Comparison of Dropout vs. Batch Normalization accuracy and efficiency

Christian Garbin
College of Engineering and Computer Science
Florida Atlantic University
Boca Raton, Florida, USA
CAP-6619 Deep Learning, Fall 2018
Term Project

***Abstract*—**
Overfitting and long training times are fundamental problems in machine learning. Dropout [Sri+14] significantly improved overfitting, while Batch Normalization [IS15] significantly reduced training time.

These two techniques overlap. Using them is not as simple as adding Dropout layers to improve overfitting and adding Batch Normalization layers to speed up training. Although there are guidelines to use them, there are no well-defined rules that can be applied in all cases, for all network configurations and types of input data.

Getting them to work well in practice requires informed trial and error of different combinations of hyperparameters. Because there is a large number of combinations of hyperparameters, contradicting results have sometimes been published. For example, the paper introducing Batch Normalization [IS15] recommends removing Dropout because Batch Normalization has enough of a regularization effect in the network, while subsequent work by [HG16] and [Li+18] showed that Dropout can indeed be used together with Batch Normalization to improve results.

Compounding all these observations is the fact that, while there are hypotheses to explain how Dropout and Batch Normalization improve accuracy and training time, there are no definitive explanations for their inner works [NS18] [Bjo+18] [San+18]. Resulting again in the need to experiment with different configurations and sample data to validate empirically the theoretical assumptions.

We conducted experiments to create guidelines for using Dropout and Batch Normalization. The experiments were performed in image classifications tasks (MNIST and CIFAR-10) using multilayer perceptron networks (MLPs) and convolutional neural networks (CNNs).

The goal of these experiments is to analyze the accuracy of the different networks and hyperparameters, their usage of system resources during training and during test (prediction) time. This analysis will be used to derive some general guidelines to use and fine-tune these network configurations.

***Index Terms*—machine learning, overfitting, dropout, batch normalization, optimization, regularization**

I. Introduction

A. Generalization and Dropout

The fundamental problem in machine learning is *generalization*, the accuracy of a trained model when it evaluates previously unseen data [GBC16]. *Overfitting* happens when the model performs well on the training data, but performs poorly on new data, i.e. the model has low training error and high test error. *Regularization* is a set of techniques used to reduce overfitting. Over the years many such techniques have been developed. Some of them act on the gradient descent optimization algorithms [Rud16], others act on the input data, artificially creating new training data [PW17].

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 [GBC16]. 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.

Dropout [Sri+14] simulates the model ensembling without creating multiple networks. It has been widely adopted since its publication, in part because it doesn't 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 [Sri+14] 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[Sri+14]. Then, after adding Dropout, we need to train models with different combinations of hyperparameters that affect its behavior, further increasing training time.

Another, more significant complicating factor is that it was tested with the standard stochastic descent gradient (SGD) optimizer (as it's done in most papers [Rud16]). Most networks today use adaptive optimizers, e.g. RMSProp [TH12], commonly used in Keras examples. Some of the recommendations in the paper, for example learning rates and weight decay values, do not necessarily apply when an adaptive optimizer is used.

This leads us to techniques to help reduce the training time by helping the models converge faster. One such technique is *Batch Normalization* [IS15].

B. Training time and Batch Normalization

Before *Batch Normalization* [IS15] 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 (the original paper 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 different combinations of hyperparameters needed to optimize the use of Dropout layers (it would not speed up overall 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 [IS15], in [Luo+18] and [Koh+18].

C. The intersection of Dropout and Batch Normalization

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 three-part question:

1) 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?
2) Should it use both Dropout and Batch Normalization? Despite the claim in [IS15], other experiments showed that they can be used together to improve a network [Li+18].
3) 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?

And then, if Dropout should be used, there is one more question to answer:

1) What values should be used for the hyperparameters that affect Dropout (learning rate, weight decay, momentum, optimizer, etc.)?

We conducted experiments to derive some guidelines for using Dropout and Batch Normalization. The experiments were performed in image classifications tasks (MNIST and CIFAR-10) using multilayer perceptron networks (MLP) and convolutional neural networks (CNN).

The experiments tested 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 [Sri+14] and the Batch Normalization paper [IS15].

II. Related Work

[Ben12] is a practical guide to optimize hyperparameters. It emphasizes the need to choose a good learning rate as the main decision when optimizing networks.

[Rud16] summarizes the different optimizers and documents guidelines to choose one. It 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.

[Sri+14] introduced Dropout and described how specific hyperparameters (learning rate, momentum, max-norm, etc.) affect its behavior. Appendix A has the recommendation 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 results in a large matrix of combinations to try during training.

[IS15] introduced Batch Normalization. It shows that Batch Normalization enables higher learning rates, but doesn't 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 newer articles (some of them are referenced below). These newer investigations resulted in recommendations to use Dropout together with Batch Normalization in some scenarios.

[Li+18] 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.

[Luo+18] shows that using Dropout after a Batch Normalization layer is beneficial if the batch size is large (256 sample or more) and a small (0.125) dropout rate is used (similarly to the findings in [Li+18] in this respect). It also hypothesizes that Dropout did not work in [IS15] because it was tested with a small batch size.

III. Technical Details

A. Datasets

Tests were performed on two image classification datasets, MNIST [LeC99] and CIFAR-10 [Kri09].

MNIST

MNIST [LeC99] is a set of 60,000 training and 10,000 test images of handwritten digits from 0 to 9. All images are 28x28 pixels in grayscale, with the digits centered in the image.

Although it is sometimes mentioned as the "postal (ZIP) code" set, it was not created from actually posted letters. 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 1 shows a sample of the MNIST dataset.

