Shark Identification Neural Network

Project Report

Claire Fielden
University of Cape Town
FLDCLA001@myuct.ac.za

Danny Guttmann
University of Cape Town
GTTDAN002@myuct.ac.za

Carlton Ndhlovu
University of Cape Town
NDHCAR002@myuct.ac.za

1. Problem Description

1.1. Background

According to the Shark Research Institute, there are more than four hundred (400) different species of shark. Marine biologists, scuba divers, and any other marine enthusiasts who encounter sharks may not have time to study the sharks that they encounter while in the ocean. Those lucky enough to obtain photographs of the sharks they spot, will not be able to correctly identify the species to which the photographed shark belongs, unless they are experts.

1.2. Objective

Artificial Intelligence was applied to address this unfortunate conundrum. This was achieved by the development of an artificial neural network that was trained to classify pictures of sharks as particular shark species.

1.3. Potential User Community

This Artificial Neural Network (ANN) is targeted towards any person interested in sharks, as the network is intended to classify pictures of sharks as particular species. Due to scope limitations and time constraints, the network is only able to classify a small subset of shark species. However, the network has the potential to be expanded and improved to be able to correctly recognise and classify all the known shark species.

1.4. Ethical Considerations

The images of sharks used for training and testing were obtained from kaggle.com. This is an open source database of various datasets that are freely available. As far as the developers of the the ANN are aware, no sharks were harmed in the development and testing of the ANN.

1.5. Problem Formulation and Dateset

The original training set can be found on Kaggle.com. The dataset consisted of a training set of one thousand shark images, each allocated to its own file according to species. There were 14 species under consideration:

1. Basking
2. Blacktip
3. Blue
4. Bull
5. Hammerhead
6. Lemon
7. Mako
8. Nurse
9. Sand Tiger
10. Thresher
11. Tiger
12. Whale
13. White
14. Whitetip

The dataset was downloaded and each image in the file renamed according to its species. Each file was then resized to have 634 x 425 pixels, as well as a 64x64 resolution. After this normalization, the dataset was loaded into Python. This can be done in the program either by saving the normalised dataset in Kaggle.com or by saving the kaggle.json file to your .kaggle file in C drive and choosing "Y" when prompted by the application. This will download the data directly from Kaggle.com into your directory.

Once the data is accessible, it is split into a Test set and a Train set. 70 of the images from each file are allocated to training data, and the remaining data within each file is allocated to the Test set. All binary file header labels were discarded and the 270 000 pixel values converted from byte form to integer form. These pixel values are written out, tab-delimited, in one line of the text file. At the

end of this line, a label indicating which digit that pattern of pixels represents is included. Then the next sequence of pixels is loaded into the same text file. This process is completed for all 980 training images. In a separate text file, 566 test image pixel arrays and their labels are stored. Binarizatio of the pixel values could not take place, as the colouring of sharks was to be taken into consideration when performing classification. The string of pixel values was reshaped into a Tensor with a height and width of 48 pixels. All labels are also stored in a separate Tensor. Both Tensors are stored within the DataSet object to be accessed during training and testing.

The training dataset for all Classifiers were fed to the networks using a DataLoader iterable, created from PyTorch's data library [8]. This object, named train_dataloader for all Classifiers, was passed samples in minibatches of 64 Tensors with each epoch. After every epoch, the data in the DataLoader is reshuffled and a new batch passed to the network. This reshuffling allowed reduction in overfitting.

The hypothesis space originally consisted of 14 classes, however as experiments were conducted, there was a significant improvement in the prediction capabilities when the hypothesis space was reduced to 9 classes. The performance of a classifier is strongly influenced by its ability to generalise. Therefore, it is difficult to find the best ANN to produce a hypothesis space that is general enough to classify many images, but specific enough to classify them correctly (the maximally specific hypothesis). For a network architecture to be generalised, the number of free parameters to be determined by the learner was kept to a minimum. Free parameters in this case refer to the coefficients determined through the model's learning process. However, in doing this, the computational power of the network should not be reduced, as this would degrade the specificity of the algorithm to too high a degree. Correct generalisation is essential as it results in reduced entropy [3].

