Deeploy CV Project

Assignment : 4

Name: Vibha Narayan

Roll Number: 231141

ID : 224

Branch: PHY

Date of submission: 28/12/2024

Github Repository:

Answer 1

A large learning rate can cause the model to overshoot the optimal point, making training unstable or causing divergence whereas a small learning rate ensures gradual progress but can make training slow and may result in the model getting stuck in local minima or plateaus.
With low learning rate the model there is a possibility of underfitting of data points but with an appropriate learning rate the model is able to classify the orange and the blue dots separately in their respective regions whereas for the high learning rate case there is overfitting and the model misclassifies the datapoints.
An epoch in the context of training neural networks refers to one complete pass through the entire training dataset. During training, the model needs to learn patterns from the dataset. Each epoch allows the model to see all the data and adjust its weights accordingly. By performing multiple epochs, the model gradually improves its performance by refining its understanding of the patterns in the data. Therefore as the epoch increases while covering each dataset cycle the testing and training losses decreases and the model gets trained.

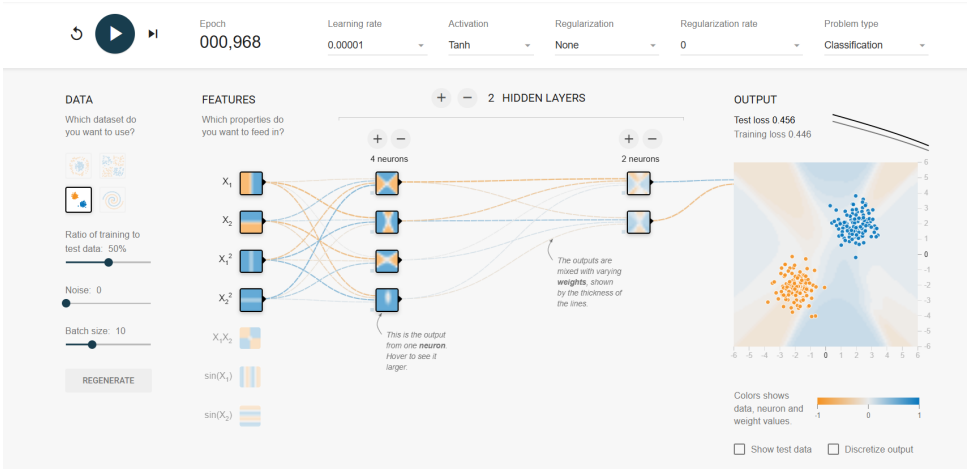

Figure 1: Output with low learning rate

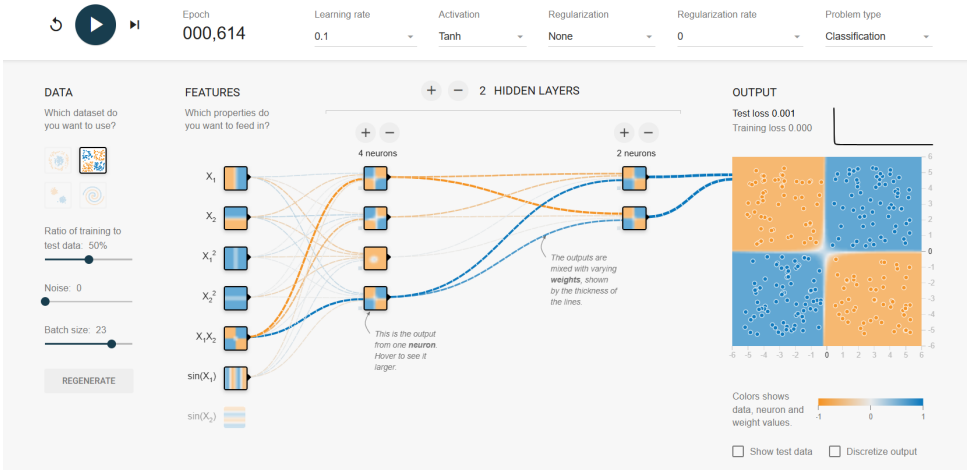

Figure 2: Output with appropriate learning rate

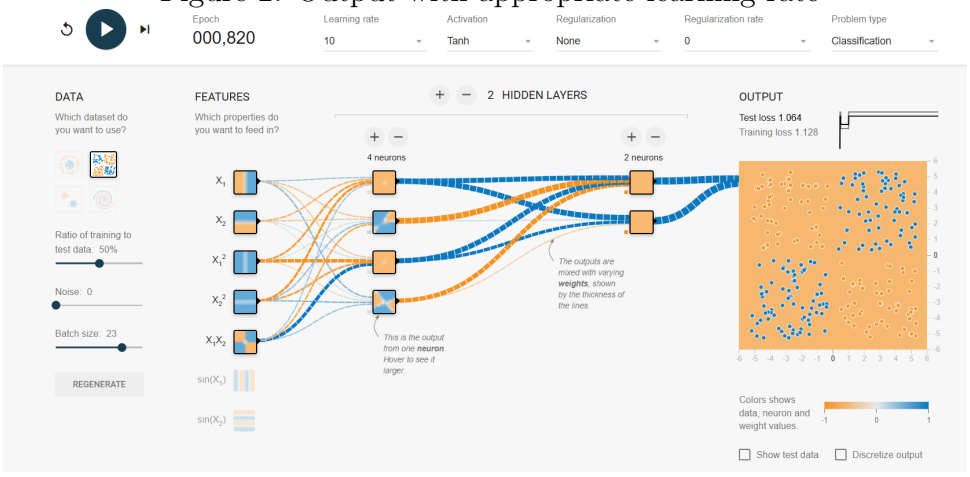

Figure 3: Output with high learning rate

Answer 2

a) We are training a model where input is a hand-written number from 0 to 9 and are expecting the output as that input number. So, basically we are trying to understand that how does Backpropagation helps neural networks learn. Here, our major objective is to minimise the cost function so that our model identifies the input correctly.
Cost Function is a layer of complexity beyond the hidden layers which takes weights and biases as inputs and many many training examples as parameters and produces one 1 number i.e the cost(how bad the weights and biases are) as the output.

