Assignment No: 04

Aim: To implement Autoencoder for anomaly detection.

.

Problem Statement:

Use Autoencoder to implement anomaly detection. Build the model by using:

a. Import required libraries

b. Upload / access the dataset

c. Encoder converts it into latent representation

d. Decoder networks convert it back to the original input

e. Compile the models with Optimizer, Loss, and Evaluation Metrics.

Objectives:

a) Apply Autoencoder deep learning architecture to determine anomalies in input dataset

b) Evaluate Model.

Theory:

Autoencoders are very useful in the field of unsupervised machine learning. You can use them to compress the data and reduce its dimensionality. The main difference between Autoencoders and Principle Component Analysis (PCA) is that while PCA finds the directions along which you can project the data with maximum variance, Autoencoders reconstruct our original input given just a compressed version of it. If anyone needs the original data can reconstruct it from the compressed data using an Autoencoder. Autoencoders are mainly a dimensionality reduction (or compression) algorithm with a couple of important properties:

- Data-specific: Autoencoders are only able to meaningfully compress data similar to what they have been trained on. Since they learn features specific for the given training data, they are different than a standard data compression algorithm like gzip. So we can't expect an Autoencoder trained on handwritten digits to compress landscape photos.
- Lossy: The output of the Autoencoder will not be exactly the same as the input, it will be a close but degraded representation. If you want lossless compression they are not the way to go. Unsupervised: To train an Autoencoder we don't need to do anything fancy, just throw the raw input data at it. Autoencoders are considered an *unsupervised* learning technique since they don't need

explicit labels to train on. But to be more precise they are *self-supervised* because they generate theirown labels from the training data.

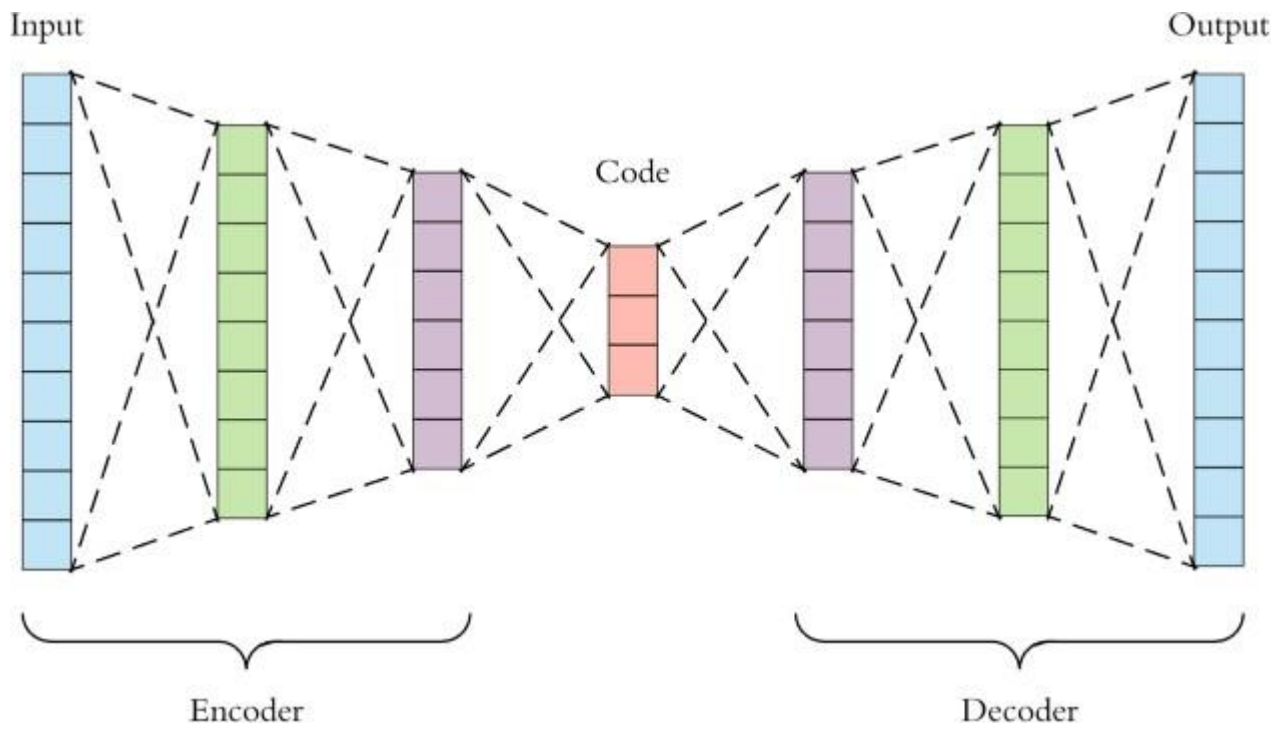
Architecture

An Autoencoder is a type of neural network that can learn to reconstruct images, text, and other data from compressed versions of themselves.

