AI Capstone Project with Deep Learning

Crack Detection in Buildings

Chan Ngar Kay
murphy3@gamil.com

Abstract

Visual recognition is one of the key components in application artificial intelligent. The architecture of different neutral networks affects the accuracy and recall of detection systems for the existence of cracks in buildings. This work uses convolutional neural networks with transfer learning to detect whether an input image with or without cracks on it. The paper compares two pre-trained networks, ResNet50 and VGG16. Modified ResNet50 and VGG16 networks are trained on images from the dataset, and the results are compared. The validation accuracies for ResNet50 and VGG16 are 99.85% and 99.39% respectively. Apart from accuracy, the other performance matrices used in this work are precision and recall.Based on these performance matrices, shows that accurate affect detection is obtained from the two networks with ResNet50 being more accurate than VGG16.

1. Introduction and Background

In contemporary society, Artificial Intelligence (AI) is one of the hottest topics around the world. AI applications become more and more common in daily life. Simple as our daily communication apps in mobile phone, the speech to text and text to speech function which is the example of AI usage. AI as Augmented Intelligence, helping experts scale their capabilities as machines do the time-consuming work. AI learns by the machine learning models based on provided inputs and the desired outputs. AI also benefits to the world in diverse areas such as Healthcare, Education, Transcription, Customer Service, as well as Financial Fraud Prevention, and more.

Machine Learning is a subset of AI which uses computer algorithms to analyze data and make intelligent decisions. While Deep Learning is a specialized subset of Machine Learning, and layers algorithms to create a neural network enabling AI systems to learn from unstructured data and continue learning on the job.

In this AI capstone project with deep learning, we focus on the Image Classification by using Keras [1]. Keras is an API which follows best practices for reducing cognitive load. Keras is a central part of the TensorFlow ecosystem, and the

machine learning workflow, from data management to hyperparameter training to deployment solutions.

The project is to create and evaluate the two pre-trained models for classification of crack detection on buildings, as to train the visual recognition by AI for structural health monitoring. This paper also analyzes the accuracy of both models.

2. Problem Description

The identification of the location and existence of a crack on buildings is an important example of structural health monitoring and has received considerable attention. Most of the approaches, in the past, involves lots of manpower and time-consuming tasks. Nowadays, AI can help for the first step scanning from the images of building surfaces, and then detects if the crack appears on that area.

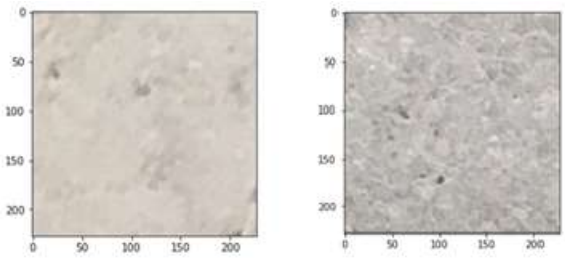
The challenges associated with the data set cracks can be confused with noise in the background texture or foreign objects in ho genius illumination and irregularities that can mislead the model for inaccurate output [2].

3. Description of Data

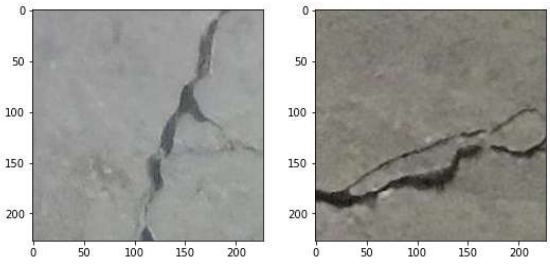
3.1 Data Collection

The data set consists of 40,000 concrete crack images of size 224 x 224 in RGB, in which, 20,000 with cracks and 20,000 with no crack. 75% of the images will be used for training the models, and the remaining 25% of the images will be used for validation. The image source is from Mendeley comm [3].

