Cat or Dog?

Azer Afram
0713566@my.msjc.edu

ABSTRACT

Image classification has been a highly studied and sophisticated field with the introduction of many large models trained on millions of images with the ability to classify thousands of classes. Throughout these models, the use of convolutional neural networks as well as stochastic gradient descent has been proven to be extremely successful. Similarly founded, this model has been built and trained to find the solution to the following problem: Given an image containing either a cat or a dog, identify whether the image contains a cat or a dog. We will cover the model architecture, convolution, and training procedure. We will then evaluate the model's performance, discuss broader questions, doubts, and limitations on neural networks as well as supervised learning.

1 Introduction

Although this problem was first solved by AlexNet [1] in 2012, convolution neural networks, backpropagation, and gradient descent are now ubiquitously used across computer vision, natural language processing, and reinforcement learning. And even though an increase in layers allows a model to represent more complex data, this task is still extremely challenging for a computer since the problem has the potential of being very dynamic which makes it impossible to simply take the image and run some algorithm specifically searching for features that we have in mind. What of the size of these features? The location? The angle? The resolution of the image? The color/texture? These types of major changes in what we are searching for cannot be easily overlooked by a computer.

Though what if we, for some reason, had tens of thousands of labeled images containing only a cat or a dog? Could we somehow have a computer deeply analyze every labeled image in order to find some underlying commonality behind each class of images? Then, maybe, the computer could use these commonalities in order to find any similarity with a given image and the images the computer previously analyzed?

Well, with the help of many conveniently labeled images and learning from its past mistakes our neural network could simply put, teach itself what to look for as well as where to look for it. But how?

2 Data Preprocessing

For the images, Kaggle's Dogs vs Cats dataset was used. This is a dataset put together by Microsoft, but do not download the dataset from the Microsoft website, all the images are corrupted. The problem that we will see very soon here is overfitting. Overfitting happens when our model learns to classify our training data very well, so well that it also learns from the noise (eg. background) of those images. This causes our model to predict images outside of our training data poorly since it is also looking for the noise it has learnt to look for in our training data. This is an obvious problem that we must crack down on since our model is slightly deep. The most intuitive solution to overfitting would be to change our input image before it is fed forward into our network. This gives the network no choice but to find the features that this image shares with its class instead of finding commonalities with the images it has been trained on previously. We only augment input images when training, augmentation is also random.

3 Convolutional Neural Network

A convolutional neural network (CNN) is basically an algorithm that, ideally, takes in an image, uses κ filters/kernels (2D/3D representations of a feature) on the image to get κ corresponding feature maps (2D). Then, to isolate the features in each feature map, we zero and/or drop unimportant points in each feature map. Then we do the same thing over again, but this time, our κ feature maps would be the input. In this case, we, similarly, repeat the above process 3 times in total. Finally, we "flatten" our 2D feature maps into a 1D vector. This 1D vector would then be simplified enough to hopefully, output the correct answer. Now unto the "why" and "how" this happens.

3.1 Convolution

In our case, the input image will be an RGB image. This means that the matrix representation of the image will have 3 channels, each channel corresponding to red, green, and blue respectively. Each one of these channels will be of size 256 × 256, each point will be a value between 0 and 255, larger numbers will correspond to darker tones of either red, green, or blue. A convolution of our image and our kernel will be represented as: image × kernel.

Matrix representation of a 3 × 3 × 3 image:

