LIP-STUDENTS-20-07

Study of jet quenching phenomenon using deep neural networks

Filipe Cunha[1,a]

[1] *Instituto Superior Técnico, Lisboa, Portugal*

Project supervisor: N. Castro, M. Romão

October 2020

Abstract. In this work several convolutional neural network architectures were trained on simulated calorimeter images to discriminate between quenched and non-quenched jets. Despite not achieving the best performance previously realized, the trained models still managed to work quite well, and prove that machine learning applications on high-energy physics are quite interesting and should be more carefully studied in the near future.

Keywords: Jet Quenching, CNNs, Monte Carlo

1 Introduction to Machine Learning in the study of jets

Quarks and gluons are *colour-charged*, a fundamental property related to their strong interactions in quantum chromodynamics (the term color is loosely related to the primary colours, with a particle with colour red, for instance, having an antiparticle with the anticolour of red). The colour confinement principle states that colour-charged particles cannot be isolated - meaning that quarks and gluons must form hadrons, and therefore only colourless states are allowed naturally.

As such, whenever a colour-charged object fragments, each resulting piece will also be colour-charged. Due to the previously mentioned colour confinement, these fragments must create other colour-charged pieces in order to form colourless objects. The formed objects tend to create a narrow cone, or *jet*, since they will all move in the same direction. So, jets are essentially collimated bunches of hadrons derived from the fragmentation of energetic quarks and gluons.

In heavy ion collisions, a dense medium, called the quark-gluon plasma (QGP) is created. Jets can interact with this medium, reducing their energy - called a *jet quenching* phenomenon. As such, it is possible to study the properties of quark-gluon plasma comparing jet quenching phenomenon with the unquenched case. This is due to the fact that the magnitude of the lost energy depends on the properties of the QGP, amongst others.

Several groups around the world have highlighted the power of machine learning approaches to study jet-related phenomenon, taking as inputs mainly images recovered from calorimeters.

The aim of this study was to analyse how convolutional neural networks could distinguish between quenched and not-quenched jets.

2 Deep Neural Networks

2.1 Basic Architecture of Neural Networks

The simplest possible neural network is known as the perceptron, a basic network that just contains a single input layer and an output node - each training instance is of the form ($\bar{X}$,y), where each $\bar{X} = [x_1, \ldots, x_n]$ contains n feature variables and each y $\in$ {-1, +1} represents the class of the variable.

The input layer contains n nodes that transmits the n features weighted by a weight vector $\bar{W} = [w_1, \ldots, w_n]$ to the output node, with the predicted value $\hat{y}$ given by:

$$\hat{y} = sgn\{\bar{X} * \bar{W}\} = sgn\{\sum_{i=1}^{d} x_i * w_i\}$$

Multilayer neural networks contain more than one computational layer - besides the input and output layers they also have additional intermediate layers referred to as hidden layers, since their outputs are not visible. The architecture of multilayer networks assumes that all nodes in one layer are connected to those in the next layer.

Therefore, when forward propagating inputs in a multilayer perceptrons, there are three main computations (network with a total of d layers):

$$\bar{h}_1 = \sigma(W_1^T \bar{x})$$

[Input to first hidden layer]

$$\bar{h}_{k+1} = \sigma(W_{k+1}^T \bar{h}_p) \quad \forall p \in \{1, \ldots, d-1\}$$

[Hidden to hidden layer]

$$\bar{o} = \sigma(W_{k+1}^T \bar{h}_d)$$

[Hidden to Output layer]

where the σ function, called the *activation function* is a function that is applied element-wise to its' vector arguments, with its most common example being the *tanh*, *relu*, *elu* or *selu* functions. The relu function will be explained later on, with the elu and selu functions being alterations of these.

In order to train a neural network, the main objective would be to minimize the error (also called *loss*) function at the output layer. This is done using the *backpropagation* [2] algorithm, which uses the chain rule of differential calculus to compute the error gradients in terms of sums of local (in the nodes) gradient products over the various paths from a node to the output. Despite sounding inefficient, this algorithm can be computed easily using dynamic programming.

