Udacity: Follow Me Report

Shane Reynolds

June 3, 2018

Contents

1 Introduction

Computer vision is a subset of robotic perception - it has been defined as the development of autonomous systems which can perform tasks achieved by human visual systems (Huang, 1996). This means the acquisition of digital image data from an optical camera, and some type of interpretation of the acquired image. A simple example of a task that is routinely performed by a human visual system, which is sought for computer vision systems, is answering the question: *Is there a puppy in Figure 1?*, or *Where is the puppy in Figure 1?*.

Figure 1: Computer vision is interested in answering questions such as *Is there a puppy in the image?* or *Where is the puppy in the image?*

There are many sub-fields of computer vision such as scene reconstruction, event detection, video tracking, object recognition, 3D pose estimation, and motion estimation. This paper will focus on classification, using an approach called semantic segmentation. Shelhamer, Long and Darrell (2016) define semantic segmentation as a method of inference which is able to categorise fine image details. This is achieved by classifying each pixel in the image, and labelling it with the class of its enclosing object or region. A Fully Convolutional Neural Network (FCN) is proposed as the architecture to implement semantic segmentation. An FCN model was trained and implemented on a robotic agent.

Figure 2: The UAV agent in the simulated environment.

Figure 3: The *hero* can be seen coloured in red, with the UAV agent following her.

The agent was tasked with the identification (and subsequent tracking) of an individual, known as the *hero*, in a 3D simulated environment built with Unity. The 3D simulated environment is a small city consisting of buildings, roads, elevated highways, and vegetation. The robotic agent is an unmanned aerial vehicle (UAV), as shown in Figure 2, which is fitted with a panning optical camera. The agent roams the simulation until it is able to locate the hero using the trained FCN, at which point the UAV will track the hero. The hero is a simulated person, as seen in Figure 3. To

satisfy the assignment criteria, the proposed FCN model must achieve above a 40% benchmark for an intersection over union metric.

2 Network Architecture

FCNs are widely used for computer vision applications, and are a type of Artificial Neural Network (ANN). ANNs are computational models which, once trained on a dataset, can be used to make classification predictions, or value estimations, based on a set of feature inputs that the model has been trained on. A typical fully connected feed-forward ANN consists of an input layer, one or more hidden layers, and an output layer, as shown in Figure 4. Hidden layers are made up of multiple nodes. The nodes themselves contain a non-linear activation function, such as a sigmoid or ReLU, and receive weighted input from the previous layers in the model. The inputs from the previous layer, and the non-linear activation of a node form a computational element called a neuron (also known as a perceptron) - these can be loosely thought of as decision making elements. An example of a neuron can seen in Figure 5.

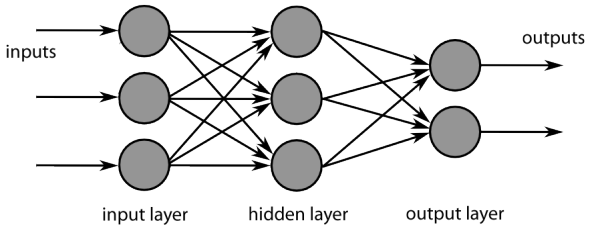

Figure 4: A feed-forward artificial neural network consists of an input layer, which receives feature inputs, some hidden layers, and an output layer for classification.

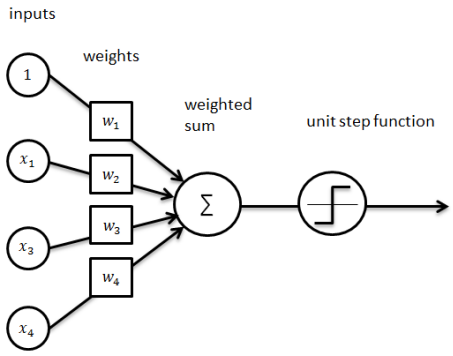

Figure 5: The structure of a neuron (perceptron) includes weighted inputs from the previous layer, and a non-linear activation function.

Changing the weights in a neuron changes the neurons's contribution to the model, which in turn affects the overall model output. Weight changes occur during model training, which uses large volumes of labelled data to adjust the weights. Hidden layers are important because they allow highly non-linear models to be constructed, providing an approach for estimating complex phenomena which may be difficult to model with classical approaches, or computationally intractable. Generally, the more hidden layers, the more non-linear the model. Network architectures with multiple hidden layers have become so wide spread that the term Deep Neural Network (DNN) was coined to describe feed-forward ANNs which use two or more hidden layers. It must be noted that whilst increased non-linearity may allow us to model more complex phenomenon, making the ANN deeper does not guarantee increased model performance. This is mainly due to the fact that deeper models may over-fit the data during training, resulting in a failure to generalise on test and validation data sets. Fully connected feed-forward DNNs have proven effective in computer vision classification problems, such as optical character recognition. One of the most widely cited examples of this is a feed-forward DNN performing classification on the MNIST dataset. The MNIST dataset contains handwritten digits, from 0 to 9, and is considered a benchmark for measuring neural net classification performance for the optical character recognition problem. Table 1, taken from paper by Ciresan, Meier, Gambardella, and Schmidhuber (2010), shows a table of feed-forward DNNs with varying numbers of hidden layers, and hidden layer depth.

