CAP 6619-001

DEEP LEARNING

ASSIGNMENT-2

Sai Supraja Chinthapalli

Z-number: Z23760554

1. If we want to change perceptron to logistic regression, what do we do?

To change a perceptron model to logistic regression, there is a need to adjust both the architecture and the training process.

Replace Activation Function: In a perceptron, the activation function is typically a step function, which outputs binary values (0 or 1). Logistic regression, on the other hand, uses a logistic (sigmoid) activation function to obtain the output between 0 and 1. So, we need to replace the step function with the logistic function.

Modify Output Transformation: Since logistic regression outputs probabilities, we need to adjust the output transformation accordingly. Instead of simply taking the sign of the dot product as in a perceptron, we'll interpret the output of the logistic function as the probability of the positive class.

Update Loss Function: Perceptron uses the perceptron loss, which penalizes misclassifications linearly. Logistic regression typically uses the cross-entropy loss function, which is better suited for probabilistic predictions. This loss function measures the difference between the predicted probabilities and the actual labels.

Training Algorithm: While both perceptron and logistic regression can use gradient descent for optimization, logistic regression's cross-entropy loss function requires the gradient to be computed differently. Ensuring that the training algorithm is adapted to compute gradients for the logistic regression loss function.

Regularization: Logistic regression often incorporates regularization techniques like L1 or L2 regularization to prevent overfitting.

So, to change from a perceptron to logistic regression, we need to replace the step function with the sigmoid function as the activation function and implement a learning algorithm like gradient descent to minimize the logistic loss function.

2. The following is the error function of perceptron.

$$E[W] = \frac{1}{2}\sum_{d \epsilon D}(t_d - O_d)^2$$

2(a): Show O_d in perceptron training rule(PTR)

The perceptron training rule is based on adjusting the weights to minimize the error function. The perceptron training rule (PTR) is commonly associated with the gradient descent algorithm. To minimize the error function E[W] we need to update the weights W to decrease the error.

The perceptron training rule is often expressed as:

$$\Delta\omega_{ij} = \eta \cdot (t_d - O_d) \cdot x_{ij}$$

Where :

$\Delta\omega_{ij}$ is the change in weight between the input neuron i and the output neuron j.

η is the learning rate, a small positive constant that determines the step size in the weight update.

t_d is the target output for the training example d.

O_d is the actual output for the training example d.

x_{ij} is the input from neuron i to neuron j.

The output O_d is determined by comparing the weighted sum of inputs to a threshold. Assuming a step function as the activation, the perceptron output O_d can be expressed as:

$$O_d = \begin{cases} 1, & if\ \Sigma_i W_i \cdot x_{di} \geq threshold \\ 0, & otherwise \end{cases}$$

So, the perceptron training rule, O_d is the output of the perceptron for the training example d, and it is used to update the weights in the direction that reduces the error.

2(b): Show O_d in Delta rule(DR):

The Delta Rule (also known as the Widrow-Hoff or LMS rule) is a generalization of the perceptron learning rule for updating weights. It is commonly used in the context of gradient descent for updating weights in a continuous-valued output neuron. The Delta Rule is expressed as:

$$\Delta\omega_{ij} = \eta \cdot (t_d - O_d) \cdot x_{ij}$$

Where :

$\Delta\omega_{ij}$ is the change in weight between the input neuron i and the output neuron j.

η is the learning rate, a small positive constant that determines the step size in the weight update.

t_d is the target output for the training example d.

O_d is the actual output for the training example d.

x_{ij} is the input from neuron i to neuron j.

To express O_d in the form of delta rule the formula is given as:

$$\boldsymbol{O_d = O_d + \Delta O_d}$$

Where is the change in output and given by:

$$\boldsymbol{\Delta O_d = \eta \cdot (t_d - O_d)}$$

Equating the equation with 0 we get:

$$0 = -\sum_{d \in D}(\boldsymbol{t_d - O_d})$$

$$\sum_{d \in D} \boldsymbol{O_d} = \sum_{d \in D} \boldsymbol{t_d}$$

Assuming D as the number of training examples $\boldsymbol{O_d}$ is given as:

$$\boldsymbol{O_d} = \frac{\boldsymbol{1}}{\boldsymbol{|D|}} \sum_{d \in D} \boldsymbol{t_d}$$

The Delta Rule is used to update the weights in the process of minimizing the error, and $\boldsymbol{O_d}$ is not explicitly expressed in terms of $\boldsymbol{t_d}$ in the Delta Rule. Instead, $\boldsymbol{O_d}$ is the output of the perceptron, which is updated during the training process based on the Delta Rule.

