathematical Form:

For a given input vector z with n dimensions, the softmax function converts the raw output scores into probabilities:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

where:

- z_i is the score for class i,
- $\hat{y}_i$ is the predicted probability for class i,
- n is the number of classes.

Properties:

- **Probability Distribution**: Softmax converts raw scores into a probability distribution where the sum of all predicted probabilities equals 1.
- **Non-linearity**: It introduces non-linearity, which is essential for classification tasks.
- **Differentiability**: The softmax function is differentiable, making it suitable for gradient-based optimization.

Explanation: The softmax function is commonly used in the output layer of neural networks for multi-class classification problems. It provides a clear probabilistic interpretation of the model's output, which is ideal for tasks where each example belongs to one of several possible classes.

2. Loss Function

Categorical Cross-Entropy Loss

Mathematical Form:

The categorical cross-entropy loss for a single example is given by:

$$L_{\text{CE}}(\hat{y}, y) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

where:

- $\hat{y}_i$ is the predicted probability for class i,
- y_i is the ground truth label for class i,
- n is the number of classes.

Explanation:

- **Suitability for Multi-Class Classification**: Categorical cross-entropy is specifically designed for multi-class classification tasks. It measures the difference between the true class distribution (one-hot encoded vector) and the predicted probability distribution.
- **Gradient-Based Optimization**: It is differentiable and suitable for gradient-based optimization methods, which are commonly used in training neural networks.

Summary

For a multi-class classification task, use:

- **Softmax Activation Function**: Converts raw output scores into a probability distribution over the classes, making the outputs interpretable as probabilities.
- **Categorical Cross-Entropy Loss**: Measures the discrepancy between the predicted probability distribution and the true class labels, guiding the model to minimize this differer ↓

These choices are well-established and effective for training neural networks on multi-class classification problems, providing a clear probabilistic framework and a suitable loss measure for optimizing the model.

Single perceptron

Given a single perceptron, can the Perceptron Learning Algorithm (PLA) find a separating boundary for the following data points: {(1, 0), (0, 1)} ∈ P and {(-1, 0) and (0, -1)} ∈ N. P denotes the positive class and N denotes the negative class. State True or False. If True, then write the equation of the separating boundary for each case.

Given Data Points

- **Positive Class (P)**: $(1, 0)$, $(0, 1)$
- **Negative Class (N)**: $(-1, 0)$, $(0, -1)$

Checking Separability

Let's visualize the points:

- **Positive Class (P)**:
  - $(1, 0)$
  - $(0, 1)$
- **Negative Class (N)**:
  - $(-1, 0)$
  - $(0, -1)$

These points are plotted as follows ↓

- The positive points $(1, 0)$ and $(0, 1)$ lie on the right and upper sides, respectively.
- The negative points $(-1, 0)$ and $(0, -1)$ lie on the left and lower sides, respectively.

To determine if a linear boundary exists, let's consider the line equation $w_1 x_1 + w_2 x_2 + b = 0$, where w_1, w_2, and b are the weights and bias of the perceptron.

Deriving the Linear Boundary

1. **Positive Points**: The points should satisfy $w_1 x_1 + w_2 x_2 + b > 0$.
2. **Negative Points**: The points should satisfy $w_1 x_1 + w_2 x_2 + b < 0$.

For the positive point $(1, 0)$:

$$w_1 \cdot 1 + w_2 \cdot 0 + b > 0 \implies w_1 + b > 0$$

For the positive point $(0, 1)$:

$$w_1 \cdot 0 + w_2 \cdot 1 + b > 0 \implies w_2 + b > 0$$

For the negative point $(-1, 0)$:

$$w_1 \cdot (-1) + w_2 \cdot 0 + b < 0 \implies -w_1 + b < 0 \implies b < w_1$$

For the negative point $(0, -1)$:

$$w_1 \cdot 0 + w_2 \cdot (-1) + b < 0 \implies -w_2 + b < 0 \implies b < w_2$$

Combining these conditions, we get:

1. $w_1 + b > 0$
2. $w_2 + b > 0$
3. $b < w_1$
4. $b < w_2$

We need to check if these conditions can be satisfied simultaneously.