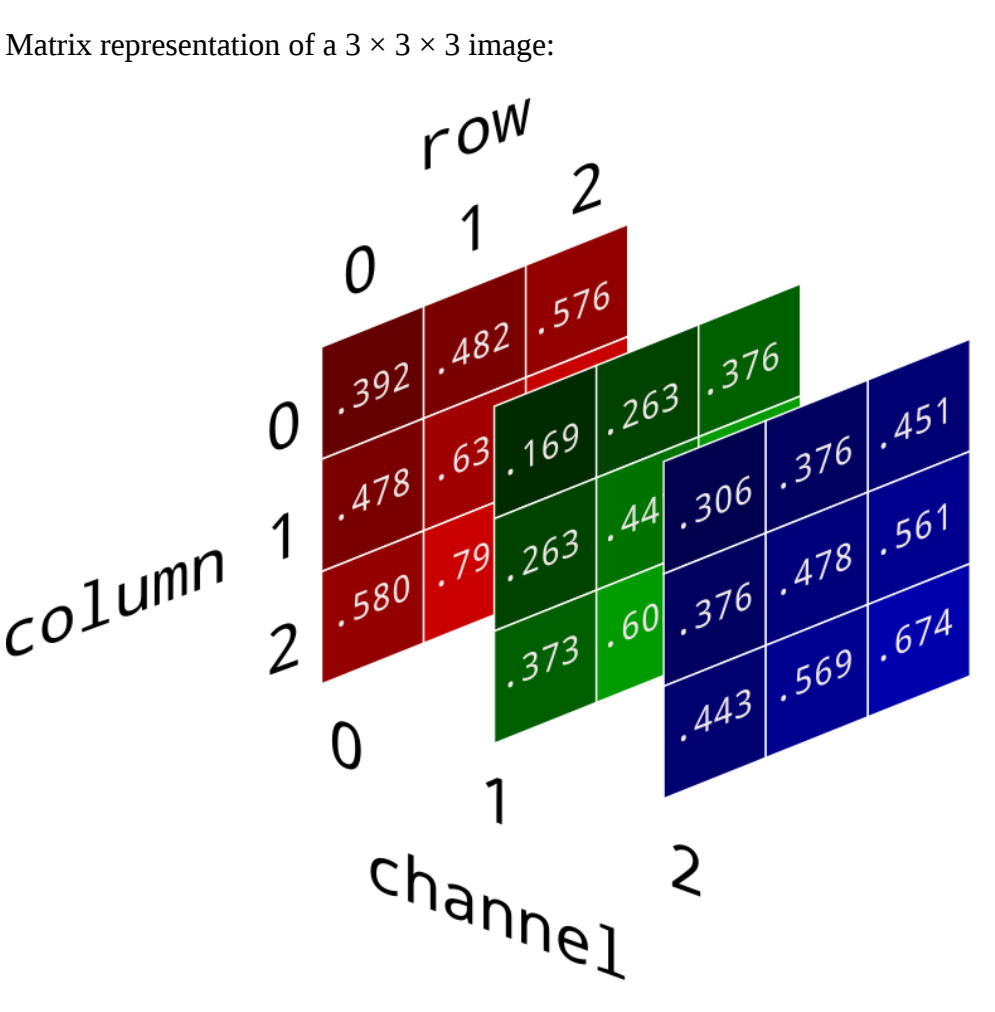


Image from: https://e2eml.school/convert_rgb_to_grayscale.html

Note that the values of the matrix above are only so after we normalize the image. This is done by dividing each value in our matrix by 255. This effectively decreases computation needed to train/test our model. Limiting the matrix to the range [0,1] also causes our input to be on the same scale as our output (which is also in range [0,1]) which helps us avoid the vanishing gradient problem

Convolution of a 256 × 256 × 3 image by a 3 × 3 × 3 kernel (no padding, stride of 1):

$$image \times kernel = v$$

$$R' = R - 2 = 254$$

$$C' = C - 2 = 254$$

$$v_{rc} = \sum_{h=0}^{2} \sum_{r'=r}^{r+2} \sum_{c'=c}^{c+2} image_{r'c'h} \ * \ kernel_{r'-r, c'-c, h}$$

$$where:$$

$$\begin{gathered} R = number\ of\ rows\ in\ image \\ R' = number\ of\ rows\ in\ v \\ C = number\ of\ columns\ in\ image \\ C' = number\ of\ columns\ in\ v \\ h = channels, 3 \\ r \in [0, 253] \\ c \in [0, 253] \end{gathered}$$

In other words, we convolute each channel in the image with its corresponding channel in the kernel for all channels, then add them up to get our first pixel in the convoluted vector. Since our kernel has 3 rows and 3 columns and we do not use padding to convolute, our rows and columns decrease by 2. Note also, we get only one feature map, v, from this convolution.

3.2 ReLU

Kernel values, ideally, would be positive if that point is part of our feature, otherwise negative. This causes our feature map to have high values where a feature was detected, and negative values if that that portion of our image was not similar to the feature. In other words, in our feature map, higher values of a point correspond with a higher similarity to our kernel. To amplify our features then, it makes since to (literally) zero out any negative values in our feature map. This removes any unwanted noise and brings full attention to the features found by our kernel. That is exactly what a rectified linear unit (ReLU) activation function does.

