Deep Learning (DL)

Q. History of Deep Learning.

Ans : The history of deep learning is a fascinating journey that spans several decades and has seen significant breakthroughs in artificial intelligence and machine learning. Here is an overview of the key milestones and developments in the history of deep learning:

1. Perceptrons (1950s-1960s):

   - The concept of artificial neural networks (ANNs) was first introduced by Warren McCulloch and Walter Pitts in the 1940s, but it gained more attention in the 1950s and 1960s with the development of the perceptron.

   - The perceptron was a single-layer neural network designed for binary classification tasks. It had limitations in solving complex problems, which led to the first "AI winter."

2. Backpropagation (1970s-1980s):

   - In the 1970s, the backpropagation algorithm was developed, allowing for the training of multi-layer neural networks.

   - Despite this advancement, neural networks faced challenges in training deep networks due to the vanishing gradient problem.

3. Neural Network Renaissance (2000s):

   - In the early 2000s, there was a resurgence of interest in neural networks, driven by the availability of more data and computational power.

   - Researchers explored various techniques to train deeper networks, including unsupervised pre-training and initialization methods.

4. Deep Learning Breakthroughs (2010s):

   - The 2010s marked a significant turning point for deep learning:

     - The introduction of deep neural networks, such as convolutional neural networks (CNNs) for image recognition and recurrent neural networks (RNNs) for sequence modeling.

     - The ImageNet Large Scale Visual Recognition Challenge in 2012, won by a deep CNN, demonstrated the power of deep learning in computer vision.

     - Advances in GPU technology provided the necessary computational resources for training deep networks.

     - The development of deep learning frameworks like TensorFlow and PyTorch made it easier to build and train deep models.

5. Application Domains (2010s-present):

- Deep learning has seen widespread adoption in various application domains, including natural language processing (NLP), speech recognition, recommendation systems, autonomous vehicles, and more.

- Breakthroughs in NLP, such as the introduction of transformer-based models like BERT and GPT-3, revolutionized language understanding and generation tasks.

6. Ongoing Research (2020s and beyond):

- Deep learning continues to evolve with ongoing research in areas like reinforcement learning, self-supervised learning, and multi-modal AI.

- Efforts are being made to address the challenges of ethical AI, fairness, and interpretability.

Throughout its history, deep learning has been shaped by a combination of theoretical advancements, algorithmic improvements, increased data availability, and more powerful hardware. It has transformed many industries and has become an integral part of modern AI research and applications. The field of deep learning is likely to continue evolving with new breakthroughs and applications in the coming years.

Q. Thresholding in Deep Learning

Thresholding logic in deep learning refers to the application of a threshold function to determine the activation state of a neuron or unit in a neural network. It is a fundamental concept in artificial neural networks, including deep neural networks (DNNs).

Here's how thresholding logic works in the context of deep learning:

1. **Neuron Activation**: In artificial neural networks, each neuron or unit takes inputs, performs a weighted sum of those inputs, and applies an activation function to produce an output or activation.

2. **Thresholding Function**: The thresholding logic is implemented through an activation function that compares the weighted sum of inputs to a certain threshold value. If the weighted sum exceeds the threshold, the neuron is said to be "activated" or "fired," and it produces a non-zero output. Otherwise, it remains in an "inactive" state and produces an output of zero.

3. **Mathematical Representation**: The thresholding logic can be mathematically represented as follows:

- If $\Sigma(wi * xi) \geq \theta$, where Σ represents the weighted sum of inputs, xi are inputs, wi are weights, and θ is the threshold, then the output of the neuron is set to 1 (activated).

- If $\Sigma(wi * xi) < \theta$, then the output of the neuron is set to 0 (inactive).

4. **Activation Function Types**: There are different types of activation functions used in deep learning, and some of them implement thresholding logic to varying degrees. For example:

- The step function implements a strict threshold, where any input sum greater than or equal to the threshold results in an output of 1, and anything below it results in an output of 0.

- The sigmoid activation function produces a smooth transition from 0 to 1 as the input sum varies, but it can be thought of as a type of thresholding function.

5. **Role in Deep Learning**: In deep neural networks, thresholding logic is used to introduce non-linearity into the network's activations. This non-linearity enables the network to learn complex relationships and patterns in the data. Without activation functions implementing thresholding logic, a deep neural network would essentially reduce to a linear model, which is limited in its ability to capture intricate features.

6. **Common Activation Functions**: Some common activation functions that incorporate thresholding logic include the step function, sigmoid function, rectified linear unit (ReLU), and variants like Leaky ReLU. These functions allow for different degrees of thresholding and non-linearity, which impact how neural networks learn and generalize from data.

In summary, thresholding logic in deep learning is a crucial component that introduces non-linearity and allows neural networks to learn and represent complex patterns in data. Activation functions implementing thresholding play a significant role in the success of deep learning models. Different activation functions offer varying degrees of thresholding and non-linearity, which can affect the performance and training dynamics of neural networks.

Q. Perceptrons

Perceptrons are foundational building blocks in the history of neural networks and deep learning. While they are relatively simple compared to modern deep learning models, understanding perceptrons provides valuable insights into the development of more complex neural network architectures.

Here are key points about perceptrons in the context of deep learning:

1. **Introduction**: Perceptrons were introduced by Frank Rosenblatt in 1957. They are one of the earliest forms of artificial neural networks.

2. **Architecture**: A perceptron consists of one or more binary inputs, weights assigned to each input, a weighted sum of inputs, and a threshold function.

3. **Activation Function**: The threshold function, often referred to as a step function, compares the weighted sum of inputs to a threshold. If the sum is greater than or equal to the threshold, the perceptron outputs 1 (activated); otherwise, it outputs 0 (not activated).

4. **Mathematical Representation**: The output of a perceptron can be mathematically represented as follows:

- Output (y) = 1 if $\Sigma(wi * xi) \geq \theta$
- Output (y) = 0 if $\Sigma(wi * xi) < \theta$

Here, xi represents the input, wi represents the weight, and θ represents the threshold.

5. **Binary Classification**: Perceptrons are particularly suited for binary classification tasks. They can learn to separate two classes by adjusting their weights during training.

6. **Limitations**: Perceptrons have significant limitations. They can only solve linearly separable problems, which means they cannot learn to solve problems where a linear decision boundary is insufficient. This limitation led to the development of more complex neural network architectures.

7. **Single-Layer Networks**: Perceptrons are single-layer neural networks because they have no hidden layers. They can only represent linear transformations of the input data.

8. **The Perceptron Learning Rule**: To train a perceptron, a learning rule called the "perceptron learning rule" is used. It adjusts the weights based on errors in classification, effectively updating the parameters to better separate the classes.

9. **Role in Deep Learning**: While perceptrons are not used directly in modern deep learning, they played a pivotal role in the history of neural networks. The limitations of perceptrons led to the development of multi-layer neural networks, which can represent more complex, non-linear functions.

10. **Multi-Layer Networks**: Deep learning models, such as feedforward neural networks, convolutional neural networks (CNNs), and recurrent neural networks (RNNs), are composed of multiple layers of neurons, enabling them to capture complex patterns and relationships in data.

In summary, perceptrons are the foundational elements of neural networks, but they have limitations that led to the development of more advanced deep learning models. While perceptrons are not used in isolation in modern deep learning, the principles behind them, such as weighted inputs and thresholding, are essential components in the more complex neural networks that make up the field of deep learning.

Q. Sigmoid Neuron

A sigmoid neuron, often referred to as a sigmoid activation function, is a type of artificial neuron used in artificial neural networks and machine learning. It's named after its characteristic S-shaped (sigmoid) curve, which is used as the activation function to introduce non-linearity in the model. The most common sigmoid function used in neural networks is the logistic sigmoid function.

Here is an explanation of the sigmoid neuron and its key characteristics:

1. **Sigmoid Activation Function**: The sigmoid neuron uses a sigmoid activation function, typically represented by the logistic sigmoid function, which is defined as:

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

In this equation, z is the weighted sum of inputs and the bias term in the neuron.

2. **Output Range**: The sigmoid function maps its input to an output in the range (0, 1). This makes it suitable for applications where the output is interpreted as a probability, such as binary classification problems.

3. **Non-Linearity**: The sigmoid function introduces non-linearity into the model, allowing neural networks to learn complex relationships and perform tasks that involve classification or decision-making.

4. **Derivative**: The derivative of the sigmoid function can be calculated as:

$$\frac{d\sigma(z)}{dz} = \sigma(z) \cdot (1 - \sigma(z))$$

This derivative is essential for training neural networks using gradient descent-based optimization algorithms like backpropagation.

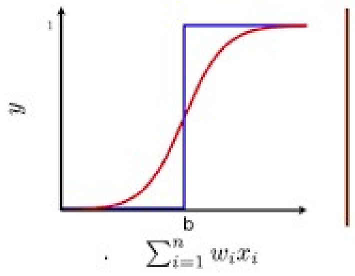
5. **Historical Significance**: Sigmoid neurons were widely used in the early days of neural network research. However, they have largely been replaced by other activation functions like the rectified linear unit (ReLU) in recent years due to some advantages of ReLU, such as faster convergence during training.

6. **Vanishing Gradient**: One of the drawbacks of sigmoid neurons is the vanishing gradient problem. When gradients become very small during backpropagation, it can slow down or impede the training process, especially in deep neural networks. This problem has led to the preference for ReLU and its variants in deep learning.

7. **Use in Binary Classification**: Sigmoid neurons are commonly used in the output layer of neural networks for binary classification tasks. The output value can be interpreted as the probability that a given input belongs to one of the two classes.

8. **Logistic Regression**: Sigmoid activation is also a key component of logistic regression, a statistical method for binary classification. In this context, the sigmoid function models the probability of an event occurring.

While sigmoid neurons are less popular in hidden layers of deep neural networks today, they still have their applications in certain scenarios, particularly in output layers where probability-like outputs are needed.



$$y = \frac{1}{1+e^{-(w^T x+b)}}$$

Q. momentum based Gradient Decent

Momentum-Based Gradient Optimizer: Introduction

- **Definition:**

  - Momentum-based gradient optimization is a technique used in machine learning to enhance the training of models.

