Colourizing black and white pictures (without user interactions)

Ákos Horváth, Gergely Mátyás, Barnabás Gurubi – *balit_learning*

Abstract—**We approach the problem of colourizing black and white (gray scale) images. The colour of pixel is highly depends on the features of its neighbors, hence we use Convolutional Neural Networks (CNNs). Our work contains an Auto-Encoder type network which is not pre-trained. The model itself is fairly large, contains a huge number of parameters therefore the training takes considerably long time. We train on 70 thousand images approximately. Our system gets these images in gray scale format and outputs the colour channels. Unlike the training, the generation of the coloured images from their pale version is rapid enough, especially compared to the training time. We propose our final solution to be automatic, so no user input should be required. This problem contains huge uncertainty, due to its difficulty, for this reason our results are not perfect, nevertheless they are worth to be considered.**

Kivonat—**A kitűzött probléma a fekete-fehér képek színezése. Egy képet alkotó pixelek színe nagymértékben függ a szomszédos pixelek tulajdonságától, ebből kifolyólag a megoldás során Konvolúciós Neurális Hálókat (CNNs) használunk. A munkánk egy Auto-Encoder típusú hálót dolgoz ki, ami nem tartalmaz előzőleg betanított részt. Maga a modell meglehetősen nagy, rendkívül sok paramétert tartalmaz, ezáltal a tanítása igen hosszú időt vesz igénybe. Hozzávetőlegesen 70 ezer képen tanítjuk a modellt. A rendszer fekete-fehér formában kapja a képeket bemenetként, és kimenetként visszaadja a szín csatornákat. A tanítással ellentétben a színes képek generálása az eredeti színtelen verzióból kellően gyors, főleg a tanuláshoz képest. A végső megoldás automata, vagyis nem igényel felhasználói beavatkozást, segítséget. Maga a probléma sok bizonytalanságot hordoz magában a nehézségéből kifolyólag, ennek következtében az eredmények nem tökéletesek, ennek ellenére hasznosnak találjuk a fontolóra vételüket.**

I. Introduction

LETS take a look at the problem. Given a gray scale image, the prediction of the colours seems to be a rather fair or even a bit too complex task, taking the possible variates into consideration. Although observing the image more carefully, we can start to rely on our experiences since we know the original (or common) colour of lots of things. For example, the sky during a bright day is blue, or trees used to be green (of course in certain conditions). Needless to say, that many objects can have more than one possible colour, like clothes or other artificial entities. Our goal was to give a plausible prediction. Albeit these predictions often imperfect and incomplete, they have creditable parts also.

To give a comparison of how complex the problem is let's examine how they colourize black and white frames during the colourization of a movie with the classic methods: The artist typically begins by dividing the picture into regions and then assigning a colour to each region. This method is called segmentation method and it is a very time-consuming process that can take hours or much more to finish. In contrast our network takes only a few seconds to colourize an image (of course if we have already trained our network before) which is obviously not as good as if it had been colourized by an artist. During our work we used a PC with 16 GB memory and a GTX 1070 graphics card. This GPU is without question one of the strongest in the market that's available and affordable for the common users nowadays. Even though we had this relatively strong setup the training still lasted a long time (see in detail at the part where we discuss the teaching process).

II. Related work

This section gives a brief overview of the previous colourization methods.

Scribble-based colourization Levin propose an effective approach that requires the user to provide colourful scribbles on the grayscale target image. The colour information on the scribbles are then propagated to the rest of the target image using least-square optimization. Huang develop an adaptive edge detection algorithm to reduce the colour bleeding artifact around the region boundaries.

Unlike scribble-based colourization methods, the example-based methods transfer the colour information from a reference image to the target gray scale image. The accuracy of these methods heavily depend on the similarities of the example and input images, because these solutions work usually with the pixel intensity informations and neighbourhood statistics, so the example-based methods not really work well in global or general usage. Furthermore, we need example images selected by humans, and it requires identical objects and scenes on the images, but the overall solution works only mainly on static pictures, for example simple landscapes.

