Convolutional Neural Network Case Study - Binary Classification of Cats and Dogs

Eric Zhuang, Kan Zhang

June 2024

1 Introduction

The purpose of a binary classification machine learning model is to distinguish which of the two categories a given sample belongs to, for example, aiming at Disease Diagnosis and Early Screening and Financial Fraud Detection and Credit Scoring. Although this model can distinguish samples, it still has uncertain accuracy, recall, and precision. Our paper provides a detailed walkthrough of a mature model code that classifies cats and dogs, including interpretive code. We also explain how a computer learns through training, validation, and tests datasets to classify deterministic factors that are critical to model development. Finally, we adjust several parameters on the mature model to test for the importance of each factor.

2 Neural Networks

2.1 What is a Neural Network?

A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by a human brain. But how does it teach the computers? Well, let's consider an example with a computer trying to identify hand-written number digits as shown in Figure 1.

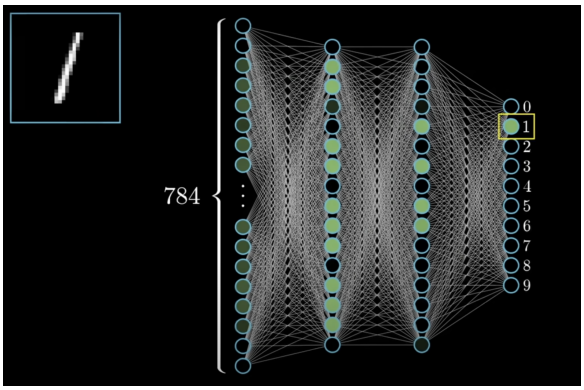


Figure 1

The hand-written number digits can be broken down to a total of $28 \times 28 = 784$ pixels, and this forms the first layer of our neural network. Each pixel can be seen as a neuron, so the first layer of our neural network has a total of 784 neurons. On top of that, each neuron is assigned a real number ranging from 0 to 1, with a value closer to 1 representing a lighter pixel. These numbers are known as the activation of neurons as shown in Figure 2.

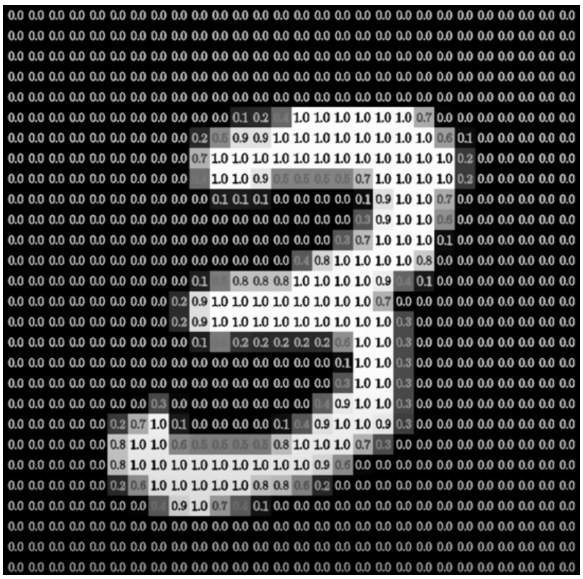


Figure 2

To make our life easier, let's go over the outline of our neural network. Our neural network has a total of 4 layers, with the first layer consisting of 784 neurons, the second and third layers with 16 layers, and 10 neurons for the last layer. Apparently, the 10 neurons in the last layer represent each integer from 0 to 9. Thus, we have a total of $784 + 16 + 16 + 10 = 826$ neurons to work with. Then, let's break down what each layer of our neural network does. As mentioned above, the first layer has all the inputs from a 28×28 black and white picture. Well, the middle two layers are where things go crazy and complicated. We can imagine the second layer aims to break down the digits into different parts. For instance, a 9 can be seen as a circle and a line. After that, the third layer can classify the sub-components of each digit like a circle and a line as shown in Figure 3.

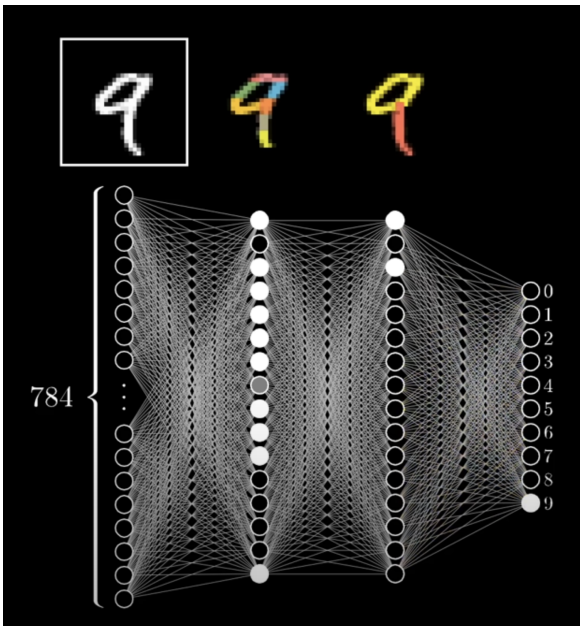


Figure 3

This is accomplished by utilizing the activations in the previous layers and a numeric value called weighted sum. Basically, each neuron is assigned a distinct weight that can be any real number. The weighted sum, as the name implies, simply adds each neuron's activation and multiplies it by its weight. The presence of weights makes the classification of sub-components more lucid. Let's say we want to classify a horizontal edge in the picture, we could then analyze the neurons that are presented in that region and its vicinity. If we assign the neurons that are in the edge, or the expected bright region, with positive weights and the expected dark regions with negative weights, the weighted sum will be large. On the other hand, if a neuron in the expected dark region is pretty bright, meaning it has a quite high activation, then the weighted sum will be relatively smaller. Therefore, the weights of each neuron help us to see a more significant difference between light and dark regions. Figure 4 shows a visual illustration of this concept.

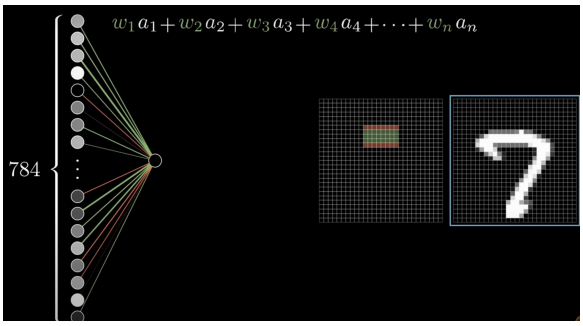


Figure 4

What if we want to artificially influence the classification of certain neurons? Well, we then have to invent a new concept named bias. Straightforward, the name suggests that it has some impact that can bias the result(the result of the classification). Recall that our classification result is based on the weighted sum of neurons. Hence, it's natural to lower the weighted sum if we don't want the computer to classify it to the specific trait. So the equation for calculating the weighted sum requires subtracting a bias term at the end. Remember, the weighted sum is calculated for the activation of each neuron in the next layer according to every neuron in the previous layer. Our weighted sum can be anywhere on the number line, but the activation must be between 0 and 1. Thus, we have to implement a function that can squeeze the number line to 0 and 1. This is called the sigmoid function, where $\sigma(x) = \frac{1}{1+e^{-x}}$, as shown in Figure 5.

