16 June 2021

Assessment 3: Capstone Report

Image Classification with Convolutional Neural Networks

Nikki Fitzherbert 13848336
MA5852 MASTER CLASS 2

Contents

Part 1: Paper Review

Convolutional neural networks (CNNs) take their inspiration from the operation of simple and complex cells in the visual cortex (LeCun et al., 2015), and use convolutions in place of general multiplication in at least one of their layers in order to use the spatial information between different features (Goodfellow et al., 2016). CNNs are a powerful deep learning algorithm that has been successfully applied to a wide variety of different areas including image and video classification, object detection, semantic segmentation and time series forecasting (Géron, 2019).

Nevertheless, CNNs were largely ignored by the mainstream computer vision and machine learning communities until the advent of the ImageNet competition[1] in 2012 (LeCun et al., 2015). The objective of this part of the report, therefore, was to review the architecture of one of the winners of that competition.

The Inception architecture was developed by Szegedy et al. and the variant that won the 2014 ImageNet challenge (nicknamed GoogLeNet) is detailed in their paper "Going deeper with convolutions". It was able to achieve a top-5 error rate of less than 7%, compared to ZFNet's 11.2% and AlexNet's 15.3%, with significantly fewer parameters (around 5 million compared to AlexNet's 60 million). However, like AlexNet before it and ResNet in 2015, Inception fundamentally changed subsequent approaches to CNN architecture design (Anwar, 2019).

The overall approach to Inception's design was partly inspired by the famous "we need to go deeper" meme from the 2010 Christopher Nolan film of the same name (Szegedy et al., 2015)[2]. The reason for this was the desire to increase both model width and depth to improve performance without also significantly increasing the number of parameters (Szegedy et al., 2015).

Inspired by the way the human visual system works, and in particular, the way that images are simultaneously processed at different scales to capture multiple salient features, as well as the hypothesis that a sparsely connected architecture would be able to address several known issues with deep neural networks, Szegedy et al. (2015) developed the keystone of their architecture – the "inception" module. This would not only allow the model to choose the best filter size, but result in an increase in model width as the input would be simultaneously passed through four different layers (three convolution layers with different filter sizes and a max pooling layer) before being concatenated depth-wise.

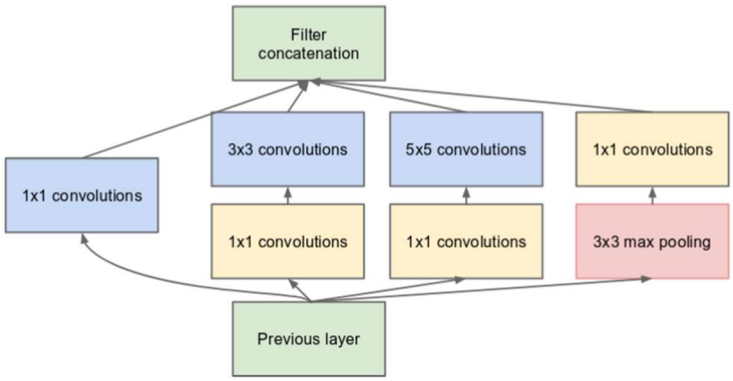


Figure 1: Inception module with dimensionality reduction

However, it was observed that the presence of even a small number of 5x5 convolution layers along with pooling layers would quickly become prohibitively expensive. Therefore, Szegedy et al. (2015) introduced a 1x1 convolution layer just before each 3x3 and 5x5 layer in the block to act as a form of dimensionality reduction (see Figure 1).

The second major innovation was the incorporation of intermediate or auxiliary classifiers, to assist with the common

[1] Also known as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) (ImageNet, n.d.)
[2] And hence the name.

issue in deep neural networks of vanishing gradients. In particular, the authors identified that the effective propagation of the gradients back through the layers was problematic and the inclusion of these classifiers would increase the gradient signal being propagated back and provide additional network regularisation (Szegedy et al., 2015).

The final GoogLeNet network structure was designed with computational efficiency and practicality in mind (Szegedy et al., 2015). It was 22 layers deep (27 including the pooling layers) and had four main components: the initial stem, nine stacked inception modules, two auxiliary classifiers, and a final classifier. Apart from the final dense layers, all convolution and dense layers in the network used a Rectified Linear Unit (ReLU) activation function. A full model schematic and topology details are located at Appendix B.

- The stem was seven layers deep, and was made up of a series of standard convolution, max pooling and normalisation layers.
- Next were the nine inception modules. These were divided into three blocks (2, 5 and 2), with a max pooling layer separating each block.
- The two auxiliary classifiers were connected to the fourth and seventh inception modules. Both were made up of an average pooling, 1x1 convolution, dense, 70% dropout, and final dense layer with softmax activation.
- The final classifier consisted of a global average pooling layer, 40% dropout, and a dense layer with softmax activation.

The GoogLeNet variant of the Inception architecture took out first place at both the ILSVRC 2014 classification and detection challenges. Whilst GoogLeNet was the only model to achieve a top-5 error rate of less than 7% in the classification challenge, the authors noted that they had used a more aggressive cropping approach than that used by Krizhevsky et al. (2012)'s AlexNet, and this may not actually be required in real applications due to the diminishing marginal benefit of additional crops after a certain point. GoogLeNet also won the detection challenge by a fair margin using an ensemble of six networks (43.9% vs 40.7% for second place). As a result, Szegedy et al. (2015) concluded that using a sparse network structure was a viable and useful method of improving neural network performance for computer vision as opposed to the previous approach of just adding more layers.

However, the Inception architecture was not without its drawbacks. In a subsequent paper, Szegedy et al. (2016) identified several issues with the first version that led to the development of versions two and three (and further) of the structure. Firstly, the complexity of the architecture and the lack of documentation on the reasoning behind many of the design decisions meant that it was difficult to make changes to the architecture or just scale it up without compromising the computational efficiency.

Secondly, the auxiliary classifiers acted much more like model regularisers than assisting in network convergence early in the training process, particularly if dropout or batch normalisation layers were incorporated.

Finally, not only did convolution layers with large filter sizes tend to be disproportionally expensive on a computation basis (for example, 963 mega FLOPS for a 5x5 layer compared to 128 mega FLOPS for an entire inception module (Smola, 2019), but feature map size reduction through the use of pooling layers led to "representational bottlenecks" and information loss. The hypothesis here was that neural networks performed better when convolutions did not try to significantly alter the dimensions of inputs.

Part 2: Computer Vision

Research proposal/Project Plan

Artificial structures such as buildings, bridges and roads are subject to gradual degradation due to a number of different stresses during their construction and period of existence, including design and construction errors, natural disasters and the impact of the surrounding environment (Mohan & Poobal, 2018). One of the earliest signs of a potential compromise in structural integrity are the emergence of cracks, which generally occur because the stresses in the concrete have exceeded its strength (Prasad, n.d.).

