Image Classification (often referred to as *Image Recognition*) is the task of associating one (*single-label classification)* or more (*multi-label classification)* labels to a given image.

Here's how it looks like in practice when classifying different birds—

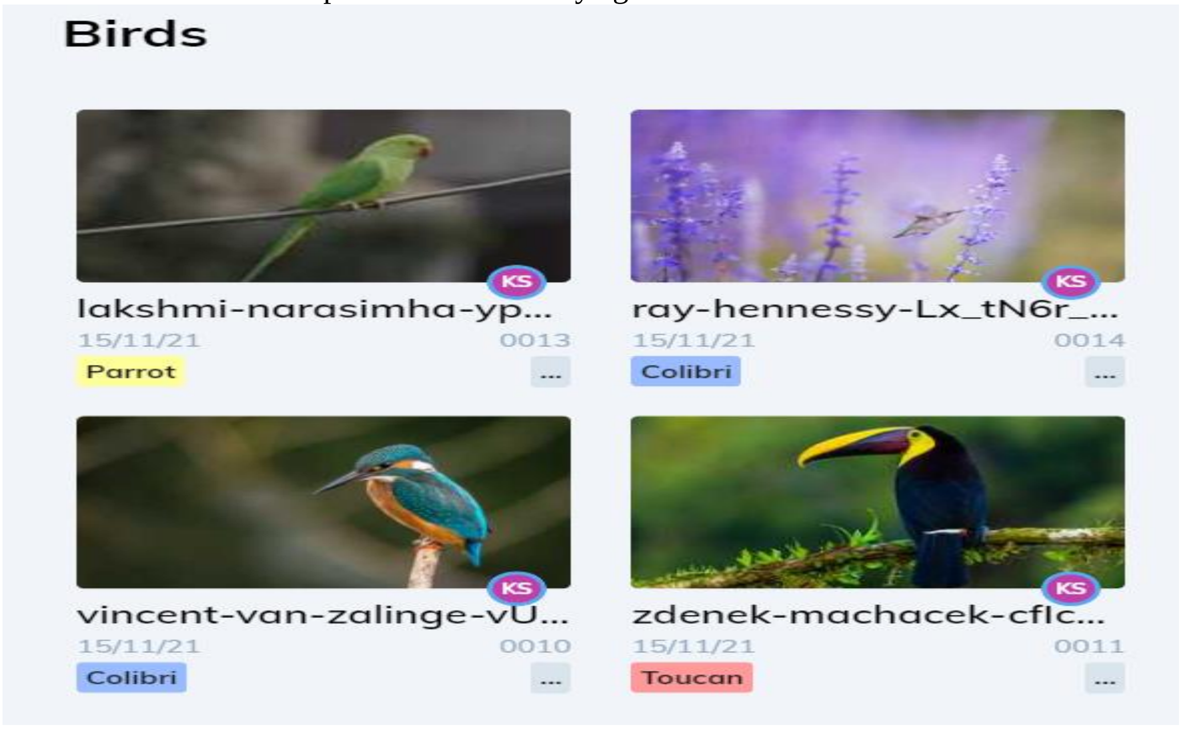


Image Classification is a solid task to benchmark modern architectures and methodologies in the domain of computer vision.
Single-label classification is the most common classification task in supervised Image Classification. As the name suggests, a single label or annotation is present for each image in single-label classification. Therefore, the model outputs a single value or prediction for each image that it sees. The output from the model is a vector with a length equal to the number of classes and value denoting the score that the image belongs to this class.
A Softmax activation function is employed to make sure the score sums up to one and the maximum of the scores is taken to form the model's output.

Softmax

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

While the Softmax initially seems not to provide any value to the prediction as the maximum probable class does not change after applying it, it helps to bound the output between one and zero, helping gauge the model confidence from the Softmax score.

Some examples of single-label classification datasets include MNIST, SVHN, ImageNet, and more. Single-label classification can be of Multiclass classification type where there are more than two classes or binary classification, where the number of classes is restricted to only two.

Multi-label Classification

Multi-label classification is a classification task where each image can contain more than one label, and some images can contain all the labels simultaneously.

While this seems similar to single-label classification in some respect, the problem statement is more complex compared to single-label classification.

Multi-label classification tasks popularly exist in the medical imaging domain where a patient can have more than one disease to be diagnosed from visual data in the form of X-rays.

Furthermore, in natural surroundings, image labeling can also be framed as a multi-label classification problem, indicating objects present in the images.

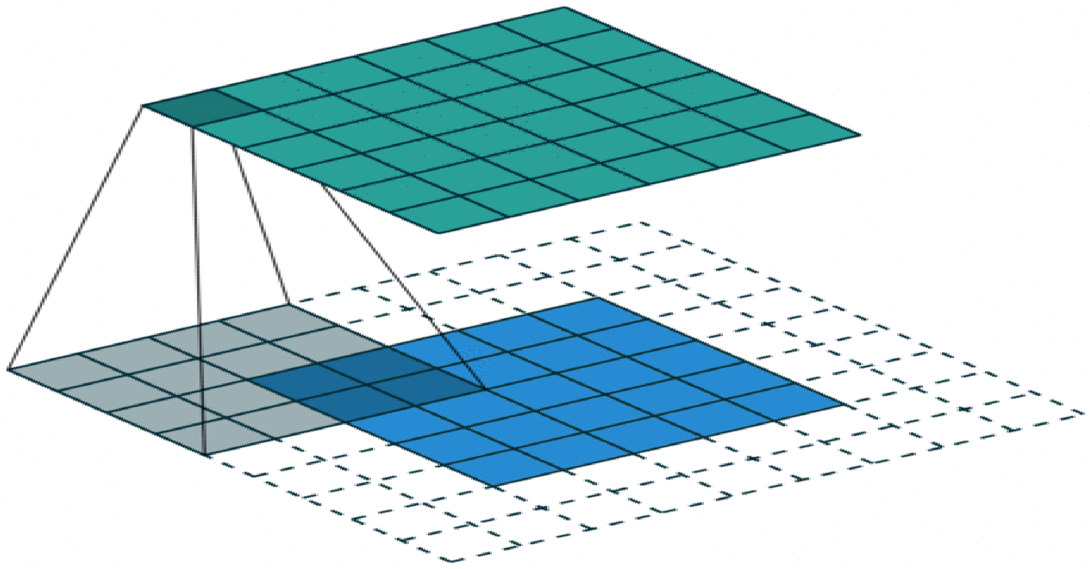
Image classification works on:

- Data collection
- Models
- Training
- Metrics

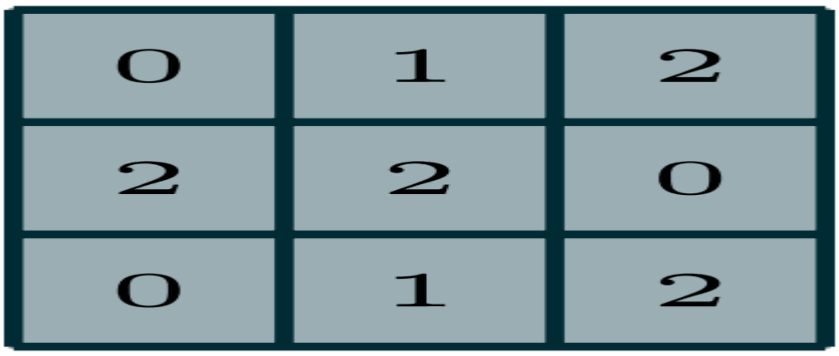
DATA COLLECTION: Due to the simplicity of the task, creating a high-quality dataset for image classification is quicker than other tasks, e.g., image segmentation.

Usually, a dataset is composed of images and a set of labels, and each image can have one or more labels. The most challenging part is to ensure the dataset is bias-free and balanced.

MODELS: In modern computer vision, the *gold-standard* model architecture to use is a convolutional neural network.The convolution operation slides a matrix, called kernel, on input while performing pair-wise matrix multiplication and summing up the result. The following gif shows this process.

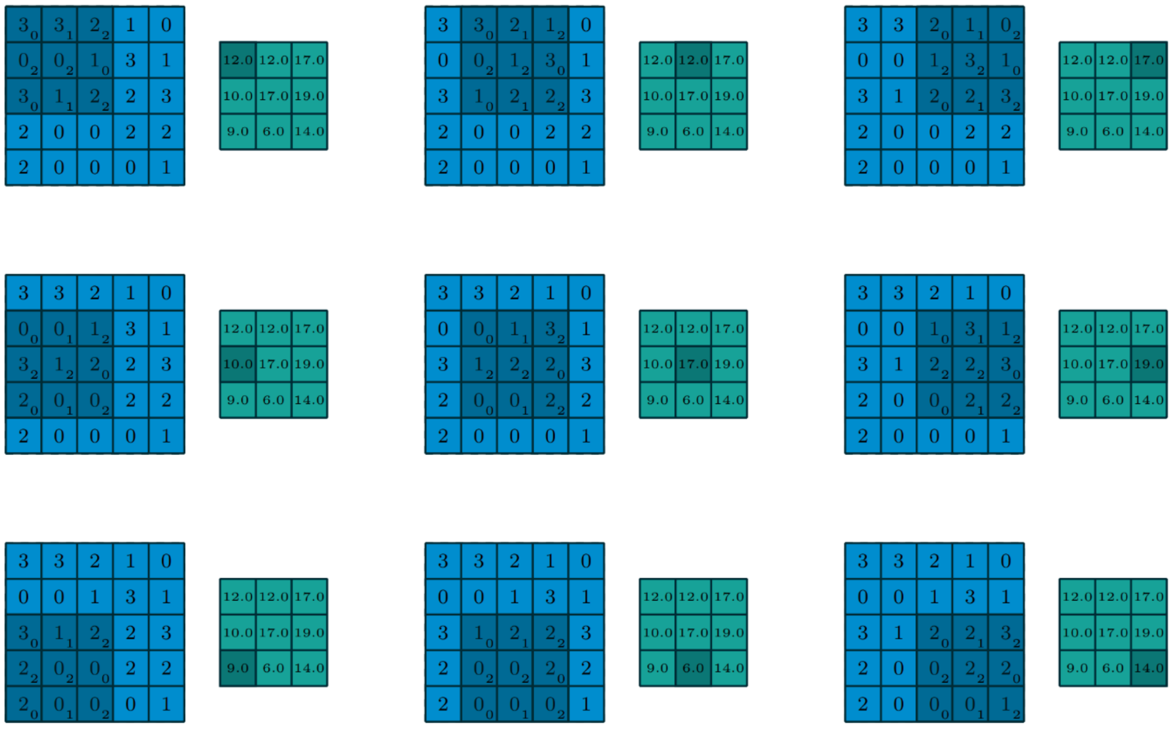


Let's see an example with numbers.
Assuming we have the following 3x3 kernel.



0	1	2
2	2	0
0	1	2

To apply the filter we have to imagine hovering it on the input matrix starting from top-right, multiplying each value with the kernel's corresponding value, and summing them up. Then we slide the kernel by a defined amount, called the *stride*, in this case, one, and we repeat the process.



For example, the first step is to multiply and sump up:
3*0 + 3*1 + 2*2 + 0*2 + 0*1 + 3 *0 + 1*1 + 2*2

he number of kernels applied at the same time is referred to as *features*. Imagine we apply four 3x3 kernels to an image; the resulting matrix will have four features.