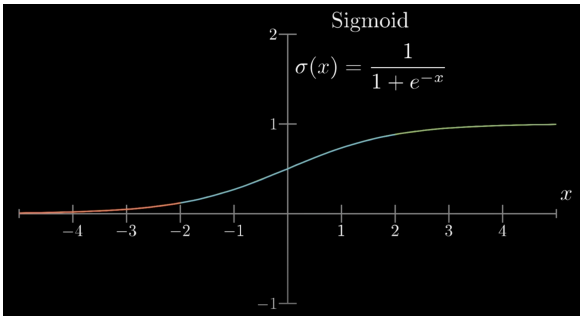


Figure 5

Although the calculation of a single weighted sum seems lucid, we have thousands of similar calculations needed to do. Since there are some similarities between these equations, can we generalize them? Of course! Let's see the matrix form of the weighted sum calculation.

$$\sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \cdot \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}\right)$$

We can name the weights matrix W, the activation vector $a^(0)$, and the bias vector b. We can then generalize the formula to

$$\sigma\left(Wa^{(0)} + b\right)$$

Now, we can change our perspective on neurons. At first, we view them as a capacitor that holds a number ranging from 0 to 1. But now we can consider each neuron as a function that takes all activations in the previous layer as input and produces a number from 0 to 1 as output, which is its activation. With the viewpoint of each neuron being changed, the neural network itself switches to a function, and learning means finding the correct values for weights and biases. This total of $784 \times 16 + 16 \times 16 + 16 \times 10 = 13,002$ variables can be adjusted for our four-layer neural network, as shown in Figure 6 [1].

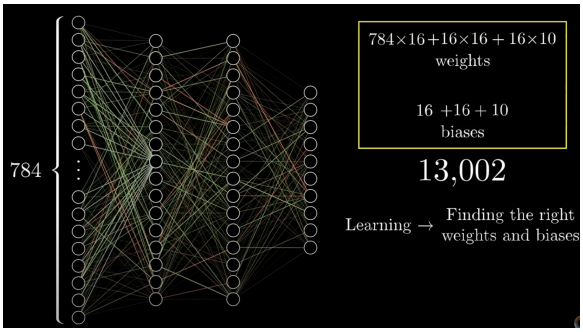


Figure 6

2.2 Gradient Descent

But how exactly does the neural network learn? We would measure the cost. If we give the network random parameters (13,002 weights and biases) and run the simulation, we would get utter trash. The cost is simply the square of differences between each of those trash output activations and the expected value, this is shown in Figure 7.

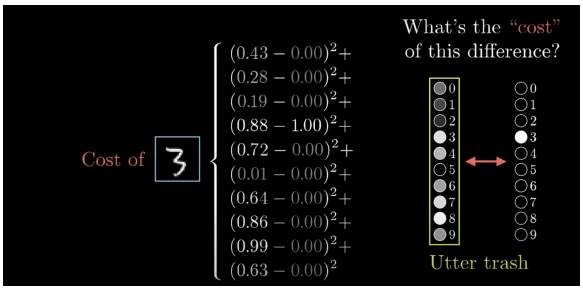


Figure 7

Let's consider the neural network as a function. In our case, this neural network function takes 784 numbers (pixels) as input and has 10 numbers as output with the parameters being the 13,002 weights and biases. If we want to analyze the quality of a neural network, we have to invent a new function - the cost function. Basically, the cost function takes 13,002 weights and biases as input and has 1 number (cost) as output with the parameters being many many training examples. To make things easier, instead of considering the cost function with 13,002 inputs, let's focus on a single input cost function. Our aim is to minimize the cost function, which said by calculus students is to find the zero derivative. However, this couldn't work for our crazy complicated neural network that takes 13,002 different weights and biases as input. What we need is an algorithm named the gradient descent. The specific steps are as follows:

1. Compute ∇C.
2. Move a small step in the $-\nabla C$ direction.
3. Repeat steps 1 and 2.

Where $\nabla C = \frac{\partial C}{\partial x} + \frac{\partial C}{\partial y} + \cdots$. A visual representation is shown in Figure 8.

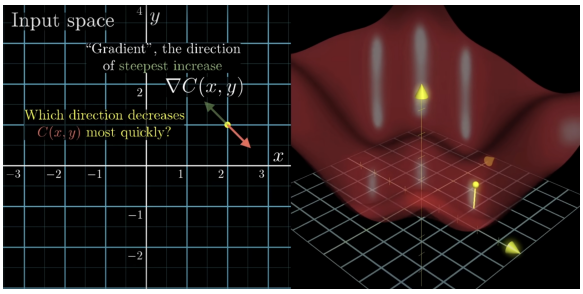


Figure 8

Imagine organizing all 13,002 weights and biases of our network into a giant column vector. The negative gradient of the cost function is just a vector, and it's some direction inside this insanely huge input space that tells you which nudges to all of those numbers are going to cause the most rapid decrease to the cost function.

$$\overrightarrow{W} = \begin{bmatrix} 2.43 \\ -1.12 \\ 1.98 \\ \vdots \\ -1.16 \\ 3.82 \\ 1.21 \end{bmatrix} \quad -\nabla C(\overrightarrow{W}) = \begin{bmatrix} 0.18 \\ 0.45 \\ -0.51 \\ \vdots \\ 0.40 \\ -0.32 \\ 0.82 \end{bmatrix}$$

Now with this in mind, the ways of determining the activation value of a certain neuron according to the neurons in the previous layer don't make sense anymore. What really determines the output neuron's activation is the backpropagation [2].

2.3 Backpropagation

Because the cost function involves averaging a certain cost per example over all the tens of thousands of training examples, the way we adjust the weights and biases for a single gradient descent step also depends on every single example. For now, let's focus on this example of the image of 2 as shown in Figure 9.

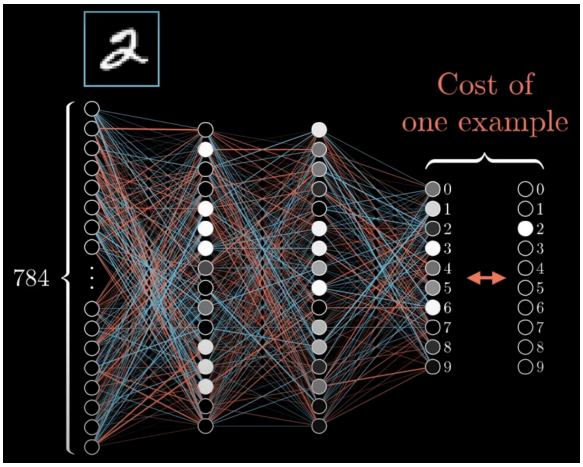


Figure 9

What effect should this one training example have on how the weights and biases get adjusted? Let's say the network is not well-trained yet, so we are going to end up with some random activations on our 4^{th} layer. We cannot directly change the activations since we can only adjust the weights and biases, but it's helpful to keep track of which adjustments we wish should take place to that output layer. Since we want to classify the image of 2, we want the activation of 2 in the 4^{th} layer to get nudged up while others get nudged down. Moreover, the size of the nudges should be proportional to how far away each current value is from its target value. Let's zoom in on the single neuron that we want to increase. Based on the equation of the activation, we have three avenues that can team up together to help increase that activation. We can increase the bias b, increase the weights w_i, and change the activations from the previous layer a_i. Focusing on increasing the weights, we see that the connections with the brightest neurons from the preceding layer have the biggest effect since those weights are tied up with larger activation values. Similarly, when we are focusing on adjusting the activations of the preceding layers, we wish to make changes that are proportional to the size of the corresponding weights. Although we cannot directly influence the activations of the previous layers, we can do similar things to the previous layer and keep a note of what those desired changes are. But remember, we are focusing on the classification of 2, and we

also want all other neurons in the last layer to have a lower activation value. And each neuron has its own tracks in the preceding layers. So, the desire of this digit 2 neuron is added together with the desires of all other output neurons for what should happen to the second-to-last layer, again in proportion to the corresponding weights, and in proportion to how much each of those neurons needs to be changed. This is shown in Figure 10.

