1. What is the COVARIATE SHIFT Issue, and how does it affect you?

1. What is Dataset Shift?

Every time you participate in a competition, your journey will look quite similar to the one shown in the figure below.

https://cdn.analyticsvidhya.com/wp-content/uploads/2017/07/10091309/drifting-300x42.png

Let me explain this with the help of a scenario depicted in the picture below. You are given a train and a test file in a competition. You complete the preprocessing, the feature engineering and the cross validation part on the model created but you do not get the same result as the one you get on the cross-validation. No matter what validation strategy you try, it seems like you are bound to get different results in comparison to the cross validation.

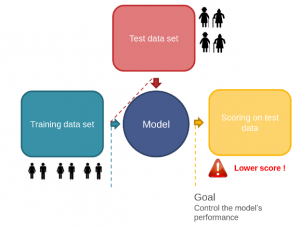


Image source: [link](https://github.com/AxeldeRomblay/MLBox/blob/master/docs/MLBox.pdf)

What can be a possible reason for this failure? So, if you carefully notice the first picture, you will find that you did all the manipulation by just looking at the train file. Therefore, you completely ignored the information contained in the test file.

Now take a look back on the second picture, you will notice that the training file contains information about male and females of fairly younger age while the test file contains information about people of older age. Therefore it means that the distribution of data contained in the train and test file is significantly different.

So, if you build your model based on the data set containing information about people having lower age and predict on a data set containing higher values of age, that will definitely give you a low score. The reason is that there will a wide gap in the interest and the activities between these two groups. So your model will fail in these conditions.

This change in the distribution of data contained in train and test file is called **dataset shift (or drifting)**.

2. What causes Dataset Shift?

Try to think some of the examples, where you can encounter the problem of dataset shift.

Basically, in the real world, dataset shift mainly occurs because of the change of environments (popularly called as non-stationary environment), where the environment can be referred as location, time, etc.

Let us consider an example. We collected the sales of various item during the period of July-September. Now your job is to predict the sales during the period of Diwali. The visual representation of sales in the train (blue line) and test (black line) file would be similar to the image shown below.

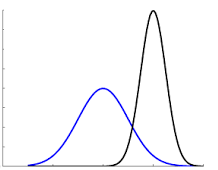


Image source: [link](http://iwann.ugr.es/2011/pdf/InvitedTalk-FHerrera-IWANN11.pdf)

Clearly, the sales during the time of Diwali would be much higher as compared to routine days. Therefore we can say that it is the situation of dataset shift, which occurred due to change of time period between our train and test file.

But our machine learning algorithms work by ignoring these changes. They presume that the train and test environments match and even if they don’t, it assumes that it makes no difference if the environment changes.

Now take a look back at both of the examples that we discussed above. Is there any difference between them?

Yes, in the first scenario, there was a shift in the age(**independent variable or predictor**) of the population due to which we were getting wrong predictions. While in the latter one, there was a shift in the sales(**target variable**) of the items. This brings the next topic to the table – Different types of Dataset shifts.

3. Types of Dataset Shift

Dataset shift could be divided into three types:

1. Shift in the independent variables (**Covariate Shift**)
2. Shift in the target variable (**Prior probability shift**)
3. Shift in the relationship between the independent and the target variable (**Concept Shift**)

In this article, we will discuss only covariate shift in this article since the other two topics are still an active research area and there has not been any substantial work to mitigate these problems.

We will also see the methods to identify Covariate shift and the proper measures that can be taken in order to improve the predictions.

4. Covariate Shift

Covariate shift refers to the change in the distribution of the input variables present in the training and the test data. It is the most common type of shift and it is now gaining more attention as nearly every real-world dataset suffers from this problem.

First, let us try to understand how does the change in distribution creates a problem for us. Take a look at the image shown below.

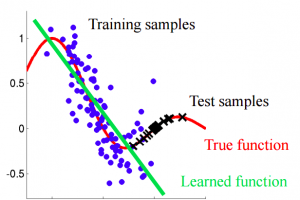




Image source: [link](http://iwann.ugr.es/2011/pdf/InvitedTalk-FHerrera-IWANN11.pdf)

If you carefully notice the image given above, our learning function tries to fit the training data. But here, we can see that the distribution of training and test is different, so predicting using this learned function will definitely give us wrong predictions.

So our first step should be to identify this shift in the distribution. Let’s try and understand it.

5. Identification

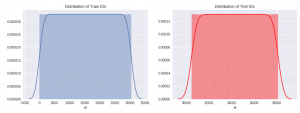
Here, I have used a quick and dirty machine learning technique to check whether there is a shift between the training data and the test data.

For this purpose, I will use [Sberbank Russian Housing Market](https://www.kaggle.com/c/sberbank-russian-housing-market/submissions?sortBy=date&group=all&page=1) dataset from Kaggle.

**The basic idea to identify shift –**If there exists a shift in the dataset, then on mixing the train and test file, you should still be able to classify an instance of the mixed dataset as train or test with reasonable accuracy. Why?

Because, if the features in both the dataset belong to different distributions then, they should be able to separate the dataset into train and test file significantly.

Let’s try to make it simple. Take a look at the distribution of the feature ‘id’ in both the dataset.



By looking at their distribution, we can clearly see that after a certain value (=30,473), all the instances will belong to test dataset.

So if we create a dataset which is a mixture of training and test instances, where we have labelled each instance of training data as ‘training’ and test as ‘test’ before mixing.

In this new dataset, if we just look at the feature ‘id’, we can clearly classify any instance that whether it belongs to training data or test data. Therefore, we can conclude that ‘id’ is a drifting feature for this dataset.

So this was fairly easy. But we can’t visualise every variable and check whether it is drifting or not. For that purpose, let us try to code this in Python as a simple classification problem and identify the drifting features.

Steps to identify drift

The basic steps that we will follow are:

1. Preprocessing: This step involves imputing all missing values and label encoding of all categorical variables.
2. Creating a random sample of your training and test data separately and adding a new feature origin which has value train or test depending on whether the observation comes from the training dataset or the test dataset.
3. Now combine these random samples into a single dataset. Note that the shape of both the samples of training and test dataset should be nearly equal, otherwise it can be a case of an unbalanced dataset.
4. Now create a model taking one feature at a time while having ‘origin’ as the target variable on a part of the dataset (say ~75%).
5. Now predict on the rest part(~25%) of the dataset and calculate the value of AUC-ROC.
6. Now if the value of AUC-ROC for a particular feature is greater than **0.80**, we classify that feature as drifting.

