The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

DLPaper2Code: Auto-Generation of Code from Deep Learning Research Papers

Akshay Sethi*
IIIT Delhi

Anush Sankaran, Naveen Panwar, Shreya Khare, Senthil Mani
IBM Research AI

Abstract

With an abundance of research papers in deep learning, reproducibility or adoption of the existing works becomes a challenge. This is due to the lack of open source implementations provided by the authors. Even if the source code is available, then re-implementing research papers in a different library is a daunting task. To address these challenges, we propose a novel extensible approach, DLPaper2Code, to extract and understand deep learning design flow diagrams and tables available in a research paper and convert them to an abstract computational graph. The extracted computational graph is then converted into execution ready source code in both Keras and Caffe, in real-time. An *arXiv*-like website is created where the automatically generated designs is made publicly available for $5,000$ research papers. The generated designs could be rated and edited using an intuitive drag-and-drop UI framework in a crowd sourced manner. To evaluate our approach, we create a simulated dataset with over $216,000$ valid deep learning design flow diagrams using a manually defined grammar. Experiments on the simulated dataset show that the proposed framework provide more than 93% accuracy in flow diagram content extraction.

Introduction

The growth of deep learning (DL) in the field of artificial intelligence has been astounding in the last decade with about $63,600$ research papers being published since 2017[1]. Keeping up with the growing literature has been a real struggle for researchers and practitioners. In one of the recent AI conferences, NIPS 2016, the maximum number of papers submitted ($\sim 685/2500$) were in the topic, "Deep Learning or Neural Networks". However, a majority of these research papers are not accompanied by their corresponding implementations. In NIPS 2016, **only** $101/567$ ($\sim 18\%$) papers made their source implementation available[2]. Implementing research papers takes at least a few days of effort for software engineers, assuming that they have limited knowledge in DL (Sankaran et al. 2011).

Another major challenge is the availability of various libraries in multiple programming languages to implement DL algorithms such as Tensorflow (Abadi et al. 2016), Theano (Bastien et al. 2012), Caffe (Jia et al. 2014), Torch (et al 2011), MXNet (Chen 2015), DL4J (Gibson 2015), CNTK (Seide and Agarwal 2016) and wrappers such as Keras (Chollet and others 2015), Lasagne (Dieleman 2015), and PyTorch (Chintala 2016). The public implementations of the DL research papers are available in various libraries offering very little interoperability or communication across them. Consider a use-case for a researcher working in "image captioning", where three of the highly referred research papers for the problem of image captioning[3] are:

1. Show and Tell (Vinyals et al. 2015): Original implementation available in Theano; https://github.com/kelvinxu/arctic-captions
2. NeuralTalk2 (Karpathy and Fei-Fei 2015): Original implementation available in Torch; https://github.com/karpathy/neuraltalk2
3. LRCN (Donahue et al. 2015): Implementation available in Caffe; http://jeffdonahue.com/lrcn/

As the implementations are available in different libraries, a researcher cannot directly combine the models. Also, for a practitioner having remaining of the code-base in Java (DL4J) directly leveraging either of these public implementations would be daunting. Thus, we highlight two highly overlooked challenges in DL:

1. Lack of public implementation available for existing research works and thus, the time incurred in reproducing their results
2. Existing implementations are confined to a single (or few) libraries limiting portability into other popular libraries for DL implementation.

We observed that most of the research papers explain the DL model design either through a figure or a table. Thus, in this research we propose a novel algorithm that automatically parses a research paper to extract the described DL model design. The design is represented as an abstract computational graph which is independent of the implementation library or language. Finally, the source code is generated in multiple libraries from this abstract computational

*Akshay Sethi interened at IBM Research, India during this work.

[1]https://scholar.google.co.in/scholar?as_ylo=2017&q=deep+learning&hl=en&as_sdt=1,5