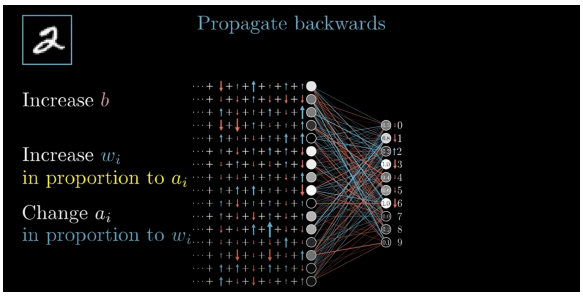


Figure 10

This right here is where the idea of propagating backward comes in. By adding all these desired effects, we basically get a list of nudges that we want to happen to this second-to-last layer. We can recursively apply the same process to the relevant weights and biases that determine those values. In addition, we are going to go through the same backpropagation routine for every other training example, recording how each of them would like to change the weights and biases and average together those desired changes, as shown in Figure 11.

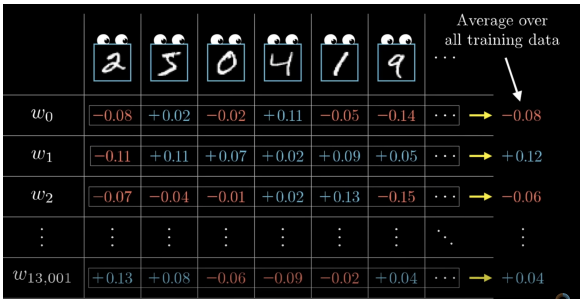


Figure 11

2.4 Stochastic Gradient Descent

In practice, it takes computers an extremely long time to add up the influence of every training example every gradient descent step. Therefore, we instead randomly shuffle and then divide the training data into a whole bunch of mini-batches. Then we compute a step according to the mini-batch. It's not going to be the actual gradient of the cost function, which depends on all the training data, not this tiny subset, so it's not the most efficient way to step downhill, but each mini-batch does give a pretty good approximation, and more importantly, it significantly speeds up the backpropagation process. Figure 12 shows a nice illustration of the gradient descent (GD) and the stochastic gradient descent (SGD) [3].

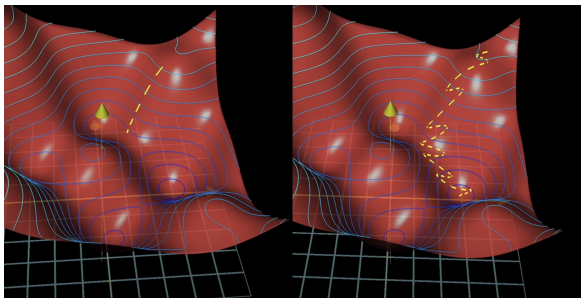


Figure 12

2.5 Backpropagation Calculus

Now, let's see the calculus version that explains this whole process. Let's start off with a simple neural network, one where each layer has a single neuron in it and a total of 4 layers. This network is determined to have 3 weights and 3 biases, and our goal is to understand how sensitive the cost function is to these variables. That way, we know which adjustments to those terms will cause the most efficient decrease in the cost function. We're just going to focus on the connection between the last two neurons. Let's label the activation of that last neuron with a superscript L ($a^{(L)}$), indicating which layer it's in, so the activation of the previous neuron is $a^{(L-1)}$. Next, let's say that the value we want this last activation to be for a given training example is y, for example, y might be 0 or 1. So the cost of this single training example is $C_0(\cdots) = (a^{(L)} - y)^2$. As a reminder, this last activation is determined by a weight $w^{(L)}$ times the previous neuron's activation plus some bias $b^{(L)}$. Then we add a sigmoid function to ensure the activation is within the proper range.

$$a^{(L)} = \sigma\left(w^{(L)}a^{(L-1)} + b^{(L)}\right)$$

To make things easier, let's name this specific weighted sum $z^{(L)}$.

$$a^{(L)} = \sigma(z^{(L)})$$

Our first goal is to understand how sensitive the cost function is to small changes in our weight $w^{(L)}$, or phrase differently, what is the derivative of C_0 with respect to $w^{(L)}$?
We find out that

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}}\frac{\partial a^{(L)}}{\partial z^{(L)}}\frac{\partial C_0}{\partial a^{(L)}}$$

This right here is the chain rule, where multiplying together these three ratios gives us the sensitivity of C_0 to small changes in $w^{(L)}$.
Now, let's compute the relative derivatives.

$$\frac{\partial C_0}{\partial a^{(L)}} = 2\left(a^{(L)} - y\right)$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'\left(z^{(L)}\right)$$

$$\frac{\partial z^{(L)}}{w^{(L)}} = a^{(L-1)}$$

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}}\frac{\partial a^{(L)}}{\partial z^{(L)}}\frac{\partial C_0}{\partial a^{(L)}} = a^{(L-1)}\sigma'\left(z^{(L)}\right)2\left(a^{(L)} - y\right)$$

All of this is the derivative with respect to $w^{(L)}$ only of the cost for a specific training example. Since the full cost function involves averaging together

all those costs across many different training examples, its derivative requires averaging the expression above over all training examples.

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}}$$

And of course, this is just one component of the gradient vector, which itself is built up from the partial derivatives of the cost function with respect to all those weights and biases.

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix}$$

Likewise, for the sensitivity of cost to the bias, we just need to change the $\frac{\partial z^{(L)}}{\partial w^{(L)}}$ for a $\frac{\partial z^{(L)}}{\partial b^{(L)}}$, which is 1.

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} = 1\sigma'\left(z^{(L)}\right) 2\left(a^{(L)} - y\right)$$

Also, we can see how sensitive this cost function is to the activation of the previous layer.

$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} = w^{(L)}\sigma'\left(z^{(L)}\right) 2\left(a^{(L)} - y\right)$$

Now, we can just keep iterating this same chain rule method backward to see how sensitive the cost function is to the previous weights and previous biases. What about we have a more complicated neural network? Honestly, not that much changes when we give the layers multiple neurons, it's just a few more indices to keep track of. Rather than the activation of a given layer simply being $a^{(L)}$, it's also going to have a subscript indicating which neuron of that layer it is. Let's use the letter k to index the layer $L-1$, and the letter j to index the layer L. For the cost, again we are looking at what the desired output is, but this time we add up the squares of the differences between these last layer activations and the desired output.

$$C_0 = \sum_{j=0}^{n_L - 1} \left(a_j^{(L)} - y_j\right)^2$$

Let's call the weight of the edge connecting the k^{th} neuron to the j^{th} neuron $w_{jk}^{(L)}$. Just as before, it's still nice to give a name to the relevant weighted sum, like $z^{(L)}$, so that the activation of the last layer is just a special function applied to $z^{(L)}$.

$$z_j^{(L)} = w_{j0}^{(L)} a_0^{(L-1)} + w_{j1}^{(L)} a_1^{(L-1)} + w_{j2}^{(L)} a_2^{(L-1)} + \cdots$$

$$a_j^{(L)} = \sigma\left(z_j^{(L)}\right)$$

Indeed, the chain-rule derivative expression describing how sensitive the cost is to a specific weight looks essentially the same.