$$C = \Sigma_{j=0}^{n_(L-1)}(a_j^L - y_j)^2$$

Cost function is the average over all the training data.
C(w1,w2,w3,.....,wn) == C(w) where w is a single input
Our purpose is to minimise C(w). Now, gradient of the cost function gives us the direction where C(w) decreases most rapidly. Backprogation helps in computing the gradient.

$-\nabla C(w_1, w_2, \ldots, w_n)$ tells us how sensitive the cost function is to each weight and bias.

Suppose the network is not trained well; then the activation values of the output neurons would be random. So, we apply nudges to reduce the activation values of the neurons that do not correspond to the input number. In order to produce the correct output neuron, we can do the following:

- Increase the **bias**.
- Increase (**wi**) in terms of (**ai**).
- Change (**ai**) in proportion to (**wi**).

size of the nudges should be proportional to how far away each current value is from the target value.

b) The first hidden layer was assumed that it picks up on the edges and the second hidden layer was assumed as which can pick upon patterns like loops and lines and the last one combines it to form digits but in actual we are assigning nudges in proportion to corresponding weights and how much each neuron should change while propagating backwards so that we can get close to the actual output by reducing random activation values.

c) The sigmoid activation function is primarily used. The sigmoid function has an easily interpretable output between 0 and 1, which makes it a popular choice for illustrating the concepts of activation, gradient, and backpropagation.

$$\sigma = 1/(1 + e^{-x})$$

$$a^L = \sigma(w^L a^{L-1} + b^L)$$

Answer 3

Local minima can cause gradient descent to get stuck in suboptimal points where the model parameters are not ideal. This can hinder the learning process, preventing the algorithm from reaching the global minimum where the true best solution lies. If the gradient descent algorithm finds a local minimum, it might get "stuck" there because the gradient is close to zero, meaning no further descent is possible. In this case, the algorithm will stop improving even though it hasn't found the global minimum.