Often cracks are non-structural, which means that they are just aesthetically displeasing and do not affect the stability or durability of the structure. Structural cracks on the other hand are more serious and generally occur as a result of poor design, faulty construction or overloading. Furthermore, without appropriate attendance and remediation, non-structural cracks can become structural if there is sufficient ingestion of moisture and other corrosive substances (Civil Site Visit, 2019; Meeedee Design Services, 2019).

As a result, early detection is key to preventing later system or catastrophic failure (Mohan & Poobal, 2018). However, visual inspection of surfaces continues to be the most widely-used technique for monitoring concrete structures and such an approach is subjective, laborious, time-consuming and sometimes hindered by difficult or hazardous access to the desired part(s) of the structure (Flah et al., 2020; Ren et al., 2020; Wan et al., 2021).

Advancements in a number of relevant areas including sensor and communication technologies, unmanned aerial systems, cloud computing and data-driven techniques such as deep learning and the use of transfer learning have fuelled a resurgence in the attempt to develop models to detect and classify cracks in civil engineering structures (Azimi et al., 2020). And in many cases, the primary objective has been to reduce the need for manual visual inspections: For example, Cha et al. (2017) proposed a deep CNN architecture with a sliding window technique for images larger than 256x256 pixels and was able to obtain an accuracy of 98%; Zhang et al. (2016) used a deep CNN to classify whether road network images contained cracks, which were then smoothed and extracted with adaptive thresholding with an accuracy of 99.9%; and Dung and Anh (2019) proposed a deep fully convolutional network (FCN) with a VGG16-based encoder as the network's backbone to classify pavement cracks. This final study obtained F1 and average precision scores of 90%.

The intent of the overall project is to take advantage of advancements in technology and data science modelling to create a system consisting of a series of models that can not only detect the presence of cracks in any provided image irrespective of surface type, but then quantify the number of cracks in the image, their location and the severity of the cracking. This would allow construction and engineering businesses, and state and federal governments, to determine where materials, equipment and human resources might be needed most before expensive and potentially irreparable damage occurs. The objective of this project was to start developing the first stage of that system by building a CNN model that can detect cracks in images of concrete surfaces. It will have two parts: the first will be to develop and train a model from scratch and the second will be to apply a pre-trained Inception model to the same dataset. The performance and training times of both models will be compared and discussed.

Data

The data used for the project was Özgenel (2019)'s "Concrete Crack Images for Classification" dataset. It consisted of 40,000 coloured images of concrete surfaces, divided equally into those with and those without cracks. Each image had a dimension of 227x227 pixels. The images had been generated from a smaller set of 458 high-resolution images (4032x3024) of surfaces belonging to a number of different concrete buildings in the Middle East Technical University in Turkey using the technique described in Zhang et al. (2016):

- Each high-resolution image was segmented into a series of 227x227 patches, and classified as containing a crack if the image centre was within five pixels of the crack centroid.
- The overlap of adjacent patches containing cracks was minimised by specifying a distance between the centre of relevant patches and disallowing the overlap of any patches that did not contain a crack.
- Candidate patches were rotated by a random angle between zero and 360 degrees.

No additional data augmentation, such as rotation, shifting or flipping was applied.

Finally, the authors noted that the surfaces varied with respect to the type of finish photographed; that is, exposed, plastered or painted, the images were taken about one metre away with the camera directly facing the target surface, and were all taken on a single day under similar illumination conditions. Figure 2 shows a small sample of images from both outcome classes.

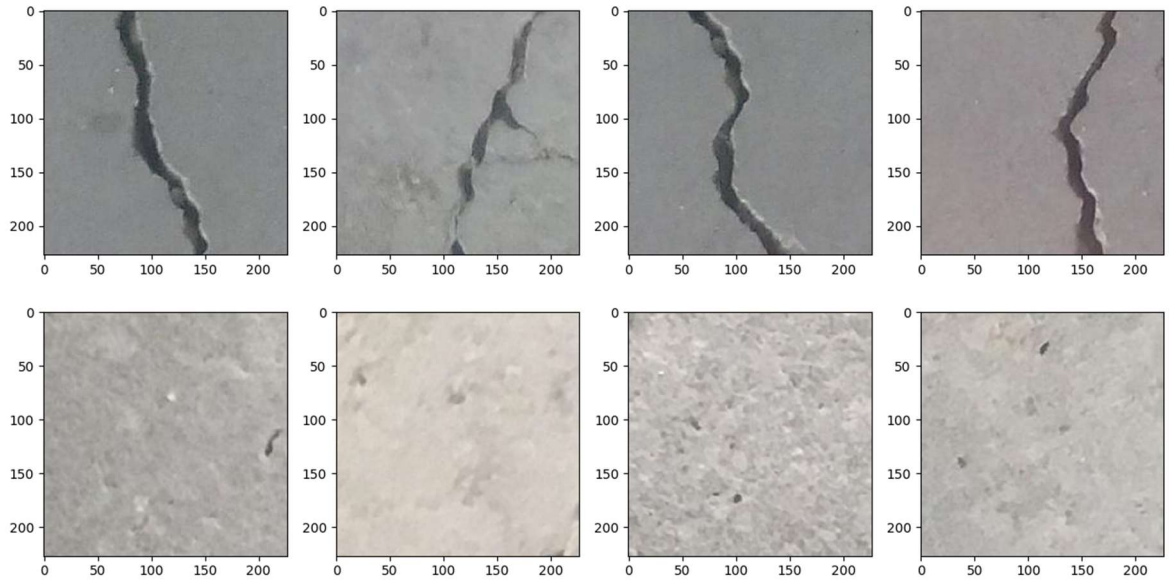


Figure 2: Sample of images from the positive class (top) and the negative class (bottom)

Whilst it was impossible to make a close visual inspection of the entire dataset to make a conclusive determination regarding its quality, perusal of the first thousand or so images indicated that the dataset was of reasonable quality. There was no indication of image corruption, which might have occurred during the segmentation process, image distortion such as from the sun or a camera flash reflecting off a highly-reflective surface or image obscuration by undesired objects such as foliage or insects. Furthermore, both image classes were reasonably diverse in the sense that they both contained a range of different colours and other non-crack features that tend to occur on outdoor surfaces

like lines, splodges of paint and spalling[3]. Furthermore, the cracks in the positive class images were of differing widths, lengths and angles, and sometimes there was more than one crack in the same image.

However, some images were blurrier than others in both classes, which could make the classifiers job harder, particularly with respect to the detection of thinner, less obvious cracks. Also, the positive class included images with cracks just in the corners, which may not be picked up by the classifier during model training due to the downsampling that occurs as an image moves through the layers of the CNN. Some of these issues are presented in Figure 3.

