1. What are the advantages of a CNN over a fully connected DNN for image classification?

Because consecutive layers are only partially connected and because it heavily reuses its weights, a CNN has many fewer parameters than a fully connected DNN, which makes it much faster to train, reduces the risk of overfitting, and requires much less training data.

The objective of this article is to provide a theoretical perspective to understand why (single layer) CNNs work better than fully-connected networks for image processing. Linear algebra (matrix multiplication, eigenvalues and/or PCA) and a property of sigmoid/tanh function will be used in an attempt to have a one-to-one (almost) comparison between a fully-connected network (logistic regression) and CNN. Finally, the tradeoff between filter size and the amount of information retained in the filtered image will be examined for the purpose of prediction. For simplicity, we will assume the following:

The fully-connected network does not have a hidden layer (logistic regression)

Original image was normalized to have pixel values between 0 and 1 or scaled to have mean = 0 and variance = 1

Sigmoid/tanh activation is used between input and convolved image, although the argument works for other non-linear activation functions such as ReLU. ReLU is avoided because it breaks the rigor of the analysis if the images are scaled (mean = 0, variance = 1) instead of normalized

Number of channels = depth of image = 1 for most of the article, model with higher number of channels will be discussed briefly

The problem involves a classification task. Therefore, C > 1

There are no non-linearities other than the activation and no non-differentiability (like pooling, strides other than 1, padding, etc.)

Negative log likelihood loss function is used to train both networks.

2.If your GPU runs out of memory while training a CNN, what are five things you could try to solve the problem?

If your GPU runs out of memory while training a CNN, what are five things you could try to solve the problem? Reduce the mini-batch size. Reduce dimensionality using a larger stride in one or more layers. Remove one or more layers.

Convolutional Neural Networks (CNN) are everywhere. It is arguably the most popular deep learning architecture. The recent surge of interest in deep learning is due to the immense popularity and effectiveness of convnets. The interest in CNN started with AlexNet in 2012 and it has grown exponentially ever since. In just three years, researchers progressed from 8 layer AlexNet to 152 layer ResNet.

CNN is now the go-to model on every image related problem. In terms of accuracy they blow competition out of the water. It is also successfully applied to recommender systems, natural language processing and more. The main advantage of CNN compared to its predecessors is that it automatically detects the important features without any human supervision. For example, given many pictures of cats and dogs it learns distinctive features for each class by itself.

CNN is also computationally efficient. It uses special convolution and pooling operations and performs parameter sharing. This enables CNN models to run on any device, making them universally attractive.

All in all this sounds like pure magic. We are dealing with a very powerful and efficient model which performs automatic feature extraction to achieve superhuman accuracy (yes CNN models now do image classification better than humans). Hopefully this article will help us uncover the secrets of this remarkable technique.

2. Architecture

All CNN models follow a similar architecture, as shown in the figure below.

There is an input image that we’re working with. We perform a series convolution + pooling operations, followed by a number of fully connected layers. If we are performing multiclass classification the output is softmax. We will now dive into each component.

2.1) Convolution

The main building block of CNN is the convolutional layer. Convolution is a mathematical operation to merge two sets of information. In our case the convolution is applied on the input data using a convolution filter to produce a feature map. There are a lot of terms being used so let’s visualize them one by one.

On the left side is the input to the convolution layer, for example the input image. On the right is the convolution filter, also called the kernel, we will use these terms interchangeably. This is called a 3x3 convolution due to the shape of the filter.

We perform the convolution operation by sliding this filter over the input. At every location, we do element-wise matrix multiplication and sum the result. This sum goes into the feature map. The green area where the convolution operation takes place is called the receptive field. Due to the size of the filter the receptive field is also 3x3.

Here the filter is at the top left, the output of the convolution operation “4” is shown in the resulting feature map. We then slide the filter to the right and perform the same operation, adding that result to the feature map as well.

We continue like this and aggregate the convolution results in the feature map. Here’s an animation that shows the entire convolution operation.

This was an example convolution operation shown in 2D using a 3x3 filter. But in reality these convolutions are performed in 3D. In reality an image is represented as a 3D matrix with dimensions of height, width and depth, where depth corresponds to color channels (RGB). A convolution filter has a specific height and width, like 3x3 or 5x5, and by design it covers the entire depth of its input so it needs to be 3D as well.

One more important point before we visualize the actual convolution operation. We perform multiple convolutions on an input, each using a different filter and resulting in a distinct feature map. We then stack all these feature maps together and that becomes the final output of the convolution layer. But first let’s start simple and visualize a convolution using a single filter.

Let’s say we have a 32x32x3 image and we use a filter of size 5x5x3 (note that the depth of the convolution filter matches the depth of the image, both being 3). When the filter is at a particular location it covers a small volume of the input, and we perform the convolution operation described above. The only difference is this time we do the sum of matrix multiply in 3D instead of 2D, but the result is still a scalar. We slide the filter over the input like above and perform the convolution at every location aggregating the result in a feature map. This feature map is of size 32x32x1, shown as the red slice on the right.

If we used 10 different filters we would have 10 feature maps of size 32x32x1 and stacking them along the depth dimension would give us the final output of the convolution layer: a volume of size 32x32x10, shown as the large blue box on the right. Note that the height and width of the feature map are unchanged and still 32, it’s due to padding and we will elaborate on that shortly.

To help with visualization, we slide the filter over the input as follows. At each location we get a scalar and we collect them in the feature map. The animation shows the sliding operation at 4 locations, but in reality it’s performed over the entire input.

Below we can see how two feature maps are stacked along the depth dimension. The convolution operation for each filter is performed independently and the resulting feature maps are disjoint.

2.2) Non-linearity

For any kind of neural network to be powerful, it needs to contain non-linearity. Both the ANN and autoencoder we saw before achieved this by passing the weighted sum of its inputs through an activation function, and CNN is no different. We again pass the result of the convolution operation through relu activation function. So the values in the final feature maps are not actually the sums, but the relu function applied to them. We have omitted this in the figures above for simplicity. But keep in mind that any type of convolution involves a relu operation, without that the network won’t achieve its true potential.

2.3) Stride and Padding

Stride specifies how much we move the convolution filter at each step. By default the value is 1, as you can see in the figure below.

We can have bigger strides if we want less overlap between the receptive fields. This also makes the resulting feature map smaller since we are skipping over potential locations. The following figure demonstrates a stride of 2. Note that the feature map got smaller.

We see that the size of the feature map is smaller than the input, because the convolution filter needs to be contained in the input. If we want to maintain the same dimensionality, we can use padding to surround the input with zeros. Check the animation below.

The gray area around the input is the padding. We either pad with zeros or the values on the edge. Now the dimensionality of the feature map matches the input. Padding is commonly used in CNN to preserve the size of the feature maps, otherwise they would shrink at each layer, which is not desirable. The 3D convolution figures we saw above used padding, that’s why the height and width of the feature map was the same as the input (both 32x32), and only the depth changed.

2.4) Pooling

After a convolution operation we usually perform pooling to reduce the dimensionality. This enables us to reduce the number of parameters, which both shortens the training time and combats overfitting. Pooling layers downsample each feature map independently, reducing the height and width, keeping the depth intact.

