

[Link to management report](#)

Max 8000 words (not including cover page, contents page, and references page)

What this report needs to cover:

- Process to get from initial brief to proposed solution
- Blueprint for final design
- Purpose of your system
- How it works at a high level
- Any limitations it may have (assumptions that you have made for its correct operation)
- Suggestions for future work (i.e. what we would do if we had more time and how we would do it) - to indicate that we've thought beyond the point we are currently in
- References (using IEEE referencing)

Marking scheme:

Introduction/background (5%) **450 Words**

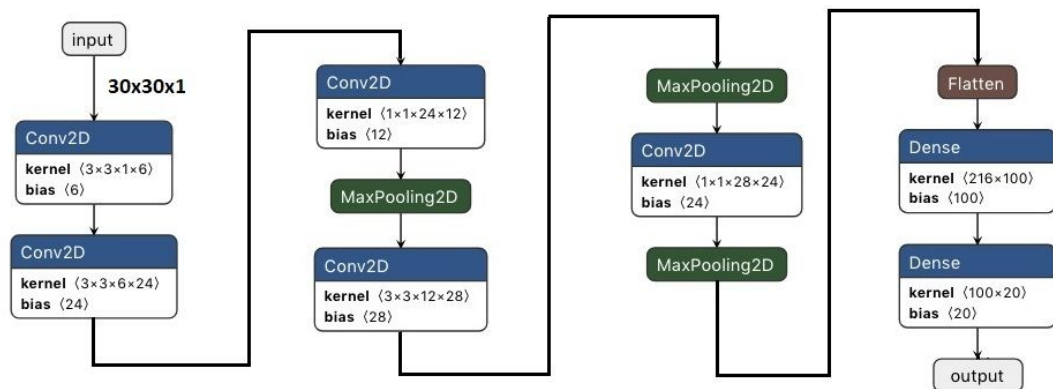
- Updated Description of the project:
 - Purpose of the proposed design
 - Mimics pictictionary
 - To be able to recognize drawings of objects from a set of trained objects
 - Explanation of the outcomes
 - Takes in the input image drawn by user and calculates the likelihood of the drawing being a certain object.
 - Displays on the screen what object it thinks it is and displays a visual indicator of the percentage of certainty
 - NB: Please do not repeat all the information from the Management Report. Simply summarise it in a couple of paragraphs to provide context

Progress made since Management Report (20%) **1780 Words**

- (Moin, Aarsal) Updated High level description of the system:
 - Explain what a CNN is, how it works, and why we are using one
 - A CNN is a way of processing visual data
 - A CNN is comprised of 'layers' that take in data from the previous layer, perform some processing on it, and send some outputs to the next layer. The input image is passed into the first layer of the network, and (in our case, where our CNN is an image classification CNN) the output layer provides the classification outputs (e.g. the confidence scores for each of our 20 classes)
 - Due to the hierarchical manner in which they process visual data, CNNs are able to able to recognize complex features in the image, by first extracting low-level features such as edges and corners, and then combining information about these features to recognize more complex shapes and textures, and eventually recognizing full objects such as faces, animals, cars, etc
 - A typical CNN is comprised of different types of layers that serve different purposes - convolutional layers, pooling layers, and fully connected layers

- Convolutional layers - in a convolutional layer, a set of kernels/filters is 'convolved' over the input image to produce a 'feature map' representing what features have been detected by the filters on different parts of the input image. This feature map (after going through an 'activation function') is then passed to the next layer - if the next layer is also a convolutional layer, it will convolve another set of filters over the outputted feature map, to detect increasingly higher level features.
 - Pooling layers - The purpose of the pooling layer is to reduce the pixel height and width of the input image so that less computations are needed. Max pooling is a technique which reduces the input sizes but retains most of significant features of the image. It has been used many times within the ML community and works very well for our purposes.
 - Fully connected layers -
 - During training, the optimal set of weights (i.e. the values for the kernels to use as well as the weights of the fully connected layers) are learned
 - In recent years there has been growing interest in developing custom hardware for running CNNs and other machine learning applications, and in our project we intend to create a system to learn and demonstrate how an FPGA can be used for low power, hardware-accelerated, on-device CNN inference.
- Has it changed from the system described in the Management Report?
 - After discussing with Dr. Bouganis, we decided that the MobileNetV2 architecture would probably be too challenging to implement as there are too many kernels per layer and too many layers -> would be too slow on the FPGA would not meet the 25 fps frame rate requirement
 - Instead, we will use a simpler architecture based on LeNet-5 (a CNN architecture developed in 1998 by Yann LeCun for recognizing handwritten numbers in images)
 - Also the aim of the project has changed from recognizing food to guessing what object a user has drawn on the screen, to ensure that it is something that a shallower network can perform well on
- What functions / operations will the system perform?
 - User draws on an online sketch board which has a white background, using a black marker.
 - This happens on a computer, and the display screen is taken as an input for our project, as opposed to a user drawing in an actual whiteboard.
 - Asking the user to draw, and video-recording them bears a far greater risk of inaccurate results since external influence, like moving whiteboard, and change in lighting, is likely to occur.
 - This method also removes the need to filter the entire input image to sift out just the whiteboard in the needed dimensions
 - The system will first transform the input image frames into a 30x30 frame and convert it to grayscale and invert the values (i.e. darker shades would become lighter and vice versa).

- The system will then perform Convolution and other image processing techniques(edge detection, etc) to recognize the input image from a set of possible objects using trained weights of the neural network.
- The system will display a confidence score for the predicted classes of what object the user has drawn.
- How will they be achieved?
 - For manipulation of frames, we use OpenCV, a python image processing library to do three main objectives:
 - Cropping and resizing the input image frames.
 - Converting them to grayscale such that they only have a single channel instead of smoothed out three.
 - Displaying the classes and their percentage certainties, using text over image. Percentage certainties are represented by a "loading bar" made up of character "|". Higher certainty means a greater number of | in the image next to the class name.
 - The displaying of the certainty on the output screen will be done by.....
 - The recognition of the drawing will require:
 - the CNN architecture to be built first in C++
 - train CNN architecture on data sets of the different objects
 - Add optimisations to make the process quicker
 - Implement on vivado HLS and add further optimisations
 - Export onto Vivado and implement on the PYNQ Board.
 - Test with further images to check for accuracy
 - Make any necessary improvements
- (All) Updated Schematic of the system:
 - Provide a diagram showing how the information flows through your system
 - High Level Schematic(How the project works overall)
 - Schematic of CNN Architecture (Turn this into a proper diagram)