Note that we generally take 0.80 as the threshold value, but the value can be altered based on the situation.

So that is enough of theory, now let’s code this and find which of the features are drifting in this problem.

https://cdn.analyticsvidhya.com/wp-content/uploads/2017/07/07232020/Screenshot-from-2017-07-07-23-20-02-300x42.png  
# Drifting features : {id, life\_sq, kitch\_sq, hospital\_beds\_raion, cafe\_sum\_500\_min\_price\_avg, cafe\_sum\_500\_max\_price\_avg, cafe\_avg\_price\_500 }

Here we have classified seven features as drifting. You can also manually check their difference in distribution through some visualisation or by using 1-way ANOVA test.

So, now the important question is how to treat them effectively such that we can improve our predictions.

6. Treatment

There are different techniques by which we can treat these features in order to improve our model. Let us discuss some of them.

1. Dropping of drifting features
2. Importance weight using Density Ratio Estimation

So let’s try to understand them.

6.1 Dropping

This method is quite simple, as in this, we basically drop the features which are being classified as drifting. But just give it a thought, that simply dropping features might result in some loss of information.

To deal with this, we have defined a simple rule.

*Features having a drift value greater than 0.8 and are****not important****in our model, we drop them.*

So, let’s try this in our problem.

Here, I have used a basic random forest model just to check which features are important.

# **using** a basic model **with** **all** the featurestraining = train.drop('origin',axis=1)testing = test.drop('origin',axis=1)

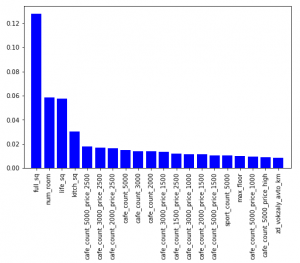
rf = RandomForestRegressor(n\_estimators=200, max\_depth=6,max\_features=10)rf.fit(training.drop('price\_doc',axis=1),training['price\_doc'])pred = rf.predict(testing)columns = ['price\_doc']sub = pd.DataFrame(data=pred,columns=columns)sub['id'] = test['id']sub = sub[['id','price\_doc']]sub.to\_csv('with\_drifting.csv', index=False)

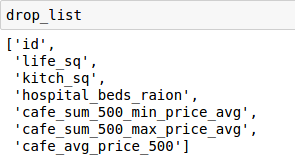
On submitting this file on Kaggle, we are getting a rmse score of **0.40116** on private leaderboard.

So, let’s check first 20 important features for this model.

### plotting importancesfeatures = training.drop('price\_doc',axis=1).columns.valuesimp = rf.feature\_importances\_indices = np.argsort(imp)[::-1][:20]

#plotplt.figure(figsize=(8,5))**plt**.bar(range(len(indices)), imp[indices], color = 'b', align='center')**plt**.xticks(range(len(indices)), features[indices], rotation='vertical')**plt**.xlim([-1,len(indices)])**plt**.show()





Now, if we compare our drop list and feature importance, we will find that the features ‘life\_sq’ and ‘kitch\_sq’ are common.

So, we will keep these two features in our model, while dropping the rest of the drifting features.

NOTE: Before dropping any feature, just make sure you if there any possibility to create a new feature from it.

Let’s try this and check whether it improves our prediction or not.

## dropping drifting features which **are** **not** important.drift\_train = training.drop(['id','hospital\_beds\_raion','cafe\_sum\_500\_min\_price\_avg','cafe\_sum\_500\_max\_price\_avg','cafe\_avg\_price\_500'], axis=1)drift\_test = testing.drop(['id','hospital\_beds\_raion','cafe\_sum\_500\_min\_price\_avg','cafe\_sum\_500\_max\_price\_avg','cafe\_avg\_price\_500'], axis=1)

rf = RandomForestRegressor(n\_estimators=200, max\_depth=6,max\_features=10)rf.fit(drift\_train.drop('price\_doc',axis=1),training['price\_doc'])pred = rf.predict(drift\_test)columns = ['price\_doc']sub = pd.DataFrame(data=pred,columns=columns)sub['id'] = test['id']sub = sub[['id','price\_doc']]sub.to\_csv('without\_drifting.csv', index=False)

On submission of this file on Kaggle, we got a rmse score of **0.39759** on the private leaderboard.

Congratulations, we have successfully improved our performance using this technique.

6.2 Importance weight using Density Ratio Estimation

In this method, the approach to importance estimation would be to first estimate the training and test densities separately and then estimate the importance by taking the ratio of the estimated densities of test and train.

Then these densities act as weights for each instance in the training data.

But giving weights to each instance based on the density ratio could be a rigorous task in higher dimensional data sets. I tried this method on an i7 processor with 128 GB RAM and it took around 3 minutes to calculate the ratio density for a single feature. Also, I could not find any improvement in the score on applying the weights to the training data.

1. What is the process of BATCH NORMALIZATION?

**What is Batch Normalization?**

***Batch normalization*** was introduced to mitigate the internal covariate shift problem in [neural networks](https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/) by Sergey Ioffe and Christian Szegedy in 2015. The normalization process involves calculating the mean and variance of each feature in a mini-batch and then scaling and shifting the features using these statistics. This ensures that the input to each layer remains roughly in the same distribution, regardless of changes in the distribution of earlier layers’ outputs. Consequently, Batch Normalization helps in stabilizing the training process, enabling higher learning rates and faster convergence.

**Need for Batch Normalization**

Batch Normalization is extension of concept of normalization from just the input layer to the activations of each hidden layer throughout the neural network. By normalizing the activations of each layer, Batch Normalization helps to alleviate the internal covariate shift problem, which can hinder the convergence of the network during training.

*In traditional neural networks, as the input data propagates through the network, the distribution of each layer’s inputs changes. This phenomenon, known as****internal covariate shift****, can slow down the training process. Batch Normalization aims to mitigate this issue by normalizing the inputs of each layer.*

The inputs to each hidden layer are the activations from the previous layer. If these activations are normalized, it ensures that the network is consistently presented with inputs that have a similar distribution, regardless of the training stage. This stability in the distribution of inputs allows for smoother and more efficient training.