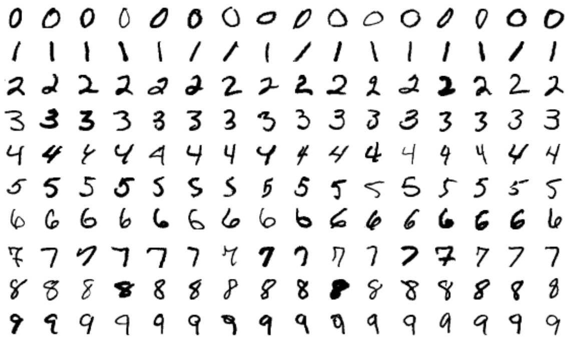


Fig. 1: MNIST sample digits [Ste]

CIFAR-10

CIFAR-10 [Kri09] is a set of 50,000 training and 10,000 test images. All images are 32x32 pixels in color (RGB channels).

It contains 10 classes of images. The training set has 5,000 images of each class. The test set has 1,000 images of 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 2 shows a sample of the CIFAR-10 dataset.

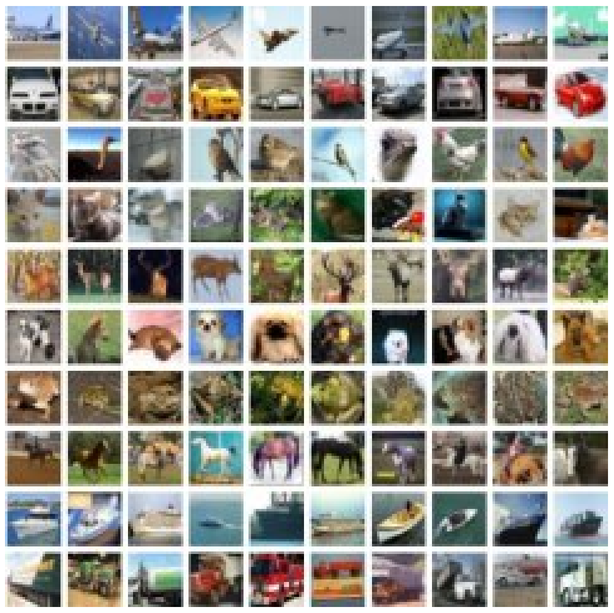


Fig. 2: CIFAR-10 sample pictures [Kar]

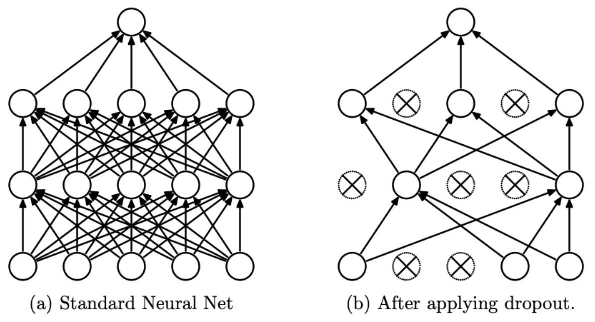


Fig. 3: How dropout works: randomly removing units from a fully connected network (left) to create a sub-model (right). Original picture from [Sri+14].

B. Dropout

Dropout [Sri+14] 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. See figure 3.

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*.[1] 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:

[1]Note that the Keras API use 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 compensates 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.

When applying these rules, note that the paper seems to have tested these guidelines with a regular, non-adaptive SGD optimizer. They may not apply exactly as described for adaptive optimizers.

C. 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 [IS15] was created to resolve this variability and speed up learning.

Normalizing the values of each sample before presenting it to the network's input layer was already a well-known technique. Batch Normalization goes one step further and normalizes the input in every layer of the network, not only the input layer. The normalization is computed for each batch. This normalization allows the use of higher learning rates during training (although the paper 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:

- 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 dropout rates.

D. Network Configurations

In this report we tested multilayer perceptron networks (MLPs) and convolutional neural networks (CNNs) with different hyperparameter configurations.

Multilayer Perceptron Network Tests

Multilayer perceptron networks are networks composed of several fully connected layers. An example is shown in figure 4.

The MNIST dataset was used to test the MLP networks and hyperparameters combinations. MNIST was chosen for this test because the original Dropout paper [Sri+14] also used MNIST in their MLP tests and formulated their guidelines for hyperparameters based on that. The Dropout paper 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.

Convolutional Neural Network Tests

Convolutional neural networks are networks composed of several convolution and max-pooling layers. An example is shown in figure 9.

CNN was used to test the CIFAR-10 dataset in the Dropout paper [Sri+14]. The Batch Normalization paper [IS15] 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.

E. Data Collected

For each experiment, the following measurements were collected.

Training CPU time

- What it measures: how much time was used across all CPUs and GPUs to run all training epochs.
- Goal: measure how much system resources a network configuration uses during the training phase.
- Why it was measured: 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.
- How it was measured: with the Python `time` package, measuring the time to it took to execute Keras' `model.fit(...)` function for the network configuration.
- Note: in 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.

Test CPU time

- What it measures: how much time was used across all CPUs and GPUs to evaluate the trained network using the test set.
- Goal: measure how much system resources the trained network uses when it is evaluating samples.
- Why it was measured: to understand how efficient (or not) the trained network is on the end-users' systems. Lower variables 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. phones).
- How it was measured: with the Python `time` package, measuring the time to it took to execute Keras' `model.evaluate(...)` function on the trained network.
- Note: similarly to the training CPU time, this is not the same as clock (wall) time. See the note for that item for more details.

Number of parameters in the model

- What it measures: the number of parameters the model has.
- Goal: measure how much memory the model uses.
- Why it was measured: to have a second parameter 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.
- How it was measured: with 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).

Training and validation loss and accuracy for each epoch

- What it measures: the training loss shows if 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.
- Goal: measure how fast the network converges and if the network will perform well on unseen data.
- Why it was measured: to have a measurement for training efficiency among the models tested. 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.
- How it was measured: 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 [MB89] or annealing the learning rate), this is an acceptable practice. [Ker16] has a similar discussion for this usage pattern in the Keras examples.

IV. Experiments

This section describes the setup used for the experiments and the results collected from them.

The first subsection describes the experiment setup. The second subsection describes the experiments and their results using multilayer perceptron networks (MLP). The third section does the same for convolutional neural networks (CNNs).

