3 Convolution Neural Network (CNN)

Unit III

Syllabus

Introduction, CNN architecture overview, The Basic Structure of a Convolutional Network- Padding, Strides, Typical Settings, the ReLU layer, Pooling, Fully Connected Layers, The Interleaving between Layers, Local Response Normalization, Training a Convolutional Network

3.1 Introduction

- The ability of artificial intelligence to close the gap between human and computer skills has been growing dramatically. Both professionals and amateurs focus on many facets of the field to achieve great results. The field of computer vision is one of several such disciplines.
- The goal of this discipline is to give robots the ability to see the environment similarly as humans do and to use that understanding for a variety of activities, including image and video recognition, image analysis, media recreation, recommendation systems, natural language processing, etc. With time, one specific algorithm a Convolutional Neural Network has been developed and optimised, largely leading to breakthroughs in computer vision with deep learning.
- The connection pattern between the neurons of convolutional neural networks, an unique kind of feed-forward artificial neural network, is modelled after that of the visual cortex.

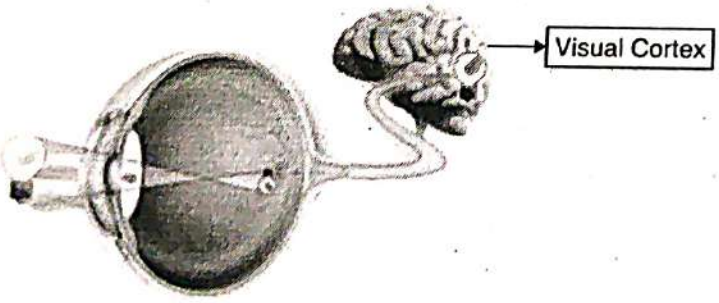


Fig. 3.1.1 : Visual Cortex

- Cells that are sensitive to visual fields are concentrated in a restricted area of the visual cortex. Convolutional Neural Networks were developed as a result of the fact that only some individual neuronal cells in the brain fire when certain orientation edges are present.

- For example, some neurons respond when exposed to vertical edges, while others respond when exposed to horizontal or diagonal edges.
- Convolutional neural networks, or covnets, are nothing more than neural networks that share parameters. Imagine that there is a picture that includes length, width, and height and is represented by a cuboid. Here the dimensions of the image are represented by the Red, Green, and Blue channels, as shown in the image given in Fig. 3.1.2.

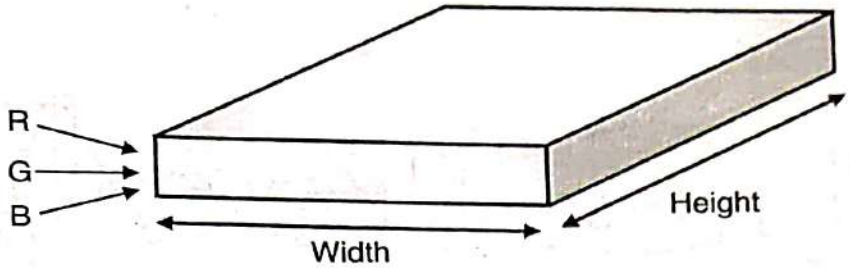


Fig. 3.1.2 : Image channel

- Consider now that we have taken a tiny piece of the same image and applied a little neural network to it. The neural network has k outputs, and each output is shown vertically.
- Now, if we move our little neural network around the image, the output will be a new image with altered width, height, and depth. As opposed to having R, G, and B channels, we now have additional channels that are also narrower and taller. This is the idea behind convolution.
- If we were successful in matching the patch size of the image, it would have been a typical neural network. Because of this little region, we have several wights.

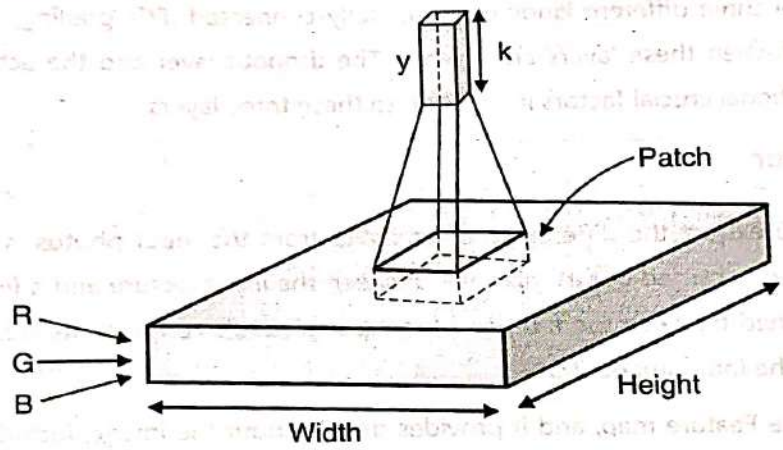


Fig 3.1.3 : Convolution of Image

3.2 CNN Architecture

- A CNN architecture consists of two fundamental components.
- Feature extraction is a procedure that uses a convolution tool to separate and identify the distinct characteristics of a picture for study.
- There are several pairs of convolutional or pooling layers in the feature extraction network.

- A fully connected layer that makes use of the convolutional process's output and determines the class of the picture using the features that were previously extracted.
- This CNN feature extraction model seeks to minimise the quantity of features in a dataset. It generates new features that compile an initial set of features' existing features into a single new feature. As seen in the Fig. 3.2.1 CNN architectural , there are several CNN levels.

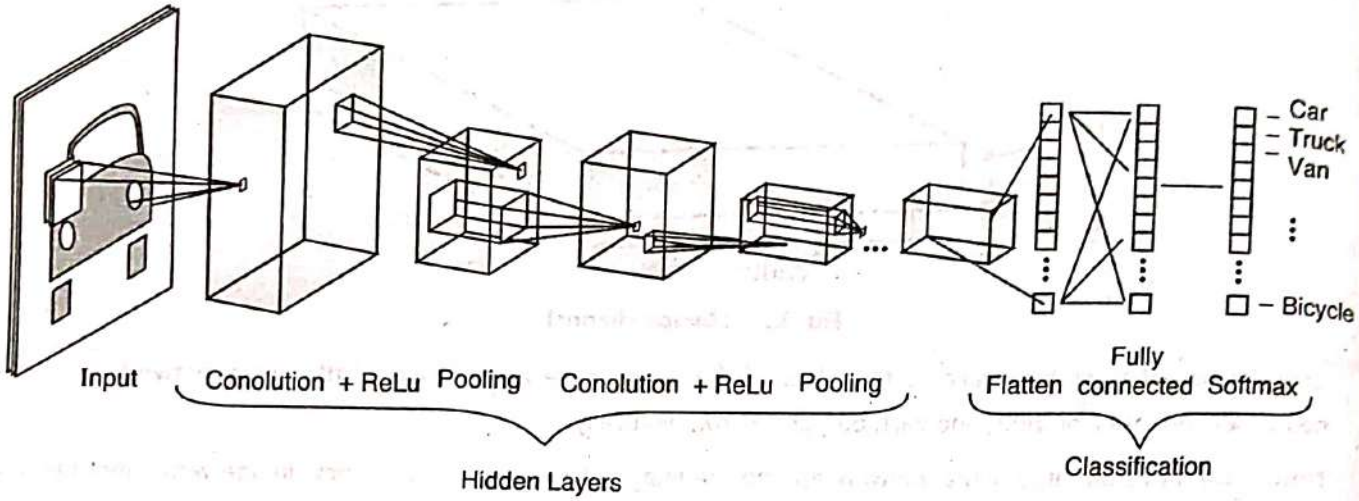


Fig. 3.2.1 : CNN Architecture

3.3 Layers

The CNN is made up of three different kinds of layers: fully-connected (FC), pooling, and convolutional layers. A CNN architecture is created when these layers are layered. The dropout layer and the activation function, which are detailed below, are two additional crucial factors in addition to these three layers.

3.3.1 Convolution Layer

- The first layer utilised to extract the different characteristics from the input photos is this one. Convolution is a mathematical process that is carried out at this layer between the input picture and a filter of a specific size, MxM. The dot product is obtained between the filter and the input picture's components with regard to the filter's size by sliding the filter over the input image (MxM).
- The result is known as the Feature map, and it provides details about the image, including its corners and edges. This feature map is later supplied to further layers to teach them more features from the input picture.
- Once the convolution operation has been applied to the input, CNN's convolution layer transfers the output to the following layer. The spatial link between the pixels is preserved thanks to convolutional layers of CNN.

Let us understand with the help of an example,

- Let's take an image with Dimensions as 5 (Height) x 5 (Breadth) x 1 (Number of channels, eg. RGB).
- A Kernel/Filter, K, is the component that performs the convolution operation in a convolutional layer. K has been chosen as a 3 x 3 x 1 matrix. With a value {101, 010, 101}.

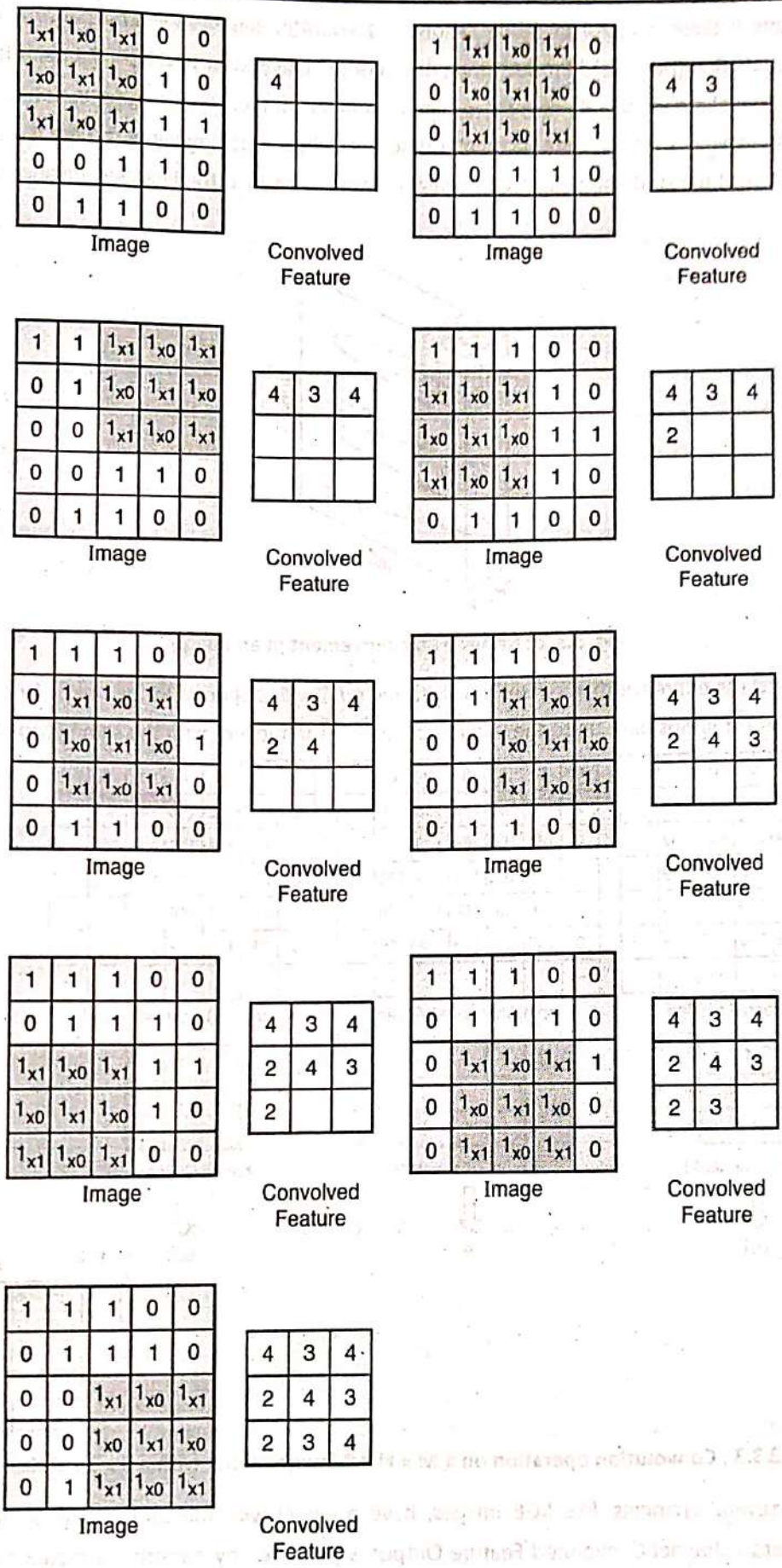


Fig. 3.3.1 : Working of Convolution Layer

- The Kernel shifts 9 times because of Stride Length = 1 (Non-Strided), every time performing an elementwise multiplication operation (Hadamard Product) the portion of the image over which the kernel is hovering. Stride is a component of convolutional neural networks, or neural networks tuned for the compression of images and video data. The neural network's filter's stride parameter determines how much movement there is across the picture or video. For instance, if the stride parameter of a neural network is set to 1, the filter will advance one pixel or unit at a time.

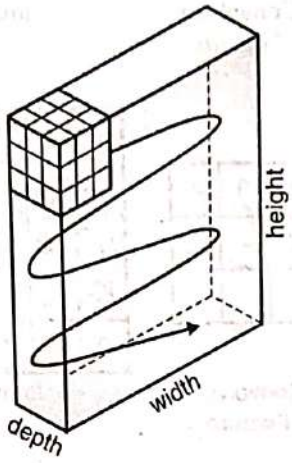


Fig. 3.3.2 : Kernal/Filter movement in an image

- Until it has parsed the entire width, the filter travels to the right with a specific Stride Value. Once the entire picture has been traversed, it jumps back up to the image's beginning (on the left) with the same Stride Value.

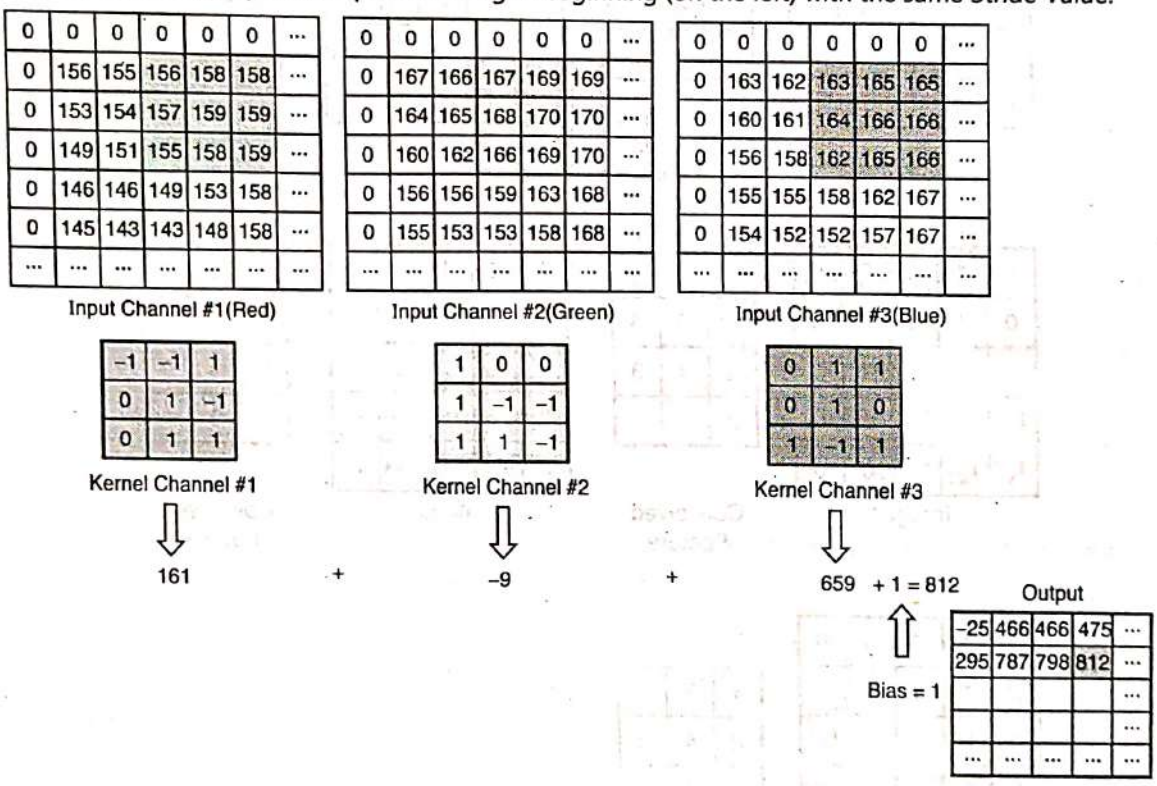


Fig. 3.3.3 : Convolution operation on a M x N x 3 image matrix with a 3 x 3 x 3 Kernel

- Pictures having several channels, like RGB images, have a kernel with the same depth as the input image. A squashed one-depth channel Convoluted Feature Output is produced by performing matrix multiplication across the Kn and In stacks ([K1, I1]; [K2, I2]; and [K3, I3]). All of the outputs are then added together with the bias.

- The Convolution Operation's goal is to take the input image's high-level characteristics, such edges, and extract them. There is no need that ConvNets have only one convolutional layer.
- Typically, low-level characteristics like edges, colour, gradient direction, etc. are captured by the first ConvLayer. With more layers, the architecture also adjusts to the High-Level characteristics, giving us a network that comprehends the dataset's pictures holistically in a way that is comparable to how we do.
- The procedure yields two different sorts of results: one where the dimensionality of the convolved feature is decreased as compared to the input, and the other where it is either increased or stays the same.
- Applying Valid Padding in the first instance or Same Padding in the second accomplishes this.
- When the 5 x 5 x 1 picture is enhanced into a 6 x 6 x 1 image and the 3 x 3 x 1 kernel is applied to it, we see that the convolved matrix has the dimensions 5 x 5 x 1. Therefore, Same Padding.
- Convolutional neural networks (CNNs) can benefit from padding since it describes the number of pixels that are added to an image during processing by the CNN kernel. For instance, if the padding in a CNN is set to zero, then any additional pixels will have a value of 0. But, if the zero padding is set to 1, an additional one-pixel border with a pixel value of zero will be applied to the picture.
- On the other hand, if we carry out the identical operation without padding, we are given a matrix called Valid Padding that has the same dimensions as the kernel itself (3 x 3 x 1).

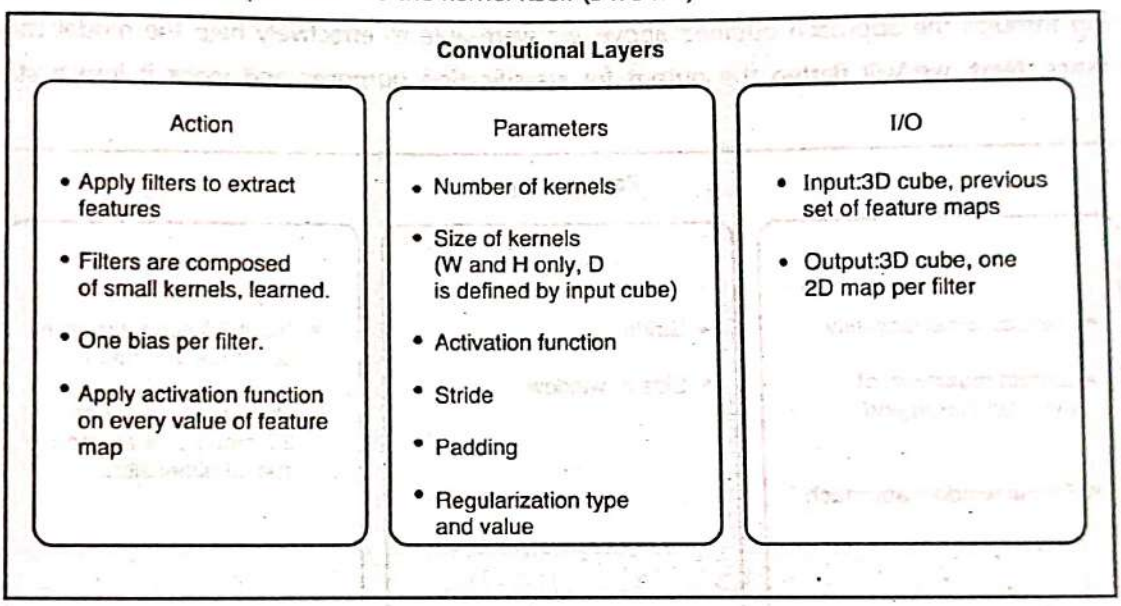


Fig. 3.3.4 : Features of a convolutional layer

3.3.2 Pooling Layer

- The Pooling layer, like the Convolutional Layer, is in charge of shrinking the Convolved Feature's spatial size. Through dimensionality reduction, the amount of computing power needed to process the data will be reduced. Furthermore, it aids in properly training the model by allowing the extraction of dominating characteristics that are rotational and positional invariant.
- Max Pooling and Average Pooling are the two different forms of pooling. The largest value from the area of the picture that the Kernel has covered is returned by Max Pooling. The average of all the values from the area of the picture covered by the Kernel is what is returned by average pooling, on the other hand.

- Additionally, Max Pooling functions as a noise suppressant. It also does de-noising and dimensionality reduction in addition to completely discarding the noisy activations. Average Pooling, on the other hand, only carries out dimensionality reduction as a noise-suppressing strategy. Therefore, we may conclude that Max Pooling outperforms Average Pooling significantly.

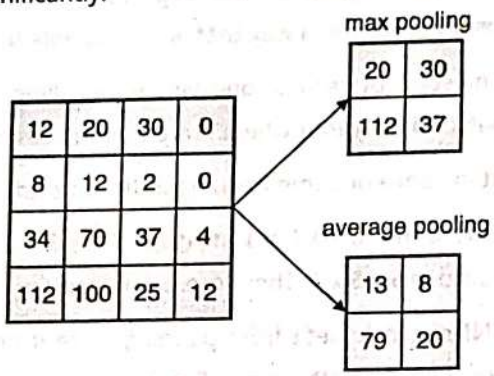


Fig. 3.3.5 : Pooling types

- The i^{th} layer of a convolutional neural network is made up of the convolutional layer and the pooling layer. The number of these layers may be expanded to capture even more minute details, but doing so will need more computer power depending on how complex the pictures are.
- After going through the approach outlined above, we were able to effectively help the model comprehend the characteristics. Next, we will flatten the output for classification purposes and input it into a standard neural network.

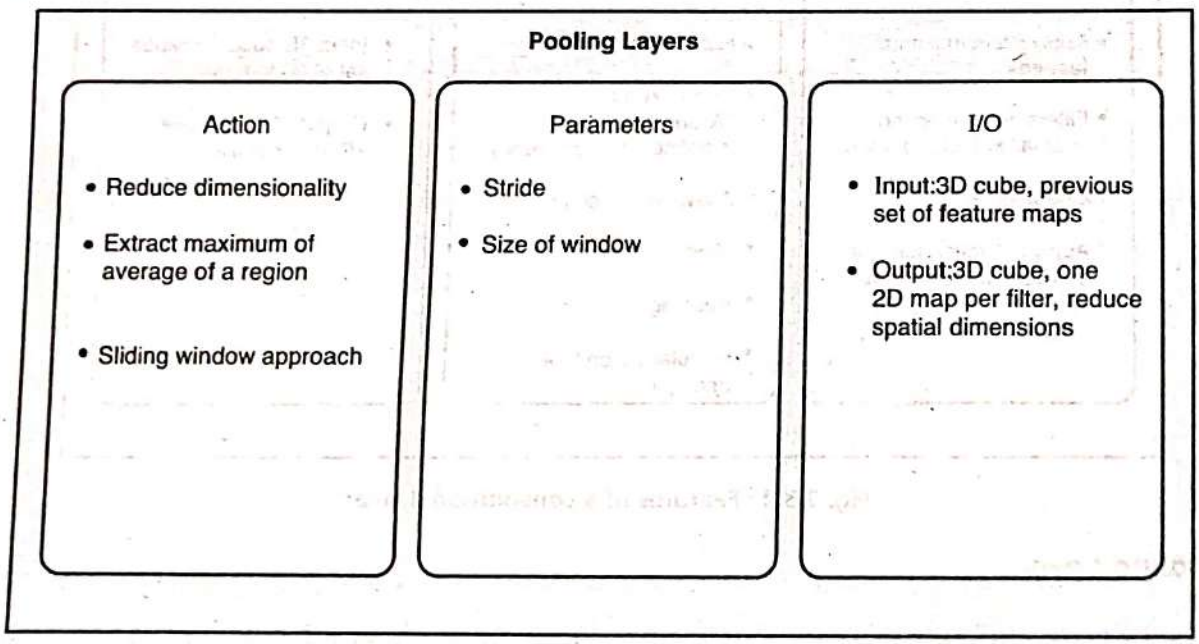


Fig. 3.3.6 : Features of a pooling layer

3.3.3 Classification - Fully Connected Layer (FC Layer)

- A (typically) inexpensive method of learning non-linear combinations of the high-level characteristics represented by the output of the convolutional layer is to add a Fully-Connected layer. In that area, the Fully-Connected layer is now learning a function that may not be linear.

- We will now flatten the input picture into a column vector after converting it to a format that is appropriate for our multi-level perceptron. A feed-forward neural network receives the flattened output, and backpropagation is used for each training iteration.
- The model can categorise pictures using the SoftMax Classification method across a number of epochs by identifying dominant and specific low-level characteristics.

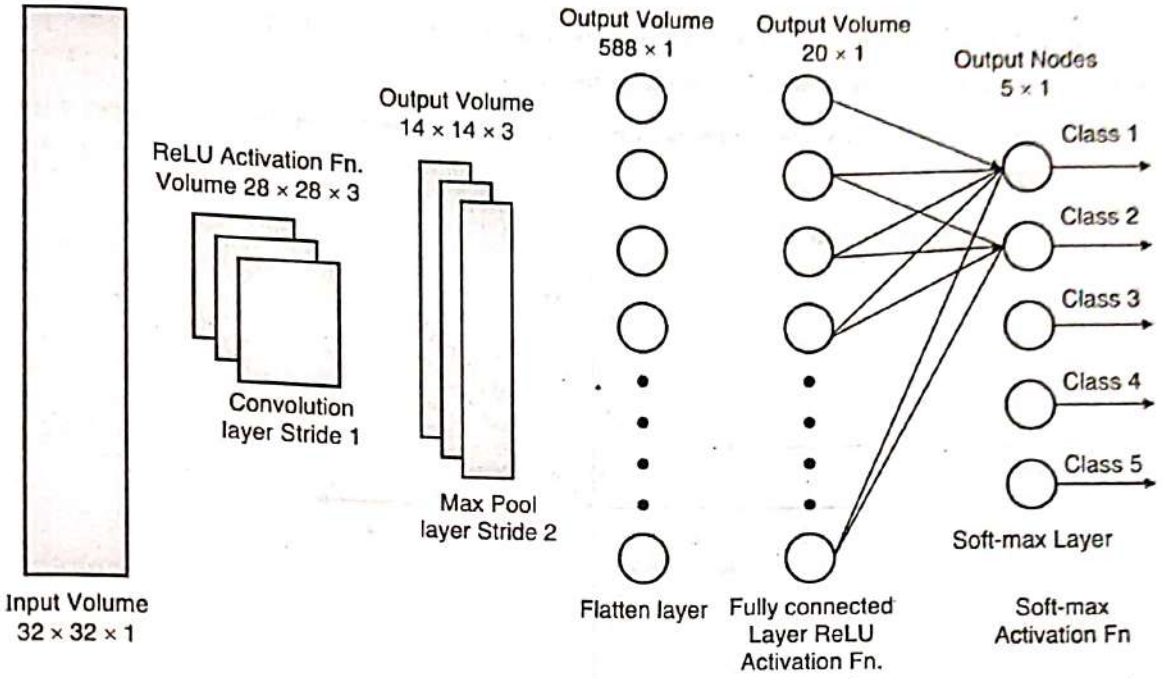


Fig. 3.3.7 : Fully Connected Layer

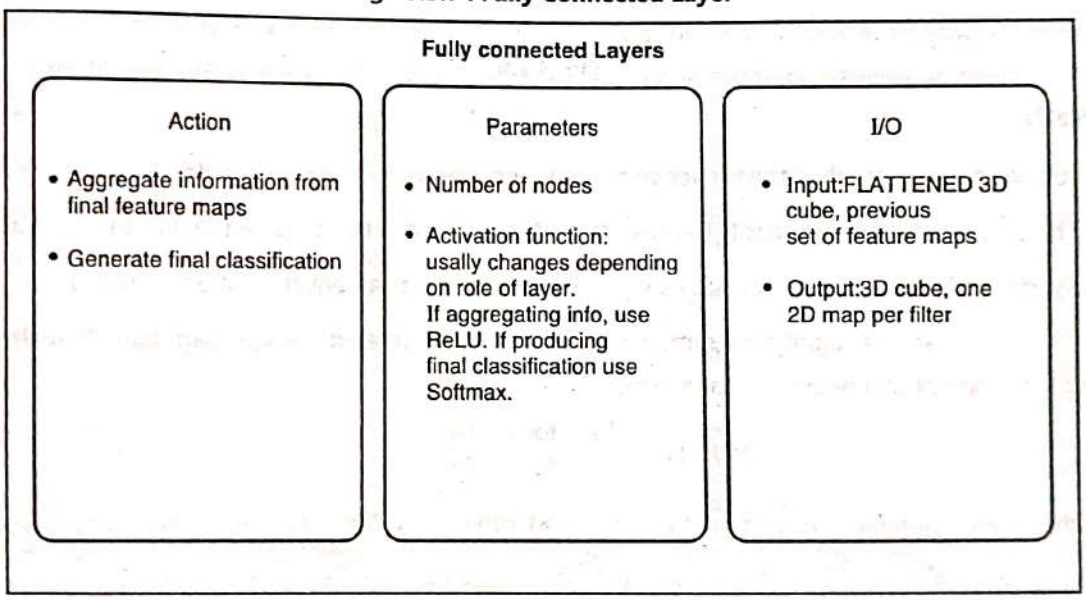


Fig. 3.3.8 : Features of a fully connected layer

3.3.4 ReLU

- Hidden layers of a trained CNN have neurons that might be abstract representations of the input characteristics. A CNN is not aware of which of the learnt abstract representations will apply to a given input when it is presented with an unknown input.

