Multimedia Tools and Applications (2020) 79:12777–12815
https://doi.org/10.1007/s11042-019-08453-9

Dropout *vs.* batch normalization: an empirical study of their impact to deep learning

Christian Garbin[1] · Xingquan Zhu[1] · Oge Marques[1]

Received: 26 April 2019 / Revised: 3 September 2019 / Accepted: 7 November 2019 /
Published online: 22 January 2020

Abstract

Overfitting and long training time are two fundamental challenges in multilayered neural network learning and deep learning in particular. Dropout and batch normalization are two well-recognized approaches to tackle these challenges. While both approaches share overlapping design principles, numerous research results have shown that they have unique strengths to improve deep learning. Many tools simplify these two approaches as a simple function call, allowing flexible stacking to form deep learning architectures. Although their usage guidelines are available, unfortunately no well-defined set of rules or comprehensive studies to investigate them concerning data input, network configurations, learning efficiency, and accuracy. It is not clear when users should consider using dropout and/or batch normalization, and how they should be combined (or used alternatively) to achieve optimized deep learning outcomes. In this paper we conduct an empirical study to investigate the effect of dropout and batch normalization on training deep learning models. We use multilayered dense neural networks and convolutional neural networks (CNN) as the deep learning models, and mix dropout and batch normalization to design different architectures and subsequently observe their performance in terms of training and test CPU time, number of parameters in the model (as a proxy for model size), and classification accuracy. The interplay between network structures, dropout, and batch normalization, allow us to conclude when and how dropout and batch normalization should be considered in deep learning. The empirical study quantified the increase in training time when dropout and batch normalization are used, as well as the increase in prediction time (important for constrained environments, such as smartphones and low-powered IoT devices). It showed that a non-adaptive optimizer (e.g. SGD) can outperform adaptive optimizers, but only at the cost of a significant amount of training times to perform hyperparameter tuning, while an adaptive optimizer (e.g. RMSProp) performs well without much tuning. Finally, it showed that dropout and batch normalization should be used in CNNs only with caution and experimentation (when in doubt and short on time to experiment, use only batch normalization).

✉ Xingquan Zhu
xzhu3@fau.edu

Extended author information available on the last page of the article.

Keywords Deep learning · Overfitting · Dropout · Batch normalization · Optimization · Regularization

1 Introduction

Machine learning is a "generalization" process which learns mathematical models from sample data (i.e. training data) to make accurate predictions on previously unseen data (i.e. test data), without using explicitly programmed rules [5]. Many methods exist to carry out learning on the training data, using supervised, unsupervised, or semi-supervised learning paradigms [5] [34]. For all these learning algorithms, two commonly used measures to evaluate their performance are learning efficiency, i.e. model training speed, and prediction accuracy. The former determines the scalability of the algorithms and defines how well they can be applied to large-scale data, and the latter determines the utility of the model and its usefulness for real-world usages.

When carrying out learning on the training data, conventional machine learning algorithms, such as nearest neighbor classification [31], decision trees [19], multilayer perceptrons [26], and support vector machines [3], require each training instance (or sample) to be represented using some feature values, and the whole training data is provided in instance-feature tabular format. Recently, deep learning approaches, such as convolutional neural networks (CNN) [16], and recurrent neural networks (RNN) [18], are emerging as a new machine learning paradigms that have shown tremendous performance gain compared to conventional machine learning algorithms, especially for applications such as voice recognition [4], image classification [13], video action recognition [32, 33], and spatio-temporal data applications [14]. The power of deep learning stems from its feature representation capability, which is able to repetitively learn new features from training data. This is fundamentally better than conventional machine learning methods, because their feature learning capabilities automatically extract features from the training data, leading to a better separation of the training data and subsequently improved performance at test time.

For all deep learning algorithms, the repetitive parameter training, carried out across different layers, is often a time-consuming process. As many stacked deep learning network designs, such as stacked Boltzmann machines or stacked autoencoders, are becoming increasingly popular, it is easy to have deep learning architectures with billions of parameters which not only make the training time consuming, but are also vulnerable to overfitting. As a result, many methods/approaches, such as layer-wise freeze training [2], dropout [28], and batch normalization [7], have been proposed to ensure steady and efficient training of reliable deep learning models.

1.1 Overfitting and dropout

Overfitting is a common challenge in training machine learning models [5]. In general, overfitting happens when the model performs well on the training data, but performs poorly on test data, i.e. the model has low training error and high test error. *Regularization* is a set of techniques used to reduce overfitting [5]. Some of these techniques are model-specific, such as preprunning and postprunning for decision trees, some techniques act on the gradient descent optimization algorithms [25], and others act on the input data, artificially creating new training data [24].

Other techniques go beyond acting on only one model. One such technique is *model ensembling*, combining the output of several models, each trained differently in some respect, to generate one final answer. It has dominated recent machine learning competitions [5]. Although model ensembling performs well, it requires a much larger training time by definition (compared to training only one model). Each model in the ensemble has to be trained either from scratch or derived from a base model in significant ways.

When tackling overfitting for deep learning, *dropout* [28] proposed to randomly change the network architecture, to minimize the risks that the learned weight values are highly customized to the underlying training data, and therefore cannot be generalized well to test data. In essence, dropout simulates model ensembling without creating multiple networks. It has been widely adopted since its publication, in part because it does not require fundamental changes to the network architecture, other than adding the dropout layers.

Despite its simplicity, dropout still requires tuning of hyperparameters to work well in different applications. The original paper [28] mentions the need to change the learning rate, weight decay, momentum, max-norm, number of units in a layer, among others. Getting dropout to work well for a given network architecture and input data requires experimentation with these hyperparameters. Adding dropout to a network increases the convergence time [28]. Then, after adding dropout, we need to train models with different combinations of hyperparameters that affect its behavior, further increasing training time.

Another equally important, if not more significant, complicating factor is that existing dropout evaluations were tested with the standard stochastic descent gradient (SGD) optimizer (as it is done in most papers [25]). Most networks today use adaptive optimizers, e.g. RMSProp [30], commonly used in Keras examples. Some of the recommendations in the dropout paper [28], for example, learning rates and weight decay values, do not necessarily apply when an adaptive optimizer is used.

This observation naturally leads to techniques focusing on improving the model training efficiency by helping the models converge faster. One such technique commonly used for deep learning is *batch normalization* [7].

1.2 Training efficiency and batch normalization

Before *batch normalization* [7] was introduced, the time to train a network to converge depended significantly on careful initialization of hyperparameters (e.g. initial weight values) and on the use of small learning rates, which lengthened the training time. The learning process was further complicated by the dependency of one layer on its preceding layers. Small changes in one layer could be amplified when flowing through the other network layers. Batch normalization significantly reduces training time by normalizing the input of each layer in the network, not only the input layer. This approach allows the use of higher learning rates, which in turn reduces the number of training steps the network need to converge ([7] reported 14 times fewer steps in some cases).

Similar to dropout, using batch normalization is simple: add batch normalization layers in the network. Because of this simplicity, using batch normalization would be a natural candidate to be used to speed up training of different combinations of hyperparameters needed to optimize the use of dropout layers (it would not speed up each epoch during training, but would help converge faster).

However, batch normalization also has a regularization effect that renders dropout unnecessary in some cases, as documented in the original paper [7], in [20], and [9].

1.3 Dropout and batch normalization combination

With the overlapping and sometimes contradicting recommendations for dropout and batch normalization usage, choosing the best architecture for a network, one that can be trained in a short amount of time and generalizes well, now becomes a four-fold question:

1. How do dropout and batch normalization behave concerning different types of deep learning architectures?
2. Should it use dropout or batch normalization? Both claim to regularize the network, but do they regularize equally well, and at the same cost of training time and network size?
3. Should it use both dropout and batch normalization? Despite the claim in [7], other experiments showed that they can be used together to improve a network [17].
4. Should it use any of them? Could an adaptive optimizer (e.g. RMSProp) be enough to quickly converge to an acceptable accuracy for some problem spaces and input data?

Indeed, if dropout should be encouraged for some (if not all) deep learning architectures, there are two more questions to answer:

– What values should be used for the hyperparameters that affect dropout (learning rate, weight decay, momentum, optimizer, etc.)?
– What are reasonable dropout parameter settings to ensure performance gain of the underlying models?

Motivated by the above observations, in this paper we conduct empirical studies to derive some guidelines for using dropout and batch normalization to train deep learning models. The experiments are performed in image classifications tasks (MNIST and CIFAR-10) using multilayer perceptron networks (MLP) and convolutional neural networks (CNN). We carry out controlled experiments to test networks without dropout or batch normalization to create a baseline, followed by networks only with dropout, only with batch normalization and with both. Each network was further tested with a combination of hyperparameters. The hyperparameters selected for the tests are the ones mentioned in the dropout paper [28] and the batch normalization paper [7].

Overall, our experiments draw valuable findings to answer the above questions.

For MLPs, our results show that:

– Dropout and batch normalization significantly increase training time.
– However, batch normalization converges faster. If used together with early stopping it may reduce overall training time; without early stopping it will increase overall training time by a large margin.
– Batch normalization also resulted in higher test (prediction) times. This may be an important factor for applications in restricted environments, such as mobile devices and low-powered IoT (Internet of Things) edge devices.

For CNNs, our results demonstrate that:

– Using batch normalization improves accuracy with only a small penalty for training time. Therefore, it should be the first technique used to improve CNNs.
– Using dropout, on the other hand, reduces accuracy in our tests. Other papers (e.g. [17]) reported that dropout helps accuracy, but not in all cases. The conclusion is to apply dropout carefully, not assuming that it will always improve the results.

Finally, we found out that a non-adaptive optimizer (such as SGD), frequently used in papers [25], performs better than an adaptive optimizer (such as RMSProp), but only if hyperparameters are carefully tuned. Therefore, papers that publish results using a non-adaptive optimizer should document the value of all hyperparameters to be reproducible, and practitioners, in general, should use an adaptive optimizer to save the large amount of training time required to fine-tune a non-adaptive one. They should use that time to focus on more consequential decisions instead (e.g. the network architecture).

We also made the framework used for the experiments available in GitHub,[1] to expose all details of the tests to other researchers, and to make the experiments reproducible for confirmation or enhancements for future investigations.

The remainder of the paper documents the findings in details and is structured as follows. Section 2 reviews related works on hyperparameters configuration for neural networks, dropout, batch normalization and works that investigate both of them together. Section 3 describes the most important design parameters that affect the experiments performed in this paper, starting with the recommendations from the original dropout and batch normalization works. Section 4 describes the datasets, network architectures, hardware and software configurations used in the experiments. Section 5 reports the results of the experiments, documents conclusions and recommendations derived from these experiments, and suggests topics for future investigations. Section 6 reviews the goals of the work and its results.

2 Related work

2.1 Parameter settings for deep learning

Due to the complex nature of the underlying deep learning architecture and the large number of parameters involved in the training, finding proper parameter settings has always been a practical challenge for the deep learning community. In order to optimize gradient-based training of deep architectures, Bengio [1] introduced a practical guide to optimizing hyperparameters. It emphasizes the need to choose a good learning rate as the main decision when optimizing networks.

Because the majority of deep learning architectures rely on gradient descent principle for optimization and a variety of approaches have been proposed, a literature review [25] summarizes the different optimizers and documents guidelines to choose one. The survey also recommends the use of adaptive learning-rate optimizers for sparse data, and names RMSProp, Adadelta and Adam as good choices, with Adam slightly outperforming RMSProp towards the end of the optimization. It also notes that many recent papers use a simple SGD optimizer (no momentum and only simple learning rate annealing schedules), even though an adaptive optimizer could have been a better choice. This points to the need for more investigations of results published in papers using SGD as the optimizer.

