Artificial Neural Networks for Sea Turtle Identification

Kane Kesler

Tuesday 9th May, 2023

Abstract

Investigating the effects of tourism and social media on wildlife animals can be done by identifying the animals on social media platforms and tracking the frequency at which they are posted (Papafitsoros, Adam, and Schofield 2023). For this, we need to employ network-based classification methods on a large dataset spanning over many years, in this case, the SeaTurtleIDHeads dataset. Using this dataset, we test and train a convolutional neural network (CNN) to identify the turtles based on the pattern segments on their heads. We review the fundamentals of deep learning required to build the model, starting from a simple model to a model that can successfully label images ($> 80\%$ accuracy). While we build the neural network, using PyTorch, we will use the established CIFAR-10 dataset to compare performance with models of a different dataset.

Contents

1 Artificial Neural Networks Fundamentals **3**
1.1 Introduction - The Perceptron Neuron 3
1.2 The Sigmoid Function 5
1.3 Gradient Descent Optimiser 7
1.4 Generalising Neural Networks 8
1.5 Backpropagation 9

2 Convolutional Neural Networks (CNNs) **10**
2.1 Convolutional layers 12
2.2 Pooling Layers 13
2.3 Overfitting 14
2.3.1 Hold-Out Method 14
2.3.2 Data Transformations 15
2.3.3 Dropout 15
2.3.4 Batch Normalisation 15
2.4 Network Architecture 15

3 Applications **16**
3.1 CIFAR-10 Dataset 16
3.1.1 CIFAR-10 Results 16
3.2 SeaTutleIDHeads Dataset 19
3.2.1 Random split vs Time-cutoff split 19
3.2.2 SeaTutleIDHeads Results 20

4 Conclusion **22**

Bibliography **22**

1 Artificial Neural Networks Fundamentals

1.1 Introduction - The Perceptron Neuron

Artificial Neural Networks (ANNs) are based on the biological phenomenon of neurons activating in the brain. Just like in the brain, the artificial neurons only activate once the input is large enough, i.e., passes a given threshold. Consider an ANN that takes in a vector of n binary components and produces an output.

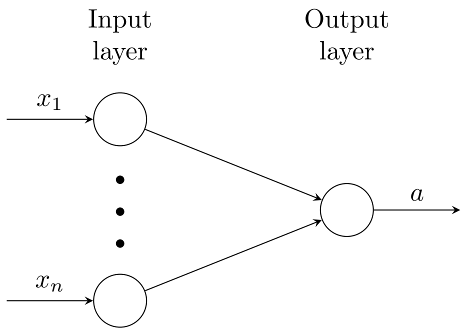


Figure 1: A network where the input is passed through the first layer, before being collected as a weighted sum in the neuron in the output layer. The overall output of the network is the output of the final node, labelled as a.

This is an example of a perceptron network, with input vector $x = (x_1, \ldots, x_n)^T$, each of which has different levels of contribution to the output neuron represented as the weight vector $w = (w_1, \ldots, w_n)$. Throughout this dissertation, the components of the input vector x are referred to as *features* or *feature variables.* We define the output or *activation* of the neuron as,

$$a(x) = \text{sign}(z) = \begin{cases} 0, & wx \leq \text{ threshold;} \\ 1, & wx > \text{ threshold,} \end{cases}$$

where the weighted input $z(x) = wx$ is called the *preactivation.*

Simply put, the neuron only activates once the weighted sum of the inputs passes a threshold. It is useful to think of the neurons after the input layer as computational nodes where each node collects an input and applies a function to it before producing an output. For example, while the i^{th} node in the input layer only passes the information along to the next layer, the subsequent layers will have nodes receiving the sum of these quantities wx as an input and then applying the sign function before producing an output. We can make this simpler by introducing the bias $b =$ -threshold as a property of the neuron and letting $z = wx + b$. Doing so allows us to rewrite the output as

$$a = \begin{cases} 0, & z \leq 0; \\ 1, & z > 0, \end{cases}$$

The bias can be interpreted as the propensity for the neuron to be activated, a higher bias indicates the neurons requires a greater input to activate. When discussing further topics, for simplicity, we will set the bias vector as $b = 0$, unless stated otherwise. To see how a perceptron can learn, consider a perceptron with an input of three features $x = (0, 0, 1)^T$ assigned to label $y = 1$. We have to find weights such that, our output correctly assigns the label 1 to the input, in other words, find w such that, $wx + b > 0$. We may start with an initial vector $w = (1, 0, 0)$ but quickly realise $wx = 1 \cdot 0 + 0 \cdot 0 + 0 \cdot 1 = 0$ hence $a = 0$. The same result happens with $w = (0, 1, 0)$ until we reach the parameter vector $w = (0, 0, 1)$, which is an appropriate vector thus, the perceptron has learned to classify the input.

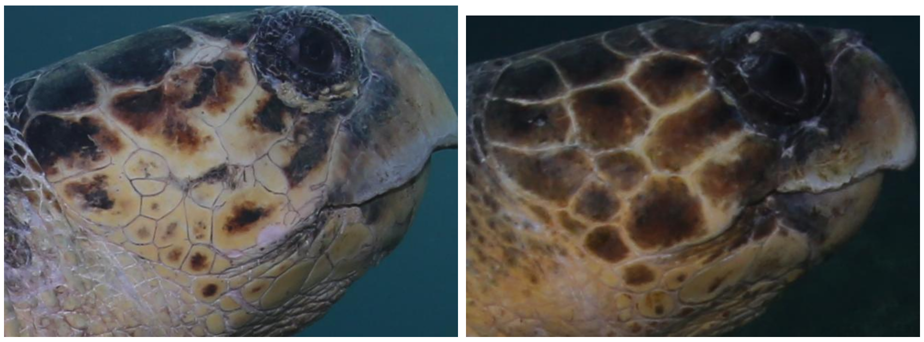


Figure 2: Every turtle has unique segmented patterns on the sides of their heads, as demonstrated by the turtles shown. This image was taken from the SeaTurtleIDHeads dataset.

For more complicated applications, such as classifying sea turtles based on head segments (see Figure 2), thousands of pixels count as input features. Thus, a lot of weights will also need to be configured for correct classification. Two problems arise from using a perceptron here. The first problem is that images are represented by floats beyond binary inputs so, most of those images are not valid inputs for a perceptron neuron. The second problem is that even if we allowed floats beyond binary inputs, changing some of the weights can potentially completely flip the prediction. If we consider the previously mentioned example, another valid vector would be $w = (0, 0, 0.001)$ but only a small change in the third weight is needed to completely flip our prediction from 1 to 0. This makes it unclear as to how we should change the parameters. A solution to this problem is to replace the rigid sign activation function with a smoothed-out version, in particular the sigmoid function.