The most common type of pooling is max pooling which just takes the max value in the pooling window. Contrary to the convolution operation, pooling has no parameters. It slides a window over its input, and simply takes the max value in the window. Similar to a convolution, we specify the window size and stride.

Here is the result of max pooling using a 2x2 window and stride 2. Each color denotes a different window. Since both the window size and stride are 2, the windows are not overlapping.

Note that this window and stride configuration halves the size of the feature map. This is the main use case of pooling, downsampling the feature map while keeping the important information.

Now let’s work out the feature map dimensions before and after pooling. If the input to the pooling layer has the dimensionality 32x32x10, using the same pooling parameters described above, the result will be a 16x16x10 feature map. Both the height and width of the feature map are halved, but the depth doesn’t change because pooling works independently on each depth slice the input.

By halving the height and the width, we reduced the number of weights to 1/4 of the input. Considering that we typically deal with millions of weights in CNN architectures, this reduction is a pretty big deal.

In CNN architectures, pooling is typically performed with 2x2 windows, stride 2 and no padding. While convolution is done with 3x3 windows, stride 1 and with padding.

2.5) Hyperparameters

Let’s now only consider a convolution layer ignoring pooling, and go over the hyperparameter choices we need to make. We have 4 important hyperparameters to decide on:

Filter size: we typically use 3x3 filters, but 5x5 or 7x7 are also used depending on the application. There are also 1x1 filters which we will explore in another article, at first sight it might look strange but they have interesting applications. Remember that these filters are 3D and have a depth dimension as well, but since the depth of a filter at a given layer is equal to the depth of its input, we omit that.

Filter count: this is the most variable parameter, it’s a power of two anywhere between 32 and 1024. Using more filters results in a more powerful model, but we risk overfitting due to increased parameter count. Usually we start with a small number of filters at the initial layers, and progressively increase the count as we go deeper into the network.

Stride: we keep it at the default value 1.

Padding: we usually use padding.

2.6) Fully Connected

After the convolution + pooling layers we add a couple of fully connected layers to wrap up the CNN architecture. This is the same fully connected ANN architecture we talked about in [Part 1](https://medium.com/towards-data-science/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6).

Remember that the output of both convolution and pooling layers are 3D volumes, but a fully connected layer expects a 1D vector of numbers. So we flatten the output of the final pooling layer to a vector and that becomes the input to the fully connected layer. Flattening is simply arranging the 3D volume of numbers into a 1D vector, nothing fancy happens here.

2.7) Training

CNN is trained the same way like ANN, backpropagation with gradient descent. Due to the convolution operation it’s more mathematically involved, and it’s out of the scope for this article. If you’re interested in the details refer [here](http://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/).

3. Intuition

A CNN model can be thought as a combination of two components: feature extraction part and the classification part. The convolution + pooling layers perform feature extraction. For example given an image, the convolution layer detects features such as two eyes, long ears, four legs, a short tail and so on. The fully connected layers then act as a classifier on top of these features, and assign a probability for the input image being a dog.

The convolution layers are the main powerhouse of a CNN model. Automatically detecting meaningful features given only an image and a label is not an easy task. The convolution layers learn such complex features by building on top of each other. The first layers detect edges, the next layers combine them to detect shapes, to following layers merge this information to infer that this is a nose. To be clear, the CNN doesn’t know what a nose is. By seeing a lot of them in images, it learns to detect that as a feature. The fully connected layers learn how to use these features produced by convolutions in order to correctly classify the images.

All this might sound vague right now, but hopefully the visualization section will make everything more clear.

4. Implementation

After this lengthy explanation let’s code up our CNN. We will use the Dogs vs Cats [dataset](https://www.kaggle.com/c/dogs-vs-cats) from Kaggle to distinguish dog photos from cats.

We will use the following architecture: 4 convolution + pooling layers, followed by 2 fully connected layers. The input is an image of a cat or dog and the output is binary.

3.Why would you want to add a max pooling layer rather than a convolutional layer with the same stride?

Max-pooling helps in extracting low-level features like edges, points, etc. While Avg-pooling goes for smooth features. If time constraint is not a problem, then one can skip the pooling layer and use a convolutional layer to do the same.

A problem with the output feature maps is that they are sensitive to the location of the features in the input. One approach to address this sensitivity is to down sample the feature maps. This has the effect of making the resulting down sampled feature maps more robust to changes in the position of the feature in the image, referred to by the technical phrase “local translation invariance.”

Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. Two common pooling methods are average pooling and max pooling that summarize the average presence of a feature and the most activated presence of a feature respectively.

In this tutorial, you will discover how the pooling operation works and how to implement it in convolutional neural networks.

After completing this tutorial, you will know:

Pooling is required to down sample the detection of features in feature maps.

How to calculate and implement average and maximum pooling in a convolutional neural network.

How to use global pooling in a convolutional neural network.

Pooling Layers

Convolutional layers in a convolutional neural network systematically apply learned filters to input images in order to create feature maps that summarize the presence of those features in the input.

Convolutional layers prove very effective, and stacking convolutional layers in deep models allows layers close to the input to learn low-level features (e.g. lines) and layers deeper in the model to learn high-order or more abstract features, like shapes or specific objects.

A limitation of the feature map output of convolutional layers is that they record the precise position of features in the input. This means that small movements in the position of the feature in the input image will result in a different feature map. This can happen with re-cropping, rotation, shifting, and other minor changes to the input image.

A common approach to addressing this problem from signal processing is called down sampling. This is where a lower resolution version of an input signal is created that still contains the large or important structural elements, without the fine detail that may not be as useful to the task.

Down sampling can be achieved with convolutional layers by changing the [stride of the convolution across the image](https://machinelearningmastery.com/padding-and-stride-for-convolutional-neural-networks/). A more robust and common approach is to use a pooling layer.

A pooling layer is a new layer added after the convolutional layer. Specifically, after a nonlinearity (e.g. ReLU) has been applied to the feature maps output by a convolutional layer; for example the layers in a model may look as follows:

Input Image

Convolutional Layer

Nonlinearity

Pooling Layer

The addition of a pooling layer after the convolutional layer is a common pattern used for ordering layers within a convolutional neural network that may be repeated one or more times in a given model.

The pooling layer operates upon each feature map separately to create a new set of the same number of pooled feature maps.

Pooling involves selecting a pooling operation, much like a filter to be applied to feature maps. The size of the pooling operation or filter is smaller than the size of the feature map; specifically, it is almost always 2×2 pixels applied with a stride of 2 pixels.

This means that the pooling layer will always reduce the size of each feature map by a factor of 2, e.g. each dimension is halved, reducing the number of pixels or values in each feature map to one quarter the size. For example, a pooling layer applied to a feature map of 6×6 (36 pixels) will result in an output pooled feature map of 3×3 (9 pixels).

The pooling operation is specified, rather than learned. Two common functions used in the pooling operation are:

Average Pooling: Calculate the average value for each patch on the feature map.