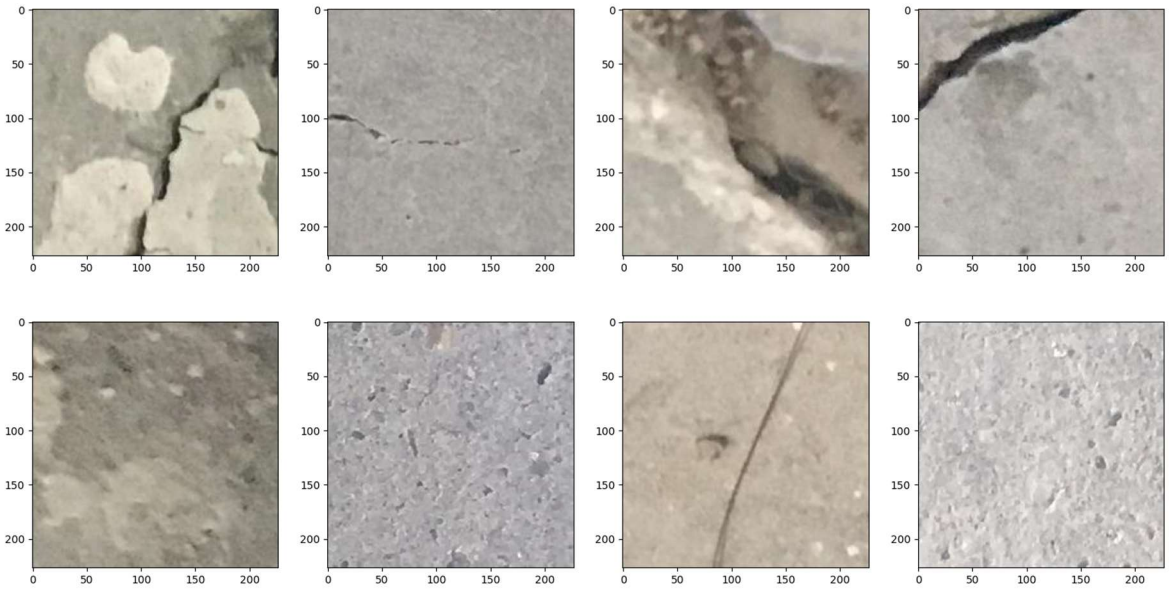


Figure 3: Sample of images indicating both the dataset quality and the types of issues present in the positive class (top) and the negative class (bottom)

Nevertheless, the issues identified above were not deemed serious enough at this stage to manually go through the dataset and remove potentially unsuitable images. Furthermore, the author hypothesised that retaining such images would assist in creating a more robust classifier as it would be likely to encounter such images if taken into production.

The pixel intensities of the images were also normalised to fall between a range of zero one. The purpose of this step was to ensure that each image had a similar data distribution and also generally increases the rate of convergence during model training.

No other necessary data cleaning or image preparation tasks were identified.

The images were then randomly allocated to training, validation and testing datasets according to a 70:15:15 ratio; that is, 14,000 of the images were used for model training, 3,000 for model validation and the final 3,000 images for final model evaluation on an unseen dataset. The terminal commands used to retrieve a copy of the dataset from Kaggle[4] and create the training, validation and testing datasets are at Appendix C. The image normalisation formed part of the python script at Appendix D.

[3] Pits and pock marks.
[4] https://www.kaggle.com/arunrk7/surface-crack-detection

Model Development

Baseline CNN architecture

The configuration of the initial CNN model was guided by the author's understanding of deep learning models, the results and conclusions of previous relevant research. In particular, the author had observed that excellent performance on the same or a similar dataset had been achieved by relatively parsimonious and shallow architectures (for example, see Ali et al. (2021), Flah et al. (2020), Kim et al. (2021)). The latter was an important consideration due to the size of the dataset, the computational resources available to the author and the project timeframe.

It used a relatively simple but classic sequential architecture; namely, two convolutional blocks followed by a pooling layer and then a dense hidden layer and a dense output layer:

- The input shape was 227x227x3 because the images are coloured. If they were greyscale then the number of channels would be 1.
- Next were two convolutional layers. Both consisted of 32 filters of size 3x3 and default striding. Zero padding was included to ensure that the images did not shrink in size as they progressed through the network.
- After that was a max pooling layer of size 2x2, which meant that it shrunk the spatial dimension in half. These also acted as regularisers to reduce model overfitting.
- The block finished with a dropout layer, which prevented a random subset of neurons (50% in this case) from activating at each training epoch.
- This structure was repeated, except that one of the convolution layers was dropped and the number of filters was doubled to 64. Increasing the number of filters enabled the model to take advantage of the fact that there are many ways to combine lower-level features to higher-level features (Géron, 2019).
- The flattening layer provided an interface between the convolutional and artificial parts of the network. Its purpose was to transform the two-dimensional feature maps into a single long vector that could be passed into a dense layer.
- The artificial neural network (ANN) part enabled the image classification to occur, and consisted of a hidden dense layer with 256 neurons, a dropout layer with the same dropout rate as previously and an output layer with a single neuron to classify the images into those with cracks (the positive class) or those without (the negative class).

Figure 4 visualises the proposed model structure:

```
Model: "initial_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 227, 227, 32)      896
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 227, 227, 32)      9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 113, 113, 32)      0
_________________________________________________________________
dropout (Dropout)            (None, 113, 113, 32)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 113, 113, 64)      18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 56, 56, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 56, 56, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 200704)            0
_________________________________________________________________
dense (Dense)                (None, 256)               51380480
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 257
=================================================================
Total params: 51,409,377
Trainable params: 51,409,377
Non-trainable params: 0
```

Figure 4: Proposed CNN model architecture

Model development considerations - Parameter and hyperparameter selection

1. A kernel size of 3x3 was used for each convolutional layer as they struck the right balance between computational cost (larger filters are very expensive) and feature extraction size (smaller filters may ignore information from neighbouring pixels) (Pandey, 2020, June 23).
2. Max pooling was chosen over average pooling for this model as it preserves the strongest features and provides a clearer signal to the next layer. A 2x2 kernel size was used for both pooling layers.
3. The ReLU activation function was chosen for all convolutional and hidden dense layers as it has a fast computation time (compared to the alternatives such as the sigmoid and tanh function). It also tends to create more stable gradients, which minimise the likelihood of vanishing or exploding gradients (Géron, 2019; Goodfellow et al., 2016).

4. The kernel initialiser was changed from the default Glorot to He based on the discussion in Géron (2019) and preliminary model testing conducted outside of the AWS environment.
5. A sigmoid function was chosen for the output layer activation function given this was a binary classification problem.
6. The model was initially compiled with a Nesterov Accelerated Gradient optimiser, which is generally faster than both vanilla momentum optimisation and plain Stochastic Gradient Descent (which can be very slow).
7. The initial choice of batch size (128 images) was chosen somewhat arbitrarily initially, noting that Géron (2019) recommended using a large batch size with a hardware accelerator such as a GPU and reducing it if model training proved unstable.
8. Again, the initial choice of initial learning rate (0.001) was chosen somewhat arbitrarily, but strove to find a balance between a lengthy model training time and an inability to converge.