1.2 The Sigmoid Function

The sigmoid function $\sigma : \mathbb{R} \longrightarrow (0,1)$ is defined as

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

and can be seen as the smoothed-out version of the sign function (see Figure 3).

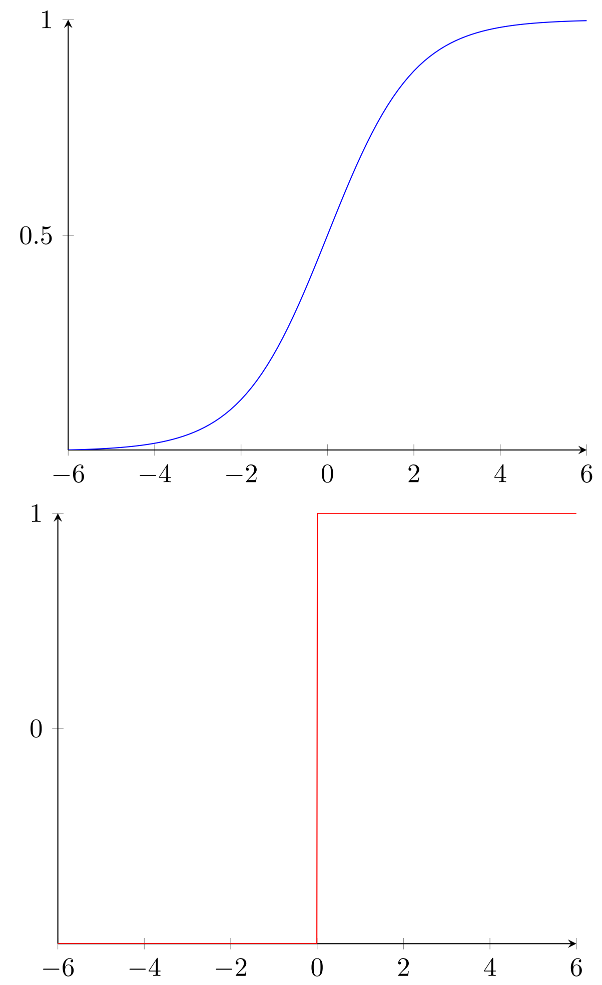


Figure 3: The sigmoid function (Top) and the sign function (Bottom).

Note that the sigmoid function is a continuous function taking any real input and producing any real output between 0 and 1. Another useful property is that the activation function $\sigma(z)$ is an increasing function where

$$\sigma(z) = \sigma\left(w_1 x_1 + \cdots + w_n x_n\right),$$

so it makes it easier to see how increasing the slight weights will only slightly increase the neuron's activation. This means that, with the aid of its continuity, we have a gradual change, rather than the complete flip in activation we saw with the sign function. Note that this also applies to non-zero bias(es) as well. We vectorise $\sigma(\cdot)$ to take inputs of higher dimensions by defining $\sigma : \mathbb{R}^m \longrightarrow \mathbb{R}^m$, such that

$$(\sigma(z))_i = \sigma(z_i)$$

for some vector $z \in \mathbb{R}^m$.

Recall the network portrayed in Figure 1. Suppose we are given N training points $\left(x^{\{i\}}\right)_{i=1}^{N}$, with $x^{\{i\}} \in \mathbb{R}^n$, and corresponding labels $\left(y\left(x^{\{i\}}\right)\right)_{i=1}^{N}$, where $y \in (0,1)$. We define our error of prediction, or *cost*, as the mean squared error

$$\mathrm{Cost}_{\mathrm{MSE}}(w,b) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left\|y\left(x^{\{i\}}\right) - a\left(x^{\{i\}}\right)\right\|_2^2 \dagger$$ [1]

Our goal is to build a network whose outputs accurately predict the label for any given input. In other words, we must find the parameters that minimise the squared error as defined above. For our simple example (see Figure 4), the parameters consist of a single weight w. The general idea is that we tune the parameters according to how much they affect the cost function. Intuitively, the more influential (larger in absolute value) parameters would affect the cost function more, so tweaking those parameters potentially decreases the cost significantly quicker than less influential neurons. We can demonstrate this with a simple example, let $(x,y) = (3, 0.5)$ be our input and label respectively. Consider a network with the input of one feature and one output (below).

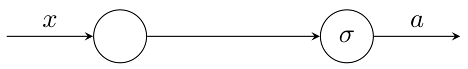


Figure 4: A network with an input of one feature, the node in the output layer applies an activation function and produces output a.

We aim to choose a weight w such that the network accurately predicts the label as 0.5, i.e., minimising the difference $y - a$. Setting $w = 0$ minimises the cost function as $y - a = 0.5 - \sigma(0 * 3) = 0.5 - 0.5 = 0$. Instead, if we set $w = 0.8$ we would have to tweak it slightly to minimise the cost but if we had set $w = 100$, we would need to drastically tweak the weight to minimise the cost. For larger networks iterating over thousands of different inputs, manually

[1]Note that in this formula, the quantity $a\left(x^{\{i\}}\right)$ denotes the total output of the network however, in the simple case of an n-dimensional input and one output neuron, the neuron's output is equal to the total activation. This is not necessarily true when we generalise the network, as we will see in later sections.

tweaking the parameters is impractical. Luckily there is an optimisation process called *gradient descent*.

1.3 Gradient Descent Optimiser

Recall the formula of the cost function of a neural network summing over N training points with an arbitrary number of features with corresponding labels. For simplicity, we will replace the arguments of the cost function with p so we have

$$\text{Cost}_{\text{MSE}}(p) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left\|y\left(x^{\{i\}}\right) - a\left(x^{\{i\}};p\right)\right\|_2^2,$$

where p is the vector of the weights and biases.

We iteratively update the parameters p such that it coincides with decreasing the cost function until we reach some minimum. To speed up the process we choose the direction of steepest descent $-\nabla Cost_{MSE}(p)$, where $\nabla Cost_{MSE}(p)$ is the gradient and points in the direction of steepest increase. Once we find the gradient, using the chain rule, we then do the following update

$$p \longrightarrow p - \eta\nabla Cost_{MSE}(p),$$

where η is called the *learning rate*.

