# Image Colorization with Neural Networks

Weiheng Xia
EURECOM
Antibes, France
weiheng.xia@eurecom.fr

Lamia Salhi
EURECOM
Antibes, France
lamia.salhi@eurecom.fr

## ABSTRACT

The project aims to perform image colorization on gray-scale images, and produce an automatic colorful image. We explored several potential architectures of multilayer perceptron (MLP) and the auto-encoder (AE), while fine tuning their hyper-parameters for the best performance. To represent the metrics of the problem, we used a mean-square error (MSE) function to evaluate our colorization task as a regression problem. We also discuss the problems we encountered with during the training, and how to solve these problems. Finally, we provide a qualitative and quantitative analysis of our experiment results and possible improvements for future research.

**Code Availability:**
https://github.com/LamiaLaBg/Colorization-CIFAR10.

## 1 INTRODUCTION

Recently, there has been an increase interest in image colorization of gray-scale images, using machine learning algorithms. The motivation of image colorization is to add color to old movies, pictures or to assist artists to add colors to their works. It is a powerful tool to restore colors in history and to add colors to the future. The input of our algorithm is a gray scale image, and we use a neural network to predict a colorful image as our final result.

## 2 RELATED WORK

There has been many research works in the field of colorization, and we discover that there are mainly three methods: multi-layer perceptron (MLP), support vector machine (SVM), and convolutional neural network (CNN). As for MLP, [1] used a MLP for auto-colorization of old gray-scale images, but their work is limited in terms of exploring different combinations of training techniques and they didn't discuss the problem of activation functions that could cause problems to the colorization output. As for SVM, [2] used a support vector machine for automatic colorization, but their method requires in-depth knowledge about image processing so we didn't choose SVM as our approach. As for CNN, [3, 4] utilized a convolutional neural network for image colorization and they explored other training details such as drop-out and batch-norm for efficient training. Since our class is dedicated to machine learning, we decide to choose MLP as our main approach for this colorization task.

## 3 DATASET AND FEATURES

This project uses CIFAR-10-python data set which contains images classified by the type of object (dogs, cars, planes, etc.). We obtained our data set from https://www.cs.toronto.edu/~kriz/cifar.html [5]. This data set has been created for classification but here it is used for colorization. It consists of 60000 color images of size 32x32. There are 50000 training images and 10000 test images. The resolution of one image is small, which allows us to do training faster. To simplify the training, we created 3 data sets from it: 5000 images for training, 500 images for validation, and 1000 images for test.

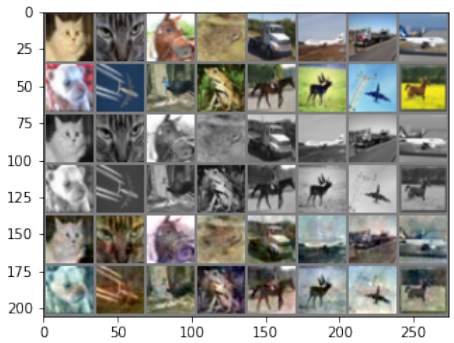



**Figure 1: Inference on test data of our MLP model. The 1-2 rows are groud-truth, the 3-4 rows are gray-scale input, and the 5-6 rows are our colorized outputs.**

In order to train with the dataset, prepocessing is necessary. First, we normalized each image into the range of [-1,1], to make our training easier. Then, we shuffled the data set to make our model independant of the order of the data. Thirdly, we transformed colorful images into gray-scale images as our input. Figure 1 is an example of our original data, processed gray images and output.

## 4 METHODS

Our project uses two architectures to deal with the problem of colorization. Our approaches include the multilayer perceptron method and the auto-encoder with fully connected layers.

### 4.1 MultiLayer Perceptron

Multilayer perceptron (MLP) is a feedforward network of fully connected neurons as shown in the figure Figure 2. It is inspired by the biological neurons in the real world, and it has three main components: an input layer,hidden layers, and an output layer. Each node of the previous layer is connected with each node in the next layer, which is why it's sometimes called fully-connected neural network.