Verifying Linear Separability

Consider a potential separating line. For example, if we use $w_1 = 1$, $w_2 = 1$, and $b = -0.5$, the equation of the line is:

$$x_1 + x_2 - 0.5 = 0$$

- For $(1, 0)$:

$$1 + 0 - 0.5 = 0.5 > 0 \text{ (correctly classified as positive)}$$

- For $(0, 1)$:

$$0 + 1 - 0.5 = 0.5 > 0 \text{ (correctly classified as positive)}$$

- For $(-1, 0)$:

$$-1 + 0 - 0.5 = -1.5 < 0 \text{ (correctly classified as negative)}$$

- For $(0, -1)$:

$$0 - 1 - 0.5 = -1.5 < 0 \text{ (correctly classified as negative)}$$

Since we found a linear boundary that correctly separates the points into their respective classes, the statement is:

True

Equation of the Separating Boundary:

A possible equation of the separating boundary is:

$$x_1 + x_2 - 0.5 = 0$$

This equation can separate the given positive and negative points correctly.

Perceptron Problem 3

A student implemented PLA using the following train data and computed the separating boundary as: x1 + x2 - 1 = 0.

x_1	x_2	t
1	2	1
2	3	1
4	4.5	1
0	0	0
0.5	0	0
0.5	1	0
-1	0	0

When he applied this model on the validation samples, he noticed that the accuracy was low. He debugged the code and found that the separating boundary is not computed accurately. Your task is to find:

Which point(s) are misclassified in the above samples?

Suggest appropriate corrections to the parameters of the separating boundary. Write the new equation

To determine which points are misclassified by the separating boundary $x_1 + x_2 - 1 = 0$, we need to evaluate each point using this boundary equation and check if it aligns with the expected class labels.

Classification by the Given Boundary

Separating Boundary Equation:

$$x_1 + x_2 - 1 = 0$$

Classification Rule:

- If $x_1 + x_2 - 1 > 0$, classify as positive (1).
- If $x_1 + x_2 - 1 < 0$, classify as negative (0).

Given Data Points and Expected Labels

x_1	x_2	t (True Class)
1	2	1
2	3	1
4	4.5	1
0	0	0
0.5	0	0
0.5	1	0
-1	0	0

Classification of Each Point

1. **Point (1, 2)**:

$$1 + 2 - 1 = 2 \text{ (positive, should be 1)}$$

   - Classified as positive (1), correct.

2. **Point (2, 3)**:

$$2 + 3 - 1 = 4 \text{ (positive, should be 1)}$$

   - Classified as positive (1), correct.

3. **Point (4, 4.5)**:

$$4 + 4.5 - 1 = 7.5 \text{ (positive, should be 1)}$$

   - Classified as positive (1), correct.

4. **Point (0, 0)**:

$$0 + 0 - 1 = -1 \text{ (negative, should be 0)}$$

   - Classified as negative (0), correct.

5. **Point (0.5, 0)**:

$$0.5 + 0 - 1 = -0.5 \text{ (negative, should be 0)}$$

   - Classified as negative (0), correct.

6. **Point (0.5, 1)**:

$$0.5 + 1 - 1 = 0.5 \text{ (positive, should be 0)}$$

- Classified as positive (1), incorrect (should be 0).

7. **Point (-1, 0)**:

$$-1 + 0 - 1 = -2 \text{ (negative, should be 0)}$$

- Classified as negative (0), correct.

Misclassified Points

- **Point (0.5, 1)**: Misclassified as positive (1), but it should be negative (0).

Suggesting Corrections

To correct the separating boundary, we need to adjust the boundary to ensure it separates the points accurately.