Layer (type)	Output Shape	Param #
conv2d_46 (Conv2D)	(None, 28, 28, 6)	60
conv2d_47 (Conv2D)	(None, 26, 26, 24)	1320
conv2d_48 (Conv2D)	(None, 26, 26, 12)	300
max_pooling2d_28 (MaxPooling)	(None, 13, 13, 12)	0
dropout_28 (Dropout)	(None, 13, 13, 12)	0
conv2d_49 (Conv2D)	(None, 13, 13, 28)	3052
max_pooling2d_29 (MaxPooling)	(None, 6, 6, 28)	0
conv2d_50 (Conv2D)	(None, 6, 6, 24)	696
max_pooling2d_30 (MaxPooling)	(None, 3, 3, 24)	0
dropout_29 (Dropout)	(None, 3, 3, 24)	0
flatten_10 (Flatten)	(None, 216)	0
dense_19 (Dense)	(None, 100)	21700
dropout_30 (Dropout)	(None, 100)	0
dense_20 (Dense)	(None, 20)	2020
Total params: 29,148		
Trainable params: 29,148		
Non-trainable params: 0		

■ Schematic of Each Type of Layer in the CNN architecture

■ Schematic of optimization techniques:

- GEMM for matrix multiplication (fully connected layers) and convolution (when represented as a matrix multiplication operation)
- Winograd algorithm
- Depthwise separable convolution

■ Guys can we have a list of all the graphics, images etc in one place like gantt charts, blueprints etc so we can get started on them.

● (Arsal) Project planning and management:

○ Have you changed the strategy since the Interim Report?

■ We have mostly kept to the strategy as we made sure:

- everyone completed the CNN course so that they were equipped in implementing a CNN architecture
 - The data set was downloaded and trained the neural network
 - Next implementation and testing of the neural network.
- Some changes to the strategy occurred due to the change in the architecture of the CNN used and hence tasks were added to learn the architecture and implement techniques to make the CNN faster which added a lot more time needed to work on the implementation of the CNN than previously expected.
- It is the nature of machine learning at this stage that we are not always exactly certain that an implemented technique would bear fruit on our particular system. Our chosen methods, GEMM and Winograd were not as effective on the FPGA board as we had hoped, even though they were considerably better on our personal computers.
- Therefore our strategy then shifted to focusing on our main CNN implementation and improving its performance using tools like loop unrolling and pipelining rather than focusing on separate algorithms to do the job.
- Is there a Gantt chart including tasks, delivery dates and responsibilities?
 - Update Gantt Chart and add dates and new tasks
 - New Tasks:
 - Write code:
 - for direct/naive convolution operation in c++ and optimize it for vivado
 - for winograd convolution operation in c++ (and optimize it for vivado)
 - for GEMM convolution operation in c++ and optimize it for vivado
 - Integrate GEMM and winograd into naive CNN implementation, if they prove faster during testing runs
 - CNN network:
 - modify the number of layers of the mimics pictionary cnn found online
 - modify the number and/or accuracy of the parameters/weights of the kernels (quantization)
 - Training the weights of the cnn after modification
 - ...

Design Process (65%) **5780 Words**

- (Horace, Moin) Design specification:
 - Are systems requirements well defined?
 - Input image must be converted to a 28x28 image (actually the first layer has padding so you could say that the input image is a 30x30 image)
 - Input image must be drawn on a white background with dark paint (any color is fine as long as it is dark enough).
 - CNN architecture is
 - The layers within the architecture are

- The output frame rate must be at least 25 fps
- The output image size must be

- (Arsal) Design process:

- Explain the approach used by the team in designing the system
 - Each member took a course on CNN to get familiar with how to implement the system.
 - The approach used by the team was to get each member to work on a small part of the system individually and then meet up regularly to combine the works into a fully functioning system.
 - Each member was designated to research and implement a technique to speed up convolution
 - The layers of the CNN architecture was also distributed in a way to give the most challenging layers to the group members with the most experience with deep learning and neural networks.
 - Trial and error approach - after brainstorming and planning we tried something and see if it worked out - if not we tried to solve the issue (aka since winograd did not work out, we went for GEMM)
- Provide evidence of using one of the concept generating techniques covered in class (brainstorming, brainwriting, etc.)
 - Someone please go over one of these techniques and just draw a sketch of how we came up with different concepts.

- (All) Proposed solutions:

- Describe the different options considered at system and subsystem level
 - Show:
 - All the ideas we had
 - How they would work
 - Why they didn't get chosen
 - Why did we choose this one
 - CNN architectures:
 - (only contains architectures with > 89% accuracy after 20 epochs):
<https://docs.google.com/spreadsheets/d/1K877PRDfJW0hhOzTkirxRajGqLE0InGoTOBGKO3uXIs/edit#gid=0>
- Remember to include those options that did not work

- Final design selection

- Provide a rationale for the selection made
 - Explain why MobileNetV2 was too ambitious and how Dr Bouganis us to do something else
 - Explain the benefits of what our system can do and how it can be used in real life
 - Explain how this new architecture can meet the requirements
- Provide some elaboration on the key design decisions that you had to make for your system or any optimisation that you have performed

- Explain techniques explored (e.g. im2col+GEMM, Winograd algorithm, depthwise separable convolution, etc., and how we chose them) to speed up computation + show graphs of how much speed up was achieved compared to direct convolution
 - GEMM+im2col
 - Winograd algorithm
 - FFT - decided not to use this technique because it only works well for large filter sizes (whereas our CNN only had 3x3 filters and 1x1 filters for pointwise convolution)
 - Depthwise separable convolution
 - Quantization

■

- Provide evidence of using a matrix selection method in at least one aspect of your system

■ NO IDEA WHAT THIS MEANS.