Despite the recommendation to start with an adaptive learning-rate optimizer from [25], a well-tuned SGD optimizer may outperform an adaptive optimizer in some applications. However, this can only be determined by trying out network configurations with the SGD optimizer. Smith [27] makes several recommendations to help speed up the training phase.

[1] https://github.com/fau-masters-collected-works-cgarbin/cap6619-deep-learning-term-project

Specifically for SGD optimizers, it makes recommendations for learning rate, momentum, and weight decay, all major contributors to the convergence time. It recommends the use of cycling learning rates with cyclical momentum and a quick grid search to determine the weight decay, in all cases closely watching the test and validation losses of the network to prune inefficient combinations of values early on in the training cycle.

Even with guidelines to choose the initial values for hyperparameters [1] and guidelines to tune parameters during training [27], verifying that those initial values and modifications during training are indeed helping the network converge faster takes time. In [6] the authors propose to start the training phase with a lower dimension version of the input data. For example, when training with images, first start with images resized to a smaller resolution, then increase the resolution in later phases, once it has been confirmed that the hyperparameter settings are helping the network to converge. The results prove that this method reduces the time to verify that the network is converging (or not). Since it does not overlap with other hyperparameter tuning strategies, it can be used as a complementary approach to those strategies.

2.2 Dropout and batch normalization

Because deep learning architectures often have a large number of weight values for tuning, whereas the training process only has a limited number of samples, overfitting becomes a significant challenge. On the other hand, existing approaches to avoid overfitting, such as decision tree pruning or constrained optimization, are either too specific or too expensive, therefore cannot be applied to general deep learning frameworks. Finding simple and effective approaches to avoid overfitting for deep learning is a practical challenge.

To prevent neural networks from overfitting, Srivastava et al. [28] introduced dropout and described how specific hyperparameters (learning rate, momentum, max-norm, etc.) affect its behavior and provided detailed recommendations on how to train networks using dropout. Most of the recommendations are given in ranges of values for each hyperparameter. For example increase learning rate by 10 to 100 times, use momentum between 0.95 and 0.99, apply max-norm with values from 3 to 4, etc. The dropout rate itself is recommended as a range between 0.5 and 0.8 (for hidden layers). Mixing this number of hyperparameters and their ranges result in a large matrix of combinations to try during training.

To accelerate the deep network training, Ioffe and Szegedy [7] introduced batch normalization. It shows that batch normalization enables higher learning rates by reducing internal covariate shift, but does not prescribe a value or a range to be used. It also recommends to reduce L2 weight regularization and to accelerate learning rate decay. Finally, it recommends removing dropout altogether and count on the regularization effect provided by batch normalization. This claim has been studied in more recent articles (some of them are referenced below). These additional investigations resulted in recommendations to use dropout together with batch normalization in some scenarios.

Noticing the success of the dropout and batch normalization for deep learning, [17] reconciles dropout and batch normalization for some applications and shows that combining them reduces the error rate in those applications. Its specific recommendation is to apply dropout after batch normalization, with a small dropout rate.

In order to better understand regularization in batch normalization, [20] shows that using dropout after batch normalization layers is beneficial if the batch size is large (256 samples or more) and a small (0.125) dropout rate is used (similar to the findings in [17] in this respect). It also hypothesizes that dropout did not work in [7] because it was tested with a small batch size.

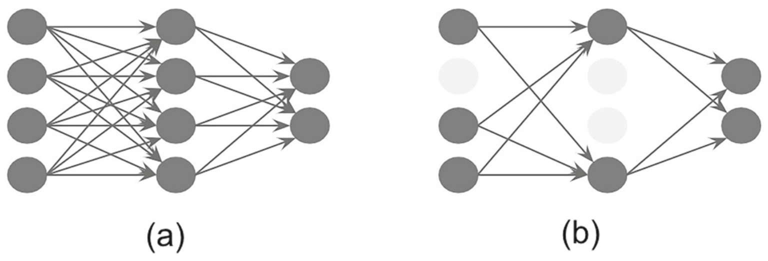


Fig. 1 A conceptual view of dropout. Instead of using a fixed network structure, dropout randomly removes units from a fully connected network (left) to create a sub-model (right)

In summary, while there is empirical research of dropout and batch normalization, with some practical recommendations for settings for each of them, our research work reported in this paper differs from existing works in two main aspects:

1. Our work not only investigates the effect of the dropout and batch normalization layers, but also studies how do they behave with respect to different optimizers: the SGD optimizer, commonly used in publications, and an adaptive optimizer (RMSProp), commonly used in commercial applications.
2. Our research investigates the efficiency and effectiveness of the networks using dropout and batch normalization combined training. The networks are measured in terms of the time to train the networks, the time a trained network takes to predict values, and the memory consumption of the network. These factors are important for real-life applications, where the best possible accuracy is not the only deciding factor to adopt a deep neural network architecture.

3 Empirical study framework and designs

3.1 Dropout

Dropout [28] is a technique to reduce overfitting. Its central idea is to take a model that is overfitting and train sub-models derived from it by randomly removing units for each training batch. A conceptual view of the standard dense network *vs.* the network structure after applying dropout is shown in Fig. 1.

By repeatedly eliminating random units, dropout forces the units to be more robust, learning feature on their own, without depending on other units. In this context it can be thought of as simplified model ensembling.

The number of units to retain is controlled by a new hyperparameter, the *dropout rate*.[2] The recommended values for the dropout rate are 0.1 for the input layer and between 0.5 and 0.8 for internal layers.

Using dropout requires some adjustments to the hyperparameters. The more significant changes are:

[2]Note that the Keras API uses this parameter to control the number of units to *remove* (the opposite meaning of what is used in the dropout paper). This paper follows the Keras API, i.e. units to remove.

- **Increase the network size:** dropout removes units during training, reducing the capacity of the network. To compensate for that the number of units has to be adjusted by the dropout rate, i.e. the number of units has to be multiplied by 1/(dropout rate). For example, if the dropout rate is 0.5, it will double the number of units.
- **Increase learning rate and momentum:** dropout introduces noise in the gradients, causing them to cancel each other sometimes. Increasing the learning rate by 10-100x and adding momentum between 0.95 and 0.99 compensate that effect.
- **Add max-norm regularization:** increasing the learning rate and adding momentum may result in large weight values. Adding max-norm regularization counteracts that effect.

It is worth noting that in the original paper studying dropout [28], the authors have tested some of these guidelines with a regular, non-adaptive SGD optimizer. They may not apply exactly as described for adaptive optimizers.

3.2 Batch normalization

During training of a neural network, the distribution of the input values of each layer is affected by all layers that come before it. This variability reduces training speed (lower learning rates). Batch normalization [7] was created to resolve this variability and speed up learning.

Normalizing the values of each sample before using it to the network's input layer is already a well-known technique. Batch normalization goes one step further and normalizes every layer of the network, not only the input layer. The normalization is computed for each mini-batch. A conceptual view of the standard dense network *vs.* the network structure after applying Batch normalization is shown in Fig. 2.

This normalization allows the use of higher learning rates during training (although the batch normalization paper [7] does not recommend a specific value or a range).

The way batch normalization operates, by adjusting the value of the units for each batch, and the fact that batches are created randomly during training, results in more noise during the training process. The noise acts as a regularizer. This regularization effect is similar to the one introduced by dropout. As a result, dropout can be removed completely from the network or should have its rate reduced significantly if used in conjunction with batch normalization.

Using batch normalization requires some adjustments to the hyperparameters. The more significant changes are:

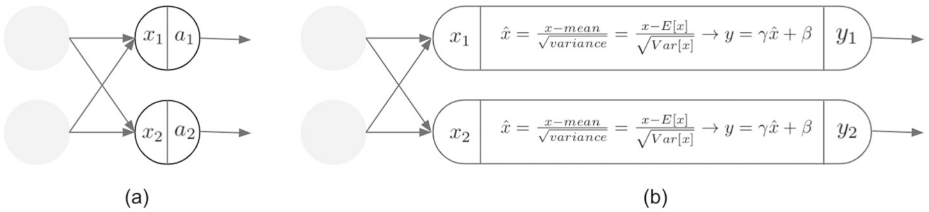

x_1 a_1
x_2 a_2
(a)
x_1 $\hat{x} = \frac{x - mean}{\sqrt{variance}} = \frac{x - E[x]}{\sqrt{Var[x]}} \rightarrow y = \gamma\hat{x} + \beta$ y_1
x_2 $\hat{x} = \frac{x - mean}{\sqrt{variance}} = \frac{x - E[x]}{\sqrt{Var[x]}} \rightarrow y = \gamma\hat{x} + \beta$ y_2
(b)

Fig. 2 A conceptual view of batch normalization. Instead of using the values from the previous layer unchanged (**a**), batch normalization normalizes the input values to have mean of zero and variance of one (**b**): $\hat{x} = \frac{x-mean}{\sqrt{variance}} = \frac{x-E[x]}{\sqrt{Var[x]}}$. γ and β are newly-introduced, learnable parameters that scale and shift the normalized values

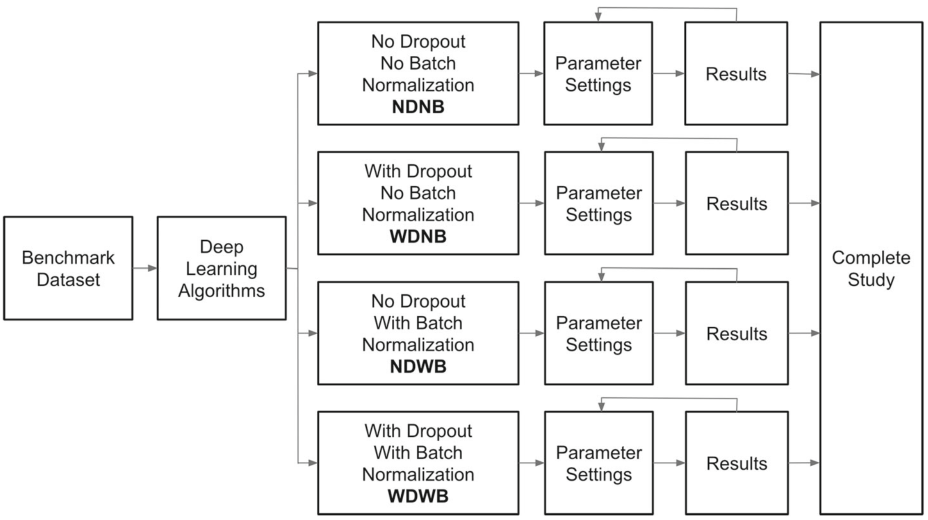

Benchmark Dataset
Deep Learning Algorithms
No Dropout No Batch Normalization NDNB
With Dropout No Batch Normalization WDNB
No Dropout With Batch Normalization NDWB
With Dropout With Batch Normalization WDWB
Parameter Settings
Results
Complete Study

Fig. 3 Empirical study framework and designs. We systematically test four combinations to train deep neural networks, including No dropout, no batch normalization (NDNB); with dropout, no batch normalization (WDNB); no dropout, with batch normalization (NDWB); and with dropout, with batch normalization (WDWB), and comparatively study the impact of dropout and batch normalization for deep neural network learning

- **Increase the learning rate**: the normalization stabilizes the training process, allowing higher learning rates.
- **Remove dropout or use lower dropout rates**: batch normalization also has a regularization effect. This effect reduces the need for dropout to the point it is no longer needed. If it is used, it should be used with lower rates.