- There are two conceivable (fuzzy) scenarios for each neuron in the buried layer that represents a particular learn abstract representation: either the neuron is relevant or it isn't.
- The absence of the neuron's relevance does not always imply that other potential abstract representations are les likely as a result. If we were to employ an activation function with the image R-, it would indicate that, for specific input values to a neuron, that neuron's output would have a negative impact on the neural network's output.
- We presume that all learnt abstract representations are independent of one another, which is often undesirable. Therefore, non-negative activation functions are preferred for CNNs.
- The Rectified Linear function is the most prevalent of these functions, and a neuron that employs it is known as a Rectified Linear Unit (ReLU),

$$F(x) = \text{argmax}(0, x)$$

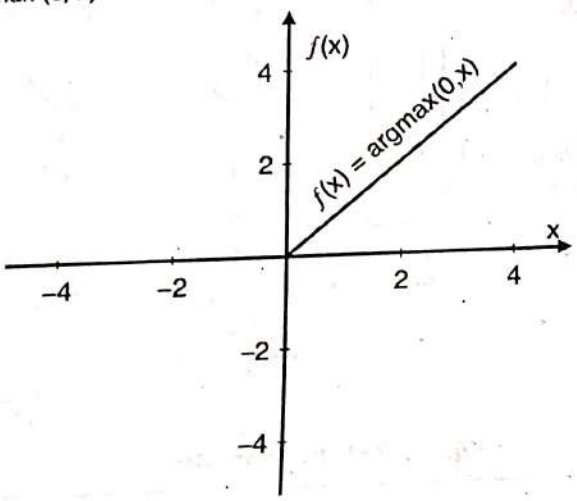


Fig. 3.3.9

Computing ReLU

- This function has two major advantages over sigmoidal functions such as σ(x) or tanh(x).
- ReLU may be calculated extremely easily because all that is required is to compare the input to the value 0.
- Additionally, depending on whether or not its input is negative, it has a derivative of either 0 or 1.
- The latter in particular has significant ramifications for training-related backpropagation. It really means that computing the gradient of a neuron is not expensive :

$$\text{ReLU}'(x) = \begin{cases} 0, & \text{for } x < 0 \\ 1, & \text{for } x \geq 0 \end{cases}$$

- On the other hand, sigmoidal activation functions and other non-linear activation functions typically lack this property.
- As a result, using ReLU lessens the likelihood that the computing needed to run the neural network would rise exponentially. The computational cost of adding more ReLUs rises linearly as the CNN's size scales.
- The so-called "vanishing gradient" problem, which frequently occurs when employing sigmoidal functions, is likewise avoided by ReLUs. This issue relates to a neuron's propensity for its gradient to approach zero for large input levels.

- ReLU always stays at a constant 1, unlike sigmoidal functions, whose derivatives trend to 0 as they approach positive infinity. Due to this, even with large input values to the activation function, backpropagation of the mistake and learning can continue:

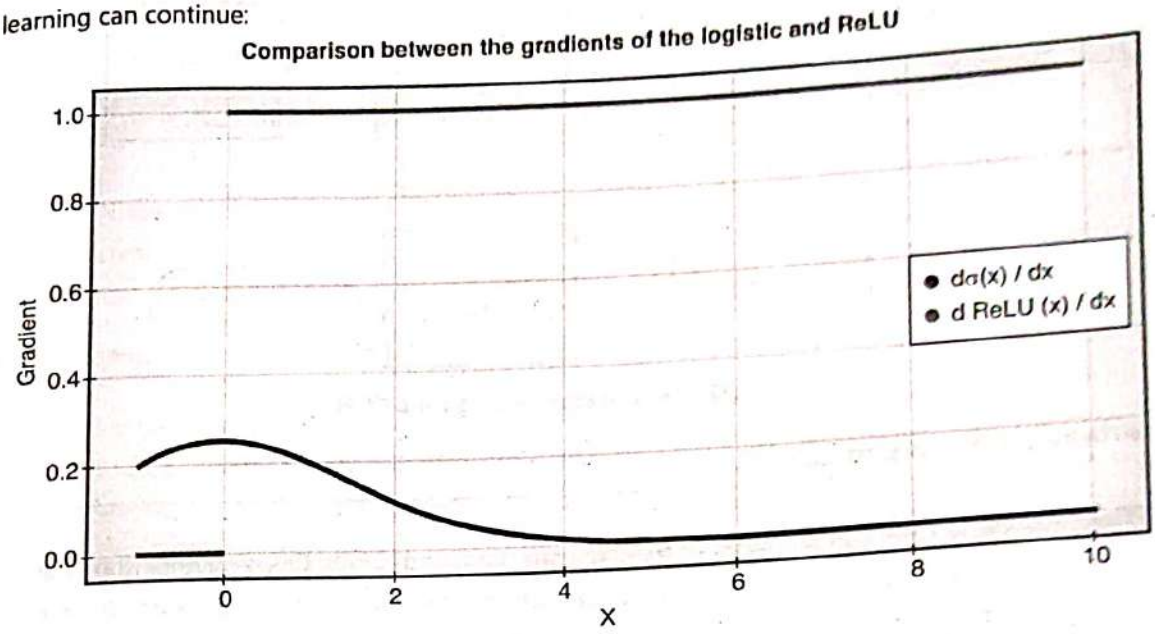


Fig. 3.3.10 : ReLU vs Sigmoid

3.3.5 Dropout Layer

- A Dropout layer is another prominent feature of CNNs. The Dropout layer acts as a mask, eliminating some neurons' contributions to the subsequent layer while maintaining the functionality of all other neurons. If we apply a Dropout layer to the input vector, some of its characteristics are eliminated; however, if we apply it to a hidden layer, some hidden neurons are eliminated.
- Because they avoid overfitting on the training data, dropout layers are crucial in the training of CNNs. If they are absent, the first set of training samples has an excessively large impact on learning. As a result, traits that only show in subsequent samples or batches would not be learned.

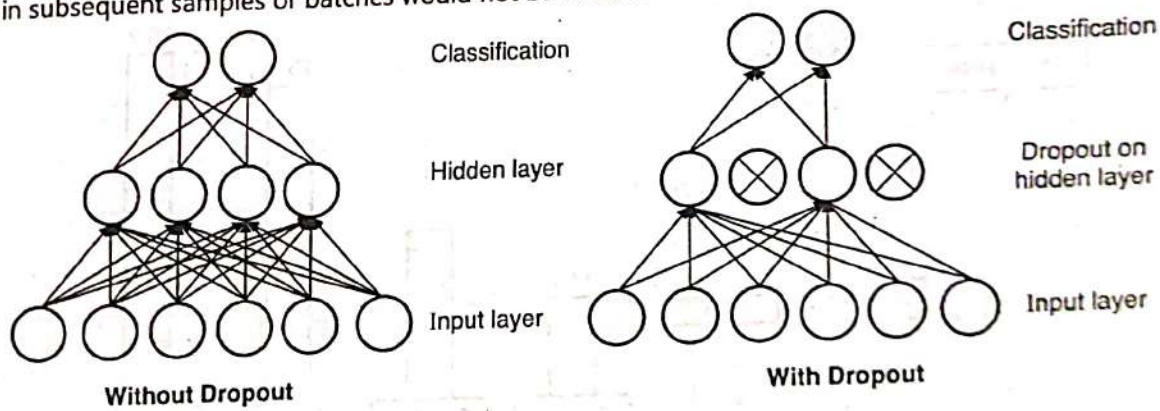


Fig. 3.3.11 : Dropout Layer

- Let's imagine that during training, we display a CNN 10 images of a circle one after the other. If we subsequently show the CNN an image of a square, it won't understand that straight lines exist and will be quite perplexed. By incorporating Dropout layers into the network's architecture to minimise overfitting, we may avoid these situations.

- The Fig. 3.3.12 shows a typical architecture for a CNN with a ReLU and a Dropout layer. This type of architecture is very common for image classification tasks.

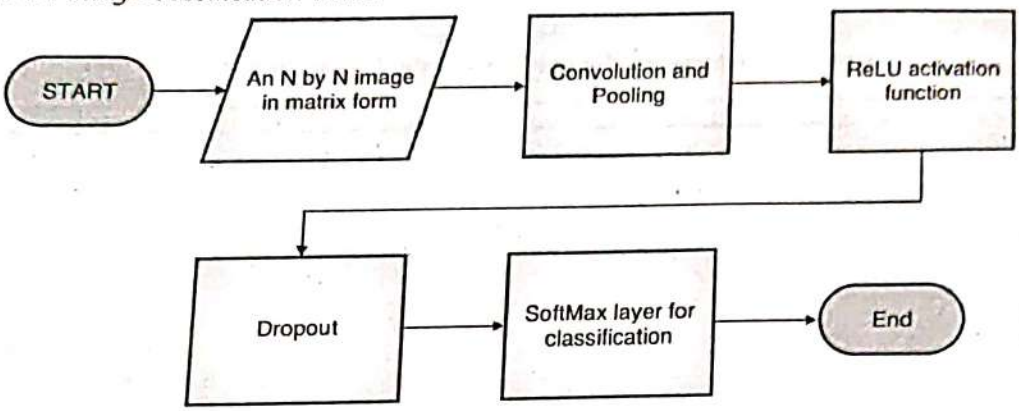


Fig. 3.3.12 : Typical architecture for a CNN

3.4 Interleaving between Layers

- These different layers in CNN can be mixed or repeated any number of times, this is interleaving of layers in CNN. Interleaving of is an important feature provided in CNN which makes research and experimentation in CNN that much more interesting.
- There are several CNN designs that may be used, and these architectures have been essential in creating the algorithms that power and will continue to power AI as a whole in the near future. Below is a list of a few of them:
  - LeNet
  - AlexNet
  - VGGNet
  - GoogLeNet
  - ResNet
  - ZFNet

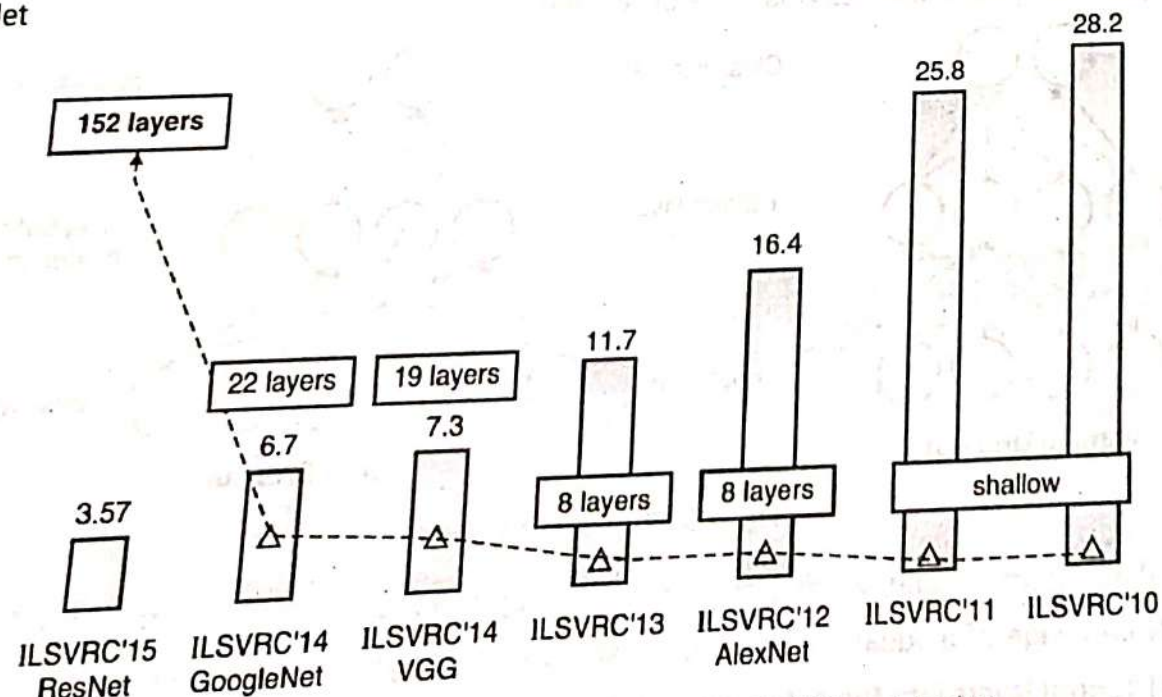


Fig. 3.4.1 : Different CNN architectures with different layers

The Fig. 3.4.2 the architecture of VGGNet

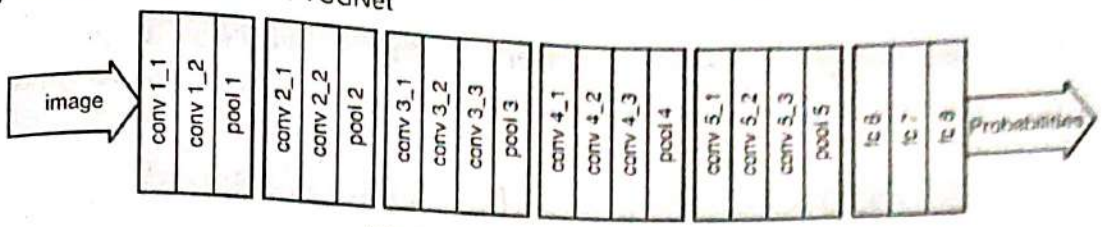
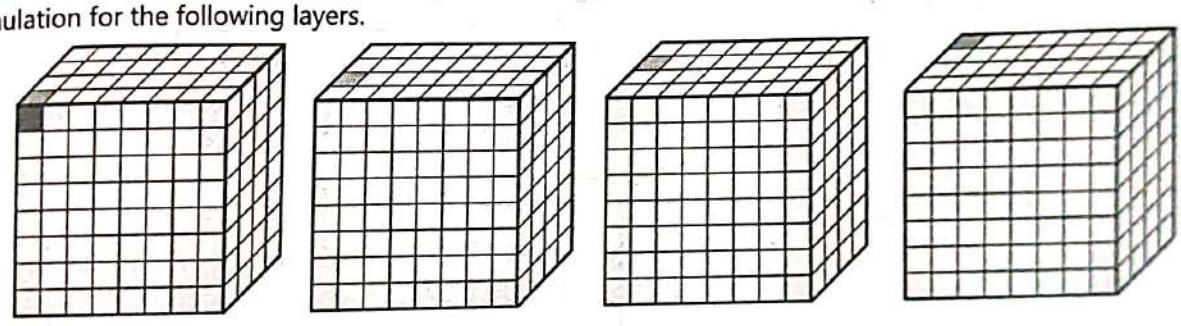


Fig. 3.4.2 : VGGNet Architecture

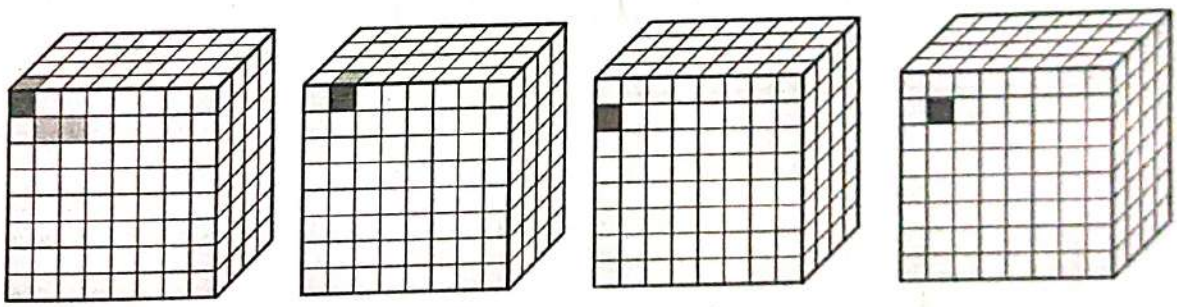
- Different CNN designs have been explained in the above figures. Over the years complexities of the architectures have been increased from 8 layers of combination of Convolution, Pooling and Fully connected layers in AlexNet to 19 layers in VGG to 152 layers in ResNet.

3.5 Local Response Normalization

- For deep neural networks that make up for the unbounded nature of some activation functions like ReLU, ELU, etc., normalisation has grown in importance. With these activation functions, the output layers can expand as high as the training permits rather than being limited to a finite range (like [-1,1] for tanh).
- Normalization is employed shortly before the activation function to prevent the unbounded activation from boosting the output layer values.
- Local Response Normalization is one of the most important normalization techniques employed, especially in CNN.
- ReLU was employed as the activation function instead of the then-common tanh and sigmoid, which led to the initial introduction of Local Response Normalization (LRN) in the AlexNet architecture.
- In addition to the aforementioned justification, the use of LRN was made to promote lateral inhibition. The ability of a neuron to lessen the activity of its neighbours is a notion in neuroscience. This lateral inhibition function in DNNs is utilised to carry out local contrast enhancement such that the highest pixel values are used as local stimulation for the following layers.



a) Inter-Channel LRN (n = 2)



b) Inter-Channel LRN (n = 2)

Fig. 3.5.1 : Local Response Normalization

1. Inter-Channel LRN

- This is what the AlexNet paper first employed. The specified neighbourhood is across the water. The normalisation is done in the depth dimension for each (x,y) point and is determined by the following formula.

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

- Where N is the total number of channels, I is the output of filter I a(x,y), b(x,y) denotes the pixel values at (x, y) position before and after normalisation, respectively, and The (k,,,,n) constants are hyper-parameters. A singularity is avoided by using k, which is also employed as a normalisation constant and as a contrasting constant (division by zero). The neighbourhood length, or how many successive pixel values must be taken into account while doing the normalisation, is defined by the constant n. The standard normalisation is the situation when (k, α, β, n) =(0, 1, 1, N). In the image above, n is assumed to be 2 and N is equal to 4.
- Let's have a look at an example of Inter-channel LRN. Consider the following Fig. 3.5.2.

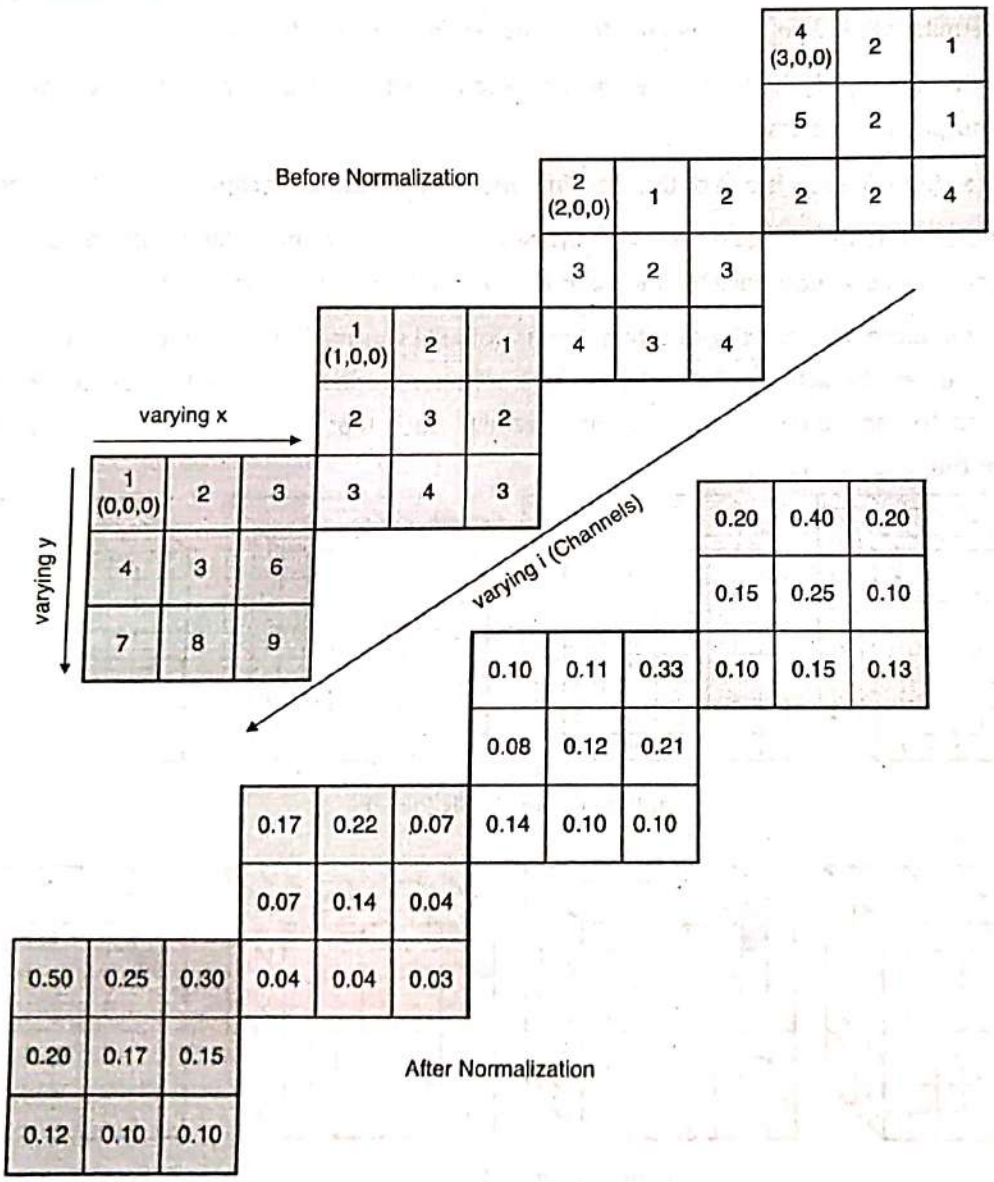


Fig. 3.5.2 : Inter-Channel LRN

- Different colours denote different channels and hence N = 4. Let's take the hyper-parameters to be (k, α, β, n) = (0, 1, 1, 2). The value of n = 2 means that while calculating the normalized value at position (i, x, y), we consider the values at the same position for the previous and next filter i.e. (i-1, x, y) and (i+1, x, y). For (i, x, y) = (0, 0, 0) we have value (i, x, y) = 1, value (i-1, x, y) doesn't exist and value (i+, x, y) = 1. Hence normalized-value (i, x, y) = $1/(1^2+1^2)$ = 0.5 and can be seen in the lower part of the figure above. The rest of the normalized values are calculated in a similar way.

2. Intra-Channel LRN

- As seen in the above graphic, intra-channel LRN only extends the neighbourhood inside the same channel. The equation is provided by

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

- Where (W, H) are the width and height of the feature map (for example in the figure above (W, H) = (8, 8)). The only difference between Inter and Intra Channel LRN is the neighbourhood for normalization. In Intra-channel LRN, a 2D neighbourhood is defined (as opposed to the 1D neighbourhood in Inter-Channel) around the pixel under-consideration. As an example, the figure below shows the Intra-Channel normalization on a 5 × 5 feature map with n = 2 (i.e. 2D neighbourhood of size (n + 1) × (n + 1) centred at (x, y)).

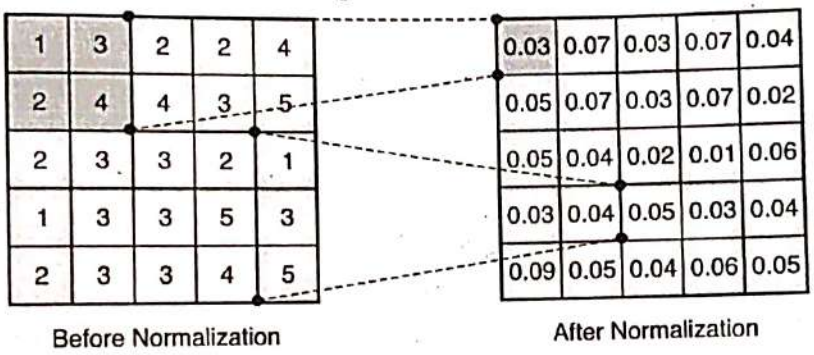


Fig. 3.5.3 : Intra-Channel LRN

3.6 Training a Convolutional Network

Generally, in order to build and train a CNN following steps are followed

1. Input Layer

This step resets the data. Size is equal to the square root of the number of pixels. For example, if a picture has 156 pixels, the figure is 26x26. We need to specify whether the image contains color or not. If so, we had a size 3 to 3 for RGB-, otherwise 1.

2. Convolution Layer

Here, We need to create consistent layers. We apply various filters to learn important features of the network. We define the size of the kernel and volume of the filter.

3. Pooling Layer

In the third step, we add a pooling layer. This layer reduces the size of the input. It does by taking the maximum value of the sub-matrix.

4. **Additional Convolution and Pooling Layers**

   In this step, we can add as many convolution and pooling layers as we want.

5. **Dense Layer / Fully Connected Layer**

   Step 5 flattens the previous to form fully joined layers. In this step, we can use a different activation function and the dropout effect.

6. **Logit Layer**

   The final step is the prediction.

7. **Train the Model**

   Once the architecture has been finalized the CNN can be trained. For practical implementation packages TensorFlow, Keras or PyTorch can be used for building the CNN architecture/model and train/test it.

8. **Test and Evaluate the Model**

   One the model has been trained, the final step is to test the mode and evaluate its performance.

TensorFlow Code

```
# Input Layer
input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])

# Convolutional Layer
conv1 = tf.layers.conv2d(
  inputs=input_layer,
  filters=32,
  kernel_size=[5, 5],
  padding="same",
  activation=tf.nn.relu)

# Pooling Layer
pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)

# Convolutional Layer #2 and Pooling Layer
conv2 = tf.layers.conv2d(
  inputs=pool1,
  filters=36,
  kernel_size=[5, 5],
  padding="same",
  activation=tf.nn.relu)
pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)

# Dense Layer
```

```
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 36])
dense = tf.layers.dense(inputs=pool2_flat, units=7 * 7 * 36, activation=tf.nn.relu)
dropout = tf.layers.dropout(
  inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

# Logits Layer
logits = tf.layers.dense(inputs=dropout, units=10)

predictions = {
  # Generate predictions (for PREDICT and EVAL mode)
  "classes": tf.argmax(input=logits, axis=1),
  "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
}

if mode == tf.estimator.ModeKeys.PREDICT:
 return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

# Calculate Loss
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

# Configure the Training Op (for TRAIN mode)
if mode == tf.estimator.ModeKeys.TRAIN:
 optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
 train_op = optimizer.minimize(
   loss=loss,
   global_step=tf.train.get_global_step())
 return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

# Add evaluation metrics Evaluation mode
 eval_metric_ops = {
 "accuracy": tf.metrics.accuracy(
 labels=labels, predictions=predictions["classes"])}
 return tf.estimator.EstimatorSpec(
 mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)

# Set up logging for predictions
tensors_to_log = {"probabilities": "softmax_tensor"}
logging_hook =tf.train.LoggingTensorHook(tensors=tensors_to_log, every_n_iter=50)
```

```
#Train the model
#Train the model
train_input_fn = tf.estimator.inputs.numpy_input_fn(
  x={"x": X_train_scaled},
  y=y_train,
  batch_size=100,
  num_epochs=None,
  shuffle=True)
  mnist_classifier.train(
  input_fn=train_input_fn,
  steps=18000,
  hooks=[logging_hook])
 # Evaluate the model and print the results
eval_input_fn = tf.estimator.inputs.numpy_input_fn(
  x= {"x": X_test_scaled},
  y=y_test,
  num_epochs=1,
  shuffle=False)
eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
print(eval_results)
```

Review Questions

Q. 1 Draw and explain CNN architecture.

Q. 2 Explain working of Convolutional layer.

Q. 3 Explain all the features of pooling layer.

Q. 4 Explain pooling layers and its different types.

4 Recurrent and Recursive Nets

Unit IV

Syllabus

Recurrent and Recursive Nets : Unfolding Computational Graphs, Recurrent Neural Networks, Bidirectional RNNs, Encoder-Decoder Sequence-to-Sequence Architectures, Deep Recurrent Networks, Recursive Neural Networks, The Challenge of Long-Term Dependencies, Echo State Networks, Leaky Units and Other Strategies for Multiple Time Scales, The Long Short-Term Memory and Other Gated RNNs, Optimization for Long-Term Dependencies, Explicit Memory. **Practical Methodology :** Performance Metrics, Default Baseline Models, Determining Whether to Gather More Data, Selecting Hyper parameters.

4.1 Unfolding Computational Graphs

- The structure of a collection of computations, such as the mapping of inputs and parameters to outputs and loss, may be formalised using a computational graph.
- We can unfold a recursive or recurrent computation into a repeated computational network.
- Unfolding this graph leads to sharing of parameters across a deep network structure, correlating to a series of occurrences.

4.1.1 Example of Unfolding a Recurrent Equation

Classical form of a dynamical system is

$$s^{(t)} = f(s^{(t-1)}; \theta)$$

Where, $s^{(t)}$ is called the state of the system

- Equation is recurrent because the definition of s at time t refers back to the same definition at time t-1
- For a finite no. of time steps τ, the graph can be unfolded by applying the definition τ-1 times

For example : for $\tau = 3$ time steps we get

$$s^{(3)} = f(s^{(2)}; \theta) = f(f(s^{(1)}; \theta); \theta)$$

- Unfolding equation by repeatedly applying the definition in this can yieldexpression without recurrence $s^{(1)}$ is ground state and $s^{(2)}$ computed by applying f. Such an expression can be represented by **a traditional acyclic computational graph.**

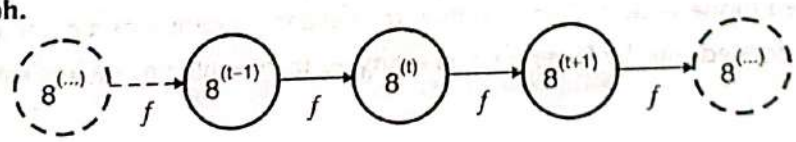


Fig. 4.1.1 : Unfolding

- Each node represents state at some time t
- Function f maps state at time t to the state at t + 1
- The same parameters (the same value of θ used to parameterize f) are used for all time steps

4.1.2 Dynamical System Driven by External Signal

- As another example, consider a dynamical system driven by external (input) signal $x^{(t)}$

$$s^{(t)} = f\left(s^{(t-1)}, x^{(t)}; \theta\right)$$

- State now contains information about the whole past input sequence
- Note that the previous dynamic system was simply $s^{(t)} = f(s^{(t-1)}; \theta)$
- Recurrent neural networks can be built in many ways. Much as almost any function is a feedforward neural network, any function involving recurrence can be considered to be a recurrent neural network.

4.1.3 Unfolding Recurrent Neural Net

- Many recurrent neural nets use same equation, as dynamical system with external input, to define values of hidden units.
- To indicate that the state is hidden rewrite using variable h for state :

$$h(t) = f\left(h(t-1), x(t); \theta\right)$$

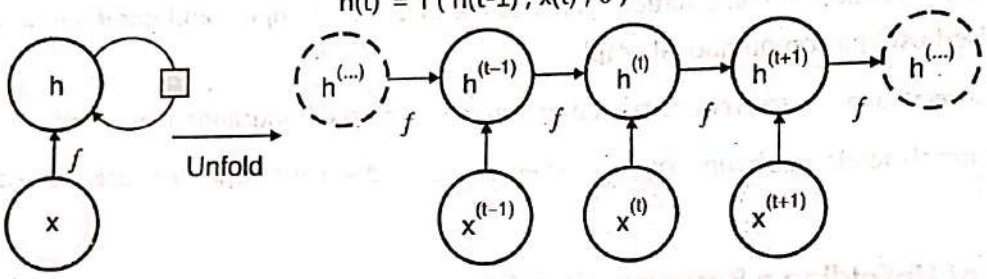


Fig. 4.1.2 : Unfolding RNN

- Typical RNNs have extra architectural features such as output layers that read information out of state h to make predictions.

4.1.4 Advantages of Unfolding Process

- The unfolding process introduces two major advantages :
  1. Regardless of sequence length, learned model has same input size. This is because it is specified in terms of transition from one state to another state rather than specified in terms of a variable length history of states
  2. Possible to use same function f with same parameters at every step.
- These two factors make it possible to learn a single model f, that operates on all time steps and all sequence lengths, rather than needing separate model g(t) for all possible time steps
- Learning a single shared model allows : Generalization to sequence lengths that did not appear in the training and Allows model to be estimated with far fewer training examples than would be required without parameter sharing.

4.2 Recurrent Neural Networks

- Recurrent neural networks (RNNs) are a type of neural network in which the results of one step are fed into the next step's computations.
- Traditional neural networks have inputs and outputs that are independent of one another, but there is a need to remember the previous words in situations where it is necessary to anticipate the next word in a sentence.
- As a result, RNN was developed, which utilised a Hidden Layer to resolve this problem. The Hidden state, which retains some information about a sequence, is the primary and most significant characteristic of RNNs.

Fig. 4.2.1 : General RNN

- RNNs have a "memory" that retains all data related to calculations. It executes the same action on all of the inputs or hidden layers to produce the output, using the same settings for each input. In contrast to other neural networks, this minimises the complexity of the parameter set.