[a] e-mail: filipe.miranda.cunha@tecnico.ulisboa.pt

2.2 Overfitting

Despite being universal function approximators, some challenges are still present in neural network training, the most important of them being overfitting. This occurs when a network performs well on training data - meaning that it accurately predicts the output classes - but makes a large amount of mispredictions on unseen data.

In order to stop a model from overfiiting, several strategies can be applied, the most common being regularization, dropout and early stopping.

2.2.1 L^p Regularization

In layman terms, regularization constrains a model to use fewer non-zero parameters. This is usually done by adding a penalty $\lambda||\bar{W}||^p$ (λ being a new hyperparameter) to the loss function. The most common regularization technique used sets $p = 2$, called the *Tikhonov regularization*

2.2.2 Early Stopping

The early stopping technique makes the gradient descent end after a few iterations. In order to decide the stoppage point, it is common to set aside a part of the training data and test the error on this set (called the validation set), and stops the algorithm if the error of the validation set stops decreasing, thus preventing the model from overfitting.

2.2.3 Dropout

The last commonly used techinique to prevent overfitting, dropout makes the model ignore stochastically chosen units (nodes) during the training phase. As such, during each epoch, nodes in the network are dropped out with a certain probability p or frozen (not considered during forward or backpropagation) with probability *1-p*. This prevents the units from becoming co-dependent in one another, thus preventing overfitting.

2.3 Gradient Descent Optimization

Gradient descent is still the preferred way to optimize neural networks. This objective of this algorithm is to minimize the loss function ($J(\theta), \theta \in \mathbb{R}^d$), by updating the parameters in the direction opposing its gradient ($\nabla_\theta J(\theta)$), with a parameter called the *learning rate* (η) determining the size of the steps.

2.3.1 Momentum

The gradient descent algorithm usually has trouble navigating areas where the surface curves more steeply in one dimension than another (called *ravines*). Momentum [4] accelerates the algorithm by 'directioning' it to the relevant direction, doing this by adding a fraction (γ) of the update vector of the past time step to the current update vector:

$$\upsilon_t = \gamma\upsilon_{t-1} + \eta\nabla_\theta J(\theta)$$
$$\theta = \theta - \upsilon_t$$

2.3.2 Nesterov accelerated gradient (NAG)

Momentum alone does not satisfy gradient descent's problems around ravines, since it keeps accelerating the algorithm even when it is nearing a minimum, often getting stuck at local minimum. NAG [5] solves this by adding the momentum term ($-\gamma\upsilon_{t-1}$) to the parameters:

$$\upsilon_t = \gamma\upsilon_{t-1} + \eta\nabla_\theta J(\theta - \gamma\upsilon_{t-1})$$
$$\theta = \theta - \upsilon_t$$

2.3.3 Adagrad

Adagrad [6] is an an algorithm for gradient-descent based optimization that performs smaller updates (with lower leerning rates) for features that occur frequently across classes, and larger updates for infrequent features, making it well-suited for dealing with sparse data.

Let g_t be the gradient at time step t, with $g_{ti} = \Delta_\theta J(\theta_{ti})$. The adagrad learning rate at step t will be given by:

$$\eta_t = \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}},$$

With $G_t \in \mathbb{R}^{d*d}$ being a diagonal matrix where each nonzero element (i, i) is the sum of the gradients with respect to θ_i up to time t. As such, the general learning rule will be given by:

$$\theta_{t+1,i} = \theta_t + \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \circ g_t$$

The main issue with this algorithm is that it can make the learning rate shrink until it becomes infinitesimally small, making the network unable to acquire any new knowledge.