Specifically, for all points $f[r][c]$ in all feature maps $f[i]$:

$$f[r][c] = \max(0, f[r][c])$$

3.3 Pooling

In total, our model generates 448 feature maps, each feature has its own kernel that was used to create it. So, if all those feature maps were of size 256×256 our model would have generated 29360128 values/pixels. That is computationally difficult to do and thus would increase training time. Also, 256×256 is generous amount of space for one feature, it is better for the model to know that there exists a certain feature in a certain portion of the image and not what that feature looks like exactly. And an even larger issue that we must address even further is overfitting, if we do not simplify these features down our model would learn too much from one certain training image and not others that are similar to it. To address all these issues, max pooling takes the maximum point in a 2×2 portion the feature map, then moves on to the next point of our feature map and does it again, effectively amplifying features as well as reducing computation.

In other words:

$$image\ post\ max\ pooling = p$$

$$\begin{gathered} ans = max(image[r][c], image[r+1][c]) \\ ans = \max(ans, image[r][c+1]) \\ ans = \max(ans, image[r+1][c+1]) \end{gathered}$$

$$p[r][c] = ans$$

We use a stride of 1.

3.4 Dropout

Personally, the most challenging part of implementing this model was trying to avoid overfitting. And, even after all precautions that I have taken, I am still forced to overfit the model in order to reach higher validation percentages. Internally, overfitting is caused by our trainable parameters (e.g. kernel values) being overly tweaked in favor of our training images. As we will soon see, it is not the model's fault. The model is trained with 1 mission: reduce the cost function/increase the predicted accuracy as much as possible. So, for a certain image, the model does all it can in order to optimize our output for that image. Optimizing for that image though, can cause the model to overly value or overlook important features that aren't in that image. One of the most effective ways of solving this is dropout. We convolute our image into feature maps, convolute those feature maps, we pool those feature maps, then, we literally "dropout", in this case, 30% of the feature maps. This will cause our model to learn less from a certain image and instead spread out its learning throughout all images.

3.5 Batch Normalization

After we pool the feature maps, we then normalize them. The "why" there is not very clear, yet still batch normalization can be justified using the same reason for normalizing our input image. Again, our input image can have any values from 1 to 255. Now, assume we have a darker input image, this means that the average point/pixel will be around 255. In theory, our model would have performed greater transformations and abstractions on this image in order to get from an input average of 255 to an output in between 0 and 1. This causes the absolute value of the gradients of loss function, with respect to the trainable parameters, to be less. In other words, we are forced to make drastic changes to our trainable parameters in order to affect the output, this can lead to the vanishing gradient problem. Batch normalization is more than that though, we not only normalize the data but also organize it by setting the standard deviation/variance as well as the mean of the data. We do batch normalization at the end of layer l, so the data that layer $l+1$ gets would be predictable. Instead of the input data varying from x to y it would vary at a known range. On top of that, layer $l+1$ would know where the data is varying from since we also set the mean. And since we don't even know what the values of the feature maps would be; it is not optimal for us to arbitrarily set the variance and mean ourselves, so we initially set the mean to 0 and the variance to 1; setting the variance any lower than 1 would take the slight regularization effect of batch normalization to the extreme.

For all feature maps f in the set of feature maps F:

$$\mu = \frac{\sum_{r=1}^{R} \sum_{c=1}^{C} f_{rc}}{R*C}$$

$$\sigma^2 = \frac{\sum_{r=1}^{R} \sum_{c=1}^{C} (f_{rc} - \mu)^2}{R*C}$$

$$f_{rc} = \gamma * \left(\frac{f_{rc} - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) \ + \ \beta$$

$$\begin{gathered} where: \\ \mu = current\ mean \\ \beta = desired\ mean \\ \sigma = current\ variance \\ \gamma = desired\ variance \\ \epsilon = 0.001 \end{gathered}$$

γ, β are learnable parameters, initially set to 1, 0 respectively. ϵ is set to 0.001 so that it wouldn't greatly impact the variance, we take the square root, so there is no point in setting it any smaller. We take the square of the difference, $f_{rc} - \mu$, since we want the absolute value. Simply, dividing the difference between f_{rc} and μ by σ has the same answer as the question: what number, x must we multiply the absolute average difference from the mean (σ) to get the difference from a certain data point f_{rc} to the mean of all of the data (μ)? The thing is the variance is the average of the differences from the data points to the total mean, so on average, we would get $x = 1$. Sometimes x is more than the average (if $f_{rc} - \mu$ is less than σ) and other times its less (if $f_{rc} - \mu$ is more than σ), but on average it will be 1. Thus, the standard deviation of all our updated data points in f would be 1. γ could than scale the differences, increase the standard deviation accordingly, and β would add a constant to all data points and thus change the mean, which would be at 0, accordingly. For more information: Batch Normalization [2], though beware that the title of the paper has been disproved.