Let's propose a new boundary by manually adjusting the parameters. For simplicity, we can try a boundary that better fits the given data.

Finding a Suitable Boundary

To adjust the boundary, we can aim for a new boundary that correctly separates the positive and negative classes. For example, consider the boundary:

$$x_1 + x_2 - 0.5 = 0$$

Verification:

1. **Point (1, 2)**:

$$1 + 2 - 0.5 = 2.5 \text{ (positive, correct)}$$

2. **Point (2, 3)**:

$$2 + 3 - 0.5 = 4.5 \text{ (positive, correct)}$$

3. **Point (4, 4.5)**:

$$4 + 4.5 - 0.5 = 8 \text{ (positive, correct)}$$

4. **Point (0, 0)**:

$$0 + 0 - 0.5 = -0.5 \text{ (negative, correct)}$$

5. **Point (0.5, 0)**:

$$0.5 + 0 - 0.5 = 0 \text{ (negative, correct)}$$

6. **Point (0.5, 1)**:

$$0.5 + 1 - 0.5 = 1 \text{ (positive, correct)}$$

7. **Point (-1, 0)**:

$$-1 + 0 - 0.5 = -1.5 \text{ (negative, correct)}$$

The new boundary equation $x_1 + x_2 - 0.5 = 0$ correctly classifies all the points according to their true labels.

New Separating Boundary Equation

The corrected separating boundary equation is:

$$x_1 + x_2 - 0.5 = 0$$

Back Propagation

Show that for a binary classification problem, minimising the cross entropy loss is the same as minimising the KL divergence between the true and predicted distributions

To show that minimizing the cross-entropy loss is equivalent to minimizing the Kullback-Leibler (KL) divergence between the true and predicted probability distributions in a binary classification problem, we need to establish a mathematical relationship between these two concepts.

Definitions

1. **Cross-Entropy Loss (Binary Classification)**

   Given a binary classification problem with true label $y \in \{0, 1\}$ and predicted probability $\hat{y}$ (the probability that the true class is 1), the cross-entropy loss L_{CE} is defined as:

$$L_{\text{CE}} = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

2. **KL Divergence**

   The KL divergence D_{KL} between the true distribution p and the predicted distribution q is defined as: ↓

$$D_{\mathrm{KL}}(p\|q) = \sum_i p_i \log \frac{p_i}{q_i}$$

For binary classification, where the true distribution p is a Bernoulli distribution with probability $p_1 = y$ and $p_0 = 1 - y$, and the predicted distribution q is $\hat{y}$ and $1 - \hat{y}$, the KL divergence is:

$$D_{\mathrm{KL}}(p\|q) = p_1 \log \frac{p_1}{q_1} + p_0 \log \frac{p_0}{q_0}$$

Deriving the Equivalence

For binary classification, let's use:

- True probability distribution: $p = (y, 1 - y)$
- Predicted probability distribution: $q = (\hat{y}, 1 - \hat{y})$

The KL divergence between these distributions is:

$$D_{\mathrm{KL}}(p\|q) = y \log \frac{y}{\hat{y}} + (1 - y) \log \frac{1 - y}{1 - \hat{y}}$$

We need to show that this KL divergence is minimized when the cross-entropy loss is minimized.

Expanding KL Divergence:

$$D_{\mathrm{KL}}(p\|q) = y \log \frac{y}{\hat{y}} + (1 - y) \log \frac{1 - y}{1 - \hat{y}}$$

To simplify, let's split and re-arrange:

$$D_{\mathrm{KL}}(p\|q) = y \log y - y \log \hat{y} + (1 - y) \log(1 - y) - (1 - y) \log(1 - \hat{y})$$

When y is either 0 or 1, the terms involving $\log y$ or $\log(1-y)$ vanish because $\log 0$ is undefined (or considered 0 when multiplied by 0). The relevant parts are:

- For $y = 1$:

$$D_{\mathrm{KL}}(p\|q) = \log \frac{1}{\hat{y}} = -\log \hat{y}$$

- For $y = 0$:

$$D_{\mathrm{KL}}(p\|q) = \log \frac{1}{1-\hat{y}} = -\log(1-\hat{y})$$

Thus, combining these with the weights of y and $1-y$, the KL divergence simplifies to:

$$D_{\mathrm{KL}}(p\|q) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

Conclusion

We see that the expression for the KL divergence matches the negative of the cross-entropy loss function:

$$L_{\mathrm{CE}} = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

Therefore:

$$D_{\mathrm{KL}}(p\|q) = L_{\mathrm{CE}}$$

Summary

Minimizing the cross-entropy loss in binary classification is equivalent to minimizing the KL divergence between the true probability distribution and the predicted probability distribution. This is because both expressions represent the same quantity in different forms, with cross-entropy being the negative of the KL divergence.

Fully Connected CNN

Consider a fully connected network with 3 inputs x1, x2, x3. Suppose there are two hidden layers with 4 neurons having sigmoid activation functions. Further, the output layer is a softmax layer. Assume that all the weights in the network are set to 1 and all biases are set to 0. Write down the output of the network as a function of x = [x1, x2, x3].
Find y.

To find the output y of a fully connected network with the given structure and parameters, we need to follow these steps:

1. **Define the Network Structure:**
   - **Input Layer**: 3 inputs x1,x2, x3.
   - **Hidden Layers**: 2 hidden layers with 4 neurons each, using sigmoid activation functions.
   - **Output Layer**: Softmax layer for the final output.
2. **Network Parameters:**
   - All weights are set to 1.
   - All biases are set to 0.
3. **Calculate the Output Step-by-Step:**

Step 1: Input to First Hidden Layer

For each neuron in the first hidden layer, the output is computed as follows:

$$z_i^{(1)} = \sum_j W_{ij} x_j + b_i$$

where $W_{ij} = 1$ and $b_i = 0$. Therefore:

$$z_i^{(1)} = x_1 + x_2 + x_3$$

For 4 neurons in the first hidden layer, the output of each neuron is:

$$a_i^{(1)} = \sigma(z_i^{(1)}) = \sigma(x_1 + x_2 + x_3)$$

where $\sigma(z)$ is the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Step 2: Input to Second Hidden Layer

The output of the first hidden layer is fed into the second hidden layer. Each neuron in the second hidden layer computes:

$$z_i^{(2)} = \sum_j W_{ij}^{(2)} a_j^{(1)} + b_i^{(2)}$$

Since all weights $W_{ij}^{(2)} = 1$ and biases $b_i^{(2)} = 0$:

$$z_i^{(2)} = \sum_j a_j^{(1)} = 4 \cdot \sigma(x_1 + x_2 + x_3)$$

where $a_j^{(1)}$ is the output from each neuron in the first hidden layer. Thus:

$$z_i^{(2)} = 4 \cdot \sigma(x_1 + x_2 + x_3)$$

The output of each neuron in the second hidden layer is:

$$a_i^{(2)} = \sigma(z_i^{(2)}) = \sigma(4 \cdot \sigma(x_1 + x_2 + x_3))$$

Step 3: Output Layer (Softmax)

The output of the second hidden layer is used as input to the softmax layer. Each output neuron computes:

$$z_i^{(3)} = \sum_j W_{ij}^{(3)} a_j^{(2)} + b_i^{(3)}$$

Since all weights $W_{ij}^{(3)} = 1$ and biases $b_i^{(3)} = 0$:

$$z_i^{(3)} = \sum_j a_j^{(2)}$$

Since there are 4 neurons in the second hidden layer, the output of the softmax layer will be:

$$z^{(3)} = \sum_j a_j^{(2)} = 4 \cdot \sigma(4 \cdot \sigma(x_1 + x_2 + x_3))$$

Each softmax output y_i is:

$$y_i = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

where z_i is the same for all output neurons and $\sum_k e^{z_k} = 4 \cdot e^z$:

$$y_i = \frac{e^{4 \cdot \sigma(x_1+x_2+x_3)}}{4 \cdot e^{4 \cdot \sigma(x_1+x_2+x_3)}} = \frac{1}{4}$$

