Environment Recognition

NCML Project

David Håkansson, Alejandro Villaron, Laleh Aftabi

April 2021

1 Introduction

Currently, one of the most popular solutions when it comes to image analysis is the use of convolutional neural networks (CNN) and their variants, such as Fast Region-CNN, Region-based Fully Convolutional Network or RESidual Networks [10]. While it is true that image analysis through these types of networks can have as many applications as one can imagine, this report is not focused that much on the application, but on the study of the implementation itself.

To do so, a data-set of more than 20.000 images of 150x150 pixels will be fed to the network. These images belong to one out of six categories of different landscapes (buildings','forest', 'glacier', 'mountain', 'sea', 'street').

The aim of this report to understand and implement CNN and transfer learning models on image classification problems.

The network and the algorithms necessary to carry away the experiments will be developed in Python. In addition, the libraries Keras and Tensorflow will be used due to their optimization and ease of use for the human development of deep learning algorithms.

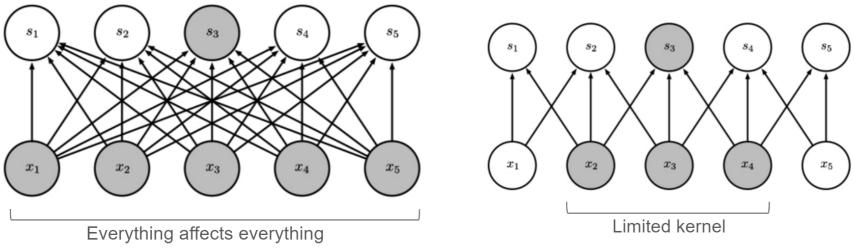
2 Background

2.1 CNN

Artificial Neural Networks(ANN) are the most common solutions to almost all machine learning problems. However, they are not the best when dealing with a large input such as images. For instance, for a simple low-quality image of 64*64 in RGB format, 12288 nodes are just needed for the input(64*64*3). Approaching image data with ANNs and just more than 12000 for the input nodes can lead to more than a billion nodes when dealing with a better quality input and the required hidden layers to reach the desired output. ANNs will require a huge amount of storage usage and a huge amount of time to train the model. Convolutional Neural Networks(CNNs) are the architecture developed to overcome the difficulties faced by ANNs. CNNs have three main properties that make them a good option when having large size input data.

- Sparse Connectivity
- Parameter Sharing
- Translation invariance

Sparse Connectivity is the criteria in CNNs that solves the problem which is mentioned regarding the storage and node connections. As there is no need that all the outputs of the nodes from one layer(excluding the output layer) to be connected to the other layers.



(a) Full connection: all the nodes from layer x is connected to each node in layer s. Node s_3 along with all other nodes in layer x are coloured for clarity.

(b) Sparse Connection: Some nodes from layer x is connected to node s_3. The difference can be viewed even in a small network.

Figure 1: Taken from:[2]

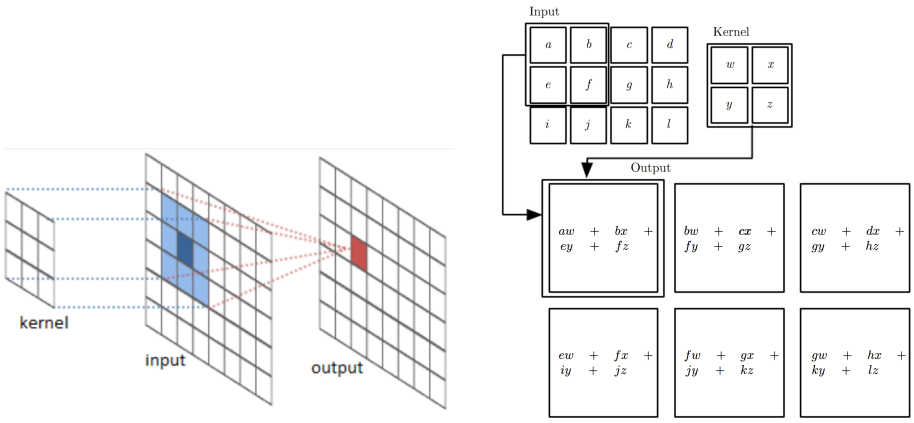
Parameter Sharing This property allows the weights of the neurons to be shared through the convolutional layer in order to reduce training time and the risk of overfitting. Once the filter and the data are convoluted, the weights are set and shared for the neurons.

Translation invariance is a property in CNNs that makes sure even if there are some slight changes in the input data the model will still give out the expected output. For example, slight changes in an image that contains a cat will not make the model not being able to classify the image as "cat".

How do convolutions actually work? The picture 2a shows how generally convolutions work over an input, while the next figure2b shows how convolutions work from a mathematical point of view.

Stride is the parameter that indicates the size of the jumps in the rows or columns a filter makes while processing the input image.

Pooling or *Pooling layer* is one of the important parts of CNNs as it shrinks down the size of the input to this layer. Therefore, this layer helps with decreasing the memory usage and the time needed to train the model. The following figure shows the two most common ways of pooling. As it can be viewed from the figure also the Max Pooling is just choosing the maximum number out of each window that is processing while the Average Pooling just computes the average value of the window.



(a) Taken from: [14] (b) Taken from: [2]

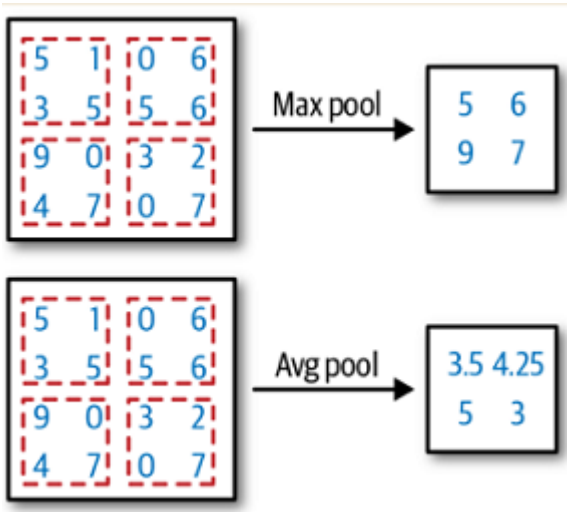


Figure 3: Pooling. Taken from: [16]

2.1.1 Relevant CNNs

Thanks to its versatility and multiple applications, the implementation of CNNs keeps growing and growing. The development of new models of CNN's and the inspiration and improvements amongst them leads to new and more advanced models. Two examples of these evolved models are Very Deep Convolutional Networks (VGG) and Densely Connected Convolutional Networks (DenseNet). Being developed in 2015 and 2018 respectively, these two networks explore the benefits of adding more depth to a convolutional network and share the base implementations as the famous model AlexNet developed in 2012.