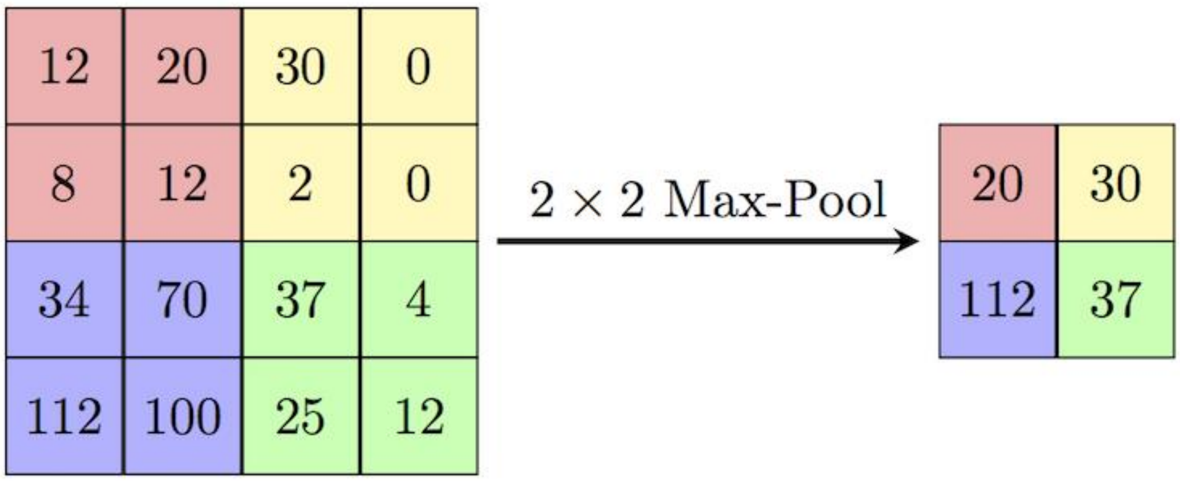
In practice, we randomly initialize the kernel's numbers, stack more of them together and update them during training using gradient descent.

One key aspect of the convolution operation is *weights sharing;* you may have noticed that the same kernel, the grey matrix, is reused in each step covering the whole image. This drastically reduces the number of weights needed, lowering computational cost.

Moreover, this introduces a positional bias since we teach the network to look at the input matrix by aggregating pixels close to each other. This helps because neighbor pixels are usually semantically connected.

Usually, in neural networks, multiple blocks of convolutional operation are stacked together to form a layer. Multiple layers are then composed together, like LEGOs, to form the final model. The number of layers is called *depth;* the more layers, the deeper the network.

To further reduce computation overhead, different kinds of poolings are applied between layers. A pool operation is a very simple thing. It takes a matrix and reduces it by aggregating its value. The most common one is *max pooling* where you define a window size and take the max inside it.



The final vector resulting by feeding a batch of images in this series of stacked convolution layers is then passed to a fully connected layer that outputs a vector with the classes.

MLPs (Multilayer Perceptron) use one perceptron for each input (e.g. pixel in an image) and the amount of weights rapidly becomes unmanageable for large images. It includes too many parameters because it is fully connected. Each node is connected to every other node in next and the previous layer, forming a very dense web — resulting in redundancy and inefficiency. As a result, difficulties arise whilst training and *overfitting* can occur which makes it lose the ability to generalize.
Another common problem is that MLPs react differently to an input (images) and its shifted version — they are not translation invariant. For example, if a picture of a cat appears in the top left of the image in one picture and the bottom right of another picture, the MLP will try to correct itself and assume that a cat will always appear in this section of the image.
Hence, MLPs are not the best idea to use for image processing. One of the main problems is that spatial information is lost when the image is flattened(matrix to vector) into an MLP.
We thus need a way to leverage the spatial correlation of the image features (pixels) in such a way that we can see the cat in our picture no matter where it may appear. Solution? — **CNN** !
Convolutional Neural Network (CNN): More generally, CNNs work well with data that has a spatial relationship. Therefore CNNs are go-to method for any type of prediction problem involving image data as an input.
The benefit of using CNNs is their ability to develop an internal representation of a two-dimensional image. This allows the model to learn position and scale in variant structures in the data, which is important when working with images.

AlexNet
AlexNet was the first real convolutional neural network to achieve superhuman abilities. In 2012 it won the ImageNet large Scale Visual Recognition Channalnge by a 10% margin.
It is composed of stacked convolutional layers separated by max pooling

VGG
Very Deep Convolutional Networks **(**VGG) was the winner of 2014 ImageNet challenge. It is deeper than AlexNet pushing the depth up to 19 layers. It utilizes a smaller 3x3 kernel size filter, the standard today.
Inception V1-V3
Inception V1 was a family of networks developed by google. They have parallel connections in which the input flows in different convolutional blocks on different kernel sizes independently and then its output is aggregated. Inception V3 won the ImageNet competition in 2015
ResNet
ResNet introduced residual connection allowing the network to scale up to 150 layers. ResNet-50 is the most widely used backbone for computer vision tasks in the industry, due to its good accuracy and medium parameter count.
Multiple variants of ResNet have been proposed in the following years, some of the most interesting: Se-ResNet, ResNetXt, ResNeST and RegNet
So, we got our data and our model, but how do we train it to successfully classify images?
Well, first of all, you need to know that there are two main categories of *learning*.
Supervised Learning
When we provide our model with training errors signals, e.g. you classify this image as a cat but it was a dog, we perform supervised learning. This is the most common scenario in which we have labeled datasets with image and class pairs.
Neural Networks are trained by minimizing a function, called *loss*, using gradient descend.
For single-label classification, we rely on the, defined as follows:

$$L = -\frac{1}{m} \sum [y_i \ln(\hat{y}_i) + (1 - y_i) \ln(1 - \hat{y}_i)]$$

Binary Cross-Entropy Loss mathematically
Don't be scared by the notation, *m* is the number of samples, *y_i* is the label, *hat y_i* is the prediction and *ln* is the natural logarithm.
Softmax
The model outputs a vector with a length equal to a number of classes, they are the classes' scores. Since they can be anything, we apply a special function called Softmax.
It makes these numbers sum up to one, so each individual item is bounded between 0 and 1. The more close to one the model thinks that class is the correct one.
The Softmax is defined as follows.

$$\text{Softmax}(\hat{y}_i) = \frac{e^{\hat{y}_i}}{\sum e^{\hat{y}_j}}$$

Softmax mathematically
So we take the exponential of the output and we divide by the sum of the exponential of all the outputs. Let's take an example, imagine we have two classes and our output is:

$$\hat{y} = [2, 5]$$

Output value

Clearly, the value for class 2, *hat y_2*, is the biggest one, but we really want them to be between 0 and 1 (you'll see it later). So let's apply the function, first let's find the denominator.

$$\sum e^{\hat{y}_j} = 7.3 + 148 = 155$$

Then we take the exponential of each output and we divide by it.

$$\text{Softmax}(\hat{y}) = [\frac{e^2}{155}, \frac{e^5}{155}] = [0.05, 0.95]$$

Dividing by the exponential of each output

The number sums up to one and they capture the correct score the model assigned previously. You may are wondering why we don't just normalize the output.

Here's the answer—

If we normalize the previous values we obtain [0.28, 0.71], while if with Softmax [0.05, 0.95]. Softmax pushes the values more away than vanilla normalization, effectively almost taking the maximum value, this is from where the name S*oftmax* comes from.

Cross Entropy

Let's take an example if our target is 0 but our model output 0.9, so very close to 1, the loss value would be.

Remember, in this case, the model outputs a vector with a length equal to a number of classes, they are the class's score. These numbers sum up to one, so each individual item is bounded between 0 and 1.

The more to close to one the model thinks that class is the correct one.

$$L = -(0 - 2.30)$$

If you know about regression you may think that we can apply Mean Square Error and call if a day.

MSE doesn't penalize the model as much as BCE does. If we compute the gradient with respect to *hat y_i*.

$$\frac{\delta L}{\delta \hat{y}_i} = \frac{y_i}{\hat{y}_i} + \frac{1-y_i}{1-\hat{y}_i} = 0 + \frac{1}{0.1} = 10$$

While with MSE:

$$L_{MSE} = (y_i - \hat{y}_i)^2|$$

$$\frac{\partial L_{MSE}}{\partial \hat{y}_i} = 2 * (0 - 0.9) = 1.8$$

The gradient from BCE is 5 times more, thus the model will be penalized more.

Multi-Labels Classification

For multi-labels classification we want our outputs to be independent of each other since multi classes can exist at the same time. For example, if we have two classes at the same time our output could be [20, 21], we cannot apply Softmax since it will give more importance to one specific value, the biggest one.

In this case, we apply the Sigmoid function before BCE that maps between [-1,1], it is defined as follows.

Sigmoid / Logistic

$$f(x) = \frac{1}{1 + e^{-x}}$$

This will make sure that each individual value in our model outputs is bounded by the same range.

Unsupervised Learning

Here we don't have annotations, just a big collection of raw images and we let our model figure out what they are during training.

There are different approaches to solving this task—let's go through each one of them.

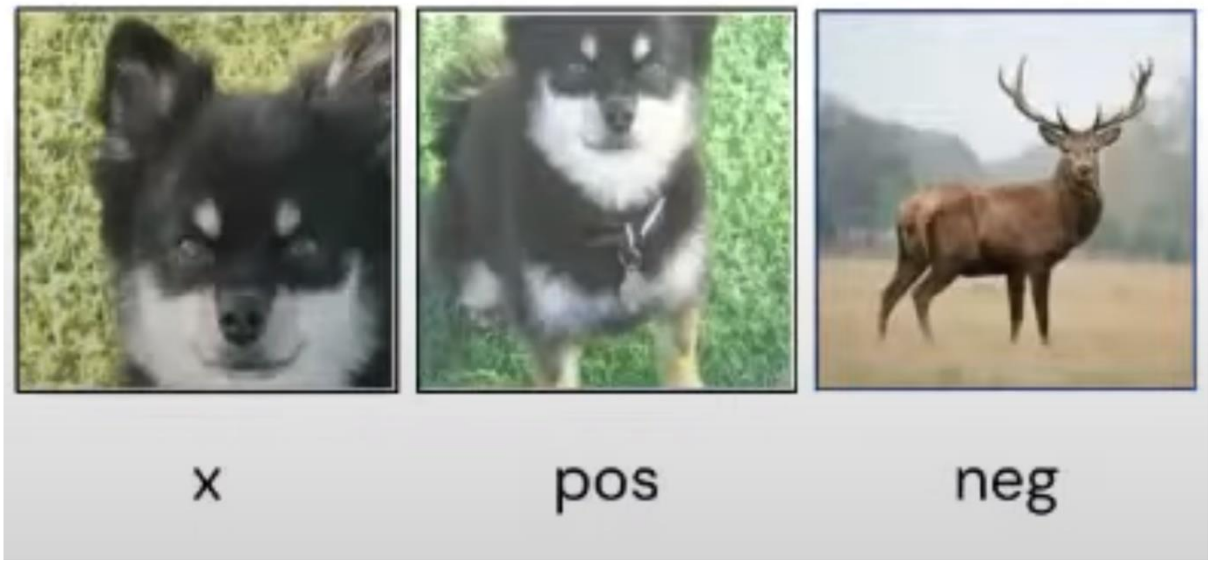
Contrastive learning

In contrastive learning, we take an input image, x, sample a “positive” image, *x_pos*, and a “negative” image, *x_pos*, in some way and try to train a scoring model such that *score(x, x_pos)* > *score(x, x_neg)*

A sample with positive and negative images

We draw these samples randomly by enforcing that each time we have a different positive and negative example.

If we have labels, the negative sample must be from a different class, while the positive one is from the same one. When there are missing labels, one easy way to get the sample is to sample two parts of the same image for the positive and one part of another random image for the negative.



Sample of the same image negative and positive

Given a vector representation of an image, we try to push the two positive as close as possible and the negative them far away. One way is to learn a similarity metric, *D*.