The learning technique of MLP is to first do a forward-propagation of the input through the network, then calculate the loss between

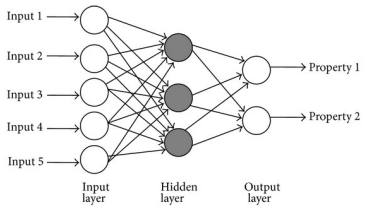



**Figure 2: Multilayer perceptron [6]**

the output and the target. After gaining the loss, it uses backpropagation to compute the gradient of each weight and bias, then update.

A MLP normally uses a non-linear activation function to map the weighted inputs to the outputs of each neuron. If it uses a linear activation function, it essentially becomes a two-layer input-output model.

## 4.2 Auto-encoder

An auto-encoder is a neural network that learns to copy its input to its output. Its particularity is on one internal hidden layer that describes a code used to represent the input. As shown on the figure Figure 3, it is divided in two parts: the encoder and the decoder. The encoder maps the input into the code, while the decoder maps the code to a reconstruction of the original input [7]. We can represent the encoding part by an encoding function h=f(X) and the decoding part as a decoding function Y=g(h). By copying the input to the output, we hope that the latent representation h will make our model learn some useful properties. By constraining h to have smaller dimensions than x, we force the autoencoder to learn the key features of the training data. The autoencoder is then called undercomplete [8]. Autoencoder are unsupervised learning model, because it doesn't require labeled data to train the model.

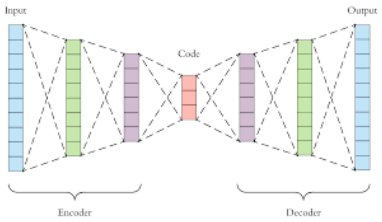



**Figure 3: Auto-encoder scheme [9]**

There are different kinds of auto-encoders the one that we tried to implement is the most basic one in the form of a feedforward through MLP.

# 5 EXPERIMENTS

## 5.1 First Attempt

### *5.1.1 No activation function.*

Our first attempt was to build a MLP using Pytorch framework. Since we were unfamiliar with PyTorch framework, we made a critical mistake in the first attempt of building the MLP network, shown in the code snippet below:

```
self.layer1 = nn.Linear(1024,512)
self.bn1 = nn.BatchNorm1d(512)
self.layer2 = nn.Linear(512,256)
self.bn2 = nn.BatchNorm1d(256)
self.layer3 = nn.Linear(256,3072)
```

**Figure 4: First attempt: MLP implementation in PyTorch**

As you can see, we didn't add any activation functions after the linear layers, so our network is essentially a combination of many linear functions and all the hidden layers can be merged into one hidden layer by linear algebra. However, much to our surprise, we worked in this implementation with many different layers and the result is not so bad, as shown in Figure 5a.

Later after our discussion with our professor, we think it shows a good sign, that the MLP is powerful enough for such colorization tasks, even if we didn't add activation functions. After realizing this, we added activation functions in all our models. As for training details, We chose Mean Square Error as our loss function and used SGD optimizer. Our learning rate is about 0.01, and batch size is 16.

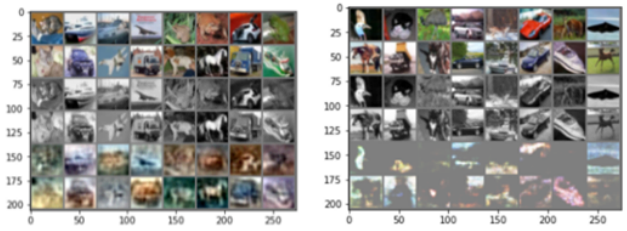
### *5.1.2 Loss function.*

The reason why we chose MSE as our loss function is because we want to represent our task as a regression problem, and we minimize the squared distance between the predicted color value and the ground-truth color value. This MSE loss function is not perfect for our task, since colorization is a multi-modality problem. So the model will tend to choose desaturated colors to avoid large penalty [4]. We decide to continue with MSE in our current work.

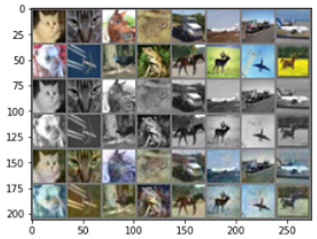
$$Loss := ||y^{color} - y^{truth}||^2 \tag{1}$$

### *5.1.3 Blurry Problems.*

After adding the activation functions, we then encountered with another serious problem in our early attempts. Our images are really blurry and the contrast is very weak, as shown in Figure 5b.