2.1.2 AlexNet

The most well-known CNN architecture is LeNet-5 which was created by Yann LeCunn in 1998 [8] and is used for the MNIST dataset or handwritten number detection. The following figure 5 shows how this CNN is structured. The input images to this model are grey-scaled.

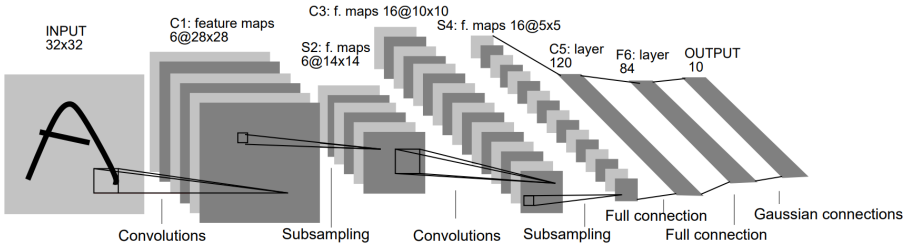


Figure 4: LeNet5 architecture. Taken from: [8]

AlexNet [7] model which won the ImageNet challenge in 2012 has lots of similarities to LeNet5's model structure. However, AlexNet is deeper and larger and has some of its convolution layers in a row.

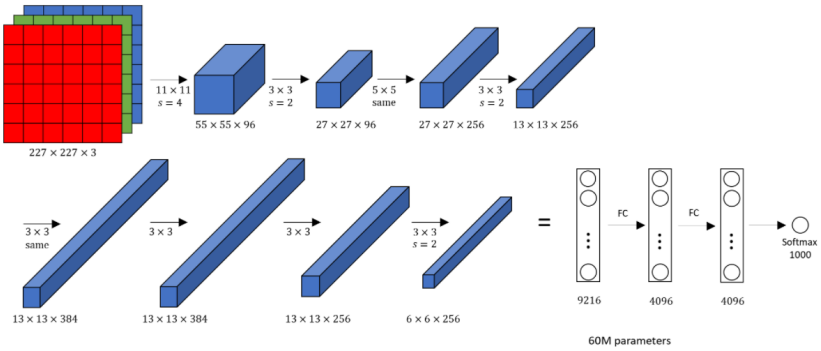


Figure 5: AlexNet architecture. Taken from: [1]

2.1.3 VGG

In this example [11], starting from 11 weight layers of depth are added up to a total of 19, showing an accuracy increase. The network was trained during 74 epochs over a dataset of 224 × 224 RGB images, which were filtered by a 3x3 filters through the layers. These layers were structured as follows:

- Stack of convolutional layers of varying depth
- Two Fully Connected layers, where each one has 4096 channels
- A third Fully Connected layer with one channel for each classification class (in this case 1000)
- A soft-max layer

Thanks to the inclusion of more weight layers, they managed to reduce the error percentage in individual tests from 29.6 to 25.5, and from 28.2 to 24.8 in multiple tests scales.

2.1.4 DenseNet

Following the idea of VGG of increasing the accuracy thanks to an increase in the depth of the network, G. Huang ***et al***, 2018 proposed a new model called DenseNet in which all layers are connected amongst each other[4].

There are two main points behind DenseNet, the first is in a layer x take full use of all the previously generated features. The second is to reduce the distance between the classification layer and the early layers in order to decrease the vanishing and exploding gradients problems.

If we compare a model that has access to all previously generated feature maps and a model that only has access to the feature maps of the previous layer. Then it is easy to imagine that the first example requires fewer layers to get the same accuracy as the latter. Having a layer connected to all previous layers can pose a problem since that is a lot of connections. DenseNet solves this by using bottleneck layers, transition layers and blocks. In each block, all the feature maps generated by layer x is available to layer $x+1, x+2, ...$ Each layer in a block is a 1x1 convolutional layer followed by a 3x3 convolutional layer. What the 1x1 convolutional layer does is restricts the amount of data passed to the 3x3 convolutional layer, which is why it is called the bottleneck layer. Specifically, the bottleneck layer feeds $4 * k$ feature maps to the 3x3 convolutional layer, where k is the growth factor, which is how many feature maps each 3x3 convolutional layer produces.

Between each block there is a transition layer, what a transition layer contains typically is a 1x1 convolutional layer followed by a pooling layer. The 1x1 convolution is done for compressing the amount of information that is passed to the next block in a way a bit like bottleneck layers but for an entire block. The 1x1 convolutional layer generates $\lfloor m\theta \rfloor$ feature maps, where m is the total number of feature maps fed to that layer and θ is the compression factor, which should have a value between 0 and 1.

By using these techniques DenseNet has access to all the previous generated feature maps while at the same time filtering out unimportant information. Because of the high connectivity, it does not need to create so many feature maps, which is why the growth factor used for most tests in the DenseNet paper was 8. Since there is not a lot of feature maps created at each layer and

the distance between the input layer and classification layer is low, this means DenseNet can have a lot of layers and it is known that having a deep network is better than having a wide.

