1. CNN Architecture:

A Convolutional Neural Network (CNN) is a type of Deep Learning neural network architecture commonly used in Computer Vision. Computer vision is a field of Artificial Intelligence that enables a computer to understand and interpret the image or visual data.

When it comes to Machine Learning, Artificial Neural Networks perform really well. Neural Networks are used in various datasets like images, audio, and text. Different types of Neural Networks are used for different purposes, for example for predicting the sequence of words we use Recurrent Neural Networks more precisely an LSTM, similarly for image classification we use Convolution Neural networks.

CNN is very useful as it minimises human effort by automatically detecting the features. For example, for apples and mangoes, it would automatically detect the distinct features of each class on its own.

CNNs are a class of Deep Neural Networks that can recognize and classify particular features from images and are widely used for analysing visual images. Their applications range from image and video recognition, image classification, medical image analysis, computer vision and natural language processing.

Basic Architecture:

There are two main parts in a CNN architecture as shown in Fig1.a are i) feature learning and classification.
Convolutional Neural Network consists of multiple layers like the input layer, Convolutional layer, Pooling layer, and fully connected layers .

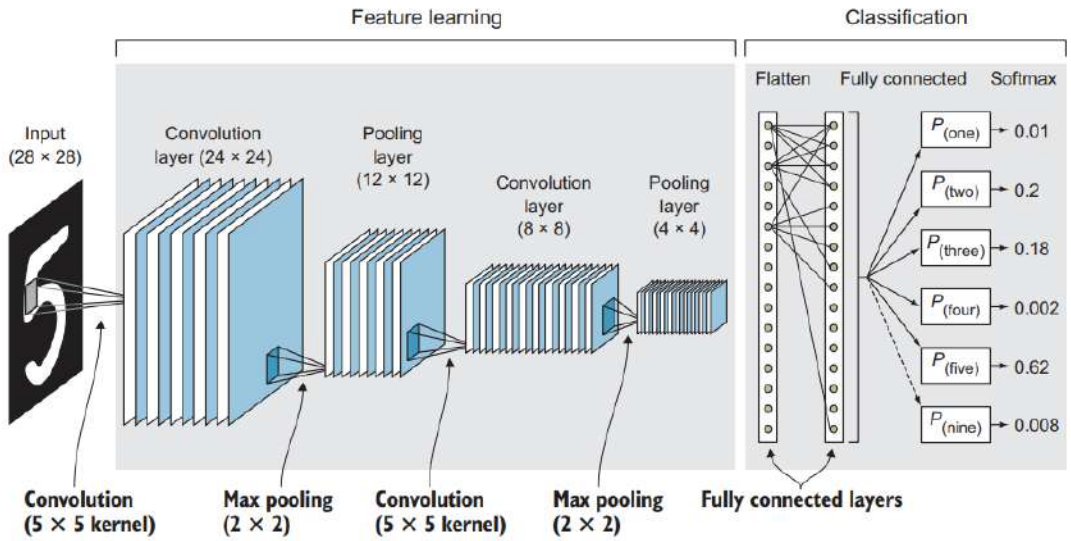


Fig 1.a The CNN architecture

a. Input layer:

The 2D images need to preprocess before feeding them. The preprocessing results which can understand and interpret by network. Consider figure 1.a, the image fed is 28 pixels wide ×

28 pixels high. This image is seen by the computer as a 28 × 28 matrix, with pixel values ranging from 0 to 255 (0 for black, 255 for white, and the range in between for grayscale as shown in Fig 1.b.

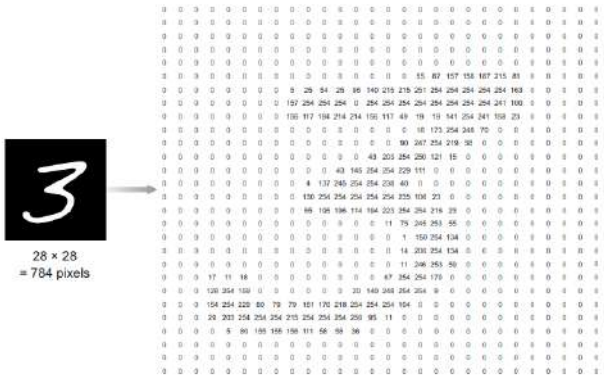


Figure 1.b The image as a 28 × 28 matrix of pixel values ranging from 0 to 255.

***Image flattening*:** MLPs takes input 1D vectors with dimensions (1, n), they cannot take a raw 2D image matrix with dimensions (x, y). To fit the image in the input layer, it needs to transform image into one large vector with the dimensions (1, n) that contains all the pixel values in the image.

example, the total number (n) of pixels in this image is 28 × 28 = 784. In order to feed this image to our network, we need to flatten the (28 × 28) matrix into one long vector with dimensions (1, 784).

2D image looks as input vector: x = [row1, row2, row3, ..., row28]

Image Flattening

Results as 1D large vector with 784 nodes: x1, x2, …, x784.

b. Hidden layers

The hidden layers in the network provide a basic building block to transform the data (input layer or the output of the previously hidden layer). Most of the commonly used hidden layers (not all) follow a pattern. It begins with applying a function to its input, moving onto pooling, normalization, and finally applying the activation before it can be fed as input to the next layer. Thus, each layer can be decomposed into the following 4 sub-functions. As shown in figure below

- **Layer function:** Basic transforming function such as convolutional or fully connected layer.
- **Pooling:** Used to change the spatial size of the feature map either increasing (up-sampling) or decreasing (most common) it. For example maxpooling, average pooling, and unpooling.
- **Normalization:** This subfunction normalizes the data to have zero mean and unit variance. This helps in coping up with problems such as vanishing gradient, internal covariate shift, etc. The two most common normalization techniques used are local response normalization and batch normalization.
- **Activation:** Applies non-linearity and bounds the output from getting too high or too low.

i) Convolutional Layer:

These layers are applied to 2D (and 3D) input feature maps. The trainable weights are a 2D (or 3D) kernel/filter that moves across the input feature map, generating dot products with the overlapping region of the input feature map.

Following are the 3 parameters used to define a convolutional layer:
Kernel Size K: The size of the sliding kernel or filter.
Stride Length S: Defines how much is the kernel slide before the dot product is carried out to generate the output pixel
Padding P: The frame size of zeros inserted around the input feature map.

If we have an input of size W x W x D and Dout number of kernels with a spatial size of F with stride S and amount of padding P, then the size of output volume can be determined by the following formula:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$

Filters or Kernels: The number of convolutional filters or kernels in each layer. This represents the depth of its output.

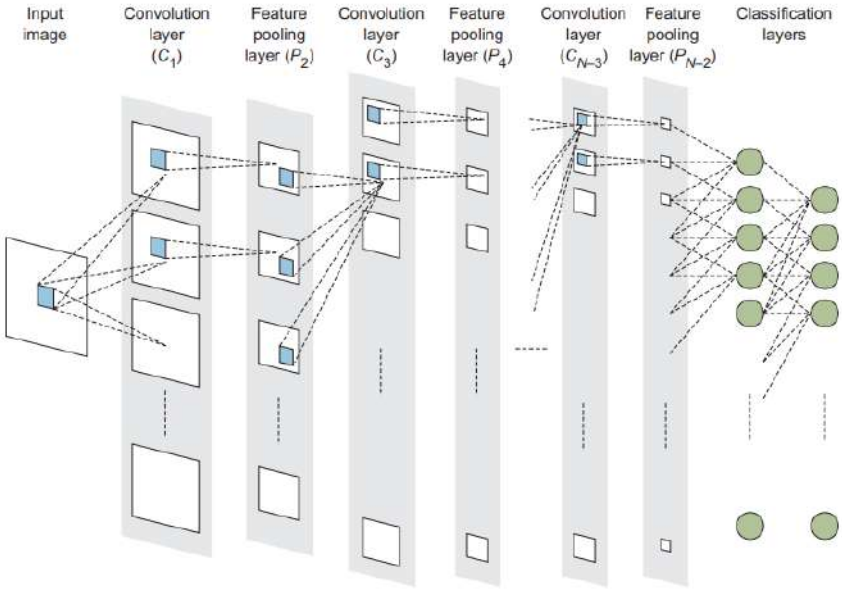


Figure 1.c Representation of the CNN layers that shows the number-of-kernels idea

By sliding the convolutional filter or *kernel* over the input image, the network breaks the image into little chunks and processes those chunks individually to assemble the modified image, a feature map as shown in figure 1.c and 1.e .

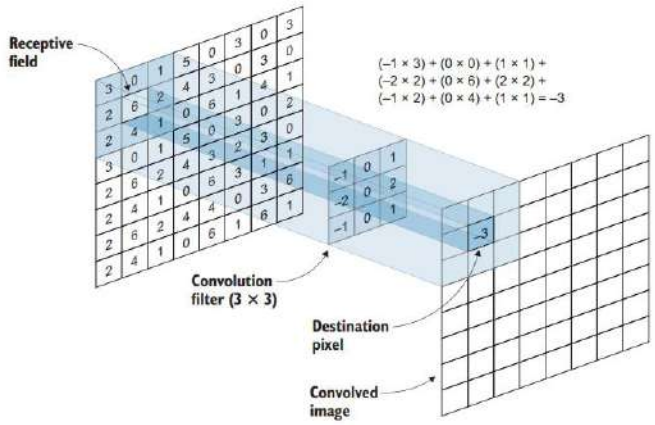


Figure 1.d. A 3 × 3 convolutional filter is sliding over the input image.

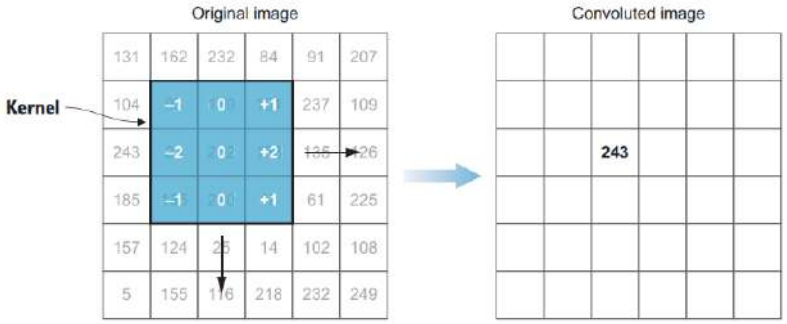


Figure 1.e multiplying each pixel in the receptive field by the corresponding pixel in the convolution filter and summing them gives the value of the center pixel in the new image.

The snippet to create convolutional layers in Keras :

```
from keras.layers import Conv2D

model.add(Conv2D(filters=16, kernel_size=2, strides='1', padding='same',
          activation='relu'))
```

Stride* and *Padding: The two hyper parameters are used together, because they both control the shape of the output of a Convolutional layer.

Strides—The amount by which the filter slides over the image.