2.3.4 Nadam

Nadam [8] combines the Adam algorithm [7] with the NAG. The Adam algorithm is centered on the momemtum update rule, given by:

$$g_t = \nabla_{\theta_t} J(\theta_t)$$
$$m_t = \gamma_{t-1} + \eta g_t$$
$$\theta_{t+1} = \theta_t - m_t$$

Where m_t is a new term, called the *first moment* - corresponding to the mean. Nadam changes these equations by combining the momentum term with NAG, obtaining:

$$g_t = \nabla_{\theta_t} J(\theta_t)$$
$$m_t = \gamma_{t-1} + \eta g_t$$
$$\theta_{t+1} = \theta_t - (\gamma m_{t-1} + \eta g_t)$$

The plain Adam update rule is given by:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}$$

Expanding the second equation:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}(\frac{\beta_1 m_{t-1}}{1-\beta_1^t} + \frac{(1-\beta_1)g_t}{1-\beta_1^t})$$

And replacing $\frac{m_{t-1}}{1-\beta_1^t}$ with $\hat{m}_t$, one obtains the final Nadam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}(\beta_1 \hat{m}_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t})$$

2.4 Introduction to CNNs

Convolutional neural networks excel in the analysis of grid-structured inputs with strong spatial dependencies in local regions. As such, they are the best kinds of neural networks to perform image analysis, since adjacent spatial locations in an image are often related to one another.

Each layer in a convolutional network is, as such, a three-dimensional grid structure composed of width, height and depth, referring to the number of channels in the layer (in this particular case, the depth would be two, representing the order of magnitude of the energy).

The main important characteristic of a CNN is the convolution operation, a dot product operation between a grid-structured set of weights and similar grid-structured inputs drawn from different spatial localities in the input volume [1]. In order to explain this operation, consider the input of the qth layer is of size $L_q \times B_q \times d_q$ (in the current case, for the input layer, $L_q = 33$, $B_q = 33$, $d_q = 2$).

The parameters to be considered for the convolution operation are three-dimensional square sets known as filters, with an arbitrary width but with the property that the depth of the filter is always the same as the depth of the layer on which it is applied.

The convolution operation will then place the filter at each possible position in the image, performing a dot product operation between the $F_q \times F_q \times d_q$ parameters of the filter and the corresponding grid on the input. As such, the number of filters in the model will control the capacity of the model, since they directly control the number of model parameters. The operation itself is represented in Fig 1:

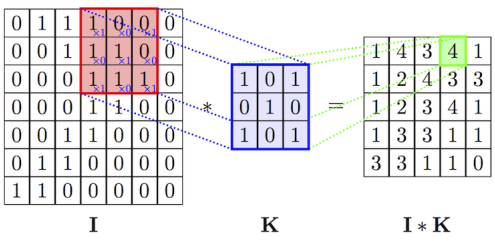

Figure 1. Convolution Operation [9]

For a more rigorous definition, suppose that the pth filter in the qth layer has parameters $W^{p,q} = [w_{ijk}^{(p,q)}]$ and that the feature maps in this same layer are represented by the tensor $H^q = [w_{ijk}^{(q)}]$. The convolution operation to the $(q+1)$th layer will be defined as:

$$w_{ijk}^{(q)} = \sum_{r=1}^{F_q}\sum_{s=1}^{F_q}\sum_{k=1}^{d_q} w_{rsk}^{(p,q)} h_{i+r-1,j+s-1,k}^{(q)}$$

$$\forall i \in \{1,\ldots,L_q - F_q + 1\}$$

$$\forall j \in \{1,\ldots,B_q - F_q + 1\}$$

$$\forall k \in \{1,\ldots,d_q + 1\}$$

It can be observed that the convolution operation reduces the size of the $(q+1)$th layer when comparing with the size of the qth layer, which can lead to information loss. In order to avoid this, padding can be used - adding $(F_q - 1)/2$ pixels around the borders of the image. It is relevant that these pixels don't add information that may lead to an incorrect diagnosis, and as such the most commonly used type of padding is the 'same-padding': just adding zeros.