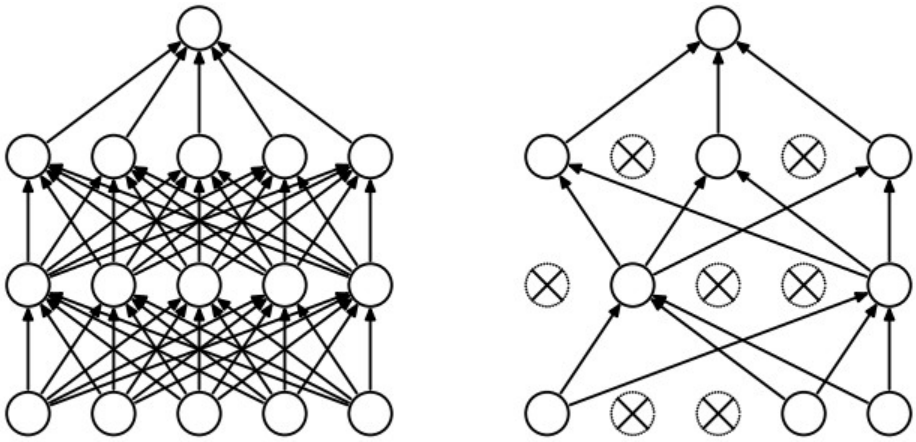
2.2 Regularization

Regularization is a term used for any techniques that help a model avoid overfitting. The following techniques are the ones that have been used in our project.

- Early stopping: The underlying idea of Early stopping is to stop the training as soon as the validation error reaches the minimum. The validation error curves are not smooth when using stochastic and mini-batch gradient descent. In order to find the minimum in these cases is that training can be stopped after a period of time that the minimum doesn't decrease any more, there the model parameters can be rolled back to the minimum validation error parameter sets.

- Dropout: A dropout is a form of regularization which prevents the model(CNNs or NNs) to memorize the dataset and actually learns the important features of the dataset. This method works in a way that randomly and periodically removes some nodes from the model. It applies to the hidden layer nodes and removes the nodes along with their input and output connections. This approach makes it possible that the neurons don't have connections to specific layers, neurons and their outputs!

  The following figure 6 shows an example of a network before and after applying dropout.



(a) Standard Neural Net (b) After applying dropout.

Figure 6: Dropout. Taken from: [12]

- Bottleneck layer(mentioned in section 2.1.4)

- Transition layers(mentioned in section 2.1.4)
- Batch Normalization(mentioned in section 2.3)
- Pooling(mentioned in section 2.1)
- Augmentation(mentioned in section 3.1)

2.3 Batch Normalization and Normalization

Normalization is a technique commonly used on the input layer of a Neural Network to decrease training time. The normalization generally used in Neural Networks is the one where you move the mean to zero and set the variance to one. A Normalized data set $\hat{X}$ is generated from X by $\hat{X} = \frac{X - mean(X)}{\sqrt{Variance(X)}}$. The first reason why this can be a good idea is that ensure that there exist both positive and negative values. If we look at how we change weights in a typical Neural Network we have the formula $\Delta w_{ji} = \eta \delta_j x_i$, where j corresponds to the nodes in one layer and i the layer before it. We can see that $\eta \delta_j$ will be the same for all i if we look at a specific j. This means the only part that separate the weight changes for the weights from the previous layer to the node j is the x_i and if all the x_i:s have the same sign that means they all will decrease or increase at the same time. This can be problematic since when we converge we probably want to decrease the weights of some nodes at the same time we increase other ones.

Another reason why normalization is good is the scaling. Imagine if the inputs are not scaled, that could be seen as equivalent to having the inputs scaled but having the learning rate different for different nodes and having different learning rates in a randomized way is probably not a good idea.

Batch normalization is a technique that can decrease training time, increase accuracy and reduce over-fitting[5]. The main point behind batch normalization is something called internal covariance shift, Which is when the input distribution to a node changes. For example, let's say we have a network and we inspect a node at the later layers of the network. Then the input to that activation function is dependant on the previous layers, which means that after a round of backpropagation even if the changes in weights to all layers individually were small, the cumulative change to the input of the node we are inspecting can be large. Large enough to change the input distribution which means that all the previous changes to the activation function in order to move it towards the optimum is wasted or at least the effect of the work is diminished. This can be even worse if we for example imagine that our activation function is a sigmoid and the change in the input distribution moved the output distribution to the saturated region of the sigmoid(where the gradients are close to zero).

Reducing the problem of internal covariance shift can reduce the training time since we train for fewer problems and are allowed to have higher learning rates since we do not need to be worried about the output distribution jumping around as much. The way Batch Normalization does this is that it takes the

output of an activation function of a batch and normalizes it to have a mean of 0 and variance of 1. It then later moves the mean and rescales it with two parameters β, γ respectively. This means that even if the output distribution of the activation function change drastically, that does not mean that the output of the Batch Normalization will change drastically.

Batch Normalization can also be done after multiplication with the weights instead of directly after the output of the activation function.

For CNNs you tend to want to batch normalization per feature map rather than per node, This is because you would probably want a 1 in location (a,b) to mean the same as a 1 in the location(x,y).

2.4 Transfer Learning

Large dataset and huge computer resources are usually needed in order to train a model from scratch especially when the model is dealing with an image dataset. Transfer learning is a technique to overcome these barriers. The underlying idea is to use the models that have already been trained on similar tasks and have already taken their time and resources to train. The approaches to implement this idea after choosing the pre-trained model is as follows:

- "Feature-extraction": Getting rid of the last layers in the model and implement new ones based on the desired output. "Freezing" is a term that is used when the weights in a layer remain the same as the previous model. In this approach, all layers excluding the layer or layers that removed get freezing while training on the new input and the newly implemented last layer/layers.
- "Fine-tuning": The model might need more changes than just the last layers, in which there are two ways to play with the chosen model. To use the weights of the chosen model as initial weights for the new model or to chose some layers as initial layers (initial weights), freeze them and get rid of some layers to implement new ones.

3 Dataset

3.1 Augmentation

The small dataset is often a barrier that is faced in machine learning. However, it is most common in the computer vision field. Data augmentation is one of the ways to solve this matter. There are some common methods that are used/implemented on image data-sets such as:

- Mirroring: Flipping the image horizontally or vertically.
- Random cropping: Randomly cropping apart from the original image.
- Colour shifting: Changing colours in the image.

- Rotation: Rotate the image randomly

Implementing each one of the above techniques will result in new data out of the original data which will improve the results of the model by increasing the size of the data-set.

Some example techniques of image augmentation can be viewed in figure 7.