By applying Batch Normalization into the hidden layers of the network, the gradients propagated during backpropagation are less likely to vanish or explode, leading to more stable training dynamics. This ultimately facilitates faster convergence and better performance of the neural network on the given task.

**Fundamentals of Batch Normalization**

In this section, we are going to discuss the steps taken to perform batch normalization.

**Step 1: Compute the Mean and Variance of Mini-Batches**

For mini-batch of activations x1​,x2​,…,xm​*x*1​​,*x*2​​,…,*xm*​​, the mean μB*μB*​​ and variance σB2*σB*2​ of the mini-batch are computed.

**Step 2: Normalization**

Each activation xi​*xi*​​is normalized using the computed mean and variance of the mini-batch.

The normalization process subtracts the mean μB*μB*​​ from each activation and divides by the square root of the variance σB2*σB*2​​, ensuring that the normalized activations have a zero mean and unit variance.

Additionally, a small constant ϵ*ϵ* is added to the denominator for numerical stability, particularly to prevent division by zero.

xi^=xi–μBσB2+ϵ*xi*​​=*σB*2​+*ϵ*​*xi*​–*μB*​​

**Step 3: Scale and Shift the Normalized Activations**

The normalized activations xi*xi* are then scaled by a learnable parameter γ*γ* and shifted by another learnable parameter β*β*. These parameters allow the model to learn the optimal scaling and shifting of the normalized activations, giving the network additional flexibility.

yi=γxi^+β*yi*​=*γxi*​​+*β*

**Batch Normalization in TensorFlow**

In the provided pseudo code, we have used a simple neural network model with batch normalization using [TensorFlow’s Keras API](https://www.geeksforgeeks.org/how-to-create-models-in-keras/" \t "_blank). We have added, the batch normalization layer using ‘***tf.keras.layers.BatchNormalization()***‘ to normalize the activations of the previous layer.

import tensorflow as tf  
  
# Define a simple model  
model = tf.keras.Sequential([  
 tf.keras.layers.Dense(64, input\_shape=(784,)),  
  **tf.keras.layers.BatchNormalization(), # Add Batch Normalization layer**  
 tf.keras.layers.Activation('relu'),  
 tf.keras.layers.Dense(10),  
 tf.keras.layers.Activation('softmax')  
])  
  
# Compile the model  
model.compile(optimizer='adam',  
 loss='sparse\_categorical\_crossentropy',  
 metrics=['accuracy'])  
  
# Train the model  
model.fit(x\_train, y\_train, epochs=5, batch\_size=32)

**Batch Normalization in PyTorch**

In the following pseudo code, we have build a simple neural network with batch normalization using [PyTorch](https://www.geeksforgeeks.org/getting-started-with-pytorch/" \t "_blank). We have define a subclass of ‘***nn.Module***‘ and added the ‘***nn.BatchNorm1D***‘ after the first fully connected layer to normalize the activations.

We have used ***‘nn.BatchNorm1D’***as the input data is one-dimensional, but for two-dimensional data, especially for Convolutional Neural Networks ***‘BatchNorm2D’*** is used.

import torch  
import torch.nn as nn  
  
# Define a simple model  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, self).\_\_init\_\_()  
 self.fc1 = nn.Linear(784, 64)  
  **self.bn = nn.BatchNorm1d(64) # Add Batch Normalization layer**  
 self.relu = nn.ReLU()  
 self.fc2 = nn.Linear(64, 10)  
 self.softmax = nn.Softmax(dim=1)  
   
 def forward(self, x):  
 x = self.fc1(x)  
 x = self.bn(x) # Apply Batch Normalization  
 x = self.relu(x)  
 x = self.fc2(x)  
 x = self.softmax(x)  
 return x  
  
# Instantiate the model  
model = Model()  
  
# Define loss function and optimizer  
criterion = nn.CrossEntropyLoss()  
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)  
  
# Train the model  
for epoch in range(5):  
 for inputs, labels in train\_loader:  
 optimizer.zero\_grad()  
 outputs = model(inputs)  
 loss = criterion(outputs, labels)  
 loss.backward()  
 optimizer.step()

**Benefits of Batch Normalization**

* **Faster Convergence:** Batch Normalization reduces internal covariate shift, allowing for faster convergence during training.
* **Higher Learning Rates:** With Batch Normalization, higher learning rates can be used without the risk of divergence.
* **Regularization Effect:** Batch Normalization introduces a slight regularization effect that reduces the need for adding regularization techniques like dropout.

1. Using our own terms and diagrams, explain LENET ARCHITECTURE.

In the late 1990s, Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner created a convolutional neural network (CNN) based architecture called LeNet. The LeNet-5 architecture was developed to recognize handwritten and machine-printed characters, a function that showcased the potential of deep learning in practical applications. This article provides an in-depth exploration of the LeNet-5 architecture, examining each component and its contribution in deep learning.

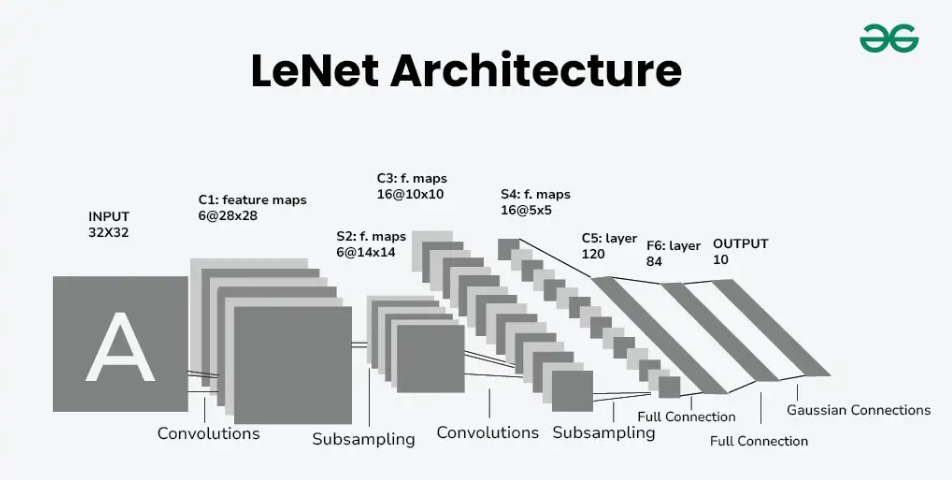
## Introduction to LeNet-5

LeNet-5 is a [convolutional neural network (CNN)](https://www.geeksforgeeks.org/introduction-convolution-neural-network/" \t "_blank)architecture that introduced several key features and innovations that have become standard in modern deep learning. It demonstrated the effectiveness of CNNs for image recognition tasks and introduced key concepts such as convolution, pooling, and hierarchical feature extraction that underpin modern deep learning models.