**(a) MLP with no activation** **(b) MLP with ReLU at last**



**(c) MLP with Tanh at last**

**Figure 5: Inference on MLP with different activations**

We then look at each activation function's formula and shape, in order to find a possible reason why this whitened phenomenon happened after we added activation functions.

As shown later in Figure 10, ReLU function will discard all information when its input value is negative, and Sigmoid function will also discard most of negative information. Then we did another 2 experiments with Tanh and no activation as our last layer's activation function, this weakened phenomenon disappear as in Figure 5c. Thus, we came to a conclusion that Sigmoid and ReLU cannot be used as the activation function after the last layer in our application, because we want to keep the negative pixel values as well in our final output.

## 5.2 Models

We made a comparison between MLP and Auto-encoder models to select the best model for further tuning in training details.

*5.2.1 MLP experiments.*

We designed 3 versions of MLP models to test its performance thoroughly. We tried 1 hidden layer, 2 hidden layers, and 3 hidden layers(you may find the code on the github which link is in the abstract). Here we used training parameters shown in Table 3. The one hidden layer architecture is shown on Figure 6.

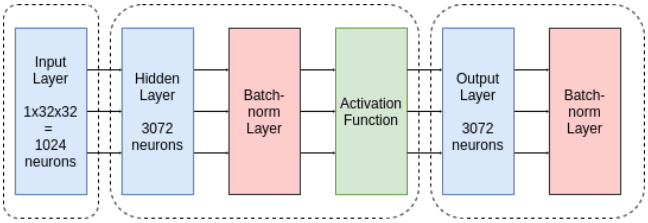



**Figure 6: Our MLP architecture**

**Table 1: MLP results with different layers**

| Number of hidden layer | 1 | 2 | 3 |
|---|---|---|---|
| Train loss | 3.5 % | 3.2 % | 3.7 % |
| Valid loss | 3.4 % | 3.8 % | 4.1 % |
| Test loss | 3.2 % | 3.8 % | 4.1% |

*5.2.2 Autoencoder experiments.*

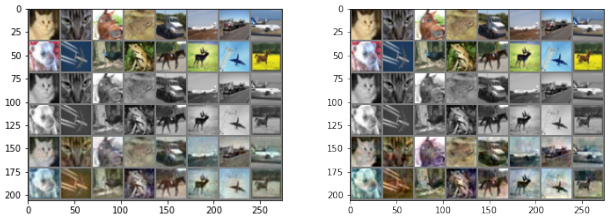
After running several trainings with different number of layers, here is a comparative table which gather all of our tests in the tab Table 2. In order to compare with MLP, we kept all the other training details exactly the same as MLP trainings. For the one hidden layer, we used 1024 neurons in the first layer, 512 for the hidden layer and 3072 for the output layer. As we can see on Table 2 the more hidden layers we add the higher the losses are. In fact one hidden layer is enough to get good results. And the more hidden layer we add the more complex our model is. And it seems that more than one layer is already a too complex model for our data set which is quite simple.

**Table 2: Autoencoder results with different layers**

| number of hidden layer | 1 | 2 | 3 | 5 |
|---|---|---|---|---|
| train loss | 4.2 % | 4.0 % | 5.4 % | 5.7 % |
| valid loss | 3.5 % | 4.1 % | 5.1 % | 5.9 % |
| test loss | 3.408 % | 4.1 % | 5.1% | 5.870% |

*5.2.3 Auto-encoder vs MLP.*

Finally, comparing one layer MLP and one layer Auto-encoder we can say that there are not much differences, which is normal because those two architectures are really similar. To really compare those two methods we would have to work on a more complex data set where pictures contains more pixels (better resolution). Since MLP is slightly better, we choose to continue with MLP for further tuning.



