Project: Effect of Batch Normalization and Dropout Layers in Deep Learning Networks

ADEIKALAM Pierre
CHEN Guangyue
XU Kevin

February 2020

1 Introduction

Batch Normalization by (Ioffe and Szegedy, 2015)[1] and Dropout by (Srivastava et al, 2014)[2] have become standard techniques used daily by practitioners for the regularisation of deep networks. As with any regularisation technique, it is very important to understand the effect they have in order to use them correctly, as their improper use can lead to sub-optimal performance at best and poor performance at worst.

The aim of this project is to clearly and cleanly explain how these two techniques work, what problem they aim to solve and how to use them effectively. To this end, this report has been structured as follows:

1. Mechanism of Batch Normalization and its Effects
2. Mechanism of Dropout and its Effects
3. Illustrative Experiments

2 Mechanisms of Batch Normalization and its Effects

2.1 Motivation and Mechanism

Normalization is the traditional technique of setting the empirical mean and variance of a dataset to be 0 and 1. Formally, if we let $x_1, ..., x_n$ be a dataset of n observations, μ be its mean and σ^2 be its variance, then normalizing this dataset corresponds to the operation:

$$\forall i \in \{1, ..., n\} \qquad \hat{x}_i \leftarrow \frac{x_i - \mu}{\sigma} \tag{1}$$

For linear models such as SVM and Linear Regression, normalization of the input dataset is known to speed up and regularize training as it puts all the features at the same scale.

In the context of deep learning, the output of one layer becomes the input of the next one. Due to the non-linearity of the activation functions, we can see how the normalization of the dataset can vanish as the output of a layer no longer has the same distribution as its input. This means that each layer trains on different distributions that do not have mean 0 and variance 1. If we could keep the normalization effect as the inputs go through the layers, then training could maybe be sped up just like in the linear models.

However, gradient descent in deep learning is often done via Mini-Batch Stochastic Gradient Descent, meaning we do not compute the full gradient over the whole dataset but instead compute it over a *mini-batch* of samples of size $m < n$. In practice this is done with the following two operations:

- Forward Operation: Evaluate the loss of the network on a mini-batch of uniformly sampled observations.
- Backward Operation: Perform gradient descent on the parameters of the network with the gradient of the loss obtained on this mini-batch.

Therefore, we cannot directly normalize the outputs of a layer like we do when first normalizing the dataset. This means that the normalization should happen in between the layers during the Forward operation, hence the name Batch Normalization.

To perform this operation, the authors propose the following algorithm.

Algorithm 1 Batch Normalization Forward Operation

Input: Batch of inputs $(x_1, ..., x_n)$
Output: Batch of normalized inputs $(\hat{x}_1, ..., \hat{x}_n)$

procedure BatchNormalization$(x_1, ..., x_n)$
  $\hat{\mu} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i$ ▷ Mini-batch empirical mean
  $\hat{\sigma}^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \hat{\mu})^2$ ▷ Mini-batch empirical variance
  $\forall i \in \{1, ..., m\}, \quad \hat{x}_i \leftarrow \gamma \frac{x_i - \hat{\mu}}{\hat{\sigma}} + \beta$ ▷ Normalization of the inputs

The main difference one can notice with the standard normalization operation is the presence of the scaling constant γ and shift constant β. These constants are actually trainable parameters that give more flexibility to the model and allow the Batch Normalization operation to become the identity function if it becomes optimal while training (if $\gamma \approx \hat{\sigma}$ and $\beta \approx \hat{\mu}$, then $\hat{x}_i \approx x_i$). Much like the other parameters of the network, γ and β are trained via gradient descent.

2.2 Effects of Batch Normalization

As we will see in our experiments, the Batch Normalization technique accomplishes its main goal which is to speed up training of deep networks. However, it comes with other beneficial effects that make this operation an actual regularization technique.

In order to build an intuition as to how Batch Normalization enables us to train a deep network faster, we can look at the simple case of a "dead" linear unit (also known as "neuron").

One of the most widely used activation function is the ReLU function given by $\text{ReLU}(x) = x1_{x>0}$. This function is of practical interest because its gradient is constant and it avoids the problem of *vanishing gradient*.

The main problem of this activation function is that, if due to unfortunate initialization the output of a linear unit is almost always negative, then the activation of this neuron will almost always be 0. Since the gradient of the ReLU function at $x < 0$ is 0, if the neuron does not activate, it does not update. Thus, this neuron will never update its weights during training and always output 0.