A. Experimental Settings

Test environment

Tests were executed on a Google Cloud virtual machine with the following specification:

- 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 was essential to explore the combination of hyperparameters described in the following sections. Training was 30x faster in that environment, compared to a relatively high-performing machine without a CPU (Intel i7 2.9 GHz, 16 GB RAM).

Tools

The following tools were used in the experiments:

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

B. Results for Multilayer Perceptron Networks

This section describes the experiments performed with multilayer perceptron networks and reviews the results. First the details of the experiments are described, then the results are analyzed.

MLPs are the networks that have only dense layers (no convolution or other operation is applied).

Baseline

Since the goal of these 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.

Configurations Tested

Three MLP configurations were tested using the MNIST dataset:

1) Baseline: a standard MLP, with dense layers connected to each other without Dropout or Batch Normalization. This configuration is used as the baseline.
2) Dropout: added Dropout layers to the standard network, following the guidelines in the Dropout paper [Sri+14].
3) Batch Normalization: added Batch Normalization layers to the standard network, following the guidelines in the Batch Normalization paper [IS15].

Hyperparameters Tested

Each of these networks was tested with a combination of these hyperparameters:

- Hidden layers: 2, 3 and 4 hidden layers. These numbers were chosen because they were described in the Dropout paper [Sri+14].
- Units in each hidden layer: 1,024 and 2,048 units. These numbers were chosen for the same reason as above.
- Batch size: 128 samples.
- Epochs, 5, 20 and 50 epochs.
- Optimizer: a non-adaptive stochastic gradient descent (SGD) optimizer and RMSProp [TH12]. The SGD optimizer was chosen because there are indications that most published results use such an optimizer [Rud16]. The RMSProp optimizer was chosen to compare the performance of SGD with an adaptive optimizer.
- Activation function: ReLU [NH10] in all cases. It was chosen because the Dropout paper [Sri+14] and the Batch Normalization paper [IS15] 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.

Standard MLP Network Tests - Baseline

The standard MLP, without Dropout or Batch Normalization was used as the baseline.

Network structure

A standard MLP network in this context is a network made of only dense layers, without any Dropout or Batch Normalization layer.

Figure 4 shows an example of a standard MLP network used in the tests.

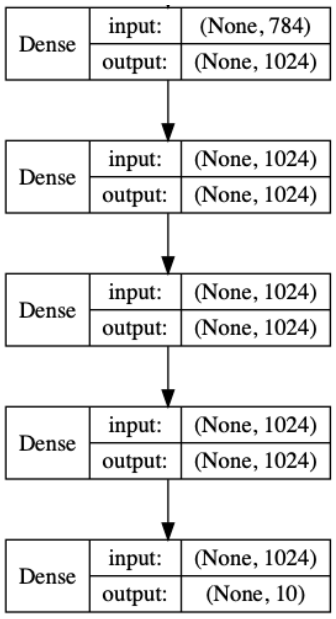


Fig. 4: Sample standard MLP network used in the tests

Hyperparameters tested

- 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 a 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 a 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.

Best results

The top 10 results of these tests are listed in table I on page 7.

TABLE I: Top 10 test accuracy for the standard MLP network (no Dropout or Batch Normalization)

Optimizer	Test accuracy	Hidden layers	Units per layer	Epochs	Batch size	Learning rate	Decay	SGD moment.	max-norm	Parameters count	Training time (s)	Test time (s)
SGD	0.9879	2	2048	50	128	0.1	0	0.95	none	5,824,522	137	0.609
SGD	0.9878	3	2048	50	128	0.1	0	0.95	2	10,020,874	165	0.677
SGD	0.9869	3	2048	50	128	0.1	0	0.95	none	10,020,874	167	0.650
SGD	0.9866	4	2048	50	128	0.1	0	0.95	none	14,217,226	191	0.696
SGD	0.9865	2	2048	5	128	0.1	0.001	0.95	none	5,824,522	17	0.704
SGD	0.9864	4	2048	50	128	0.1	0	0.95	2	14,217,226	191	0.698
SGD	0.9863	2	1024	50	128	0.1	0	0.95	none	1,863,690	125	0.584
SGD	0.9861	3	2048	50	128	0.1	0.001	0.95	2	10,020,874	170	0.787
RMSprop	0.9860	2	2048	20	128	0.001	0.00001	0	2	5,824,522	69	0.644
SGD	0.9859	3	1024	50	128	0.1	0	0.95	2	2,913,290	135	0.583

MLP Network with Dropout

Network structure

The MLP Dropout network was modeled after the Dropout paper [Sri+14]. The significant changes compared to the standard MLP network are:

- 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 5 shows an example of a Dropout network used in the tests.

Hyperparameters tested

- Dropout rate for the input layer: 0.1, as recommended in [Sri+14].
- Dropout rate for hidden layers: 0.5, the high end of the rate recommended in [Sri+14].
- Learning rates for SGD: the default Keras rate 0.01 and 10x the default rate (0.1) as recommended in [Sri+14].
- 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 [Sri+14].
- 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 a 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 a 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 [Sri+14]: 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 [Sri+14]: 2 and 3.

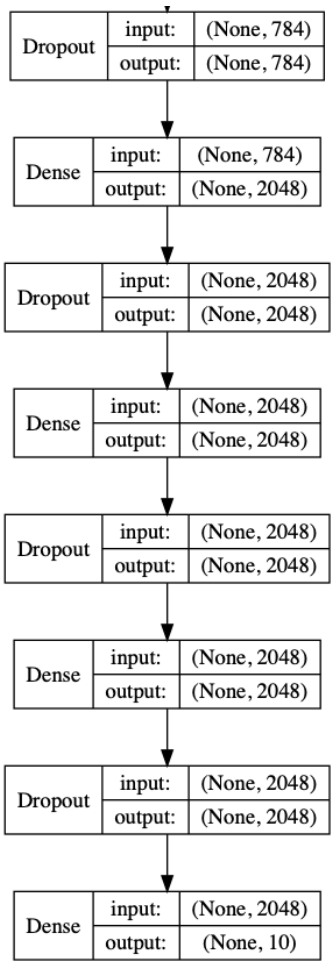


Fig. 5: Sample Dropout MLP network used in the tests

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.