- (Kimon) Outcomes: WE can only write about this after we have implemented the project!
 - What are the outcomes (solution) of the project?
 - Are they what you had expected at the beginning of the project?
 - Consider the hardware resources dedicated to the design (FPGA resources)
- (Horace) Suggestions for **future work** / improvements:
 - What improvements could be made to the system if you had more time?
 - Improve the speed-accuracy tradeoff - figure out ways to make the CNN run faster so that we could accommodate a larger network (for higher accuracy and to support a higher number of output classes)
 - Finish Winograd
 - Learn more techniques in Vivado HLS to figure out how we can optimize the system better
 - Use automated neural architecture search through python packages like AutoKeras to find a better architecture that uses less weights or requires a lower number of operations but still has the same accuracy
 - Data augmentation - enlarge the training set to make the network more accurate - make it work with something drawn on paper / variety of backgrounds, stroke colors, stroke widths, stroke textures, etc.
 - See whether performing pre-processing and post-processing in hardware rather than in software improves latency
 - Explore other uses of our CNN - the code we wrote could be easily used for running other CNNs trained on other datasets (not just the QuickDraw dataset)

Report (10%)

- Has the **word limit** of 8,000 words been respected?
- Does the report have a logical **structure** which is easy to follow?

- Is the **writing** suitable for a technical report? Includes grammar and spelling
- Ability to communicate effectively
- Report **presentation**: Cover and contents pages?
- Have all sources been **referenced**?

References:

- <https://github.com/googlecreativelab/quickdraw-dataset>
- <https://github.com/subarnop/Kiddo/>
- <https://adeshpande3.github.io/assets/Cover.png> (picture of CNN layers)
- <https://qph.fs.quoracdn.net/main-qimg-98ecf7ba49710bf56042d035a74505b6>
- (picture of pooling layer)
- https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/assets/tfdl_0401.png (picture of FC layer)

<https://arxiv.org/pdf/1509.09308.pdf> [winograd]

https://trace.tennessee.edu/cgi/viewcontent.cgi?referer=https://www.google.com/&httpsredir=1&article=5786&context=utk_gradthes [winograd]

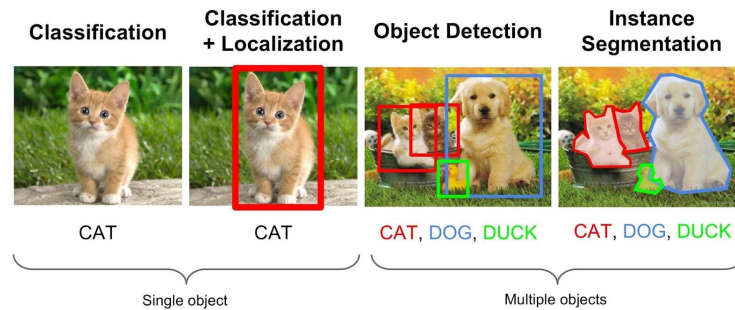
References

[1] J. Jongejan and I. Johnson, "googlecreativelab/quickdraw-dataset", *GitHub*, 2019. [Online]. Available: <https://github.com/googlecreativelab/quickdraw-dataset>. [Accessed: 17- May- 2019].

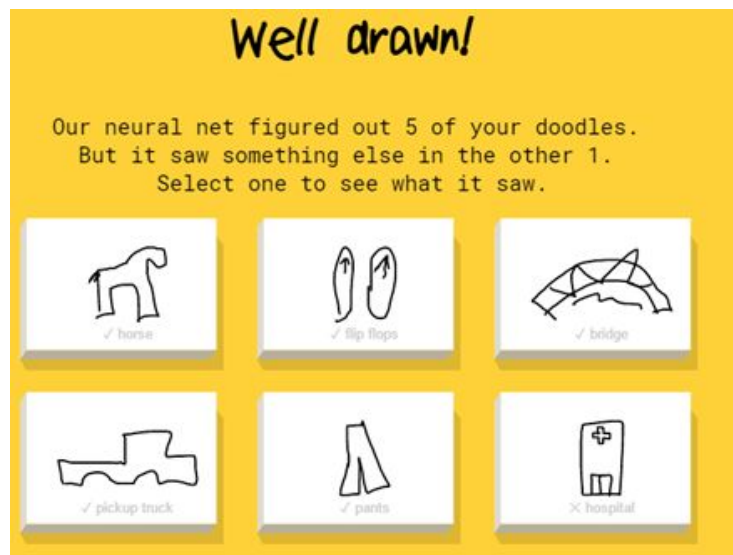
- [2]S. Pal and S. Basak, "subarnop/Kiddo", *GitHub*, 2019. [Online]. Available: <https://github.com/subarnop/Kiddo/>. [Accessed: 17- May- 2019].
- [3]*Adeshpande3.github.io*, 2019. [Online]. Available: <https://adeshpande3.github.io/assets/Cover.png>. [Accessed: 17- May- 2019].
- [4]*Qph.fs.quoracdn.net*, 2019. [Online]. Available: <https://qph.fs.quoracdn.net/main-qimg-98ecf7ba49710bf56042d035a74505b6>. [Accessed: 17- May- 2019].
- [5]*Oreilly.com*, 2019. [Online]. Available: https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/assets/tfdl_0401.png. [Accessed: 17- May- 2019].
- [6]*Arxiv.org*, 2019. [Online]. Available: <https://arxiv.org/pdf/1509.09308.pdf>. [Accessed: 17- May- 2019].
- [7]*Trace.tennessee.edu*, 2019. [Online]. Available: https://trace.tennessee.edu/cgi/viewcontent.cgi?referer=https://www.google.com/&httpsredir=1&article=5786&context=utk_gradthes. [Accessed: 17- May- 2019].
- [8]*Cdn-images-1.medium.com*, 2019. [Online]. Available: https://cdn-images-1.medium.com/max/1600/1*Mj8WKVKf_RpiAsX3SC1ZdQ.png. [Accessed: 17- May- 2019].
- [9]*Cdn-images-1.medium.com*, 2019. [Online]. Available: https://cdn-images-1.medium.com/max/1200/1*yG6z6ESzsRW-9q5F_neOsg.png. [Accessed: 17- May- 2019].
- [10]*Cdn-images-1.medium.com*, 2019. [Online]. Available: https://cdn-images-1.medium.com/max/1200/1*Q7a20gyuunpJzXGnWayUDQ.png. [Accessed: 17- May- 2019].
- [11]*Redcatlabs.com*, 2019. [Online]. Available: http://redcatlabs.com/2018-07-19_TFandDL_MixedPrecision/img/float32and16_800x333.png. [Accessed: 17- May- 2019].
- [12]P. Warden, "Why GEMM is at the heart of deep learning", *Pete Warden's blog*, 2019. [Online]. Available: <https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/>. [Accessed: 17- May- 2019].
- [13]"ResearchGate | Share and discover research", *ResearchGate*, 2019. [Online]. Available: <https://www.researchgate.net/>. [Accessed: 17- May- 2019].
- [14]2019. [Online]. Available: <https://www.oreilly.com/library/view/learn-unity-ml-agents/>. [Accessed: 17- May- 2019].
- [15]*Systems.ethz.ch*, 2019. [Online]. Available: <https://www.systems.ethz.ch/sites/default/files/parallel-distributed-deep-learning.pdf>. [Accessed: 17- May- 2019].
- [16]*Image.slidesharecdn.com*, 2019. [Online]. Available: <https://image.slidesharecdn.com/ti2-04armiodice-160729141820/95/using-sgemm-and-ffts-to-accelerate-deep-learning-a-presentation-from-arm-8-638.jpg?cb=1470496898>. [Accessed: 17- May- 2019].