[2]https://www.kaggle.com/benhamner/nips-papers

[3]https://competitions.codalab.org/competitions/3221#results

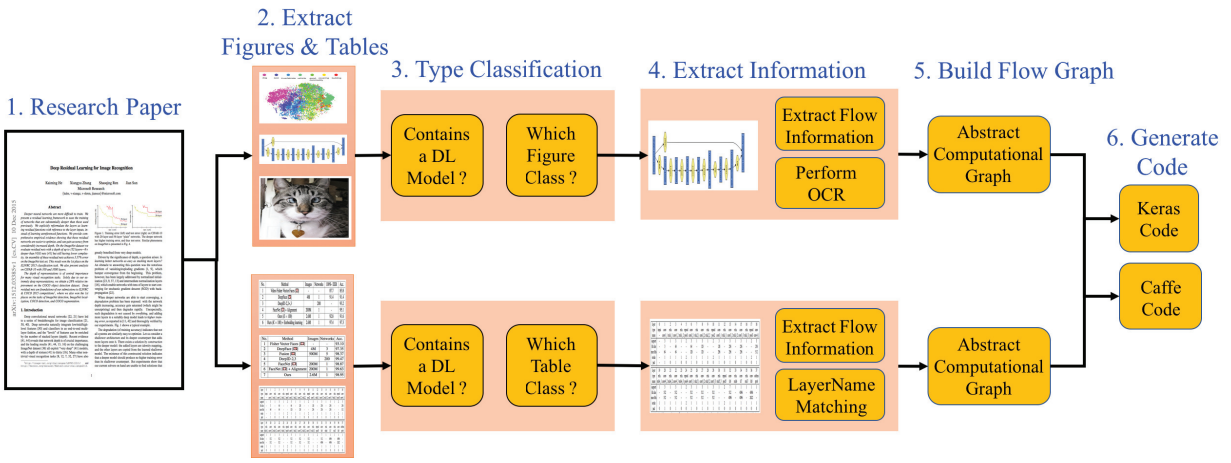

Figure 1: The architecture of the proposed system that extracts and understands the flow diagrams and tables from a deep learning research paper and generates an execution ready deep learning code in two different platforms: Keras and Caffe.

graph of the DL design. The results are shown by automatically generating the source code of $5,000$ *arXiv* papers both in CAFFE (pyCaffe + prototxt) and Keras (python). However, evaluating the generated source code is debatable due to the lack of ground truth. To overcome this challenge, we simulated a large image dataset of $216,000$ valid DL model designs in both Caffe and Keras. To generate the dataset, we manually defined a grammar for DL models. As these visualizations are highly varying, they are comparable to the figures present in research papers. The major research contributions are:

1. a technique to automatically understand a DL model design by parsing the figures and tables in a research paper,
2. generate source code both in both Keras and Caffe from the abstract computation graph of a DL design,
3. automatically generate design for $5,000$ *arXiv* papers and build a UI system[4] for editing in a crowd sourced way,
4. simulate a dataset of $216,000$ DL model visualizations using a manually defined grammar, where the proposed approach achieves more than 95% accuracy in automated flow extraction.

The rest of the paper is organized as follows: Section 2 explains the entire proposed approach for auto generation of DL source code from research paper, Section 3 talks about the simulated dataset and the experimental performance of the individual components of the proposed approach, Section 4 discusses the experimental results on $5,000$ *arXiv* DL papers, and Section 5 concludes this paper with some discussion on our future efforts.

DLPaper2Code: Proposed Approach

Consider a state-of-art paper DL paper (Szegedy et al. 2017) published in AAAI 2017, which explains the DL design model through figures, as shown in Figure 3(a). Similarly, in the AAAI 2017 paper by (Parkhi et al. 2015), the DL model design was explained using a table, as shown in Figure 4. Given the PDF of a research paper in deep learning, the proposed DLPaper2Code architecture consists of five major steps, as shown in Figure 1: (i) Extract all the figures and tables from a research paper. Handling the figures and tables are done independently, although they follow a similar pipeline, (ii) As there could be many descriptive images and results tables in a paper, classify each figure or table if whether it contains a DL model design. Also, perform a fine grained classification on the class of figure or table used to describe the DL model, (iii) Extract the flow and the text information from the figures and tables, independently, (iv) Construct an abstract computational graph which is agnostic of the implementation library, and (v) generate source code in both Caffe and Keras from the computational graph.

Characterizing Research Papers

We observed that in a research paper the DL design is mostly explained using figures or tables. Thus, it is our assertion that by parsing the figure, as an image, and the table content, the respective novel DL design could be obtained. The primary challenges with the figures in research papers is that the DL design figures typically do not follow any definition and show extreme variations. Similarly, tables can have different structures and can entail different kind of information.