Example of images without cracks:



Example of images with cracks:



3.2 Data Preparation

The image data is representing 2 classes that 1 is positive for image with crack, 0 is negative for image without crack. Import the library ImageDataGenerator from Keras preprocessing.image. Then, the train image set for training the model can be created by apply the flow_from_directory function from the defined class ImageDataGenerator[4],

```
train_generator = data_generator.flow_from_directory(
    'concrete_data_week4/train',
    target_size=(image_resize, image_resize),
    batch_size=batch_size_training,
    class_mode='categorical')
```

```
Found 30001 images belonging to 2 classes.
```

Similarly, the validation_generator is created from the validation image set for validation steps when fitting the models.

```
validation_generator = data_generator.flow_from_directory(
    'concrete_data_week4/valid',
    target_size=(image_resize, image_resize),
    batch_size=batch_size_validation,
    class_mode='categorical')
```

```
Found 9501 images belonging to 2 classes.
```

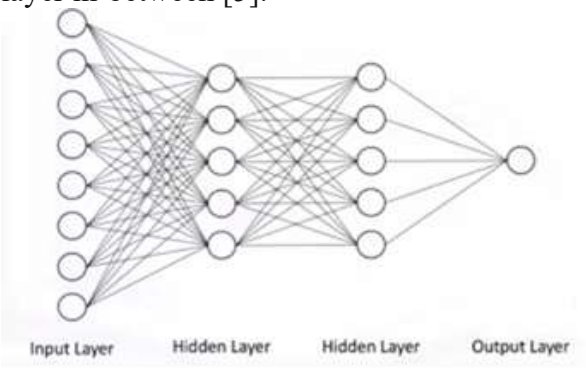
Lastly, , the test_generator is created from the test image set for the evaluation use.

```
test_generator = data_generator.flow_from_directory(
    "concrete_data_week4/test",
    target_size=image_size,
    batch_size=batch_size,
    class_mode="categorical",
)
```

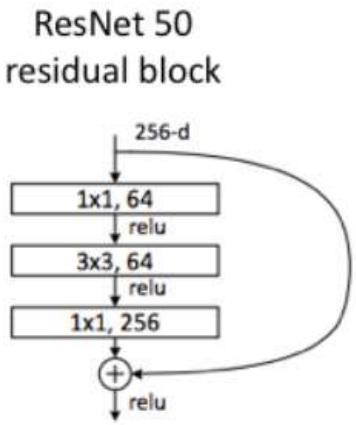
```
Found 500 images belonging to 2 classes.
```

4. Methodology

A Neural Network consists of an input and output layer with one or multiple hidden layer in-between [5].



In order to create model with different layers, a Sequential model class is used from Keras. The Sequential model is a linear stack of layers that a model can be created by passing a list of layer instances to the constructor.
ResNet is a short form of residual networks. ResNet50 is a variant of ResNet model which has 48 Convolution layers along with 1 MaxPool and 1 Average Pool layer. It has 3.8 x 10^9 Floating points operations. It is a widely used ResNet model. It is comparable to VGG-16 accept that Resnet50 has an additional identity mapping capability [6].



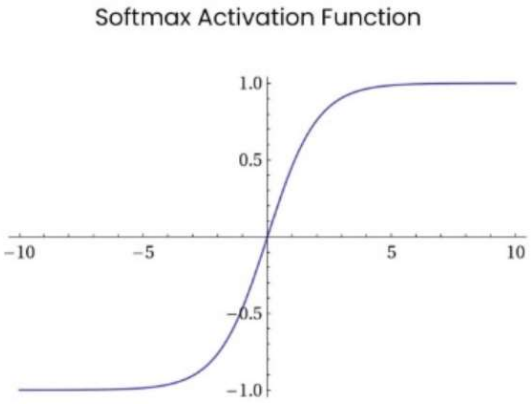
Now, this pre-trained model is used for creating the model for the first model of crack detection, ResNet50_model. To define an output layer and train it so that it is optimized for the image dataset. As to leave out the output layer of the pre-trained model the argument include_top is set to False.

```
ResNet50_model.add(ResNet50(
    include_top=False,
    pooling='avg',
    weights='imagenet',
    ))
```

Then, to define the output layer as a Dense layer, that consists of two nodes and uses the Softmax function as the activation function**.**

```
: ResNet50_model.add(Dense(num_classes, activation='softmax'))
```

The Softmax activation function compresses values to positive values between 0.0 and 1.0. Typically placed in output layers of networks used for classification, the Softmax neurons allow the prediction of outputs to certain classes.