4 Flatten and Dense

Now that we have all these high-level features, that have gone through the above process many times, what then? How do we take many output feature maps and turn them into a predicted value of the image's contents? Neural networks can be used to answer that exact question by further abstracting the data while learning what to abstract. Here is one below:

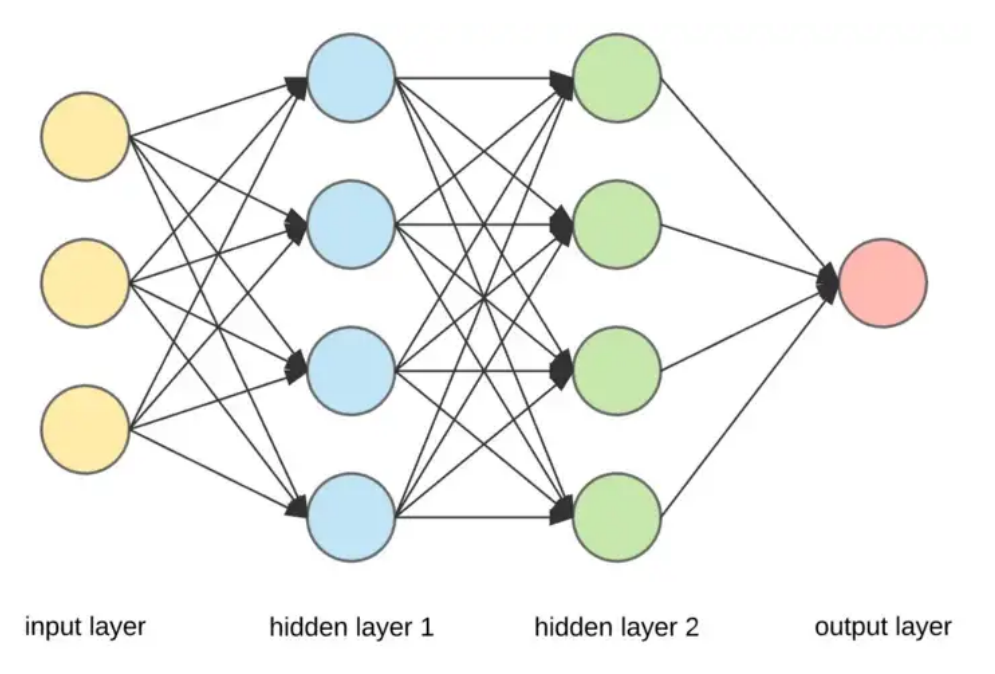


Image from: https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6

The weights (black arrows above) are learnable parameters. The neurons, on the other hand, are not, they are simply the “squishified” summation of all weights connected to it multiplied by the previous corresponding neurons plus the bias (trainable parameter). Looking at the bigger picture, the inputs must be neurons, yet we would currently have feature maps. To go from feature maps to a normal vector we simply "flatten" the feature maps. In other words, we would stack each row right next to each other in order of the initial feature map to make up a 1D vector.

5 Training

The model's weights and biases are initialized randomly which cause it to make random guesses. Yet, with a model deep enough, there is some combination of weights and biases that would yield desired results. In theory, this combination of weights and biases (function) should satisfy all images in our dataset. Knowing this, it is obvious that each image in our dataset should have some impact on these weights and biases. Though, images that were classified inaccurately by the model should not have as much impact as an image that was classified accurately. Yet, at the same time, our model should not change too much for one image since this would cause overfitting.