For example, to slide the convolution filter one pixel at a time, the strides value is 1. If we want to jump two pixels at a time, the strides value is 2. Strides of 3 or more are uncommon and rare in practice. Jumping pixels produces smaller output volumes spatially. Strides of 1 will make the output image roughly the same height and width of the input image, while strides of 2 will make the output image roughly half of the input image size.

Padding— is also called as zero-padding because we add zeros around the border of an image (figure 1.f).

Padding is most commonly used to allow us to preserve the spatial size of the input volume so the input and output width and height are the same. This way, we can use convolutional layers without necessarily shrinking the height and width of the volumes. This is important for building deeper networks, since, otherwise, the height/width would shrink as we went to deeper layers. For example, Zero-padding=2 adds two layers of zeros around the border of a image.

The goal when using strides and padding hyperparameters is one of two things: keep all the important details of the image and transfer them to the next layer.

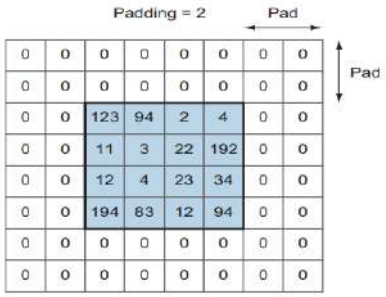


Fig 1.f Zero-padding

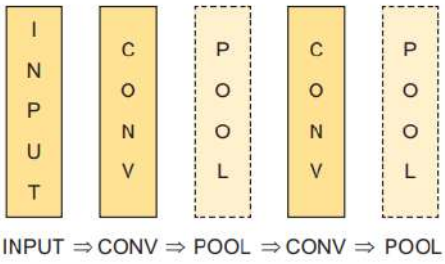
C)Pooling layers or subsampling:

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.

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.

A pooling or subsampling layer often immediately follows a convolution layer in CNN. Its role is to down sample the output of a convolution layer along both the spatial dimensions of height and width.

The goal of the sub sampling and pooling layer is to down sample the feature maps produced by the convolutional layer into a smaller number of parameters, thus reducing computational complexity. It is a common practice to add pooling layers after every one or two convolutional layers in the CNN architecture (figure 1.g).




```
from keras.layers import MaxPooling2D

model.add(MaxPooling2D(pool_size=(2, 2), strides = 2))
```

There are two main types of pooling layers: ***max pooling and average pooling***.

i) Max pooling kernels are windows of a certain size and strides value that slide over the image. The difference with max pooling is that the windows don't have weights or any values. All they do is slide over the feature map created by the previous convolutional layer and select the max pixel value to pass along to the next layer, ignoring the remaining values.

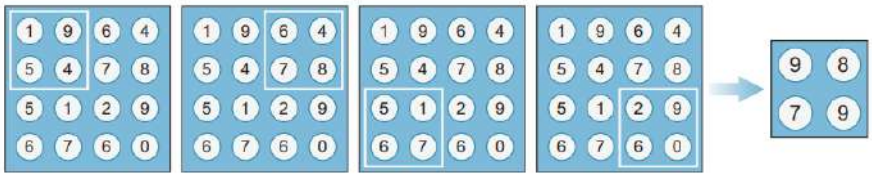


Fig 1.h.pooling filter with a size of 2 × 2 and strides of 2

ii) Global average pooling is a more extreme type of dimensionality reduction. Instead of setting a window size and strides, global average pooling calculates the values of all pixels in the feature map (figure 1.j).

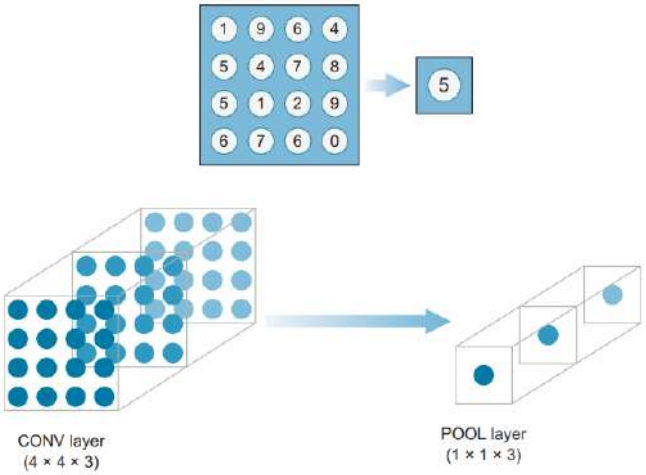
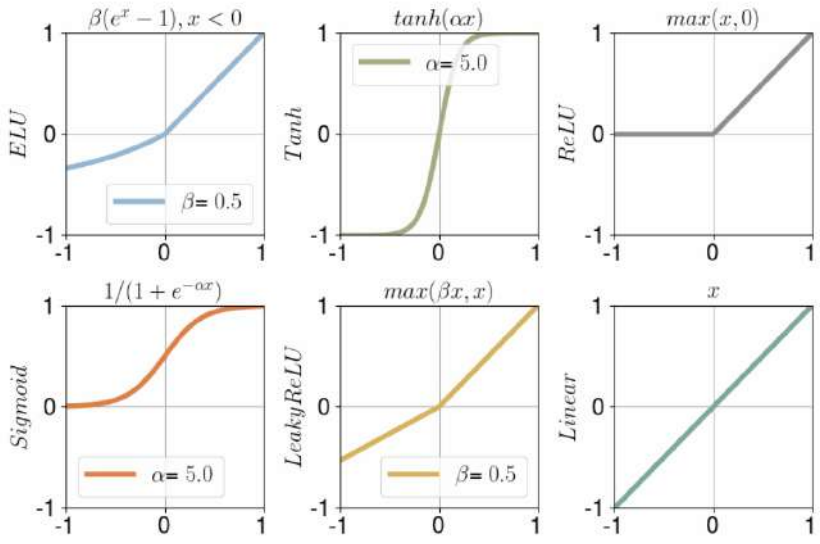


Figure 1.j The global average pooling layer turns a 3D array into a vector.

D. Activation Layers: The main purpose of activation functions is to introduce non-linearity so CNN can efficiently map non-linear complex mapping between the input and output. Multiple activation functions are available and used based on the underlying requirements. As shown in figure 1.k

Non-parametric/Static functions: Linear, ReLU
Parametric functions: ELU, tanh, sigmoid, Leaky ReLU
Bounded functions: tanh, sigmoid



1.k. Activation Functions

D. Fully connected layers:

After passing the image through convolutional and pooling layers, we have extracted all the features and put them in a long tube. Now it is time to use these extracted features to classify images.

Input flattened vector—As illustrated in figure 1.k, for classification feed the all features to flatten it to a vector with the dimensions (1, n). For example, if the features tube has the dimensions of 5 × 5 × 40, the flattened vector will be (1, 1000).

Hidden layer—We add one or more fully connected layers, and each layer has one or more neurons (similar to what we did when we built regular MLPs).

Output layer—The softmax activation function is used for classification problems involving more than two classes. In this example, we are classifying digits from 0 to 9: 10 classes. The number of neurons in the output layer is equal to the number of classes; thus, the output layer will have 10 nodes.

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

Given sample vector input **x** and weight vectors $\{\mathbf{w}_j\}$, the predicted probability of y = j

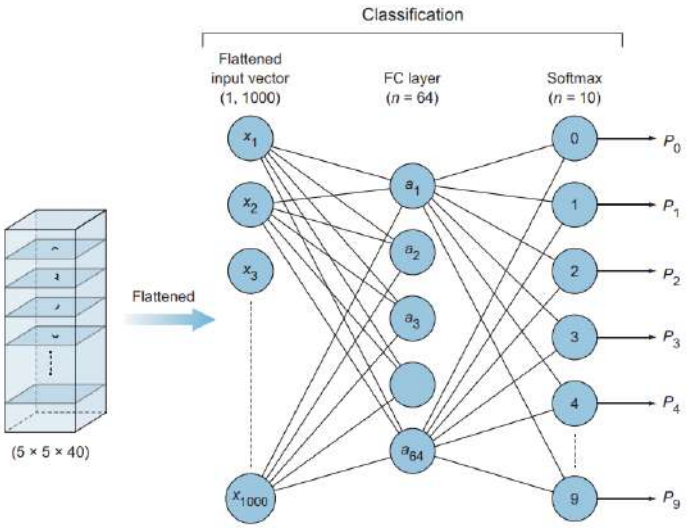


Figure 1.k Fully connected layers for an MLP

Finally, the output layer of a CNN can also be used to perform regularization techniques, such as dropout or batch normalization, to improve the network's performance.

In conclusion we can say CNN architecture is a powerful tool for image and video processing tasks because of combination of convolutional layers, pooling layers, and fully connected layers. These layers allow to extracting features from images, videos, and other data sources and can be used for various tasks, such as object recognition, image classification, and facial recognition. Overall, this type of architecture is highly effective when applied to suitable functions and datasets.

How Training is done in CNN with better accuracy:

The Image data is fed into the model based on convolutions, pooling the outputs are predicted this process is called feedforward, we then calculate the error between the actual and predicted using error functions like cross-entropy, square loss error, etc. The error function measures how well the network is performing. If the error is maximum and accuracy is minimum, we backpropagate into the model by calculating the derivatives. This step is called Backpropagation which basically is used to minimize the loss.

