

# Multi-Image Classification

# TensorFlow - Keras

EC327 Final Project Documentation

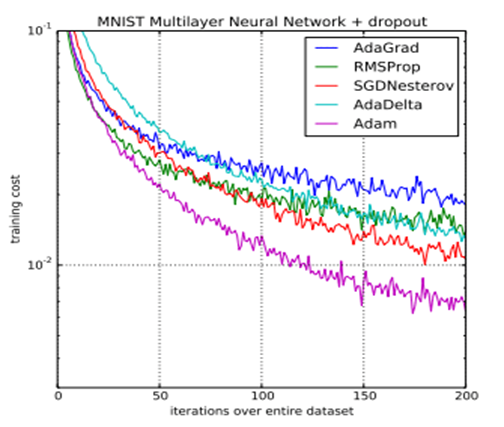
Todd Tian, Purna

# 

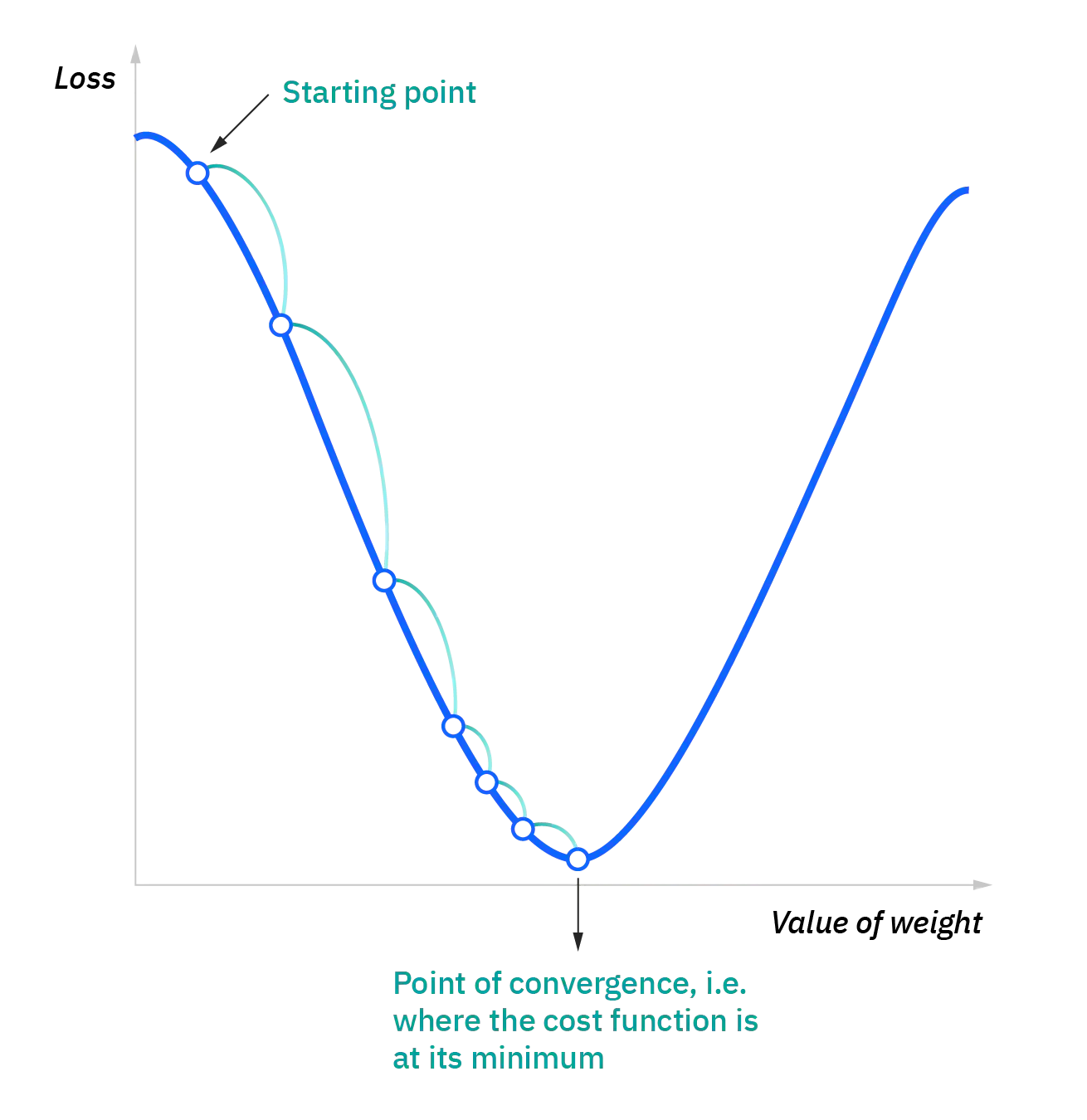
# Introduction

Our model employs the use of an open-source platform called TensorFlow which is developed by Google primarily for deep learning applications. TensorFlow provides pre-built functions and advanced operations to ease the task of building different neural networks. This platform provides a comprehensive and flexible ecosystem of tools, community tools and libraries. Additionally, a high level interface, called Keras, is used in TensorFlow to solve machine learning problems. Keras provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity.