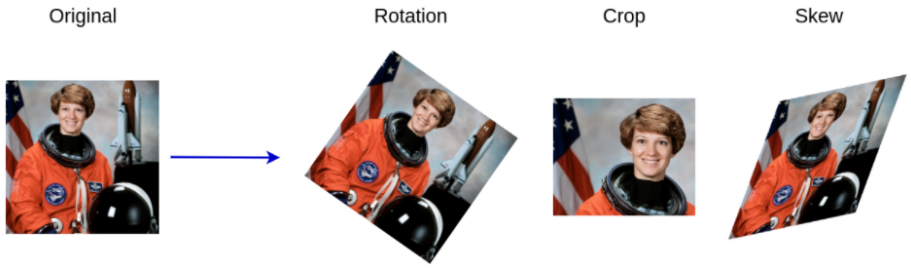


Figure 7: Examples of Augmentation. Taken from: [15]

The augmentation we used was mirroring and rotation. Where in rotation the image was rotated between $-0.1 * 2\pi$ and $0.1 * 2\pi$ radians and in mirroring we randomly flipped the image horizontally.

4 Deep-Learning Models

We made two models, one that was inspired by DenseNet and the other was inspired by VGG. Both models used a normalization and augmentation layer, except when we tested the performance of not using it. The normalization layer performed normalization as described in the background section with a mean of zero and a variance of one. The classification layer consisted of six nodes that were fully connected to the previous layer. All models used relu as the activation function and used adam as the optimizer. The classification layer did not have softmax as an activation function on the final layer and instead had $from\_logits = True$.

The DenseNet model used between 1 and 3 blocks and used transition and bottleneck layers. Bottleneck layers were used before every convolutional layer that was within a block and transition layers were used in between blocks. The compression factor used was 0.5. The transition layers used a 1x1 convolutional layer along with average pooling of size and stride 2. We had a 7x7 convolutional layer followed by a 3x3 max pooling both with stride 2 before the block section of the model. Inside the blocks, all the convolutional layers(except the bottleneck layers) used a 3x3 kernel. After all the blocks we had a global average pooling layer and then the classification layer. We used Batch Normalization for all the convolutional layers, the Batch Normalization was done in between the weighted sum and the activation function.

The VGG based model used 3x3 convolutions, followed by 2x2 max pooling and batch normalization. There was also a dropout layer in the CNN part of the

model. After the CNN part we flattened the model and used two fully connected layers with 128 nodes and a dropout layer in between.

4.1 Transfer Learning

Besides making our own models we tested using Transfer Learning for classifying the data set. When we used transfer learning we had a base model from Keras that was trained on the image-net data set and had the classification layer removed. After the base model, we had either a flattening layer of a global average pooling layer. Which in turn was connected to two fully connected layers with 1024 nodes and a dropout layer in between. After that, we just had the classification layer which was the same as described in the model part. We used normalization for our transfer learning models, the normalization used was the same as for our own models. We used relu as activation functions for our added layers and adam as the optimizer.

The base models we tested was DenseNet121[4], MobileNetV2[9], EfficientNetB0[13] and Xception[3]. They were mainly chosen since they were among the smallest networks.

4.2 Metrics

When testing the models we focused on looking at accuracy. Accuracy is just the number of correct classifications divided by the size of the data set. The reason why we used this is simply that we saw no reason to use anything else since we have the same amount of inputs for each class and there is no point in trying to better at classifying building than a street, counter to classifying when a person is infected by a disease compared to not. We also recorded time but since the experiments were done on the cloud it is hard to compare those results since we do not control the number of computing resources we have available. All models did take about 30-200 seconds per epoch so there was no extreme difference in time taken. Although the transfer learning models tended to be slower than our own.

5 Results

First, we look at the results of testing different hyperparameters for the models in order to learn what helps and what does not, then we compare the performance of our best transfer learning as well as VGG and DenseNet inspired model.

With our VGG inspired model, we look at the effects of adding augmentation, normalization and batch normalization layers.

With our DenseNet based model, we tried changing the number of blocks and the growth factor.

With our transfer learning models, we tried average pooling compared to flattening as well as seeing how well the different models performed on this

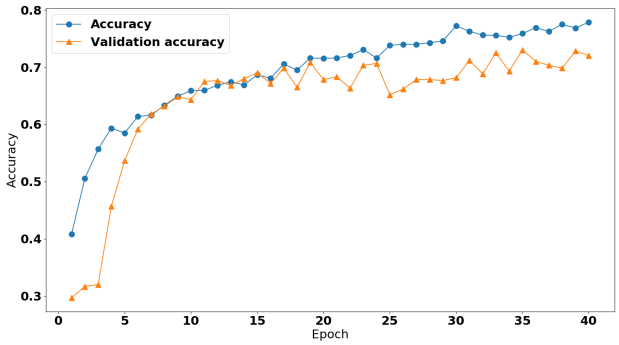
problem.

All the tests using the VGG and the DenseNet based models were done over 40 epochs. While the ones done for transfer learning used 20 epochs. When we write accuracy without giving context it refers to training accuracy.

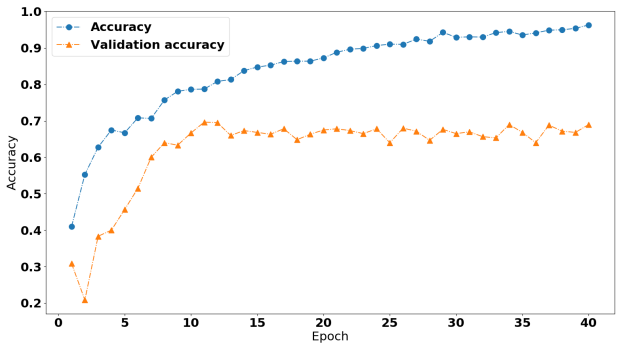
5.1 VGG

In order to test the effects of augmentation, normalization and batch normalization, we have created a baseline model with all of those techniques. Then we compare the baseline model to a model where one of those techniques neglected and look at if adding those types of are worth it.

We also look at the training accuracy compared to the validation accuracy in order to see if any of the techniques help against overfitting.