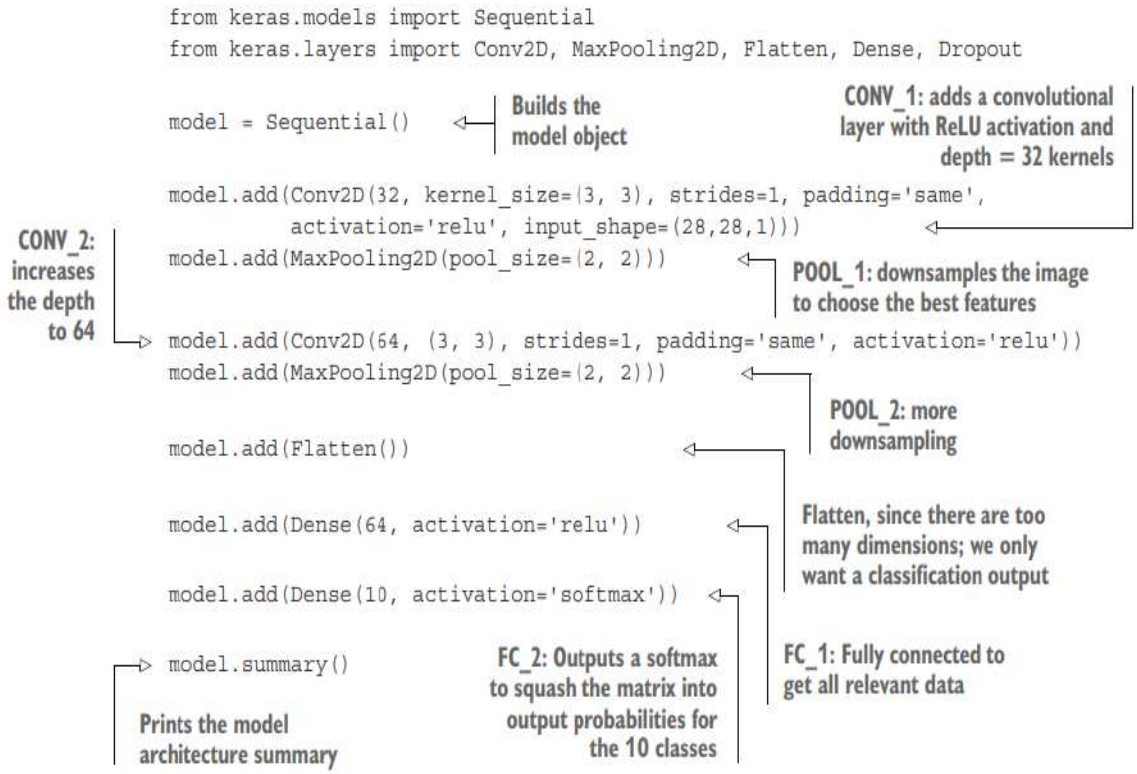
2. Building CNN to Classify 2D Images (Python code)

The basic idea of neural networks is that neurons learn features from the input. In CNNs, a feature map is the output of one filter applied to the previous layer. It is called a feature map because it is a mapping of where a certain kind of feature is found in the image. CNNs look for features such as straight lines, edges, or even objects. Whenever they spot these features, they report them to the feature map. Each feature map is looking for something specific: one could be looking for straight lines and another for curves.

i) Steps to Classify Images Using CNN:

Data set: the MNIST dataset

Snippet Code:



Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 64)	18496
max_pooling2d_2 (MaxPooling2	(None, 7, 7, 64)	0
flatten_1 (Flatten)	(None, 3136)	0
dense_1 (Dense)	(None, 64)	200768
dense_2 (Dense)	(None, 10)	650

Total params: 220,234
Trainable params: 220,234
Non-trainable params: 0

Model Summary Line by Line:

CONV_1—The input shape is (28 × 28 × 1) and output shape of conv2d: (28 × 28 × 32). Since we set the strides parameter to 1 and padding to same, the dimensions of the input image did not change. But depth increased to 32. Why? Because we added 32 filters in this layer. Each filter produces one feature map.

POOL_1—The input of this layer is the output of its previous layer: (28 × 28 × 32). After the pooling layer, the image dimensions shrink, and depth stays the same. Since we used a 2 × 2 pool, the output shape is (14 × 14 × 32).

CONV_2— Same as before, convolutional layers increase depth and keep dimensions. The input from the previous layer is (14 × 14 × 32). Since the filters in this layer are set to 64, the output is (14 × 14 × 64).

POOL_2—Same 2 × 2 pool, keeping the depth and shrinking the dimensions. The output is (7 × 7 × 64).

Flatten—Flattening a features tube that has dimensions of (7 × 7 × 64) converts it into a flat vector of dimensions (1, 3136).

Dense_1—We set this fully connected layer to have 64 neurons, so the output is 64.

Dense_2—This is the output layer that we set to 10 neurons, since we have 10 classes.

ii) Number of parameters (weights)

Parameters are the weights with which the network learns. The of network's goal is to update the weight values during the gradient descent and backpropagation processes until it finds the optimal parameter values that minimize the error function.

How are these parameters calculated?

In MLP, we know that the layers are fully connected to each other, so the weight connections or edges are simply calculated by multiplying the number of neurons in each layer. In CNNs, weight calculations are not as straightforward.

Fortunately, there is an equation for this:

number of params = filters × kernel size × depth of the previous layer + number of filters (for biases)

to calculate the parameters at the second layer CONV_2

model.add(Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'))

the depth of the previous layer is 32, then

⇒ Params = 64 × 3 × 3 × 32 + 64 = 18,496.

Pooling and flatten layers don't add parameters, so Param # is 0 after pooling and flattening layers in the model summary.

Layer (type)	Output Shape	Param #
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 64)	18496
max_pooling2d_2 (MaxPooling2	(None, 7, 7, 64)	0
flatten_1 (Flatten)	(None, 3136)	0

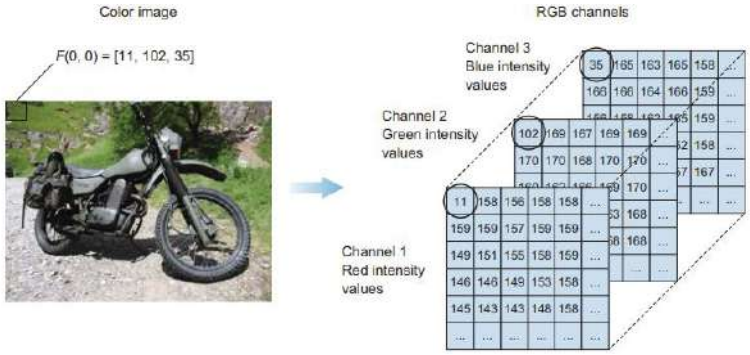
The total number of parameters that this network needs to optimize: 220,234.
The Trainable parameters and non-Trainable parameters are:

```
Total params: 220,234
Trainable params: 220,234
Non-trainable params: 0
```

Using a pretrained network and combining it with your own network for faster and more accurate results: in such a case, you may decide to freeze some layers because they are pretrained. So, not all of the network params will be trained. This is useful for understanding the memory and space complexity of your model before starting the training process.

3. Convolution in Colour images (3D images):

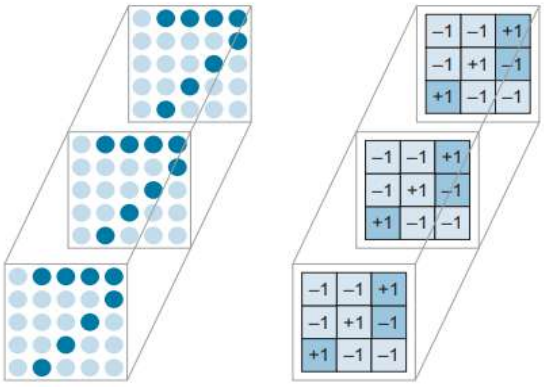
Color images are interpreted by the computer as 3D matrices with height, width, and depth. In the case of RGB images (red, green, and blue) the depth is three: one channel for each color. For example, a color 28 × 28 image will be seen by the computer as a 28 × 28 × 3 matrix. Think of this as a stack of three 2D matrices—one each for the red, green, and blue channels of the image. Each of the three matrices represents the value of intensity of its color. When they are stacked, they create a complete color image (figure 1.l).



For generalization, we represent images as a 3D array: height × width × depth. For grayscale images, depth is 1; and for color images, depth is 3.

How do we perform a convolution on a color image?

Sliding the convolutional kernel over the image and compute the feature maps, resulting in a 3D kernel. The kernel is itself three-dimensional: one dimension for each color channel (figure below).



To perform convolution in color images , sum is three times as many terms. As shown in figure 1.m.

- Each of the color channels has its own corresponding filter.
- Each filter will slide over its image, multiply every corresponding pixel elementwise, and then add them all together to compute the convolved pixel value of each filter. This is similar to what we did previously.
- We then add the three values to get the value of a single node in the convolved image or feature map. And don't forget to add the bias value of 1. Then we slide the filters over by one or more pixels (based on the strides value) and do the same thing.

We continue this process until we compute the pixel values of all nodes in the feature map.

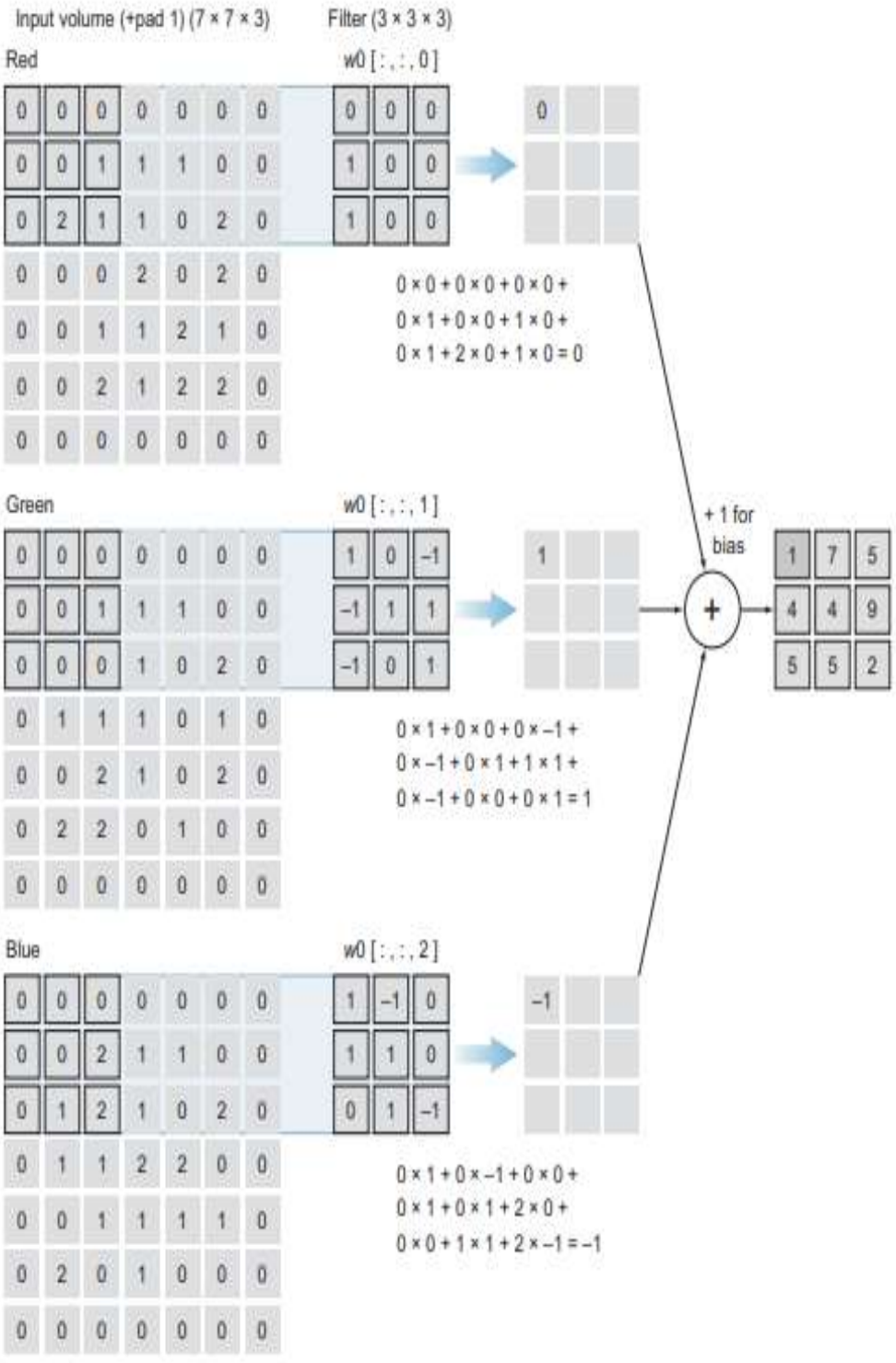


Figure 3.35 Performing convolution in Color images

Problem:

Calculate a value in a Feature Map for 3x3 Image feature convoluted with a 3x3 filter or a Kernel where , Image Feature= [[0,0,0],[0,0,2],[0,1,2]] and kernel =[[1,-1,0],[1,1,0],[0,1,-1]] .