2. Baseline Explanation

Classifier_1.py was written to represent a baseline for the model. It has a topology resembling a Multi-Layer Perceptron [4]. Boasting three layers, the hidden layer receives input from every input node, and the output node receives input from each neuron in the hidden layer. Thus, it is fully connected.

The goal of this network is simply to train a multi-layered neural network to learn appropriate internal representations for mapping from input to output. The Multi-Layer Perceptron is likely the simplest form of a deep neural network and provides a good basis from which to create other classifiers. The topology of this network is represented in the table in Figure 1.

2.1. Topology

This network is built entirely out of neurons, each with their own weights and biases. If a neuron has two inputs, it requires three weights, the third being dedicated to the bias. The bias adjusts the output along with the weighted sum of the inputs to the neuron and is a measure of a perceptron's probability to output a 1.

At the input layer there are 6912 nodes, arranged in such a way that each node accepts a singular feature from the 48x48 Tensor that is passed to the model in the form of a one-dimensional array. Each element in the array is a pixel containing three values: the intensities of the red, green, and blue sub-pixels. The inputs are combined with the initial, stochastic weights in a weighted sum and subjected to the activation function. The result of the activation function is then passed to 64 neurons that reside in the Hidden Layer.

Every neuron in the Input Layer feeds every neuron in the Hidden Layer their internal representation of the data. The Hidden Layer creates a form of "bottleneck", which aids in the reduction of parameters in the network. However, the main reason behind implementation of this layer is to allow compression of feature representations. This allows salient features to be pinched to best fit the available space. Improvements to the network's compression occur as weights are updated.

Each neuron in the Hidden Layer constructs its own collection of pixels. These neurons produce outputs by weighing up the result received from the input layer, thus making their decisions more complex. Depending on the weights in the hidden layer and the combination of hidden neurons that are "activated", the Multi-Layer Perceptron can make a prediction on which image was fed into the network.

The Hidden Layer performs its weighted sum of inputs, feeds this to the ReLU activation function, and allows the results to be subjected to the 9 output neurons. These neurons represent the 9 possible classes for classification. Each neuron will return a value between 0 and 1, an output that is used by the Cross Entropy Loss function as a probability distribution. All true values are one-hot encoded so that a 1 appears in the column corresponding to the correct category, otherwise a 0 is produced.

Classifier_1 returns a prediction tensor that displays the probability of each class label decided

Topology of Classifier_1				
LAYER	**INPUT**	**MODULE**	**ACTIVATION FUNCTION**	**OUTPUT**
Input Layer	48x48 Tensor of Pixels	Linear	ReLU	Weighted sum of inputs
Hidden Layer	64 weighted sums	Linear	ReLU	64 weighted sums
Output Layer	64 weighted sums	Linear	None	9 outputs corresponding to relevant classes

Figure 1: Topology of Classifier_1

by the model. Cross Entropy encourages classification by comparing the prediction to the Tensor provided by the training dataset. When the predicted class probabilities are identical to the training set probabilities, there is a Cross Entropy Loss of 0.

2.2. Activation Function

The activation function of a Multi-Layer Perceptron governs the threshold at which a neuron is "fired", as well as the strength of the output signal. In this case, the ReLU() activation function was chosen as it will output the input directly if it is positive, otherwise it will output zero.

This is useful in such a case where the value of the pixels fed to a neuron are either 0 or 1, allowing the output of a neuron to directly represent the weighted sum. Thus, when the Output Layer receives input from the Hidden Layer, it can simply choose the input with the highest value and make a prediction according to that classification.