D is the distance between these representations.

$$D_i = ||x, x_i||_2$$

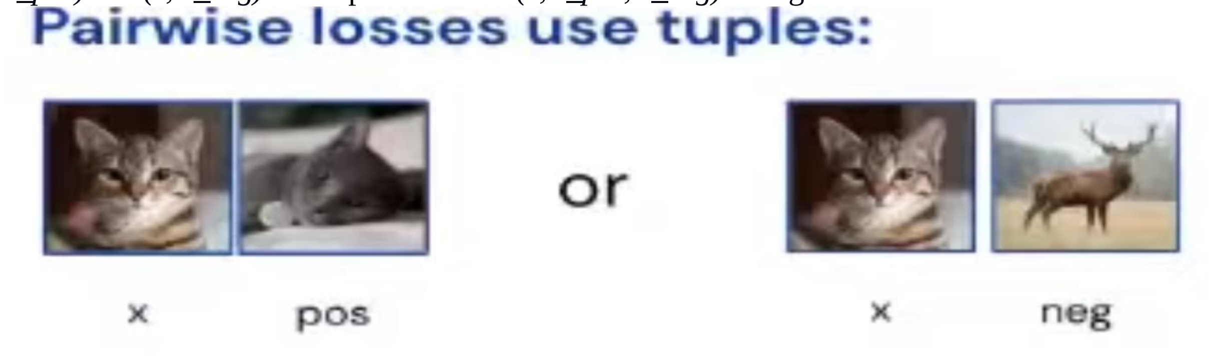
Distance between the positive and negative representations
If positive, we want our loss *L* to push them close.

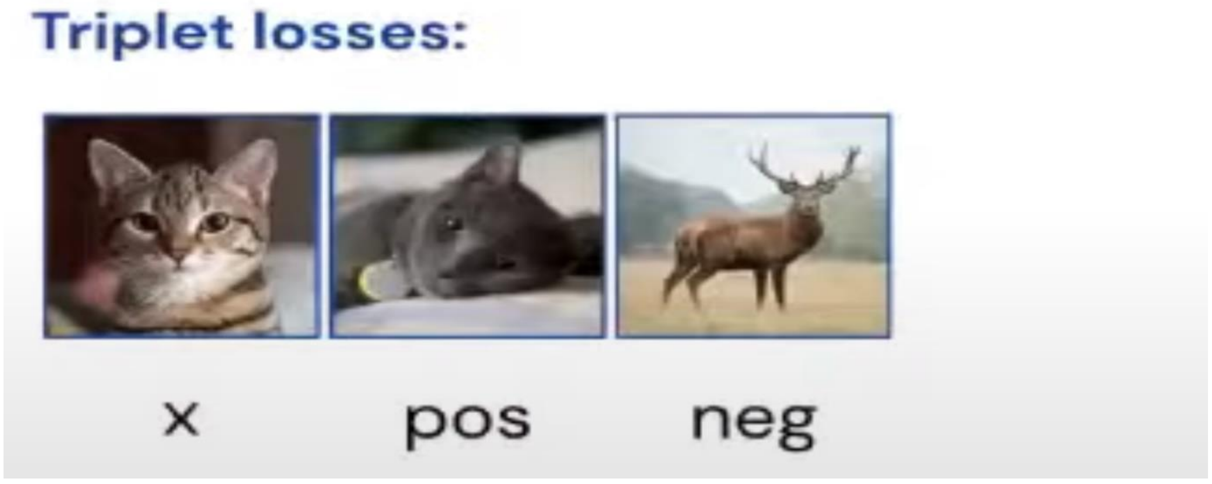
$$L_i = D_i^2$$

So we want *x* and *x_pos* to have a big dot product. On the other hand, if we have a negative sample we do the opposite.

$$L_i = \max(0, \epsilon - D_i))^2$$

The epsilon acts like a margin value.
Depending on how we decide to sample our example, we can have a pairwise loss, so with *(x, x_pos)* and *(x, x_neg)* or a triplet loss with *(x, x_pos, x_neg)* all together.





With Tripet losses, we first compute the score between *x,_pos* and *x* and *x_neg*, and then we use them in the loss.

$$D_{\text{pos}} = ||x, x_{\text{pos}}||^2$$

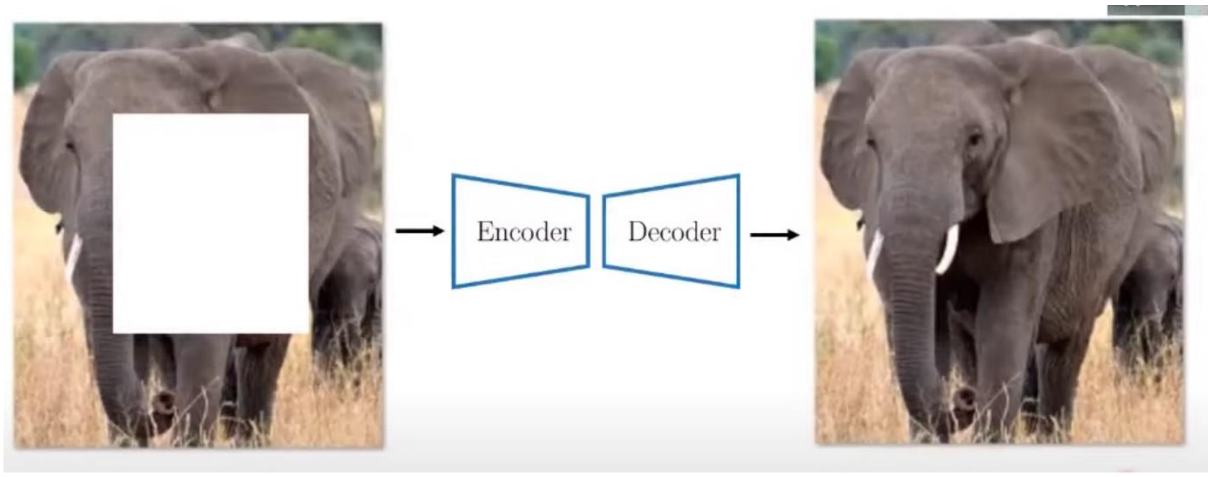
$$D_{\text{neg}} = ||x, x_{\text{neg}}||^2$$

$$L = \max(0, D^2_{\text{pos}} - D^2_{\text{neg}} - m)$$

Our variable *m* acts like a margin.
Another very common loss is InfoNCE in which here we can have multiple negative examples.
InfoNCE is the log-likelihood to predict the correct example from a given set of examples based on the score of the model.

$$\text{InfoNCE} = -\mathbb{E}[\log \frac{s(x,y)}{\sum_{y_i} s(x,y_i)}]$$

InfoNCE mathematically
The score function, *s*, is usually the inner product. The higher the more similar they are and vice versa.
One very successful method that uses contrastive learning is SimCLR in which you have a network *f*, usually a ResNet-50, and a linear projection, *g*.
The way it works is simple—
We sample a minibatch from the dataset and for each data point, *x*, we apply data augmentation and pass them through the network, *f*, and the linear projection obtained two representations. This effectively creates two positive pairs since we augmented the same images, then we apply a contrastive loss on all the augmented data points in the batch keeping track of the positive pairs.
Generative Models
Another idea is to tackle the unsupervised problem by using generative networks.
The idea is to give the networks reconstruction images tasks.
For example, we can crop some part of the image and train the network to fill in the gap.



Generative model

This task requires the network to develop context-aware skills to reconstruct the missing input. Unfortunately, there are some limitations to this approach.

For example, there are way too many pixels details that the network has to learn that are not essentials, like the color.

We can overcome these issues by using some ideas from contrastive learning, like applying an augmentation to create a pair of positive examples and training the network to reconstruct the permuted example.

However, there are some caveats here as well—

To make this work we also need a lot of negative examples, which is not ideal.

Teacher student models

One of the most efficient methodologies is the teacher-student approach.

The teacher and the student are two networks, usually the same. An input image, *x*, is first augmented in two different ways and then each of the new augmented images is passed to the teacher and student respectively.

They generate two features vector that we score with a reconstruction loss, like Mean Square Error or Cosine Similarity. Since they come from the same image, they are a positive pair and we want the result features from both models to be as similar as possible.

We don't need negative examples.

One approach that we want to highlight here is BYOL.

In BYOL, at each training step, after creating two augmented images from a data point and computing their score in the features space, the parameters of the student are updated using gradient descent while the teacher's ones are updated with the exponential moving average of the student's weights.

Even though it is a very straightforward procedure, it is highly effective.

Image Classification Metrics

Image Classification models have to be evaluated to determine how well they perform in comparison to other models.

Here are some well-known metrics used in Image Classification.

Precision

Precision is a metric that is defined for each class. Precision in a class tells us what proportion of data predicted by the ML model to belong to the class was actually part of the class in the validation data.

A simple formula can demonstrate this:

Precision= TPTP+FP

Where:

- TP represents True Positives: The number of samples that were predicted to belong to a class and correctly belong to the class.

- FP represents False Positives: The number of samples that were predicted to belong to a class while they do not belong to the class at all.

Recall

Recall similar to precision is defined for each class.
Recall tells us what proportion of the data from the validation set belonging to the class was identified correctly (as belonging to the class).
Recall can be represented as:

Recall=TPTP+FN

Where:

- FN represents False Negatives: The number of samples that the model predicted as not belonging to a class while they actually belong to that particular class.

F1 Score

F1 Score helps us achieve a balance between precision and recall to get an average idea of how the model performs.
F1 score as a metric is calculated as follows.

$$\text{F1 Score} = \frac{2 \text{ x } (\text{Precision x Recall})}{\text{Precision + Recall}}$$

Precision and Recall scores largely depend on the problem the classification model is trying to address.
Recall is a critical metric, particularly in problems referring to the medical image analysis, like detection of pneumonia from chest X-rays, where false negatives cannot be present to prevent diagnosing a patient as healthy when they actually have the disease.
Precision is needed where the false positives have to be avoided, like email spam detection. If an important email is classified as spam, then a user would face significant issues.

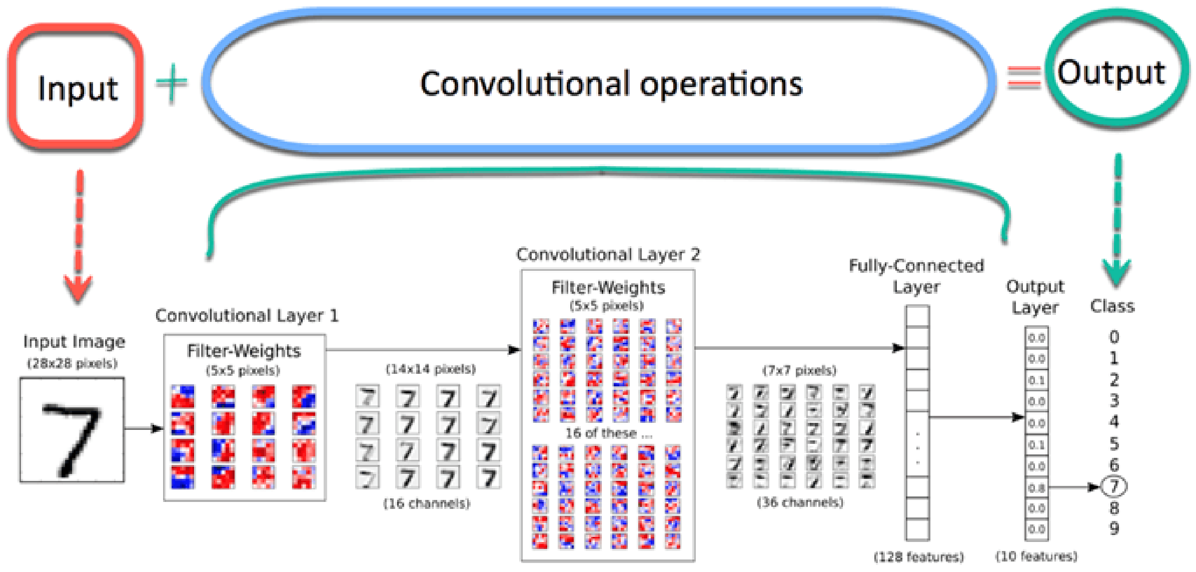
CNN ARCHITECTURE:

Convolution Layer
Convolutional Neural Network**,** also known as convnets or CNN, is a well-known method in computer vision applications. It is a class of deep neural networks that are used to analyze visual imagery. This type of architecture is dominant to recognize objects from a picture or video. It is used in applications like image or video recognition, neural language processing, etc.
The Conv layer is the core building block of a CNN.The parameters consist of a set of learnable filters. Every filter is small spatially (width and height), but extends through the full depth of the input volume, eg, 5x5x3.During the forward pass, we slide (convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. Produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature.A set of filters in each CONV layer – each of them will produce a separate 2-dimensional activation map – We will stack these activation maps along the depth dimension and produce the output volume.