Best results

The top 10 results of these tests are listed in table II on page 8.

MLP Network with Batch Normalization

Network structure

TABLE II: Top 10 test accuracy for the Dropout MLP network

Optimizer	Test accuracy	Hidden layers	Units per layer	Epochs	Batch size	Dropout rate input	Dropout rate hidden	Learning rate	Decay	SGD moment.	max-norm	Parameters count	Training time (s)	Test time (s)
SGD	0.9881	2	2048	50	128	0.1	0.5	0.01	0.001	0.99	3	20,037,642	244	0.701
SGD	0.9879	3	2048	50	128	0.1	0.5	0.01	0	0.95	3	36,818,954	335	0.779
SGD	0.9879	4	2048	50	128	0.1	0.5	0.01	0.001	0.99	2	53,600,266	429	0.808
SGD	0.9876	3	1024	50	128	0.1	0.5	0.01	0	0.95	none	10,020,874	180	0.725
SGD	0.9876	4	2048	50	128	0.1	0.5	0.01	0.001	0.99	3	53,600,266	429	0.826
SGD	0.9875	2	1024	50	128	0.1	0.5	0.01	0.001	0.99	none	5,824,522	161	0.662
SGD	0.9875	2	2048	50	128	0.1	0.5	0.1	0	0	3	20,037,642	241	0.730
SGD	0.9875	4	1024	50	128	0.1	0.5	0.01	0	0.95	none	14,217,226	204	0.671
SGD	0.9874	2	2048	50	128	0.1	0.5	0.01	0	0.95	2	20,037,642	242	0.722
SGD	0.9874	3	2048	50	128	0.1	0.5	0.01	0.001	0.99	3	36,818,954	340	0.771

The MLP Batch Normalization network was modeled after the Batch Normalization paper [IS15]. The significant change compared to the standard MLP network is the addition of a Batch Normalization layer after the dense layers.

Figure 6 shows an example of a Batch Normalization network used in the tests.

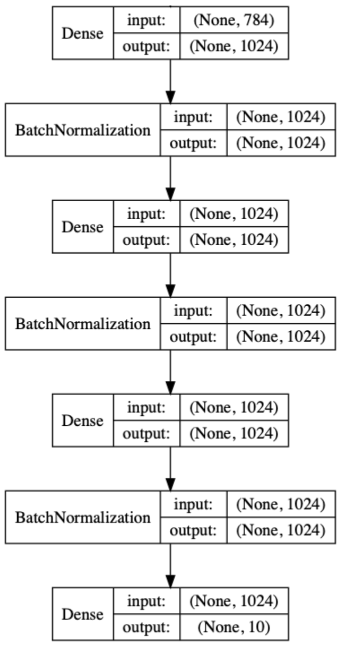

Dense | input: (None, 784) | output: (None, 1024)
BatchNormalization | input: (None, 1024) | output: (None, 1024)
Dense | input: (None, 1024) | output: (None, 1024)
BatchNormalization | input: (None, 1024) | output: (None, 1024)
Dense | input: (None, 1024) | output: (None, 1024)
BatchNormalization | input: (None, 1024) | output: (None, 1024)
Dense | input: (None, 1024) | output: (None, 10)

Fig. 6: Sample Batch Normalization MLP network used in the tests

Hyperparameters tested

- Learning rates for SGD: the default Keras rate 0.01 and 10x the default rate (0.1). [IS15] 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, [IS15] 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 a 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 others tests where a 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 [Sri+14]: 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 [Sri+14]: 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.

Best results

The top 10 results of these tests are listed in table III on page 9.

Analysis of the results

Accuracy

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

Figure 7 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.

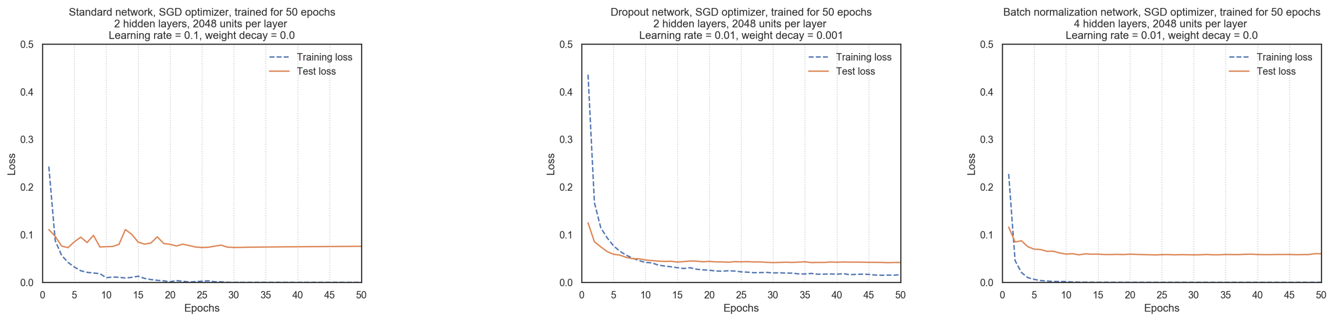
The Batch Normalization network reaches its lower value much earlier than other networks, as expected, given that one of its purposes is to accelerate learning [IS15]. This can be taken advantage of to shorten training times when a very high accuracy is not needed. Early stopping would stop training with the Batch Normalization sooner than with the other networks.

