Convolutional Neural Network towards image classification of the CIFAR-10 dataset

Julien Guinot
The University of Adelaide
julien.guinot@student.adelaide.edu.au

Abstract

The Exponential augmentation of computing capacity in computers over the course of the recent decades has allowed for developing of deeper and deeper neural networks [1], especially convolutional neural networks (CNN) over the course of recent years [2],[3],[4],[5].Image classification problems have been the playground of deeper and deeper CNNs, and have been well explored for problems such as the CIFAR-10 dataset [6].

This paper explores the basics of CNN structures and architectures, as well as well-known CNN architectures with the goal of applying these principles to building a custom CNN for the CIFAR-10 dataset.

A structure inspired by the VGG16 architecture is proposed, and attains comparable results to structures which are a few years old, while lacking some accuracy compared to newer structures, with satisfying accuracy results.

Finally, solutions to reduce overfitting while training are proposed, and a data augmentation pipeline is set up for said purpose, proving useful in reducing overfitting.

1. Introduction

Convolutional Neural Networks were implemented as soon as 1989 to address a problem that has today become widespread and essential to many applications, integral to the field of computer vision: Image classification. Since then, many new architectures of increasing complexity have emerged to address more and more complex problems. Some of these famous architectures include AlexNet, GoogLeNet, and ResNet.

Today, Many computer vision problems are well explored. However, some still hold pedagogical value towards exploring the performance of Convolutional Neural Networks (CNN) and understanding the competing architectures in the field. The CIFAR-10 dataset is one of these well-explored problems. It is comprised of images belonging to one of 10 classes, which will be discussed later. Through modern architectures, close to 100% of accuracy can be achieved on this dataset. This paper will focus on a more naive approach, the technical approach of which remains within the scope of what was covered in the *Deep learning fundamentals* course of the University of Adelaide.

The goal of this paper is to present the approach used to design, implement, and test our own CNN on the CIFAR-10 Dataset to correctly classify sample data. In the first place, the contents of the CIFAR-10 dataset will be overviewed as well as explored in terms of usability for the problem. The basics of CNNs will also be covered as well as some of the most well-known architectures. The architecture chosen for our CNN will be presented, and the various choices for hyperparameters will be explained. Finally, performance analysis of the final structure will be conducted, and its performance will be compared to the previously-discussed architectures. Optional ways to improve the performance of our algorithm will also be briefly covered in the last section of this paper.

2. Overview of provided data

The data used for this project is a well-known dataset called the CIFAR-10 Dataset, comprised of 60000 RGB images of dimensions 32×32 pixels. So, the processed training data will be individual arrays of shape $3 \times 32 \times 32$, with 3 entry channels for the CNN. The dataset contains 10 classes, which are the following: **Plane, Car, Bird, Cat, Deer, Dog, Frog, Horse, Ship, Truck**.

The backend for the building and training of the model, Pytorch, also provides a built-in CIFAR-10 dataset, which will be used for the purpose of training and testing the model: `https://pytorch.org/vision/stable/datasets.html#cifar`. The code for visualizations shown in this section is contained in the following repo: `https://github.com/Pliploop/ConvNN_From_Scratch`

2.1. Exploratory data analysis

The goal of this section is mainly to visualize some of the samples for each class, and consider the issue of class imbalance. Below are shown 4 samples for some of the aforementioned classes (Fig. 1 through 3)

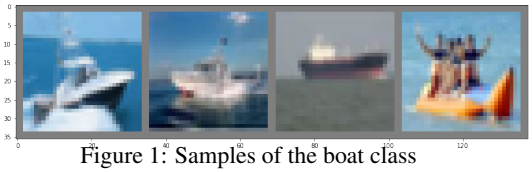

Figure 1: Samples of the boat class

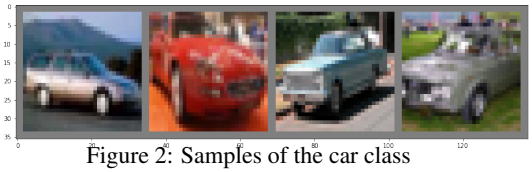

Figure 2: Samples of the car class

Figure 3: Samples of the deer class

Concerning class balance, each class contains 6000 samples in the CIFAR-10 dataset [6], and pytorch generates class-balanced data through the dataset module, which allows us to verify that all classes are perfectly balanced in our training data. This is shown in Figure 4:

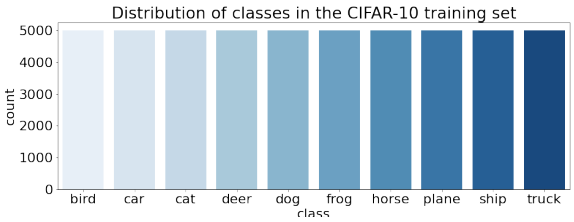

Figure 4: Distribution of data in the training set for the CIFAR-10 dataset

Now that the data has been covered, and is deemed appropriate for the task at hand, both in terms of format and of class balance, we can explore the mathematical principles behind convolutional neural networks.

3. Convolutional Neural Networks

The goal of this section is to present the mathematic principles behind CNNs, as well as explain some of the various layers that can be used in CNN architecture design, and their functions. A brief overview of a few well-known CNN architectures will be undertaken in this section as well.