Table 1: Reproduced from Ciresan, Meier, Gambardella, and Schmidhuber (2010) - DNN architectures of varying size for classifying the MNIST data set, and the associated performance of each network.

Architecture (number of neurons in each layer)	**Test Error Best Validation [%]**	**Best Test Error [%]**	**Simulation Time [min]**	**Weights [Millions]**
1000, 500, 10	0.49	0.44	23.4	1.34
1500, 1000, 500, 10	0.46	0.40	44.2	3.26
2000, 1500, 1000, 500, 10	0.41	0.39	66.7	6.69
2500, 2000, 1500, 1000, 500, 10	0.35	0.32	114.5	12.11
9×1000, 10	0.44	0.43	107.7	8.86

Notably, every model listed presents an error rate of less than 1%. Generally, the deeper a network, the better the model's predictive performance, although this is not always the case as previously outlined. Figure 5, taken from the same paper, shows a small set of the misclassified images. Despite the misclassification it can be seen that, in most cases, the handwritten digit bears a high resemblance to the predicted value, and that the second prediction is generally correct. Remarkably, DNNs are not considered state of the art for image classification problems - even with simple tasks like MNIST classification. This is due to model inefficiencies that arise from image variation in the spatial domain.

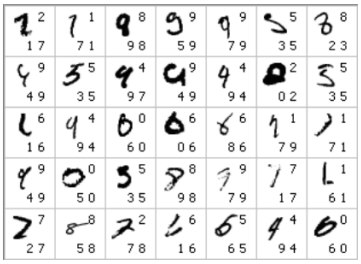

Figure 6: Misclassified hand written digits by the top performing DNN from Ciresan, Meier, Gambardella, and Schmidhuber (2010). The digit in the top right hand corner of the box is the observation label, and the two digits in the bottom right hand corner are the predictions from the DNN model.

The problem can be better understood by considering Figures 7 and 8. Figure 7 shows an image with a puppy on the left, and Figure shows an image with a puppy on the right. Suppose we create a simple model to classify whether an image has a puppy in it or not, and assume we train this model with lots of images like the one shown in Figure 7. If the trained model was then used to classify images like the one shown in Figure 8 it would perform poorly. This is because our model would have only learned to classify pixel features on the left side of the picture with puppies, which says nothing about identifying a puppy on the right hand side of an image. Put simply, there is no *translational invariance* in the model.

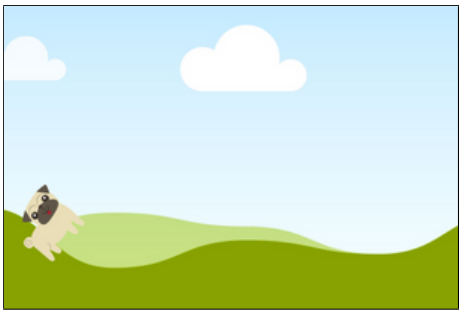

Figure 7: An image in which a puppy is located on the left hand side of the image.

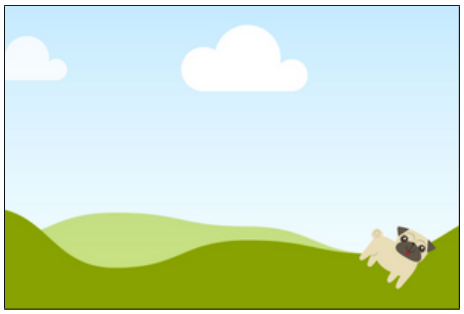

Figure 8: An image in which a puppy is located on the right hand side of the image.

2.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a class of ANN, which has an underlying network structure which is better at learning shapes, edges, and colours meaning it is less reliant on the spatial location of a classification object in an image. Recall that vanilla feed-forward neural nets only have neuron connections from the previous layer, and there are no connections from neurons in the same layer - weights are not shared. In contrast, CNNs share neuron weights by using filters which are convolved over an input image. Consider a raw input image of say 32 $\times$ 32 pixels, with a depth of 3 colour channels, as shown in Figure 9.

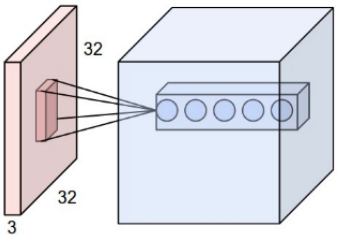

Figure 9: The raw image input is 32 $\times$ 32, with a 3-channel depth (R,G,B). The filter is a 3 $\times$ 3 patch with the same depth as the input. The filter is convolved over the image using some stride - each convolution creates a single element output which forms part of the 2D activation map (i.e. the output). There are K filters convolved over the image, a parameter chosen as part of the architecture, and the output volume represents the stacked 2D activation maps.