Performance: 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

TABLE III: Top 10 test accuracy for the Batch Normalization MLP network

Optimizer	Test accuracy	Hidden layers	Units per layer	Epochs	Batch size	Learning rate	Decay	SGD moment.	Parameters count	Training time (s)	Test time (s)
SGD	0.9868	4	2048	50	128	0.01	0	0.95	14,243,850	394	0.933
SGD	0.9867	2	1024	50	128	0.1	0.0001	0.95	1,870,858	241	0.765
RMSprop	0.9867	4	2048	50	128	0.001	0.0001	0	14,243,850	439	0.927
SGD	0.9865	3	1024	50	128	0.1	0.0001	0.95	2,923,530	293	0.857
SGD	0.9864	2	2048	50	128	0.1	0.0001	0.95	5,838,858	255	0.795
SGD	0.9864	4	2048	50	128	0.01	0.0001	0.95	14,243,850	388	0.901
RMSprop	0.9862	3	1024	20	128	0.001	0.0001	0	2,923,530	125	0.868
SGD	0.9860	2	2048	50	128	0.01	0	0.95	5,838,858	255	0.780
SGD	0.9860	4	2048	20	128	0.01	0	0.95	14,243,850	161	0.927
SGD	0.9859	2	2048	20	128	0.01	0	0.95	5,838,858	101	0.764



(a) Loss for standard MLP

(b) Loss for Dropout MLP

(c) Loss for Batch Normalization MLP

Fig. 7: Loss graphs for the top MLP network in each test

layers was extracted into table IV.[2]

It shows these behaviors of the different networks:

- Training CPU time: Dropout increases training time by approximately 17%. Batch Normalization increases training time by approximately 86%. As shown in figure 8, 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. 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 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).

[2]Note that the Dropout network is listed in the top 10 results as "1024 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 1024 units in a layer to effectively have 2048 units in that layer.

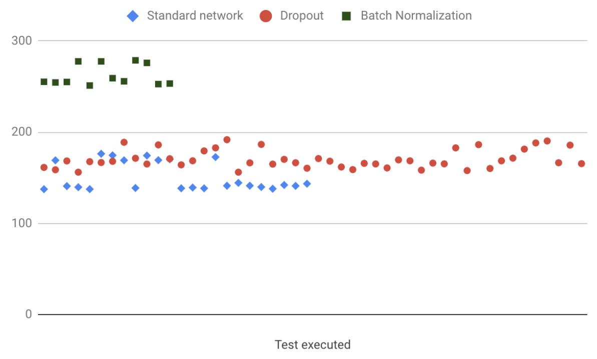


Fig. 8: Training time in seconds for MLP standard, Dropout and Batch Normalization networks using two hidden layers, 2048 units in each layer, trained for 50 epochs. Showing all combination of hyperparameters (learning rate, decay, momentum, etc.). Each dot represents a test executed with the different hyperparameters.

TABLE IV: Performance evaluation comparing networks with two hidden layers and 2048 hidden units (note that Dropout is divided by 0.5, the dropout rate). All tests 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

Effects of some combinations of hyperparameters

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

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 test time.

Recommendations based on the results

- Use Batch Normalization if the convergence time is more important than absolute accuracy. Together with early stopping, it could significantly reduce training time.
- But be aware of Batch Normalization's training time increase. 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. 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 report, these are some improvements that could be done in the experiments and data collection process:

- Batch Normalization test time validation: tests showed that 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 [Sri+14] 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 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.

C. Results for Convolutional Neural Networks

This section describes the experiments performed with convolutional neural networks and reviews the results. First the details of the experiments are described, then the results are analyzed.

CNNs are the networks that have convolution and max-pooling layers as their main feature, followed by flattening and dense layers before an output layer.

Baseline

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 the text.

Configurations Tested

The following CNN configurations were tested:

1) Baseline: a standard CNN, with convolution and max-pooling layers, without using any Dropout or Batch Normalization layers. This configuration is used as the baseline. This CNN was based on the official Keras example [ref] and similar to the CNN used in [Sri+14] for the Google Street View House Numbers tests. Figure 9 shows this network configuration.
2) 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 10 shows this network configuration.
3) 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 11 shows this network configuration.
4) Batch Normalization: starting with the standard CNN, added Batch Normalization layers between the convolution and the max-pooling layers. Although [IS15] adds Batch Normalization before the non-linearity, subsequent experiments reported that adding Batch Normalization after the non-linearity improves accuracy [Mis16]. Because of such reports, tests were executed with the Batch Normalization layer after the non-linearity layer. Figure 12 shows this network configuration.
5) Dropout + Batch Normalization: a combination of the Dropout and Batch Normalization networks. Figure 13 shows this network configuration.

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.

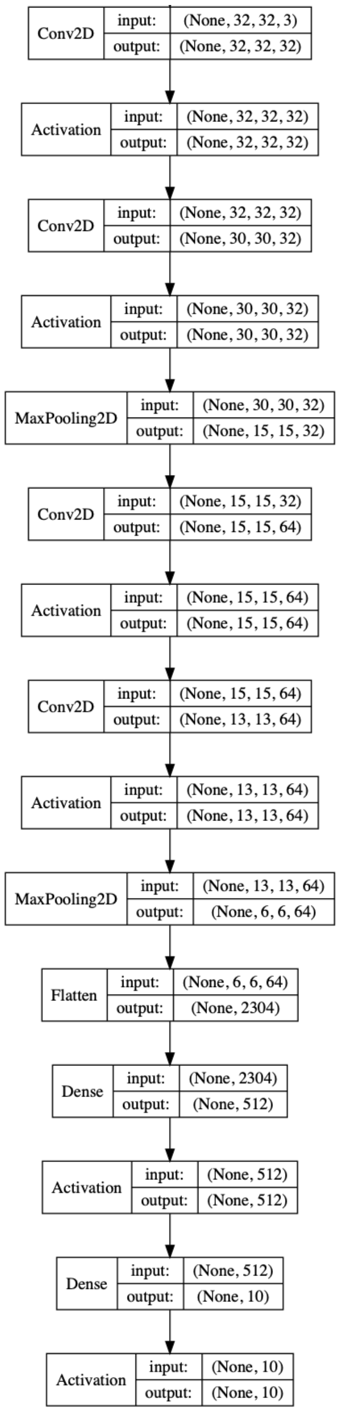

Conv2D input: (None, 32, 32, 3) output: (None, 32, 32, 32)
Activation input: (None, 32, 32, 32) output: (None, 32, 32, 32)
Conv2D input: (None, 32, 32, 32) output: (None, 30, 30, 32)
Activation input: (None, 30, 30, 32) output: (None, 30, 30, 32)
MaxPooling2D input: (None, 30, 30, 32) output: (None, 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, 512)
Activation input: (None, 512) output: (None, 512)
Dense input: (None, 512) output: (None, 10)
Activation input: (None, 10) output: (None, 10)