This update coincides with our intuition in the last section, the smaller in absolute value, i.e., the less influential the weights and biases in parameter vector p, the smaller the cost, meaning we only need to make subtle updates to the parameters. Ideally, every update to the argument p will cause the cost function to decrease to the cost function as quickly as possible (with the help of the steepest decreasing gradient) until we reach some minimum. This process is called *gradient descent* and is defined as an *optimizer*, a process that optimizes the weights and biases to minimize the cost function. For larger data sets, we use a less computationally heavy variation of this process called *mini-batch stochastic gradient descent*. We define the cost function of an individual training point $x^{\{i\}}$ as

$$C_{x^{\{i\}}}(p) = \frac{1}{2}\left\|y\left(x^{\{i\}}\right) - a\left(x^{\{i\}};p\right)\right\|_2^2.$$

Thus, the overall cost function is the sum of the cost of each training point. The gradient of the cost function is now

$$\nabla\,\text{Cost}_{MSE}(p) = \frac{1}{N}\sum_{i=1}^{N}\nabla C_{x^{\{i\}}}(p).$$

For mini-batch stochastic gradient descent, we randomly split the N training points into K batches of r training points. Then we take the mean cost function of each batch and use that to update our parameters. Thus for a batch of r integers, we choose $k_1, k_2, \dots, k_r$ from the set $\{1, 2, \dots, N\}$ and perform the updates

$$p \longrightarrow p - \eta \frac{1}{r} \sum_{i=1}^{r} \nabla C_{x^{\{k_i\}}}.$$

We repeat this process K times to get our optimised weights.

We now have a simple updating rule that we can use to tune the parameters but our goal is to build a network to classify sea turtle images, which will require multiple hidden layers between the input layer and output layer. This makes calculating the gradient not so obvious, fortunately, there is an algorithm that does just that called *backpropagation*, which we will explore later in the paper. But before we introduce this algorithm, we first need to generalise the networks and concepts we have been working with so far and introduce new notation.

1.4 Generalising Neural Networks

The notation we will be using is adapted from (C. F. Higham and D. J. Higham 2018, p. 7). For a network of L layers, we have each layer l for $l = 1, \dots, L$ containing n_l neurons. Bear in mind that neurons in the input layer ($l = 1$) only pass on the input to neurons in the next layer and so do not possess weights or biases. With that said we introduce the weight matrix at layer l, $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$, where each entry $w^{[l]}_{jk}$ represents the weight of the connection between the neuron j and k in layers l and $l-1$, respectively. Finally, we have our bias vector $b^{[l]} \in \mathbb{R}^{n_l}$ at layer l, where $b^{[l]}_j$ represents the bias of neuron j. Using the notation we introduced so far, we can effectively treat the network as a function from $\mathbb{R}^{n_1}$ to $\mathbb{R}^{n_L}$ where the input is fed forward into the following process

$$\begin{aligned} a^{[1]} &= x \in \mathbb{R}^{n_1}, \\ z^{[l]} &= W^{[l]} a^{[l-1]} + b^{[l]}, \\ a^{[l]} &= \Phi(z^{[l]}) \in \mathbb{R}^{n_l}, \end{aligned}$$

We do this process for $l = 2, \dots L$, where $\Phi(\cdot)$ is an activation function to be chosen.

This process defines a *feed forward network* (FFN), where input is fed from left to right. We can reproduce Figure 1 by choosing to configure a network with $L = 2$ layers, with $n_1 = n$ and $n_2 = 1$. From this, we see that our input vector is $x \in \mathbb{R}^n$ and the weight matrix for layer 2 is $W^{[2]} \in \mathbb{R}^{1 \times n}$.

1.5 Backpropagation

To put it simply, backpropagation allows us to see how the small changes in the parameters affect the cost of the network. We do this by recursively using the chain rule to get the gradients $\partial C/\partial w_{jk}^{[l]}$ and $\partial C/\partial b_j^{[l]}$, where $w_{jk}^{[l]}$ and $b_j^{[l]}$ are defined as in the last section. First, we introduce the *error signal* $\delta_j^{[l]} \in \mathbb{R}^{n_l}$ defined as

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}}, \text{ for } 1 \leq j \leq n_l \text{ and } 2 \leq l \leq L,$$

where $C = C_{x^{\{i\}}}$ is the cost of an individual training point.

This quantity measures the rate at which the cost function changes for small changes in the input of the j-th neuron in the l-th layer. With this, we can see with a small example how backpropagation works.

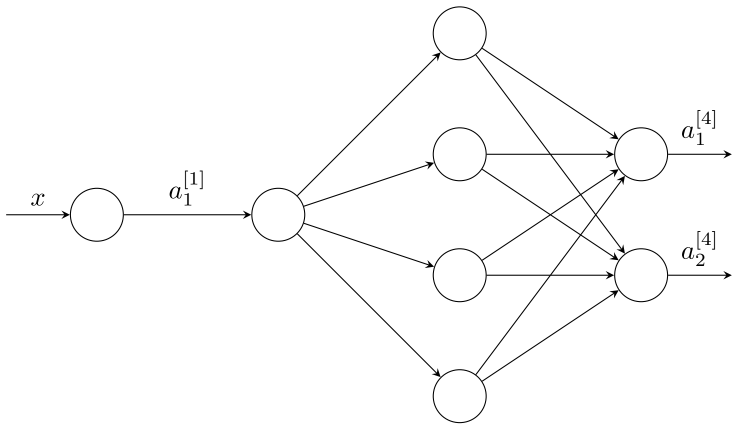


Figure 5: A small network with an input of one feature and two-dimensional output. The activations of the hidden layers and layer labels have been omitted for simplicity.

Our example consists of one training point with one feature $x \in \mathbb{R}^1$ and contains the following weight matrices $W^{[2]} \in \mathbb{R}^{1\times 1}$, $W^{[3]} \in \mathbb{R}^{4\times 1}$ and $W^{[4]} \in \mathbb{R}^{2\times 4}$. In our example, we have already propagated the input through our network with output $a^{[4]}$ and so, our cost of this training point is $C = \frac{1}{2}(y - a^{[4]})^2$. To update the weights we need to find the gradient $\nabla_w C$, which means computing the gradient $\partial C/\partial w_{jk}^{[l]}$ for all j and k in the network. This can be solved using the backpropagation equations, see (C. F. Higham and D. J. Higham 2018, pp. 12–14) for derivation,

$$\begin{aligned}
\delta^{[L]} &= \Phi'\left(z^{[L]}\right) \circ \left(a^{[L]} - y\right), \\
\delta^{[l]} &= \Phi'\left(z^{[l]}\right) \circ \left(W^{[l+1]}\right)^T \delta^{[l+1]}, && \text{for } 2 \le l \le L-1, \\
\frac{\partial C}{\partial b_j^{[l]}} &= \delta_j^{[l]}, && \text{for } 2 \le l \le L, \\
\frac{\partial C}{\partial w_{jk}^{[l]}} &= \delta_j^{[l]} a_k^{[l-1]}, && \text{for } 2 \le l \le L.
\end{aligned}$$

The operator $\circ : \mathbb{R}^m \times \mathbb{R}^m \longrightarrow \mathbb{R}^m$, is the Hadamard product defined as the component-wise multiplication of two vectors, which is useful as it can be used to simplify the equations produced from the chain-rule into in matrix form.

