BTP Assessment Report

"Automatic Conversion of DL models to RTL"

For the period

July 2023 to November 2023

by

Suyash Jaiswal

Roll No: B20EE070

Under the supervision of Dr. Binod Kumar

Department of Electrical Engineering

॥ त्वं ज्ञानमयो विज्ञानमयोऽसि ॥

भारतीय प्रौद्योगिकी संस्थान, जोधपुर

Indian Institute of Technology Jodhpur
NH 62, Nagaur Road, Karwar,
Jodhpur 342030 INDIA

Introduction

Within the context of deep learning, the rapid development and complexity of neural network structures have propelled the field to new heights, attaining extraordinary outcomes across a wide range of applications. With the increasing computing requirements of these models, there is a crucial requirement for specialised hardware solutions that can efficiently handle and enhance the execution of complex deep learning algorithms. Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) are notable platforms that show promise in enabling custom hardware design and parallelized processing. This makes them well-suited for overcoming the limitations in performance that are associated with standard computing architectures.

The Register-Transfer Level (RTL) is an essential abstraction in hardware description languages like Verilog. It is vital in converting high-level algorithms into efficient hardware implementations. This project aims to connect advanced deep learning models with specialised hardware by creating an automated method that converts deep learning models into Verilog RTL code.

The difficulties inherent in this undertaking are numerous. Deep learning models possess a wide range of structures and configurations in their layers, requiring a versatile and adaptive method for generating RTL code. Integrating pre-trained models from widely-used frameworks like TensorFlow and PyTorch presents distinct issues that necessitate thoughtful deliberation. The project aims to address these difficulties by creating a collection of strong Verilog templates and employing optimisation techniques to improve the efficiency of the resulting hardware.

This project seeks to enhance the capabilities of developers and researchers in utilising unique hardware for deep learning applications by automating the conversion process. The optimised Verilog RTL code has the potential to uncover new possibilities in real-time processing. It can be deployed in scenarios where low latency and energy efficiency are of utmost importance, due to its performance and resource utilisation. This introduction provides a foundation for a thorough examination of the project, including the reasons behind it, the objectives, the difficulties, and the expected influence of the suggested automated conversion method.

Problem Statement: Design and Implement an automated system which takes Deep Learning models as input and generates its corresponding Verilog code by using the predefined templates.

- Develop a mechanism to identify the input DL models from user.
- Create a set of various Verilog templates that can be used in various networks as a layer (function)
- Develop a mapping mechanism to associate DL model components with corresponding sections in Verilog templates.
- Develop a method to generate and merge the complete verilog code from each layer of the network.
- Derive methods to check whether the Generated code is correct or not.

Background:
Deep learning has come a long way in the past few years, allowing for the creation of complicated models that are very good at things like image recognition, natural language processing, and self-driving systems. As the need for efficient adoption of these models grows, so does the need to look into specialised hardware solutions that can speed up their execution.

Register-Transfer Level (RTL) is an important part of designing hardware. It is a level of abstraction in hardware description languages (HDLs) like Verilog. RTL shows how data moves between registers in a digital device and is used as a foundation for making hardware that works well. To get the most out of custom hardware, it's important to convert deep learning models to RTL. This lets you make efficient and optimised hardware architectures for faster inference jobs.

This project's motivate comes from the desire to be more efficient and specialised in how deep learning hardware is implemented. The project's goal is to make FPGAs and ASICs work as well as they can by automatically turning different DL models into Verilog RTL code.

Our work mainly included these 3 steps:

- Finding the model name from the .py file.
- Generating an output file containing the structure of the model.
- Verilog code generation for the architecture.

First we have written the code to identify the structures of the DL model. This code identifies and extracts the model names from Python files related to deep learning networks. Here's a breakdown of its functionality:

- `find_model(`string `file_name) Function:`
  - Opens a Python file specified by `file_name`
  - Scans through the file line by line.
  - Breaks the line into words.
    * `device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")`
  - Word are:
    * `device` `=` `torch.device("cuda")` `if` `torch.cuda.is_available()` `else` `torch.device("cpu")`
  - From all the words, we find the word `"torch.hub.load"` that was used to import the model.
    * `resnet50 = torch.hub.load('NVIDIA/DeepLearningExamples:torchhub', 'nvidia_resnet50', pretrained=True)`
  - The next word to this will be the model name ie,
    * `The model in the .py file is nvidia_resnet50`
  - If no model is found it returns:
    * `The model in the .py file is Model Not Found`

Using this model name, an output file, containing the architecture of the model is produced. From the list of files of architecture, it matches the model name and copies the architecture from the file to the output file.