3.1. CNN principles

Convolutional Neural Networks (CNNs), are a special kind of neural networks with an architectural specificity: convolution layers, which will be explored in a later section. Though this specificity makes them very apt to classify and identify data with clear spatial or temporal structure, they do not differ by much in comparison to classical neural networks. Their structure is fairly similar, and the backpropagation, loss function and training principles which are true for traditional Neural Networks such as the MultiLayerPerceptron, which was covered in a previous Paper, remain largely the same. For reference, the general structure of a Neural Network with one hidden layer is shown in the following figure (Fig. 5):

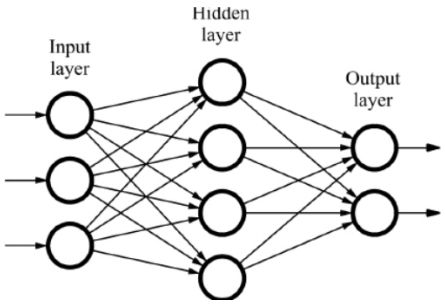

Figure 5: General Structure of a Neural Network

With weights influencing the inter-correlations and influences the layers have between them. CNNs take the assumption that NN neurons behave quite like biological neurons in that they are influenced by a weighted and activated summation of signals by the neurons around them in the brain, and adds to this the biological realization that when the brain is tasked with a complex problem, some neurons deal with simple operations, and others with complex operations. For instance, some neurons deal with recognizing edges in vision, and input this information to other neurons, which provide simpler operation functionality.

CNNs aim to translate the behaviour of these complex neurons by what are called convolution layers. In the following section, an overview of the basic building blocks of a CNN, layers will be undertaken.

3.2. Common CNN Layers and their functions

Firstly, CNNs have two basic blocks: **A feature learning block and a Classification block**. This, again, mimics the aforementioned neural behaviour of the brain. When humans classify an object in their field of view, they first aim to identify features relative to the object that they have learned before (Wheels for a car, wings for a bird...). This is the goal of the feature learning block. Layers that comprise the feature learning block can be of the following types:

- Convolution Layers
- Pooling Layers

Which will all be discussed in detail in the following section. The classification Block addresses the more simple operations provided by less complex neurons in the brain. It's essentially a simple NN with any amount of hidden layers, and emulates the biological operation of *"given the features identified by my complex neurons, what am I looking at?"*. Possible blocks in this structure include:

- Fully connected layers
- Softmax classification layer

Furthermore, Whichever of these blocks can contain additional layers which functions will be discussed later on :

- Activation layers
- Dropout layers
- Batch Normalization layers

Many improvements upon these basic layers have been built over the years, including more complex structures containing these basic building blocks. These are outside the scope of this paragraph, and will be covered when addressed in their respective well-Known architectures in Section **3.3**

Convolution Layers

Convolution layers are the basic building blocks of CNNs. Convolution layers take as input a batch of n images with m channels (usually 3 for RGB images) and of dimension $W \times H$. So an array of size $n \times m \times W \times H$ is given. A series of convolutions, or "filters", are applied to the batch of images. To visualize the action of a convolution, Fig. 6 below schematizes the action of a convolution layer with $k = 6$ output channels (so 6 filters), of size 5×5

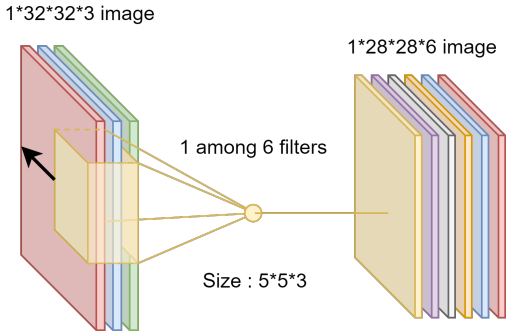

Figure 6: Action of a convolutional layer on an image

Here, the yellow filter (corresponding to the yellow layer of the output array) "slides" over the image, computing dot products for every stop allong the way, and each dot product is the new value in the yellow filter. the **filter** w_k computes the dot product by the following:

$$y_{ijk} = w_k^T x + b \tag{1}$$

where y_{ijk} is the i, j pixel of the output array's k^{th} layer, and x is the array encapsulated by the filter. What is learned during backpropagation is the filter weights w_k for each filter.

Additional parameters, such as **stride,dilation,padding** can be used to dictate the manner in which the filter goes over the original image, and modify the dimension of the output array. Their exact influence on the size of the array is outside of the scope of this paper, but it is possible to decide the shape of the output as desired. Activation layers, which will be discussed later, are put after the convolution layer to simulate neuron activation on the output array layers. These activated layers are called **activation maps**

Pooling layers

The second type of layers the feature extraction block of the algorithm has are pooling layers. Pooling layers serve the purpose of reducing the size of convolutional layers' outputs as to make the calculations conducted by the algorithm more manageable from a computational standpoint. Pooling layers can be **Max pooling layers** or **Average pooling layers**. They are applied on each of the activation maps. Fig. 7 summarizes the action of a 2×2 **Max pooling layer** on a $28 \times 28 \times 6$ array.

Softmax and Fully connected

if the classification block, Softmax layers and fully connected layers will be covered succintly, as they are relatively simple in comparison to convolutional layers. Fully connected layers are reminiscent of the structure of a traditional

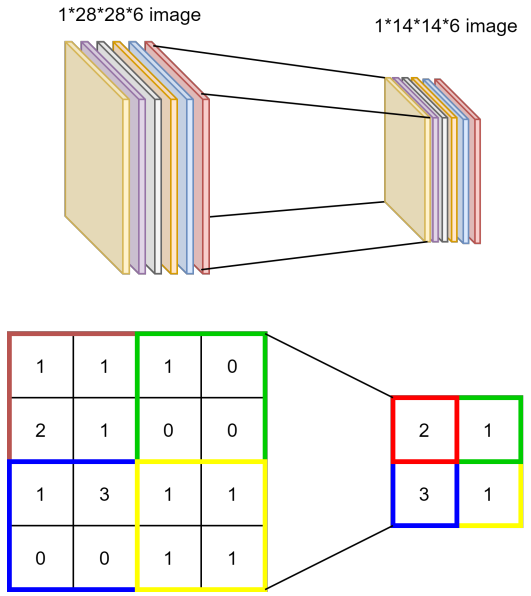

Figure 7: Action of a max pooling layer on an image

NN. The only difference is that arrays must be flattened before being fed into fully connected layers. In the following figure (Fig. 8), the passage from a $1 \times 14 \times 14 \times 6$ through two fully connected layers is shown.