## What is an Image?

Since an image is read as a 2D array with each pixel being represented by a number from 0 to 255. These pixels can be modified to create a grayscale version of the image, or any combination of the colors red, blue, and green. We import these with their original pixel values, but we can modify these values later to ‘stress test’ our model. We decided to standardize our images so that each pixel value would now be between 0 to 1 and we do this for all our models, however, it would be interesting to compare the difference in accuracy for standardized and no standardized images.

## Optimizers and Gradient Descent

Finally, we compile our model using the **‘Adam’ optimizer** which is a type of **gradient descent algorithm**. To put it simply, the Adam optimizer is trying to find the minimal value in our loss function by training an x number of images, and the frequency at which it updates its loss value is dependent on which optimizer we use (**Figure 1**); our model may try to update our loss value for every single image, but this creates more sensitivity and may direct our model away from the minimum of our loss function. Alternatively, we can update our loss value only after a batch of images of x size to decrease the frequency 

that our optimizer updates the loss value. (See **Figure 2** for a visual explanation of what an optimizer seeks to do) By industry standard, the Adam optimizer usually performs much better than the other options and at a better computational efficiency as well.

### Loss Functions

We use S**parseCategoricalCrossentropy** as our loss function as it is the most effective multi-image classification loss function available. There are many types of loss functions available, for example, the BinaryCrossentropy loss function is used for binary classification models (think cat or dog), and MeanSquaredError loss function returns the value with respect to how ‘sure’ the model is when predicting image classes compared to the true class (A 99.99% accuracy would result in a very low MSE vice versa).

## Conv2D

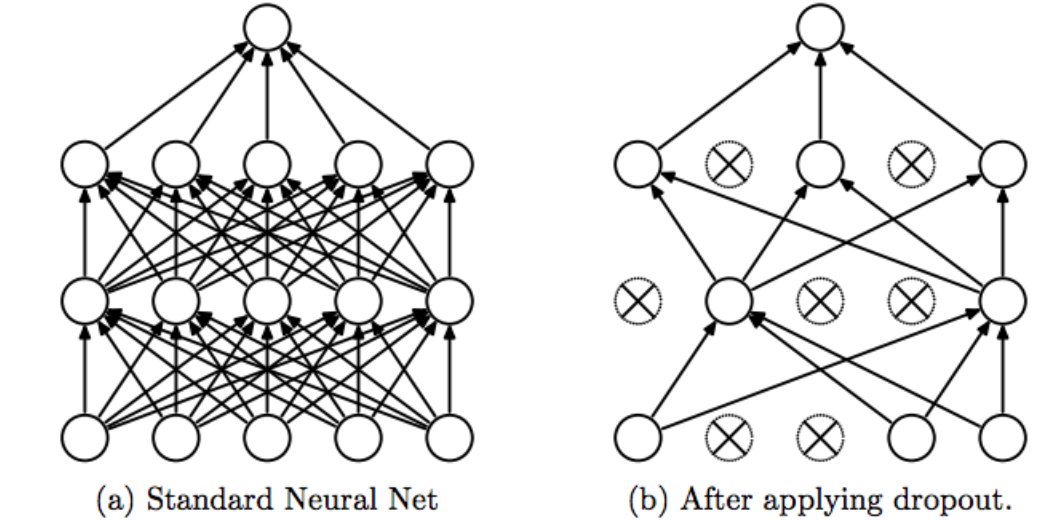
**Conv2D** is a 3x3 filter with each index containing a certain numerical value that we multiply our feature with. There are a variety of features Conv2D can extract given its numerical values. For example, a matrix containing a combination of (-1, 1, 0) could extract edges and form a general shape of the object it’s trying to classify. More complex values could extract textures, depth, and hues. We try to show a relatively intuitive example of a Conv2D layer extracting features from a bird and you can see how it slowly recognizes the general shape and texture as it runs through multiple convolutions on the image. **(Figure 3)**