An Autoencoder consists of three layers:

1. Encoder
2. Code
3. Decoder

The Encoder layer compresses the input image into a latent space representation. It encodes the input image as a compressed representation in a reduced dimension. The compressed image is a distorted version of the original image.



The Code layer represents the compressed input fed to the decoder layer. The decoder layer decodes the encoded image back to the original dimension. The decoded image is reconstructed from latent space representation, and it is reconstructed from the latent space representation and is a lossy reconstruction of the original image.

There are 4 hyperparameters that we need to set before training an Autoencoder:

- Code size: number of nodes in the middle layer. Smaller size results in more compression.
- Number of layers: the Autoencoder can be as deep as we like. In the figure above we have 2 layers in both the encoder and decoder, without considering the input and output.
- Number of nodes per layer: the Autoencoder architecture is called a *stacked Autoencoder* since the layers are stacked one after another. The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder. Also the decoder is symmetric to the encoder in terms of layer structure. As noted above this is not necessary and we have total control over these parameters.
- Loss function: we either use *mean squared error (MSE)* or *binary cross entropy*. If the input values are in the range [0, 1] then we typically use cross entropy, otherwise we use the mean squared error.

Training of Autoencoder

Training of an Auto-encoder for data compression: For a data compression procedure, the most important aspect of the compression is the reliability of the reconstruction of the compressed data. This requirement dictates the structure of the Auto-encoder as a bottleneck.

Step 1: Encoding the input data The Auto-encode first tries to encode the data using the initialized weights and biases.

Step 2: Decoding the input data The Auto-encoder tries to reconstruct the original input from the encoded data to test the reliability of the encoding.

Step 3: Backpropagating the error after the reconstruction, the loss function is computed to determine the reliability of the encoding. The error generated is backpropagated.

The above-described training process is reiterated several times until an acceptable level of reconstruction is reached.

After the training process, only the encoder part of the Auto-encoder is retained to encode a similar type of data used in the training process. The different ways to constrain the network are:-

- Keep small Hidden Layers: If the size of each hidden layer is kept as small as possible, then the network will be forced to pick up only the representative features of the data thus encoding the data.
- Regularization: In this method, a loss term is added to the cost function which encourages the network to train in ways other than copying the input.
- Denoising: Another way of constraining the network is to add noise to the input and teach thenetwork how to remove the noise from the data.

- Tuning the Activation Functions: This method involves changing the activation functions of various nodes so that a majority of the nodes are dormant thus effectively reducing the size of the hidden layers.

The different variations of Auto-encoders are:-

- Denoising Auto-encoder: This type of auto-encoder works on a partially corrupted input and trains to recover the original undistorted image. As mentioned above, this method is an effective way to constrain the network from simply copying the input.
- Sparse Auto-encoder: This type of auto-encoder typically contains more hidden units than the input but only a few are allowed to be active at once. This property is called the sparsity of the network.The sparsity of the network can be controlled by either manually zeroing the required hidden units, tuning the activation functions or by adding a loss term to the cost function.
- Variational Auto-encoder: This type of auto-encoder makes strong assumptions about the distribution of latent variables and uses the Stochastic Gradient Variational Bayes estimator in the trainingprocess.

Conclusion:

Autoencoders can be used as an anomaly detection algorithm when we have an unbalanced dataset where we have a lot of good examples and only a few anomalies. Autoencoders are trained to minimize reconstruction error. When we train the Autoencoders on normal data or good data, we can hypothesis that the anomalies will have higher reconstruction errors than the good or normal data.

Assignment No: 05

Title: Implement the Continuous Bag of Words (CBOW) Model

Assignment No: 05

Aim: Implement the continuous Bag of Words (CBOW) model for text recognition in given dataset.

Problem Statement:

Implement the Continuous Bag Of Words (CBOW) Model. Task to build the model are

.

a. Data preparation

b. Generate training data

c. Train model

d. Output

Objective:

- Build CBOW model to predict the current word given context words within a specific window.
- Evaluate Model.