Originally designed for [handwritten digit recognition](https://www.geeksforgeeks.org/handwritten-digit-recognition-using-neural-network/), the principles behind LeNet-5 have been extended to various applications, including:

* Handwriting recognition in postal services and banking.
* Object and face recognition in images and videos.
* Autonomous driving systems for recognizing and interpreting road signs.

## Architecture of LeNet-5



*LeNet-5 Architecture for Digit Recognition*

The architecture of LeNet 5 contains 7 layers excluding the input layer. Here is a detailed breakdown of the LeNet-5 architecture:

### 1. Input Layer

* **Input Size**: 32×32 pixels.
* The input is larger than the largest character in the database, which is at most 20×20 pixels, centered in a 28×28 field. The larger input size ensures that distinctive features such as stroke endpoints or corners can appear in the center of the receptive field of the highest-level feature detectors.
* **Normalization**: Input pixel values are normalized such that the background (white) corresponds to a value of 0, and the foreground (black) corresponds to a value of 1. This normalization makes the mean input roughly 0 and the variance roughly 1, which accelerates the learning process.

### 2. ****Layer C1 (Convolutional Layer)****

* **Feature Maps**: 6 feature maps.
* **Connections**: Each unit is connected to a 5×5 neighborhood in the input, producing 28×28 feature maps to prevent boundary effects.
* **Parameters**: 156 trainable parameters and 117,600 connections.

### ****3. Layer S2 (Subsampling Layer)****

* **Feature Maps**: 6 feature maps.
* **Size**: 14×14 (each unit connected to a 2×2 neighborhood in C1).
* **Operation**: Each unit adds four inputs, multiplies by a trainable coefficient, adds a bias, and applies a sigmoid function.
* **Parameters**: 12 trainable parameters and 5,880 connections.

***Partial Connectivity****: C3 is not fully connected to S2, which limits the number of connections and breaks symmetry, forcing feature maps to learn different, complementary features.*

### 4. ****Layer C3 (Convolutional Layer)****

* **Feature Maps**: 16 feature maps.
* **Connections**: Each unit is connected to several 5×5 neighborhoods at identical locations in a subset of S2’s feature maps.
* **Parameters and Connections**: Connections are partially connected to force feature maps to learn different features, with 1,516 trainable parameters and 151,600 connections.

### 5. ****Layer S4 (Subsampling Layer)****

* **Feature Maps**: 16 feature maps.
* **Size**: 7×7 (each unit connected to a 2×2 neighborhood in C3).
* **Parameters**: 32 trainable parameters and 2,744 connections.

### 6. ****Layer C5 (Convolutional Layer)****

* **Feature Maps**: 120 feature maps.
* **Size**: 1×1 (each unit connected to a 5×5 neighborhood on all 16 of S4’s feature maps, effectively fully connected due to input size).
* **Parameters**: 48,000 trainable parameters and 48,000 connections.

### 7. ****Layer F6 (Fully Connected Layer)****

* **Units**: 84 units.
* **Connections**: Each unit is fully connected to C5, resulting in 10,164 trainable parameters.
* **Activation**: Uses a scaled hyperbolic tangent function f(a)=Atan⁡(Sa)*f*(*a*)=*A*tan(*Sa*), where A = 1.7159 and S = 2/3

### 8. ****Output Layer****

In the output layer of LeNet, each class is represented by an Euclidean Radial Basis Function (RBF) unit. Here’s how the output of each RBF unit yi*yi*​is computed:

yi=∑jxj.wij*yi*​=∑*j*​*xj*​.*wij*​​

In this equation:

* xj*xj*​ represents the inputs to the RBF unit.
* wij*wij*​ represents the weights associated with each input.
* The summation is over all inputs to the RBF unit.

In essence, the output of each RBF unit is determined by the Euclidean distance between its input vector and its parameter vector. The larger the distance between the input pattern and the parameter vector, the larger the RBF output. This output can be interpreted as a penalty term measuring the fit between the input pattern and the model of the class associated with the RBF unit.

## Detailed Explanation of the Layers

* **Convolutional Layers (Cx)**: These layers apply convolution operations to the input, using multiple filters to extract different features. The filters slide over the input image, computing the dot product between the filter weights and the input pixels. This process captures spatial hierarchies of features, such as edges and textures.
* **Subsampling Layers (Sx)**: These layers perform pooling operations (average pooling in the case of LeNet-5) to reduce the spatial dimensions of the feature maps. This helps to control overfitting, reduce the computational load, and make the representation more compact.
* **Fully Connected Layers (Fx)**: These layers are densely connected, meaning each neuron in these layers is connected to every neuron in the previous layer. This allows the network to combine features learned in previous layers to make final predictions.

The overall architecture of LeNet-5, with its combination of convolutional, subsampling, and fully connected layers, was designed to be both computationally efficient and effective at capturing the hierarchical structure of handwritten digit images. The careful normalization of input values and the structured layout of receptive fields contribute to the network’s ability to learn and generalize from the training data effectively.

1. Using our own terms and diagrams, explain ALEXNET ARCHITECTURE.

This article is focused on providing an introduction to the AlexNet architecture. Its name comes from one of the leading authors of the AlexNet [paper](https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf)– Alex Krizhevsky. It won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 with a top-5 error rate of **15.3%** (beating the runner up which had a top-5 error rate of 26.2%).