Fig. 9: Standard CNN network used in the tests

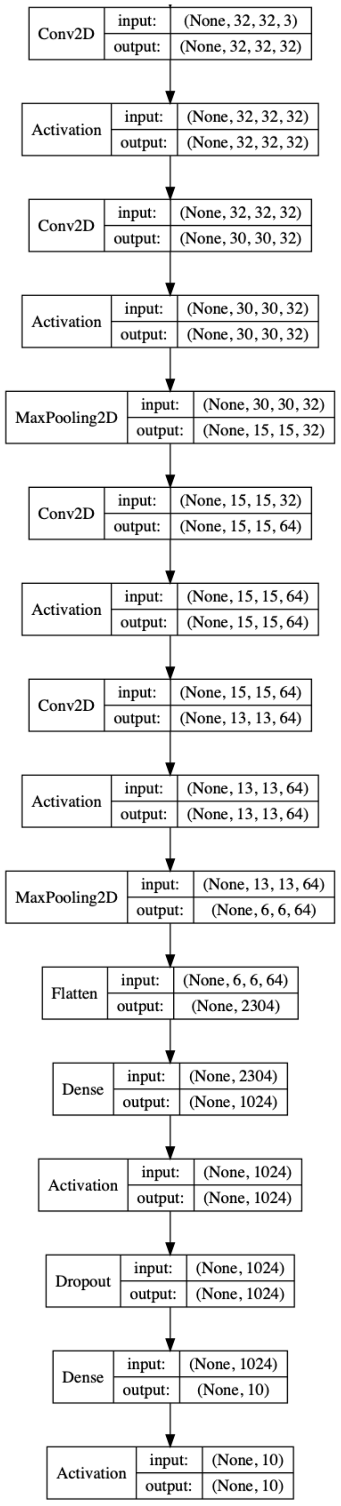

Conv2D input: (None, 32, 32, 3) output: (None, 32, 32, 32)
Activation input: (None, 32, 32, 32) output: (None, 32, 32, 32)
Conv2D input: (None, 32, 32, 32) output: (None, 30, 30, 32)
Activation input: (None, 30, 30, 32) output: (None, 30, 30, 32)
MaxPooling2D input: (None, 30, 30, 32) output: (None, 15, 15, 32)
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. 10: CNN network with Dropout after the dense layer used in the tests

Hyperparameters Tested

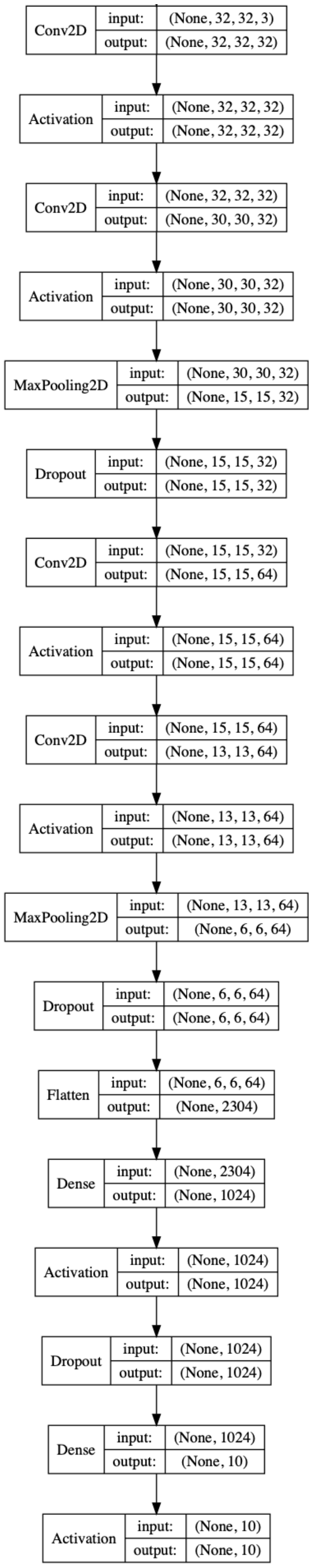

Conv2D input: (None, 32, 32, 3) output: (None, 32, 32, 32)
Activation input: (None, 32, 32, 32) output: (None, 32, 32, 32)
Conv2D input: (None, 32, 32, 32) output: (None, 30, 30, 32)
Activation input: (None, 30, 30, 32) output: (None, 30, 30, 32)
MaxPooling2D input: (None, 30, 30, 32) output: (None, 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. 11: CNN network with Dropout in all layers used in the tests