**(a) auto-encoder outputs** **(b) MLP outputs**

**Figure 7: Auto-encoder and MLP outputs**

## 5.3 Overfitting and Batch Normalization

In a neural network, the model is updated layer per layer backward, which means we assume that the weights in the previous layers are fixed. But this isn't the case, as this paper highlights [10], "the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities". This paper names this issue "internal covariate shift" and it can be solved using Batch Normalization. Batch Normalization standardizes the activations of each input variable per mini-batch, to stabilize the training and prevent overfitting [11]:

$$y = \frac{x - E(x)}{\sqrt{V(x) + \epsilon}} * \alpha + \beta \tag{2}$$

$x$ is our input and and $y$ our output. We recognize the classic normalization formula applied on the output of each layer Figure 6. This part is multiply by $\alpha$ (standard deviation) to which we add $\beta$(new mean). Those are learnable parameters which allow the network to learn the distribution of the activations. (By default at the beggining of our training $\epsilon = 1e - 05$)

*5.3.1 Influence of Batch Normalization.*

Let's see now the influence of Batch Normalization on our one hidden layer architecture model with the features of Table 3.

Batch Normalization here overcomes overfitting. As we can see on Table 4 the orange curve which is validation loss is above the

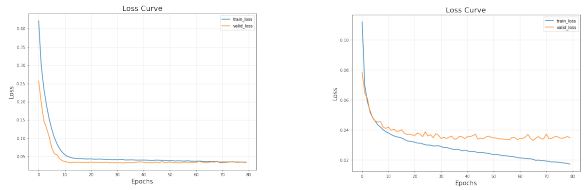
**Table 3: Standard parameters used for experiments**

| Features | Values |
|---|---|
| Network | MLP with one hidden layer (see Figure 6) |
| Loss | MSE |
| Optimizer | Adam |
| Learning rate | 0.0002 |
| Epochs | 80 |
| Batchsize | 16 |

blue curve which is training loss. The validation loss and testing loss are at least 1.5% higher than training loss. By not using Batch Normalization we would have to struggle to find the good initialization parameters.

**Table 4: batchnormalization efficiency**

| | with Batch Normalization | without |
|---|---|---|
| train loss | 3.5 % | 1.7 % |
| valid loos | 3.4 % | 3.5 % |
| test loss | 3.298 % | 3.356 % |




**(a) with batchnormalization** **(b) without regularizer**

**Figure 8: Comparison of loss curves**

*5.3.2 Influence of Batch Normalization after the output layer.*

We then tried to put batch Normalization to understand why we have to put Batch Normalization after the output layer. Here is the curve on Figure 9.

The results without standardization is really unstable and the different losses are higher than with it. Indeed, The distribution of the output layer inputs changes during training, as the parameters of the first layer are changing. Thus, we have to standardize the output (this is "internal covariate shift").

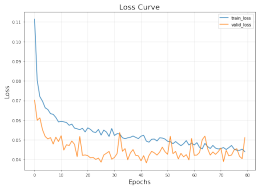



**Figure 9: without regularizer after the last output**

## 5.4 Parameters Tuning

On Table 3 (part 5.3), you can see the model that we chose for the different experiments introduced in this part. In each section of this part only one of those features will change and Batch Normalization is used.

*5.4.1 Activation function of the hidden layer.*

This part highlights the choice of the activation function after the hidden layers. We tried 3 different activation functions with the same other features as in Table 3. Here are the results for each one of them in Table 5:

**Table 5: comparison activation functions**

| | Sigmoid | ReLu | Tanh |
|---|---|---|---|
| train loss | 3.5 % | 2.9% | 3.0 % |
| valid loss | 3.4 % | 3.9% | 3.8 % |
| test loss | 3.298% | 3.899% | 3.745 % |

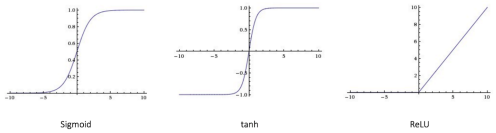



**Figure 10: Activation functions [12]**

Sigmoid seems to give the best results, in fact even if ReLU and Tanh give good results on training loss, it isn't the case for validation loss and testing loss. Indeed Sigmoid function gives lower validation and testing loss and it seems like ReLU and Tanh make the model overfit.

Indeed, ReLU is a pretty simple function which is faster to compute than the sigmoid or the tanh function, and its derivative is faster to compute. For, neural networks this can make a significant difference to training and inference time. In our case it performs too well [13]. Secondly, the model with tanh is also preforming better on training than sigmoid. According to this paper[14], the explanation lies on the fact that tanh function is symmetric about the origin and as our inputs have been normalized on [-1;1] thus tanh is more likely to produce outputs which are the inputs to next layer.