2(c): Explain the differences between PTR and DR in terms of convergence.

The differences between PTR and DR are as follows:

Aspect	Perceptron Training Rule (PTR)	Delta Rule (DR)
Objective Function	Minimize the error function for perceptrons.	Minimize the mean squared error for continuous-valued outputs.
Output Representation	Typically binary (0 or 1) due to step function.	Can handle continuous-valued outputs, more flexible.
Weight Update Rule	Proportional to the difference between target and actual output, multiplied by the input.	More generalized formula with a learning rate, suitable for various activation functions.

Convergence Guarantee	Converges if the data is linearly separable.	Convergence depends on network architecture, learning rate, and dataset characteristics.
Convergence Speed	May converge quickly for linearly separable datasets.	Convergence speed depends on factors like learning rate and network architecture.
Applicability	Specific to single-layer perceptrons.	More versatile, applicable to a broader range of neural network architectures.
Performance on Nonlinear Data	May not converge for datasets that are not linearly separable.	Can handle more complex patterns and nonlinear relationships.
Training Complexity	Simpler due to binary output and straightforward weight update.	More complex, especially for networks with continuous outputs and the need for tuning the learning rate.

2(d): Suppose perceptron uses ReLU activation function. In this case, show the formula of $\frac{\partial E}{\partial \omega_i}$

Given error function is:

$$E[W] = \frac{1}{2}\sum_{d \epsilon D}(t_d - O_d)^2$$

Considering a perceptron with ReLU activation function, the output of the perceptron is computed as:

$O_d = ReLU(w.x_d)$

where w is the weight vector, x_d is the input vector for the d-th training example, and ReLU(z) is the Rectified Linear Unit activation function, defined as:

ReLU(z)= max(0,z)

the partial derivative of the error with respect to the i-th weight of ω_i **is:**

$$\frac{\partial E}{\partial \omega_i} = -\Sigma_d(t_d - O_d) \cdot \frac{\partial O_d}{\partial \omega_i}$$

Since ReLU is a piecewise function, we consider two cases:

If w. $x_d > 0$**:**

$$\frac{\partial O_d}{\partial \omega_i} = \frac{\partial (w.x_d)}{\partial \omega_i} = x_{di}$$

If w. $x_d <= 0$**:**

$$\frac{\partial O_d}{\partial \omega_i} = \frac{\partial (w.x_d)}{\partial \omega_i} = 0$$

Combining both the cases we get:

$$\frac{\partial O_d}{\partial \omega_i} = x_{di}.\ \Pi(\text{w.}\ x_d > 0)$$

Where Π(.) is the indicator function

Finally we get,

$$\frac{\partial E}{\partial \omega_i} = -\Sigma_d(t_d - O_d).x_{di}.\ \Pi(\text{w.}\ x_d > 0)$$

3.Given $f(x, y) = 2x^2 + y^2 + y + 1$

3(a): compute the derivative of $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ respectively

Finding $\frac{\partial f}{\partial x}$:

$$\frac{\partial f}{\partial x} = \frac{\partial(2x^2+y^2+y+1)}{\partial x}$$

Taking the derivative of each term with respect to x we get:

$\frac{\partial f}{\partial x}$ = 4x

So $\frac{\partial f}{\partial x}$ = 4x

Finding $\frac{\partial f}{\partial y}$:

$$\frac{\partial f}{\partial y} = \frac{\partial(2x^2+y^2+y+1)}{\partial y}$$

Taking the derivative of each term with respect to y we get:

$\frac{\partial f}{\partial y}$ = 2y+1

So $\frac{\partial f}{\partial y}$ = 2y+1

3(b) Suppose that we start at position ($x1$,$y1$)=(300,150). Compute the gradient vector using q. 1)