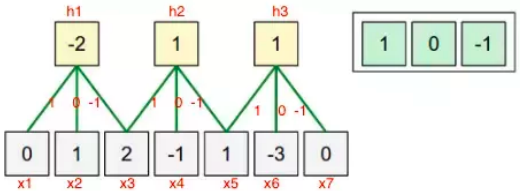

Figure 10: An simple example of weight sharing that takes place in a CNN. The white 1D array represents the input image, and the blue 1D array represents the filter. The filter (blue) is convolved across the input (white) using a stride of 2. The convolved output, which represents the activation map, can be seen in yellow. This architecture allows for the sharing of the weights in the model.

The convolving filter, which contains the model weights, is the small patch which is incident on the raw input image surface. The image section in contact with the filter is called the receptive field - this changes as the filter convolves an image. Filter width and height are parameters chosen as part of the network architecture, and filter depth is identical to the input image depth. The filter is moved around the image according to the number of pixels in each stride. After each movement, the filter weights are multiplied by the receptive field and added together - this makes up a single entry in the 2D activation map which forms part of the output volume. The width and height of the output volume are dependent on the stride, and the type of padding used during the convolution. Finally, the output depth is dependent on the number of filters specified in the network architecture - typically this parameter is denoted as K. The output volume is simply the stacked 2D activation maps from each filter. It is the convolutional process, whereby the filter weights are used over an entire image, that provide the weight sharing seen in CNNs. Figure 10 provides a simple example of how this works. In this example, the filter is a 1D array (shown in blue), and the input image is also a 1D array (shown in white). The filter (blue) is convolved over the input image (white) with a stride of 2. The filter weights are multiplied with the image values, and added together to form the output volume - a 1D array (shown in yellow).

CNNs provide superior performance over DNNs in the image classification domain. This was demonstrated on the MNIST dataset by LeCun (XXXX) with his LeNet (pictured in Figure 11), and again by Krizhevsky (XXXX) on the ImageNet dataset with AlexNet (shown in Figure 12). Whilst CNNs can achieve noteworthy performance in the task of image classification, there is a notable performance loss when they are re-tasked with pixel by pixel classification known as semantic segmentation. This is due to the fact that fully connected layers don't preserve spatial information.

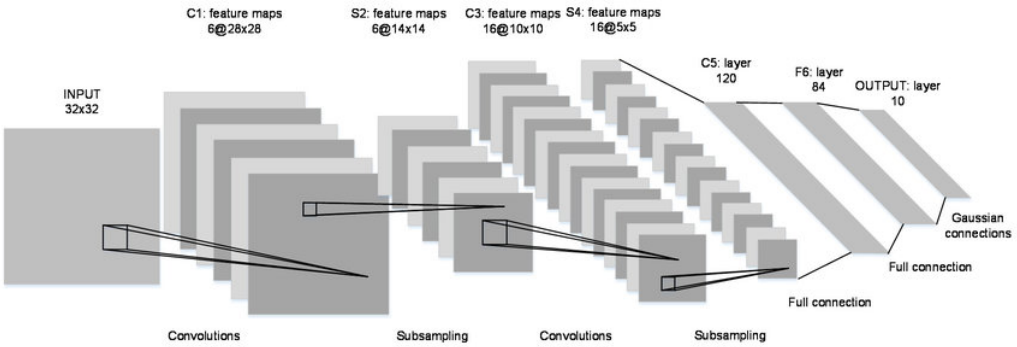

Figure 11: text

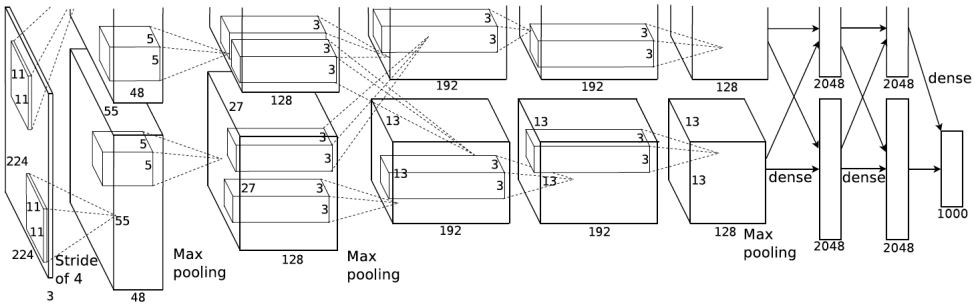

Figure 12: text

2.2 Fully Convolutional Neural Networks

FCNs preserve spatial information, and represent a state-of-the-art approach to semantic segmentation. Structurally, they can be thought of as two distinct parts: encoders and decoders. This is shown in Figure 13. The encoder is comprised of several convolutional layers, which are typically arranged to progressively concentrate the spatial domain, and increase the number of channels in the image. These different layers of spatial compression are useful for training the model on different image resolutions. The decoder, in contrast, upsamples encoder compressions, restoring the spatial information to the output - typically the final layer restores the output to the initial input image dimensions, before being passed to a convolutional layer with a softmax activation function.