So our process is as follows: we fed forward training points into the network and then, after computing the output, we use backpropagation to compute the gradient of our overall cost function and use that to update our parameters with mini-stochastic gradient descent. The process of forward feeding and backpropagation once is called an *epoch*. Ideally, after a set number of epochs, our cost function is minimised, i.e. the parameters become optimised.

One significant drawback is that pixel values in non-random images are correlated; e.g., a red pixel is more likely to be surrounded by others. This correlation acts as a way to determine the pixel's location. So if we just fed each pixel value to each node in an FFN, we lose this spatial information, as there is no inherent order in which we provide the pixel values nodes in the input layer. This problem is solved using a spatial-orientated network called a *convolutional neural network.*

2 Convolutional Neural Networks (CNNs)

The general idea of CNNs is that instead of taking each pixel as a singular input for each input node, each input node will take patches of the input image. Each progressive layer will analyse larger patches from the input image until we reach the output layer. These networks learn in a way similar to FFNs, both use backpropagation and an optimiser to train the network parameters and pass on information to the next layer using an activation function. Once they have finished learning the training data of images, we can see how the networks classify unseen images. Before we mention how these networks preserve spatial information, we need to state the differences in the optimiser and cost functions commonly used for CNNs. First, we mention that we use the exponential linear unit (ELU) activation function

$$\mathrm{ELU}(z) = \begin{cases} z, & z > 0, \\ \alpha(e^z - 1), & z \leq 0, \end{cases} \qquad \alpha > 0,$$

in place of the sigmoid function. This is because if a neuron is *saturated*, i.e., the activation $\sigma(z)$ of a neuron saturated is close to either 0 or 1, the gradient of the activation $\sigma'(z)$ is closer to zero. Thus the weights get updated by small amounts, causing it to learn too slowly. The ELU function fixes this by increasing the range the activations can take to decrease the chance of saturation and hence is quicker than the sigmoid function for large inputs, in particular, images (Clevert et al. 2015, pp. 9–10).

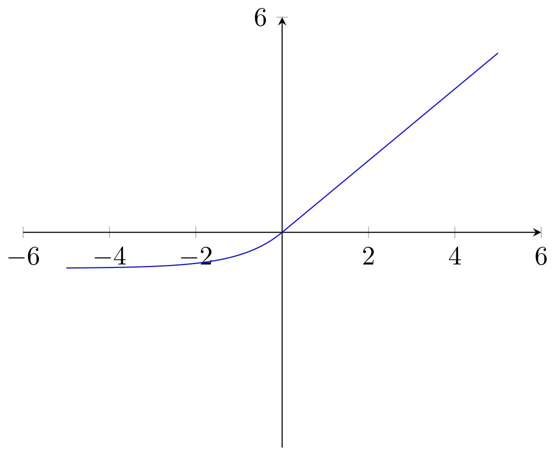


Figure 6: ELU plot with $\alpha = 1$. Unlike the sigmoid function, there is no exponential decay as $z > 0$ increases, which increases the range of values the activation can take.

For the last layer of the network, however, we use the *softmax* activation which assigns the s-th component of the activation of the last layer $a^{[L]}$ as

$$a_s^{[L]} = \frac{e^{z_s^{[L]}}}{\sum_k^C e^{z_k^{[L]}}},$$

where the arguments have been emitted for simplicity. The normalisation constant $\sum_k e^{z_k^{[L]}}$ effectively gives us a probability of how likely the category the input data belongs in the s-th category out of C categories, corresponding to the s-th component of the vector $a^{[L]}$.

In place of the mean-squared error cost, we use the cross-entropy cost function, see (Nielsen 2019, pp. 68–70) for more detail,

$$\text{Cost}_{cross-entropy} = -\sum_{s=1}^{C} y_s \log a_s^{[L]}.$$

It is common practice to use the cross-entropy function alongside the softmax activation function and so we will be using this function.

For our optimiser, we use the ADAM algorithm (Kingma and Ba 2014), a variant of the gradient descent that is commonly used for models with large datasets, e.g., a dataset size of 6,000 32x32 images. Now we can explain how CNNs efficiently preserve spatial information by introducing *convolutional layers*.

2.1 Convolutional layers

Convolutional layers are used to transfer spatial information of regions of the input image onto the next layer and achieve this using *kernels*. Kernels are used for extracting specific features of an image, as well as its location. For example, if you want to detect horizontal edges in the image, the kernel could extract the spatial information of that feature and pass that information on to the neurons in the next layer. This is not limited to one feature, we could use as many kernels to detect many independent features as we deem necessary. Mathematically, kernels are small matrices of weights and are applied to the regions of pixel values in the input image. They extract patches of information from the previous layer and store that information as a single value for each neuron in the next layer. Each neuron in the subsequent layer contains information about the different regions of the image in the previous layer and will be trained to analyse those patches. In this way, the position of each pixel relative to other pixels is accounted for, thereby preserving spatial information. This is achieved using the 2D convolutional operation, as illustrated in Figure 7. For the mathematical formulation of the 2D convolutional operation see (Aggarwal 2018, pp. 320–321)

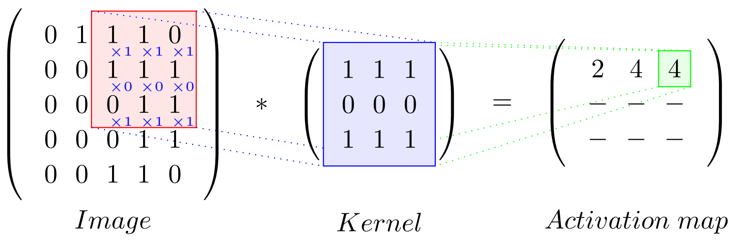


Figure 7: A 3x3 convolutional kernel applied on the local region (the region in red) of a 5x5 image before "sliding" one unit to the left or down, until all the entries in the activation map are filled.

The result is called an activation map, where each entry can be thought of as the activation of each neuron in the next layer. The benefit of this is that the entries in the kernel matrix act as weights on the input values of the image so, instead of having a unique weight applied to each input we can use the same matrix of weights across multiple regions of the image. This reduces the number of weights in the network needed to be trained, making it much more computationally effective when applied to many images containing a lot of pixel information.