  - It's an improvement over basic gradient descent methods.

- **Objective:**

  - Minimize the training loss and help the model converge faster to the optimal solution.

- **Key Concept:**

  - Inspired by physics, momentum introduces a "velocity" term to the optimization process.

  - It helps overcome challenges like oscillations and slow convergence associated with basic gradient descent.

- **Momentum in Physics Analogy:**

  - Think of a ball rolling down a hill.

  - Momentum is the speed and direction of the ball, influencing how it moves.

  - Similarly, in optimization, momentum affects the model's parameter updates.

- **Algorithm:**

  - In each iteration, the optimizer not only considers the current gradient but also takes into account the accumulated past gradients.

  - This accumulated information helps the optimizer "build momentum" in the direction of consistent gradients.

- **Equation:**

  - Momentum is mathematically represented as a moving average of past gradients.

  - The current update is a combination of the current gradient and a fraction of the past accumulated gradient.

- **Advantages:**

  - **Faster Convergence:**

    - Momentum helps the optimizer move more quickly in the direction of consistent gradients.

  - **Escape Local Minima:**

    - It aids in escaping local minima, contributing to a better chance of finding the global minimum.

- **Implementation:**

  - Commonly used in conjunction with other optimization algorithms like stochastic gradient descent (SGD).

  - Popular libraries like TensorFlow and PyTorch include momentum-based optimizers.

- **Adjustable Hyperparameter:**

  - The momentum term is a hyperparameter that can be tuned based on the characteristics of the dataset and the model.

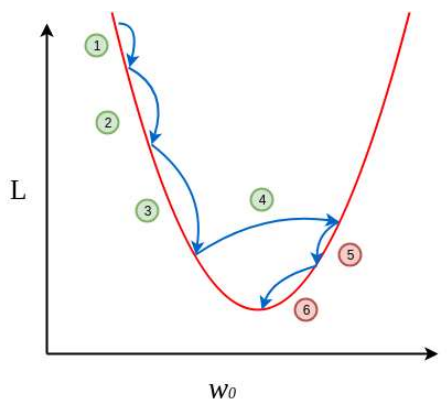
- **Conclusion:**

  - Momentum-based optimization is a valuable tool for training machine learning models, offering improved convergence speed and the ability to navigate through challenging optimization landscapes.

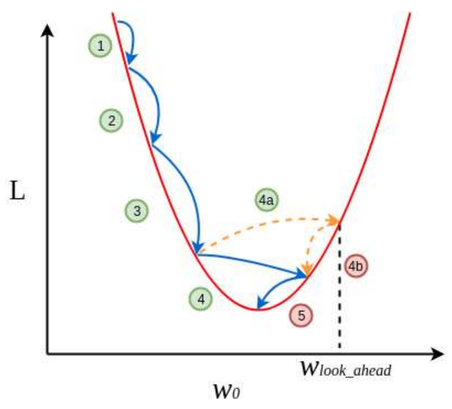
In summary, momentum-based gradient optimization introduces a physics-inspired concept to enhance the efficiency of model training. It helps overcome challenges associated with basic gradient descent methods, contributing to faster convergence and improved optimization.

Q. Nesterov Accelerated Gradient Decent

NAG resolves this problem by adding a look ahead term in our equation. The intuition behind NAG can be summarized as 'look before you leap'. Let us try to understand this through an example.



(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

🟢 $\Longrightarrow \dfrac{\partial L}{\partial w_0} = \dfrac{Negative(-)}{Positive(+)}$ 🔴 $\Longrightarrow \dfrac{\partial L}{\partial w_0} = \dfrac{Negative(-)}{Negative(-)}$

As can see, in the momentum-based gradient, the steps become larger and larger due to the accumulated momentum, and then we overshoot at the 4th step. We then have to take steps in the opposite direction to reach the minimum point.
However, the update in NAG happens in two steps. First, a partial step to reach the look-ahead point, and then the final update. We calculate the gradient at the look-ahead point and then use it to calculate the final update. If the gradient at the look-ahead point is negative, our final update will be smaller than that of a regular momentum-based gradient. Like in the above example, the updates of NAG are similar to that of the momentum-based gradient for the first three steps because the gradient at that point and the look-ahead point are positive. But at step 4, the gradient of the look-ahead point is negative.
In NAG, the first partial update 4a will be used to go to the look-ahead point and then the gradient will be calculated at that point without updating the parameters. Since the gradient at step 4b is negative, the overall update will be smaller than the momentum-based gradient descent.
We can see in the above example that the momentum-based gradient descent takes six steps to reach the minimum point, while NAG takes only five steps.
This looking ahead helps NAG to converge to the minimum points in fewer steps and reduce the chances of overshooting.

How NAG actually works?

We saw how NAG solves the problem of overshooting by 'looking ahead'. Let us see how this is calculated and the actual math behind it.
Update rule for gradient descent:
$w_{t+1} = w_t - \eta \nabla wt$
In this equation, the weight (W) is updated in each iteration. η is the learning rate, and ∇wt is the gradient.
Update rule for momentum-based gradient descent:
In this, momentum is added to the conventional gradient descent equation. The update

equation is
$w_{t+1} = w_t - update_t$
$update_t$ is calculated by:
$update_t = \gamma \cdot update_{t-1} + \eta\nabla w_t$

$$\begin{aligned}
update_0 &= 0 \\
update_1 &= \gamma \cdot update_0 + \eta\nabla w_1 = \eta\nabla w_1 \\
update_2 &= \gamma \cdot update_1 + \eta\nabla w_2 = \gamma \cdot \eta\nabla w_1 + \eta\nabla w_2 \\
update_3 &= \gamma \cdot update_2 + \eta\nabla w_3 = \gamma(\gamma \cdot \eta\nabla w_1 + \eta\nabla w_2) + \eta\nabla w_3 \\
&= \gamma \cdot update_2 + \eta\nabla w_3 = \gamma^2 \cdot \eta\nabla w_1 + \gamma \cdot \eta\nabla w_2 + \eta\nabla w_3 \\
update_4 &= \gamma \cdot update_3 + \eta\nabla w_4 = \gamma^3 \cdot \eta\nabla w_1 + \gamma^2 \cdot \eta\nabla w_2 + \gamma \cdot \eta\nabla w_3 + \eta\nabla w_4 \\
&\vdots \\
update_t &= \gamma \cdot update_{t-1} + \eta\nabla w_t = \gamma^{t-1} \cdot \eta\nabla w_1 + \gamma^{t-2} \cdot \eta\nabla w_1 + \ldots + \eta\nabla w_t
\end{aligned}$$

This is how the gradient of all the previous updates is added to the current update.
Update rule for NAG:
$w_{t+1} = w_t - update_t$
While calculating the $update_t$, We will include the look ahead gradient (∇w_{look_ahead}).
$update_t = \gamma \cdot update_{t-1} + \eta\nabla w_{look\_ahead}$
∇w_{look_ahead} is calculated by:
$w_{look\_ahead} = w_t - \gamma \cdot update_{t-1}$

Stochastic Gradient Descent (SGD):

Stochastic Gradient Descent (SGD) is a variant of the Gradient Descent algorithm that is used for optimizing machine learning models. It addresses the computational inefficiency of traditional Gradient Descent methods when dealing with large datasets in machine learning projects.
In SGD, instead of using the entire dataset for each iteration, only a single random training example (or a small batch) is selected to calculate the gradient and update the model parameters. This random selection introduces randomness into the optimization process, hence the term "stochastic" in stochastic Gradient Descent

The advantage of using SGD is its computational efficiency, especially when dealing with large datasets. By using a single example or a small batch, the computational cost per iteration is significantly reduced compared to traditional Gradient Descent methods that require processing the entire dataset.

Stochastic Gradient Descent Algorithm

- **Initialization**: Randomly initialize the parameters of the model.
- **Set Parameters**: Determine the number of iterations and the learning rate (alpha) for updating the parameters.
- **Stochastic Gradient Descent Loop**: Repeat the following steps until the model converges or reaches the maximum number of iterations:
  a. Shuffle the training dataset to introduce randomness.

  b. Iterate over each training example (or a small batch) in the shuffled order.

c. Compute the gradient of the cost function with respect to the model parameters using the current training example (or batch).

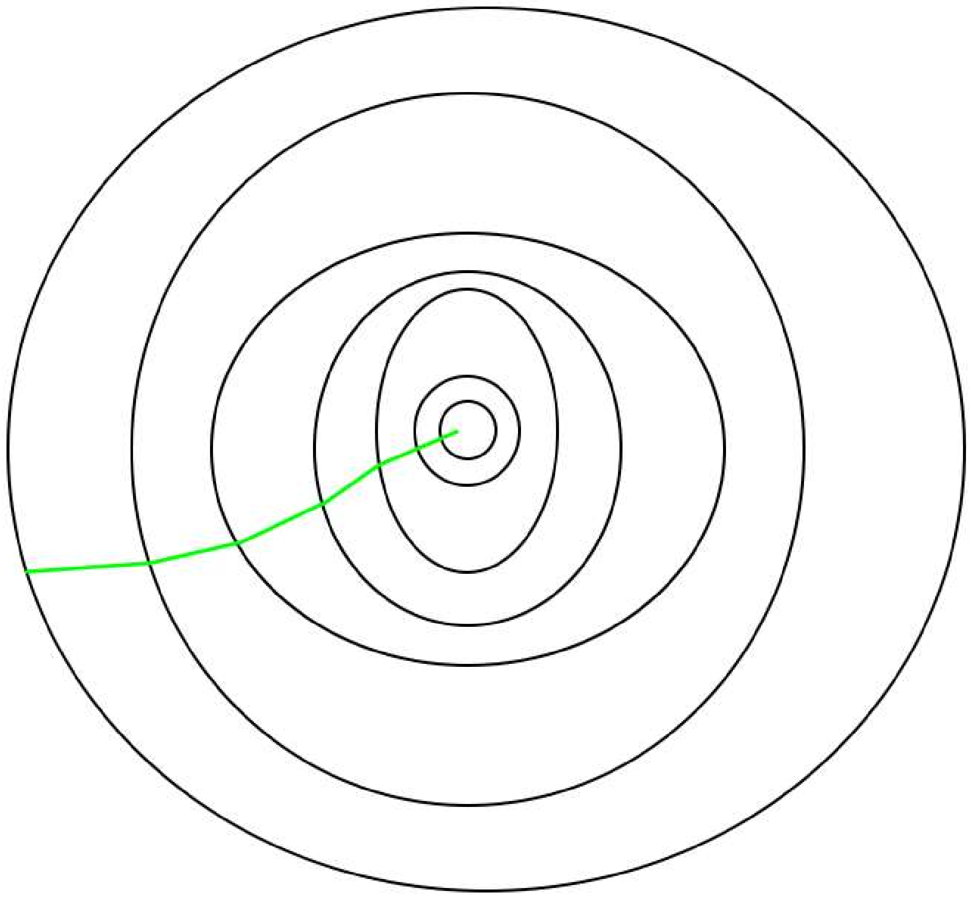
d. Update the model parameters by taking a step in the direction of the negative gradient, scaled by the learning rate.

e. Evaluate the convergence criteria, such as the difference in the cost function between iterations of the gradient.