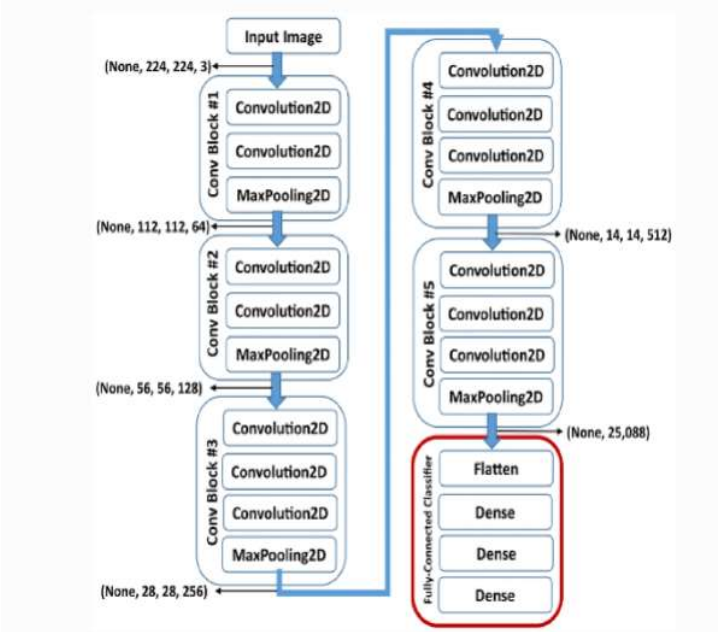


The ResNet50_model is composed of two sets of layers. The first set is the layers pertaining to ResNet50 and the second set is a single layer, which is the Dense layer.

Summary of the created ResNet50_model.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet50 (Model)             (None, 2048)              23587712
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 4098
=================================================================
Total params: 23,591,810
Trainable params: 4,098
Non-trainable params: 23,587,712
_________________________________________________________________
```

The VGG-16 network was trained on the ImageNet database. VGG16 is a convolution neural net (CNN) architecture. VGG16 focused on having convolution layers of 3x3 filter with a stride 1 and always used same padding and maxpool layer of 2x2 filter of stride 2. It follows this arrangement of convolution and max pool layers consistently throughout the whole architecture [7].



Like the previous model creation, using the Sequential model class to add the VGG16 layer and a Dense layer, that consists of two nodes and uses the Softmax function as the activation function**.**

```
VGG16_model.add(VGG16(
    include_top=False,
    pooling='avg',
    weights='imagenet',
    ))

VGG16_model.add(Dense(num_classes, activation='softmax'))
```

Summary of the created VGG16_model.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 512)               14714688
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 1026
=================================================================
Total params: 14,715,714
Trainable params: 1,026
Non-trainable params: 14,714,688
_________________________________________________________________
```

Finally, is to compile and fit the models. Using the “adam” as the optimizer, because it can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data.

5. Results

Fitting the models of configuration: num_epochs = 2, batch_size = 100

First round of fitting, fit the two models with steps_per_epoch=50, validation_steps=10.

Result for ResNet50

```
]: fit_history = model_ResNet50.fit_generator(
    training_generator,
    steps_per_epoch=50,
    epochs=num_epochs,
    validation_data=validation_generator,
    validation_steps=10,
    verbose=1,
)
```

```
Epoch 1/2
50/50 [==============================] - 1047s 21s/step - loss: 0.1091 - accuracy: 0.9556 - val_loss: 0.0271 - val_accuracy: 0.9890
Epoch 2/2
50/50 [==============================] - 1042s 21s/step - loss: 0.0166 - accuracy: 0.9958 - val_loss: 0.0164 - val_accuracy: 0.9940
```

Result for VGG16 model,

```
fit_history = model_vgg16.fit(
    training_generator,
    steps_per_epoch=50,
    epochs=num_epochs,
    validation_data=validation_generator,
    validation_steps=10,
    verbose=1,
)
```

```
Epoch 1/2
50/50 [==============================] - 2256s 45s/step - loss: 0.3720 - accuracy: 0.8346 - val_loss: 0.1107 - val_accuracy: 0.9790
Epoch 2/2
50/50 [==============================] - 2259s 45s/step - loss: 0.0795 - accuracy: 0.9850 - val_loss: 0.0734 - val_accuracy: 0.9810
```