If one were to normalize the outputs of this neuron **before** the activation, then necessarily this neuron will be updated **every** iteration of the Backward Operation, thus "ressurecting" this neuron regardless of its initialization.

While this effect is beneficial for the ReLU activation, it can be detrimental when using the *hyperbolic tangent* activation function given by $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$. As we can see from Figure 1, when x is close to 0, $\tanh(x) \approx x$, meaning that Batch Normalization could potentially remove the non-linearity effect of the tanh activation function.

A similar result can be said for the *sigmoid* activation function given by $S(X) = \frac{1}{1+e^{-x}}$ as it can be linearly approximated by $\frac{1}{4}x + \frac{1}{2}$ near 0, which is also a linear function.

As these activation functions are often used for binary classification, Batch Normalization **should not** be used before the last layer of a network.

Another beneficial effect of Batch Normalization is that at each epoch the normalization will be computed with different parameters since the samples in the mini-batch will change. This adds noise to the input and works like a data augmentation technique, thus regularizing the network.

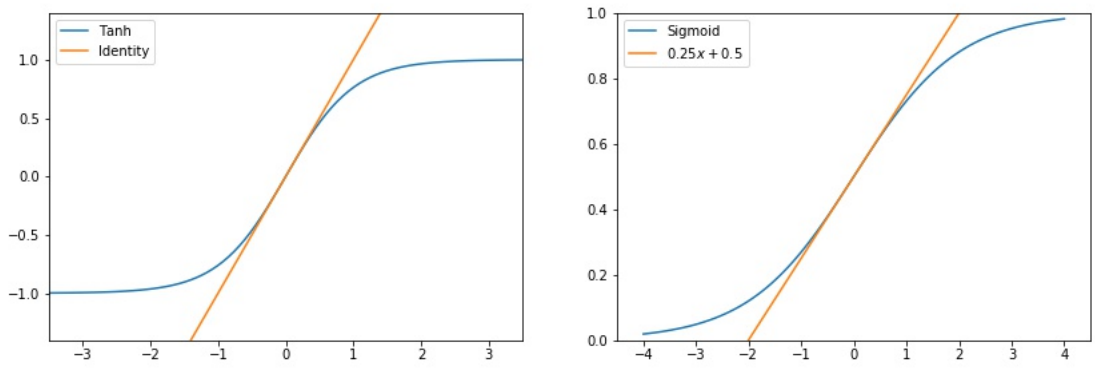

Figure 1: Comparison of tanh and sigmoid with the identity function.

3 Mechanism of Dropout and its Effects

3.1 Motivation and Mechanism

A typical machine learning regularisation technique is to make Ensemble models that combine a multitude of *weak* regressors into a strong, more powerful regressor. This approach often helps combat overfitting, especially for models that have a very large capacity, such as neural networks or deep decision trees. A typical example of Ensemble models is the Random Forest model that constructs a large amount of Decision Trees, each with a random selection of the features, and averages their individual predictions to make the final prediction.

The idea behind the Dropout operation is similar. Intuitively, if we imagine a neuron being a regressor and the outputs of the previous layer being the features, implementing this idea would consist in using only *part* of the outputs to train this neuron.

Since making many neural networks with a random subset of features would be very expensive, the Dropout operation achieves this by randomly setting some of the outputs of a layer to 0 before passing it to the next layer during training. This effectively removes neurons from the network for a brief amount of time, allowing other neurons to fill in for the missing signals.

Another way to see this is that at each iteration of training, we sample a thinner version of the network and train it for a single step. As the authors put it, training a neural network of n neurons with dropout can be seen as training a collection of 2^n thinned networks with extensive weight sharing, where each thinned network gets trained very rarely, if at all.

At test time, the averaging of the predictions is done by re-scaling the weights of every neuron. If the probability of keeping the outputs of a specific layer is given by $p > 0$, then the weights of this layer are multiplied p in order to "average" the outputs of the previous layer. However, since each Dropout layer can have a different retention probability p, each Dropout operation potentially adds a new dimension of fine-tuning to the network.

3.2 Effects of Dropout

The main benefit of Dropout is that it prevents *co-adaptation* of the features. What is co-adaptation exactly?