- **Return Optimized Parameters**: Once the convergence criteria are met or the maximum number of iterations is reached, return the optimized model parameters.

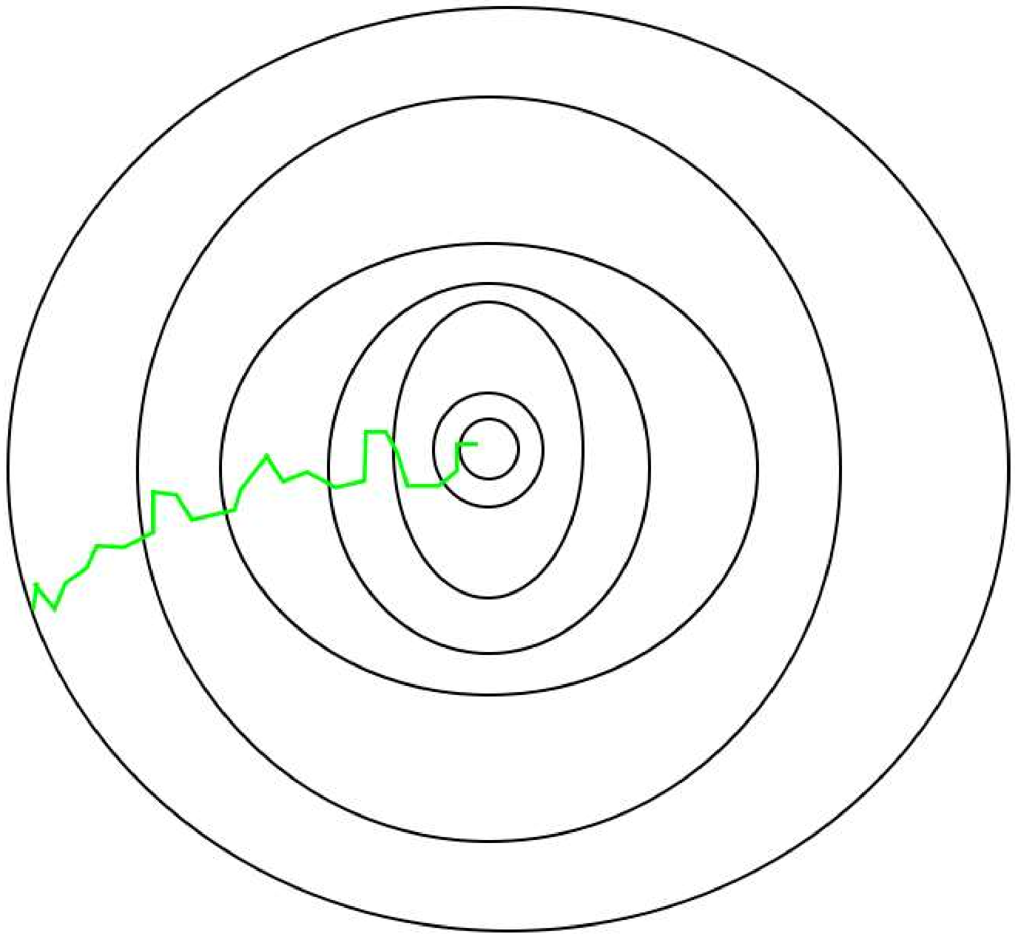
In SGD, since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than your typical Gradient Descent algorithm. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minimum and with a significantly shorter training time.

The path taken by Batch Gradient Descent is shown below:



Batch gradient optimization path

A path taken by Stochastic Gradient Descent looks as follows –



stochastic gradient optimization path

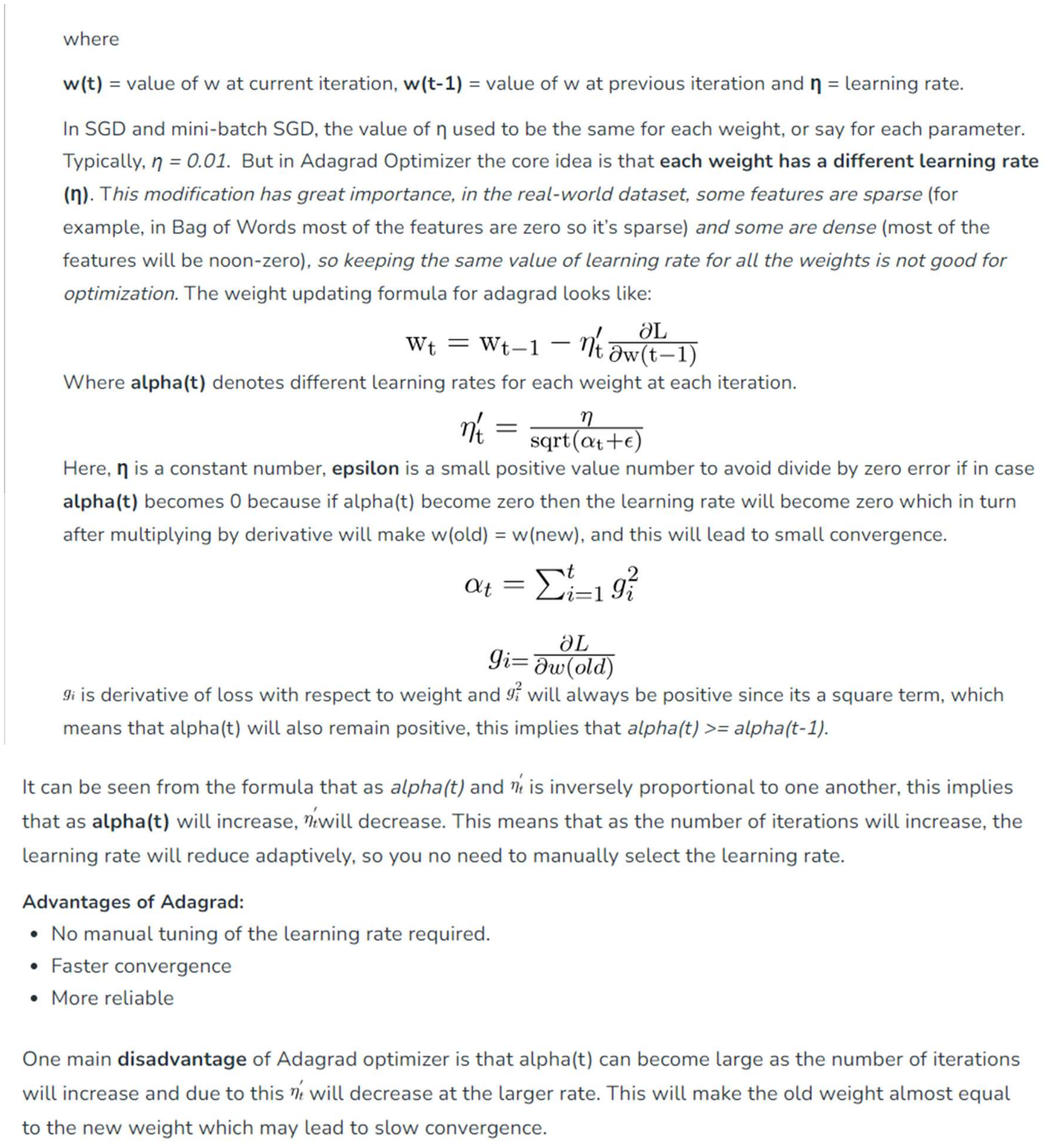
One thing to be noted is that, as SGD is generally noisier than typical Gradient Descent, it usually took a higher number of iterations to reach the minima, because of the randomness in its descent. Even though it requires a higher number of iterations to reach the minima than typical Gradient Descent, it is still computationally much less expensive than typical Gradient Descent. Hence, in most scenarios, SGD is preferred over Batch Gradient Descent for optimizing a learning algorithm.

Adagrad stands for **Adaptive Gradient Optimizer**. There were optimizers like Gradient Descent, Stochastic Gradient Descent, mini-batch SGD, all were used to reduce the loss function with respect to the weights. The weight updating formula is as follows:

$$(\mathrm{w})_{\text{new}} = (\mathrm{w})_{\text{old}} - \eta \frac{\partial \mathrm{L}}{\mathrm{dw}(\text{ old })}$$

Based on iterations, this formula can be written as:

$$w_t = w_{t-1} - \eta \frac{\partial L}{\partial w(t-1)}$$



where

w(t) = value of w at current iteration, **w(t-1)** = value of w at previous iteration and **η** = learning rate.