3.3 Empirical study framework

In order to study the interplay between dropout, batch normalization, and deep neural network training, we carry out an empirical study by following the framework shown in Fig. 3. Given a benchmark dataset and a deep learning algorithm (detailed in Section 4), we systematically test four combinations: No dropout, no batch normalization (NDNB); with dropout, no batch normalization (WDNB); no dropout, with batch normalization (NDWB); and with dropout, with batch normalization (WDWB). By empirically testing each of the four combinations using different parameter settings (the feedback loop showing in Fig. 3), and collecting outcomes of the trained models, including training times, classification accuracy, number of parameters, our study intends to understand how dropout and batch normalization behave to improve (or negatively impact) the deep learning architectures.[3]

In the following section, we will detail experimental settings, including benchmark data, deep learning algorithms, and performance measures, followed by detailed experimental results and analysis in Section 5.

[3]The source code used in the experiments is available in Github at https://github.com/fau-masters-collected-works-cgarbin/cap6619-deep-learning-term-project

4 Experiments and settings

4.1 Benchmark datasets

We carry out experiments and comparisons on two benchmark image classification datasets: MNIST [15] and CIFAR-10 [10].

4.1.1 MNIST: handwritten digits

MNIST (Modified National Institute of Standards and Technology) [15] is a set of 60,000 training and 10,000 test images of handwritten digits from 0 to 9. All images are 28×28 pixels in grayscale, with the digits centered in the image.

It was created by combining a subset of two NIST digit sets, one containing digits from employees of the Census Bureau, and one containing digits from high-school students. These sets were chosen to provide a mix of samples that are considered easy to classify (the ones from the Census Bureau employees) and samples that are relatively harder to classify (the ones from the high-school students).

To make the validation process more meaningful the training and test sets were split by the original writers in a way that they do not overlap (digits from a given writer are either in the training set or the test set, but not in both) and the two types of writers are equally represented (half of the samples in the training and test set comes from the Census Bureau employees, the other half from the high-school students). This careful split results in a robust evaluation of a trained model (it will be presented with samples from writers it has not been trained on).

Figure 4 shows a sample of the MNIST dataset.

4.1.2 CIFAR-10: natural images

CIFAR-10 (Canadian Institute For Advanced Research) dataset [10] is a collection of natural images that are commonly used to train machine learning and computer vision algorithms. The dataset contains 50,000 training images and 10,000 test images. All images are 32×32 pixels in color (RGB channels). The dataset has 10 classes of images, representing airplanes,

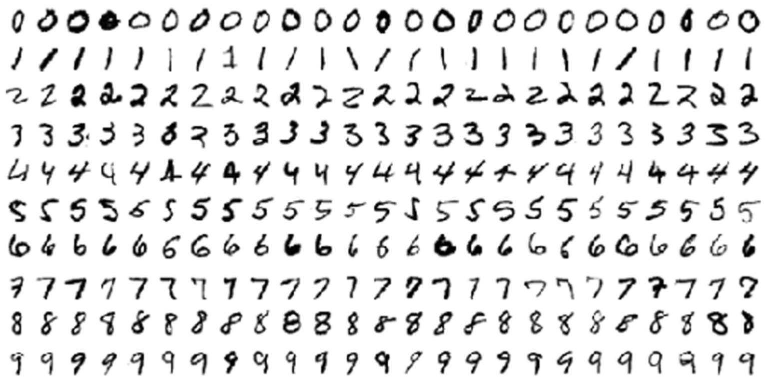


Fig. 4 Sample handwritten digits of the MNIST (Modified National Institute of Standards and Technology) dataset. Each row represents a digit written in different styles

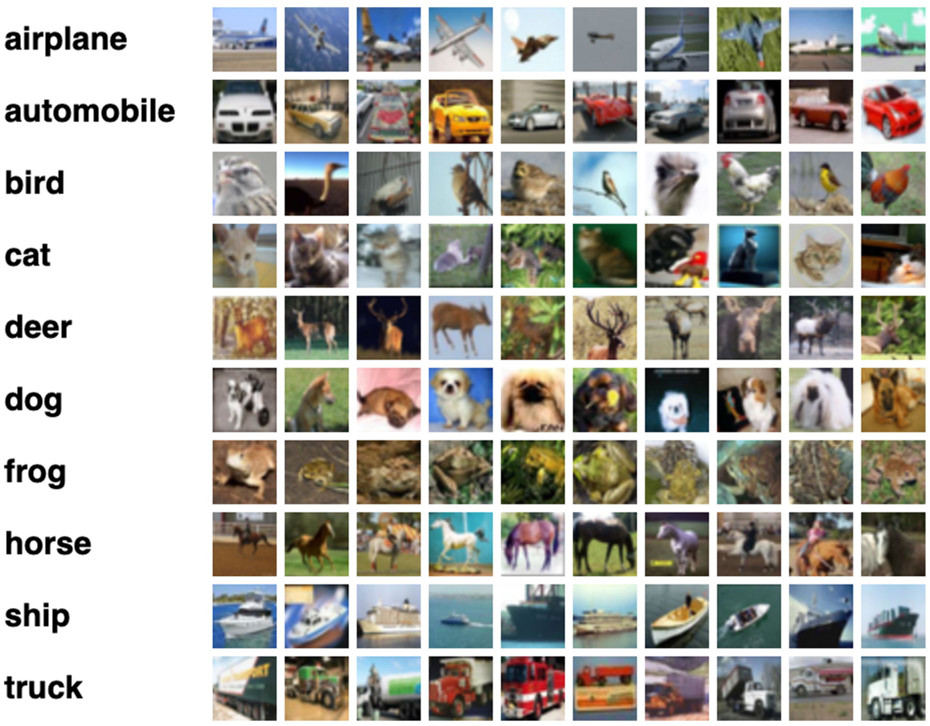


Fig. 5 Samples images of the CIFAR-10 (Canadian Institute For Advanced Research) dataset [12]. Each row represents a type of objects (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, or trucks) under different background or visual appearance

cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks, as shown in Fig. 5. The training set has 5,000 images in each class, and the test set has 1,000 images in each class.

The set was created from images collected from the Internet and manually labeled by humans. The creation process ensured that no duplicates or synthetically-created images are present in the set.

Figure 5 shows a sample of the CIFAR-10 dataset.

4.2 Deep learning algorithms

We carry out comparisons using two types of deep learning architectures, multilayer perceptron networks (MLPs) and convolutional neural networks (CNNs) with different hyperparameter configurations. The architectures of the networks are detailed in Section 5.

4.2.1 MLP: multilayer perceptron network

Multilayer perceptron networks are networks composed of several fully connected layers. An example is shown in Fig. 6.

The MNIST dataset was used to test the MLP networks and hyperparameters combinations. MNIST was chosen for this test because the original dropout paper [28] also used MNIST in their MLP tests and formulated their guidelines for hyperparameters based on

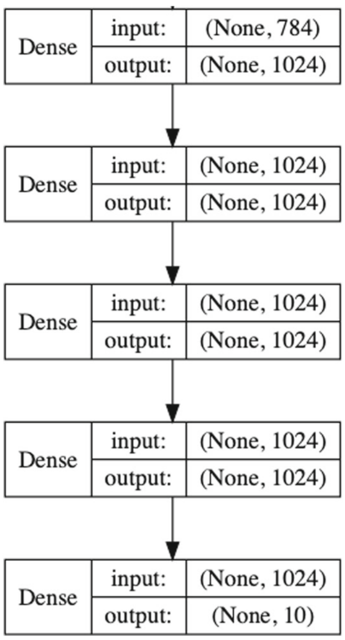


Fig. 6 MLP-NDNB: Sample standard MLP network architecture used in the tests (without dropout and without batch normalization)

that. The dropout paper [28] recommends ranges of values for hyperparameters, including number of layers (2 to 4), number of units in the hidden layer (1024 to 4096 and a special case of 8192), learning rate (10x to 100x what would normally be used without dropout), max-norm (3 to 4), among others.

We tested these combinations to document the effect they have on the network. The goal is to test combinations of the recommendations from that paper, verifying how they affect the training performance and the accuracy of the training model of an MLP network.

4.2.2 CNN: convolutional neural network

Convolutional neural networks are networks composed of several convolution and max-pooling layers. An example is shown in Fig. 11.

CNN was used to test the CIFAR-10 dataset in the dropout paper [28]. The batch normalization paper [7] used the ImageNet dataset for a similar test. For simplicity, given the hardware and time available for the tests performed here, CIFAR-10 (instead of ImageNet) was used for the batch normalization tests. Since the goal is to compare the relative performance of the strategies and hyperparameters, using CIFAR-10 should suffice in this application.

4.3 Performance metrics

In order to comparatively study the model performance, the following performance measurements are collected in our experiments.

4.3.1 Training CPU time

Training CPU time records the total CPUs and GPUs time used to run all training epochs. Our goal is to measure how much system resources a network configuration uses during the training phase.

This performance metric is used to understand how taxing the training phase is for a network configuration. Lower values are desirable to allow efficient use of expensive GPUs and to speed up the experimental phase, where different hyperparameter combinations have to be tried to find an optimal one.

In our experiments, we collected the training CPU time using the Python `time` package, which measures the time to it took to execute Keras' `model.fit(...)` function for the network configuration.

It is worth noting that for systems with more than one CPU or GPU this is not the same as clock (wall) time. Using a simplified example: if a network needs 20 seconds to be trained, in a system with two CPUs the training will complete in about 10 seconds. This item will report 20 seconds in this case. Measuring total CPU and GPU utilization gives a better view of the utilization of systems resources. Measuring only the clock time (10 seconds in this example) would not give an idea of how taxing the training phase really is on the system.

4.3.2 Test CPU time

Test CPU time records the total amount of time used across all CPUs and GPUs to evaluate the trained network using the test set. Our goal is to capture the amount of system resources the trained network uses when it is evaluating samples.

Test CPU time can help understand how efficient (or not) the trained network is on the end-users' systems. Lower values are desirable here to be responsive to the end-user and, perhaps more importantly nowadays, to make efficient use of batteries in portable devices (e.g. smartphones).

Similar to the training CPU time, we collect the test CPU time using Python `time` package, measuring the time to it took to execute Keras' `model.evaluate(...)` function on the trained network. (similar to the training CPU time, this is not the same as clock (wall) time. See the note in the *Training CPU time* section for more details).

4.3.3 Number of parameters in the model

Number of model parameters record the number of parameters the model has. Our goal is to evaluate and compare how much memory the models use.

While deep learning normally emphasizes the model accuracy, the model size, in terms of the number of parameters, provides a second metric to decide among models that have the same accuracy. The model with fewer parameters should be used because it will be faster to experiment with (run more training experiments), will use less memory on an end-user's device (and thus contribute to overall performance device by not forcing the operating system to eject other processes from memory) and use less battery (because it executes fewer calculations to predict the output).

In the experiments, we use Keras' `model.count_params()` function, called on the model object after all layers have been created. Note that this number includes non-trainable parameters, such as auxiliary variables used in batch normalization. Therefore size at test time is an approximation (but it is close enough).

4.3.4 Training loss, validation loss and accuracy for each epoch

This performance metric records the training loss and shows whether the network is improving as training progresses and how fast it is doing so. The validation loss checks if the network is overfitting or underfitting. Our goal is to measure how fast the network converges and if the network will perform well on unseen data.

Given two models that have the same accuracy, the one that achieves that accuracy faster (fewer epochs) saves system resources at training time (assuming a technique to decide that in real-time, such as early stopping, is being used) and allows faster experimentation cycles. Therefore, this measure allows us to have a measurement for training efficiency among the models tested.

In the experiments, we collect the results by setting the parameter `validation_data` when executing Keras' `model.fit(...)`. Keras returns a `History` object when validation data is supplied. This object contains the training and validation loss and accuracy for each training epoch. Note that the code uses the test portion of the datasets for this purpose. Since the code is not making decisions based on that (e.g. early stopping [22] or annealing the learning rate), this is an acceptable practice. KerasTeam [8] has a similar discussion for this usage pattern in the Keras examples.