Maximum Pooling (or Max Pooling): Calculate the maximum value for each patch of the feature map.

The result of using a pooling layer and creating down sampled or pooled feature maps is a summarized version of the features detected in the input. They are useful as small changes in the location of the feature in the input detected by the convolutional layer will result in a pooled feature map with the feature in the same location. This capability added by pooling is called the model’s invariance to local translation.

4.When would you want to add a local response normalization layer?

Local Response Normalization

Local Response Normalization (LRN) was first introduced in AlexNet architecture where the activation function used was ReLU as opposed to the more common tanh and sigmoid at that time. Apart from the reason mentioned above, the reason for using LRN was to encourage lateral inhibition. It is a concept in Neurobiology that refers to the capacity of a neuron to reduce the activity of its neighbors [1]. In DNNs, the purpose of this lateral inhibition is to carry out local contrast enhancement so that locally maximum pixel values are used as excitation for the next layers.

LRN is a non-trainable layer that square-normalizes the pixel values in a feature map within a local neighborhood.

Inter-Channel LRN: This is originally what the AlexNet paper used. The neighborhood defined is across the channel. For each (x,y) position, the normalization is carried out in the depth dimension and is given by the following formula

LRN used in AlexNet [2]

where i indicates the output of filter i, a(x,y), b(x,y) the pixel values at (x,y) position before and after normalization respectively, and N is the total number of channels. The constants (k,α,β,n) are hyper-parameters. k is used to avoid any singularities (division by zero), α is used as a normalization constant, while β is a contrasting constant. The constant n is used to define the neighborhood length i.e. how many consecutive pixel values need to be considered while carrying out the normalization. The case of (k,α, β, n)=(0,1,1,N) is the standard normalization). In the figure above n is taken to be to 2 while N=4.

Different colors denote different channels and hence N=4. Lets take the hyper-parameters to be (k,α, β, n)=(0,1,1,2). The value of n=2 means that while calculating the normalized value at position (i,x,y), we consider the values at the same position for the previous and next filter i.e (i-1, x, y) and (i+1, x, y). For (i,x,y)=(0,0,0) we have value(i,x,y)=1, value(i-1,x,y) doesn’t exist and value(i+,x,y)=1. Hence normalized\_value(i,x,y) = 1/(¹²+¹²) = 0.5 and can be seen in the lower part of the figure above. The rest of the normalized values are calculated in a similar way.

Intra-Channel LRN: In Intra-channel LRN, the neighborhood is extended within the same channel only as can be seen in the figure above.

where (W,H) are the width and height of the feature map (for example in the figure above (W,H) = (8,8)). The only difference between Inter and Intra Channel LRN is the neighborhood for normalization. In Intra-channel LRN, a 2D neighborhood is defined (as opposed to the 1D neighborhood in Inter-Channel) around the pixel under-consideration. As an example, the figure below shows the Intra-Channel normalization on a 5x5 feature map with n=2 (i.e. 2D neighborhood of size (n+1)x(n+1) centered at (x,y)).

5.Can you name the main innovations in AlexNet, compared to LeNet-5? What about the main innovations in GoogLeNet, ResNet, SENet, and Xception?

A Convolutional Neural Network (CNN, or ConvNet) are a special kind of multi-layer neural networks, designed to recognize visual patterns directly from pixel images with minimal preprocessing.. The ImageNet project is a large visual database designed for use in visual object recognition software research. The ImageNet project runs an annual software contest, the ImageNet Large Scale Visual Recognition Challenge ([ILSVRC](https://en.wikipedia.org/wiki/ImageNet#ImageNet_Challenge)), where software programs compete to correctly classify and detect objects and scenes. Here I will talk about CNN architectures of ILSVRC top competitors .

LeNet-5 (1998)

LeNet-5, a pioneering 7-level convolutional network by LeCun et al in 1998, that classifies digits, was applied by several banks to recognise hand-written numbers on checks (cheques) digitized in 32x32 pixel greyscale inputimages. The ability to process higher resolution images requires larger and more convolutional layers, so this technique is constrained by the availability of computing resources.

AlexNet (2012)

In 2012, [AlexNet](https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf" \t "_blank) significantly outperformed all the prior competitors and won the challenge by reducing the top-5 error from 26% to 15.3%. The second place top-5 error rate, which was not a CNN variation, was around 26.2%.

The network had a very similar architecture as [LeNet](http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf" \t "_blank) by Yann LeCun et al but was deeper, with more filters per layer, and with stacked convolutional layers. It consisted 11x11, 5x5,3x3, convolutions, max pooling, dropout, data augmentation, ReLU activations, SGD with momentum. It attached ReLU activations after every convolutional and fully-connected layer. AlexNet was trained for 6 days simultaneously on two Nvidia Geforce GTX 580 GPUs which is the reason for why their network is split into two pipelines. AlexNet was designed by the SuperVision group, consisting of Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever.

ZFNet(2013)

Not surprisingly, the ILSVRC 2013 winner was also a CNN which became known as ZFNet. It achieved a top-5 error rate of 14.8% which is now already half of the prior mentioned non-neural error rate. It was mostly an achievement by tweaking the hyper-parameters of AlexNet while maintaining the same structure with additional Deep Learning elements as discussed earlier in this essay.

GoogLeNet/Inception(2014)

The winner of the ILSVRC 2014 competition was GoogLeNet(a.k.a. Inception V1) from Google. It achieved a top-5 error rate of 6.67%! This was very close to human level performance which the organisers of the challenge were now forced to evaluate. As it turns out, this was actually rather hard to do and required some human training in order to beat GoogLeNets accuracy. After a few days of training, the human expert (Andrej Karpathy) was able to achieve a top-5 error rate of 5.1%(single model) and 3.6%(ensemble). The network used a CNN inspired by LeNet but implemented a novel element which is dubbed an inception module. It used batch normalization, image distortions and RMSprop. This module is based on several very small convolutions in order to drastically reduce the number of parameters. Their architecture consisted of a 22 layer deep CNN but reduced the number of parameters from 60 million (AlexNet) to 4 million.

VGGNet (2014)

The runner-up at the ILSVRC 2014 competition is dubbed VGGNet by the community and was developed by Simonyan and Zisserman. VGGNet consists of 16 convolutional layers and is very appealing because of its very uniform architecture. Similar to AlexNet, only 3x3 convolutions, but lots of filters. Trained on 4 GPUs for 2–3 weeks. It is currently the most preferred choice in the community for extracting features from images. The weight configuration of the VGGNet is publicly available and has been used in many other applications and challenges as a baseline feature extractor. However, VGGNet consists of 138 million parameters, which can be a bit challenging to handle.

ResNet(2015)

At last, at the ILSVRC 2015, the so-called Residual Neural Network (ResNet) by Kaiming He et al introduced anovel architecture with “skip connections” and features heavy batch normalization. Such skip connections are also known as gated units or gated recurrent units and have a strong similarity to recent successful elements applied in RNNs. Thanks to this technique they were able to train a NN with 152 layers while still having lower complexity than VGGNet. It achieves a top-5 error rate of 3.57% which beats human-level performance on this dataset.

