LEGO® Brick Classifier

- Image Recognition with CNNs - Available on Github

Maria Emine Nylund
Adrian Kleven

Contents

1 Abstract **2**

2 Introduction **2**

3 Theory **2**
3.1 Convolutional Neural Networks . . . 2
3.1.1 Discrete Convolution . . . 3
3.1.2 Pooling . . . 4
3.1.3 CNN architecture . . . 5
3.1.4 Translational equivariance . . . 6
3.1.5 Translational invariance . . . 6
3.2 Transfer Learning . . . 7

4 Implementation **7**
4.1 Dataset . . . 7
4.2 CNN package . . . 7

5 Results **9**
5.1 NN Model . . . 9
5.2 CNN Model . . . 11
5.3 Transfer Learning . . . 12

6 Discussion **13**
6.1 NN vs. CNN . . . 13
6.2 Failure of Transfer Learning . . . 14
6.3 Data augmentation . . . 14

7 Conclusion **15**

List of Figures

1 Discrete 2D- Convolution . . . 4
2 Pooling of a 2D input feature map . . . 5
3 Convolutional Neural Network . . . 6
4 Kernels trained on ImageNet dataset . . . 7
5 Dataset examples . . . 7
6 Code Structure . . . 8
7 Training Models . . . 9
8 NN validation accuracy . . . 10
9 NN parameter sweep . . . 10
10 CNN validation accuracy . . . 11
11 CNN training accuracy . . . 11
12 CNN parameter sweep, best models . . . 12
13 CNN parameter relative correlation with validation accuracy . . . 12
14 CNN w/ transfer learning validation accuracy . . . 13
15 CNN w/ transfer learning training accuracy . . . 13
16 Edge detection on white noise . . . 14

1 Abstract

Convolutional Neural Networks (CNNs) are a common adaptation to Ordinary Neural Networks (ONNs), especially in the field of image classification and natural language processing, where it has seen immense success [1][2][3]. Using Keras [4], we implemented CNNs and ONNs in classifying images on a limited dataset of rendered LEGO® bricks. We also examined the effect of transfer learning on the training and validation accuracy of the CNNs.
We found that an ONN with 2,510,080 parameters is 92.56% accurate on the validation dataset, while a CNN using 660,000 parameters is 93.62% accurate. Introducing transfer learning, sharing weights from MobileNetV2 [5], increased training accuracy from around 90% to 97% while dropping the validation accuracy to 66.69% for the best model. It also achieved its best model at 10 epochs as opposed to the other CNN's 27 epochs. Not counting network architecture, the parameters that correlated most highly with validation accuracy were the batch size, learning rate and decay rate.
In this particular case, within the range of hyper parameters we examined, it seems that CNNs performed only marginally better than ONNs, while using many times fewer parameters. While transfer learning appeared promising in training the network, we likely need greater specificity in which filters to freeze and which to train.

2 Introduction

The purpose of this article is to explore the theory of CNNs and how they relate to ONNs. We explore how the theory translates to a real world classification problem and in what ways CNNs offer advantages over ONNs in image classification. We also examine the effect of transfer learning using the MobileNetV2 model [5]. For classification, we chose to work with a limited version of a dataset provided by Joost Hazelzet [6], containing rendered images of LEGO® bricks. We used data augmentation to expand the dataset, as it contains only 800 images per class.
We did parameter searches (sweeps) of all the models using the python library wandb [7].The ONNs and CNNs were implemented using the python library Keras [4], an API for the TensorFlow library developed by Google [8].

In the Theory section, we review some of the theory behind CNNs and how they modify ONNs to improve performance in tasks such as image classification. In the Implementation section we cover the overall structure and function of the program that was used to create the models. In the Results section we cover quantitatively the accuracy of a selection of ONNs, CNNs and CNNs using transfer learning. In the Discussion section we discuss the results qualitatively and reason about why we might expect the results we got. Finally, in the Conclusion we discuss the areas for improvement and possible subject matter for further studies.

3 Theory

3.1 Convolutional Neural Networks

ONNs are capable of performing image classification [9], but they underperform in a couple of respects. By flattening the input image to a $N \times 1$- array of N pixels, structural information is lost about how one pixel relates to its neighbours. They cannot trivially classify, spatially translated instances of old training data, since they result in entirely different activations. An ordinary, fully connected neural network also cannot classify separate instances of classes within the same input image since all neurons in the input layer are fully connected with the first hidden layer.
There is also the problem of scalability. For a $n \times m$ image with 3 color channels, a fully connected neural network would require $n \cdot m \cdot 3$ weights for the input layer alone. For a reasonable pixel resolution, say 256 by 256 pixels, that equates to a parameter space of over 196,000 parameters, untenable to most desktop computer owners. This also inevitably results in overfitting, as the system is easily overparameterized.

CNNs address these problems and add extra functionality by introducing a few new ideas:

1. Preserve the underlying structure of the data by employing discrete convolutions to the input, rather than the affine transformation used in ONNs (matrix- vector multiplication).

2. Exploit the full feature space by adding layer depth to the network.

3. Exploit strong local correlations in the input to reduce the total parameter space of the model.

3.1.1 Discrete Convolution

We introduce multidimensional discrete convolution by considering a particularly simple example; 2D - Convolution. We consider a matrix, called the kernel (K) and another matrix called the input feature map (I). For a $n \times m$ image with 3 color channels, the input feature map would be a $n \times m \times 3$ matrix of pixel values. We will consider 1 color channel and a 5×5 grid of pixels with small integer values for simplicity. The kernel represents our choice of weights, each of which can be applied to multiple pixel values in the input. The kernel would usually be smaller than the input feature map, lest we end up with a fully connected layer.

$$I = \begin{bmatrix} 3 & 3 & 2 & 1 & 0 \\ 0 & 0 & 1 & 3 & 1 \\ 3 & 1 & 2 & 2 & 3 \\ 2 & 0 & 0 & 2 & 2 \\ 2 & 0 & 0 & 0 & 1 \end{bmatrix}, \; K = \begin{bmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix} \tag{1}$$

It is pertinent now, before proceeding, to mention two more choices for the discrete convolution. Namely stride length and zero padding. When performing the convolution we have the choice of adding rows and columns of zeros to the perimeter of the input feature map. This is known as zero padding. When the kernel is panned across this new matrix, we also have a choice of how many rows and columns to shift the kernel by after each operation. This is known as stride length. We will consider the example of stride length 1 vertically and horizontally and no zero padding.

We denote the submatrices of I by $I_{[o_m:k_m-1+o_m,\, o_n:k_n-1+o_n]}$, where k_m and k_n are the number of rows and columns of K, respectively. o_m and o_n are positive integers. The matrix elements of the output feature map is given by

$$O_{[s_m,s_n]} = \mathbf{e}^T \left(I_{[o_m:k_m-1+o_m,\, o_n:k_n-1+o_n]} \circ K \right) \mathbf{e} \tag{2}$$

where $\mathbf{e}$ is the $k_m \times 1$ one- vector and $\circ$ is the Hadamard product between matrices.
A more intuitive approach is seen in figure 1 where we take the Hadamard product of the two overlapping regions and sum all the resulting matrix elements to calculate the corresponding region in the output feature map.