Verilog code for the architecture:

- The parameters for different layers are stored in the map.
  -
```
{"vgg16" : {"parameter WIDTH" : [224, 224, 112, 112, 56, 56, 56, 28, 28, 28, 14, 14, 14],
            "parameter HEIGHT" : [224, 224, 112, 112, 56, 56, 56, 28, 28, 28, 14, 14, 14],
            "parameter CHANNELS" : [3, 64, 64, 128, 128, 256, 256, 256, 512, 512, 512, 512, 512],
            "parameter FILTERS" : [64, 64, 128, 128, 256, 256, 256, 512, 512, 512, 512, 512, 512],
            "parameter KERNEL_SIZE" : [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
            "parameter STRIDE": [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
            "parameter PADDING" : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]},
```

- The architecture of the code was written in the following way:
  - No of inputs = n
  - Previous output, x1, x2...xn-1 = The inputs to the current layer
  - (n, Inputs, Name of the layer)
    - 1, 2, ReLu,
    - 2, 3, 1, Adder
- For each layer, we append the layer number too.
  - LayerName_LayerNumber
  - This is done so as to have different names for the modules, else module with the same name cannot be declared again.
  - The same is done to the output of the layer too:
  - New output = LayerOutput_LayerNumber
  - This is done so that all the outputs have a unique name and are not redeclared again.
  - All the outputs are stored in a list. This is done so that we can use the previous outputs easily using the index.
- For each of the Verilog files, the input is replaced with the previous layer's output and the output is replaced with the new output.
- Adder is a Verilog file that can take atmost 10 inputs and sum all of them.

```
module vectorAdder(
  input [31:0][0:(1024*1024*1024)-1] vector_a ,
  input [31:0][0:(1024*1024*1024)-1] vector_b ,
  input [31:0][0:(1024*1024*1024)-1] vector_c ,
  input [31:0][0:(1024*1024*1024)-1] vector_d ,
  input [31:0][0:(1024*1024*1024)-1] vector_e ,
  input [31:0][0:(1024*1024*1024)-1] vector_f ,
  input [31:0][0:(1024*1024*1024)-1] vector_g ,
  input [31:0][0:(1024*1024*1024)-1] vector_h ,
  input [31:0][0:(1024*1024*1024)-1] vector_i ,
  input [31:0][0:(1024*1024*1024)-1] vector_j ,
  output reg [31:0][0:(1024*1024*1024)-1] result_vector
);

  integer idx;

  // Vector addition
  always @* begin
    for (idx = 0; idx < (1024*1024*1024); idx = idx + 1) begin
      result_vector[idx] = vector_a[idx] +
                           vector_b[idx] +
                           vector_c[idx] +
                           vector_d[idx] +
                           vector_e[idx] +
                           vector_f[idx] +
                           vector_g[idx] +
                           vector_h[idx] +
                           vector_i[idx] +
                           vector_j[idx]
                                        ;
    end
  end

endmodule
```

- Here we made sure all the inputs are in different lines. This helps us comment the particular input if they are not in use. Line = "//" + Line. We had to comment out these unused vectors as they contain garbage value and will give us wrong results if they are added.

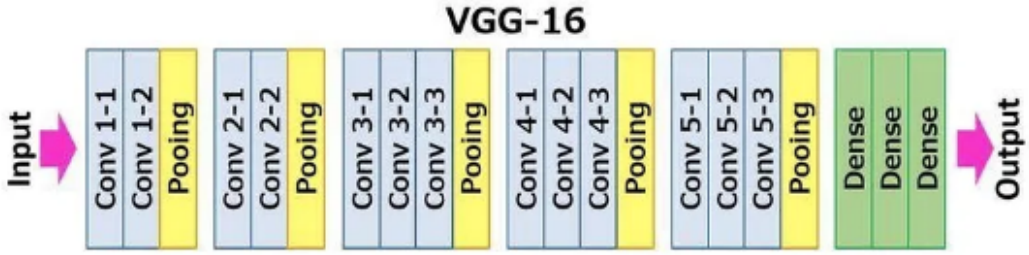
```
// Define parameters
parameter WIDTH = 224;
parameter HEIGHT = 224;
parameter CHANNELS = 3;
parameter FILTERS = 64;
parameter KERNEL_SIZE = 3;
parameter STRIDE = 1;
parameter PADDING = 1;
```

- .
  - Iterating over the Verilog file line by line, if we find these parameters, these are replaced with the value that was earlier stored in the dictionary.
- For the final Verilog file, we require a top module which contains the layers' name along with their inputs and outputs. While iterating over the layers, we had stored all the inputs and outputs which helped us to form the top module.