Solution : New Value in a Feature Map = 0*1 +0*-1+0*0+0*1+0*1+2*0+0*0+1*1+2*-1

Answer is -1

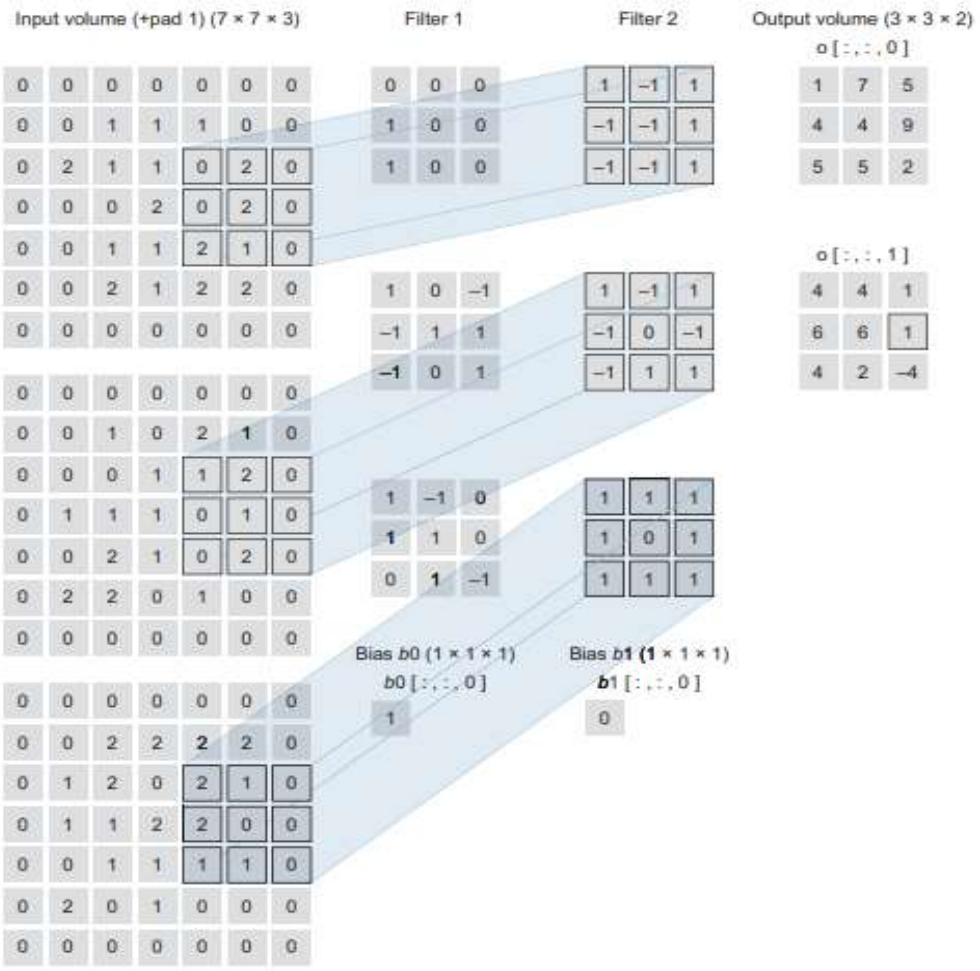


Fig: input image has dimensions (7 × 7 × 3), and we add two convolution filters of dimensions (3 × 3). The output feature map has a depth of 2

4. PROJECT : Implementation of Image Classification for Color Images

Step by Step Implementation an image classifier using CNN:

- Step 1: Load The Dataset
- Step 2: Image Preprocessing
- Step 3 : Define The Model Architecture
- Step 4: Compile The Model
- Step 5: Train /Fit The Model
- Step 6: Load The Model With The Best Val_acc
- Step 7: Evaluate The Model

5. Performance Metrics for Classification Models:

Performance metrics allow us to evaluate our model. The simplest way to measure the "goodness" of our model is by measuring its accuracy. To measure Accuracy, precision, recall and F1 score we use Confusion matrix.

Confusion matrix:

The confusion matrix is a matrix used to determine the performance of the classification models for a given set of test data. It can only be determined if the true values for test data are known. The matrix itself can be easily understood, but the related terminologies may be confusing. Since it shows the errors in the model performance in the form of a matrix, hence also known as an error matrix.

The goal is to describe model performance from different angles other than prediction accuracy. For example, suppose we are building a classifier to predict whether a patient is sick or healthy. The expected classifications are either positive (the patient is sick) or negative (the patient is healthy). We run our model on 1,000 patients and enter the model predictions in Figure below.

	Predicted sick (positive)	Predicted healthy (negative)
Sick patients (positive)	100 True positives (TP)	30 False negative (FN)
Healthy patients (negative)	70 False positives (FP)	800 True negatives (TN)

The most basic terms, which are whole numbers (not rates):

- True positives (TP)—The model correctly predicted yes (the patient has the disease).
- True negatives (TN)—The model correctly predicted no (the patient does not have the disease).
- False positives (FP)—The model falsely predicted yes, but the patient actually does not have the disease (in some literature known as a Type I error or error of the first kind).
- False negatives (FN)—The model falsely predicted no, but the patient actually does have the disease (in some literature known as a Type II error or error of the second kind).

Metrics of Classification with Confusion Matrix:

i)Accuracy: The accuracy metric measures how many times our model made the correct prediction. So, if we test the model with 100 input samples, and it made the correct prediction 90 times, this means the model is 90% accurate.

Here is the equation used to calculate model accuracy:

$$\text{accuracy} = \frac{\text{correct predictions}}{\text{total number of examples}}$$

Is accuracy the best metric for evaluating a model?

We have been using accuracy as a metric for evaluating our model in earlier projects, and it works fine in many cases. Consider the following problem: you are designing a medical diagnosis test for a rare disease. Suppose that only one in every million people has this disease. Without any training or even building a system at all, if you hardcode the output to be always negative (no disease found), your system will always achieve 99.999% accuracy. Is that good? The system is 99.999% accurate, which might sound fantastic, but it will never capture the patients with the disease. This means the accuracy metric is not suitable to measure the "goodness" of this model. We need other evaluation metrics that measure different aspects of the model's prediction ability which are precision ,Recall and F1 Score.

Precision and recall

The patients that the model predicts are negative (no disease) are the ones that the model believes are healthy, and we can send them home without further care. The patients that the model predicts are positive (have disease) are the ones that we will send for further investigation. Which mistake would we rather make? Mistakenly diagnosing someone as positive (has disease) and sending them for more investigation is not as bad as mistakenly diagnosing someone as negative (healthy) and sending them home at risk to their life. The obvious choice of evaluation metric here is that we care more about the number of false negatives (FN). We want to find all the sick people, even if the model accidentally classifies some healthy people as sick. This metric is called recall.

Recall (also known as sensitivity) tells us how many of the sick patients our model incorrectly diagnosed as well. In other words, how many times did the model incorrectly diagnose a sick patient as negative (false negative, FN)?

Recall is calculated by the following equation:

$$\text{Recall} = \frac{\text{true positive}}{\text{true positive} + \text{false negative}}$$

Precision (also known as specificity) is the opposite of recall. It tells us how many of the well patients our model incorrectly diagnosed as sick. In other words, how many times did the model incorrectly diagnose a well patient as positive (false positive, FP)?

Precision is calculated by the following equation:

$$\text{Precision} = \frac{\text{true positive}}{\text{true positive} + \text{false positive}}$$

F-score :

In many cases, we want to summarize the performance of a classifier with a single metric that represents both recall and precision. To do so, we can convert precision (p) and recall (r) into a single F-score metric. In mathematics, this is called the harmonic mean of p and r:

$$\text{F-score} = \frac{2pr}{p+r}$$

The F-score gives a good overall representation of how your model is performing.

Let's take a look at the health-diagnostics example again. We agreed that this is a high recall model. But what if the model is doing really well on the FN and giving us a high recall score, but it's performing poorly on the FP and giving us a low precision score? Doing poorly on FP means, in order to not miss any sick patients, it is mistakenly diagnosing a lot of patients as sick, to be on the safe side. So, while recall might be more important for this problem, it is good to look at the model from both scores—precision and recall—together:

	Precision	Recall	F-score
Classifier A	95%	90%	92.4%
Classifier B	98%	85%	91%

Defining the model evaluation metric is a necessary step because it will guide your approach to improving the system. Without clearly defined metrics, it can be difficult to tell whether changes to a DL system result in progress or not.

Problem:

A DL Image classification model is trained to predict tumour in images. The test dataset consists of 100 people. Out which **10 people** who have tumours are predicted positively, **60 people** who don't have tumors are predicted negatively, **22 people** are predicted as positive of having a tumor, although they don't have a tumor and **8 people** who have tumors are predicted as negative. Draw the Confusion Matrix and Find the Accuracy, recall, Precision and F1 Score.

Solution:

Confusion Matrix:

		ACTUAL	
		Negative	Positive
PREDICTION	Negative	60	8
	Positive	22	10

TP= 10 ; TN= 60; FP= 22; FN =8

Accuracy = TP+TN/(TP+TN+FP+FN) = 70 /100 = 0.70 *100**=70%**

Recall(R) = TP/(TP+FN) = 10/18 = 0.5555 *100**= 55.55%**

Precision (P) = TP/(TP+FP) = 10/32 =0.3125*100 **=31.25 %**

F1score=2*P*R/(P+R) = 0.34 /0.86 =0.39* 100 **= 39%**

6. Evaluating the DL model:

a. Interpreting its performance (Underfitted or Overfitted):

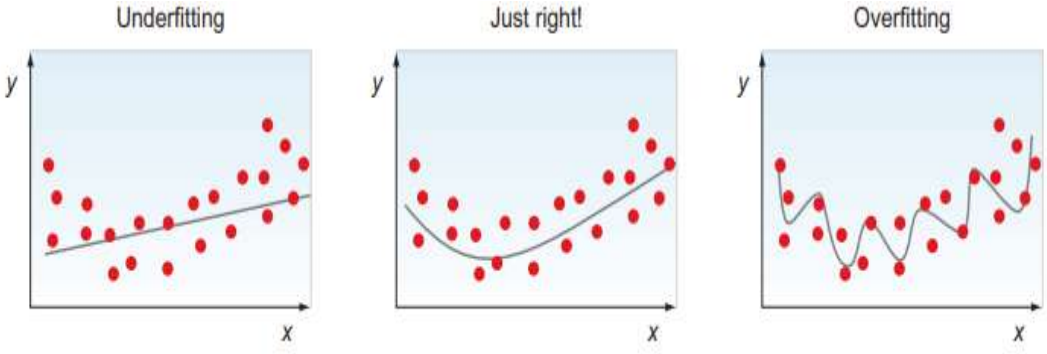
After completing the training of the model, we need to determine the bottlenecks,

- diagnose which CNN components are performing poorly,
- determine whether the poor performance is due to overfitting, underfitting, or a defect in the training data.