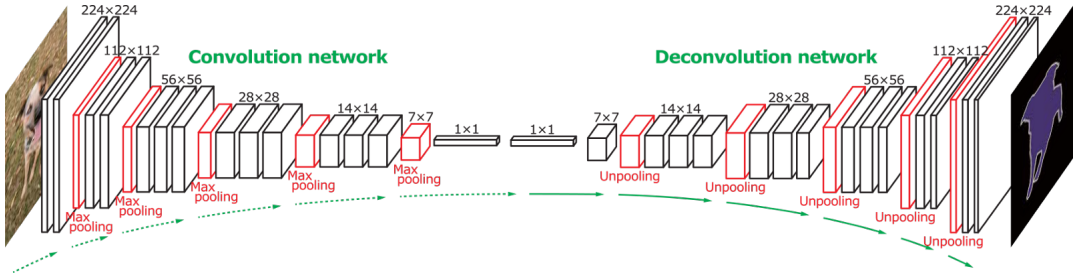

Figure 13: Structurally FCNs can be thought of as an encoder (convolutional network) connected to a decoder (deconvolutional network)

FCNs are often make use of 1×1 convolutions, which are just a 2D convolution of patch size 1×1, and a stride of 1 (NETWORK IN NETWORK). The spatial dimensions of the convolved image are preserved in the output volume, but the desired number of filters (K) changes the output volume depth. Indeed this is one of the main incentives to use 1×1 convolutions - they are computationally cheap, and can reduce the image depth making subsequent operations less expensive. One approach that has been used is to intersperse convolutional layers with 1×1 convolutional layers, to reduce the depth of the image being convolved by larger patches (e.g. 3×3, or 5×5). An example of this can be seen in Figure XXXX. Of course, this is not the only way that 1×1 convolutional layers can be used. They can also be used to add more depth to a model, and additional non-linearity since the 1 *times* 1 layer contains an activation function like ReLU. Using 1×1 convolutions is considered a cheap way of doing this.

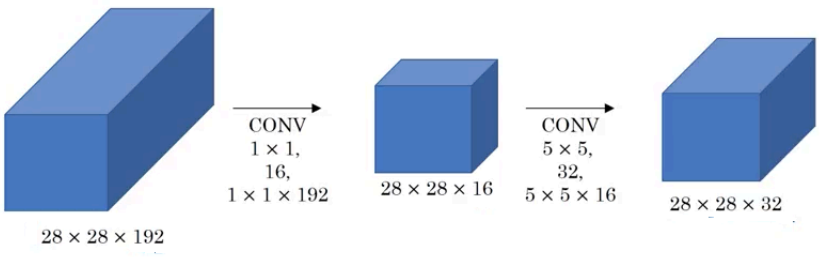

Figure 14: The input is $28 \times 28 \times 192$ and the desired 2D convolution is a 5×5 patch using 32 filters. This would result in a total of XXXX multiplications. If a 1×1 convolution is used to reduce the depth of the input image from 192 to 16 before performing the 2D convolution with the 5×5 patch, this reduces the total number of multiplications to XXXX.

Finally, FCNs see an improvement in model performance with the use of something called skip connections. A skip connection attaches a residual output in an FCN architecture to an output occurring later in the network. Their use was originally proposed by XXXX XXXXX and XXXXX (XXXXX) who found that adding more layers to their model *worsened* the performance during training. This

work led to the development of the well known Resnet architecture. The skip connections essentially concatenate the 2 layers by adding the outputs - an example of the operation can be seen in Figure XXXX.

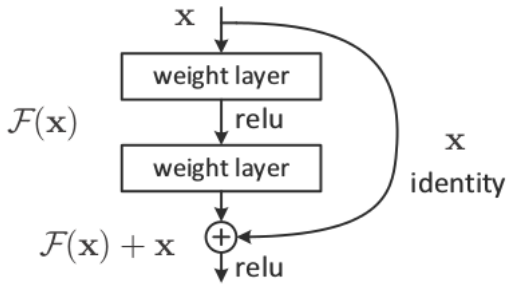

Figure 15: An example of a skip connection - the residual output in the network literally skips a section of the network and is concatenated with a later output of the model.

2.3 Proposed Architectures & Implementation

The information presented so far makes a compelling case for the use of FCNs to perform the semantic segmentation task, however, there is some uncertainty as to what model architecture should be employed. State of the art FCNs involve multiple 2D convolution layers interspersed with pooling layers in the encoder. The decoder consists of up-sampling layers interspersed with un-pooling layers. Additionally, these state of the art networks make use of 1×1 convolutions, and skip connections.An example of this can be seen in Figure XXXX, which shows an FCN network called SegNet taken from a paper by XXXX, XXXX, and XXXX (XXXX).