In SGD and mini-batch SGD, the value of η used to be the same for each weight, or say for each parameter. Typically, *η = 0.01*. But in Adagrad Optimizer the core idea is that **each weight has a different learning rate (η)**. T*his modification has great importance, in the real-world dataset, some features are sparse* (for example, in Bag of Words most of the features are zero so it's sparse) *and some are dense* (most of the features will be noon-zero)*, so keeping the same value of learning rate for all the weights is not good for optimization.* The weight updating formula for adagrad looks like:

$$\mathrm{w_t} = \mathrm{w_{t-1}} - \eta'_{\mathrm{t}} \frac{\partial \mathrm{L}}{\partial \mathrm{w(t-1)}}$$

Where **alpha(t)** denotes different learning rates for each weight at each iteration.

$$\eta'_{\mathrm{t}} = \frac{\eta}{\mathrm{sqrt}(\alpha_{\mathrm{t}}+\epsilon)}$$

Here, **η** is a constant number, **epsilon** is a small positive value number to avoid divide by zero error if in case **alpha(t)** becomes 0 because if alpha(t) become zero then the learning rate will become zero which in turn after multiplying by derivative will make w(old) = w(new), and this will lead to small convergence.

$$\alpha_t = \sum_{i=1}^{t} g_i^2$$

$$g_i = \frac{\partial L}{\partial w(old)}$$

g_i is derivative of loss with respect to weight and g_i^2 will always be positive since its a square term, which means that alpha(t) will also remain positive, this implies that *alpha(t) >= alpha(t-1).*

It can be seen from the formula that as *alpha(t)* and η'_t is inversely proportional to one another, this implies that as **alpha(t)** will increase, η'_twill decrease. This means that as the number of iterations will increase, the learning rate will reduce adaptively, so you no need to manually select the learning rate.

Advantages of Adagrad:

- No manual tuning of the learning rate required.
- Faster convergence
- More reliable

One main **disadvantage** of Adagrad optimizer is that alpha(t) can become large as the number of iterations will increase and due to this η'_t will decrease at the larger rate. This will make the old weight almost equal to the new weight which may lead to slow convergence.

Bias is one type of error that occurs due to wrong assumptions about data such as assuming data is linear when in reality, data follows a complex function. On the other hand, variance gets introduced with high sensitivity to variations in training data. This also is one type of error since we want to make our model robust against noise. There are two types of error in machine learning. Reducible error and Irreducible error. Bias and Variance come under reducible error.

What is Bias?

Bias is simply defined as the inability of the model because of that there is some difference or error occurring between the model's predicted value and the actual value. These differences between actual or expected values and the predicted values are known as error or bias error or error due to bias. Bias is a systematic error that occurs due to wrong assumptions in the machine learning process.

Let be the true value of a parameter, and let be an estimator

of based on a sample of data. Then, the bias of the estimator is given by:

where is the expected value of the estimator . It is the measurement of the model that how well it fits the data.

- **Low Bias:** Low bias value means fewer assumptions are taken to build the target function. In this case, the model will closely match the training dataset.
- **High Bias:** High bias value means more assumptions are taken to build the target function. In this case, the model will not match the training dataset closely.

The high-bias model will not be able to capture the dataset trend. It is considered as the underfitting model which has a high error rate. It is due to a very simplified algorithm.
For example, a linear regression model may have a high bias if the data has a non-linear relationship.

Ways to reduce high bias in Machine Learning:

- **Use a more complex model:** One of the main reasons for high bias is the very simplified model. it will not be able to capture the complexity of the data. In such cases, we can make our mode more complex by increasing the number of hidden layers in the case of a deep neural network. Or we can use a more complex model like Polynomial regression for non-linear datasets, CNN for image processing, and RNN for sequence learning.

- **Increase the number of features:** By adding more features to train the dataset will increase the complexity of the model. And improve its ability to capture the underlying patterns in the data.
- **Reduce Regularization of the model:** Regularization techniques such as L1 or L2 regularization can help to prevent overfitting and improve the generalization ability of the model. if the model has a high bias, reducing the strength of regularization or removing it altogether can help to improve its performance.
- **Increase the size of the training data:** Increasing the size of the training data can help to reduce bias by providing the model with more examples to learn from the dataset.

What is Variance?

Variance is the measure of spread in data from its mean position. In machine learning variance is the amount by which the performance of a predictive model changes when it is trained on different subsets of the training data. More specifically, variance is the variability of the model that how much it is sensitive to another subset of the training dataset. i.e. how much it can adjust on the new subset of the training dataset.

Let Y be the actual values of the target variable, and be the predicted values of the target variable. Then the variance of a model can be measured as the expected value of the square of the difference between predicted values and the expected value of the predicted values.

where is the expected value of the predicted values. Here expected value is averaged over all the training data.

Variance errors are either low or high-variance errors.

- **Low variance:** Low variance means that the model is less sensitive to changes in the training data and can produce consistent estimates of the target function with different subsets of data from the same distribution. This is the case of underfitting when the model fails to generalize on both training and test data.
- **High variance:** High variance means that the model is very sensitive to changes in the training data and can result in significant changes in the estimate of the target function when

trained on different subsets of data from the same distribution. This is the case of overfitting when the model performs well on the training data but poorly on new, unseen test data. It fits the training data too closely that it fails on the new training dataset.

Ways to Reduce the reduce Variance in Machine Learning:

- **Cross-validation:** By splitting the data into training and testing sets multiple times, cross-validation can help identify if a model is overfitting or underfitting and can be used to tune hyperparameters to reduce variance.
- **Feature selection:** By choosing the only relevant feature will decrease the model's complexity. and it can reduce the variance error.
- **Regularization:** We can use L1 or L2 regularization to reduce variance in machine learning models
- **Ensemble methods:** It will combine multiple models to improve generalization performance. Bagging, boosting, and stacking are common ensemble methods that can help reduce variance and improve generalization performance.
- **Simplifying the model:** Reducing the complexity of the model, such as decreasing the number of parameters or layers in a neural network, can also help reduce variance and improve generalization performance.
- **Early stopping:** Early stopping is a technique used to prevent overfitting by stopping the training of the deep learning model when the performance on the validation set stops improving.

Underfitting in Machine Learning

A statistical model or a machine learning algorithm is said to have underfitting when a model is too simple to capture data complexities. It represents the inability of the model to learn the training data effectively result in poor performance both on the training and testing data. In simple terms, an underfit model's are inaccurate, especially when applied to new, unseen examples. It mainly happens when we uses very simple model with overly simplified assumptions. To address underfitting problem of the model, we need to use more complex models, with enhanced feature representation, and less regularization.

Note: The underfitting model has High bias and low variance.

Reasons for Underfitting

1. The model is too simple, So it may be not capable to represent the complexities in the data.
2. The input features which is used to train the model is not the adequate representations of underlying factors influencing the target variable.
3. The size of the training dataset used is not enough.
4. Excessive regularization are used to prevent the overfitting, which constraint the model to capture the data well.
5. Features are not scaled.

Techniques to Reduce Underfitting

1. Increase model complexity.
2. Increase the number of features, performing feature engineering.
3. Remove noise from the data.
4. Increase the number of epochs or increase the duration of training to get better results.

Overfitting in Machine Learning

A statistical model is said to be overfitted when the model does not make accurate predictions on testing data. When a model gets trained with so much data, it starts learning from the noise and inaccurate data entries in our data set. And when testing with test data results in High variance. Then the model does not categorize the data correctly, because of too many details and noise. The causes of overfitting are the non-parametric and non-linear methods because these types of machine learning algorithms have more freedom in building the model based on the dataset and therefore they can really build unrealistic models. A solution to avoid overfitting is using a linear algorithm if we have linear data or using the parameters like the maximal depth if we are using decision trees.

In a nutshell, Overfitting is a problem where the evaluation of machine learning algorithms on training data is different from unseen data.

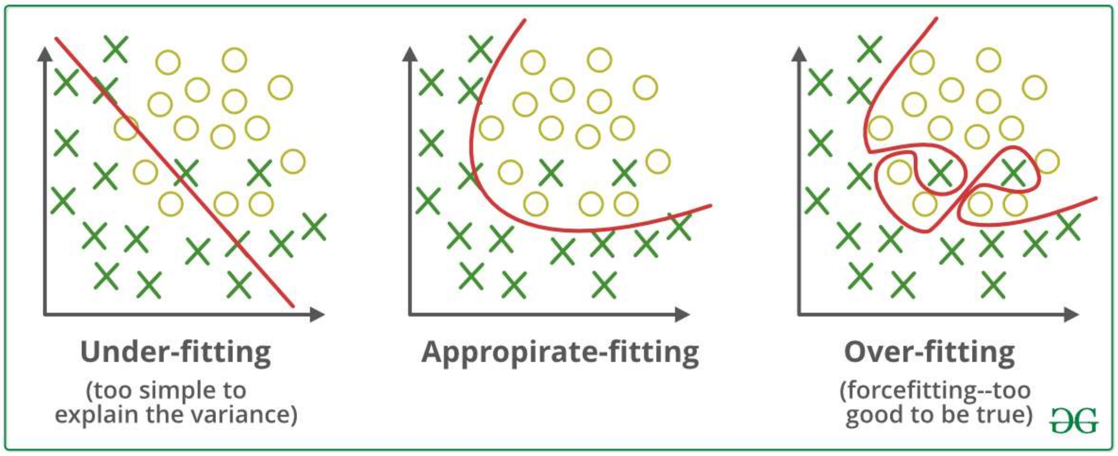
Reasons for Overfitting:

1. High variance and low bias.
2. The model is too complex.
3. The size of the training data.

Techniques to Reduce Overfitting

1. Increase training data.
2. Reduce model complexity.

3. Early stopping during the training phase (have an eye over the loss over the training period as soon as loss begins to increase stop training).
4. Ridge Regularization and Lasso Regularization.
5. Use dropout for neural networks to tackle overfitting.


Under-fitting
(too simple to explain the variance)
Appropirate-fitting
Over-fitting
(forcefitting--too good to be true)

What is L1 regularization?

L1 regularization, also known as Lasso regularization, is a machine-learning strategy that inhibits overfitting by introducing a penalty term into the model's loss function based on the absolute values of the model's parameters. L1 regularization seeks to reduce some model parameters toward zero in order to lower the number of non-zero parameters in the model (sparse model).

L1 regularization is particularly useful when working with high-dimensional data since it enables one to choose a subset of the most important attributes. This lessens the risk of overfitting and also makes the model easier to understand. The size of a penalty term is controlled by the hyperparameter lambda, which regulates the L1 regularization's regularization strength. As lambda rises, more parameters will be lowered to zero, improving regularization.