Another commonly used operation in CNNs is the pooling layer, which works on small grid regions and outputs another layer with the same depth but different height and width. This layer essencialy applies an arbitrary operation to the previously mentioned grid region, increasing the size of the receptive field while reducing the spatial footprint of the layer (assuming that use strides used are greater than one). The most commonly used types of pooling are max and average pooling, and the latter is represented in Fig 2:

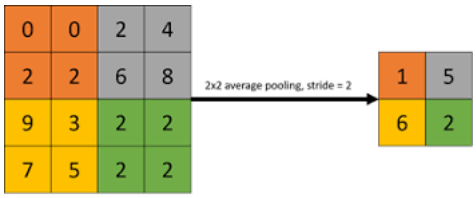

Figure 2. Average Pooling Operation [10]

The final commonly used layer is the ReLU layer. The rectified linear activation function or ReLU for short is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. This layer usually follows a convolutional layer in CNNs.

2.5 CNN Architectures

2.5.1 VGG

In 2014, the Visual Geometry Group (VGG) [11] research lab at Oxford University further emphasized a developing trend in that time: increased depth in networks = better results. As such, several networks with 11 to 19 layers were developed, and the top-performers were the ones with more than 16 layers.

The main important innovation VGG gave was that reduced filters required increased depth, meaning that a smaller filters with an bigger number of layers generally outperformed networks with larger filters per layer but with smaller depth. This is due to the fact that more depth allows for more non-linearity (due to the presence of more ReLU layers), and, as such, allows the network to 'discover' more common patterns amongst the images.

Bearing this in mind, the VGG network was designed with a repeated pattern of 2/3 convolutional layers with stride 1, 3×3 filters and a pooling-size of 2×2. In essence, allowed the network to reduce its spacial footprint by a factor of 2 whenever its depth was increased by a factor of 2.

The main issue with this architecture is that increased depth led to more instability in the network.

2.5.2 GoogLeNet

In spite of the good performance of the VGG, the best architecture developed in 2014 was the GoogLeNet [12], that proposed the novel concept of an inception network (a network inside a network). The initial part of this architecture is a regular CNN (known as the *stem*), with the original part being the intermediate layer (the *inception module*).

This idea was based on the fact that key information in images is available at different levels of detail, so different networks should capture this information better than a single one. This flexibility allows for different images to be represented at different levels of granularity, leading to a much better network than the previous state of the art (winner of the 2014 ILSVRC competition). A simple inception module in Fig 3:

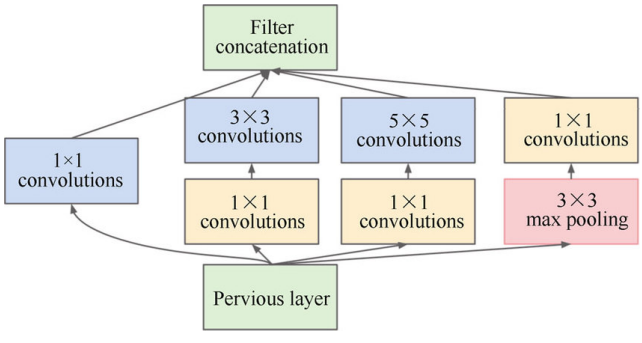

Figure 3. Simple Inception Module [12]

The main issue with this architecture is that it results in some computational inefficiency, due to the large number of convolutions with different sizes. This can be solved by adding 1×1 convolutions to first reduce the depth of the feature map (eg: 128 1×1 convolutions can be used to reduce a depth of 256 to 128). This approach is known as a bottleneck operation.

The initial GoogLeNet was made of 9 such inception modules.

2.5.3 ResNet

In 2015, the ResNet [13] model was introduced with 152 layers, almost an order of magnitude more than the previous architectures. This large number of layers was possible due to the development of skip connections (or *residual layers*), allowing the input of a certain layer q to be 'copied' directly to a layer $q+r$, without suffering any transformation in the r layers. In Fig 4, a simple residual layer is shown, where the input (x) is allowed to skip two weight layers.