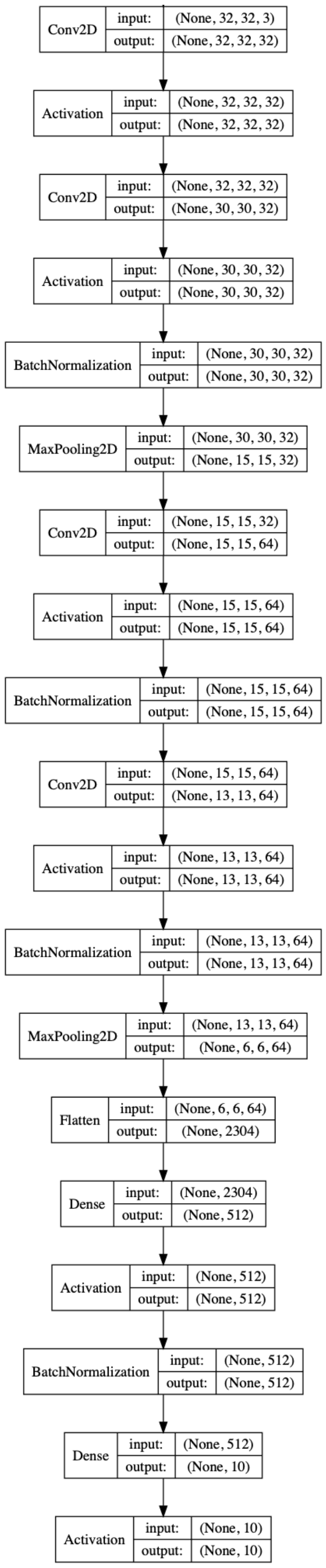

Conv2D input: (None, 32, 32, 3) output: (None, 32, 32, 32)
Activation input: (None, 32, 32, 32) output: (None, 32, 32, 32)
Conv2D input: (None, 32, 32, 32) output: (None, 30, 30, 32)
Activation input: (None, 30, 30, 32) output: (None, 30, 30, 32)
BatchNormalization input: (None, 30, 30, 32) output: (None, 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. 12: CNN network with Batch Normalization in all layers used in the tests

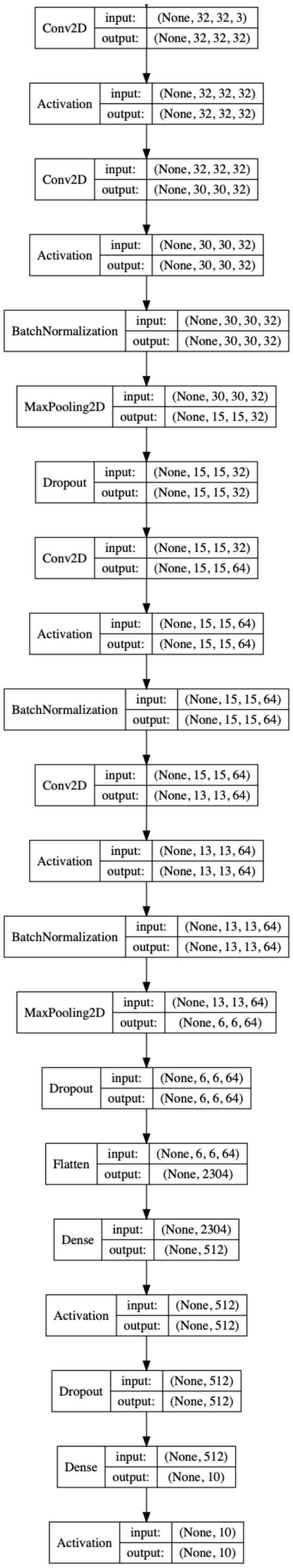

Conv2D | input: (None, 32, 32, 3) | output: (None, 32, 32, 32)
Activation | input: (None, 32, 32, 32) | output: (None, 32, 32, 32)
Conv2D | input: (None, 32, 32, 32) | output: (None, 30, 30, 32)
Activation | input: (None, 30, 30, 32) | output: (None, 30, 30, 32)
BatchNormalization | input: (None, 30, 30, 32) | output: (None, 30, 30, 32)
MaxPooling2D | input: (None, 30, 30, 32) | output: (None, 15, 15, 32)
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)

Fig. 13: CNN network with Batch Normalization and Dropout in all layers used in the tests

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 [Sri+14]); Batch Normalization CNNs were tested with 0.0005, a higher rate as recommended in general terms (without providing specific values) in [IS15]. 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 [Sri+14] 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 [Sri+14].
- Activation function: ReLU [NH10] in all cases. [KSH12] showed that ReLU speeds up the training significantly.

Analysis of the results

Results from the tests are shown in table V.

Accuracy

Adding Batch Normalization significantly improves the accuracy.

Dropout, on the other hand, was always detrimental to accuracy (as used in these experiments - see recommendations for further tests).

Performance: training CPU time, 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%.

Effects of learning rate

Increasing the learning yields better accuracy only when Batch Normalization is used. In all other cases it is detrimental to accuracy.

TABLE V: Results from the CNN tests

Test	Network	Test accuracy	Learning rate	Units in dense layer	Epochs	Parameters count	Training time (s)
1	Standard, no Dropout or Batch Normalization	0.3226	0.001	1024	50	2,436,138	2920
2	Standard, no Dropout or Batch Normalization	0.6945	0.0005	512	50	1,250,858	2878
3	Standard, no Dropout or Batch Normalization	0.8041	0.0001	1024	50	2,436,138	2884
4	Standard, no Dropout or Batch Normalization	0.5717	0.001	512	50	1,250,858	2872
5	Standard, no Dropout or Batch Normalization	0.8021	0.0001	512	50	1,250,858	2866
6	Dropout only for dense layer	0.7426	0.0001	512	50	1,250,858	2881
7	Dropout only for dense layer	0.1001	0.001	1024	50	2,436,138	2894
8	Dropout in all layers	0.7575	0.0001	512	50	1,250,858	2904
9	Dropout in all layers	0.3226	0.001	1024	50	2,436,138	2920
10	Batch Normalization	0.8447	0.001	512	50	1,253,546	3185
11	Batch Normalization	0.8266	0.0001	512	50	1,253,546	3174
12	Batch Normalization	0.8395	0.0005	512	50	1,253,546	3189
13	Dropout + Batch Normalization	0.8002	0.0001	512	50	1,251,498	3092
14	Dropout + Batch Normalization	0.8087	0.001	1024	50	2,436,778	3094
15	Dropout + Batch Normalization	0.7774	0.0005	512	50	1,251,498	3083

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 a much lower accuracy.

Recommendations based on the results

- 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. 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. [IS15] recommends to increase it and it does make a significant difference. Tests 10 and 11 in table V shows that increasing it by 10x improves accuracy by 3%, without any other change in the test parameters. However, adding it for any other configuration, including Dropout, reduces the accuracy.