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}$$

What does change here, though, is the derivative of the cost with respect to one of the activations in the $L-1$ layer.

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_l-1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}$$

In this case, the difference is that the neuron influences the cost function through multiple different paths. That is, on the one hand, it influences $a_0^{(L)}$, which plays a role in the cost function, but it also has an influence on $a_0^{(L)}$, which also plays a role in the cost function, and we have to add those up. Once you know how sensitive the cost function is to the activations in the second-to-last layer, we can just repeat the process for all the weights and biases feeding into that layer [4].

3 Convolutional Neural Networks (CNN)

Now that we understand the basics of neural networks, we are going to delve into the convolutional neural network (CNN), which works surprisingly well for image classification. We can still use the basic model for the neural network from the previous section, but we have to change the neurons in the first layer a bit. We are NOT simply naming each pixel as a neuron in our network. Convolutional neural networks replace each neuron in the input layer with a kernel convolution, like a Sobel edge detector.

Let's say our network is still going to classify a human face. Now what we have as input is called a 3-D image, where each image can be broken down into 3 matrices of RGB values ranging from 0 to 255. If we perform a Sobel horizontal edge detection on this, the network will produce another image that is slightly smaller and only one deep as shown in Figure 13.

Figure 13

Hypothetically, it would be an image where the horizontal edges were highlighted since we implemented the Sobel horizontal edge detector. There will only be one output because Sobel just outputs a number between 0 and 255 as soon as we scale it. However, we don't know if Sobel is the best thing for this task. What we do have in convolutional neural networks is let's say 64 of these images on the first layer. The first image will be some convolution process applied to the whole image that takes three input channels and outputs one output channel. The next image will be a different kernel convolution operation, and each image in the first layer has a different kernel. We can imagine the kernel as weights in the previous section about normal learning. With different kernels, one of the images might detect edges, one of them might detect corners, and so on. Then we use these images as our features for learning as a start of doing more features based on these features. We find combinations of edges and combinations of corners since the actual human face is not just a circle of edges, it's several corners and edges combining together.

Kernel windows, which can be visualized as a square matrix, will go through every pixel and produce the output image by giving a single number. For instance, if our kernel window is 3 by 3, which can be viewed as a 3×3 matrix, with 9 numbers, we can use this kernel to slide through every 9 pixels of the input image and output a number by summing the product of corresponding numbers of the kernel window and the original RGB values.

Then we do the same thing on the images in the first layer that produces another image on the second layer, which is a slight combination of, maybe, corners and edges, and there will also be lots of images in the second layer. Moreover, as we keep sliding the kernel through images and produce new images, which is known as the convolution operation, we reduce the size of these images as shown in Figure 14.

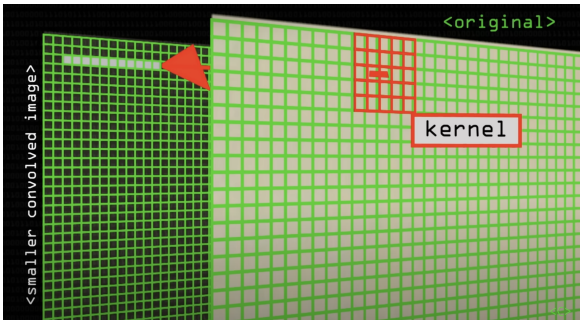


Figure 14

Therefore, we end up with lots of much smaller images in a layer. Each image looks different and represents different things that we don't know what that is, but it's useful for classifying human faces. Eventually, as the number of layers increases, these images will get down to just one pixel and a very very long list of them. Essentially, we completely remove the spatial dimension. There's no more spatial information left, we don't know where anything is. But we know what it is because it's listed in all features as shown in Figure 15.

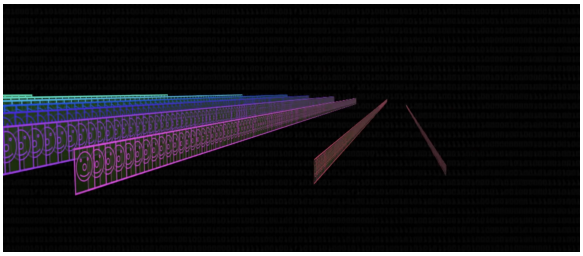


Figure 15

These now are our neurons in the input layer. We then have a couple more layers that point to these neurons, and everything else is the same as the traditional neural network except that we are going to use ReLU instead of the standard sigmoid function. The ReLU activation function is specifically used as a non-linear activation function, as opposed to other non-linear functions such as Sigmoid because it has been empirically observed that convolutional neural networks using ReLU are faster to train than their counterparts, as shown in Figure 16 [5].

$$\text{ReLU}(x) = \max(0, x)$$

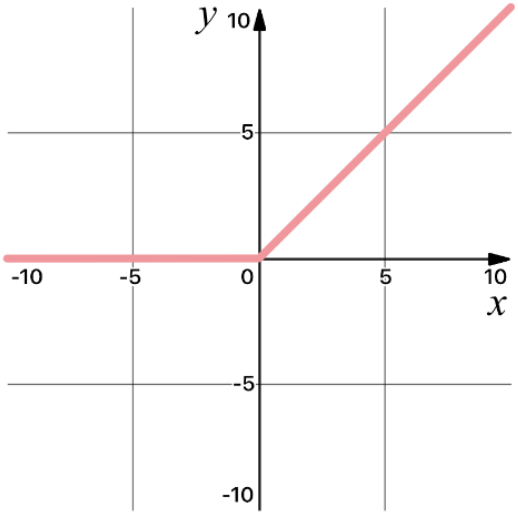


Figure 16

We also conduct the training process by adjusting weights inside all these kernel convolutions [6].

A more intuitive picture of how the convolutional neural network works is shown in Figure 17 [7].

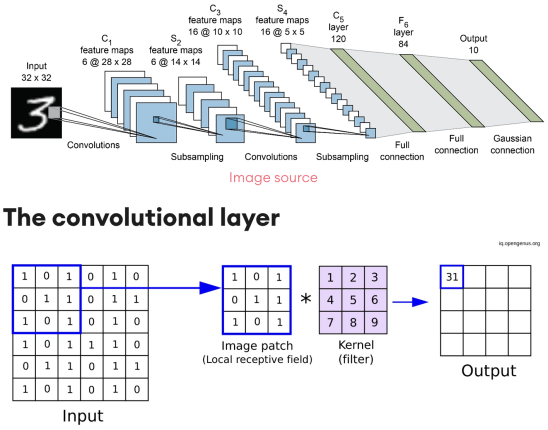


Figure 17

4 Code Analysis

Let's use our current knowledge about machine learning and go through a model that classifies cats and dogs!

In [1]:
```
# importing libraries
import numpy as np
import random
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.metrics import confusion_matrix
import seaborn as sns
sns.set(style='darkgrid', font_scale=1.4)
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
```

Figure 18

In [2]:
```
# Tensorflow
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications.inception_resnet_v2 import InceptionResNetV2, preprocess_input
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
```

Figure 19

These 16 lines of code in Figure 18 and Figure 19 first import the libraries to be used and set up the environment, using the pre-trained Inceptionresv2 model as the base model. Here is a detailed explanation of each library [8].