From question 3a,

$\frac{\partial f}{\partial x}$ **= 4x**

At (x1,y1)=(300,150), the partial derivative with respect to x is :

$\frac{\partial f}{\partial x}$ **= 4* 300=1200**

From question 3a**,**

$\frac{\partial f}{\partial y}$ **= 2y+1**

At (x1,y1)=(300,150), the partial derivative with respect to y is :

$\frac{\partial f}{\partial y}$ **= 2*150+1 = 301**

So the gradient vector at the point (x1,y1) = (300,150) is given by: [1200,301]

3(c): Compute next position (learning rate=1)

To find the next position using gradient descent, The update rule is given by:

New Position=Old Position−Learning Rate×∇f

Given the function $\boldsymbol{f(x, y) = 2x^2 + y^2 + y + 1}$ and the gradient vector ∇f=[1200,301], and assuming the learning rate is 1 the new position is as follows:

New position = (300,150) -1 *[1200,301]

New position = [300-1200, 150-301]

New position = [-900, -151]

Therefore the next position using a learning rate of 1 is (-900,-151)

4. Perceptron is a linear classifier. To change perceptron non-linear classifier, the following two things must be done

4(a): Activation function:

To transform a perceptron into a non-linear classifier, we need to change the activation function. The perceptron, as a linear classifier, uses a step function as its activation. This step function results in a binary output (0 or 1) based on whether the weighted sum of inputs is above or below a threshold.

To introduce non-linearity, you can replace the step function with a non-linear activation function. Here are some commonly used non-linear activation functions:

- **Sigmoid Function(Logistic):**

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

This function provides the output between 0 and 1 and is commonly used in the hidden layers of a neural network.

- **Hyperbolic Tangent Function(tanh):**

$$\textbf{tanh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Similar to the sigmoid function, but it squashes the output between -1 and 1.

- **Rectified Linear Unit(ReLU):**

$$\textbf{ReLU(z)= max(0,z)}$$

It outputs the input if it's positive; otherwise, it outputs zero. ReLU is popular due to its simplicity and effectiveness.

By introducing any of these non-linear activation functions we can transform it into a non-linear classifier.

4(b): Layer

A single-layer perceptron can only learn linear decision boundaries. To transform it into a non-linear classifier, we need to add one or more hidden layers between the input and output layers.

Each hidden layer applies a linear transformation to the input followed by the non-linear activation function.

The depth of the network, represented by the number of hidden layers, allows the model to learn hierarchical representations of the input data, capturing more intricate patterns.

In summary, to change a perceptron into a non-linear classifier:

Replace the activation function with a non-linear one.

Add one or more hidden layers between the input and output layers.

These modifications enable the model to learn and represent non-linear relationships in the data, making it more powerful and suitable for a wider range of tasks.

5. We use backpropagation algorithm (p. 16 in slide) in the following neural network. Suppose output $O_{1.}$=0.6 and target value (of $O_{1.}$) $t1$ = 1. Also $w_{11}^{(2)}$ = 0.5, $O_{1.}(2)$ = 0.7, and η =1.

(In p. 16 pseudo code, x_{ji} means $O_{1.}^{(2)}$ in this network)

5(a): Compute δ_1

To compute $\boldsymbol{\delta_1}$ the error term for the output layer in a neural network using the backpropagation algorithm, we can use the following formula:

$$\boldsymbol{\delta_1 = (t_1 - O_{1.}) \cdot O_1(1 - O_{1.})}$$

Given:

$O_{1.}$ = 0.6(output of the neuron)

t_1 = 1(target value for $O_{1.}$)

Substitute these values into the formula:

δ_1 = (1- 0.6) . 0.6 . (1- 0.6)

Calculating the result we get:

δ_1 = 0.4*0.6*0.4

δ_1 = 0.096

So δ_1 = 0.096

This is the error term for the output layer, and it is used in the weight update step during backpropagation.