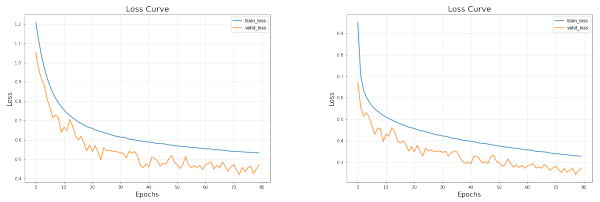
In conclusion, ReLU et Tanh perform better than Sigmoid on training but with regard to the other parameters they would not be a good choice as they lead to overfitting.

*5.4.2 Optimizer.*

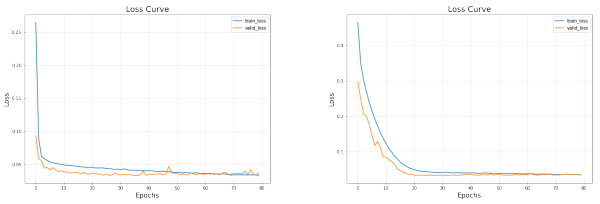
We compared 4 optimizers: SGD, SGD with weight decay, RMSprop, and Adam. Stochastic gradient descent (SGD) is the one of the most popular optimizers in machine learning, which computes each sample's gradient to converge to the minimum value.

However, SGD method may get into local minimum easily and there's a lot of fluctuation near ravines, as you can see in Figure 11a. Therefore, we tried SGD with momentum to try to accelerate

the convergence and dampen the oscillation, as shown in Figure 11b. But the result it not satisfying as well, with a loss around 27%. So we decide to try RMSprop and Adam optimizers, which are self-adaptive learning-rate methods, and their results are shown in Figure 11c and Figure 11d, with their test loss at 3.5% and 3.2%. Here we can see the power of self-adaptive learning rate methods. They can avoid local minimum traps and allow faster convergence, so we finally decide to use Adam as our best tuning optimizer.




**(a) SGD without momentum** **(b) SGD with momentum**




**(c) RMSprop** **(d) Adam**

**Figure 11: Comparison of optimizers**

*5.4.3 Learning Rate.*

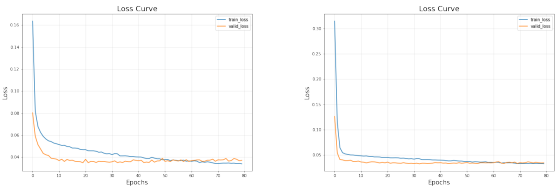
Learning rate is an important parameter for quick convergence and to find the global minimum. If a learning rate is too large, it can result in fast convergence into local minimum and unstable training process. If a learning rate is too small, it may result in very slow training process.

To find the best learning rate, we did 3 experiments with learning rate 0.01, 0.001, and 0.0001. In order to compare their results, we keep the rest of the training details the same as shown in Table 3. The results in Figure 12 are similar, and the main difference is the convergence speed. After comparison, we decide to choose 0.001 as our our best learning rate.

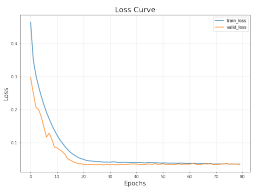
*5.4.4 Validation loss < Training loss.*

As you can see on most of our graphs it seems that the validation curves are often under the training loss curve. Which means that the validation set had better results than the training set. We struggled during a long time to figure out why this happens. This can be explained by 3 reasons (more detailed on source [15]):

- Training loss is measured during each epoch, when validation loss is measured after each epoch. The training loss is continually reported over the course of an entire epoch, however, validation features are computed over the validation set only once the current training epoch is completed. Thus, to see the correct graph we have to shift the curve for in average $\frac{1}{2}$ epoch.




**(a) LR=0.01** **(b) LR=0.001**




**(c) LR=0.0001**

**Figure 12: Comparison of learning rate**

- Secondly we thought our validation set was simpler or not large enough. After testing different features, we found it wasn't the issue.
- Thirdly we think it might be because we didn't consider the fact that the regularizer is applied only on the training set. In fact in the training loss we consider the regularizer loss which is not the case of the validation loss because it is not back-propagated.