Model development considerations – Regularisation and overfitting

In acknowledgement that CNNs are prone to overfitting, three regularisation techniques were employed:

1. As previously mentioned, pooling layers were used in the model to divide the feature maps into small, non-overlapping kernels for the next layer. This layer also performs subsampling of the feature maps, which means that in addition to reducing data dimensionality, the number of parameters, and computational time they also act as model regularisers and therefore help to minimise overfitting (Ali et al., 2021; Géron, 2019).
2. Dropout is one of the most popular, and most effective methods of preventing model overfitting (Goodfellow et al., 2016). It operates by forcing a random proportion of neurons in each layer to deactivate during model training at each epoch and therefore forces the model to look for more generalisable features. In this case, a fairly aggressive rate of dropout was used (50%).
3. Early stopping is another popular approach to minimising overfitting and reduces, if not eliminates, the need to tune the number of epochs. This is because the model will halt training once the rate of improvement (or deterioration) in the chosen metric is sufficiently small over a pre-specified number of epochs. The monitored metric in this case was validation loss and the patience parameter was set to five epochs, primarily to limit computation time. The initial model was permitted to run for a maximum of 20 epochs.

Hyperparameter optimisation

The model described in the previous sections was able to achieve training and validation accuracies and F1 scores of 98.7% and 98.4%[5] after 20 epochs, which indicated that the baseline model configuration choices were reasonable. However, a small hyperparameter tuning job was also run to determine if there might be a better configuration with respect to selected hyperparameters; that is, the batch size, learning rate, optimiser and number of nodes in the hidden dense layer. The permitted ranges were:

[5] The accuracy and F1 scores were exactly the same for both datasets.

- Batch size: Any integer value between 8 and 128
- Hidden layer nodes: Any integer value between 32 and 256
- Learning rate: Any float value between 0.0001 and 0.1 on a reverse logarithmic scale.
- Optimisation algorithm: Nesterov Accelerated Gradient (NAG), Adaptive Moment Estimation (Adam), Nesterov Accelerated Adaptive Moment Estimation (Nadam) or RMSprop.

Despite the performance metric charts indicating that the number of epochs was too short, and increased performance could be gained by increasing the allowable total training time (see figure 5), it was kept at 20 epochs largely to keep model training times reasonable. Furthermore, as stated previously, the model was still able to obtain very acceptable performance scores on the validation data (98.4% accuracy, 99.8% AUC and a 98.4% F1 score).

Figure 5: CloudWatch algorithm charts during initial model training

Before proceeding to the results, the author would like to provide a brief explanation on why those particular hyperparameters were chosen for the tuning job:

- Batch size: This is one of the most important hyperparameters to tune but there is a compromise to be made in choice of value. Large batch sizes can speed up training times if hardware accelerators such as GPUs are present, but can also lead to model instability and an inability to perform well on new data. On the other hand, smaller batch sizes tend to lead to faster convergence, but the model could land on a local instead of global optima (Kandel & Castelli, 2020; Naincyjain, 2020). The upper range of permissible values for model training was limited by the computational resources available.
- Learning rate: The learning rate refers to the amount that the model weights are updated during training. Because of that, it is often also called the step size. Like the choice of batch size, the choice of value indicates what is considered more important. A larger learning rate will result in rapid changes and decreased training times, but the model could converge too quickly to a suboptimal solution. In contrast, models with smaller learning rates require longer training times but if it is too small could result in the model never converging (Brownlee, 2019).
- Optimisation algorithm: Choice of optimisation algorithm can have a significant impact on model performance and model training times. This was why a diverse set of algorithms were tested. For example, whilst SGD was a popular choice in the crack detection literature, it can be very slow to train and the Adam algorithms (or its

variants such as Nadam, which adds moment estimation) can be much faster (Géron, 2019).

- Dense hidden layer nodes: There are no hard and fast rules, or even general rules-of-thumb for this particular hyperparameter. For example, one suggestion is to start with a value somewhere between half and two-thirds of the average of the number of inputs and number of output classes. Increasing the number of nodes means more underlying patterns and features of the dataset can be learned, but will also increase the computational complexity. The author found that she could only use a maximum of 256 nodes without overloading AWS SageMaker.

The results of the tuning job indicated that the initial model configuration was the best in this instance. No other combination of hyperparameter values could come close to the initial model (see Figure 6).

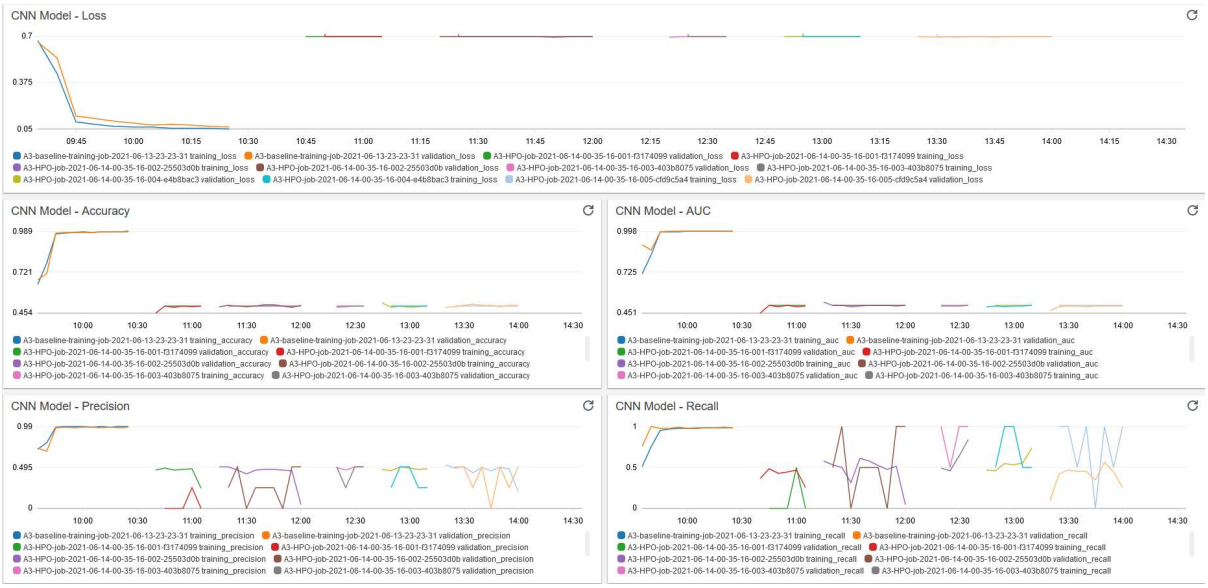


Figure 6: CloudWatch algorithm charts for the initial and models trained during hyperparameter optimisation

However, the oddness of the values of the performance metrics in that they all maxed out at a value well below the performance of the initial model trained irrespective of the hyperparameter configuration suggested (see Table 1) that something else might have been going on. This is discussed further in the performance evaluation section of the report.