AlexNet has parallel two CNN line trained on two GPUs with cross-connections, GoogleNet has inception modules ,ResNet has residual connections.

6.What is a fully convolutional network? How can you convert a dense layer into a convolutional layer?

Convolutional neural networks (CNN) work great for computer vision tasks. Using a pre-trained model that is trained on huge datasets like ImageNet, COCO, etc. we can quickly specialize these architectures to work for our unique dataset. This process is termed as transfer learning. However, there’s a catch! Pre-trained models for image classification and object detection tasks are usually trained on fixed input image sizes. These typically range from 224x224x3 to somewhere around 512x512x3 and mostly have an aspect ratio of 1 i.e. the width and height of the image are equal. If they are not equal then the images are resized to be of equal height and width.

Newer architectures do have the ability to handle variable input image sizes but it’s more common in object detection and segmentation tasks as compared to image classification tasks. Recently, I came across an interesting use case wherein I had 5 different classes of image and each of the classes had minuscule differences. Also, the aspect ratio of the images was higher than usual. The average height of the image was around 30 pixels and the width was around 300 pixels. This was an interesting one for the following reasons:

Resizing the images easily distorted the important features

Pre-trained architectures were gargantuan and always overfitted the dataset

The task demanded low latency

The need for a CNN with variable input dimensions

I tried base models of MobileNet and EfficientNet but nothing worked. There was a need for a network which didn’t have any restrictions on input image size and could perform image classification task at hand. The first thing that struck me was fully convolutional networks (FCNs). FCN is a network that does not contain any “Dense” layers (as in traditional CNNs) instead it contains 1x1 convolutions that perform the task of fully connected layers (Dense layers). Though the absence of dense layers makes it possible to feed in variable inputs, there are a couple of techniques that enable us to use dense layers while cherishing variable input dimensions. This tutorial delineates some of those techniques. In this tutorial, we will go through the following steps:

Building a fully convolutional network (FCN) in TensorFlow using Keras

Downloading and splitting a sample dataset

Creating a generator in Keras to load and process a batch of data in memory

Training the network with variable batch dimensions

Deploying the model using TensorFlow Serving

7.What is the main technical difficulty of semantic segmentation?

In semantic segmentation, our aim is to extract features before using them to separate the image into multiple segments. However, the issue with convolutional networks is that the size of the image is reduced as it passes through the network because of the max-pooling layers.

Deep learning has been very successful when working with images as data and is currently at a stage where it works better than humans on multiple use-cases. The most important problems that humans have been  interested in solving with computer vision are image classification, object detection and segmentation in the increasing order of their difficulty.

In the plain old task of image classification we are just interested in getting the labels of all the objects that are present in an image. In object detection we come further a step and try to know along with what all objects that are present in an image, the location at which the objects are present with the help of bounding boxes. Image segmentation takes it to a new level by trying to find out accurately the exact boundary of the objects in the image.

What is image segmentation

We know an image is nothing but a collection of pixels. Image segmentation is the process of classifying each pixel in an image belonging to a certain class and hence can be thought of as a classification problem per pixel. There are two types of segmentation techniques

Semantic segmentation :- Semantic segmentation is the process of classifying each pixel belonging to a particular label. It doesn't different across different instances of the same object. For example if there are 2 cats in an image, semantic segmentation gives same label to all the pixels of both cats

Instance segmentation :- Instance segmentation differs from semantic segmentation in the sense that it gives a unique label to every instance of a particular object in the image. As can be seen in the image above all 3 dogs are assigned different colours i.e different labels. With semantic segmentation all of them would have been assigned the same colour.

So we will now come to the point where would we need this kind of an algorithm

Use-cases of image segmentation

[Handwriting Recognition](https://nanonets.com/blog/handwritten-form-ocr-handwriting-recognition/) :- Junjo et all demonstrated how semantic segmentation is being used to extract words and lines from handwritten documents in their [2019 research paper](https://arxiv.org/pdf/1906.05229.pdf)to recognise handwritten characters

Google portrait mode :- There are many use-cases where it is absolutely essential to separate foreground from background. For example in Google's portrait mode we can see the background blurred out while the foreground remains unchanged to give a cool effect

YouTube stories :- Google recently released a feature YouTube stories for content creators to show different backgrounds while creating stories.

Virtual make-up :- Applying virtual lip-stick is possible now with the help of image segmentation

4.Virtual try-on :- Virtual try on of clothes is an interesting feature which was available in stores using specialized hardware which creates a 3d model. But with deep learning and image segmentation the same can be obtained using just a 2d image

Visual Image Search :- The idea of segmenting out clothes is also used in image retrieval algorithms in eCommerce. For example Pinterest/Amazon allows you to upload any picture and get related similar looking products by doing an image search based on segmenting out the cloth portion

Self-driving cars :- Self driving cars need a complete understanding of their surroundings to a pixel perfect level. Hence image segmentation is used to identify lanes and other necessary information

8.Build your own CNN from scratch and try to achieve the highest possible accuracy on MNIST.

The MNIST handwritten digit classification problem is a standard dataset used in computer vision and deep learning.

Although the dataset is effectively solved, it can be used as the basis for learning and practicing how to develop, evaluate, and use convolutional deep learning neural networks for image classification from scratch. This includes how to develop a robust test harness for estimating the performance of the model, how to explore improvements to the model, and how to save the model and later load it to make predictions on new data.

In this tutorial, you will discover how to develop a convolutional neural network for handwritten digit classification from scratch.

After completing this tutorial, you will know:

How to develop a test harness to develop a robust evaluation of a model and establish a baseline of performance for a classification task.

How to explore extensions to a baseline model to improve learning and model capacity.

How to develop a finalized model, evaluate the performance of the final model, and use it to make predictions on new images.

The [MNIST dataset](https://en.wikipedia.org/wiki/MNIST_database) is an acronym that stands for the Modified National Institute of Standards and Technology dataset.

It is a dataset of 60,000 small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9.

The task is to classify a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9, inclusively.

It is a widely used and deeply understood dataset and, for the most part, is “solved.” Top-performing models are deep learning convolutional neural networks that achieve a classification accuracy of above 99%, with an error rate between 0.4 %and 0.2% on the hold out test dataset.

9.Use transfer learning for large image classification, going through these steps:

* 1. Create a training set containing at least 100 images per class. For example, you could classify your own pictures based on the location (beach, mountain, city, etc.), or alternatively you can use an existing dataset (e.g., from TensorFlow Datasets).
  2. Split it into a training set, a validation set, and a test set.
  3. Build the input pipeline, including the appropriate preprocessing operations, and optionally add data augmentation.
  4. Fine-tune a pretrained model on this dataset.

a [tf.keras.Sequential](https://www.tensorflow.org/api_docs/python/tf/keras/Sequential) model and load data using [tf.keras.utils.image\_dataset\_from\_directory](https://www.tensorflow.org/api_docs/python/tf/keras/utils/image_dataset_from_directory). It demonstrates the following concepts:

Efficiently loading a dataset off disk.

Identifying overfitting and applying techniques to mitigate it, including data augmentation and dropout.