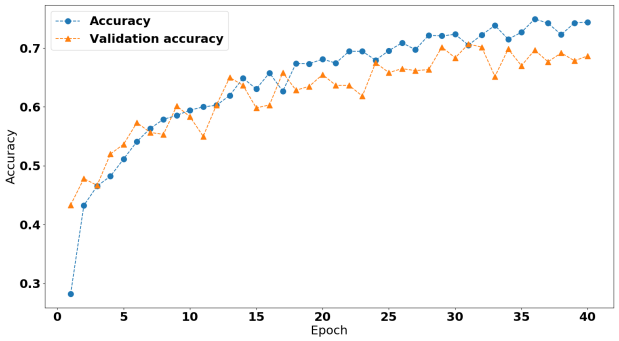


(a) Validation and training accuracy for the baseline model

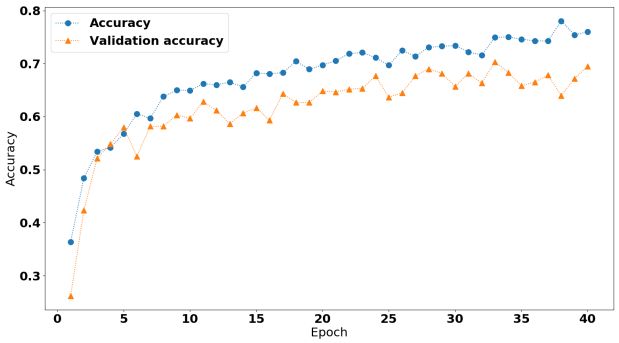


(b) Validation and training accuracy for the model without augmentation layer

We observe in figure 8b that without augmentation we seem to start overfitting. This is understandable since when using the random flip and especially



(a) Validation and training accuracy for the model without batch normalization



(b) Validation and training accuracy for the model without normalization

random rotation we drastically increase the number of images the network would have to memorize, if it wants to memorize the images instead of being generalized and interpret images.

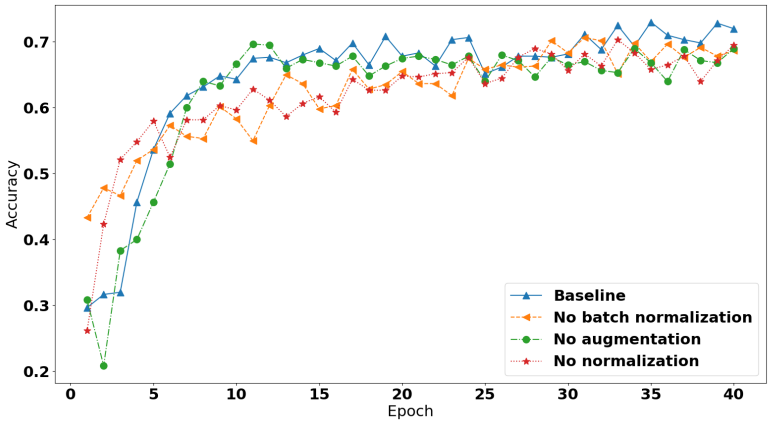


Figure 10: Validation accuracy for the different versions of VGG

We observe in the in figure 10 that including augmentation, normalization and batch normalization layers increases the performance of the network. This is to be expected since there are good reasons to use the techniques mentioned in the background and dataset sections.

5.2 DenseNet

In order to learn how to derive an efficient DenseNet based architecture, we have tested a few hyperparameters, namely the number of blocks the length of the blocks and the growth factor. The number of blocks we tested was 1, 2 or 3, these tests were done in conjunction with testing the growth factor where we had either $k = 4$ or $k = 8$. For these tests, we used a block length, the number of 3x3 convolutional layers in a block, of six.

Afterwards, we tested the effects of block length by changing the block length from six to ten on the 3 blocks, $k = 8$ version.

If we look at fig 11 we can see that the optimal parameter configuration is $k = 8$ along with 2 blocks. The reason why the 2 blocks version performs better than the 3 blocks one is likely due to having fewer weights makes the network easier to train. The reason why this is the case and not overfitting can be seen in figure 12, where for the 6 length block version the training and validation accuracy is so close together that overfitting is unlikely.

We observe in fig 12 that using a block length of six is preferable to a block length of ten. The difference in validation accuracy is not that large but its starting to look like the length ten version is beginning to overfit, with how the

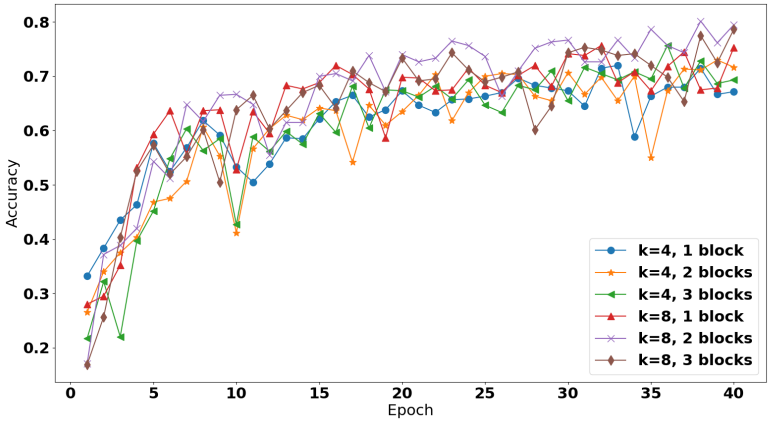


Figure 11: Validation accuracy for different DenseNet hyperparameters

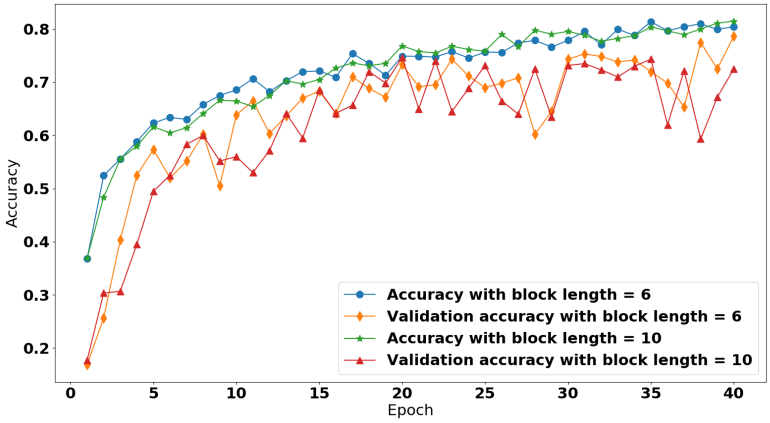


Figure 12: Validation and Training accuracy when comparing block length

validation accuracy looks stagnant from epoch 20 while the training accuracy is increasing.