Model	Batch size	Hidden layer	Learning rate	Optimiser	Epochs
Initial	128	256	0.001	NAG	20
Alt 1	48	201	0.085	Nadam	8
Alt 2	107	256	0.010	Adam	15
Alt 3	117	105	0.030	NAG	6
Alt 4	103	105	0.1	RMSprop	9
Alt 5	55	46	0.007	Adam	14

Figure 7: Hyperparameter configurations for the initial and alternative models

Model Evaluation and Deployment

Training, validation and test data assessment

As mentioned in the previous section, initial indications of model performance were excellent. The initial model produced scores over 98% across all the major metrics being monitored for both the training and validation datasets and there were indications that even better performance could have been obtained by letting the model train for longer than just 20 epochs (see Table 1 and Figure 8 below). Furthermore, the learning charts (see Figure 5 from the previous section) indicated that the model was not yet overfitting: the validation loss was still trending downward and the values of all metrics were very similar and converging.

Dataset	Loss	Accuracy	Precision	Recall	AUC	F1
Training	0.0560	0.9870	0.9895	0.9845	0.9986	0.9870
Validation	0.0642	0.9838	0.9853	0.9823	0.9978	0.9838
Test	-	0.8889	0.9130	0.8750	0.8899	0.8936

Table 1: Performance metrics on the training, validation and test data

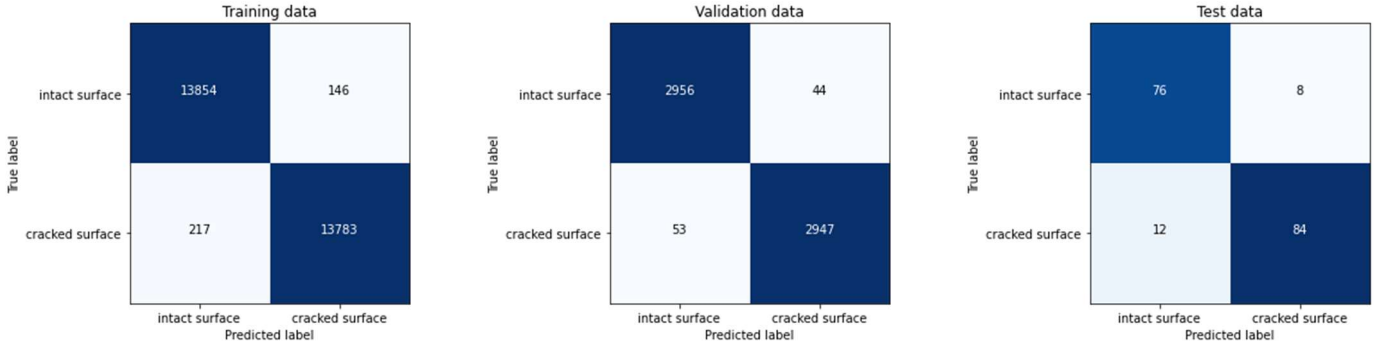


Figure 8: Confusion matrices for the training, validation and test data

Whilst not as good, the results were equally as promising when the model was deployed to an endpoint and predictions made on a small subset (180 images) of the 6,000 images in total originally set aside for an evaluation on an unseen test set. It had mistaken a cracked surface for an intact one around 6.5% of the time and made the opposite mistake 4% of the time. Whilst in this situation it would be preferable to have it around the opposite way given that mislabelling a cracked surface as intact is potentially more serious than a wasted inspection trip, it must be observed that a robust conclusion regarding performance would require assessment on the full test dataset. Nevertheless, the relative stability of the performance metrics from the training to the validation and test datasets provided further evidence that the classifier was not overfitting to the training data and was able to apply what it had learned of the differences between cracked and intact surfaces to a set of images it had never seen before.

On the other hand, this sort of consistent high-performance meant that the lacklustre performance during the hyperparameter optimisation stage somewhat puzzling. The author can only conclude that perhaps Amazon SageMaker was not reading in the entire dataset correctly despite the fact that the script and other code used had been applied successfully in the initial training of the model and/or applied successfully in the past to other datasets.

Evidence of endpoint creation and deployment of this CNN models is in the final section discussing the implementation on AWS.

Assessment against the project objectives

The purpose of the second half of this project was to start developing and overall system for developing and assessing the severity of cracked surfaces in order to determine whether, and how urgent, any required remediation work was. Advances in machine and deep learning and other technologies such as UAVs are enabling companies to visually inspect and or assess cracks that historically have been difficult for humans to get to. This approach can also be used as a complementary source of information to remote sensors.

Similar prior research into such systems and deep learning classification models on the same or different datasets reported roughly the same level of performance, which provides further evidence that the most is a good starting point at the very least. For example, (Ali et al., 2021) reported validation and test dataset accuracies upwards of 91% across datasets ranging from 2,800 to 25,000 images, and Kim et al. (2021) and Flah et al. (2020) reported accuracy and test set accuracy of 99.8% and 98.3% respectively on the same dataset as used for this project.

However, this model is not yet ready to be put into production for several reasons:

- Firstly, the classifier needs to undergo a more robust testing process to determine more conclusively its ability to generalise to new images.
- Secondly, the classifier needs to be exposed to a wider diversity of environments and settings that it would be likely to encounter in a production sense; such as different weather and lighting conditions, different camera angles and potentially different surfaces such as brick. One easy way to do that would be to apply techniques such as data augmentation, which can apply different transformations[6] to an image to provide a wider diversity of features for the classifier to learn.
- Finally, the classifier should be benchmarked against traditional edge-detection approaches like Canny and Sobel.

Model Comparison

The final aspect of the project involved implementing the CNN structure reviewed back at the start on the concrete surface crack dataset. The Keras Applications module provides easy access to a number of deep learning models that have already been pre-trained, which means that building the network from scratch is unnecessary. The third version of the Inception network model was used for this part of the project because the first version was not one of the available networks. The architecture for version three is described in detail in Szegedy et al. (2016). Compared to version one, it had approximately three times as many parameters (24 million compared to the original 7 million) and was 100 layers deep. The main difference to this model from the previous iteration was the addition of batch normalisation in the auxiliary classifiers after the realisation that they acted more like regularisers, changed the optimiser to RMSprop and added factorised 7x7 convolutions (Raj, 2018, May 30 2018).