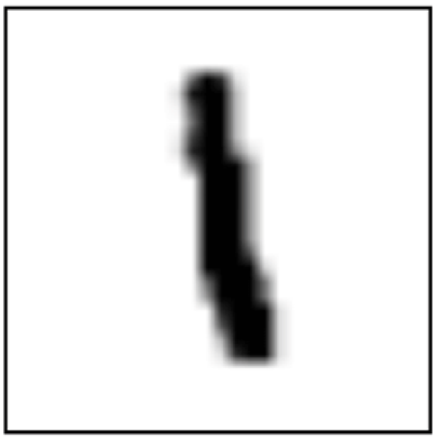
Convolutional Neural Network, also known as convnets or CNN, is a well-known method in computer vision applications. It is a class of deep neural networks that are used to analyze visual imagery. This type of architecture is dominant to recognize objects from a picture or video. It is used in applications like image or video recognition, neural language processing, etc.

Architecture of a Convolutional Neural Network:
A convolutional neural network for image classification is not very difficult to understand. An input image is processed during the convolution phase and later attributed a label.

A typical convnet architecture can be summarized in the picture below. First of all, an image is pushed to the network; this is called the input image. Then, the input image goes through an infinite number of steps; this is the convolutional part of the network. Finally, the neural network can predict the digit on the image.



An image is composed of an array of pixels with height and width. A grayscale image has only one channel while the color image has three channels (each one for Red, Green, and Blue). A channel is stacked over each other. In this tutorial, you will use a grayscale image with only one channel. Each pixel has a value from 0 to 255 to reflect the intensity of the color. For instance, a pixel equals to 0 will show a white color while pixel with a value close to 255 will be darker.
Let’s have a look at an image stored in the MNIST dataset The picture below shows how to represent the picture of the left in a matrix format. Note that, the original matrix has been standardized to be between 0 and 1. For darker color, the value in the matrix is about 0.9 while white pixels have a value of 0.



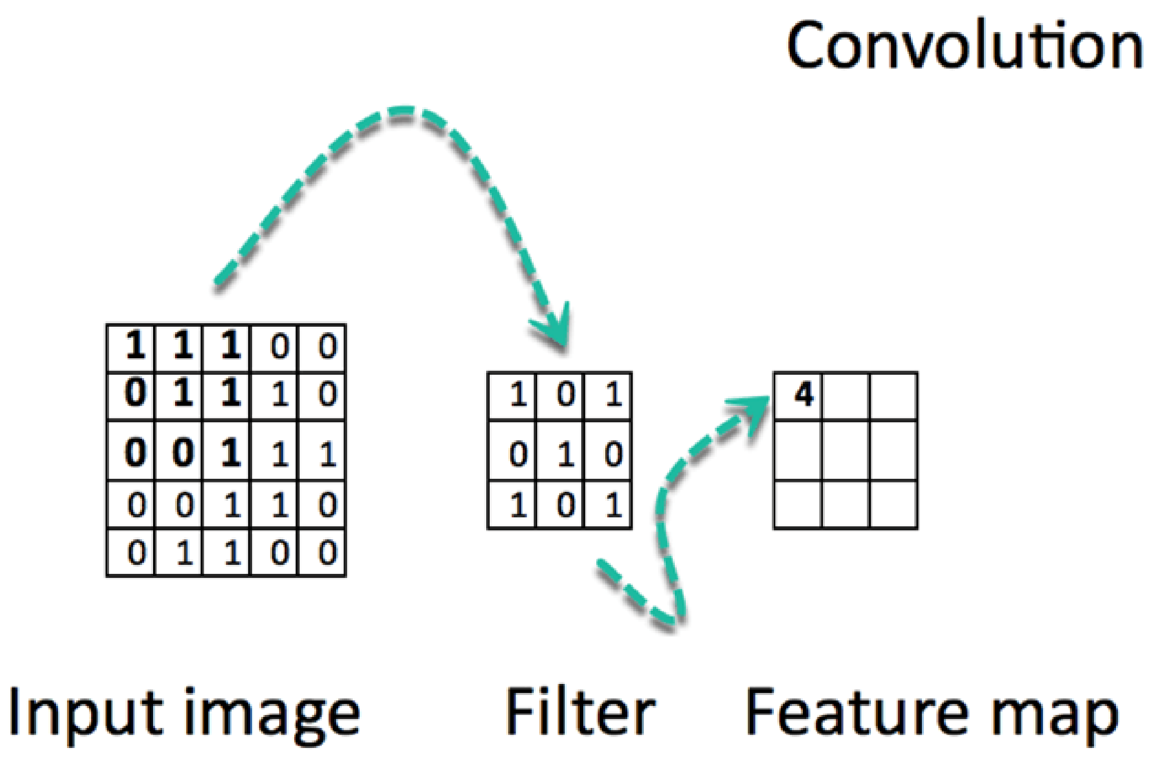
$$\simeq \begin{bmatrix}
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&.6&.8&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.5&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.7&0&0&0&0&0\\
0&0&0&0&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&.9&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&.3&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}$$

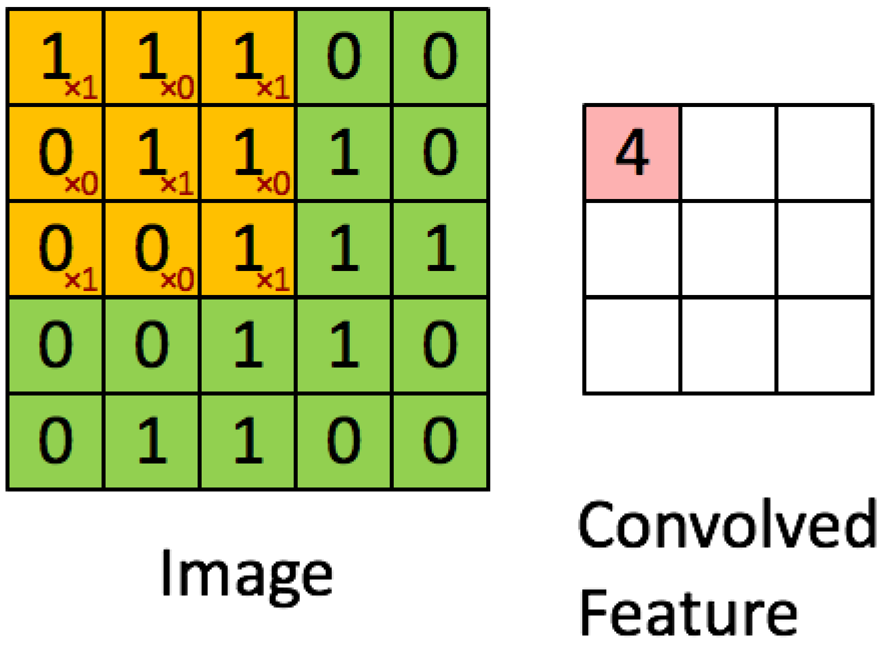
The most critical component in the model is the convolutional layer. This part aims at reducing the size of the image for faster computations of the weights and improve its generalization.
During the convolutional part, the network keeps the essential features of the image and excludes irrelevant noise. For instance, the model is learning how to recognize an elephant from a picture with a mountain in the background. If you use a traditional neural network, the model will assign a weight to all the pixels, including those from the mountain which is not essential and can mislead the network.
Instead, a Keras convolutional neural network will use a mathematical technique to extract only the most relevant pixels. This mathematical operation is called convolution. This technique allows the network to learn increasingly complex features at each layer. The convolution divides the matrix into small pieces to learn to most essential elements within each piece.
Components of Convolutional Neural Network:

There are four components of a Convnets

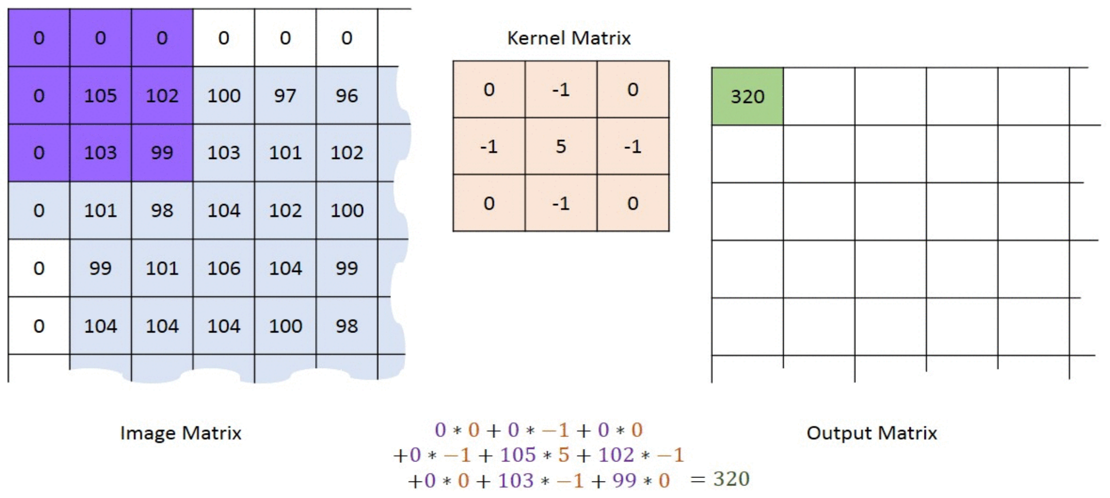
1. Convolution
2. Non Linearity (ReLU)
3. Pooling or Sub Sampling
4. Classification (Fully Connected Layer)

Convolution
The purpose of the convolution is to extract the features of the object on the image locally. It means the network will learn specific patterns within the picture and will be able to recognize it everywhere in the picture.
Convolution is an element-wise multiplication. The concept is easy to understand. The computer will scan a part of the image, usually with a dimension of 3×3 and multiplies it to a filter. The output of the element-wise multiplication is called a feature map. This step is repeated until all the image is scanned. Note that, after the convolution, the size of the image is reduced.



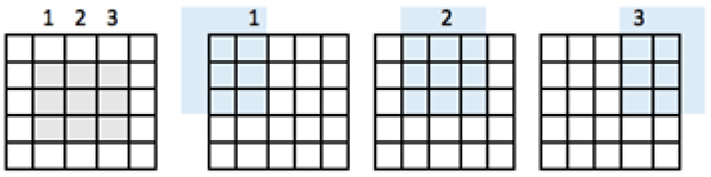


There are numerous channels available. Below, we listed some of the channels. You can see that each filter has a specific purpose. The convolutional phase will apply the filter on a small array of pixels within the picture. The filter will move along the input image with a general shape of 3×3 or 5×5. It means the network will slide these windows across all the input image and compute the convolution. The image below shows how the convolution operates. The size of the patch is 3×3, and the output matrix is the result of the element-wise operation between the image matrix and the filter.