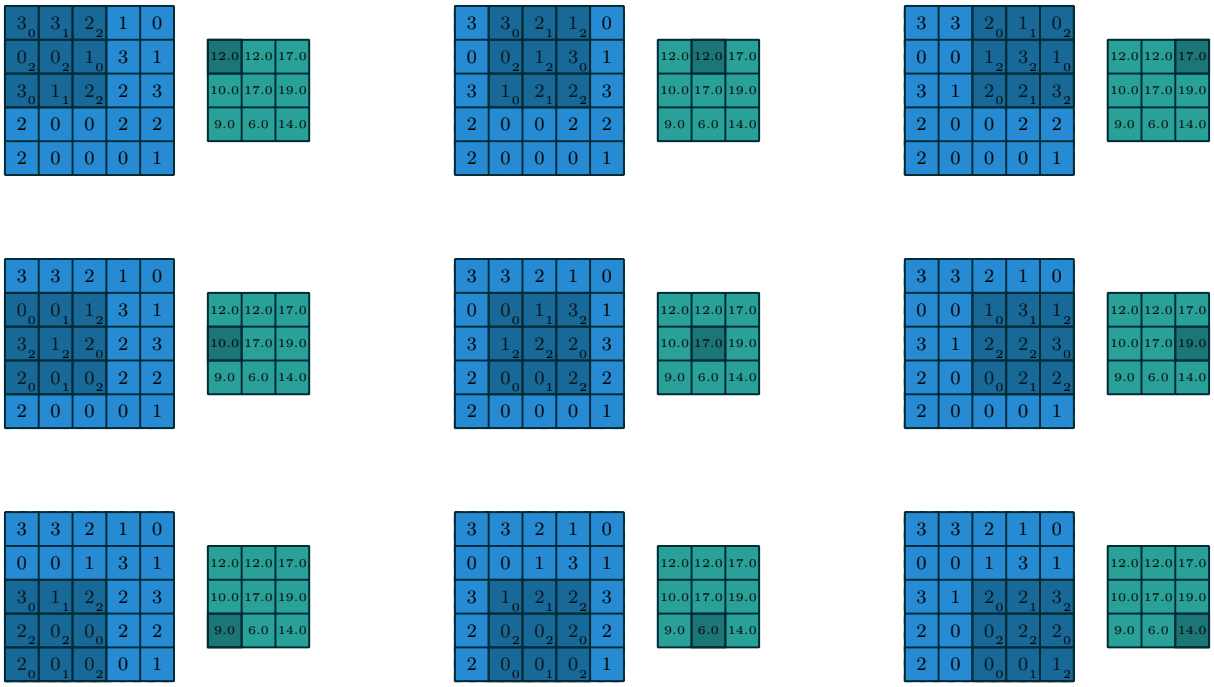


Figure 1: The kernel (shaded region) covers sections of the input feature map (blue region) in turn, calculating the regions of the output feature map (green region). Using no zero padding and a stride length of 1 in each direction. Image credit to Vincent Dumoulin and Francesco Visin [10]

This same procedure can be repeated with multiple different kernels to produce a range of varying output feature maps. Each of the output feature maps can then be stacked together and treated as a single output to the succeeding layer.

3.1.2 Pooling

Pooling is a way of downsampling the input feature space into a roughly representative set of features [10]. In a CNN, this amounts to performing some operation on the output feature map in the preceding layer. This is done in a similar manner to the discrete convolution, except to generate the output feature map, we apply the average or maximum, or other similar operation to the overlapping region. Depending on the size and shape of our input, and the size and stride length of our sampling region, we get different outputs. One example of such a downsampling is illustrated in figure 2.

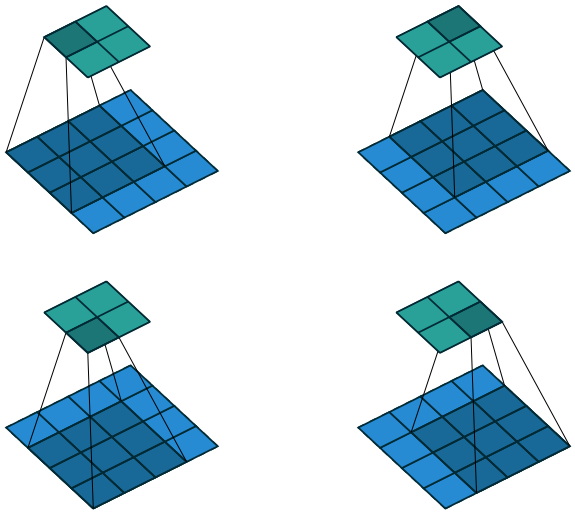


Figure 2: The shaded elements in the input feature map are passed through a $\mathbb{R}^9 \to \mathbb{R}$ function, generating the corresponding region of the output (green region). Using a stride length of 1 in each direction. Image credit to Vincent Dumoulin and Francesco Visin [10]

This process is repeated for every input feature map produced in the preceding layer. The outputs are stacked as in the inputs, meaning the width and height of the data is downsized, but the depth remains the same.

3.1.3 CNN architecture

A general CNN architecture uses several different layers, their ordering and numbers differing between models. These are

1. **Convolution Layer**- Performs convolution on local regions in each input feature map using weights contained in the kernels. Each kernel, the depth of the input feature map, contain separate weights and can resolve different features in the input.

2. **Activation Layer**- Applies an element-wise activation function to the input, producing an output of the same size. A common choice is the ReLU- activation function [11].

3. **Pooling Layer**- Downsamples the input along the width and height, while maintaining the depth.

4. **Fully connected Layer**- Performs classification and connects to the output layer, giving the scores for each class.

Figure 3 gives one example of a such CNN.

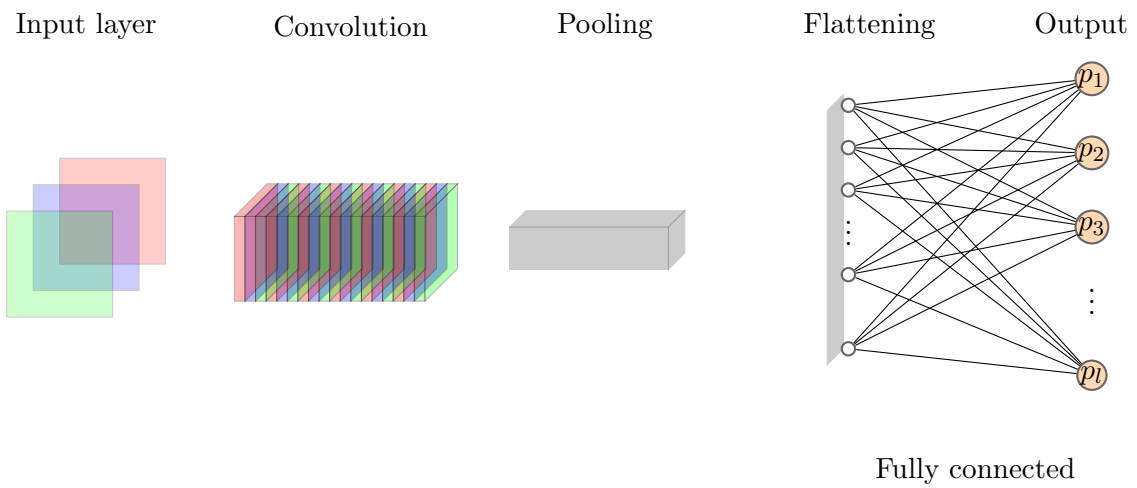


Figure 3: CNN with three color channels,a single convolution layer with 6 kernels of depth 3, one for each color. A single pooling layer, followed by a flattening procedure and a fully connected hidden layer, connected to the output which is passed through the SoftMax function. The output consists of a score for each of the l classes.