Libraries:

numpy: Used for scientific computation in Python, supporting multidimensional array and matrix operations.

random: Generate random numbers for random transformations or sample selection in data enhancement.

matplotlib.pyplot: Used to draw data visualization charts, such as line charts, scatter charts, and histograms.

matplotlib inline: Displays the graphics generated by matplotlib in the Notebook.

sklearn.metrics: Various metrics and functions are provided to evaluate the performance of the model, such as the confusion matrix.

seaborn: A library for rendering statistical data visualizations, providing higher-level interfaces and more aesthetically pleasing default styles.

os: A library that interacts with the operating system to set environment variables or manage file paths, etc.

tensorflow.keras: TensorFlow's advanced API for building and training deep learning models.

ImageDataGenerator: Image data generator for real-time data enhancement and batch data generation.

image: Tools for loading, processing, and preprocessing image data.

InceptionResNetV2: Pre-trained deep learning model InceptionResNetV2 for image feature extraction.

Dense, Flatten: Used to construct the fully connected layer and the flattened layer in the neural network model.

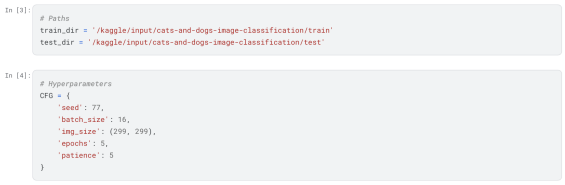
Model: Classes used to define and build Keras models.

Adam: The Adam optimizer is used to optimize the model parameters when compiling the model.

tensorflow: An open-source framework for deep learning that provides a wide range of tools and functions to build, train, and deploy deep learning models.

keras: A high-level neural network API that runs on top of TensorFlow and simplifies the process of building and training deep learning models.

callbacks: A module for Keras that provides a series of callback functions that can be triggered at different stages in the training process to perform specific actions (such as saving the model, adjusting the learning rate, stopping early, etc.).



```
In [3]:
# Paths
train_dir = '/kaggle/input/cats-and-dogs-image-classification/train'
test_dir = '/kaggle/input/cats-and-dogs-image-classification/test'

In [4]:
# Hyperparameters
CFG = {
    'seed': 77,
    'batch_size': 16,
    'img_size': (299, 299),
    'epochs': 5,
    'patience': 5
}
```

Figure 20

train_dir and test_dir in Figure 20 specify the paths of the training and test sets, respectively.

In the CFG dictionary, seed refers to the 77^{th} method and batch_size is the number of images per training. img_size is set to the size of the image (299, 299).

Epochs are the number of rounds of training, which is set here to 5 rounds of training, and patience is the parameter used for an early stop in model training, indicating how many consecutive rounds stop training when there is no lift.

Data augmentation

```
3]:
# Augment train set only
train_data_generator = ImageDataGenerator(
        validation_split=0.15,
        rotation_range=15,
        width_shift_range=0.1,
        height_shift_range=0.1,
        preprocessing_function=preprocess_input,
        shear_range=0.1,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')

val_data_generator = ImageDataGenerator(preprocessing_function=preprocess_input, val
idation_split=0.15)
test_data_generator = ImageDataGenerator(preprocessing_function=preprocess_input)
```

Figure 21

validation_split = 0.15: Divides 15% of the data from the training set as the validation set.

rotation_range = 15: The angle range of the randomly rotated image is ±15 degrees.

width_shift_range = 0.1: The range of random horizontal translation of the image is 10% of the image width.

height_shift_range = 0.1: The range of random translation in the vertical direction of the image is 10% of the image height.

preprocessing_function = preprocess_input: Preprocessing function for preprocessing images.

shear_range = 0.1: Shear strength, the magnitude of the miter transformation.

zoom_range = 0.2: The range of the randomly scaled image is 20%.

horizontal_flip = True: Flip the image horizontally at random.

fill_mode = 'nearest': The pixel is filled in the nearest neighbor mode.

This section of code as shown in Figure 21 uses TensorFlow's ImageDataGenerator class. First, train_data_generator uses 15% of the data for validation by setting validation_split = 0.15, and applies a series of data enhancement techniques. Includes rotation, horizontal and vertical translation, cross-cutting, scaling, and horizontal flipping, and makes the preprocess_input function preprocess. Second, val_data_generator creates a validation data generator that also uses the preprocess_input function and splits 15% of the data into a validation set without data enhancement.

Data

```
4]:
# Connect generators to data in folders
train_generator = train_data_generator.flow_from_directory(train_dir, target_size=CF
G['img_size'], shuffle=True, seed=CFG['seed'], class_mode='categorical', batch_size=
CFG['batch_size'], subset="training")
validation_generator = val_data_generator.flow_from_directory(train_dir, target_size
=CFG['img_size'], shuffle=False, seed=CFG['seed'], class_mode='categorical', batch_s
ize=CFG['batch_size'], subset="validation")
test_generator = test_data_generator.flow_from_[Main Notebook Content]r, target_size=CFG['
img_size'], shuffle=False, seed=CFG['seed'], class_mode='categorical', batch_size=CF
G['batch_size'])

# Number of samples and classes
nb_train_samples = train_generator.samples
nb_validation_samples = validation_generator.samples
nb_test_samples = test_generator.samples
classes = list(train_generator.class_indices.keys())
print('Classes:'+str(classes))
num_classes = len(classes)
```

Figure 22

nb_train_samples = train_generator.samples: The number of samples for the training set.

nb_validation_samples = validation_generator.samples: The number of samples for the validation set.

nb_test_samples = test_generator.samples: The number of samples in the test set.

classes = list(train_generator.class_indices.keys()): List of class names.

print('Classes:' + str(classes)): Prints the name of the class.

num_classes = len(classes): Indicates the number of classes.

The code in Figure 22 suggests that train_generator creates a training data generator object that loads images from the previous data augmentation and preprocesses and enhances them. In the train_generator part of the train, the image order is randomly disturbed; in the validation_generator part of the validation, the image order is not randomly disturbed; in the test_generator part of the test, the image order is not randomly disturbed

For the 'Number of samples and classes', Sample number and category information is obtained to ensure that data is loaded and processed correctly when training and evaluating machine learning models. The explanation of this code is to process the function of some editing of photos [9].

In [5]:

```
# Visualise some examples
plt.figure(figsize=(15,15))
for i in range(9):
    ax = plt.subplot(3,3,i+1)
    ax.grid(False)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    batch = train_generator.next()
    imgs = (batch[0] + 1) * 127.5
    label = int(batch[1][0][0])
    image = imgs[0].astype('uint8')
    plt.imshow(image)
    plt.title('cat' if label==1 else 'dog')
plt.show()
```

Figure 23

The code in Figure 23 displays 9 image samples from the training dataset, categorizing them as either cats or dogs while handling image processing.

The code initializes an image window with dimensions of 15×15 inches. It then iterates 9 times to generate 9 subplots, each within a 3x3 grid. For each subplot (indexed by $i + 1$), the code disables grid lines, hides the X and Y-axis coordinates, and retrieves a batch of data from the 'train_generator'. It preprocesses the images, converting pixel values from the range [-1, 1] to [0, 255], and then gets the label of the first image. The processed image is converted to 'uint8' type for display. Finally, the image is displayed in the subplot with a title indicating whether the image is of a cat or a dog, and all subplots are displayed together.