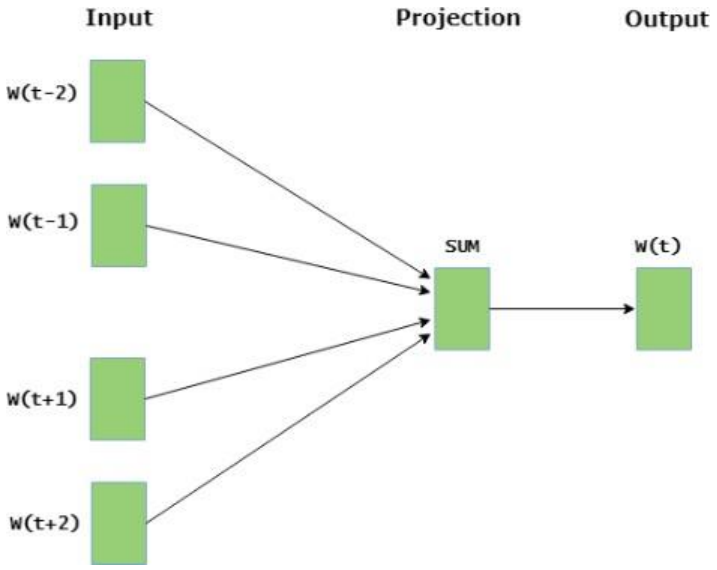
Theory:

Word Embedding is a language modeling technique used for mapping words to vectors of real numbers. Word embedding can be generated using various methods like neural networks, co-occurrence matrix,probabilistic models, etc. The basic idea of word embedding is words that occur in similar context tend to be closer to each other in vector space. Word2Vec consists of models for generating word embedding. These models are shallow two-layer neural networks having one input layer, one hidden layer, and one output layer.Word2Vec utilizes two architectures:

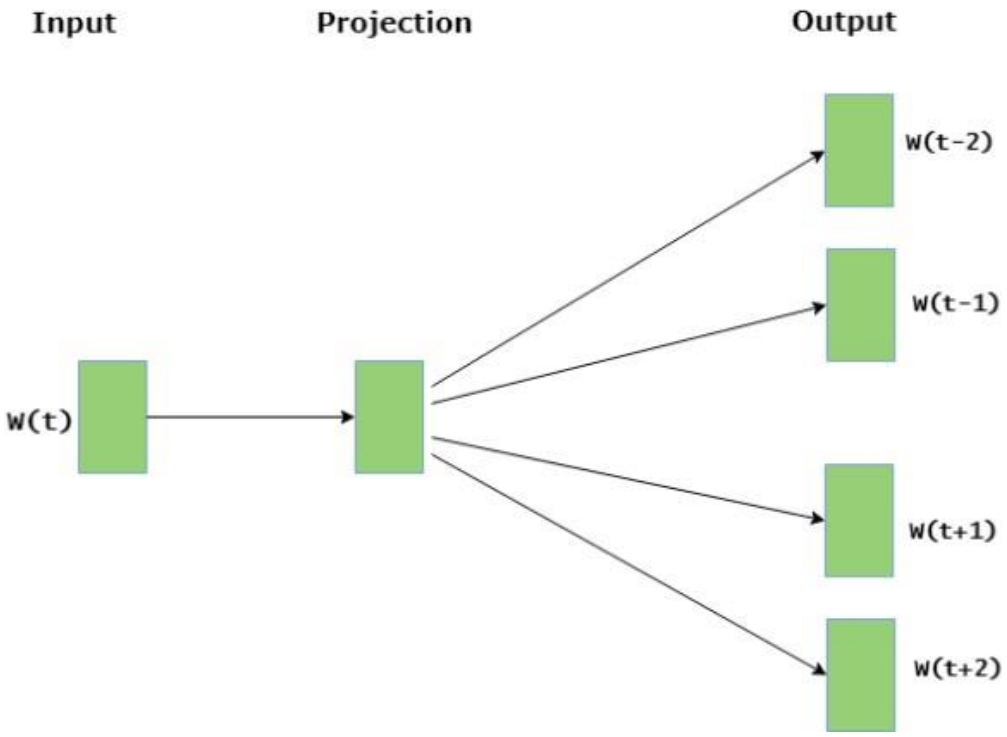
1. CBOW (Continuous Bag of Words)
2. Skip Gram

CBOW (Continuous Bag of Words): The CBOW model tries to understand the context of the words and takes this as input. It then tries to predict words that are contextually accurate.