Final Output

The final output y of the network is a vector of equal probabilities for each of the 4 output neurons:

$$y = \left[\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right]$$

Sigmoid and tanh functions

Let φ1(.) and φ2(.) denote the sigmoid and the tanh functions respectively. Find φ1(-v)

$$\phi_1(v) = \sigma(v) = \frac{1}{1 + e^{-v}}$$

To find $\phi_1(-v)$, substitute $-v$ into the sigmoid function:

$$\phi_1(-v) = \sigma(-v) = \frac{1}{1 + e^{-(-v)}}$$

Simplify $e^{-(-v)}$:

$$e^{-(-v)} = e^{v}$$

So:

$$\phi_1(-v) = \frac{1}{1 + e^{v}}$$

This expression can also be related to the original sigmoid function $\sigma(v)$:

$$\phi_1(v) = \frac{1}{1 + e^{-v}}$$

Thus:

$$\phi_1(-v) = \frac{1}{1 + e^{v}} = 1 - \phi_1(v)$$

So:

$$\phi_1(-v) = 1 - \phi_1(v)$$

Feed forward Neural network problem 3

We are given astronomical data for star classification. Stars can be classified into seven main types (O, B, A, F, G, K, M) based on their surface temperatures. Additionally, there are sub-classes identified based on their sizes: supergiants, giants, main-sequence stars, and subdwarfs. Hence, a given training sample can be a supergiant star of O type. What changes should be done to the standard feed forward neural network to handle such cases where the classes are not mutually exclusive?

To handle cases where classes are not mutually exclusive—such as classifying stars into types and sub-classes using a feedforward neural network—you need to modify the standard network architecture and its output layer to account for multi-label classification. Here's how you can adapt the neural network:

1. Modify the Output Layer

Instead of using a single output node per class (which is suitable for mutually exclusive classes in a multi-class classification problem), you need to use a separate output node for each label in a multi-label classification problem. In this case, each star can belong to multiple classes (e.g., type and size).

- **Output Nodes**: The output layer should have nodes corresponding to each possible class label. For the given problem, you would have:
    - 7 output nodes for the star types (O, B, A, F, G, K, M).

- 4 output nodes for the size categories (supergiants, giants, main-sequence stars, and subdwarfs).

Thus, the output layer should have 7+4=11 nodes.

Activation Function

For each output node, use the **sigmoid activation function** instead of the softmax function. The sigmoid function is appropriate for multi-label classification because it can output probabilities for each class independently, which allows multiple classes to be activated simultaneously.

The sigmoid function for each output node I is defined as:

$$\sigma i(vi) = \frac{1}{1 + e - vi}$$

Loss Function

Use the **binary cross-entropy loss** function instead of categorical cross-entropy. Binary cross-entropy is suitable for multi-label classification because it evaluates each label independently.

For a given sample, if the true label vector is y and the predicted probability vector is y^, the binary cross-entropy loss is:

$$L_{BCE} = -\sum_{i}[y_i \log(y_i hat\ ) + (1 - y_i)\log\ (1 - y_i hat)]$$

where yi is the true label (0 or 1) for the i-th output node, and y^i is the predicted probability for the i-th output node.

4. Network Architecture

- **Input Layer**: Receive the features related to star properties.
- **Hidden Layers**: These can remain as they are in a standard feedforward neural network.
- **Output Layer**: Modify to have 11 output nodes (7 for star types, 4 for sizes) with sigmoid activations.
- **Loss Function**: Use binary cross-entropy loss.

Summary of Changes

1. **Output Layer**: Adjust to have separate nodes for each class label.
2. **Activation Function**: Use sigmoid activation for each output node.
3. **Loss Function**: Use binary cross-entropy loss for multi-label classification.