What is L2 regularization?

L2 regularization, also known as Ridge regularization, is a machine learning technique that avoids overfitting by introducing a penalty term into the model's loss function based on the squares of the model's parameters. The goal of L2 regularization is to keep the model's parameter sizes short and prevent oversizing.

In order to achieve L2 regularization, a term that is proportionate to the squares of the model's parameters is added to the loss function. This word works as a limiter on the parameters' size, preventing them from growing out of control. A hyperparameter called lambda that controls the regularization's intensity also controls the size of the penalty term. The parameters will be smaller and the regularization will be stronger the greater the lambda.

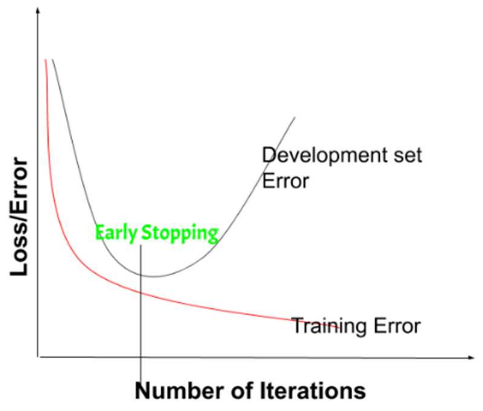
Difference between L1 & L2 regularization

L1 Regularization	**L2 Regularization**
The penalty term is based on the absolute values of the model's parameters.	The penalty term is based on the squares of the model's parameters.
Produces sparse solutions (some parameters are shrunk towards zero).	Produces non-sparse solutions (all parameters are used by the model).
Sensitive to outliers.	Robust to outliers.
Selects a subset of the most important features.	All features are used by the model.
Optimization is non-convex.	Optimization is convex.
The penalty term is less sensitive to correlated features.	The penalty term is more sensitive to correlated features.
Useful when dealing with high-dimensional data with many correlated features.	Useful when dealing with high-dimensional data with many correlated features and when the goal is to have a less complex model.
Also known as Lasso regularization.	Also known as Ridge regularization.

What is Early Stopping?

In Regularization by Early Stopping, we stop training the model when the performance on the validation set is getting worse- increasing loss decreasing accuracy, or poorer scores of the scoring metric. By plotting the error on the training dataset and the validation dataset together, both the errors decrease with a number of iterations until the point where the model starts to overfit. After this point, the training error still decreases but the validation error increases.

So, even if training is continued after this point, early stopping essentially returns the set of parameters that were used at this point and so is equivalent to stopping training at that point. So, the final parameters returned will enable the model to have low variance and better generalization. The model at the time the training is stopped will have a better generalization performance than the model with the least training error.



on the validation set is getting worse- increasing loss or decreasing accuracy or poorer scores

Early stopping can be thought of as **implicit regularization**, contrary to regularization via weight decay. This method is also efficient since it requires less amount of training data, which is not always available. Due to this fact, early stopping requires lesser time for training compared to other regularization methods. Repeating the early stopping process many times may result in the model overfitting the validation dataset, just as similar as overfitting occurs in the case of training data.

The number of iterations(i.e. epoch) taken to train the model can be considered a **hyperparameter**. Then the model has to find an optimum value for this hyperparameter (by hyperparameter tuning) for the best performance of the learning model.

Tip: *The downside of early stopping are as follows:*
*By stopping early , we can't able to optimize Cost function(J) much for the training set. So, we use a different concept Known as **Orthogonalisation** is used.*

Benefits of Early Stopping:

- Helps in reducing overfitting
- It improves generalisation
- It requires less amount of training data
- Takes less time compared to other regularisation models
- It is simple to implement

Limitations of Early Stopping:

- If the model stops too early, there might be risk of underfitting
- It may not be beneficial for all types of models
- If validation set is not chosen properly, it may not lead to the most optimal stopping

To summarize, early stopping can be best used to prevent overfitting of the model, and saving resources. It would give best results if taken care of few things like – parameter tuning, preventing the model from overfitting, and ensuring that the model learns enough from the data.

Q. Data Augmentation

Audio Data Augmentation

1. **Noise injection**: add gaussian or random noise to the audio dataset to improve the model performance.
2. **Shifting**: shift audio left (fast forward) or right with random seconds.
3. **Changing the speed**: stretches times series by a fixed rate.
4. **Changing the pitch**: randomly change the pitch of the audio.

Text Data Augmentation

1. **Word or sentence shuffling**: randomly changing the position of a word or sentence.
2. **Word replacement**: replace words with synonyms.
3. **Syntax-tree manipulation**: paraphrase the sentence using the same word.

4. **Random word insertion**: inserts words at random.
5. **Random word deletion**: deletes words at random.

Image Augmentation

Learn more about image transformation and manipulation with hands-on exercises in our Image Processing with Python skill track.

1. **Geometric transformations**: randomly flip, crop, rotate, stretch, and zoom images. You need to be careful about applying multiple transformations on the same images, as this can reduce model performance.
2. **Color space transformations**: randomly change RGB color channels, contrast, and brightness.
3. **Kernel filters**: randomly change the sharpness or blurring of the image.
4. **Random erasing**: delete some part of the initial image.
5. **Mixing images**: blending and mixing multiple images.

Q. Parameter Sharing and Typing

We usually apply limitations or penalties to parameters in relation to a fixed region or point. **L**^{2} regularisation (or weight decay) penalises model parameters that deviate from a fixed value of zero, for example.
However, we may occasionally require alternative means of expressing our prior knowledge of appropriate model parameter values. We may not know exactly what values the parameters should take, but we do know that there should be some dependencies between the model parameters based on our knowledge of the domain and model architecture.
We frequently want to communicate the dependency that various parameters should be near to one another.

Parameter Typing

Two models are doing the same classification task (with the same set of classes), but their input distributions are somewhat different.

- We have model **A** has the parameters
- Another model **B** has the parameters

Parameter Sharing

The parameters of one model, trained as a classifier in a supervised paradigm, were regularised to be close to the parameters of another model, trained in an unsupervised paradigm, using this method (to capture the distribution of the observed input data). Many of the parameters in the classifier model might be linked with similar parameters in the unsupervised model thanks to the designs. While a parameter norm penalty is one technique to require sets of parameters to be equal, constraints are a more prevalent way to regularise parameters to be close to one another. Because we view the numerous models or model components as sharing a unique set of parameters, this form of regularisation is commonly referred to as parameter sharing. The fact that only a subset of the parameters (the unique set) needs to be retained in memory is a significant advantage of parameter sharing over regularising the parameters to be close (through a norm penalty). This can result in a large reduction in the memory footprint of certain models, such as the convolutional neural network.

Convolutional neural networks (CNNs) used in computer vision are by far the most widespread and extensive usage of parameter sharing. Many statistical features of natural images are translation insensitive. A shot of a cat, for example, can be translated one pixel to the right and still be a shot of a cat. By sharing parameters across several picture locations, CNNs take this property into account. Different locations in the input are computed with the same feature (a hidden unit with the same weights). This indicates that whether the cat appears in column ***i*** or column ***i + 1*** in the image, we can find it with the same cat detector.

CNN’s have been able to reduce the number of unique model parameters and raise network sizes greatly without requiring a comparable increase in training data thanks to parameter sharing. It’s still one of the best illustrations of how domain knowledge can be efficiently integrated into the network architecture.

It can be seen from the formula that as *alpha(t)* and η_t' is inversely proportional to one another, this implies that as **alpha(t)** will increase, η_t'will decrease. This means that as the number of iterations will increase, the learning rate will reduce adaptively, so you no need to manually select the learning rate.

Advantages of Adagrad:

- No manual tuning of the learning rate required.
- Faster convergence
- More reliable

One main **disadvantage** of Adagrad optimizer is that alpha(t) can become large as the number of iterations will increase and due to this η_t' will decrease at the larger rate. This will make the old weight almost equal to the new weight which may lead to slow convergence.

The pooling operation involves sliding a two-dimensional filter over each channel of feature map and summarising the features lying within the region covered by the filter.
For a feature map having dimensions **n_h x n_w x n_c**, the dimensions of output obtained after a pooling layer is

```
(nh - f + 1) / s x (nw - f + 1)/s x nc
```

where,

```
-> nh - height of feature map
-> nw - width of feature map
-> nc - number of channels in the feature map
-> f  - size of filter
-> s  - stride length
```

A common CNN model architecture is to have a number of convolution and pooling layers stacked one after the other.

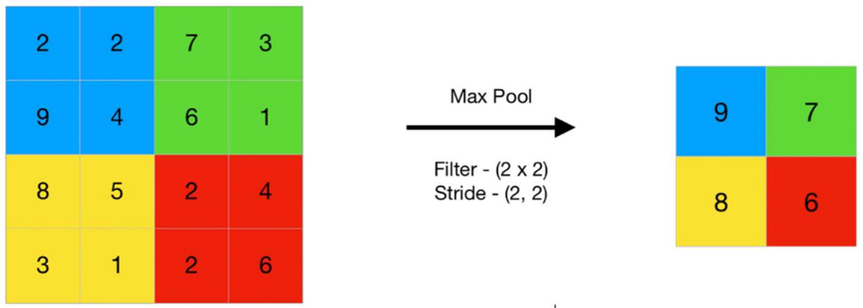
Why to use Pooling Layers?