Other CNNs can have several convolution- and pooling layers, all interwoven in between the input and fully connected layers. Activation layers may be added between any such layers.

Introducing convolution to the network has clearly reduced the parameter space dramatically. A relatively small set of weights and biases contained in each kernel are shared across much of the input. These need to be trained in addition to the weights and biases contained in the fully connected layers. We may have shrunken the parameter space, but have simultaneously introduced several new hyperparameters that will ultimately impact the success of our models. These are

1. Different network architectures.
2. The size, shape and number of kernels, stride and zero padding. Also what's called dilated convolution [12], though we will not address this further.
3. Pooling function and the degree of downsampling.
4. Activation function in the fully connected layers.
5. Choice of optimizer.
6. Choice of cost function.

3.1.4 Translational equivariance

A function $f : X \to Y$ is said to be equivariant to a function g if for all $x \in X$, $f(g(x)) = g(f(x))$ [13]. If g corresponds to some translation in the input space, applying the convolution to the output of g is the same as performing the convolution on the original input, then applying the function g to its output. The upshot is that translating an object in an image gives the same output as the original image, only translated. The convolution achieves this by having shared weights across multiple regions in the input space. This means that an object will be picked up regardless of where it is in the image. Additionally, multiple instances of the same object are classified separately, rather than trying to resolve it as a separate class, say the "3 antilopes- class".

3.1.5 Translational invariance

The translation in the output is resolved in the pooling layer. Because pooling serves to summarize the contents of entire regions in the input space, the relative positions of objects within that region

end up producing the same output. In this way, translated objects produce outputs similar to their non- translated counterparts.

3.2 Transfer Learning

Transfer learning is a technique that leverages learned behaviours from a trained model, performing similar tasks, to improve learning for a new task [14]. This is possible because the features that the kernels (filters) closest to the input end up learning, are often similar across different classification problems [15]. A filter might learn to identify edges oriented a certain way, or splotches of color. Features that are easily applicable across different datasets. An example of such filters is seen in figure 4.

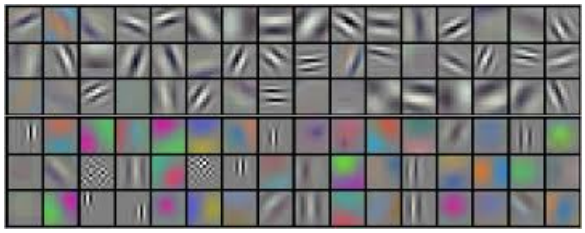


Figure 4: 96 convolutional kernels of size $11 \times 11 \times 3$ learned by the first convolutional layer on the $224 \times 224 \times 3$ input images in ImageNet[16]. Image credit to Krizhevsky et al [17].

4 Implementation

4.1 Dataset

In order to test the viability of our models to classify LEGO® bricks, we chose to work with a limited data set. The Kaggle web-site has a fitting dataset where LEGO® bricks were generated by using Maya [18], created by Joost Hazelzet [6]. For ease of reading we'll refer to the different LEGO® bricks, simply as classes. The original data set has a two- camera set up, where one data point consists of the same class captured from two angles. The reason for this is that some classes look indistinguishable from certain angles. Because we worked with a limited number of classes, we opted for only one of the camera angles. We also chose to work with grey scale images, as we wanted the networks to be trained on the shape of the classes, not by detecting splotches of color. A small example set of eight different classes are shown in figure 5.

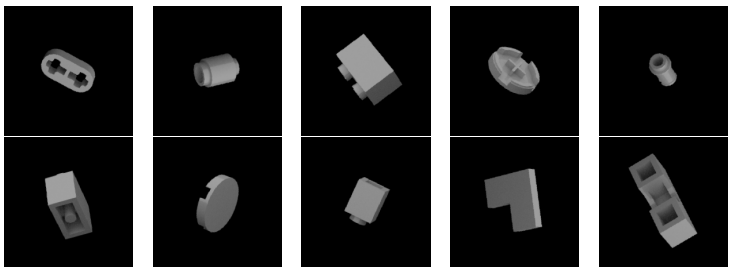


Figure 5: Example images from the dataset [6]. The models are trained to classify 10 different objects.

One thing to note about this dataset, is that the same class can look vastly different depending on its orientation. Also as we noted, because we chose the one- camera setup, different classes may look similar depending on the angle. This may be a contributing factor in the difficulty of training a neural network on this dataset.

4.2 CNN package

The code base for this project mostly lies in the CNN package. It consists of dataloader, trainer and model python files. See the diagram below showing their relationship and most important functions

and variables.

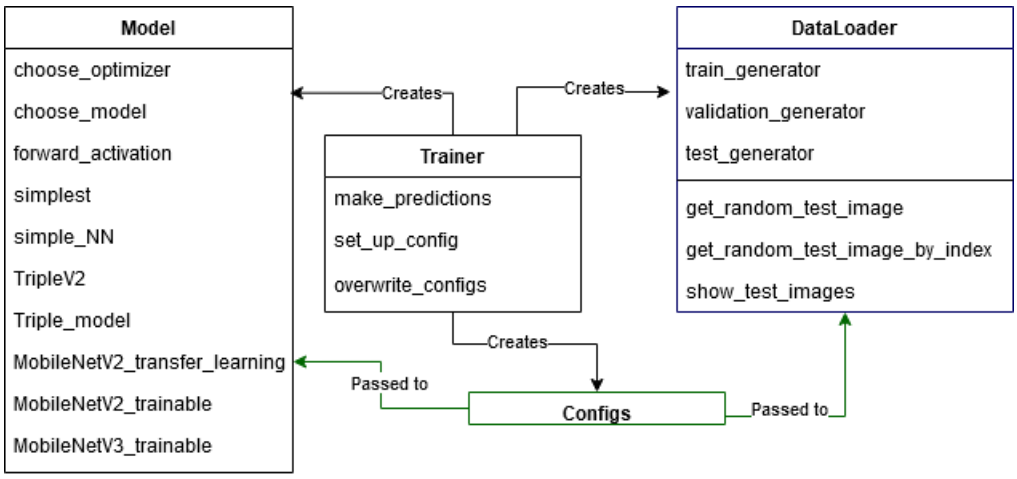


Figure 6: Diagram of classes used for training. Configs is generated using Wandb [7].

