Task-1:- Paper Review

ImageNet Classification with Deep Convolutional Neural Networks

Introduction:

The paper was published by the authors *Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton* in 2012. The report mainly focuses on the Deep Convolutional Neural Network architecture used to classify the 1.3 million high-resolution images in the LSVRC-2010 ImageNet training set into 1000 classes.

What are Convolutional Neural Networks?

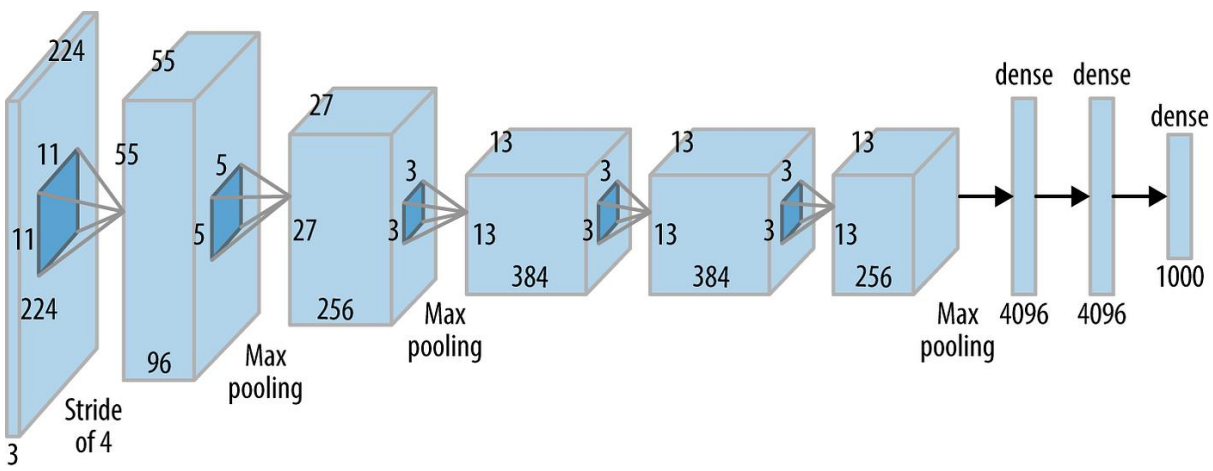
Convolutional Neural Networks (CNNs) are designed to work with grid-structured inputs like black-white images (2D-Structured images). This data type also exhibits spatial dependencies because adjacent spatial locations in an image often have similar colors, creating 3-dimensional input.

An essential defining characteristic of CNN is an operation that is referred to as *Convolution.* It is a dot product operation between a grid-structured set of weights a similar grid-structured inputs drawn from different spatial localities in the input volume. It is helpful for data with a high level of spatial locality, like image data. Hence, we call these networks Convolutional Neural Networks because they use this operation at least in one layer. Although, in most of the scenarios, more than one layer is observed to perform this operation.

About the Dataset:

The dataset used for training this network is the ImageNet dataset which has over 15 million labeled high-dimensioned images corresponding to 22,000 different classes labeled by human labelers using Amazon's Mechanical Turk crowd-sourcing tool. But to train this neural network, authors have used a subset of the dataset with nearly 1.2 million training images, 50,000 validation images, and 1,50,000 test images. Also, the dimension of each image is fixed at 256 × 256. Any other shape of the image is pre-processed to this shape dimension and then fed into the network as input.

About the Architecture:



Source: Structure of AlexNet Architecture that matches to network mentioned in the paper. Link