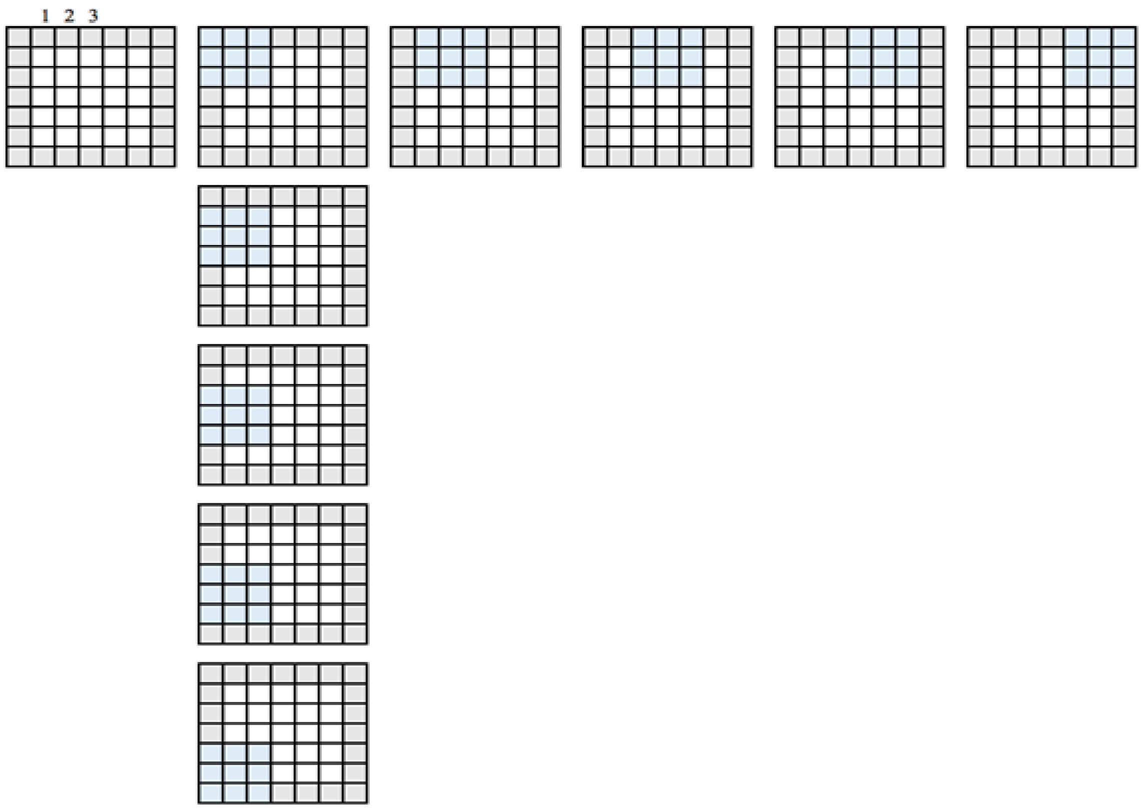


Convolution with horizontal and vertical strides = 1

Image has a 5×5 features map and a 3×3 filter. There is only one window in the center where the filter can screen an 3×3 grid. The output feature map will shrink by two tiles alongside with a 3×3 dimension.

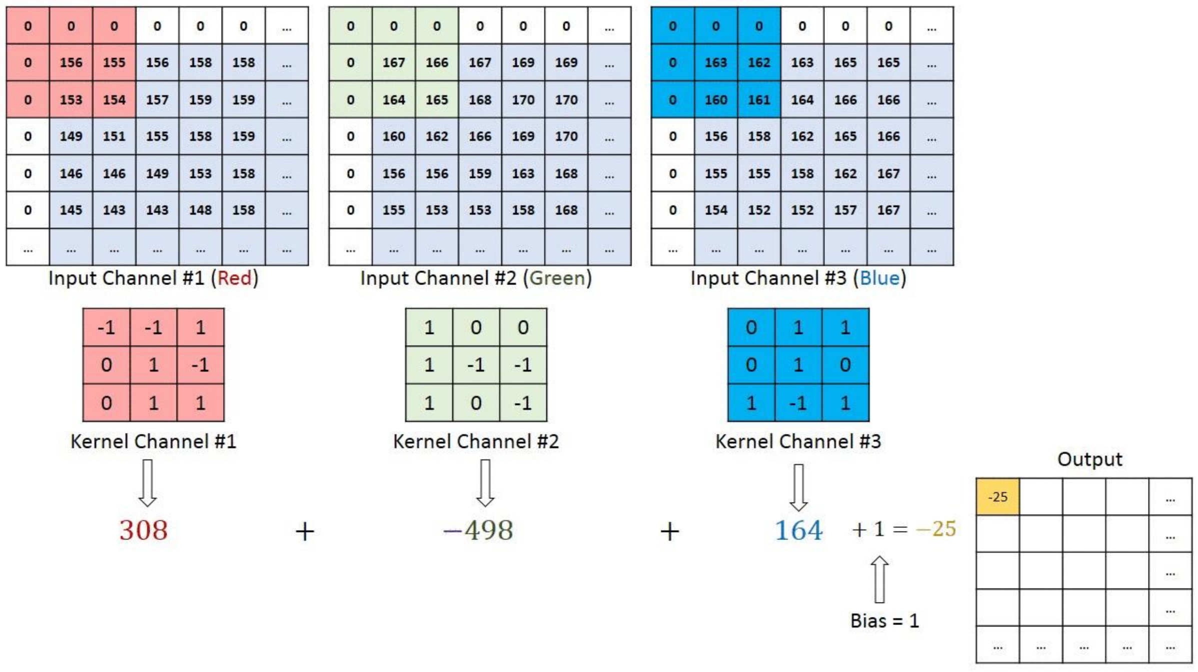


To get the same output dimension as the input dimension, you need to add padding. Padding consists of adding the right number of rows and columns on each side of the matrix. It will allow the convolution to center fit every input tile. In the image below, the input/output matrix have the same dimension 5×5



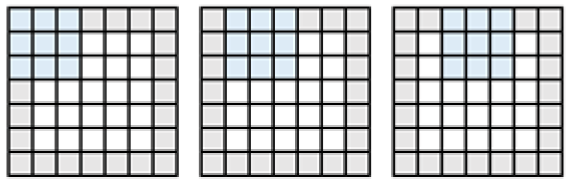
When you define the network, the convolved features are controlled by three parameters:

1. **Depth**: It defines the number of filters to apply during the convolution. In the previous example, you saw a depth of 1, meaning only one filter is used. In most of the case, there is more than one filter. The picture below shows the operations done in a situation with three filters

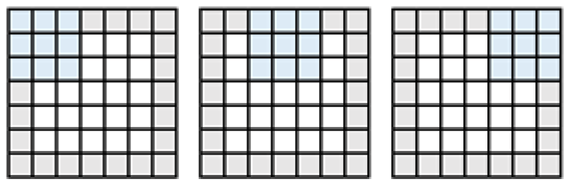


2. **Stride**: It defines the number of "pixel's jump" between two slices. If the stride is equal to 1, the windows will move with a pixel's spread of one. If the stride is equal to two, the windows will jump by 2 pixels. If you increase the stride, you will have smaller feature maps.

Example stride 1



stride 2



3. **Zero-padding**: A padding is an operation of adding a corresponding number of rows and column on each side of the input features maps. In this case, the output has the same dimension as the input.

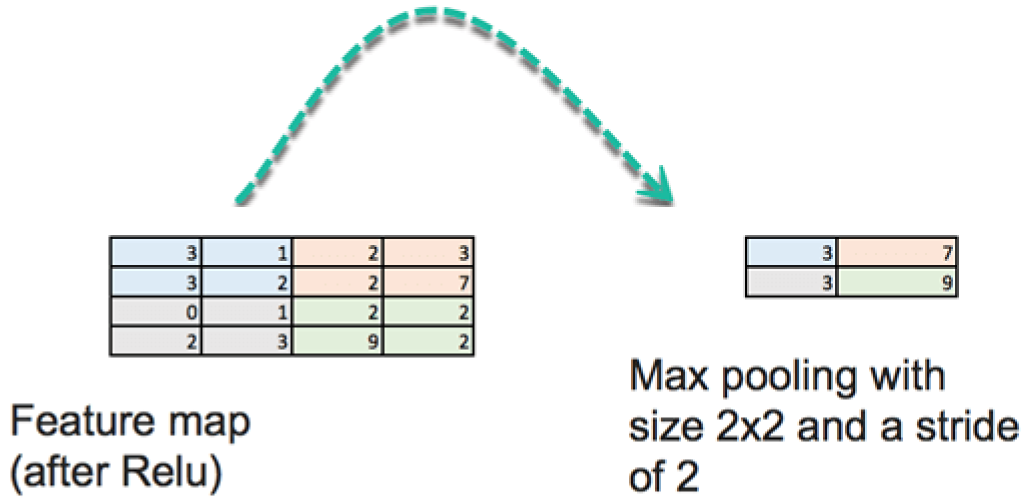
Non Linearity (ReLU)

At the end of the convolution operation, the output is subject to an activation function to allow non-linearity. The usual activation function for convnet is the Relu. All the pixel with a negative value will be replaced by zero.

Pooling Operation

This step is easy to understand. The purpose of the pooling is to reduce the dimensionality of the input image. The steps are done to reduce the computational complexity of the operation. By diminishing the dimensionality, the network has lower weights to compute, so it prevents overfitting.

In this stage, you need to define the size and the stride. A standard way to pool the input image is to use the maximum value of the feature map. Look at the picture below. The "pooling" will screen a four submatrix of the 4×4 feature map and return the maximum value. The pooling takes the maximum value of a 2×2 array and then move this windows by two pixels. For instance, the first sub-matrix is [3,1,3,2], the pooling will return the maximum, which is 3.



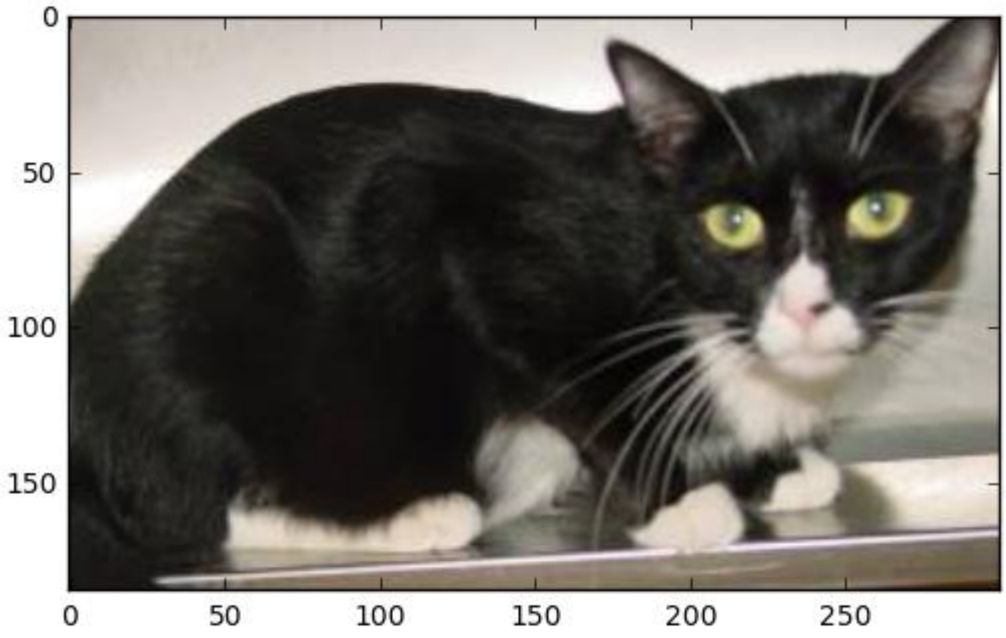
There is another pooling operation such as the mean.
This operation aggressively reduces the size of the feature map
Fully Connected Layers

The last step consists of building a traditional ANN,Connecting all neurons from the previous layer to the next layer. You use a softmax activation function to classify the number on the input image.