Introduction:

Over the years there has been increasing research and focus on artificial intelligence and computer vision. This is due to the growing potential of and investment in image recognition for uses in medical image analysis, self driving cars, face detection and more. 'Image recognition is the ability of a system or software to identify objects, people, places, and actions in images' [1]. Due to image recognition being of such importance for the future our group was motivated to make this the focus of our project and go beyond simple image processing techniques. This meant doing extensive research and completing online courses on deep learning in order to produce a system which incorporates convolutional neural networks to perform image recognition.



The application of our image recognition system is taken as an inspiration from Google's "Quick, Draw!". "Quick, Draw!" is an online game in which the user is asked to draw an object or idea and then a neural network is used to guess what the drawings represent. This is done in real time and the neural network gives its prediction as the user is drawing.























Our project works by taking into input an image by getting the user to draw one out of 20 objects and gets our implementation of CNN layers to build a neural network which calculates the likelihood of being an image from each class. Unlike "Quick, Draw!" our system along with displaying on the screen

what objects it thinks it is, also shows a visual indicator of the percentage of certainty of all objects during the drawing process.

Our project has thus aimed to implement this interactive image processing system using the PYNQ-Z1 board with a frame rate of 25Hz. This would be achieved after utilizing the massively parallel processing capabilities of the FPGA to perform hardware-accelerated image processing.

Updated High level description of the system

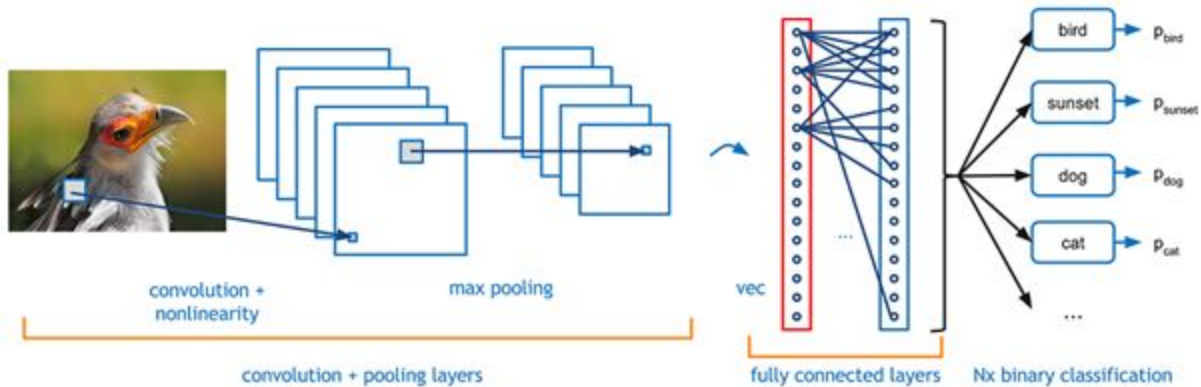
Our system is a computer vision application which uses Convolutional Neural Networks(CNN) to perform image classification on an image drawn by the user. Our System is able to identify with an average accuracy of 87% 20 different classes of images. These include cars , birds, and more as shown below. In our project, we create a system that learns and demonstrates how an FPGA can be used for low power, hardware-accelerated, on-device CNN inference.

Classes used (with example images)						
airplane 	apple 	bathtub 	bicycle 	bird 	brain 	car 
cat 	cell phone 	dolphin 	flower 	peas 	penguin 	shoe 
skyscraper 	speedboat 	television 	violin 	watermelon 	wristwatch 	

Description of CNNs

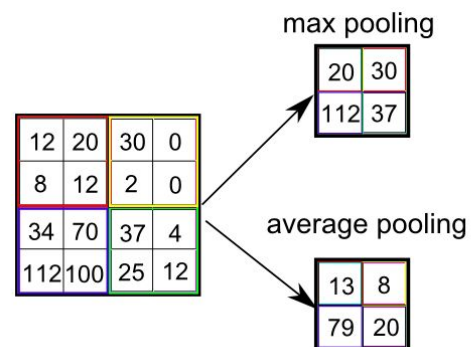
A CNN is a way of processing visual data and is comprised of ‘layers’ that take in data from the previous layer, perform some processing on it, and send outputs to the next layer.

A typical CNN is comprised of different types of layers that serve different purposes. These include convolutional layers, pooling layers, and fully connected layers.

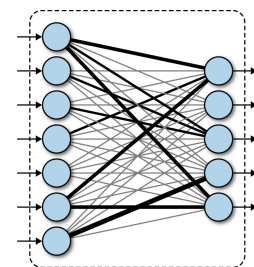


Convolutional layers - In a convolutional layer, a set of filters are 'convolved' over the input image to produce a 'feature map' representing what features have been detected by the filters on different parts of the input image. This feature map (after going through an 'activation function') is then passed to the next layer - if the next layer is also a convolutional layer, it will convolve another set of filters over the outputted feature map, to detect increasingly higher level features.

Pooling layers – Pooling is a downsampling strategy that reduces the spatial size and the number of parameters of the input, thus increasing the computation speed. The pooling layer can be of 2 main types; max pooling and average pooling. Max pooling selects the maximum number from subspace occupied by the filter while average pooling calculates the average of all the values within the subspace.