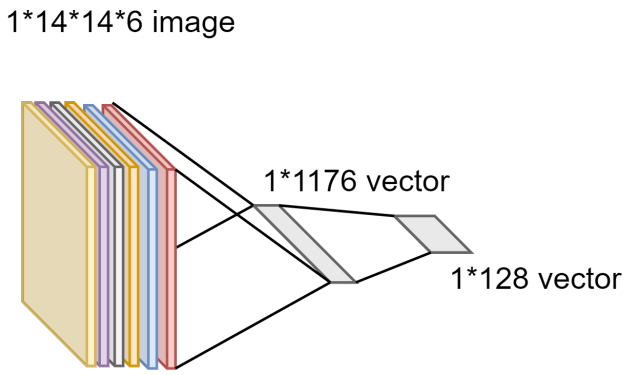

Figure 8: Transition from Feature extraction block to classification block

Concerning softmax layers, their goal is to output a probability distribution for the last layer of the network, which must be a fully-connected layer of the same amount of elements as the amount of classes for the given classification problems. For a given vector z, with a one-hot-encoded vector, to obtain a probability distribution representing the probability of belonging to the given class for each feature of the output vector , the following function is applied:

$$y^i_{pred} = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{2}$$

This gives us all the building blocks for making a basic CNN. Considering the CIFAR-10 dataset, an elementary CNN would look like this (Fig. 9):

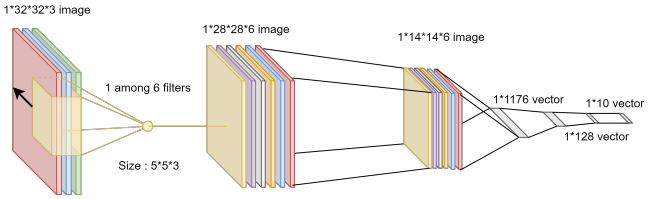

Figure 9: Full CNN for the CIFAR-10 problem

Activation layer

Activation layers can be applied after either convolution layers or fully connected layers. Their objective is to emulate the biological threshold of activation of a neuron. by applying an activation function to the input values of the layer. Many activation functions exist, and some of them are shown in the following figure (Fig. 10):

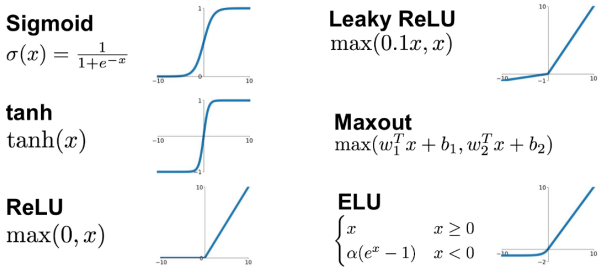

Figure 10: Common activation functions

The specifics of selecting an activation function will be discussed in section **4.2.2**.

Dropout layers

Given the amount of parameters in modern CNNs (some can go up to millions of trainable weights), the risk of overfitting becomes a common problem. Dropout layers can aid in fixing that issue, by dropping random weights in the dot products of convolutional layers and dropping random neuron connections with a given probability p in the fully connected layers. By doing this, the network does not excessively learn the intercorrelations between neurons that might arise without dropping any weights.

Batch Normalization layers

When training, if weights are not controlled, it is possible for them to become very large and diverge if some features become truly prevalent over others. Batch Normalization layers serve the purpose of rescaling the weights within the mini-batch of the training set given to the algorithm during training. To batch normalize a mini-batch B of size m, first one must empirically compute its mean and variance for the mini-batch:

$$\mu_B = \sum_{i=0}^{m} \frac{x_i}{m}, \sigma_B^2 = \frac{1}{m} \sum_{i=0}^{m} (x_i - \mu_B)^2$$

and follow by normalizing the input x:

$$\bar{x_i}^k = \frac{x_i^k - \mu_B^k}{\sqrt{\sigma_B^k{}^2}}$$

Having explored what a basic CNN looks like, some of the most known well-explored CNN structures will now be succintly overviewed to provide a basis for building our own CNN.

3.3. Well-known CNN Architectures

Over the course of the evolution of CNNs, their complexity has grown, with the introduction of new structures, such as residual blocks [4], and have grown to be deeper and deeper, as well as wider. some state of the art CNNs will be surveyed here in chronological order, and the structure of one of them will be used as inspiration for our own network:

3.3.1 AlexNet - 2012

One of the first CNNs to be considered deep (8 layers) Was AlexNet, which performed stunningly on the *ImageNet Large Scale Visual Recognition Challenge* [2]. its structure is shown below (Fig. 11). It used the ReLU activation function for its activation layers.