We took following trained deep neural network:

- VGG16
- AlexNET
- ResNET-18
- Squeezenet

VGG16 Network Structure:
Overview: The VGG16 architecture consists of multiple convolutional and fully connected layers, culminating in a SoftMax output layer. Each layer's specifications include the filter size, stride, zero padding, input and output dimensions, and activation function applied.

- 

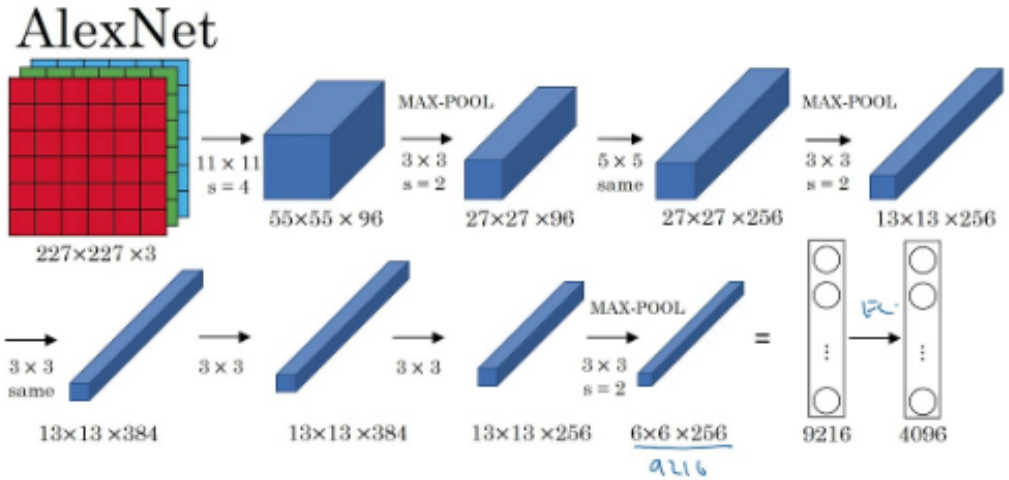
- Layer Configuration:

VGG16 - Structural Details													
#	Input Image			output			Layer	Stride	Kernel		in	out	Param
1	224	224	3	224	224	64	conv3-64	1	3	3	3	64	1792
2	224	224	64	224	224	64	conv3064	1	3	3	64	64	36928
	224	224	64	112	112	64	maxpool	2	2	2	64	64	0
3	112	112	64	112	112	128	conv3-128	1	3	3	64	128	73856
4	112	112	128	112	112	128	conv3-128	1	3	3	128	128	147584
	112	112	128	56	56	128	maxpool	2	2	2	128	128	65664
5	56	56	128	56	56	256	conv3-256	1	3	3	128	256	295168
6	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
7	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
	56	56	256	28	28	256	maxpool	2	2	2	256	256	0
8	28	28	256	28	28	512	conv3-512	1	3	3	256	512	1180160
9	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
10	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
	28	28	512	14	14	512	maxpool	2	2	2	512	512	0
11	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
12	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
13	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
	14	14	512	7	7	512	maxpool	2	2	2	512	512	0
14	1	1	25088	1	1	4096	fc		1	1	25088	4096	102764544
15	1	1	4096	1	1	4096	fc		1	1	4096	4096	16781312
16	1	1	4096	1	1	1000	fc		1	1	4096	1000	4097000
Total													138,423,208

-

AlexNET Network Structure:

Overview: The AlexNet architecture encompasses convolutional, pooling, fully connected layers, and SoftMax output. Each layer's specifications detail the filter size, stride, padding, input and output dimensions, and activation functions applied