This tutorial follows a basic machine learning workflow:

Examine and understand data

Build an input pipeline

Build the model

Train the model

Test the model

Improve the model and repeat the process

In addition, the notebook demonstrates how to convert a [saved model](https://www.tensorflow.org/guide/saved_model) to a [TensorFlow Lite](https://www.tensorflow.org/lite/) model for on-device machine learning on mobile, embedded, and IoT devices.

Setup

Import TensorFlow and other necessary libraries:

import matplotlib.pyplot as plt  
import numpy as np  
import PIL  
import tensorflow as tf  
  
from tensorflow import keras  
from tensorflow.keras import layers  
from tensorflow.keras.models import Sequential

2022-10-27 01:21:02.763442: W tensorflow/compiler/xla/stream\_executor/platform/default/dso\_loader.cc:64] Could not load dynamic library 'libnvinfer.so.7'; dlerror: libnvinfer.so.7: cannot open shared object file: No such file or directory

2022-10-27 01:21:02.763541: W tensorflow/compiler/xla/stream\_executor/platform/default/dso\_loader.cc:64] Could not load dynamic library 'libnvinfer\_plugin.so.7'; dlerror: libnvinfer\_plugin.so.7: cannot open shared object file: No such file or directory

2022-10-27 01:21:02.763550: W tensorflow/compiler/tf2tensorrt/utils/py\_utils.cc:38] TF-TRT Warning: Cannot dlopen some TensorRT libraries. If you would like to use Nvidia GPU with TensorRT, please make sure the missing libraries mentioned above are installed properly.

This tutorial uses a dataset of about 3,700 photos of flowers. The dataset contains five sub-directories, one per class:

flower\_photo/  
  daisy/  
  dandelion/  
  roses/  
  sunflowers/  
  tulips/

import pathlib  
dataset\_url = "https://storage.googleapis.com/download.tensorflow.org/example\_images/flower\_photos.tgz"  
data\_dir = tf.keras.utils.get\_file('flower\_photos', origin=dataset\_url, untar=True)  
data\_dir = pathlib.Path(data\_dir)

Downloading data from https://storage.googleapis.com/download.tensorflow.org/example\_images/flower\_photos.tgz

228813984/228813984 [==============================] - 1s 0us/step

After downloading, you should now have a copy of the dataset available. There are 3,670 total images:

image\_count = len(list(data\_dir.glob('\*/\*.jpg')))  
print(image\_count)

3670

Load data using a Keras utility

Next, load these images off disk using the helpful [tf.keras.utils.image\_dataset\_from\_directory](https://www.tensorflow.org/api_docs/python/tf/keras/utils/image_dataset_from_directory) utility. This will take you from a directory of images on disk to a [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset) in just a couple lines of code. If you like, you can also write your own data loading code from scratch by visiting the [Load and preprocess images](https://www.tensorflow.org/tutorials/load_data/images) tutorial.

Create a dataset

Define some parameters for the loader:

batch\_size = 32  
img\_height = 180  
img\_width = 180

It's good practice to use a validation split when developing your model. Use 80% of the images for training and 20% for validation.

train\_ds = tf.keras.utils.image\_dataset\_from\_directory(  
  data\_dir,  
  validation\_split=0.2,  
  subset="training",  
  seed=123,  
  image\_size=(img\_height, img\_width),  
  batch\_size=batch\_size)

Found 3670 files belonging to 5 classes.

Using 2936 files for training.

val\_ds = tf.keras.utils.image\_dataset\_from\_directory(  
  data\_dir,  
  validation\_split=0.2,  
  subset="validation",  
  seed=123,  
  image\_size=(img\_height, img\_width),  
  batch\_size=batch\_size)

Found 3670 files belonging to 5 classes.

Using 734 files for validation.

You can find the class names in the class\_names attribute on these datasets. These correspond to the directory names in alphabetical order.

class\_names = train\_ds.class\_names  
print(class\_names)

['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']

Visualize the data

Here are the first nine images from the training dataset:

import matplotlib.pyplot as plt  
  
plt.figure(figsize=(10, 10))  
for images, labels in train\_ds.take(1):  
  for i in range(9):  
    ax = plt.subplot(3, 3, i + 1)  
    plt.imshow(images[i].numpy().astype("uint8"))  
    plt.title(class\_names[labels[i]])  
    plt.axis("off")

You will pass these datasets to the Keras [Model.fit](https://www.tensorflow.org/api_docs/python/tf/keras/Model" \l "fit) method for training later in this tutorial. If you like, you can also manually iterate over the dataset and retrieve batches of images:

for image\_batch, labels\_batch in train\_ds:  
  print(image\_batch.shape)  
  print(labels\_batch.shape)  
  break

(32, 180, 180, 3)

(32,)

The image\_batch is a tensor of the shape (32, 180, 180, 3). This is a batch of 32 images of shape 180x180x3 (the last dimension refers to color channels RGB). The label\_batch is a tensor of the shape (32,), these are corresponding labels to the 32 images.

You can call .numpy() on the image\_batch and labels\_batch tensors to convert them to a numpy.ndarray.

Configure the dataset for performance

Make sure to use buffered prefetching, so you can yield data from disk without having I/O become blocking. These are two important methods you should use when loading data:

[Dataset.cache](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#cache) keeps the images in memory after they're loaded off disk during the first epoch. This will ensure the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache.

[Dataset.prefetch](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#prefetch) overlaps data preprocessing and model execution while training.

Interested readers can learn more about both methods, as well as how to cache data to disk in the Prefetching section of the [Better performance with the tf.data API](https://www.tensorflow.org/guide/data_performance) guide.

AUTOTUNE = tf.data.AUTOTUNE  
  
train\_ds = train\_ds.cache().shuffle(1000).prefetch(buffer\_size=AUTOTUNE)  
val\_ds = val\_ds.cache().prefetch(buffer\_size=AUTOTUNE)

Standardize the data

The RGB channel values are in the [0, 255] range. This is not ideal for a neural network; in general you should seek to make your input values small.

Here, you will standardize values to be in the [0, 1] range by using [tf.keras.layers.Rescaling](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Rescaling):

normalization\_layer = layers.Rescaling(1./255)

There are two ways to use this layer. You can apply it to the dataset by calling [Dataset.map](https://www.tensorflow.org/api_docs/python/tf/data/Dataset" \l "map):

normalized\_ds = train\_ds.map(lambda x, y: (normalization\_layer(x), y))  
image\_batch, labels\_batch = next(iter(normalized\_ds))  
first\_image = image\_batch[0]  
# Notice the pixel values are now in `[0,1]`.  
print(np.min(first\_image), np.max(first\_image))

WARNING:tensorflow:From /tmpfs/src/tf\_docs\_env/lib/python3.9/site-packages/tensorflow/python/autograph/pyct/static\_analysis/liveness.py:83: Analyzer.lamba\_check (from tensorflow.python.autograph.pyct.static\_analysis.liveness) is deprecated and will be removed after 2023-09-23.