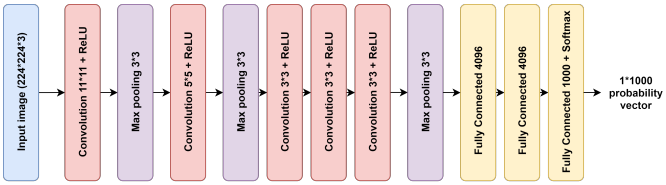

Figure 11: AlexNet CNN

3.3.2 VGG16 and VGG19 - 2014

VGG16 and VGG19 are two variants of the same structure, VGG19 having 19 trainable layers while VGG16 has 16. It achieved 92.7% Accuracy on The *ImageNet dataset* [3] its structure is shown below. It also uses the ReLU activation function between its convolutional blocks (Fig. 12):

Figure 12: VGG16 CNN

3.3.3 ResNet50 - 2015

ResNet50 was the first true deep CNN, jumping from a 22-Layer blocking point model accuracy progression directly up to a whopping 152 layer-Deep-Model. ResNet solved the previous vanishing gradients issue which was limiting the layer depth possibility by introducing what were called *residual blocks*. [7]. This prevents the CNN forgetting features after a great amount of layers. The structure for a residual block is shown in Fig. 13:

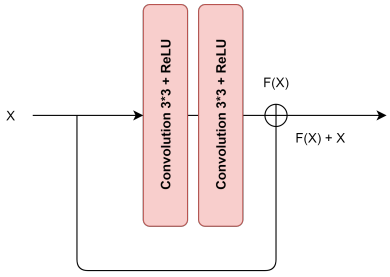

Figure 13: Residual block in ResNet50

And the global structure for ResNet is shown Below (Fig. 14). It also uses the ReLU activation function between its layers.

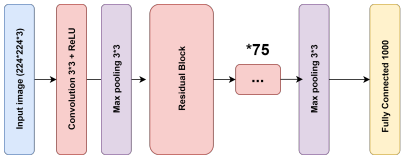

Figure 14: ResNet50 Network

3.3.4 ShuffleNet - 2017

ShuffleNet was built with the goal to be an extremely efficient network usable by mobile devices with less computing power. It achieves this by introducing two new elements:

- Pointwise group convolution
- Channel shuffle

Though the precise workings of these new elements are out of the scope of this paper, both serve to speed up the calculations of the network. Pointwise group convolution separates the convolution filters of the image onto multiple devices to compute parallel activation maps. Channel shuffle allows for the flow of information between channels to happen before connecting to the next layer.

Both elements aid [8] in developing a new residual block structure proper to the ShuffleNet structure, which is shown below in Fig. 15, taken directly from [8]:

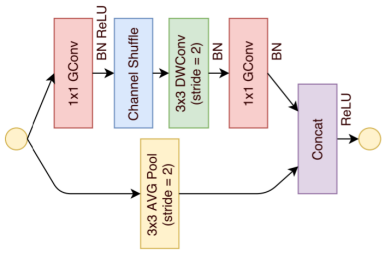

Figure 15: ShuffleNet Residual Block structure

3.3.5 DenseNet - 2018

DenseNet, the most recent amongst the networks we will be overviewing, proposes a new method of connecting layers between them. Indeed, in [5], Huang et Al. Propose to use all feature maps from preceding layers as inputs for the following layers, according to the following schematic (Fig. 16):

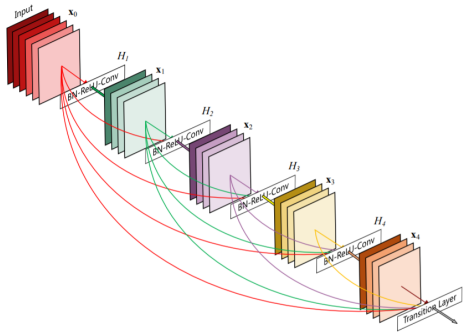

Figure 16: DenseNet Feed-forward strucure

So, instead of L connections between layers in a traditional CNN with L layers, this network has $\frac{L(L+1)}{2}$ [5], which allows to greaty densify the network and reduce the number of parameters, thus speeding up computing greatly. Furthermore, this densely-connected network greatly alleviates the vanishing gradient problem by encouraging feature reuse at every layer and providing all old feature maps for the newest layer to use. [5]

3.4. Results of aforementioned Architectures on CIFAR-10

Below are shown the results for the previously mentioned architectures on the CIFAR-10 Dataset. However, because of lack of computational power and of time for training the models (a single model took about 6 hours to train with 500 epochs), the results are taken from other papers, github repositories, and kaggle notebooks. The sources for each of the accuracy metrics can be found in the references section. This will serve for later comparison with our own model.

Model	AlexNet	VGG16	ResNet50	ShuffleNet	DenseNet
Global accuracy	83.47	92.6	93.6	92	91.47

Table 1: table of accuracy metrics for the well-known CNNs on CIFAR-10

4. Building a custom CNN architecture

This section focuses on the building of our own CNN architecture based on what was explored and given before. The reasons for choosing any given architecture will be explored.

4.1. Code

All of the code for the implemented CNN can be found in the following github repository: `https://github.com/Pliploop/ConvNN_From_Scratch`. Though it has been upgraded with a data augmentation pipeline, the underlying structure remains the same.

4.2. Architecture Overview

This section focuses on the chosen architecture for the CNN presented by this paper. We will firstly discuss the reasons for settling on the architecture used as inspiration, then describe the architecture, and finally elaborate on any additional elements that were added.

4.2.1 Choice of architecture

Two elements are limiting in our choice of architecture. Indeed, building a very deep and complex network inspired by ResNet for instance is tempting, but the time resource necessary to train, test, re-train the algorithm multiple times is lacking given the alloted time to complete this paper and the computing power provided. Furthermore, the size of the dataset images (32×32) does not allow for very deep networks that halve the dimensions of the image at each iteration.