As a network is trained, what often happens is that the neurons of a layer do not uniformly contribute to the prediction. Indeed, high weights are associated to neurons that are often active while smaller weights are given to the others. Each time a neuron is active, it has to fight for relevancy by shifting down the weights associated with the inactive neurons. The network then stabilizes around a local minimum where only a small amount of neurons are effectively trained, meaning that a large part of the network is useless and the quality of the predictions does not generalize at test time. This is co-adaptation.

An ideal scenario would be that each individual neuron learns to detect a powerful feature such that taking a random sample of these neurons would be enough to make a proper prediction. This is what Dropout aims to do. By shutting down some neurons at each iteration, the network is forced to learn to train each neuron so that it detects *independent* signals that do not have to fight for relevancy.

However, it is often unclear how many neurons should be dropped out to maximize the effectiveness of each individual neuron, so many tests are needed to find the optimal dropout probability of a layer.

4 Illustrative Experiments

In this section, we illustrate the effects of Batch Normalization and Dropout with training experiments on the CIFAR-10[1] classification dataset. It provides 50000 32x32-pixel training and 10000 test images, classified into 10 categories. All the experiments were carried out on Google Colab. For the three architectures, we use a batch-size of 32 and a learning rate of 10^{-3} with the Adam Optimizer. For the first two architectures, we train for 20 epochs. For the third architecture, we train for 100 epochs and decay the learning rate by a factor of 10 after 80 epochs.

[1] https://www.cs.toronto.edu/ kriz/cifar.html

4.1 Batch Normalization experiments

In order to measure the effect of Batch Normalization on the speed of training, we test 3 different neural network architectures and watch the speed at which the objective loss function decreases over time during training. For each architecture, we created two versions of it where one of them had a Batch Normalization layer between every two consecutive layers and one without. A third version was added and it has a Batch Normalization layer **after** every activation operation. This was done to show that the Batch Normalization operation should be applied **before** the activation operation. Indeed, we have noticed that it is very common for users of the Keras library to make the mistake of including the Batch Normalization Layer after the activation and we believe this is due to the Keras API that facilitates this overlook.

4.1.1 Experiment 1: A Very Simple CNN Model

The first architecture we tested was a simple Convolution Neural Network with 4 Convolution Layers, 1 Dense Layer and 1 Soft-max Layer with a Max-Pooling after every other Convolution Layer. Its architecture is represented in Figure 2.

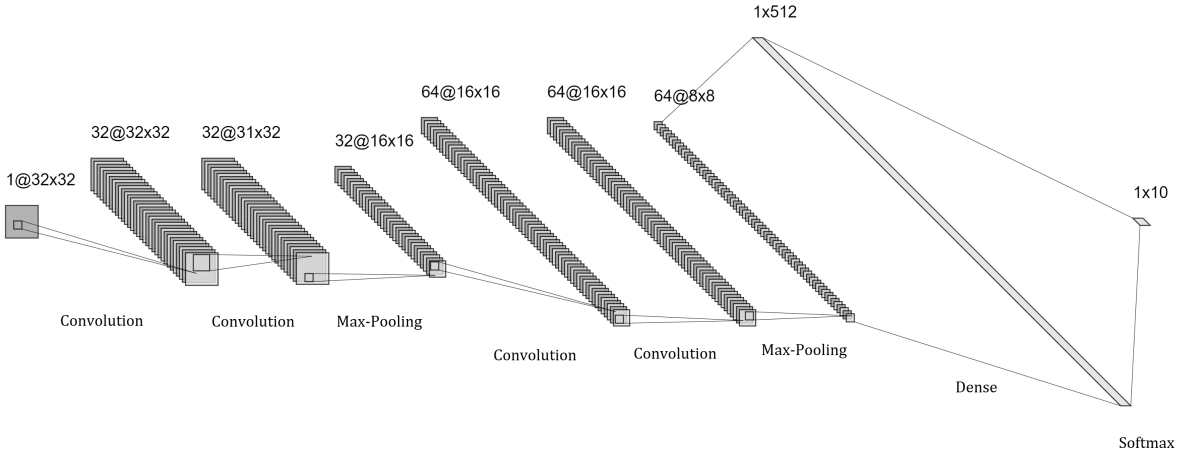

Figure 2: Architecture of the first model used for testing Batch Normalization.