Underfitting vs. Overfitting
How do you know if your model is underfitting? Your model is underfitting if the accuracy on the validation set is higher than the accuracy on the training set. Additionally, if the whole model performs bad this is also called underfitting. For example, using a linear model for image recognition will generally result in an underfitting model. Alternatively, when experiencing underfitting in your deep neural network this is probably caused by dropout. Dropout randomly sets activations to zero during the training process to avoid overfitting. This does not happen during prediction on the validation/test set. If this is the case you can remove dropout. If the model is now massively overfitting you can start adding dropout in small pieces.
Overfitting happens when your model fits too well to the training set. It then becomes difficult for the model to generalize to new examples that were not in the training set. For example, your model recognizes specific images in your training set instead of general patterns. Your training accuracy will be higher than the accuracy on the validation/test set. So what can we do to reduce overfitting?
Steps for reducing overfitting:

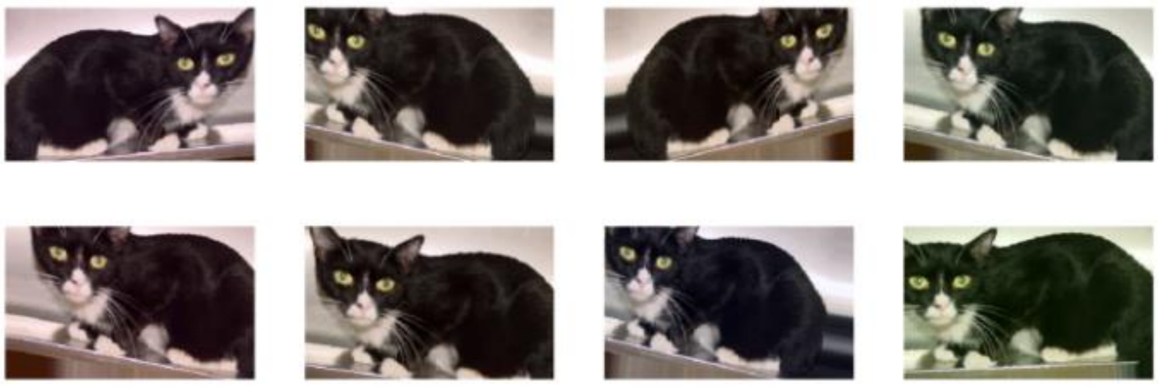
1. Add more data
2. Use data augmentation
3. Use architectures that generalize well
4. Add regularization (mostly dropout, L1/L2 regularization are also possible)
5. Reduce architecture complexity.

The first step is of course to collect more data. However, in most cases you will not be able to. Let’s assume you have collected all the data. The next step is data augmentation: something that is always recommended to use.Data augmentation includes things like randomly rotating the image, zooming in, adding a color filter etc. Data augmentation only happens to the training set and not on the validation/test set. It can be useful to check if you are using too much data augmentation. For example, if you zoom in so much that features of a cat are not visible anymore, than the model is not going to get better from training on these images.



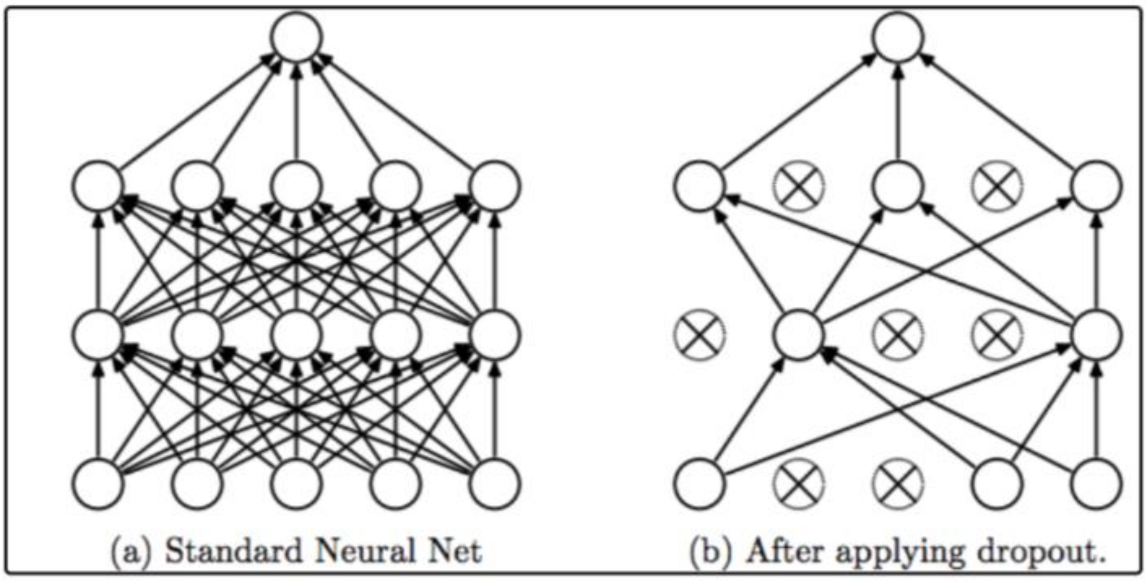
Original image

Let’s now have a look at the image after performing data augmentation. All of the ‘cats’ are still clearly recognizable as cats



Augmented cats

The third step is to use an architecture that generalizes well. However, much more important is the fourth step: adding regularization. The three most popular options are: dropout, L1 regularization and L2 regularization. In deep learning you will mostly see dropout, which I discussed earlier. Dropout deletes a random sample of the activations (makes them zero) in training. In the Vgg model this is only applied in the fully connected layers at the end of the model. However, it can also be applied to the convolutional layers. Be aware that dropout causes information to get lost. If you lose something in the first layer, it gets lost for the whole network. Therefore, a good practice is to start with a low dropout in the first layer and then gradually increase it. The fifth and final option is to reduce the network complexity. In reality, various forms of regularization should be enough to deal with overfitting in most cases.



Visualization of dropout

Batch Normalization

Finally, let’s discuss batch normalization. This is something you should always do! Batch normalization is a relatively new concept and therefore was not yet implemented in the Vgg model.

Standardizing the inputs of your model is something that you have definitely heard about if you are into machine learning. Batch normalization takes this a step further. Batch normalization adds a ‘normalization layer’ after each convolutional layer. This allows the model to converge much faster in training and therefore also allows you to use higher learning rates.

Simply standardizing the weights in each activation layer will not work. Stochastic Gradient Descent is very stubborn. If wants to make one of the weights very high it will simply do it the next time. With batch normalization the model learns that it can adjust all the weights instead of one each time.

Defining performance metrics

Performance metrics allow us to evaluate our system. When we develop a model, wewant to find out how well it is working. The simplest way to measure the “goodness”of our model is by measuring its accuracy. The accuracy metric measures how manytimes 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:

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

ACCURACY:

We have been using accuracy as a metric for evaluating our model in earlier projects, and it works fine in many cases. But let’s 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.

CONFUSION MATRIX:

To set the stage for other metrics, we will use a confusion matrix: a table that describes the performance of a classification model. The confusion matrix itself is relatively simple to understand, but the related terminology can be a little confusing at first. Once you understand it, you’ll find that the concept is really intuitive and makes a lot of sense. Let’s go through it step by step. 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).

 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).

PRECESION AND 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

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

following equation:

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.

DESIGN BASELINE MODEL:

Now that you have selected the metrics you will use to evaluate your system, it is time to establish a reasonable end-to-end system for training your model. Depending on the problem you are solving, you need to design the baseline to suit your network type and architecture. In this step, you will want to answer questions like these:

- Should I use an MLP or CNN network (or RNN, explained later in the book)?
- Should I use other object detection techniques like YOLO or SSD (explained inlater chapters)?
- How deep should my network be?
- Which activation type will I use?
- What kind of optimizer do I use?
- Do I need to add any other regularization layers like dropout or batch normalization to avoid overfitting?

If your problem is similar to another problem that has been studied extensively, you will do well to first copy the model and algorithm already known to perform the best for that task. You can even use a model that was trained on a different dataset for your own problem without having to train it from scratch. This is called transfer learning.For example, in the last chapter’s project, we used the architecture of the popular AlexNet as a baseline model. Figure 4.1 shows the architecture of an AlexNet deep CNN, with the dimensions of each layer. The input layer is followed by five convolutional layers (CONV1 through CONV5), the output of the fifth convolutional layer is fed into two fully connected layers (FC6 through FC7), and the output layer is a fully connected layer (FC8) with a softmax function:

INPUT ⇒ CONV1 ⇒ POOL1 ⇒ CONV2 ⇒ POOL2 ⇒ CONV3 ⇒ CONV4 ⇒ CONV5⇒ POOL3 ⇒ FC6 ⇒ FC7 ⇒ SOFTMAX_8

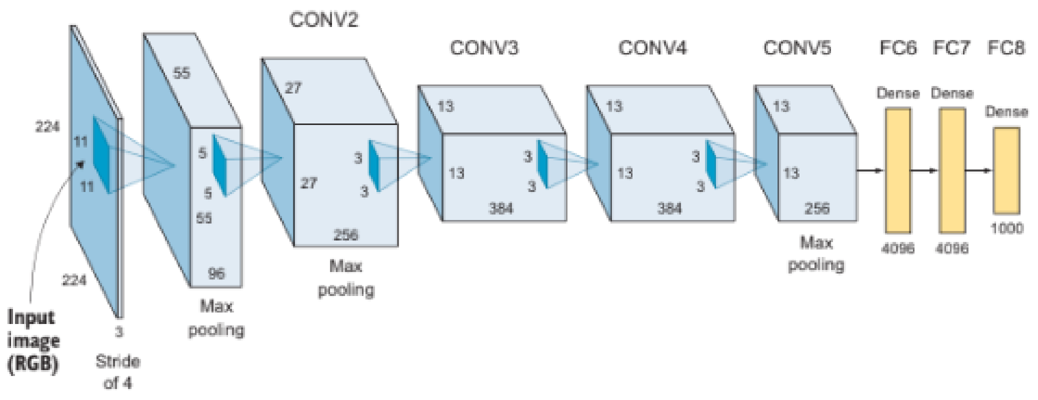


Figure 4.1 The AlexNet architecture consists of five convolutional layers and three FC layers.

 Network depth (number of layers): 5 convolutional layers plus 3 fully connected layers

 Layers' depth (number of filters): CONV1 = 96, CONV2 = 256, CONV3 = 384, CONV4 = 385, CONV5 = 256

 Filter size: 11 × 11, 5 × 5, 3 × 3, 3 × 3, 3 × 3

 ReLU as the activation function in the hidden layers (CONV1 all the way to FC7)

 Max pooling layers after CONV1, CONV2, and CONV5

 FC6 and FC7 with 4,096 neurons each

 FC8 with 1000 neurons, using a softmax activation function

IMPROVING THE NETWORK AND TUNING HYPERPARAMETERS

After you run your training experiment and diagnose for overfitting and underfitting,you need to decide whether it is more effective to spend your time tuning the network, cleaning up and processing your data, or collecting more data. The last thing you want to do is to spend a few months working in one direction only to find out that it barely improves network performance. So, before discussing the different hyperparameters to tune, let's answer this question first: should you collect more data?

Collecting more data vs. tuning hyperparameters

We know that deep neural networks thrive on lots of data. With that in mind, ML novices often throw more data to the learning algorithm as their first attempt to improve its performance. But collecting and labeling more data is not always a feasible option and, depending on your problem, could be very costly. Plus, it might not even be that effective.

In other scenarios, it is much better to collect more data than to improve the learning

algorithm. So it would be nice if you had quick and effective ways to figure out

whether it is better to collect more data or tune the model hyperparameters.