Fully connected layers – This layer is attached to the end of the CNN and it transforms the input volume into a vector consisting of all vital information about the input. It holds the most important aggregated information of all the previous convolutional layers and thus allows the layer to do classification based on features from the previous layers.



Once the CNN architecture is built, training is done so that the optimal set of weights (i.e. the values to use for the filters as well as the weights of the fully connected layers) are learned. This allows the CNNs to be useful in applications such as image classification and object detection.

Has it changed from the system described in the Management Report?

After discussing our project idea with Dr. Bouganis, we decided that the MobileNetV2 CNN architecture we were originally thinking of implementing would be too challenging to implement as it uses a significant number of kernels per layer and there are too many layers in total to be realistically implemented in a short timeframe on the PYNQ board. In order to ensure that we can have a CNN that works in real time, we decided to use a smaller network based on LeNet-5, a CNN architecture developed in 1998 by Yann LeCun for recognizing handwritten digits in images, as it has less layers and filters and therefore requires less computational power to process a single image.

Since we were using a smaller CNN, we changed the use case of the project from food recognition to handwritten drawing recognition (i.e. guessing what object the user has drawn on a screen), in order to ensure that aim of the CNN is something that a shallower network can perform well on.

What functions / operations will the system perform?

First the user will draw an image on an online sketch board. The sketch board will have a white background and the user will draw with a black marker. This happens on a computer, and the display screen is taken as an input for our project, as opposed to a user drawing in an actual whiteboard. This is because asking the user to draw, and video-recording them bears a far greater risk of inaccurate results since external influence, like moving whiteboard, and change in lighting, is likely to occur. This method also removes the need to crop the entire input image to sift out just the whiteboard in the needed dimensions

The system will then transform the input image frames into 30x30 frames and convert them to grayscale and invert the values (i.e. darker shades would become lighter and vice versa). This is done to meet the input image requirements of our CNN.

The image will then enter our CNN which will try to recognize the input image from a set of possible classes using trained weights of the neural network.

Due to the hierarchical manner in which CNNs process visual data, they are able to recognize complex features in the image, by first extracting low-level features such as edges and corners, and then combining information about these features to recognize more complex shapes and textures, and eventually recognizing full objects such as faces, animals, cars, etc

The system will finally display the confidence scores for each of our 20 classes trained by our CNN.

How will they be achieved?

For manipulation of frames, we use OpenCV, a python image processing library to do three main objectives:

1. Cropping and resizing the input image frames.
2. Converting them to grayscale such that they only have a single channel instead of smoothed out three.
3. Displaying the classes and their percentage certainties, using text over image. Percentage certainties are represented by a "loading bar". Higher certainty means a greater proportion of green to red in the loading bar over the image next to the class name.

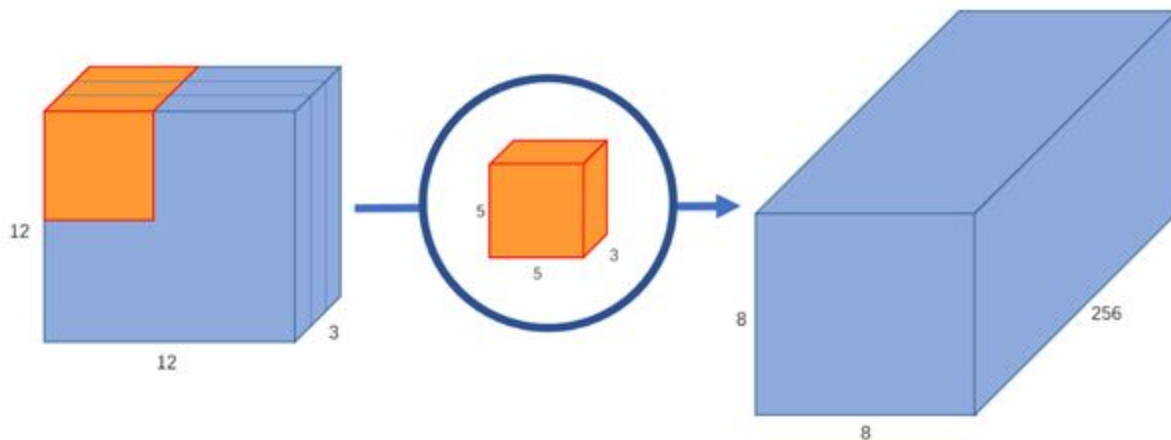
The recognition of the drawing will require the use of convolutional neural networks, this is a multi step process:

1. The CNN architecture to be built first in C++ .
2. Train CNN architecture on data-sets of the different classes
3. Add optimisations to make the process quicker
4. Implement on vivado HLS and add further optimisations
5. Export onto Vivado and implement on the PYNQ Board.
6. Test with further images to check for accuracy
7. Make any necessary improvements

Depthwise Separable Convolutions

Depthwise Separable Convolution is an alternative to regular 2D convolution that requires roughly 8-9x less operations when using 3x3 kernels:

To understand the significance of the depthwise separable convolution lets take a Normal Convolution with a 12x12x3 input and a 8x8x256 output.



https://cdn-images-1.medium.com/max/1200/1*XloAmCh5bwE4j1G7yk5THw.png

To compare we can compute the number of multiplications done in normal convolution. Since we have 256 5x5x3 kernels that form an 8x8 output, the total multiplications are $256 \times 5 \times 5 \times 3 \times 8 \times 8 = 1,228,800 \sim 1.3 \text{ Million}$.