Whilst a reasonable amount of modification is permitted to these models, the author only changed the very top of the network so it could be applied to a binary classification problem with 227x227 pixel images instead of the 1,000-class ImageNet dataset, which used 224x224 images. In addition, the scaling factor needed to be changed from 0 to 1 to -1 to 1. Figure 9 provides a screenshot of the code used to implement the InceptionV3

[6] Including shifting the image slightly, rotating it, flipping it, increasing the level of noise or pixilation, or changing the contrast or colour saturation.

network with the modified top layer. The python script and accompanying Jupyter notebook can be found at Appendices E and F.

```
# define the inception model
def keras_model_fn(model_name=MODEL_NAME,
                   metrics=METRICS):

    # add the InceptionV3 base
    inception_model = InceptionV3(input_shape = (227, 227, 3),
                                  include_top = False, # Leave out the last fully connected layer
                                  weights = 'imagenet')
    # set all layers to be non-trainable (note that this could increase the possibility over overfitting)
    for layer in inception_model.layers:
        layer.trainable = False

    # flatten the output layer to one dimension
    flatten = Flatten()(inception_model.output)
    # add a fully connected layer with 256 hidden units, ReLU activation and he_uniform initialisation
    dense = Dense(256, activation='relu', kernel_initializer='he_uniform')(flatten)
    # add a dropout rate of 0.5
    dropout = Dropout(0.5)(dense)
    # add a final sigmoid layer for classification
    classification = Dense(1, activation='sigmoid')(dropout)

    # define the model
    model = Model(inputs=inception_model.input, outputs=classification)

    model.compile(optimizer=Nadam(learning_rate=0.0001), loss='binary_crossentropy', metrics=metrics)

    return model
```

Figure 9: InceptionV3 code

The InceptionV3 network was trained on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) dataset. The classes of images were very diverse and ranged from animals, to fruit, vegetables and manufactured foods, objects like cups and bicycles, flora, and landscapes.

	Proposed CNN				**InceptionV3**			
Dataset	Loss	Accuracy	AUC	F1	Loss	Accuracy	AUC	F1
Training	0.056	0.987	0.999	0.987	754.6	0.500	0.5	0.667
Validation	0.064	0.984	0.998	0.984	754.2	0.500	0.5	0.666
Test	-	0.889	0.889	0.894	-	0.533	0.5	0.696

Table 2: Performance metrics on the training, validation and test data

These metrics, along with the confusion matrices in figure 10 clearly showed that the InceptionV3 network performed very poorly on this dataset. On the one hand it, the poor performance could be explained by the fact that this dataset was very narrow in its focus and the InceptionV3 network may have never seen images quite like it before. However, it had been trained on a wide variety of images with a respectable Top-1 accuracy of 779.9% and top-5 accuracy of 93.7% so it would not have been unreasonable to assume that the performance would at least approach those sorts of values. Furthermore, Kim and Cho (2018) and Özgenel and Sorguç (2018) are examples of previous work validating this hypothesis. The other hypothesis was that the issue that occurred with the hyperparameter tuning of the proposed CNN re-occurred when training this model and therefore prevented the model from being able to accurately classify the images.

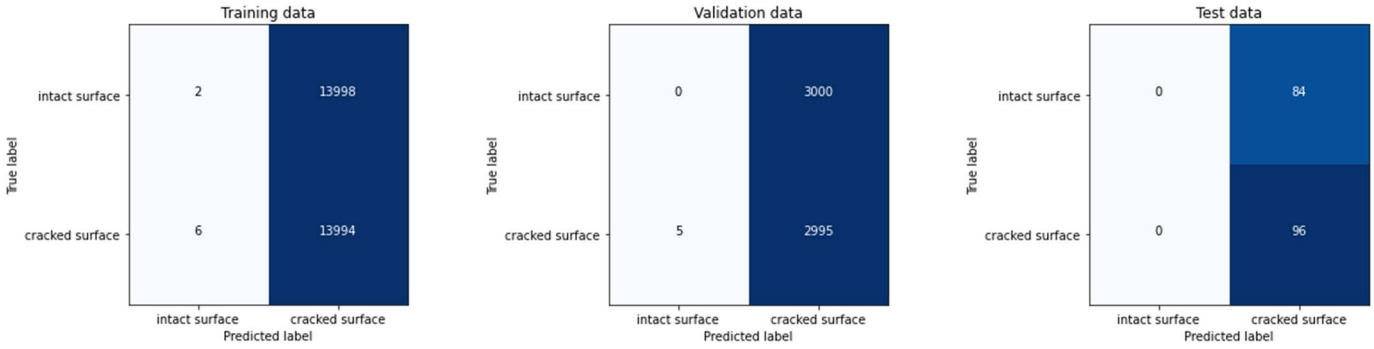


Figure 10: Confusion matrices for the training, validation and test data

Application on AWS

Evidence of model training and deployment

All data preparation, training and inference was performed on Amazon SageMaker and model training was monitored with CloudWatch. The data used was initially loaded into a local SageMaker notebook instance before being uploaded to an Amazon Simple Service (S3) bucket. Evidence of the successful training of all models described in this report, including those trained inside the hyperparameter optimisation job is shown in Figure 11. In addition, Figure 12 shows that both the proposed CNN and the Inception-based CNN were successfully deployed to an AWS SageMaker endpoint in order to determine performance on a 180-image subset of the full test dataset.

Training jobs

Actions ▼ | Create training job

Search training jobs

Status : Completed ✕ | Creation time after : Jun 09, 2021 07:31 UTC ✕

< 1 2 >

Name	Creation time	Duration	Status
A3-inception-training-job-2021-06-15-11-53-52	Jun 15, 2021 11:58 UTC	40 minutes	Completed
A3-HPO-job-2021-06-14-00-35-16-005-cfd9c5a4	Jun 14, 2021 03:16 UTC	an hour	Completed
A3-HPO-job-2021-06-14-00-35-16-004-e4b8bac3	Jun 14, 2021 02:41 UTC	33 minutes	Completed
A3-HPO-job-2021-06-14-00-35-16-003-403b8075	Jun 14, 2021 02:12 UTC	26 minutes	Completed
A3-HPO-job-2021-06-14-00-35-16-002-25503d0b	Jun 14, 2021 01:12 UTC	an hour	Completed
A3-HPO-job-2021-06-14-00-35-16-001-f3174099	Jun 14, 2021 00:36 UTC	33 minutes	Completed
A3-baseline-training-job-2021-06-13-23-23-31	Jun 13, 2021 23:24 UTC	an hour	Completed

Figure 11: Evidence of successful Amazon SageMaker training jobs

```
# configure and deploy a model endpoint
model_name = best_configuration_name

model_uri = "s3://{}/{}/output/{}/output/model.tar.gz".format(bucket, prefix, model_name)
model_endpoint_name = model_name + '-ep'

CNN_model = TensorFlowModel(
    model_data=model_uri,
    role=role,
    framework_version='2.1.0')
```

```
# deploy the model
CNN_model_predictor = CNN_model.deploy(
    initial_instance_count=1,
    instance_type='ml.m4.xlarge',
    endpoint_name=model_endpoint_name
)
```

```
update_endpoint is a no-op in sagemaker>=2.
See: https://sagemaker.readthedocs.io/en/stable/v2.html for details.

-------------!
```

```
# configure and deploy a model endpoint
model_name = best_configuration_name

model_uri = "s3://{}/{}/output/{}/output/model.tar.gz".format(bucket, prefix, model_name)
model_endpoint_name = model_name + '-ep'

inception_model = TensorFlowModel(
    model_data=model_uri,
    role=role,
    framework_version='2.1.0')
```

```
# deploy the model
inception_model_predictor = inception_model.deploy(
    initial_instance_count=1,
    instance_type='ml.m4.xlarge',
    endpoint_name=model_endpoint_name
)
```

```
update_endpoint is a no-op in sagemaker>=2.
See: https://sagemaker.readthedocs.io/en/stable/v2.html for details.

----------!
```