The process I use to make this decision is as follows:

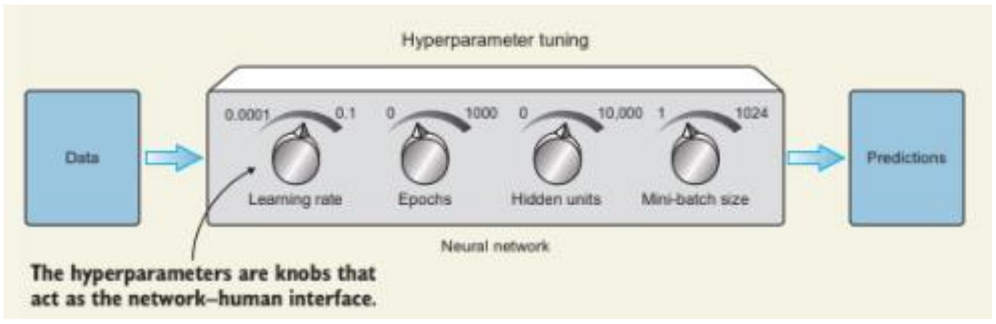
1 Determine whether the performance on the training set is acceptable as-is.

2 Visualize and observe the performance of these two metrics: training accuracy (train_acc) and validation accuracy (val_acc).

3 If the network yields poor performance on the training dataset, this is a sign of underfitting. There is no reason to gather more data, because the learning algorithm is not using the training data that is already available. Instead, try tuning the hyperparameters or cleaning up the training data.

4 If performance on the training set is acceptable but is much worse on the test dataset, then the network is overfitting your training data and failing to generalize to the validation set. In this case, collecting more data could be effective.

Hyperparameters are the variables that we set and tune. Parameters are the variables that the network updates with no direct manipulation from us. Parameters are variables that are learned and updated by the network during training, and we do not adjust them. In neural networks, parameters are the weights and biases that are optimized automatically during the backpropagation process to produce the minimum error. In contrast, hyperparameters are variables that are not learned by the network. They are set by the ML engineer before training the model and then tuned. These are variables that define the network structure and determine how the network is trained. Hyperparameter examples include learning rate, batch size, number of epochs, number of hidden layers, and others.



Generally speaking, we can categorize neural network hyperparameters into three main categories:

→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

Hyperparameters that define the neural network architecture:

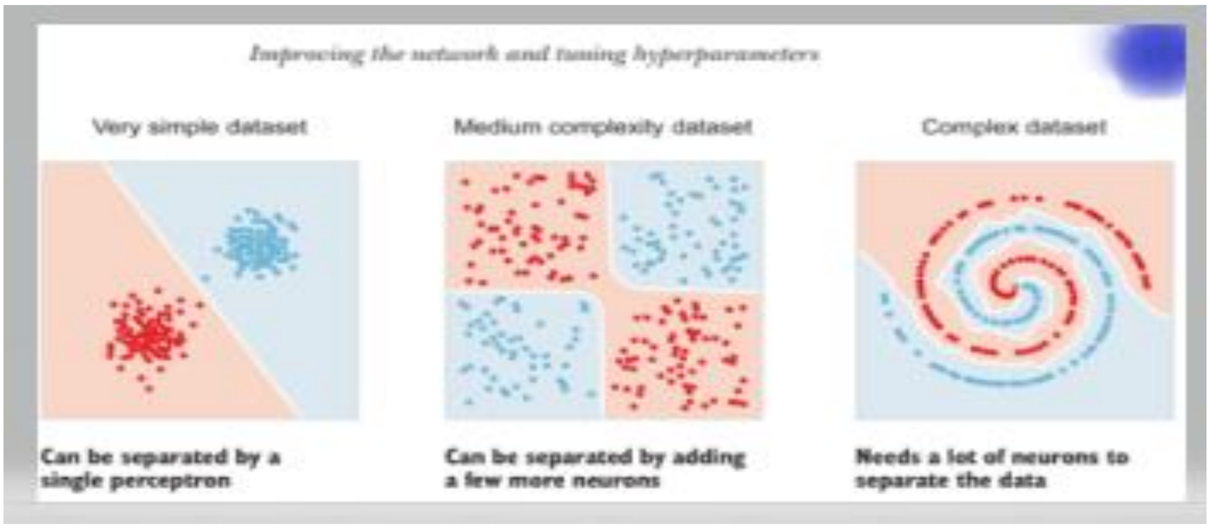
 Number of hidden layers (representing the network depth)

 Number of neurons in each layer, also known as hidden units (representing the network width)

 Activation functions

DEPTH AND WIDTH OF THE NEURAL NETWORK

Whether you are designing an MLP, CNN, or other neural network, you need to decide on the number of hidden layers in your 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.


Improving the network and tuning hyperparameters
Very simple dataset
Medium complexity dataset
Complex dataset
Can be separated by a single perceptron
Can be separated by adding a few more neurons
Needs a lot of neurons to separate the data

OPTIMIZATION ALGORITHMS:

Gradient descent with momentum

Recall that SGD ends up with some oscillations in the vertical direction toward the minimum error (figure 4.23). 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.

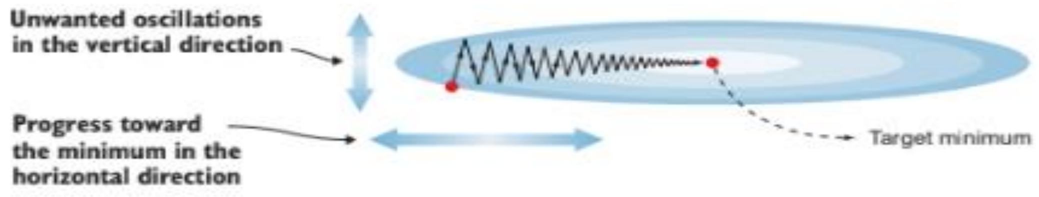

Unwanted oscillations in the vertical direction
Progress toward the minimum in the horizontal direction
Target minimum

Figure 4.23 SGD oscillates in the vertical direction toward the minimum error.

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.

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 techniques we have seen earlier.

Again, we have new hyperparameters to tune. But the good news is that the defaultvalues of major DL frameworks often work well, so you may not need to tune at all except for the learning rate, which is not an Adam-specific hyperparameter:

keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None,decay=0.0)

The authors of Adam propose these default values:

 The learning rate needs to be tuned.

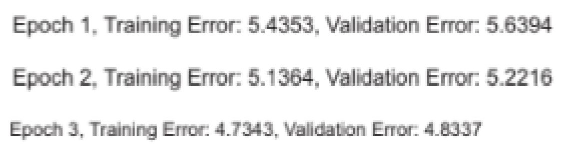
 For the momentum term β1, a common choice is 0.9.

 For the RMSprop term β2, a common choice is 0.999.

 ε is set to 10–8.

Number of epochs and early stopping criteria

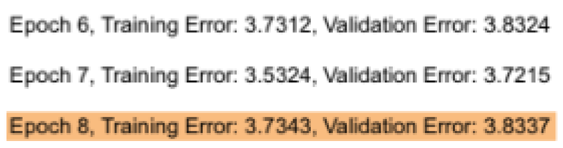
A training iteration, or epoch, is when the model goes a full cycle and sees the entire training dataset at once. The epoch hyperparameter is set to define how many iterations our network continues training. The more training iterations, the more our model learns the features of our training data. To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



```
Epoch 1, Training Error: 5.4353, Validation Error: 5.6394

Epoch 2, Training Error: 5.1364, Validation Error: 5.2216

Epoch 3, Training Error: 4.7343, Validation Error: 4.8337
```

Figure 4.24 **Sample verbose output of the first five epochs. Both training and validation errors are improving.**

You can see that both training and validation errors are decreasing. This means the network is still learning. It doesn't make sense to stop the training at this point. The network is clearly still making progress toward the minimum error.



```
Epoch 6, Training Error: 3.7312, Validation Error: 3.8324

Epoch 7, Training Error: 3.5324, Validation Error: 3.7215

Epoch 8, Training Error: 3.7343, Validation Error: 3.8337
```

Figure 4.25 **The training error is still improving, but the validation error started oscillating from epoch 8 onward.**

It looks like the training error is doing well and still improving. That's good. This means the network is improving on the training set. However, if you look at epochs 8 and 9, you will see that val_error started to oscillate and increase. Improving train_error while not improving val_error means the network is starting to overfit the training data and failing to generalize to the validation data.

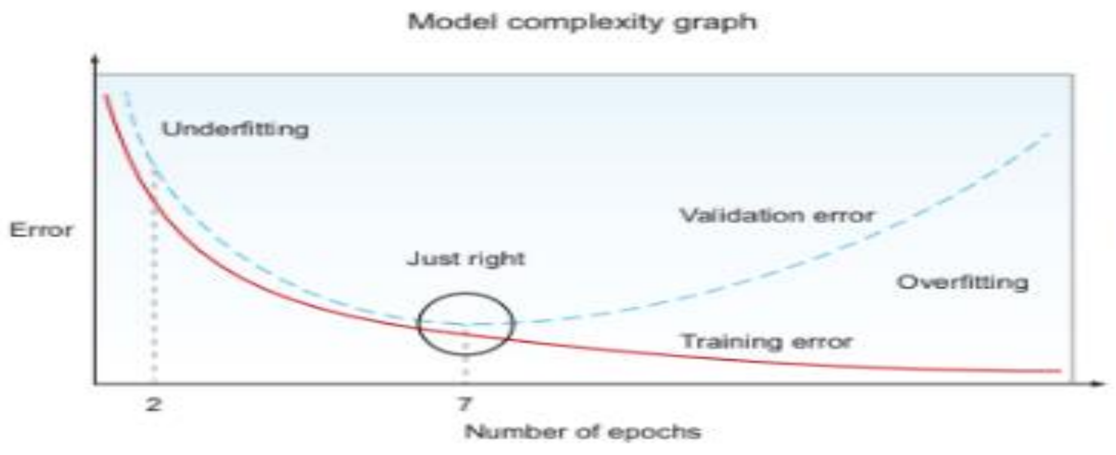


Figure 4.26 **Improving `train_error` while not improving `val_error` means the network is starting to overfit.**

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. Here is 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.

Regularization techniques to avoid overfitting

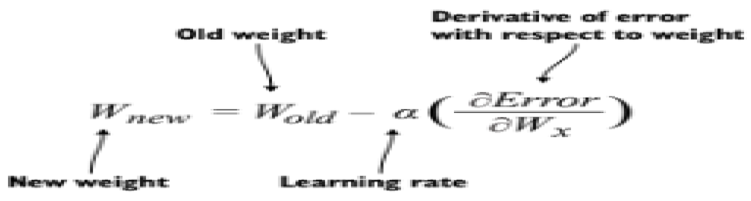
If you observe that your neural network is overfitting the training data, your network might be too complex and need to be simplified. One of the first techniques you should try is regularization. In this section, we will discuss three of the most common regularization techniques: L2, dropout, and data augmentation.

→ 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.