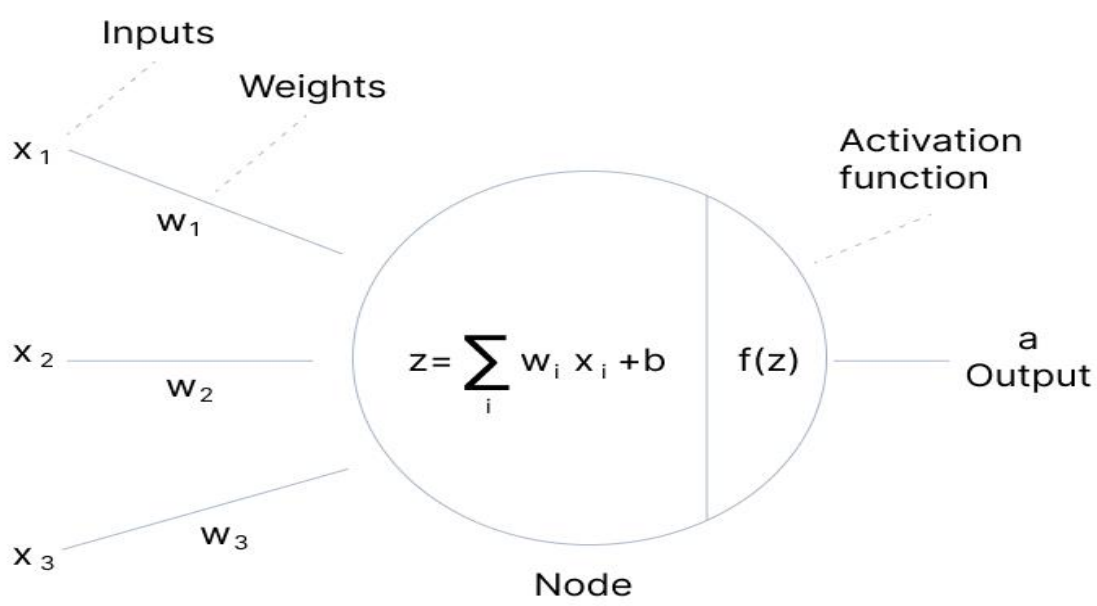
The network mentioned in the paper corresponds to AlexNet architecture (if not assumed) with eight layers, including the output layer [3]. The first five of these eight layers correspond to Convolutional layers, and the last three are part of a simple Artificial Neural Network (ANN). The network was trained on two parallel pipelines caused by two GPUs that worked together to build the training model with faster speed and memory sharing. The network was trained on GTX 580 GPU with 3GB memory. The parallelization approach we use places half of the kernels (or neurons) on each GPU, with one added trick: the GPUs only interact at certain levels. This implies that the kernels of layer 3, for example, accept input from all kernel maps in layer 2. However, kernels in layer four only get input from kernel maps in layer three that are located on the same GPU.

What are the Activation Functions, and why do we need them?

In simple words, the activation function is a normal mathematical function that helps decide whether the input into a neuron is essential. Another vital function is transforming the weighted input into an output value that must be fed into the next layer. The following image depicts the role of the activation function in deep neural networks. Now answering the why part: the activation function adds to a non-linearity in the neural network. If we do not include an activation function, i.e., use linear transformation between the layers of weights and bias; all the layers will behave similarly without knowing how many hidden layers we have [2]. Hence, it is worth having an activation function.

There are several activation functions available like $f(x) = tanh(x)$, $f(x) = (1+e^{-x})^{-1}$, $f(x) = max(0, x)$, $f(x) = |tanh(x)|$. They have used the Rectified Linear Unit (ReLU) activation function in this architecture. The reason for using the ReLU activation function is due to its lower processing time. It was observed that the CNN architectures having ReLU activation functions converged faster than the other activations. An experiment aimed to study the convergence of training error rate, which concluded that the architectures with ReLU activation converged six times faster than one with tanh activation. Also, we don't need to normalize values before feeding into the ReLU activation.

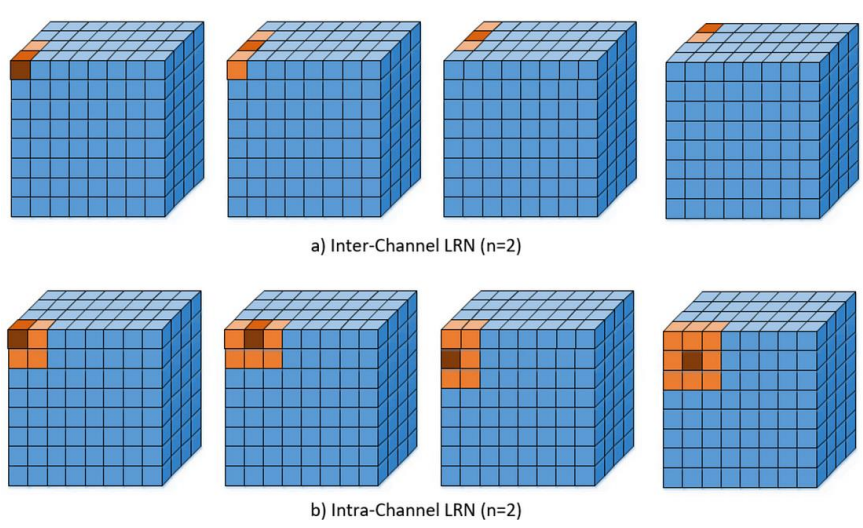
ReLU activation function:- f(x) = max(0, x)