# 6 CONCLUSION AND FUTURE WORK

## 6.1 Conclusion

In conclusion, after struggling a lot on different varieties of issues we succeeded in implementing a pretty satisfying model for image colorization. Our best model is the one hidden layer MLP with Batch Normalization, which performs slightly better than the auto-encoder model. Autoencoder may be more helpful for encoding problems. We used MSE loss to represent our task as a regression problem, but it is not perfect in the colorization task, since it will tend to use a desaturated color to avoid large penalty. To overcome "internal covariate shift" we used Batch Normalization as for the activation function, Sigmoid seems to be the best choice for us to prevent the loss of negative information. We found SGD and SGD with momentum couldn't really solve our regression task of colorization, while RMSprop and Adam can adapt better to our problem.

## 6.2 Future Work

If we had more time, we would try a more complex data set, with higher resolution and complicated patterns. We would also like to work more on Auto-encoder to discover its power in encoding. We also want to try deep neural networks like CNN and cGAN, which can extract neighboring information to recognize the patterns of outlines. Another challenge for further research is the loss function. As colorization is a multi-modal task, it's hard to evaluate its performance and avoid the desaturation of colors.

## 7 CONTRIBUTIONS

We both worked equally on our project or at least we tried to. At the beginning of the project we split in two. Weiheng implemented the project using Pytorch and Lamia tried implementing our own MLP using Lab2 code of the eurecom MALIS course (see on github). But then as our data set contains a lot of data it was complicate to continue running our own implementation on CPU. Thus we decided to focus both on the Pytorch implemented model which we can be run on Nvidia GPU. In a second part, after struggling a lot on some issues we decided to split the different tests to run: Weiheng trained the MLP model, and he explored the influence of different kinds of optimizers, loss functions and learning rates, and he also trained an auto-encoder with convolutional layers on anime dataset for fun. Lamia implemented an auto-encoder model, and she explored the impacts of different activation functions and she studied the effect of Batch Normalization.

## ACKNOWLEDGMENTS

We thank Ms. Zuluaga teacher or machine learning at EURECOM for her guidance and help.

## REFERENCES

[1] Junkyo Suh, K. N. Nazif, and A. Kumar. Old image denoising and auto-colorization using linear regression , multilayer perceptron and convolutional nearual network.

[2] Guillaume Charpiat, Ilja Bezrukov, Matthias Hofmann, Yasemin Altun, Bernhard Schölkopf, and R. Lukac. Machine learning methods for automatic image colorization. *Computational Photography: Methods and Applications, 395-418 (2010),* 01 2011.

[3] M. Agrawal and Kartik Sawhney. Exploring convolutional neural networks for automatic image colorization.

[4] Richard Zhang, Phillip Isola, and Alexei A. Efros. Colorful image colorization, 2016.

[5] Alex Krizhevsky. Cifar-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html, October 2009.

[6] Jelena Djuris, Djordje Medarević, Marko Krstić, Ivana Vasiljević, Ivana Aleksić, and Svetlana Ibrić. Design space approach in optimization of fluid bed granulation and tablets compression process. *TheScientificWorldJournal*, 2012:185085, 07 2012.

[7] Wikipedia. "auto-encodeur". https://fr.wikipedia.org/wiki/Auto-encodeur, January 2021.

[8] Nathan Hubens. "deep inside: Autoencoders", towards data science. https://towardsdatascience.com/deep-inside-autoencoders-7e41f319999f, February 2018.

[9] Arden Dertat. Applied deep learning - part 3: Autoencoders. https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798, October 2017.

[10] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[11] Pytorch. Batchnorm1d. https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html.

[12] IBM. Deep learning. https://developer.ibm.com/recipes/tutorials/deep-learning/, January 2020.

[13] StackExchange. "what are the advantages of relu over sigmoid function in deep neural networks?". https://stats.stackexchange.com/questions/126238/what-are-the-advantages-of-relu-over-sigmoid-function-in-deep-neural-networks#:~:text=Efficiency, December 2012.

[14] Chamanth mvs. "activation functions : Why “tanh” outperforms “logistic sigmoid”?". https://medium.com/analytics-vidhya/activation-functions-why-tanh-outperforms-logistic-sigmoid-3f26469ac0d1, December 2019.

[15] Adrian Rosebrock. " why is my validation loss lower than my training loss?". https://www.pyimagesearch.com/2019/10/14/why-is-my-validation-loss-lower-than-my-training-loss/.