4.2.1 How RNN Works ?

- The working of a RNN can be understood with the help of following example.
- Consider a deeper network that has three hidden layers, one output layer, one input layer, and three hidden levels. Each hidden layer will thus, like other neural networks, have its own set of weights and biases.
- For example, let's suppose that the weights and biases for hidden layer 1 are (w1, b1), (w2, b2) for the second hidden layer, and (w3, b3) for the third hidden layer. This indicates that each of these layers is independent of the others and does not retain information from earlier outputs.

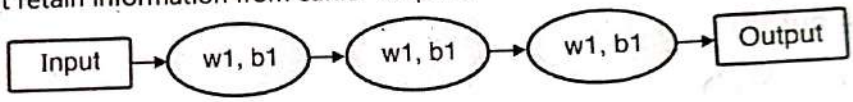


Fig. 4.2.2 : RNN as Deep Neural Network

- The RNN will now perform the following :
- By giving all of the layers the same weights and biases, RNN transforms independent activations into dependent activations, decreasing the complexity of raising parameters and memorising each previous output by using each output as an input to the following hidden layer.
- Thus, all three layers can be combined into a single recurrent layer so that the weights and bias of all the hidden levels are the identical.

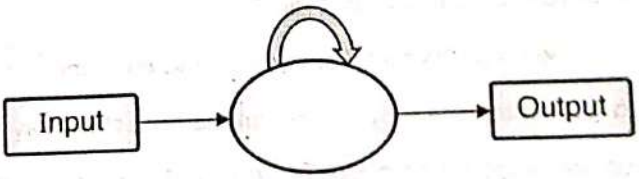


Fig. 4.2.2(a) : Recurrent state in RNN

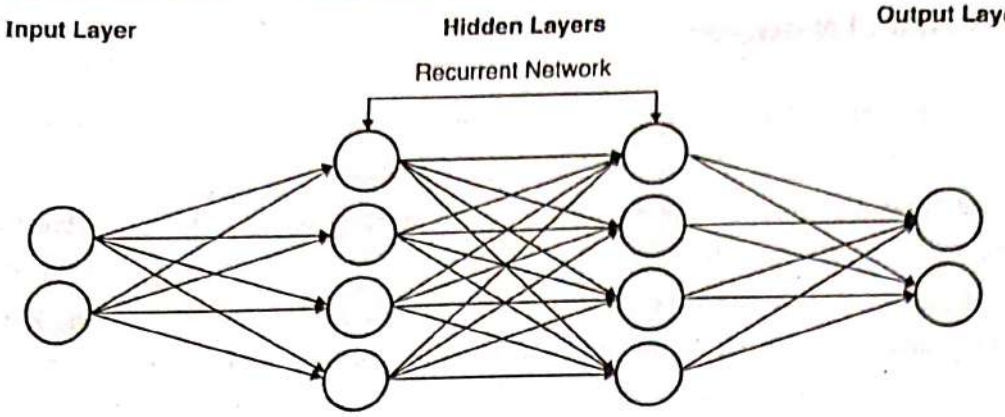


Fig. 4.2.2 (b) : Recurrent state in RNN

Formula for Calculating current state

$$h_t = f(h_{t-1}, X_t)$$

Where :

h_t -> current state

h_{t-1} -> previous state

x_t -> input state

Formula for applying Activation function(tanh)

$$h_t = \tan h\,(W_{hh}\, h_{t-1} + W_{xh}\, X_t)$$

Where :

w_{hh} -> weight at recurrent neuron

w_{xh} -> weight at input neuron

Formula for calculating output

$$y_t = W_{hy}\, h_t$$

Y_t -> output

W_{hy} -> weight at output layer

4.2.2 Training

- The input is given to the network in a single time step.
- Then, using the set of current input and the prior state, calculate the present state of the system.
- For the following time step, the current time becomes time-1.
- Depending on the issue, one can travel back as many time steps and combine the data from all the prior states.
- The final current state is used to determine the output after all the time steps have been finished.
- The error is then generated once the output is compared to the goal output, which is the actual output.
- The network (RNN) is trained after the error is back-propagated to it in order to update the weights.

4.2.3 Recurrent Neural Networks : Why use them?

- There were a few problems with the feed-forward neural network, which led to the development of RNN :
  - can't deal with consecutive data
  - merely takes into account current input
  - unable to remember earlier inputs
  - The RNN offers a remedy for these problems. An RNN can handle sequential data, accepting both the input data being used at the moment and inputs from the past. RNNs' internal memory allows them to remember prior inputs.

4.2.4 Working of each Recurrent Unit

- Take the current input vector and the previously hidden state vector as input. Keep in mind that each element of the vector is positioned in a different dimension that is orthogonal to the other dimensions because the hidden state and current input are both considered as vectors. As a result, when one element is multiplied by another, only non-zero elements in the same dimension and that element's own dimension provide non-zero values.
- The hidden state vector is multiplied by the hidden state weights element-wise, while the current input vector and current input weights are also multiplied element-wise. This produces the current input vector and the parameterized hidden state vector. Note that the trainable weight matrix contains weights for several vectors.
- To create the new hidden state vector, perform the vector addition of the two parameterized vectors and then compute the element-wise hyperbolic tangent.

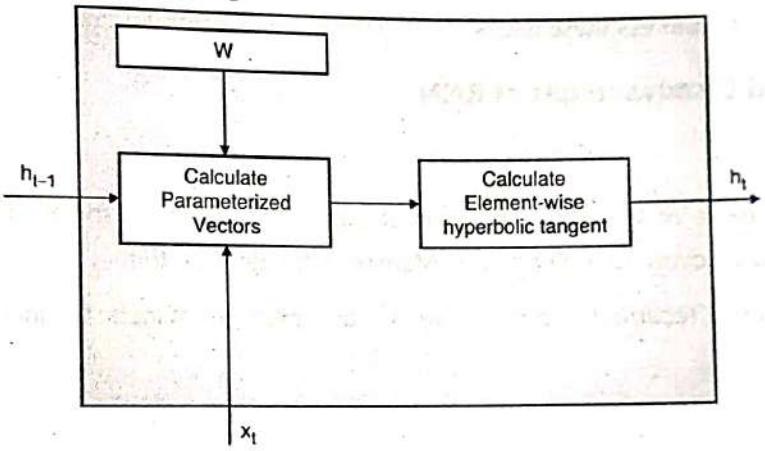


Fig. 4.2.3 : Training of Recurrent Unit

- The recurrent network produces an output at each time step while it is being trained. Gradient descent is used to train the network using this output.

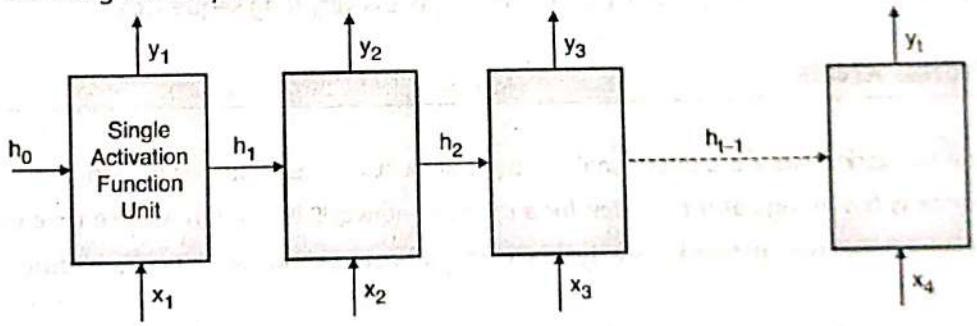


Fig. 4.2.4 : Training RNN

- With a few slight modifications, the Back-Propagation used here is comparable to the one used in a conventional Artificial Neural Network.

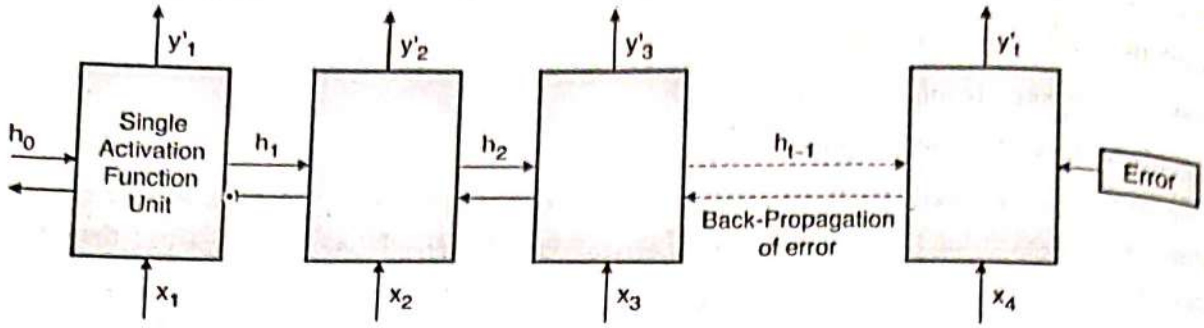


Fig. 4.2.5 : Back Propagation in RNN

- Although the fundamental Recurrent Neural Network is reasonably effective, it can have a serious issue. The back propagation technique for deep networks can result in the following problems:
  - **Vanishing gradients :** These are gradients that are so small that they gravitate to zero.
  - **Exploding Gradients :** These happen when back-propagation causes the gradients to grow excessively large.
- By placing a threshold on the gradients being transported back in time, it may be possible to use a hack to overcome the Exploding Gradients problem. However, this technique is not regarded as resolving the issue and could harm the network's effectiveness.
- Long Short Term Memory Networks and Gated Recurrent Unit Networks, two key versions of Recurrent Neural Networks, were created to address these issues.

4.2.5 Advantages and Disadvantages of RNN

Advantages

- An RNN retains every piece of knowledge throughout time. Only the ability to remember past inputs makes it helpful for time series prediction. Long Short Term Memory is the term for this.
- Convolutional layers and recurrent neural networks are even combined to increase the effective pixel neighbourhood.

Disadvantages

- Problems with gradient disappearing and explosions.
- It is exceedingly tough to train an RNN.
- If tanh or relu are used as the activation function, it cannot process very long sequences.

4.3 Bidirectional RNNs

- Recurrent neural networks that are bidirectional are basically just two separate RNNs combined. For one network the input sequence is fed in regular time order; for a different network, it is fed in reverse time order. At each time step, the outputs of the two networks are typically concatenated, however there are other choices, such as summation.

- The networks may access both forward and backward information about the sequence at each time step because to this structure. The idea sounds simple enough.

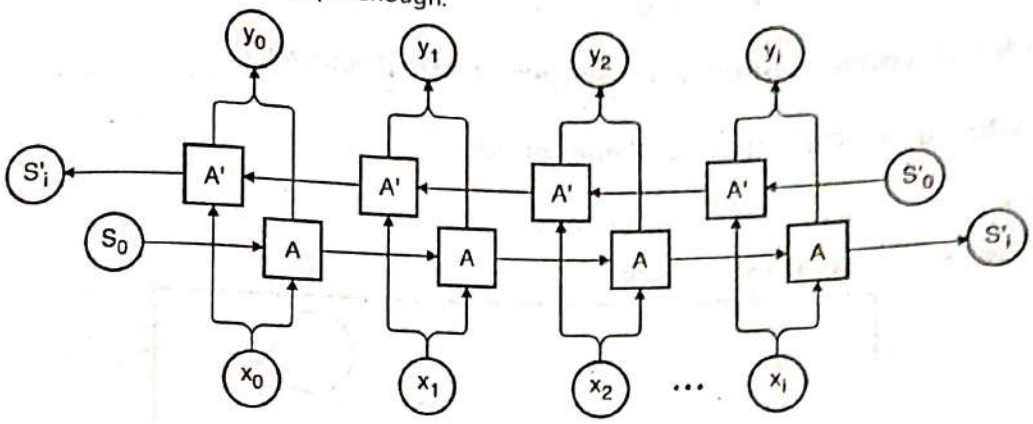


Fig. 4.3.1 : Bidirectional Neural Network

- But there remains uncertainty when it comes to actually putting a neural network with a bidirectional topology into practise.

4.3.1 The Confusion

- The first area of uncertainty is how to send a bidirectional RNN's outputs to a dense neural network. The next image, which I obtained using Google, depicts a similar strategy on a bidirectional RNN, demonstrates how we might simply forward the outputs at the last time step for conventional RNNs.

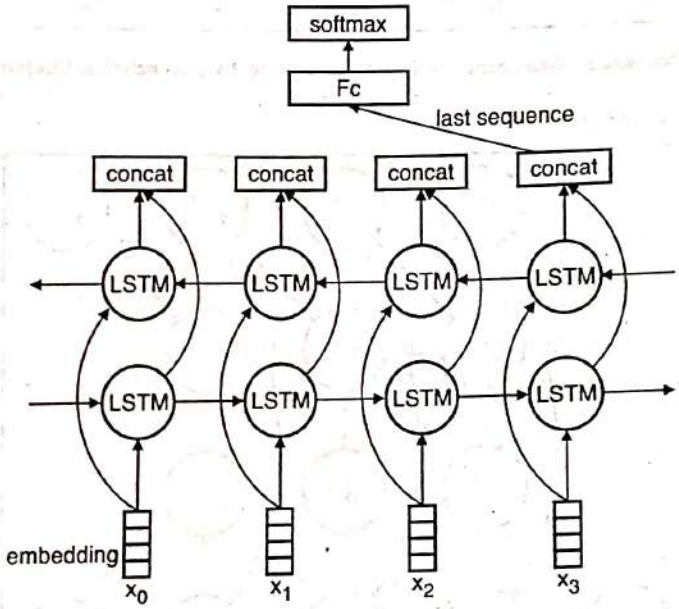


Fig. 4.3.2 : Formulation of Bidirectional Neural Network

- The reverse RNN will only have viewed the most recent input (x 3 in the illustration) if we choose the output at the final time step. It won't have much predictive power.
- The returning hidden states are the second area of uncertainty. We need the encoder's hidden states to initialise the decoder's hidden states in seq2seq models. It seems sense that, if we can only select hidden states at a single time step, we would prefer the state where the RNN has just finished consuming the most recent input in the sequence.

- We will, however, get the hidden states of the reversed RNN with only one step of inputs seen if the hidden states of time step n (the last one) are returned, as previously.

4.4 Encoder-Decoder Sequence-to-Sequence Architectures

4.4.1 RNN when Input/Output are of Same Length

RNNs a very useful and can be used under three cases,

1. To map an input sequence to a fixed-size vector

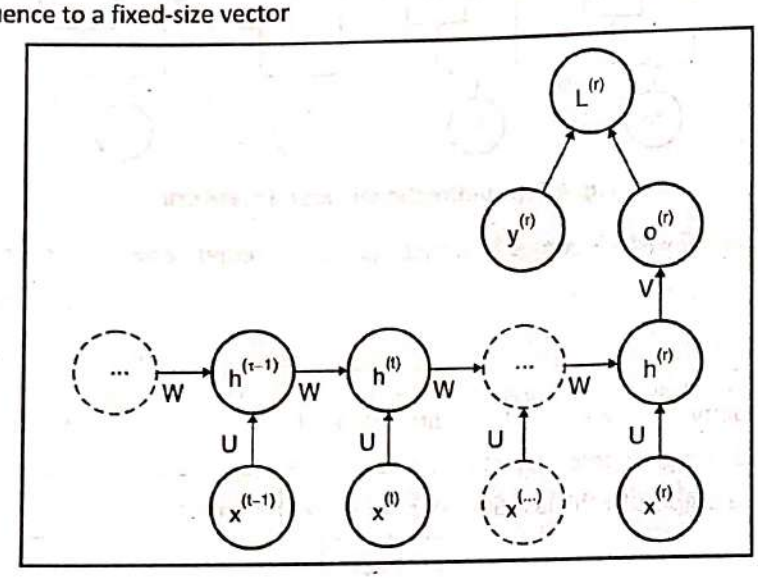


Fig. 4.4.1 : Mapping an input sequence to a fixed-size vector

2. To map a fixed-size vector to a sequence

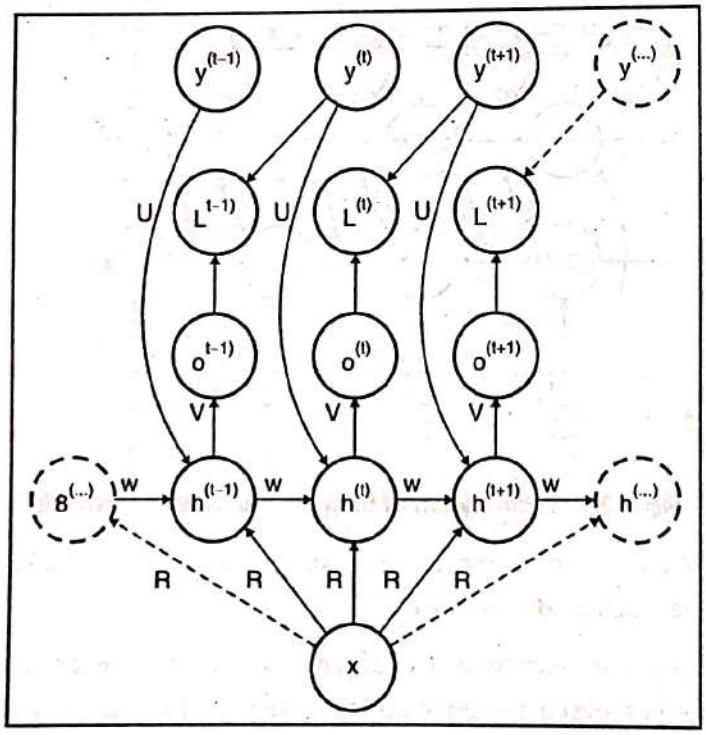


Fig. 4.4.2 : Mapping a fixed-size vector to a sequence

3. To map an input sequence to an output sequence of the same length

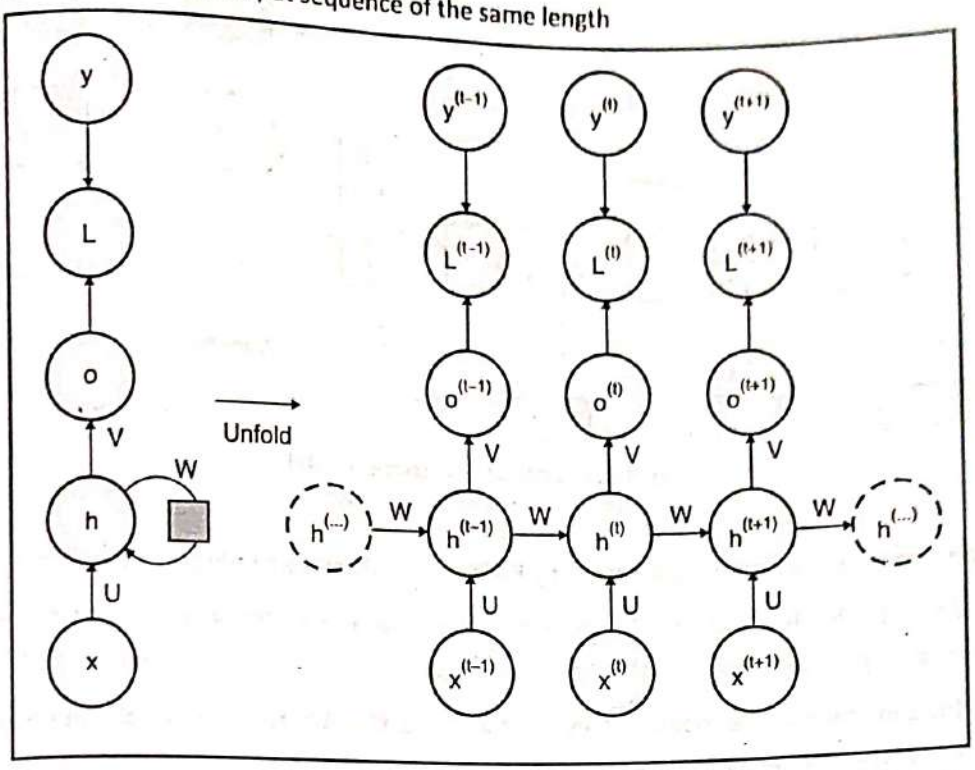


Fig. 4.4.3 : Mapping an input sequence to an output sequence of the same length

4.4.2 RNN when Input / Output are not Same Length – Encoder-Decoder Model

- Here we consider how an RNN can be trained to map an input sequence to an output sequence which is not necessarily the same length.
- Better neural network design can be created in deep learning to address many complex challenges. In sequence-to-sequence learning, the RNN (Recurrent Neural Network) and its derivatives are quite helpful. The most often utilised cell in seq-seq learning tasks is the RNN variant LSTM (Long Short-Term Memory).
- The typical neural machine translation technique that competes with and occasionally exceeds traditional statistical machine translation techniques is called encoder-decoder architecture for recurrent neural networks.
- The encoder-decoder model is composed on three primary building blocks :
  - Encoder
  - Hidden Vector / Encoder Vector
  - Decoder
- The encoder will create a one-dimensional vector from the input sequence (hidden vector). The hidden vector will be transformed into the output sequence by the decoder.
- In order to optimise the conditional probabilities of the target sequence given the input sequence, encoder-decoder models are jointly trained.

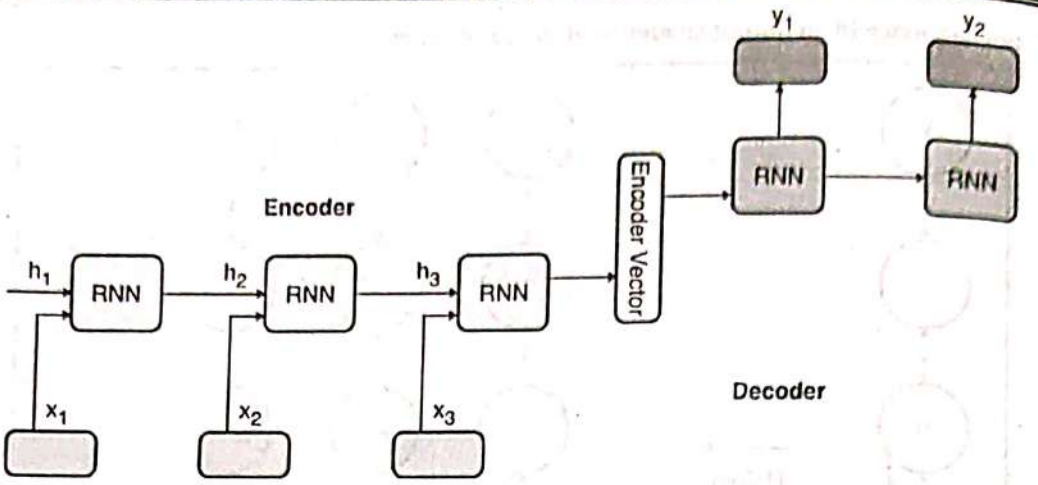


Fig. 4.4.4 : Encoder-Decoder Model

1. Encoder

- The encoder may be created by stacking many RNN cells. RNN successively scans each input
- The hidden state (hidden vector) h is updated in accordance with the input at that timestep X[i] for ea timestep (each input) t.
- The final hidden state of the model represents the context/summary of the whole input sequence after th encoder model has read all of the inputs.
- **Example :** Consider encoding the input phrase "I am a Student." The Encoder model will have a total of timesteps (4 tokens). The previous hidden state and the current input are used to update the hidden state h each time step.

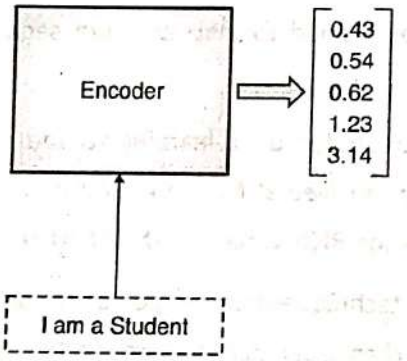


Fig. 4.4.5 : Encoder

- The preceding hidden state h0 will be regarded as zero or randomly selected at timestep t1. Therefore, usin the initial input and h0, the first RNN cell will update the current hidden state. Updated hidden state and th output for each level are the two outputs that each layer produces.
- Only the concealed states will be transmitted to the subsequent layer at each level, with the outputs at eac stage being discarded.
- The following formula is used to calculate the hidden states h i:

$$h_t = f\left(W^{(hh)} h_{t-1} + W^{(hx)} x_t\right)$$

- The second input X[2] and the hidden state h1 will be provided as input at timestep t2, and the hidden stat h2 will be modified in accordance with both inputs. The hidden state h2 is then created by updating th hidden state h1 with the new input. With regard to the chosen case, this occurs for all four steps.

- A group of recurrent units (LSTM or GRU cells for greater performance) stacked one on top of the other, each of which receives a single input sequence element, gathers data for that element, and propagates it forward.
- The input sequence for the task entails gathering every word from the question. Each word is denoted by the symbol x i, where I denotes the word's order.
- This simple formula represents the result of an ordinary recurrent neural network. As you can see, we just apply the appropriate weights to the previously hidden state h(t-1) and the input vector xt.

2. **Encoder Vector**

- This is the model's final hidden state created by the encoder. The formula above is used to compute it.
- In order to aid the decoder in producing precise predictions, this vector seeks to include the data for all input components.
- It serves as the decoder portion of the model's first hidden state.

3. **Decoder**

- By anticipating the subsequent output Yt given the hidden state ht, the Decoder creates the output sequence.
- The final hidden vector obtained at the conclusion of the encoder model serves as the decoder's input.
- Each layer will have three inputs: the original hidden vector h, the hidden vector from the preceding layer (ht-1), and the layer output (yt-1).
- The first layer's inputs are the encoder's output vector and the random symbol START. The outputs are the updated hidden state h1 and the empty hidden state ht-1 (the information of the output will be subtracted from the hidden vector).
- The second layer creates the hidden vector h2 and output y2 using the updated hidden state h1, the previous output y1, and the original hidden vector h as current inputs.
- The real output is what the decoder produced at each timestep. Up to the END sign, the model will continue to forecast the output.
- A collection of several recurrent units, each of which forecasts an output at time step t, y t.
- Each recurrent unit receives a hidden state from the preceding unit, accepts it, and generates both an output and a hidden state of its own.
- The output sequence in the question-answering problem is a compilation of each word in the response. Each word is denoted by the symbol y i, where I denotes the word's order.

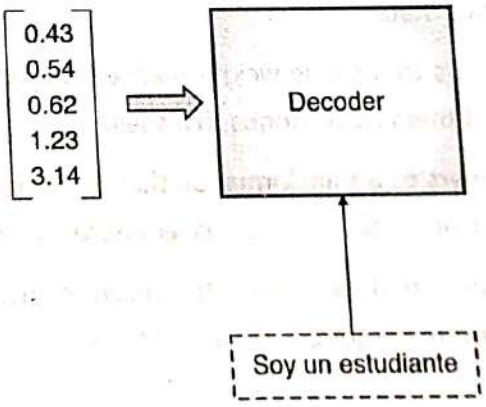


Fig. 4.4.6 : Decoder

- Any hidden state h_i is computed using the formula :

$$h_t = f(W^{(hh)} h_{t-1})$$

- As you can see, we are just using the previous hidden state to compute the next one.
- At the output layer, we employ Softmax activation function.
- With the goal class of high probability, it is used to generate the probability distribution from a vector of values.
- The formula is used to calculate the output y t at time step t :

$$y_t = \text{softmax}(W^S h_t)$$

- Using the hidden state at the current time step and the appropriate weight W, we calculate the outputs (S). To build a probability vector that will assist us in determining the outcome, Softmax is applied (e.g. word in the question-answering problem).
- This model's strength rests in its ability to map diverse length sequences to one another. As you can see, the inputs and outputs do not always have the same length. As a result, a whole new set of issues may now be resolved utilising such an architecture.

4.4.3 Applications of Encoder-Decoder RNN Models

- It possesses many applications such as :
- Google's Machine Translation
- Question answering chatbots
- Speech recognition
- Time Series Application etc.,
- It is only that in these applications enocoder-decoder RNN models have been used.

4.5 Deep Recurrent Networks

- Most recurrent neural networks' computation may be broken down into three blocks of parameters and related transformations:
  1. From the input to the concealed state
  2. Moving on to the following concealed state from the previous one
  3. From the output to the concealed state
- Each of these three blocks corresponds to a single weight matrix in the RNN architecture depicted, meaning that when the network is unfurled, each of them corresponds to a shallow transformation.
- The term "shallow Transformation" refers to a transformation that would typically be represented by a learnt affine transformation followed by a fixed nonlinearity as a single layer inside a deep MLP.
- Experimental evidence strongly suggests that deepening the recurrent neural network is advantageous. That we need enough depth in order to perform the required transformations.

Depth can be introduced in RNN in three ways :

1. **Recurrent states broken down into groups**
   - It is possible to imagine that lower levels of the hierarchy have a part to play in converting the raw input into a representation that is more suitable for the upper levels of the hidden state.

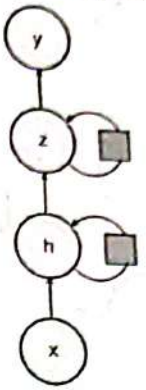


Fig. 4.5.1 : Deep RNN 1

2. **Deeper computation in hidden-to-hidden**
   - Go a step further and propose to have a separate Multi LP (possibly deep) for each of the three blocks:
     1. From the input to the hidden state
     2. From the previous hidden state to the next hidden state
     3. From the hidden state to the output
   - Considerations of representational capacity suggest that to allocate enough capacity in each of these three steps.
   - But doing so by adding depth may hurt learning by making optimization difficult.
   - In general it is easier to optimize shallower architectures.
   - Adding the extra depth makes the shortest time of a variable from time step t to a variable in time step t + 1 become longer.

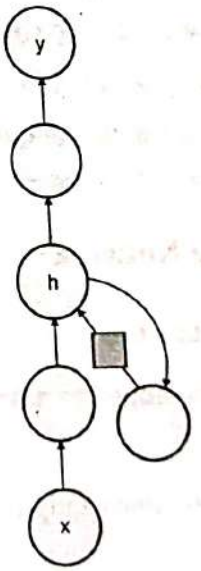


Fig. 4.5.2 : Deep RNN 2

3. Introducing skip connections

The length of the shortest path between variables in any two different time steps has doubled when an MLP with a single hidden layer is used for the state-to-state transition in comparison to an ordinary RNN. This can be mitigated by adding skip connections in the hidden-to-hidden path, as shown here.

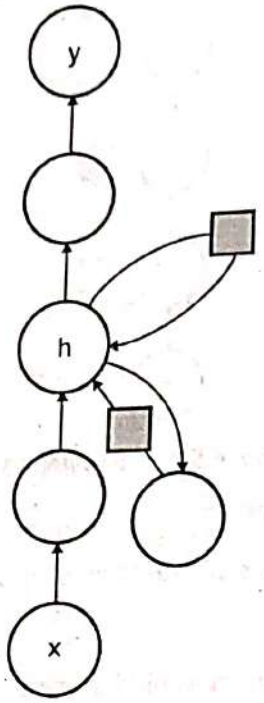


Fig. 4.5.3 : Deep RNN 3

4.6 Recursive Neural Networks

- A subset of deep neural networks called recursive neural networks (RvNNs) are capable of learning organised and detailed data. By repeatedly using the same set of weights on structured inputs, RvNN enables you to obtain a structured prediction. Recursive refers to the neural network's application to its output.
- Recursive neural networks are capable of handling hierarchical data because of their in-depth tree-like structure. In a tree structure, parent nodes are created by combining child nodes.
- There is a weight matrix for every child-parent bond, and similar children have the same weights. To allow for recursive operations and the use of the same weights, the number of children for each node in the tree is fixed. When it's necessary to parse an entire sentence, RvNNs are employed.