## MaxPooling2D

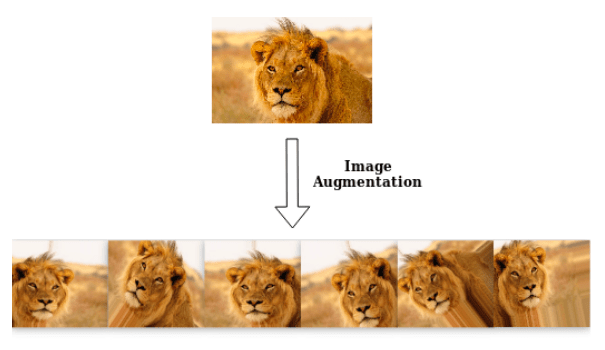
**Pooling** refers to downsampling an image alongs its height and width. For example, if we have a simple convoluted image that is size 2x2, MaxPooling would reduce that image to a size 1x1. Remember that the pixels in our image are represented by

values that range from 0 to 1 after standardization. MaxPooling aggregates the values of a specific section and down samples it to a chosen size—**Figure 4** gives a good visual representation of this process. MaxPooling is useful for our process because it simplifies the feature space after our convolutions making our stack more process efficient. However, as we increase the volume of hidden layers we risk the possibility of overfitting on our training set because we are using the same layers to convolve and pool at each hidden layer and this model quickly becomes rather inflexible for our validation data set.

# Dropout

In our third model we try to combat the problem of overfitting by adding a **Dropout layer** at the end of our neural network. What Dropout does is relatively simple when it comes to functionality, and it is rather elegant because it automates the process without adding in new variables **(Figure 5)**. Say we have a hidden layer with ten neurons, instead of using all ten for all batches of images passed through, we use multiple different combinations of five neurons for each batch. This helps our overfitting problem especially when the number of hidden layers increases because at each level of our NN, our image batches are trained by different subsets of neurons making the model more flexible across our training and validation sets. However, this model still performs relatively poorly because this model now trains double the amount of parameters compared to Model 2. It still overfits significantly but we can see that a simple Dropout layer can lead to a significant improvement in validation accuracy. Model 3 did well to [increase validation accuracy](#_1g3ztqbnyeix), but ultimately was overfitting to our training data due to our increase of hidden layers.

## Data Augmentation

Our first addition was a data augmentation module that introduced images that are randomly flipped, rotated, and zoomed in. This is essentially a stress test that we now force our model to adapt to with every new epoch and this helps because any improvements in accuracy now imply that the model has a deeper understanding of distinct features of each animal class that are not heavily 

affected by translations performed on the image, as shown in **Figure 6**. Other options inherent to the data augmentation module include rgb to grayscale, brightness levels, and cropping. This technique is best used on a relatively small data set because it helps generate multiple synthetic versions of an image that are similar in class but different in orientation.

## Kernel Constraints

One of the major issues with neural networks is the vanishing gradient problem. In a neural network, neurons in a layer have weights that are constantly being changed given our models ‘feel’ of how useful those neurons are in improving accuracy. This sometimes results in our model setting certain weights to near zero, essentially shutting down the layers in the shallow area of our stack. To avoid any large discrepancies between the weights of neurons, we set a maximum weight for each Conv2D filter in each hidden layer so that weights between layers are evenly distributed.

## Intermediate Dropout

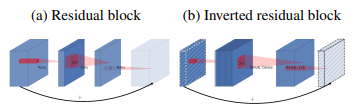
Instead of solely dropping out neurons at the end of the neural net before the last output layer, we can also put these dropout layers between hidden layers. We set the frequency of dropout between layers to 0.5 due to popular research in image classification neural networks. The conclusion is that a frequency of 0.5 is large enough to provide meaningful dropouts that improve on the overfitting problem, but still small enough to retain image information.

# Model 5

## Pretraining on MobileNetV2

For our fifth and final model, we ran our data on **MobileNetV2**, an open source model trained by Google for image recognition. Specifically, MobileNetV2 is an improved convolutional neural network (CNN) algorithm that is based on an *inverted residual structure* where the input and output of the residual blocks are *thin bottleneck layers*, as opposed to how traditional residual models operate. Model 5, unsurprisingly, had the best accuracy rate even on the first epoch. This is because this open source image classifier has been trained on millions of images so that this model would be able to classify virtually any category of images. It’s extremely efficient and able to run through each epoch at a much faster learning rate with no overfitting at all. These models serve as the foundation for image classification that extends to products such as your phone's facial recognition abilities. It is virtually perfect when it comes to static images even if there are several classes.