Additionally, ReLU does not suffer from the vanishing gradient problem. Weights do not experience insignificantly small updates during the training process. This model is considered a "faster" learning model since the gradients are faster to compute and the weight updates are substantial. However, it should be noted that in cases where such a "shallow" network exists, the vanishing gradient problem is not usually an issue.

2.3. Optimiser

In **Classifier_1**, the Stochastic Gradient Descent optimiser was chosen as it is extremely basic. The main goal of an optimiser is to allow a network to be trained to reach maximum accuracy given resource constraints such as time. This PyTorch optimizer allowed a learning rate of 1 x 10^{-3} to be selected, a hyperparameter referring to the step of backpropagation. It scales the impact the optimizer will allow the gradient to have on the weight. Thus, a small learning rate will cause the network to converge slowly and may cause it to suffer from the vanishing gradient problem.

In SGD, the gradient of the loss function is calculated using a single, randomly selected batch to perform each iteration of the gradient descent algorithm. This contrasts typical gradient descent in which the sum of the gradient is calculated across all training examples. A few samples are selected randomly for each iteration, making this form of optimization useful when the training datasets are particularly large. SGD is less computationally expensive because gradient descent does not have to be performed for every iteration until the local minimum is reached [7].

However, too large a learning rate puts the network at risk of missing the local minimum, causing the network to incur an "exploding gradient" and impose oscillations. When used correctly, the momentum provided by SGD can render some speed to the optimization and may allow local minima to be avoided.

2.4. Loss Function

Loss increases exponentially as the network's prediction diverges from the labelled, expected outcome. How the loss is measured directly affects the optimization process, as it acts as a feedback signal. Based on this feedback signal, the weights in the network are adjusted. If the probability distribution produced by the training label is 1, the network should produce a probability estimate that is as close to 1 as possible to reduce the loss incurred.

Cross Entropy Loss was chosen for the initial classifier because it is the most common loss function used for classification problems. The target in this case is a Tensor of class probabilities that is the same shape as the prediction, where each value ranges between 0 and 1. The goal of this loss function is to measure the difference between two averages of the number of bits of distribution. Thus,

it can compute the error between two probability distribution functions to a satisfactory degree.

The number of bits required to transmit an event from a probability distribution is known as entropy. A distribution in which events have equal probability has a large, balanced entropy. Cross Entropy builds on this idea by calculating the number of bits required to transmit an event based on the comparison of two events. If the output of this loss function is "none", this indicates that the calculation is the same shape as the target. Any other value will contribute to the average loss accumulated for the network.

Cross Entropy Loss also has a role to play in sound network generalization capabilities and a faster training pace. This is due to its ability to directly base its performance on the error in the output. If the error is extremely large, the neuron will update its weights more drastically and thus learn faster.

3. Model Design

3.1. Enhancements to Baseline

Classifier_1 was enhanced to create **Classifier_2**. This Classifier was inspired by Y. Le Cun's "Handwritten Digit Recognition with a Backpropagation Network" [5]. In this case, a multi-layer network with adaptive connections is implemented. These connections are heavily constrained and trained using backpropagation. This Restrictive Connection Scheme is based on the Restricted Boltzmann Machine, which introduces the concept that restriction allows more efficient algorithms. By restricting the weights locally, the number of learnable parameters is reduced, producing a more generalized network.

Unlike **Classifier_1**, **Classifier_2** has a topology characterized by multiple convolutional layers. Convolutional layers are unlike linear layers as they are not fully connected. The neurons receive inputs only from a specific subsection of the previous layer. Each neuron is responsible for its own receptive field, allowing a sumptuous ability to perceive small details that make up the defined region of space that is exposed to the neuron. This increases the expressive power of the network architecture and in turn broadens its representation of the hypothesis space.

It is for this reason that the design of Classifier_1 evolved into Classifier_2. All connections are fully adaptive despite being highly constrained. This forced locality, particularly amongst the first few layers, is beneficial as, if the feature detector is advantageous on a certain segment of the image, it is likely to be useful in other segments as well. This is because salient features of a distorted feature may be displaced slightly from their position in a typical character. Convolutional layers [1] and the receptive fields they provide allow recognition of visual features within a small region of the input array. As these receptive fields share weights, units in the feature map are constrained to perform identical operations on different segments of the input.