Answer 4

a) In the context of neural networks, the learning rate is a hyperparameter that controls the size of the steps the model takes while adjusting its weights during training. It is used in optimization algorithms (such as gradient descent) to update the model's weights and minimize the loss function. The learning rate essentially determines how quickly or slowly the model learns.

b) A higher learning rate means larger updates to the weights during each iteration of training. This can lead to faster learning, but if it's too large, it can cause the algorithm to overshoot the optimal solution, potentially causing instability and poor convergence. A lower learning rate results in smaller updates, leading to slower learning but more controlled, precise convergence. However, if it's too small, it may take too long to converge or get stuck in suboptimal minima.

Learning rate seems intuitive because it implies the pace at which the model learns, but it could be misleading because a higher learning rate doesn't always equate to better or faster learning. Sometimes, it leads to instability or diverging results.

Answer 5

1) So, we have one input, K Hidden layers and one output as our model structure.
Cost function will be function of weights and biases of each layer.

$$C(w_1, w_2, ..., w_k, b_1, b_2, ..., b_k)$$

$$C = (a^L - y)^2$$

$$a^L = \sigma(w^L a^{L-1} + b^L)$$

$$z^L = w^L a^{L-1} + b^L$$

$$a^L = \sigma(z^L)$$

$$\sigma = 1/(1 + e^{-x})$$

C : cost function
w : weight
b : bias
y : output
where a raised to L represents the activation value of last neuron
the sigmoid function produces the output between 0 and 1

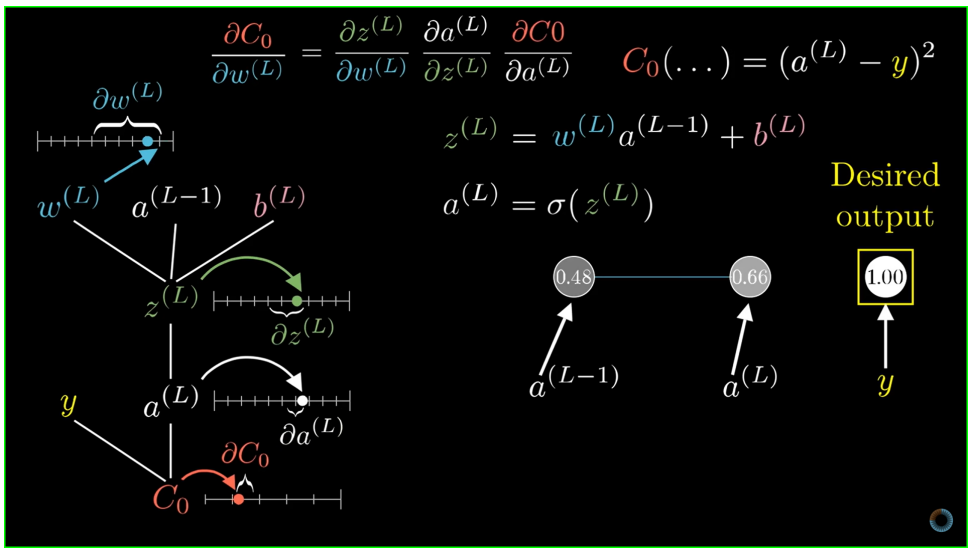

Figure 4: Initializing the Problem Statement of finding gradient of C

Now, since **z** depends on weights and bias so any tiny changes to **w** and **b** would lead to change **z** and since **a** depends on **z** so **a** also changes. **C** depends on **a** and **y** thus any change in a would lead to change C.

$$\delta C/\delta w^L = (\delta z^L/\delta w^L)(\delta a^L/\delta z^L)(\delta C/\delta a^L)$$

$$\delta C/\delta b^L = (\delta z^L/\delta b^L)(\delta a^L/\delta z^L)(\delta C/\delta a^L)$$

For the last derivative, the amount of the small nudge to the weight influenced the last layer depends on how strong the previous neuron is