5.1 Backpropagation

Backpropagation is an algorithm to get the gradient of the loss function (binary cross entropy) with respect to each learnable parameter. The gradient of the loss of the output with respect to some weight, w, is $\frac{\partial L(o)}{\partial w}$. This is the same thing as taking the first order derivative but only looks different since we will need to consider the change it has when fed forward through our network. For example, say we want to get the gradient of the loss of the output with respect to some weight, w_1^2, which we will say is the first weight of the second layer from the output. To get this desired gradient we would have to use the previously calculated gradients of every element that our weight would encounter. In this case, w_1^2 would get multiplied with its neuron (the corresponding output of the previous layer) O_1^3 and then added up with all other neurons and weight multiplications as well as the bias of layer 2, that would make up the input of one neuron on the next layer I_1^2. This input would then go through the ReLU activation function to get the output of that neuron O_1^2, and the process would repeat for neurons from the rest of the layer, and so on. Thus, assuming we begin from the output, we would have the desired gradients of all elements behind us. And so, to get the desired gradient of w_1^2 we would only need the gradient of the input of the neuron from the next layer, I_1^2, with respect to w_1^2. We would than multiply that gradient with the gradient of the loss of the output with respect to the output of that neuron $ReLU(I_1^2)$. This is called the chain rule. Note that the output of neuron whose input is I is ReLU(I).

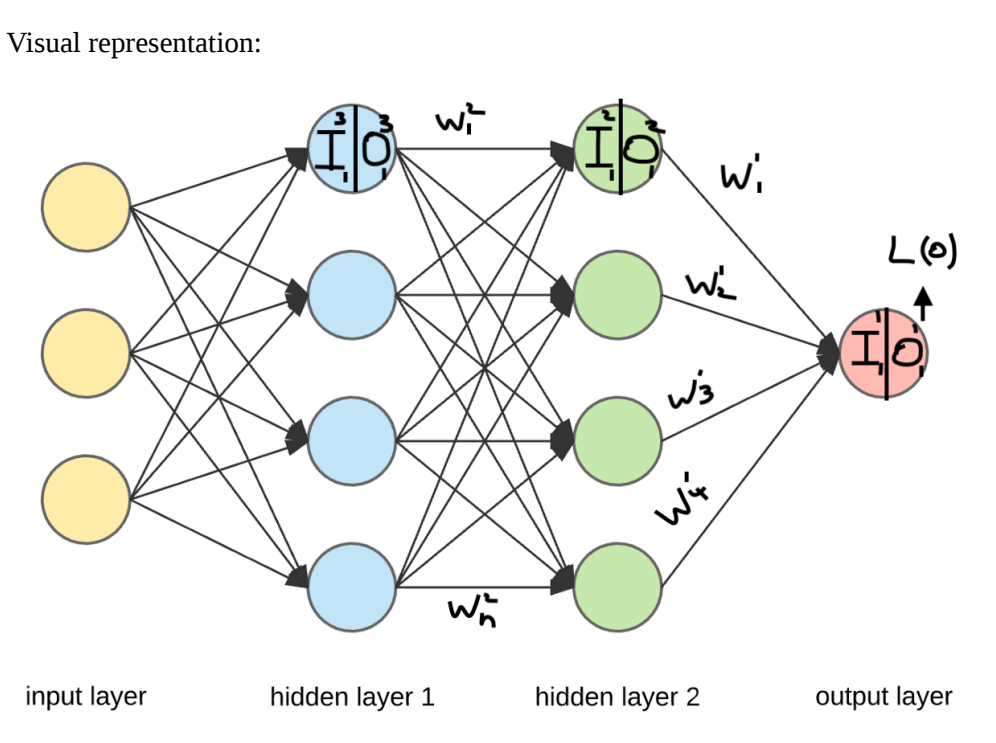
The gradient of the loss of the ouput with respect to weight, w_1^2:

$$o = ReLU(I_1^1) = output$$

$$\frac{\partial L(o)}{\partial w_1^2} = \frac{\partial I_1^2}{\partial w_1^2} * \frac{\partial ReLU(I_1^2)}{\partial I_1^2} * \frac{\partial I_1^1}{\partial ReLU(I_1^2)} * \frac{\partial ReLU(I_1^1)}{\partial I_1^1} * \frac{\partial L(ReLU(I_1^1))}{\partial ReLU(I_1^1)}$$

$$\frac{\partial L(o)}{\partial w_1^2} = \frac{\partial I_1^2}{\partial w_1^2} * \frac{\partial L(o)}{\partial I_1^2}$$

Visual representation:



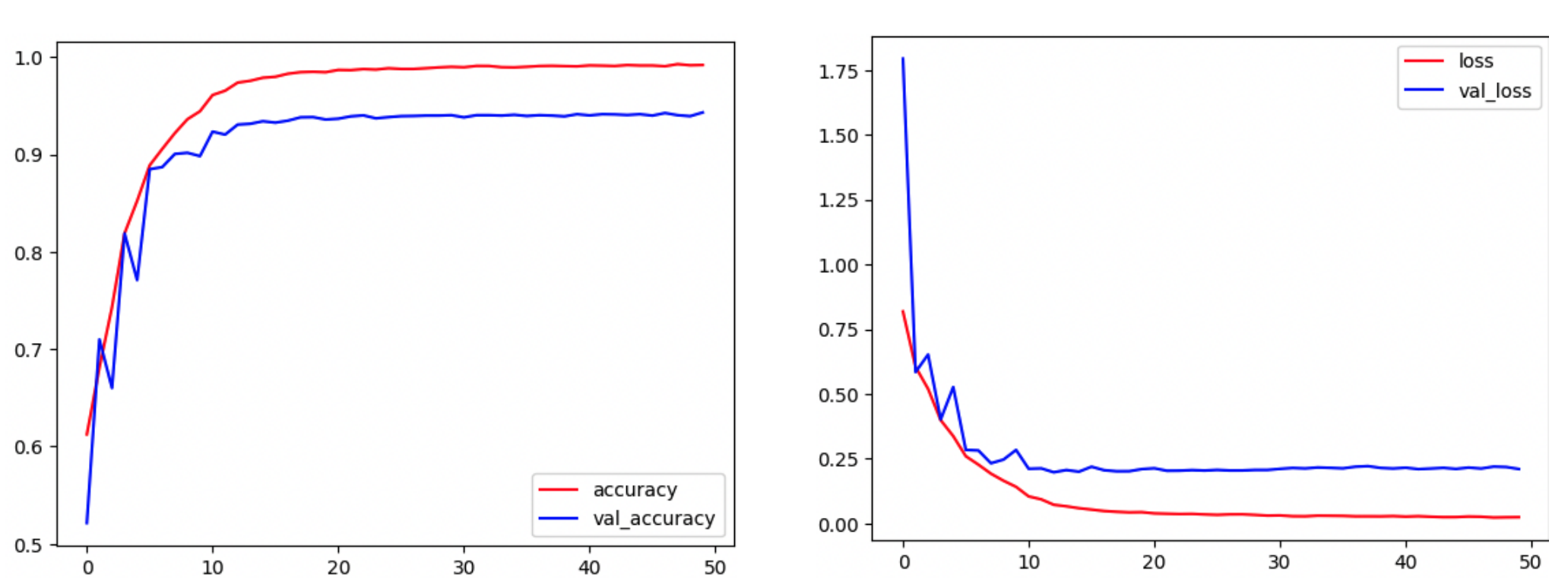
For more information: Backpropagation [3].

5.2 Stochastic Gradient Descent

With the gradients of each learnable parameter, we can tell just how much a small change would affect the loss function which we want to decrease. The loss function essentially tells us how wrong our answer was given the correct answer. So, a decrease in the loss function means an increase in accuracy, we can also choose to increase our accuracy directly instead of decreasing the loss. Since our loss function is convex all we must do is find the local minimum of the function, the input (weights) that would have the least output. To do this we have to change each weight in proportion to the learning rate as well as its gradient. Yet at the same time we must make sure not to change the weights too much, this is to avoid overfitting and overshooting or too little, which would cause our network to be overly "stubborn" with the data. Adam, which is a combination of RMSProp and AdaGrad, does this by giving weight to recent past gradients but more weight to current gradients. Still, it is sometimes beneficial to lower learning rate when needed while training. What makes Adam stochastic is that we do not change the weights of the network for each input, instead we choose a random input from our batch of inputs. This has been proven to converge just as well as other optimizers while using much less computation. The same process is done for the biases of each layer. For more information: Adam [4]

6 Evaluation

After 31,300 calls to gradient descent, and more than 40 hours of training on a CPU. The network achieved 99% accuracy on its training data (20000 images) and a trailing 94% accuracy on the validation data (5000 images). The network, as usual, was never trained on the validation data, only evaluated.



Although the network did slightly overfit to the training data, accuracy on the validation data was always close behind. What's interesting is that the validation accuracy would at times fall behind for a few epochs to then spike back up to the training accuracy. This, I believe, is a sign of stochastic gradient descent [4] being used.