Consequently, the number of trainable parameters within each layer is drastically reduced by these local receptive fields. This reduction plays a very important part in reducing the capacity of the network and thus improves generalization [2]. The network's capacity was weakened in the hopes of preventing Classifier_2 from memorizing the training set and exploiting accidental features such as noise, thus improving overfitting. A further step taken in the prevention of overfitting was the reduction in the size of the images fed from the dataloader. In Classifier_1, they were 48x48, however Classifier_2 takes in 28x28 arrays.

Additionally, this weight sharing technique introduces shift invariance, allowing the result to be shifted upon modification of the input, however leaving the result unchanged otherwise.

Lastly, **CLassifier_2** has shown to be an extremely fast learner due to the use of an Adam optimizer and Leaky ReLU activation function.

3.2. Topology

The topology of **Classifier_2** is described in the table in Figure 2.

The initial four feature maps are produced as output after applying the first filter of size 5x5 to the input image. This propagation of the image into a channel is performed by a singular input neuron producing a local receptive field. These feature maps experience a reduction in size from the original 28x28 Tensor due to the size of the kernel and the addition of padding. The padding, in this case, was not implemented by Le Cun. It was added due to an observation that as the image is filtered from layer to layer in a convolutional network, the output may become so small that it loses meaning. Thus, padding was introduced to prevent loss of valuable data. The original input size is preserved to a certain degree, preventing data from the input from being discarded at too high a rate. These four feature maps then feed their weighted sums into the LeakyReLU activation function.

The first Hidden Layer is a Max Pool layer, which allows meaningful data to be segregated from unmeaningful data, consequently removing noise from the input image. By extracting signif-

Topology of Classifier_2					
LAYER	**INPUT**	**MODULE**		**ACTIVATION FUNCTION**	**OUTPUT**
Input Layer	3 (28x28) Tensor of Pixels	Convolutional		LeakyReLU	4 (28x28) Feature Maps
		Kernel size	5x5		
		Padding	1		
		Stride	1		
Hidden Layer 1	4 (28x28) Feature Maps	Max Pool		None	4 (13x13) Feature Maps
		Kernel size	2x2		
		Padding	1		
		Stride	2x2		
Hidden Layer 2	4 (13x13) Feature Maps	Convolutional		LeakyReLU	12 (11x11) Feature Maps
		Kernel size	5x5		
		Padding	1		
		Stride	1		
Hidden Layer 3	12 (11x11) Feature Maps	Max Pool		None	12 (5x5) Feature Maps
		Kernel size	2x2		
		Padding	1		
		Stride	2x2		
Flatten()	The "cube" received from the convolutional layer is flattened into a 1-dimensional linear vector before being passed to the fully connected linear layer.				
Output Layer	A single 1-dimensional vector with 300 features	Linear		None	9 outputs corresponding to relevant classes

Figure 2: Topology of Classifier_2

icant featured from the given Tensor, the dimensionality is reduced by placing only the most activated pixels in the outputted feature map, whilst discarding the less activated features. This layer performs a 2-to-1 subsampling, reducing the size of the feature maps received from the previous layer to half. This was aided by the implementation of a 2x2 stride, which, in combination with padding and the 2x2 kernel, halved the input received from the feature maps in the previous layer. This is a lot of detail being discarded. However, it is likely this will improve the performance of the network by improving its ability to generalize.

Hidden Layer 2 performed another convolution on these feature maps using a 5x5 kernel and a 1-pixel zero-padding border that reduced the size of the output feature maps by two pixels. It is composed of groups of 169 units, coordinated as 4 independent 13x13 feature maps. Each unit in a feature map obtains input from the kernel. As aforementioned, all 169 units will share the same set of 171 weights (including bias). Units in a different feature map on the same layer will share another set of 171 weights.