Nowadays, existing fully automated colourization solutions, usually modelled with convolutional neural networks. These models often contain fusion layers, with classification models to classify the pictures during the propagation. These classifications can be done by pre-trained models, to reach faster teaching period.

Initially, research focused on small set of outputs, in particular the classification task. However, they are now successfully applied to many different tasks in which the output is an image, such as optical flow, super-resolution, contour detection, and semantic segmentation. These are based on convolutional neural networks and can process images of any resolution.

We have also dealt with a fusion model, which contains a convolutional neural network with *ResNet* classification model, but it slows down the teaching process with almost 50%, and the results were almost the same. We tried to optimize our model with other parameters, activation functions without the fusion layers. In our opinion, the almost no improvement in result with the fusion model is caused by the relatively small dataset. (~70k)

III. The model

A. The network

The network we used is an Auto-Encoder-like CNN (Convolutional Neural Network). Which means it has an encoder, a narrow bottleneck and a decoder part. The difference is in the input and output. In an AE network we usually use the same data as input and output and goal is to compress the data. In our case we give the gray scale image as input and the a and b channels of the colourized pictures as the output (we use Lab colour space, see later at the implementation part). So not just the input and output are different, but moreover their shapes are also dissimilar.
Firstly, let's look at the input, which is a 256x256x1 sized matrix. It means that it has 256x256 pixels and only 1 channel.

The model consists of three main parts. *Encoder*, *Bottleneck*, *Decoder*. It goes through the *Encoder* first which was built from several convolutional layers (see the table below, conv2d_1 – conv2d_5). The *Bottleneck* two alike convolutional layers (conv2d_6 – conv2d_7). The task of the *Decoder* part is to make this 512@32x32 tensor to fit the desired output dimensions. To achieve the height and width we use Keras UpSampling2D layers, which repeats the rows and columns of the data by the given size respectively. (in our case we used size = (2,2) in every UpSampling2D layer) It uses the interpolation what's given as a parameter, we used the default nearest interpolation method. The depth of the output is 2 (the a and b colour channels), to make our data the wished depth we use several convolutional layers. This part is almost symmetrical to the encoder part, since we half the depth with each layer. (opposite to the encoder where we double the depth with each layer).

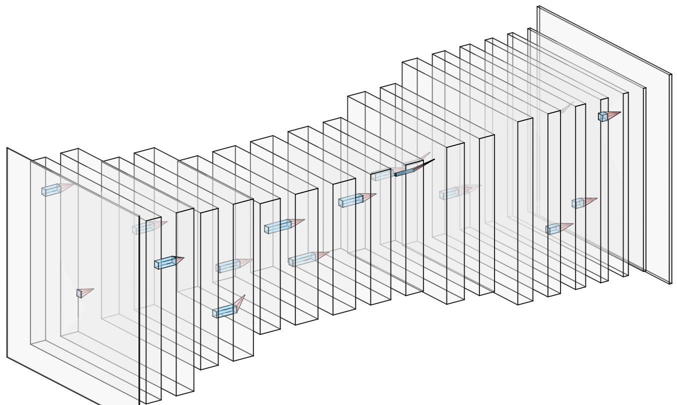


Fig. 1. The visualization of our model. The layers proceed from left to right (the leftmost one is the input gray scale image, the rightmost is the output ab colour channels).