Source:- Activation function f(z) taking weighted sum z as input and giving output a, Link

Local Response Normalisation:

LRN is a non-trainable layer that square-normalizes the pixel values in a feature map within a local neighborhood [4]. There are two types of LRN based on neighborhood, as depicted in the following image:



Source: Types of LRN, Link

The paper has used Inter-Channel LRN, where the neighborhood is defined across the channels. For the (x,y) position, the normalization is carried out by

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a^j_{x,y})^2 \right)^{\beta}$$

Source: Inter-channel LRN, Link

Where i indicates the output of filter i, an (x,y), b(x,y) the pixel values at (x,y) position before and after normalization respectively, and N is the total number of channels, the constants (k,α,β,n) are hyper-parameters, k is used to avoid any singularities (division by zero), α is used as a normalization constant, while β is a contrasting constant. The constant n defines the neighborhood length, i.e., how many consecutive pixel values must be considered while normalizing. The case of (k, α, β, n)=(0, 1, 1, N) is the standard normalization) [4]. In the figure above, n is taken to be two while N=4. Apart from Inter/Intra-channel LRN, their batch normalization exists as well.

$$b_{x,y}^{k} = a_{x,y}^{k} / \left(k + \alpha \sum_{i=max(0,x-n/2)}^{min(W,x+n/2)} \sum_{j=max(0,y-n/2)}^{min(H,y+n/2)} (a_{i,j}^{k})^2 \right)^{\beta}$$

Source: Inter-channel LRN, Link

Pooling Operation:

The pooling operation works on the small grid region in each layer. The maximum values are returned when using the max pooling technique for each square region of the size equal to the pooling kernel size. Otherwise, we also have min pooling and average pooling. The pooling operation reduces the size of the original grid, and the new size is given by the formula (L-P+1) × (L-P+1)/s [1] , where L is the size of the input layer and P = 3 is the size of the pooling kernel and assuming the stride of s = 2. In most cases, the stride of 2 or 3 is used along with a kernel size of 3 or 5. Although, in some exceptional cases, a kernel size of P = 7 has also been observed. This pooling operation reduces the error rate by 0.4% to 0.3%. If pooling is not used, there are high chances of overfitting our model.

Data Augmentation:

This technique was used to reduce the overfitting challenge in the network. Data Augmentation refers to artificially making some changes/transformations in the existing dataset (in this case, images) that creates a new set of data, which can be used to train the model and reduces the chances of overfitting. These operations are performed on the CPU rather than GPU, so training operations are not disturbed. The first transformation was of horizontal reflections by extracting 224 × 224 patches from 256 × 256 images, which reduces the size of images by 2048. The second transformation is based on altering the intensities of RGB channels using Principle Component Analysis in the training set. A random variable with mean = 0 and std = 0.1 is chosen and is multiplied with magnitudes corresponding to the eigenvalues of RGB.

$$[p_1 , p_2 , p_3] [\alpha_1\lambda_1 , \alpha_2\lambda_2, \alpha_3\lambda_3]^T$$

where p_i and λ_i are i^{th} eigenvectors and eigenvalue of the 3 × 3 covariance matrix of RGB pixel values, respectively, and αi is the random variable. This reduces the top-1 error rate by 1%.

Dropout:

This technique refers to setting the output as zero for neurons with a probability less than or equal to 0.5. these dropped neurons don't participate in backpropagation and forward pass operations, reducing overfitting. But consequently, dropout increases the number of iterations required to converge. This technique forces us to learn the parameters from other different neurons rather than fixed ones.