Lastly, 12 feature maps are fed into Hidden Layer 3, which once again performs the Max Pool operation. The input plane is transformed, via the use of a 2x2 kernel and 4-pixel stride, to consist of 12 5x5 feature maps. This is a large reduction in size from the original 28x28 Tensor. These 12 maps containing 25 pixels each are flattened into a vector and passed to the linear layer. The Output Layer subsequently makes its predictions based on the linear combination of the feature maps fed into the output layer.

3.3. Activation

LeakyReLU is a variant of ReLU. It does not result in input values below 0 being set to 0, but allows a small negative gradient in this domain. This is beneficial because the backward pass can alter weights that produce negative pre-activation. Due to these negative pre-activations, negative values can be produced instead of eliminating the problem of weights only being updated in one direction that was associated with the ReLU. This is known as the Dying ReLU problem and arises when neurons become inactive no matter what input is supplied. No gradient flows in this region, which poses a threat to the network if a large number of "dead" neurons are in existence.

Additionally, the LeakyReLU was chosen over the Sigmoid due to its ability to produce zero-centred distribution, thus improving the time taken to train the network. Despite the sparsity of the outputs, LeakyReLU does not guarantee elimination of vanishing gradients because if the output of a filter set in a layer is always negative, the gradient is likely to vanish. This probability is worsened by the depth of this network. Neither does this activation function solve one-sided saturation, which puts the model at risk of poor convergence. Additionally, as it possesses linearity, the LeakyReLU may in fact lag behind the Sigmoid function in certain classification problems. However, this model did not experience this consequence.

3.4. Optimizer

The Adam optimizer is an adaptive moment estimation algorithm and an extension of SGD. However, it may be considered more beneficial in cases like **Classifier_2** as it is known to implement an adaptive learning rate. The adaptive learning rate affects each parameter differently, which is useful when so many parameters are changing from layer to layer.

By updating the learning rate for each network weight individually, each feature map can acquire its own learning rate. Thus, parameters that would ordinarily receive smaller or less frequent updates, such as ones in the lower layers, may receive larger updates with this optimizer. This allows the network to learn more swiftly.

Adam's ability to traverse saddle points adeptly has gifted this optimizer very high accuracy. Thus, it requires less iterations than the SGD optimizer to train the network. However, the cost of this high computational speed is generalization. Although the results produced by **Classifier_2** are expected to be extremely favourable, the ability of this model to generalize may be poor.

3.5. Loss Function

As with Classifier_1, Cross Entropy Loss is used for Classifier_2. The classification decision is still based on the probability distribution produced by the network. Cross Entropy is positive and tends toward zero as a given node becomes more efficient in its computation of the desired output.

4. Model Testing and Evaluation

4.1. Experimental Setup

Experimentation took place in three phases:

1. Tuning of hyperparameters
2. Implementation of gradient clipping
3. Change in data size

4.1.1. Tuning of Hyperparameters

The following hyperparameters were changed in order to find the optimal model:

1. The number and type of layers, including Batch Normalization [6] and Dropout [10]
2. The number of neurons in hidden layers
3. Use of different activation functions, namely Linear, Sigmoid, and "flavours" of ReLU
4. Use of different loss functions, namely Stochastic Gradient Descent, Negative Log Likelihood, and different "flavours" of the Adam Optimiser
5. Increasing and decreasing the learning rate

Initially, the model was trained on 5 epochs. However, models that displayed promise by quickly improving accuracy within these epochs were re-trained on 10 epochs and classification of a random shark image evaluated.

4.1.2. Implementation of Gradient Clipping

When making changes to the loss function and learning rate, the model would at times become extremely unstable. The loss values would become extremely large or small, known as the vanishing or exploding gradients problem. In order to rectify this issue, Gradient Clipping was implemented [9]. Not only was the clip value changed, but the threshold at which it was implemented was also experimented with. However, gradient clipping did not improve the accuracy with which the model made predictions and was eventually discarded as an improvement to the model.