-

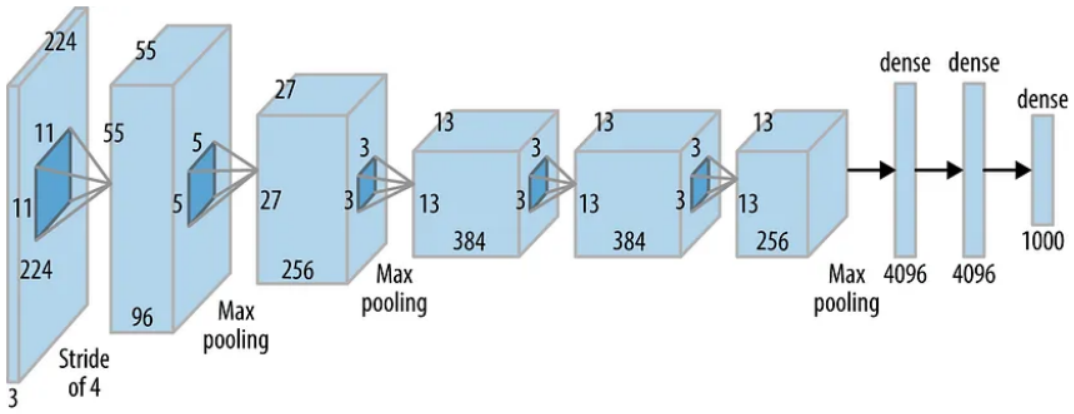
- Layer configuration:

AlexNet Network - Structural Details													
Input			Output			Layer	Stride	Pad	Kernel size		in	out	# of Param
227	227	3	55	55	96	conv1	4	0	11	11	3	96	34944
55	55	96	27	27	96	maxpool1	2	0	3	3	96	96	0
27	27	96	27	27	256	conv2	1	2	5	5	96	256	614656
27	27	256	13	13	256	maxpool2	2	0	3	3	256	256	0
13	13	256	13	13	384	conv3	1	1	3	3	256	384	885120
13	13	384	13	13	384	conv4	1	1	3	3	384	384	1327488
13	13	384	13	13	256	conv5	1	1	3	3	384	256	884992
13	13	256	6	6	256	maxpool5	2	0	3	3	256	256	0
						fc6			1	1	9216	4096	37752832
						fc7			1	1	4096	4096	16781312
						fc8			1	1	4096	1000	4097000
Total													62,378,344

-

ResNET-18 Network Structure:

Overview: The network consists of 5 Convolutional (CONV) layers and 3 Fully Connected (FC) layers. The activation used is the Rectified Linear Unit (ReLU). The structural details of each layer in the network can be found in the table below.

- 

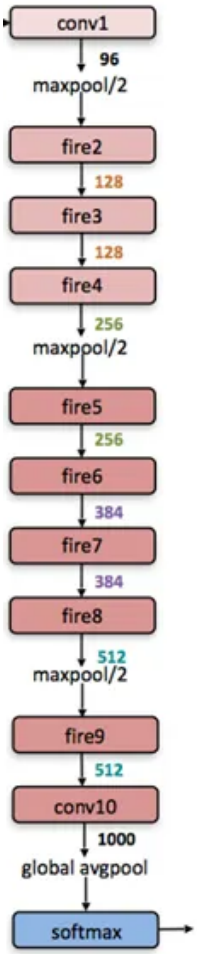
- Layer configuration:

AlexNet Network - Structural Details												
Input			Output			Layer	Stride	Pad	Kernel size		in	out	# of Param
227	227	3	55	55	96	conv1	4	0	11	11	3	96	34944
55	55	96	27	27	96	maxpool1	2	0	3	3	96	96	0
27	27	96	27	27	256	conv2	1	2	5	5	96	256	614656
27	27	256	13	13	256	maxpool2	2	0	3	3	256	256	0
13	13	256	13	13	384	conv3	1	1	3	3	256	384	885120
13	13	384	13	13	384	conv4	1	1	3	3	384	384	1327488
13	13	384	13	13	256	conv5	1	1	3	3	384	256	884992
13	13	256	6	6	256	maxpool5	2	0	3	3	256	256	0
						fc6			1	1	9216	4096	37752832
						fc7			1	1	4096	4096	16781312
						fc8			1	1	4096	1000	4097000
Total													**62,378,344**

-

SqueezeNet Network Structure:

Overview: SqueezeNet, as the name suggests is a deep neural architecture that has something to do with a "squeezed" network for training networks for image classification. It deals with the prospect of deploying ML models in embedded systems which are highly resource constrained applications.