Figure 12: Evidence of successful endpoint creation and model deployment

Model deployment, monitoring and maintenance

The Jupyter notebooks used to prepare and pre-process the data, perform general investigation, and construct and run the python scripts for model training and inference were powered by a ml.t2.medium elastic computing (EC2) instance that is free for the first 250 hours of usage each month for the first two months. These are one of AWSs standard instances and suitable for general use as they are powered by 2 virtual CPUs and 5 GiB of memory. If the 250 hours is used up in a month, then this instance is very economical at around 6 US cents per hour in the Asia Pacific (Sydney) region (Amazon Web Services, n.d.-c).

For the slightly more resource-intensive task of model deployment and inference tasks, a ml.m4.xlarge instance was used. For the first two months of Amazon SageMaker usage, the first 50 hours of model training and first 150 hours of model inference with this EC2 instance is free, otherwise it is still a very economical 30 US cents per hour. The ml.m4.xlarge is one of the smaller standard training instances offered and is computationally is powered by four virtual CPUs and 16 GiB of memory. Model deployment was primarily done using model artifacts from previous successful training jobs. This significantly cut down on the deployment time as the model did not have to first be trained. It also meant that model deployment and test set performance evaluation did not have to occur in the same session as model training; that is, it could be hours or days later if necessary.

Finally, all model training was performed with an accelerated computing ml.p2.xlarge EC2 instance. These types of instances are designed for general-purpose compute applications using CUDA like Tensorflow and machine learning (Amazon Web Services, n.d.-b). This instance type provided access to a NVIDIA K80 GPU with 2,496 parallel processing cores and 12 GiB of GPU memory, 4 virtual CPUs and 61 GiB of memory. It cost USD1.93 for each hour of usage (Amazon Web Services, n.d.-a, n.d.-c).

Amazon SageMaker provides the user a wide range of tools with which to monitor the different jobs available within the service. For example, a top-level dashboard provides an overview of the number, status and types of jobs running, completed and failed, and type of jobs running and their type. This provides an entry-point into more detailed monitoring, including near real-time access to log data, computational resource usage and algorithm performance metric information.

If set up, performance metric information such as training and validation loss, accuracy, AUC, and precision and recall values are sent directly to CloudWatch as an (easier) alternative to Tensorboard. The former permits the construction of customised charts to monitor model training. For example, Figure 13 below shows examples of customised dashboards used to monitor algorithm performance metrics and computational resource utilisation during training of the initial CNN model.

Figure 13: CloudWatch dashboards for model training monitoring

Conclusion

This project had multiple parts. The first, and arguably smaller, task was to review a seminal computer vision paper from a list of three provided . The first option was Krizhevsky et al. (2012)'s AlexNet, the second was He et al. (2016)'s ResNet and the final option was Szegedy et al. (2015)'s GoogLeNet or InceptionV1. The author chose to review the latter paper.

InceptionV1 was a complex and heavily-engineered network that was attempting to push classifier performance boundaries on the widely-known ImageNet dataset by going both wider and deeper. Up until that point, many previous architecture designs had just stacked convolutional layers on top of one another to that aim. This goal was largely achieved, with the model achieving a top-5 error rate of less than 7% compared to previous years' achievements of 11-15% and far fewer parameters. However, the complexity of the model architecture meant that modifications could not be easily made and it could not be scaled up without compromising on its computational efficiency.

The second half of the project aimed to develop a computer vision classifier that could detect cracks in concrete surfaces and hence feed into a wider system to determine the severity of that crack. Organisations could then determine the resourcing required for remediation work (if any) without always requiring a physical inspection.

Whilst the model was able to perform very well on the provided dataset of 40,000 images from a Turkish university (achieving 88.9% accuracy and a 89.4% F1 score on the test set), the authors cautioned that additional work was required before it could even be considered in a production setting. Of most importance was the issue that a robust assessment on the test dataset could not be provided due to the limitations of AWS endpoint for predictions involving image data. The authors also recommended that the classifier be tested on a more diverse dataset to simulate what it might be exposed to in a production environment, although an intermediate step might be to apply data augmentation as per the example in Figure 14.

Figure 14: An example of image augmentation

The final task involved implementing the Inception architecture and training it on the concrete crack dataset. However, performance across all three datasets was disappointing and unexpected with respect to the type of performance reported by others applying pre-trained models to similar datasets. One hypothesis the author had regarding this was that the data was not being read in correctly, as a similar event had occurred during hyperparameter tuning of the proposed CNN model.

Appendix A – References

Ali, L., Alnajjar, F., Jassmi, H. A., Gocho, M., Khan, W., & Serhani, M. A. (2021). Performance evaluation of deep CNN-based crack detection and localization techniques for concrete structures. *Sensors*, *21*(5), 1688. https://www.mdpi.com/1424-8220/21/5/1688

Amazon Web Services. (n.d.-a). *Amazon EC2 instance types*. https://amzn.to/2R4DCKD

Amazon Web Services. (n.d.-b). *Amazon EC2 p2 instances*. https://aws.amazon.com/ec2/instance-types/p2/

Amazon Web Services. (n.d.-c). *Amazon SageMaker pricing*. https://aws.amazon.com/sagemaker/pricing/

Anwar, A. (2019, June 7). *Difference between AlexNet, VGGNet, ResNet and Inception*. Towards Data Science. https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96

Azimi, M., Eslamlou, A. D., & Pekcan, G. (2020). Data-driven structural health monitoring and damage detection through deep learning: State-of-the-art review. *Sensors*, *20*(10). https://doi.org/10.3390/s20102778

Brownlee, J. (2019, January 25). *Understand the impact of learning rate on neural network performance*. Machine Learning Mastery. https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

Cha, Y.-J., Choi, W., & Büyüköztürk, O. (2017). Deep learning-based crack damage detection using convolutional neural networks. *Computer-Aided Civil and Infrastructure Engineering*, *32*(5), 361-378. https://doi.org/10.1111/mice.12263

Civil Site Visit. (2019, May 20). *Types of cracks in concrete structures and its prevention*. https://civilsitevisit.com/types-of-cracks-in-concrete-structures/