We manually observed more than $30,000$ images from research papers and characterized the DL design flow diagrams images into five broad classes, as shown in Figure 2. The five classes are: (i) Neurons plot: the classical representation of a neural network with each rectangular layer having circular nodes inside them, (ii) 2D Box: each hidden layer is represented as a 2D rectangular box, (iii) Stacked2D Box: each layer is represented as a stack of 2D rectangular boxes, describing the depth of the layer, (iv) 3D Box: each hidden layer is represented as a 3D cuboid structure, and (v) Pipeline plot: along with the DL model design, the end-to-end pipeline and mostly some intermediate results of

[4]DARVIZ: https://darviz.mybluemix.net/

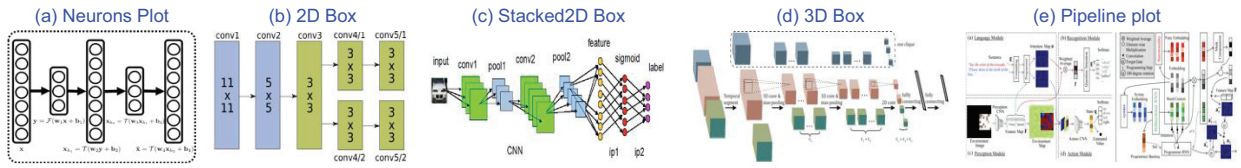

Figure 2: Characterizing the DL model designs available in research papers and grouping them into five different categories.

image/ text is shown as well. Similarly, based on the representation, tables can be classified as, (i) row-major table: where the DL model design flows along the row (Springenberg et al. 2014), and (ii) column-major table: where the DL model design flows along the column (Parkhi et al. 2015). It is essential to account for these variations in the proposed pipeline, as they indicate the DL design flow represented in the paper. Following our assumptions, proposed approach does not identify a DL design flow that is neither in a table nor in a figure.

Extracting Figures and Tables

Extracting visual figures from a PDF document, especially from a scholarly report is a well studied problem (Choudhury and Giles 2015). Common challenges include extracting vector images as they are embedded in the PDF document and extracting a large figure as a whole instead of multiple sub-figures. To this end, we have used a publicly existing tool called *PDFFigures 2.0*[5] (Clark and Divvala 2016) for extracting a list of figures from a scholarly paper. However, none of the existing open source tools preserve the table structure, which is essential for us. Thus, we built a PDF table extraction tool combining PDFMiner[6] and Poppler-utils[7]. Poppler-utils provide high level information about the document such as the text dump, while using PDFMiner certain low level document details such as vertical line spacing are obtained. The table structure, along with the table caption, is retrieved by building the heuristics over the horizontal and vertical line spacing.

Figure and Table Classification

The aim is to classify and retrieve only those figures and tables in a research paper that contains a DL design flow. Further, a fine-grained classifier is required to classify the figure into one of the identified five broad classes and classify the table as a row-major or column-major flow.

In case of figures, the classifier is trained to perform the prediction using the architecture shape and the flow. For example, figures having result graphs and showing sample images from dataset has different shape compared to an architecture flow diagram. All the figures are resized to 224×224 and $4,096$ features (*fc2*) are extracted from a fully connected layer of a popular deep learning model *VGG19* (Simonyan and Zisserman 2014) pre-trained on ImageNet dataset. We have two classification levels: (i) Coarse classifier: a binary neural network (NNet) classifier trained on *fc2* features to classify if the figure contains a DL model or not, and (ii) Fine-grained classifier: a five class neural network classifier trained on *fc2* features to identify the type of DL design, only for those figures classified positive by the coarse classifier. Having a sequence of two classifiers provided better performance as compared to a single classifier with six classes (sixth class being no DL design flow).

In case of tables, a bag-of-words model is built using keywords from the caption text as well as the table text. A cosine distance based classifier is used to identify if there is a DL design flow in the given table as compared to tables containing results. Further based on the number of rows and columns in the table, as extracted in the previous section, the table is classified as a row-major or column-major flow.

Content Extraction from Figure

Content extraction from figures has two major steps: (i) flow detection to identify the nodes and the edges, and (ii) OCR to extract the flow content. Identifying the flow is the challenges, as there is a huge variation in the type of DL design flow diagrams. In this section, we explain the details of the approach for a 2D Box type, as shown in Figure 3, while similar approach could be extended to other classes, as well. Flow detection involves identifying the nodes first, followed by the edges connecting the nodes. As the image is usually of high resolution and quality, they are directly binarized using an adaptive Gaussian thresholding and a Canny edge detection approach is used to identify all the lines. An iterative region grown algorithm is adopted to identify closed contours in the figure, as they represent the nodes as shown in Figure 3(b). All the detected nodes are masked out from the figure and the contour detection algorithm is applied again to detect the edges, as shown in Figure 3(d). The direction of the edge flow is obtained by analyzing the pixel distribution within each edge contour. The node and edge contours are then sorted based on the location and direction to obtain the flow of the entire DL model design. As shown in Figure 3, the proposed approach could also handle branchings and forking in a design flow diagram.

Once the flow is extracted, the text in each node/ layer is obtained through OCR using Tesseract[8].Based on our manual observation, we assume that the a layer description would be available within the detected node. A dictionary of possible DL layer names is created to perform spell correction of the extracted OCR text.