For implementation of the networks, we have chosen to use Keras [4], an API for the TensorFlow library developed by Google [8]. It allows for an easy implementation of model architectures, weight decay and training loop. It contains pre-trained models, that are ready to use for transfer learning. It also works well with the Wandb [7] package that keeps track of the parameters and creates visualisations based on the results on their web-site. This package also allows for parameter searches for the models. This is done by creating a .yaml file containing the names of parameters and their constraints. Then one uploads that file to set up the search (called sweep on their web-site). For it to work, one needs to use the configuration class in the code, you can see that it is passed to other classes in the diagram 6 above.

Trainer

Trainer is a class written by us, it creates the configurations, dataloader and model classes. Then initiates the Wandb [7] callback to keep track of results during training. For the training itself, it calls fit function on the model with the data from the dataloader class. In the end, it makes predictions on the test dataset and uploads the images to the Wandb [7] web-site.

Dataloader

First we have downloaded the dataset and programatically divided it into train, validation and test folders. Each folder is subdivided into one folder of images for each class. We have chosen 10 different classes out of 50 to lessen the scope of the project. There are only 800 images per class.
Since this is a very limited dataset, we have used data augmentation to expand the number of training examples available to the model. This was implemented using the Keras ImageDataGenerator class using shear and flip transformations on the original dataset. We have also normalised the images by dividing it by 255. Dataloader takes in the configuration class, where we can specify the number of channels we need to upload, if it is 1, then images are loaded greyscaled otherwise they are loaded as rgb. Train, test and validation datasets are each saved as a variable in the class. The file also contains some helper functions for easier visualisation of the images.

Model

Model class contains the different neural networks wrapped in their own functions. Parameters are set from the configuration class passed on from the trainer. Now I will briefly explain each implemented model:

- *Simplest*: Contains only one hidden layer with one neuron. This was used for testing of the code.

- *Simple_NN*: Simple neural network with four hidden layers, each using ReLU as its activation function. Used to compare with the CNN
- *Triple_model*: First version of CNN created by us, contains filters of sizes 7, 5, 3 and 1. Max pooling layer comes after each convolutional layer. It contains dropout and dense layers at the model of the model. It used ReLU as activation function for all layers. See its architecture in the figure 7.
- *TripleV2*: Second version of CNN, mostly the same architecture, uses tanh as its activation function.
- *MobileNetV2_transfer_learning*: Transfer learning using a pre-trained MobileNetV2 model. Only bottom dense layers are trainable.
- *MobileNetV2_trainable*: Tested out if it works better if some layers are re-trained.
- *MobileNetV3_trainable*: Tested if only BatchNormalization layers are trainable.

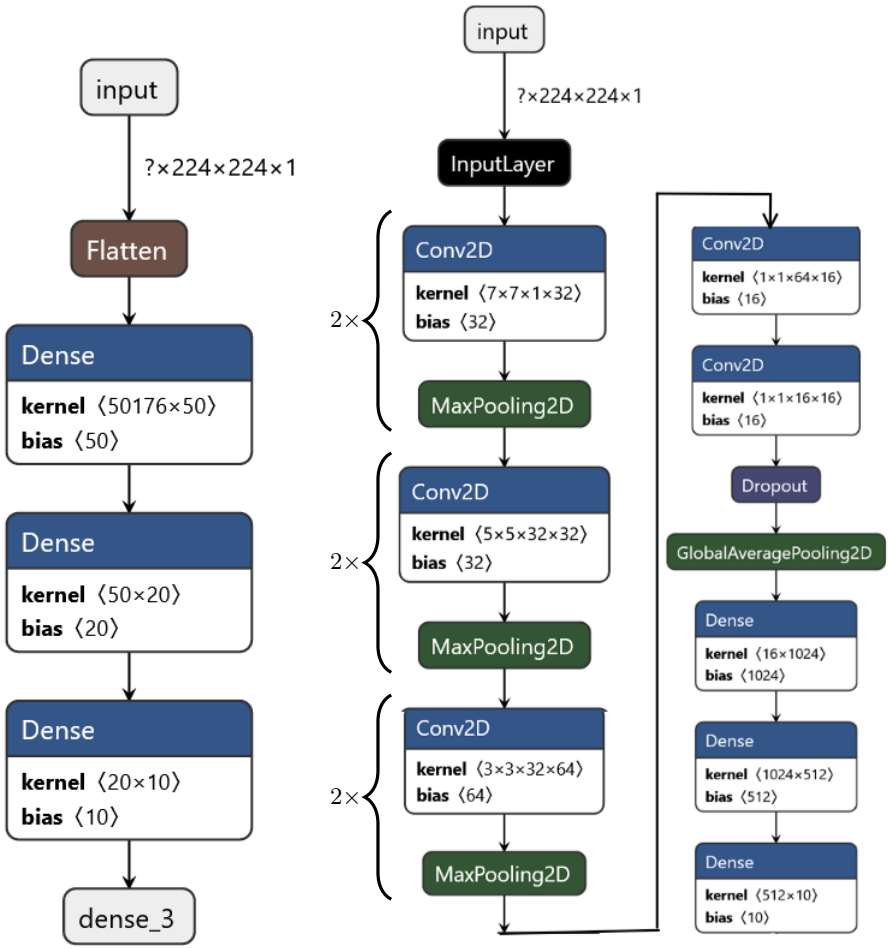


Figure 7: To the left simple NN and to the right CNN. The CNNs have alternating series of a Conv2D layer followed by a MaxPooling2D layer. Generated using Wandb [7].

5 Results

5.1 NN Model

Simple_NN_V2, a simple fully connected neural network using 4 hidden layers with 512, 400, 200 and 50 neurons from input to output. In all, the model has nearly 2,510,080 parameters. Figure 8 below shows the maximum, minimum and mean validation accuracy from a sweep of 4 different models over activation functions, batch sizes and learning rates.