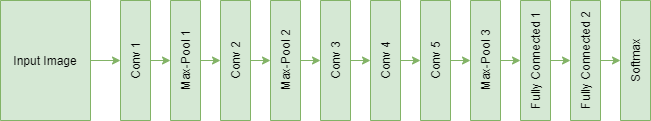
The most important features of the AlexNet paper are:

* As the model had to train 60 million parameters (which is quite a lot), it was prone to overfitting. According to the paper, the usage of Dropout and Data Augmentation significantly helped in reducing overfitting. The first and second fully connected layers in the architecture thus used a dropout of 0.5 for the purpose. Artificially increasing the number of images through data augmentation helped in the expansion of the dataset dynamically during runtime, which helped the model generalize better.
* Another distinct factor was using the ReLU activation function instead of tanh or sigmoid, which resulted in faster training times (a decrease in training time by 6 times). Deep Learning Networks usually employ ReLU non-linearity to achieve faster training times as the others start saturating when they hit higher activation values.

### The Architecture

The architecture consists of 5 Convolutional layers, with the 1st, 2nd and 5th having Max-Pooling layers for proper feature extraction. The Max-Pooling layers are overlapped having strides of 2 with filter size 3×3. This resulted in decreasing the top-1 and top-5 error rates by **0.4%** and **0.3%** respectively in comparison to non-overlapped Max-Pooling layers. They are followed by 2 fully-connected layers (each with dropout) and a softmax layer at the end for predictions.

The figure below shows the architecture of AlexNet with all the layers defined.



**Code: Python code to implement AlexNet for object classification**

|  |
| --- |
| model = Sequential()    # 1st Convolutional Layer  model.add(Conv2D(filters = 96, input\_shape = (224, 224, 3),              kernel\_size = (11, 11), strides = (4, 4),              padding = 'valid'))  model.add(Activation('relu'))  # Max-Pooling  model.add(MaxPooling2D(pool\_size = (2, 2),              strides = (2, 2), padding = 'valid'))  # Batch Normalisation  model.add(BatchNormalization())    # 2nd Convolutional Layer  model.add(Conv2D(filters = 256, kernel\_size = (11, 11),              strides = (1, 1), padding = 'valid'))  model.add(Activation('relu'))  # Max-Pooling  model.add(MaxPooling2D(pool\_size = (2, 2), strides = (2, 2),              padding = 'valid'))  # Batch Normalisation  model.add(BatchNormalization())    # 3rd Convolutional Layer  model.add(Conv2D(filters = 384, kernel\_size = (3, 3),              strides = (1, 1), padding = 'valid'))  model.add(Activation('relu'))  # Batch Normalisation  model.add(BatchNormalization())    # 4th Convolutional Layer  model.add(Conv2D(filters = 384, kernel\_size = (3, 3),              strides = (1, 1), padding = 'valid'))  model.add(Activation('relu'))  # Batch Normalisation  model.add(BatchNormalization())    # 5th Convolutional Layer  model.add(Conv2D(filters = 256, kernel\_size = (3, 3),              strides = (1, 1), padding = 'valid'))  model.add(Activation('relu'))  # Max-Pooling  model.add(MaxPooling2D(pool\_size = (2, 2), strides = (2, 2),              padding = 'valid'))  # Batch Normalisation  model.add(BatchNormalization())    # Flattening  model.add(Flatten())    # 1st Dense Layer  model.add(Dense(4096, input\_shape = (224\*224\*3, )))  model.add(Activation('relu'))  # Add Dropout to prevent overfitting  model.add(Dropout(0.4))  # Batch Normalisation  model.add(BatchNormalization())    # 2nd Dense Layer  model.add(Dense(4096))  model.add(Activation('relu'))  # Add Dropout  model.add(Dropout(0.4))  # Batch Normalisation  model.add(BatchNormalization())    # Output Softmax Layer  model.add(Dense(num\_classes))  model.add(Activation('softmax')) |

1. Describe the vanishing gradient problem.

In the realm of deep learning, the optimization process plays a crucial role in training neural networks. Gradient descent, a fundamental optimization algorithm, can sometimes encounter two common issues: vanishing gradients and exploding gradients. In this article, we will delve into these challenges, providing insights into what they are, why they occur, and how to mitigate them. We will build and train a model, and learn how to face vanishing and exploding problems.

## What is Vanishing Gradient?

The vanishing [gradient](https://www.geeksforgeeks.org/gradient-descent-in-linear-regression/) problem is a challenge that emerges during backpropagation when the derivatives or slopes of the activation functions become progressively smaller as we move backward through the layers of a neural network. This phenomenon is particularly prominent in deep networks with many layers, hindering the effective training of the model. The weight updates becomes extremely tiny, or even exponentially small, it can significantly prolong the training time, and in the worst-case scenario, it can halt the training process altogether.

### Why the Problem Occurs?

During backpropagation, the gradients propagate back through the layers of the network, they decrease significantly. This means that as they leave the output layer and return to the input layer, the gradients become progressively smaller. As a result, the weights associated with the initial levels, which accommodate these small gradients, are updated little or not at each iteration of the optimization process.

**The vanishing gradient problem** is particularly associated with the sigmoid and hyperbolic tangent (tanh) [activation functions](https://www.geeksforgeeks.org/activation-functions/) because their derivatives fall within the range of 0 to 0.25 and 0 to 1, respectively. Consequently, extreme weights becomes very small, causing the updated weights to closely resemble the original ones. This persistence of small updates contributes to the vanishing gradient issue.

The sigmoid and tanh functions limit the input values ​​to the ranges [0,1] and [-1,1], so that they saturate at 0 or 1 for sigmoid and -1 or 1 for Tanh. The derivatives at points becomes zero as they are moving. In these regions, especially when inputs are very small or large, the gradients are very close to zero. While this may not be a major concern in shallow networks with a few layers, it is a more pronounced issue in deep networks. When the inputs fall in saturated regions, the gradients approach zero, resulting in little update to the weights of the previous layer. In simple networks this does not pose much of a problem, but as more layers are added, these small gradients, which multiply between layers, decay significantly and consequently the first layer tears very slowly , and hinders overall model performance and can lead to convergence failure.

### How can we identify?

Identifying the vanishing gradient problem typically involves monitoring the training dynamics of a deep neural network.