5.3 Transfer Learning

In order to see which is better for our problem global average pooling or flattening, we test the performance in the form of validation accuracy for three transfer learning models. The three base models used are DenseNet121[4], MobileNetV2[9] and EfficientNetB0[13].

We also compare the models DenseNet121, MobileNetV2, EfficientNetB0 and Xception[3], when using average pooling by looking at validation accuracy.

Afterwards, we test how the augmentation impacts transfer learning, by testing two versions of our DenseNet121 transfer learning model. Both have global average pooling, one has an augmentation layer, the same way as described in section 6, while the other does not.

If we look at fig 13a and 13b we observe that flattening seems to perform slightly better than global average pooling, This is understandable since by flattening you preserve the features generated by the base model better.

We also note that EfficientNetB0 does not perform well for this problem.

We observe in fig 14 that MobileNetV2, DenseNet121 and Xception, roughly have the same performance. Although a small edge should be given to Xception.

We observe in fig 15 that augmentation improves validation accuracy as well as reduces the difference between training and validation accuracy.

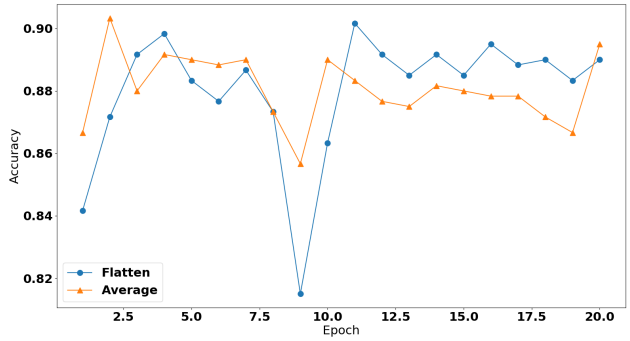
5.4 Final Results

We took the best version of each model and trained it on the entire Kaggle training set(14000 files) and collected the accuracy on the test set(3000 files). The DenseNet model chosen had $k = 8$, 2 blocks, and a block length of 6. The VGG model was the one with all the techniques. The transfer learning one was DenseNet121 with augmentation and global average pooling.

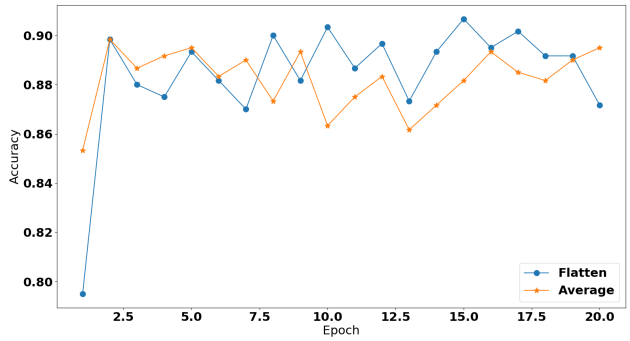
The training for the DenseNet and VGG models was done for 40 epochs while the transfer learning model trained for 20 epochs, this was to compensate the difference in time taken per epoch.

For each of the models, we generated a confusion matrix.

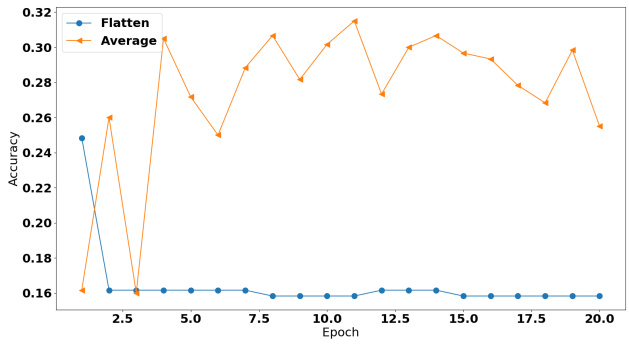
Model	Testing accuracy
DenseNet	0.8126
VGG	0.7478
Transfer Learning	0.9163



(a) Validation accuracy for the DenseNet121 model when using flattening compared to global average pooling



(b) Validation accuracy for the MobileNetV2 model when using flattening compared to global average pooling



(c) Validation accuracy for the EfficientNetB0 model when using flattening compared to global average pooling