.6.1 Computational Graph of a Recursive Network

It generalizes a recurrent network from a chain to a tree.

A variable sequence x(1), x(2), ... x(t) can be mapped to a fixed size representation (the output o), with a fixed set of parameters (the weight matrices U, V, W).

Fig. 4.6.1 illustrates supervised learning case in which target y is provided that is associated with the whole sequence

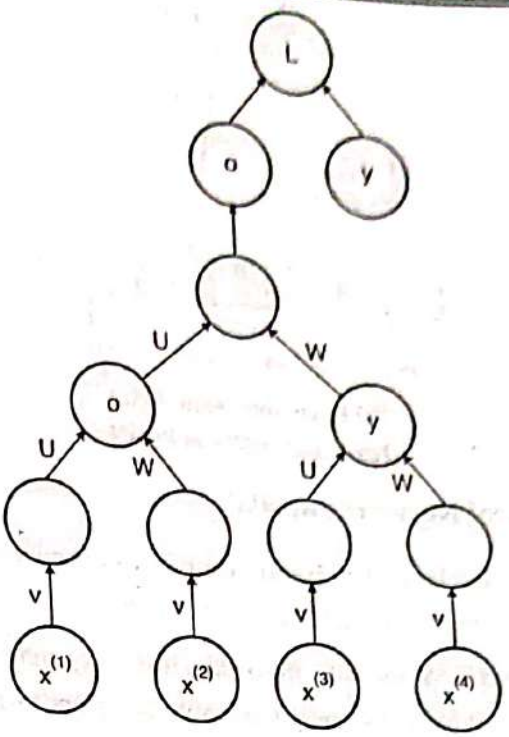
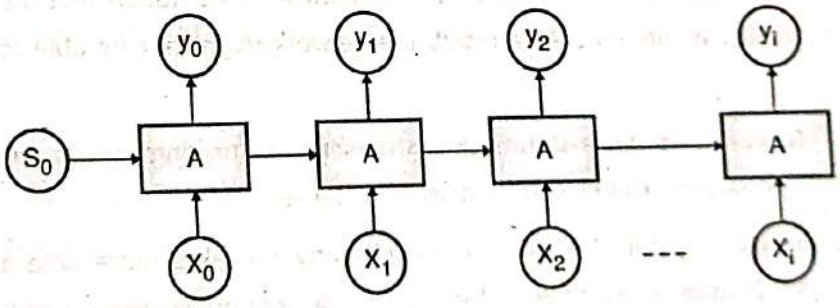


Fig. 4.6.1 : Recursive Neural Net Tree

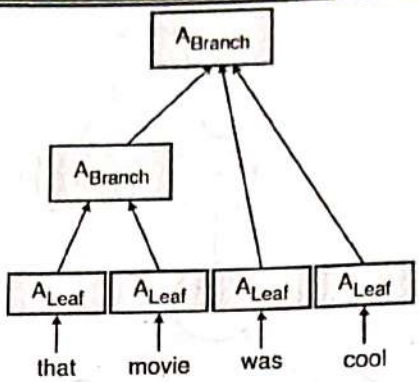
4.6.2 Recurrent Neural Network vs. Recursive Neural Networks

- Another well-known class of neural networks for processing sequential data is recurrent neural networks (RNNs). They are connected to the recursive neural network in a close way.
- Given that language-related data like sentences and paragraphs are sequential in nature, recurrent neural networks are useful for representing temporal sequences in natural language processing (NLP). Chain structures are frequently used in recurrent networks. By distributing the weights over the entire chain length, the dimensionality is maintained.
- Recursive neural networks, on the other hand, work with hierarchical data models because of their tree structure. The tree can perform recursive operations and use the same weights at each step because each node has a fixed number of children. Parent representations are created by combining child representations.
- The efficiency of a recursive network is higher than a feed-forward network.
- Recurrent Networks are recurrent over time, meaning recursive networks are just a generalization of the recurrent network.



(a) Recurrent Neural Net

Fig. 4.6.2 : RNN vs RvNN



(b) Recursive Neural Net

Fig. 4.6.2 : RNN vs RvNN

4.6.3 Need for Recursive Neural Networks in NLP

- Methods using deep learning using a large data corpus, learn low-dimensional, realvalued vectors for word tokens that successfully capture the syntactic and semantic aspects of text.
- In order to aggregate tokens into a vector with fixed dimensionality that can be used for other NLP tasks, a compositional model is first required for tasks where the inputs are larger text units, such as phrases, sentences, or documents.
- Recursive and recurrent models are two types of models for accomplishing this.

4.6.4 Advantages of RvNN in NLP

- The structure and decrease in network depth of recursive neural networks are their two main advantages for natural language processing.
- Recursive Neural Networks' tree structure, as previously mentioned, can manage hierarchical data, such as in parsing issues.
- The ability for trees to have a logarithmic height is another advantage of RvNN. A recursive neural network may depict a binary tree with a height of O (log n) when there are O(n) input words. The distance between the first and last input elements is shortened as a result. As a result, the long-term dependence becomes more manageable and shorter.

4.6.5 Disadvantages of RvNN in NLP

- The tree structure of recursive neural networks may be their biggest drawback. Using the tree structure suggests giving our model a special inductive bias. The bias is consistent with the notion that the data are organised in a tree hierarchy. But the reality is different. As a result, the network might not be able to pick up on the current trends.
- The Recursive Neural Network also has a drawback in that sentence parsing can be cumbersome and slow. It's interesting that different parse trees can exist for the same sentence.
- Additionally, labelling the training data for recursive neural networks takes more time and effort than building recurrent neural networks. It takes more time and effort to manually break down a sentence into smaller parts than it does to give it a label.

4.7 The Challenge of Long-Term Dependencies

- Deep computational graphs pose a challenge for neural network optimization, as seen, for example, in feedforward networks with several layers and RNNs that continually execute the same operation at each time step of a lengthy temporal sequence.
- Gradients propagated over numerous stages typically disappear or explode (damaging optimization).
- Long-term dependencies are challenging because long-term interactions are given exponentially smaller weights (involving multiplication of many Jacobians)
- Vanishing and Exploding Gradient Problem
- Suppose a computational graph consists of repeatedly multiplying by a matrix W.
- After t steps this is equivalent to multiplying by Wt.
- Suppose W has an eigen decomposition $W=V\text{diag}(\lambda)V^{-1}$.
- In this case it is straightforward to see that $W^t = (V\text{diag}(\lambda)V^{-1})^t = V\text{diag}(\lambda)^t V^{-1}$.
- Any eigenvalues λ_i that are not near an absolute value of 1 will either explode if they are greater than 1 in magnitude and vanish if they are less than 1 in magnitude.
- Vanishing gradients make it difficult to know which direction the parameters should move to improve cost.
- Exploding gradients make learning unstable.

4.8 Echo State Networks

- Recurrent neural networks, such as echo state networks, which are a component of the reservoir computing framework, have the following characteristics:

1. The ratios of the input to the buried layer (the "reservoir"): as well as the "reservoir's" weights Wr are assigned at random and cannot be trained.
2. In order for the network to replicate particular temporal patterns, the weights of the output neurons (the "readout" layer) can be trained and learnt.
3. The buried layer, also known as the reservoir, is often just 10% linked.
4. A recurrent nonlinear embedding of the input is created by the reservoir architecture (H in the image below), which can then be connected to the desired output. The resulting weights are trainable.
5. It is possible to connect the embedding to a different predictive model (a trainable NN or a ridge regressor/SVM for classification problems).

Reservoir Computing

- A higher dimension representation is produced via reservoir computing, which is an extension of neural networks in which the input signal is coupled to a fixed (non-trainable) and random dynamical system (the reservoir) (embedding). The required output is then coupled to this embedding using trainable units.
- The Extreme Learning Machine, which consists only of feed forward networks with only the readout layer trainable, is the non-recurrent equivalent of reservoir computing.

Working

In response to an input of the form N, T, V, where N denotes the number of observations, T the number of time steps, and V the number of variables, we will :

1. Select the reservoir's R size as well as other factors affecting the degree of connection sparsity, whether or not we want to represent leakage, the appropriate number of components following dimensionality reduction, etc.
2. By selecting samples from a random binomial distribution, create (V, R) input weights Win.
3. Create (R, R) reservoir weights Wr by taking a sample from a uniform distribution with a specified density, where density controls the degree of sparsity.
4. Calculate the high-dimensional state representation H as a nonlinear function of the input at the current time step (N, V) multiplied by the internal weights plus the prior state multiplied by the reservoir matrix (usually tanh) (R, R).
5. An alternative is to use a dimensionality reduction algorithm, like PCA to D components, to reduce the number of dimensions from (N, T, D).
6. One way to create an input representation is to train a regressor to map states from t to t+1 using, for instance, the complete reservoir. Another representation may be the matrix of all calculated slopes and intercepts. Using the mean or the most recent value of H might also be an option.
7. Connect this embedding to the intended output using either a trainable NN structure or another kind of predictor.

Why and when should you use Echo State Networks?

- The parameters in the hidden layers of traditional NN architectures don't change all that much or they cause numeric instability and chaotic behaviour because of the vanishing/exploding gradient problem. These issues don't exist in echo state networks.
- Because there is no back propagation phase on the reservoir, Echo State Networks are extremely fast compared to traditional NN architectures, which are computationally expensive.
- Bifurcations can cause traditional NN to break down.
- Chaotic time series can be handled by ESN with ease.

4.9 Leaky Units and Other Strategies for Multiple Time Scales

The Goal is to deal with long-term dependencies.

1. Some model parts operate at fine-grained time scales and handle small details

Strategies which are useful to build fine and coarse time scales are

(i) Adding skip connections through time:

- Add direct connections from variables in the distant past to variables in the present.
- In an ordinary RNN, recurrent connection goes from time t to time t+1. Can construct RNNs with longer delays.
- Gradients can vanish/explode exponentially wrt no. of time steps.
- Introduce time delay of d to mitigate this problem.

- Gradients diminish as a function of τ/d rather than τ.
- Allows learning algorithm to capture longer dependencies(Not all long-term dependencies can be captured this way).

(ii) Leaky units and a spectrum of different time scales to integrate signals with different time constants

- Rather than an integer skip of d time steps, the effect can be obtained smoothly by adjusting a real-valued α.
- Running Average: Running average $\mu^{(t)}$ of some value $v^{(t)}$ is $\mu^{(t)} \leftarrow \alpha\mu^{(t-1)}+(1-\alpha)v^{(t)}$. It is called a linear self-correction. When α is close to 1, running average remembers information from the past for a long time and when it is close to 0, information is rapidly discarded.
- Hidden units with linear self connections behave similar to running average. They are called leaky units.
- Can obtain product of derivatives close to 1 by having linear self-connections and a weight near 1 on those connections.

(iii) Removal of some connections to model finegrained time scales

- Another approach to handle long-term dependencies.
- Organize state of the RNN at multiple time scales.
- Information flowing more easily through long distances at the slower time scales.
- It involves actively removing length one connections and replacing them with longer connections.
- Skip connections add edges

2. Other parts operate at coarse time scales

Transfer information from the distant past to the present more efficiently.

4.10 The Long Short-Term Memory and Other Gated RNNs

As mentioned earlier Vanishing gradients and Exploding Gradients are two major issues in recurrent neural network and to solve these issues Long Short Term Memory Networks and Gated Recurrent Unit have been put forward.

4.10.1 Long Short Term Memory Networks

- An enhanced RNN, or sequential network, called a long short-term memory network, permits information to endure. It is capable of resolving the RNN's vanishing gradient issue. RNNs, also referred to as recurrent neural networks, are utilised for persistent memory.
- RNN retain the prior knowledge and apply it to the processing of the incoming data. Due to diminishing gradient, RNN have the flaw of being unable to recall long-term dependencies. Long-term dependency issues are specifically avoided when designing LSTMs.

LSTM Architecture

- LSTM functions on a high level very similarly to an RNN cell. The LSTM network's internal operation is seen below. As seen in the Fig. 4.10.1, the LSTM is composed of three sections, each of which has a distinct function.

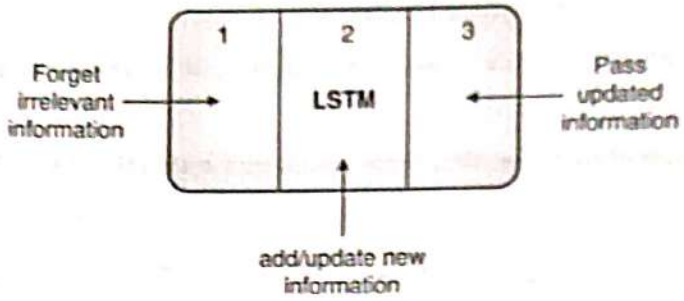


Fig. 4.10.1 : LSTM Components

- The first section determines whether the information from the preceding timestamp needs to be remembered or can be ignored. The cell attempts to learn new information from the input to this cell in the second section. The cell finally transmits the revised data from the current timestamp to the next timestamp in the third section.
- Gates refer to these three LSTM cell components. The Forget gate, Input gate, and Output gate are the names of the three components, respectively.

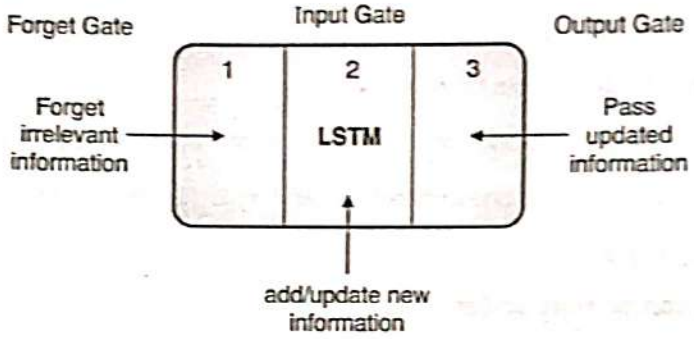


Fig. 4.10.2 : LSTM Gates

- An LSTM has a hidden state, just like a straightforward RNN, with H(t-1) standing for the hidden state of the prior timestamp and Ht for the hidden state of the present timestamp. Additionally, LSTMs have a cell state that is denoted by the timestamps C(t-1) and C(t), which stand for the prior and current timestamps, respectively.
- In this case, the cell state is referred to as the long-term memory and the hidden state as the short-term memory. See the illustration in Fig. 4.10.3.

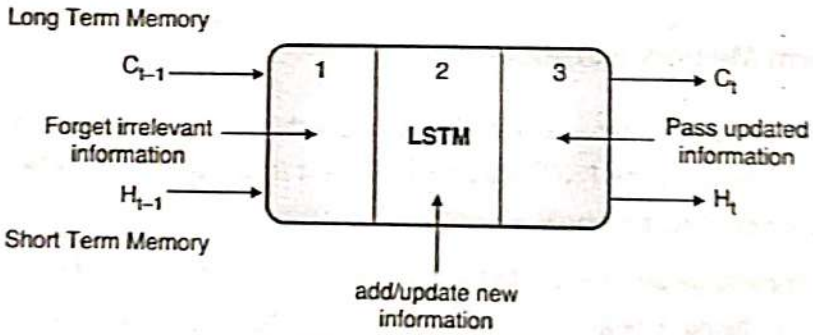


Fig. 4.10.3 : LSTM Cell State

- It's interesting to note that all of the timestamps are carried by the cell state along with the information.

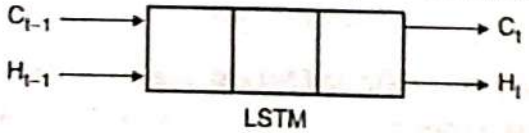


Fig. 4.10.4 : LSTM Gate

Bob is a nice person. Dan on the other hand is evil.

To comprehend how LSTM functions, let's look at an example. Here, a full stop separates two phrases. Dan, on the other hand, is nasty, and the first sentence reads, "Bob is a lovely person." It is crystal evident that we are talking about Bob in the first phrase and switching to Dan as soon as we reach the full stop (.).

Our network should understand that we have stopped talking about Bob as soon as we go from the first to the second phrase. Now Dan is the focus. Here, the network's Forget gate enables it to forget about it. Let's examine the functions these gates perform in the LSTM architecture.

1. **Forget Gate**

- The initial step in an LSTM network cell is to choose whether to keep or discard the data from the preceding timestamp. The forget gate equation is given below.

Forget gate : $f_t = \sigma (x_t \times U_f + H_{t-1} \times W_f)$

Here,

X_t: input to the current timestamp.

U_i: weight associated with the input

H_{t-1}: The hidden state of the previous timestamp

W_i: It is the weight matrix associated with hidden state

- Later, a sigmoid function is applied over it. That will make ft a number between 0 and 1. This ft is later multiplied with the cell state of the previous timestamp as shown below.

$C_{t-1} \times f_t = 0$ ...if $f_t = 0$ (forget everything)

$C_{t-1} \times f_t = C_{t-1}$ ...if $f_t = 1$ (forget nothing)

- The network will forget everything if ft is set to 0, but nothing if ft is set to 1. Returning to our example, the first line discussed Bob, and following a full stop, the network will come across Dan. Ideally, the network should have forgotten about Bob.

2. **Input Gate**

- "Bob has swimming skills. He told me over the phone that he had spent four arduous years in the navy."
- So, Bob is the subject of both of these phrases. However, each offers a unique perspective on Bob. He knows how to swim, as indicated in the first sentence. The second sentence, on the other hand, mentions that he uses a phone and spent four years in the navy.
- Just consider which details in the second line are crucial in light of the context provided in the first. He first mentioned his service in the navy over the phone. It makes no difference in this situation whether he sent the information via phone or another method of contact. We want our model to keep in mind that it is significant information that he served in the navy. The Input gate's job is to perform this.
- The value of the fresh information carried by the input is measured by the input gate. The input gate's equation is shown below.

Input gate : $i_t = \sigma (x_t \times U_i + H_{t-1} \times W_i)$

Here,

X_t: Input at the current timestamp t

U_i: weight matrix of input

H_{t-1}: A hidden state at the previous timestamp

W_i: Weight matrix of input associated with hidden state

We once more applied the sigmoid function to it. As a result, I will have a value between 0 and 1 at timestamp t.

New Information : $N_t = \tanh(x_t \times U_c + H_{t-1} \times W_c)$ (new information)

- The new data that had to be sent to the cell state now depends on a concealed state at timestamp t-1 in the past and input x at timestamp t. Tanh is the activation function in this case. The tanh function causes the value of fresh information to range from – 1 to 1. The information is deducted from the cell state if the value of N_t is negative, and added to the cell state at the current timestamp if the value is positive.
- The N_t won't, however, be added to the cell state immediately. This is the revised equation.

$$C_t = (f_t \times C_{t-1} + i_t \times N_t) \text{ (updating cell state)}$$

- In this case, C_{t-1} represents the cell state at the current timestamp, and the other variables are those we previously calculated.

3. Output Gate

- Consider the following sentence.
- "Bob faced the enemy alone and gave his life for his nation. brave_____, for his contributions."
- We have to finish the second sentence in this task. Today, we immediately recognise a person when we hear the word brave. Bob is the only one being brave in the phrase; we cannot say that the nation or the enemy is brave. We must therefore provide a pertinent word to fill in the blank depending on the existing expectation. This is the purpose of our Output gate, and that term is our output.
- The Output gate's equation, which is quite similar to the equations for the two earlier gates, is shown below.

Output gate : $o_t = \sigma(x_t \times U_o + H_{t-1} \times W_o)$

- Due to this sigmoid function, it will also have a value between 0 and 1. We will now use O_t and tanh of the updated cell state to determine the current hidden state. As displayed below.

$$H_t = O_t \times \tanh(C_t)$$

- It turns out that the concealed state depends on both the present output and long-term memory (C_t). Simply activate SoftMax on hidden state H_t if you need to take the output of the current timestamp.

$$\text{Output} = \text{Softmax}(H_t)$$

- Prediction is the token in this case having the highest score in the output.
- This is the LSTM network's more understandable diagram.

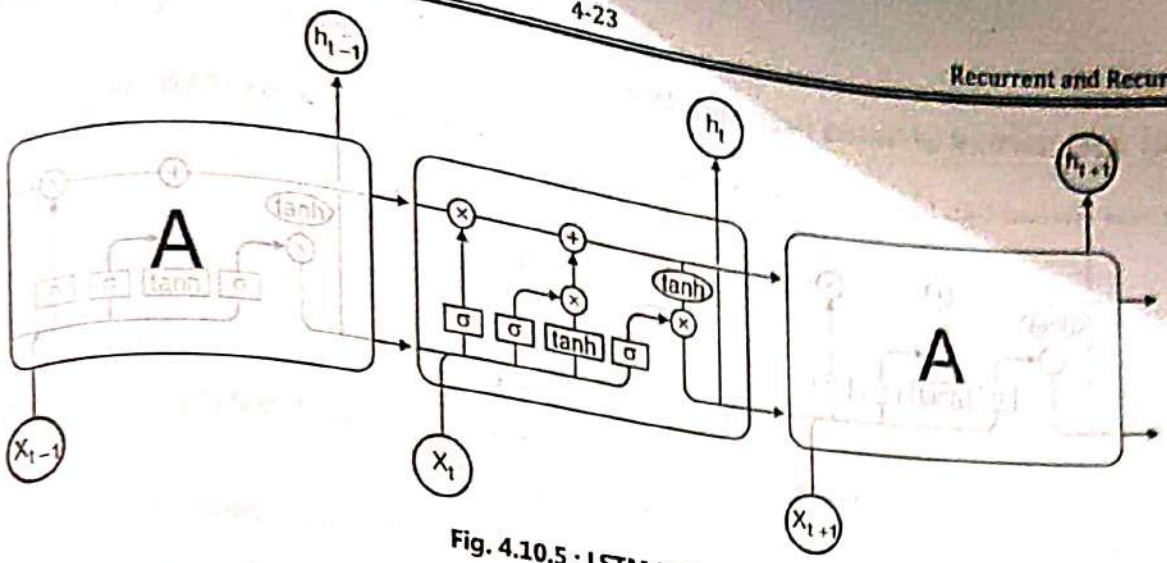


Fig. 4.10.5 : LSTM Network

4.10.2 Gated Recurrent Unit

- A development of the traditional RNN, or recurrent neural network, is the GRU, or gated recurrent unit. In the year 2014, Kyunghyun Cho and others introduced it.
- GRUs and Long Short Term Memory are quite similar (LSTM). GRU use gates to regulate the information flow, just like LSTM. When compared to LSTM, they are quite new. They have a simpler architecture and provide certain improvements over LSTM because of this.

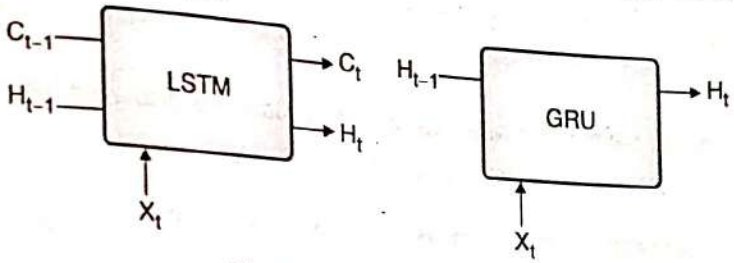


Fig. 4.10.6 : LSTM and GRU

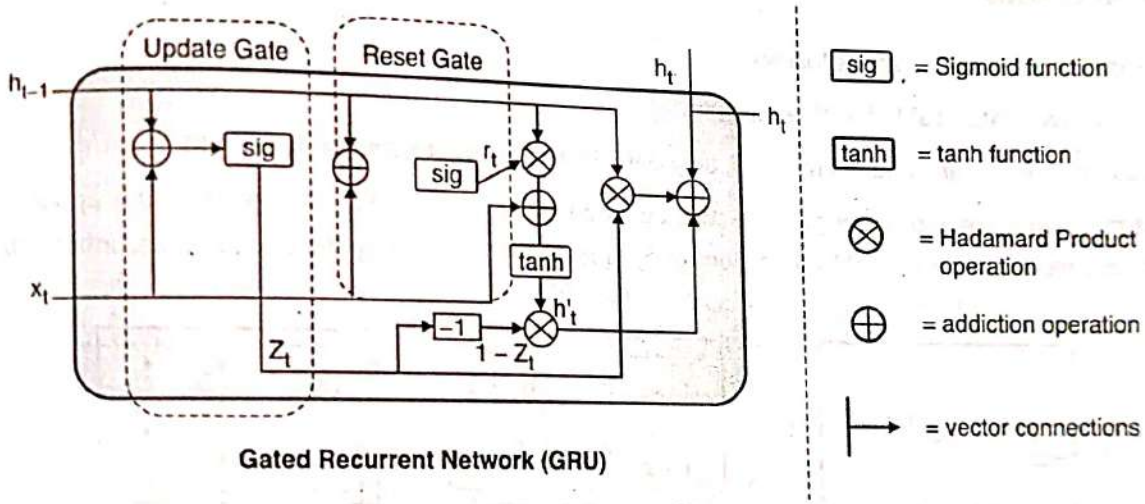


Fig. 4.10.7 : GRU Architecture

- Another intriguing feature of GRU is that, in contrast to LSTM, it lacks a distinct cell state (Ct). There is just a hidden state (Ht). GRUs are quicker to train because of the architecture's simplicity.

4.10.2(A) Architecture of Gated Recurrent Unit

- Let's now see how GRU functions. Here, we have a GRU cell that resembles an LSTM or RNN cell more or less.

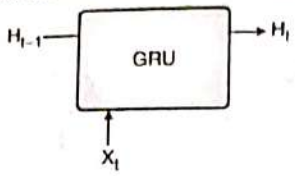


Fig. 4.10.8 : GRU Gates

- It accepts an input Xt and the hidden state Ht-1 from timestamp t-1 for each timestamp t. Later, a new hidden state, Ht, is output and again given to the following timestamp.
- Currently, a GRU has primarily two gates as opposed to an LSTM cell's three gates. The Reset gate is the first gate, while the Update gate is the second.

1. Reset Gate

- The network's short-term memory, or hidden state, is handled by the reset gate (Ht). The Reset gate's equation is given below.

$$r_t = \sigma (x_t * U_r + H_{t-1} * w_f)$$

- This is pretty similar to the LSTM gate equation, if you recall. The sigmoid function will lead rt to have a value between 0 and 1. The reset gate's weight matrices, Ur and Wr, are shown here.

2. Update Gate

- For long-term memory, we have an Update gate, and its equation is displayed below.

$$u_t = \sigma (x_t * U_u + H_{t-1} * W_u)$$

- The only distinction is between the weight measurements, Uu and Wu.

4.10.3 LSTM vs GRU

- The few differencing points are as follows:
- The GRU has two gates, LSTM has three gates
- GRU does not possess any internal memory, they don't have an output gate that is present in LSTM
- In LSTM the input gate and target gate are coupled by an update gate and in GRU reset gate is applied directly to the previous hidden state. In LSTM the responsibility of reset gate is taken by the two gates i.e., input and target.

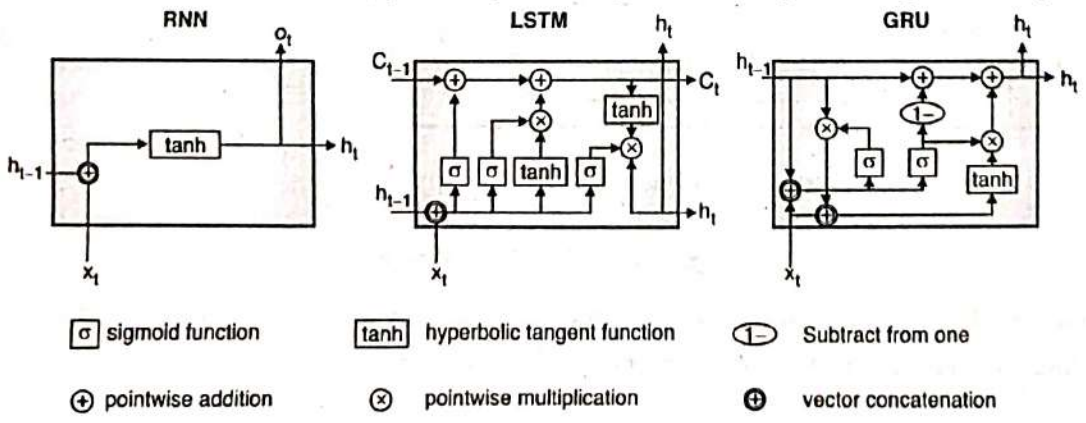


Fig. 4.10.9 : RNN vs LSTM vs GRU

4.11 Optimization for Long-Term Dependencies

- There are two ways to optimize Long-Term Dependencies,
  - **Clipping gradient :** This avoids gradient explode but NOT gradient vanish. One option is to clip the parameter gradient from a minibatch element-wise just before the parameter update. Another is to clip the norm ||g|| of the gradient 'g' just before the parameter update.
  - **Regularizing :** Regularizing is done to encourage the information flow. Favor gradient vector being back-propagated to maintain its magnitude, i.e. penalize the L2 norm differences between such vectors.
- It is important to note that since using LSTM will usually solve the long-term dependency problem, these techniques are not very useful today. Even so, it's useful to be familiar with time-tested techniques.
- Creating a model that is simple to optimise is frequently easier than creating a powerful optimization algorithm.

4.12 Explicit Memory

- Implicit knowledge is a specialty of neural networks. They have trouble, nevertheless, remembering facts. Therefore, adding explicit memory components can be useful for sequential reasoning as well as for quickly and "intentionally" storing and retrieving specific facts.
- Memory networks have a collection of memory cells they can access using an addressing mechanism, but they also need a supervision signal telling them how to use those memory cells.
- Neural Turing machine, which can read from and write any material to memory cells without explicit guidance on what to do next.
- The employment of a content-based soft attention mechanism in the neural Turing machine (NTM) enables end-to-end training without the need for an external supervision signal (Bahdanau et al., 2014). Functions that output accurate, integer addresses are challenging to optimise. NTMs actually read to or write from a lot of memory cells at once to solve this issue.

Note : Reading the original paper may help you better understand NTM. However, the following graph serves as an example:

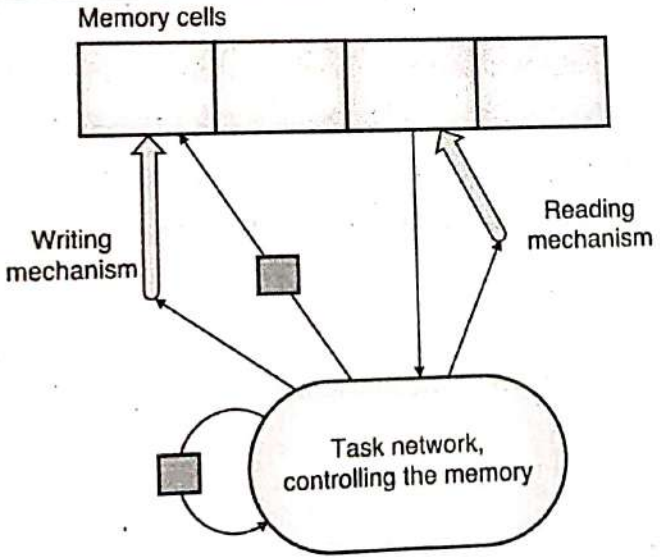


Fig. 4.12.1 : Explicit Memory

- Fig. 4.12.1 showa a schematic of an example of a network with an explicit memory, capturing some of the design elements of the neural Turing machine. In this diagram we distinguish the "representation" part of model (the "task network," here a recurrent net in the bottom) from the "memory" part of the model (the cells), which can store facts.
- The task network learns to "control" the memory, deciding where to read from and where to write to within memory (through the reading and writing mechanisms, indicated by bold arrows pointing at the reading writing addresses).