On the other hand, we do not wish to over-simplify the architecture of our CNN to the point of AlexNet, as this might not attain reasonable results on the dataset.

We settle for a structure inspired by VGG16 and VGG19, which provides reasonable depth while maintaining a modulable amount of layers to not reduce the dimensions of the convolution layer outputs, and a reduced amount of parameters to speed up training. This allows training and testing the model multiple times with multiple hyperparameters to zero in on the best settings.

4.2.2 Building the model

As mentioned before, this model was inspired by VGG16 and VGG19, with a reduced number of convolutional blocks to not shrink the activation maps too much. The resulting architecture is shown below (Fig.17), as well as the dimensions of each of the outputs.

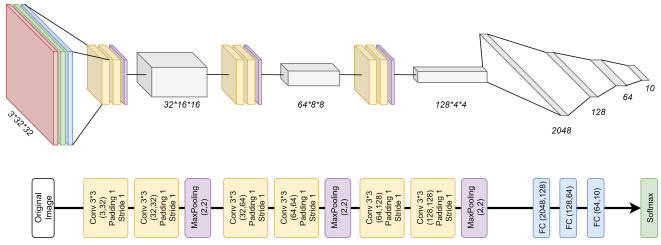

Figure 17: JNet without regularization

ReLU activation layers are set after each convolutional layer and fully connected layer to reproduce the widespread use of this specific activation function in all the previously discussed literature. To this basic structure, batch normalization layers and dropout layers were added after a first round of training to prevent what was observed to be drastic overfitting. The final structure is the following (Fig 18). The training curves for the overfitting algorithm are shown in section **5.3.2**

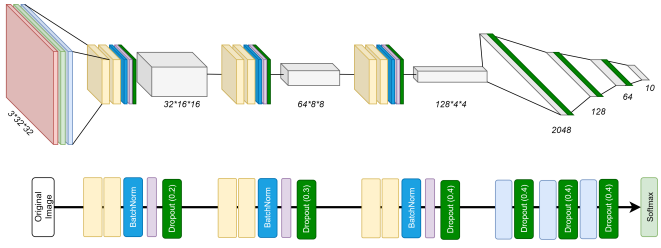

Figure 18: JNet with regularization

The green layers are dropout layers with increasing dropout probability over the length of the network. The blue layers are batch normalization layers. This represents the final state of the network, to which we will give the nickname JNet in the following.

5. Training and performance analysis

The network being built, this section focuses on training JNet and analyzing its performance on the CIFAR-10 dataset. : Choice of data splitting and hyperparameters, as well as the actual training of the model, and the modifications brought to the model over the course of training will be addressed in the first part of this section.

A second part will be dedicated to performance analysis, and visualizing what the model has actually learnt over the course of training, including activation maps for a random image, and feature dimension reduction on the last fully connected layer.

5.1. Data splitting

as a reminder, the CIFAR-10 dataset is comprised of 60000 $32 \times 32 \times 3$ images of varying classes. this a consequential amount of data, and thus the train-validation-test split can be a bit more oriented towards training data. We choose to split the dataset into 45000 training images, 5000 validation images, and 10000 test images.

this represents a **75% - 8% - 12%** training split, quite close to the usual **80-10-10** for a training-oriented split. The strange distribution is due to the pytorch backend, which makes it quite awkward to generate a validation set. We will, however, move forward with this split.

K-Fold validation

One technique to ensure that the validation accuracy obtained on the train-validation-test split is accurate is to perform K-Fold validation. What this means is that the data is split K times (into K folds of data), and trained each time on the corresponding training set, and validating it on the corresponding validation set. By averaging the accuracy obtained on all validation sets, we obtain a representative estimation of the accuracy of the model on data it has never seen before. This is shown in Fig. 19.

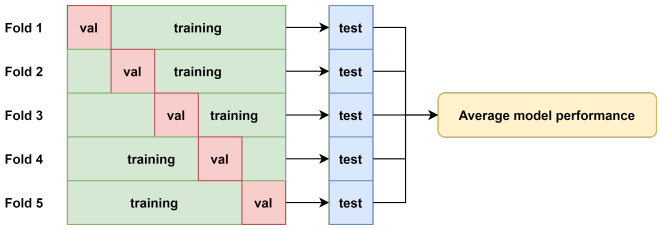

Figure 19: K-fold validation with $K = 5$

Though this would provide additional conviction that our model is performing well, given the training time for a single model (6h), this would take too much time resources given the allotted time to write this paper. So, this idea is not pursued even though its interest is noted.

5.2. Hyperparameter choice

This section focuses on choosing and optimizing hyperparameters for JNet's training. Number of epochs, Learning rate, and Optimization algorithm will be overviewed. hyperparameters such as weight decay and weight initialization will be left as default in the pytorch backend, which already provides generally satisfactory values and algorithms

Learning rate

It is useful to visualize the learning rate for our algorithm before choosing the number of epochs, as the learning rate we settle on will define the number of epochs at which the algorithms' weights start to converge. To do this, we train the algorithm on 100 epochs for various learning rate values. The learning rate is notated α here. we try: $\alpha = 0.1, 0.01, 0.001, 0.0001$

This is common practice and will allow us to zero in on a satisfactory order of magnitude for the learning rate. Below are shown loss and accuracy curves for the train set and validation set over 100 epochs of training with these varying values (Fig. 20-23).

