UNIT IV

Convolutional Neural Networks: Nerual Network and Representation Learing, Convolutional Layers, Multichannel Convolution Operation, **Recurrent Neural Networks:** Introduction to RNN, RNN Code, PyTorch Tensors: Deep Learning with PyTorch, CNN in PyTorch

……

Convolutional neural networks (CNNs) are a type of deep learning neural network that is specifically designed for image processing and computer vision tasks. CNNs are inspired by the structure and function of the human visual cortex, which is the part of the brain that is responsible for processing visual information.

CNNs have a number of advantages over other types of neural networks for image processing and computer vision tasks:

- **They are able to extract features from images that are invariant to translation, rotation, and scaling.** This means that the same features can be detected even if the object in the image is in a different position or orientation.
- **They are able to extract features from images that are hierarchically organized.** This means that they can start by extracting simple features, such as edges and corners, and then use those features to extract more complex features, such as faces and objects.
- **They are computationally efficient.** This is because the convolution operation at the heart of CNNs can be implemented using the fast Fourier transform (FFT).

CNNs have revolutionized the field of computer vision. They are now used in a wide range of applications, including:

- **Image classification:** CNNs can be used to classify images into different categories, such as cats, dogs, and cars.
- **Object detection:** CNNs can be used to detect objects in images, such as pedestrians, cars, and traffic signs.
- **Facial recognition:** CNNs can be used to recognize faces in images.
- **Medical imaging:** CNNs can be used to analyze medical images, such as X-rays and MRI scans, to diagnose diseases and identify abnormalities.
- **Natural language processing:** CNNs can be used to extract features from text, which can then be used for tasks such as sentiment analysis and machine translation.

CNNs are a powerful tool for a wide range of applications. They are still under active development, and new applications for CNNs are being discovered all the time.

Here are some specific examples of how CNNs are being used today:

- Facebook uses CNNs to recognize faces in photos.
- Google uses CNNs to power its image search engine.
- Tesla uses CNNs to power its self-driving cars.
- Doctors use CNNs to analyze medical images and diagnose diseases.
- Researchers are using CNNs to develop new methods for machine translation and text analysis.

CNNs are a powerful and versatile tool that is transforming the way we interact with the world around us.

Artificial neural networks (ANNs) Vs convolutional neural networks (CNNs)

Artificial neural networks (ANNs) and convolutional neural networks (CNNs) are both types of deep learning neural networks. However, they have different architectures and are used for different types of tasks.

ANNs are general-purpose neural networks that can be used for a variety of tasks, including classification, regression, and clustering. They are typically made up of a series of fully connected layers, meaning that each neuron in one layer is connected to every neuron in the next layer.

CNNs are a type of ANN that is specifically designed for image processing and computer vision tasks. They are made up of a series of convolutional layers, which are able to extract features from images that are invariant to translation, rotation, and scaling.

Here is a table that summarizes the key differences between ANNs and CNNs:

Characteristic	ANN	CNN
Architecture	Fully connected layers	Convolutional layers
Applications	General-purpose	Image processing, computer vision
Advantages	Flexible and versatile	Able to extract invariant features from images
Disadvantages	Can be computationally expensive	Requires a large amount of labeled training data

Which type of neural network to use depends on the specific task at hand. If you are working on a general-purpose task, such as classification or regression, then an ANN may be a good choice. If you are working on an image processing or computer vision task, then a CNN is likely to be a better choice.

Here are some examples of when to use ANNs and CNNs:

- **ANNs:**
    - Classifying text documents into different categories
    - Predicting customer churn
    - Recommending products to customers
- **CNNs:**
    - Classifying images of objects
    - Detecting objects in images
    - Segmenting images

1. Nerual Network and Representation Learing

- Neural networks initially receive data on observations, with each observation represented by some number n features.
- A simple neural network model with one hidden layer performed better than a model without that hidden layer.
- One reason is that the neural network could learn nonlinear relationships between input and output.
- However, a more general reason is that in machine learning, we often need linear combinations of our original features in order to effectively predict our target.
- Let's say that the pixel values for an MNIST digit are x1 through x784.
- There may be many other such combinations, all of which contribute positively or negatively to the probability that an image is of a particular digit.
- Neural networks can automatically discover combinations of the original features that are important through their training process.
- This process of learning which combinations of features are important is known as representation learning, and it's the main reason why neural networks are successful across different domains.

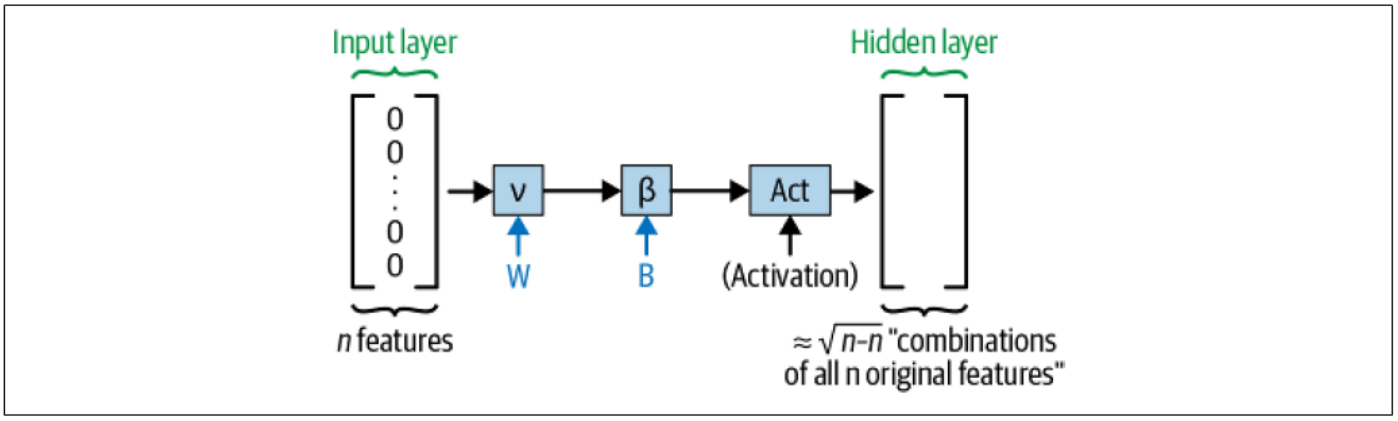


Figure 5-1. The neural networks we have seen so far start with n features and then learn somewhere between $\sqrt{n}$ and n "combinations" of these features to make predictions

Is there any reason to modify this process for image data?

The fundamental insight that suggests the answer is "yes" is that in images, the interesting "combinations of features" (pixels) tend to come from pixels that are close together in the image.

We want to exploit this fundamental fact about image data: that the order of the features matters since it tells us which pixels are near each other spatially.

But how do we do it?

1.1. A Different Architecture for Image Data

The solution, at a high level, will be to create combinations of features, as before, but an order of magnitude more of them, and have each one be only a combination of the pixels from a small rectangular patch in the input image. Figure 5-2 describes this.

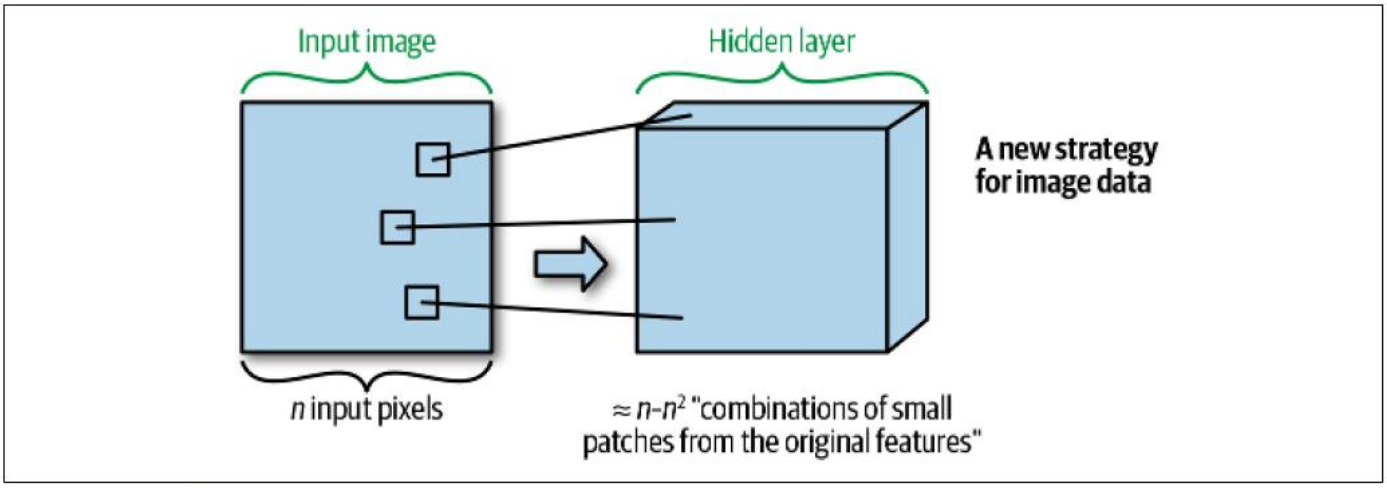


Figure 5-2. With image data, we can define each learned feature to be a function of a small patch of data, and thus define somewhere between n and n^2 output neurons