1. CBOW model predicts the current word given context words within a specific window.
2. The input layer contains the context words and the output layer contains the current word.
3. The hidden layer contains the number of dimensions in which we want to represent the current word present at the output layer.



- **Skip Gram :** Skip gram predicts the surrounding context words within specific window given current word. The input layer contains the current word and the output layer contains the context words. The hidden layer contains the number of dimensions in which we want to represent current word present at the input layer.



The basic idea of word embedding is words that occur in similar context tend to be closer to each other in vector space. For generating word vectors in Python, modules needed are nltk and gensim. Run these commands in terminal to install nltk and gensim:

- pip install nltk
- pip install gensim

Following are the steps to implement the CBOW Model,

Step 1: Download or collect the dataset from https://www.gutenberg.org/files/11/11-0.txt

Step 2: Remove all special characters and digits

Step 3: Remove all punctuation marks

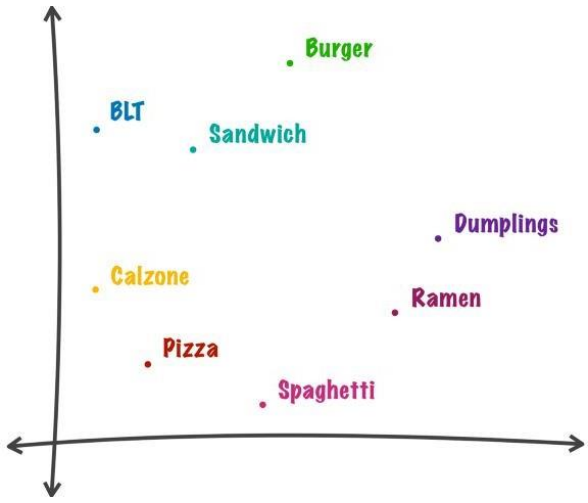
Step 4: Bring all letters to lower case

Step 5: Tokenize the document into sentences and words

Step 6: Remove all stop words.

Step 7: Train the model (Use Gensim word2vec for training the model)

To make words understood by machine learning algorithms, word embedding is used to map words into vectors of real numbers. There are various word embedding models and word2vec is one of them. In simple words, word2vec is a group of related models that are used to produce word embedding. These models are trained to construct the linguistic contexts of words. Word2vec takes a large corpus of text and produces a vector space, with each unique word in the corpus being assigned to a corresponding vector in the space.



To understand how word2vec works, let's explore some methods that we can use with word2vec such as finding similarities between words or solving analogies like this

$$f(\text{"canada"}) - f(\text{"us"}) = f(\text{"??"}) - f(\text{"hamburger"})$$

Step 8: Load the dataset and Model

Step 9: Find the most similar Words

Now we use model.most_similar( ) to find the top-N most similar words. Positive words contribute positively towards the similarity, negative words negatively. This method computes cosine similarity between a simple mean of the projection weight vectors of the given words and the vectors for each word in the model. The method corresponds to the word-analogy and distance scripts in the original word2vec implementation. If topn is False, most_similar returns the vector of similarity scores. restrict_vocab is an optional integer which limits the range of vectors which are searched for most-similar values. For example, restrict_vocab=10000 would only check the first 10000 word vectors in the vocabulary order. (This may be meaningful if you've sorted the vocabulary by descending frequency.)

Step 9: Predict the output word [Use predict_output_word (context_word_list, topn=10) function]

Conclusion:

Thus we have implemented the CBOW model on collected dataset and found that word embedding approach is very useful for text recognition. CBOW can also be apply to convert the speech to text and has many more application in natural language processing.

Assignment No: 06

Title: Object detection using Transfer Learning of CNN architectures

Assignment No: 06

Aim: Implementation of object detection using transfer learning of CNN architectures

Problem Statement:

Implementation of object detection using transfer learning of CNN architecture.

a. Load in a pre-trained CNN model trained on a large dataset
b. Freeze parameters (weights) in model's lower convolutional layers
c. Add custom classifier with several layers of trainable parameters to model
d. Train classifier layers on training data available for task
e. Fine-tune hyperparameters and unfreeze more layers as needed

Objectives:

- To detect the objects in an image by implementing transfer learning deep learning approach using existing CNN architectures.