4.4 Testing environment

All experiments reported were carried out on a Google Cloud virtual machine with the following specification

4.4.1 Machine configuration

- Machine type: n1-standard-4
- Number of CPUs: 4
- Memory: 15 GB
- GPU: 1 x NVIDIA Tesla P100

The base image used for this virtual machine was *Intel® optimized Deep Learning image*, described by Google Cloud as *A Debian based image with TensorFlow (With CUDA 10.0 and Intel® MKL-DNN, Intel® MKL) plus Intel® optimized NumPy, SciPy, and scikit-learn.*

Using a GPU is essential to explore the combination of hyperparameters described in the following sections. Training is normally 30× faster on GPU platform, compared to a relatively high-performing machine without a GPU (Intel i7 2.9 GHz, 16 GB RAM).

4.4.2 Programming tools

The following programming tools were used to implement the empirical study framework and the experiment designs:

- Python 3.5.3
- Keras 2.2.4
- TensorFlow 1.12.0 (with GPU support enabled)

5 Results and analysis

Based on the designed framework and the experimental settings detailed in the above sections, this section reports and analyzes the experiments and their results using multilayer perceptron networks (MLPs) and convolutional neural networks (CNNs).

5.1 Multilayer perceptron network results

This section reports the experiments performed with multilayer perceptron networks and analyzes the results under different settings.

5.1.1 MLP architectures, configurations, and parameter settings

Since the goal of our experiments is to compare the relative performance of the different configurations tested, the baseline for those experiments is an MLP that does not use dropout or batch normalization. This network is referred to as *Standard MLP* in the text. Therefore, we use the following three network configurations for comparisons

1. **MLP-NDNB (Standard MLP no dropout, no batch normalization:** A standard multilayer perceptron network, with dense layers connected to each other without dropout or batch normalization. This configuration is used as the baseline.
2. **MLP-WDNB (MLP with dropout, no batch normalization):** added dropout layers to the standard network, following the guidelines in the dropout paper [28].
3. **MLP-NDWB (MLP no dropout, with batch normalization):** added batch normalization layers to the standard network, following the guidelines in the batch normalization paper [7].

For each of the above three network architectures, we test their performance with a combination of these hyperparameters:

- **Hidden layers:** We use 2, 3 and 4 hidden layers. These numbers were chosen because they were described in the dropout paper [28].
- **Units in each hidden layer:** we use 1,024 and 2,048 units in the experiments. These numbers were chosen for the same reason as above.
- **Batch size:** 128 samples in each batch.
- **Epochs:** We use 5, 20 and 50 epochs in the experiments.
- **Optimizer:** a non-adaptive stochastic gradient descent (SGD) optimizer and RMSProp [30]. The SGD optimizer was chosen because there are indications that most published results use such an optimizer [25]. The RMSProp optimizer was chosen to compare the performance of SGD with an adaptive optimizer.
- **Activation function:** ReLU [23] in all cases. It was chosen because the dropout paper [28] and the batch normalization paper [7] also use it. A small-scale test was performed with sigmoid to test its behavior and found that accuracy was comparable to the ReLU (slightly lower, but not significantly). Because that test did not point to major differences, the investigations proceeded only with ReLU.

Besides the list above, each network was also tested with different values for learning rate, weight decay, SGD momentum and max-norm values. These hyperparameters depend on the network being tested. The range of values for them is documented within the respective sections below.

5.1.2 MLP-NDNB: no dropout, no batch normalization

In this subsection, we report the results of MLP-NDNB, which are standard MLPs, without dropout or batch normalization.

Network architecture A standard MLP network in this context is a network made of only dense layers, without any dropout or batch normalization layer. Figure 6 shows an example of a standard MLP network used in the tests.

Hyperparameter settings We use following hyperparameter settings in the experiments.

- Learning rates for SGD: the default Keras rate 0.01 and a higher rate 0.1 to compare the behavior of this network with the dropout network (also tested with higher learning rate).
- Learning rate for RMSProp: the default Keras rate 0.001 and a higher rate 0.002. The higher rate is not as high as the SGD based on experiments. A 10x higher rate resulted in low accuracy in small-scale tests.
- Decay for SGD: the default Keras value 0.0 (no decay applied) and a small decay 0.001 to compare with the dropout and batch normalization tests where decay was also applied.
- Decay for RMSProp: the default Keras value 0.0 (no decay applied) and a small decay 0.00001 to compare with the dropout and batch normalization tests where decay was also applied. The small decay value was chosen based on values used in the Keras examples, then verified empirically with small-scale tests.
- Momentum for SGD: the default Keras value of 0.0 (no momentum applied) and the value 0.95, also used in dropout and batch normalization test. Momentum is not applicable to RMSProp.
- Max-norm constraint: no max-norm and a max-norm constraint with max value set to 2.

The combination of these hyperparameters and the hyperparameters applicable to all MLP networks listed above resulted in 288 tests using the SGD optimizer and 144 tests with the RMSProp optimizer.

Results The top 10 best results of these tests are listed in Table 1.

5.1.3 MLP-WDNB: with dropout, no batch normalization

In this subsection, we report the results of MLP-WDNB, which are MLPs with dropout layers but without batch normalization.

Network architecture The MLP dropout network was modeled using a similar approach as the original dropout paper [28]. The significant changes compared to the standard MLP network are summarized as follows:

- A dropout layer was added after the input layer, using a lower dropout rate (compared to the dropout rate in dense layers).
- A dropout layer was added after each dense layers, with a higher dropout rate, also as recommended.

Figure 7 shows an example of a dropout network used in the tests.

Table 1 Top 10 test accuracy for the standard MLP network MLP-NDNB (no dropout or batch normalization). All tests executed with a batch size of 128 samples

Optimizer	Test accuracy	Hidden layers	Units per layer	Epochs	Learning rate	Decay	SGD moment.	max-norm	Parameters count	Training time (s)	Test time (s)
SGD	0.9879	2	2048	50	0.1	0	0.95	none	5,824,522	137	0.609
SGD	0.9878	3	2048	50	0.1	0	0.95	2	10,020,874	165	0.677
SGD	0.9869	3	2048	50	0.1	0	0.95	none	10,020,874	167	0.650
SGD	0.9866	4	2048	50	0.1	0	0.95	none	14,217,226	191	0.696
SGD	0.9865	2	2048	5	0.1	0.001	0.95	none	5,824,522	17	0.704
SGD	0.9864	4	2048	50	0.1	0	0.95	2	14,217,226	191	0.698
SGD	0.9863	2	1024	50	0.1	0	0.95	none	1,863,690	125	0.584
SGD	0.9861	3	2048	50	0.1	0.001	0.95	2	10,020,874	170	0.787
RMSprop	0.9860	2	2048	20	0.001	0.00001	0	2	5,824,522	69	0.644
SGD	0.9859	3	1024	50	0.1	0	0.95	2	2,913,290	135	0.583

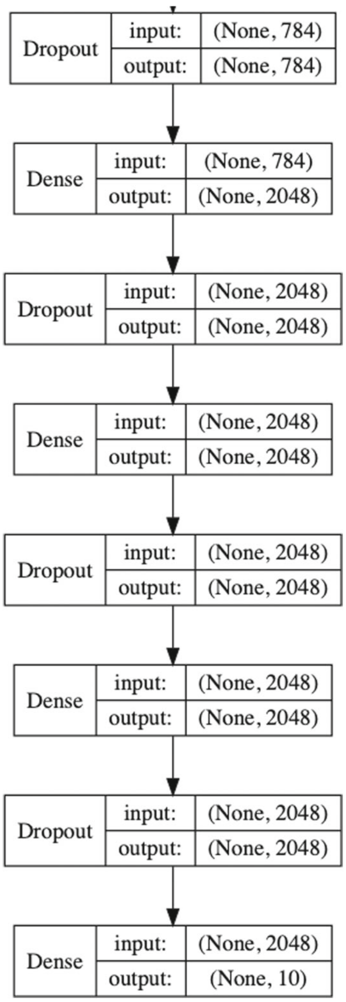


Fig. 7 MLP-WDNB: Sample dropout MLP network architecture used in the tests (with dropout but without batch normalization)

Hyperparameter settings We use following hyperparameter settings in the experiments.

- Dropout rate for the input layer: 0.1, as recommended in [28].
- Dropout rate for hidden layers: 0.5, the high end of the rate recommended in [28].
- Learning rates for SGD: the default Keras rate 0.01 and 10x the default rate (0.1) as recommended in [28].
- Learning rate for RMSProp: the default Keras rate 0.001 and a higher rate 0.01. Although the standard MLP network did not use such a high rate, the 10x rate was used here to test the recommendation in [28].
- Decay for SGD: the default Keras value 0.0 (no decay applied) and a small decay 0.001 to compare with the dropout and batch normalization tests where decay was also applied.
- Decay for RMSProp: the default Keras value 0.0 (no decay applied) and a small decay 0.00001 to compare with the dropout and batch normalization tests where decay was

also applied. The small decay value was chosen based on values used in the Keras examples, then verified empirically with small-scale tests.
- Momentum for SGD: the default Keras value of 0.0 (no momentum applied) and the two extremes of the range recommended in [28]: 0.95 and 0.99. Momentum is not applicable to RMSProp.
- Max-norm constraint: no max-norm to test the behavior of the network when using the same value as the standard MLP and two of the values recommended in [28]: 2 and 3.

The combination of these hyperparameters and the hyperparameters applicable to all MLP networks listed above resulted in 648 tests using the SGD optimizer and 216 tests with the RMSProp optimizer.

Results The top 10 results of these tests are listed in Table 2.

5.1.4 MLP-NDWB: no dropout, with batch normalization

In this subsection, we report the results of MLP-NDWB, which are MLPs without dropout layers but with batch normalization.

Network architecture The MLP batch normalization network was modeled after the batch normalization paper [7]. The significant change compared to the standard MLP network is the addition of a batch normalization layer after the dense layers.

Figure 8 shows an example of a batch normalization network used in the tests.

Hyperparameter settings We use following hyperparameter settings in the experiments.

- Learning rates for SGD: the default Keras rate 0.01 and 10x the default rate (0.1). Ioffe and Szegedy [7] recommends a higher learning rate but does not give a range of values. These values were chosen to match the ones used in the dropout MLP tests, making the comparison more meaningful.
- Learning rate for RMSProp: the default Keras rate 0.001 a higher rate 0.005. Although the standard MLP network did not use such a high rate, Ioffe and Szegedy [7] recommends a higher rate.
- Decay for SGD: the default Keras value 0.0 (no decay applied) and a small decay 0.001 to compare with the other tests where decay was also applied.
- Decay for RMSProp: the default Keras value 0.0 (no decay applied) and a small decay 0.00001 to compare with the other tests where decay was also applied. The small decay value was chosen based on values used in the Keras examples, then verified empirically with small-scale tests.
- Momentum for SGD: the default Keras value of 0.0 (no momentum applied) and the two extremes of the range recommended in [28]: 0.95 and 0.99. Momentum is not applicable to RMSProp.
- Max-norm constraint: no max-norm to test the behavior of the network when using the same value as the standard MLP and two of the values recommended in [28]: 2 and 3.

The combination of these hyperparameters and the hyperparameters applicable to all MLP networks listed above resulted in 144 tests using the SGD optimizer and 72 tests with the RMSProp optimizer.

Results The top 10 results of these tests are listed in Table 3.