Review Questions

Q. 1 Explain RNN

Q. 2 Explain LSTM and Bidirectional LSTM

Q. 3 What the varients of RNN and explain

Q. 4 What is Deep recurrent networks

Q. 5 What are the challenges of long term dependencies

Q. 6 Explain Encoder-Decoder RNN model

5 Deep Generative Models

Unit V

Syllabus

Introduction to deep generative model, Boltzmann Machine, Deep Belief Networks, Generative adversarial network (GAN), discriminator network, generator network, types of GAN, Applications of GAN networks

5.1 Introduction to Deep Generative Model

In this chapter, we present several of the specific kinds of generative models that can be built and trained using the techniques like Structured Probabilistic Models, Monte Carlo Methods, Inference etc.

All of these models represent probability distributions over multiple variables in some way. Some allow the probability distribution function to be evaluated explicitly. Others do not allow the evaluation of the probability distribution function, but support operations that implicitly require knowledge of it, such as drawing samples from the distribution.

Some of these models are structured probabilistic models described in terms of graphs and factors, using the language of graphical models. Others cannot easily be described in terms of factors, but represent probability distributions, nonetheless.

5.2 Boltzmann Machine

- Boltzmann machines were originally introduced as a general "connectionist" approach to learning arbitrary probability distributions over binary vectors (Fahlman et al., 1983; Ackley et al., 1985; Hinton et al., 1984; Hinton and Sejnowski, 1986).
- Variants of the Boltzmann machine that include other kinds of variables have long ago surpassed the popularity of the original. In this section we briefly introduce the binary Boltzmann machine and discuss the issues that come up when trying to train and perform inference in the model.
- We define the Boltzmann machine over a d-dimensional binary random vector $x \in \{0, 1\}^d$. The Boltzmann machine is an energy-based model, meaning we define the joint probability distribution using an energy function :

$$P(x) = \frac{\exp(-E(x))}{Z} \qquad ...(5.2.1)$$

where E (x) is the energy function and Z is the partition function that ensures that $\sum_x P(x) = 1$. The energy function of the Boltzmann machine is given by

$$E(x) = -x^T Ux - b^T x \qquad ...(5.2.2)$$

where U is the "weight" matrix of model parameters and b is the vector of bias parameters.

- In the general setting of the Boltzmann machine, we are given a set of training examples, each of which are n-dimensional. Equation 5.2.1 describes the joint probability distribution over the observed variables. While this scenario is certainly viable, it does limit the kinds of interactions between the observed variables to those described by the weight matrix. Specifically, it means that the probability of one unit being on is given by a linear model (logistic regression) from the values of the other units.
- The Boltzmann machine becomes more powerful when not all the variables are observed. In this case, the latent variables, can act similarly to hidden units in a multi-layer perceptron and model higher-order interactions among the visible units. Just as the addition of hidden units to convert logistic regression into an MLP results in the MLP being a universal approximator of functions, a Boltzmann machine with hidden units is no longer limited to modeling linear relationships between variables. Instead, the Boltzmann machine becomes a universal approximator of probability mass functions over discrete variables (Le Roux and Bengio, 2008). Formally, we decompose the units x into two subsets: the visible units v and the latent (or hidden) units h. The energy function becomes

$$E(v, h) = -v^T Rv - v^T Wh - h^T Sh - b^T v - c^T h \quad \ldots(5.2.3)$$

- Boltzmann Machine Learning Learning algorithms for Boltzmann machines are usually based on maximum likelihood. All Boltzmann machines have an intractable partition function, so the maximum likelihood gradient must be approximated using the techniques like one interesting property of Boltzmann machines when trained with learning rules based on maximum likelihood is that the update for a particular weight connecting two units depends only the statistics of those two units, collected under different distributions: P_{model} (v) and P^*_{data} (v)P_{model} (h | v).
- The rest of thenetwork participates in shaping those statistics, but the weight can be updated without knowing anything about the rest of the network or how those statistics were produced. This means that the learning rule is "local," which makes Boltzmann machine learning somewhat biologically plausible. It is conceivable that if each neuron were a random variable in a Boltzmann machine, then the axons and dendrites connecting two random variables could learn only by observing the firing pattern of the cells that they actually physically touch.
- In particular, in the positive phase, two units that frequently activate together have their connection strengthened. This is an example of a Hebbian learning rule (Hebb, 1949) often summarized with the mnemonic "fire together, wire together." Hebbian learning rules are among the oldest hypothesized explanations for learning in biological systems and remain relevant today (Giudice et al., 2009).
- Other learning algorithms that use more information than local statistics seem to require us to hypothesize the existence of more machinery than this. For example, for the brain to implement back-propagation in a multilayer perceptron, it seems necessary for the brain to maintain a secondary communication network for transmitting gradient information backwards through the network.
- Proposals for biologically plausible implementations (and approximations) of back-propagation have been made (Hinton, 2007a; Bengio, 2015) but remain to be validated, and Bengio (2015) links back-propagation of gradients to inference in energy-based models similar to the Boltzmann machine (but with continuous latent variables).

5.3 Deep Belief Networks

- *Deep belief networks (DBNs) were one of the first non-convolutional models to successfully admit training of deep architectures (Hinton et al., 2006; Hinton, 2007b).*

The introduction of deep belief networks in 2006 began the current deep learning renaissance. Prior to the introduction of deep belief networks, deep models were considered too difficult to optimize. Kernel machines with convex objective functions dominated the research landscape.

Deep belief networks demonstrated that deep architectures can be successful, by outperforming kernelized support vector machines on the MNIST dataset (Hinton et al., 2006).

Today, deep belief networks have mostly fallen out of favor and are rarely used, even compared to other unsupervised or generative learning algorithms, but they are still deservedly recognized for their important role in deep learning history.

Deep belief networks are generative models with several layers of latent variables. The latent variables are typically binary, while the visible units may be binary or real. There are no intralayer connections.

Usually, every unit in each layer is connected to every unit in each neighbouring layer, though it is possible to construct more sparsely connected DBNs. The connections between the top two layers are undirected. The connections between all other layers are directed, with the arrows pointed toward the layer that is closest to the data.

A DBN with l hidden layers contains l weight matrices: W(1), . . . ,W(l). It also contains l + 1 bias vectors : $b^{(0)}$, ... $b^{(l)}$,with $b^{(0)}$ providing the biases for the visible layer. The probability distribution represented by the DBN is given by :

$$P(h^{(l)}, h^{(l-1)}) \propto \exp\left(b^{(l)^T} h^{(l)} + b^{(l-1)T} h^{(l-1)} + h^{(l-1)T} W^{(l)} h^{(l)}\right)$$

$$P\left(h_i^{(k)} 1 \mid h^{(k+1)}\right) = \sigma\left(b_i^{(k)} + W_i^{(k+1)T} h^{(k+1)}\right) \forall i, \forall k \in 1, \ldots, l-2,$$

$$P(v_i = 1 \mid h(1)) = \sigma\left(b_i^{(0)} + W_i^{(1)T} h^{(1)}\right) \forall i$$

- In the case of real-valued visible units, substitute

$$v \sim N(v; b^{(0)} + W^{(1)T} h^{(1)}, \beta^{-1})$$

with β diagonal for tractability. Generalizations to other exponential family visible units are straightforward, at least in theory. A DBN with only one hidden layer is just an RBM.

- To generate a sample from a DBN, we first run several steps of Gibbs sampling on the top two hidden layers. This stage is essentially drawing a sample from the RBM defined by the top two hidden layers.
- We can then use a single pass of ancestral sampling through the rest of the model to draw a sample from the visible units.
- Deep belief networks incur many of the problems associated with both directed models and undirected models.
- Inference in a deep belief network is intractable due to the explaining away effect within each directed layer, and due to the interaction between the two hidden layers that have undirected connections. Evaluating or maximizing the standard evidence lower bound on the log-likelihood is also intractable, because the evidence lower bound takes the expectation of cliques whose size is equal to the network width.
- Evaluating or maximizing the log-likelihood requires not just confronting the problem of intractable inference to marginalize out the latent variables, but also the problem of an intractable partition function within the undirected model of the top two layers.

- To train a deep belief network, one begins by training an RBM to maximize **Ev~pdata log p(v)** using contrastive divergence or stochastic maximum likelihood. The parameters of the RBM then define the parameters of the first layer of the DBN. Next, a second RBM is trained to approximately maximize

$$E_{v \sim p_{data}} E_{h^{(1)} \sim p^{(1)}(h^{(1)} \mid v)} \log p^{(2)}(h^{(1)})$$

where p(1) is the probability distribution represented by the first RBM and p(2) is the probability distribution represented by the second RBM.

- In other words, the second RBM is trained to model the distribution defined by sampling the hidden units of the first RBM, when the first RBM is driven by the data. This procedure can be repeated indefinitely, to add as many layers to the DBN as desired, with each new RBM modeling the samples of the previous one. Each RBM defines another layer of the DBN. This procedure can be justified as increasing a variational lower bound on the log-likelihood of the data under the DBN (Hinton et al., 2006).
- In most applications, no effort is made to jointly train the DBN after the greedy layer-wise procedure is complete. However, it is possible to perform generative fine-tuning using the wake-sleep algorithm.
- The trained DBN may be used directly as a generative model, but most of the interest in DBNs arose from their ability to improve classification models. We can take the weights from the DBN and use them to define an MLP :

$$h^{(1)} = \sigma(b^{(1)} + v^T W^{(1)})$$

$$h^{(1)} = \sigma\left(b_i^{(1)} + h^{(l-1)T} W^{(l)}\right) \forall l \in 2, \ldots, m,$$

- After initializing this MLP with the weights and biases learned via generative training of the DBN, we may train the MLP to perform a classification task. This additional training of the MLP is an example of discriminative fine-tuning.
- This MLP is a heuristic choice that seems to work well in practice and is used consistently in the literature. Many approximate inference techniques are motivated by their ability to find a maximally tight variational lower bound on the log-likelihood under some set of constraints.
- One can construct a variational lower bound on the log-likelihood using the hidden unit expectations defined by the DBN's MLP, but this is true of any probability distribution over the hidden units, and there is no reason to believe that this MLP provides a particularly tight bound. In particular, the MLP ignores many important interactions in the DBN graphical model.
- The MLP propagates information upward from the visible units to the deepest hidden units, but does not propagate any information downward or sideways. The DBN graphical model has explaining away interactions between all of the hidden units within the same layer as well as top-down interactions between layers.
- The term "deep belief network" is commonly used incorrectly to refer to any kind of deep neural network, even networks without latent variable semantics. The term "deep belief network" should refer specifically to models with undirected connections in the deepest layer and directed connections pointing downward between all other pairs of consecutive layers.
- The term "deep belief network" may also cause some confusion because the term "belief network" is sometimes used to refer to purely directed models, while deep belief networks contain an undirected layer.
- Deep belief networks also share the acronym DBN with dynamic Bayesian networks (Dean and Kanazawa, 1989), which are Bayesian networks for representing Markov chains.

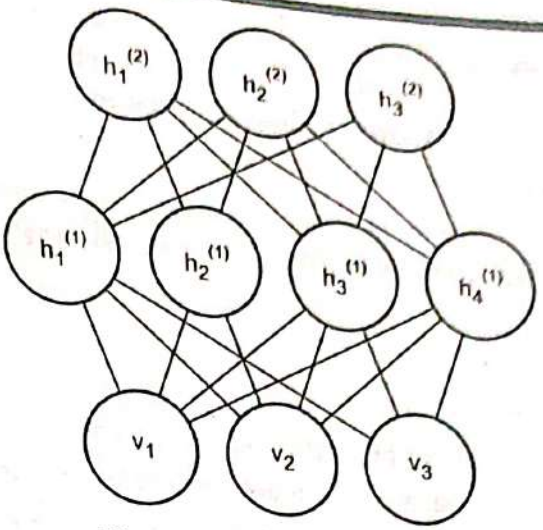


Fig. 5.3.1 : Deep belief network

- The graphical model for a deep Boltzmann machine with one visible layer (bottom) and two hidden layers. Connections are only between units in neighboring layers.
- There are no intralayer layer connections.

5.4 Generative Adversarial Network (GAN)

- Generative Adversarial Networks (GANs) are a kind of exceptionally potent neural networks used to support unsupervised learning. They were developed and debuted for the first time in 2014 by Ian J. Goodfellow. GANs can analyse changes within a set of data since they are made up of two neural networks that compete with one another.
- A generative modelling technique called GANs uses CNN and other deep learning techniques (Convolutional Neural Network). Generative modelling is an unsupervised learning technique that uses patterns in input data to automatically find and learn them so the model may be applied to fresh instances from the original dataset.
- By framing the task as a supervised learning problem and utilising GANs, generative adversarial networks can be trained.
- Discriminator This compares the images with real-world examples to classify fake and real images.
- Example The Generator generates random images (e.g., The Generator generates some random images (e.g., tables), and then the Discriminator compares these images with real-world table images. Finally, the Generator sends the feedback directly to Generator. See the GAN structure in the Fig. 5.4.1.

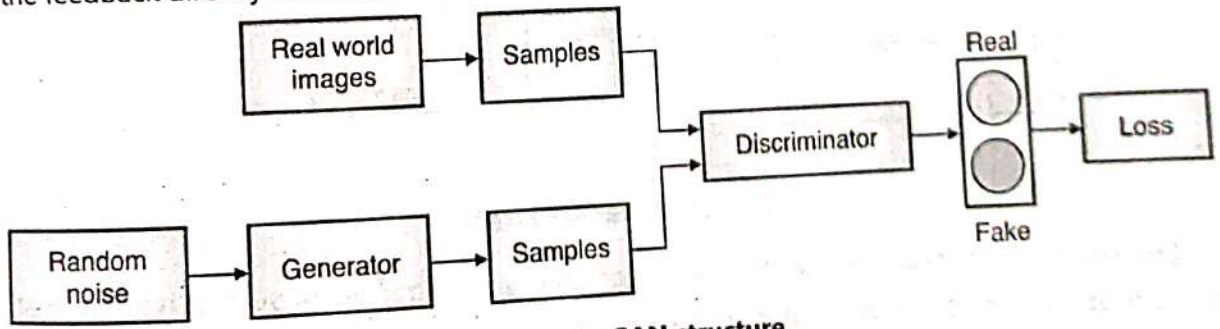


Fig. 5.4.1 : GAN structure

5.4.1 Why GANs was Developed?

- Machine learning algorithms and neural networks can easily be fooled to misclassify things by adding some amount of noise to data. After adding some amount of noise, the chances of misclassifying the images increase.
- Hence the small rise that, is it possible to implement something that neural networks can start visualizing new patterns like sample train data. Thus GANs were built that generate new fake results similar to the original.

5.4.2 How does GANs Work?

- GAN is basically an approach to generative modeling that generates a new set of data based on training data that look like training data. GANs have two main blocks(two neural networks) which compete with each other and are able to capture, copy, and analyze the variations in a dataset. The two models are usually called Generator and Discriminator which we will cover in Components on GANs. To understand the term GAN let's break it into separate three parts
- Generative – To learn a generative model, which describes how data is generated in terms of a probabilistic model. In simple words, it explains how data is generated visually.
- **Adversarial** – The training of the model is done in an adversarial setting.
- **Networks** – Use deep neural networks for training purposes.
- In GANs, there is a generator and a discriminator. The Generator generates fake samples of data(be it an image, audio, etc.) and tries to fool the Discriminator.
- The Discriminator, on the other hand, tries to distinguish between the real and fake samples. The Generator and the Discriminator are both Neural Networks and they both run in competition with each other in the training phase.
- The steps are repeated several times and in this, the Generator and Discriminator get better and better in their respective jobs after each repetition. The working can be visualized by the Fig. 5.4.2.

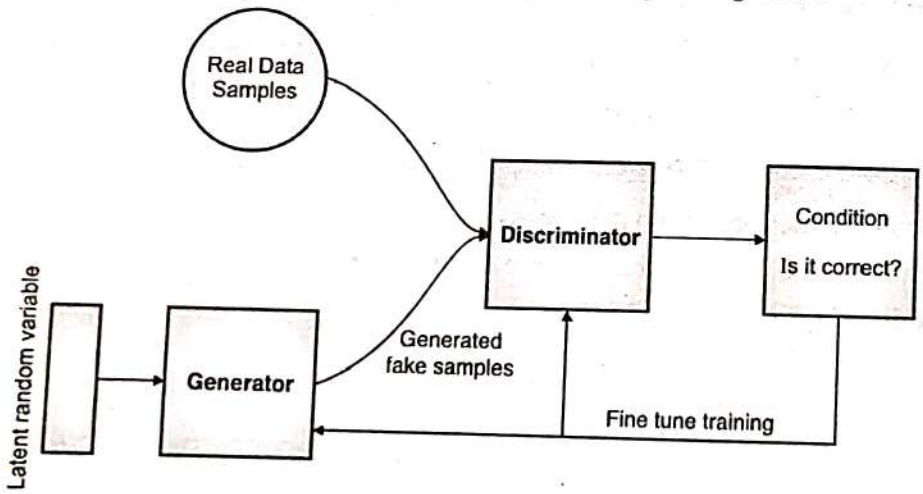


Fig. 5.4.2 : Working of GAN

- *Here, the generative model captures the distribution of data and is trained in such a manner that it tries to maximize the probability of the Discriminator in making a mistake. The Discriminator, on the other hand, is based on a model that estimates the probability that the sample that it got is received from the training data and not from the Generator.*

- The GANs are formulated as a minimax game, where the Discriminator is trying to minimize its reward V(D, G) and the Generator is trying to minimize the Discriminator's reward or in other words, maximize its loss. It can be mathematically described by the formula below :

$$\max_{D} V(D, G)$$

$$V(D, G) = E_{x \sim pdata(x)} [\log D(x)] + E_{z \sim pz(z)} [\log (1 - D(G(z)))]$$

where,

G = Generator

D = Discriminator

Pdata(x) = distribution of real data

P(z) = distribution of generator

x = sample from Pdata(x)

z = sample from P(z)

D(x) = Discriminator network

G(z) = Generator network

5.4.3 Training a GAN

- So, basically, training a GAN has two parts:

Part 1 : The Discriminator is trained while the Generator is idle. In this phase, the network is only forward propagated and no back-propagation is done. The Discriminator is trained on real data for n epochs, and see if it can correctly predict them as real. Also, in this phase, the Discriminator is also trained on the fake generated data from the Generator and see if it can correctly predict them as fake.

Part 2 : The Generator is trained while the Discriminator is idle. After the Discriminator is trained by the generated fake data of the Generator, we can get its predictions and use the results for training the Generator and get better from the previous state to try and fool the Discriminator.

- The above method is repeated for a few epochs and then manually check the fake data if it seems genuine. If it seems acceptable, then the training is stopped, otherwise, its allowed to continue for few more epochs.

5.5 Discriminator Network

- The discriminator in a GAN is simply a classifier. It tries to distinguish real data from the data created by the generator. It could use any network architecture appropriate to the type of data it's classifying.

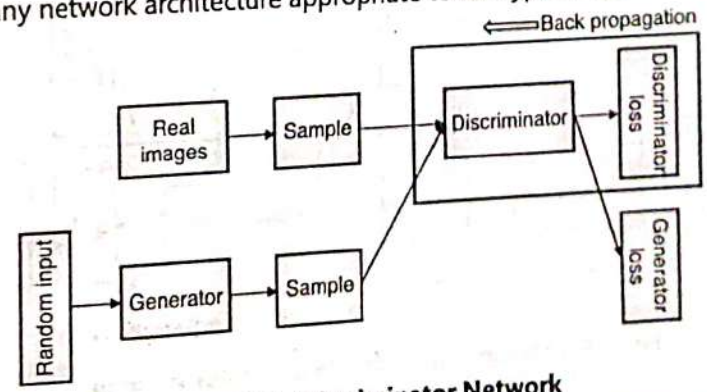


Fig. 5.5.1 : Discriminator Network

Discriminator Training Data

- The discriminator's training data comes from two sources:
  1. **Real data** instances, such as real pictures of people. The discriminator uses these instances as positive examples during training.
  2. **Fake data** instances created by the generator. The discriminator uses these instances as negative examples during training.
- In Fig. 5.5.1, the two "Sample" boxes represent these two data sources feeding into the discriminator. During discriminator training the generator does not train. Its weights remain constant while it produces examples for the discriminator to train on.

Training the discriminator

The discriminator connects to two loss functions. During discriminator training, the discriminator ignores the generator loss and just uses the discriminator loss. We use the generator loss during generator training, as described in the next section.

During discriminator training

1. The discriminator classifies both real data and fake data from the generator.
2. The discriminator loss penalizes the discriminator for misclassifying a real instance as fake or a fake instance as real.
3. The discriminator updates its weights through backpropagation from the discriminator loss through the discriminator network.

5.6 Generator Network

- The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to make the discriminator classify its output as real.
- Generator training requires tighter integration between the generator and the discriminator than discriminator training requires. The portion of the GAN that trains the generator includes:
- random input
- generator network, which transforms the random input into a data instance
- discriminator network, which classifies the generated data
- discriminator output
- generator loss, which penalizes the generator for failing to fool the discriminator

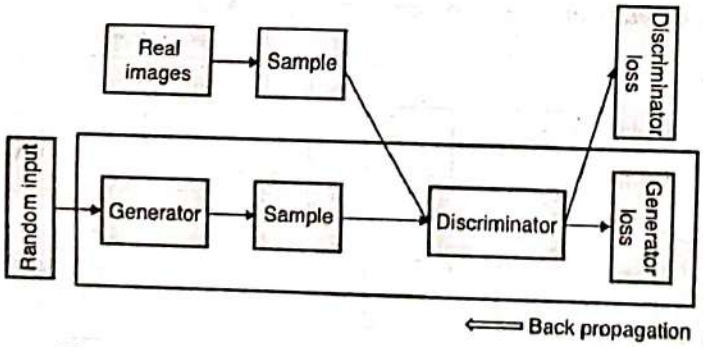


Fig. 5.6.1 : Backpropagation in generator training

Using the Discriminator to Train the Generator

- To train a neural net, we alter the net's weights to reduce the error or loss of its output. In our GAN, however, the generator is not directly connected to the loss that we're trying to affect.
- The generator feeds into the discriminator net, and the discriminator produces the output we're trying to affect. The generator loss penalizes the generator for producing a sample that the discriminator network classifies as fake.
- This extra chunk of network must be included in backpropagation. Backpropagation adjusts each weight in the right direction by calculating the weight's impact on the output how the output would change if you changed the weight. But the impact of a generator weight depends on the impact of the discriminator weights it feeds into. So backpropagation starts at the output and flows back through the discriminator into the generator.
- At the same time, we don't want the discriminator to change during generator training. Trying to hit a moving target would make a hard problem even harder for the generator.
- So we train the generator with the following procedure :
  - Sample random noise.
  - Produce generator output from sampled random noise.
  - Get discriminator "Real" or "Fake" classification for generator output.
  - Calculate loss from discriminator classification.
  - Backpropagate through both the discriminator and generator to obtain gradients.
  - Use gradients to change only the generator weights.
- This is one iteration of generator training. In the next section we'll see how to juggle the training of both the generator and the discriminator.

5.7 Types of GAN

1. **Vanilla GAN :** This is the simplest type GAN. Here, the Generator and the Discriminator are simple multi-layer perceptrons. In vanilla GAN, the algorithm is really simple, it tries to optimize the mathematical equation using stochastic gradient descent. The generator captures the data distribution meanwhile, the discriminator tries to find the probability of the input belonging to a certain class, finally the feedback is sent to both the generator and discriminator after calculating the loss function , and hence the effort to minimize the loss comes into picture.

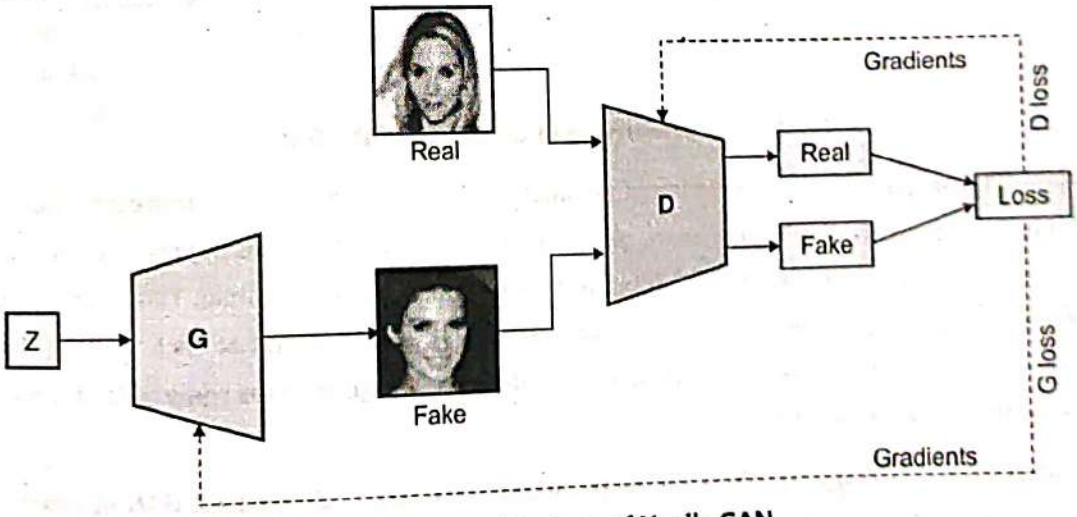


Fig. 5.7.1 : Architecture of Vanila GAN

2. **Conditional GAN (CGAN) :** CGAN can be described as a deep learning method in which some conditional parameters are put into place. In CGAN, an additional parameter 'y' is added to the Generator for generating the corresponding data. Labels are also put into the input to the Discriminator in order for the Discriminator to help distinguish the real data from the fake generated data.

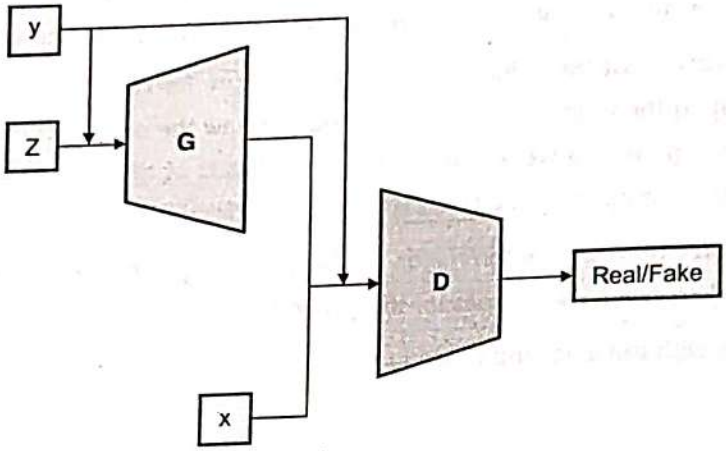


Fig. 5.7.2 : Architecture of Conditional GAN

The loss function of the conditional GAN is as below

$$\min_G \max_D E_{x \sim p_r} \log [D(x \mid y)] + E_{z \sim p_z} \log [1 - D(G(z \mid y))]$$

3. **Deep Convolutional GAN (DCGAN) :** DCGAN is one of the most popular also the most successful implementation of GAN. It is composed of ConvNets in place of multi-layer perceptrons. The ConvNets are implemented without max pooling, which is in fact replaced by convolutional stride. Also, the layers are not fully connected.

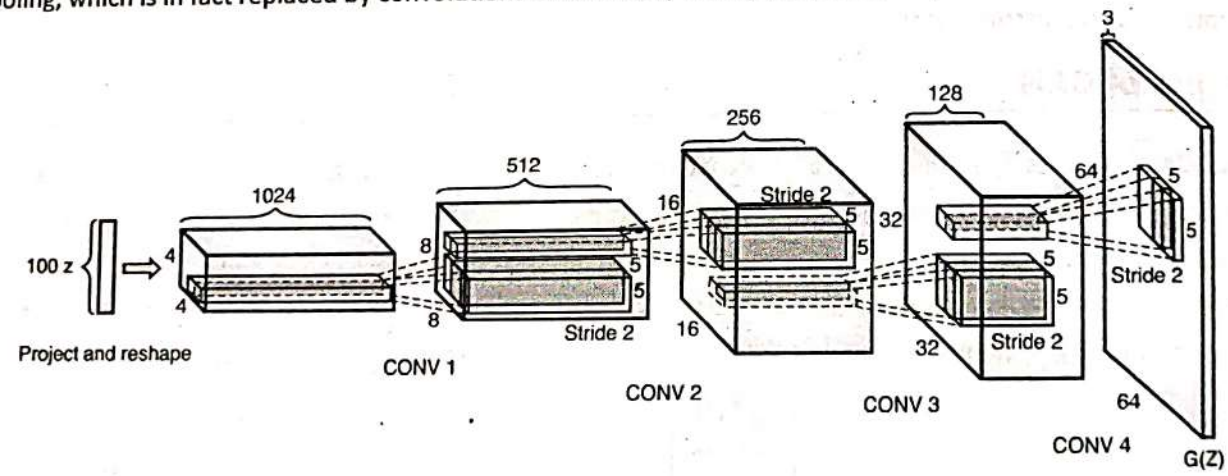


Fig. 5.7.3 : Architecture of generator of the GAN

4. **Laplacian Pyramid GAN (LAPGAN) :** The Laplacian pyramid is a linear invertible image representation consisting of a set of band-pass images, spaced an octave apart, plus a low-frequency residual. This approach uses multiple numbers of Generator and Discriminator networks and different levels of the Laplacian Pyramid. This approach is mainly used because it produces very high-quality images. The image is down-sampled at first at each layer of the pyramid and then it is again up-scaled at each layer in a backward pass where the image acquires some noise from the Conditional GAN at these layers until it reaches its original size.

5. **Super Resolution GAN (SRGAN) :** SRGAN as the name suggests is a way of designing a GAN in which a deep neural network is used along with an adversarial network in order to produce higher resolution images. This type of GAN is

particularly useful in optimally up-scaling native low-resolution images to enhance its details minimizing errors while doing so.