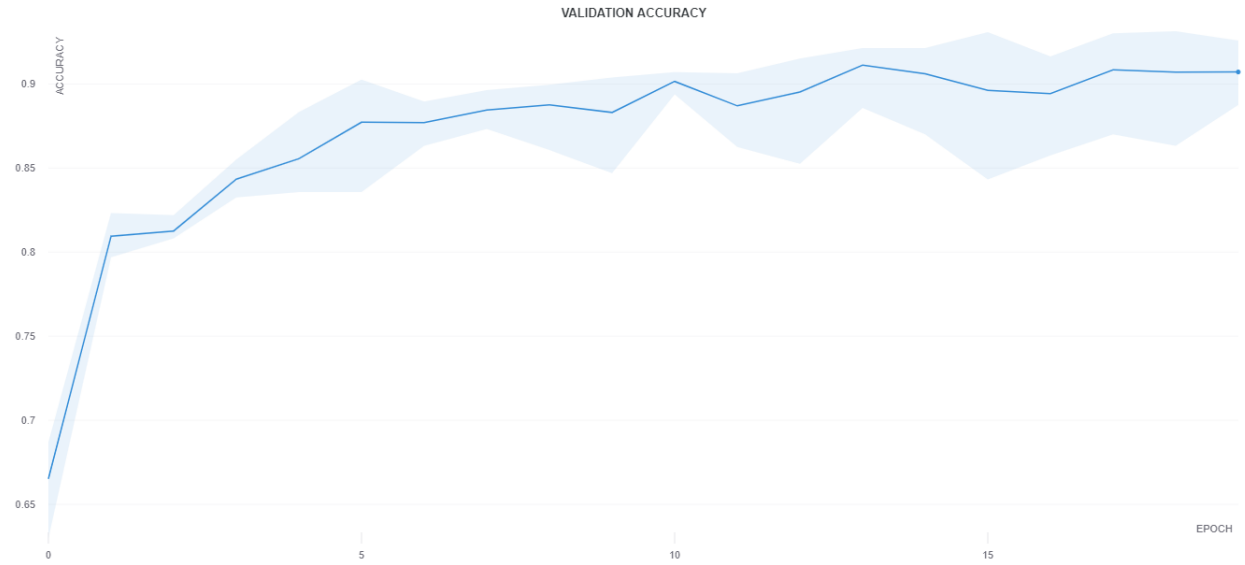


Figure 8: Validation accuracy of multiple NNs shown in figure 9 as a function of the epoch number. Dark blue line indicates the mean accuracy of all four models. Light blue regions indicates the range of the worst- to best- performing models. Generated using Wandb [7].

The best model had 92.56% validation accuracy at 19 epochs. The hyper parameters of this model and the three others are shown in figure 9.

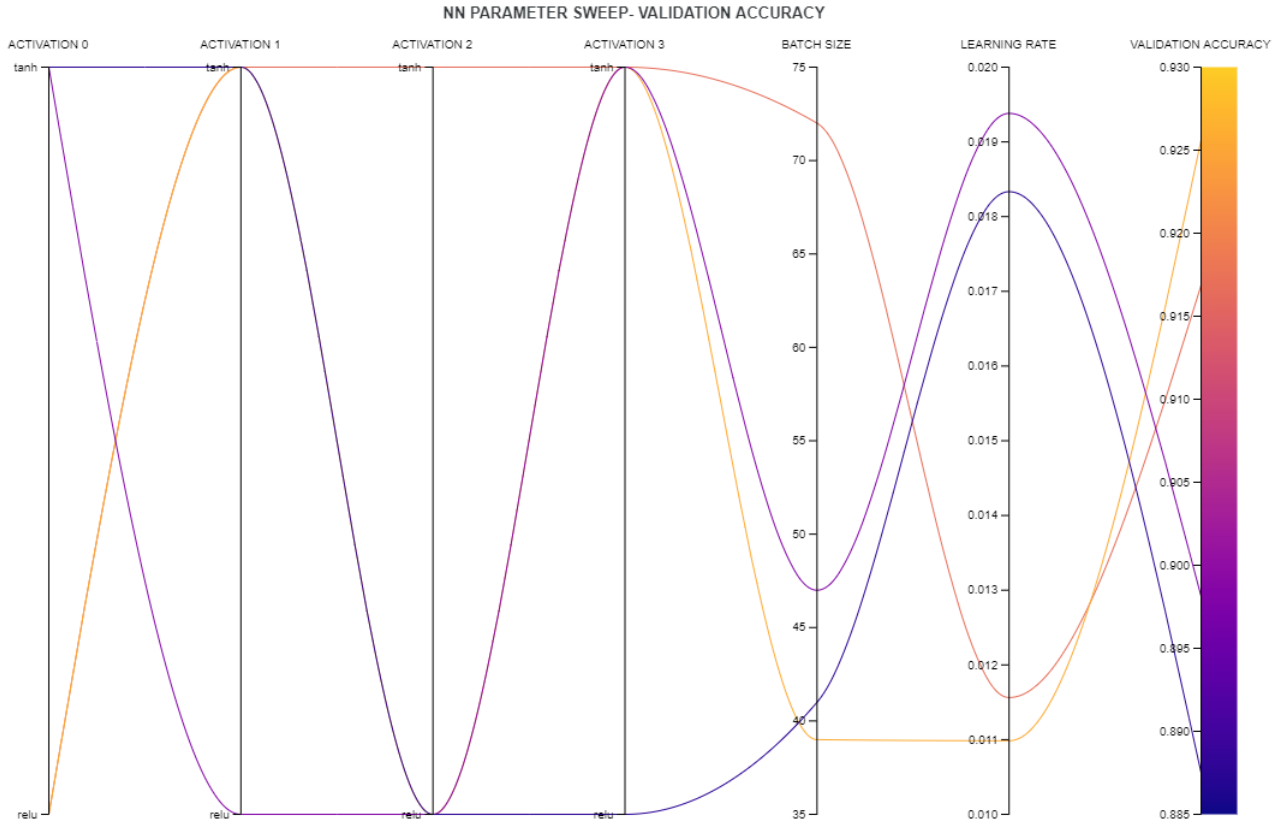


Figure 9: Sweep of four NN models. Validation accuracy determined for four different models, varying in activation functions, batch sizes and learning rates. Generated using Wandb [7].

There are not enough models to make a statistically sound inference from this graph, but a lower learning rate seems to correspond to higher accuracy. We also note that the two best performing models are equal in all respects except for batch size and slightly w.r.t. learning rate. Here, the lowest batch size corresponds to a better accuracy.

5.2 CNN Model

TrippleV2, the best performing CNN we found, follows the network architecture found in figure 6. In all, the model has nearly 660,000 parameters, wherein almost 550,000 of them are used to parameterize the fully connected layers. Figure 10 below shows the performance range of 54 different CNNs trained using varying hyperparameters. These are batch size, dacay rate (related to the decay of the learning rate), the number of decay steps and the initial learning rate.