5(b) Update $w_{11}^{(2)}$

To update the weight $w_{11}^{(2)}$ using the backpropagation algorithm, we use the following formula:

$$w_{11}^{(2)} \leftarrow w_{11}^{(2)} + \eta \cdot \delta_1 \cdot O_{1.}^{(2)}$$

Given:

$w_{11}^{(2)}$ = 0.5(initial weight)

η = 1 (learning rate)

δ_1 = 0.096(error term for the output layer)

$O_{1.}^{(2)}$ = 0.7 (output of the neuron in the hidden layer connected to $w_{11}^{(2)}$)

Substituting the values in the formula we get:

$w_{11}^{(2)} \leftarrow$ 0.5+1*0.096*0.7

$w_{11}^{(2)} \leftarrow$ 0.5+0.0672

$w_{11}^{(2)} \leftarrow$0.5672

Therefore after the update, $w_{11}^{(2)}$ becomes approximately 0.5672. This is the new weight for the connection between the hidden layer neuron and the output layer neuron.

6. Comparing MAE and MSE

6(a): What are the advantages of MSE over SSE?

In the context of error metrics like Mean Squared Error (MSE) and Sum of Squared Errors (SSE), MSE is essentially the average of SSE. The advantages of using MSE over SSE are as follows:

- **Normalization by Sample Size:**

  One key advantage of MSE is that it normalizes the error by the number of samples. MSE is calculated as the sum of squared errors divided by the number of samples (mean squared error), providing a measure of the average error per data point.

  This normalization makes MSE more interpretable and less dependent on the absolute scale of the dataset, which is especially useful when dealing with datasets of varying sizes.

- **Consistency in Evaluation:**

  MSE provides a consistent and standardized measure of the average error across different datasets and problems. By normalizing the sample size, MSE ensures that the error metric is not biased towards larger datasets, making it easier to compare models across various scenarios.

- **Mathematical Properties:**

  From a mathematical standpoint, MSE has desirable properties. It is differentiable, which is crucial for optimization algorithms that rely on derivatives. This property makes MSE well-suited for optimization problems in machine learning, where gradient-based optimization techniques are commonly employed.

- **Statistical Interpretation:**

  MSE can be interpreted as an estimate of the variance of the underlying distribution of errors. This provides insights into the variability of the model's predictions, helping practitioners understand the overall quality of the model's performance.

- **Reduced Sensitivity to Outliers:**

  MSE is less sensitive to outliers compared to SSE. Squaring the errors in MSE penalizes larger errors more heavily than smaller errors, but it has the effect of reducing the impact of extreme outliers. This makes MSE more robust in the presence of data points with unusually large errors.

In summary, MSE offers advantages over SSE by normalizing the error metric, providing consistency in evaluation across different datasets, possessing desirable mathematical properties, offering statistical interpretation, and exhibiting reduced

sensitivity to outliers. These factors contribute to the widespread use of MSE in various applications.

6(b). Show the error function formula of MAE and MSE, respectively.

The error functions for Mean Absolute Error (MAE) and Mean Squared Error (MSE) are as follows:

Mean Absolute Error(MAE):

MAE is calculated as the difference between the predicted values (MSE is calculated as the average of the squared differences between the predicted values ($\hat{y}$) and the true values and the true values (y).

$$\mathbf{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Where :

n = The number of samples

y_i= The true value for the i-th sample

$\hat{y}_i$ = The predicted value for the i-th sample

Mean Square Error (MSE):

MSE is calculated as the average of the squared differences between the predicted values ($\hat{y}$) and the true values (y)

$$\mathbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where:

n = The number of samples

y_i= The true value for the i-th sample

$\hat{y}_i$ = The predicted value for the i-th sample

MAE measures the average absolute deviation between predictions and true values, while MSE measures the average squared deviation.

6(c): Explain why MSE is more sensitive to outliers than MAE using the definition of the formula

The Mean Squared Error (MSE) is more sensitive to outliers compared to the Mean Absolute Error (MAE) because of the squaring operation in its formula

Mean Absolute Error(MAE):

$$\mathbf{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

In the MAE formula, the absolute differences $|y_i - \hat{y}_i|$ are taken for each data point. The absolute value function ensures that the contribution of each data point to the overall error is linear and does not depend on the direction (overestimation or underestimation).