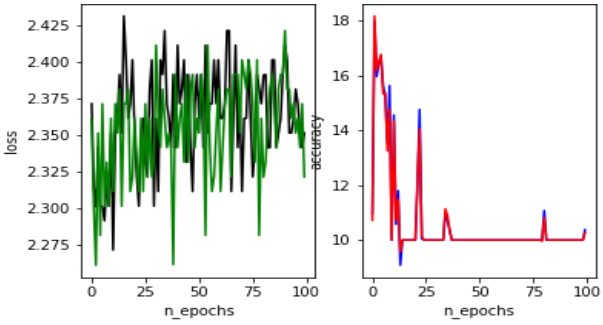

Figure 20: Training curves for $\alpha = 0.1$

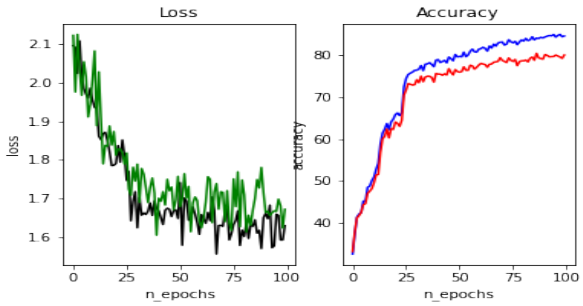

Figure 21: Training curves for $\alpha = 0.01$

while $\alpha = 0.1$ yields a non-converging model, all the other values seem to converge nicely. One caveat we wish to avoid is to converge to a local minima instead of a global minima, and so choose the largest value of α for which the

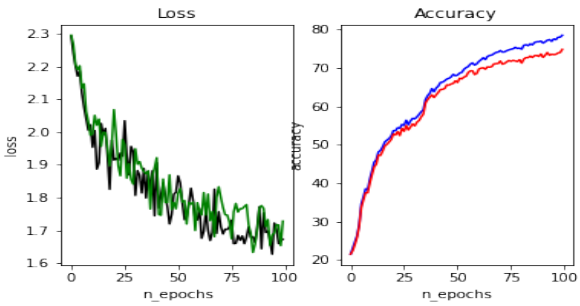

Figure 22: Training curves for $\alpha = 0.001$

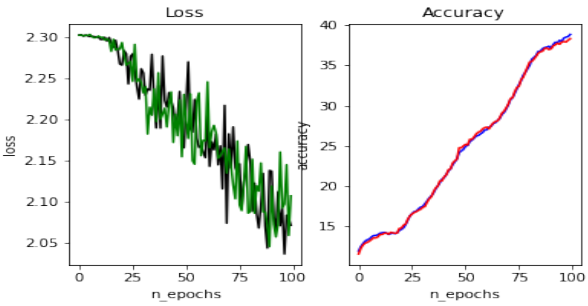

Figure 23: Training curves for $\alpha = 0.0001$

model converges. For the rest of the study, we will use $\alpha = 0.01$. This also gives us an appropriate estimate of the number of epochs.

Number of epochs

We decide upon the appropriate number of epochs given Fig. 27. The model seems to already be converging after 100 epochs but seems like it could be gaining accuracy on the validation set still after 100 epochs. So, we decide to push the training to 200 epochs, and perhaps more if we truly want the model to stabilize and converge later on. so, we get $n_{epochs} = 200$

OPtimization algorithm

The Chosen optimization algorithm is a classically used optimization algorithm for CNN learning, Stochastic gradient descent. We choose to use the given algorithm with **Weight decay** and **momentum**. These are succintly explained below:

- **Weight decay:** The weight decay parameter is the factor λ in front of the regularization term when performing computation of the loss function L with the MSE loss function for instance:

$$L = MSE(y, y_{pred}) + \lambda \sum w_i^2$$

This allows to avoid overfitting in some manner while training

- **Momentum:** Momentum is a algorithm which aids in performing a more reasonable estimate of the gradient when performing gradient descent, by performing weighted averages over the last derivative values obtained. This aids in avoiding local minima by providing "momentum" to the gradient, which avoid ravines and other such caveats.

5.3. Training

The model is trained with Stochastic gradient descent, with momentum set to 0.9, weight decay set to 10^{-4}, and $\alpha = 0.01$, during 200 epochs.

5.3.1 Baseline training curves

A first round of training is conducted to diagnose any potential problems. The training curves are shown below (Fig.24):

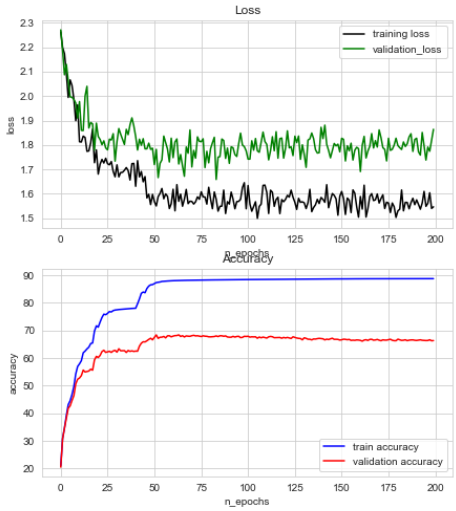

Figure 24: Training curves without regularization

Though the model converges nicely for the training set, there is clearly a very large amount of overfitting happening, as the training accuracy goes up to 100%, while validation accuracy stagnates at 60%. This means that the model is overfitting the training data.

As discussed previously, dropout layers and batch normalization layers are set up to alleviate this issue, as shown in Fig. 20.