Table 2 Top 10 test accuracy for the dropout MLP network MLP-WDNB. All tests executed for 50 epochs, with a batch size of 128 samples. Dropout rate for the input layer is 0.1 and 0.5 for the hidden layers

Optimizer	Test accuracy	Hidden layers	Units per layer	Learning rate	Decay	SGD moment.	max-norm	Parameters count	Training time (s)	Test time (s)
SGD	0.9881	2	2048	0.01	0.001	0.99	3	20,037,642	244	0.701
SGD	0.9879	3	2048	0.01	0	0.95	3	36,818,954	335	0.779
SGD	0.9879	4	2048	0.01	0.001	0.99	2	53,600,266	429	0.808
SGD	0.9876	3	1024	0.01	0	0.95	none	10,020,874	180	0.725
SGD	0.9876	4	2048	0.01	0.001	0.99	3	53,600,266	429	0.826
SGD	0.9875	2	1024	0.01	0.001	0.99	none	5,824,522	161	0.662
SGD	0.9875	2	2048	0.1	0	0	3	20,037,642	241	0.730
SGD	0.9875	4	1024	0.01	0	0.95	none	14,217,226	204	0.671
SGD	0.9874	2	2048	0.01	0	0.95	2	20,037,642	242	0.722
SGD	0.9874	3	2048	0.01	0.001	0.99	3	36,818,954	340	0.771

Table 3 Top 10 test accuracy for the batch normalization MLP network MLP-NDWB. All tests executed for 50 epochs, with a batch size of 128 samples

Optimizer	Test accuracy	Hidden layers	Units per layer	Learning rate	Decay	SGD moment.	Parameters count	Training time (s)	Test time (s)
SGD	0.9868	4	2048	0.01	0	0.95	14,243,850	394	0.933
SGD	0.9867	2	1024	0.1	0.0001	0.95	1,870,858	241	0.765
RMSprop	0.9867	4	2048	0.001	0.0001	0	14,243,850	439	0.927
SGD	0.9865	3	1024	0.1	0.0001	0.95	2,923,530	293	0.857
SGD	0.9864	2	2048	0.1	0.0001	0.95	5,838,858	255	0.795
SGD	0.9864	4	2048	0.01	0.0001	0.95	14,243,850	388	0.901
RMSprop	0.9862	3	1024	0.001	0.0001	0	2,923,530	125	0.868
SGD	0.9860	2	2048	0.01	0	0.95	5,838,858	255	0.780
SGD	0.9860	4	2048	0.01	0	0.95	14,243,850	161	0.927
SGD	0.9859	2	2048	0.01	0	0.95	5,838,858	101	0.764

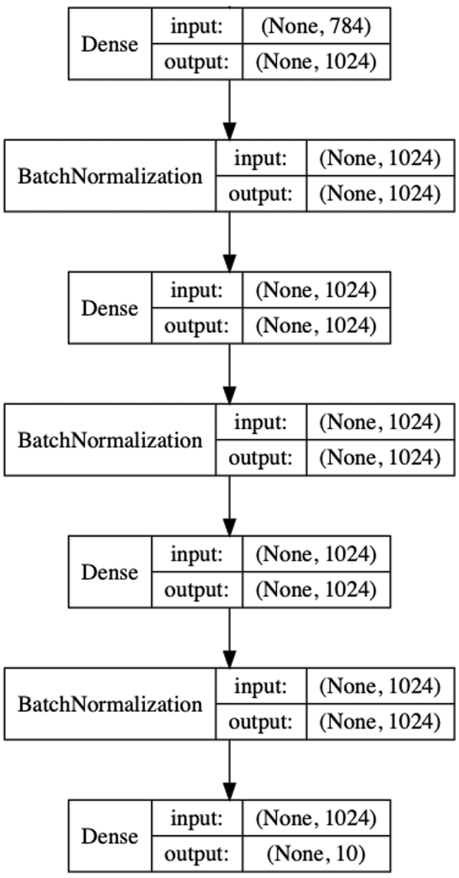


Fig. 8 MLP-NDWB: Sample batch normalization MLP network used in the tests (with batch normalization but without dropout)

5.1.5 MLP result analysis

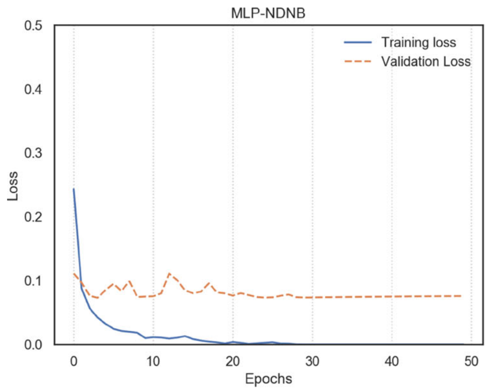
We compare and analyze the MLP results using classification accuracy, CPU time, model sizes, and combination of hyperparameters.

Classification accuracy All networks resulted in similar accuracy, with a small edge for dropout.

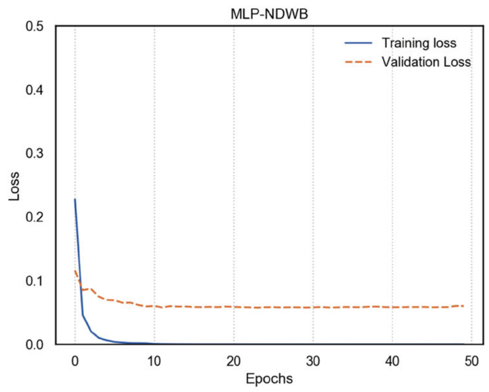
Figure 9 shows that the accuracy is not reached at the same time. The figure plots the training (blue, dotted line) and test (solid orange line) loss during training.

The batch normalization network (MLP-NDWB) (see Fig. 9c) reaches its lower value much earlier than other networks, as expected, given that one of its purposes is to accelerate learning [7]. This can be taken advantage of to shorten training times when very high accuracy is not needed. Early stopping would stop training with the batch normalization sooner than with the other networks.

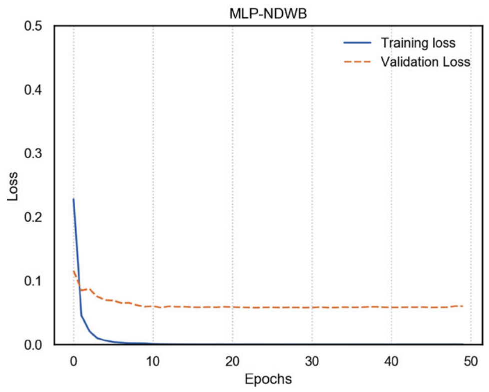
Fig. 9 Model training and test loss with respect to epochs for the top network in each category (MLP-NDNB, MLP-WDNB, and MLP-NDWB). All networks configured with 2,048 units in the hidden layers, trained with an SGD optimizer for 50 epochs. Parameters specific to a network noted under each graph



(a) Loss for MLP-NDNB (Standard MLP) - 2 hidden layers, learning rate = 0.1, no weight decay



(b) Loss for MLP-WDNB (MLP with only dropout) - 2 hidden layers, learning rate = 0.01, weight decay = 0.001



(c) Loss for MLP-NDWB (MLP with only batch normalization) - 4 hidden layers, learning rate = 0.01, no weight decay

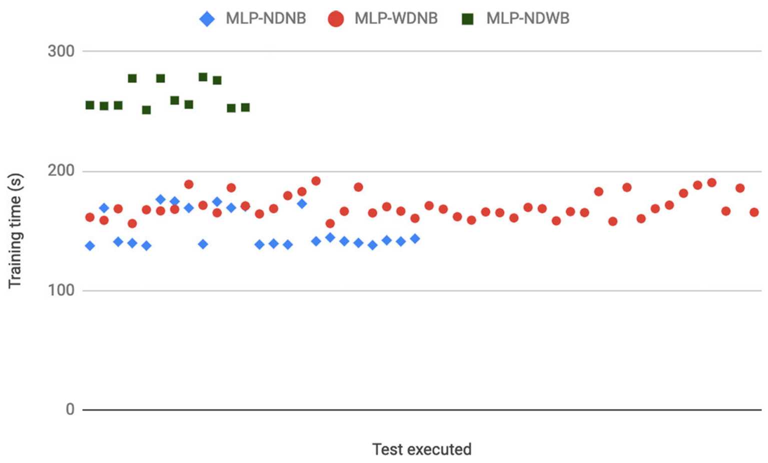


Fig. 10 Training time in seconds for MLP-NDNB (Standard MLP), MLP-WDNB (MLP with only dropout), and MLP-NDWB (MLP with only batch normalization) networks using two hidden layers, 2,048 units in each layer, trained for 50 epochs. The vertical axis shows the training time (time to execute all epochs) in seconds. The horizontal axis shows the test executed, with different hyperparameters (MLP-WDNB has more tests because of its larger combination of hyperparameters to test)

Training and test CPU time, parameter count To evaluate the training and test CPU and parameter count, the best results of each network configuration using two hidden layers was extracted into Table 4.[4]

It shows these behaviors of the different networks:

- Training CPU time: dropout increases training time by approximately 17% (line 2 in Table 4). Batch normalization increases training time by approximately 86% (line 3 in 4). As shown in Fig. 10, this increase in training time happens in all combinations of hyperparameters. It is not the product of a specific set of hyperparameters. From that we can make the general statement that batch normalization training time is approximately 80% longer than the standard and dropout networks.
- Test CPU time: batch normalization is significantly slower (30+%) at test time (line 3 in Table 4). This result is surprising, given that the network architecture is effectively the same. It could be a fluctuation of the environment. It needs some further research. If it does indeed increase the test time by this much, it has significant implications for uses in restricted environments, e.g. mobile phones, were battery conservation is a high-priority concern.
- Parameter count: as expected, batch normalization (line 3 in Table 4) uses more parameters than the standard MLP and dropout and therefore more memory (parameter count is used as a proxy for memory usage). However, the increase is small (less than 1%) and occurs only at training time (where usually more memory is available).

Combination of hyperparameters Inspecting the top 10 results for each network reveals some patterns.

[4]Note that the dropout network is listed in the top 10 results as "1,024 hidden units". The number of units is adjusted by the dropout rate, 0.5 in this case. The adjustment results in a dropout network configured to run with 1,024 units in a layer to effectively have 2,048 units in that layer.

Table 4 Performance evaluation comparing networks with two hidden layers and 2,048 hidden units (note that dropout is divided by 0.5, the dropout rate). All tests were done with 50 epochs, SGD optimizer, 0.1 learning rate and 0.99 momentum for dropout, 0.95 for the other networks

Network	Accuracy	Training time (s)	Test time (s)	Parameters count
Standard	0.9879	137	0.609	5,824,522
Dropout	0.9875	161	0.662	5,824,522
Batch normalization	0.9864	255	0.795	5,838,858

The non-adaptive optimizer SGD performed unexpectedly well compared to the adaptive RMSProp. However, changing the default learning rate and adding momentum were needed to achieve that performance. Adding a max-norm constraint was needed in most cases as well.

Getting to that performance level requires experimentation with those hyperparameters, which translates in more training time. The small gain in accuracy compare to RMSProp may not justify the time invested.

5.1.6 Recommendations and discussions

The MLP results suggest the following findings and recommendations:

- Use batch normalization if the convergence time is more important than absolute accuracy (Fig. 9). Together with early stopping, it could significantly reduce training time.
- But be aware of batch normalization's training time increase (Table 4). Unless it can be shown that it is helping converge faster during training, it may not be worth using it for experiments. Each experiment will take significantly longer to complete. It may be better to start with a standard network to run experiments faster, then switch to batch normalization in a later phase.
- Start with an adaptive optimizer (e.g. RMSProp). The experiments show that a non-adaptive SGD optimizer can be fine-tuned to outperform an adaptive one, but that comes at the cost of trying combinations of hyperparameters to find one that performs well (see for example the varying learning rate, decay, momentum and max-norm of the entries in Table 2). This adds to the training time. The accuracy of the adaptive optimizer with its default settings is not much lower. Starting with that configuration quickly provides a baseline for the tests and frees up time to experiment with other hyperparameters (e.g. the number of hidden layers, batch size, etc.).