Theory:

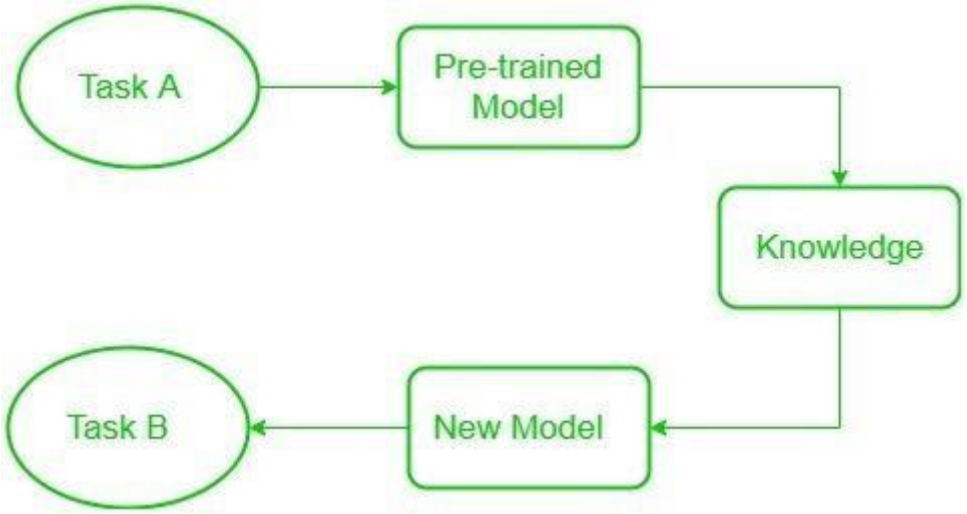
Whenever we encounter a new problem or a task, we recognize it and apply our relevant knowledge from our previous learning experiences. This makes our work easy and fast to finish. For instance, if you know how to ride a bicycle and if you are asked to ride a motorbike which you have never done before. In such a case, our experience with a bicycle will come into play and handle tasks like balancing the bike, steering, etc. This will make things easier compared to a complete beginner. Following the same approach, a term was introduced *Transfer Learning*. This approach involves the use of knowledge that was learned in some task and applying it to solve the problem in the related target task.

Need of transfer learning:

Many deep neural networks trained on images have a curious phenomenon in common: in the early layers of the network, a deep learning model tries to learn a low level of features, like detecting edges, colors, variations of intensities, etc. Such kind of features appears not to be specific to a particular dataset or a task because no matter what type of image we are processing either for detecting a lion or car. In both cases, we have to detect these low-level features. All these features occur regardless of the exact cost function or imagedataset. Thus learning these features in one task of detecting lion can be used in other tasks like detecting humans.

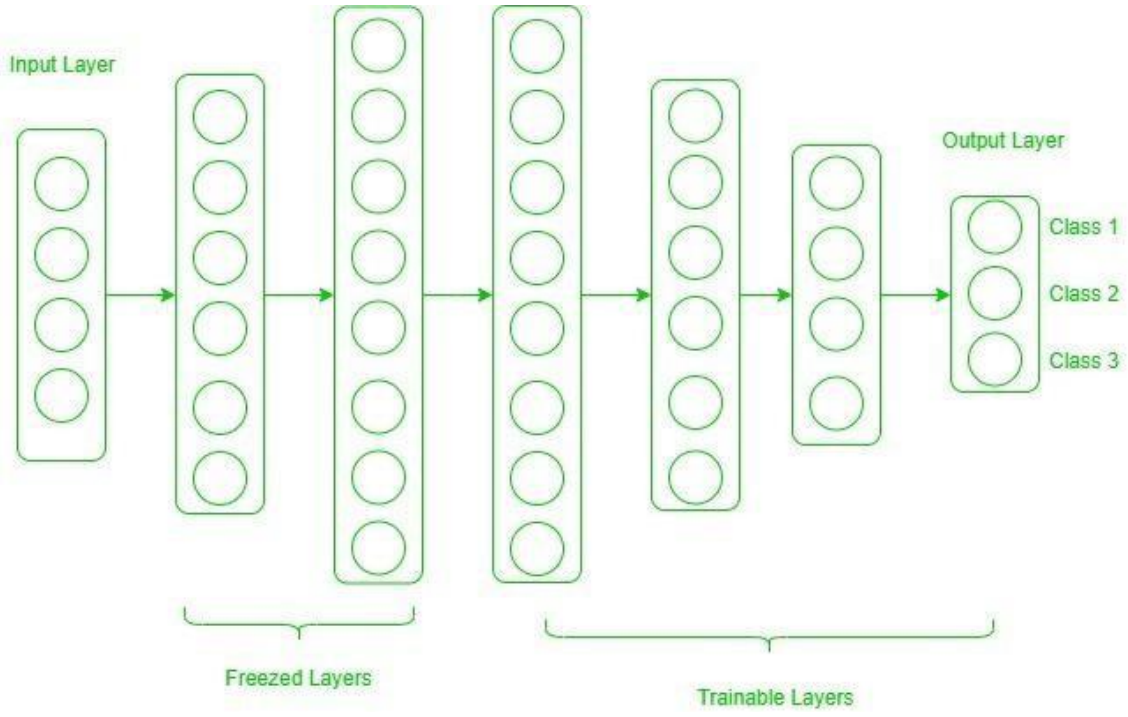
Necessity for transfer learning: Low-level features learned for task A should be beneficial for learning of model for task B. This is what transfer learning is. Nowadays, it is very hard to see people training a whole convolutional neural network from scratch, and it is common to use a pre-trained model trained on a variety of images in a similar task, e.g models trained on ImageNet.