[5]https://github.com/allenai/pdffigures2

[6]https://euske.github.io/pdfminer/

[7]https://poppler.freedesktop.org/

[8]https://github.com/tesseract-ocr/

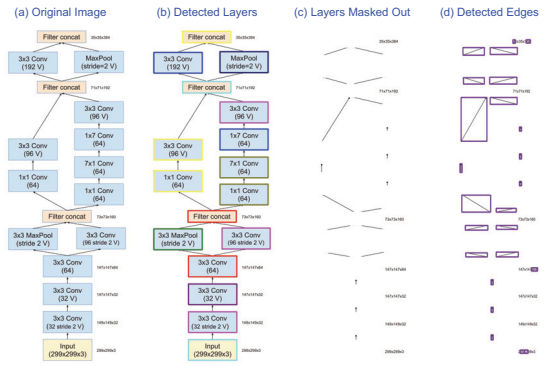

Figure 3: Illustration of the proposed flow detection approach from complex figures (Szegedy et al. 2017) (AAAI 2017) involving (b) node/ layer detection, and (d) edge detection.

layer	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
type	input	conv	relu	conv	relu	mpool	conv	relu	conv	relu	mpool	conv	relu	conv	relu	conv	relu	mpool	conv
name	–	conv1_1	relu1_1	conv1_2	relu1_2	pool1	conv2_1	relu2_1	conv2_2	relu2_2	pool2	conv3_1	relu3_1	conv3_2	relu3_2	conv3_3	relu3_3	pool3	conv4_1
support	–	3	1	3	1	2	3	1	3	1	2	3	1	3	1	3	1	2	3
filt dim	–	3	–	64	–	–	64	–	128	–	–	128	–	256	–	256	–	–	256
num filts	–	64	–	64	–	–	128	–	128	–	–	256	–	256	–	256	–	–	512
stride	–	1	1	1	1	2	1	1	1	1	2	1	1	1	1	1	1	2	1
pad	–	1	0	1	0	0	1	0	1	0	0	1	0	1	0	1	0	0	1

layer	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
type	relu	conv	relu	conv	relu	mpool	conv	relu	conv	relu	conv	relu	mpool	conv	relu	conv	relu	conv	softmx
name	relu4_1	conv4_2	relu4_2	conv4_3	relu4_3	pool4	conv5_1	relu5_1	conv5_2	relu5_2	conv5_3	relu5_3	pool5	fc6	relu6	fc7	relu7	fc8	prob
support	1	3	1	3	1	2	3	1	3	1	3	1	2	7	1	1	1	1	1
filt dim	–	512	–	512	–	–	512	–	512	–	512	–	–	512	–	4096	–	4096	–
num filts	–	512	–	512	–	–	512	–	512	–	512	–	–	4096	–	4096	–	2622	–
stride	1	1	1	1	1	2	1	1	1	1	1	1	2	1	1	1	1	1	1
pad	0	1	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0

Figure 4: An example table showing the DL design flow as explained in tabular format in (Parkhi et al. 2015).

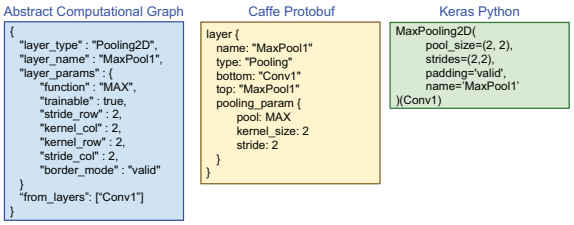

Figure 5: An illustration for a Pooling2D layer showing the rule base of the inference engine, converting the abstract JSON format into Caffe protobuf and Keras python code.

Content Extraction from Table

In a row major table, every row corresponds to a layer in the DL design flow, as shown in Figure 4. Similarly, in a column major table, every column corresponds to a layer along with other parameters of the layer. The layer name is extracted by matching it with a manually created dictionary. Further, the parameters are extracted by mapping the corresponding row or column header with a pre-defined list of parameter names corresponding to the layer. Thus, sequentially the entire DL design flow is extracted from a table.

Generating Source Code

Overall, after detecting DL design flow, an abstract computational graph is represented in JSON format, as shown in Figure 5. Two rule based converters are written to convert the abstract computational graph extracted in the previous step to either (i) Keras code (Python) or (ii) Caffe code (pyCaffe + prototxt). An inference engine acts as the converter to map the abstract computational graph to the grammar of the library. The inference engine consists of a comprehensive list of templates and dictionaries built manually for both Keras and Caffe. Template based structures transfer each component from the abstract representation to a platform specific structure using the dictionary mappings. Further, another set of templates, consisting of a set of assertions, are designed for translating each layer's parameters. The inference engine is highly flexible allowing easy extension and addition of new layer definitions. An example of the inference engine's dictionary mapping for a Pooling2D layer is shown in Figure 5.