Second round of fitting the two models with steps_per_epoch=100, validation_steps=10:

Result for ResNet50

```
: fit_history = model_ResNet50.fit(
    training_generator,
    steps_per_epoch=100,
    epochs=num_epochs,
    validation_data=validation_generator,
    validation_steps=10,
    verbose=1,
)
```

```
Epoch 1/2
100/100 [==============================] - 1882s 19s/step - loss: 0.0119 - accuracy: 0.9975 - val_loss: 0.0095 - val_accuracy: 0.9990
Epoch 2/2
100/100 [==============================] - 1901s 19s/step - loss: 0.0067 - accuracy: 0.9986 - val_loss: 0.0104 - val_accuracy: 0.9960
```

Result for VGG16

```
fit_history = model_vgg16.fit(
    training_generator,
    steps_per_epoch=100,
    epochs=num_epochs,
    validation_data=validation_generator,
    validation_steps=10,
    verbose=1,
)
```

```
Epoch 1/2
100/100 [==============================] - 4191s 42s/step - loss: 0.0463 - accuracy: 0.9905 - val_loss: 0.0355 - val_accuracy: 0.9900
Epoch 2/2
100/100 [==============================] - 4202s 42s/step - loss: 0.0318 - accuracy: 0.9923 - val_loss: 0.0218 - val_accuracy: 0.9940
```

Third round of fitting the two models with

steps_per_epoch_training = len(train_generator)

steps_per_epoch_validation = len(validation_generator)

Result for ResNet50

```
fit_history = ResNet50_model.fit(
    train_generator,
    steps_per_epoch=steps_per_epoch_training,
    epochs=num_epochs,
    validation_data=validation_generator,
    validation_steps=steps_per_epoch_validation,
    verbose=1,34
)
```

```
Epoch 1/2
301/301 [==============================] - 9640s 32s/step - loss: 0.0379 - accuracy: 0.9853 - val_loss: 0.0085 - val_accuracy: 0.9975
Epoch 2/2
301/301 [==============================] - 9850s 33s/step - loss: 0.0066 - accuracy: 0.9985 - val_loss: 0.0086 - val_accuracy: 0.9985
```

Result for VGG16

```
fit_history = model.fit_generator(
    train_generator,
    steps_per_epoch=steps_per_epoch_training,
    epochs=num_epochs,
    validation_data=validation_generator,
    validation_steps=steps_per_epoch_validation,
    verbose=1
)
```

```
301/301 [==============================] - 19940s 66s/step - loss: 0.1562 - accuracy: 0.9370 - val_loss: 0.0372 - val_accuracy: 0.9921
Epoch 2/2
301/301 [==============================] - 19902s 66s/step - loss: 0.0272 - accuracy: 0.9939 - val_loss: 0.0206 - val_accuracy: 0.9945
```

Consolidate the 3 round results, compare the results for the 2 two models as below. Both models can perform very well on the classification of crack detection with accuracy for ResNet50 is 99.76% and for VGG16 is average 99.047%. Using the evaluate function by passing the test_generator as input, the accuracy for ResNet50 is 100% and for VGG16 is average 99.467%.

			1st epochs			2nd epochs			Evaluate	
Model	steps_per_epoch	validation_steps	Time (s)	Loss	Accuracy	Time (s)	Loss	Accuracy	Loss	Accuracy
ResNet50	50	10	1047	0.1091	0.9556	1042	0.0166	0.9958	0.00818	1
VGG16	50	10	2256	0.372	0.8346	2259	0.0795	0.985	0.05824	0.992
ResNet50	100	10	1882	0.0119	0.9975	1901	0.0067	0.9986	0.00346	1
VGG16	100	10	4191	0.0463	0.9905	4202	0.0318	0.9923	0.02158	0.996
ResNet50	301	96	9640	0.0379	0.9853	9850	0.0066	0.9985	0.0021	1
VGG16	301	96	19940	0.1562	0.937	19902	0.0272	0.9939	0.0178	0.996

The above table shows the models performance under the same number of epoch, when changing the epoch to 10 and compare the results again.