The results of training are given in Figure 3. The benefits of Batch Normalization are immediately noticeable: The model without Batch Normalization needs 20 epochs to reach a Training Loss of 0.2 while the model that incorporates Batch Normalization before every activation achieves this in 6 epochs. The Validation Accuracy is also a lot higher for the models with Batch Normalization by a margin of more than 10%. Even though Batch Normalization is beneficial even when performed after the activation, we see that it is sub-optimal.

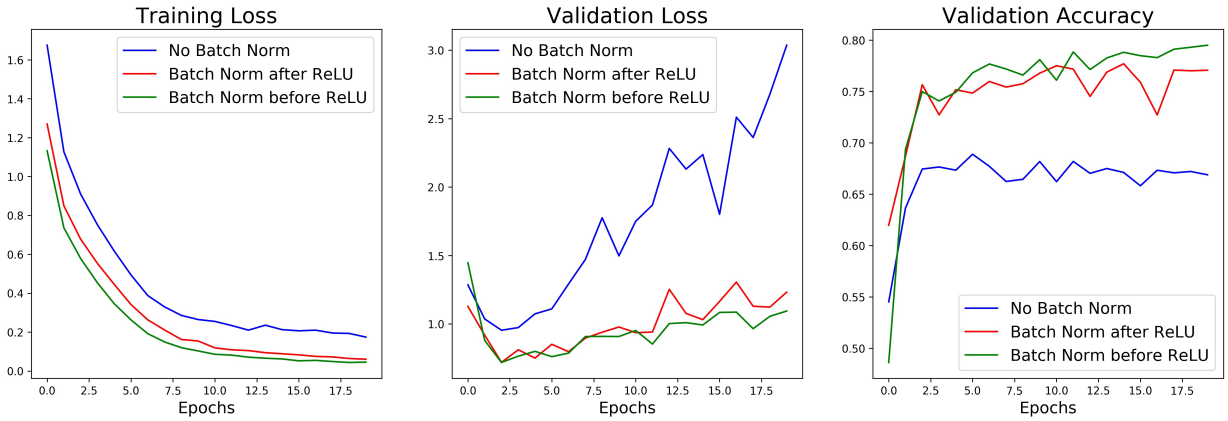

Figure 3: Training results for the first tested architecture.

4.1.2 Experiment 2: A Deeper CNN Model

We use the same architecture as in the previous section but twice as deep. We have now 8 Convolution Layers, 2 Dense Layers and 1 Soft-max Layer. The results are shown in Figure 4.

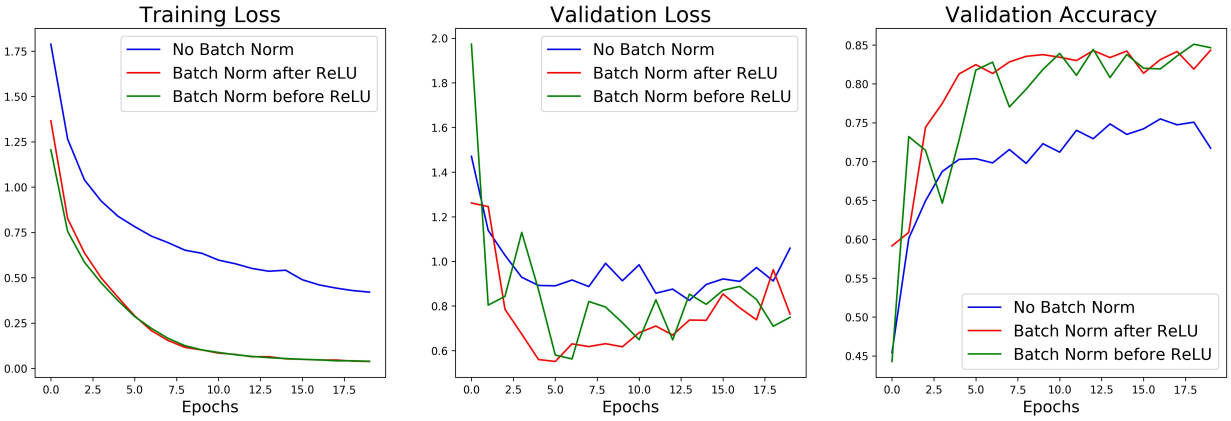

Figure 4: Training results for the second tested architecture.