Future investigations Considering the results so far and what was learned in producing this paper, these are some improvements that could be done in the experiments and data collection process:

- Batch normalization test time validation: tests showed that batch normalization test time is significantly higher than the standard MLP and dropout. This is unexpected and warrants more investigations.
- Force overfitting in each test: to better evaluate the effect of the hyperparameters, the test should begin by verifying that overfitting is taking place and where it does so (which epoch). Forcing overfitting would have triggered more differences in accuracy across the network types, providing more actionable recommendations for the readers.

Once the network is overfitting we can verify if the hyperparameter changes resolve the overfitting and how soon it does so (which epoch). A possible way to force overfitting in these tests is to reduce the number of samples in the training set.

- Effect of different dropout rates: the tests were performed with the dropout rates recommended in [28] because of the limited amount of time. Since the dropout rate is a key hyperparameter, another investigation path could be to explore its effect on the top 10 results (e.g. could we improve the dropout network further with different dropout rates?).
- Capture tensorboard data: Keras can save data during training into a format that tensorboard can read. Making the data available in this format allows a deeper, more detailed exploration of the results by other readers, potentially resulting in more insights.

5.2 Convolutional neural network results

This section reports empirical studies using convolutional neural networks. CNNs are one of the most commonly used deep learning architectures. Convolution and max-pooling layers are their main feature, followed by flattening and dense layers before an output layer.

5.2.1 CNN architectures, configurations, and parameter settings

Since the goal of these experiments is to compare the relative performance of the different configurations tested, the baseline for those experiments is a CNN that does not use dropout or batch normalization. This network is referred to as *Standard CNN*. In summary, we use the following five network configurations for comparisons:

- **CNN-NDNB (Standard CNN no dropout, no batch normalization):** a standard CNN, with convolution and max-pooling layers, without using any dropout or batch normalization layer. This configuration is used as the baseline. This CNN was based on the official Keras example [29] and similar to the CNN used in [28] for the Google Street View House Numbers tests. Figure 11 shows this network configuration.
- **CNN-WDNB$_d$ (CNN with dropout, no batch normalization):** This architecture adds dropout before the dense layer. Starting with the standard CNN, added a dropout layer right before the dense layer. No other dropout or batch normalization layer was added. Figure 12 shows this network configuration.
- **CNN-WDNB$_a$ (CNN with dropout, no batch normalization):** This architecture includes dropout after all layers. Starting with the standard CNN, added dropout to the convolutions and also before the dense layer. The dropout layer was added after the max-pooling layers. Figure 13 shows this network configuration.
- **CNN-NDWB (CNN no dropout, with batch normalization):** starting with the standard CNN, added batch normalization layers between the convolution and the max-pooling layers. Although [7] adds batch normalization before the non-linearity, subsequent experiments reported that adding batch normalization after the non-linearity improves accuracy [21]. Because of such reports, tests were executed with the batch normalization layer after the non-linearity layer. Figure 14 shows this network configuration.
- **CNN-WDWB (CNN with dropout and batch normalization):** This architecture includes both dropout and batch normalization networks. Figure 15 shows this network configuration. Dropout was applied after batch normalization, following the recommendation from [17], with the variation that we add dropout after each batch normalization layer.

All tests were executed with data augmentation using Keras `ImageDataGenerator` using these transformations:

- Random vertical shift with a factor of 0.1 (of total height), filling the new pixels with the nearest neighbor.
- Random horizontal shift with a factor of 0.1 (of total width), filling the new pixels with the nearest neighbor.
- Random horizontal flipping of images.

Hyperparameter settings Each of these networks was tested with a combination of the hyperparameters listed below, using the CIFAR-10 dataset.

Because testing a CNN with a meaningful data set such as CIFAR-10, even with a small number of layers, is time-consuming, a small combination of hyperparameters was tested (compared to the MLP test).

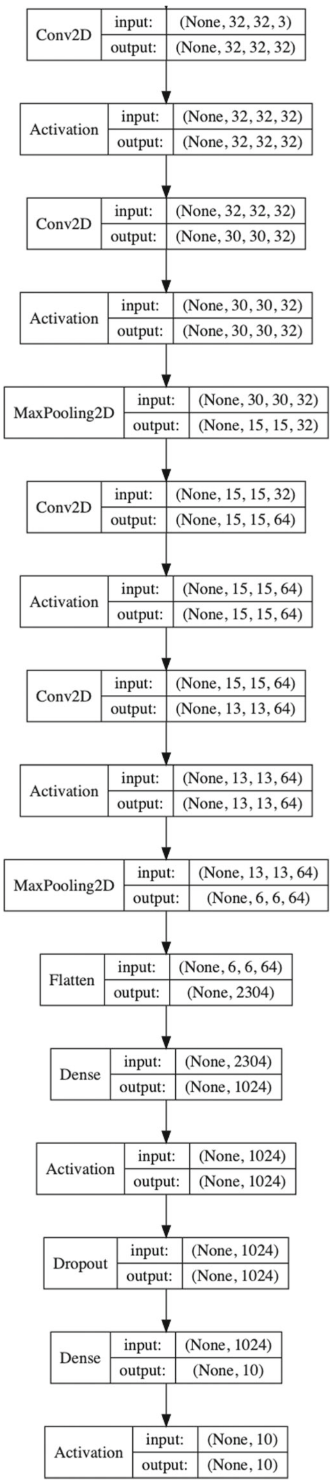
- Optimizer: all tests used RMSProp. Although testing with SGD may have provided a useful contrast between an adaptive optimizer (RMSProp) and a non-adaptive one (SGD), time and resources considerations limited the tests to RMSProp.
- Learning rate: all CNNs were tested with learning rates 0.0001, the default Keras value. In addition to that the dropout CNNs were tested with 0.001 (the 10x value recommended in the dropout paper [28]); batch normalization CNNs were tested with 0.0005, a higher rate as recommended in general terms (without providing specific values) in [7]. The standard CNN was tested with 0.001 and 0.0005 for comparison.
- Units in the dense layer: All CNNs were tested with 512 units in the dense layer, as shown in the official Keras example. In addition to that test, the dropout CNNs were tested with 1024 units to follow the recommendation in [28] to adjust the number of units based on the dropout rate (0.5 in this case).
- Epochs: all networks were tested with 50 epochs. This number was chosen to let the networks stabilize and to have a reasonable test execution time. Even with this relatively small number of epochs, training one network in a GPU-enable machine took 20 minutes.
- Dropout rate: a dropout rate of 0.25 was used after convolution layers and 0.5 after dense layers. A smaller dropout rate for the convolution layers was used as documented in [28].
- Activation function: ReLU [23] in all cases. Krizhevsky et al. [11] showed that ReLU speeds up the training significantly.

5.2.2 CNN result analysis

Results from the tests are summarized in Table 5.

Classification accuracy Adding batch normalization significantly improves the accuracy. Dropout, on the other hand, was always detrimental to accuracy (as used in these experiments - see the next section for recommendations).

Training CPU time and parameter count Adding batch normalization increased training time, as expected. However, contrary to the MLP test, adding batch normalization did not result in a large increase in training time. It increased training time by approximately 10%.



Conv2D input: (None, 32, 32, 3) output: (None, 32, 32, 32)
Activation input: (None, 32, 32, 32) output: (None, 32, 32, 32)
Conv2D input: (None, 32, 32, 32) output: (None, 30, 30, 32)
Activation input: (None, 30, 30, 32) output: (None, 30, 30, 32)
MaxPooling2D input: (None, 30, 30, 32) output: (None, 15, 15, 32)
Conv2D input: (None, 15, 15, 32) output: (None, 15, 15, 64)
Activation input: (None, 15, 15, 64) output: (None, 15, 15, 64)
Conv2D input: (None, 15, 15, 64) output: (None, 13, 13, 64)
Activation input: (None, 13, 13, 64) output: (None, 13, 13, 64)
MaxPooling2D input: (None, 13, 13, 64) output: (None, 6, 6, 64)
Flatten input: (None, 6, 6, 64) output: (None, 2304)
Dense input: (None, 2304) output: (None, 1024)
Activation input: (None, 1024) output: (None, 1024)
Dropout input: (None, 1024) output: (None, 1024)
Dense input: (None, 1024) output: (None, 10)
Activation input: (None, 10) output: (None, 10)

Fig. 11 CNN-NDNB: Standard CNN network architecture used in the tests (without dropout and without batch normalization)

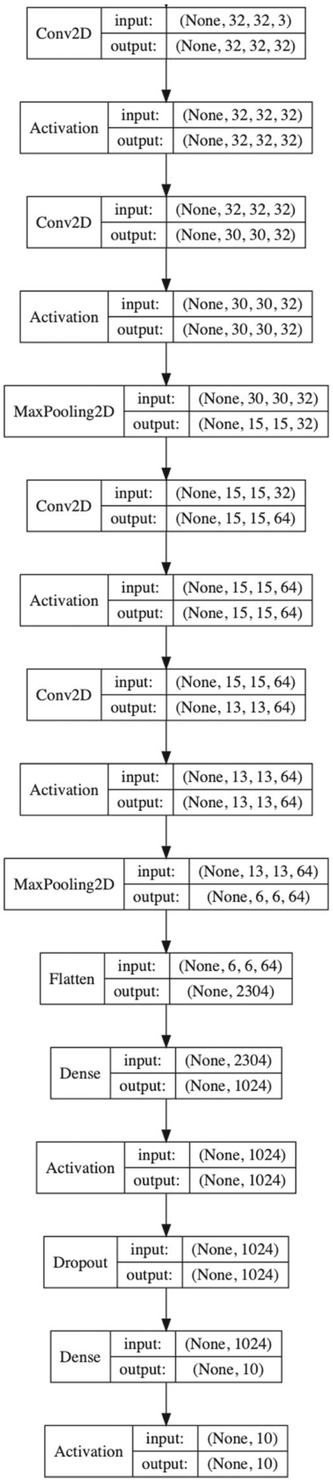

Conv2D input: (None, 32, 32, 3) output: (None, 32, 32, 32)
Activation input: (None, 32, 32, 32) output: (None, 32, 32, 32)
Conv2D input: (None, 32, 32, 32) output: (None, 30, 30, 32)
Activation input: (None, 30, 30, 32) output: (None, 30, 30, 32)
MaxPooling2D input: (None, 30, 30, 32) output: (None, 15, 15, 32)
Conv2D input: (None, 15, 15, 32) output: (None, 15, 15, 64)
Activation input: (None, 15, 15, 64) output: (None, 15, 15, 64)
Conv2D input: (None, 15, 15, 64) output: (None, 13, 13, 64)
Activation input: (None, 13, 13, 64) output: (None, 13, 13, 64)
MaxPooling2D input: (None, 13, 13, 64) output: (None, 6, 6, 64)
Flatten input: (None, 6, 6, 64) output: (None, 2304)
Dense input: (None, 2304) output: (None, 1024)
Activation input: (None, 1024) output: (None, 1024)
Dropout input: (None, 1024) output: (None, 1024)
Dense input: (None, 1024) output: (None, 10)
Activation input: (None, 10) output: (None, 10)

Fig. 12 CNN-WDNB$_d$: CNN network architecture with dropout after the dense layer used in the tests

Fig. 13 CNN-WDNB$_a$: CNN network architecture with dropout in all layers used in the tests