Having our neural network learn combinations of all of the input features—that is, combinations of all of the pixels in the input image—turns out to be very inefficient, since it ignores the insight described in the prior section: that most of the interesting combinations of features in images occur in these small patches.

What operation can we use to compute many combinations of the pixels from local patches of the input image?

The answer is the convolution operation.

1.2. The Convolution Operation

The convolution operation is a fundamental operation in deep learning, especially in convolutional neural networks (CNNs). CNNs are a type of neural network that is specifically designed for image processing and computer vision tasks.

CNNs use convolution operations to extract features from images. Features are patterns in the image that can be used to identify and classify objects. For example, some features of a face might include the eyes, nose, and mouth.

Convolution operations are performed by sliding a small filter over the image and computing the dot product of the filter and the image pixels at each location. The filter is typically a small square or rectangular array of weights. The result of the convolution operation is a new image that is smaller than the original image.

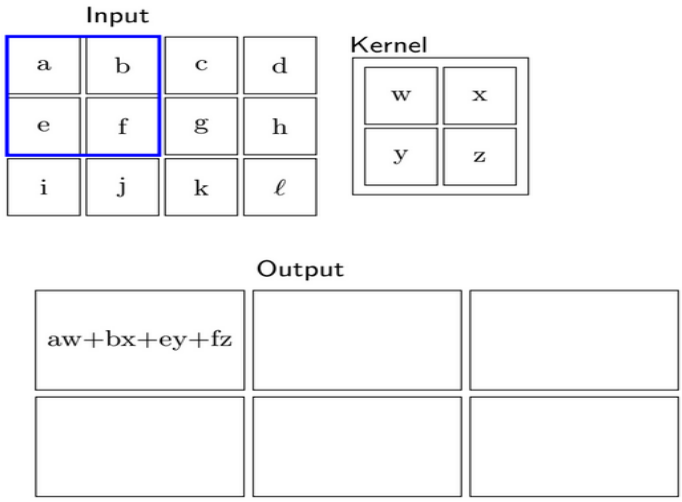
The new image contains the features that were extracted by the filter. For example, a filter might be designed to extract edge features from an image. The output of the convolution operation with this filter would be an image that highlights the edges in the original image.

CNNs typically have multiple convolutional layers, each of which uses a different filter to extract different features from the image. The output of the convolutional layers is then fed into a fully connected neural network, which performs classification or other tasks.

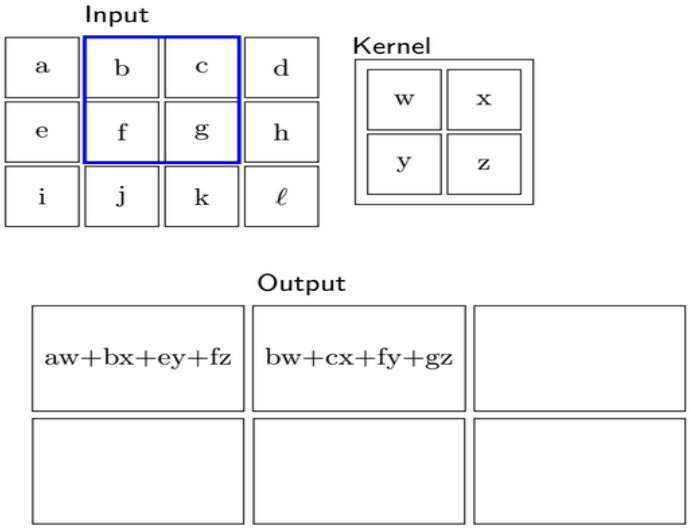
We compute the output(re-estimated value of current pixel) using the following formula:

$$S_{ij} = (I * K)_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{n-1} I_{i+a,j+b} K_{a,b}$$

Here m refers to the number of rows(which is 2 in this case) and n refers to the number of columns(which is 2 i this case).

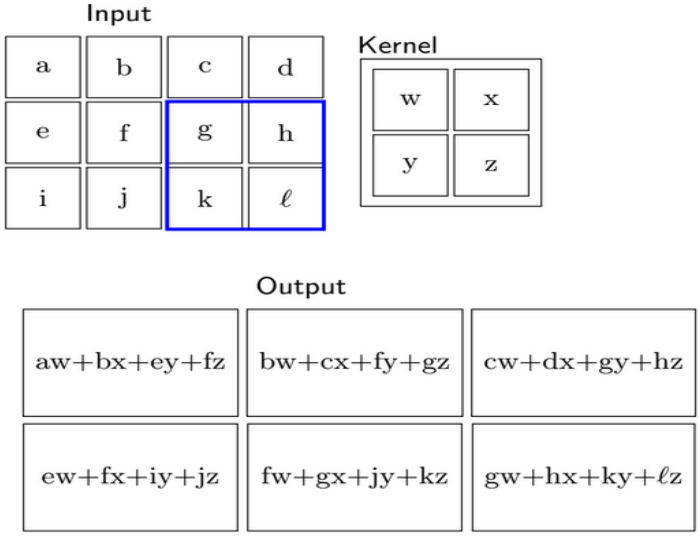


- Let us apply this idea to a toy example and see the results



- Let us apply this idea to a toy example and see the results

Similarly, we do the rest



- Let us apply this idea to a toy example and see the results

2. The Multichannel Convolution Operation

To review: convolutional neural networks differ from regular neural networks in that they create an order of magnitude more features, and in that each feature is a function of just a small patch from the input image.

Now we can get more specific: starting with *n* input pixels, the convolution operation just described will create *n* output features, one for each location in the input image.

What actually happens in a convolutional Layer in a neural network goes one step further: there, well create *f sets* of *n* features, *each* with a corresponding (initially random) set of weights defining a visual pattern whose detection at each location in the input image will be captured in the feature map.

These *f* feature maps will be created via *f* convolution operations. This is captured in Figure 5-3.