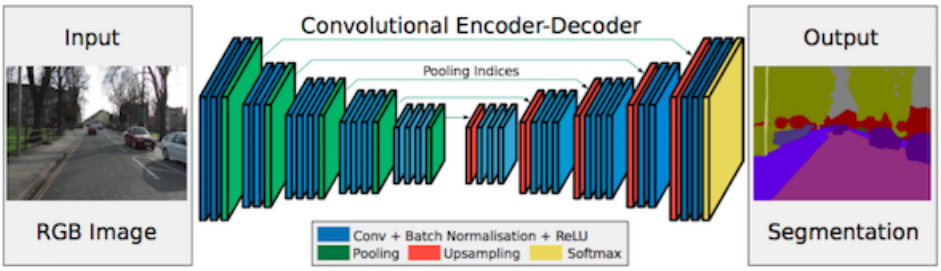

Figure 16: text

FCN networks with many layers, like the original FCN architecture by XXXX, XXXX, and XXXX (XXXX) and SegNet, are used to provide semantic segmentation on images with high visual complexity taken from the real world. In contrast, the models proposed in this paper are tasked with semantic segmentation of images taken from the 3D environment simulated in Unity. Given this, it may not be appropriate to adopt the depth seen in state-of-the-art FCN architectures since reasonable performance may be achieved with smaller models. Moreover, given that the dataset from the simulation only consists of approximately 4000 images (about half of the 9,993 found in the VOC2012 segmentation data set), a more complex model may result in worse performance. In this light, three different FCN architectures are proposed for investigation: a deep model with 1×1 convolutions; a shallow model with 1×1 convolutions; and a shallow model without 1×1 convolutions. These models are discussed in more detail in Sections 2.3.2, 2.3.3, and 2.3.4 respectively.

2.3.1 Tensorflow Implementation

To make the model implementation easier, two functions were developed: `encoder_block()` (shown in Listing 1), and `decoder_block()` (shown in Listing 2). An interesting feature of the `encoder_block()` is that it uses the a seperable 2D convolution. The reason for this is that separable convolutions are more computationally efficient. To see this, consider a 3×3 patch with one filter ($K = 1$) in a single convolution. The number of operations to perform this is XXXX PROVIDE EXPLANATION. A matrix is seperable if it can be written as follows:

$$\begin{bmatrix} A \cdot a & A \cdot b & A \cdot c \\ B \cdot a & B \cdot b & B \cdot c \\ C \cdot a & C \cdot b & C \cdot c \end{bmatrix} = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \cdot \begin{bmatrix} a & b & c \end{bmatrix} \tag{1}$$

The definition shown in (1) can be used to first undertake a convolution with XXXX, and then convolve the result of the first convolution with XXXX. This results in a total of XXXX operations. Given that FCNs have to perform this type of operation many times to convolve an entire image, there are significant computational savings made by using separable 2D convolutions.

Listing 1: text

```
def encoder_block(input_layer, filters, strides):

    # TODO Create a separable convolution layer using the separable_conv2d_batchnorm() function.
    output_layer = separable_conv2d_batchnorm(input_layer, filters, strides)

    return output_layer
```

The `decoder_block()` function takes three arguements, two of them being layers. The smallest input layer is upsampled using the `bilinear_upsample()` function - this simply doubles the size of the small layer, and fills the additional spaces with copies of the small layer rows and columns. The second layer input is a skip connection which comes from the encoder of the FCN. The upsampled layer and the layer from the skip connection are added together using a concatenate function. Note that care must be taken to ensure there is no mismatch in the layer dimensions. The concatenated layers are then passed through a 2D convolution to create the final layer.

Listing 2: text

```
def decoder_block(small_ip_layer, large_ip_layer, filters):

    # TODO Upsample the small input layer using the bilinear_upsample() function.
    small_ip_upsample = bilinear_upsample(small_ip_layer)

    # TODO Concatenate the upsampled and large input layers using layers.concatenate
    concat_layer = layers.concatenate([small_ip_upsample, large_ip_layer])

    # TODO Add some number of separable convolution layers
    output_layer = separable_conv2d_batchnorm(concat_layer, filters)

    return output_layer
```

Both the `encoder_block()` and `decoder_block()` functions use ReLUs as the non-linear activation functions, and both functions employ batch normalisation. Batch normalisation is a technique proposed by XXXX and XXXX (XXXX) which helps to improve the optimisation process which occurs during training. Additionally, it provides some regularisation to the model as well.