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, [Li+18] reported that Dropout can be used to improve accuracy. Investigating the reasons for the low accuracy 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. Further tests should explore other dropout rates. Two places to start are increasing the batch size, as recommended in [Luo+18], and reducing the dropout rate, as recommended in [Li+18].
- Add max-norm and momentum: as seen in the MLP results, max-norm and momentum 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-enable 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.
- 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.

V. CONCLUSIONS

Dropout is a popular regularization strategy. Batch Normalization is frequently used with deep neural networks to mitigate the gradient vanishing problem and also has a regularization effect. In this respect, Dropout and Batch Normalization overlap in some applications. Guidelines to use them are sometimes contradicting and often lacking in details.

This report examined the application of Dropout and Batch Normalization in multilayer perceptron networks (MLPs) and convolutional neural networks (CNNs), both in isolation and together. A network that does not use Dropout or Batch Normalization was used as a baseline.

The goal of the experiments was to analyze combinations of hyperparameters mentioned in the Dropout paper [Sri+14] and the Batch Normalization paper [IS15], both separately (only Dropout or only Batch Normalization) or in combination (both Dropout and Batch Normalization).

The MLP experiments 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% slower. This may be a factor for some applications because it also results in more energy use, draining batteries faster. This was an unexpected result of the tests and needs further validation.

The CNN tests 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 [IS15] improves accuracy by 2 to 3%. Since 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, remove all Dropout layers from the network and test it again to check it is not harming performance.

References

[MB89] Nelson Morgan and Hervé Bourlard. "Generalization and Parameter Estimation in Feedforward Netws: Some Experiments". In: *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*. Ed. by David S. Touretzky. Morgan Kaufmann, 1989, pp. 630–637. URL: http://papers.nips.cc/paper/275-generalization-and-parameter-estimation-in-feedforward-nets-some-experiments.

[LeC99] Yann LeCun. "The MNIST database of handwritten digits". In: (1999). URL: http://yann.lecun.com/exdb/mnist/.

[Kri09] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. 2009. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf (visited on 12/09/2018).

[NH10] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*. Ed. by Johannes Fürnkranz and Thorsten Joachims. Omnipress, 2010, pp. 807–814. URL: http://www.icml2010.org/papers/432.pdf.

[Ben12] Yoshua Bengio. "Practical recommendations for gradient-based training of deep architectures". In: (June 24, 2012). arXiv: http://arxiv.org/abs/1206.5533v2 [cs.LG].

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *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, United States*. Ed. by Peter L. Bartlett et al. 2012, pp. 1106–1114. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

[TH12] T. Tieleman and G. Hinton. *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning. 2012.

[Sri+14] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=2627435.2670313.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: (Feb. 11, 2015). arXiv: http://arxiv.org/abs/1502.03167v3 [cs.LG].

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. ISBN: 978-0-262-03561-3. URL: http://www.deeplearningbook.org/.

[HG16] Dan Hendrycks and Kevin Gimpel. "Adjusting for Dropout Variance in Batch Normalization and Weight Initialization". In: (July 8, 2016). arXiv: http://arxiv.org/abs/1607.02488v2 [cs.LG].

[Ker16] KerasTeam. *Using test data as validation data during training*. 2016. URL: https://github.com/keras-team/keras/issues/1753 (visited on 12/09/2018).

[Mis16] Dmytro Mishkin. 2016. URL: https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md (visited on 12/09/2018).

[Rud16] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: (Sept. 15, 2016). arXiv: http://arxiv.org/abs/1609.04747v2 [cs.LG].

[PW17] Luis Perez and Jason Wang. "The Effectiveness of Data Augmentation in Image Classification using Deep Learning". In: (Dec. 13, 2017). arXiv: http://arxiv.org/abs/1712.04621v1 [cs.CV].

[Bjo+18] Johan Bjorck et al. "Understanding Batch Normalization". In: (June 1, 2018). arXiv: http://arxiv.org/abs/1806.02375v4 [cs.LG].

[Koh+18] Jonas Moritz Kohler et al. "Towards a Theoretical Understanding of Batch Normalization". In: *CoRR* abs/1805.10694 (2018). arXiv: 1805.10694. URL: http://arxiv.org/abs/1805.10694.

[Li+18] Xiang Li et al. "Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift". In: (Jan. 16, 2018). arXiv: http://arxiv.org/abs/1801.05134v1 [cs.LG].

[Luo+18] Ping Luo et al. "Towards Understanding Regularization in Batch Normalization". In: (Sept. 4, 2018). arXiv: http://arxiv.org/abs/1809.00846v3 [cs.LG].

[NS18] Eric Nalisnick and Padhraic Smyth. "Unifying the Dropout Family Through Structured Shrinkage Priors". In: (Oct. 9, 2018). arXiv: http://arxiv.org/abs/1810.04045v1 [stat.ML].

[San+18] Shibani Santurkar et al. "How Does Batch Normalization Help Optimization?" In: (May 29, 2018). arXiv: http://arxiv.org/abs/1805.11604v3 [stat.ML].

[Kar] Karpathy. *ConvNetJS*. URL: https://cs.stanford.edu/people/karpathy/convnetjs/ (visited on 12/08/2018).

[Ste] Josef Steppan. *MNIST Examples*. URL: https://commons.wikimedia.org/wiki/File:MnistExamples.png (visited on 12/08/2018).

VI. Appendix

A. *Source code used for experiments*

The source code used in the experiments is available in Github at https://github.com/cgarbin/cap6619-deep-learning-term-project.

B. *How network model pictures were generated*

Pictures showing network models were generated in Keras with `plot_model(...)`, e.g.

```
plot_model(model, to_file="modelplot.png",
           show_layer_names=False,
           show_shapes=True)
```

The pictures were created right after adding the layers to the models and calling `model.compile()`. The pictures reflect the model exactly as they were trained and tested.