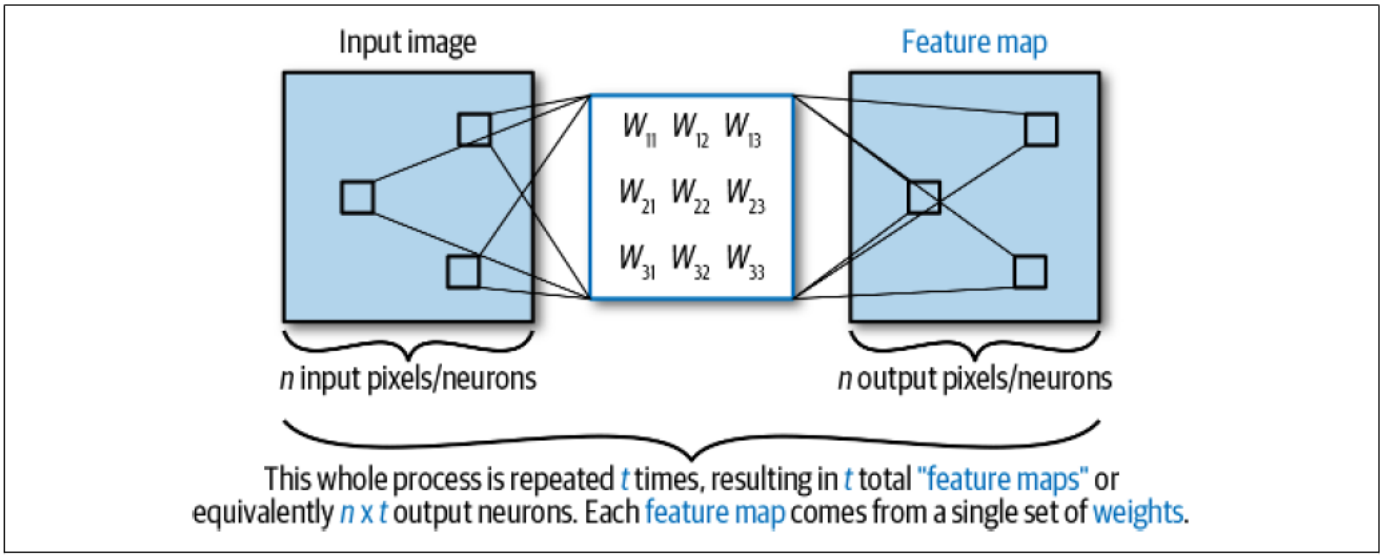


Figure 5-3. More specifically than before, for an input image with n pixels, we define an output with f feature maps, each of which has about the same size as the original image, for a total of n × f total output neurons for the image, each of which is a function of only a small patch of the original image

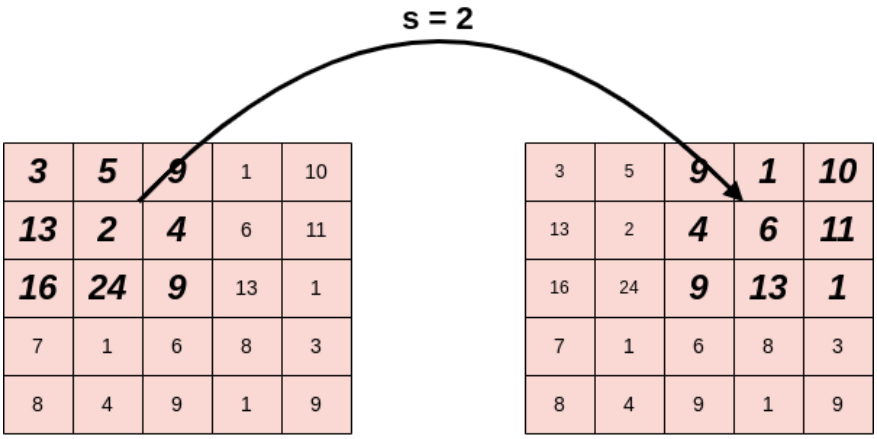
While each "set of features" detected by a particular set of weights is called a **feature map**, in the context of a convolutional Layer, the number of feature maps is referred to as the number of **channels** of the Layer—this is why the operation involved with the Layer is called the multichannel convolution. In addition, the f sets of weights Wi are called the convolutional filters.

Q) Relation between input size, output size and filter size

Stride

During convolution, the filter slides from left to right and from top to bottom until it passes through the entire input image. We define **stride as the step of the filter.** So, when we want to down sample the input image and end up with a smaller output, we set S>0.

Below, we can see example when:

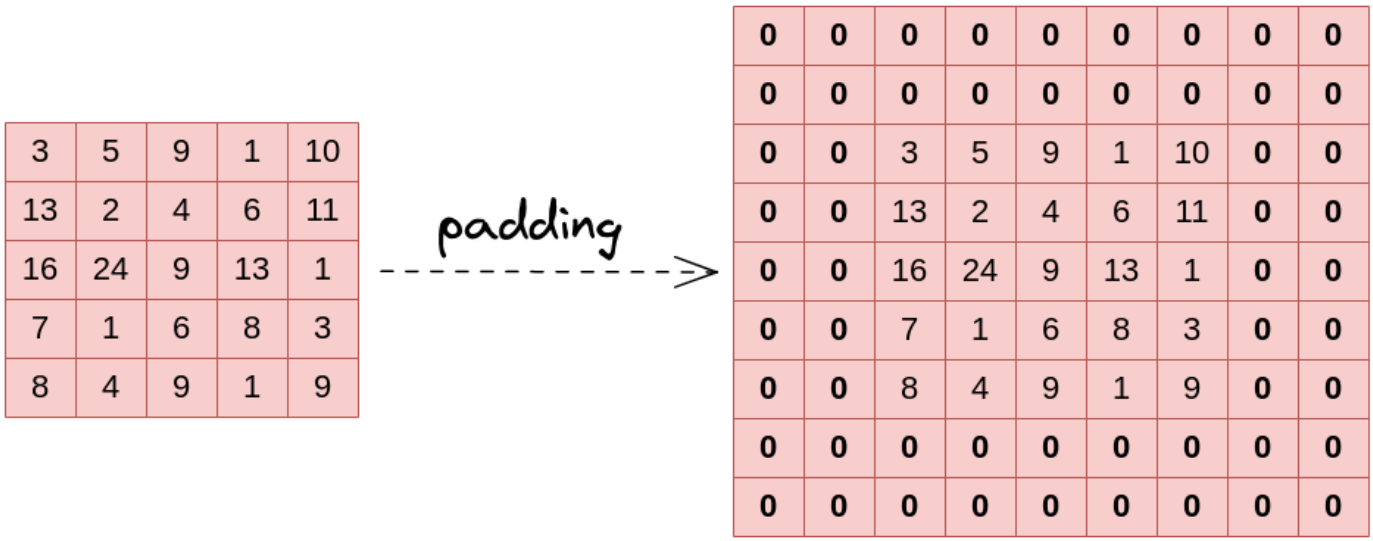


Padding

In a convolutional layer, we observe that **the pixels located on the corners and the edges are used much less than those in the middle.**

A simple and powerful solution to this problem is padding, which adds rows and columns of zeros to the input image. If we apply padding in an input image of size HXH ,the output image has dimensions (W+2P)X(H+2P).

Below we can see an example image before and after padding with $P = 2$. As we can see, the dimensions increased from 5×5 to 9×9:



By using padding in a convolutional layer, we increase the contribution of pixels at the corners and the edges to the learning procedure.

More Edge Detection

The type of filter that we choose helps to detect the vertical or horizontal edges. We can use the following filters to detect different edges:

1	0	-1
1	0	-1
1	0	-1

Vertical

1	1	1
0	0	0
-1	-1	-1

Horizontal

Some of the commonly used filters are:

1	0	-1
2	0	-2
1	0	-1

Sobel filter

3	0	-3
10	0	-10
3	0	-3

Scharr filter

The Sobel filter puts a little bit more weight on the central pixels. Instead of using these filters, we can create our own as well and treat them as a parameter which the model will learn using backpropagation.

To present the formula for computing the output size of a convolutional layer. We have the following input:

- An image of dimensions $W_{in} \times H_{in}$.
- A filter of dimensions $K \times K$.
- Stride S and padding P.

The output activation map will have the following dimensions:

- $\mathbf{W_{out} = \frac{W_{in}-K+2P}{S} + 1}$
- $\mathbf{H_{out} = \frac{H_{in}-K+2P}{S} + 1}$

If the output dimensions are not integers, it means that we haven't set the stride S correctly.

We have two exceptional cases:

- When there is no padding at all, the output dimensions are $\left(\frac{W_{in}-K}{S} + 1, \frac{H_{in}-K}{S} + 1\right)$.
- In case we want to keep the size of the input unchanged after the convolution layer, we apply same padding where $W_{out} = W_{in}$ and $H_{out} = H_{in}$. If s=1, we set $p = \frac{K-1}{2}$.

Example:

Let's suppose that we have an input image of size 125x49, a filter of size 5x5, padding P=2 and stride S=2. Then the output dimensions are the following:

- $W_{out} = \frac{125-5+2*2}{2} = \frac{124}{2} = 62$
- $H_{out} = \frac{49-5+2*2}{2} = \frac{48}{2} = 24$

So,the output activation map will have dimensions $(62, 24)$.

In fact these involve different aspects of parameters in a CNN.

1. Parameter sharing, means one parameter may be shared by more than one input/connection. So this reduces total amount of independent parameters. Parameters shared are non-zero.