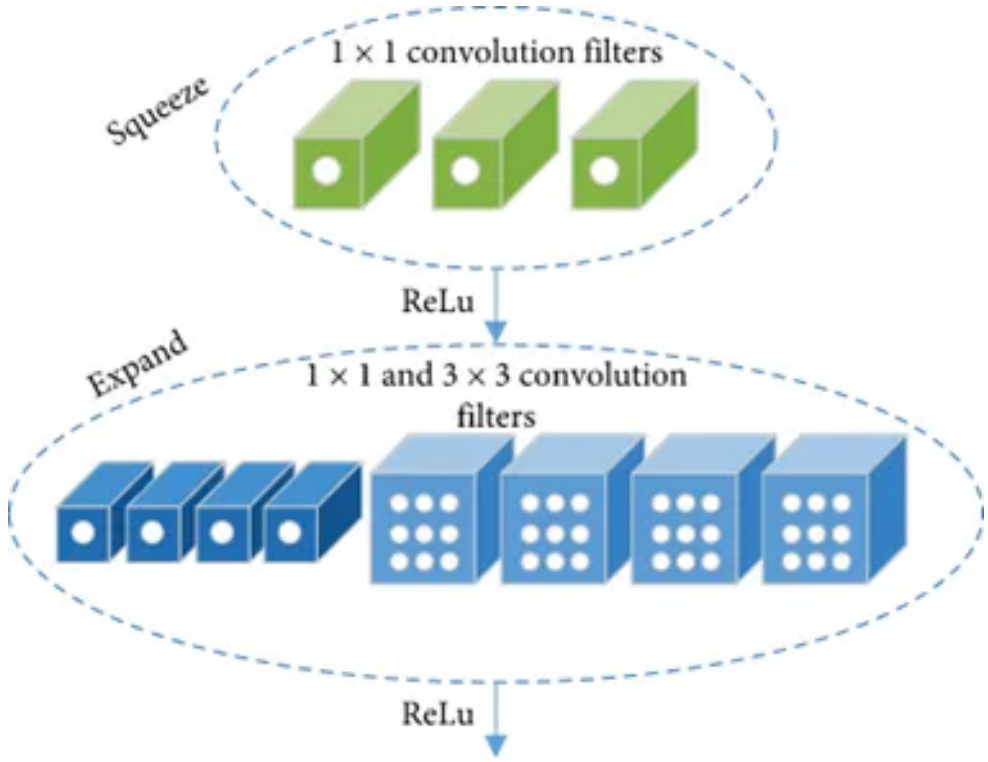


-

- Layer configuration:

layer name/type	output size	filter size / stride (if not a fire layer)	depth	s_{1x1} (#1x1 squeeze)	e_{1x1} (#1x1 expand)	e_{3x3} (#3x3 expand)	s_{1x1} sparsity	e_{1x1} sparsity	e_{3x3} sparsity	# bits	#parameter before pruning	#parameter after pruning
input image	224x224x3										-	-
conv1	111x111x96	7x7/2 (x96)	1				100% (7x7)			6bit	14,208	14,208
maxpool1	55x55x96	3x3/2	0									
fire2	55x55x128		2	16	64	64	100%	100%	**33%**	6bit	11,920	5,746
fire3	55x55x128		2	16	64	64	100%	100%	**33%**	6bit	12,432	6,258
fire4	55x55x256		2	32	128	128	100%	100%	**33%**	6bit	45,344	20,646
maxpool4	27x27x256	3x3/2	0									
fire5	27x27x256		2	32	128	128	100%	100%	**33%**	6bit	49,440	24,742
fire6	27x27x384		2	48	192	192	100%	**50%**	**33%**	6bit	104,880	44,700
fire7	27x27x384		2	48	192	192	**50%**	100%	**33%**	6bit	111,024	46,236
fire8	27x27x512		2	64	256	256	100%	**50%**	**33%**	6bit	188,992	77,581
maxpool8	13x12x512	3x3/2	0									
fire9	13x13x512		2	64	256	256	**50%**	100%	**30%**	6bit	197,184	77,581
conv10	13x13x1000	1x1/1 (x1000)	1				**20%** (3x3)			6bit	513,000	103,400
avgpool10	1x1x1000	13x13/1	0									
	activations		parameters				compression info				1,248,424 (total)	**421,098** (total)

- Fire Modules:
  - The Fire module comprises: A squeeze convolution layer (which has only 1x1 filters), feeding into an expand layer that has a mix of 1x1 and 3x3 convolution filters.



Conclusion: Successful Verilog Code Generation for VGG16, AlexNet, Resnet and Squeezenet.

The successful generation of Verilog code for several models highlights the flexibility and versatility of the automated conversion method that was built. The complex structures of models presented distinct difficulties that have been effectively resolved, demonstrating the system's capability to handle a range of deep learning models.