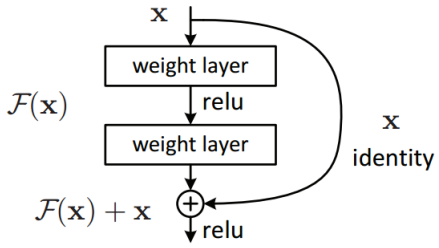

Figure 4. Basic Skip Connection [13]

These residual layers provide paths of unimpeded gradient flow and so allow the learning algorithm to choose the appropriate level of non-linearity required for the given particular classification task. As such, skip connections permit the development of very deep networks that are computationally efficient to train and that do not necessarily cause problems during backpropagation.

3 Results

Bearing the above concepts in mind, the conducted study focused on the deployment of deep CNNs architectures to distinguish between quenched and non-quenched jets in simulated calorimeter images. The diagrams where generated using stochastic Monte-Carlo techniques, and were the inputs of the developed networks: corresponding to $(33 * 33 * 2)$ sparse arrays.

The sparsity of the arrays makes this a difficult problem, since it is extremely hard to distinguish an array representing a quenched jet from an array representing a non-quenched one.

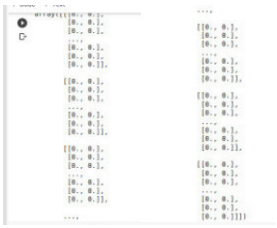

Figure 5. Image example

The studied architectures were implemented using the Keras package [14] on top of a Tensorflow backend [15] and were optimized using Optuna [16]. The required data

preprocessing was made using the sparse functionalities of Scipy [16] and the postprocessing required the Pandas [17] and Scikit-Learn [18] libraries.

To continuously improve the models, the area under the ROC (receiver operation characteristics) curve was chosen as the performance metric. In order to make a fair comparison between the chosen architectures, a maximum number of 100 epochs was fixed, making use of EarlyStopping after 10 consecutive with no improvements in the area under the ROC curve. The same loss function was also chosen for all models - binary cross-entropy - since the problem at hand is a binary classification problem. Normalized physical weights were also chosen as training, validation and test sample weigths.

This section will be divided in two subsections: in the first, small convolutional networks were designed, following basic architectures, and afterwards the complex contemporary architectures explained in Section 2 were used.

3.1 Results of basic CNNs

The first constructed networks were basic, having a depth of 3 layers with a variable number of filters in each (32 and 64) and kernel size (3 and 5). The Nadam optimizer was used.

	Train AUC	Val Loss	Val AUC
filters: 32 kernel_size: 3 lr: 0.155407	0.5010	84.3672	0.5001
filters: 64 kernel_size: 3 lr: 0.0612185	0.5007	42.8984	0.500
filters: 64 kernel_size: 5 lr: 0.064864	0.4988	4625.5	0.4993
filters: 64 kernel_size: 3 lr: 0.0112074	0.5500	3.2453	0.6112
filters: 64 kernel_size: 3 lr: 0.029714	0.5012	14.0709	0.5003

Table 1. Base Network Results

It is possible to observe that the obtained results were far from optimal - the simple networks created did not learn the dataset well, with the best option being using more filters with a smaller kernel size, as one should expect.

In order to study the effect of pooling, the next designed network was essentially the same as before, with added *Batch Normalization* layers between the convolutional layers and with an extra *MaxPooling* layer behind the classifier:

	Train AUC	Val Loss	Val AUC
filters: 32 kernel_size: 3 lr: 0.155407	0.5010	84.3672	0.5001
filters: 64 kernel_size: 3 lr: 0.0612185	0.5007	42.8984	0.500
filters: 64 kernel_size: 5 lr: 0.064864	0.4988	4625.5	0.4993
filters: 64 kernel_size: 3 lr: 0.0112074	0.5500	3.2453	0.6112
filters: 64 kernel_size: 3 lr: 0.029714	0.5012	14.0709	0.5003

Table 2. Base Network Results with Pooling

As a final 'small' model, a network known as the AlexNet (best model designed in 2012) was trained on the dataset. This network consists of four consecutive convolutional layers with an increasing number of filters followed by a pooling layer and a classifier at the end. The 'pure' network was not used, with a tweaked model being preferred.