5.3.2 Regularization

Training is reconducted for the new regularized architecture, and given the positive results obtained for 200 epochs, the number of epochs was bumped up to 1000 to allow for the model to converge longer. The training cruves are shown Fig. 25 :

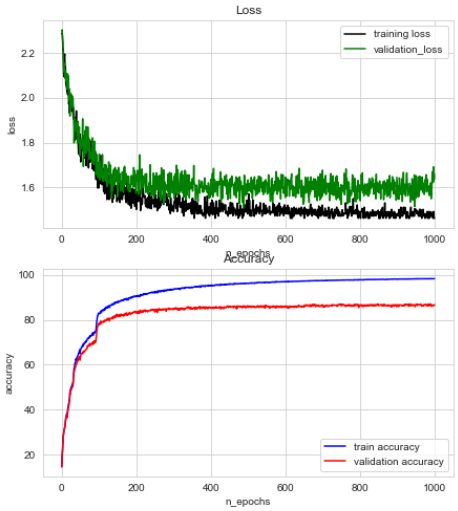

Figure 25: Training curves with regularization

The model now converges nicely for both training and validation, and though the overfitting problem has not entirely been alleviated, it is lesser, as validation accuracy goes up to over 80% with regularization activated.

5.4. Performance analysis

5.4.1 Performance metrics

The problem at hand is a multiclass classification problem. Though it is interesting to look at the global accuracy for the model, it can also be interesting to calculate some metrics by class in a one-against-all fashion to evaluate the proficiency of the model on each class:

The metrics used to evaluate the model on each class will be accuracy (acc), precision (p), recall (r), and F_1 score (F_1). By notating the amount of predicted true positives TP, true negatives TN, false positives FP and false negatives FN, the formulae for these metrics are shown below:

$$acc = \frac{TN + TP}{TN + TP + FN + FP}, \; p = \frac{TP}{FP + TP}$$

$$r = \frac{TP}{TP + FN}, \; F_1 = \frac{2TP}{2TP + FP + FN}$$

Furthermore, a confusion matrix of actual predictions vs given predictions will be computed to better visualize where the model is not a proficient as needed.

5.4.2 CNN performance

Firstly, the main statistic we aim to look at is global accuracy for the model, or how many times its prediction was correct over the whole test set. The following accuracy value is obtained for JNet:

$$acc_{global} = 85.9\% \tag{3}$$

Which is clearly better than chance, as guessing would yield about 10% accuracy. In terms of previous models, JNet is a reasonable contender to AlexNet, as discussed in section **3.4**. Though it pales in comparison to deeper models like VGG16 and the brand-new DenseNet, this is understandable as the size of the images, the allotted time for the report and the computational power required prevented building a more complex model.

The previously discussed metrics for each class are shown in Table 2:

Class	Accuracy	Precision	Recall	F1 score
Plane	88.8	1	88.8	
Car	94.1	1	94.1	
Cat	69.4	1	69.4	
Ship	92.1	1	92.1	
Horse	88.7	1	88.7	
Deer	84.8	1	84.8	
Truck	91.7	1	91.7	
Frog	91.8	1	91.8	
Dog	80.3	1	80.3	
Bird	77.8	1	77.8	

Table 2: Metrics by class for JNet

Some classes clearly perform more poorly than others, with a 20% discrepancy between Cats and Frogs. It might be interesting to see which classes are being misclassified as which classes to better understand why this is the case for future solving. The following confusion matrix shows actual class versus class predicted by JNet, which serves this purpose nicely, in fig. 26:

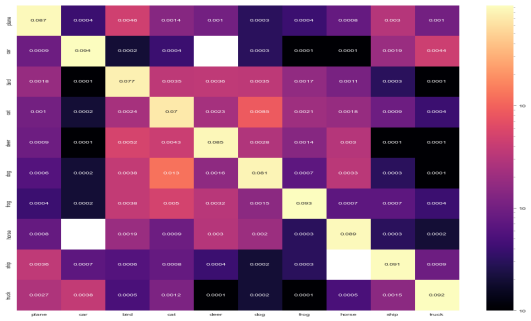

Figure 26: Confusion matrix for class prediction

Among the most notable misclassifications, cats and dogs are often exchanged, as well as cars and trucks. An amusing result to note is that cats are often misclassified as frogs.

5.4.3 Feature extraction and dimension reduction

It is interesting to visualize what the network has learned over the course of training to understand how it classifies images. Though the deeper a network goes, the more its workings are that of a black box, at this level some activation maps of layers for the sake of curiosity are still interpretable. The following figure (Fig. 27) shows 4 activation maps for each of the layers of the network for a random image:

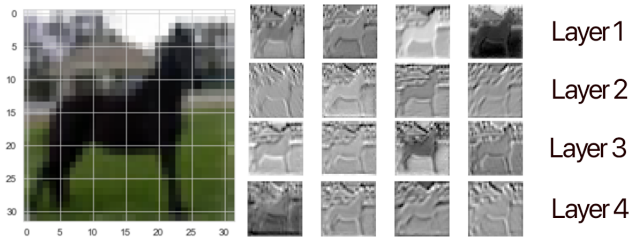

Figure 27: Activation maps for a sample of the horse class

We can also perform dimension reduction on the features learned by the classification block to visualize the separation between classes. In the following figure, t-SNE was applied to all images going through the 64-element fully connected layer of the network, and the features corresponding to each class were visualized in the following chart (Fig. 28)

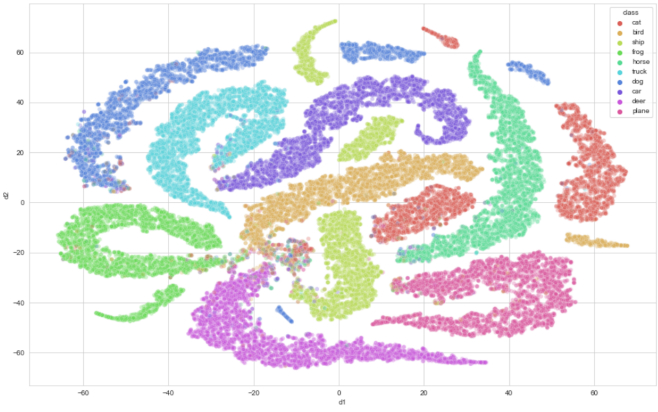

Figure 28: TSNE performed on last-layer features

The clear separation between classes shows that even through dimension reduction, the algorithm has learned co-

herent features which provide the classification layer with good data to distinguish classes.