2. Sparsity of connections means that some parameters are simply missing (ie are zero), **nothing to do with sharing same non-zero parameter**. Parameters in this case are zero, ignored. That means that **not necessarily all (potential) inputs to a layer are actually connected to that layer**, only some of them, rest are ignored. Thus sparsity of connections.

Q) Weight sharing is an old-school technique for reducing the number of weights in a network that must be trained. It is exactly what it sounds like: the reuse of weights on nodes that are close to one another in some way.

Weight sharing in CNNs

A typical application of weight sharing is in convolutional neural networks. CNNs work by passing a filter over the image input. For the trivial example of a 4x4 image and a 2x2 filter with a stride size of 2, this would mean that the filter (which has four weights, one per pixel) is applied four times, making for 16 weights total. A typical application of weight sharing is to share the same weights across all four filters.

In this context weight sharing has the following effects:

- It reduces the number of weights that must be learned (from 16 to 4, in this case), which reduces model training time and cost.
- It makes feature search insensitive to feature location in the image.

So we reduce training cost at the cost of model flexibility. Weight sharing is for all intents and purposes a form of regularization. And as with other forms of regularization, it can actually *increase* the performance of the model, in certain datasets with high feature location variance, by decreasing variance more than they increase bias.

Q) Pooling Layer

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

$$(n_h - f + 1) / s \times (n_w - f + 1)/s \times n_c$$

where,

-> n_h - height of feature map

-> n_w - width of feature map

-> n_c - number of channels in the feature map

-> **f** - size of filter

-> **s** - stride length

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

Why to use Pooling Layers?

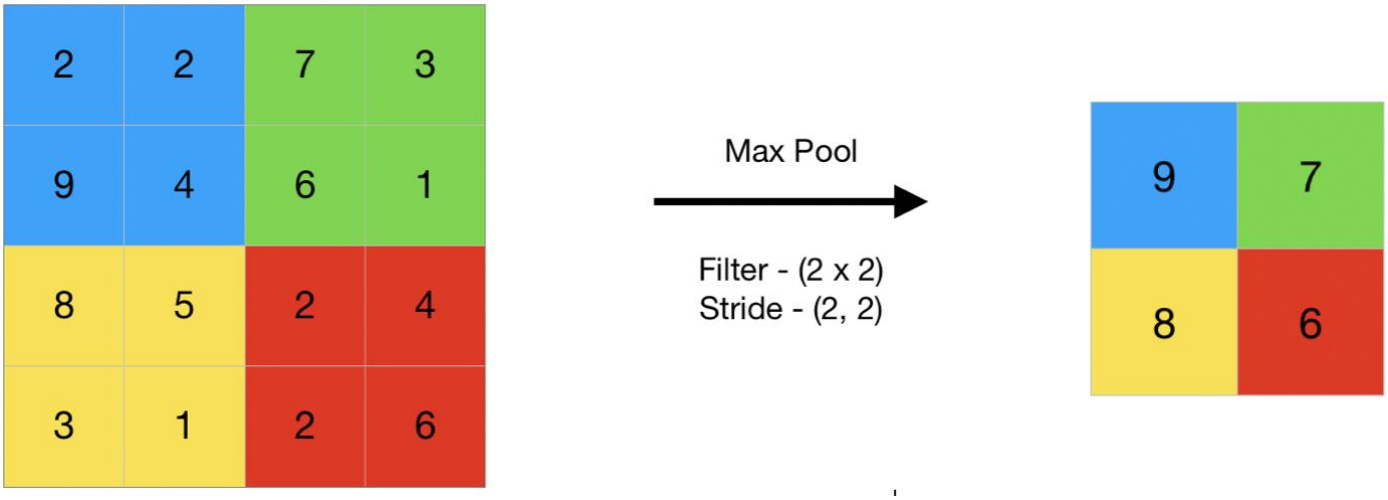
- Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn and the amount of computation performed in the network.

The pooling layer summarises the features present in a region of the feature map generated by a convolution layer. So, further operations are performed on summarised features instead of precisely positioned features generated by the convolution layer. This makes the model more robust to variations in the position of the features in the input image.

Types of Pooling Layers

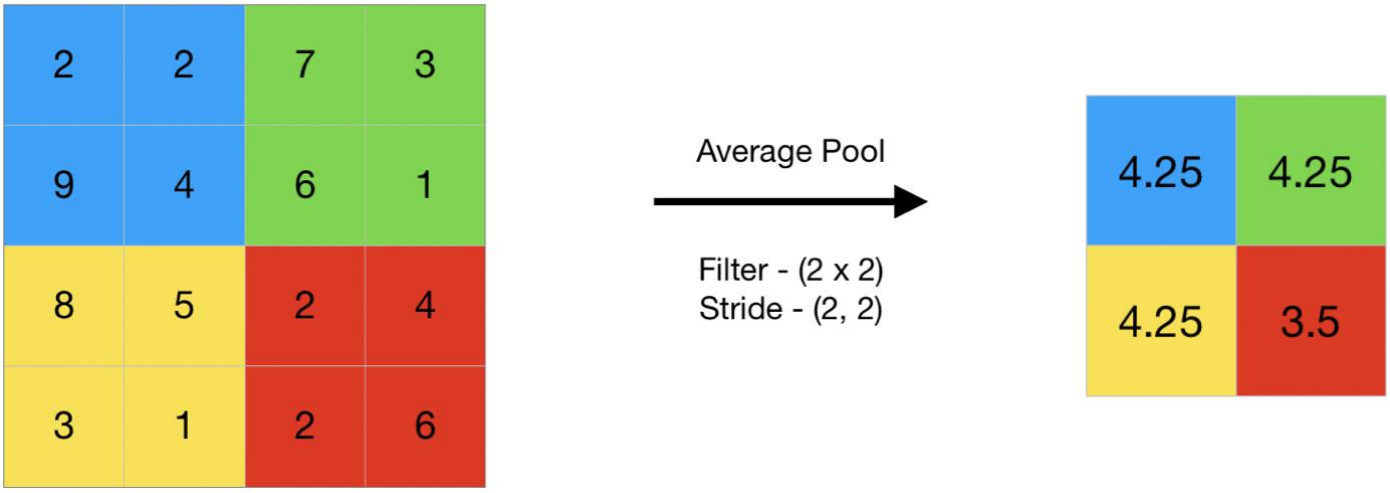
Max Pooling

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



Average Pooling

Average pooling computes the average of the elements present in the region of feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch.



In convolutional neural networks (CNNs), the pooling layer is a common type of layer that is typically added after convolutional layers. The pooling layer is used to reduce the spatial dimensions (i.e., the width and height) of the feature maps, while preserving the depth (i.e., the number of channels).

1. The pooling layer works by dividing the input feature map into a set of non-overlapping regions, called pooling regions. Each pooling region is then transformed into a single output value, which represents the presence of a particular feature in that region. The most common types of pooling operations are max pooling and average pooling.
2. In max pooling, the output value for each pooling region is simply the maximum value of the input values within that region. This has the effect of preserving the most salient features in each pooling region, while discarding less relevant information. Max pooling is often used in CNNs for object recognition tasks, as it helps to identify the most distinctive features of an object, such as its edges and corners.

3. In average pooling, the output value for each pooling region is the average of the input values within that region. This has the effect of preserving more information than max pooling, but may also dilute the most salient features. Average pooling is often used in CNNs for tasks such as image segmentation and object detection, where a more fine-grained representation of the input is required.