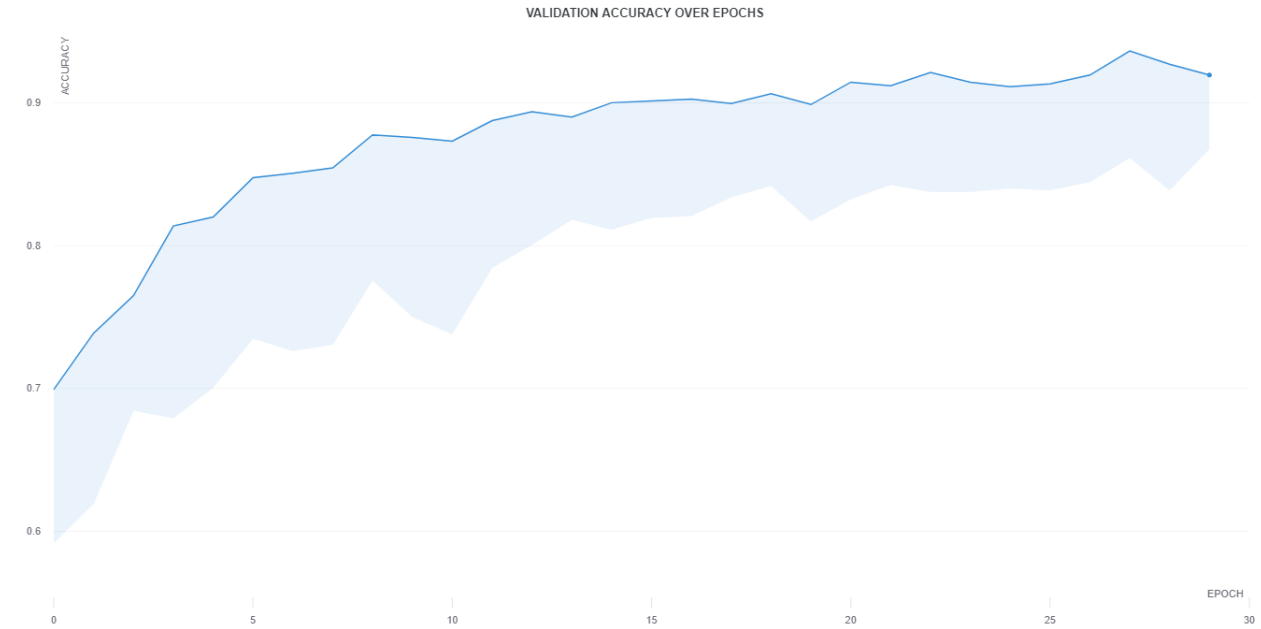


Figure 10: Validation accuracy of 54 CNNs as a function of the epoch number. Dark blue line indicates the accuracy of the best model. Light blue regions indicates the range of the worst- to best-performing models. All the models share the same network architecture, found in figure 6. Generated using Wandb [7].

The best CNN model accomplished 93.62% accuracy at 27 epochs and 91.94% accuracy at 29 epochs. The hyper parameters of this model and a small subset of the best performing models are shown in figure 12. Figure 11 below shows the training accuracy of the same models.

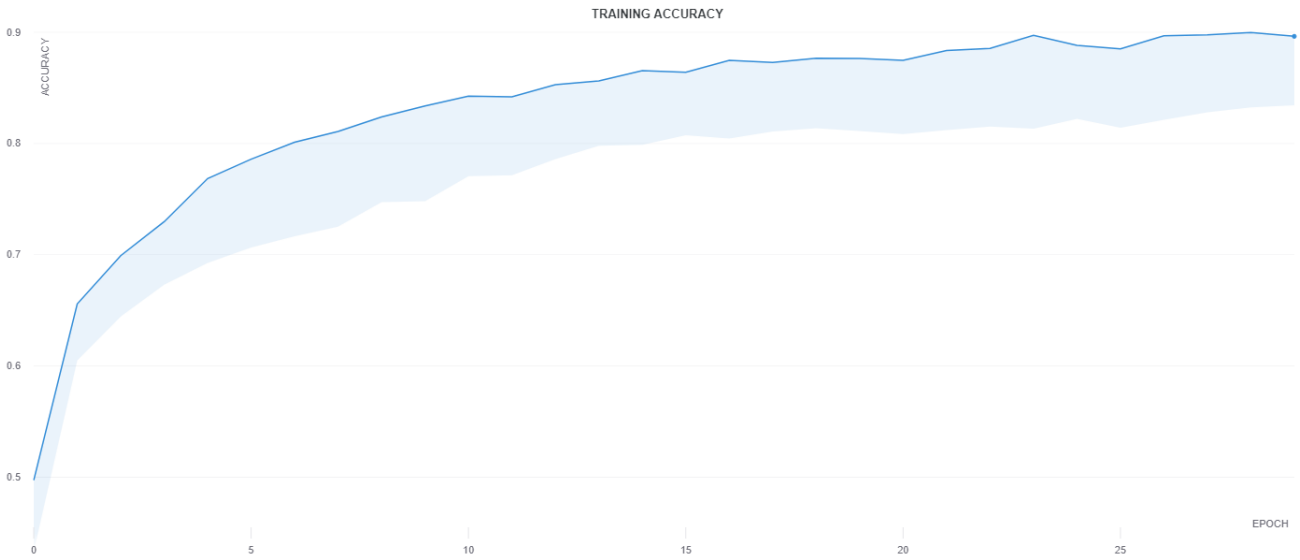


Figure 11: Training accuracy of 54 CNNs as a function of the epoch number. Dark blue line indicates the accuracy of the best model. Light blue regions indicates the range of the worst- to best- performing models. Generated using Wandb [7].

We see from this that the best model reaches about 90% training accuracy after about 25 epochs. This will become notable later when evaluating the success of transfer learning.

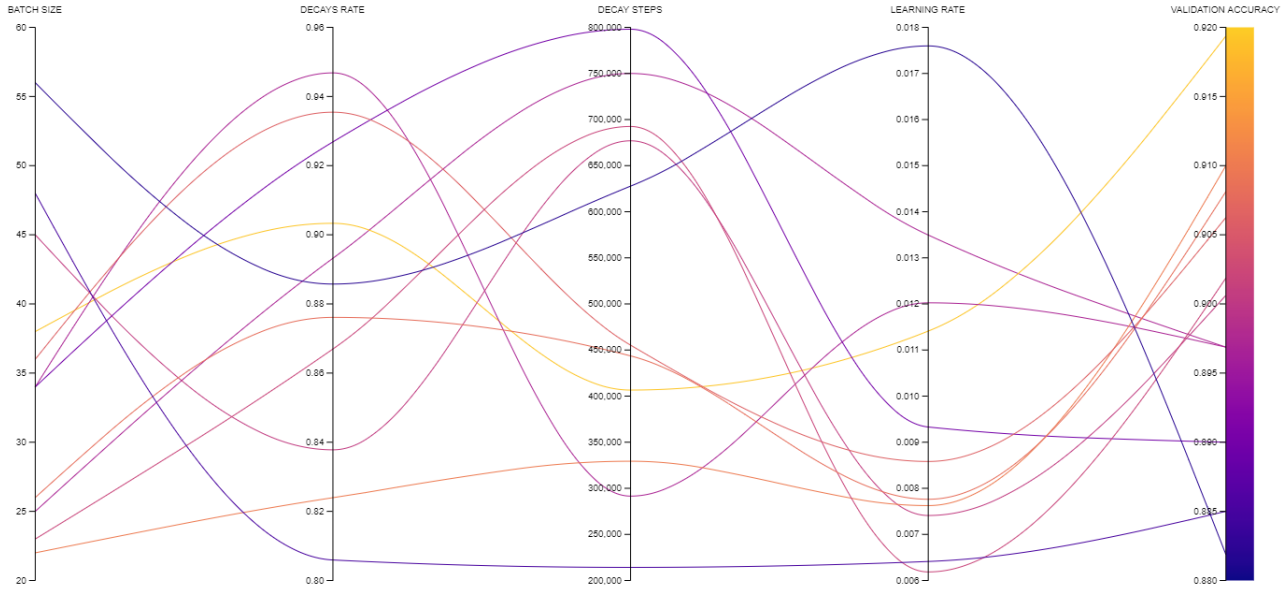


Figure 12: Best models after 54 Sweeps of different CNN models. All models follow the architecture shown in figure 6. Their other parameters are the same as in figure 10. Generated using Wandb [7].