Thus for a given research paper, by processing both the figure and table content, the DL model design flow is obtained which is converted to execution ready code in both Keras and Caffe.

Evaluation on Simulated Dataset

The aim of this process is to simulate and generate ground truth deep learning designs and their corresponding flow visualizations figures. Thus, the proposed pipeline of DL model design could be quantitatively evaluated. To this end, we observed that both Keras and Caffe have an in-built visualization routine for a DL design model. Further, both Keras and Caffe have their internal DL model validator and a visualization can be exported only when the simulated design is deemed valid by the validator.

Grammar for DL Design Generation

To be able to generate meaningful and valid DL design models, we manually defined a grammar for the model flow as well as for the hyper-parameters. We considered 10 unique layers for our dataset simulation - $\{$*Conv2D, MaxPool2D, AvgPool2D*$\}$ for building convolution neural network like architectures, $\{$*Embed, SimpleRNN, LSTM*$\}$ for building recurrent neural network like architectures, $\{$*Dense, Flatten, Dropout, Concat*$\}$ as the core layers. The use of *Concat* enables our designed models to be non-sequential as well as with a combination of recurrent and convolution architectures. This allows us to create random, complex, and highly varying DL models. Also, *RNN* and *LSTM* layers have an additional binary parameter of *return seq*, which when set true returns the output of every hidden cell, otherwise, returns the output of only the last hidden cell in the layer. Table 1 explains the proposed grammar for generating DL design models. The grammar defines the set of all possible next layers for a given current layer. This is determined by the shape of the tensor flowing through each of the layer's operation. For example, a *Dense* layer strictly expects the input to be a vector of shape $n \times 1$. Thus, the *Dense* cannot appear after a *Conv2D* layer without the presence of a *Flatten* layer. The proposed grammar further includes the set of possible values for each hyper-parameter of a layer, as explained in Table 2. While hyper-parameter values beyond the defined bounds are possible, the table values indicate the assumed set of values in the model simulation process.

Current Layer	Dense	Conv2D	Flatten	Dropout	MaxPool	AvgPool	Concat	Embed	RNN	RNN (seq)	LSTM	LSTM (seq)
Input	✔	✔						✔				
Dense	✔			✔			✔	✔				
Conv2D		✔	✔	✔	✔	✔	✔					
Flatten	✔			✔			✔	✔				
Dropout	Same as previous layer											
MaxPool		✔	✔	✔	✔	✔	✔					
AvgPool		✔	✔	✔	✔	✔	✔					
Concat	If input is one dimensional, same as Dense layer; else same as previous layer											
RNN	✔			✔			✔	✔				
RNN (seq)			✔	✔			✔		✔	✔	✔	✔
LSTM		✔		✔			✔	✔				
LSTM (seq)			✔	✔			✔		✔	✔	✔	✔

Table 1: The proposed grammar for creating valid deep learning design models defining the list of possible next layers for a given current layer.

Layer	Hyper-parameters
Dense	#nodes - {[5:5:500]}
Dropout	probability - {[0:0.1:1]}
Conv2D	#filters - {[16:16:256]}
	filter size - {[1:2:11]}
MaxPool	stride - {[2:1:5]}
	filter size - {[1:2:11]}
AvgPool	stride - {[2:1:5]}
	filter size - {[1:2:11]}
Embed	embed size - {64, 100, 128, 200}
	vocab - {[10000, 20000, 50000, 75000]}
SimpleRNN	#units - {[3:1:25]}
LSTM	#nodes - {[3:1:25]}
InputData	MNIST - {28, 28, 1}
	CIFAR-10 - {32, 32, 3}
	ImageNet - {224, 224, 3}
	IMDB Text

Table 2: The set of hyper-parameter options considered for each layer in our simulated dataset generation. The parameters [a:b:c] is a list of values from a to c in steps of b.

Simulated Dataset

A model simulation starts with an *Input* layer, where there are four possible options - *MNIST, CIFAR, ImageNet, IMDBText*. From the set of all possible next layers for the given *Input* layer, a completely random layer is decided. For the given next layer, a random value is picked for every possible hyper-parameter. For example, for *MNIST* being the input layer, *Conv2D* could be picked as the random next layer. For *Conv2D* the hypar-parameters are determined randomly as 32 filters, 5×5 filter size with a stride of 2. The design always ends with a *Dense* layer with number of nodes equal to the number of classes of the corresponding *Input* layer.