The use of pre-existing templates has been crucial in simplifying the conversion process, by providing a basis that can adapt to the complexities of each model's structure. This strategy not only guarantees precision in the depiction of the models but also simplifies the capacity to expand and incorporate more models in the future.

Ultimately, the achievement of generating Verilog code for well-known deep learning models such as VGG16, AlexNet, Resnet and Squeezenet. represents a noteworthy advancement in the effort to create efficient and customised hardware implementations. This accomplishment signifies the successful completion of efforts to connect high-level deep learning models with the Register-Transfer Level (RTL), allowing for effortless incorporation into hardware description languages such as Verilog.

The successful generation of Verilog code for several models highlights the flexibility and versatility of the automated conversion method that was built. The complex structures of models presented distinct difficulties that have been effectively resolved, demonstrating the system's capability to handle a range of deep learning models.

The use of pre-existing templates has been crucial in simplifying the conversion process, by providing a basis that can adapt to the complexities of each model's structure. This strategy not only guarantees precision in the depiction of the models but also simplifies the capacity to expand and incorporate more models in the future.

Future Works:

Support for Additional Models: Provided we get the correct verilog files for the layers, we can extend to other models such as shufflenet etc.

FPGA Implementation: In future stages of the research, a key objective is to efficiently incorporate and enhance the Verilog RTL code produced for deep learning models on Field-Programmable Gate Arrays (FPGAs). This entails enhancing the generated code to achieve optimal utilisation of resources on FPGAs, investigating optimisations that are specific to FPGAs, and creating reliable interfaces to facilitate efficient communication. The study will investigate the use of dynamic reconfiguration capabilities to adjust the hardware accelerator according to different workloads. Additionally, energy-efficient design techniques such as dynamic voltage and frequency scaling and power gating will be applied. The focus will be on attaining the ability to make immediate deductions, and thorough evaluation will confirm the effectiveness and acceleration gained on FPGA technology. To enhance the practical applicability of the project in accelerating deep learning tasks, it is important to ensure compatibility with commonly used FPGA development tools, offer user guidance, and explore high-level synthesis languages such as OpenCL. These measures will facilitate the smooth deployment of Verilog code on FPGAs.

Using More Templates:

For future versions of the project, a planned enhancement of the system's functionality includes augmenting the quantity and variety of templates accessible for Verilog code production. This update seeks to adapt to a wider variety of architectures, layer types, and specialised operations often seen in evolving neural network structures, in order to address the diverse landscape of deep learning models. Expanding the template library will enhance the automated conversion system's flexibility, enabling it to easily adjust to new and complex model configurations. This expansion could include customised templates designed for certain deep learning frameworks, addressing differences in layer setups and activation functions. Moreover, the expanded variety of templates will enable users to select or modify templates according to their own hardware limitations and optimisation preferences, resulting in a more adaptable and user-focused experience. The objective is to cultivate a complete and dynamic template ecosystem that corresponds to the ever-changing nature of deep learning model architectures. This will guarantee the project's ongoing relevance and capacity to adjust to increasing industry standards and research advancements.

References: https://www.researchgate.net/publication/340470168/figure/fig3/AS:877502188769280@1586224233016/of-the-MobileNet-architecture-A-The-overall-MobileNet-architecture-and.png
https://github.com/AniketBadhan/Convolutional-Neural-Network
https://www.analyticsvidhya.com/blog/2021/03/introduction-to-the-architecture-of-alexnet/
https://medium.com/@mygreatlearning/everything-you-need-to-know-about-vgg16-7315defb5918
https://iq.opengenus.org/vgg19-architecture/
https://www.analyticsvidhya.com/blog/2021/03/introduction-to-the-architecture-of-alexnet/?
https://www.researchgate.net/figure/Original-ResNet-18-Architecture$_fig1_336642248$
https://iq.opengenus.org/resnet50-architecture/
https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96

https://towardsdatascience.com/review-squeezenet-image-classification-e7414825581a
https://www.analyticsvidhya.com/blog/2021/03/introduction-to-the-architecture-of-alexnet/?
https://medium.com/@avidrishik/squeezenets-architecture-compressed-neural-network-7741d24ca56f
https://towardsdatascience.com/review-shufflenet-v1-light-weight-model-image-classification-5b253dfe982f
https://medium.com/syncedreview/shufflenet-an-extremely-efficient-convolutional-neural-network-for-mobile-devices-72c6f5b01651
https://www.researchgate.net/figure/Proposed-Modified-ResNet-18-architecture-for-Bangla-HCR-In-the-diagram-conv-stands-for$_fig1_323063171$