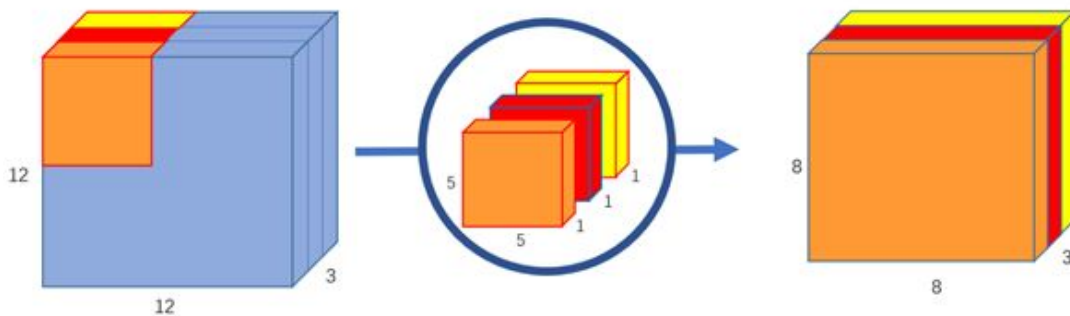
A Depthwise separable convolution separates this convolution process into 2 steps:

1. Depthwise Convolution:

The input volume is given a convolution without changing the depth. This is done by keeping each channel separate when performing convolution. Depthwise convolution can be thought of as 3 steps:

- First split the input image into its channels and split the kernel into the same number of channels.
- Now convolve each of the channels with their corresponding kernels producing a 2D output .
- Finally stacking these convolutions together will form the output of this step.

So for example if we have a 12x12x3 image and we use a 5x5 kernel with stride 1. We will use 3 5x5x1 kernels, each performing convolution on 1 channel each leading to a combined output of depth 3.

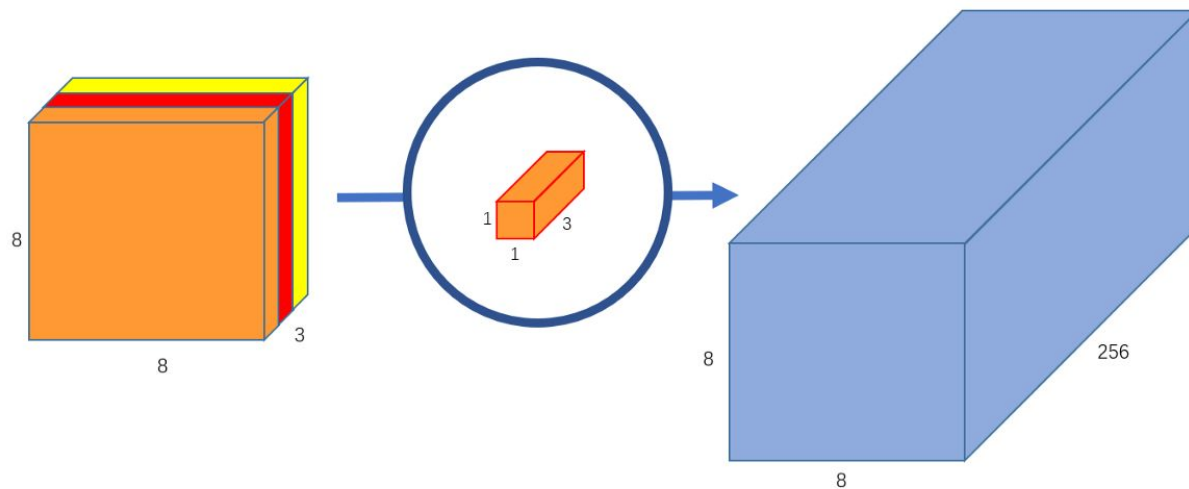


https://cdn-images-1.medium.com/max/1200/1*yG6z6ESzsRW-9q5F_neOsg.png

Now since we have 3 5x5x1 kernels that form an 8x8 output, the total multiplications are $3 \times 5 \times 5 \times 1 \times 8 \times 8 = 4800$.

2. Pointwise Convolution:

This step uses a 1x1 kernel with the depth the same as the output from the previous step. The 1x1 filter convolves the whole input volume and forms an output volume with the depth of 1. Now by choosing the number of 1x1 kernels you can form a volume of the depth you require. Since our example output had a depth of 256, we can perform pointwise convolution using 256 1x1x3 kernels.



https://cdn-images-1.medium.com/max/1200/1*Q7a20gyuunpJzXGnWayUDQ.png

Having 256 1x1x3 kernels that form an 8x8 output, the total multiplications are $256 \times 1 \times 1 \times 3 \times 8 \times 8 = 49,152$.

Hence, adding up the total multiplications for both depthwise and pointwise convolution, we get $4800 + 49,152 = 53,952$ multiplications. This is about 24 times less multiplications than the 1.3 million multiplications used by normal convolution.

Therefore, we have produced a similar output volume as normal convolution but with a significantly lower computational cost. The benefit of this technique is that it reduces the number of parameters and hence with fewer parameters the computation requires less operations and hence is faster and cheaper to implement.

After implementing and testing depthwise separable convolution in Vivado HLS, we were able to achieve a 2.45x speedup (on a 13x13x64 input volume) compared to our implementation of direct convolution, before adding any pragma directives. However, in the end we did not use depthwise separable convolution as we experienced an issue with the accuracy rate being significantly reduced when attempting to apply post-training quantization to our depthwise separable convolution model (more details on quantization further below).

Provide a rationale for the selection made

After meeting with Dr Bouganis we concluded that the MobileNetV2 architecture discussed in the management report was too ambitious. This is because it would be very difficult to get all the filters and layers of that architecture working at real-time speed on the board. The architecture would also has a high number of parameters which means it may have exceeded the PYNQ board's limited memory

The architecture we decided to use instead is based on LeNet-5. The original LeNet-5 was developed by Yann LeCun and it has 98% accuracy on recognizing handwritten digits, so after adding modifications and optimisations we achieve a similar architecture with a slightly higher number of layers and parameters allowing it perform well in recognizing hand drawn images. We also used images from the QuickDraw dataset which have a similar size (28x28) as the input size of LeNet-5 (32x32).

After all the optimisations added to the execution and reduction in parameters and retraining this architecture allows the CNN to be successfully implemented on the PYNQ board with a accuracy of% while still meeting the 25Hz frame rate requirement.

Have you changed the strategy since the Interim Report?

We have maintained our original action plan, since the conditions needed to maintain the strategy were cleared i.e. we were on our schedules with no significant delays in our objectives which were outlined in the interim report. All team members completed the CNN course by deeplearning.ai course from Coursera. This ensured everyone was equipped with skills and knowledge needed to implement a CNN architecture. We managed to download the Google Quickdraw datasets, published by Google, and used them to train our neural network. This meant that we obtained several thousand images of doodles drawn by real users without actually having to draw hundred of doodles (of several objects that we wanted our network to recognise) ourselves.