Mean Square Error (MSE):

$$\mathbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

In the MSE formula, the squared differences $(y_i - \hat{y}_i)^2$ are taken for each data point. Squaring the differences amplifies the impact of larger deviations. Consequently, outliers with substantial differences from the predicted values have a disproportionately larger effect on the overall error.

For example:

Suppose the predicted value $\hat{y}_i$ is 5, and the true value y_i is 10 for a particular data point.

The absolute difference for this point in MAE is $|10-5|=5$

The squared difference for this point in MSE is $(10-5)^2=25$

From the above example squared difference is larger than the absolute difference. When dealing with outliers, which may have significantly larger deviations, the squaring operation in MSE magnifies these differences, making MSE more sensitive to extreme values.

Therefore, the squaring operation in MSE gives more weight to larger errors, making it more sensitive to outliers compared to MAE, which treats all errors with equal importance.

7. Explain why (in most cases) minimizing KL divergence is equivalent to minimizing cross-entropy

In most cases, minimizing the Kullback-Leibler (KL) divergence between two probability distributions is equivalent to minimizing the cross-entropy between the same distributions. This equivalence holds true under certain conditions, typically when one of the distributions is the true (or target) distribution and the other is the predicted distribution.

Kullback-Leibler (KL) Divergence:

KL divergence measures the difference between two probability distributions P and Q. It is defined as:

$$D_{KL}(P||Q) = \Sigma_i P(i) \log\left(P(i)/Q(i)\right)$$

KL divergence quantifies the amount of information lost when using Q to approximate P It's not symmetric and measures the "divergence" or "distance" between the distributions.

Cross-Entropy:

Cross-entropy is commonly used as a loss function in machine learning and is related to KL divergence.

For a true probability distribution P and a predicted (or estimated) probability distribution Q, the cross-entropy loss is defined as:

$$H(P,Q) = -\Sigma_i P(i) \log(Q(i))$$

Cross-entropy measures the average number of bits needed to encode events from P, using the optimal code based on Q.

In classification tasks, where P is typically the true distribution (one-hot encoded labels) and Q is the predicted distribution (softmax output of the model), minimizing cross-entropy encourages the predicted distribution to be similar to the true distribution.

The equivalence between minimizing KL divergence and minimizing cross-entropy arises from the fact that the cross-entropy between two probability distributions

P and Q can be expressed as the sum of the entropy of P and the KL divergence between P and Q:

$$H(P,Q) = H(P) + D_{KL}(P||Q)$$

When minimizing cross-entropy, the entropy of the true distribution P is constant. Thus, minimizing cross-entropy is equivalent to minimizing the KL divergence between P and Q, as the term H(P) becomes a constant during optimization. This equivalence makes cross-entropy a convenient and widely-used loss function in various machine learning tasks, particularly in classification problems, where it is often used in conjunction with softmax activation functions.

8. Explain why sigmoid causes vanishing gradient

The vanishing gradient problem is a phenomenon that can occur during the training of neural networks, particularly deep networks, and it is associated with the use of activation functions that provide their input into a small range. The sigmoid activation function is one such function that can contribute to the vanishing gradient problem.

The sigmoid function is defined as:

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

The primary issue with the sigmoid function in the context of deep learning is that its output saturates and becomes very close to 0 or 1 for extreme input values (very positive or very negative). This saturation leads to gradients that are close to zero during backpropagation, and when these small gradients are multiplied during the update of the weights through gradient descent, they can cause the gradients to vanish as they are propagated backward through the layers.

The vanishing gradient problem is particularly problematic in deep networks with many layers. During backpropagation, as the gradients are multiplied layer by layer, if the gradients are consistently very small (close to zero), the updates to the weights become negligible, and the network fails to learn effectively.

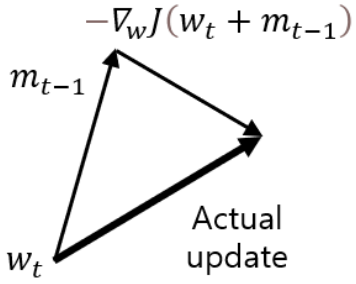
The sigmoid function's derivative with respect to its input is given by:

$$\boldsymbol{\sigma'(x) = \sigma(x).(1 - \sigma(x))}$$

We can observe that the maximum value of $\sigma'(x)$ is 0.25 when $\sigma(x)$ = 0.5. This means that for extreme values of x where $\sigma(x)$ is close to 0 or 1, the gradient $\sigma'(x)$ becomes very small, contributing to the vanishing gradient problem.