The downside is that we have lost some information about the image boundaries after each convolution (notice how the size of the image decreases after the kernel is applied to it in Figure 7). This is solved with *half-padding*, (Aggarwal 2018, pp. 322–323), so for any image, we add pixels of value 0 around the border of the image, such that half of the kernel "sticks out". This has the effect of maintaining the image size, so no information is lost during the convolution process.

2.2 Pooling Layers

In order to maintain computational efficiency, however, we will need to reduce the size of the image throughout each convolutional layer without losing information about the image boundaries. This is the purpose of pooling layers, layers that can downsize an image while still retaining the general structure of the image it was downsized from.

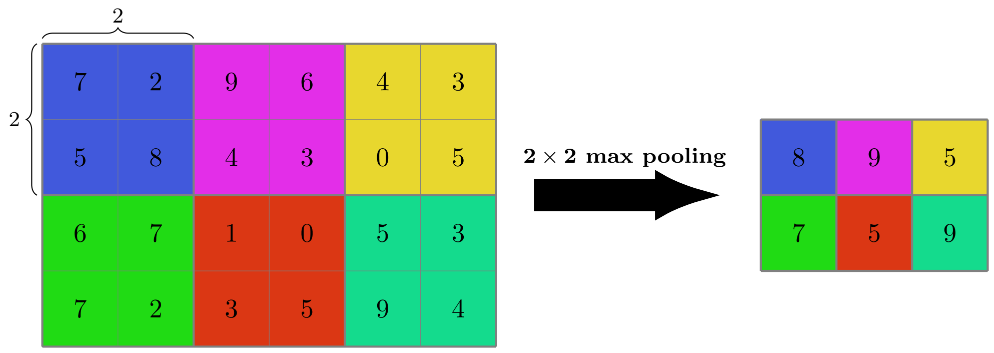


Figure 8: Max pooling using a 2 × 2 max-pool filter. We take a 2 × 2 grid and place it over a region of the input matrix, where we sample the largest value in the 2 × 2 grid. Then, we slide the grid two steps to the left and apply it until all regions have been sampled.

In particular, we will use a 2 × 2 max-pool filter (Figure 8). As a result, we get a downsized representation of the input image, such that it does not remove any border information while also making the network more computationally efficient.

2.3 Overfitting

Overfitting is when the model can identify features of the training images and classify them well but cannot generalise its classification performance to unseen images (test images). This occurs when the neuron's weights are over-trained on the random noise of the training images, making it perform worse on unseen data compared to the known data in terms of accuracy. The techniques used in our CNN model are to combat this issue.

2.3.1 Hold-Out Method

This method consists of splitting the dataset into the train, validation and test datasets. The training data is used to tune our network parameters via the method described. For every epoch, we alternate between using the tuning parameters with the training data and then gauging its generalising power with the validation set. The way we gauge generalising power is to plot the average loss of the two models against epochs.

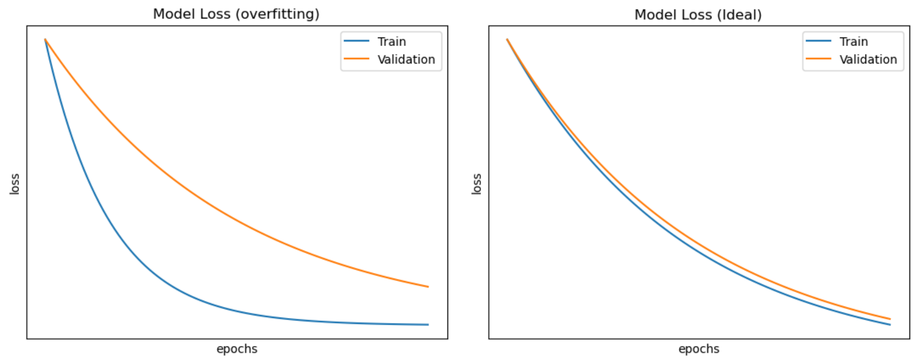


Figure 9: Dummy loss against epochs plots.

When a model is overfitting, the validation loss will be higher than the training loss, as shown in Figure 9 (left), meaning the generalising power of the model is low. Ideally, we want the two losses to be similar, as shown in Figure 9 (right)†[2], which indicates that the model performs well on the training and validation data. In some instances, the training and validation set can be similar, the datasets may consist of images taken at roughly the same time. After we finished the training process, we use the testing set to get the true performance of the model.

For our model, we randomly split the training, validation and test datasets of the ratio 80:10:10, a commonly recommended split.

[2]It is possible that the validation loss is greater than the training loss. It is still ideal to have the losses be similar, regardless.

2.3.2 Data Transformations

It may also be important to diversify your training images if you have a limited amount. This is done with image transformations such as vertical/horizontal flips and rotation by some angle. Applying such transformations allows your model to analyse the subjects of the image in different environments, for example, rotating an image of a car on flat ground may help the model classify an image of a car on an incline. This increases the generalising power of the model, mitigating overfitting.

2.3.3 Dropout

In the network, neurons will co-adapt with each other, e.g., a neuron with a small weight may learn very slowly and so another neuron will increase its weight to compensate, which in turn causes overfitting. Dropout resolves this by, randomly and independently, removing neurons with probability p for each epoch. This means each epoch will have a different configuration of neurons removed. By doing this, neurons can not consistently adapt with other neurons, as any neuron including itself could be removed and so decreases overfitting in the model (Nielsen 2019, pp. 88–90).

2.3.4 Batch Normalisation

The purpose of batch normalisation is to normalise the activations passed through each layer of the network such that all of the activations have the same distribution for all input images in every batch. It has been shown through experiments that, it improves generalising power (Ioffe and Szegedy 2015, p. 5).

2.4 Network Architecture

For the experiments, we used the following architecture

$$(2^nC3 - 2^nBN - MP2)_{n=4,5,6} - (250mFC - 250mBN - D)_{m=1,2} - 10FC,$$

with a dropout rate of $p = 0.25$ and a learning rate decay of $\eta = 0.001$. Starting with the first block, 2^nC3 is a convolutional layer with 2^n kernels each of size 3×3; 2^nBN is the corresponding batch normalisation of each kernel and $MP2$ is the max-pooling layer of size 2×2. The subscript $n = 4, 5, 6$ indicates the three iterations of the first block kernel size growing by a factor of 4. In the second block, we have $250mFC - 250mBN - D$ representing a fully connected layer of $250m$ neurons followed by batch normalisation and dropout. The subscript notation is analogous to the first block. Finally, we reach the last layer of size 10, representing each label in the dataset. For the following experiments, we will the model for 50 epochs.