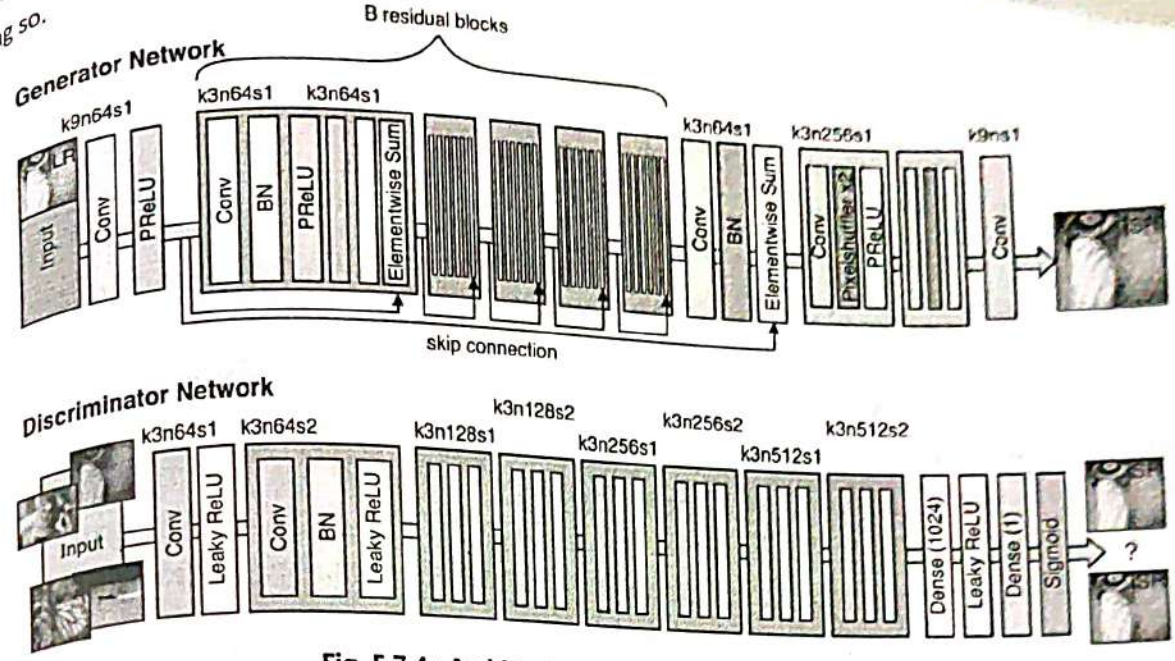


Fig. 5.7.4 : Architecture of Super Resolution GAN

5.8 Applications of GAN Networks

- GANs are a much popular approach in machine learning and have various applications in real-world scenarios. Below are a few most famous applications of generative adversarial networks (GANs), which are as follows:
- Fashion, art, and advertising
- Science
- Video games
- Audio synthesis
- Transfer learning
- Besides these applications, there are so many miscellaneous applications of GANs in machine learning, which are as follows:
- It is used to diagnose partial or total vision loss by detecting glaucomatous images.
- It is used to visualize the interior design, industrial design, shoes, bags, and clothing items by generating photorealistic images.
- It is used to reconstruct 3D models of objects from images and model motion patterns in the video.
- It is used to develop age face photographs that determine individuals' faces according to their age.
- It is used to denoise welding images by removing the random light reflection on the dynamic weld pool surface.
- It is being used in data augmentation.
- It is used to reconstruct individual faces after listening to their voice. It is known as GAN Speech2Face technology.
- It is used to visualize the effects of climate change on particular locations.

- It is used to develop intelligent games and animations by creating anime characters.
- GANs generate text, articles, songs, poems, etc.
- As soon as research on GANs in machine learning is going at its peak, in the future, we will see GAN applications in producing high-quality video, audio, and images also. Further, Microsoft has already collaborated with OpenAI to work on GPT and explore the power of GAN at the next level.

5.8.1 Case Study : GAN for Detection of Real or Fake Images

- It is common knowledge nowadays, that it's hard to tell real media from fake. May it be text, audio, video or images.
- Each type of media has it's own forgery methods. And while faking texts is (still) mostly done in the old fashioned way, faking images and videos have taken a great leap forward. Some of us even feel that we can't tell what's real and what is fake anymore. If two years ago, the photoshop battle sub-reddit was the state of the art for faking images, and photoshop experts were the wizards of this field, new techniques have changed things quite a bit.
- You've probably heard of some cuttingedge forgery methods that seriously threaten our perception of what real and what is fake: deep-fake technique allows planting every face in every video, and different re-enactment techniques allow allowing to move every face as you want: make expressions, talk, etc. And this is only the beginning.

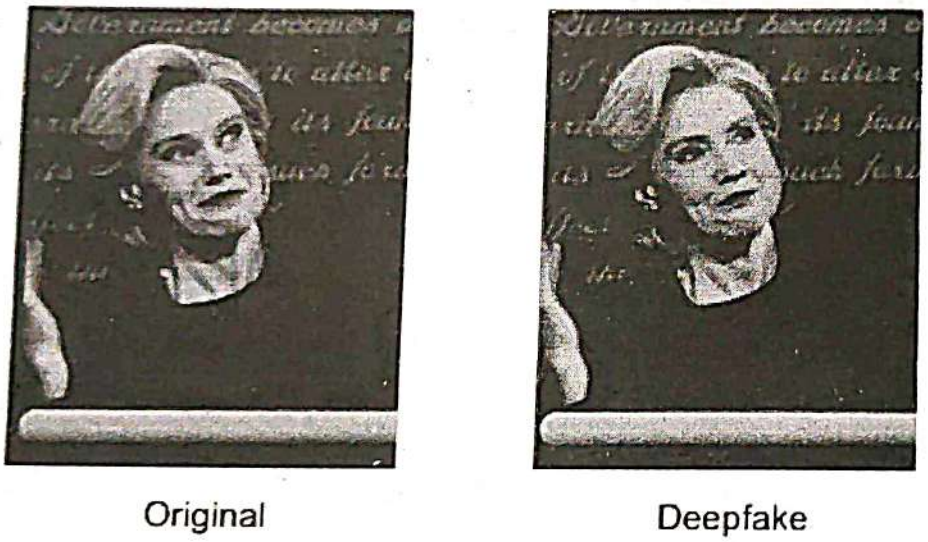


Fig. 5.8.1: Deepfake image

Fig. 5.8.2 : Re-enactment : making a face talk

- What image tampering techniques are there?
- First, let's discuss what are we dealing with.
- Photography forging is almost as old as photography itself. In the pictures below, you can see a tampered photography from the 19th century:

Fig. 5.8.3 : Photograph from 19th Century

- With the emergence of digital photography, image tampering became easier and more common: the most common one is known as "splicing" and "copy-move", or as most people call it "photoshopping".
- These methods include taking a part of one image and plant it into another one. To make it look more realistic, forgers also do some digital retouching. The results of these techniques may be very hard to tell apart with the naked eye.
- However, technology made things much harder (or easier, depends on which side are you). In an ironic twist of fate, deep learning, which became a leading method for classification images, also allowed a new, ground breaking forgery the generative model.

5.8.2 CycleGan

- In 2017, 2 amazing works came out of Alexei Efros laboratory — pix2pix and CycleGan. You can read about them in my previous post.
- Both works allowed "copying" images from one domain to another: converting horses to zebras, dogs to cats and more.

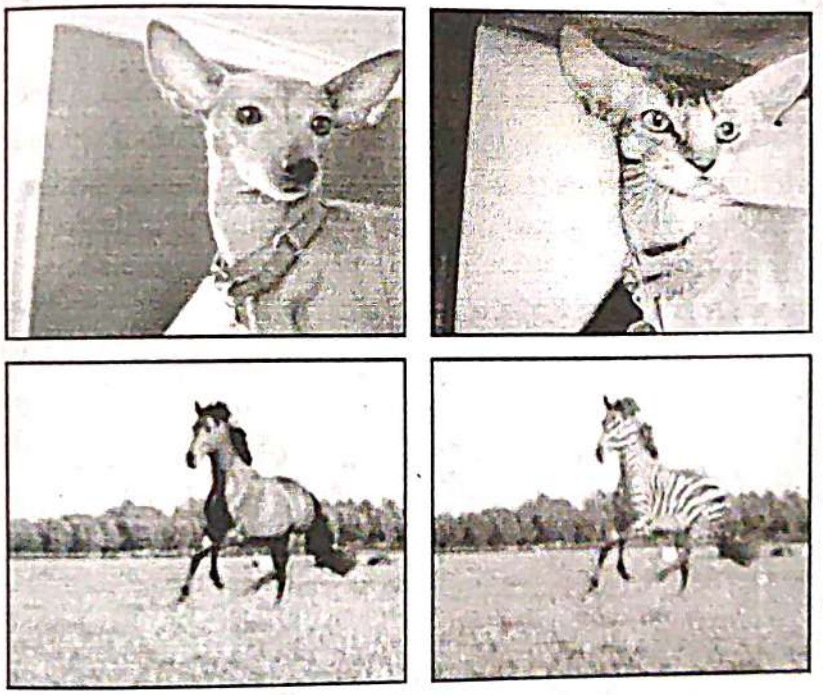


Fig. 5.8.4 : Fake video : re-enactment

Fig. 5.8.5 : Nicholas Cage as Marlon Brando as The Godfather

- Apart from creating fake images, deep learning techniques went further and allowed creating fake videos.
- The real tipping point of the field was (as in many other occasions) not technological, but cultural. Around January 2018, high-quality fake videos started appearing on Reddit. Many of them were planting Nicholas Cage's face on different scenes, but not all were SFW. This has sprung immediate media attention (along with apocalyptic prophecies) to this field.
- But Deep fake is far from being alone : recently there is a surge in such techniques, which become more and more advanced in their realism and ease of training. Starting from the toy "face swap" app, through more serious stuff such as face2face, deep-fake, Synthesizing Obama, to the recent "few shot talking heads" by Samsung.

Fig. 5.8.6 : Deep video expressions

- The currently most impressive work in this field (currently, things are moving fast) is **Deep video portrait.** In this work, researchers use a multi-step approach, to "puppeteer" a full face. They first extract face features such as pose, expression, eye coordinates and more, from source and target videos, then use an encoder-decoder to generate a fake video, allowing control not only in mouth movements and facial expressions, but head movements as well.

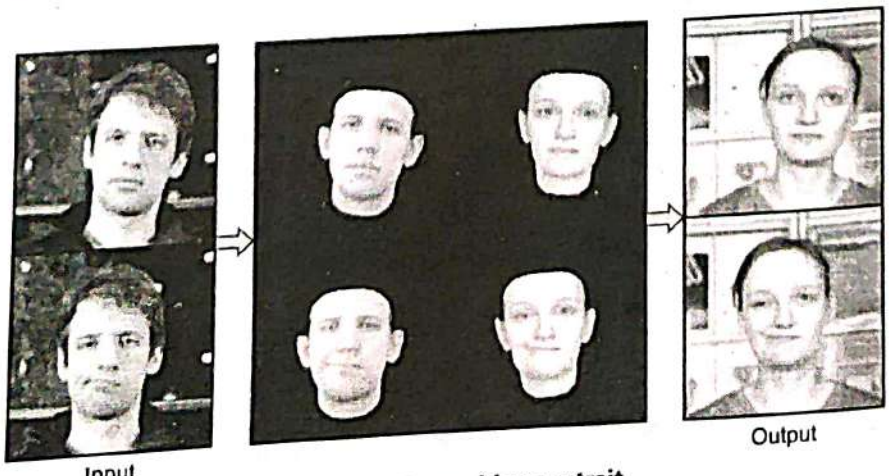


Fig. 5.8.7 : Deep video portrait

5.9 Digital Forensics

Since this post is not about image generation, but about how we detect them, after seeing some forging techniques, we would like to examine what the defensive team has to offer. As briefly discussed earlier, digital forensic methods can be divided into 3 kinds. A good detection operation should use a combination of all:

1. **Feature-based :** where there is a kind of artifact in a certain (or multiple) types of forgeries — mostly applicable for classic methods.
2. **Supervised learning :** using deep learning classifiers (mostly CNN) to learn certain types of fake images — applicable for classical methods as well as generative models, e.g GANs.
3. **Unsupervised/Universal :** an attempt to capture some essence of a genuine image, to detect new kinds of forgeries (that the model hasn't seen before). It can be seen as a kind of anomaly detection.

All the above methods have their advantages and disadvantages, but since generative methods became more realistic, 2 and 3 became more prominent.

The tampering techniques we've witnessed in the previous part are all (or most) publicly available with courtesy of the deep learning community. However, it doesn't have to stay like this forever: it is very reasonable that in the following years, different companies and regimes will have their own secret techniques.

5.9.1 Feature-based

In 2004, **Hany Farid** and **Alin Popescu** have published the first work about identifying fake images, using digital artifacts. Digital cameras have different artifacts stemming either from the photographing hardware, software or from the compression techniques, which are image specific.

Therefore, it was only a matter of time until a digital method would emerge, to find out these fake photos. Farid and Popescu have used a specific camera filter (CFA) to identify fake parts of the images.

Since then, many more handcrafted techniques have emerged:

1. **Using JPEG "signature"**

JPEG is the most common image compression protocol in digital media. You can read about its specifics here. In short, every image has its own encoding which optimizes the file size. it is possible to exploit the difference in the encoding of different images to detect forged images (splice). Here is an example of such work.

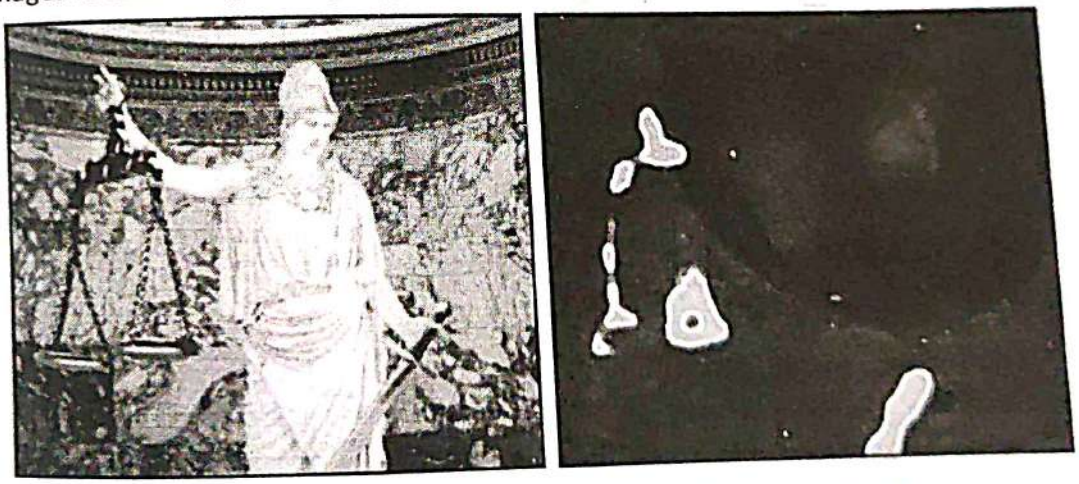


Fig. 5.9.1 : Splicing artifacts caused by JPEG compression

2. Camera artifacts

As mentioned, digital cameras also have their artifacts, structures in the hardware or software. Each camera manufacturer, model or software version, may have its own signature. comparing such features in different parts of an image or camera noise may result in a good classifier.

3. Photography artifacts

With the same logic as above, tampering an image may distort the natural conditions of the photograph. Using a numeric measure of such a feature (e.g the lighting, "aberration") may also be useful in forgery detection.

4. Generative model artifacts

Current generative models also suffer from known artifacts, some even visible, such different asymmetries, strange shape of the mouth, asymmetric eyes and more. These artifacts are exploited in this work, using classic computer vision to separate the fake images from real ones. However, considering the progress of generative models, these artifacts won't necessarily exist in tomorrows forgeries.

Fig. 5.9.2 : Fake Eyes

5. Asymmetric eyes in GAN face

All the above methods are great and clever, however once, exposed they are penetrable by forgers. This is where machine learning comes in: being somewhat of a black box, it can learn fake images without its specifics being known to forgers.

5.9.2 Supervised Deep Learning

- With the rise of deep learning, it was only natural that researchers will start using deep networks to detect fake images.
- Intuitively, it is easy to take images from different types of faking classes and start training classifiers and detectors. Let's examine some high profile works.

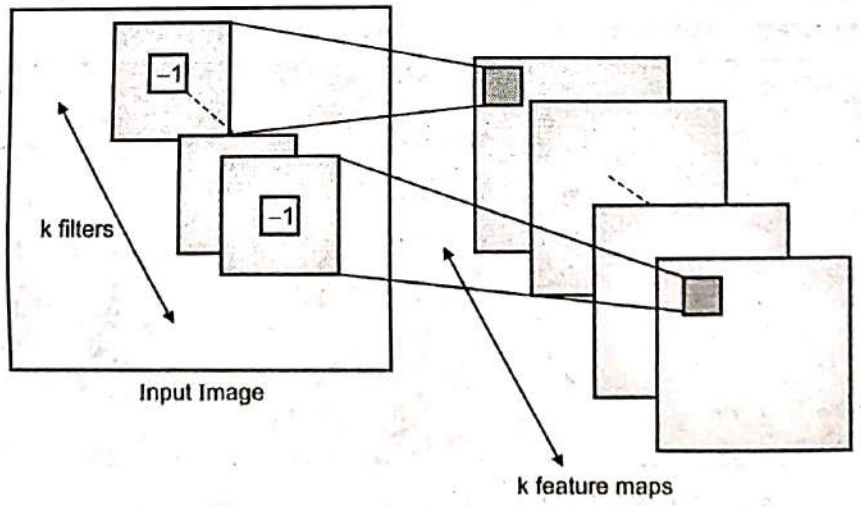


Fig. 5.9.3 : Universal Image Manipulation Detection Using a New conv layer

- In this work, researchers design a special convolutional layer, which by putting a constraint on the filters, is intended to capture *manipulations* instead of image *semantic content*.
- Tested on different retouching methods such as median filtering, Gaussian blurring and more, the method reached >95% accuracy. This work is intended to be universal, however, it's designed allegedly limits it to photoshopping and tampering, not to GANs and the like.

Mesonet

- Mesonet is a work focused on perhaps the most painful problems: tampering of faces in videos.
- Specifically, face2face and deep fakes (see above). Since video (specifically digitized) in its essence is a sequence of images, researchers address this task using a deep network and reach good results, using pretty standard networks.
- All in all, the work is not very special apart from being one of the first addressing this task.

5.9.3 GAN Experiment

- As we've seen earlier, the GANs just keep coming. It is not very hard to train a model telling apart a real image from a specific GAN. But it is also impractical to train a model for *each* GAN.
- Some researchers were optimistic enough to try and generalize a classification model to different GANs from those they were trained on. In other words, they trained a deep network on one type of GAN (PG-GAN — the predecessor of StyleGAN) and tried to infer on another one (DC-GAN, WGAN).
- The results, as expected, showed that there is no generalization whatsoever, even with the preprocessing the researchers have applied.
- There are many more similar methods, but to generalize meaning, identifying forgeries never seen before, learning research needs to start becoming more creative.

5.9.4 Universal Methods

- As we know, deep learning is not only limited to naive classifiers. There are many kinds of un/self/semi-supervised models that can handle small amounts of data, n shots, and other tasks.
- Let's see what kind of ideas are used to address the problem of "universal fake images".

Self-consistency

A good example of such a method can be found in the work **Fighting Fake News: Image Splice Detection via Learned Self-Consistency**. This work has shared some elements with his earlier works. The researchers incorporate a 4 step workflow where they:

1. Learn to predict the EXIF* metadata of an image.
2. Slice an image into many patches, comparing predicted EXIF values of each pair.
3. Slices that seem to have un-matching EXIF values will be classified as taken from different images, therefore the image will be fake.
4. Classifications will be used to provide a heat-map of the transplanted regions.

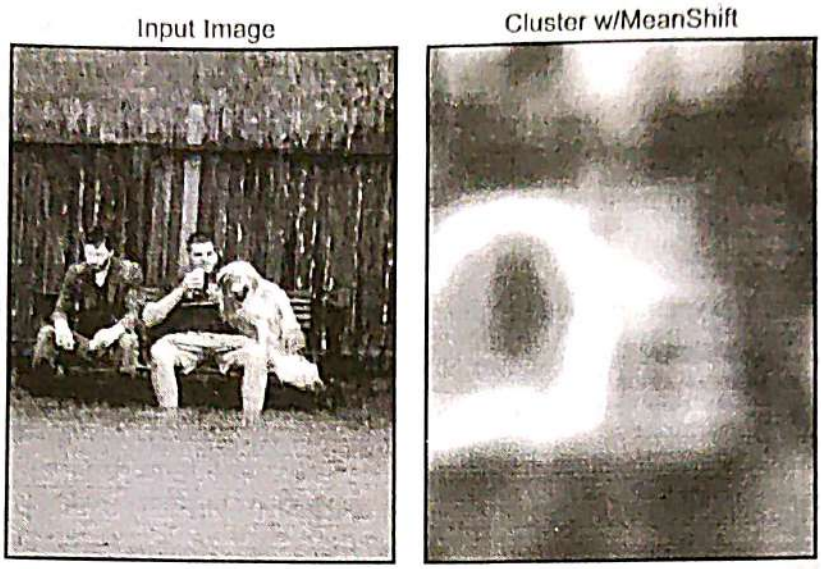


Fig. 5.9.4 A : detection of "spliced" Keanu Reeves

- But what's an EXIF? in digital media, images (and some sound files as well), EXIF, **Exchangeable image file format**, is a kind of a metadata signature of the file. An EXIF of an image should include a camera (or scanner) model, original image size, photo attributes (flash, shutter opening time) and more.

```
EXIF CameraMake: NIKON CORPORATION
EXIF CameraModel: NIKON D5300
EXIF ColorSpace: sRGB
EXIF DateTimeOriginal: 2016:09:13 16:58:26
EXIF ExifImageLength: 3947
EXIF ExifImageWidth: 5921
EXIF Flash: No
EXIF FocalLength: 31.0mm
EXIF WhiteBalance: Auto
EXIF CompressedBitsPerPixel: 2
```

- Clearly, not all online photos have an intact EXIF, especially not the fraudulent ones. That's exactly the reason the researchers engaged in predicting the EXIF for each image/patch.
- Now you can see that this task is somehow unsupervised since there are a plethora of online images with readily available EXIF to learn from. To be more exact, 400K images were used to train this model.
- This model reached good results on photoshopped images, but surprisingly, it has also some success on GaN generated images.

5.9.5 Forensic Transfer

- **Luisa** Verdoliva is an Italian researcher that along with her team took some interesting shots at generalizing image forensics.
- In this work, they train a bit different model, that will hopefully be more generalizable. What they did is using an autoencoder, which is a network that is intended to "shrink" an image into a vector, and then to reconstruct it. This vector was trained to be the **classifier** to determine whether the image is real or fake:

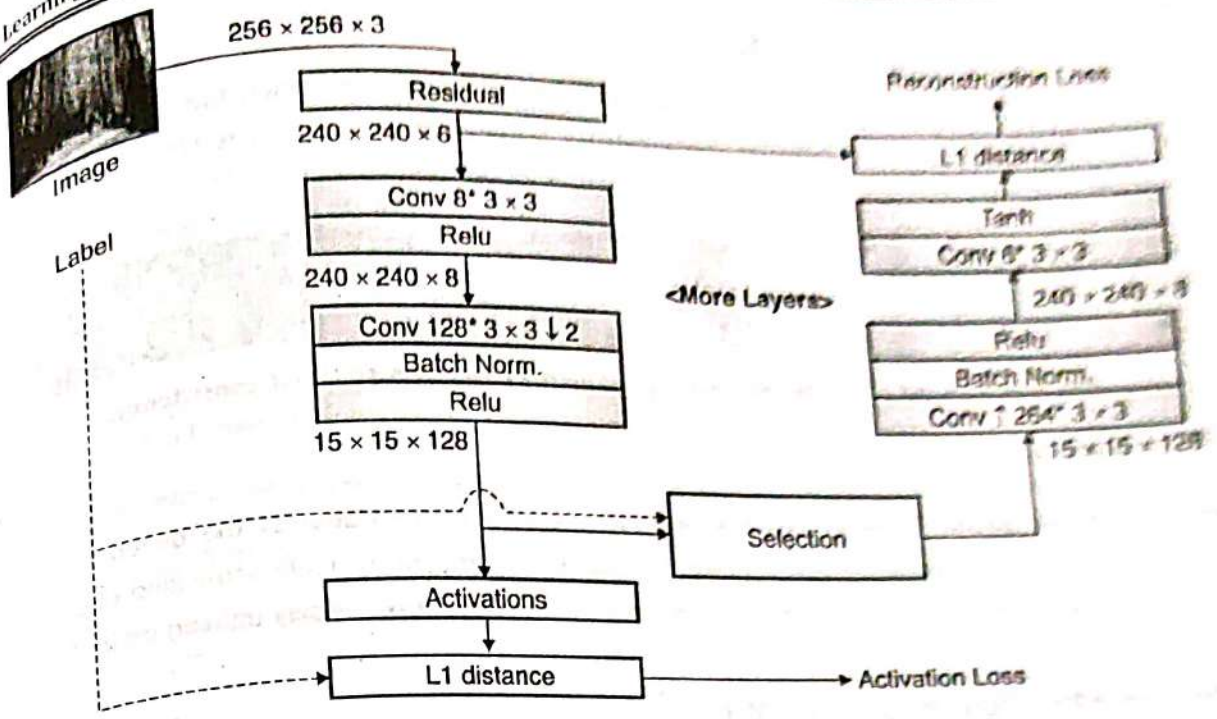


Fig. 5.9.5 : Classifier using GAN

They also experiment with transfer learning : train their network on dataset A, re-train it with a small subset of dataset B and try to infer on dataset B.

They do this task on a few data-sets, back and forth, and reach reasonable results (75–85% accuracy). These results are better than other networks (some of are discussed above in **supervised learning** part)

	Pro.GAN[22] vs CycleGAN[45]	
	Source	Target
Bayer16[7]	99.92	50.42
Cozzolino17[11]	99.92	52.43
Rahmouni17[32]	100.0	49.87
MesoInception-4[3]	97.47	44.19
XceptionNet[10]	100.0	58.79
FT(ours)	100.0	51.48
FT-Res (ours)	100.0	85.00

Fig. 5.9.6 : Forensic Transfer : An example transfer learning results

5.10 Noise print

- Another unsupervised approach from the above team, similarly to self-consistency, tries to predict PRNU noise (a specific type of camera noise) between image patches. it reports state of the art results on multiple data sets (0.4 average Matthews correlation metric in compare to 0.33 for self-consistency).

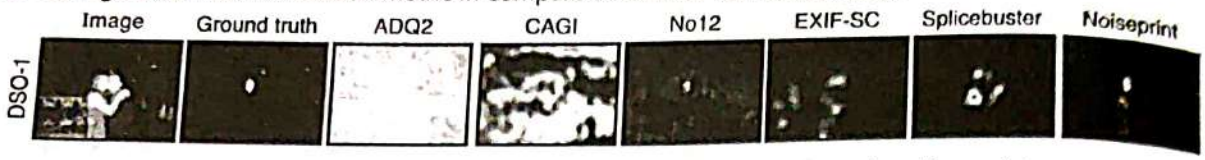


Fig. 5.10.1 : A set of predictions: noise print on the right. EXIF-SC is self-consistency.

5.10.1 Deep Fake Spin

- Considering all the above, it seems that the universal methods have to address the generative fakes more aggressively. And they do: some researchers take the chance and attempt to create some kind of a general GAN hunter. Which means be able to identify a GAN-generated image without specifically training on it's kind. Let's see a few of them:

Learning to Detect Fake Face Images in the Wild

In a somewhat hasty researchers use uncoupled pairs of images to train a deep network to classify same/different images (real and real, fake and fake, real and fake) The somehow naive result reaches reasonable results on different GANs, although training took place on all of them.

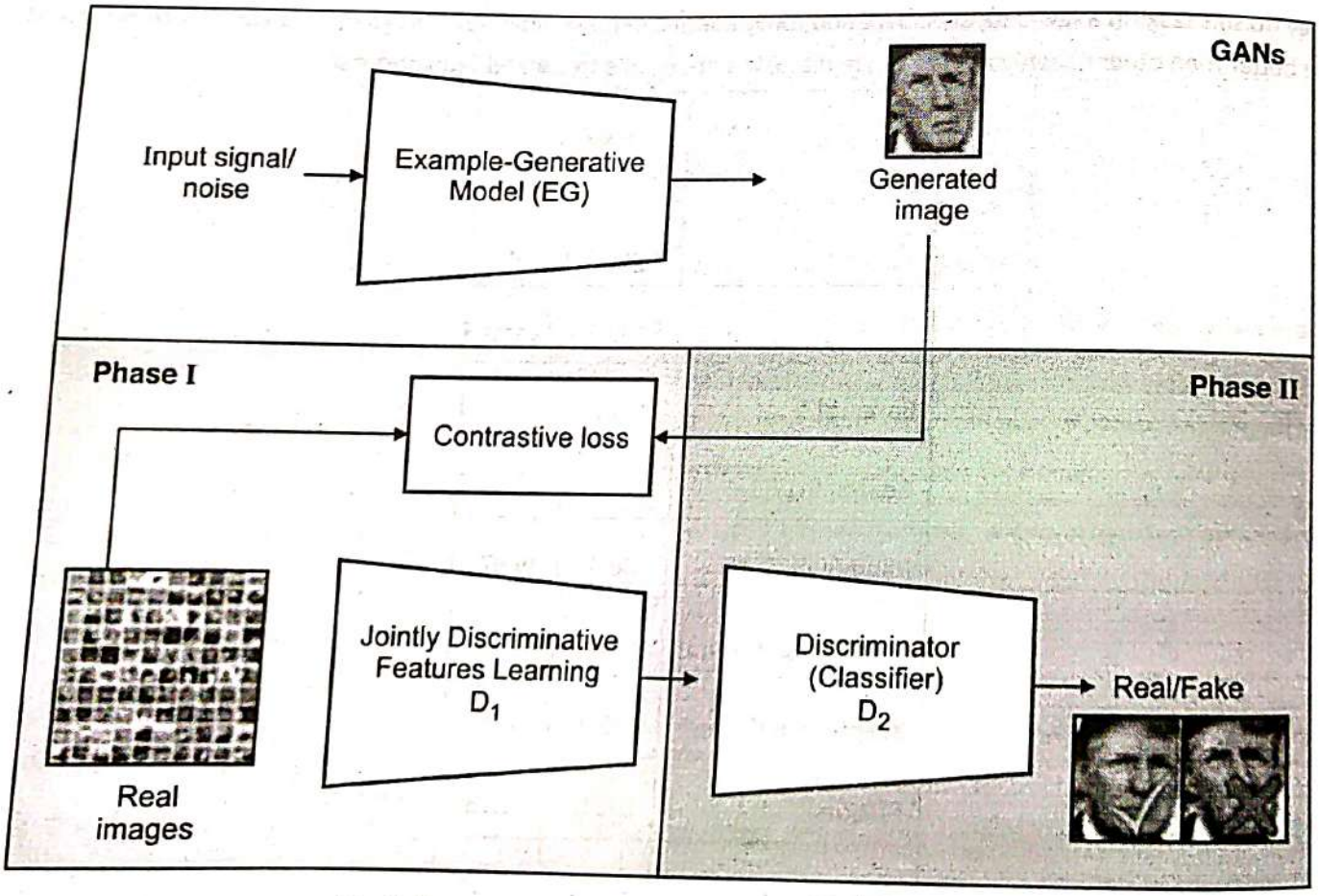


Fig. 5.10.2 : Detection of Fake Face images using GAN

Review Questions

5.1 What is the role of the generator and discriminator?

5.2 Explain the difference between the discriminative and generative models.

5.3 Can we control and modify the images generated by GAN? If yes then how?

5.4 Explain Boltzmann Machine in details.

5.5 Explain GAN architecture with an example.

5.6 How to train the Generative adversarial network.

5.7 Explain the types of Generative adversarial network.

5.8 What are the applications of Generative adversarial network.

5.9 Do GANs find real or Fake Images?

□□□

6 Reinforcement Learning

Unit VI

Syllabus

Introduction of deep reinforcement learning, Markov Decision Process, basic framework of reinforcement learning, challenges of reinforcement learning, Dynamic programming algorithms for reinforcement learning, Q Learning and Deep Q-Networks, Deep Q recurrent networks, Simple reinforcement learning for Tic-Tac-Toe.