To address the vanishing gradient problem, alternative activation functions like Rectified Linear Unit (ReLU) and its variants have been introduced, as they do not saturate for positive input values and mitigate the vanishing gradient issue to some extent.

9. Explain NAG method using the following picture.



The Nesterov Accelerated Gradient (NAG) method, is an optimization algorithm used in training neural networks. In the NAG method, the gradient is modified to take into account the momentum from the previous iteration.

NAG is commonly visualized as follows:

Gradient Descent Update:

In standard gradient descent, the weight update (Δw_t) is proportional to the negative gradient of the loss function (J (w_t))

$$\boldsymbol{\Delta w_t = -\eta\, \Delta J(w_t)}$$

Where η is the learning rate

NAG Update:

NAG introduces a momentum term (m_{t-1}) that takes into account the momentum from the previous iteration:

$$\boldsymbol{\Delta w_t = -\eta\, \Delta J(w_t + m_{t-1})}$$

Visualization:

In a triangle diagram, we can see that w_t at the apex of the triangle and m_{t-1} at the base of the triangle and $-\Delta w J(w_t + m_{t-1})$ as the vector connecting w_t and m_{t-1}.

This vector represents the "correction" term, adjusting the weight update based on the momentum from the previous iteration.

Actual Update:

The actual update is the sum of the standard gradient descent update and the momentum correction term:

$$\boldsymbol{\omega_{t+1} = \omega_t + \Delta \omega_t}$$

This includes both the negative gradient term and the momentum correction term.

The Nesterov Accelerated Gradient method adjusts the standard gradient descent update by taking into account the momentum term from the previous iteration. The visualization with the triangle helps depict how the correction term influences the

weight update. The actual update involves adding the corrected update to the current weights.

10. Using the following formula, explain how RMSProp improves AdaGrad

$$G_t = \gamma G_{t-1} + (1-\gamma)(\nabla_w J(w_t))^2$$

RMSProp (Root Mean Square Propagation) is an optimization algorithm designed to address some of the limitations of AdaGrad by adapting the learning rates for each parameter during training. Let's break down how RMSProp improves upon AdaGrad using the given formula:

$$G_t = \gamma G_{t-1} + (1-\gamma)(\nabla_w J(w_t))^2$$

Here Gt represents the accumulated squared gradients up to time step t, γ is the decay parameter (typically close to 1), $\nabla w J(w_t)$ is the gradient of the loss function with respect to the parameters (w_t) , and $(1-\gamma)(\nabla w J(wt))^2$ represents the squared gradients for the current time step, adjusted by the decay parameter.

RMSProp improves upon AdaGrad as follows:

- **Adaptive Learning Rates:**

  In AdaGrad, the learning rates are adapted based on the accumulated squared gradients. However, AdaGrad's learning rates can become overly conservative over time, as the accumulated squared gradients keep increasing.

  RMSProp addresses this issue by introducing a decay term (γ) that scales the influence of past squared gradients. This decay term allows RMSProp to adapt the learning rates more effectively, preventing them from decreasing too aggressively over time.

- **Effective Learning Rates:**

  RMSProp computes the root mean square (RMS) of the gradients, rather than the sum of squared gradients as in AdaGrad. This modification ensures that the learning rates are adjusted based on a more stable estimate of the gradient magnitudes.

  By taking the square root of the accumulated squared gradients, RMSProp effectively normalizes the learning rates, preventing them from becoming too large or too small.

- **Less Aggressive Accumulation:**

The inclusion of the decay term γ in RMSProp helps to attenuate the influence of past squared gradients. This prevents the accumulation of squared gradients from growing too rapidly, allowing RMSProp to maintain a balance between adaptability and stability.

In summary, RMSProp improves upon AdaGrad by introducing adaptive learning rates that are scaled based on a decayed estimate of the squared gradients. This approach ensures more effective and stable learning rates, preventing them from becoming overly aggressive or overly conservative during training.

