* Loss Function
* Gradient descent vs momentum vs nestrov vs adam
* Backprop
* Layer norm vs batch norm
* Activation Functions:
* Regularization
* Forward pass

**Batch Normalization (BN):**

Address the internal covariance shift. It normalizes the activation by subtracting mean and variance,

1. Fixes slow and unstable training
2. Interdependence between distribution
3. Normalization is done before the activation function
4. Layer smoothness

Implementation

Training phase :





→ The BN layer first determines the mean 𝜇 and the variance σ² of the activation values across the batch,

→ normalize the vector

→ linear transformation using gamma and beta (scale and shift)

→ these are trained using the exponential moving average

**Layer Normalization (LN):**

Layer normalization operates on a single training example but normalizes across the features of that example. It normalizes the activations of a layer by subtracting the mean and dividing by the standard deviation computed across the features.

LN does not depend on batch size, making it suitable for tasks with varying batch sizes or for models that don't use batches, such as RNNs.

LN does not introduce additional parameters to learn; the normalization is applied directly.

LN is often used in RNNs, as it helps in stabilizing the training process by reducing the impact of vanishing or exploding gradients.

The choice between BN and LN depends on the specific problem, network architecture, and requirements. BN is commonly used in feedforward neural networks, especially deep convolutional neural networks (CNNs). LN, on the other hand, is often preferred in RNNs or when batch sizes are small or inconsistent.

It's worth noting that there are also other normalization techniques available, such as instance normalization, group normalization, and adaptive normalization, each with its own characteristics and use cases.

**Loss Function**

Binary Cross-Entropy Loss: This loss function is commonly used in binary classification problems. It measures the dissimilarity between predicted probabilities and true binary labels. It is based on the concept of information entropy and is effective when dealing with imbalanced datasets.

Intuition behind Cross-Entropy Loss:

The intuition behind using cross-entropy loss for classification is to encourage the model to assign high probabilities to the correct classes. If the predicted probabilities are close to one for the correct class and close to zero for the other classes, the loss will be low. However, if the predicted probabilities deviate from the true class labels, the loss will increase.

The logarithm is used in the loss calculation because it amplifies the error when the predicted probability is far from the true label. Taking the logarithm of a value close to one yields a small negative number, while taking the logarithm of a value close to zero yields a large negative number. By using logarithms, the loss function penalizes larger deviations from the true class label more severely.

**Activation Function**

**Sigmoid Function:** The sigmoid function maps the input to a range between 0 and 1. It is given by the formula: *f(x) = 1 / (1 + e^(-x)).* Sigmoid functions were popular in the past but are now less commonly used in hidden layers due to the vanishing gradient problem.

**Tanh (Hyperbolic Tangent) Function**: The hyperbolic tangent function maps the input to a range between -1 and 1. It is given by the formula: *f(x) = (e^x - e^(-x)) / (e^x + e^(-x))*. Like the sigmoid function, the tanh function can suffer from the vanishing gradient problem.

**Rectified Linear Unit (ReLU)** Function: The ReLU function returns 0 for negative inputs and the input value for non-negative inputs. It is given by the formula: f(x) = max(0, x). ReLU is widely used in deep learning due to its simplicity and effectiveness in addressing the vanishing gradient problem.

Leaky ReLU Function: The leaky ReLU function is an extension of the ReLU function that allows a small, non-zero gradient for negative inputs. It is given by the formula: f(x) = max(αx, x), where α is a small constant. Leaky ReLU helps mitigate the "dying ReLU" problem where neurons can become stuck in a state of zero gradient.

Parametric ReLU (PReLU) Function: PReLU is similar to leaky ReLU, but the α parameter is learned during training rather than being fixed. This allows the network to adaptively determine the best value for α.

Exponential Linear Unit (ELU) Function: The ELU function smoothly handles negative inputs and has an exponential curve for positive inputs. It is given by the formula: f(x) = x if x > 0, and f(x) = α \* (e^x - 1) if x <= 0, where α is a small constant.

Swish Function: The swish function applies a sigmoid-like transformation to the input. It is given by the formula: f(x) = x / (1 + e^(-βx)), where β is a parameter that controls the slope of the function. Swish has been shown to be an effective activation function in some cases.

**Softmax Function:** The softmax function is commonly used in the output layer of classification problems. It takes a vector of real numbers as input and converts them into a probability distribution over multiple classes. The output values are in the range [0, 1], and their sum is equal to 1.

def softmax(x):

e\_x = np.exp(x) # Subtracting the maximum value for numerical stability

return e\_x / np.sum(e\_x, axis=0)

# Example usage

x = np.array([1, 2, 3])

output = softmax(x)

**Optimization Algorithms:**

Stochastic Gradient Descent (SGD): SGD is the most basic and widely used optimization algorithm. It updates the model's parameters by computing the gradient of the loss function with respect to the parameters on a small random subset of the training data at each iteration.

Momentum: Momentum builds upon SGD by introducing a momentum term that accumulates gradients over time. It helps accelerate convergence and navigate shallow local minima in the loss landscape.

AdaGrad: AdaGrad adapts the learning rate of each parameter based on the historical gradients. It scales down the learning rate for frequently updated parameters and scales up the learning rate for infrequently updated parameters. This algorithm is particularly useful for sparse datasets.

RMSprop: RMSprop is an extension of AdaGrad that addresses its diminishing learning rate problem. It divides the learning rate by an exponentially decaying average of the squared gradients.

Adam: Adam combines the ideas of momentum and RMSprop. It maintains exponentially decaying averages of both the gradients and their squared values. Adam is widely used in practice due to its efficiency and good performance across different types of neural networks.

Adadelta: Adadelta is similar to RMSprop but uses a more advanced update rule that dynamically adapts the learning rate based on a moving window of the recent gradient updates.

Adamax: Adamax is a variant of Adam that uses the infinity norm (maximum) of the gradients instead of their L2 norm. It is less sensitive to the scale of the gradients and can be useful when dealing with sparse gradients.

Nadam: Nadam combines the Nesterov accelerated gradient (NAG) and Adam algorithms. It incorporates the NAG technique, which allows the optimization algorithm to look ahead of the current parameter update.