The next stage of the plan was to implement and test the Convolutional Neural Network, but we had to re-organise ourselves and our goals to accommodate for the switch to LeNet5-based architecture as opposed to previously decided MobileNetV2- based architecture. This not only added a layer of complexity to an already complex task, but also increased the amount of work as a whole: we now not only needed to research and familiarise ourselves with this new kind of architecture, learn the skills needed to implement it, and most importantly, research and implement techniques to make the CNN faster and more efficient on the FPGA. This created a significant vacuum in our plan, and took a lot more time and effort then expected. We decided to dedicate one team member each to implement GEMM, and Winograd algorithms to speed up the convolution mechanisms, while some members worked on finalising a direct convolution implementation on the actual FPGA. This ensured maximum productivity; once the GEMM and Winograd algorithms were finalised, they could simply be integrated into the direct convolution code to a single unified network.

At this point however, our goals were not perfectly met:

- While GEMM was implemented and proved to be faster on personal computers, it didn't perform as impressively on the hardware provided to us.
- Winograd was only partially implemented given the time that remained.
- Our direct convolution implementation required more resources than were available for the IP block on the Pynq board, particularly the look-up tables which were 40% over-utilised, because of which our implementation failed to synthesise on Vivado.

These unforeseen circumstances called for some drastic reductions to our final implementation goals, and we decided to:

- Gather our complete focus on getting the Direct Convolution to work on the FPGA, without GEMM, without Winograd, and with significantly reduced parameters for our network
- Forgo the implementation of the Winograd algorithm
- Debug and improve the performance of GEMM on FPGA hardware
- Focus entirely on simply using methods like loop unrolling and pipelining techniques outlined during labs to speed up direct convolution
- Use a technique called quantization, which would allow the use of 8 bit floating point numbers in place of 32 bit floating point numbers. This would significantly reduce the resource utilization.





















After some time and effort, we managed to bring the fps of our naive implementation up to 66 fps from 2.5 fps. GEMM was also improved to ~15 times its original speed. Our neural network managed to retain 89% accuracy rate. These numbers met our goals, and we decided to implement 16 bit quantisation in lieu of 8 bit quantisation, which is more complicated and error prone.

The remaining time was delegated to exporting the bitstream into the board, streamline the python pre and post processing scripts, and test the final design!

Training (Horace) - I'm writing this section here first but I'll figure out later where to move it to

Training overview

In order to be able to recognize hand-drawn objects with reasonable accuracy, our CNN was trained on a subset of Google’s [QuickDraw Dataset](#), which contains over 50 million drawings in total across 345 classes, taken from the “Quick, Draw!” online doodling game. We arbitrarily chose the following 20 classes from the dataset and used a total of 200,000 to train and validate our model.

Classes used (with example images)						
airplane 	apple 	bathtub 	bicycle 	bird 	brain 	car 
cat 	cell phone 	dolphin 	flower 	peas 	penguin 	shoe 
skyscraper 	speedboat 	television 	violin 	watermelon 	wristwatch 	

To train our CNN, we used Keras (a high-level deep learning library in Python) so that we could prototype quickly. Our code was based on a GitHub repository called [Kiddo](#) which provided the basic code for downloading the dataset and performing training, though we’ve made various modifications to the code to meet our project needs. Due to the high computational requirements of training a CNN, we rented a cloud GPU from a free service called Google Colab to accelerate our training.

CNN architecture

We experimented with training different network architectures (such as by adding/removing different layers, changing the number of kernels in the convolutional layers, etc.), and through this process we settled on the following architecture which we used in our final product. This architecture achieves a 30x reduction in parameter count and a 5.7x reduction in the number of multiply-accumulate (MAC) operations compared with the original architecture provided in the Kiddo GitHub repository, with a slight reduction in the classification accuracy rate (from ~93% to ~89%).

Original architecture				Our final architecture			
Layer	Output dimensions	MACs	Parameter count	Layer	Output dimensions	MACs	Parameter count

	(H, W, C)		
Input	28, 28, 1	-	-
3x3 conv with padding	28, 28, 6	42,336	60
3x3 conv	26, 26, 32	1,168,128	1,760
2x2 maxpool	13, 13, 32	-	-
dropout	13, 13, 32	-	-
3x3 conv with padding	13, 13, 64	3,115,008	18,496
3x3 conv	11, 11, 64	4,460,544	36,928
2x2 maxpool	5, 5, 64	-	-
dropout	5, 5, 64	-	-
flatten	1600	-	-
fully connected	512	819,200	819,712
dropout	512	-	-
fully connected (softmax activation)	20	10,240	10,260
Total		9,615,456	887,216

	(H, W, C)		
Input	28, 28, 1	-	-
3x3 conv with padding	28, 28, 6	42,336	60
3x3 conv	26, 26, 24	876,096	1,320
1x1 conv (pointwise conv)	26, 26, 12	194,688	300
2x2 maxpool	13, 13, 12	-	-
dropout	13, 13, 12	-	-
3x3 conv	13, 13, 28	511,056	3,052
2x2 maxpool	6, 6, 28	-	-
1x1 conv (pointwise conv)	6, 6, 24	24,192	696
2x2 maxpool	3, 3, 24	-	-
dropout	3, 3, 24	-	-
flatten	216	-	-
fully connected	100	21,600	21,700
dropout	100	-	-
fully connected (softmax activation)	20	2,000	2,020
Total		1,673,968	29,148

(Note: all the convolutional layers and fully connected layers use the ReLU activation function, apart from the last layer which uses the softmax activation function.)

Some of the changes introduced between the original network and our final architecture include:

- Reducing the number of filters in the convolutional layers and the number of neurons in the fully connected layers - we discovered through experimentation that removing filters and neurons from our network does not significantly impact its accuracy (if removed in a certain way), suggesting that many of the parameters (i.e. filters, weights, etc.) were used inefficiently in the original network. By using less kernels and neurons, we not only decrease the parameter count (and hence use less resources/memory to store those parameters on the board), but we also reduce the amount of computation required to perform the convolutions and matrix multiplications (for the fully connected layers).
- Added an extra max-pooling layer to further reduce the height and width of the feature maps to reduce the computational cost of the subsequent layers.
- Added pointwise convolution - pointwise convolution is a technique that can be used to reduce the amount of computation needed in the following layer by shrinking the size of the feature map. We introduced 2 pointwise convolution layers to our network: one was placed between

the 2 largest convolutional layers of the network to help reduce the computational cost of the latter layer, while another was positioned as the last convolutional layer to reduce the computational cost of the first fully connected layer. Apart from being used for reducing computational cost, pointwise convolutions add an extra layer of nonlinearity to the network, which can help improve model accuracy (17). As such, we were able to bring down the number of parameters and multiply-accumulate operations without significantly impacting the CNN's accuracy.