Let's see how regularization works. First, we update the error function by adding the regularization term:

error functionnew = error functionold + regularization term

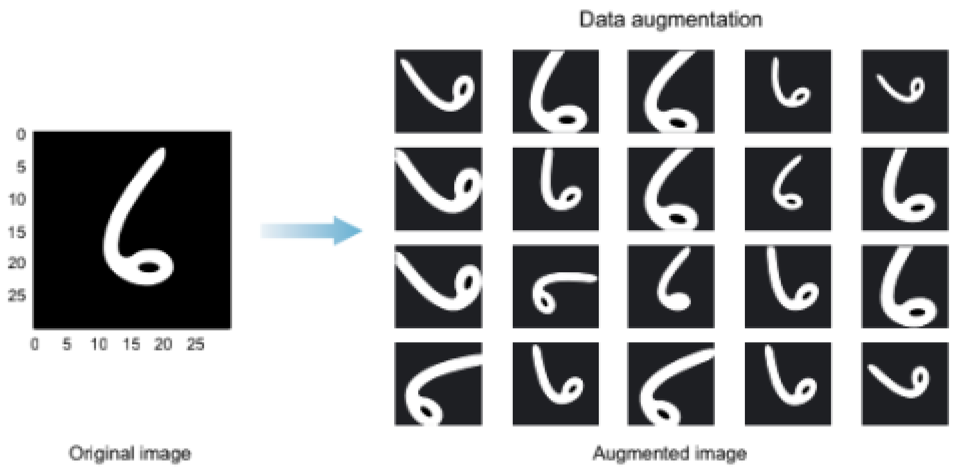

Old weight
Derivative of error with respect to weight
$W_{new} = W_{old} - \alpha\left(\frac{\partial Error}{\partial W_x}\right)$
New weight
Learning rate

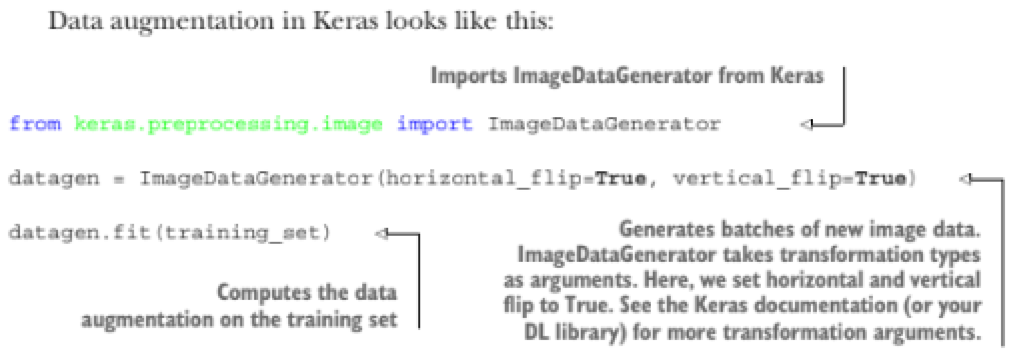
Dropout layers

Dropout is another regularization technique that is very effective for simplifying a neural network and avoiding overfitting.The dropout algorithm is fairly simple: at every training iteration, every neuron has a probability p of being temporarily ignored (dropped out) during this training iteration. This means it may be active during subsequent iterations. While it is counterintuitive to intentionally pause the learning on some of the network neurons, it is quite surprising how well this technique works. The probability p is a hyperparameter that is called dropout rate and is typically set in the range of 0.3 to 0.5. Start with 0.3, and if you see signs of overfitting, increase the rate.

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 yourdataset to provide a variety of images to train on.




Data augmentation in Keras looks like this:

Imports ImageDataGenerator from Keras

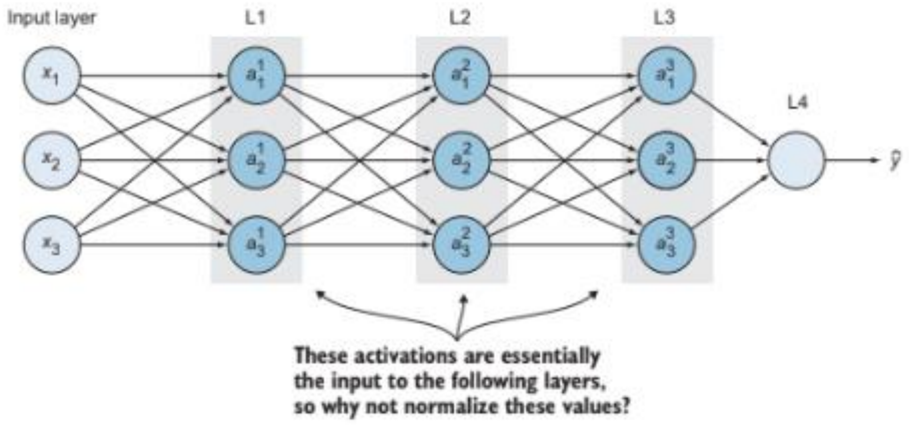
```
from keras.preprocessing.image import ImageDataGenerator

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

datagen.fit(training_set)
```

Computes the data augmentation on the 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.

Batch normalization

The normalization techniques we discussed were focused on preprocessing the training set before feeding it to the input layer. If the input layer benefits from normalization, why not do the same thing for the extracted features in the hidden units, which are changing all the time and get much more improvement in training speed and network resilience.This process is called batch normalization (BN).



The covariate shift problem

Before we define covariate shift, let's take a look at an example to illustrate the problem that batch normalization (BN) confronts. Suppose you are building a cat classifier, and you train your algorithm on images of white cats only. When you test this classifier on images with cats that are different colors, it will not perform well. Why?Because the model has been trained on a training set with a specific distribution(white cats). When the distribution changes in the test set, it confuses the model. If a model is learning to map dataset X to label y, then if the distribution of X changes, it's known as covariate shift. When that happens, you might need to retrain your learning algorithm.

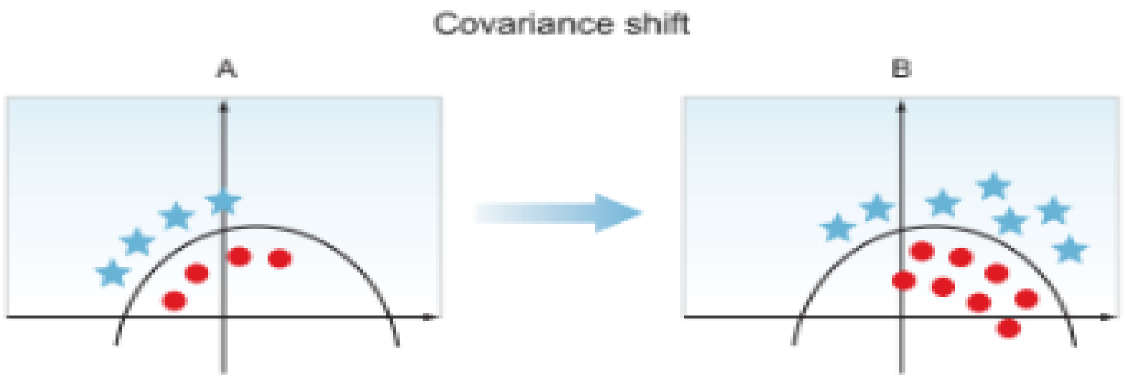


Figure 4.29 Graph A is the training set of only white cats, and graph B is the testing set with cats of various colors. The circles represent the cat images, and the stars represent the non-cat images.

Covariate shift in neural networks

To understand how covariate shift happens in neural networks, consider the simplefour-layer MLP in figure 4.30. Let's look at the network from the third-layer (L3) perspective. Its input are the activation values in L2 (a12, a22, a32, and a42), which are the features extracted from the previous layers. L3 is trying to map these inputs to yˆ to make it as close as possible to the label y. While the third layer is doing that, the network is adapting the values of the parameters from previous layers. As the parameters (w, b) are changing in layer 1, the activation values in the second layer are changing,too. So from the perspective of the third hidden layer, the values of the second hidden layer are changing all the time: the MLP is suffering from the problem of covariate shift. Batch norm reduces the degree of change in the distribution of the hidden unit values, causing these values to become more stable so that the later layers of the neural network have firmer ground to stand on.

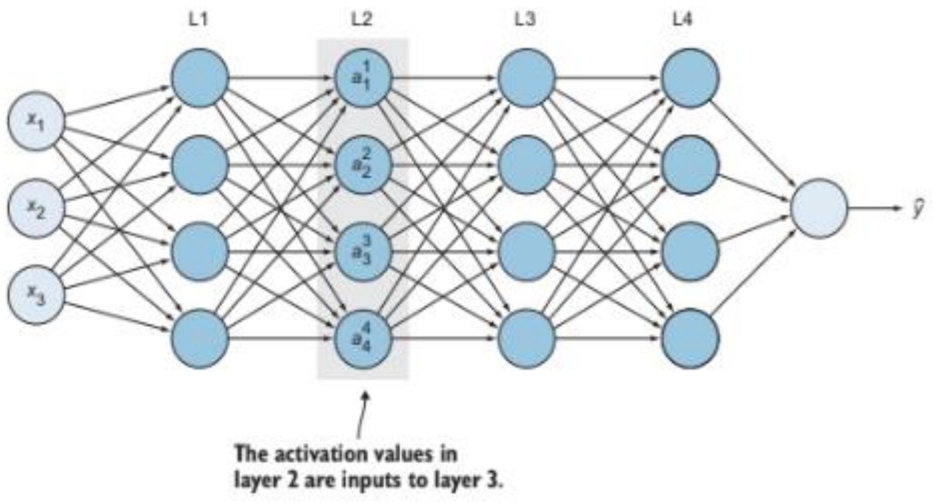
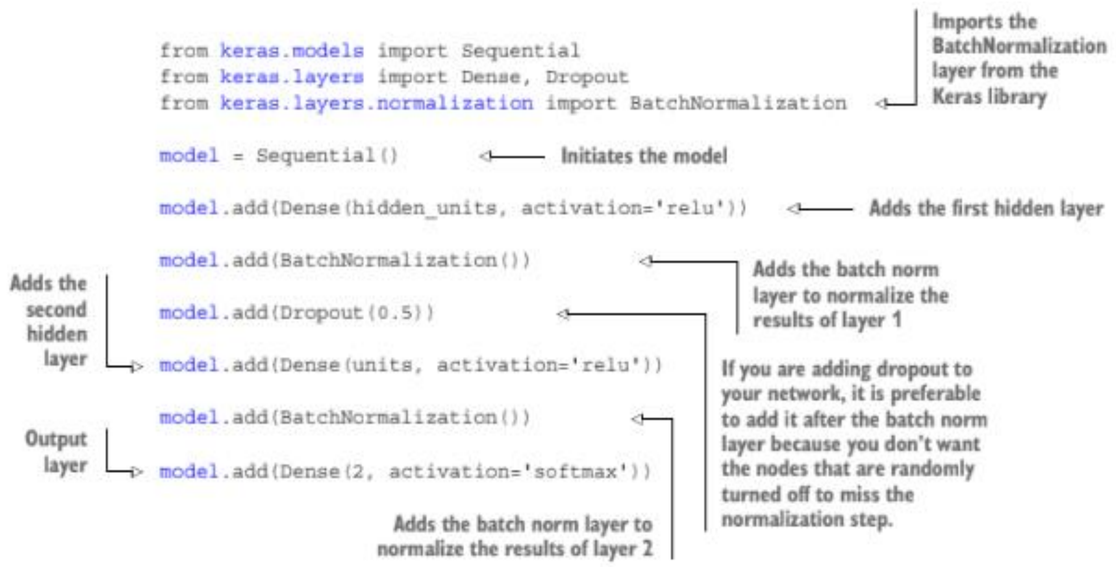


Figure 4.30 A simple four-layer MLP. L1 features are input to the L2 layer. The same is true for layers 2, 3, and 4.

Batch normalization implementation in Keras



```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers.normalization import BatchNormalization

model = Sequential()

model.add(Dense(hidden_units, activation='relu'))

model.add(BatchNormalization())

model.add(Dropout(0.5))

model.add(Dense(units, activation='relu'))

model.add(BatchNormalization())

model.add(Dense(2, activation='softmax'))
```

Imports the BatchNormalization layer from the Keras library

Initiates the model

Adds the first hidden layer

Adds the batch norm layer to normalize the results of layer 1

If you are adding dropout to your network, it is preferable to add it after the batch norm layer because you don't want the nodes that are randomly turned off to miss the normalization step.

Adds the second hidden layer

Adds the batch norm layer to normalize the results of layer 2

Output layer