Result from ResNet50:

```
fit_history = ResNet50_model.fit(
    train_generator,
    steps_per_epoch=steps_per_epoch_training,
    epochs=num_epochs,
    validation_data=validation_generator,
    validation_steps=steps_per_epoch_validation,
    verbose=1
)
```

```
Epoch 1/10
30/30 [==============================] - 678s 23s/step - loss: 0.1701 - accuracy: 0.9310 - val_loss: 0.0197 - val_accuracy: 0.9967
Epoch 2/10
30/30 [==============================] - 675s 22s/step - loss: 0.0241 - accuracy: 0.9913 - val_loss: 0.0229 - val_accuracy: 0.9900
Epoch 3/10
30/30 [==============================] - 679s 23s/step - loss: 0.0144 - accuracy: 0.9960 - val_loss: 0.0149 - val_accuracy: 0.9978
Epoch 4/10
30/30 [==============================] - 681s 23s/step - loss: 0.0102 - accuracy: 0.9990 - val_loss: 0.0092 - val_accuracy: 0.9989
Epoch 5/10
30/30 [==============================] - 654s 22s/step - loss: 0.0124 - accuracy: 0.9973 - val_loss: 0.0103 - val_accuracy: 0.9978
Epoch 6/10
30/30 [==============================] - 684s 23s/step - loss: 0.0120 - accuracy: 0.9967 - val_loss: 0.0107 - val_accuracy: 0.9967
Epoch 7/10
30/30 [==============================] - 681s 23s/step - loss: 0.0081 - accuracy: 0.9987 - val_loss: 0.0110 - val_accuracy: 0.9944
Epoch 8/10
30/30 [==============================] - 659s 22s/step - loss: 0.0086 - accuracy: 0.9976 - val_loss: 0.0083 - val_accuracy: 0.9978
Epoch 9/10
30/30 [==============================] - 660s 22s/step - loss: 0.0046 - accuracy: 0.9997 - val_loss: 0.0058 - val_accuracy: 0.9978
Epoch 10/10
30/30 [==============================] - 681s 23s/step - loss: 0.0097 - accuracy: 0.9973 - val_loss: 0.0056 - val_accuracy: 0.9978
```

Result from VGG16:

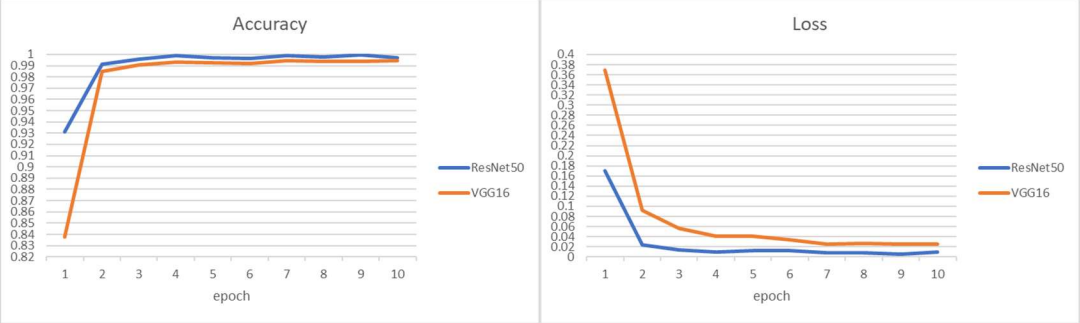
```
history_vgg16 = VGG16_model.fit_generator(
    train_generator,
    steps_per_epoch=steps_per_epoch_training,
    epochs=num_epochs,
    validation_data=validation_generator,
    validation_steps=steps_per_epoch_validation,
    verbose=1,
)
```

```
Epoch 1/10
30/30 [==============================] - 1448s 48s/step - loss: 0.3697 - accuracy: 0.8377 - val_loss: 0.1406 - val_accuracy: 0.9700
Epoch 2/10
30/30 [==============================] - 1450s 48s/step - loss: 0.0925 - accuracy: 0.9847 - val_loss: 0.0643 - val_accuracy: 0.9900
Epoch 3/10
30/30 [==============================] - 1448s 48s/step - loss: 0.0558 - accuracy: 0.9907 - val_loss: 0.0477 - val_accuracy: 0.9900
Epoch 4/10
30/30 [==============================] - 1453s 48s/step - loss: 0.0404 - accuracy: 0.9930 - val_loss: 0.0411 - val_accuracy: 0.9922
Epoch 5/10
30/30 [==============================] - 1453s 48s/step - loss: 0.0415 - accuracy: 0.9923 - val_loss: 0.0352 - val_accuracy: 0.9922
Epoch 6/10
30/30 [==============================] - 1454s 48s/step - loss: 0.0336 - accuracy: 0.9917 - val_loss: 0.0303 - val_accuracy: 0.9956
Epoch 7/10
30/30 [==============================] - 1454s 48s/step - loss: 0.0253 - accuracy: 0.9947 - val_loss: 0.0261 - val_accuracy: 0.9956
Epoch 8/10
30/30 [==============================] - 1451s 48s/step - loss: 0.0273 - accuracy: 0.9940 - val_loss: 0.0225 - val_accuracy: 0.9956
Epoch 9/10
30/30 [==============================] - 1458s 49s/step - loss: 0.0249 - accuracy: 0.9940 - val_loss: 0.0337 - val_accuracy: 0.9911
Epoch 10/10
30/30 [==============================] - 1452s 48s/step - loss: 0.0245 - accuracy: 0.9943 - val_loss: 0.0216 - val_accuracy: 0.9967
```