It is more difficult here to spot the correlations among these 10 models. Luckily, 54 models are somewhat better for doing statistical analysis, although it is still not great. We see in figure 13, the three hyper parameters that are most highly correlated (in magnitude) with the the validation accuracy.

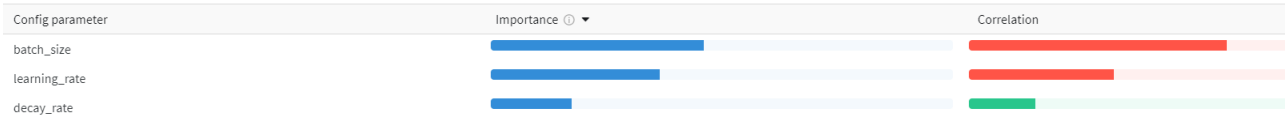


Figure 13: Relative parameter correlation with validation accuracy using 54 Sweeps of different CNN models. All models follow the architecture shown in figure 6. Generated using Wandb [7].

We can see that batch size has a relatively strong negative correlation with the validation accuracy. After that, the learning rate is also negatively correlated with the validation accuracy. The decay rate has a positive correlation with the validation accuracy.

5.3 Transfer Learning

MobileNetV2 (so named from the model it takes parameters from) is our best performing CNN that uses transfer learning. The parameters are taken from the MobileNetV2 CNN [5]. Figure 14 below shows the performance range of 18 CNNs using different hyper parameters. In addition to the usual hyper parameters (see figure 12), we have also varied the the number of layers that are frozen (meaning they do not undergo training, but retain their initial values from the source model).

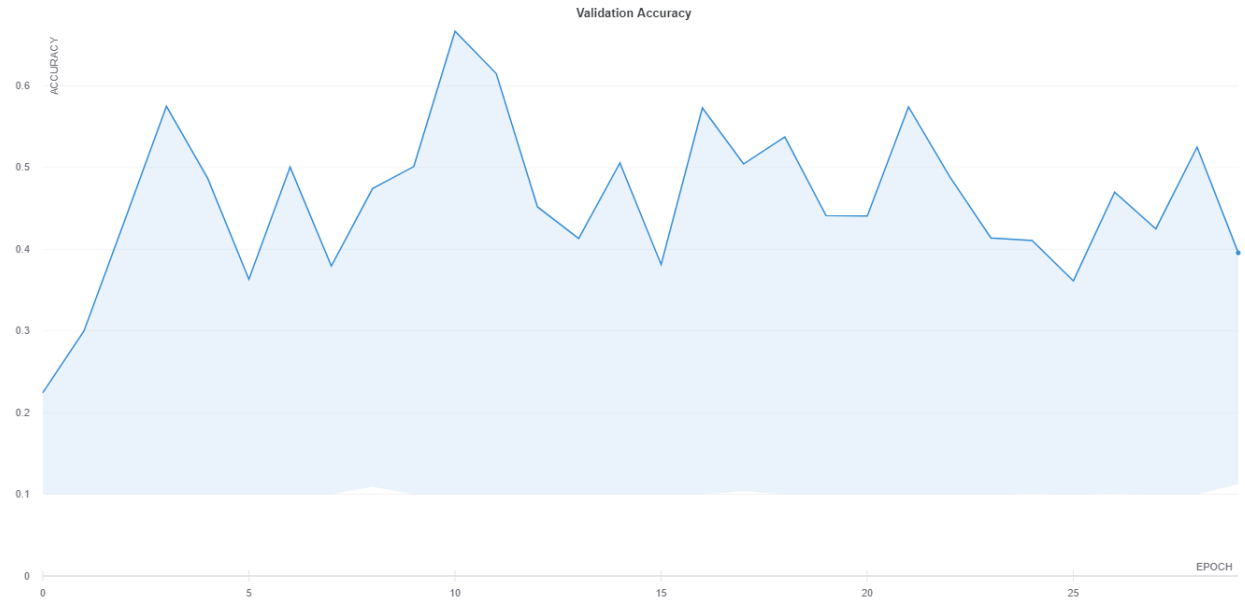


Figure 14: Validation accuracy of 18 CNNs using transfer learning, as a function of the epoch number. The dark blue line indicates the best validation accuracy achieved by any model at that epoch number. The light blue regions indicates the range of the worst- to best- performing models. Generated using Wandb [7].

These results are markedly different from the other CNN. The best accuracy is achieved at 10 epochs with 66.69%. We also see models that have accuracy barely above 10%. Figure 15 below shows the training accuracy of the same models as a function of the number of epochs.

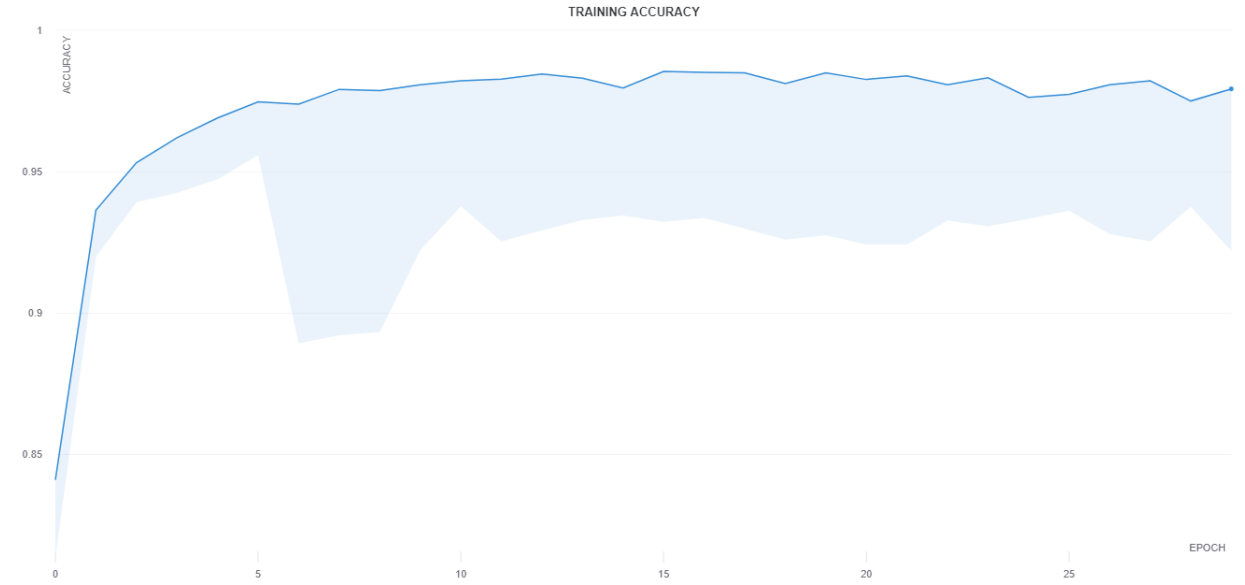


Figure 15: Training accuracy of 18 CNNs using transfer learning, as a function of the epoch number. The dark blue line indicates the best accuracy achieved by any model at that epoch number. The light blue regions indicates the range of the worst- to best- training accuracies. Generated using Wandb [7].