2.3.2 Model 1: Deep Model with 1 × 1 Convolution

The proposed deep model with 1 × 1 convolutions is comprised of 9 layers, including the input layer - the network architecture can be seen in Figure XXXX, and a detailed description of each layer can be found in Table XXXX

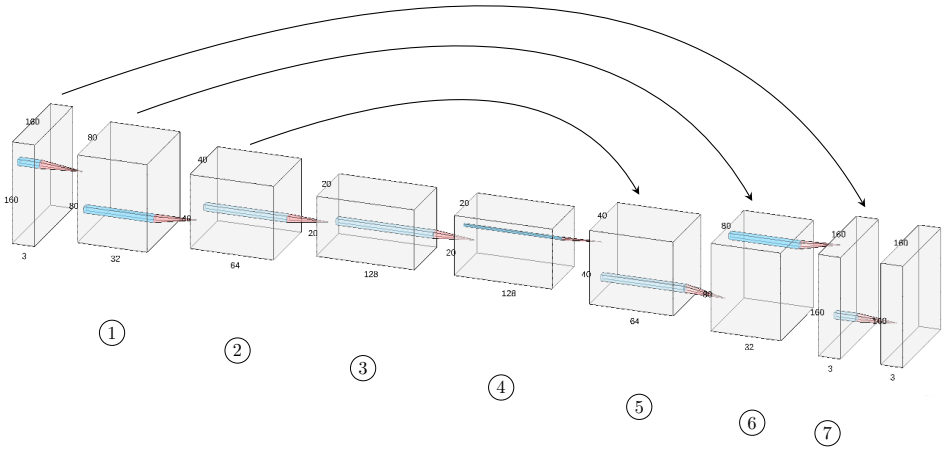

Figure 17: text

Table 2: text

Name	**Width (W)**	**Height (W)**	**Depth (D)**	Layer Details
Input	160	160	3	Raw image input
1	80	80	32	Created from 2D convolution of raw image input with 32 filters of patch size 3 × 3, and the stride is 2 with same padding
2	40	40	64	Created from 2D convolution of layer 1 with 64 filters of patch size 3 × 3, and the stride is 2 with same padding
3	20	20	128	Created from 2D convolution of layer 2 with 128 filters of patch size 3 × 3, and the stride is 2 with same padding
4	20	20	128	Created from 2D convolution of layer 3 with 128 filters of patch size 1 × 1, and the stride is 1 with same padding
5	40	40	64	Created by adding upsampled layer 4 to layer 2 with a skip connection, and passed through a 2D convolution with 64 filters of patch size 3 × 3 and stride XXXX
6	80	80	32	Created by adding upsampled layer 5 to layer 1 with a skip connection, and passed through a 2D convolution with 32 filters of patch size 3 × 3 and stride XXXX
7	160	160	3	Created by adding upsampled layer 6 to the inputs with a skip connection, and passed through a 2D convolution with 3 filters of patch size 3 × 3 and stride XXXX
Output	160	160	3	Created from a 2D convolution

2.3.3 Model 2: Shallow Model with 1 × 1 Convolution

The proposed deep model with 1 × 1 convolutions is comprised of 9 layers, including the input layer - the network architecture can be seen in Figure XXXX, and a detailed description of each layer can be found in Table XXXX