TABLE I
Model summary

Layer	Quantity	Description [a]
conv2d_1	Conv2D	1@256x256, f:3x3@64, s:2x2, z:s[b]
conv2d_2	Conv2D	64@128x128, f:3x3@128, s:1x1, z:s
conv2d_3	Conv2D	128@128x128, f:3x3@128, s:2x2, z:s
conv2d_4	Conv2D	128@64x64, f:3x3@256, s:1x1, z:s
conv2d_5	Conv2D	256@64x64, f:3x3@256, s:2x2, z:s
conv2d_6	Conv2D	256@32x32, f:3x3@512, s:1x1, z:s
conv2d_7	Conv2D	512@32x32, f:3x3@512, s:1x1, z:s
conv2d_8	Conv2D	512@32x32, f:3x3@256, s:1x1, z:s
conv2d_9	Conv2D	256@32x32, f:3x3@128, s:1x1, z:s
up_sampling2d_1	UpSampling2D	128@32x32, (2,2) [c]
conv2d_10	Conv2D	128@64x64, f:3x3@64, s:1x1, z:s
up_sampling2d_2	UpSampling2D	64@64x64, (2,2)
conv2d_11	Conv2D	64@128x128, f:3x3@32, s:1x1, z:s
conv2d_12	Conv2D	32@128x128, f:3x3@16, s:1x1, z:s
conv2d_13	Conv2D	16@128x128, f:3x3@2, s:1x1, z:s
up_sampling2d_3	UpSampling2D	2@128x128, (2,2)

[a]*Description of the given layer in the following format:{depth}@{input_x}x{input_y}, f:{filter_x}x{filter_y}@{filter_depth}, s: {stride_x}x{stride_y}, z:{zeropadding}*
[b]*'s' in this case refers to 'same', in the Keras framework you can set the parameter padding='same', this takes proper care of the zeropadding (for more information, see the Keras framework documentation)*
[c]*UpSampling2D layers, which repeat the rows and columns of the data by the given size respectively. (in our case we used size = (2,2))*

B. About the activation functions

At the beginning we used the *ReLu* function as activation in all the layers except the last convolutional layer in the decoder part, where we used *tanh* since we wanted to get a result between -1 and 1. (the standardized values for a and b channel)
When we heard about the *'swish'* activation on lecture we did a small research about it. We found out that it was fairly similar to the *ReLu* function but it outperformed *ReLu* most of the time. We decided to implement and try *swish*. (in the teaching part where we write about optimization you can find the details)

The *swish* function: f(x) = x*sigmoid(x)

IV. Implementation

A. Preprocessing and acquisition of data

1) Getting and preprocessing the images

Since our goal was to colourize any pictures, we needed a dataset that includes pictures from different themes. Because of the high complexity we also needed the dataset to be large. Although we read in many articles that they used around a million images, we were limited by our resources. We decided to use a dataset of ~70k 256x256 sized images, which we downloaded from the site: https://pixabay.com/ using a script.

2) Loading and formatting the images

After we downloaded the database, we had to load it to our program and format them. As we read in a lot of articles for this problem it is better if we change the usual RGB channels to Lab representation. (where L is for lightness, a and b are for the colour spectrums green–red and blue–yellow) This way the L channel provides us the black and white pictures which represent the input and the network has to predict the a and b values. Also, it is better for the network if the values are between 0 and 1 so we 'normalized' them. (we know that L is between 0-100 and a is between -128 – 128, so as the b channel).

3) Splitting the images into datasets

The first vital step of organizing and processing the data is to split it into train, validation and test sets.

Considering our dataset size (circa 70k 256x256 images) loading all of them once into a huge list, or other data structure, then separate it into the desired sets would be very resource consuming, hence not too wise. Instead we introduce a rather easy way of data organization, which can be useful when dealing with large datasets.

The idea is straightforward. We give IDs to all data samples after this all data elements can be simply referred using its ID. Our goal is to organize not the dataset itself but the IDs since it is way cheaper regarding to resources (all IDs are numbers).

Our way to correlate IDs and images is done through the image names. All the images are renamed incrementally from 1, thus all the names are numbers. These names are excellent IDs. After we have the ID set (numbers in 1 to image number ~70k range, incremented by one) it can be shuffled then split. The result is a Python dictionary with the following keys: “train”, “validation”, “test” and values of the corresponding id set. The train split is 80%, validation and test 10% each.