Instructions for updating:

Lambda fuctions will be no more assumed to be used in the statement where they are used, or at least in the same block. https://github.com/tensorflow/tensorflow/issues/56089

0.0 0.9970461

Or, you can include the layer inside your model definition, which can simplify deployment. Use the second approach here.

Note: You previously resized images using the image\_size argument of [tf.keras.utils.image\_dataset\_from\_directory](https://www.tensorflow.org/api_docs/python/tf/keras/utils/image_dataset_from_directory). If you want to include the resizing logic in your model as well, you can use the [tf.keras.layers.Resizing](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Resizing) layer.

A basic Keras model

Create the model

The Keras [Sequential](https://www.tensorflow.org/guide/keras/sequential_model) model consists of three convolution blocks ([tf.keras.layers.Conv2D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D)) with a max pooling layer ([tf.keras.layers.MaxPooling2D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D)) in each of them. There's a fully-connected layer ([tf.keras.layers.Dense](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense)) with 128 units on top of it that is activated by a ReLU activation function ('relu'). This model has not been tuned for high accuracy; the goal of this tutorial is to show a standard approach.

num\_classes = len(class\_names)  
  
model = Sequential([  
  layers.Rescaling(1./255, input\_shape=(img\_height, img\_width, 3)),  
  layers.Conv2D(16, 3, padding='same', activation='relu'),  
  layers.MaxPooling2D(),  
  layers.Conv2D(32, 3, padding='same', activation='relu'),  
  layers.MaxPooling2D(),  
  layers.Conv2D(64, 3, padding='same', activation='relu'),  
  layers.MaxPooling2D(),  
  layers.Flatten(),  
  layers.Dense(128, activation='relu'),  
  layers.Dense(num\_classes)  
])

Compile the model

For this tutorial, choose the [tf.keras.optimizers.Adam](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam) optimizer and [tf.keras.losses.SparseCategoricalCrossentropy](https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy) loss function. To view training and validation accuracy for each training epoch, pass the metrics argument to [Model.compile](https://www.tensorflow.org/api_docs/python/tf/keras/Model" \l "compile).

model.compile(optimizer='adam',  
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),  
              metrics=['accuracy'])

Model summary

View all the layers of the network using the Keras [Model.summary](https://www.tensorflow.org/api_docs/python/tf/keras/Model" \l "summary) method:

model.summary()

Model: "sequential

Layer (type) Output Shape Param #

rescaling\_1 (Rescaling) (None, 180, 180, 3) 0

conv2d (Conv2D) (None, 180, 180, 16) 448

max\_pooling2d (MaxPooling2D (None, 90, 90, 16) 0

)

conv2d\_1 (Conv2D) (None, 90, 90, 32) 4640

max\_pooling2d\_1 (MaxPooling (None, 45, 45, 32) 0

2D)

conv2d\_2 (Conv2D) (None, 45, 45, 64) 18496

max\_pooling2d\_2 (MaxPooling (None, 22, 22, 64) 0

2D)

flatten (Flatten) (None, 30976) 0

dense (Dense) (None, 128) 3965056

dense\_1 (Dense) (None, 5) 645

Total params: 3,989,285

Trainable params: 3,989,285

Non-trainable params: 0

Train the model

Train the model for 10 epochs with the Keras [Model.fit](https://www.tensorflow.org/api_docs/python/tf/keras/Model" \l "fit) method:

epochs=10  
history = model.fit(  
  train\_ds,  
  validation\_data=val\_ds,  
  epochs=epochs  
)

Epoch 1/10

92/92 [==============================] - 10s 19ms/step - loss: 1.3114 - accuracy: 0.4414 - val\_loss: 1.0733 - val\_accuracy: 0.5681

Epoch 2/10

92/92 [==============================] - 1s 16ms/step - loss: 0.9782 - accuracy: 0.6233 - val\_loss: 1.0065 - val\_accuracy: 0.5954

Epoch 3/10

92/92 [==============================] - 1s 16ms/step - loss: 0.8202 - accuracy: 0.6802 - val\_loss: 0.9714 - val\_accuracy: 0.6199

Epoch 4/10

92/92 [==============================] - 1s 16ms/step - loss: 0.6437 - accuracy: 0.7558 - val\_loss: 0.9029 - val\_accuracy: 0.6390

Epoch 5/10

92/92 [==============================] - 1s 16ms/step - loss: 0.4155 - accuracy: 0.8450 - val\_loss: 1.2644 - val\_accuracy: 0.5695

Epoch 6/10

92/92 [==============================] - 1s 16ms/step - loss: 0.3055 - accuracy: 0.8968 - val\_loss: 1.1000 - val\_accuracy: 0.6240

Epoch 7/10

92/92 [==============================] - 1s 16ms/step - loss: 0.1507 - accuracy: 0.9554 - val\_loss: 1.4502 - val\_accuracy: 0.5940

Epoch 8/10

92/92 [==============================] - 1s 15ms/step - loss: 0.0914 - accuracy: 0.9772 - val\_loss: 1.4260 - val\_accuracy: 0.6376

Epoch 9/10

92/92 [==============================] - 1s 16ms/step - loss: 0.0491 - accuracy: 0.9884 - val\_loss: 1.9028 - val\_accuracy: 0.5981

Epoch 10/10

92/92 [==============================] - 1s 15ms/step - loss: 0.0422 - accuracy: 0.9884 - val\_loss: 2.0496 - val\_accuracy: 0.6090

Visualize training results

Create plots of the loss and accuracy on the training and validation sets:

acc = history.history['accuracy']  
val\_acc = history.history['val\_accuracy']  
  
loss = history.history['loss']  
val\_loss = history.history['val\_loss']  
  
epochs\_range = range(epochs)  
  
plt.figure(figsize=(8, 8))  
plt.subplot(1, 2, 1)  
plt.plot(epochs\_range, acc, label='Training Accuracy')  
plt.plot(epochs\_range, val\_acc, label='Validation Accuracy')  
plt.legend(loc='lower right')  
plt.title('Training and Validation Accuracy')  
  
plt.subplot(1, 2, 2)  
plt.plot(epochs\_range, loss, label='Training Loss')  
plt.plot(epochs\_range, val\_loss, label='Validation Loss')  
plt.legend(loc='upper right')  
plt.title('Training and Validation Loss')  
plt.show()

The plots show that training accuracy and validation accuracy are off by large margins, and the model has achieved only around 60% accuracy on the validation set.

The following tutorial sections show how to inspect what went wrong and try to increase the overall performance of the model.

Overfitting

In the plots above, the training accuracy is increasing linearly over time, whereas validation accuracy stalls around 60% in the training process. Also, the difference in accuracy between training and validation accuracy is noticeable—a sign of [overfitting](https://www.tensorflow.org/tutorials/keras/overfit_and_underfit).

When there are a small number of training examples, the model sometimes learns from noises or unwanted details from training examples—to an extent that it negatively impacts the performance of the model on new examples. This phenomenon is known as overfitting. It means that the model will have a difficult time generalizing on a new dataset.

There are multiple ways to fight overfitting in the training process. In this tutorial, you'll use data augmentation and add dropout to your model.