11. In LeCun or Xavier initialization, explain why variance is divided by n_{in} (or $n_{in} + n_{out}$)

LeCun and Xavier (Glorot) initializations are techniques used to initialize the weights of neural networks in a way that helps address the challenges of training deep networks. In these initialization methods, the variance of the weights is divided by the number of input units n_{in} or the sum of the number of input and output units ($n_{in} + n_{out}$).

Normalizing Variance:

The purpose of weight initialization is to set the initial values of weights in a way that avoids vanishing or exploding gradients during the training of deep neural networks.

By dividing the variance, the goal is to normalize the scale of the weights. Normalization helps in maintaining a reasonable scale for the weights, preventing them from becoming too large or too small.

Glorot Initialization (Xavier):

In the context of Glorot initialization (also known as Xavier initialization), the weights are initialized from a distribution with zero mean and a variance given by:

$$\textbf{Var(W)} = \frac{2}{n_{in}+n_{0ut}}$$

The division by $n_{in} + n_{0ut}$ ensures that the variance of the weights is proportional to the number of input and output units. This normalization is designed to account for the number of connections to each neuron and helps in balancing the scales.

LeCun Initialization:

LeCun initialization is a specific case of Xavier initialization where the weights are initialized with a variance given by:

$$\mathbf{Var(W)} = \frac{\mathbf{1}}{\boldsymbol{n_{in}}}$$

In LeCun initialization, the division is by n_{in} to normalize the variance based on the number of input units. This normalization is considered beneficial for activation functions with linear regions, such as the hyperbolic tangent (tanh) activation function.

In both cases, the division by the number of input units or the sum of input and output units helps to prevent the weights from being initialized with overly large or small values, promoting more stable and effective training of neural networks. These initialization techniques are particularly useful in deep networks to ensure that the gradients during backpropagation neither vanish nor explode, thus facilitating the optimization process.

12. Normalizing with Gaussian N(0,1) with sigmoid function might make DNN a linear classifier. Explain it.

Normalizing the weights using a Gaussian distribution with a mean of 0 and standard deviation of 1, commonly denoted as N(0,1) can lead to the behavior where deep neural networks (DNNs) with sigmoid activation functions effectively operate as linear classifiers. This phenomenon is related to the characteristics of the sigmoid activation function and the nature of the normalization.

- **Sigmoid Function(Logistic):**

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

This function provides the output between 0 and 1 and is commonly used in the hidden layers of a neural network.

- **Gaussian Initialization (N(0,1)):**

When the weights of a neural network are initialized from a Gaussian distribution with mean 0 and standard deviation 1, it implies that the weights are randomly sampled from a standard normal distribution.

The effect of the combination of these factors are as follows:

Effect of Sigmoid Activation:

The sigmoid activation function tends to saturate for extreme input values (very positive or very negative).

In the saturated regions of the sigmoid, the derivative is close to zero. This characteristic can lead to the vanishing gradient problem, making it challenging for the network to learn complex patterns.

Effect of Gaussian Initialization:

If the weights are initialized with a standard normal distribution N(0,1)), the initial output of neurons will also be centered around zero.

When the outputs are around zero, the sigmoid function maps them to the linear region near its midpoint (approximately 0.5).

Linear Behavior near Sigmoid Midpoint:

In the linear region of the sigmoid around its midpoint, the sigmoid function behaves approximately like a linear transformation.

The outputs of neurons in this region are not subject to the saturation issues associated with the extreme values, and the network tends to produce nearly linear responses.

Overall Impact on Network Behavior:

Due to the Gaussian initialization and the sigmoid activation, the network might operate in a regime where the transformations applied by the neurons resemble linear operations.

As a result, the deep neural network effectively acts as a linear classifier, and the capacity to capture non-linear relationships is limited.

In summary, normalizing the weights using a Gaussian distribution with mean 0 and standard deviation 1, along with the sigmoid activation function, can lead to a situation where the DNN behaves as a linear classifier. This linear behavior can limit the expressive power of the network, and more advanced techniques, such as different weight initialization strategies or alternative activation functions, might be explored to enhance the capacity of the network to capture non-linear patterns.