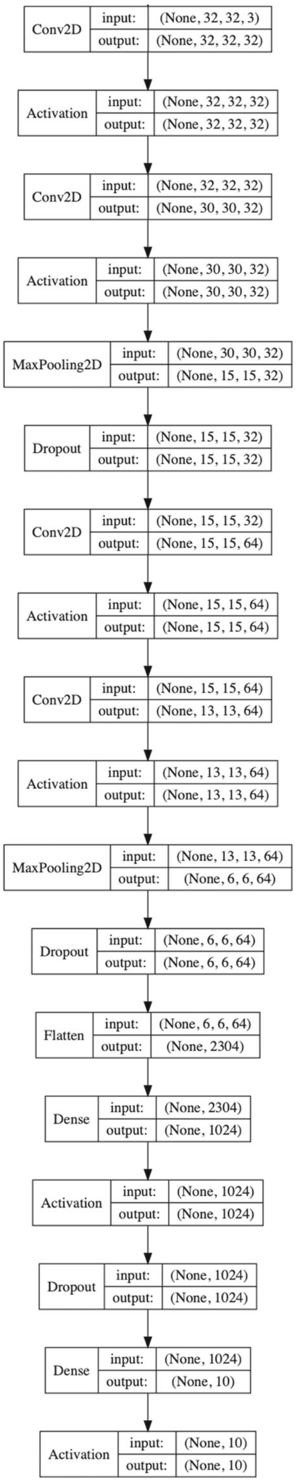

Conv2D input: (None, 32, 32, 3) output: (None, 32, 32, 32)
Activation input: (None, 32, 32, 32) output: (None, 32, 32, 32)
Conv2D input: (None, 32, 32, 32) output: (None, 30, 30, 32)
Activation input: (None, 30, 30, 32) output: (None, 30, 30, 32)
MaxPooling2D input: (None, 30, 30, 32) output: (None, 15, 15, 32)
Dropout input: (None, 15, 15, 32) output: (None, 15, 15, 32)
Conv2D input: (None, 15, 15, 32) output: (None, 15, 15, 64)
Activation input: (None, 15, 15, 64) output: (None, 15, 15, 64)
Conv2D input: (None, 15, 15, 64) output: (None, 13, 13, 64)
Activation input: (None, 13, 13, 64) output: (None, 13, 13, 64)
MaxPooling2D input: (None, 13, 13, 64) output: (None, 6, 6, 64)
Dropout input: (None, 6, 6, 64) output: (None, 6, 6, 64)
Flatten input: (None, 6, 6, 64) output: (None, 2304)
Dense input: (None, 2304) output: (None, 1024)
Activation input: (None, 1024) output: (None, 1024)
Dropout input: (None, 1024) output: (None, 1024)
Dense input: (None, 1024) output: (None, 10)
Activation input: (None, 10) output: (None, 10)

Fig. 14 CNN-NDWB: CNN network architecture with batch normalization in all layers used in the tests

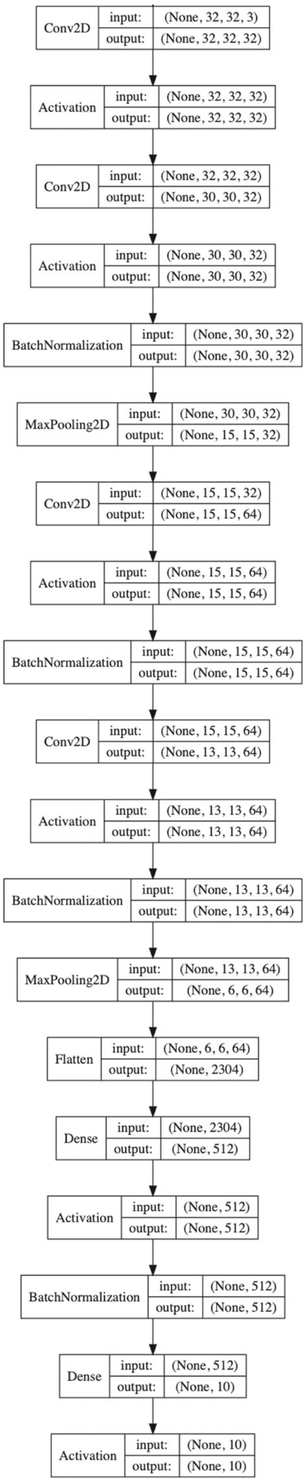

Conv2D
input: (None, 32, 32, 3)
output: (None, 32, 32, 32)
Activation
input: (None, 32, 32, 32)
output: (None, 32, 32, 32)
Conv2D
input: (None, 32, 32, 32)
output: (None, 30, 30, 32)
Activation
input: (None, 30, 30, 32)
output: (None, 30, 30, 32)
BatchNormalization
input: (None, 30, 30, 32)
output: (None, 30, 30, 32)
MaxPooling2D
input: (None, 30, 30, 32)
output: (None, 15, 15, 32)
Conv2D
input: (None, 15, 15, 32)
output: (None, 15, 15, 64)
Activation
input: (None, 15, 15, 64)
output: (None, 15, 15, 64)
BatchNormalization
input: (None, 15, 15, 64)
output: (None, 15, 15, 64)
Conv2D
input: (None, 15, 15, 64)
output: (None, 13, 13, 64)
Activation
input: (None, 13, 13, 64)
output: (None, 13, 13, 64)
BatchNormalization
input: (None, 13, 13, 64)
output: (None, 13, 13, 64)
MaxPooling2D
input: (None, 13, 13, 64)
output: (None, 6, 6, 64)
Flatten
input: (None, 6, 6, 64)
output: (None, 2304)
Dense
input: (None, 2304)
output: (None, 512)
Activation
input: (None, 512)
output: (None, 512)
BatchNormalization
input: (None, 512)
output: (None, 512)
Dense
input: (None, 512)
output: (None, 10)
Activation
input: (None, 10)
output: (None, 10)

Fig. 15 CNN-WDWB: CNN network architecture with dropout and batch normalization in all layers used in the tests

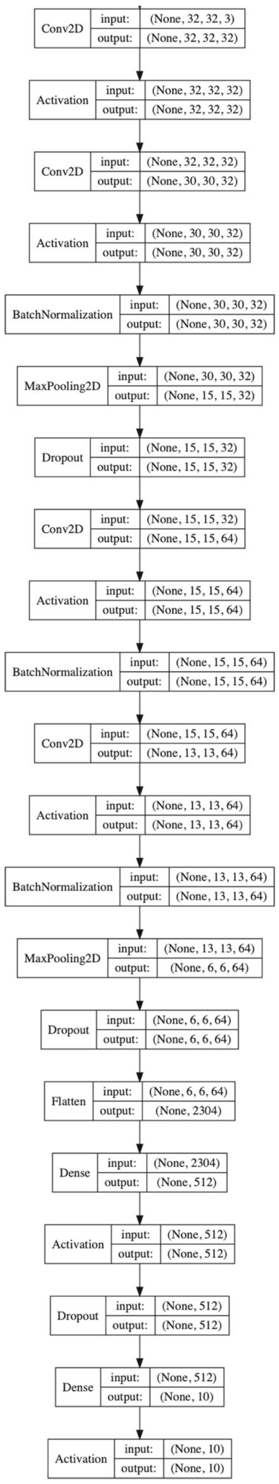

Conv2D | input: (None, 32, 32, 3) | output: (None, 32, 32, 32)
Activation | input: (None, 32, 32, 32) | output: (None, 32, 32, 32)
Conv2D | input: (None, 32, 32, 32) | output: (None, 30, 30, 32)
Activation | input: (None, 30, 30, 32) | output: (None, 30, 30, 32)
BatchNormalization | input: (None, 30, 30, 32) | output: (None, 30, 30, 32)
MaxPooling2D | input: (None, 30, 30, 32) | output: (None, 15, 15, 32)
Dropout | input: (None, 15, 15, 32) | output: (None, 15, 15, 32)
Conv2D | input: (None, 15, 15, 32) | output: (None, 15, 15, 64)
Activation | input: (None, 15, 15, 64) | output: (None, 15, 15, 64)
BatchNormalization | input: (None, 15, 15, 64) | output: (None, 15, 15, 64)
Conv2D | input: (None, 15, 15, 64) | output: (None, 13, 13, 64)
Activation | input: (None, 13, 13, 64) | output: (None, 13, 13, 64)
BatchNormalization | input: (None, 13, 13, 64) | output: (None, 13, 13, 64)
MaxPooling2D | input: (None, 13, 13, 64) | output: (None, 6, 6, 64)
Dropout | input: (None, 6, 6, 64) | output: (None, 6, 6, 64)
Flatten | input: (None, 6, 6, 64) | output: (None, 2304)
Dense | input: (None, 2304) | output: (None, 512)
Activation | input: (None, 512) | output: (None, 512)
Dropout | input: (None, 512) | output: (None, 512)
Dense | input: (None, 512) | output: (None, 10)
Activation | input: (None, 10) | output: (None, 10)

Table 5 CNN test results and comparisons, trained for 50 epochs. The best result for each network architecture is highlighted in bold

Test	Network	Test accuracy	Learning rate	Units in dense layer	Parameters count	Training time (s)
1	CNN-NDNB (Standard CNN, no dropout, no batch normalization)	0.3226	0.001	1024	2,436,138	2920
2	CNN-NDNB (Standard CNN, no dropout, no batch normalization)	0.6945	0.0005	512	1,250,858	2878
3	**CNN-NDNB (Standard CNN, no dropout, no batch normalization)**	**0.8041**	**0.0001**	**1024**	**2,436,138**	**2884**
4	CNN-NDNB (Standard CNN, no dropout, no batch normalization)	0.5717	0.001	512	1,250,858	2872
5	CNN-NDNB (Standard CNN, no dropout, no batch normalization)	0.8021	0.0001	512	1,250,858	2866
6	CNN-WDNB$_d$ (CNN with dropout, no batch normalization)	0.7426	0.0001	512	1,250,858	2881
7	CNN-WDNB$_d$ (CNN with dropout, no batch normalization)	0.1001	0.001	1024	2,436,138	2894
8	**CNN-WDNB$_a$ (CNN with dropout, no batch normalization)**	**0.7575**	**0.0001**	**512**	**1,250,858**	**2904**
9	CNN-WDNB$_a$ (CNN with dropout, no batch normalization)	0.3226	0.001	1024	2,436,138	2920
10	**CNN-NDWB (CNN no dropout, with batch normalization)**	**0.8447**	**0.001**	**512**	**1,253,546**	**3185**
11	CNN-NDWB (CNN no dropout, with batch normalization)	0.8266	0.0001	512	1,253,546	3174
12	CNN-NDWB (CNN no dropout, with batch normalization)	0.8395	0.0005	512	1,253,546	3189
13	CNN-WDWB (CNN with dropout and batch normalization)	0.8002	0.0001	512	1,251,498	3092
14	**CNN-WDWB (CNN with dropout and batch normalization)**	**0.8087**	**0.001**	**1024**	**2,436,778**	**3094**
15	CNN-WDWB (CNN with dropout and batch normalization)	0.7774	0.0005	512	1,251,498	3083

Effects of learning rate Increasing the learning rate yields better accuracy only when batch normalization is used. In all other cases it is detrimental to accuracy. Although not a surprising result for the standard CNN, increasing the learning rate when dropout is used in all layers (test 9) also resulted in much lower accuracy.

5.2.3 Recommendations and discussions

- Add batch normalization before attempting other changes: combined with increasing the learning rate (see next item), adding batch normalization improved accuracy by a significant value without a significant increase of training time (tests 10, 11 and 12 in Table 5). Because it is simple to add batch normalization, it is recommended to add it as a baseline for further improvements in the network performance, before attempting more costly hyperparameter changes.
- Learning rate value: increase it only when using batch normalization. Ioffe and Szegedy [7] recommends to increase it and it does make a significant difference. Tests 10 and 11 in Table 5 shows that increasing it by $10\times$ improves accuracy by 3%, without any other change in the test parameters. However, increasing it for any other configuration, including dropout, reduces accuracy.
- Dropout rate and layer placement: try several dropout rates and dropout layer placement. While the tests in this paper resulted in lower accuracy (Table 5 – tests 6 to 9, using only dropout, and tests 13 to 15, where dropout cancels out the batch normalization accuracy gains), [17] showed improved accuracy when dropout is used, but only for specific dropout rates and in specific places in the network. The specific rates that result in better accuracy vary by network configuration. Trial-and-error is still the best method to find a good dropout rate and layer placement.