This approach allows the neural network to handle cases where multiple classes or labels can apply to a single instance, accommodating the scenario where a star could be of a certain type and size simultaneously.

Pseudo code for the perceptron learning algorithm

Initialize weights W to zeros (or small random values)
Initialize bias b to zero (or a small random value)
Set learning rate η (eta), a small positive number (e.g., 0.01)

Repeat until convergence (or for a fixed number of iterations):
  For each training sample (x, t) in the training set:
    Compute the predicted output y_hat:
      y_hat = sign(W * x + b)
      where W * x represents the dot product of weights and input vector x

    Compute the error:
      error = t - y_hat
      where t is the true label (1 or -1) and y_hat is the predicted label

    Update the weights and bias if there is an error:
      W = W + η * error * x
      b = b + η * error

  Optionally: Check for convergence by evaluating performance on a validation set

Output the final weights W and bias b

Cross entropy error

Consider the following class labels and predicted probability values of a classification model. Calculate the cross entropy error of the given model.

Class label	Apple	Banana	Mango
Apple	0.8	0.1	0.1
Banana	0.15	0.75	0.1
Apple	0.6	0.2	0.2
Mango	0.2	0.1	0.7

To calculate the cross-entropy error for the given classification model, you need to follow these steps:

1. **Identify the True Labels and Predicted Probabilities**:

   Each row corresponds to the predicted probabilities for a particular class, and the true label is indicated by the row's class.

2. **Apply the Cross-Entropy Formula**:

   The cross-entropy loss for a single example is given by:

$$L_{\text{CE}} = -\sum_{i} y_i \log(\hat{y}_i)$$

   where y_i is the true label (1 if the class is correct, 0 otherwise) and $\hat{y}_i$ is the predicted probability for that class.

Given Data

- **Example 1**: True class is `**apple**`, predicted probabilities are `**[0.8, 0.1, 0.1]**`
- **Example 2**: True class is `**banana**`, predicted probabilities are `**[0.15, 0.75, 0.1]**`
- **Example 3**: True class is `**apple**`, predicted probabilities are `**[0.6, 0.2, 0.2]**`
- **Example 4**: True class is `**mango**`, predicted probabilities are `**[0.2, 0.1, 0.7]**`

Cross-Entropy Calculation

1. **Example 1 (True class: apple)**:

   True label vector: `**[1, 0, 0]**`
   Predicted probabilities: `**[0.8, 0.1, 0.1]**`

   Cross-entropy loss:

$$L_{\text{CE1}} = -\left[1 \cdot \log(0.8) + 0 \cdot \log(0.1) + 0 \cdot \log(0.1)\right] = -\log(0.8)$$

2. **Example 2 (True class: banana)**:

   True label vector: `**[0, 1, 0]**`
   Predicted probabilities: `**[0.15, 0.75, 0.1]**`

   Cross-entropy loss:

$$L_{\text{CE2}} = -\left[0 \cdot \log(0.15) + 1 \cdot \log(0.75) + 0 \cdot \log(0.1)\right] = -\log(0.75)$$

3. **Example 3 (True class: apple)**:

   True label vector: `**[1, 0, 0]**`
   Predicted probabilities: `**[0.6, 0.2, 0.2]**`

   Cross-entropy loss:

$$L_{\text{CE3}} = -\left[1 \cdot \log(0.6) + 0 \cdot \log(0.2) + 0 \cdot \log(0.2)\right] = -\log(0.6)$$

4. **Example 4 (True class: mango)**:

   True label vector: `[0, 0, 1]`
   Predicted probabilities: `[0.2, 0.1, 0.7]`

   Cross-entropy loss:

$$L_{\text{CE4}} = -\left[0 \cdot \log(0.2) + 0 \cdot \log(0.1) + 1 \cdot \log(0.7)\right] = -\log(0.7)$$