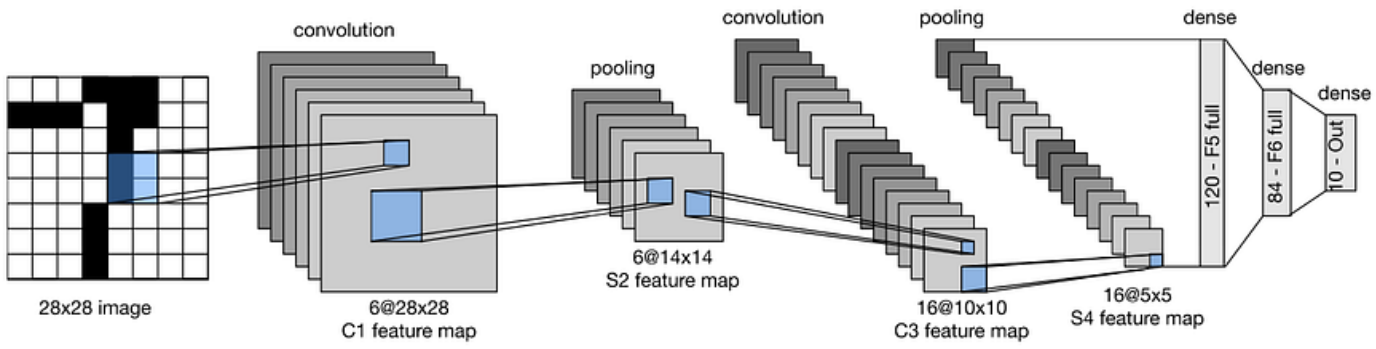
Pooling layers are typically used in conjunction with convolutional layers in a CNN, with each pooling layer reducing the spatial dimensions of the feature maps, while the convolutional layers extract increasingly complex features from the input. The resulting feature maps are then passed to a fully connected layer, which performs the final classification or regression task.

Q) LeNet-5 for handwritten character recognition

LeNet-5 is a convolutional neural network (CNN) architecture that was first proposed in 1998 for handwritten digit recognition. It is one of the earliest and most successful CNN architectures, and it has been used as a benchmark for many other CNN models.

LeNet-5 has a relatively simple architecture, consisting of the following layers:

- **Input layer:** This layer takes the input image, which is typically a 28x28 grayscale image of a handwritten digit.
- **Convolutional layer 1:** This layer extracts features from the input image using a set of convolution filters.
- **Pooling layer 1:** This layer reduces the dimensionality of the feature maps produced by the convolutional layer by downsampling them.
- **Convolutional layer 2:** This layer extracts more complex features from the feature maps produced by the first convolutional layer.
- **Pooling layer 2:** This layer further reduces the dimensionality of the feature maps produced by the second convolutional layer.
- **Fully connected layer:** This layer takes the flattened feature maps produced by the second pooling layer and produces a vector of outputs, one for each digit class.
- **Output layer:** This layer is a softmax layer that produces a probability distribution over the digit classes.



LeNet-5 can be trained to recognize handwritten digits by feeding it a dataset of labeled handwritten digit images. The network learns to extract features from the images that are discriminative for the different digit classes. Once the network is trained, it can be used to predict the digit class of a new handwritten digit image.

LeNet-5 has been shown to achieve high accuracy on the MNIST handwritten digit dataset, with an accuracy of over 99%. It has also been used to successfully recognize handwritten characters in other datasets, such as the USPS zip code dataset and the IAM handwritten text database.

While LeNet-5 is a relatively simple CNN architecture, it is still a powerful tool for handwritten character recognition. It is also a good architecture for beginners to learn about CNNs, as it is relatively easy to implement and train.

Q) How do we train a convolutional neural network ?

To train a convolutional neural network (CNN), you need to follow these steps:

1. **Collect a dataset of labeled images.** The dataset should be as large and diverse as possible, in order to train the network to generalize well to new images.
2. **Preprocess the images.** This may involve resizing the images, normalizing the pixel values, or converting the images to grayscale.
3. **Design the CNN architecture.** This involves choosing the number and type of layers, as well as the hyperparameters for each layer.
4. **Initialize the weights of the CNN.** This is typically done by randomly initializing the weights.
5. **Choose a loss function and optimizer.** The loss function measures how well the network is performing on the training data, and the optimizer updates the weights of the network to minimize the loss function.
6. **Train the CNN.** This involves feeding the training data to the network and updating the weights of the network using the optimizer.
7. **Evaluate the CNN.** Once the CNN is trained, you should evaluate its performance on a held-out test dataset. This will give you an idea of how well the network will generalize to new images.

Here is a more detailed explanation of each step:

Collect a dataset of labeled images: The dataset should be as large and diverse as possible, in order to train the network to generalize well to new images. The images should be labeled with the correct class, such as "cat", "dog", or "car". You can find pre-labeled datasets online, or you can collect your own dataset.

Preprocess the images: This may involve resizing the images to a consistent size, normalizing the pixel values, or converting the images to grayscale. Preprocessing the images helps to improve the performance of the CNN and makes it easier to train.

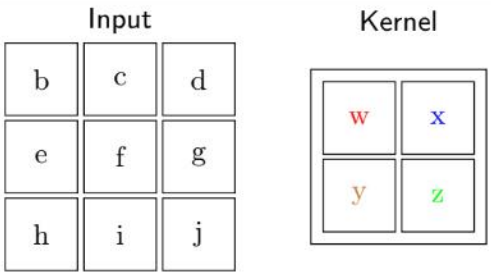
Design the CNN architecture: This involves choosing the number and type of layers, as well as the hyperparameters for each layer. The most common type of layer in a CNN is the convolutional layer. Convolutional layers extract features from the input images. Other common types of layers include pooling layers, fully connected layers, and output layers.

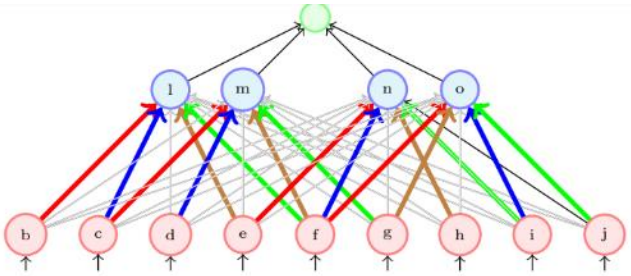
Initialize the weights of the CNN: This is typically done by randomly initializing the weights. However, there are also a number of techniques for initializing the weights in a way that improves the performance of the CNN.

Choose a loss function and optimizer: The loss function measures how well the network is performing on the training data. Common loss functions for CNNs include cross-entropy loss and mean squared error loss. The optimizer updates the weights of the network to minimize the loss function. Common optimizers for CNNs include Adam and stochastic gradient descent (SGD).

Train the CNN: This involves feeding the training data to the network and updating the weights of the network using the optimizer. The training process is typically repeated for a number of epochs, until the network converges to a good solution.

Evaluate the CNN: Once the CNN is trained, you should evaluate its performance on a held-out test dataset. This will give you an idea of how well the network will generalize to new images.





- A CNN can be implemented as a feedforward neural network
- wherein only a few weights(in color) are active
- the rest of the weights (in gray) are zero

- We can thus train a convolution neural network using backpropagation by thinking of it as a feedforward neural network with sparse connections

Q) AlexNet, ZF-Net, VGGNet, GoogLeNet and ResNet

AlexNet, ZF-Net, VGGNet, GoogLeNet, and ResNet are all convolutional neural networks (CNNs) that have achieved state-of-the-art results in image classification tasks.

AlexNet was the first CNN to win the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. It has a relatively simple architecture, consisting of five convolutional layers followed by three fully connected layers. AlexNet was trained on a massive dataset of over 1.2 million images, and it achieved a top-5 error rate of 15.3% on the ILSVRC test set.

ZF-Net was inspired by AlexNet, but it introduced several improvements, such as the use of smaller convolutional filters and a deeper network architecture. ZF-Net achieved a top-5 error rate of 14.8% on the ILSVRC 2013 test set.

VGGNet is a family of CNNs that were developed by the University of Oxford. VGGNets are characterized by their use of very small convolutional filters and a very deep network architecture. VGGNet-16 achieved a top-5 error rate of 13.6% on the ILSVRC 2014 test set.

GoogLeNet is a CNN that was developed by Google. It is characterized by its use of the Inception module, which allows the network to learn multiple levels of abstraction in parallel. GoogLeNet achieved a top-5 error rate of 6.7% on the ILSVRC 2014 test set, which was a significant improvement over the previous state-of-the-art.

ResNet is a CNN that was developed by Microsoft. It is characterized by its use of residual blocks, which allow the network to learn deeper representations of the data without overfitting. ResNets have achieved state-of-the-art results on a wide range of image classification tasks, including the ILSVRC and COCO benchmarks.

All of these CNNs have played a significant role in the development of deep learning and computer vision. They have demonstrated the power of CNNs to learn complex representations of data and to solve challenging problems such as image classification.