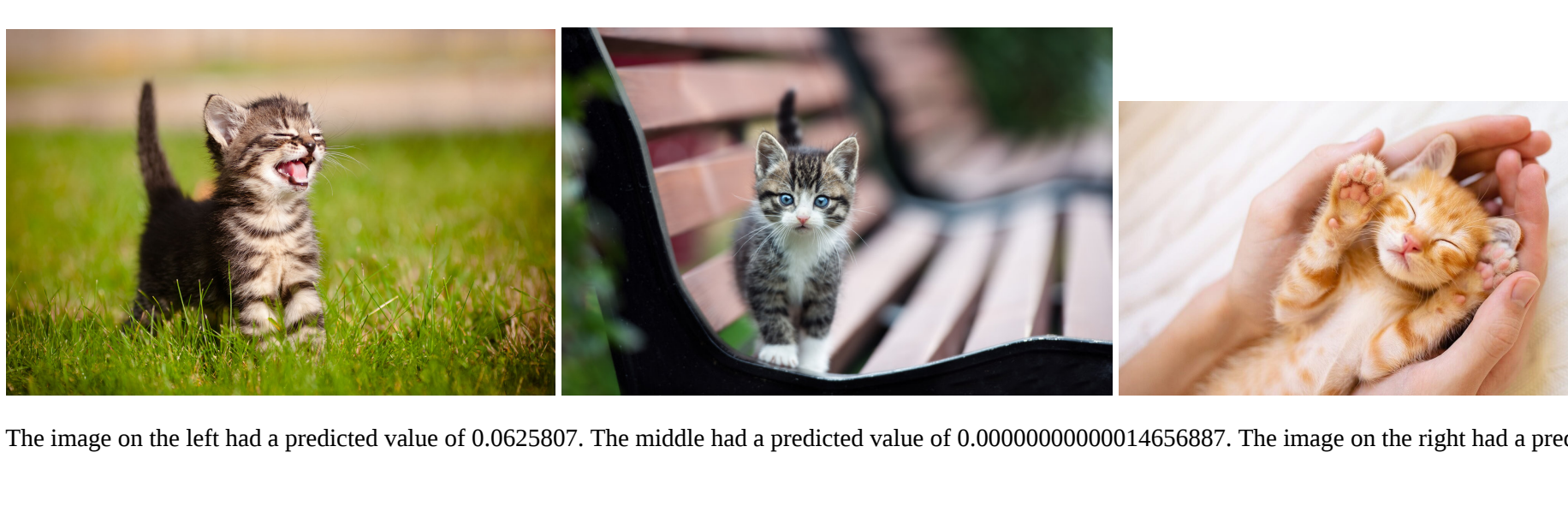
When a convolutional layer is discarded, the validation accuracy plateaus at about 84% accuracy, yet the training accuracy stays the same at about 98%. This is most likely due to the model not being able to represent the high-level features of cats and dogs, but it is still able to "memorize" the training data. In that case, even with more batch normalization done and with higher percentage dropout the model does not do any better than 85% accuracy on the validation data. Validation accuracy improves significantly only when either the number of filters or layers are increased.

The main limitation to the training of this model is the training data. The model sometimes incorecly predicts images when the cat or dog's face is not visible. Looking at the training data, it is obvious that the model was rarely ever trained on such an image. This brings us to another major limitation here and that is the need for more than 20,000 images. This number of images is usually too expensive and time consuming to scrape up. And even with quality training data, that training data is usually never a full, unbiased, realistic representation of the data. These are not only limitations of image classification but supervised learning in general.