6.1 Introduction of Deep Reinforcement Learning

- Human beings do not learn from a concrete notion of training data. Learning in humans is a continuous experience-driven process in which decisions are made, and the reward/punishment received from the environment are used to guide the learning process for future decisions.
- In other words, learning in intelligent beings is by reward-guided trial and error. Furthermore, much of human intelligence and instinct is encoded in genetics, which has evolved over millions of years with another environment-driven process, referred to as evolution.
- Therefore, almost all of biological intelligence, as we know it, originates in one form or other through an interactive process of trial and error with the environment. In his interesting book on artificial intelligence [453], Herbert Simon proposed the ant hypothesis:
- "Human beings, viewed as behaving systems, are quite simple. The apparent complexity of our behaviour over time is largely a reflection of the complexity of the environment in which we find ourselves."
- Human beings are considered simple because they are one-dimensional, selfish, and reward driven entities (when viewed as a whole), and all of biological intelligence is therefore attributable to this simple fact. Since the goal of artificial intelligence is to simulate biological intelligence, it is therefore natural to draw inspirations from the successes of biological greed in simplifying the design of highly complex learning algorithms.
- A reward-driven trial-and-error process, in which a system learns to interact with a complex environment to achieve rewarding outcomes, is referred to in machine learning parlance as reinforcement learning. In reinforcement learning, the process of trial and error is driven by the need to maximize the expected rewards over time.
- Reinforcement learning can be a gateway to the quest for creating truly intelligent agents such as game-playing algorithms, self-driving cars, and even intelligent robots that interact with the environment.
- Simply speaking, it is a gateway to general forms of artificial intelligence. We are not quite there yet. However, we have made huge strides in recent years with exciting results:

- Deep learners have been trained to play video games by using only the raw pixels of the video console as feedback. A classical example of this setting is the Atari 2600 console, which is a platform supporting multiple games.
- The input to the deep learner from the Atari platform is the display of pixels from the current state of the game. The reinforcement learning algorithm predicts the actions based on the display and inputs them into the Atari console.
- Initially, the computer algorithm makes many mistakes, which are reflected in the virtual rewards given by the console. As the learner gains experience from its mistakes, it makes better decisions. This is exactly how humans learn to play video games.
- Video games are excellent test beds for reinforcement learning algorithms, because they can be viewed as highly simplified representations of the choices one has to make in various decision-centric settings. Simply speaking, video games represent toy microcosms of real life.
- Deep Mind has trained a deep learning algorithm Alpha Go to play the game of Go by using the reward-outcomes in the moves of games drawn from both human and computer self-play.
- Go is a complex game that requires significant human intuition, and the large tree of possibilities (compared to other games like chess) makes it an incredibly difficult candidate for building a game-playing algorithm.
- Alpha Go has not only convincingly defeated all top-ranked Go players it has played against, but has contributed to innovations in the style of human play by using unconventional strategies in defeating these players.
- These innovations were a result of the reward-driven experience gained by Alpha Go by playing itself over time. Recently, the approach has also been generalized to chess, and it has convincingly defeated one of the top conventional engines.
- In recent years, deep reinforcement learning has been harnessed in self-driving cars by using the feedback from various sensors around the car to make decisions. Although it is more common to use supervised learning (or imitation learning) in self-driving cars, the option of using reinforcement learning has always been recognized as a viable possibility. During the course of driving, these cars now consistently make fewer errors than do human beings.
- The quest for creating self-learning robots is a task in reinforcement learning. For example, robot locomotion turns out to be surprisingly difficult in nimble configurations. Teaching a robot to walk can be couched as a reinforcement learning task, if we do not show a robot what walking looks like.
- In the reinforcement learning paradigm, we only incentivize the robot to get from point A to point B as efficiently as possible using its available limbs and motors. Through reward-guided trial and error, robots learn to roll, crawl, and eventually walk.
- Reinforcement learning is appropriate for tasks that are simple to evaluate but hard to specify. For example, it is easy to evaluate a player's performance at the end of a complex game like chess, but it is hard to specify the precise action in every situation.
- As in biological organisms, reinforcement learning provides a path to the simplification of learning complex behaviours by only defining the reward and letting the algorithm learn reward-maximizing behaviours.

- The complexity of these behaviours is automatically inherited from that of the environment. Reinforcement learning systems are inherently end-to-end systems in which a complex task is not broken up into smaller components, but viewed through the lens of a simple reward.
- The simplest example of a reinforcement learning setting is the multi-armed bandit problem, which addresses the problem of a gambler choosing one of many slot machines in order to maximize his payoff. The gambler suspects that the (expected) rewards from the various slot machines are not the same, and therefore it makes sense to play the machine with the expected reward.
- Since the expected payoffs of the slot machines are not known in advance, the gambler has to explore different slot machines by playing them and also exploit the learned knowledge to maximize the reward. Although exploration of a particular slot machine might gain some additional knowledge about its payoff, it incurs the risk of the (potentially fruitless) cost of playing it.
- Multi-armed bandit algorithms provide carefully crafted strategies to optimize the trade-off between exploration and exploitation. However, in this simplified setting, each decision of choosing a slot machine is identical to the previous one.
- This is not quite the case in settings such as video games and self-driving cars with raw sensory inputs (e.g., video game screen or traffic conditions), which define the state of the system. Deep learners are excellent at distilling these sensory inputs into state-sensitive actions by wrapping their learning process within the exploration / exploitation framework.

6.2 Markov Decision Process

- Markov Decision Process or MDP, is used to **formalize the reinforcement learning problems**. If the environment is completely observable, then its dynamic can be modelled as a **Markov Process**.
- In MDP, the agent constantly interacts with the environment and performs actions; at each action, the environment responds and generates a new state.

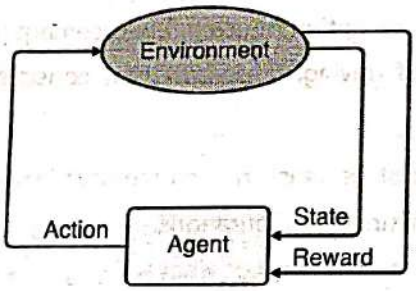


Fig. 6.2.1 : Markov Decision Process

- MDP is used to describe the environment for the RL, and almost all the RL problem can be formalized using MDP.
- MDP contains a tuple of four elements (S, A, P_a, R_a):
- A set of finite States S
- A set of finite Actions A
- Rewards received after transitioning from state S to state S', due to action a.
- Probability P_a.
- MDP uses **Markov property**, and to better understand the MDP, we need to learn about it.

6.2.1 Markov Property

- It says that "If the agent is present in the current state S1, performs an action a1 and move to the state s2, then the state transition from s1 to s2 only depends on the current state and future action and states do not depend on past actions, rewards, or states".
- In other words, as per Markov Property, the current state transition does not depend on any past action or state. Hence, MDP is an RL problem that satisfies the Markov property. Such as in a **Chess game, the players only focus on the current state and do not need to remember past actions or states**.

Finite MDP

A finite MDP is when there are finite states, finite rewards, and finite actions. In RL, we consider only the finite MDP.

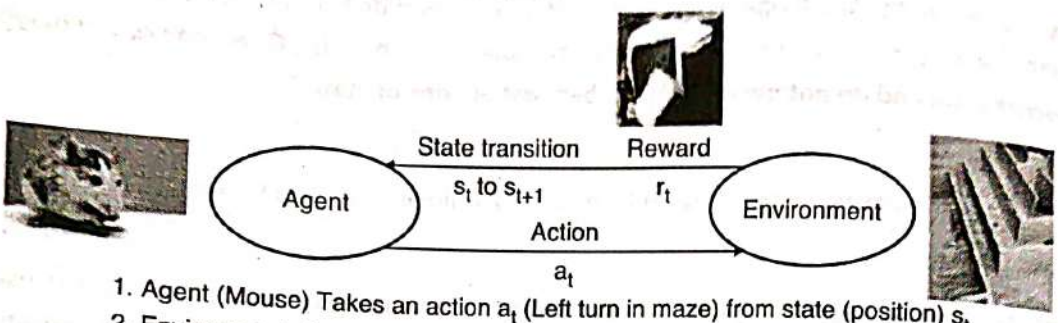
Markov Process

Markov Process is a memory less process with a sequence of random states $S_1, S_2, ..., S_t$ that uses the Markov Property. Markov process is also known as Markov chain, which is a tuple (S, P) on state S and transition function P. These two components (S and P) can define the dynamics of the system.

6.3 The Basic Framework of Reinforcement Learning

- The bandit algorithms of the previous section are stateless. In other words, the decision made at each time stamp has an identical environment, and the actions in the past only affect the knowledge of the agent (not the environment itself).
- This is not the case in generic reinforcement learning settings like video games or self-driving cars, which have a notion of state.
- In generic reinforcement learning settings, each action is associated with a reward in isolation. While playing a video game, you do not get a reward only because you made a particular move.
- The reward of a move depends on all the other moves you made in the past, which are incorporated in the state of the environment. In a video game or self-driving car, we would need a different way of performing the credit assignment in a particular system state.
- For example, in a self-driving car, the reward for violently swerving a car in a normal state would be different from that of performing the same action in a state that indicates the danger of a collision. In other words, we need a way to quantify the reward of each action in a way that is specific to a particular system state.
- In reinforcement learning, we have an agent that interacts with the environment with the use of actions. For example, the player is the agent in a video game, and moving the joystick in a certain direction in a video game is an action.
- The environment is the entire set up of the video game itself. These actions change the environment and lead to a new state. In a video game, the state represents all the variables describing the current position of the player at a particular point. The environment gives the agent rewards, depending on how well the goals of the learning application are being met.
- For example, scoring points in a video game is a reward. Note that the rewards may sometimes not be directly associated with a particular action, but with a combination of actions taken some time back.

- For example, the player might have cleverly positioned a cursor at a particularly convenient point a few moves back, and actions since then might have had no bearing on the reward. Furthermore, the reward for an action might itself not be deterministic in a particular state (e.g., pulling the lever of a slot machine).
- One of the primary goals of reinforcement learning is to identify the inherent values of actions in different states, irrespective of the timing and stochasticity of the reward.



1. Agent (Mouse) Takes an action a_t (Left turn in maze) from state (position) s_t
2. Environment gives mouse reward r_t (Cheese / No cheese
3. The state of agent is changed To s_{t+1}
4. Mouse's neurons update synaptic weights based on whether action earn rewards

Fig. 6.3.1 : The broad framework of reinforcement learning

- The learning process helps the agent choose actions based on the inherent values of the actions in different states. This general principle applies to all forms of reinforcement learning in biological organisms, such as a mouse learning a path through a maze to earn a reward.
- The rewards earned by the mouse depend on an entire sequence of actions, rather than on only the latest action. When a reward is earned, the synaptic weights in the mouse's brain adjust to reflect how sensory inputs should be used to decide future actions in the maze.
- This is exactly the approach used in deep reinforcement learning, where a neural network is used to predict actions from sensory inputs (e.g., pixels of video game). This relationship between the agent and the environment is shown in Fig. 6.3.1.
- The entire set of states and actions and rules for transitioning from one state to another is referred to as a *Markov decision process.*
- The main property of a Markov decision process is that the state at any particular time stamp encodes all the information needed by the environment to make state transitions and assign rewards based on agent actions.
- Finite Markov decision processes (e.g., tic-tac-toe) terminate in a finite number of steps, which is referred to as an *episode.* A particular episode of this process is a finite sequence of actions, states, and rewards.
- An example of length (n + 1) is the following:

  s0a0r0s1a1r1 . . . start. . . snanrn

- Note that *st* is the state *before* performing action *at*, and performing the action *at* causes are ward of *rt* and transition to state *st*+1.The outputs *rt*+1 in response to action *at* in state *st*(which slightly changes the subscripts in all the results). Infinite Markov decision processes (e.g., continuously working robots) do not have finite length episodes and are referred to as *non-episodic.*

Examples

Although a system state refers to a complete description of the environment, many practical approximations are often made. For example, in an Atari video game, the system state might be defined by a fixed-length window of game snapshots. Some examples are as follows:

1. **Game of tic-tac-toe, chess, or Go :** The state is the position of the board at any point, and the actions correspond to the moves made by the agent. The reward is +1, 0, or−1 (depending on win, draw, or loss), which is received at the end of the game. Note that rewards are often not received immediately after strategically astute actions.
2. **Robot locomotion :** The state corresponds to the current configuration of robot joints and its position. The actions correspond to the torques applied to robot joints. There ward at each time stamp is a function of whether the robot stays upright and the amount of forward movement from point A to point B.
3. **Self-driving car :** The states correspond to the sensor inputs from the car, and the actions correspond to the steering, acceleration, and braking choices. The reward is a hand-crafted function of car progress and safety. Some effort usually needs to be invested in defining the state representations and corresponding rewards. However, once these choices have been made, reinforcement learning frameworks are end-to-end systems.

6.4 Challenges of Reinforcement Learning

Reinforcement learning is more difficult than traditional forms of supervised learning for the following reasons :

1. When a reward is received (e.g., winning a game of chess), it is not exactly known how much each action has contributed to that reward. This problem lies at the heart of reinforcement learning and is referred to as the credit-assignment problem. Furthermore, rewards may be probabilistic (e.g., pulling the lever of a slot machine), which can only be estimated approximately in a data-driven manner.
2. The reinforcement learning system might have a very large number of states (such as the number of possible positions in a board game), and must be able to make sensible decisions in states it has not seen before. This task of model generalization is the primary function of deep learning.
3. A specific choice of action affects the collected data in regard to future actions. As in multi-armed bandits, there is a natural trade-off between exploration and exploitation. If actions are taken only to learn their reward, then it incurs a cost to the player. On the other hand, sticking to known actions might result in suboptimal decisions.
4. Reinforcement learning merges the notion of data collection with learning. Realistic simulations of large physical systems such as robots and self-driving cars are limited by the need to physically perform these tasks and gather responses to actions in the presence of the practical dangers of failures. In many cases, the early portion of learning in a task may have few successes and many failures. The inability to gather sufficient data in real settings beyond simulated and game-centric environments is arguably the single largest challenge to reinforcement learning.

6.5 Dynamic Programming Algorithms for Reinforcement Learning

- Dynamic programming is an optimization method for sequential problems. DP algorithms are able to solve complex 'planning' problems. Given a complete MDP, dynamic programming can find an optimal policy.
- This is achieved with two principles:
  1. Breaking down the problem into subproblems
  2. Caching and reusing optimal solutions to subproblems to find the overall optimal solution

- In reinforcement learning, we want to use dynamic programming to solve MDPs. So given an MDP hS, A, P, R, γi and a policy π:
  - First, we want to find the value function vπ for that policy:
  - This is done by policy evaluation (the prediction problem)
- Then, when we're able to evaluate the policy, we want find the best policy v* (the control problem). This is done with two strategies:
  1. Policy iteration
  2. Value iteration

6.5.1 Grid World Game

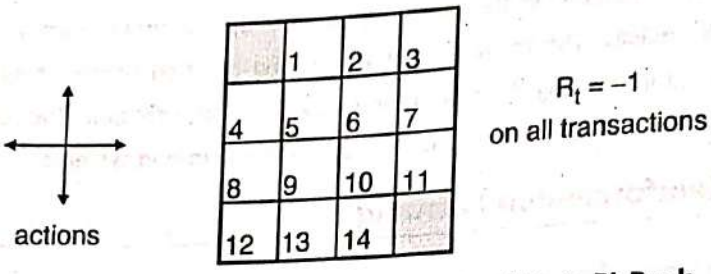


Fig. 6.5.1 : Simple Grid World via Sutton and Barto RL Book

- In this problem, we are given a grid (4 x 4 in this case). The goal is to reach either the top-left or the bottom-right square (gray coloured) from any other square on the grid, with maximum reward.
- You can jump one square in either of the North, South, East, or West directions from any given square.
- Each jump (in any direction) earns a reward of -1 except from the gray squares where the reward is 0.
- Jumping off the grid earns you -1, but you stay in the same square.
- Once you reach the goal, the game is over, and you cannot jump anywhere (i.e., even on jumping, you stay in the same square with a 0 reward).
- To approach this problem, we consider our grid to be an MDP, where each square is a state. The state transition reward for all the states is -1 except for the terminal states where the reward is 0. Our goal is to find an optimal policy for this MDP.

6.5.2 Policy Iteration

The algorithm is pretty straight forward:

Given a policy π,

1. Evaluate the policy
2. Improve the policy

6.5.2(A) Policy Evaluation

- To improve a policy, we first need to evaluate it. So in this section, we will use **Bellman's Expectation Equation iteratively** to evaluate the **value function** for all the states in an MDP for a given policy.

In our grid-world problem, we will start with a random policy initially, i.e., $\pi(\text{North}|.) = \pi(\text{South}|.) = \pi(\text{East}|.) = \pi(\text{West}|.) = 0.25$ (. means any state). To evaluate the policy:

At each iteration (k + 1): For all states $s \in S$ Update $v_{(k+1)(s)}$ using Bellman's Equation as follows :

$$v_{k+1}(s) = \sum_{a \in A} \pi(a \mid s) \left(R_s^a + \gamma \sum_{s' \in s} P_{ss'}^a v_k(s') \right) \quad \text{....(6.5.1)}$$

Initially, we assume that all the states yield nothing, i.e., we initialize the state value for all states with 0. Let's start:

k = 0

0.0	(0.0)	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

State Values at k = 0

Now, we will iteratively update these values with a **one-state lookup** for each state. By one state lookup, I mean just considering the state value of the immediate next state with a discount (γ) of 1.

For understanding, **we will calculate the state value for just one state (circled in red).**

With respect to the Bellman's equation, we have :

$$V_k(S') = 0$$

$$\pi(a \mid s) = 0.25$$

$$R_s = R_t = -1$$

$$\gamma = 1$$

$$P_{ss'} = 1$$

Values of Variables in the Bellman Equation at k = 1

Note that $v_k(s')$ is 0 for all the north, south, east and west states. In case of north, as mentioned earlier, going off the grid results in the same state with a reward of -1. Therefore : $v_{(k+1)(s)} = 0.25(-1 + (1)(1)(0)) + 0.25(-1 + (1)(1)(0)) + 0.25(-1 + (1)(1)(0)) + 0.25(-1 + (1)(1)(0))$ $v_{(k+1)(s)} = 0.25(-1 + -1 + -1 + -1) = \mathbf{-1.0}$

We repeat this for all the states in the MDP :

k = 1

0.0	–1.0	–1.0	–1.0
–1.0	–1.0	–1.0	–1.0
–1.0	–1.0	–1.0	–1.0
–1.0	–1.0	–1.0	–1.0

State Values at k = 1

Based on the values we calculated in the previous iteration we have:

$v_{k(s')}$ for north, south and east states is -1 and for the west state, it is 0. Therefore : $v_{(k+1)(s)} = 0.25(-1 + (1)(1)(-1)) + 0.25(-1 + (1)(1)(-1)) + 0.25(-1 + (1)(1)(-1)) + 0.25(-1 + (1)(1)(0))$ $v_{(k+1)(s)} = 0.25(-2 + -2 + -2 + -1) = \mathbf{-1.75}$

- **Note that** the values in the figures are rounded off to one decimal place:

k = 2

0.0	–1.7	–2.0	–2.0
–1.7	–2.0	–2.0	–2.0
–2.0	–2.0	–2.0	–1.7
–2.0	–2.0	–1.7	0.0

State Values at k = 2

- Similarly we compute for k = 3 :

$v_{k(s')}$ for south and east states is -2.0, for north it is -1.7 and for the west state, it is 0. Therefore : $v_{(k+1)(s)} = 0.25(-1 + (1)(1)(-1.7)) + 0.25 (-1 + (1)(1)(-2)) + 0.25 (-1 + (1)(1)(-2)) + 0.25 (-1 + (1)(1)(0))$ $v_{(k+1)(s)} = 0.25(-2.7 + -3 + -3 + -1) = -2.425$

k = 3

0.0	–2.4	–2.9	–3.0
–2.4	–2.9	–3.0	–2.9
–2.9	–3.0	–2.9	–2.4
–3.0	–2.9	–2.4	0.0

State Values at k = 3

- If we keep iterating, we will reach to a point where we can barely see any changes in the state value function. This indicates that **we have converged to the true value function v_π** :

k = 10

0.0	–6.1	–8.4	–9.0
–6.1	–7.7	–8.4	–8.4
–8.4	–8.4	–7.7	–6.1
–9.0	–8.4	–6.1	0.0

k = ∞

0.0	–14	–20	–22
–14	–18	–20	–20
–20	–20	–18	–14
–22	–20	–14	0.0

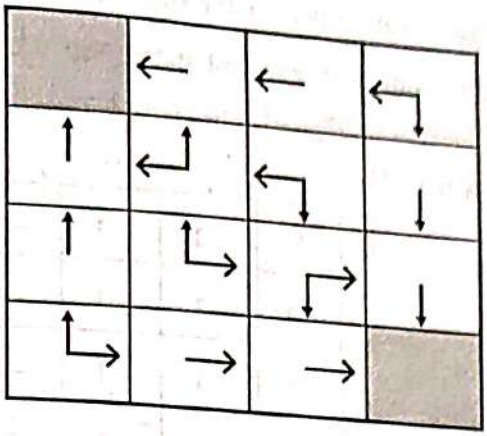
6.5.2(B) Policy Improvement

- In this step, we simply pick an action greedily based on our true value function :

$$\pi' = \text{greedy } (v_\pi)$$

- Greedily Updating the Current Policy

So basically, we assign probabilities to the actions based on the max value function values of the adjacent states. Therefore, based on v_{π}, we have:



Updated Policy, π'

Turns out, **this is the optimal policy for our grid-world problem**! However, in most cases, we have to follow the evaluation-improvement iterative pattern and stop once the policy stops improving.

In hindsight, if you notice, we could have obtained the above optimal policy even at k = 3 in the policy evaluation.

So why go up to $k = \infty$? Why not update the policy after every iteration (i.e., at $k = 1$)? This is exactly what's ne in **Value Iteration**.

3 Value Iteration

A policy π (a | s) achieves the optimal value from state s, $v\pi$ (s) = v* (s), if and only if

For any state s' reachable from s π achieves the optimal value from state s', v_{π} (s') = v* (s')

From the above definition, we can see that this technique uses recursion for each state accessible from s. It says that **if we know the solution to subproblems v*(s') (s' are all the states reachable from s), then the solution v*(s) can be found out with a simple one state look ahead** using:

$$v_*(s) \leftarrow \sum_{a \in A} R_s^a + \gamma \sum_{s' \in s} P_{ss'}^a v_*(s') \qquad \text{....(6.5.3)}$$

Equation 6.5.2 shows Bellman's Optimality Equation for Value Function

Since we are assuming the optimal value for the future states, we will use the **Bellman's Optimality Equation** (as opposed to the Bellman's Expectation Equation used in Policy Iteration).

The idea is to go backwards (i.e., start from the final state and then trace the path back). This will ensure the optimal value for the future states, as we are calculating them first.

Just like policy iteration, we initially assume that all of the states yield nothing, i.e., 0 :

k = 0

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Optimal State Values at k = 0

Now, we will take the optimality equation and calculate the state values iteratively.

- **Note that** $v_*(s')$ is 0 for all the north, south, east, and west states. Also, γ is 1. Therefore : $v_*(s) = \max(-1 + (1)(1)(0), -1 + (1)(1)(0), -1 + (1)(1)(0), -1 + (1)(1)(0))$ $v_*(s) = \max(-1, -1, -1, -1) = -1$
- We do this for all the states in the MDP :

k = 1

0	−1	−1	−1
−1	−1	−1	−1
−1	−1	−1	−1
−1	−1	−1	0

- Optimal State Values at k = 1
- Since we are going backwards, we now have the optimal values for the states adjacent to both goals. In the next step, we calculate the values for all the states except for these states (I'll explain why in a while). So we will obtain the optimal values for the states adjacent to these states, and so on... Therefore :

k = 2

0	−1	−2	−2
−1	−2	−2	−2
−2	−2	−2	−1
−2	−2	−1	0

k = 3

0	−1	−2	−3
−1	−2	−3	−2
−2	−3	−2	−1
−3	−2	−1	0

Optimal State Values at k = 2 and k = 3

If you notice, now we have the optimal policy at k = 3.

So what exactly happened?

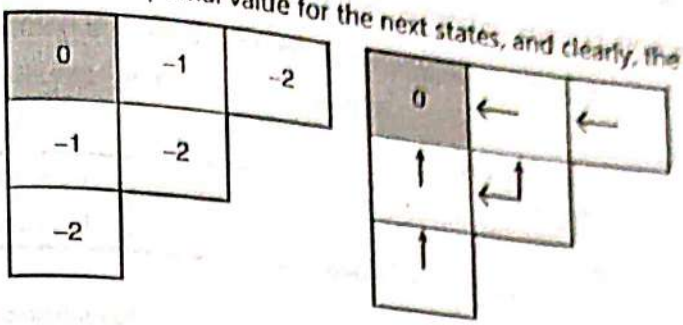
- Let's trace the path for the top left goal state. In the first iteration, we calculate the state values as we did for policy iteration.
- As said earlier, value iteration is equivalent to updating the policy at k=1 in policy iteration.
- Hence, after this step, our updated policy will have probability 1 to go towards the goal for the following two states :

0	−1
−1	

	←
↑	

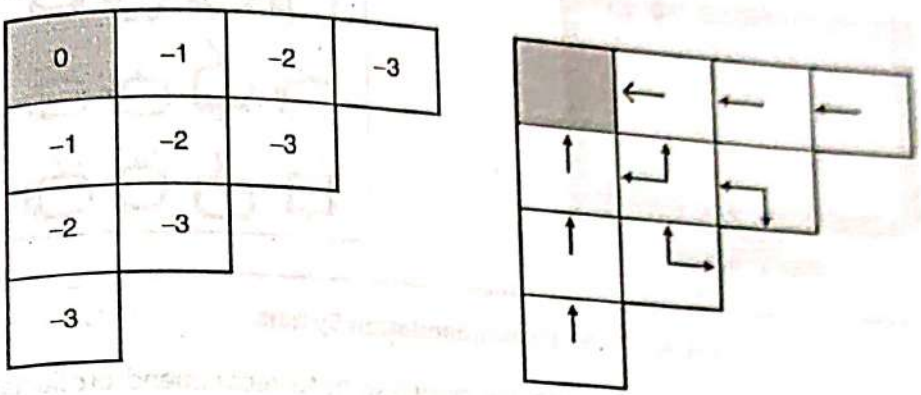
k = 1

Similarly, at k = 2, we will obtain the optimal value for the next states, and clearly, the optimal policy shown:



k = 2

And finally, at k = 3:



k = 3

we can see that two of these states are also influenced by the other goal (bottom right square) and hence the tgoing arrows.

Notice **that the state values in Value Iteration are more of a 'Shortest Path' from any state to the goal, contrary to Policy Iteration where the values are the 'Expected Values' for all the states.**

6 Q Learning and Deep Q-Networks

6.1 Q Learning

Q-Learning is a Reinforcement learning policy that will find the next best action, given a current state. It chooses this action at random and aims to maximize the reward.

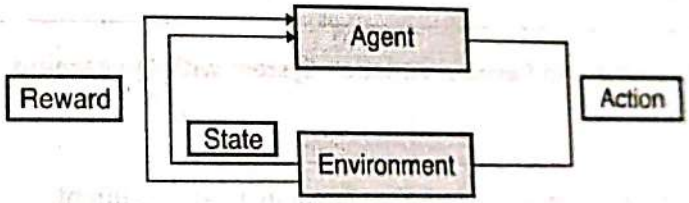


Fig. 6.6.1 : Components of Q-Learning

Q-learning is a model-free, off-policy reinforcement learning that will find the best course of action, given the current state of the agent. Depending on where the agent is in the environment, it will decide the next action to be taken.

The objective of the model is to find the best course of action given its current state. To do this, it may come up with rules of its own or it may operate outside the policy given to it to follow. This means that there is no actual need for a policy, hence we call it off-policy.

- Model-free means that the agent uses predictions of the environment's expected response to move forward. It does not use the reward system to learn, but rather, trial and error.
- An example of Q-learning is an Advertisement recommendation system. In a normal ad recommendation system, the ads you get are based on your previous purchases or websites you may have visited. If you've bought a TV, you will get recommended TVs of different brands.

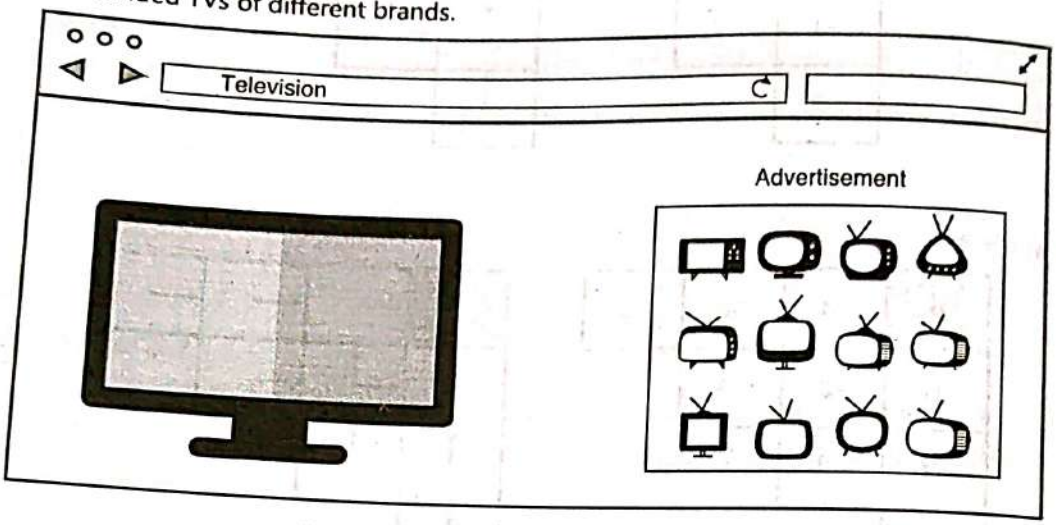


Fig. 6.6.2 : Ad Recommendation System

- Using Q-learning, we can optimize the ad recommendation system to recommend products that are frequently bought together. The reward will be if the user clicks on the suggested product.

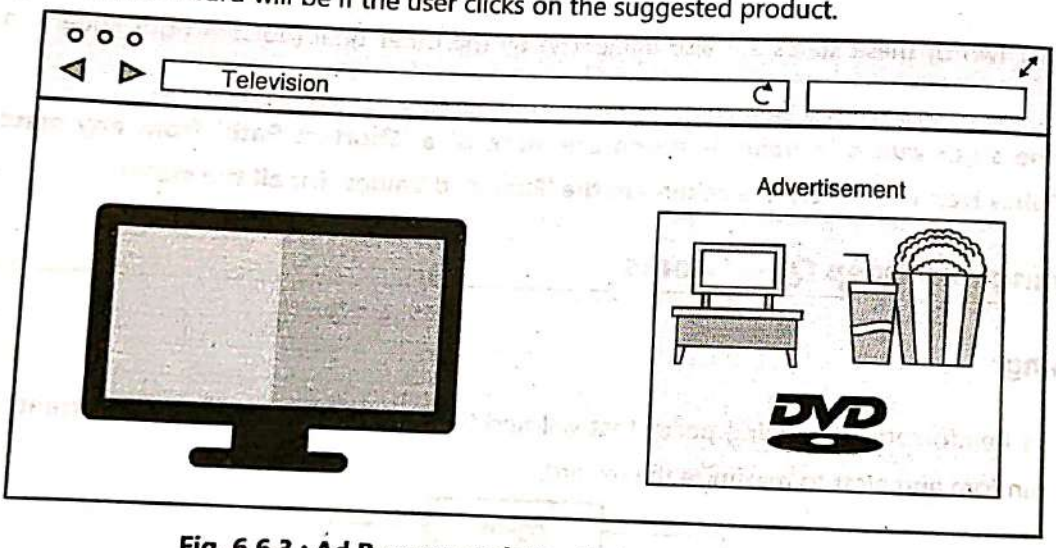


Fig. 6.6.3 : Ad Recommendation System with Q-Learning

Important Terms in Q-Learning

1. **States :** The State, S, represents the current position of an agent in an environment.
2. **Action :** The Action, A, is the step taken by the agent when it is in a particular state.
3. **Rewards :** For every action, the agent will get a positive or negative reward.
4. **Episodes :** When an agent ends up in a terminating state and can't take a new action.
5. **Q-Values :** Used to determine how good an Action, A, taken at a particular state, S, is. Q (A, S).
6. **Temporal Difference :** A formula used to find the Q-Value by using the value of current state and action and previous state and action.