Year	CNN	Developed by	Place	Top-5 error rate	No. of parameters
1998	LeNet(8)	Yann LeCun et al			60 thousand
2012	AlexNet(7)	Alex Krizhevsky, Geoffrey Hinton, Ilya Sutskever	1st	15.3%	60 million
2013	ZFNet()	Matthew Zeiler and Rob Fergus	1st	14.8%	
2014	GoogLeNet(19)	Google	1st	6.67%	4 million
2014	VGG Net(16)	Simonyan, Zisserman	2nd	7.3%	138 million
2015	ResNet(152)	Kaiming He	1st	3.6%	

Feature	AlexNet	ZF-Net	VGGNet	GoogLeNet	ResNet
Year	2012	2013	2014	2014	2015
Top-5 error on ILSVRC test set	15.3%	14.8%	13.6%	6.7%	3.57%
Depth	8	8	16	22	50
Number of parameters	61 million	58 million	138 million	6 million	25.6 million
Strengths	Simple architecture, effective for image classification	Deeper architecture, smaller convolutional filters	Very deep architecture, very small convolutional filters	Efficient architecture, learns multiple levels of abstraction in parallel	State-of-the-art results on a wide range of image classification tasks
Weaknesses	Relatively large number of parameters, difficult to train	Deeper architecture, more difficult to train	Very deep architecture, difficult to train	Requires more computational resources to train	More difficult to train than other CNNs

Q) What are Filters/Kernels?

- A filter provides a measure for how close a patch or a region of the input resembles a feature. A feature may be any prominent aspect – a vertical edge, a horizontal edge, an arch, a diagonal, etc.
- A filter acts as a single template or pattern, which, when convolved across the input, finds similarities between the stored template & different locations/regions in the input image.
- Let us consider an example of detecting a vertical edge in the input image.
- Each column of the 4×4 output matrix looks at exactly three columns & three rows (the coloured boxes show the output of the filter as it moves over the input image). The values in the output matrix represent the change in the intensity along the horizontal direction w.r.t the columns in the input image.
- The output image has the value 0 in the 1st & last column. It means there is no change in intensity in the first three columns & the previous three columns of the input image. On the other hand, the output

is 30 in the 2nd & 3rd column, indicating a change in the intensity of the corresponding columns of the input image.

Dimensions of the Convolved Output?

If the input image size is 'n x n' & filter size is 'f x f ', then after convolution, the size of the output image is: (Size of input image – filter size + 1)x (Size of input image – filter size + 1).

How is the Filter Size Decided?

By convention, the value of 'f,' i.e. filter size, is usually odd in computer vision. This might be because of 2 reasons:

- If the value of 'f' is even, we may need asymmetric padding (Please refer above eqn. 1). Let us say that the size of the filter i.e. 'f' is 6. Then by using equation 1, we get a padding size of 2.5, which does not make sense.
- The 2nd reason for choosing an odd size filter such as a 3×3 or a 5×5 filter is we get a central position & at times it is nice to have a distinguisher.

Multiple Filters for Multiple Features

- We can use multiple filters to detect various features simultaneously. Let us consider the following example in which we see vertical edge & curve in the input RGB image. We will have to use two different filters for this task, and the output image will thus have two feature maps.

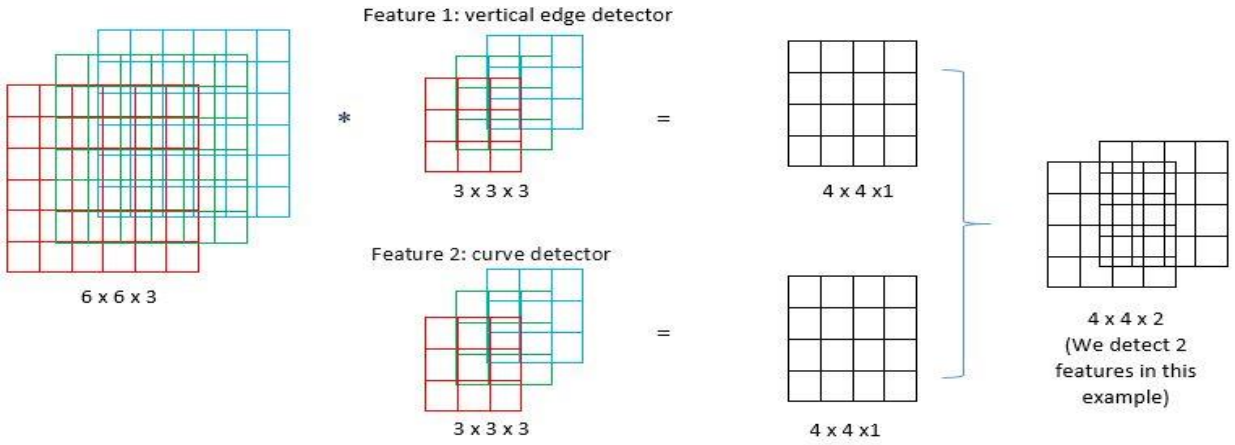


Fig: Convolution using multiple filters

- Let us understand the dimensions mathematically,

$$(n \times n \times n_c) * (f \times f \times n_c) * (f \times f \times n_c) \longrightarrow (n-f+1) \times (n-f+1) \times \text{no. of filters}$$

Here are some examples of filters that can be used in CNNs:

- **Edge detectors:** These filters are designed to detect edges in images. They can be used to extract features such as horizontal edges, vertical edges, and diagonal edges.
- **Corner detectors:** These filters are designed to detect corners in images. They can be used to extract features such as right angles, acute angles, and obtuse angles.
- **Texture detectors:** These filters are designed to detect textures in images. They can be used to extract features such as bumps, grooves, and patterns.

Introduction to RNN, RNN Code

……

1. Introduction to RNN

1.1. Sequence Learning Problems

Sequence learning problems are different from other machine learning problems in two key ways:

- The inputs to the model are not of a fixed size.
- The inputs to the model are dependent on each other.

Examples of sequence learning problems include:

- Auto completion
- Part-of-speech tagging
- Sentiment analysis
- Video classification

Recurrent neural networks (RNNs) are a type of neural network that are well-suited for solving sequence learning problems. RNNs work by maintaining a hidden state that is updated at each time step. The hidden state captures the information from the previous inputs, which allows the model to predict the next output.

Example:

Consider the task of auto completion. Given a sequence of characters, we want to predict the next character. For example, given the sequence "d", we want to predict the next character, which is "e".

An RNN would solve this problem by maintaining a hidden state. The hidden state would be initialized with the information from the first input character, "d". Then, at the next time step, the RNN would take the current input character, "e", and the hidden state as input and produce a prediction for the next character. The hidden state would then be updated with the new information.

This process would be repeated until the end of the sequence. At the end of the sequence, the RNN would output the final prediction.

Advantages of RNNs for sequence learning problems:

- RNNs can handle inputs of any length.
- RNNs can learn long-term dependencies between the inputs in a sequence.

Disadvantages of RNNs:

- RNNs can be difficult to train.
- RNNs can be susceptible to vanishing and exploding gradients.

RNNs are a powerful tool for solving sequence learning problems. They have been used to achieve state-of-the-art results in many tasks, such as machine translation, text summarization, and speech recognition.

1.2. Recurrent Neural Networks

Recurrent neural networks (RNNs) are a type of neural network that are well-suited for solving sequence learning problems. RNNs work by maintaining a hidden state that is updated at each time step. The hidden state captures the information from the previous inputs, which allows the model to predict the next output.

RNNs have several advantages over other types of neural networks for sequence learning problems:

- RNNs can handle inputs of any length.
- RNNs can learn long-term dependencies between the inputs in a sequence.
- RNNs can be used to solve a wide variety of sequence learning problems, such as natural language processing, machine translation, and speech recognition.

How to model sequence learning problems with RNNs:

To model a sequence learning problem with an RNN, we first need to define the function that the RNN will compute at each time step. The function should take as input the current input and the hidden state from the previous time step, and output the next hidden state and the prediction for the current time step.

Once we have defined the function, we can train the RNN using backpropagation through time (BPTT). BPTT is a specialized training algorithm for RNNs that allows us to train the network even though it has recurrent connections.