The number of layers in between the *Input* layer and the final *Dense* layer denotes the depth of the DL model. For our simulation, we generated $3,000$ DL models for each depth unique between 5 and 40, creating a total dataset of $108,000$ models. Each model contains the Keras JSON representation, Keras image visualization, Caffe protobuf files, and Caffe image visualization, resulting in a total of $216,000$ DL model design visualizations. These models are valid by construct since they follow a well-defined grammar. However, these models need not be the best from an execution perspective, or with respect to their training performance. Also, the visualizations from Caffe and Keras are very similar to the one found in research papers (covering more than 70% of the visualizations actually found in research papers).

Observation	Train	Validation	Test
#DataPoints	$195,296$	$48,824$	$48,824$
Naive Bayes	98.29%	98.30%	98.39%
Decision Tree	100%	99.57%	99.55%
Logistic Regression	100%	99.98%	99.99%
RDF	100%	99.72%	99.68%
SVM (RBF Kernel)	100%	99.89%	99.83%
Neural Network	100%	99.93%	99.94%

Table 3: The performance of various binary classifiers to distinguish KerasCaffeVisulizations vs. other often occuring images in research papers.

Figure Type Classification Performance

In this experiment, a binary NNet classifier with two hidden layers of size [1024, 256] is trained on *fc2* features of *VGG19* model to differentiate $216,000$ simulated DL visualizations from a set of $28,120$ other kind of diagrams often available in research papers (scraped from PDF). The whole dataset is split as 60% for training, 20% for validation, and 20% for testing, making it a total of $195,296$ images for training and validation, and $48,824$ images for testing. The performance of the NNet classifier is compared with six different classifiers as shown in Table 3. As it can be observed most of the classifier provide a classification accuracy of 100%, showing that from a set of figures obtained from a research paper, it would be possible to distinguish only the

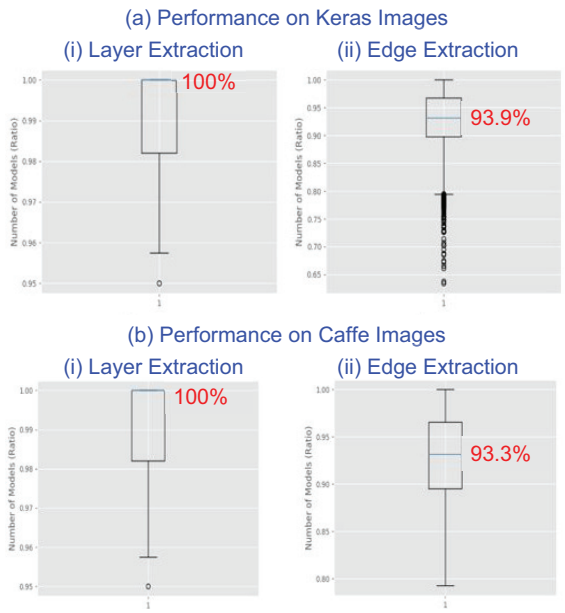

Figure 6: Box plots showing the performance accuracy of flow detection in Keras and Caffe visualizations.

deep learning design flow diagrams. All the classifiers use default parameters as provided by the *scikit-learn* package.

Computational Graph Extraction Performance

In this experiment, the performance of flow and content extraction from the $216,000$ Keras and Caffe visualizations is evaluated against the ground truth. By performing OCR, on the extracted flow, the unique layer names are obtained and two detection accuracies are reported,

1. blob (or layer) detection accuracy: evaluates the performance of blob detection and layers identified using OCR and is computed as the ratio of correct blobs detected per model (in percent)
2. edge detection accuracy: evaluates the performance of the detected flow and is computed as the ratio of correct edges detected per model (in percent)

Figure 6 is the box plot showing the performance of the proposed computational graph extraction pipeline in both Keras and Caffe. As it can be observed, the proposed pipeline provides on an average 100% accuracy in layer extraction and more than 93% accuracy in extracting the edges. As the edges can be curved and can be of any length, even connecting the first with the last layer, the variations caused a reduction in performance.

Results on Deep Learning Scholarly Papers

$5,000$ papers were downloaded from *arXiv.org* using *"deep learning"* as the input query. $30,987$ figures were extracted from these downloaded papers, out of which $28,120$ figures did not contain a DL design flow while the remaining $2,867$ contained. These represent typically found figures in a deep learning research paper that does not contain a design flow.