6. Bonus - Possible improvements

Though the model performs quite well on the CIFAR-10 dataset, it is still very much overfitting to the training split of the Dataset. Indeed, while validation accurecy reaches 85%, training accuracy reaches 100%, which shows that the model is still overfitting.

Though, as discussed previously, K-Fold validation is an interesesting endeavour to pursue for improving upon this basic model, it is extremely time-consuming to train the model even on 5 folds of the data. This section will focus on the description and implementation of a well-known technique to avoid overfitting: Data augmentation

6.1. Data Augmentation

Data augmentation is the method of applying a transformation, many of which exist, to the training data in order to avoid the algorithm getting too accustomed to it. This reduces overfitting by **Presenting more data to the algorithm**, as one can show both the original image and the augmented image, and can even show many augmented images of the same sample, but also **Varying the data presented to the algorithm**. This means that instead of learning the training sample too much, the algorithm will learn some variation of it, which makes it less likely to be too proficient on the training set.

Common augmentations in computer vision

This section will not cover augmentations which consist of mixing two or more images, as described in [9]. We will cover two broad types of data augmentations, without going into specifics about any particular techniques:

- **Geometric augmentations**: These augmentations consist of modifying the spatial distribution of the image. Techniques include **Flipping,rotating,shearing,cropping, translation**. [9]
- **Color space augmenations**: It is also possible to modify the values of the traditional RGB channels of an image. This can be done by modifying **Contrast, color jitter, white balance, hue, saturation**, or by injecting noise.

Implementation of data augmentation for JNet

A total of three augmentation pipelines in the training set were set up for JNet. These were set up using the albumentations package, which allows for randomizing which transform is applied to the image. the pipelines are shown in Fig. 29:

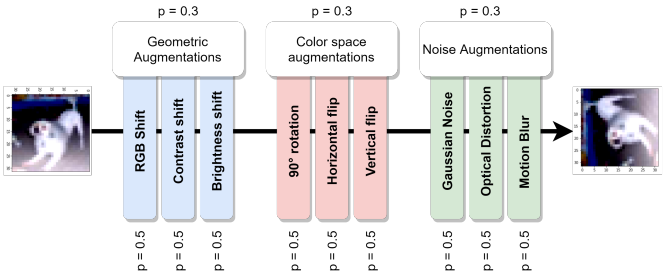

Figure 29: Augmentation pipeline for JNet

JNet was re-trained on the augmented data for 200 epochs and the training curves are shown in Fig. 30

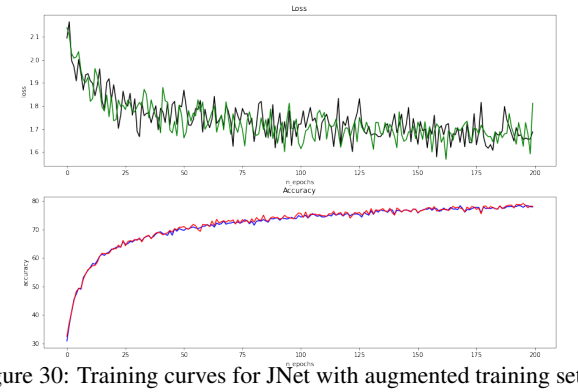

Figure 30: Training curves for JNet with augmented training set

As can be seen on Fig. 30, the overfitting problem is now nonexistent, meaning that data augmentations have served their purpose in aiming to reduce overfitting. Though time does not allow for training 1000 epochs to compare the accuracy with that of Fig. 25, no harsh loss of accuracy can be witnessed, as the model quickly converges to juste under 80% training and validation accuracy.

7. Conclusion

Through this paper, the basics of CNNs were covered in an attempt to create a custom network towards classifying images of the CIFAR-10 dataset. by compromising between complxity and computational requirements, a network was built that achieved comparable results to earlier networks such as VGG16 and AlexNet on the dataset, by achieving 85% of global accuracy on the problem.

List of Figures

List of Tables

References

[1] P. J. Denning and T. G. Lewis, "Exponential laws of computing growth," *Communications of the ACM*, vol. 60, no. 1, pp. 54–65, 2016.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[4] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[5] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[6] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[8] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 6848–6856, 2018.

[9] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, no. 1, pp. 1–48, 2019.

- AlexNet :

  `https://www.kaggle.com/drvaibhavkumar/alexnet-in-pytorch-cifar10-clas-83-test-accuracy`

- VGG16 :

  `https://github.com/kuangliu/pytorch-cifar`

- ResNet50 :

  `https://github.com/kuangliu/pytorch-cifar`

- DenseNet :

  `https://www.kaggle.com/asrsaiteja/densenet-tf-keras-on-cifar-10`

- ShuffleNet :

  `https://github.com/tinyalpha/shuffleNet-cifar10`