* One key indicator is observing model weights**converging to 0** or stagnation in the improvement of the model’s performance metrics over training epochs.
* During training, if the **loss function fails to decrease**significantly, or if there is erratic behavior in the learning curves, it suggests that the gradients may be vanishing.
* Additionally, examining the gradients themselves during backpropagation can provide insights. **Visualization techniques**, such as gradient histograms or norms, can aid in assessing the distribution of gradients throughout the network.

### How can we solve the issue?

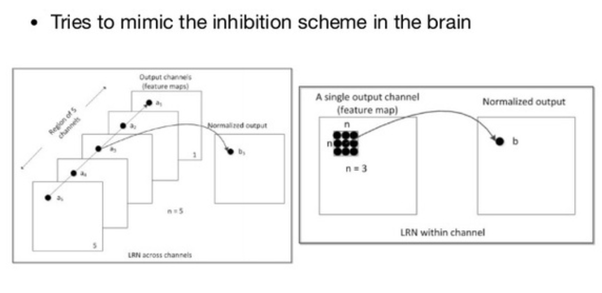
* **Batch Normalization**: Batch normalization normalizes the inputs of each layer, reducing internal covariate shift. This can help stabilize and accelerate the training process, allowing for more consistent gradient flow.
* **Activation function**: Activation function like **Rectified Linear Unit (ReLU)** can be used. With **ReLU,** the gradient is 0 for negative and zero input, and it is 1 for positive input, which helps alleviate the vanishing gradient issue. Therefore, ReLU operates by replacing poor enter values with 0, and 1 for fine enter values, it preserves the input unchanged.
* **Skip Connections and Residual Networks (ResNets)**: Skip connections, as seen in ResNets, allow the gradient to bypass certain layers during backpropagation. This facilitates the flow of information through the network, preventing gradients from vanishing.
* **Long Short-Term Memory Networks (LSTMs) and Gated Recurrent Units (GRUs)**: In the context of recurrent neural networks (RNNs), architectures like LSTMs and GRUs are designed to address the vanishing gradient problem in sequences by incorporating gating mechanisms .
* **Gradient Clipping**: Gradient clipping involves imposing a threshold on the gradients during backpropagation. Limit the magnitude of gradients during backpropagation, this can prevent them from becoming too small or exploding, which can also hinder learning.

1. What is NORMALIZATION OF LOCAL RESPONSE?

**Local Response Normalization (LRN)** is a technique used in neural networks to normalize the activity of neurons within a local neighborhood. It's particularly common in convolutional neural networks (CNNs) to improve their generalization and performance.

**Important Points Local Response Normalization:**

1. **Normalization within Local Neighborhood:** LRN computes the activity of a neuron by normalizing it based on the activities of its neighboring neurons within a local region.
2. **Normalization Formula:** The normalization formula typically involves dividing the activity of a neuron by a factor computed from the sum of squares of activities of neighboring neurons.
3. **Parameters:** LRN usually includes parameters such as the size of the local neighborhood, a scaling parameter, and an offset to control the degree of normalization.
4. **Role in CNNs:** In CNNs, LRN helps in enhancing the contrast between different features, promoting the detection of salient features while suppressing irrelevant ones.
5. **Normalization Across Channels:** In some implementations, LRN also normalizes across channels, ensuring that responses in different feature channels are relatively balanced.



**Real-world Example:**  
In image classification tasks, consider a CNN designed to classify objects in images. During the processing of an image, LRN helps enhance the responses of neurons to certain localized features, such as edges or textures, while suppressing responses to others. For example, if the network is trained to recognize cats, LRN might enhance the responses to specific features like cat ears or whiskers, making them more prominent in the classification process. As a result, the network becomes more effective at discriminating between images containing cats and those without, improving its overall accuracy.

7. In AlexNet, what WEIGHT REGULARIZATION was used?

AlexNet achieved good results on the ImageNet classification task in 2012 without explicitly using regularization or dropout techniques primarily because of two reasons:

1. Availability of large labeled dataset (ImageNet): At the time, ImageNet was one of the largest labeled computer vision datasets available with over 1 million images. Having a big dataset acts as an implicit regularizer, preventing overfitting to some extent.

2. Use of ReLU activations: AlexNet was one of the early CNN models to make extensive use of ReLU activations in place of sigmoid/tanh units. ReLUs help address the vanishing gradient problem and make networks easier to optimize. This architectural choice provided better generalization without explicit regularization.

3. Transfer learning from smaller datasets: The initial layers of AlexNet were pre-trained on smaller datasets like CIFAR-10 before finetuning on ImageNet. This stage-wise pre-training acts as a crude form of regularization.

4. Data augmentation: Random transformations like cropping, mirroring etc. were applied to input images during training to artificially increase the diversity of dataset. This helps reduce overfitting.

So in summary, having a big labeled dataset, use of ReLUs, pre-training and data augmentation helped AlexNet generalize well without techniques like dropout that became popular later with smaller datasets.

8. Using our own terms and diagrams, explain VGGNET ARCHITECTURE.

The***Visual Geometry Group (VGG) models***, particularly ***VGG-16 and VGG-19***, have significantly influenced the field of computer vision since their inception. These models, introduced by the Visual Geometry Group from the University of Oxford, stood out in the 2014 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) for their deep convolutional neural networks (CNNs) with a uniform architecture. VGG-19, the deeper variant of the VGG models, has garnered considerable attention due to its simplicity and effectiveness.

***This article delves into the architecture of VGG-19, its evolution, and its impact on the development of deep learning models.***

**Table of Content**

* [Evolution of VGG Models](https://www.geeksforgeeks.org/vgg-net-architecture-explained/#evolution-of-vgg-models)
* [VGG-19 Architecture](https://www.geeksforgeeks.org/vgg-net-architecture-explained/#vgg19-architecture)
* [Detailed Layer-by-Layer Architecture of VGG-Net 19](https://www.geeksforgeeks.org/vgg-net-architecture-explained/#detailed-layerbylayer-architecture-of-vggnet-19)
* [Architectural Design Principles](https://www.geeksforgeeks.org/vgg-net-architecture-explained/#architectural-design-principles)
* [Impact and Legacy of VGG-19](https://www.geeksforgeeks.org/vgg-net-architecture-explained/#impact-and-legacy-of-vgg19)
* [Additional Information about VGGNet-19](https://www.geeksforgeeks.org/vgg-net-architecture-explained/#additional-information-about-vggnet19)
* [Conclusion](https://www.geeksforgeeks.org/vgg-net-architecture-explained/#conclusion)

## Evolution of VGG Models

Before the advent of VGG models, [CNN](https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/) architectures like [LeNet-5](https://www.geeksforgeeks.org/lenet-5-architecture/)and [AlexNet](https://www.geeksforgeeks.org/ml-getting-started-with-alexnet/" \t "_blank) laid the groundwork for deep learning in computer vision. LeNet-5, introduced in the 1990s, was one of the first successful applications of CNNs in recognizing handwritten digits. AlexNet, which won the ILSVRC in 2012, marked a significant breakthrough by leveraging deeper architectures and GPU acceleration.