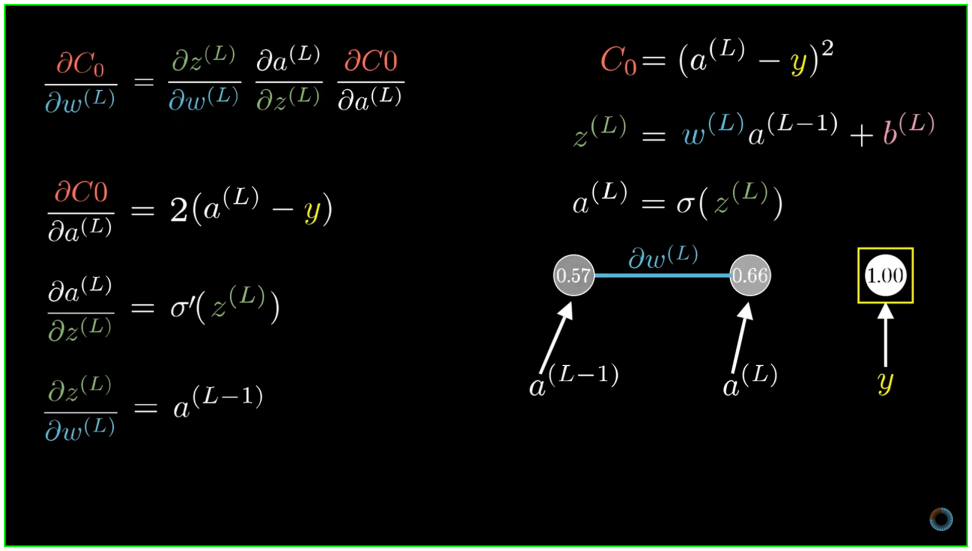

Figure 5: Understanding what the derivative terms implies

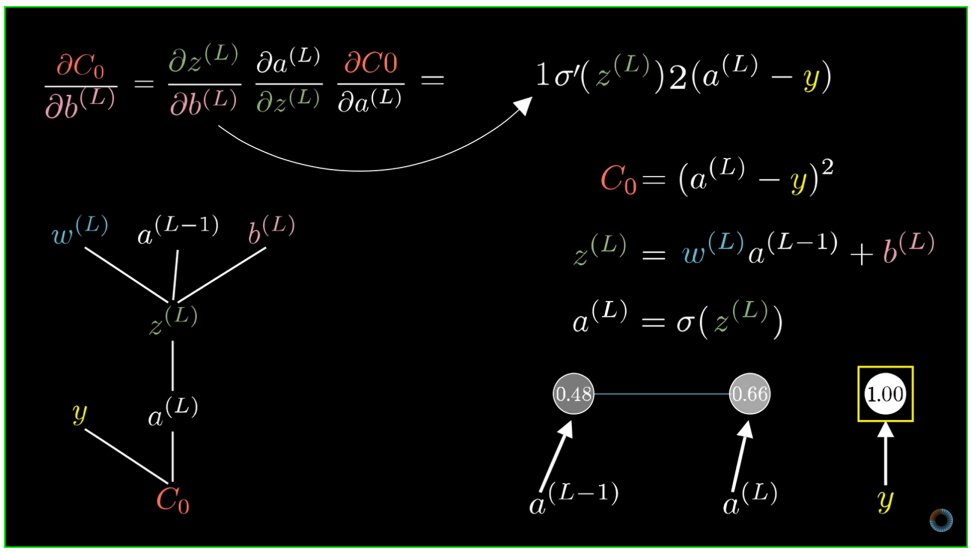

Figure 6: Derivative w.r.t bias term

2) We take the average of all the training examples to get the full derivative of the cost function so that later on this model could also be tested on other examples