4) Introducing DataGenerator

The next issue is still pertained to the size of the data. We cannot load the entire set and send it through our network due to the lack of memory. Taking the formerly mentioned set size into consideration we can calculate the required storage for the images represented with float numbers (floats are necessary if we want to normalize our data). One image (gray channel for input, the other to for output) represented with floats: 256x256x3x4 = 0.78 MB thereby ~70k images take more than 50 (!) GB, so this way is not an alternative.

This is a common dilemma among image oriented deep learning problems. To overcome this vast dilemma data generators are come in handy. The principal idea is to feed the network with chunks of data hence we do not need to store the entire set in the memory only a small part of it which we change time to time. There are lots of pre-implemented data generators in deep learning and data processing libraries, still we decided to make our own, this way we can use it exactly to our cause. Instances of this class can be used with the Keras *_generator functions. These functions are made directly for using networks with generators.

Our DataGenerator implementation makes use of the afore mentioned data organization practice. It loads the images in batches by the aid of the IDs. The output of the generator is a batch of input and output data from the adequate set, from which the generator is constructed.

B. Teaching and evaluation

We use a CNN which described in the *III.* section to colourize images. Firstly, we must train our model with several pictures from the previously mentioned source. In our case the dataset contains ~70k random images, which is enough for us to test our model and colourize gray scaled images, although a much accurate result could be achieved, if we would use a much larger dataset, unfortunatley with our resources and with the ~70k images, one epoch takes ~21 minutes (despite the afore mentioned Geforce GTX1070), and with 40-50 epoch necessary to train the model, a significantly larger dataset is out of the question in our case hence the training would last for several days.

With the well-known loss functions are slightly problematic for colourization due to the multi-modality of the problem. For example, if a gray dress could be red or blue, and our model picks the wrong colour, it will be harshly penalized. As a result, our model will usually choose desaturated colours that are less likely to be "very wrong" than bright, vibrant colours. We tried different sets of parameters and optimizers to get the best, but not too complex one.

We have trained our model with different optimization settings and activation functions, the two construction which seem to be work well were the *rms-prop* optimizer with *mean-squared* error and *Relu* activation function, and the *adam optimiser* with *mean-absolute error* and *swish* activation function.

The former construction produces quite good colours, but the edge detection is not as good as in the latter solution. We have tried to train our model with normalized gray scaled input images and non-normalized as well, and in the first case, with the non-normalised input data, the results look brownish and dull. Nevertheless, in the first case the normalization helps, in the second case, the model has not learned after the second epoch, but with non-normalized input data, the results look like almost the same as in the first case, but with better edge detection, with more accurately separated colours and less random colour patches generated in the test pictures.