The VGG models were introduced by Karen Simonyan and Andrew Zisserman in their 2014 paper titled “Very Deep Convolutional Networks for Large-Scale Image Recognition.” The primary objective was to investigate the effect of increasing the depth of CNNs on large-scale image recognition tasks. VGG-16 and VGG-19, with 16 and 19 weight layers respectively, were among the most notable models presented in the paper. Their design was characterized by using small 3×3 convolution filters consistently across all layers, which simplified the network structure and improved performance.

***You can refer to –***[***VGG-16 | CNN model***](https://www.geeksforgeeks.org/vgg-16-cnn-model/)***to study the architecture of VGG-16 Architecture.***

## VGG-19 Architecture

VGG-19 is a deep convolutional neural network with 19 weight layers, comprising 16 convolutional layers and 3 fully connected layers. The architecture follows a straightforward and repetitive pattern, making it easier to understand and implement.

The key components of the VGG-19 architecture are:

1. **Convolutional Layers**: 3×3 filters with a stride of 1 and padding of 1 to preserve spatial resolution.
2. **Activation Function**: ReLU (Rectified Linear Unit) applied after each convolutional layer to introduce non-linearity.
3. **Pooling Layers**: Max pooling with a 2×2 filter and a stride of 2 to reduce the spatial dimensions.
4. **Fully Connected Layers**: Three fully connected layers at the end of the network for classification.
5. **Softmax Layer**: Final layer for outputting class probabilities.

## Detailed Layer-by-Layer Architecture of VGG-Net 19

The VGG-19 model consists of five blocks of convolutional layers, followed by three fully connected layers. Here is a detailed breakdown of each block:

### ****Block 1****

* Conv1\_1: 64 filters, 3×3 kernel, ReLU activation
* Conv1\_2: 64 filters, 3×3 kernel, ReLU activation
* Max Pooling: 2×2 filter, stride 2

### ****Block 2****

* Conv2\_1: 128 filters, 3×3 kernel, ReLU activation
* Conv2\_2: 128 filters, 3×3 kernel, ReLU activation
* Max Pooling: 2×2 filter, stride 2

### ****Block 3****

* Conv3\_1: 256 filters, 3×3 kernel, ReLU activation
* Conv3\_2: 256 filters, 3×3 kernel, ReLU activation
* Conv3\_3: 256 filters, 3×3 kernel, ReLU activation
* Conv3\_4: 256 filters, 3×3 kernel, ReLU activation
* Max Pooling: 2×2 filter, stride 2

### ****Block 4****

* Conv4\_1: 512 filters, 3×3 kernel, ReLU activation
* Conv4\_2: 512 filters, 3×3 kernel, ReLU activation
* Conv4\_3: 512 filters, 3×3 kernel, ReLU activation
* Conv4\_4: 512 filters, 3×3 kernel, ReLU activation
* Max Pooling: 2×2 filter, stride 2

### ****Block 5****

* Conv5\_1: 512 filters, 3×3 kernel, ReLU activation
* Conv5\_2: 512 filters, 3×3 kernel, ReLU activation
* Conv5\_3: 512 filters, 3×3 kernel, ReLU activation
* Conv5\_4: 512 filters, 3×3 kernel, ReLU activation
* Max Pooling: 2×2 filter, stride 2

### Fully Connected Layers

* **FC1**: 4096 neurons, ReLU activation
* **FC2**: 4096 neurons, ReLU activation
* **FC3**: 1000 neurons, softmax activation (for 1000-class classification)

## Architectural Design Principles

The VGG-19 architecture follows several key design principles:

1. **Uniform Convolution Filters**: Consistently using 3×3 convolution filters simplifies the architecture and helps maintain uniformity.
2. **Deep Architecture**: Increasing the depth of the network enables learning more complex features.
3. **ReLU Activation**: Introducing non-linearity helps in learning complex patterns.
4. **Max Pooling**: Reduces the spatial dimensions while preserving important features.
5. **Fully Connected Layers**: Combines the learned features for classification.

## Impact and Legacy of VGG-19

### Influence on Subsequent Models

The simplicity and effectiveness of VGG-19 influenced the design of subsequent deep learning models. Architectures like ResNet and Inception drew inspiration from the depth and uniformity principles established by VGG models. VGG-19’s deep yet straightforward architecture demonstrated that increasing depth could significantly improve performance in image recognition tasks.

### Use in Transfer Learning

VGG-19 has been extensively used in transfer learning due to its robust feature extraction capabilities. Pre-trained VGG-19 models on large datasets like ImageNet are often fine-tuned for various computer vision tasks, including object detection, image segmentation, and style transfer.

### Research and Industry Applications

VGG-19 has found applications in numerous research and industry projects. Its architecture has been used as a baseline in academic research, enabling comparisons with newer models. In industry, VGG-19’s pre-trained weights serve as powerful feature extractors in applications ranging from medical imaging to autonomous vehicles.