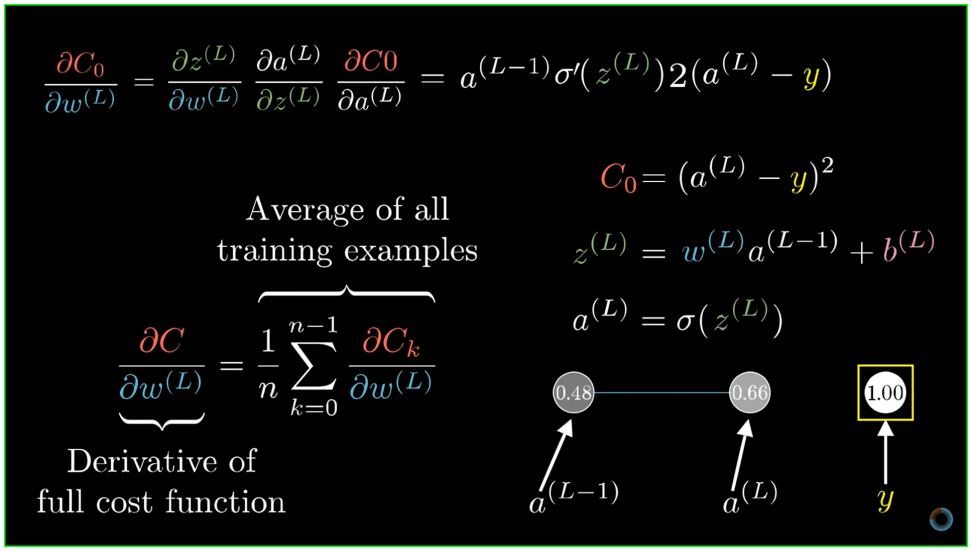

Figure 7: Averaging to all training examples

3) Calculated the derivative of the Cost function w.r.t the activation value of previous neuron and found out that it depends on the weight of last neuron. This represents how sensitive the cost function is with respect to the previous activation value. We can keep iterating to the previous activation values to see the sensitivity of the weights and biases.

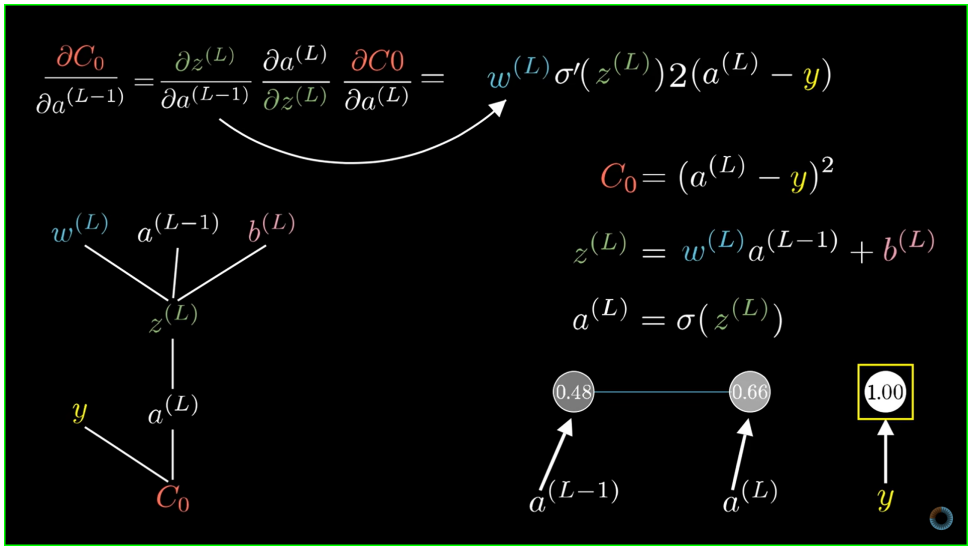

Figure 8: computing derivative w.r.t to the (L-1)th neuron's activation value

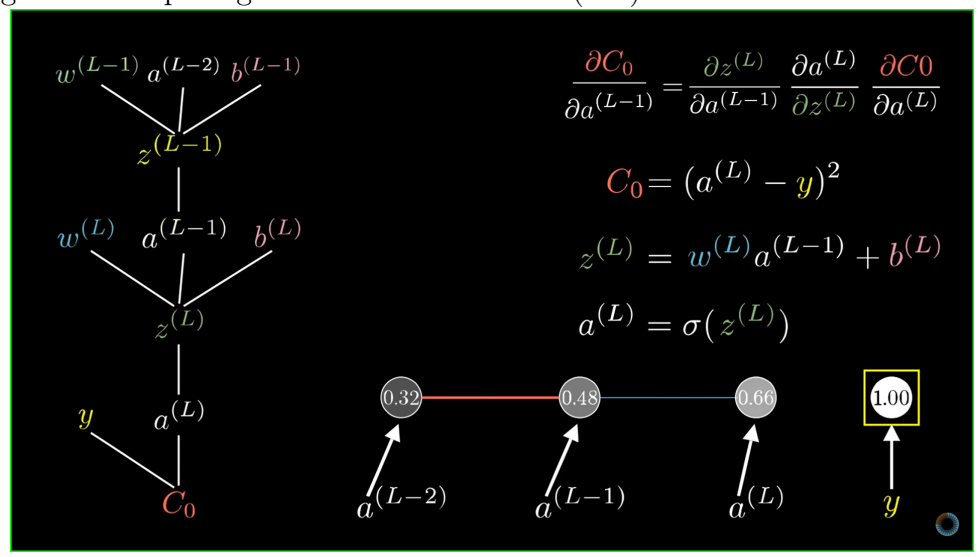

Figure 9: Propagating backwards following the same procedure with previous neurons