After 10 epochs for the two models, the following results of loss and accuracy data is found.

	ResNet50			VGG		
epoch	Time (s)	Loss	Accuracy	Time (s)	Loss	Accuracy
1	678	0.1701	0.931	1448	0.3697	0.8377
2	675	0.0241	0.9913	1450	0.0925	0.9847
3	679	0.0144	0.996	1448	0.0558	0.9907
4	681	0.0102	0.999	1453	0.0404	0.993
5	654	0.0124	0.9973	1453	0.0415	0.9923
6	684	0.012	0.9967	1454	0.0336	0.9917
7	681	0.0081	0.9987	1454	0.0253	0.9947
8	659	0.0086	0.9976	1451	0.0273	0.994
9	660	0.0046	0.9997	1458	0.0249	0.994
10	681	0.0097	0.9973	1452	0.0245	0.9943

From each round of epoch, the ResNet50 model performs better than VGG16 in terms of accuracy and recall.



6. Discussions

For both models perform as expected with accuracy more than 99% and loss less than 0.06. However, their costs are different. Below table shows the time consumed per step in second.

Model	Avg Time/ Step
ResNet50	14.06
VGG16	29.24

Time consumed per each of model VGG16 is nearly double that of ResNet50. Also, for ResNet50 model after fitting with 100 steps per epoch, it reaches almost the same accuracy as fitting with 301 steps per epoch, where for model VGG16, the model is till improving the accuracy after 301 steps per epoch. Based on all the performance matrices mentioned here, we have shown that the two models are capable of detecting cracks with very high accuracy, precision and recall, with ResNet50 performing slightly better than the VGG16.

7. Conclusion

Two pre-trained models ResNet50 and VVG16, were used, and their top fully connected layer was modified. We have successfully compared ResNet50 and VGG16 which demonstrated the accuracy on crack detection, the 2 pre-trained networks give very high accuracy, precision and recall values. The validation accuracies for ResNet50 and VGG16 are 99.85% and 99.39% respectively. In conclusion, ResNet50 achieves the highest precision and recall and takes much lesser epochs to train. In addition, ResNet50 also performs a cost saving characteristic during the processing of fitting model.

Hence, CNN with transfer learning helps to identify the various affect states very accurately thereby improving the interaction between humans and computers. We could further probe if the same technique of using CNN with transfer learning can help solve the problem of occlusions.

8. Reference:

1 Saeed Aghabozorgi, Joseph Santarcangelo, Machine Learning with Python, Coursera, 2019

2 Alex Aklson, Joseph Santarcangelo, AI Capstone Project with Deep Learning, Coursera, 2019

3 X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks, 2010.

4 A. Krizhevsky. Learning multiple layers of features from tiny images. Tech Report, 2009.

5 Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for

fast feature embedding. arXiv:1408.5093, 2014.

6 K. He, X. Zhang, S. Ren, and J. Sun, Deep Residual Learning for Image Recognition, Cornell University 2015

7 Karen Simonyan, Andrew Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition, Cornell University 2015