Training Neural Architecture:

- The model was trained using Stochastic Gradient Descent (SGD), with a batch size of 128, momentum of 0.9, and weight decay of 0.0005, reducing the training error. The update rule was as follows:

$$\begin{aligned} v_{i+1} &:= 0.9 \cdot v_i - 0.0005 \cdot \epsilon \cdot w_i - \epsilon \cdot \left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i} \\ w_{i+1} &:= w_i + v_{i+1} \end{aligned}$$

  where i is the iteration index, v is the momentum variable, ϵ is the learning rate and $(\frac{\partial L}{\partial w}\Big| w_i)D_i$ is average over ith batch D_i of the derivative of objective with respect to w.
- The weights were initialised using Gaussian Distribution with mean = 0 and standard deviation = 0.01.
- The bias terms were set to 1 for all the second, fourth, and fifth convolutional layers and all fully connected layers, and for the remaining layers, the bias was set to 0.
- The learning rate was initialized to 0.01 and was reduced manually three times when the validation error stopped improving with the current learning rate.

Results & Conclusions:

The model was submitted in ILSVRC-2010 & ILSVRC-2012 competitions and achieved 37.5% & 17% top1 and top5 error rates. This shows that large and deep convolutional neural networks can achieve pleasing results on highly complicated datasets, especially supervised learning datasets.

To summarise, the paper introduced several effective techniques, including deep CNN architectures, ReLU activations, GPU computing, and data augmentation. These techniques have since become standard practices in the deep learning community and have been instrumental in various other computer vision tasks. This architecture has become a basis on which several different architectures like GoogLeNet, VGGNet, and ResNet are built. These architectures now have shown excellent performance on the ImageNet dataset.

Scopes of Improvement:

- **Scaling images on Multiple scales**:- We can train our neural network architecture by scaling images on various scales, which may increase the ability of the network to locate the object at different locations efficiently.
- **Changing the pooling techniques**:- The paper has used the max-pooling technique for feature detection. Using average, fractional, and spatial pooling may improve the network's learning ability.
- **Using other Normalisation Techniques:-** As mentioned earlier (in the LRN section), the network used inter-channel Local Response Normalisation techniques. Instead, we can also use intra-channel Local Response Normalisation or batch Normalisation for the same.[1]
- **Ensemble Learning Approach**:- Instead of creating only one neural network architecture, now that we have much-advanced GPUs, we can build multiple networks with a different number of layers and use various hyperparameters (mentioned in earlier points) and then combine the results from all networks to predict the final answer. This will increase training time, but we can achieve good accuracy using this approach.

References:

1. C. C. Aggarwal, *Neural Networks and Deep Learning: A Textbook*. Springer, 2018.

2. J. W. Siegel and J. Xu, "Approximation rates for neural networks with general activation functions," *Neural Networks*, vol. 128, pp. 313–321, Aug. 2020, doi: 10.1016/j.neunet.2020.05.019. [Online]. Available: http://dx.doi.org/10.1016/j.neunet.2020.05.019

3. Atulanand, "AlexNet Complete Architecture," *Medium*, Nov. 09, 2022. [Online]. Available: https://medium.com/codex/alexnet-complete-architecture-dc3a9920cdd

4. Anwar, "Difference between Local Response Normalization and Batch Normalization," *Medium*, Apr. 07, 2021. [Online]. Available: https://towardsdatascience.com/difference-between-local-response-normalization-and-batch-normalization-272308c034ac

Here is the link to Code for AlexNet Architecture (using libraries Keras and TensorFlow).

Task-2:- Model Selection

Predicting Bacterial Property:

Before selecting the model to predict properties, let's list different models that can be used to predict.

- Artificial Neural Networks (not using but just mentioned it)
- Simple Linear Regression
- Multiple Linear Regression
- Decision Trees Regressor
- Random Forest Regressor

Since we are trying to avoid Neural Networks, we will not consider ANN for discussions (another reason can also be the small size of data, i.e., only 20,000, which may lead to overfitting of the network).

Simple Linear Regression:- Since we have 150 columns in the dataset, there may be better choices than the simple linear regression model. Since it is simple linear regression, the model's training time would be less than other ML models. The simple linear regression tries to find the linear relationship among the columns, which may not exist for all, in this case, 150 columns. Although we may use correlation analysis or feature important analysis to find out the sub-domain of parameters with a strong linear relationship with predicting parameters, that would not be a good fit for test data. And since it could not be a good fit, the values of evaluation metrics like Mean Squared Error, Mean Absolute Error, Root Mean Squared Error, and R^2 Error will be high.