The gap in performance between the models that incorporate Batch Normalization and the model that does not grows even larger. The non-regularized model achieves a loss of about 0.5 in 20 epochs while the regularized ones achieve this in 3 epochs. The Validation Accuracy of the non-regularized model has improved from the previous experiment but the regularized models still show a double-digit increase in accuracy.

4.1.3 Experiment 3: A Residual Network Model

We now make the same experiment but on the ResNet20 architecture. Residual networks are extremely deep neural networks that have been notorious for taking a very large amount of epochs to properly train. For this reason, training at optimal speed is of practical interest. Each network has been trained with the Adam Optimizer and a learning rate that would decrease by a factor of 10 after 80 iterations. The results of this experiment can be found in Figure 5.

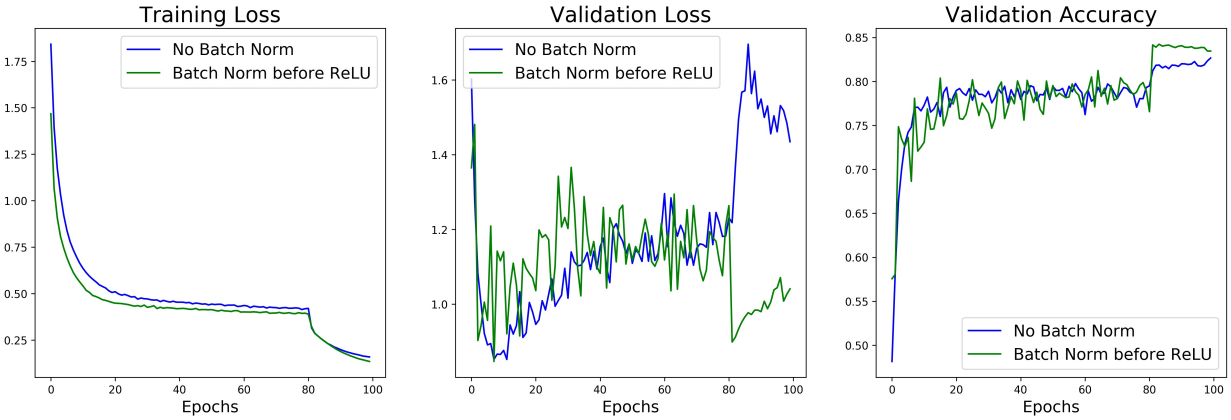

Figure 5: Training results for the ResNet20 architecture.

While the results are not as dramatic as in the previous experiment, we see that the model that incorporates Batch Normalization before every activation maintains a solid lead over the model that does not. However, when performing the training on Google Colab, the regularized model would take on average 26 seconds to perform back-propagation each epoch while the model without would take only 18 seconds on average. This represents a 44% increase in computation time required to perform Batch Normalization before every activation for the ResNet20 architecture and this should keep increasing for even deeper architectures like ResNet164 or ResNet1001. This cost should be taken into account when evaluating the *practical* training speed of an architecture.

4.2 Dropout experiments

As in the previous section, we carry out some experiments in order to observe the performance of Dropout operation. We test here 2 different neural network architectures similar the ones used previously.

4.2.1 Experiment 1: A Very Simple CNN Model

The first architecture is the one represented in Figure 2. We compare here a model following this architecture with Dropout on each of the layers and a

baseline model. Usually, the probability of dropping a unit $1 - p$ is set between 0.2 and 0.5. It can be tuned by performing a grid/random search cross-validation for optimal results. For our experiment, we have used the same probability $p = 0.8$ for all the layers as our objective is to see the concrete effect of Dropout.