The main cause of poor performance in ML and DL models is either overfitting or underfitting on the data.

Underfitting means it fails to learn the training data, so it performs poorly on the training data. This happens when the model is too simple to fit the data or because of more assumptions on data (more Bias) . for example, using one perceptron to classify a nonlinear dataset.

Overfitting, means fitting the data too much: memorizing the training data and not really learning the features. This happens when we build a super network that fits the training dataset perfectly (very low error while training) but fails to generalize to other data samples that it hasn't seen before or variance in the Data is more. In overfitting, the network performs very well in the training dataset but performs poorly in the test dataset (figure below)



To diagnose underfitting and overfitting, the two values to focus on while training are the training error and the validation error:

- If the model is doing very well on the training set but relatively poorly on the validation set, then it is overfitting. For example, if train_error is 1% and val_error is 10%, it looks like the model has memorized the training dataset but is failing to generalize on the validation set. In this case, you might consider tuning your hyperparameters to avoid overfitting and iteratively train, test, and evaluate until you achieve an acceptable performance.
- If the model is performing poorly on the training set, then it is underfitting. For example, if the train_error is 14% and val_error is 15%, the model might be too simple and is failing to learn the training set. You might want to consider adding more hidden layers or training longer (more epochs), or try different neural network architectures.

b. Plotting Training and validation loss curves Vs Epochs

Looking at the training verbose output and comparing the error numbers, one way to diagnose overfitting and underfitting is to plot your training and validation errors throughout the training (epochs vs Losses) , as you see in figure below.

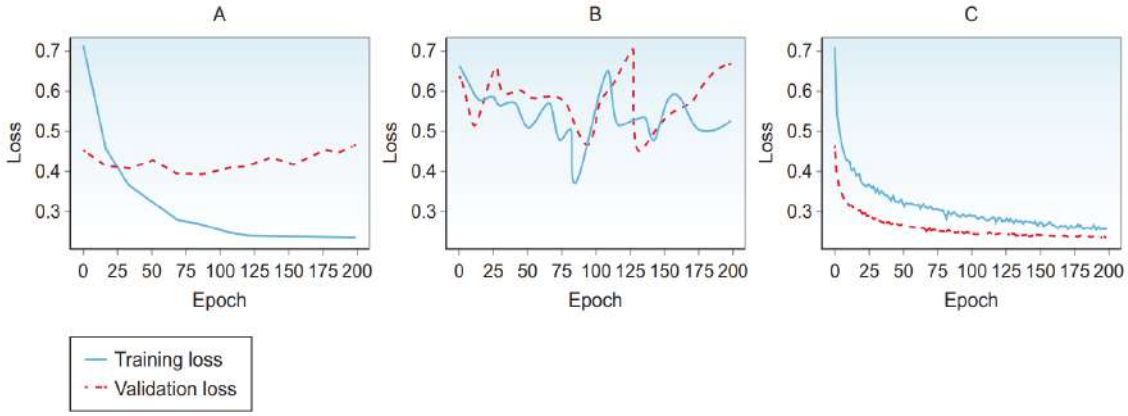


Fig : Evaluating the model with the learning curves

Figure 4.10A shows that the network improves the loss value (aka learns) on the training data but fails to generalize on the validation data. Learning on the validation data progresses in the first couple of epochs and then flattens out and maybe decreases. This is a form of overfitting.

Figure 4.10B shows that the network performs poorly on both training and validation data. In this case, your network is not learning. You don't need more data, because the network is too simple to learn from the data you already have. Your next step is to build a more complex model.

Figure 4.10C shows that the network is doing a good job of learning the training data and generalizing to the validation data. This means there is a good chance that the network will have good performance out in the wild on test data.

7. Improving the NN with Hyperparameter Tuning:

Neural network hyperparameters DL algorithms come with several hyperparameters that control many aspects of the model's behavior. Some hyperparameters affect the time and memory cost of running the algorithm, and others affect the model's prediction ability. The challenge with hyperparameter tuning is that there are no magic numbers that work for every problem. Good hyperparameter values depend on the dataset and the task at hand. Choosing the best hyperparameters and knowing how to tune them require an understanding of what each hyperparameter does.

Hyperparameters tuning are done on three main categories of parameters in a Neural Network

- Network architecture
  - Number of hidden layers (network depth)
  - Number of neurons in each layer (layer width)
  - Activation type
- Learning and optimization
  - Learning rate and decay schedule
  - Mini-batch size
  - Optimization algorithms

  - Number of training iterations or epochs (and early stopping criteria)
- Regularization techniques to avoid overfitting
  - L2 regularization
  - Dropout layers
  - Data augmentation

Tuning each knob up or down and how to know which hyperparameter to tune.

a. Network architecture:

The hyperparameters that define the neural network architecture:

- Number of hidden layers (representing the network depth)
- Number of neurons in each layer, also known as hidden units (representing the network width)
- Activation functions

i) Depth And Width of The Neural Network:

Once the we decide the number of hidden layers in network (depth) and the number of neurons in each layer (width). The number of hidden layers and units describes the learning capacity of the network. The goal is to set the number large enough for the network to learn the data features. A smaller network might underfit, and a larger network might overfit. To know what is a "large enough" network, you pick a starting point, observe the performance, and then tune up or down.

The more complex the dataset, the more learning capacity the model will need to learn its features.

If you provide the model with too much learning capacity (too many hidden units), it might tend to overfit the data and memorize the training set. If your model is overfitting, you might want to decrease the number of hidden units.

Generally, it is good to add hidden neurons until the validation error no longer improves. The trade-off is that it is computationally expensive to train deeper networks. Having a small number of units may lead to underfitting, while having more units is usually not harmful, with appropriate regularization (like dropout)

ii) Activation functions introduces nonlinearity to our neurons. Without activations, our neurons would pass linear combinations (weighted sums) to each other and not solve any nonlinear problems. The non-linear activation functions like ReLU and its variations (like Leaky ReLU) perform the best in hidden layers. But at output layer using the softmax function solve classification problems.

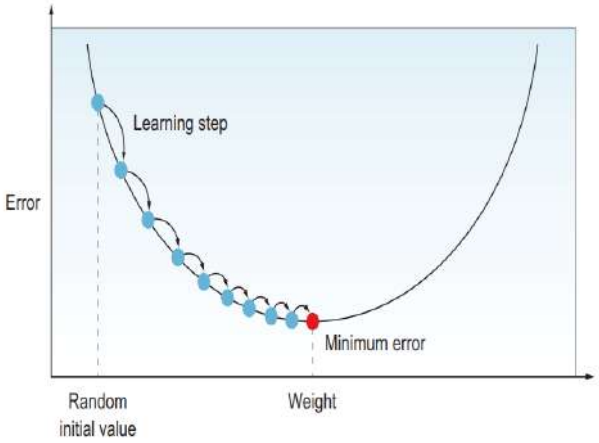
b. Learning and optimization

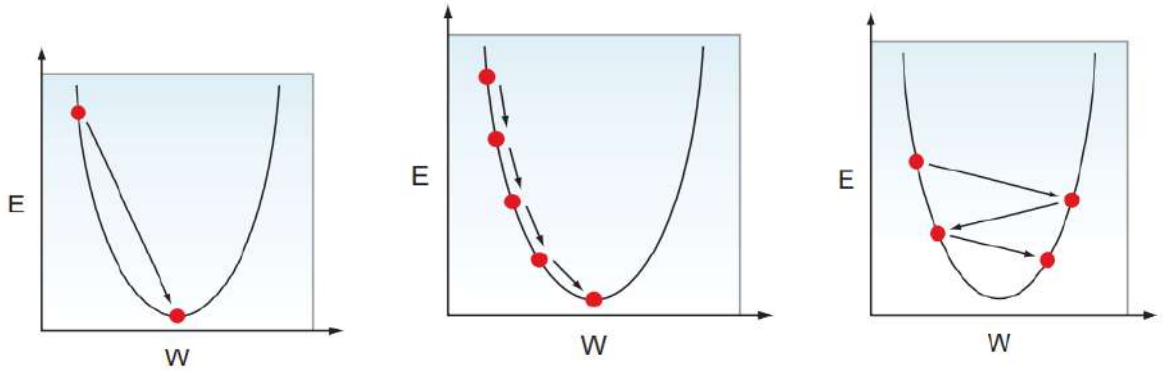
The hyper parameters tuning in an architecture determine how the network learns and optimize its parameter to achieve the minimum error.

i)Learning rate and decay schedule:

The learning rate is the single most important hyperparameter, and one should always make sure that it has been tuned. In GD optimizer searches for the optimal values of weights that yield the lowest error possible. When setting up our optimizer, we need to define the step size

that it takes when it descends the error mountain. This step size is the learning rate. It represents how fast or slow the optimizer descends the error curve. When we plot the cost function with only one weight, we get the oversimplified U-curve in figure 4.14, where the weight is randomly initialized at a point on the curve.

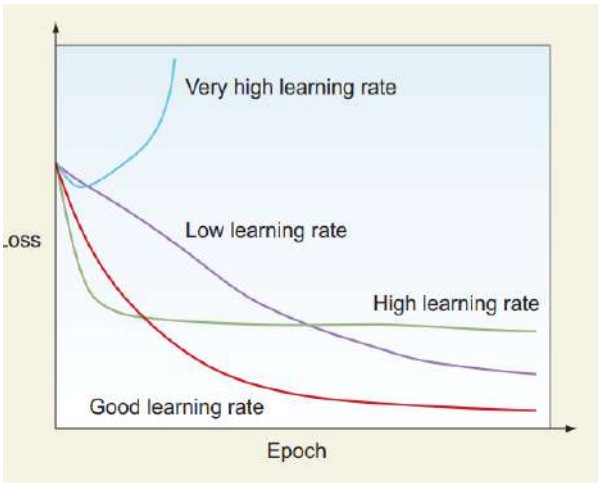




Too-high vs. too-low learning rate Setting the learning rate high or low is a trade-off between the optimizer speed versus performance.