Observation	Train	Validation	Test
#DataPoints	$18,592$	$6,197$	$6,198$
Naive Bayes	77.29%	64.39%	62.56%
Decision Tree	99.96%	76.67%	74.35%
Logistic Regression	99.96%	86.17%	85.27%
RDF	99.96%	83.72%	82.94%
SVM (RBF Kernel)	99.96%	86.89%	85.25%
Neural Network	99.96%	87.93%	86.25%

Table 4: The performance of coarse level binary classifier to distinguish DL design flow figures from other figures that usually appear in a research paper.

Observation	Train	Validation	Test
#DataPoints	$1,720$	573	574
Naive Bayes	40.42%	54.30%	52.84%
Decision Tree	99.65%	50.57%	49.13%
Logistic Regression	99.65%	69.98%	68.47%
RDF	99.65%	68.72%	66.02%
SVM (RBF Kernel)	99.65%	72.94%	69.68%
Neural Network	100%	74.93%	71.60%

Table 5: The performance of fine level five class classifier to identify the type of DL design flow figure obtained from the research paper.

Figure Type Classification Accuracy

To evaluate the coarse level binary classification, a two hidden layer NNet was trained on the *fc2* features obtained from the $30,987$ images extracted from research papers. The whole dataset is split as 60% for training, 20% validation, and 20% for testing and the results are computed for seven different classifiers as shown in Table 4. More than 86% average accuracy is obtained on binary classification.

Further, to evaluate the fine level, five-class, figure type classification, the $2,867$ DL design flow diagrams were manually labeled. The distribution of figures were as follows: (i) Neurons plot: 586 figures, (ii) 2D box: $1,204$, (iii) Stacked2D box: 407, (iv) 3D box: 561, and (v) Pipeline plot: 109. A $60-20-20$ train, validation, and test split is performed to train the NNet classifier in comparison with six other classifiers, to perform this five class classification. The results are in Table 5. The results show that even on highly varying DL flow design images, identifying the exact class of DL flow is more than 71% accurate.

Crowd sourced Improvement of Extracted Designs

Using the proposed DLPaper2Code framework, we extracted the DL design model flow diagrams for all the $5,000$ downloaded papers. However, quantitatively evaluating the extracted design flow would be challenging due to the lack of a ground truth. Hence, we created an *arXiv*-like website[9], as shown in Figure 7, where the papers, the corresponding design, and the generated source code is available. The community could rate the extracted designs which acts as a feedback or performance measure of our automated approach.

[9]https://darviz.mybluemix.net/

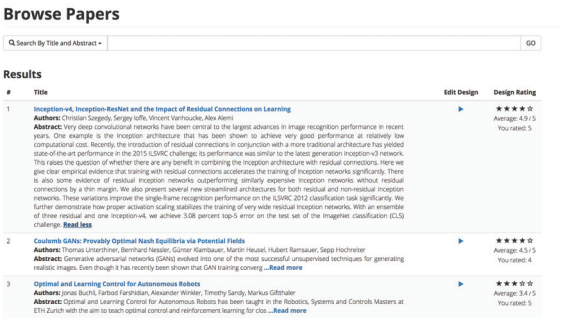

Figure 7: An *arXiv*-like website where DL papers along with their extracted design, and generated source code in Caffe and Keras is made available.

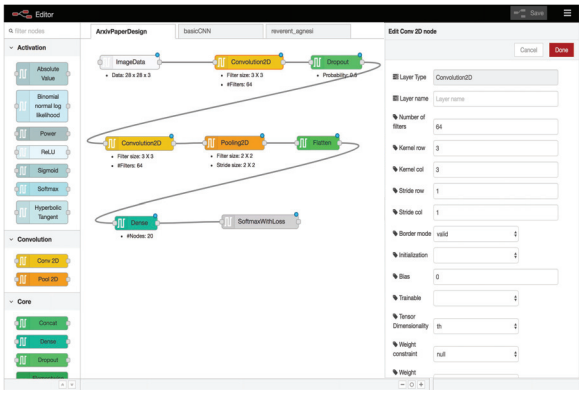

Figure 8: An intuitive drag-and-drop UI based framework to edit the extracted DL model designs and make them publicly available.

Further, an intuitive drag-and-drop based UI framework is generated for the community to edit the generated DL flow design, as shown in Figure 8. Ideally the respective papers' author or the DL community could edit the generated designs, wherever an error was found. The edited design could be further made publicly available for other researcher to reproduce the design. Further our system could generate the source code of the edited design in both Keras and Caffe, in real-time. Thus, we have a two-fold advantage through this UI system: (i) the public system could act as a one-stop repository for any DL paper and it's corresponding design flow and source code, (ii) the community feedback would enable us to continuously learn and improve the system.