3 Applications

3.1 CIFAR-10 Dataset

The CIFAR-10 dataset consists of a total of 6,000 32×32 images such that they can be classified into 10 categories: dog, cat, deer, frog, horse, bird, plane, car, truck and ship. Since this is a standard dataset used to practice different CNN models, we build our model using this dataset first to gauge performance.

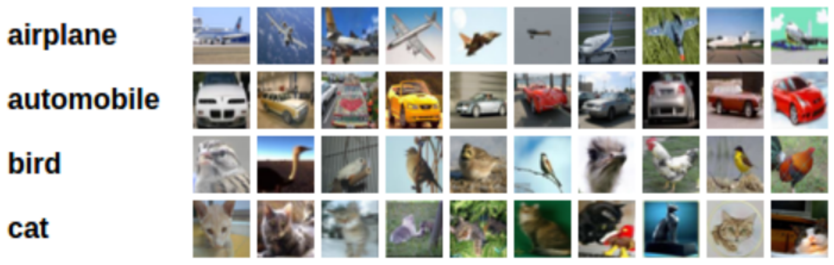


Figure 10: Samples of CIFAR-10 dataset, taken from the Pytorch tutorial: Training a Classifier.

3.1.1 CIFAR-10 Results

Using the architecture on the CIFAR-10, we observed an 82.47% accuracy rate, where accuracy is defined as the number of correct predictions out of the total number of predictions made by the model. A plot of the loss against 50 epochs is shown below

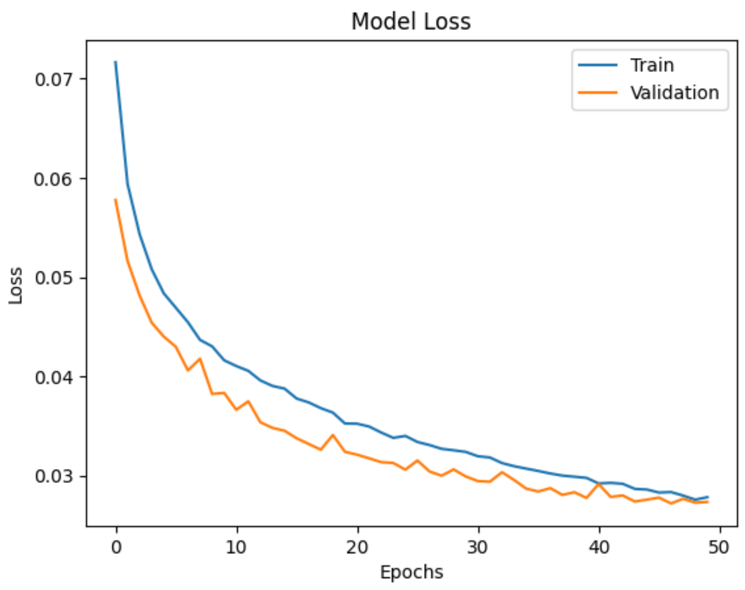
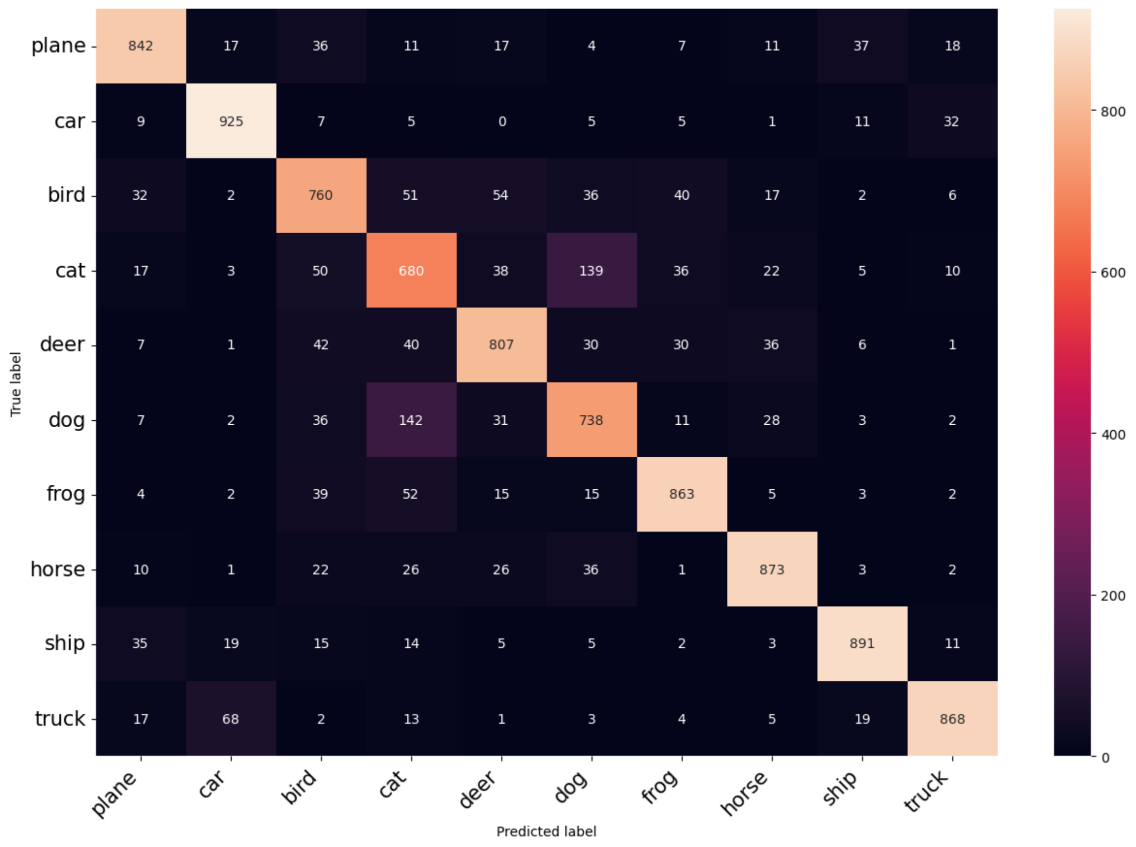


Figure 11: Plot of average loss against 50 epochs using the CIFAR-10 dataset.

Our previous discussion about the loss against epoch plots shows that the model does fairly well in both the training and validation data. This means that the model was able to generalise to unseen data very well, which coincides with the accuracy rate we got for the testing dataset.

Another interesting observation can be seen in its confusion matrix (Figure 12).