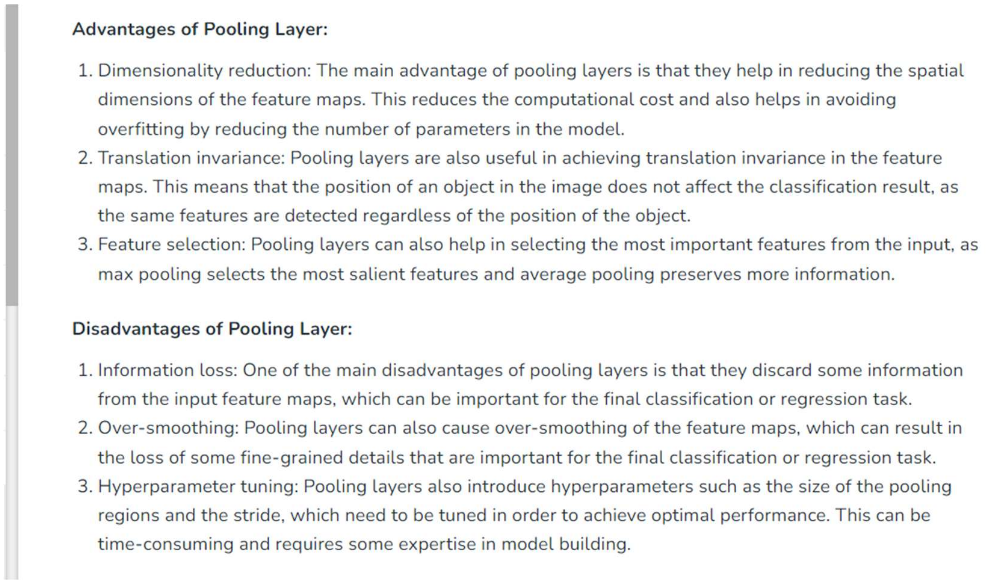
- Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn and the amount of computation performed in the network.
- The pooling layer summarises the features present in a region of the feature map generated by a convolution layer. So, further operations are performed on summarised features instead of precisely positioned features generated by the convolution layer. This makes the model more robust to variations

Types of Pooling Layers:

Max Pooling

1. Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.





Advantages of Pooling Layer:

1. Dimensionality reduction: The main advantage of pooling layers is that they help in reducing the spatial dimensions of the feature maps. This reduces the computational cost and also helps in avoiding overfitting by reducing the number of parameters in the model.
2. Translation invariance: Pooling layers are also useful in achieving translation invariance in the feature maps. This means that the position of an object in the image does not affect the classification result, as the same features are detected regardless of the position of the object.
3. Feature selection: Pooling layers can also help in selecting the most important features from the input, as max pooling selects the most salient features and average pooling preserves more information.

Disadvantages of Pooling Layer:

1. Information loss: One of the main disadvantages of pooling layers is that they discard some information from the input feature maps, which can be important for the final classification or regression task.
2. Over-smoothing: Pooling layers can also cause over-smoothing of the feature maps, which can result in the loss of some fine-grained details that are important for the final classification or regression task.
3. Hyperparameter tuning: Pooling layers also introduce hyperparameters such as the size of the pooling regions and the stride, which need to be tuned in order to achieve optimal performance. This can be time-consuming and requires some expertise in model building.

What is LeNet 5?

LeNet is a convolutional neural network that Yann LeCun introduced in 1989. LeNet is a common term for **LeNet-5**, a simple convolutional neural network.

The LeNet-5 signifies CNN's emergence and outlines its core components. However, it was not popular at the time due to a lack of hardware, especially GPU (Graphics Process Unit, a specialised electronic circuit designed to change memory to accelerate the creation of images during a buffer intended for output to a show device) and alternative algorithms, like SVM, which could perform effects similar to or even better than those of the LeNet.

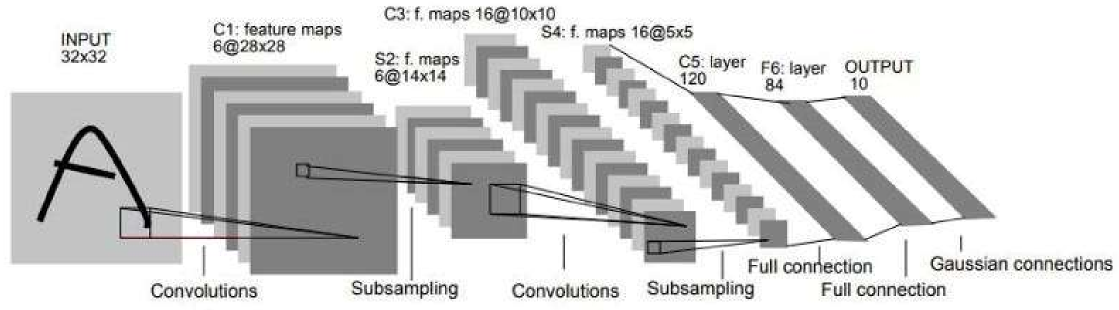
Features of LeNet-5

- Every convolutional layer includes three parts: convolution, pooling, and nonlinear activation functions

- Using convolution to extract spatial features (Convolution was called receptive fields originally)
- **The average pooling layer** is used for subsampling.
- ‘**tanh**’ is used as the activation function
- Using **Multi-Layered Perceptron** or **Fully Connected Layers** as the last classifier
- The sparse connection between layers reduces the complexity of computation

Architecture

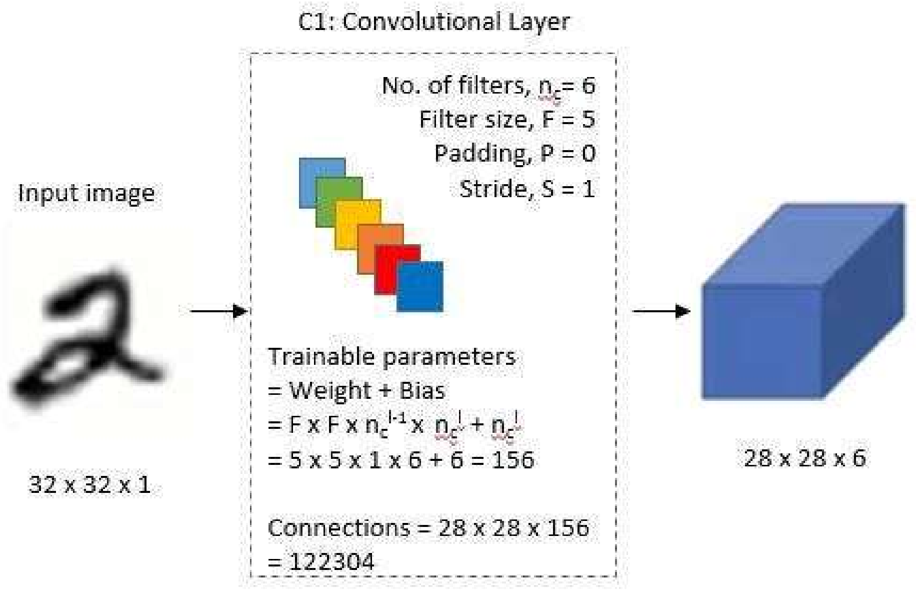
The LeNet-5 CNN architecture has seven layers. Three convolutional layers, two subsampling layers, and two fully linked layers make up the layer composition.



LeNet-5 Architecture

First Layer

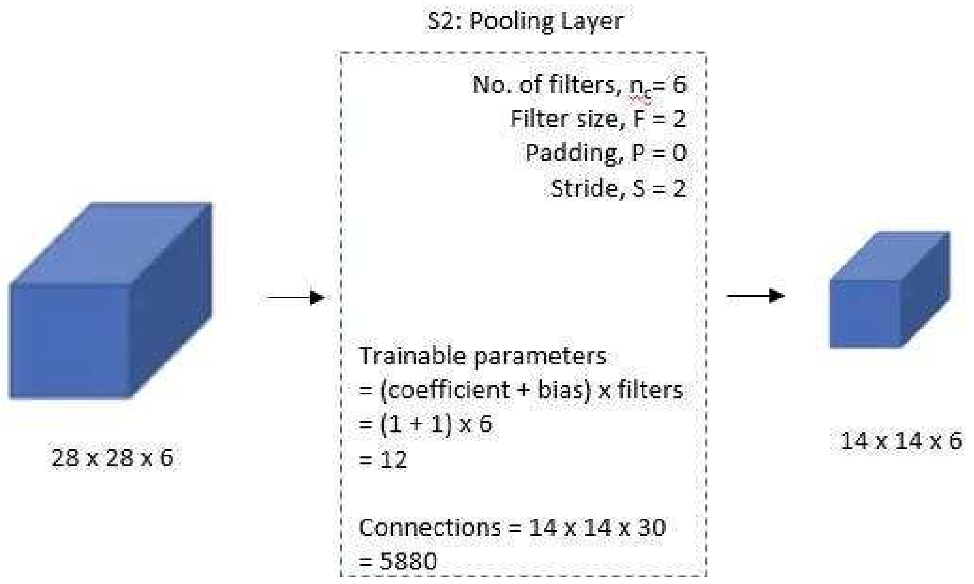
A 32x32 grayscale image serves as the input for LeNet-5 and is processed by the first convolutional layer comprising six feature maps or filters with a stride of one. From 32x32x1 to 28x28x6, the image’s dimensions shift.



First Layer

Second Layer

Then, using a filter size of 22 and a stride of 2, the LeNet-5 adds an average pooling layer or sub-sampling layer. 14x14x6 will be the final image's reduced size.



Second Layer

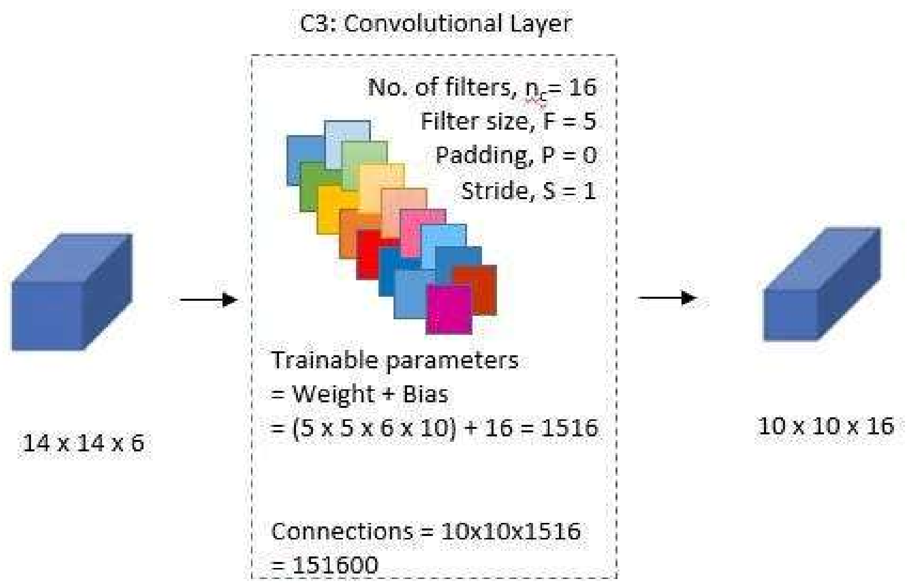
Third Layer

A second convolutional layer with 16 feature maps of size 55 and a stride of 1 is then present. Only 10 of the 16 feature maps in this layer are linked to the six feature maps in the layer below, as can be seen in the illustration below.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE I
EACH COLUMN INDICATES WHICH FEATURE MAP IN S2 ARE COMBINED BY THE UNITS IN A PARTICULAR FEATURE MAP OF C3.