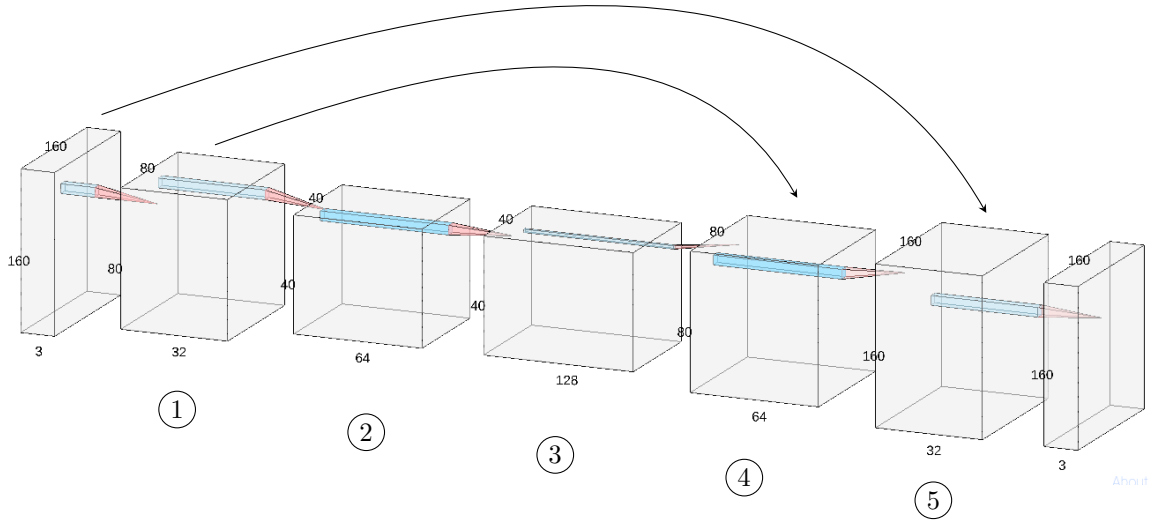

Figure 18: text

Table 3: text

Name	Width (W)	Height (W)	Depth (D)	Layer Details
Input	160	160	3	Raw image input
1	80	80	32	Created from 2D convolution of raw image input with 32 filters of patch size 3 × 3, and the stride is 2 with same padding
2	40	40	64	Created from 2D convolution of layer 1 with 64 filters of patch size 3 × 3, and the stride is 2 with same padding
3	40	40	128	Created from 2D convolution of layer 2 with 128 filters of patch size 1 × 3, and the stride is 1 with same padding
4	80	80	64	Created by adding upsampled layer 3 to layer 1 with a skip connection, and passed through a 2D convolution with 64 filters of patch size 3 × 3
5	160	160	32	Created by adding upsampled layer 4 to the raw input with a skip connection, and passed through a 2D convolution with 32 filters of patch size 3 × 3
Output	160	160	3	Created from a 2D convolution

2.3.4 Model 3: Shallow Model without 1 × 1 Convolution

The proposed deep model with 1 × 1 convolutions is comprised of 8 layers, not including the input layer - the network architecture can be seen in Figure XXXX, and a detailed description of each layer can be found in Table XXXX

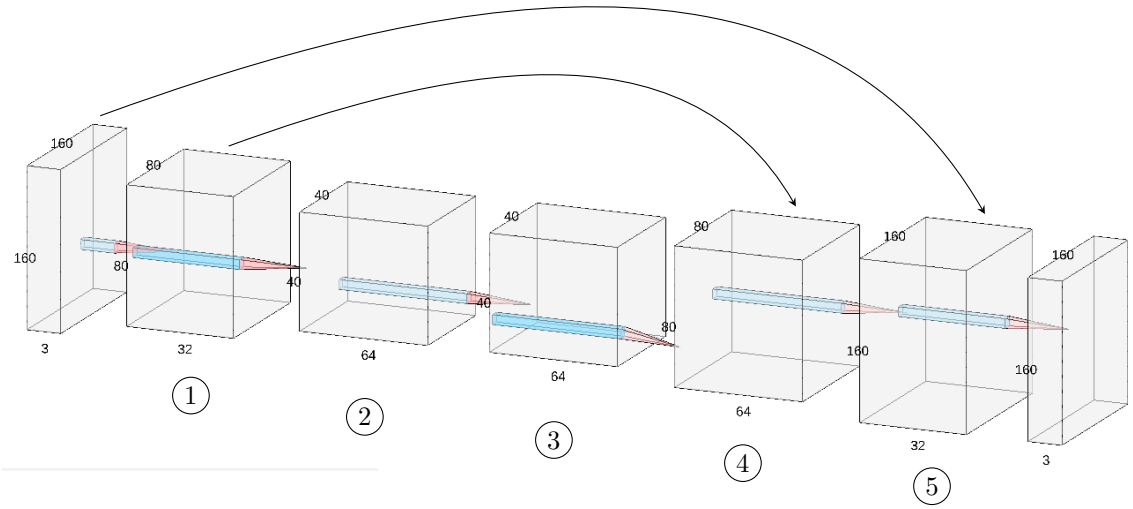

Figure 19: text

Table 4: text

Name	Width (W)	Height (W)	Depth (D)	Layer Details
Input	160	160	3	Raw image input
1	80	80	32	Created from 2D convolution of raw image input with 32 filters of patch size 3 × 3, and the stride is 2 with same padding
2	40	40	64	Created from 2D convolution of layer 1 with 64 filters of patch size 3 × 3, and the stride is 2 with same padding
3	40	40	64	Created from 2D convolution of layer 2 with 64 filters of patch size 3 × 3, and the stride is 1 with same padding
4	80	80	64	Created by adding upsampled layer 3 to layer 1 with a skip connection, and passed through a 2D convolution with 64 filters of patch size 3 × 3
5	160	160	32	Created by adding upsampled layer 4 to raw inputs with a skip connection, and passed through a 2D convolution with 32 filters of patch size 3 × 3
Output	160	160	3	Created from a 2D convolution

3 Data & Network Training

The initial data set provided with the problem was comprised of XXXX images. There are three rough categories that the images can be placed in:

1. XXXX
2. XXXX
3. XXXX

To provide the network with additional training data, XXXX, XXXX, and XXXX images were captured from the simulator in each of the fields, respectively. When an image is captured, pixel label data is also provided in the form of image masks. There are three types of masks that accompany each image: a mask highlighting the hero; a mask highlighting people who are not the hero; and a mask highlighting the background. Examples of these masks can be seen in Figures XXXX, XXXX, and XXXX, respectively. It must be noted that these labels would not be available with real world images, and the masks are able to be extracted only because the image data is collected from a simulation.