True label \ Predicted label	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	842	17	36	11	17	4	7	11	37	18
car	9	925	7	5	0	5	5	1	11	32
bird	32	2	760	51	54	36	40	17	2	6
cat	17	3	50	680	38	139	36	22	5	10
deer	7	1	42	40	807	30	30	36	6	1
dog	7	2	36	142	31	738	11	28	3	2
frog	4	2	39	52	15	15	863	5	3	2
horse	10	1	22	26	26	36	1	873	3	2
ship	35	19	15	14	5	5	2	3	891	11
truck	17	68	2	13	1	3	4	5	19	868

Figure 12: The confusion matrix of the CIFAR-10 model, where each entry shows the number of predictions per label. The diagonal and non-diagonal entries show the number of correct and incorrect predictions, respectively. Summing the entries in each row gives you the number of images in the testing dataset per label.

We see that the number of correct predictions for the images of cats and dogs is significantly lower than the rest of the group, 680 and 738 respectively. In addition, the images of cats and dogs were also mistaken for one another by the model quite frequently, with 142 cat images predicted as dog images and 139 dog images predicted as cat images. This could be explained by the fact that dogs and cats share the most similarities with each other out of all of the labels whereas planes, boats, ships and trucks have a more distinguishable shape compared to the other labels. In turn, this could make it harder for the model to identify specific features that would distinguish a cat from a dog hence, producing the results shown in the confusion matrix.

This is not limited to cats and dogs but other animal-based labels as well, with the confusion matrix showing that the animals in the dataset have a lower number of correct predictions on average when compared to the vehicle-based images. Knowing this, we can expect that the model will have a lower accuracy rate on the sea turtle data since the model will need to identify even more specific features to distinguish similar-looking sea turtles.

3.2 SeaTutleIDHeads Dataset

From the dataset, we take a subset of data of size 10 corresponding to the turtles with the most images. We also reduce the size of the images to 32×32 to reduce the computational burden.

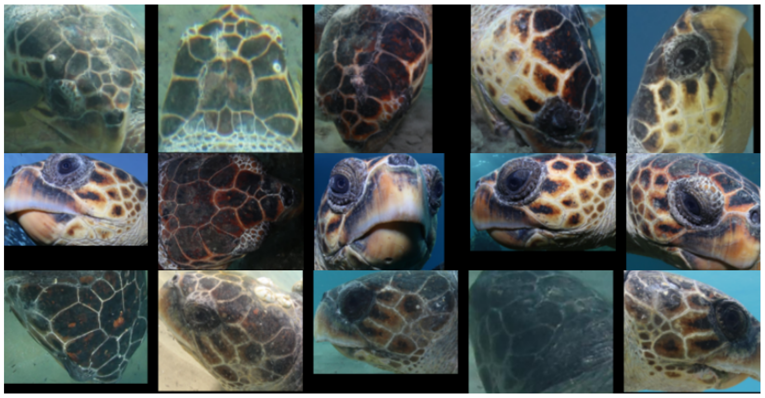


Figure 13: Samples of SeaTurtleIDHeads dataset with turtles t025, t243, t094 as the first, second and third rows respectively. These images were taken from the Kaggle page containing the SeaTureleIDHeads images.

3.2.1 Random split vs Time-cutoff split

For the previous section, we split the dataset randomly, however, there are three ways to split the SeaTurtleIDHeads dataset (Papafitsoros, Adam, Čermák, et al. 2022, p. 7). The two of interest are random split and time-cutoff split. The random split is performed just as before whereas, the time-cutoff split splits the training and testing data based on some date and time. This split is desirable for designing a model that can recognise future instances of familiar sea turtles. The date and time used are 2020-01-01 00:00:00, forming a 60:20:20 split of training, validation and testing data respectively. We use this same ratio for our random split model for the sake of continuity. Before we train the model using this data, we can hypothesise that the time-cutoff model will perform worse than the random model. A possible issue could be due to the discrepancies in the quality of the dataset images (see Figure 14).

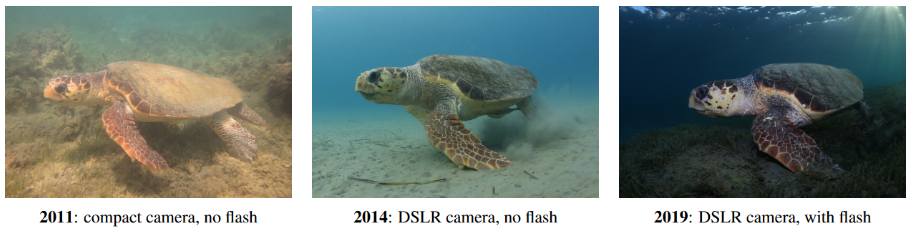

2011: compact camera, no flash
2014: DSLR camera, no flash
2019: DSLR camera, with flash

Figure 14: Image of the same sea turtle at different times. These images were taken from the 'SeaTurtleID: A novel long-span dataset highlighting the importance of timestamps in wildlife re-identification' paper.

The use of different cameras and filters, for example, the fish-eye lens filter in the third image in Figure 14, can distort the features of the turtles which could make it harder for the model to classify the same sea turtle from the data it was trained on. This could effect could be mitigated by using a consistent camera with the same lens to avoid image distortion.

3.2.2 SeaTutleIDHeads Results

From the results, we observed accuracies from the random split and time-cutoff split of 62.15% and 33.89% respectively. We observe that both the random split and time-cutoff split models performed significantly worse when compared to the random split model using the CIFAR-10 dataset, as reflected in Figure 15.

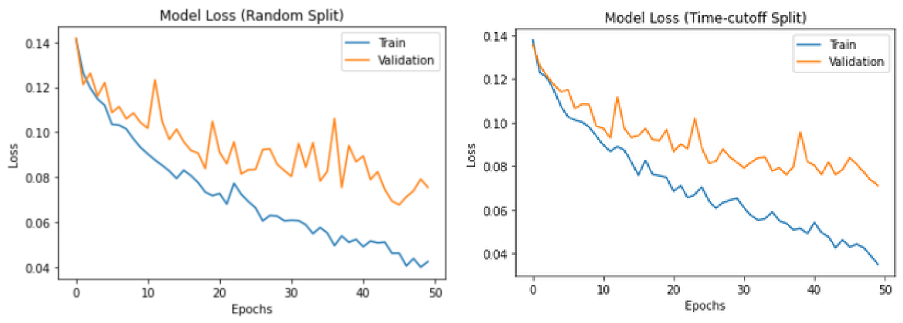


Figure 15: Two plots of loss against 50 epochs, using the SeaTurtleIDHeads dataset.