Figure 24

The code in Figure 24 uses the InceptionResNetV2 model in the Keras library and loads the pre-trained weights.

Figure 25

The code in Figure 25 shows a new fully connected layer being added based on the InceptionResNetV2 model to adapt to specific classification tasks. Specifically, the code uses Keras's functional API to build the model and adds a flat layer and a fully connected layer. This can continue to help the computer learn, avoid training from scratch, and solve some of the scarcity problems. By the way, the softmax activation function is

$$\sigma(\overrightarrow{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Load InceptionResNetV2: Full connection layer without the top layer.

Add new layers: Add a flat layer and two fully connected layers.

Create and compile the model: Combine the new layer with the base model, create a new model, and compile it, specifying the optimizer, loss function, and evaluation metric.

Print Model structure: Output an overview of the model to view the structure and parameters of the entire model.

Figure 26

The code in Figure 26 builds a model and freezes all layers in a pre-trained model

such as InceptionResNetV2 and accelerates the training process for transfer learning.

Figure 27

The definition employs the Adam optimizer, an adaptive learning rate optimization algorithm known for its effectiveness and fast convergence in most deep learning tasks. The model is then compiled in preparation for training. The loss function is set to categorical_crossentropy, appropriate for multi-class classification problems. Accuracy is utilized as a metric to evaluate the model's performance during both the training and validation phases [10].

filepath = ' Model.h5 ': Specifies the filepath to save the model.

monitor='val_loss' : Indicates the loss of the monitoring verification set.

save_best_only = True: Save the model only when the validation set loss is reduced, which ensures that the best-performing model is saved.

verbose = 1: Set to 1 to output one piece of information each time the model is saved.

Figure 28

The code in Figure 28 generates training data dynamically, preventing memory overflow by not loading all the data simultaneously. It allows you to specify the number of training steps per epoch, with each step representing a batch size to be processed, thereby determining the total number of epochs for training. An epoch signifies that the entire training dataset is processed once. You can also pass a list of callback functions to customize the training process and control the level of detail in the information displayed—setting it to 'true' will show comprehensive information, including training and validation results for each

epoch. Additionally, you can specify the number of validation steps per epoch, with each step corresponding to a batch of validation data.

```
[ ]:
# Accuracy and Loss Curves
def plot_accuracy_and_loss(history):
    plt.figure(figsize=(15, 6))

    # Accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Training Accuracy', marker='o')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy', marker='o')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)

    # Loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Training Loss', marker='o')
    plt.plot(history.history['val_loss'], label='Validation Loss', marker='o')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()

plot_accuracy_and_loss(history)
```

Figure 29

This code in Figure 29 creates a function called ‘plot_accuracy_and_loss‘ to visualize the accuracy and loss curves for both the training and validation phases. By passing the training history object, ‘history‘, to this function, we can generate and display a graphical representation of the model’s performance throughout the training process.

Figure 30

This code in Figure 30 shows how to draw a confusion matrix for a deep learning model. [11]

y_true = test_generator.classes: Get the real labels from the test data generator.

y_pred = Model.predict (test_generator): A trained model is used to predict the test data, returning the probability for each class.

y_pred = np.argmax(y_pred, axis=1): Converts the probability of the prediction into the category label of the prediction, np.argmax returns the index of the maximum value on the specified axis, i.e., the category of the prediction.

cm: Obfuscate matrix data.

annot=True: Displays the value on each cell.

fmt='d': The value is displayed as an integer.

cmap='Blues': Color mapping uses blue tones to better visualize numerical sizes.

xticklabels=classes and yticklabels=classes: Set the labels for the x and y axes to the class names, respectively (assuming classes are a list containing class names).

Figure 31

The code in Figure 31 selects an image at random from the test set and displays predictions [12].

Load image: Load the image and resize it to the desired size for the model input.

Convert to array: Converts an image object to a NumPy array for further processing.

Extended dimension: Add a dimension so that it becomes a 4D tensor of batch size 1 to conform to the model input format.

Preprocessing: The image data is preprocessed, such as normalization, so that the image data conforms to the input format of the model training.

Model prediction: A trained model is used to make predictions on the preprocessed images, returning probabilities for each category.

Determine the category: Find the category index with the greatest probability, that is, the category index predicted by the model, and find the corresponding category name according to the category index.

Display image: Displays the loaded image in the graphics window.

Set the title: Displays the actual file name and prediction results above the image.

Display graph: Displays an image containing the prediction results.

5 Changing Parameters

Before we change any parameters, let's evaluate the performance of the original model.

Figure 32

The two graphs in Figure 32 show the relationship between each epoch to the accuracy(left) and the loss function (right). As we can see, the accuracy got near 99%! Nevertheless, something strange - our validation accuracy got 100% after training the first epoch! Moreover, our training accuracy slightly decreased after training through the first epoch. Why did this happen? Well, if we look closer at the confusion matrix (Figure 33), we see that only 140 images are being categorized. It's way too small for our model to train 5 full epochs through. This thus can explain why the dataset is too small and only 1 epoch will reach the optimization of our parameters and the weird decrease in training accuracy due to over-fitting.

We also have a confusion matrix from the code to evaluate the performance. The confusion matrix of the original code is shown in Figure 33.

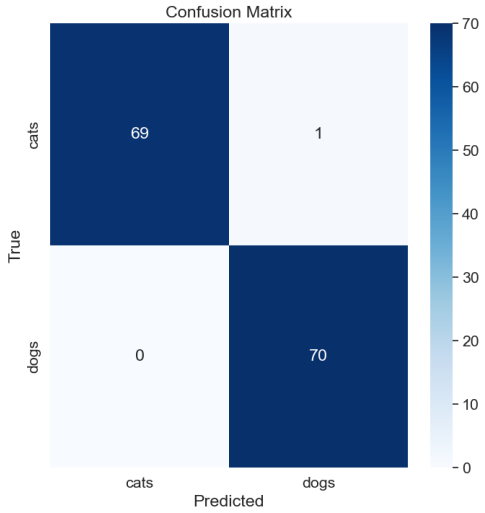


Figure 33

We only have 1 wrong prediction, indicating that the model's performance is extremely high. Let's randomly select a picture from our dataset and see if our prediction matches our labels, as shown in Figure 34.