Data augmentation

Overfitting generally occurs when there are a small number of training examples. [Data augmentation](https://www.tensorflow.org/tutorials/images/data_augmentation) takes the approach of generating additional training data from your existing examples by augmenting them using random transformations that yield believable-looking images. This helps expose the model to more aspects of the data and generalize better.

You will implement data augmentation using the following Keras preprocessing layers: [tf.keras.layers.RandomFlip](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomFlip), [tf.keras.layers.RandomRotation](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomRotation), and [tf.keras.layers.RandomZoom](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomZoom). These can be included inside your model like other layers, and run on the GPU.

data\_augmentation = keras.Sequential(  
  [  
    layers.RandomFlip("horizontal",  
                      input\_shape=(img\_height,  
                                  img\_width,  
                                  3)),  
    layers.RandomRotation(0.1),  
    layers.RandomZoom(0.1),  
  ]  
)

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

Visualize a few augmented examples by applying data augmentation to the same image several times:

plt.figure(figsize=(10, 10))  
for images, \_ in train\_ds.take(1):  
  for i in range(9):  
    augmented\_images = data\_augmentation(images)  
    ax = plt.subplot(3, 3, i + 1)  
    plt.imshow(augmented\_images[0].numpy().astype("uint8"))  
    plt.axis("off")

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:5 out of the last 5 calls to <function pfor.<locals>.f at 0x7fb800397ee0> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce\_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling\_retracing and https://www.tensorflow.org/api\_docs/python/tf/function for more details.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:6 out of the last 6 calls to <function pfor.<locals>.f at 0x7fb800397f70> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce\_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling\_retracing and https://www.tensorflow.org/api\_docs/python/tf/function for more details.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

You will add data augmentation to your model before training in the next step.

Dropout

Another technique to reduce overfitting is to introduce [dropout](https://developers.google.com/machine-learning/glossary#dropout_regularization) regularization to the network.

When you apply dropout to a layer, it randomly drops out (by setting the activation to zero) a number of output units from the layer during the training process. Dropout takes a fractional number as its input value, in the form such as 0.1, 0.2, 0.4, etc. This means dropping out 10%, 20% or 40% of the output units randomly from the applied layer.

Create a new neural network with [tf.keras.layers.Dropout](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout) before training it using the augmented images:

model = Sequential([  
  data\_augmentation,  
  layers.Rescaling(1./255),  
  layers.Conv2D(16, 3, padding='same', activation='relu'),  
  layers.MaxPooling2D(),  
  layers.Conv2D(32, 3, padding='same', activation='relu'),  
  layers.MaxPooling2D(),  
  layers.Conv2D(64, 3, padding='same', activation='relu'),  
  layers.MaxPooling2D(),  
  layers.Dropout(0.2),  
  layers.Flatten(),  
  layers.Dense(128, activation='relu'),  
  layers.Dense(num\_classes, name="outputs")  
])

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

Compile and train the model

model.compile(optimizer='adam',  
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),  
              metrics=['accuracy'])

model.summary()

Model: "sequential\_2"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

sequential\_1 (Sequential) (None, 180, 180, 3) 0

rescaling\_2 (Rescaling) (None, 180, 180, 3) 0

conv2d\_3 (Conv2D) (None, 180, 180, 16) 448

max\_pooling2d\_3 (MaxPooling (None, 90, 90, 16) 0

2D)

conv2d\_4 (Conv2D) (None, 90, 90, 32) 4640

max\_pooling2d\_4 (MaxPooling (None, 45, 45, 32) 0

2D)

conv2d\_5 (Conv2D) (None, 45, 45, 64) 18496

max\_pooling2d\_5 (MaxPooling (None, 22, 22, 64) 0

2D)

dropout (Dropout) (None, 22, 22, 64) 0

flatten\_1 (Flatten) (None, 30976) 0

dense\_2 (Dense) (None, 128) 3965056

outputs (Dense) (None, 5) 645

Total params: 3,989,285

Trainable params: 3,989,285

Non-trainable params: 0

epochs = 15  
history = model.fit(  
  train\_ds,  
  validation\_data=val\_ds,  
  epochs=epochs  
)

Epoch 1/15

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

2022-10-27 01:21:51.338790: E tensorflow/core/grappler/optimizers/meta\_optimizer.cc:954] layout failed: INVALID\_ARGUMENT: Size of values 0 does not match size of permutation 4 @ fanin shape insequential\_2/dropout/dropout/SelectV2-2-TransposeNHWCToNCHW-LayoutOptimizer

92/92 [==============================] - 11s 86ms/step - loss: 1.2819 - accuracy: 0.4598 - val\_loss: 1.0469 - val\_accuracy: 0.5804

Epoch 2/15

92/92 [==============================] - 8s 84ms/step - loss: 1.0261 - accuracy: 0.6029 - val\_loss: 0.9627 - val\_accuracy: 0.6240

Epoch 3/15

92/92 [==============================] - 8s 84ms/step - loss: 0.8992 - accuracy: 0.6563 - val\_loss: 0.9083 - val\_accuracy: 0.6308

Epoch 4/15

92/92 [==============================] - 8s 85ms/step - loss: 0.8223 - accuracy: 0.7010 - val\_loss: 0.9381 - val\_accuracy: 0.6662

Epoch 5/15

92/92 [==============================] - 8s 85ms/step - loss: 0.7830 - accuracy: 0.6972 - val\_loss: 0.8087 - val\_accuracy: 0.6839

Epoch 6/15

92/92 [==============================] - 8s 85ms/step - loss: 0.7028 - accuracy: 0.7422 - val\_loss: 0.7947 - val\_accuracy: 0.6907

Epoch 7/15

92/92 [==============================] - 8s 84ms/step - loss: 0.6888 - accuracy: 0.7296 - val\_loss: 0.7773 - val\_accuracy: 0.7030

Epoch 8/15

92/92 [==============================] - 8s 83ms/step - loss: 0.6288 - accuracy: 0.7582 - val\_loss: 0.7378 - val\_accuracy: 0.7125

Epoch 9/15

92/92 [==============================] - 8s 83ms/step - loss: 0.6088 - accuracy: 0.7694 - val\_loss: 0.8243 - val\_accuracy: 0.6812

Epoch 10/15

92/92 [==============================] - 8s 85ms/step - loss: 0.5679 - accuracy: 0.7875 - val\_loss: 0.7593 - val\_accuracy: 0.7003

Epoch 11/15

92/92 [==============================] - 8s 85ms/step - loss: 0.5125 - accuracy: 0.8127 - val\_loss: 0.8118 - val\_accuracy: 0.7016

Epoch 12/15

92/92 [==============================] - 8s 85ms/step - loss: 0.4971 - accuracy: 0.8161 - val\_loss: 0.7949 - val\_accuracy: 0.7084

Epoch 13/15

92/92 [==============================] - 8s 84ms/step - loss: 0.4511 - accuracy: 0.8334 - val\_loss: 0.8342 - val\_accuracy: 0.7030

Epoch 14/15

92/92 [==============================] - 8s 84ms/step - loss: 0.4417 - accuracy: 0.8341 - val\_loss: 0.8273 - val\_accuracy: 0.7016