We see that, unlike the CIFAR-10 loss plot, there is a greater separation between the validation and training losses, signifying a worse classification performance when using the SeaTurtleIDHeads data. There are several potential reasons for this, first, we only had 851 images as opposed to the 6,000 images to work with the CIFAR-10 dataset. Such limited data limits the model's capacity to generalise to unseen data, e.g., test and validation data. The consequences of the limited dataset are demonstrated in the following confusion matrices (Figure 16).

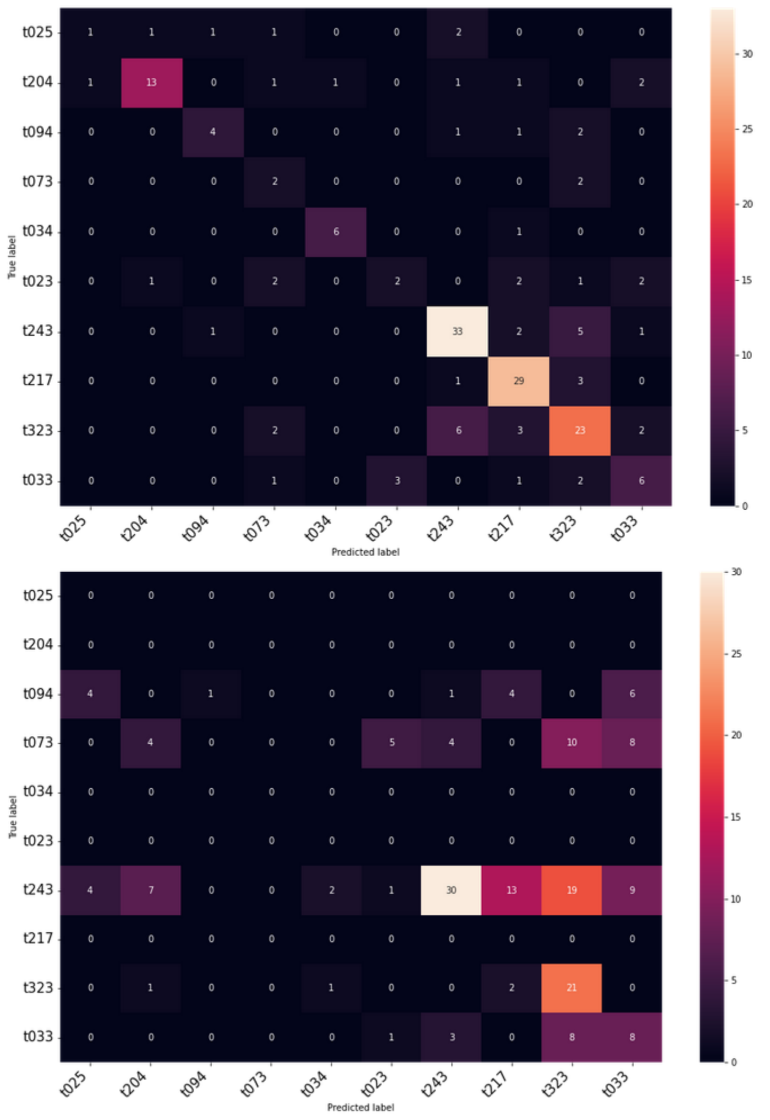


Figure 16: Two confusion matrices, one from using the random split training data (Top) and the other with the time-cutoff split (Bottom).

From both the random and time-cutoff split confusion matrices, we see a positive correlation between the number of images of a turtle in the dataset and the number of correct predictions. For example, in the random split confusion matrix in Figure 16 (Top), we that turtle t243 has a total of 42 images with 33 of them being correctly predicted, in contrast, turtle t072 has 2 correct predictions out of a total of 4 images. In other words, only half of the images of t072 were correctly predicted. In addition, when we perform the time-cutoff split, the limited amount of turtle images means that not all of the turtles are present in the resulting testing set. This is illustrated in 16 (Bottom) where the row-sum of, for example, turtle t025 is zero and therefore means there are no images of t025 present. Despite this, both turtles t243 and t094 had images wrongly predicted as t025. This can be observed with other turtles present in the training set but missing in the testing set. These false predictions lower the accuracy rate, as the model does not account for the missing data and thus performs worse than both the random split model and the CIFAR-10 model. Both of these issues could be resolved by increasing the number of photos taken of each turtle.

Another reason is that the model is trying to perform fine-grained image classification, i.e., classifying a species of animals by niche features, the segmented patterns on the sides of their heads, instead of comparing drastically different objects such as cats and planes in the CIFAR-10 dataset. Fine-grained problems are known to be more difficult due to the model needing to distinguish between similar objects on more subtle features. We could potentially improve the performance by resizing the images from 32×32 to a larger resolution, as it would add more detail for the model to work with.

4 Conclusion

Using the foundations of deep learning, we constructed a CNN model that could achieve a high accuracy rate on the CIFAR-10 dataset but performed worse on the SeaTurtleIDHeads dataset. From here, we encountered the fine-grading problem of intra-species classification based on subtle features, compared to the significant variation of the CIFAR-10 data. The main solutions proposed from the experiments suggest increasing the number of pictures of each sea turtle with more consistent quality. It also suggests a different method for prepossessing the sea turtle data, such as changing the size from 32×32 to something larger, which could increase model accuracy.

Bibliography

Aggarwal, Charu C. (2018). *Neural Networks and Deep Learning. A Textbook.* Springer Cham. ISBN: 978-3-319-94463-0. DOI: https://doi.org/10.1007/978-3-319-94463-0.

Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter (2015). *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).* arXiv: 1511.07289.

Higham, Catherine F. and Desmond J. Higham (2018). *Deep Learning: An Introduction for Applied Mathematicians.* arXiv: 1801.05894.

Ioffe, Sergey and Christian Szegedy (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv: 1502.03167.

Kingma, Diederik P. and Jimmy Ba (2014). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980.

Nielsen, Michael (2019). *Neural Networks and Deep Learning*. last visited 05/05/23. URL: http://neuralnetworksanddeeplearning.com/index.html.

Papafitsoros, Kostas, Lukáš Adam, Vojtěch Čermák, et al. (2022). *SeaTurtleID: A novel long-span dataset highlighting the importance of timestamps in wildlife re-identification*. arXiv: 2211.10307.

Papafitsoros, Kostas, Lukáš Adam, and Gail Schofield (2023). "A social media-based framework for quantifying temporal changes to wildlife viewing intensity". In: *Ecological Modelling* 476. ISSN: 0304-3800. DOI: https://doi.org/10.1016/j.ecolmodel.2022.110223. URL: https://www.sciencedirect.com/science/article/pii/S0304380022003210.