Conclusion and Discussion

Thus using this research work, the DL model design explained in a research paper could be automatically extracted. Using an intuitive drag-and-drop based UI editor, developed as a part of this research, the extracted design could be manually edited and perfected. Further, for an extracted DL design, the source code could be generated in Keras (Python) and Caffe (pyCaffe + prototxt), in real-time. The proposed DLpaper2Code framework extracts both figure and table information from a research paper and converts it into source code. Currently, an *arXiv*-like website is created that contains the DL design and the source code for $5,000$ research papers. To evaluate our approach, we simulated a dataset of $108,000$ unique deep learning designs validated by a proposed DL model grammar and their corresponding Keras and Caffe visualizations. On a total dataset of $216,000$ deep learning model visualization flow diagrams and $28,120$ diagrams that appeared in deep learning research papers and did not contains a model visualization, the proposed binary classification using NNet classifier obtained 99.9% accuracy. The performance of extracting information from computational graph figures using the proposed pipeline is more than 93% accurate. The system and DLPaper2Code feature is found at: https://darviz.mybluemix.net/

While this research could have a high impact in the reproducibility of DL research, we have planned for plenty of possible extensions for the proposed pipeline:

1. The proposed pipeline detects only the layers (blobs) and the edges from the diagram. It could be extended to detect and extract the hyper-parameter values of each layer, to make the computational graph more content rich.
2. Currently, we have two independent pipelines for generating abstract computational graphs from tables and figures. Combining the information obtained from the multimodal sources could enhance the accuracy of the extracted DL design flow.
3. The entire DLPaper2Code framework could be extended to support additional libraries, apart from Keras and Caffe, such as Torch, Tensorflow
4. The broader aim would be to propose a standard definition of representing DL model design in research papers and better readability. Further, authors of future papers could also release their design in the created website for easy accessibility to the community.

References

Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Bastien, F.; Lamblin, P.; Pascanu, R.; Bergstra, J.; and Goodfellow, I. J. e. a. 2012. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.

Chen, T. e. a. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

Chintala, S. 2016. Pytorch. https://github.com/pytorch/pytorch.

Chollet, F., et al. 2015. Keras. https://github.com/fchollet/keras.

Choudhury, S. R., and Giles, C. L. 2015. An architecture for information extraction from figures in digital libraries. In *WWW (Companion Volume)*, 667–672.

Clark, C., and Divvala, S. 2016. Pdffigures 2.0: Mining figures from research papers. In *Digital Libraries (JCDL), 2016 IEEE/ACM Joint Conference on*, 143–152.

Dieleman, S. 2015. Lasagne: First release.

Donahue, J.; Anne Hendricks, L.; Guadarrama, S.; Rohrbach, M.; Venugopalan, S.; Saenko, K.; and Darrell, T. 2015. Long-term recurrent convolutional networks for visual recognition and description. In *Computer vision and pattern recognition*, 2625–2634.

et al, R. C. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

Gibson, A. 2015. Dl4j. https://github.com/deeplearning4j/deeplearning4j.

Jia, Y.; Shelhamer, E.; Donahue, J.; Karayev, S.; Long, J.; Girshick, R.; Guadarrama, S.; and Darrell, T. 2014. Caffe: Convolutional architecture for fast feature embedding. In *ACM international conference on Multimedia*, 675–678.

Karpathy, A., and Fei-Fei, L. 2015. Deep visual-semantic alignments for generating image descriptions. In *Computer Vision and Pattern Recognition*, 3128–3137.

Parkhi, O. M.; Vedaldi, A.; Zisserman, A.; et al. 2015. Deep face recognition. In *BMVC*, volume 1, 6.

Sankaran, A.; Aralikatte, R.; Mani, S.; Khare, S.; Panwar, N.; and Gantayat, N. 2011. DARVIZ: Deep abstract representation, visualization, and verification of deep learning models: Nier track. In *International Conference on Software Engineering*, 804–807.

Seide, F., and Agarwal, A. 2016. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, 2135–2135. New York, NY, USA: ACM.

Simonyan, K., and Zisserman, A. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Springenberg, J. T.; Dosovitskiy, A.; Brox, T.; and Riedmiller, M. 2014. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.

Szegedy, C.; Ioffe, S.; Vanhoucke, V.; and Alemi, A. A. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, 4278–4284.

Vinyals, O.; Toshev, A.; Bengio, S.; and Erhan, D. 2015. Show and tell: A neural image caption generator. In *Computer Vision and Pattern Recognition*, 3156–3164.