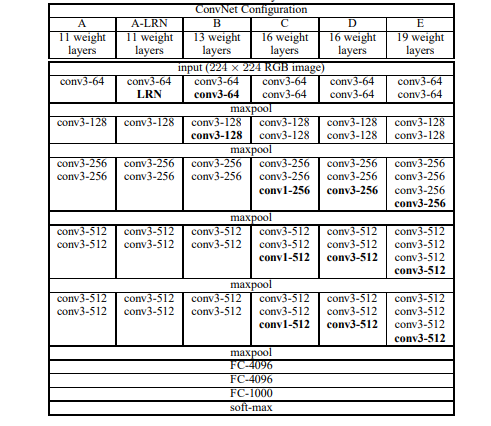
## Additional Information about VGGNet-19

1. **Model Simplicity and Effectiveness:** The VGG-19 architecture’s simplicity, characterized by its uniform use of 3×3 convolution filters and repetitive block structure, makes it a highly effective and easy-to-implement model for various computer vision tasks.
2. **Computational Requirements:**One of the key trade-offs of the VGG-19 model is its computational demand. Due to its depth and the use of small filters, it requires significant memory and computational power, making it more suited for environments with robust hardware capabilities.
3. **Robust Feature Extraction:**The depth of the VGG-19 model allows it to capture intricate features in images, making it an excellent feature extractor. This capability is particularly useful in transfer learning, where pre-trained VGG-19 models are fine-tuned for specific tasks, leveraging the rich feature representations learned from large datasets.
4. **Data Augmentation:** To enhance the performance and generalization capability of VGG-19, data augmentation techniques such as random cropping, horizontal flipping, and color jittering are often employed during training. These techniques help the model to better handle variations and improve its robustness.
5. **Influence on Network Design:**The principles established by the VGG-19 architecture, such as the use of small convolution filters and deep networks, have influenced the design of subsequent state-of-the-art models. Researchers have built upon these concepts to develop more advanced architectures that continue to push the boundaries of what is possible in computer vision.

9. Describe VGGNET CONFIGURATIONS.

#### ****VGG-16 Configuration:****

The main difference between VGG-16 configurations C and D lies in the use of filter sizes in some of the convolutional layers. While both versions predominantly use 3×3 filters, in version D, there are instances where 1×1 filters are used instead. This slight variation results in a difference in the number of parameters, with version D having a slightly higher number of parameters compared to version C. However, both versions maintain the overall architecture and principles of the VGG-16 model.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200217112031/VGG16conf.PNG)

*Different VGG Configuration*

### ****Object Localization In Image:****

To perform localization, we need to replace the class score by bounding box location coordinates. A bounding box location is represented by the 4-D vector (center coordinates(x,y), height, width). There are two versions of localization architecture, one is bounding box is shared among different candidates (the output is 4 parameter vector) and the other is a bounding box is class-specific (the output is 4000 parameter vector). The paper experimented with both approaches on VGG -16 (D) architecture. Here we also need to change loss from classification loss to regression loss functions (such as [MSE](https://www.geeksforgeeks.org/python-mean-squared-error/)) that penalize the deviation of predicted loss from the ground truth.

**Results:** VGG-16 was one of the best performing architectures in the ILSVRC challenge 2014.It was the runner up in the classification task with a top-5 classification error of 7.32% (only behind GoogLeNet with a classification error of 6.66%). It was also the winner of localization task with 25.32% localization error.

### ****Limitations Of VGG 16:****

* It is very slow to train (the original VGG model was trained on Nvidia Titan GPU for 2-3 weeks).
* The size of VGG-16 trained imageNet weights is 528 MB. So, it takes quite a lot of disk space and bandwidth which makes it inefficient.
* 138 million parameters lead to exploding gradients problem.

10. What regularization methods are used in VGGNET to prevent overfitting?

## Regularization Technique

[Regularization](https://www.geeksforgeeks.org/regularization-in-machine-learning/) is a technique in machine learning that helps prevent from overfitting. It works by introducing penalties term or constraints on the model’s parameters during training. These penalties term encourage the model to avoid extreme or overly complex parameter values. By doing so, regularization prevents the model from fitting the training data too closely, which is a common cause of overfitting. Instead, it promotes a balance between model complexity and performance, leading to better generalization on new, unseen data.

### How Regularization used to prevent overfitting

1. By introducing the regularization term in loss function that act like a constrain function of the model’s parameter. This function penalize certain parameter values in model, discouraging them from becoming too large or complex.
2. Regularization introduces a trade-off between fitting the training data and keeping the model’s parameters small. The strength of regularization is controlled by a hyperparameter, often denoted as lambda (λ). A higher λ value leads to stronger regularization and a simpler model.
3. Regularization techniques help control the complexity of the model. They make the model more robust by constraining the parameter space. This results in smoother decision boundaries in the case of classification and smoother functions in regression, reducing the potential for overfitting.
4. Regularization oppose overfitting by discouraging the model from fitting the training data too closely. It prevents parameters from taking extreme values, which might be necessary to fit the training data.

Let’s discuss about two common techniques that involve in regularization which can prevent model from overfitting

* **L1 Regularization**
* **L2 Regularization**

### L1 Regularization

[L1 regularization](https://www.geeksforgeeks.org/implementation-of-lasso-regression-from-scratch-using-python/), also known as Lasso **(Least Absolute Shrinkage and Selection Operator)** regularization, is a statistical technique used in machine learning to avoid overfitting. It is used to add a penalty term to the model’s [loss function](https://www.geeksforgeeks.org/ml-common-loss-functions/). This penalty term encourages the model to keep some of its coefficients exactly equal to zero, effectively performing feature selection. L1 regularization is employed to prevent overfitting, simplify the model, and enhance its generalization to new, unseen data. It is particularly useful when dealing with datasets containing many features, as it helps identify and focus on the most essential ones, disregarding less influential variables.

Let’s derive the mathematical formulation for L1 regularization (Lasso) in simple terms.

In linear regression, the standard model’s goal is to minimize the [mean squared error](https://www.geeksforgeeks.org/python-mean-squared-error/) (MSE), represented as:

In the above equation, ‘y’ is the actual target, and ‘ŷ’ is the predicted target. Now, to add L1 regularization, we introduce a new term to the model’s loss function:

Here, ‘w’ represents the model’s coefficients, and ‘α’ is the regularization strength, a hyperparameter that controls how much regularization is applied. The term **Σ|w|** sums up the **absolute values of all the coefficients**.What this additional term does is encourage the model to have some of its **coefficients exactly equal to zero**. It’s like a feature selector it helps the model pick only the most important features, effectively ignoring the less relevant ones.

The larger the ‘α’ value, the stronger the regularization, and the more coefficients will become zero. This balance between the **MSE (model fitting the data)**and the **Σ|w| term (coefficient sparsity)** ensures that the model remains relatively simple and less prone to overfitting while still fitting the data effectively.