When plotting the loss value against the number of training iterations (epochs), you will notice the following:

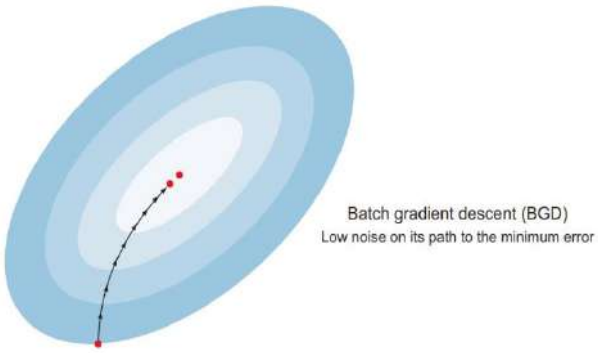
- Much smaller lr—The loss keeps decreasing but needs a lot more time to converge.
- Larger lr—The loss achieves a better value than what we started with, but is still far from optimal.
- Much larger lr—The loss might initially decrease, but it starts to increase as the weight values get farther and farther away from the optimal values.
- Good lr—The loss decreases consistently until it reaches the minimum possible value.



The difference between very high, high, good, and low learning rates are shown in the plot below.

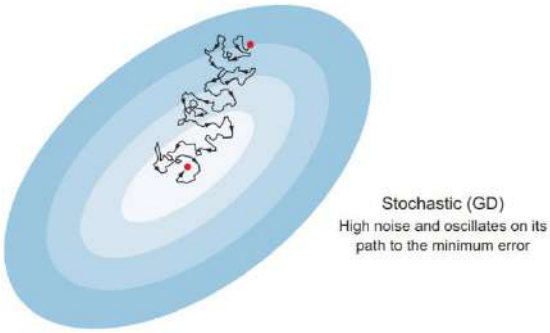
ii) Mini-batch size:

Mini-batch size is another hyperparameter that you need to set and tune in the optimizer algorithm. The batch_size hyperparameter has a big effect on resource requirements of the training process and speed.



*Batch gradient descent (BGD)—T*he entire dataset is fed into the network all at once, apply the feedforward process, calculate the error, calculate the gradient, and backpropagate to update the weights. The optimizer calculates the gradient by looking at the error generated after it sees all the training data, and the weights are updated only once after each epoch. So, in this case, the mini-batch size equals the entire training dataset. The main advantage of BGD is that it has relatively low noise and bigger steps toward the minimum (see figure 4.21). The main disadvantage is that it can take too long to process the entire training dataset at each step, especially when training on big data. BGD also requires a huge amount of memory for training large datasets, which might not be available. BGD might be a good option if you are training on a small dataset.

Stochastic gradient descent (SGD)—Also called online learning. We feed the network a single instance of the training data at a time and use this one instance to do the forward pass, calculate error, calculate the gradient, and backpropagate to update the weights (figure 4.22). In SGD, the weights are updated after it sees each single instance (as opposed to processing the entire dataset before each step for BGD). SGD can be extremely noisy as it oscillates on its way to the global minimum because it takes a step down after each single instance, which could sometimes be in the wrong direction. This noise can be reduced by using a smaller learning rate, so, on average, it takes you in a good direction and almost always performs better than BGD. With SGD you get to make progress quickly and usually reach very close to the global minimum. The main disadvantage is that by calculating the GD for one instance at a time, you lose the speed gain that comes with matrix multiplication in the training calculations.



Mini-batch gradient descent (MB-GD)—A compromise between batch and stochastic GD. Instead of computing the gradient from one sample (SGD) or all training samples (BGD), we divide the training sample into mini-batches to compute the gradient from. This way, we can take advantage of matrix multiplication for faster training and start making progress instead of having to wait to train the entire training set.

iii) Optimization algorithms:

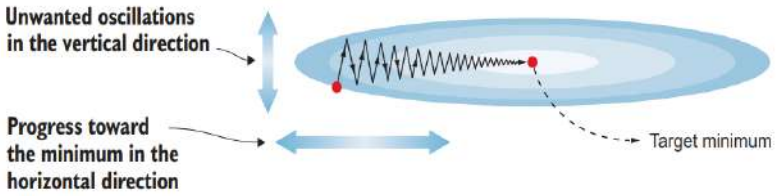
The DL community believed that the batch, stochastic, and mini-batch GD algorithms works well , But not. Another by choosing a proper learning rate is challenging because a too small learning rate leads to painfully slow convergence, while a too-large learning rate can hinder

convergence and cause the loss function to fluctuate around the minimum or even diverge. We need more creative solutions to further optimize GD.

The most popular gradient-descent-based optimizers are Momentum and Adam

Gradient descent with momentum

SGD ends up with some oscillations in the vertical direction toward the minimum error (figure below). These oscillations slow down the convergence process and make it harder to use larger learning rates, which could result in your algorithm overshooting and diverging.



To reduce these oscillations, a technique called momentum was invented that lets the GD navigate along relevant directions and softens the oscillation in irrelevant directions. In other words, it makes learning slower in the vertical-direction oscillations and faster in the horizontal-direction progress, which will help the optimizer reach the target minimum much faster.

The momentum is built by adding a velocity term to the equation that updates the weight:

$$w_{new} = w_{old} - \alpha \frac{dE}{dw_i} \quad \longleftarrow \text{Original update rule}$$

$$w_{new} = w_{old} - \text{learning rate} \times \text{gradient} + \text{velocity term} \quad \longleftarrow \text{New rule after adding velocity}$$

Adam

Adam stands for adaptive moment estimation. Adam keeps an exponentially decaying average of past gradients, similar to momentum. Whereas momentum can be seen as a ball rolling down a slope, Adam behaves like a heavy ball with friction to slow down the momentum and control it. Adam usually outperforms other optimizers because it helps train a neural network model much more quickly than the other techniques.

Adam proposes these default values:

- The learning rate needs to be tuned.
- For the momentum term β1, a common choice is 0.9.
- For the RMSprop term β2, a common choice is 0.999.
- ε is set to 10–8.

A training iteration, or epoch, is when the model goes a full cycle and sees the entire training dataset at once. The epoch hyperparameter is set to define how many iterations our network continues training. The more training iterations, the more our model learns the features of our training data. To diagnose whether your network needs more or fewer training epochs, keep your eyes on the training and validation error values.

iv) Early stopping:

Early stopping is an algorithm widely used to determine the right time to stop the training process before overfitting happens. It simply monitors the validation error value and stops the training when the value starts to increase.

The early stopping function in Keras:

EarlyStopping(monitor='val_loss', min_delta=0, patience=20)

The EarlyStopping function takes the following arguments:

monitor—The metric you monitor during training. Usually we want to keep an eye on val_loss because it represents our internal testing of model performance. If the network is doing well on the validation data, it will probably do well on test data and production. ***min_delta***—The minimum change that qualifies as an improvement. There is no standard value for this variable. To decide the min_delta value, run a few epochs and see the change in error and validation accuracy. Define min_delta according to the rate of change. The default value of 0 works pretty well in many cases.

patience—This variable tells the algorithm how many epochs it should wait before stopping the training if the error does not improve. For example, if we set patience equal to 1, the training will stop at the epoch where the error increases. We must be a little flexible, though, because it is very common for the error to oscillate a little and continue improving. We can stop the training if it hasn't improved in the last 10 or 20 epochs.

The good thing about early stopping is that it allows you to worry less about the epochs hyperparameter. You can set a high number of epochs and let the stopping algorithm take care of stopping the training when error stops improving.

C. Regularization techniques to avoid overfitting

when neural network is overfitting on the training data, we should optimize and avoid overfit. The three of the most common regularization techniques: L2, dropout, and data augmentation.

i) L2 regularization: The basic idea of L2 regularization is that it penalizes the error function by adding a regularization term to it. This, in turn, reduces the weight values of the hidden units and makes them too small, very close to zero, to help simplify the model.

we update the error function by adding the regularization term:

$$\text{error function}_{\text{new}} = \text{error function}_{\text{old}} + \text{regularization term}$$

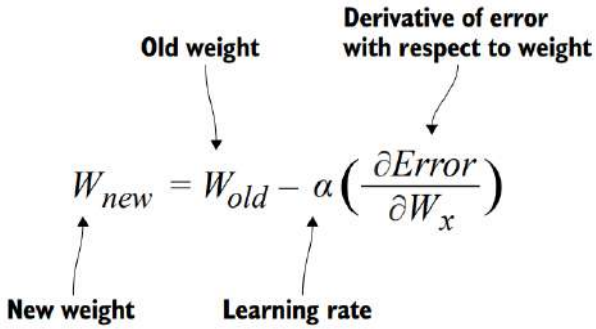
$$\text{L2 regularization term} = \frac{\lambda}{2m} \times \sum \|w\|^2$$

where lambda (λ) is the regularization parameter, m is the number of instances, and w is the weight. The updated error function looks like this:

$$\text{error function}_{\text{new}} = \text{error function}_{\text{old}} + \frac{\lambda}{2m} \times \sum \|w\|^2$$

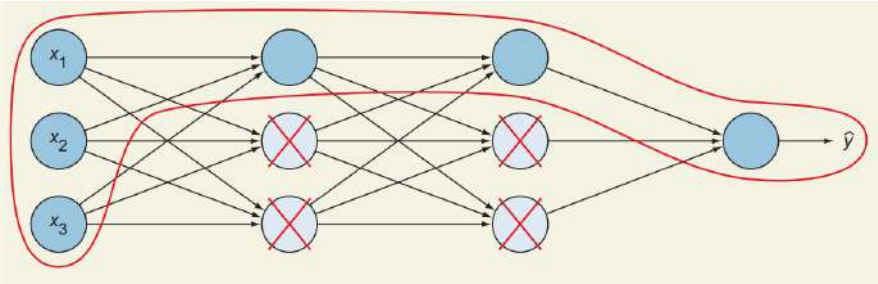
Why does L2 regularization reduce overfitting? Well, we saw how the weights are updated during the backpropagation process. The optimizer calculates the derivative of the error,

multiplies it by the learning rate, and subtracts this value from the old weight. Here is the backpropagation equation that updates the weights:



Since we add the regularization term to the error function, the new error becomes larger than the old error. This means its derivative (∂Error/∂Wx) is also bigger, leading to a smaller Wnew. L2 regularization is also known as weight decay, as it forces the weights to decay toward zero (but not exactly zero).

L2 regularization does not make the weights equal to zero. It just makes them smaller to reduce their effect. A large regularization parameter (λ) lead to negligible weights. When the weights are negligible, the model will not learn much from these units. This will make the network simpler and thus reduce overfitting.