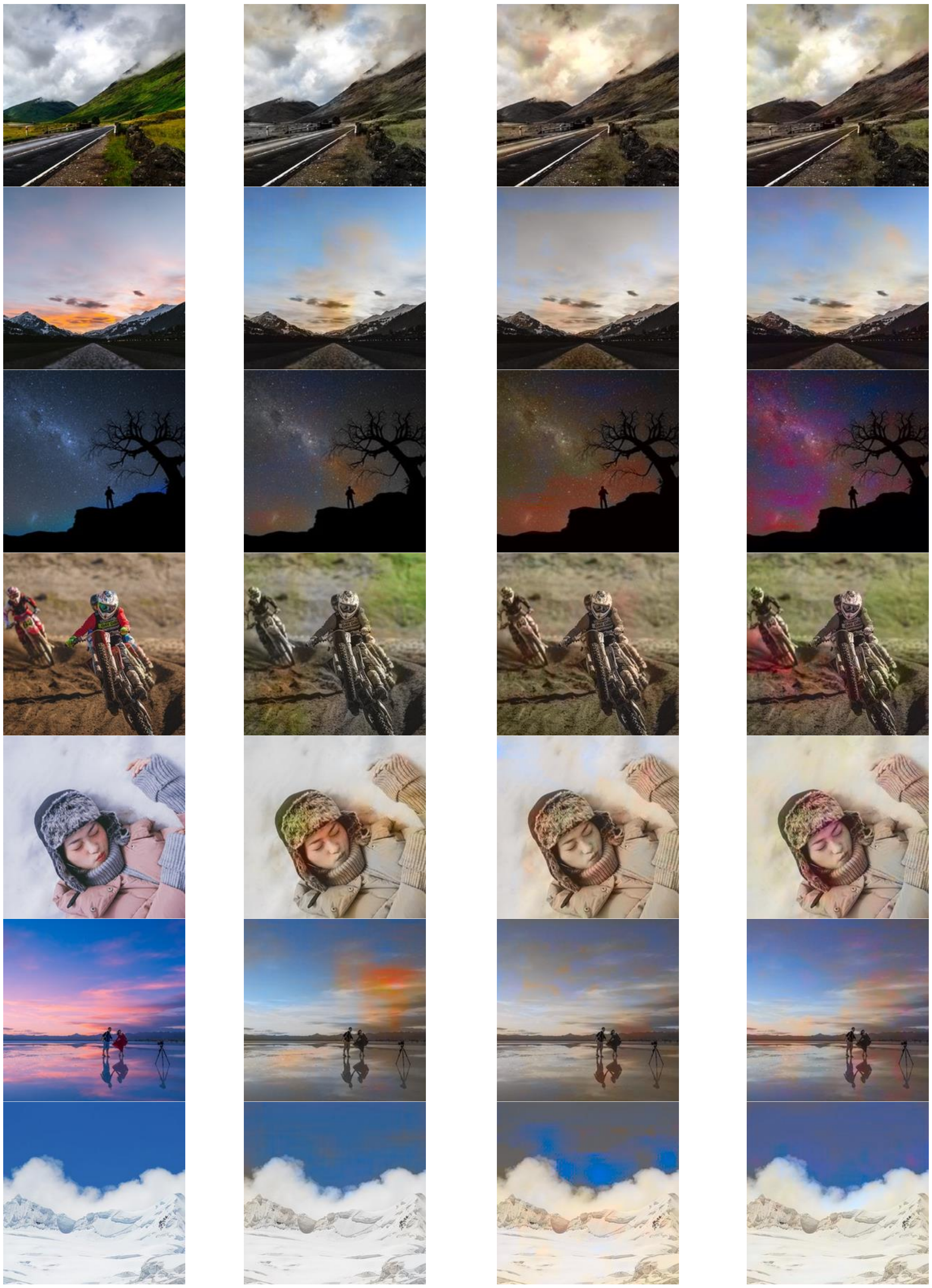
C. Testing

At this stage we need to keep in mind the original problem, colourization. We cannot purely rely on common testing and evaluating metrics, since the answer of the “How incorrect the colour is?” is not elementary. Taking all this into consideration, we have decided to test and evaluate the results of each learning with some given test images and our eyes.

A crucial point of every testing is the origin of the test data. It must be unknown to our model thereby we have chosen images from a completely different dataset.

We came upon some interesting revelation, for example the swish activation in our case was incompatible with the *rms-prop* optimizer (it did not learn) so as the *mean-absolute* error or with the *adam* optimizer the normalization of the gray channel only had disadvantageous effects on the outcomes. As previously mentioned, “each model” (they only differ in the activation, loss and occurrence of normalization) has its own style of colouring, we cannot squarely declare a best one.

Some colourization results are presented below. In all cases the original form of the image is also given. We also labeled the applied optimizer, loss function and the presence of the input normalization.


Original
Adam
mean_absolute_error
swish
without normalization
rsmprop
mean_squared_error
relu
without normalization
rsmprop
mean_squared_error
relu
with normalization

V. Future work

Of course, it is almost impossible to make the perfect colourization method because of the several reasons we mentioned in the previous parts. Yet we would like to achieve a better result we had done with the current network. We have several future plans that could make it work better.