4.1.3. Change in Data Size

As the model's accuracy was still not rising above 12% with the chosen hyperparameters, the size of the input tensor was reduced by 1/3. The height and width of the pixels was transformed from 48x48 to 28x28. Additionally, the number of classes was reduced to 9 in order to analyze the effects this would have in "memorization" of specific species in specific classes. The set of hyperparameters values that provided a better overall accuracy for the model was selected and used in the evaluation of the final model. In addition to the hyperparameter tuning, both the classifiers (**Classifier_1** and **Classifier_2**) were trained and evaluated using 9 and 14 classes for their outputs to determine which would perform better.

Due to the stochastic nature in which the weights are initialised at the start of training, this meant that the model was re-trained and evaluated for four iterations as to obtain more accurate results for the models performance.

5. Discussion and Results

The results displayed in Appendix 7 showcase the accuracy for each classifier for its respective output classes. Firstly, Table 7 displays Classifier_1's percentage accuracy when operating on 14 classes. Additionally, each image has a size of 48x48. The average accuracy is 8.92% with the baseline MLP.

A small improvement in accuracy can be seen when reducing from 14 classes to 9 classes. As seen in Table 7, the average accuracy increases to 10.38%. This displays the prediction capabilities of the model improves when there are less classes to choose from, as this reduces the complexity of the task. Across the epochs, **Classifier_1** with 14 output classes was stagnant and made no improvements in terms of its accuracy suggesting the limitations of that classifier which is further backed up by its 9 output classes equivalent which also made no improvements. On the other hand, **Classifier_2**, made an average from 60% improvement between the first and tenth epoch for the 14 output classes version and a 40% increase in 9 output classes version. This can be attributed to **Classifier_1** being less complex than **Classifier_2** by having fewer layers meaning that there are far fewer weights that can be modified and thus less computation preventing it from improving.

When implementing the Restricted Boltzmann Machine, even on 14 classes, the accuracy improved to 10.76% in Table 7. This is most probably due to the restricted connection scheme, which constrained the connections in the first few layers to be local. This constriction led to far fewer learnable parameters in each layer. The max pooling layers provided a compression of parameters to an extremely high degree, allowing the number of parameters in **Classifier_2** to be nearly half of that in **Classifier_1**. Thus, this model is much less prone to overfitting, resulting in better accuracy.

Furthermore, the accuracy displayed in Table 7 shows a promising convergence curve, where the starting accuracy starts relatively low and grows to a value of 20 or more within 10 epochs. This displays that. given more training, the model could most probably reach even higher states of prediction accuracy. Having been trained on images of size 28x28, with 9 output classes, it averages an accuracy of 16.15%. Thus, it outperforms all of the previous models.

6. Conclusion

This assignment has shown that **Classifier_2** performs better than **Classifier_1** at classifying shark images given as input. However, both these classifiers have very low accuracy which makes them

infeasible solutions for solving this task. To further simplify the task, images could be presented to the Restricted Boltzmann Classifier in greyscale, or fewer classes could be presented for classification and more training data provided. However, as these improvements would not improve the usability of this model in real-world problems, the optimal outcome is that further research is required to investigate using a neural network to classify sharks whilst taking their colouring into consideration. With this, **Classifier_2** can be used as a basis to work from whereby it could be tested for longer epochs or even looking into other neural network architectures such as a Recurrent Neural Network [3].