Average Cross-Entropy Loss

To find the average cross-entropy error for the given model, sum up the losses and divide by the number of examples:

$$L_{\text{avg}} = \frac{L_{\text{CE1}} + L_{\text{CE2}} + L_{\text{CE3}} + L_{\text{CE4}}}{4}$$

Substitute the values:

$$L_{\text{avg}} = \frac{-\log(0.8) - \log(0.75) - \log(0.6) - \log(0.7)}{4}$$

Using a calculator:

- $\log(0.8) \approx -0.2231$
- $\log(0.75) \approx -0.2877$
- $\log(0.6) \approx -0.5108$
- $\log(0.7) \approx -0.3554$

So:

$$L_{\text{avg}} \approx \frac{-(-0.2231) - 0.2877 - 0.5108 - 0.3554}{4}$$

$$L_{\text{avg}} \approx \frac{0.2231 - 0.2877 - 0.5108 - 0.3554}{4}$$

$$L_{\text{avg}} \approx \frac{-0.9308}{4}$$

$$L_{\text{avg}} \approx -0.2327$$

Thus, the average cross-entropy error of the given model is approximately 0.2327.

Convolution Operation

Question: Given an input image of size 32×3232 \times 3232×32 with a single channel, apply a convolutional layer with the following parameters:

- Filter size: 5×5
- Number of filters: 6
- Stride: 1
- Padding: 2

Calculate the output dimensions of the convolutional layer.

Solution:
Calculate Output Height and Width:
The formula for the output size of a convolutional layer is:

$$Output\ size = \frac{(Input\ size - Filter\ size + 2\ x\ Padding)}{Stride} + 1$$

For height and width:

$$Output\ Height = \frac{(32 - 5 + 2x2)}{1} + 1 = \frac{32 - 5 + 4}{1} + 1 = \left(\frac{31}{1}\right) + 1$$

$$Output\ width = \frac{(32 - 5 + 2x2)}{1} + 1 = 32$$

The output dimensions are 32×32, and since there are 6 filters, the output depth is 6. Output Shape: 32×32×6

Pooling Operation

Question: Given an input feature map of size 28×28 with a pooling layer with the following parameters:

- Pooling size: 2×2
- Stride: 2

Calculate the output dimensions of the pooling layer.
Solution:

1. Calculate Output Height and Width:

For pooling, the formula is similar to convolution but with the pooling size and stride:
Output Size=$Input\ Size - \frac{Pool\ Size}{Stride}$+1
For height and width:
Output Height= $\frac{28-2}{2} + 1 = \left(\frac{26}{2}\right) + 1$=13+1=14
Output Width=$\frac{28-2}{2} + 1 = 14$
The output dimensions are 14×14
Output Shape: 14×14

Flattening

Question: After applying several convolutional and pooling layers, suppose the output feature map has dimensions 7×7×64. Calculate the size of the flattened vector.
Solution:

1. Flatten the Feature Map:

The flattened vector size is the product of the feature map dimensions:
Flattened Size=7×7×64=3136
Flattened Vector Size: 3136
4. Number of Parameters in a Convolutional Layer
Question: Calculate the number of parameters in a convolutional layer with the following specifications:

- Input channels: 3
- Number of filters: 10
- Filter size: 3×3

- Bias terms: 10

Solution:

- Calculate the Number of Parameters:
- Each filter has a size of 3×3×3 (including input channels):
- Parameters per Filter=3×3×3=27
- There are 10 filters, so:
- Total Parameters (Weights)=27×10=270
- Plus 10 bias terms:
- Total Parameters=270+10=280
- Total Parameters: 280

5. Output Dimensions of a Fully Connected Layer

Question: Given a fully connected layer with 1000 input neurons and 500 output neurons, calculate the number of parameters in this layer (excluding biases).

Solution:

1. Calculate the Number of Parameters: Each input neuron is connected to each output neuron:
    - Number of Parameters=1000×500=500,000
    - Number of Parameters (Weights): 500,000