	Train AUC	Val Loss	Val AUC
activation: relu strides: 2 lr: 0.264609	0.4986	22.2533	0.5000
activation: selu strides: 2 lr: 0.077206	0.5014	4.6076	0.5000
activation: relu strides: 2 lr: 3.31145	0.8249	0.8246	0.6331
activation: selu strides: 2 lr: 0.010146	0.5011	1.3742	0.5000

Table 3. Tweaked AlexNet

It is possible to see that the best instance of the model overfit massively, despite being equipped with regularization and having dropout layers, mainly due to its large number of parameters. Nonetheless, it managed to outperform the previously designed netwotks by quite a margin. As such, a more thorough analysis of this architecture was conducted.

If it were possible to prevent these models from overfitting as much, an area under the ROC curve of approximately 0.9 could be achieved. Unfortunately, in this particular study all instances of this architecture overfitted massively, and, as such, new architectures needed to be experimented with, being the focus of the next sections.

4 Contemporary models

With these results in mind, the next section will focus on the optimization of the contemporary architectures explained in section 2 and their applications to the problem at hand.

The first experimented model was the VGG, and no noticeable rise in accuracy was observed. As the table below demonstrates, several depths were studied, with a notorious following increase in the number of parameters (for instance, a depth of 21 corresponded to a total number of 53261057 trainable parameters.

Using padding did not improve the results and, as such, in the following networks padding will be replaced mainly by batch normalization layers.

Depth	Train AUC	Val Loss	Val AUC
10	0.6531	0.7563	0.6321
16	0.7125	0.6875	0.6799
21	0.7041	0.6385	0.6888

Table 4. VGG Models

Despite not overfitting, the VGG model performed worse than the previously analysed AlexNet model on the training set, a suprising result considering that the VGG is a much more complex architecture, with many more parameters than the previous one.

Using skip layers proved to be a good strategy, leading to a noticeable increase in the auc score. As before, several depths were experimented with, working with the resnetv1 and resnetv2 models.

Depth	Train AUC	Val Loss	Val AUC
32 (v1)	0.7147	0.6356	0.6984
38 (v1)	0.7051	0.6414	0.6986
70 (v1)	0.7456	0.6128	0.7123
56 (v2)	0.7796	0.71656	0.6820
110 (v2)	0.8346	0.7056	0.6983

Table 5. ResNet Models

Unlike the previous ones, tweaking the ResNet architectures lead to an AUC of over 0.7, with only certain instances overfitting. It is also possible to conclude that the usage of regularization did not make any considerable changes in performance, while massively increasing the temporal complexion of the model.

Like using skip layers, using tweaked inception modules with a variable number of modules as well as towers per module also proved to be an inspired idea. The network did not overfit, with some instances achieving more than 0.7 in the AUC metric.

Bearing this in mind, the last experimented architecture used a combination of skip layers and inception modules - the XCeption architecture. Surprisingly, the models designed using this architecture performed more poorly than their previous counterparts.