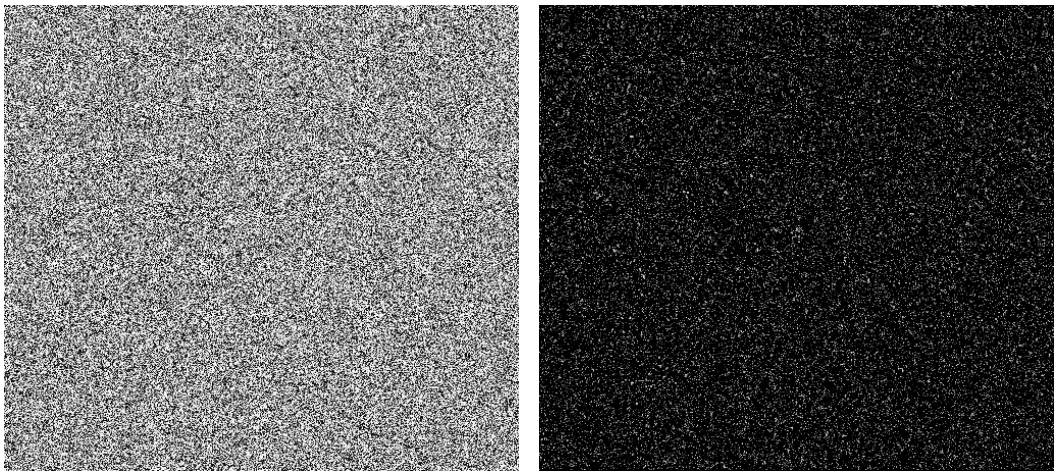
6 Discussion

6.1 NN vs. CNN

One result that became particularly clear, was that switching to a CNN did not improve the validation accuracy. The best CNN accomplished 93.62% accuracy, compared to 92.56% for the best NN. Where

the CNN excelled, was in reducing the the number of parameters to 660,000 from 2,510,080.
Having had high expectations of CNNs, this result was somewhat startling, so it is worth examining why we might expect such a result.

In the Theory section we mentioned some of the aspects that make CNNs superior in image classification. How do these apply here? All the classes in the data set are centered and surrounded by black pixels. There is also ever only one model per data point. Because of this, we should not expect translational equivariance/invariance to play a role in the success of any model. There is also the matter of the background. We might have seen an advantage in using convolutions if the classes were partly occluded or the background noisy. This would presumably impact CNNs less as certain kernels (such as the edge detecting Sobel- Feldman [19]) will tend to give zero activation in regions with randomly distributed noise.



(a) White noise image (b) Sobel filter applied

Figure 16: Effect of applying a Sobel filter to white noise

6.2 Failure of Transfer Learning

Transfer learning had an interesting effect on the CNNs. Every model that utilized transfer learning achieved a better training accuracy in one epoch than the best CNN could achieve in 25 epochs. This is partly expected. We expect that some of the pre- trained filters will be directly applicable to our data set, such as edge detection. These would otherwise take time to train.
The MobileNetV2 model [5] from which we got the weights, was trained on the ImageNet dataset [16], which contains color images. Inevitably then, some of the filters will be trained to recognize splotches of color. These are not expected to contribute to training our models, since the dataset is entirely greyscale.
We would hope then, that while the model retains the edge detecting filters, it repurposes the color detecting ones. Due to the way the weight sharing is implemented, this is not necessarily a trivial task. In the hyper parameter sweep, certain weight are either frozen or learnable. If a set of useless filters are frozen for a model, then we are under-utilizing our parameter space.

Even though this method achieved great learning accuracy, for most models at most epochs the validation accuracy was far below the other CNNs. This is indicative of overfitting and might be due to over- parameterization. These models ended up with approximately 3,500,000 parameters, which is many times more then the CNNs not using transfer learning.

6.3 Data augmentation

As is, the dataset is very limited with only 800 images per model. This is a strong contributing factor leading to overfitting. We implemented a few methods for data augmentation, such as shear

and flip transformation. There were however opportunity for other data augmentation that were left unexplored due to being somewhat more difficult to implement. These are scaling and translation. Because the classes vary so much in size, we would have to pad the input images with a lot of black pixels to be able to meaningfully translate or scale the bigger classes without clipping.

7 Conclusion

Preliminarily, we can judge CNNs to have been largely successful in this classification problem. While performing similarly to ONNs, they were considerably easier to train. As only a tiny subset of the hyper parameters were explored in the course of this project, it is hard to draw any firm conclusions. Using a few different, fixed architectures, we chose to mainly focus on the hyper parameters that correlated most highly with validation accuracy. Using those as a guideline for optimizing the best overall performing architecture. Transfer learning proved to be valuable in speeding up training, but ultimately overfitted this small dataset.

This project was ultimately limited in scope due to a lack of computational resources. As is, there are several avenues that warrant exploration.

- **K- fold cross validation** to make our results more statistically sound and to decrease overfitting.
- **More hyper parameter sweeps** would give a better chance at finding better models, while also providing better analysis for parameter correlations.
- **Testing different CNN architectures** and their effect on training and validation accuracy.
- **Experimenting with different kernel sizes** to possibly influence the type and size of features the model captures.
- **Using only the first layers in transfer learning.** Only the first layers in an image classifying CNN are completely transferable, as these filters are broadly applicable to image classification [15]. The next step would be to identify the filters that pick up of colored features and exclude them or make them available for training.
- **Expanding the dataset** to include more classes and use the two- camera setup to increase accuracy. This could also include increasing the resolution of the images.
- **Training the regular CNN on more epochs with learning rate decay.** The best model did not seem to overfit the data at the number of epochs we attempted, so more epochs might give a better accuracy.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[2] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[3] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[4] François Chollet et al. Keras. https://keras.io, 2015.

[5] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

[6] Joost Hazelzet. Images of lego bricks dataset, 2019.

[7] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[9] A. Kleven and S. Schrader. Project 2 in fys-stk4155. 2020.

[10] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *ArXiv e-prints*, mar 2016.

[11] Morten Hjorth-Jensen. Data analysis and machine learning: Lecture notes. 2020.

[12] Xiaohu Zhang, Yuexian Zou, and Wei Shi. Dilated convolution neural network with leakyrelu for environmental sound classification. In *2017 22nd International Conference on Digital Signal Processing (DSP)*, pages 1–5. IEEE, 2017.

[13] Andrew M Pitts. *Nominal sets: Names and symmetry in computer science*. Cambridge University Press, 2013.

[14] Lisa Torrey and Jude Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI global, 2010.

[15] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks?, 2014.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[18] Autodesk Inc. Maya: 3d computer animation, modeling, simulation, and rendering software, 2007. Software available from https://www.autodesk.com/products/maya.

[19] Irwin Sobel, R Duda, P Hart, and John Wiley. Sobel-feldman operator.