Epoch 15/15

92/92 [==============================] - 8s 83ms/step - loss: 0.4096 - accuracy: 0.8529 - val\_loss: 0.7895 - val\_accuracy: 0.7193

Visualize training results

After applying data augmentation and [tf.keras.layers.Dropout](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout), there is less overfitting than before, and training and validation accuracy are closer aligned:

acc = history.history['accuracy']  
val\_acc = history.history['val\_accuracy']  
  
loss = history.history['loss']  
val\_loss = history.history['val\_loss']  
  
epochs\_range = range(epochs)  
  
plt.figure(figsize=(8, 8))  
plt.subplot(1, 2, 1)  
plt.plot(epochs\_range, acc, label='Training Accuracy')  
plt.plot(epochs\_range, val\_acc, label='Validation Accuracy')  
plt.legend(loc='lower right')  
plt.title('Training and Validation Accuracy')  
  
plt.subplot(1, 2, 2)  
plt.plot(epochs\_range, loss, label='Training Loss')  
plt.plot(epochs\_range, val\_loss, label='Validation Loss')  
plt.legend(loc='upper right')  
plt.title('Training and Validation Loss')  
plt.show()

Predict on new data

Use your model to classify an image that wasn't included in the training or validation sets.

Note: Data augmentation and dropout layers are inactive at inference time.

sunflower\_url = "https://storage.googleapis.com/download.tensorflow.org/example\_images/592px-Red\_sunflower.jpg"  
sunflower\_path = tf.keras.utils.get\_file('Red\_sunflower', origin=sunflower\_url)  
  
img = tf.keras.utils.load\_img(  
    sunflower\_path, target\_size=(img\_height, img\_width)  
)  
img\_array = tf.keras.utils.img\_to\_array(img)  
img\_array = tf.expand\_dims(img\_array, 0) # Create a batch  
  
predictions = model.predict(img\_array)  
score = tf.nn.softmax(predictions[0])  
  
print(  
    "This image most likely belongs to {} with a {:.2f} percent confidence."  
    .format(class\_names[np.argmax(score)], 100 \* np.max(score))  
)

Downloading data from https://storage.googleapis.com/download.tensorflow.org/example\_images/592px-Red\_sunflower.jpg

117948/117948 [==============================] - 0s 0us/step

1/1 [==============================] - 0s 114ms/step

This image most likely belongs to sunflowers with a 95.92 percent confidence.

Use TensorFlow Lite

TensorFlow Lite is a set of tools that enables on-device machine learning by helping developers run their models on mobile, embedded, and edge devices.

Convert the Keras Sequential model to a TensorFlow Lite model

To use the trained model with on-device applications, first [convert it](https://www.tensorflow.org/lite/models/convert) to a smaller and more efficient model format called a [TensorFlow Lite](https://www.tensorflow.org/lite/) model.

In this example, take the trained Keras Sequential model and use [tf.lite.TFLiteConverter.from\_keras\_model](https://www.tensorflow.org/api_docs/python/tf/lite/TFLiteConverter" \l "from_keras_model) to generate a [TensorFlow Lite](https://www.tensorflow.org/lite/) model:

# Convert the model.  
converter = tf.lite.TFLiteConverter.from\_keras\_model(model)  
tflite\_model = converter.convert()  
  
# Save the model.  
with open('model.tflite', 'wb') as f:  
  f.write(tflite\_model)

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting RngReadAndSkip cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting Bitcast cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.

WARNING:tensorflow:Using a while\_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.

WARNING:absl:Found untraced functions such as \_jit\_compiled\_convolution\_op, \_jit\_compiled\_convolution\_op, \_jit\_compiled\_convolution\_op while saving (showing 3 of 3). These functions will not be directly callable after loading.

INFO:tensorflow:Assets written to: /tmpfs/tmp/tmp3ntoypci/assets

INFO:tensorflow:Assets written to: /tmpfs/tmp/tmp3ntoypci/assets

2022-10-27 01:23:59.635916: W tensorflow/compiler/mlir/lite/python/tf\_tfl\_flatbuffer\_helpers.cc:362] Ignored output\_format.

2022-10-27 01:23:59.635970: W tensorflow/compiler/mlir/lite/python/tf\_tfl\_flatbuffer\_helpers.cc:365] Ignored drop\_control\_dependency.

The TensorFlow Lite model you saved in the previous step can contain several function signatures. The Keras model converter API uses the default signature automatically. Learn more about [TensorFlow Lite signatures](https://www.tensorflow.org/lite/guide/signatures).

Run the TensorFlow Lite model

You can access the TensorFlow Lite saved model signatures in Python via the [tf.lite.Interpreter](https://www.tensorflow.org/api_docs/python/tf/lite/Interpreter) class.

Load the model with the Interpreter:

TF\_MODEL\_FILE\_PATH = 'model.tflite' # The default path to the saved TensorFlow Lite model  
  
interpreter = tf.lite.Interpreter(model\_path=TF\_MODEL\_FILE\_PATH)

Print the signatures from the converted model to obtain the names of the inputs (and outputs):

interpreter.get\_signature\_list()

{'serving\_default': {'inputs': ['sequential\_1\_input'], 'outputs': ['outputs']} }

In this example, you have one default signature called serving\_default. In addition, the name of the 'inputs' is 'sequential\_1\_input', while the 'outputs' are called 'outputs'. You can look up these first and last Keras layer names when running Model.summary, as demonstrated earlier in this tutorial.

Now you can test the loaded TensorFlow Model by performing inference on a sample image with [tf.lite.Interpreter.get\_signature\_runner](https://www.tensorflow.org/api_docs/python/tf/lite/Interpreter" \l "get_signature_runner) by passing the signature name as follows:

classify\_lite = interpreter.get\_signature\_runner('serving\_default')  
classify\_lite

<tensorflow.lite.python.interpreter.SignatureRunner at 0x7fb84cc7ff10>

Similar to what you did earlier in the tutorial, you can use the TensorFlow Lite model to classify images that weren't included in the training or validation sets.

You have already tensorized that image and saved it as img\_array. Now, pass it to the first argument (the name of the 'inputs') of the loaded TensorFlow Lite model (predictions\_lite), compute softmax activations, and then print the prediction for the class with the highest computed probability.

predictions\_lite = classify\_lite(sequential\_1\_input=img\_array)['outputs']  
score\_lite = tf.nn.softmax(predictions\_lite)

print(  
    "This image most likely belongs to {} with a {:.2f} percent confidence."  
    .format(class\_names[np.argmax(score\_lite)], 100 \* np.max(score\_lite))  
)

This image most likely belongs to sunflowers with a 95.92 percent confidence.

The prediction generated by the lite model should be almost identical to the predictions generated by the original model:

print(np.max(np.abs(predictions - predictions\_lite)))

1.66893e-06

Of the five classes—'daisy', 'dandelion', 'roses', 'sunflowers', and 'tulips'—the model should predict the image belongs to sunflowers, which is the same result as before the TensorFlow Lite conversion.