Answer 6

- The core idea of backpropagation is using the chain rule of calculus to compute gradients efficiently. It allows you to compute the gradient of the loss with respect to the weights layer by layer, working backward from the output layer to the input layer. This reduces the need for redundant calculations, as each weight update depends on the gradient of the error from the layer immediately following it, rather than recalculating from scratch for each layer.

- During training, instead of updating the weights after each training example (which would be computationally expensive), backpropagation can work on mini-batches of data. This allows the network to compute gradients for several examples at once, making the process more efficient and parallelizable.

- For a network with L layers, each layer having nl neurons and N training examples, the time complexity of backpropagation is approximately:
  - Forward Pass: O(N.d.L) where d is the average number of weights per neuron.
  - Backward pass: O(N.d.L) (since you are performing gradient calculations for each layer).
  - Weight Update: O(N.d) since you need to update each weight based on the gradient.

  So, in total, the time complexity for one pass through the network (forward + backward pass) is approximately O(N.d.L).
  This is much more efficient than trying to compute gradients or weight updates individually for each neuron without the use of the chain rule or the layered approach used in backpropagation.

  Why does this works

  - Computational Efficiency: Backpropagation ensures that each weight's contribution to the loss is computed efficiently by using the chain rule and propagating errors backwards, rather than recalculating from scratch.
  - Reuse of Computations: It reuses computations from the forward pass to compute gradients during the backward pass, which avoids redundant calculations and speeds up training.
  - Parallelism and Vectorization: By processing multiple examples in parallel and leveraging matrix operations, backpropagation is highly optimized in modern implementations, leading to reduced computational time.

Answer 7

Task 1

- Loading the dataset:

  load_iris() loads the Iris dataset with four features and three target classes (0, 1, 2).

- Splitting the dataset:

  train_test_split() splits the dataset into training (80%) and testing (20%) sets.

- Feature scaling:

  StandardScaler standardizes the features by removing the mean and scaling to unit variance.

- One-hot encoding:

  to_categorical() converts the integer target labels into one-hot encoded vectors. For example, label 0 becomes [1, 0, 0].

Task 2

- Input Layer:

  The input layer corresponds to the 4 features of the Iris dataset. This is specified using input_dim=4 in the first Dense layer.

- Hidden Layer:

  A single hidden layer with 8 neurons and a ReLU activation function (activation='relu').

- Output Layer:

  The output layer has 3 neurons (one for each class in the dataset) and uses the softmax activation function, which is suitable for multi-class classification.

- Compilation:

  Optimizer: Adam for efficient gradient-based optimization.
  Loss: Categorical Crossentropy, used for multi-class classification.
  Metric: Accuracy, to monitor model performance.

- Training:

  The model is trained for 50 epochs with a batch size of 8. A validation split of 20% of the training data is used to monitor performance on unseen data during training.

- Evaluation:

  The model's accuracy is evaluated on the test set after training.

Task 3

- Compilation:

  The model is compiled with:
  Adam optimizer: Efficient and commonly used for training neural networks. Categorical Crossentropy loss: Standard loss for multi-class classification tasks.

- Training Configuration:

  Epochs: Increased to 100, providing more opportunities for the model to learn.
  Batch Size: Set to 5, meaning the model updates its weights after every 5 examples.

- Validation Split:

  During training, 20% of the training data is used as a validation set to monitor performance and prevent overfitting.

Task 4

- model.evaluate():

  Computes the loss and metrics (accuracy in this case) on the test dataset.