Multiple Linear Regression:- The idea of Simple Linear Regression can be extended up to Multiple Linear Regression. This model uses the idea to find the relation between the independent columns (which, in this case, are various properties of bacteria) and the dependent column (predictive parameter). The relationship is built using a simple linear equation with the parameters equal to independent parameters in the training model and bias. Its main idea is to find optimum values of those parameters corresponding to independent columns. Its training time is less than other complex models like Decision Trees and Random Forest Regressors. We may get low values of errors in this case as compared to Simple Linear Regression. It can be considered a good going-ahead with multiple linear regression.

Decision Tree Regressor:- Decision tree regression builds a tree-like model that predicts the target variable based on a series of binary decisions on the predictor variables. It can capture non-linear relationships and interactions between variables. But I would not suggest moving with a decision tree since we have a vast number of independent columns; this may lead to overfitting of the tree and may not be a good fit for the testing data.

Random Forest Regressor:- However, the idea of a Decision Tree Regressor can be extended to Multiple Decision Tree Regressors and combining its result by taking mean or other mathematical operations. This is called ensemble learning, where we predict the final output from various models and combine them to give the final output. Our dataset has too many features, so it could be time-consuming for Random Forest Regressor to get trained. Still, talking about its accuracy/error values, we can get the best results (i.e., low values of errors mentioned above). Since it is one of the most powerful machine learning models, it can be considered the best fit for the given data [1]. In the cited paper, we can see various models were deployed out of which Random Forest Regressor showed highest accuracy (nearly 80%) followed by linear regression (73.5%) and Decision Tree (73.4%)

Predicting the number of people on the beach:

Again we have same models available in this case as well. But I would go ahead with Support Vector Regression in this case because this dataset has nearly ten independent features, and most probably, they may have non-linear relationships between them, which can be checked using statistical methods like covariance analysis, correlation analysis, and SVR can capture non-linear relationships by mapping the weather features into a higher-dimensional space. It finds a hyperplane that maximizes the margin around the predicted crowd size values. SVR is adequate for capturing complex patterns in the data. The values of evaluation metrics like MAE, MSE, and RMSE will be low in this model compared to the Simple Decision Tree model, Simple/Multiple Linear Regression model. The explanation remains the same as above for other models also. In the same paper, as mentioned earlier, the accuracy of Random Forest Regressor and Support Vector Machines near each other (78.6% & 80% respectively). If we consider a non-linear relationship, then SVM can be the best fit for the data.

Matrix Multiplication:

If you want to optimise for speed in matrix multiplication and have access to a GPU, one of the most effective approaches is to use GPU-accelerated libraries and frameworks to do the matrix multiplication, such as cuBLAS (for NVIDIA GPUs) or ROCm (for AMD GPUs). These libraries are designed for parallel calculations and can considerably accelerate the process, particularly for big

matrices. You can gain quicker results by shifting matrix multiplication operations to the GPU, which has parallel computing capacity. If you don't have access to a GPU but still want to optimise for performance, you can use highly optimised CPU-based linear algebra libraries like Intel MKL, OpenBLAS, or ATLAS. These libraries have been optimized to take advantage of CPU-specific optimizations and efficiently execute matrix multiplication on the CPU.

If the aim is to optimize for accuracy, the approach or library used is less important as long as it adheres to the standard mathematical definition of matrix multiplication. A typical matrix multiplication method, such as the naive technique, or optimized variations such as blocked matrix multiplication, would be appropriate in this scenario. These algorithms produce accurate results by employing matrix multiplication mathematics concepts.

References

[1] M. M. Churpek, T. C. Yuen, C. Winslow, D. O. Meltzer, M. W. Kattan, and D. P. Edelson, "Multicenter Comparison of Machine Learning Methods and Conventional Regression for Predicting Clinical Deterioration on the Wards," *PubMed Central (PMC)*. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4736499/

Task-3: Mathematics

Gaussian Processes

Click **Here** for the theoretical implementation pdf

Click **Here** for python implementation of Gaussian Process

Task-4: Implementation

UNet-Architecture

Click **Here** for the Jupyter Notebook