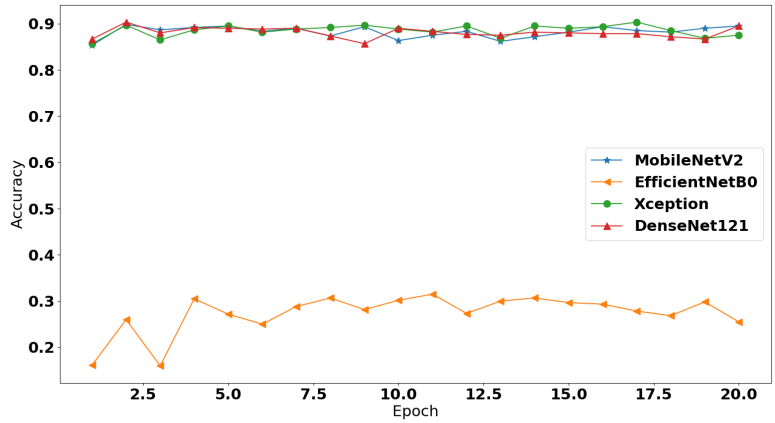


Figure 14: Validation accuracy for different transfer learning models

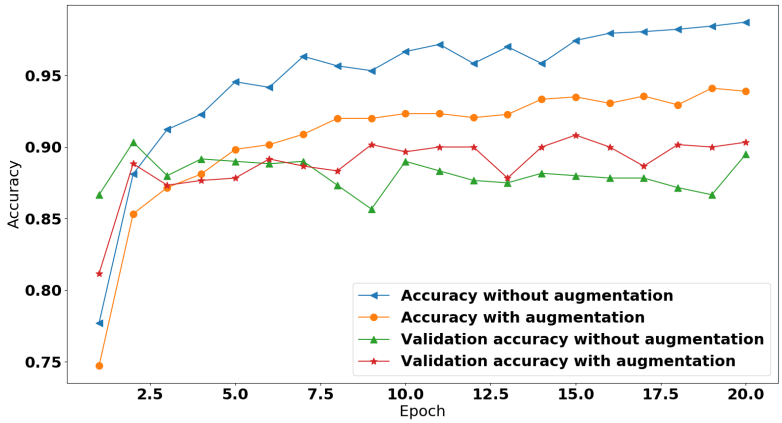


Figure 15: Validation and training accuracy for DenseNet121 model with and without augmentation layer

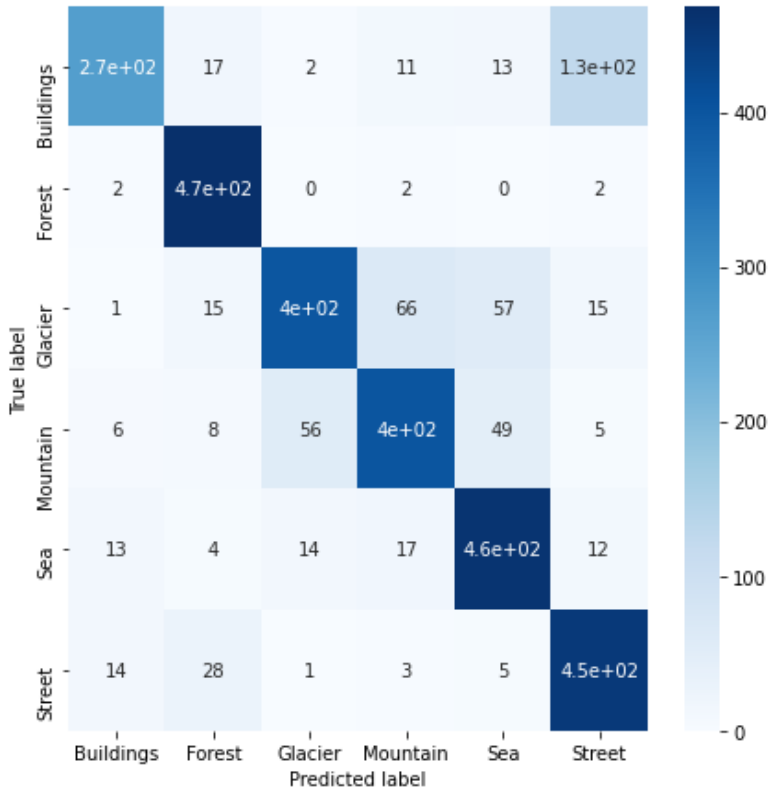


Figure 16: DenseNet confusion matrix

From the table and confusion matrices, we can quite clearly see that transfer learning outperforms our own models.

6 Discussions and Future Work

If we compare all the validation accuracies we can clearly see that the transfer learning models performs the best. This is understandable since there is a significant amount of thought behind those models, they have more complicated models since they had access to more computing power and they have done a lot of training already. Afterwards comes our DenseNet version and then our VGG version, although it is worth noting that more parameter testing had been done for DenseNet than VGG.

Using augmentation seemed to have significant effects against overfitting both for transfer learning and the VGG based model. It is reasonable to assume it also had a significant impact on our DenseNet based model as well as most models that used the same dataset. This is likely because our dataset is small with only 1800 images for training and the importance of augmentation would