- Accuracy: 0.97

Softmax
1) The softmax function is defined as:

$$y_i = e^{x_i} / \Sigma_{j=1}^{n}(e^{x_j})$$

where xi are the input values, yi are the outputs of the softmax function, and n is the number of input values.
(a) Non-negativity:

$$y_i = e^{x_i} / \Sigma_{j=1}^{n}(e^{x_j})$$

Since the exponential function $e_i^x > 0$ for all real numbers x_i, and the denominator $\Sigma_{j=1}^{n}(e^{x_j}) > 0$, it follows that $y_i > 0$

(b) Sum to 1:
To show the outputs sum to 1:
$\Sigma_{i=1}^{n}(y_i) = \Sigma_{i=1}^{n}(e^{x_i} / \Sigma_{j=1}^{n}(e^{x_j}))$
Since the denominator is constant with respect to i, we can factor it out: $\Sigma_{i=1}^{n}(y_i) = \Sigma_{i=1}^{n}(e^{x_i}) / \Sigma_{j=1}^{n}(e^{x_j})$
The numerator equals the denominator so $\Sigma_{i=1}^{n}(y_i) = 1$

2) **Intuition Behind the Softmax Operation:**
The softmax function transforms a set of arbitrary real numbers into probabilities, making it useful for multi-class classification tasks. It ensures the outputs are non-negative and sum to 1, satisfying the requirements of a probability distribution.

Intuitively, softmax amplifies the differences between the input values: larger inputs result in significantly higher probabilities, while smaller inputs are suppressed. This behavior helps emphasize the most likely class in classification problems.

Link to Github Code : Click Here

CNN

1) MNIST Digits Dataset
Insights

- **Classification Accuracy:**

  The MNIST dataset consists of grayscale images of handwritten digits (0–9). A well-trained model on this dataset achieves high accuracy (¿99

- **Feature Importance:**

  MNIST digits have distinct patterns, such as loops and strokes, making them ideal for models to capture low-level (edges, corners) and high-level (shapes) features during training.

- **Model Generalizability:**

  Models trained on MNIST often serve as a benchmark for computer vision tasks and transfer learning applications. Despite its simplicity, insights from MNIST model performance help refine architectures for more complex datasets.

Model Functionality
Understanding of How the Models Work:

- Input Layer: Accepts 28x28 grayscale images, normalized to a range of [0, 1].
- Convolutional Layers: Extract spatial features by applying convolution filters, which detect edges, textures, and patterns.
- Pooling Layers: Reduce spatial dimensions while retaining critical information, making computation efficient and preventing overfitting.
- Activation Functions: Introduce non-linearity to learn complex mappings between input features and outputs.
- Fully Connected Layers: Aggregate extracted features and map them to the final output classes (digits 0–9).
- Output Layer: Uses softmax to produce probabilities for each digit class.

Learned Features

- Low-Level Features:

  Edges, corners, and basic shapes, extracted by initial convolutional layers.
- Mid-Level Features:

  Patterns like loops, straight strokes, and diagonal lines, which are parts of digits.
- High-Level Features:

  Entire digit shapes and their distinguishing characteristics (e.g., closed loops in "8").

Layer Names

- Convolutional Layers: Responsible for feature extraction.
- Pooling Layers (Max Pooling): Downsample feature maps while retaining key information.
- Fully Connected Layers: Perform classification by combining all learned features.
- Activation Layers (ReLU): Introduce non-linearity and prevent linear mappings.
- Output Layer (Softmax): Converts logits to probability distributions across the 10 digit classes.

Layer Importance

- Convolutional Layers:

  Role: Detect patterns in the input images.

  Importance: Essential for capturing spatial hierarchies in data.

- Pooling Layers:

  Role: Reduce feature map size, enhancing computational efficiency and robustness.

  Importance: Prevent overfitting and make the model invariant to small transformations.

- Activation Layers (ReLU):

  Role: Apply non-linear transformations to the feature maps.

  Importance: Helps learn complex functions and ensures effective backpropagation.

- Fully Connected Layers:

  Role: Aggregate features and map them to output classes.

  Importance: Final decision-making layers that classify the digit.

- Output Layer (Softmax):

  Role: Produce probabilities for each digit class.

  Importance: Ensures the output is interpretable as a probability distribution.