The results obtained are given by the following Figure 6

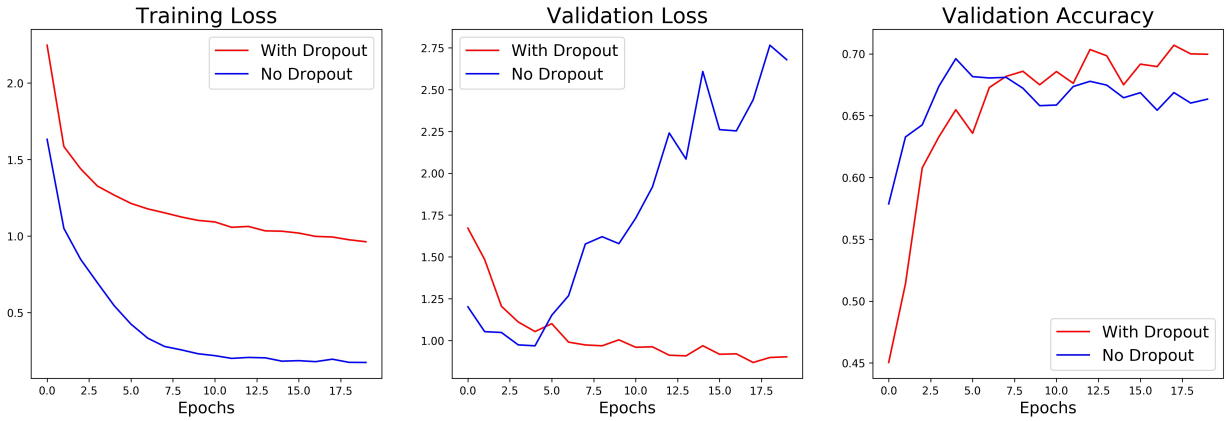

Figure 6: Training results for the first architecture with Dropout

We clearly notice by monitoring the validation loss that adding Dropout regularizes the network on the validation set. Even though the training loss of the model with Dropout is higher, we observe that the network with Dropout keeps the loss on the validation data small, while the loss with the baseline model begins to increase rapidly after only a few epochs. Even the validation accuracy still remains correct while the accuracy of the model without Dropout starts to decrease.

4.2.2 Experiment 2: A Deeper CNN Model

To see the effects of Dropout with $p = 0.8$ on a bigger model, we have applied the method on the layers of the same architecture as described in section 4.1.2. With this deeper model, we obtain the results shown in Figure 7

We observe here that the results are not satisfactory at all as the model with Dropout achieves a worse validation performance than the model without. This means that Dropout cannot be blindly applied like Batch Normalization and its application requires tuning. Indeed, the model with Dropout shows signs of underfitting as both its validation accuracy and training loss are worse.

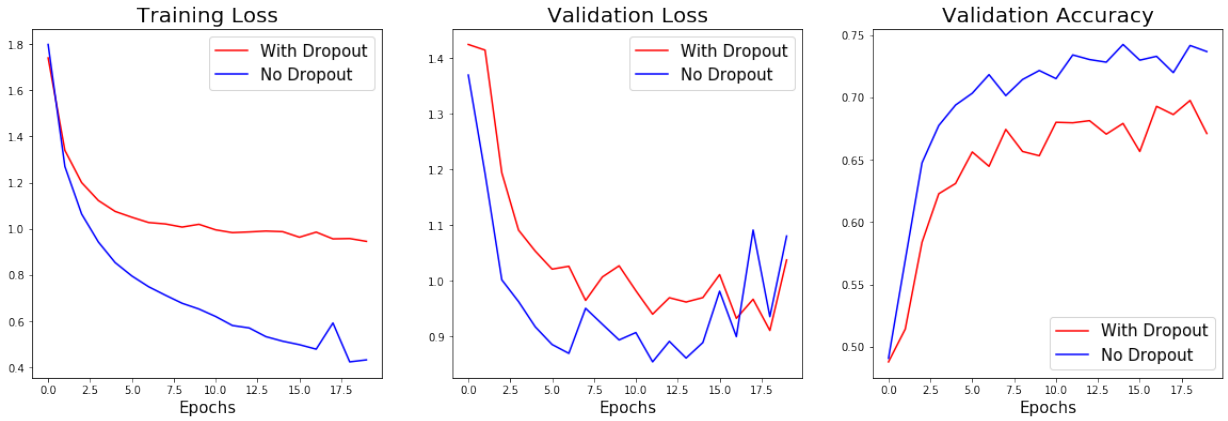

Figure 7: Training results for the second architecture with Dropout

5 Conclusion

In conclusion, Batch Normalization and Dropout both reach their objectives.

Batch Normalization is able to significantly speed the training of networks while also regularizing them. The experiments we have performed suggest that it could be potentially applied universally without worrying of potential downsides and with next to no tuning.

Dropout on the other hand has shown that its regularisation effect is very powerful but its application requires careful tuning to reach an optimal performance. Dropout can also be problematic for deep networks as each dropout operation adds a dimension of hyperparameter tuning. Our experiments have shown that we cannot blindly apply Dropout in every layer as the network can begin to underfit the data.

References

[1] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[2] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.