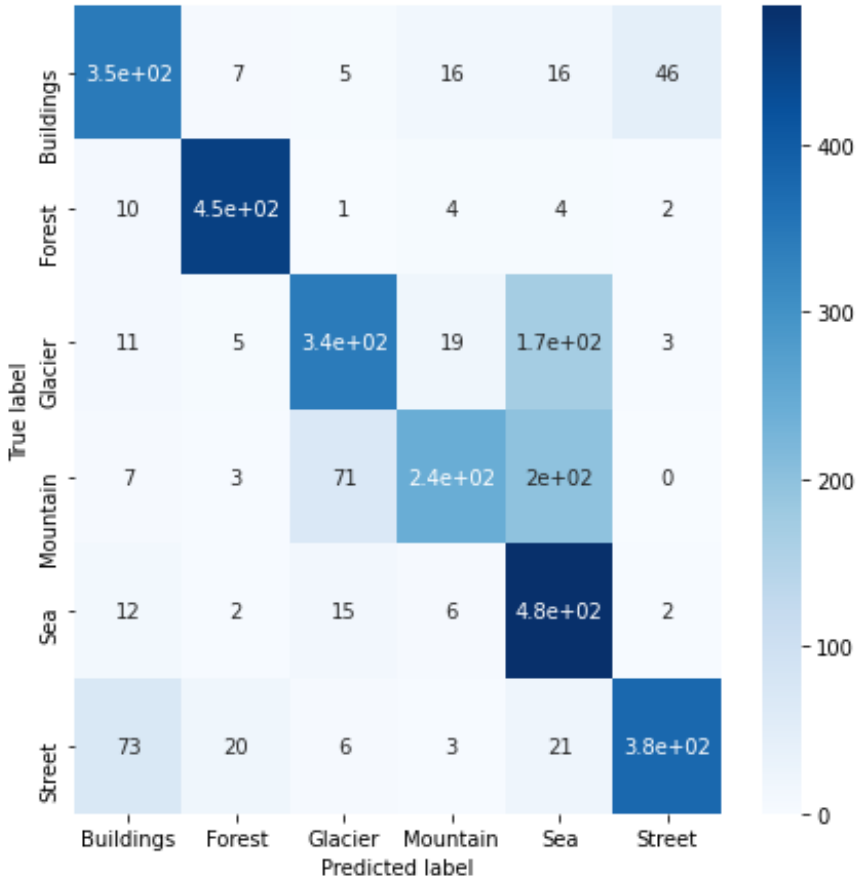


Figure 17: VGG confusion matrix

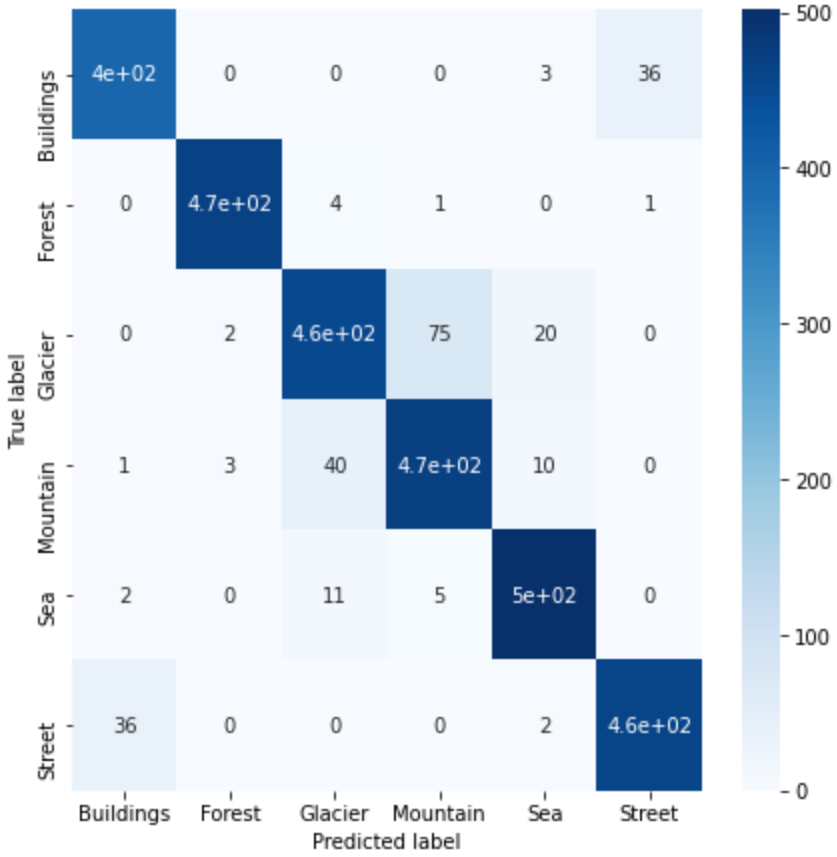


Figure 18: Transfer learning confusion matrix

probably decrease the larger the dataset. It would have been interesting to see which would perform better, using our current dataset with augment or using the entire dataset from the Kaggle competition and sampling 1800 images for training each epoch. Which could show what is more important quality or quantity of images since the use of random rotation gives us orders of magnitudes more images at the cost of the quality.

It would have been interesting to try the effects of block length more since in the paper for DenseNet[4], they have different block length for different blocks when using the architecture on the ImageNet dataset. Although this might not necessarily be good for our problem since that was the only data set they did not have an equal block size. Also having more experiments with block length would help us draw more rigorous conclusions for the dataset and block length, we can not draw many conclusions since we only tested on one configuration of parameters(excluding the block length parameter).

It is hard to say which transfer learning model was best, on one hand, Xception had the highest performance. On the other hand, it was also the largest model both in terms of memory size [6] and in terms of time per epoch, MobileNetV2 and DenseNet121 took about 150 seconds per epoch while Xception took about 200. Taking this into account we think that DenseNet121 and MobileNetV2 were the best models for the problem and went with DenseNet121 for the augmentation and final experiments on a whim.

We used global average pooling rather than flattening for the final experiments despite flattening having slightly better performance. The reason for this is the same as the reason for not choosing Xception, the difference in performance was small and we took other factors into consideration such as time.

If we look at the confusion matrices we can see that the hardest classes to differentiate are mountain and glacier, building and street, and sea, mountain and glacier. This is all very understandable since glaciers tend to be on mountains, buildings tend to have streets and seas and glaciers are made of water. It is a bit weirder confusion of mountains and seas, although this can be explained as the combination of the sea and glacier connection with the mountain and glacier connection. There is also the fact at if you look from a beach(where some of the sea images are taken) and look at where the sea meets the land there are gonna be some places where there are quite the difference in elevation, which if that difference were on land that would be an indicator for a mountain. There are also some sea images that have mountains in the background since seas do not obstruct what is behind them. This problem could be considered a multi-label instead of a multi-class. It would have been interesting to see which images we misclassified in order to check whether our speculation is correct or there is some other factor that is the reason for the misclassifications.

7 Conclusions

Transfer learning is really good and will probably be better than design your own model unless you have a lot of time and computing resources. Augmentation

can be of significant help when reducing overfitting. More layers do not mean better even if we are not overfitting. The line between multi-class and multi-label problems can be blurry.

References

[1] 013 b cnn alexnet, 2018. `http://datahacker.rs/deep-learning-alexnet-architecture/`.

[2] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*, volume 1. MIT press Massachusetts, USA:, 2017.

[3] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[4] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[5] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[6] Keras. Keras applications. `https://keras.io/api/applications/`.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[8] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[9] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[10] Neha Sharma, Vibhor Jain, and Anju Mishra. An analysis of convolutional neural networks for image classification. *Procedia computer science*, 132:377–384, 2018.

[11] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[12] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[13] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[14] River Trail. Edge detection and sharpening. `http://intellabs.github.io/RiverTrail/tutorial/`.

[15] Ivan Vasilev, Daniel Slater, Gianmario Spacagna, Peter Roelants, and Valentino Zocca. *Python Deep Learning: Exploring deep learning techniques and neural network architectures with PyTorch, Keras, and TensorFlow*. Packt Publishing Ltd, 2019.

[16] Alice Zheng and Amanda Casari. *Feature engineering for machine learning: principles and techniques for data scientists*. " O'Reilly Media, Inc.", 2018.