What Is The Bellman Equation?

- The Bellman Equation is used to determine the value of a particular state and deduce how good it is to be in/take that state. The optimal state will give us the highest optimal value.
- The equation is given below. It uses the current state, and the reward associated with that state, along with the maximum expected reward and a discount rate, which determines its importance to the current state, to find the next state of our agent. The learning rate determines how fast or slow, the model will be learning.

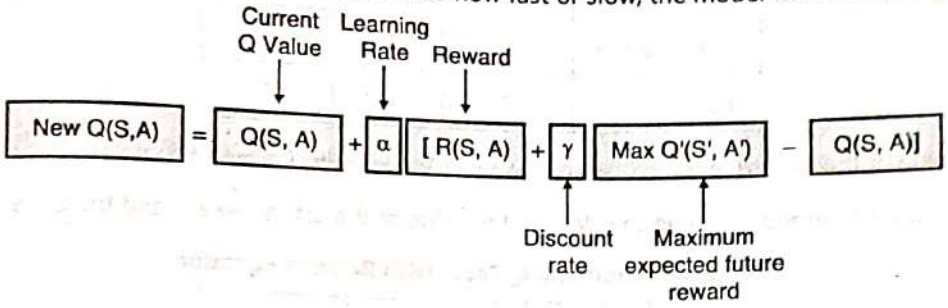


Fig. 6.6.4 : Bellman Equation

How to Make a Q-Table?

- While running our algorithm, we will come across various solutions and the agent will take multiple paths. How do we find out the best among them? This is done by tabulating our findings in a table called a Q-Table.
- A Q-Table helps us to find the best action for each state in the environment. We use the Bellman Equation at each state to get the expected future state and reward and save it in a table to compare with other states.
- Lets us create a q-table for an agent that has to learn to run, fetch and sit on command. The steps taken to construct a q-table are :

Step 1 : Create an initial Q-Table with all values initialized to 0

When we initially start, the values of all states and rewards will be 0. Consider the Q-Table shown below which shows a dog simulator learning to perform actions :

Table 6.6.1 : Initial Q-Table

Action	Fetching	Sitting	Running
Start	0	0	0
Idle	0	0	0
Wrong action	0	0	0
Correct action	0	0	0
End	0	0	0

Step 2 : Choose an action and perform it. Update values in the table

This is the starting point. We have performed no other action as of yet. Let us say that we want the agent to sit initially, which it does. The table will change to :

Table 6.6.2 : Q-Table after performing an action

Action	Fetching	Sitting	Running
Start	0	1	0
Idle	0	0	0
Wrong action	0	0	0
Correct action	0	0	0
End	0	0	0

Step 3 : Get the value of the reward and calculate the value Q-Value using Bellman Equation

For the action performed, we need to calculate the value of the actual reward and the Q (S, A) value

Table 6.6.3 : Updating Q-Table with Bellman Equation

Action	Fetching	Sitting	Running
Start	0	1	0
Idle	0	0	0
Wrong action	0	0	0
Correct action	0	34	0
End	0	0	0

Step 4 : Continue the same until the table is filled or an episode ends

The agent continues taking actions and for each action, the reward and Q-value are calculated and it updates the table.

Table 6.6.3 : Final Q-Table at end of an episode

Action	Fetching	Sitting	Running
Start	5	7	10
Idle	2	5	3
Wrong action	2	6	1
Correct action	54	34	17
End	3	1	4

6.6.1(A) Advantages of Q learning

- Long-term outcomes, which are exceedingly challenging to accomplish, are best achieved with this strategy.
- This learning paradigm closely resembles how people learn. Consequently, it is almost ideal.
- The model has the ability to fix mistakes made during training.
- Once a model has fixed a mistake, there is virtually little probability that it will happen again.
- It can produce the ideal model to address a certain issue.

6.6.2 Deep Q-Network

- Q-Learning is required as a pre-requisite as it is a process of Q-Learning creates an exact matrix for the working agent which it can "refer to" to maximize its reward in the long run.
- Although this approach is not wrong in itself, this is only practical for very small environments and quickly loses its feasibility when the number of states and actions in the environment increases.
- The solution for the above problem comes from the realization that the values in the matrix only have relative importance i.e. the values only have importance with respect to the other values. Thus, this thinking leads us to Deep Q-Learning which uses a deep neural network to approximate the values.
- This approximation of values does not hurt as long as the relative importance is preserved. The basic working step for Deep Q-Learning is that the initial state is fed into the neural network and it returns the Q-value of all possible actions as an output.
- The difference between Q-Learning and Deep Q-Learning can be illustrated as follows :

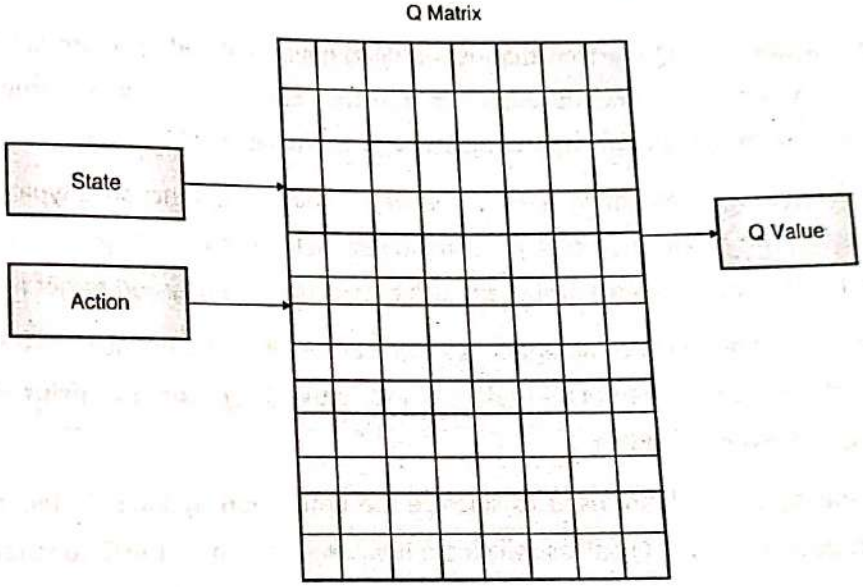


Fig. 6.6.5

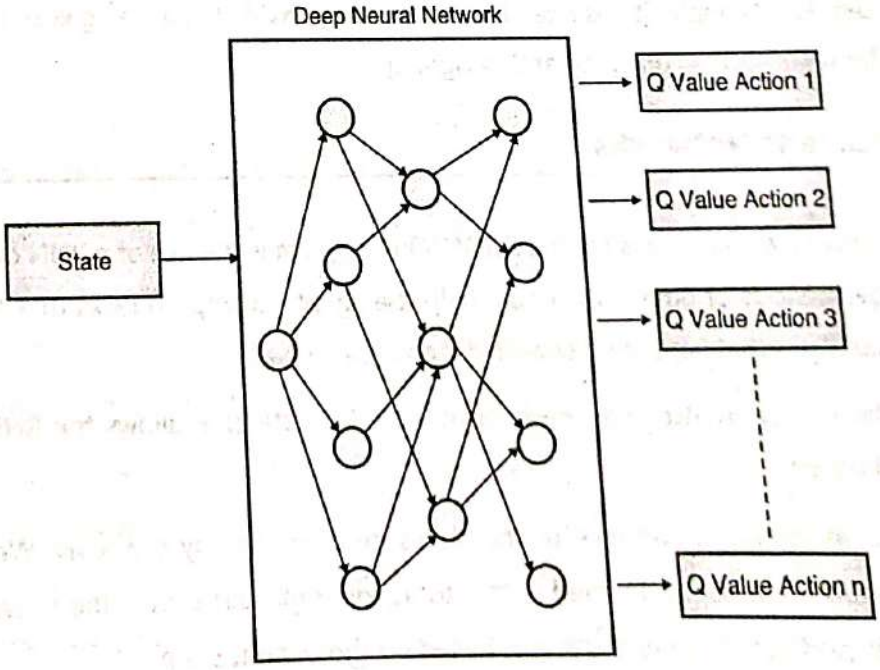


Fig. 6.6.6 : Q learning and Deep Q learning

- Observe that in the equation **target = R(s,a,s') +** , the term $\gamma \max_{a'} Q_k(s', a')$ is a variable term. Therefore in this process, the target for the neural network is variable unlike other typical Deep Learning processes where the target is stationary.
- This problem is overcome by having two neural networks instead of one. One neural network is used to adjust the parameters of the network and the other is used for computing the target and which has the same architecture as the first network but has frozen parameters. After an x number of iterations in the primary network, the parameters are copied to the target network.
- Deep Q-Learning is a type of reinforcement learning algorithm that uses a deep neural network to approximate the Q-function, which is used to determine the optimal action to take in a given state.
- The Q-function represents the expected cumulative reward of taking a certain action in a certain state and following a certain policy. In Q-Learning, the Q-function is updated iteratively as the agent interacts with the environment. Deep Q-Learning is used in various applications such as game playing, robotics and autonomous vehicles.
- Deep Q-Learning is a variant of Q-Learning that uses a deep neural network to represent the Q-function, rather than a simple table of values. This allows the algorithm to handle environments with a large number of states and actions, as well as to learn from high-dimensional inputs such as images or sensor data.
- One of the key challenges in implementing Deep Q-Learning is that the Q-function is typically non-linear and can have many local minima. This can make it difficult for the neural network to converge to the correct Q-function. To address this, several techniques have been proposed, such as experience replay and target networks.
- Experience replay is a technique where the agent stores a subset of its experiences (state, action, reward, next state) in a memory buffer and samples from this buffer to update the Q-function. This helps to decorrelate the data and make the learning process more stable.
- Target networks, on the other hand, are used to stabilize the Q-function updates. In this technique, a separate network is used to compute the target Q-values, which are then used to update the Q-function network.
- Deep Q-Learning has been applied to a wide range of problems, including game playing, robotics, and autonomous vehicles. For example, it has been used to train agents that can play games such as Atari and Go, and to control robots for tasks such as grasping and navigation.

6.7 Deep Q Recurrent Networks

- Here We examined several architectures for the DRQN. One of them is the use of a RNN on top of a DQN, to retain information for longer periods of time. This should help the agent accomplish tasks that may require the agent to remember a particular event that happened several dozens screen back.
- We also examined the use of an attention mechanism in RNNs. Attention allows the RNN to focus on particular states it has seen in the past.
- One can think of it as assigning importance to the states iterated over by the RNN. We investigate 2 forms of attention, a linear attention that uses a learned vector to assign importance over the previous states and a global attention that assigns importance to previous states based on the current state.

- The first architecture is a basic extension of DQN. The architecture of DRQN augments DQN's fully connected layer with a LSTM. We accomplish this by looking at the last L states, {st−(L−1), . . . ,st}, and feed these into a convolutional neural network (CNN) to get intermediate outputs CNN (st−i) = xt−i .
- These are then fed into a RNN (we use an LSTM for this but it can be any RNN), RNN (xt−i , ht−i−1) = ht−i , and the final output ht is used to predict the Q value which is now a function of Q({st−(L−1), . . . , st}, at).

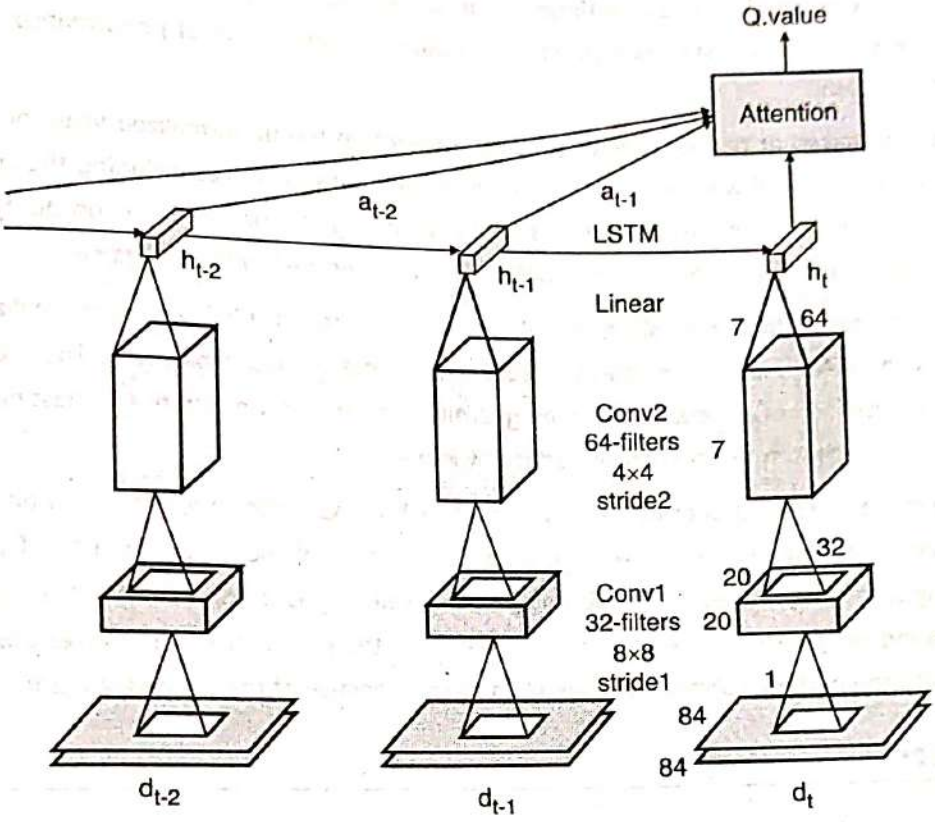


Fig. 6.7.1 : Architecture of the Attention DRQN

- The second architecture we used was a version of an attention RNN, we are calling linear attention. For the linear attention RNN, we take the L hidden states outputted by the RNN, {ht−(L−1), . . . ,ht}, and calculate an inner product with va, {v T a ht−(L−1), . . . , vT a ht}.
- This allows the model to focus more on nearby hidden states or further away states depending on what values of va are learned. We then take a SoftMax over these values, at−i = SoftMax (v T a ht−i). We use this SoftMax to take a weighted sum over the hidden states to get a context vector, ct = PL−1 i=0 at−iht−i . This context vector is then used to predict the Q value.
- We treat the current state st as the "decoder" input and the previous L − 2 states as the "encoder" inputs. We compute the following scores, {ht−(L−1) T ht, . . . , ht−1 T ht}.
- We then take a SoftMax over these values, at−i = SoftMax (h_t−i T h_t). The context vector is computed as a weighted sum over the previous hidden states, c_t = PL−1 i=1 at−ih_t−i . Finally, the context vector is used to compute h˜ = tanh (Wa[ht; ct] + ba), which is then used to predict the Q value. This type of attention allows the model to focus on previous states depending on the current state ht, as opposed to a fixed vector such as va.

6.8 Simple Reinforcement Learning for Tic-Tac-Toe

- One can generalize the stateless _-greedy algorithm to learn to play the game of tic-tac-toe. In this case, each board position is a state, and the action corresponds to placing 'X' or 'O' at a valid position. The number of valid states of the 3×3board is bounded above by 39 = 19683, which corresponds to three possibilities ('X', 'O', and blank) for each of 9 positions. Instead of estimating the value of each (stateless) action in multi-armed bandits, we now estimate the value of each state-action *pair* (*s*, *a*) based on the historical performance of action *a* in states against a fixed opponent.
- Shorter wins are preferred at discount factor γ <1, and therefore the unnormalized value of action a in state sis increased with γr−1 in case of wins and −γr−1 in case of losses after *r* moves(including the current move). Draws are credited with 0. The discount also reflects the fact that the significance of an action decays with time in real-world settings. In this case, the table is updated only after all moves are made for a game.
- The normalized values of the action sin the table are obtained by dividing the unnormalized values with the number of times the state-action pair was updated (which is maintained separately). The table starts with small random values, and the action a in state sis chosen greedily to be the action with the highest normalized value with probability 1 −_, and is chosen to be a random action otherwise.
- All moves in a game are credited after the termination of each game. Over time, the values of all state-action pairs will be learned and the resulting moves will also adapt to the play of the fixed opponent. Furthermore, one can even use self-play to generate these tables optimally. When self-play is used, the table is updated from a value in {−γr, 0, γr}depending on win/draw/loss from the perspective of the player for whom moves are made. At inference time, the move with the highest normalized value from the perspective of the player are made.

6.9 Case Studies

6.9.1 Self Driving Cars

- Reinforcement learning (RL) is a type of machine learning where an agent learns by exploring and interacting with the environment. In this case, the self-driving car is an agent.
- At the core of DRL, we have three important variables:
- State describes the current situation in a given time. In this case, it would be a position on the road.
- Action describes all the possible moves that the car can make.
- Reward is feedback that the car receives whenever it takes a certain action.
- Generally, the agent is not told what to do or what actions to take. So far as we have seen, in supervised learning, the algorithm maps input to the output. In DRL, the algorithm learns by exploring the environment and each interaction yields a certain reward. The reward can be both positive and negative. The goal of the DRL is to maximize the cumulative rewards.
- In self-driving cars, the same procedure is followed: the network is trained on perception data, where it learns what decision it should make. Because the CNNs are very good at extracting features of representations from the input, DRL algorithms can be trained on those representations. Training a DRL algorithm on these representations can yield good results because these extracted representations are the transformation of higher-dimensional manifolds

into simpler lower-dimensional manifolds. Training on lower representation yields efficiency which is required at the inference.

- One key point to remember is that self-driving cars can't be trained in real-world scenarios or roads because they will be extremely dangerous. Instead, self-driving cars are trained on a simulator where there's no risk at all.
- Some open-source simulators are:
  - CARLA
  - SUMMIT
  - AirSim
  - DeepDrive
  - Flow

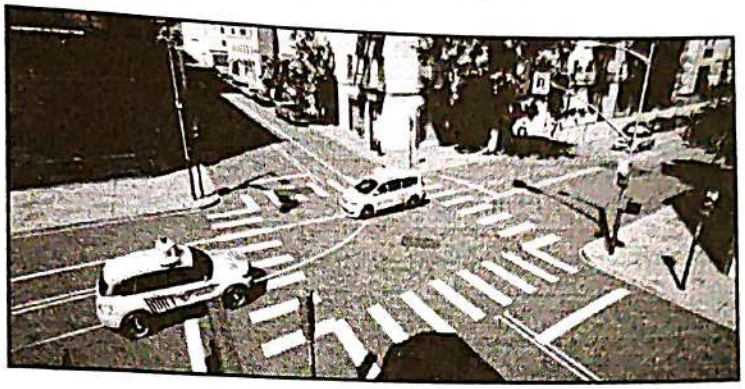


Fig. 6.9.1 : A snapshot from Voyage Deepdrive

- These cars (agents) are trained for thousands of epochs with highly difficult simulations before they're deployed in the real world.
- During training, the agent (the car) learns by taking a certain action in a certain state. Based on this state-action pair, it receives a reward. This process happens over and over again. Each time the agent updates its memory of rewards. This is called the policy.
- The policy is described as how the agent makes decisions. It's a decision-making rule. The policy defines the behaviour of the agent at a given time.
- For every negative decision the agent makes, the policy is changed. So in order to avoid the negative rewards, the agent checks the quality of a certain action. This is measured by the state-value function. State-value can be measured using the Bellman Expectation Equation.
- The Bellman expectation equation, along with Markov Decision Process (MDP), makes up the two core concepts of DRL But when it comes to self-driving cars, we have to keep in mind that the observations from the perception data should be mapped with the appropriate action and not just map the underlying state to the action. This is where a partially observed decision process or a Partially Observable Markov Decision Process (POMDP) is required, which can make decisions based on the observation.

Partially Observable Markov Decision Process used for self-driving cars

- The Markov Decision Process gives us a way to sequentialize decision-making. When the agent interacts with the environment, it does so sequentially over time.

- Each time the agent interacts with the environment, it gives some representation of the environment state. Given the representation of the state, the agent selects the action to take, as in the Fig. 6.9.2.

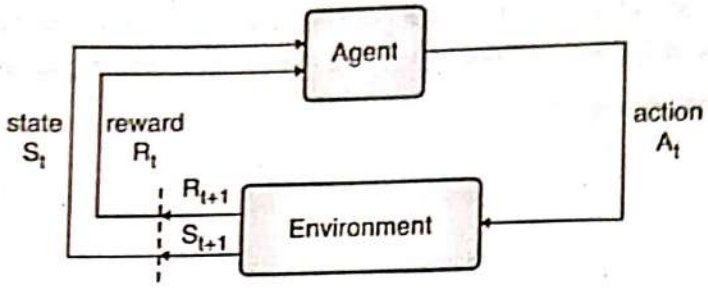


Fig. 6.9.2 : Markov Process

- The action taken is transitioned into some new state and the agent is given a reward. This process of evaluating a state, taking action, changing states, and rewarding is repeated. Throughout the process, it's the agent's goal to maximize the total amount of rewards.
- Let's get a more constructive idea of the whole process:
  - At a give time t, the state of the environment is at St
  - The agent observes the current state St and selects an action At
  - The environment is then transitioned into a new state St+1, simultaneously the agent is rewarded Rt
- In a partially observable Markov decision process (POMDP), the agent senses the environment state with observations received from the perception data and takes a certain action followed by receiving a reward.
- The POMDP has six components and it can be denoted as POMDP M:= (I, S, A, R, P, γ), where,

I: Observations

S: Finite set of states

A: Finite set of actions

R: Reward function

P: transition probability function

γ – discounting factor for future rewards.

Q-learning used for self-driving cars

- Q-learning is one of the most commonly used DRL algorithms for self-driving cars. It comes under the category of model-free learning. In model-free learning, the agent will try to approximate the optimal state-action pair. The policy still determines which action-value pairs or Q-value are visited and updated (see the equation below). The goal is to find optimal policy by interacting with the environment while modifying the same when the agent makes an error.
- With enough samples or observation data, Q-learning will learn optimal state-action value pairs. In practice, Q-learning has been shown to converge to the optimum state-action values for a MDP with probability 1, provided that all actions in all states are infinitely available.

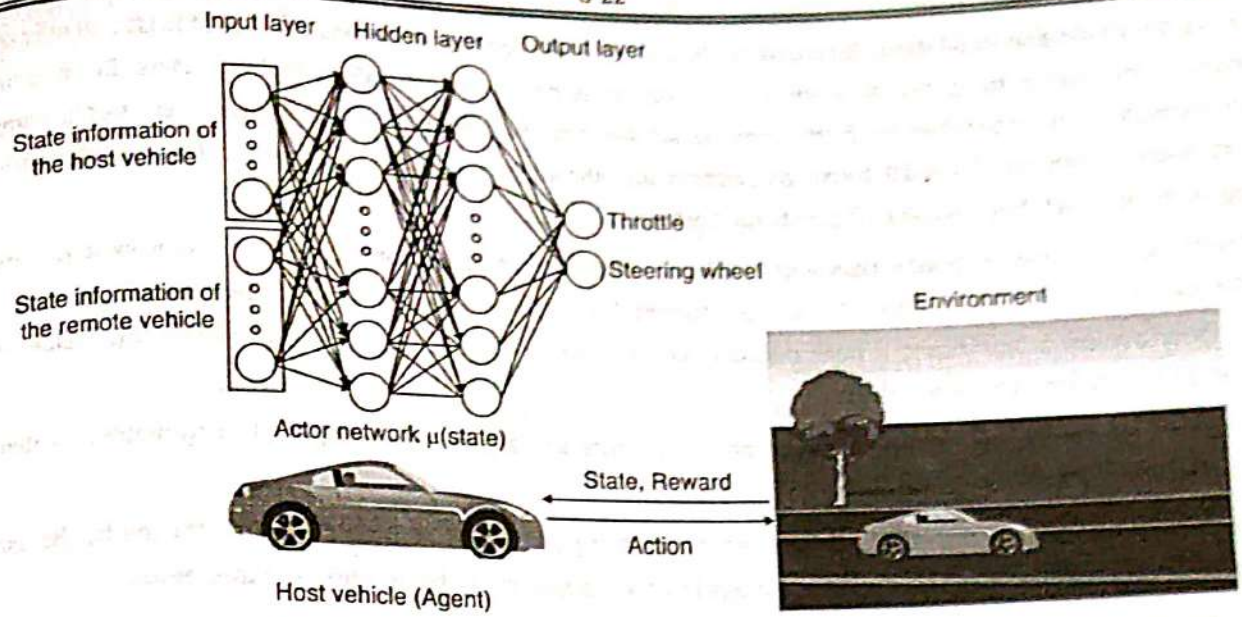


Fig. 6.9.3 : Architecture of proposed decision-making system

6.9.2 Deep Learning for Chatbots

- Chatbots are also referred to as conversational systems or dialog systems. The ultimate goal of a chatbot is to build an agent that can freely converse with a human about a variety of topics in a natural way.
- We are very far from achieving this goal. However, significant progress has been made in building chatbots for specific domains and particular applications (e.g., negotiation or shopping assistant).
- An example of a relatively general-purpose system is Apple's Siri, which is a digital personal assistant. One can view Siri as an open-domain system, because it is possible to have conversations with it about a wide variety of topics.
- It is reasonably clear to anyone using Siri that the assistant is sometimes either unable to provide satisfactory responses to difficult questions, and in some cases hilarious responses to common questions are hard-coded.
- This is, of course, natural because the system is relatively general-purpose, and we are nowhere close to building a human-level conversational system. In contrast, closed-domain systems have a specific task in mind, and can therefore be more easily trained in a reliable way.
- In the following, we will describe a system built by Facebook for end-to-end learning of negotiation skills. This is a closed-domain system because it is designed for the particular purpose of negotiation.
- As a test-bed, the following negotiation task was used. Two agents are shown a collection of items of different types (e.g., two books, one hat, three balls). The agents are instructed to divide these items among themselves by negotiating a split of the items.
- A key point is that the value of each of the types of items is different for the two agents, but they are not aware of the value of the items for each other. This is often the case in real-life negotiations, where users attempt to reach a mutually satisfactory outcome by negotiating for items of value to them.
- The values of the items are always assumed to be non-negative and generated randomly in the test-bed under some constraints.

- First, the total value of all items for a user is 10. Second, each item has non-zero value to at least one user so that it makes little sense to ignore an item. Last, some items have nonzero values to both users. Because of these constraints, it is impossible for both users to achieve the maximum score of 10, which ensures a competitive negotiation process. After 10 turns, the agents are allowed the option to complete the negotiation with no agreement, which has a value of 0 points for both users.
- The three item types of books, hats, and balls were used, and a total of between 5 and 7 items existed in the pool. The fact that the values of the items are different for the two users (without knowledge about each other's assigned values) is significant; if both negotiators are capable, they will be able to achieve a total value of larger than 10 for the items between them.
- Nevertheless, the better negotiator will be able to capture the larger value by optimally negotiating for items with a high value for them.
- The reward function for this reinforcement learning setting is the final value of the items attained by the user. One can use supervised learning on previous dialogs in order to maximize the likelihood of utterances.
- A straightforward use of recurrent networks to maximize the likelihood of utterances resulted in agents that were too eager to compromise. Therefore, the approach combined supervised learning with reinforcement learning.
- The incorporation of supervised learning within the reinforcement learning helps in ensuring that the models do not diverge from human language.
- A form of planning for dialogs called dialog roll-out was introduced. The approach uses an encoder-decoder recurrent architecture, in which the decoder maximizes the reward function rather than the likelihood of utterances. This encoder-decoder architecture is based on sequence-to-sequence learning.
- To facilitate supervised learning, dialogs were collected from Amazon Mechanical Turk. A total of 5808 dialogs were collected in 2236 unique scenarios, where a scenario is defined by assignment of a particular set of values to the items. Of these cases, 252 scenarios corresponding to 526 dialogs were held out. Each scenario results in two training examples, which are derived from the perspective of each agent.
- A concrete training example could be one in which the items to be divided among the two agents correspond to 3 books, 2 hats, and 1 ball. These are part of the input to each agent. The second input could be the value of each item to the agent, which are (i) Agent A: book:1, hat:3, ball:1, and (ii) Agent B : book:2, hat:1, ball:2.
- Note that this means that agent A should secretly try to get as many hats as possible in the negotiation, whereas agent B should focus on books and balls.
- An example of a dialog in the training data is given below:
  - Agent A : I want the books and the hats, you get the ball.
  - Agent B : Give me a book too and we have a deal.
  - Agent A : Ok, deal.
  - Agent B : choose
- The final output for agent A is 2 books and 2 hats, whereas the final output for agent B is 1 book and 1 ball. Therefore, each agent has her own set of inputs and outputs, and the dialogs for each agent are also viewed from their own perspective in terms of the portions that are reads and the portions that are writes.

- Therefore, each scenario generates two training examples and the same recurrent network is shared for generating the writes and the final output of each agent. The dialog x is a list of tokens x0 ...xT , containing the turns of each agent interleaved with symbols marking whether the turn was written by an agent or their partner. A special token at the end indicates that one agent has marked that an agreement has been reached.
- The supervised learning procedure uses four different gated recurrent units (GRUs). The first gated recurrent unit GRUg encodes the input goals, the second gated recurrent unit GRUq generates the terms in the dialog, a forward-output gated recurrent unit GRUo, and a backward-output gated recurrent unit GRUo.
- The output is essentially produced by a bidirectional GRU. These GRUs are hooked up in end-to-end fashion. In the supervised learning approach, the parameters are trained using the inputs, dialogs, and outputs available from the training data.
- The loss for the supervised model for a weighted sum of the token-prediction loss of the dialog and the output choice prediction loss of the items.
- However, for reinforcement learning, dialog roll-outs are used. Note that the group of GRUs in the supervised model is, in essence, providing probabilistic outputs.
- Therefore, one can adapt the same model to work for reinforcement learning by simply changing the loss function. In other words, the GRU combination can be considered a type of policy network.
- One can use this policy network to generate Monte Carlo roll-outs of various dialogs and their final rewards. Each of the sampled actions becomes a part of the training data, and the action is associated with the final reward of the roll-out.
- First, the supervised learning methods often tended to give up easily, where as the reinforcement learning methods were more persistent in attempting to obtain a good deal. Second, the reinforcement learning method would often exhibit human-like negotiation tactics. In some cases, it feigned interest in an item that was not really of much value in order to obtain a better deal for another item.

Review Questions

Q. 1 What is Reinforcement Learning?

Q. 2 Write short note on Markov Chain.

Q. 3 Explain Characteristics of Reinforcement Learning

Q. 4 What are the Challenges in Reinforcement Learning?

Q. 5 Explain in details dynamic programming algorithms for reinforcement learning.

Q. 6 Explain the process of Deep Q-learning?

Q. 7 Is LSTM a reinforcement learning?

Q. 8 Explain reinforcement learning for Tic-Tac-Toe game.

Q. 9 Write case study on Self driving cars

Q. 10 Write case study on Deep learning for Chatbots.

□□□