L2 regularization reduces the weights and simplifies the network to reduce overfitting.

```
model.add(Dense(units=16, kernel_regularizer=regularizers.l2(λ),
    activation='relu'))
```

When adding a hidden layer to your network, add the kernel_regularization argument with the L2 regularizer

ii) Dropout layers:

A dropout layer is one of the most commonly used layers to prevent overfitting. Dropout turns off a percentage of neurons (nodes) that make up a layer of your network (figure 3.31). Dropout is another regularization technique that is very effective for simplifying a neural network and avoiding overfitting.

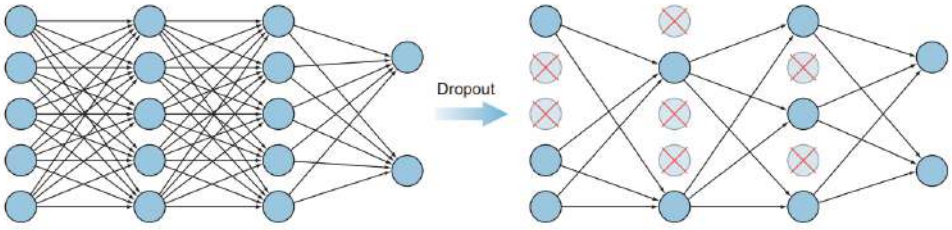


Figure 3.31 Dropout turns off a percentage of the neurons that make up a network layer.

Both L2 regularization and dropout aim to reduce network complexity by reducing its neurons' effectiveness. The difference is that dropout completely cancels the effect of some neurons with every iteration, while L2 regularization just reduces the weight values to reduce the neurons' effectiveness. Both lead to a more robust, resilient neural network and reduce overfitting. It is recommended that you use both types of regularization techniques in your network.

iii)Data augmentation:

One way to avoid overfitting is to obtain more data. Since this is not always a feasible option, we can augment our training data by generating new instances of the same images with some transformations. Data augmentation can be an inexpensive way to give your learning algorithm more training data and therefore reduce overfitting. The many image-augmentation techniques include flipping, rotation, scaling, zooming, lighting conditions, and many other transformations that you can apply to your dataset to provide a variety of images to train on.

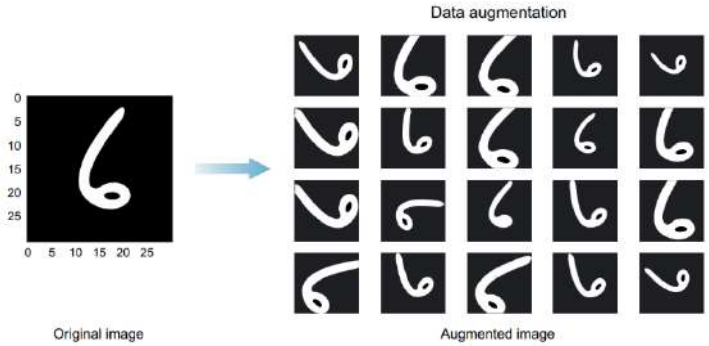


Figure : Various image augmentation techniques applied to an image of the digit 6

In figure above, we created 20 new images that the network can learn from. The main advantage of synthesizing images like this is that now you have more data (20×) that tells your algorithm that if an image is the digit 6, then even if you flip it vertically or horizontally or rotate it, it's still the digit 6. This makes the model more robust to detect the number 6 in any form and shape.

Data augmentation is considered a regularization technique because allowing the network to see many variants of the object reduces its dependence on the original form of the object during feature learning. This makes the network more resilient when tested on new data.

Data augmentation in Keras:

Imports ImageDataGenerator from Keras

```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)

datagen.fit(training_set)
```

Generates batches of new image data. ImageDataGenerator takes transformation types as arguments. Here, we set horizontal and vertical flip to True. See the Keras documentation (or your DL library) for more transformation arguments.

Computes the data augmentation on the training set

5. What is Batch Normalization?

Batch normalization is a deep learning approach that has been shown to significantly improve the efficiency and reliability of neural network models. It is particularly useful for training very deep networks, as it can help to reduce the internal covariate shift that can occur during training.

Batch normalization is a supervised learning method for normalizing the interlayer outputs of a neural network. As a result, the next layer receives a "reset" of the output distribution from the preceding layer, allowing it to analyze the data more effectively.

The term "internal covariate shift" is used to describe the effect that updating the parameters of the layers above it has on the distribution of inputs to the current layer during deep learning training. This can make the optimization process more difficult and can slow down the convergence of the model.

Since normalization guarantees that no activation value is too high or too low, and since it enables each layer to learn independently from the others, this strategy leads to quicker learning rates.

By standardizing inputs, the "dropout" rate (the amount of information lost between processing stages) may be decreased. That ultimately leads to a vast increase in precision across the board.

How does batch normalization work?

Batch normalization is a technique used to improve the performance of a deep learning network by first removing the batch mean and then splitting it by the batch standard deviation.

Stochastic gradient descent is used to rectify this standardization if the loss function is too big, by shifting or scaling the outputs by a parameter, which in turn affects the accuracy of the weights in the following layer.

When applied to a layer, batch normalization multiplies its output by a standard deviation parameter (gamma) and adds a mean parameter (beta) to it as a secondary trainable parameter. Data may be "denormalized" by adjusting just these two weights for each output, thanks to the synergy between batch normalization and gradient descents. Reduced data loss and improved network stability were the results of adjusting the other relevant weights.

The goal of batch normalization is to stabilize the training process and improve the generalization ability of the model. It can also help to reduce the need for careful initialization of the model's weights and can allow the use of higher learning rates, which can speed up the training process.

Batch normalization overfitting

While batch normalization can help to reduce overfitting, it is not a guarantee that a model will not overfit. Overfitting can still occur if the model is too complex for the amount of training data, if there is a lot of noise in the data, or if there are other issues with the training process. It is important to use other regularization techniques like dropout, and to monitor the performance of the model on a validation set during training to ensure that it is not overfitting.

Batch normalization equations

During training, the activations of a layer are normalized for each mini-batch of data using the following equations:

To zero-center the inputs, the algorithm needs to calculate the input mean and standard deviation (the input here means the current mini-batch: hence the term batch normalization):

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \quad \text{Mini-batch mean}$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 \quad \text{Mini-batch variance}$$

where m is the number of instances in the mini-batch, μB is the mean, and σB is the standard deviation over the current mini-batch. 2 Normalize the input:

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

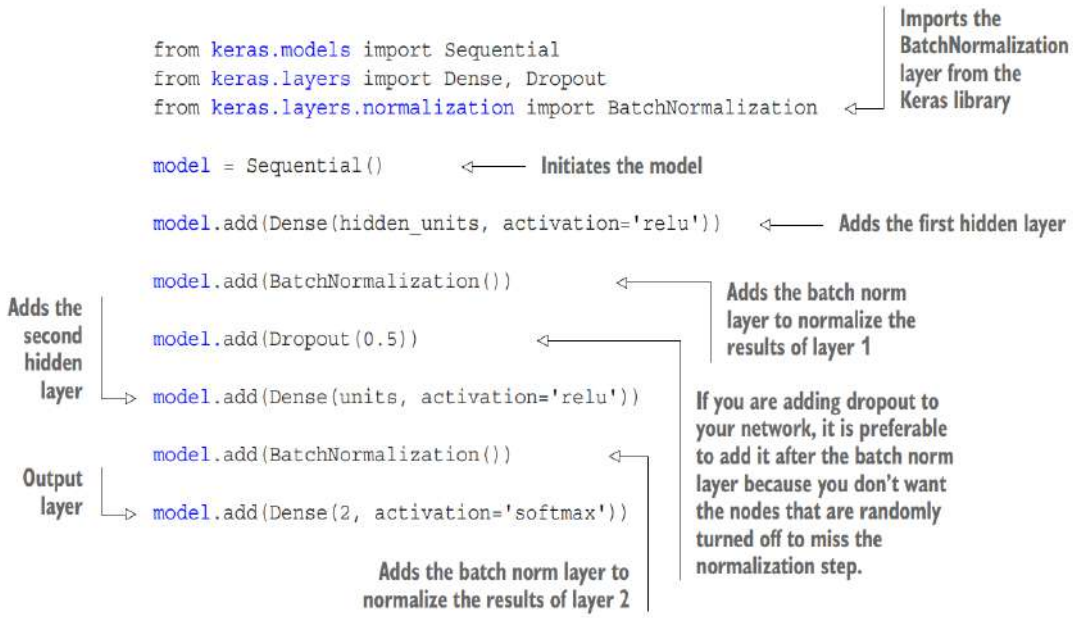
where xˆ is the zero-centered and normalized input. Note that there is a variable here that we added (ε). This is a tiny number (typically 10–5) to avoid division by zero if σ is zero in some estimates. 3 Scale and shift the results. We multiply the normalized output by a variable γ to scale it and add (β) to shift it

$$y_i \leftarrow \gamma x_i + \beta$$

where yi is the output of the BN operation, scaled and shifted.

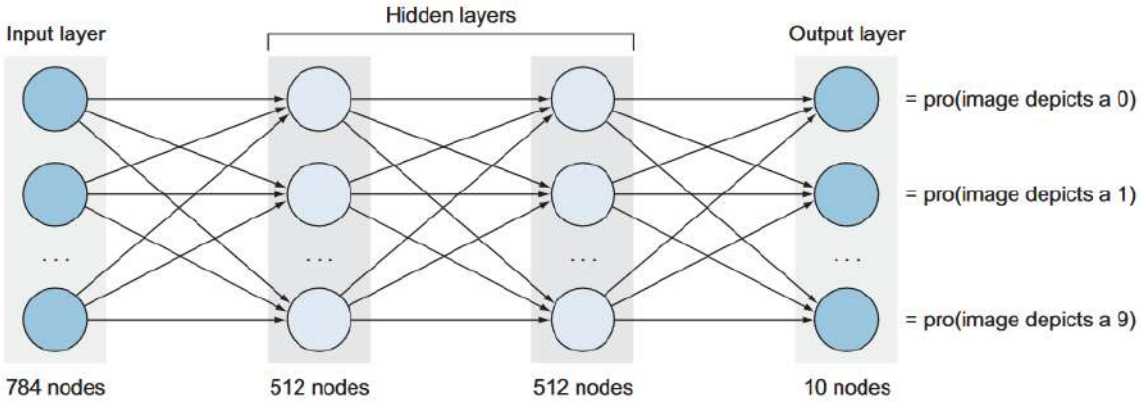
BN introduces two new learnable parameters to the network: γ and β. So our optimization algorithm will update the parameters of γ and β just like it updates weights and biases. In practice, this means you may find that training is rather slow at first, while GD is searching for the optimal scales and offsets for each layer, but it accelerates once it's found reasonably good values.

The following code snippet shows you how to add a BN layer when building your neural network:



Implementation of MLP (Simple ANN or Fully connected Layers) using Keras

The MLP architecture has neurons stacked in layers on top of each other, with weight connections. The MLP architecture consists of an input layer, one or more hidden layers, and an output layer (figure below).