Figure 20: text

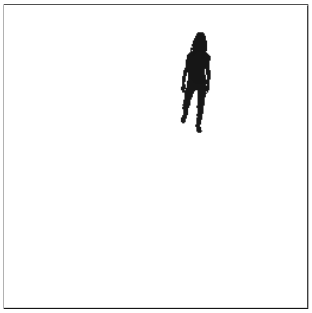

Figure 21: text

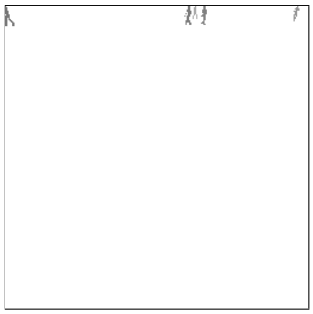

Figure 22: text

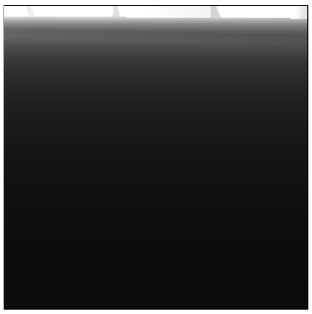

Figure 23: text

Prior to model training, there is some pre-processing which concatenates the mask images into a single image. The network is trained using an optimiser called Adam, which is similar to Stochastic Gradient Descent (SGD). It differs in one important way: SGD typically uses a static learning rate, whereas Adam computes adaptive learning rates for each parameter (XXXX, XXXX). Talk about the use of hyperparameters for training the network

3.1 Batch Size

Batch size defines the number of training samples/images propagated through the network in a single pass. Batch size has an upper bound dependent on the amount of RAM in the GPU - we cannot accommodate more images than can be stored in RAM for a pass. A small experiment was conducted to gain some intuition on how batch size affects model performance. Model performance was tested for a series of increasing batch sizes, setting learning rate at 0.01, number of epochs at 5, and number of steps per epoch to 80. Figure XXXX shows the training loss for increasing batch size.

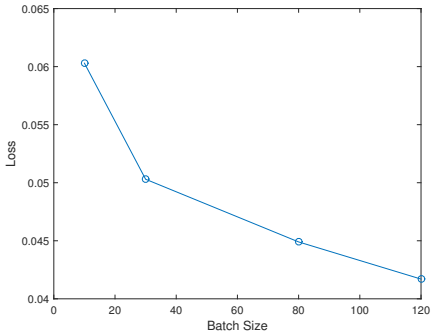

Figure 24: Change in loss function as batch size is increased

Clearly increased batch size allows more training images to pass through the model resulting in better performance - which is in line with intuition. This relationship is most likely follows a law of diminishing returns, with increases in performance reaching some asymptotic limit as batch size increases further.

3.2 Epochs

Epochs are the number of times the entire data set gets propagated through the network. Intuition would suggest that the more epochs, the more the model would be exposed to the training dataset, resulting in a better performance. Evidence supporting this intuition can be seen in Figure XXXX, which shows a small experiment in which the loss function was measured for an increasing number epochs. Note that the other model hyper-parameters were held constant with learning rate of 0.01, batch size 20, and number of steps 80.

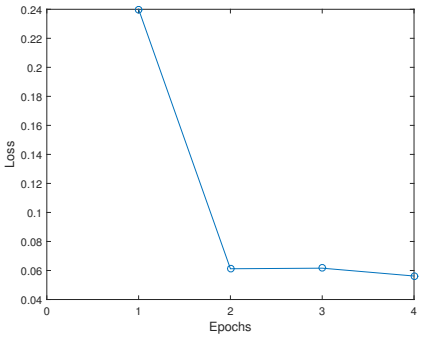

Figure 25: Change in loss function as the number of epochs is increased

Whilst it is clear that more data means better performance, care must be taken to avoid over exposing the model to the training dataset. Failure to do so will often result in excellent performance on the training data set, but poor performance on test and validation sets due to over fitting.

3.3 Steps Per Epoch

The number of steps per epoch defines the number of batches of training images that go through the network in one epoch. It is considered desirable for one epoch to cover the entire dataset, and therefore this hyper-parameter can be thought of as a function of the total number of images, N, and the batch size, $N_{\texttt{batch}}$:

$$\texttt{steps\_per\_epoch} = \frac{N}{N_{batch}} \tag{2}$$

Setting the steps per epoch equal to the value shown in equation (2) will ensure that all of the images are covered in each epoch.

3.4 Learning Rate

Talk about the learning rate being critical to

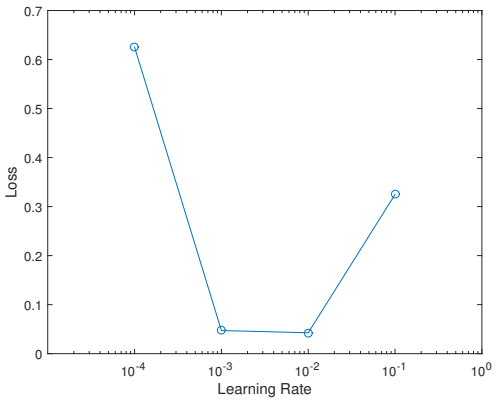

Figure 26: text

4 Performance & Model Generalisation

5 Future Enhancements