In our opinion it would highly improve the results if we had used significantly more images for the training. Although we were limited by our resources, with more research and maybe stronger hardware it is not impossible to use e.g. ~1million images like we read in the paper written by Richard Zhang, Phillip Isola, Alexei A. Efros.

Another improvement could be if we used user interactions, where the user can influence the colourization by giving the colour of an area. As we mentioned it is impossible for the program to predict e.g. the colour of a T-shirt, so if the user helps with information it can achieve a better result as you can see it in in the paper written at University of California, Berkeley

We would like to test our model with even more losses and optimizers. We read about them in papers. But we would like to research the problem more and understand better the results of each loss function and optimizer.

Last but not least we were thinking about using this for videos as well. The simple solution would be if we sent through all the frames one by one on the current network. But somehow with other methods like LSTM we could use the information about the frames time and sequence.

VI. Conclusion

We have presented a quite elaborate problem and an (of course not superior) approach for the solution. Our main aim was to understand the issue and overcome the emerging difficulties. Our model solves the plain colouring tasks (e.g. colouring of the sky) more successfully. The model with *adam, swish and mean-absolute error* is probably the most accurate, leastways in our consideration, however every imparted model has its assets and drawbacks.

References

Vivek Kumar Bagaria (Stanford University) Kedar Tatwawadi (Stanford University) “CS231N Project: Coloring black and white world using Deep Neural Nets” [Online]. Available: http://cs231n.stanford.edu/reports/2016/pdfs/205_Report.pdf

Richard Zhang, Jun-Yan Zhu, Phillip Isola, Xinyang Geng, Angela S. Lin, Tianhe Yu, Alexei A. Efros (University of California, Berkeley) “Real-Time User-Guided Image Colorization with Learned Deep Priors”, 8 May 2017 [Online]. Available: https://arxiv.org/pdf/1705.02999

Richard Zhang, Phillip Isola, Alexei A. Efros “Colorful Image Colorization” v5, 5 Oct 2016 [Online]. Available: https://arxiv.org/abs/1603.08511

Aditya Deshpande, Jiajun Lu, Mao-Chuang Yeh, Min Jin Chong, David Forsyth (University of Illinois at Urbana Champaign) “Learning Diverse Image Colorization” v2, 27 Apr 2017 [Online]. Available: https://arxiv.org/abs/1612.01958

Prajit Ramachandran, Barret Zoph, Quoc V. Le (Google Brain) “Searching for activation functions” v2, 27 Oct 2017 [Online]. Available: https://arxiv.org/abs/1710.05941

Carl Doersch (Carnegie Mellon / UC Berkeley) “Tutorial on Variational Autoencoders” v2, 13 Aug 2016 [Online]. Available: https://arxiv.org/abs/1606.05908

Tung Nguyen, Kazuki Mori, Ruck Thawonmas (College of Information Science and Engineering, Ritsumeikan University)
“Image Colorization Using a Deep Convolutional Neural Network” 27 Apr 2016 [Online]. Available: https://arxiv.org/abs/1604.07904

Diederik P. Kingma (University of Amsterdam, OpenAI), Jimmy Ba (University of Toronto) “Adam: A Method for Stochastic Optimization” v9, 30 Jan 2017 [Online]. Available: https://arxiv.org/abs/1412.6980

Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, Benjamin Recht (University of California, Berkeley,Toyota Technological Institute at Chicago) “The Marginal Value of Adaptive Gradient Methods in Machine Learning” v2, 22 May 2018 [Online]. Available: https://arxiv.org/abs/1705.08292

Soham De, Anirbit Mukherjee, Enayat Ullah (Department of Computer Science, University of Maryland, Department of Applied Mathematics and Statistics, Johns Hopkins University, Department of Computer Science, Johns Hopkins University) “Convergence guarantees for RMSProp and ADAM in non-convex optimization and an empirical comparison to Nesterov acceleration” v3, 20 Nov 2018 [Online]. Available: https://arxiv.org/abs/1807.06766