From the above fig the input 28x28 image is flatted in a vector with 784 nodes, connected to the two hidden layers produce 512 nodes each; and, finally, the output layer (dense_3) produces a layer with 10 nodes that is output 10 class classification.

The snippet code is to create MLP in Keras :

```
from keras.models import Sequential
from keras.layers import Flatten, Dense
model = Sequential()   # Defines the neural network architecture
model.add( Flatten(input_shape = (28,28) ))  # Adds the  Flatten layer
model.add(Dense(512, activation = 'relu'))  # Adds 2 hidden layers with 512 nodes
# each. Using the ReLU activation function is recommended in hidden layers.
model.add(Dense(512, activation = 'relu'))
model.add(Dense(10, activation = 'softmax')) #Adds 1 output Dense layer with
#10 nodes. Using the softmax activation function is recommended in the output layer for
#multiclassclassification problems.
model.summary()
```

The Param # field represents the number of parameters (weights) produced at each layer. These are the weights that will be adjusted and learned during the training process. They are calculated as follows:

1 Params after the flatten layer = 0, because this layer only flattens the image to a vector for feeding into the input layer. The weights haven t been added yet.
2 Params after layer 1 = (784 nodes in input layer) (512 in hidden layer 1) +(512 connections to biases) = 401,920.
3 Params after layer 2 = (512 nodes in hidden layer 1) (512 in hidden layer 2) +(512 connections to biases) = 262,656.
4 Params after layer 3= (512 nodes in hidden layer 2) (10 in output layer) + (10connections to biases) = 5,130.
5 Total params in the network = 401,920 + 262,656 + 5,130 = 669,706.

PROJECT 1: Implementation of Image Classification for Color Images

Data Set: CIFAR-10 dataset (www.cs.toronto.edu/ ~kriz/cifar.html). CIFAR-10 is an established CV dataset used for object recognition. It is a subset of the 80 Million Tiny Images dataset1 and consists of 60,000 (32 × 32) color images containing 1 of 10 object classes, with 6,000 images per class.

Steps to develop an image classifier for a custom dataset:

Step 1: Load the Dataset

The first step is to load the dataset into our train and test objects. Keras provides the CIFAR dataset for us to load using the load_data() method. All we have to do is import keras.datasets and then load the data:

```
import keras
from keras.datasets import cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(20,5))
for i in range(36):
    ax = fig.add_subplot(3, 12, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_train[i]))
```

Loads the preshuffled train and tests the data

Step 2: Image Preprocessing

Based on the dataset and the problem do some data cleanup and preprocessing to get it ready learning the model.

A cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales. Figure 3.38 shows gradient descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right). When using gradient descent, you should ensure that all features have a similar scale; otherwise, it will take much longer to converge.

Rescale the images:

```
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255
```

Rescales the images by dividing the pixel values by 255: [0,255] ⇒ [0,1]

Prepare the labels (one-hot encoding) :

Every image in our dataset has a specific label that explains (in text) how this image is categorized. In this particular dataset, for example, the labels are categorized by the following 10 classes: ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']. We need to convert these text labels into a form that can be processed by computers. Computers are good with numbers, so we will do something called one-hot encoding. One-hot encoding is a process by which categorical variables are converted into a numeric form.

Image	Label
image_1	dog
image_2	automobile
image_3	airplane
image_4	truck
image_5	bird

After one-hot encoding, we have the following:

	airplane	bird	cat	deer	dog	frog	horse	ship	truck	automobile
image_1	0	0	0	0	1	0	0	0	0	0
image_2	0	0	0	0	0	0	0	0	0	1
image_3	1	0	0	0	0	0	0	0	0	0
image_4	0	0	0	0	0	0	0	0	1	0
image_5	0	1	0	0	0	0	0	0	0	0

```
from keras.utils import np_utils

num_classes = len(np.unique(y_train))
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

One-hot encodes the labels

Split the dataset for training and validation

In addition to splitting our data into train and test datasets, it is a standard practice to further split the training data into training and validation datasets (figure below)Why? Because each split is used for a different purpose:

Training dataset—The sample of data used to train the model.

Validation dataset—The sample of data used to provide an unbiased evaluation of model fit on the training dataset while tuning model hyperparameters. The evaluation becomes more biased as skill on the validation dataset is incorporated into the model configuration.

Test dataset—The sample of data used to provide an unbiased evaluation of final model fit on the training dataset.

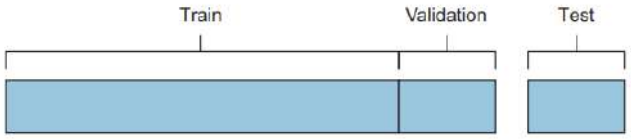


Fig :Splitting the data into training, validation, and test subsets

```
(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]
```
Breaks the training set into training and validation sets

```
print('x_train shape:', x_train.shape)
```
Prints the shape of the training set

```
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print(x_valid.shape[0], 'validation samples')
```
Prints the number of training, validation, and test images

Step 3 : Define The Model Architecture:

The core building block of CNNs (and neural networks in general) is the layer. The main CNN layers are convolution, pooling, fully connected, and activation functions.
How many convolutional layers should you create?
How many pooling layers?

- The more layers you add, the better your network will learn; but this will come at the cost of increasing the computational and memory space complexity, because it increases the number of parameters to optimize. You will also face the risk of the network overfitting your training set.
- As the input image goes through the network layers, its depth increases, and the dimensions (width, height) shrink, layer by layer.
- In general, two or three layers of 3 × 3 convolutional layers followed by a 2 × 2 pooling can be a good start for smaller datasets. Add more convolutional and pooling layers until your image is a reasonable size (say, 4 × 4 or 5 × 5), and then add a couple of fully connected layers for classification.
- You need to set up several hyperparameters (like filter, kernel_size, and padding). Remember that you do not need to reinvent the wheel: instead, look in the literature to see what hyperparameters usually work for others. Choose an architecture that worked well for someone else as a starting point, and then tune these hyperparameters to fit your situation.

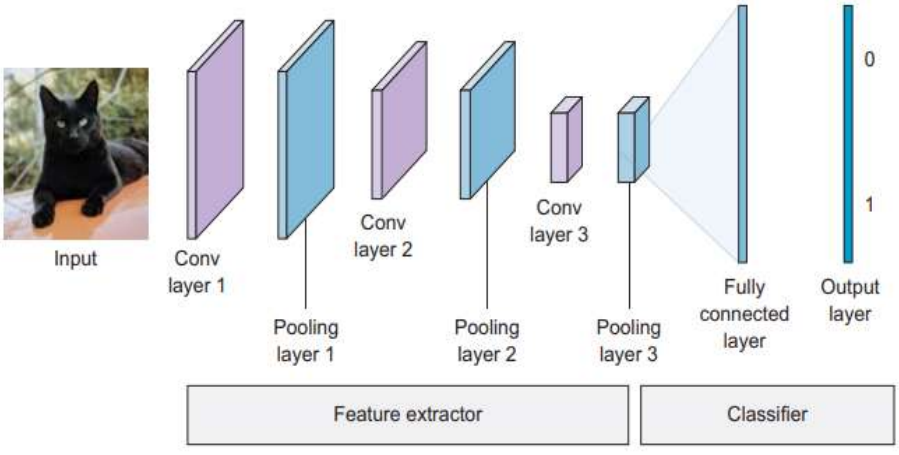


Fig: building a small CNN consisting of three convolutional layers and two dense layers.

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()
model.add(Conv2D(filters=16, kernel_size=2, padding='same',
    activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=2))

model.add(Conv2D(filters=32, kernel_size=2, padding='same',
    activation='relu'))
model.add(MaxPooling2D(pool_size=2))

model.add(Conv2D(filters=64, kernel_size=2, padding='same',
    activation='relu'))
model.add(MaxPooling2D(pool_size=2))

model.add(Dropout(0.3))

model.add(Flatten())

model.add(Dense(500, activation='relu'))
model.add(Dropout(0.4))

model.add(Dense(10, activation='softmax'))

model.summary()
```

First convolutional and pooling layers. Note that we need to define input_shape in the first convolutional layer only.

Second convolutional and pooling layers with a ReLU activation function

Third convolutional and pooling layers

Dropout layer to avoid overfitting with a 30% rate

Flattens the last feature map into a vector of features

Adds the first fully connected layer

Another dropout layer with a 40% rate

The output layer is a fully connected layer with 10 nodes and softmax activation to give probabilities to the 10 classes.

Prints a summary of the model architecture

When we run this cell, we will see the model architecture and how the dimensions of the feature maps change with every successive layer, as illustrated in figure below. We discussed previously how to understand this summary.

As you can see, our model has 528,054 parameters (weights and biases) to train. We also discussed previously how this number was calculated.

Step 4: Compile The Model:

The last step before training our model is to define three more hyperparameters—a loss function, an optimizer, and metrics to monitor during training and testing:

Loss function—How the network will be able to measure its performance on the training data.

Optimizer—The mechanism that the network will use to optimize its parameters (weights and biases) to yield the minimum loss value. It is usually one of the variants of stochastic gradient descent.

Metrics—List of metrics to be evaluated by the model during training and testing. Typically, we use metrics=['accuracy'].

```
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
    metrics=['accuracy'])
```

Step 5: Train the Model

Train the network with Keras,

```
from keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5', verbose=1,
    save_best_only=True)

hist = model.fit(x_train, y_train, batch_size=32, epochs=100,
    validation_data=(x_valid, y_valid), callbacks=[checkpointer],
    verbose=2, shuffle=True)
```

Analysing how your network is performing and suggest which knobs (hyperparameter) to tune.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 16)        208
_________________________________________________________________
max_pooling2d_1 (MaxPooling 2 (None, 16, 16, 16)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 32)        2080
_________________________________________________________________
max_pooling2d_2 (MaxPooling 2 (None, 8, 8, 32)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 64)          8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling 2 (None, 4, 4, 64)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 4, 4, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 500)               512500
_________________________________________________________________
dropout_2 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5010
=================================================================
Total params: 528,054
Trainable params: 528,054
Non-trainable params: 0
_________________________________________________________________
```

STEP 6: LOAD THE MODEL WITH THE BEST VAL_ACC

Now that the training is complete, we use the Keras method load_weights() to load into our model the weights that yielded the best validation accuracy score.

```
model.load_weights('model.weights.best.hdf5')
```

STEP 7: EVALUATE THE MODEL The last **step** is to evaluate our model and calculate the accuracy value as a percentage indicating how often our model correctly predicts the image classification. When you run this cell, you will get an accuracy of about 70%. That is not bad. But we can do a lot better. Try playing with the CNN architecture by adding more convolutional and pooling layers, and see if you can improve your model. In the next chapter, we will discuss strategies to set up your DL project and **hyperparameter** tuning to improve the model's performance. At the end of chapter 4, we will revisit this project to apply these strategies and improve the accuracy to above 90%.

```
score = model.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', score[1])
```