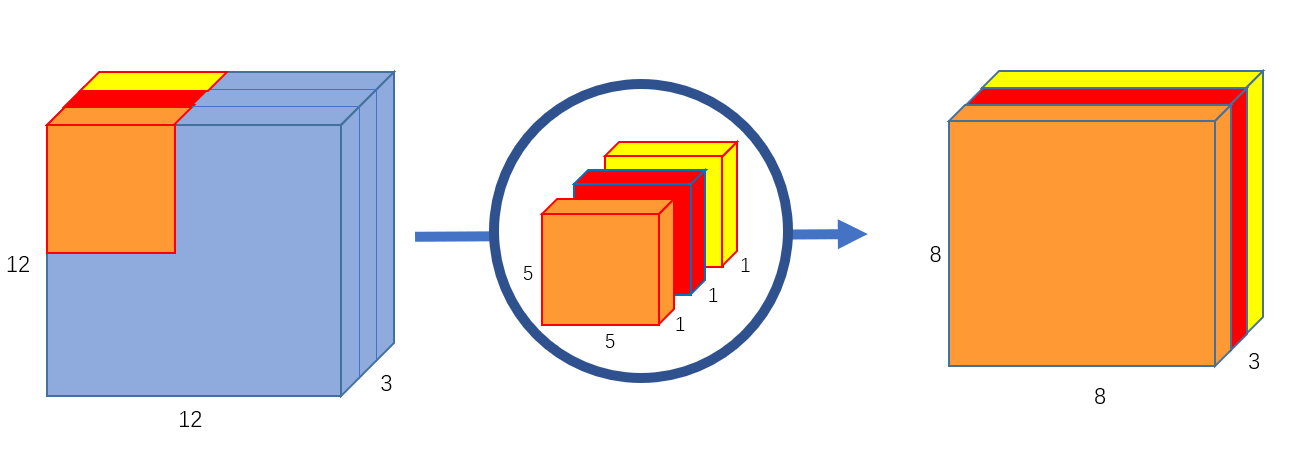
## Inverted Residual Structure and Bottlenecks

To understand what an inverted residual structure is, it helps to understand the differences between that and a normal residual structure. When running a normal residual structure, the hidden layer first filters the features of the images, then squeezes out the most important features from those images. An **inverted residual structure**, on the other hand, squeezes the features using a 1x1 block, then filters the features in a 3x3 block, and finally squeezes the most important features of the image again using a 1x1 block (**Figure 7**). The initial problem with this, however, is that this structure traditionally runs all blocks with the relu activation function, meaning that the squeeze on the final block does not take into account the negative values of the image. This can be detrimental to the model because negative values that are not taken into account can be important when classifying an image. 

For example, referring to **Figure 8**, it would be better to squeeze the features of the dog along with residuals of the grass, compared to squeezing the image where part of the dog’s head may be missing. As a result, MobileNetV2 implements **thin bottleneck layers**. The bottleneck layers remove the Relu activation function at the last 1x1 block, which guarantees the preservation of the entire object of the image we are trying to classify. As a result, this makes the model much more confident when classifying which 

animal is which by creating a model that is much more flexible across the training and validation sets, and decreases the bias for both datasets substantially.

## Depthwise Conv2D and Batch Normalization

Model 5 also makes use of depthwise Conv2D, unlike the traditional Conv2D discussed earlier, and batch normalization. Unlike Conv2D, where the features of an image are split into one layer, **depthwise Conv2D** (as displayed in **Figure 9**) splits the features of an image into multiple layers within the hidden layer, and returns features in separate layers, each representing specific attributes of the image (E.g. One layer representing texture, a second layer representing color, a third layer representing size, etc.). While this separates the features, the problem with this is the layers become very messy because as you go deeper into the neural network and the number of filters applied increases. As a result, **batch normalization** is needed to renormalize each layer that represents the features before you pass into the next hidden layer. Because the features and inputs of the model are normalized, the implementation of depthwise conv2D and batch normalization allows the model to become much more efficient, requiring the running of less epochs.

# Conclusion

What we’ve taken away from diving into this platform is that computational power is key. Along with model efficiency, it is rather difficult to iterate on a data set when each epoch run takes several hours. During our workflow, we’ve gained a new appreciation and admiration for open source R&D projects and supercomputers in general. This project would’ve been virtually impossible if not for the open source tools developed by the Tensorflow and Keras team. Machine learning is hard but creating machine learning tools is even harder.