The Block diagram is shown below as follows:



When dealing with transfer learning, we come across a phenomenon called freezing of layers. A layer, it can be a CNN layer, hidden layer, a block of layers, or any subset of a set of all layers, is said to be fixed when it is no longer available to train. Hence, the weights of freezed layers will not be updated during training. While layers that are not freezed follows regular training procedure. When we use transfer learning in solving a problem, we select a pre-trained model as our base model. Now, there are two possible approaches to use knowledge from the pre-trained model. First way is to freeze a few layers of pre-trained model and train other layers on our new dataset for the new task. Second way is to make a new model, but also take out somefeatures from the layers in the pre-trained model and use them in a newly created model. In both cases, we take out some of the learned features and try to train the rest of the model. This makes sure that the only feature that may be same in both of the tasks is taken out from the pre-trained model, and the rest of the model is changed to fit new dataset by training.

Freezed and Trainable Layers:



Now, one may ask how to determine which layers we need to freeze and which layers need to train. The answer is simple, the more you want to inherit features from a pre-trained model, the more you have to freezelayers. For instance, if the pre-trained model detects some flower species and we need to detect some new species. In such a case, a new dataset with new species contains a lot of features similar to the pre-trained model. Thus, we freeze less number of layers so that we can use most of its knowledge in a new model. Now,consider another case, if there is a pre-trained model which detects humans in images, and we want to usethat knowledge to detect cars, in such a case where dataset is entirely different, it is not good to freeze lots of layers because freezing a large number of layers will not only give low level features but also give high-levelfeatures like nose, eyes, etc which are useless for new dataset (car detection). Thus, we only copy low-level features from the base network and train the entire network on a new dataset. Let's consider all situations where the size and dataset of the target task vary from the base network.

Target dataset is small and similar to the base network dataset: Since the target dataset is small, that means we can fine-tune the pre-trained network with target dataset. But this may lead to a problem of overfitting. Also, there may be some changes in the number of classes in the target task. So, in such a casewe remove the fully connected layers from the end, maybe one or two, and add a new fully-connected layer

satisfying the number of new classes. Now, we freeze the rest of the model and only train newly added layers.

Target dataset is large and similar to base training dataset: In such case when the dataset is large and it can hold a pre-trained model there will be no chance of overfitting. Here, also the last full-connected layer is removed, and a new fully-connected layer is added with the proper number of classes. Now, the entire model is trained on a new dataset. This makes sure to tune the model on a new large dataset keeping the model architecture the same.

Target dataset is small and different from the base network dataset: Since the target dataset is different, using high-level features of the pre-trained model will not be useful. In such a case, remove most of the layers from the end in a pre-trained model, and add new layers the satisfying number of classes in a new dataset. This way we can use low-level features from the pre-trained model and train the rest of the layers to fit a new dataset. Sometimes, it is beneficial to train the entire network after adding a new layer at the end.

Target dataset is large and different from the base network dataset: Since the target network is large and different, best way is to remove last layers from the pre-trained network and add layers with a satisfying number of classes, then train the entire network without freezing any layer.

Transfer learning is a very effective and fast way, to begin with, a problem. It gives the direction to move, and most of the time best results are also obtained by transfer learning.

Conclusion:

Thus we have studied to implement transfer learning deep learning approach whereby a neural network model is first trained on a problem similar to the problem that is being solved. One or more layers from the trained model are then used in a new model trained on the problem of interest. We also observed that transferlearning has the benefit of decreasing the training time for a neural network model and can result in lower generalization error.