Dung, C. V., & Anh, L. D. (2019). Autonomous concrete crack detection using deep fully convolutional neural network. *Automation in Construction*, *99*, 52-58. https://doi.org/https://doi.org/10.1016/j.autcon.2018.11.028

Flah, M., Suleiman, A. R., & Nehdi, M. L. (2020). Classification and quantification of cracks in concrete structures using deep learning image-based techniques. *Cement and Concrete Composites*, *114*, 103781. https://doi.org/https://doi.org/10.1016/j.cemconcomp.2020.103781

Géron, A. (2019). *Hands-on machine learning with scikit-learn, keras, and tensorflow: Concepts, tools, and techniques to Build Intelligent Systems* (2nd ed.). O'Reilly Media, Inc.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org

He, K., Zhang, X., Ren, S., & Sun, J. (2016, 27-30 June 2016). *Deep Residual Learning for Image Recognition* 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), https://ieeexplore.ieee.org/document/7780459

Kandel, I., & Castelli, M. (2020). The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset. *ICT Express*, *6*(4), 312-315. https://doi.org/https://doi.org/10.1016/j.icte.2020.04.010

Kim, B., & Cho, S. (2018). Automated Vision-Based Detection of Cracks on Concrete Surfaces Using a Deep Learning Technique. *Sensors*, *18*(10). https://doi.org/10.3390/s18103452

Kim, B., Yuvaraj, N., Sri Preethaa, K. R., & Arun Pandian, R. (2021). Surface crack detection using deep learning with shallow CNN architecture for enhanced computation. *Neural Computing and Applications*. https://doi.org/10.1007/s00521-021-05690-8

Krizhevsky, A., Sutskever, I., & Hinton, G. (2012). ImageNet classification with deep convolutional neural networks. *Neural Information Processing Systems*, *25*(2), 1106-1114. https://doi.org/10.1145/3065386

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436-444. https://doi.org/10.1038/nature14539

Meeedee Design Services. (2019, November 5). *Structural cracks: Causes and their identification*. https://www.meeedesignservices.com/2019/11/05/structural-cracks-causes-and-their-identification/

Mohan, A., & Poobal, S. (2018). Crack detection using image processing: A critical review and analysis. *Alexandria Engineering Journal*, *57*(2), 787-798. https://doi.org/https://doi.org/10.1016/j.aej.2017.01.020

Naincyjain. (2020, April 14). *Effect of batch size on training process and results by gradient accumulation*. Analytics Vidhya. https://medium.com/analytics-vidhya/effect-of-batch-size-on-training-process-and-results-by-gradient-accumulation-e7252ee2cb3f

Özgenel, Ç. F. (2019). Concrete crack images for classification. *Mendeley Data, V2*. https://doi.org/10.17632/5y9wdsg2zt.2

Özgenel, Ç. F., & Sorguç, A. G. (2018). Performance comparison of pretrained convolutional neural networks on crack detection in buildings. In J. Teizer, M. König, & T. Hartmann (Eds.), *Proceedings of the 35th International Symposium on Automation and Robotics in Construction (ISARC)* (pp. 693-700). International Association for Automation and Robotics in Construction (IAARC). https://doi.org/10.22260/ISARC2018/0094

Pandey, S. (2020, June 23). *How to choose the size of the convolution filter or Kernel size for CNN?* Analytics Vidhya. https://medium.com/analytics-vidhya/how-to-choose-the-size-of-the-convolution-filter-or-kernel-size-for-cnn-86a55a1e2d15

Prasad. (n.d.). *Durability of concrete [requirements and problems]*. https://www.structuralguide.com/durability-requirements-in-reinforced-concrete-design/

Raj, B. (2018, May 30). *A simple guide to the versions of the Inception network*. Towards Data Science. https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202

Ren, Y., Huang, J., Hong, Z., Lu, W., Yin, J., Zou, L., & Shen, X. (2020). Image-based concrete crack detection in tunnels using deep fully convolutional networks. *Construction and Building Materials*, *234*, 117367. https://doi.org/https://doi.org/10.1016/j.conbuildmat.2019.117367

Smola, A. (2019, March 8). *L13/1 Inception (GoogLeNet)* [Video]. YouTube. https://www.youtube.com/watch?v=jikAvn68mYY

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 2818-2826). IEEE. https://doi.org/10.1109/CVPR.2016.308

Szegedy, C., Wei, L., Yangqing, J., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. In

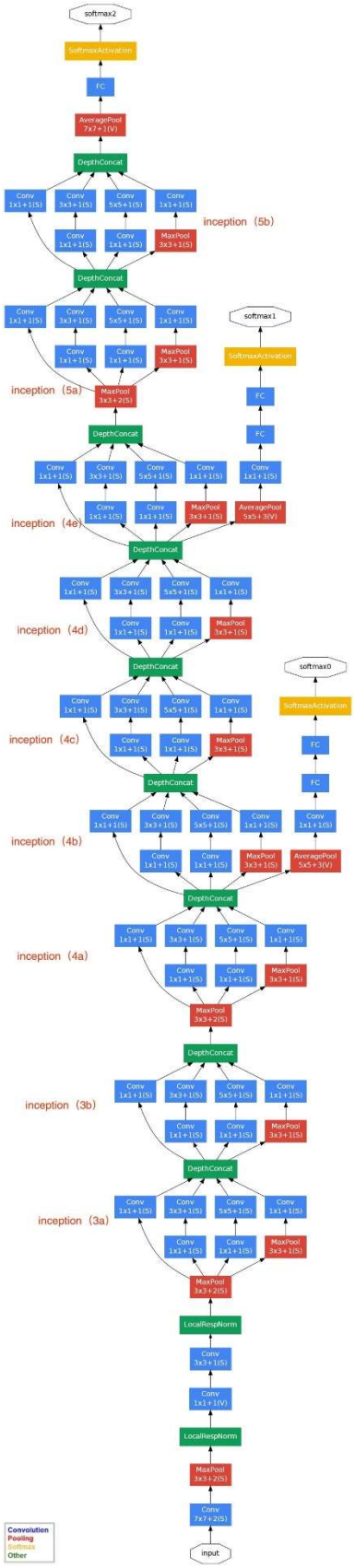
2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 1-9). IEEE. https://doi.org/10.1109/CVPR.2015.7298594

Wan, H., Gao, L., Su, M., Sun, Q., & Huang, L. (2021). Attention-Based Convolutional Neural Network for Pavement Crack Detection. *Advances in Materials Science and Engineering*, *2021*, 5520515. https://doi.org/10.1155/2021/5520515

Zhang, L., Yang, F., Zhang, Y. D., & Zhu, Y. J. (2016). Road crack detection using deep convolutional neural network. In *2016 IEEE International Conference on Image Processing (ICIP)* (pp. 3708-3712). IEEE. https://doi.org/10.1109/ICIP.2016.7533052

Appendix B – GoogLeNet Architecture Maps

Overall Schematic



Topology Details

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	β×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

These are attached as a separate zip file.