Examples of sequence learning problems that can be solved with RNNs:

- Natural language processing: tasks such as part-of-speech tagging, named entity recognition, and machine translation.
- Speech recognition: tasks such as transcribing audio to text and generating text from speech.
- Video processing: tasks such as video classification and captioning.
- Time series analysis: tasks such as forecasting and anomaly detection.

How RNNs solve the problems on the wishlist:

- **The same function is executed at every time step:** This is achieved by sharing the same network parameters at every time step.
- **The model can handle inputs of arbitrary length:** This is because the RNN can keep updating its hidden state based on the previous inputs, regardless of the length of the input sequence.
- **The model can learn long-term dependencies between the inputs in a sequence:** This is because the RNN's hidden state can capture information from the previous inputs, even if they are many time steps ago.

1.3. Backpropagation through time (BPTT)

BPTT is a training algorithm for recurrent neural networks (RNNs). It is used to compute the gradients of the loss function with respect to the RNN's parameters, which are then used to update the parameters using gradient descent.

To compute the gradients using BPTT, we need to first compute the explicit derivative of the loss function with respect to the RNN's parameters. This is done by treating all of the other inputs to the RNN as constants.

However, RNNs also have implicit dependencies, which means that the output of the RNN at a given time step depends on the outputs of the RNN at previous time steps. This makes it difficult to compute the gradients using the explicit derivative alone.

To address this problem, BPTT uses the chain rule to recursively compute the implicit derivatives of the loss function with respect to the RNN's parameters. This involves summing over all of the paths from the loss function to each parameter, where each path is a sequence of RNN outputs and weights.

BPTT can be computationally expensive, but it is a powerful tool for training RNNs. It has been used to achieve state-of-the-art results on a variety of sequence learning tasks, such as natural language processing, machine translation, and speech recognition.

Example:

Consider the following RNN, which is used to predict the next character in a sequence:

s_t = W * s_{t-1} + x_t

y_t = softmax(V * s_t)

where:

- s_t is the hidden state of the RNN at time step t
- x_t is the input at time step t
- y_t is the output at time step t
- W and V are the RNN's parameters

Suppose we want to compute the gradient of the loss function with respect to the weight W. Using BPTT, we can do this as follows:

```
# Compute the explicit derivative

d_loss_dw = s_{t-1}

# Compute the implicit derivative

for i in range(t - 2, -1, -1):
    d_loss_dw += s_i * W^T * d_loss_dw
```

The implicit derivative is computed by recursively summing over all of the paths from the loss function to the weight W. Each path is a sequence of RNN outputs and weights, and the derivative for each path is computed using the chain rule.

Once we have computed the explicit and implicit derivatives, we can simply sum them together to get the total derivative of the loss function with respect to the weight W. This derivative can then be used to update the weight W using gradient descent.

Challenges:

BPTT can be computationally expensive, especially for RNNs with many layers or long sequences. However, there are a number of techniques that can be used to improve the efficiency of BPTT, such as truncated BPTT and gradient clipping.

Another challenge with BPTT is that it can be sensitive to the initialization of the RNN's parameters. If the parameters are not initialized carefully, the RNN may not learn to perform the desired task.

1.4. The problem of Exploding and Vanishing Gradients

The problem of vanishing and exploding gradients is a common problem when training recurrent neural networks (RNNs). It occurs because the gradients of the loss function with respect to the RNN's parameters can become very small or very large as the backpropagation algorithm progresses. This can make it difficult for the RNN to learn to perform the desired task.

There are two main reasons why vanishing and exploding gradients can occur:

1. **Bounded activations:** RNNs typically use bounded activation functions, such as the sigmoid or tanh function. This means that the derivatives of the activation functions are also bounded. This can lead to vanishing gradients, especially if the RNN has a large number of layers.
2. **Product of weights:** The gradients of the loss function with respect to the RNN's parameters are computed by multiplying together the gradients of the activations at each layer. This means that if the gradients of the activations are small or large, the gradients of the parameters will also be small or large.

Vanishing and exploding gradients can be a major problem for training RNNs. If the gradients vanish, the RNN will not be able to learn to perform the desired task. If the gradients explode, the RNN will learn very quickly, but it will likely overfit the training data and not generalize well to new data.

There are a number of techniques that can be used to address the problem of vanishing and exploding gradients, such as:

- **Truncated backpropagation:** Truncated backpropagation only backpropagates the gradients through a fixed number of layers. This helps to prevent the gradients from vanishing.
- **Gradient clipping:** Gradient clipping normalizes the gradients so that their magnitude does not exceed a certain threshold. This helps to prevent the gradients from exploding.
- **Weight initialization:** The way that the RNN's parameters are initialized can have a big impact on the problem of vanishing and exploding gradients. It is important to initialize the parameters in a way that prevents the gradients from becoming too small or too large.

Truncated backpropagation is a common technique used to address the problem of vanishing and exploding gradients in recurrent neural networks (RNNs). However, it is not the only solution.

Another common solution is to use **gated recurrent units (GRUs)** or **long short-term memory (LSTM) cells**. These units are specifically designed to deal with the problem of vanishing and exploding gradients.

GRUs and LSTMs work by using gates to control the flow of information through the RNN. This allows the RNN to learn long-term dependencies in the data without the problem of vanishing gradients.

GRUs and LSTMs have been shown to be very effective for training RNNs on a variety of tasks, such as natural language processing, machine translation, and speech recognition.

1.5. Long Short Term Memory(LSTM) and Gated Recurrent Units(GRUs)

Long Short Term Memory (LSTM) and Gated Recurrent Units (GRU) are two types of recurrent neural networks (RNNs) that are specifically designed to learn long-term dependencies in sequential data. They are both widely used in a variety of tasks, including natural language processing, machine translation, speech recognition, and time series forecasting.

Both LSTMs and GRUs use a gating mechanism to control the flow of information through the network. This allows them to learn which parts of the input sequence are important to remember and which parts can be forgotten.

LSTM Architecture

An LSTM cell has three gates: an input gate, a forget gate, and an output gate.

- The input gate controls how much of the current input is added to the cell state.
- The forget gate controls how much of the previous cell state is forgotten.
- The output gate controls how much of the cell state is output to the next cell in the sequence.

The LSTM cell also has a cell state, which is a long-term memory that stores information about the previous inputs. The cell state is updated at each time step based on the input gate, forget gate, and output gate.

GRU Architecture

A GRU cell has two gates: a reset gate and an update gate.

- The reset gate controls how much of the previous cell state is forgotten.
- The update gate controls how much of the previous cell state is combined with the current input to form the new cell state.

The GRU cell does not have a separate output gate. Instead, the output of the GRU cell is simply the updated cell state.

Comparison of LSTMs and GRUs

LSTMs and GRUs are very similar in terms of their performance on most tasks. However, there are a few key differences between the two architectures:

- LSTMs have more gates and parameters than GRUs, which makes them more complex and computationally expensive to train.
- GRUs are generally faster to train and deploy than LSTMs.
- GRUs are more robust to noise in the input data than LSTMs.

Which one to choose?

The best choice of architecture for a particular task depends on a number of factors, including the size and complexity of the dataset, the available computing resources, and the specific requirements of the task.

In general, LSTMs are recommended for tasks where the input sequences are very long or complex, or where the task requires a high degree of accuracy. GRUs are a good choice for tasks where the input sequences are shorter or less complex, or where speed and efficiency are important considerations.

2. RNN Code

```
import keras

# Define the model

model = keras.Sequential([

    keras.layers.LSTM(128, input_shape=(10, 256)),

    keras.layers.Dense(64, activation='relu'),

    keras.layers.Dense(1, activation='sigmoid')

])

# Compile the model

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model

model.fit(x_train, y_train, epochs=10)

# Evaluate the model

model.evaluate(x_test, y_test)

# Make predictions

predictions = model.predict(x_test)
```

This code defines a simple RNN model with one LSTM layer, one dense layer, and one output layer. The LSTM layer has 128 hidden units, and the dense layer has 64 hidden units. The output layer has a single unit, and it uses the sigmoid activation function to produce a probability score.

The model is compiled using the binary cross-entropy loss function and the Adam optimizer. The model is then trained on the training data for 10 epochs.

Once the model is trained, it can be evaluated on the test data to assess its performance. The model can also be used to make predictions on new data.