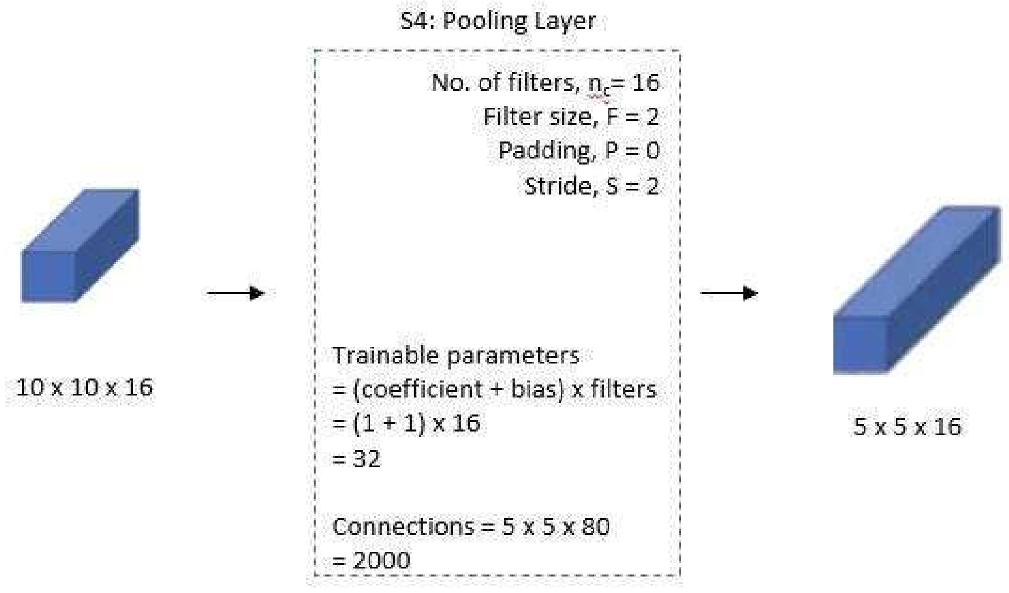
The primary goal is to disrupt the network's symmetry while maintaining a manageable number of connections. Because of this, there are 1516 training parameters instead of 2400 in these layers, and similarly, there are 151600 connections instead of 240000.



Third Layer

Fourth Layer

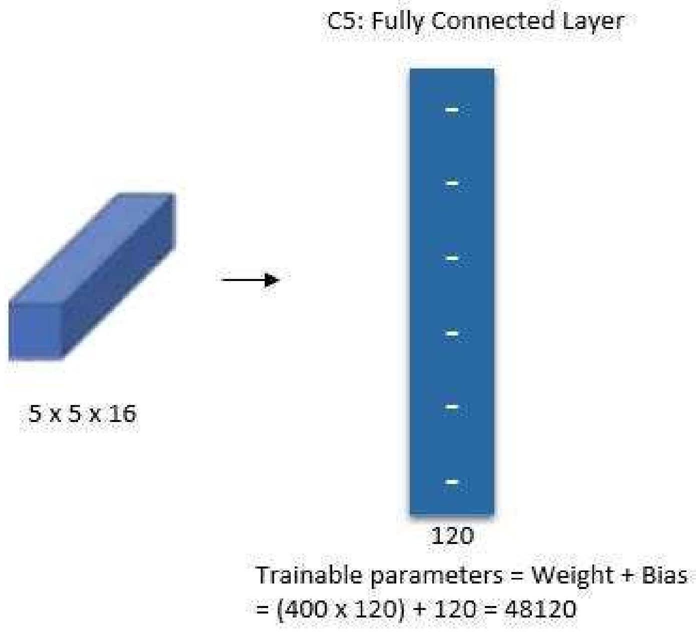
With a filter size of 22 and a stride of 2, the fourth layer (S4) is once more an average pooling layer. The output will be decreased to 5x5x16 because this layer is identical to the second layer (S2) but has 16 feature maps.



Fourth Layer

Fifth Layer

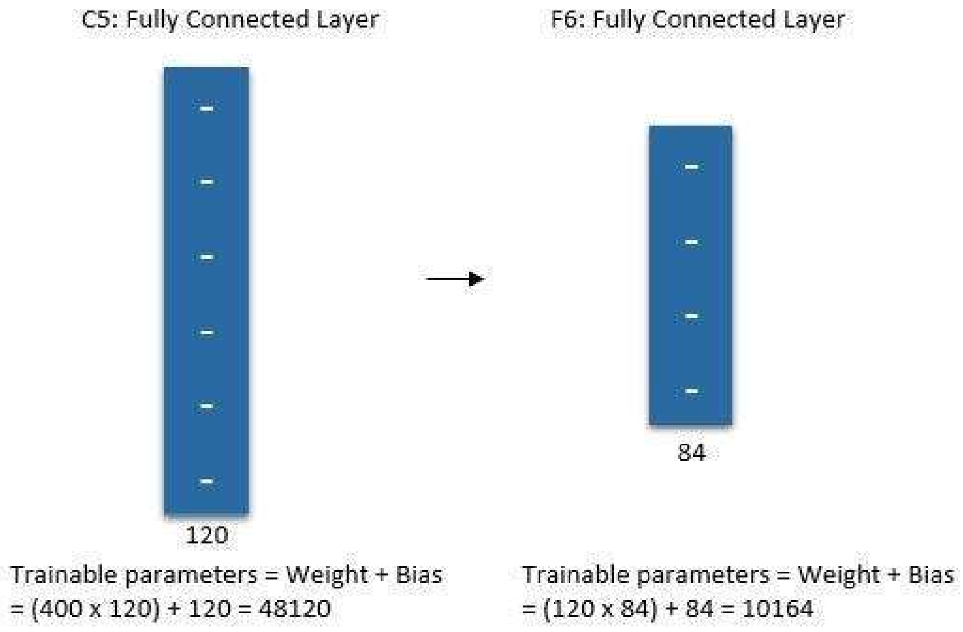
With 120 feature maps, each measuring 1 x 1, the fifth layer (C5) is a fully connected convolutional layer. All 400 nodes (5x5x16) in layer four, S4, are connected to each of the 120 units in C5's 120 units.


C5: Fully Connected Layer
5 x 5 x 16
120
Trainable parameters = Weight + Bias
= (400 x 120) + 120 = 48120

Fifth Layer

Sixth Layer

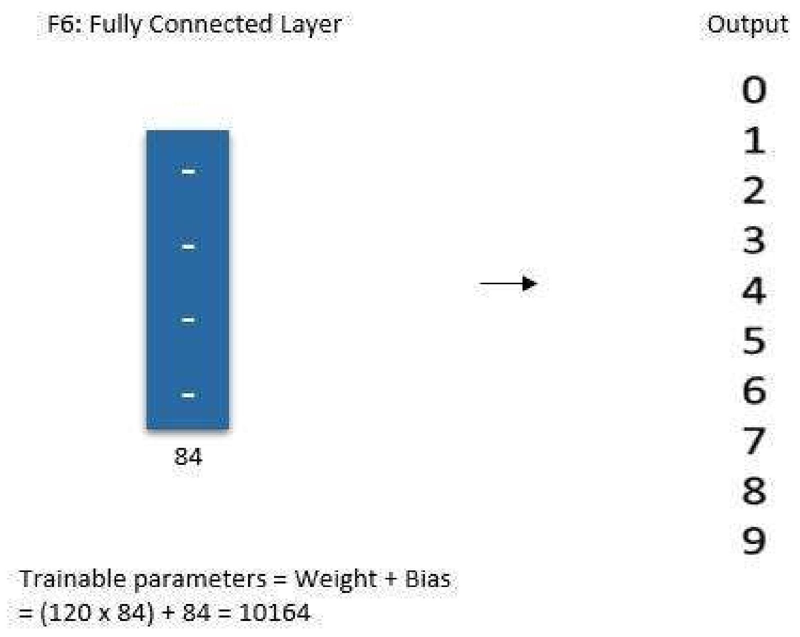
A fully connected layer (F6) with 84 units makes up the sixth layer.


C5: Fully Connected Layer
F6: Fully Connected Layer
120
84
Trainable parameters = Weight + Bias
= (400 x 120) + 120 = 48120
Trainable parameters = Weight + Bias
= (120 x 84) + 84 = 10164

Sixth Layer

Output Layer

The SoftMax output layer, which has 10 potential values and corresponds to the digits 0 to 9, is the last layer.


F6: Fully Connected Layer
Output
84
0
1
2
3
4
5
6
7
8
9
Trainable parameters = Weight + Bias
= (120 x 84) + 84 = 10164

Alexnet

AlexNet is a convolutional neural network (CNN) architecture that gained significant attention for its performance in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. It was designed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Here's a simplified overview of the AlexNet architecture in a CNN:

1. Input Layer: Accepts the input image. In the case of AlexNet, which was designed for ImageNet, the input size is 224x224x3 (RGB).

2. Convolutional Layers: The network contains five convolutional layers, each followed by a Rectified Linear Unit (ReLU) activation function. These layers learn hierarchical features from the input image.

3. Pooling Layers: After some of the convolutional layers, max-pooling layers are applied to downsample the spatial dimensions. Max-pooling helps in reducing the computational load and making the learned features more robust.