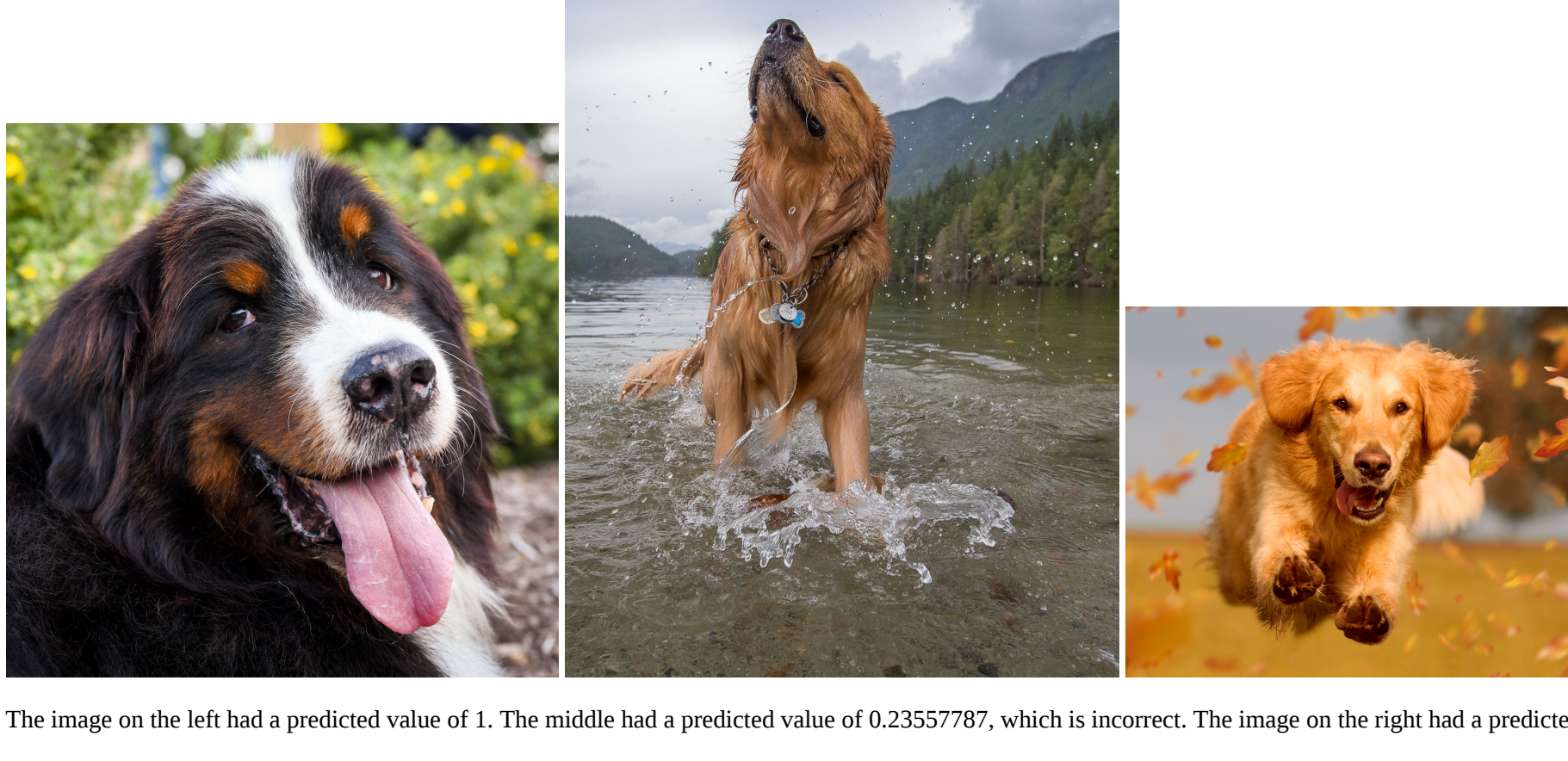
Another issue that was encountered here was the confidence of the predictions. Since the answer to each input is either a 0 or 1 the model does not learn to predict the probability of an image being part of a certain class. Instead it learns to have answer that is always very close to either 0 or 1.

7 Examples

A predicted value close to 0 corresponds with cat and a predicted value close to 1 corresponds with dog. These images are were found on Google Images and were not used to train the model.



The image on the left had a predicted value of 0.0625807. The middle had a predicted value of 0.00000000000014656887. The image on the right had a predicted value of 0.996319, which is incorrect.



The image on the left had a predicted value of 1. The middle had a predicted value of 0.23557787, which is incorrect. The image on the right had a predicted value of 0.9542546.

References

[1] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. Part of *Neural Information Processing Systems 25*, NIPS 2012.

[2] Diederik P. Kingma, Jimmy Lei Ba. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. In *International Conference on Learning Representations*, 2015.

[3] Rumelhart, D., Hinton, G. & Williams, R. Learning representations by back-propagating errors. *Nature* 323, 533–536, 1986.

[4] Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.*arXiv preprint arXiv:*1502.03167, 2015.

Implementation: https://github.com/AzerAfram/catOrDog

Thanks to Alex Renotte for the baseline as well as Serkan Peldek for the model architecture.