In order to help us decide on the most suitable network architecture to use, we used the following table and decision matrix to compare the best architectures (in terms of accuracy) we discovered through experimentation.

CNN architecture comparison (only best architectures shown)

	Architecture 1 (Original)	Architecture 2	Architecture 3	Architecture 4	Architecture 5	Architecture 6	Architecture 7	Architecture 8	Architecture 9	Architecture 10	Architecture 11 (Final)
Summary											
Total parameter count	887,216	452,016	188,328	98,568	138,908	136,800	181,496	73,944	34,680	30,932	29,148
Total multiply-accumulate operations	9,615,456	6,206,560	5,666,656	3,856,736	4,299,904	4,026,208	5,013,984	3,596,256	2,665,184	2,010,016	1,673,968
Validation accuracy at 20 epochs (%)	> 91	> 90	> 90	> 90	> 90	> 89	> 89	90	88.3	88.1	87.9
Breakdown											
Parameter count - convolutional layers	57,244	31,644	29,332	19,060	19,656	17,548	23,844	14,596	8,932	7,212	5,428
Parameter count - fully connected layers	829,972	420,372	158,996	79,508	119,252	119,252	157,652	59,348	25,748	23,720	23,720
Multiply-accumulate operations - convolutional layers	8,786,016	5,786,720	5,507,936	3,777,376	4,180,864	3,907,168	4,856,544	3,537,120	2,639,584	1,984,416	1,648,368
Multiply-accumulate operations - fully connected layers	829,440	419,840	158,720	79,360	119,040	119,040	157,440	59,136	25,600	25,600	25,600

Decision matrix

Criteria	Total parameter count (normalized score)	Total MACs (normalized score)	Validation error (normalized score)	Total weighed score (lower is better)
----------	--	-------------------------------	-------------------------------------	---------------------------------------

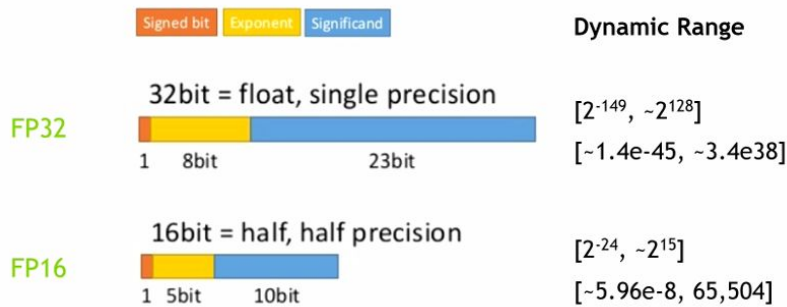
Weight	30%	50%	20%	
Arch 1	887k (1)	9.62M (1)	< 9 (0.74)	0.74
Arch 2	452k (0.51)	6.21M (0.65)	< 10 (0.83)	0.25
Arch 3	188k (0.21)	5.67M (0.59)	< 10 (0.83)	0.10
Arch 4	98.6k (0.11)	3.86M (0.40)	< 10 (0.83)	0.04
Arch 5	138.9 (0.16)	4.30M (0.45)	< 10 (0.83)	0.06
Arch 6	136.8 (0.15)	4.03M (0.42)	< 11 (0.91)	0.06
Arch 7	181.5 (0.20)	5.01M (0.52)	< 11 (0.91)	0.09
Arch 8	73.9 (0.08)	3.60M (0.37)	10 (0.83)	0.02
Arch 9	34.7 (0.039)	2.67M (0.28)	11.7 (0.97)	0.011
Arch 10	30.9 (0.035)	2.01M (0.21)	11.9 (0.98)	0.007
Arch 11	29.1 (0.033)	1.67M (0.17)	12.1 (1)	0.006 (best)

Model quantization

Quantization refers to the practice of using reduced precision values in a neural network, such as using 8-bit integers (INT8) or 16-bit floating point (FP16) numbers to represent the weights and activations, rather than using the full-precision 32-bit floating point numbers used during training. The benefit of quantization is that we can reduce the memory/resources used for storing our weights and activations (as each value is represented with less bits), and reduce latency as the DSPs (digital signal processor, used for performing multiply-accumulate operations) on the board can only accept an 18-bit number for one of its inputs.

Initially, we attempted to implement INT8 quantization using the open-source Tensorflow Lite Converter tool for quantizing neural network models, but after running the tool, the resulting quantized model was not giving the correct outputs for almost all of the input images we tested it with. We decided to try another approach where we attempted to manually quantize the model, and although we were successful in manually quantizing the first 2 layers of the CNN, we encountered a significant accuracy drop after quantizing the 3rd layer. In the end, we decided to use FP16 quantization as it was much easier to implement (we simply replaced our “float” declarations with the FP16 “half” data type provided in Vivado HLS and casted the existing weights to FP16). Both FF and LUT utilization were roughly halved after applying this.

FP32 and FP16 data types



http://redcatlabs.com/2018-07-19_TFandDL_MixedPrecision/img/float32and16_800x333.png

Data augmentation - one limitation of our current system is that it assumes the input video is of an on-screen sketchpad with a white background, whereas the user may want to use our recognition system on other media such as by drawing on a piece of paper or whiteboard. As such, we could consider using data augmentation (i.e. artificially generating additional training data from existing training data) to train our CNN to work more effectively in these scenarios, by applying various transformations to the existing training images to generate new images, such as brightening/darkening parts of the image (to mimic real-world lighting conditions), shearing/rotating the image (to mimic variations in viewing angle in the real world), and changing the stroke widths and textures if possible to the drawings in the QuickDraw dataset (to mimic the different drawing utensils users would use). Through data augmentation we may be able to get our system to perform well in a wider variety of situations, without having to collect actual training data.