7. Appendix A

Table 1: Classifier 1 with 14 classes

Epoch	1	2	3	4	5	6	7	8	9	10
Run 1	10.6%	10.5%	10.5%	10.5%	10.5%	9.5%	10.5%	10.5%	10.4%	10.5%
Run 2	6.6%	6.6%	6.6%	6.6%	6.6%	6.6%	6.6%	6.6%	6.6%	6.5%
Run 3	8.3%	8.3%	8.3%	8.3%	8.3%	8.3%	8.3%	8.3%	8.3%	8.3%
Run 4	10.4%	10.4%	10.4%	10.4%	10.4%	10.4%	10.4%	10.4%	10.4%	10.4%

Table 2: Classifier 2 with 14 classes

Epoch	1	2	3	4	5	6	7	8	9	10
Run 1	6.5%	12.1%	9.5%	8.2%	13.7%	14.2%	13.5%	15%	14.8%	15.8%
Run 2	6.5%	6.8%	6.9%	8.3%	6.9%	7.5%	5.3%	8.1%	11.4%	9.2%
Run 3	8.6%	10.8%	12.7%	11.2%	11.2%	15%	11.1%	14.2%	20.3%	12.5%
Run 4	7.8%	10.1%	7.9%	10.4%	11.8%	12.2%	12.1%	10.4%	9.9%	10.1%

Table 3: Classifier 1 with 9 classes

Epoch	1	2	3	4	5	6	7	8	9	10
Run 1	13.1%	13.1%	13.1%	13.1%	13.1%	13.1%	13.1%	13.1%	13.1%	13.1%
Run 2	10.2%	10.2%	10.2%	10.2%	10.2%	10.2%	10.2%	10.2%	10.2%	10.2%
Run 3	7.8%	7.8%	7.8%	7.8%	7.8%	7.8%	7.8%	7.8%	7.8%	7.8%
Run 4	10.4%	10.4%	10.4%	10.4%	10.4%	10.4%	10.4%	10.4%	10.4%	10.4%

Table 4: Classifier 2 with 9 classes

Epoch	1	2	3	4	5	6	7	8	9	10
Run 1	10.6%	12.6%	15.1%	17.7%	21.5%	18.6%	12.4%	21.7%	22.2%	26.6%
Run 2	18.6%	19.3%	18.4%	18.0%	17.7%	18.0%	16.4%	16%	14.9%	12.6%
Run 3	13.3%	14.2%	15.7%	15.1%	13.7%	11.8%	12.4%	13.3%	15.1%	14.2%
Run 4	10.9%	15.5%	17.3%	13.5%	14.6%	18.2%	19.5%	21.1%	22.6%	21.3%

References

[1] R. Alake, *Understand local receptive fields in convolutional neural networks*, Nov. 2021.

[2] J. Brownlee, *How to avoid exploding gradients with gradient clipping*, Aug. 2020.

[3] P. Chaber and M. Ławryńczuk, *Pruning of recurrent neural models: an optimal brain damage approach*, 2 (2018), pp. 763–780, 10.1007/s11071-018-4089-1.

[4] N. Kang, *Multi-layer neural networks with sigmoid function- deep learning for rookies (2)*, Feb. 2019.

[5] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, *Handwritten digit recognition with a back-propagation network* (1989).

[6] M. Liu, W. Wu, Z. Gu, Z. Yu, F. Qi, and Y. Li, *Deep learning based on batch normalization for p300 signal detection*, C (NLD, Jan. 2018), pp. 288–297, 10.1016/j.neucom.2017.08.039.

[7] O. Matan, H. Baird, J. Bromley, C. Burges, J. Denker, L. Jackel, Y. Le Cun, E. Pednault, W. Satterfield, C. Stenard, and T. Thompson, *Reading handwritten digits: a zip code recognition system*, English (US), July 1992, 10.1109/2.144441.

[8] J. McCaffrey02/01/2022, *Preparing mnist image data text files.*

[9] M. A. Nielsen, *Neural networks and deep learning* (Determination Press, 2015).

[10] A. Poernomo and D.-K. Kang, *Biased dropout and crossmap dropout: learning towards effective dropout regularization in convolutional neural network* (2018), pp. 60–67, 10.1016/j.neunet.2018.03.016.