	Train AUC	Val Loss	Val AUC
1 module 3 cols	0.7518	0.6891	0.6544
1 module 4 cols	0.6692	0.6587	0.6742
2 modules 3 cols	0.7231	0.6252	0.7156
2 modules 4 cols	0.7274	0.6035	0.6952

Table 6. Inception Models

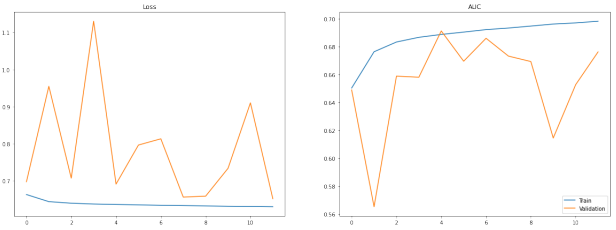

Figure 6. Xception results

It is clear that the model overfitted, despite being equipped with l^r regularization and dropout.

5 Conclusions

This study once again proves the advantages of using machine learning techniques in high-energy physics, particularly in the field of image analysis. Despite not reaching state of the art, the designed architectures achieved a relatively low false negative rate, highlighting their usefulness in this study area. It should also be noted that, due to the low capacity of the GPU used in the study as well as the limited usage of the same, more complex version of the explained contemporary architectures as well as current state of the art models, such as squeeze-and-excitation networks [19] could not be tested in the given time frame. For future studies, it should be interesting to not only test even more advanced contemporary architectures, as well as adding more depth to classical ones. Using GANs [20] instead of Monte Carlo techniques to simulate the data should also decrease the computational complexity and could add more physical relevance to the data.

References

[1] C. Aggarwall, “Neural Networks and Deep Learning,” Springer, 2018, ISBN 978-3-319-94463-0

[2] Maximilian Alber, Irwan Bello, Barret Zoph, Pieter-Jan Kindermans, Prajit Ramachandran, ”Backprop Evolution,” ICML 2018 AutoML Workshop

[3] Geoffrey Hinton et all. “ Dropout: A Simple Way to Prevent Neural Networks from Overfitting,”Journal of Machine Learning Research 15 2014,pp. 1929-1958

[4] Ilya Sutskever, James Martens, George Dahl, Geoffrey Hinton, "On the importance of initialization and momentum in deep learning," Proceedings of the

30th International Conference on Machine Learning, PMLR 28(3):1139-1147, 2013

[5] Alesandar Botev, Guy Lever, David Barber, "Nesterov's Accelerated Gradient and Momentum as approximations to Regularised Update Descent," Anchorage, AK, USA, International Joint Conference on Neural Networks, 2017

[6] John Duchi, Elad Hazan, Yoram Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," 2011 Journal of Machine Learning Research, 2011, pp. 2121-2159

[7] Diederik Kingma, Jimmy Ba, "Adam: A Method for Stochastic Optimization", 3rd International Conference for Learning Representations, San Diego, 2015

[8] Timothy Dozat, "Incorporating Nesterov Momentum Into Adam,", ICLR, 2016

[9] Ihab S. Mohamed, "Detection and Tracking of Pallets using a Laser Rangefinder and Machine Learning Techniques," 2017

[10] Albert Liu et al, "Embarc - A machine learning inference library", 2019

[11] Karen Simonyan, Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," 2014

[12] C. Szegedy et al., "Going deeper with convolutions," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, 2015, pp. 1-9.

[13] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 770-778.

[14] F. Chollet, "Keras," https://keras.io, 2015

[15] M. Abadi, "TensorFlow: Large-scale machine learning on heterogeneous systems," tensorflow.org, 2015

[16] T. Oliphant, P. Peterson, E. Jones, "Scipy, " ,2001–.

[17] W. McKinney, "Data structures for statistical computing in python," Python inScience Conference, volume 445, pages 51–56. Austin, 2010

[18] F. Pedregosa, "Scikit-learn: Machine learning in-Python, " Journal of Machine Learning Research, volume 12, pages 2825–2830, 2011.

[19] J. Hu, L. Shen, S. Albanie, G. Sun, E. Wu, "Squeeze-and-Excitation Networks, " CVPR, 2018

[20] Ian J. Goodfellow et all, "Generative Adversarial Networks, " International Conference on Neural Information Processing Systems (NIPS 2014), pp. 2672–2680