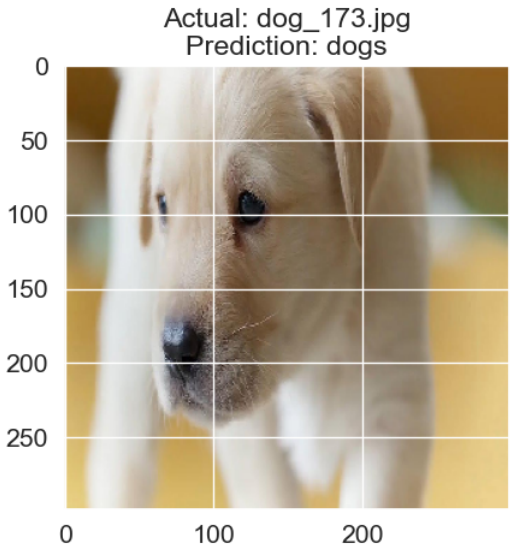


Figure 34

Now, let's change a few parameters. We are going to control the variables, meaning that we only change 1 parameter at a time. First, we changed 'epochs' = 10, and the results are as follows:

Epochs vs Accuracy/Loss graphs

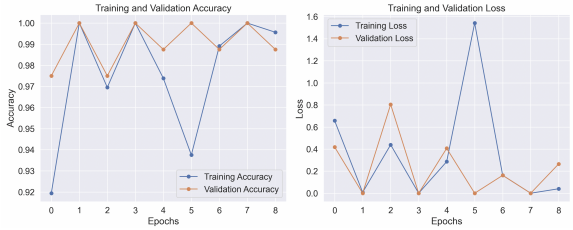


Figure 35

Confusion matrix

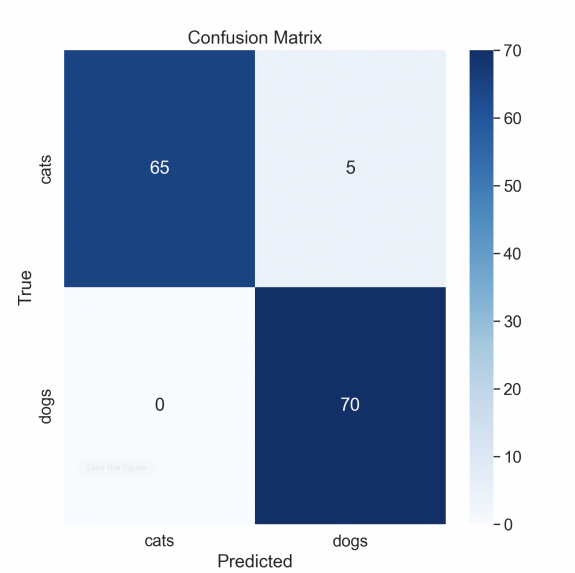


Figure 36

As we increased epochs from 5 to 10, our accuracy slightly decreased, and we had 5 wrong predictions about cats. This is because of over-fitting since we doubled the training process and our dataset is still too small. Now we used the whole training dataset 3 times to train our model, which eventually ended up in a decrease in the model's accuracy. What about decreasing the epoch? The results are as follows:

Epochs vs Accuracy/Loss graphs

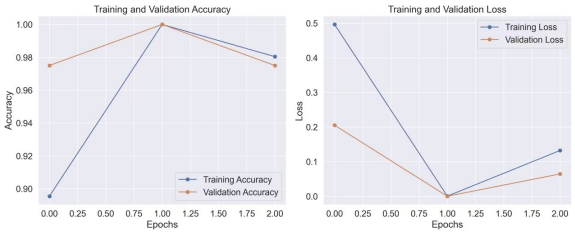


Figure 37

Confusion matrix

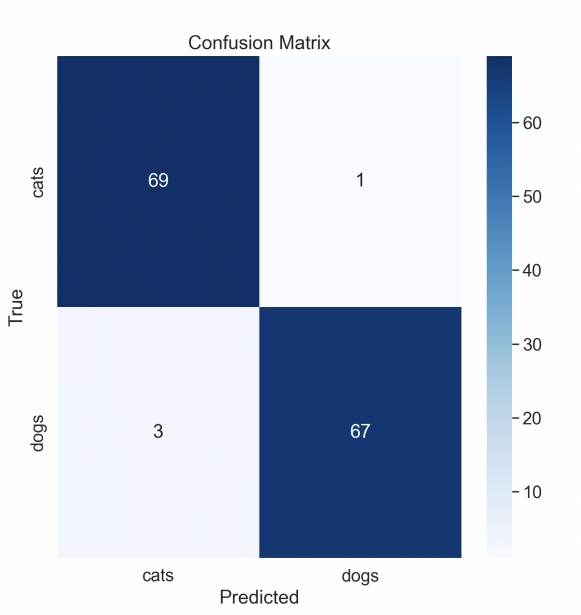


Figure 38

We see that the accuracy also decreases. It's still likely to be over-fitting due to our small dataset. Next, let's change the batch size to 20. The results are as follows:

Epochs vs Accuracy/Loss graphs

Figure 39

Confusion matrix

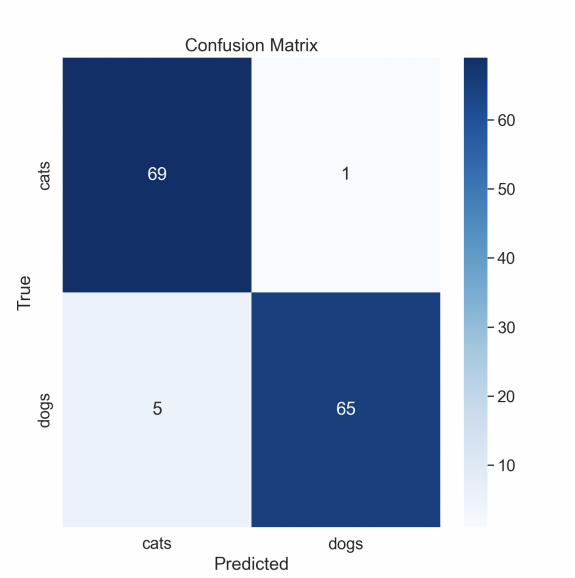


Figure 40

The model's accuracy is even worse. It improves for cat images and declines for dog images. This is might because of the random method modified by 'seed' = 77 that we select more cat images a time than dot images. Still, the weird trends of the accuracy lines are likely to be the result of modeling over such a small dataset containing only 140 pictures. Then, let's change the number of neurons in our newly added dense layer from 100 to 50. We hypothesize that the model's accuracy will decrease. The results are as follows:

Figure 41

Confusion matrix

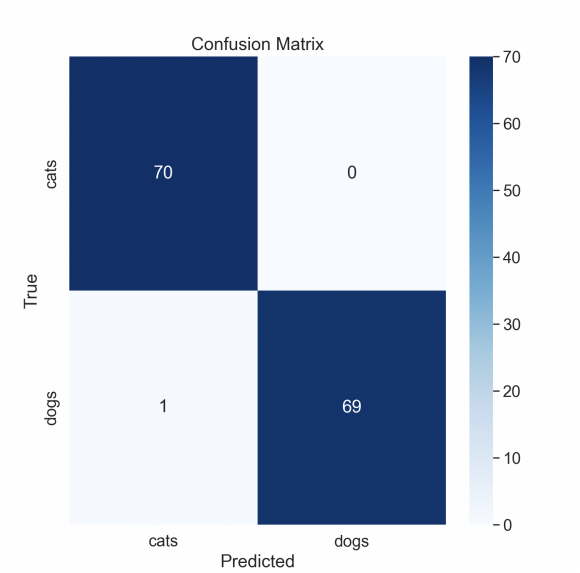


Figure 42

However, we see that the accuracy is about the same as the original model's performance. This could indicate that the number of neurons on our newly added layers doesn't really matter the model's performance since the details and characteristics are being well adopted in the previous few dense layers. To make sure, let's try changing the neurons in the additional layer to 150.