5.2.4 Future investigations

Based on the results collected so far, these are some areas that could be investigated further:

- The low accuracy of dropout combined with batch normalization: contrary to the tests performed here, [17] reported that dropout can be used to improve accuracy. However, the results depended on the dropout rate. While [17] tried several dropout rates, in this report we tried a fixed rate. Investigating the reasons for the low accuracy with specific dropout rates could provide more information to understand the interactions between dropout and batch normalization.
- The low accuracy of dropout in general: this was perhaps the most unexpected result. Several examples, including the official Keras example, add dropout to the network with the assumed intention that it improves accuracy. Further tests should explore other hyperparameters. Two places to start are increasing the batch size, as recommended in [20], and reducing the dropout rate, as recommended in [17]. Together with the previous item, future investigations in this area could reveal a pattern to choose dropout rates for specific network configurations, saving valuable training time. Until then, trying different dropout rates is still the best method to find an effective one.
- Add max-norm and momentum: as seen in the MLP results, max-norm and momentum (when testing with a non-adaptive SGD optimizer) make a difference in the behavior of the network. They were not used in the CNN tests due to the limited time (each CNN test takes 20 minutes in a GPU-enabled system). These tests could help explain the low accuracy when dropout is used.

- Deeper networks: the CNN used in the tests is relatively shallow. Further tests should be executed in deeper networks to verify these results.
- Batch normalization layer after the non-linear layer: although [21] showed that using the batch normalization layer after the non-linear layer improves accuracy, it did not happen for all network types. Testing with the batch normalization layer before the non-linear layer, together with max-norm and momentum, could provide more insights on the performance of CNNs.
- Capture tensorboard data: same as noted in the MLP recommendation. Making the data available in this format allows a deeper, more detailed exploration of the results by other readers, potentially resulting in more insights.

6 Conclusions

Deep neural network training commonly uses dropout, a popular regularization strategy, and batch normalization, a mitigation to the gradient vanishing problem, to improve the model performance. In many cases, dropout and batch normalization overlap in applications, whereas guidelines to use them are sometimes contradicting and often lack in details. In this paper, we carried out an empirical study to examine the impact of dropout and batch normalization on multilayer perceptron networks (MLPs) and convolutional neural networks (CNNs), both in isolation and together. The goal of the experiments is to analyze combinations of hyperparameters mentioned in the dropout paper [28] and the batch normalization paper [7], separately (only dropout or only batch normalization) or in combination (both dropout and batch normalization).

Overall, our experiments and analysis resulted in the following major findings. For MLP networks, the empirical study showed that:

- Training with dropout and batch normalization is slower, as expected. However, batch normalization turned out to be significantly slower, increasing training time by over 80%.
- A non-adaptive optimizer (SGD) can outperform an adaptive optimizer (RMSProp). But to do so it required experimentation with other hyperparameters (learning rate, momentum, max-norm), consuming more training time. As a general guideline tests should start with an adaptive optimizer because it will perform better with default parameters. Switching to a non-adaptive optimizer should be reserved for a later phase, when other major decisions have been made (e.g. validate the dataset, explore different network architectures, etc.).
- Test (prediction) time of a network trained with batch normalization is approximately 30% higher. This may be a factor for some applications because it also results in more energy consumption, draining batteries faster. This was an unexpected result of the tests and needs further validation.

For CNNs, the empirical study showed that:

- Adding batch normalization improved accuracy without other observable side effects. Since it can be added without major structural changes to the network architecture, adding batch normalization should be one of the first steps taken to optimize a CNN.
- Increasing the learning rate, as recommended in the batch normalization paper [7] improves accuracy by 2% to 3%. Because this is a simple step to take, it should be done in the initial optimization steps, before investing time in more complex optimizations.

– Adding dropout reduced accuracy significantly. This could be a deficiency of the experiments conducted here because other sources reported improvements when dropout was used. At a minimum, it is a cautionary sign that using dropout in CNNs require careful consideration. As a practical suggestion, one should consider removing all dropout layers from the network, re-validate and confirm that dropout does not deteriorate the peformance.

Acknowledgements This research is sponsored by the US National Science Foundation (NSF) through Grants IIS-1763452 and CNS-1828181.

References

1. Bengio Y (2012) Practical recommendations for gradient-based training of deep architectures
2. Brock A, Lim T, Ritchie JM, Weston N (2017) Freezeout: accelerate training by progressively freezing layers. arXiv:1706.04983
3. Cortes C, Vapnik V (1995) Mach Learning, 273–297
4. Deng L, Hinton G, Kingsbury B (2013) New types of deep neural network learning for speech recognition and related applications: an overview. https://doi.org/10.1109/icassp.2013.6639344
5. Goodfellow IJ, Bengio Y, Courville AC (2016) Deep learning. Adaptive Computation and Machine Learning. MIT Press, Cambridge. http://www.deeplearningbook.org/
6. Hinz T, Navarro-Guerrero N, Magg S, Wermter S (2018) Speeding up the Hyperparameter Optimization of Deep Convolutional Neural Networks. International Journal of Computation Intelligence and Applications 17(2):1850008. https://doi.org/10.1142/s1469026818500086
7. Ioffe S, Szegedy C (2015) Batch normalization: accelerating deep network training by reducing internal covariate shift
8. KerasTeam (2016) Using test data as validation data during training. https://github.com/keras-team/keras/issues/1753
9. Kohler JM, Daneshmand H, Lucchi A, Zhou M, Neymeyr K, Hofmann T (2018) Towards a theoretical understanding of batch normalization, arXiv:1805.10694
10. Krizhevsky A (2009) Learning multiple layers of features from tiny images. https://www.cs.toronto.edu/kriz/learning-features-2009-TR.pdf
11. Krizhevsky A, Sutskever I, Hinton GE (2012) In: Bartlett PL, Pereira FCN, Burges CJC, Bottou L, Weinberger KQ (eds) Advances in neural information processing systems 25: 26th annual conference on neural information processing systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, USA, pp 1106–1114. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks
12. Krizhevsky A, Nair V, Hinton G (2019) The cifar-10 dataset. https://www.cs.toronto.edu/kriz/cifar.html
13. Krizhevsky A, Sutskever I, Hinton GE (2017) Imagenet classification with deep convolutional neural networks 60: 84. https://doi.org/10.1145/3065386
14. Längkvist M, Karlsson L, Loutfi A (2014) A review of unsupervised feature learning and deep learning for time-series modeling 42: 11. https://doi.org/10.1016/j.patrec.2014.01.008
15. LeCun Y (1999) The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/
16. LeCun Y, Bengio Y, Hinton G (2015) Deep Learning 521:436. https://doi.org/10.1038/nature14539
17. Li X, Chen S, Hu X, Yang J (2018) Understanding the disharmony between dropout and batch normalization by variance shift
18. Lipton ZC, Berkowitz J, Elkan C (2015) A critical review of recurrent neural networks for sequence learning
19. Loh WY (2014) Fifty years of classification and regression trees 82: 329. https://doi.org/10.1111/insr.12016
20. Luo P, Wang X, Shao W, Peng Z (2018) Towards understanding regularization in batch normalization
21. Mishkin D, Sergievskiy N, Matas J (2017) Systematic evaluation of convolution neural network advances on the imagenet. Comput. Vision Image Understanding. https://doi.org/10.1016/j.cviu.2017.05.007. http://www.sciencedirect.com/science/article/pii/S1077314217300814

22. Morgan N, Bourlard H (1989) In: Touretzky DS (ed) Advances in neural information processing systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]. Morgan Kaufmann, pp 630–637. http://papers.nips.cc/paper/275-generalization-and-parameter-estimation-in-feedforward-nets-some-experiments
23. Nair V, Hinton GE (2010) In: Fürnkranz J, Joachims T (eds) Proceedings of the 27th international conference on machine learning (ICML-10), June 21-24, 2010, Haifa, Israel. Omnipress, pp 807–814. http://www.icml2010.org/papers/432.pdf
24. Perez L, Wang J (2017) The effectiveness of data augmentation in image classification using deep learning
25. Ruder S (2016) An overview of gradient descent optimization algorithms
26. Rumelhart DE, Hinton G, Williams RJ (1985) Learning internal representations by error propagation. https://doi.org/10.21236/ada164453
27. Smith LN, A disciplined approach to neural network hyper-parameters: part 1 – learning rate batchsize momentum and weight decay (2018)
28. Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R (2014) Dropout: a simple way to prevent neural networks from overfitting. J Mach Learn Res 15(1):1929. http://dl.acm.org/citation.cfm?id=2627435.2670313
29. Team K (2019) https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py
30. Tieleman T, Hinton G (2012) Lecture 6.5—Rmsprop: divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning
31. University S (2018) Stanford university cs231n: convolutional neural networks for visual recognition. http://cs231n.github.io/classification/#nn
32. Wang X, Gao L, Song J, Shen H, Beyond frame-level CNN (2017) Saliency-aware 3-D CNN with lstm for video action recognition. IEEE Signal Process Lett 24(4):510. https://doi.org/10.1109/LSP.2016.2611485
33. Wang X, Gao L, Wang P, Sun X, Liu X (2018) Two-stream 3-D convnet fusion for action recognition in videos with arbitrary size and length. IEEE Trans Multimed 20(3):634. https://doi.org/10.1109/TMM.2017.2749159
34. Zhu X (2005) Semi-supervised learning literature survey. Tech. Rep. 1530, Computer Sciences, University of Wisconsin-Madison

Christian Garbin has held software development, line and staff management positions in telecommunications and financial industries, working with highly-available, high-performance servers. He is currently a senior architect at a major IT company. He graduated from Florida Atlantic University (FAU) with a B.S. degree in Computer Science in 2003. After following the development of artificial intelligence and machine learning from a distance since then, he returned to FAU in 2018 to pursue his M.Sc. degree in Computer Science, focusing on machine learning.

Xingquan Zhu is a Professor in the Department of Computer & Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL, USA. He received PhD degree in computer science from Fudan University, Shanghai, China. His research interests include data mining, machine learning, and multimedia systems. Since 2000, he has authored or co-authored over 250 refereed journal and conference papers in these areas, including three Best Paper Awards and one Best Student Award. Dr. Zhu is an Associate Editor of the IEEE Transactions on Knowledge and Data Engineering (2008-2012, and 2014 - date), and an Associate Editor of the ACM Transactions on Knowledge Discvoery from Data (2017- date).

Oge Marques is *Professor of Computer Science and Engineering* at the College of Engineering and Computer Science at Florida Atlantic University (FAU) (Boca Raton, Florida, USA). He is *Tau Beta Pi Eminent Engineer, ACM Distinguished Speaker*, and *Leshner Fellow* of the American Association for the Advancement of Science (AAAS). Dr. Marques is the author of more than 100 publications in the area of *intelligent processing of visual information* — which combines the fields of image processing, computer vision, image retrieval, machine learning, serious games, and human visual perception —, including the textbook "Practical Image and Video Processing Using MATLAB" (Wiley-IEEE Press). Professor Marques is Senior Member of both the IEEE (Institute of Electrical and Electronics Engineers) and the ACM (Association for Computing Machinery) and member of the honor societies of Sigma Xi, Phi Kappa Phi and Upsilon Pi Epsilon. He has more than 30 years of teaching and research experience in different countries (USA, Austria, Brazil, France, India, Spain, Serbia, and the Netherlands).

Affiliations

Christian Garbin[1] **· Xingquan Zhu**[1] **· Oge Marques**[1]

Christian Garbin
cgarbin@fau.edu

Oge Marques
omarques@fau.edu

[1] Department of Computer & Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL 33431, USA