Here is an example of how to use the model to make predictions:

```
# Make predictions on a new data sample

x_new = [[[0.1, 0.2], [0.3, 0.4], [0.5, 0.6]]]

# Get the prediction

prediction = model.predict(x_new)

# Print the prediction

print(prediction)
```

This code will print the prediction for the new data sample, which is a probability score between 0 and 1. A probability score closer to 1 means that the model is more confident in the prediction.

This is just a simple example of RNN code, and there are many other ways to implement RNNs in Python. For more complex tasks, you may need to use a different RNN architecture or add additional layers to the model.

Chapter-III

PyTorch Tensors: Deep Learning with PyTorch, CNN in PyTorch

……..

PyTorch is **an optimized Deep Learning tensor library based on Python and Torch and is mainly used for applications using GPUs and CPUs**. PyTorch is favored over other Deep Learning frameworks like TensorFlow and Keras since it uses dynamic computation graphs and is completely Pythonic.

Why is PyTorch used for deep learning?

It is open source, and is based on the popular Torch library. PyTorch is designed to provide good flexibility and high speeds for deep neural network implementation. PyTorch is different from other deep learning frameworks in that it uses dynamic computation graphs.

3.1.Features

The major features of PyTorch are mentioned below −

Easy Interface − PyTorch offers easy to use API; hence it is considered to be very simple to operate and runs on Python. The code execution in this framework is quite easy.

Python usage − This library is considered to be Pythonic which smoothly integrates with the Python data science stack. Thus, it can leverage all the services and functionalities offered by the Python environment.

Computational graphs − PyTorch provides an excellent platform which offers dynamic computational graphs. Thus a user can change them during runtime. This is highly useful when a developer has no idea of how much memory is required for creating a neural network model.

PyTorch is known for having three levels of abstraction as given below −

- Tensor − Imperative n-dimensional array which runs on GPU.
- Variable − Node in computational graph. This stores data and gradient.
- Module − Neural network layer which will store state or learnable weights.

Advantages of PyTorch

The following are the advantages of PyTorch −

- It is easy to debug and understand the code.
- It includes many layers as Torch.
- It includes lot of loss functions.
- It can be considered as NumPy extension to GPUs.
- It allows building networks whose structure is dependent on computation itself.

3.2.TensorFlow vs. PyTorch

We shall look into the major differences between TensorFlow and PyTorch below –

PyTorch	TensorFlow
PyTorch is closely related to the lua-based Torch framework which is actively used in Facebook.	TensorFlow is developed by Google Brain and actively used at Google.
PyTorch is relatively new compared to other competitive technologies.	TensorFlow is not new and is considered as a to-go tool by many researchers and industry professionals.
PyTorch includes everything in imperative and dynamic manner.	TensorFlow includes static and dynamic graphs as a combination.
Computation graph in PyTorch is defined during runtime.	TensorFlow do not include any run time option.
PyTorch includes deployment featured for mobile and embedded frameworks.	TensorFlow works better for embedded frameworks.

3.3. Deep Learning with PyTorch

PyTorch elements are the building blocks of PyTorch models. These elements are:

- **Model:** A model is a representation of a machine learning algorithm. It is made up of layers and parameters.
- **Layer:** A layer is a unit of computation in a neural network. It performs a specific mathematical operation on the input data.
- **Optimizer:** An optimizer is an algorithm that updates the model's parameters during training.
- **Loss:** A loss function measures the error between the model's predictions and the ground truth labels.

Models are created using the torch.nn.Module class. Layers are created using the different classes provided by the torch.nn module. For example, to create a linear layer, you would use the torch.nn.Linear class.

Optimizers are created using the classes provided by the torch.optim module. For example, to create an Adam optimizer, you would use the torch.optim.Adam class.

Loss functions are created using the classes provided by the torch.nn.functional module. For example, to create a mean squared error loss function, you would use the torch.nn.functional.mse_loss function.

Once you have created the model, layers, optimizer, and loss function, you can train the model using the following steps:

1. Forward pass: The input data is passed through the model to produce predictions.
2. Loss calculation: The loss function is used to calculate the error between the predictions and the ground truth labels.

3. Backward pass: The gradients of the loss function with respect to the model's parameters are calculated.
4. Optimizer step: The optimizer uses the gradients to update the model's parameters.

This process is repeated for a number of epochs until the model converges and achieves the desired performance.

Here is a simple example of a PyTorch model:

```
import torch

class LinearModel(torch.nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearModel, self).__init__()

        self.linear = torch.nn.Linear(input_size, output_size)

    def forward(self, x):
        output = self.linear(x)

        return output

# Create the model
model = LinearModel(10, 1)

# Create the optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# Create the loss function
loss_fn = torch.nn.functional.mse_loss

# Train the model
...
```

This code defines a simple linear model with one input layer and one output layer. The model is trained using the Adam optimizer and the mean squared error loss function.

3.4. RNN with PyTorch

PyTorch is a Python library for machine learning. It is based on Torch, a scientific computing framework for Lua. PyTorch is popular for its flexibility and ease of use. It is also widely used in research and industry.

To implement RNNs in PyTorch, you can use the **torch.nn.RNN** module. This module provides a number of different RNN architectures, including LSTM and GRU.

Here is a simple example of how to implement an LSTM in PyTorch:

This code defines a simple LSTM model with one input layer, one LSTM layer, and one output layer. The LSTM layer has 128 hidden units.

The model is trained using the model.fit() method. The model can then be used to make predictions on new data using the model.predict() method.

For more complex tasks, you may need to use a different RNN architecture or add additional layers to the model. You can also use PyTorch to implement bidirectional RNNs, stacked RNNs, and other advanced RNN architectures.

PyTorch also provides a number of tools for training and evaluating RNNs, such as the torch.optim module and the torch.nn.functional module.

```
import torch

class LSTM(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super(LSTM, self).__init__()

        self.lstm = torch.nn.LSTM(input_size, hidden_size, num_layers)

    def forward(self, x):
        output, (hn, cn) = self.lstm(x)

        return output

# Define the model
model = LSTM(10, 128, 1)

# Train the model
...

# Make predictions
...
```

3.4. CNN with PyTorch

Convolutional neural networks (CNNs) are a type of neural network that are specifically designed to work with image data. CNNs are able to learn spatial features in images, which makes them very effective for tasks such as image classification, object detection, and image segmentation.

PyTorch is a popular Python library for machine learning. It provides a number of features that make it easy to build, train, and deploy CNNs.

To implement a CNN in PyTorch, you can use the torch.nn.Conv2d layer. This layer performs a convolution operation on the input data. The convolution operation is a mathematical operation that extracts features from the input data.

CNNs also use pooling layers to reduce the spatial size of the input data. This helps to reduce the number of parameters in the network and makes it more efficient to train.

Here is a simple example of a CNN in PyTorch:

```
import torch

class CNN(torch.nn.Module):
    def __init__(self):
        super(CNN, self).__init__()


        # Define the convolutional layers
        self.conv1 = torch.nn.Conv2d(3, 6, 5)
        self.conv2 = torch.nn.Conv2d(6, 16, 5)


        # Define the pooling layers
        self.pool1 = torch.nn.MaxPool2d(2, 2)
        self.pool2 = torch.nn.MaxPool2d(2, 2)
```

```
        # Define the fully connected layers
        self.fc1 = torch.nn.Linear(16 * 5 * 5, 120)
        self.fc2 = torch.nn.Linear(120, 84)
        self.fc3 = torch.nn.Linear(84, 10)

    def forward(self, x):
        # Pass the input data through the convolutional layers
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)

        # Flatten the output of the convolutional layers
        x = x.view(-1, 16 * 5 * 5)

        # Pass the flattened output through the fully connected layers
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x
# Create the model
model = CNN()
# Train the model
...
```

This code defines a simple CNN with two convolutional layers, two pooling layers, and three fully connected layers. The convolutional layers have 6 and 16 filters, respectively. The pooling layers have a kernel size of 2x2 and a stride of 2. The fully connected layers have 120, 84, and 10 units, respectively.

The model is trained using the model.fit() method. The model can then be used to make predictions on new data using the model.predict() method.

For more complex tasks, you may need to use a different CNN architecture or add additional layers to the model. You can also use PyTorch to implement other types of neural networks, such as recurrent neural networks (RNNs) and long short-term memory (LSTM) networks.