Epochs vs Accuracy/Loss graphs

Figure 43

Confusion matrix

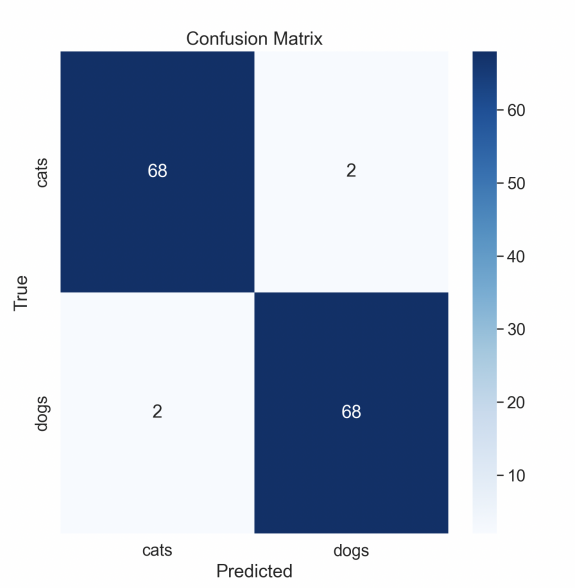


Figure 44

Surprisingly, the accuracy decreases a little bit. Having more neurons might make the model focus on unnecessary details and characteristics of the picture. Or, in this case, the unsubstantial details are useless for a small dataset.

In conclusion, despite having a small dataset, we successfully figured out that epochs, batch size, and the number of neurons in the additional layer all matter, with epochs = 5 being the optimal value. Lower values end up in under-fitting, and larger values end up in over-fitting. A batch size greater than 16 with seed = 77 doesn't ensure an equal amount of pictures of cats and dogs, due to the random method. Last but not least, the neurons in the additional layer are responsible for capturing distinct characteristics between cats and dogs, with 100 being the potential optimum number.

6 Experiment

Now, let's test our model with the images that we collected. We added 20 images of cats and dogs each to our dataset. First, let's take a look at the confusion matrix on how our model performs. The confusion matrix is shown in Figure 45.

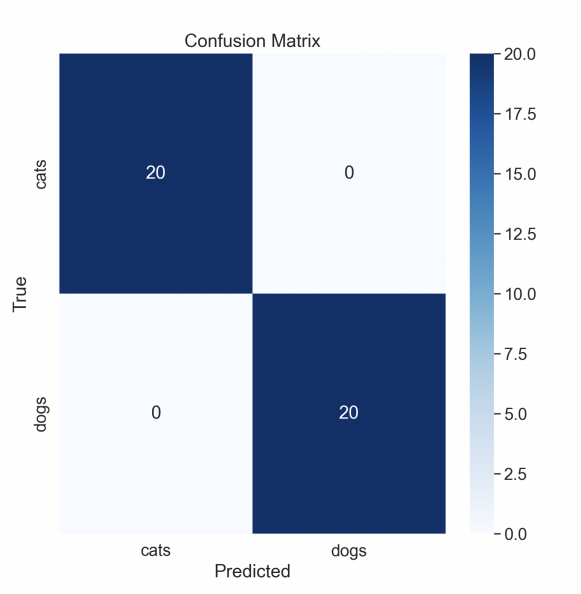


Figure 45

As we can see, our model did perfectly with 100% accuracy in categorizing both cats and dogs! Next, let's examine the accuracy and loss plot as shown in Figure 46.

Figure 46

Though the same problem of a small dataset still exists, the final accuracy

is about 99%, which is not bad. Finally, let's extract some of the images in the newly added dataset and see if the prediction matches with the labels. 4 examples are shown in Figure 47.

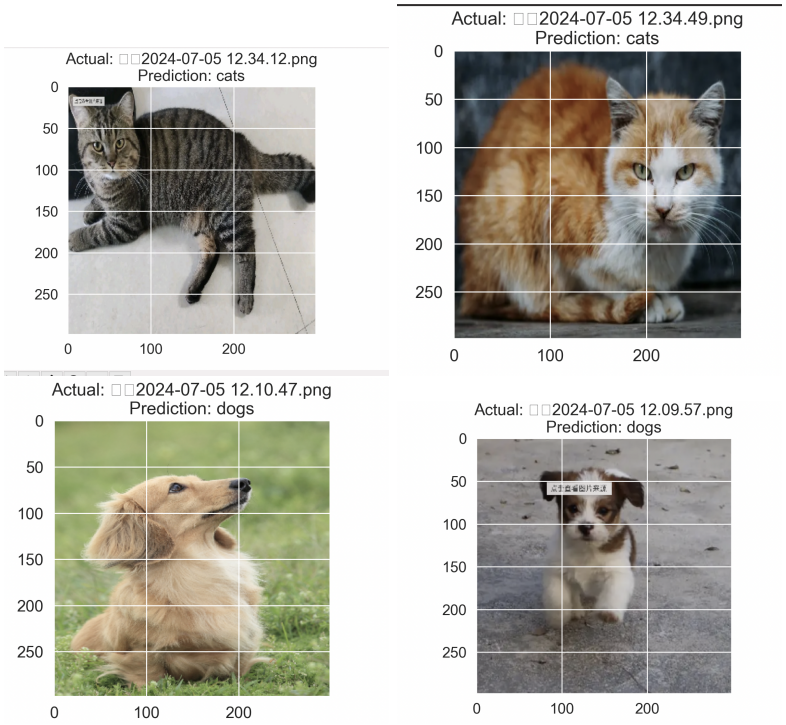


Figure 47

References

[1] "But What Is a Neural Network? — Chapter 1, Deep Learning." YouTube, YouTube, 5 Oct. 2017, https://youtube.com/watch?v=aircAruvnKk&t=2s.

[2] "Gradient Descent, How Neural Networks Learn — Chapter 2, Deep Learning." YouTube, YouTube, 16 Oct. 2017, https://www.youtube.com/watch?v=IHZwWFHWa-w&t=945s.

[3] "What Is Backpropagation Really Doing? — Chapter 3, Deep Learning." YouTube, YouTube, 3 Nov. 2017, https://www.youtube.com/watch?v=Ilg3gGewQ5U.

[4] "Backpropagation Calculus — Chapter 4, Deep Learning." YouTube, YouTube, 3 Nov. 2017, https://www.youtube.com/watch?v=tIeHLnjs5U8.

[5] "CNN Explainer." Polo Club of Data Science @ Georgia Tech: Human-Centered AI, Deep Learning Interpretation & Visualization, Cybersecurity, Large Graph Visualization and Mining. Available at: https://poloclub.github.io/cnn-explainer/. Accessed 14 June 2024.

[6] "CNN: Convolutional Neural Networks Explained - Computerphile." YouTube, YouTube, 20 May 2016, www.youtube.com/watch?v=py5byOOHZM8.

[7] Convolutional Neural Networks: 1998-2023 Overview — Superannotate, www.superannotate.com/blog/guide-to-convolutional-neural-networks. Accessed 14 June 2024.

[8] "Library functions" prompt. ChatGPT, 13 Feb. version, OpenAI, 14 June 2024, https://chat.openai.com.

[9] "Tell me the function of these 6 steps of image preprocessing" prompt. ChatGPT, 13 Feb. version, OpenAI, 18 June 2024, https://chat.openai.com.

[10] "explain the different functions" prompt. ChatGPT, 13 Feb. version, OpenAI, 20 June 2024, chat.openai.com.

[11] "explain the functions of the arguments" prompt. ChatGPT, 13 Feb. version, OpenAI, 20 June 2024, chat.openai.com.

[12] "show the important function called" prompt. ChatGPT, 13 Feb. version, OpenAI, 20 June 2024, chat.openai.com.