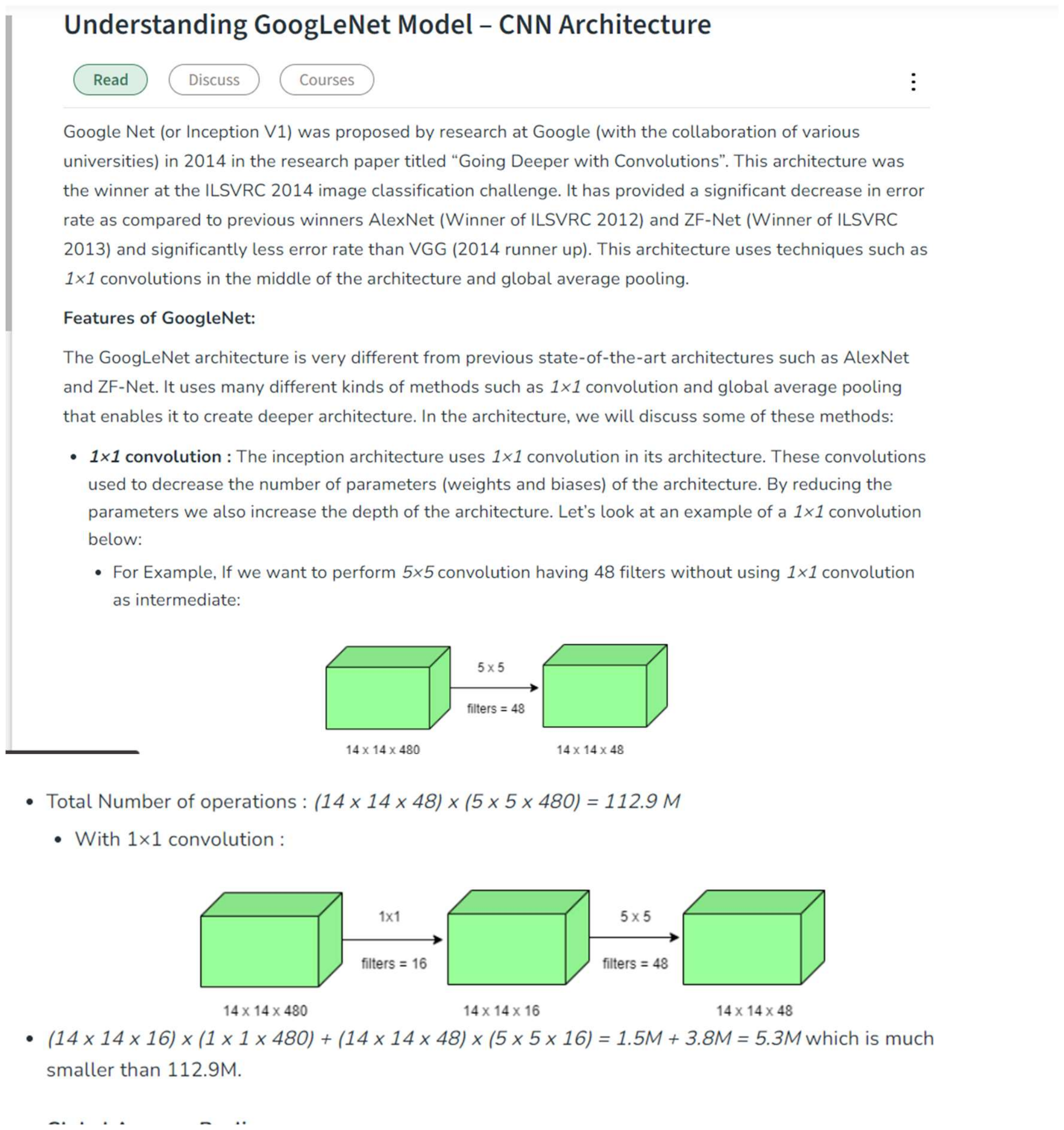
4. Local Response Normalization (LRN) Layers: LRN layers are used to normalize the responses of neighboring neurons, promoting competition among features. This helps enhance the contrast between the activated features.

5. Fully Connected Layers: Following the convolutional and pooling layers, there are three fully connected layers. These layers act as classifiers, producing the final output. The last fully connected layer has 1000 nodes, representing the 1000 classes in the ImageNet dataset.

6. Dropout: Dropout layers are used to prevent overfitting by randomly setting a fraction of input units to zero during training.

7. Softmax Layer: The final layer employs the softmax activation function to convert the network's output into probability scores for each class. The class with the highest probability is considered the predicted class.

The success of AlexNet played a pivotal role in the resurgence of interest in deep learning. It demonstrated the effectiveness of deep CNNs in image classification tasks. Implementing AlexNet in Cnn would involve using a deep learning framework like TensorFlow or PyTorch and defining the layers and connections according to the architecture described above. It's worth noting that newer architectures have been developed since AlexNet, but it remains a historically significant model.



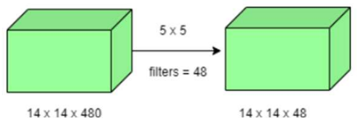
Understanding GoogLeNet Model – CNN Architecture

Google Net (or Inception V1) was proposed by research at Google (with the collaboration of various universities) in 2014 in the research paper titled "Going Deeper with Convolutions". This architecture was the winner at the ILSVRC 2014 image classification challenge. It has provided a significant decrease in error rate as compared to previous winners AlexNet (Winner of ILSVRC 2012) and ZF-Net (Winner of ILSVRC 2013) and significantly less error rate than VGG (2014 runner up). This architecture uses techniques such as *1×1* convolutions in the middle of the architecture and global average pooling.

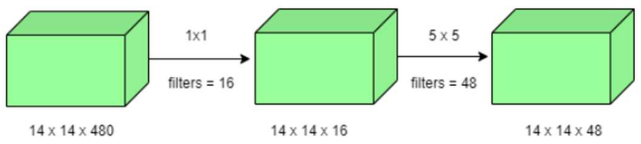
Features of GoogleNet:

The GoogLeNet architecture is very different from previous state-of-the-art architectures such as AlexNet and ZF-Net. It uses many different kinds of methods such as *1×1* convolution and global average pooling that enables it to create deeper architecture. In the architecture, we will discuss some of these methods:

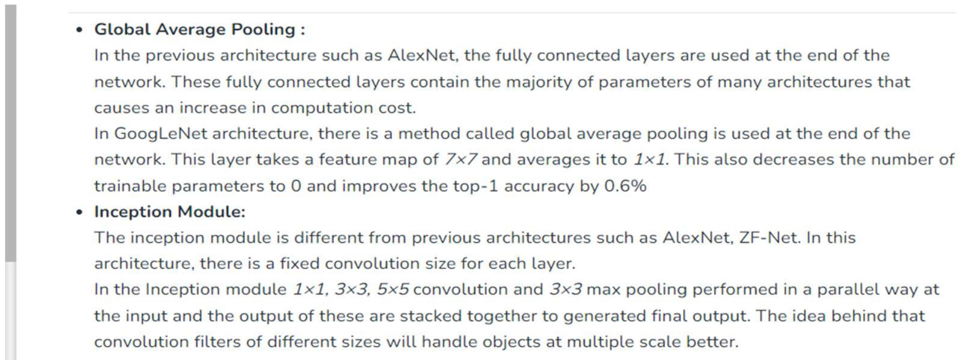
- ***1×1* convolution :** The inception architecture uses *1×1* convolution in its architecture. These convolutions used to decrease the number of parameters (weights and biases) of the architecture. By reducing the parameters we also increase the depth of the architecture. Let's look at an example of a *1×1* convolution below:
  - For Example, If we want to perform *5×5* convolution having 48 filters without using *1×1* convolution as intermediate:



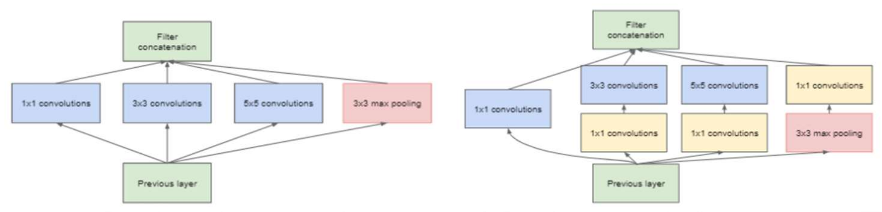
- Total Number of operations : *(14 x 14 x 48) x (5 x 5 x 480) = 112.9 M*
  - With 1×1 convolution :



- *(14 x 14 x 16) x (1 x 1 x 480) + (14 x 14 x 48) x (5 x 5 x 16) = 1.5M + 3.8M = 5.3M* which is much smaller than 112.9M.



- **Global Average Pooling :**
  In the previous architecture such as AlexNet, the fully connected layers are used at the end of the network. These fully connected layers contain the majority of parameters of many architectures that causes an increase in computation cost.
  In GoogLeNet architecture, there is a method called global average pooling is used at the end of the network. This layer takes a feature map of *7×7* and averages it to *1×1*. This also decreases the number of trainable parameters to 0 and improves the top-1 accuracy by 0.6%
- **Inception Module:**
  The inception module is different from previous architectures such as AlexNet, ZF-Net. In this architecture, there is a fixed convolution size for each layer.
  In the Inception module *1×1, 3×3, 5×5* convolution and *3×3* max pooling performed in a parallel way at the input and the output of these are stacked together to generated final output. The idea behind that convolution filters of different sizes will handle objects at multiple scale better.



(a) Inception module, naïve version (b) Inception module with dimension reductions

Q. Guided Back Propogation

Guided Backpropagation is a technique in deep learning used for visualizing and understanding the contributions of different input features to the model's predictions. It modifies the standard backpropagation algorithm by incorporating positive gradients and excluding negative gradients during the backward pass. This selective propagation of gradients enhances the interpretability of the model's decision-making process. The guided backpropagation method helps highlight the regions in the input space that have a positive impact on the model's predictions, aiding in the interpretation of complex neural network models.