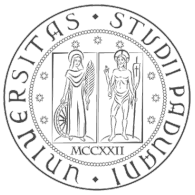


UNIVERSITÀ DEGLI STUDI DI PADOVA

SCUOLA DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA MAGISTRALE

IMPROVED TRAINING METHODS FOR NEURAL NETWORKS

Relatore: Prof. MATTEO FISCHETTI

Laureando: IACOPO MANDATELLI

Matricola 1151791

28 Settembre 2018

ANNO ACCADEMICO 2017-2018

Summary

The object of this thesis is to analyze the deep neural networks, mathematical models used in this circumstance for the classification of data, and to search for innovative solutions to the problems they are affected, for example the reduction of training time or the reduction of the number of tunable hyperparameters.

In the Machine Learning field there are many models and architectures to solve certain tasks: we focused on the neural networks because in the last few years they reached remarkable levels of accuracy in the classification task, one of the key problems of data science.

Another reason that guided us towards the analysis of these structures is the fact that there is not yet a complete understanding of their functioning or how performances are affected by the parameters, therefore it is still possible to achieve significant improvements with relatively simple techniques.

Two new methods will be presented, called *minibatch persistency* and *adaptive Nesterov*, targeted respectively to reduce training time and to automatically adapt the value of the stepsize: in addition to the different implementations, it will be made an analysis of the experimental results aimed at understanding strengths and weaknesses of each method.

The thesis is organized as follows: in Chapter 1 the deep neural networks and their functioning are described, besides giving the notation used in the following chapters. Chapter 2 describes a new method called *minibatch persistency* together with the results obtained from the experiments. In Chapter 3 we discuss the role of the learning rate in the training of neural networks, in addition to the presentation of policies such as the Cyclical Learning Rate and the adaptive learning rates. Also, in Chapter 3 we describe our new *adaptive Nesterov* method and the relative tests performed. Finally in Chapter 4 some conclusions and observations are drawn, together with future works.

Contents

Chapter 1

Deep Learning

Deep Learning it is a branch of Machine Learning based on learning data representation through the use of a neural network architecture, specifically deep neural networks.
Let's start first with defining the neural network model which is only vaguely inspired by information processing in the human nervous system: the idea is that many neurons can be connected to each other to carry out complex computations. Such structure therefore corresponds to a graph whose nodes are the neurons and each directed edge is a connection between two neurons.

1.1 Feedforward neural networks

A feedforward neural network is described by a directed acyclic graph $G = (V, E)$ and a weight function $w : E \to \mathbb{R}$; we assume that the network is organized in layers such that the set of nodes can be decomposed into a union of disjoint subsets, $V = \bigcup_{t=0}^{T} V_t$, such that every edge connects some node in V_{t-1} to some node in V_t, for $t \in T$.
The input layer, V_0, contains $n+1$ neurons where n is the dimensionality of the input space, the output of the neuron i in V_0 being denoted as x_i (the last neuron always outputs 1).
Called $v_{t,i}$ the i-th neuron of the t-th layer, $a_{t,i}(\mathbf{x})$ and $o_{t,i}(\mathbf{x})$ respectively the input and the output of $v_{t,i}$ when the network is fed with the input vector $\mathbf{x}$. Consequently the output of the neuron i at layer $t+1$ is given by:

$$\begin{aligned} a_{t+1,i}(\mathbf{x}) &= \sum_{r:(v_{t,r},v_{t+1,i})\in E} w((v_{t,r}, v_{t+1,i}))o_{t,r}(\mathbf{x}) \\ o_{t+1,i}(\mathbf{x}) &= \sigma(a_{t+1,i}(\mathbf{x})) \end{aligned} \tag{1.1.1}$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is the activation function of the neuron, typically the threshold function $\sigma(a) = \mathbb{1}_{[a>0]}$ or the sigmoid function $\sigma(a) = 1/(1+e^{-a})$. As we can see from Figure 1.1, the input of a node is the weighted sum, according to w, of the outputs of the neurons in the previous layer, and the output is the application of the activation function σ on its input.
Layers $V_1, ..., V_{T-1}$ are called hidden layers and the last layer V_T is called

output layer: the value T therefore represents the number of layers in the network (excluding the input layer) or its depth; if $T > 1$ we call the net a deep neural network.
Below is depicted an explanatory figure:

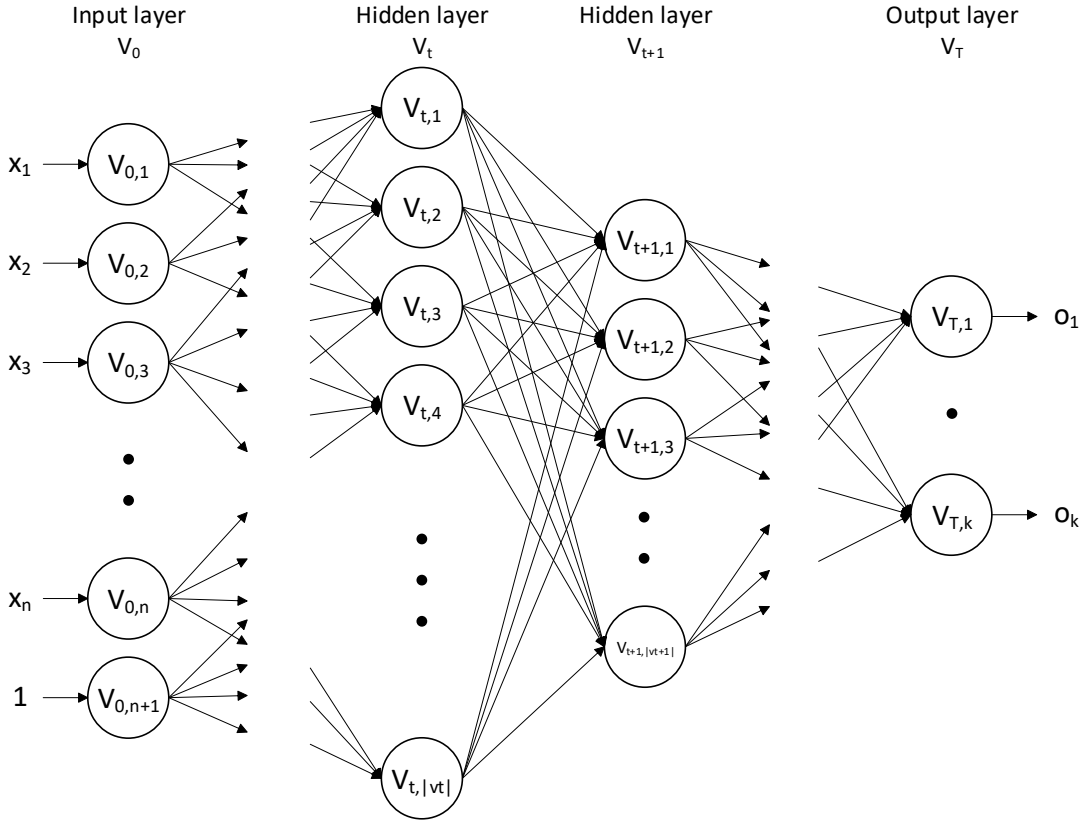


Figure 1.1: Scheme of a feedforward neural network.

1.2 Training the neural network: SGD and backpropagation

Once we have specified the architecture of the neural network by fixing V, E and σ, the hypothesis class is composed by all the functions $h_{V,E,\sigma,w}$ for any choice of $w : E \to \mathbb{R}$. Therefore the parameters specifying a hypothesis in the hypothesis class are the weights over the edges of the network.
As proved by Bartlett and Ben-David [1], it is NP hard to implement the ERM rule with respect to $H_{V,E,\sigma}$ for a network with a single hidden layer as well as finding weights that result in close-to-minimal empirical error. ERM, which stands for Empirical Risk Minimization, is a learning paradigm in which an algorithm tries to find an predictor (called also hypothesis, model) that minimizes the training error - the error the

predictor incurs over the training sample (also called empirical error). Klivans and Sherstov [13] proved an even more general result: under some cryptographic assumption, any hypothesis class which contains intersections of halfspaces cannot be learned efficiently.
Given the extreme complexity of the problem, it is preferable to look for a solution using a heuristic technique such as Stochastic Gradient Descent (SGD) [4]; it is proven that SGD is a successful learner if the loss function is convex, but in neural networks the loss function is highly non-convex: our hope is that the algorithm will find a reasonable solution anyway [5].
Once fixed the architecture of the network, that is fixing (V, E, σ), the problem of finding a hypothesis in $H_{V,E,\sigma}$ with a low risk corresponds to the problem of tuning the weights over the edges; since E is a finite set, we can think of the weights as a vector $\mathbf{w} \in \mathbb{R}^{|E|}$ and the function calculated by the network when it is fed with the input vector $\mathbf{x} \in \mathbb{R}^n$ can be denoted by $h_{\mathbf{w}} : \mathbb{R}^n \to \mathbb{R}^k$. Let's indicate by $\Delta(h_{\mathbf{w}}(\mathbf{x}), \mathbf{y})$ the loss of predicting $h_{\mathbf{w}}(\mathbf{x})$ when the target is $\mathbf{y} \in \mathcal{Y}$. A typical loss function could be the squared loss, $\Delta(h_{\mathbf{w}}(\mathbf{x}), \mathbf{y}) = \frac{1}{2} \|h_{\mathbf{w}}(\mathbf{x}) - \mathbf{y}\|^2$.
Finally, given a distribution $\mathcal{D}$ over the sample domain, the risk of the network, $L_{\mathcal{D}}$, is equal to:

$$L_{\mathcal{D}}(\mathbf{w}) = \mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim D}[\Delta(h_{\mathbf{w}}(\mathbf{x}), \mathbf{y})].$$

The resulting standard stochastic gradient descent algorithm for minimizing the risk is:

Algorithm 1 Stochastic Gradient Descent for minimizing $L_{\mathcal{D}}(\mathbf{w})$

1: **parameters:** $\eta > 0 \in \mathbb{R}^+$, $T \in \mathbb{N}$
2: initialize $\mathbf{w}^{(1)} = 0$
3: **for** $t = 1, 2, ..., T$ **do**
4: sample $(\mathbf{x}, \mathbf{y}) \sim D$
5: pick $\mathbf{v}_t \in \partial\Delta(h_{\mathbf{w}^{(t)}}(\mathbf{x}), \mathbf{y})/\partial\mathbf{w}$
6: update $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta\, \mathbf{v}_t$
7: **output:** $\mathbf{w}^{(T+1)}$

In more detail the SGD framework applied to the Neural Networks is as follows:

The most relevant differences are the initialization of $\mathbf{w}$ to a random vector with elements close to zero, and the fact that the gradient does not have a closed form solution and is instead calculated using the back-propagation algorithm. Note that it is also possible to return the best performing $\mathbf{w}^{(t)}$ on a validation set instead of $\mathbf{w}^{(T+1)}$ (value of $\mathbf{w}$ at the

Algorithm 2 Stochastic Gradient Descent for Neural Networks

1: **parameters:** $\eta > 0 \in \mathbb{R}^+$, $T \in \mathbb{N}$
2: initialize $\mathbf{w}^{(1)} \in \mathbb{R}^{|E|}$ from a zero mean distribution
3: **for** $t = 1, 2, ..., T$ **do**
4: sample $(\mathbf{x}, \mathbf{y}) \sim D$
5: calculate $\mathbf{v}_t = backpropagation(\mathbf{x}, \mathbf{y}, \mathbf{w}^{(t)}, (V, E, \sigma))$
6: update $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta\, \mathbf{v}_t$
7: **output:** $\mathbf{w}^{(T+1)}$

Algorithm 3 Backpropagation algorithm

1: denote $V_0, ..., V_T$ the layers of the graph, where $V_t = \{v_{t,1}, ..., v_{t,k_t}\}$
2: denote with σ' the derivative of the activation function σ
3: define $w_{t,i,j}$ as the weight of $(v_{t,j}, v_{t+1,i})$
4: **forward pass:**
5: set $o_0 = \mathbf{x}$
6: **for** $t = 1, 2, ..., T$ **do**
7: **for** $i = 1, 2, ..., k_t$ **do**
8: set $a_{t,i} = \sum_{j=1}^{k_{t-1}} w_{t-1,i,j}\, o_{t-1,j}$
9: set $o_{t,i} = \sigma(a_{t,i})$
10: **backward pass:**
11: set $\delta_T = o_T - \mathbf{y}$
12: **for** $t = T-1, T-2, ..., 1$ **do**
13: **for** $i = 1, 2, ..., k_t$ **do**
14: $\delta_{t,i} = \sum_{j=1}^{k_{t+1}} w_{t,j,i}\, \delta_{t+1,j}\, \sigma'(a_{t+1,j})$
15: **output:** $\forall\, (v_{t,j}, v_{t+1,i}) \in E$ set $\frac{\partial \Delta(h_{\mathbf{w}}(\mathbf{x}), \mathbf{y})}{\partial w_{t,i,j}}$ to $\delta_{t,i}\, \sigma'(a_{t,i})\, o_{t-1,j}$

last iteration).
The presented version of the SGD algorithm is the on-line variant (lines 4 and 5 of Algorithm 2) which means that the true gradient of the loss function $\Delta(h_{\mathbf{w}}(\mathbf{x}), \mathbf{y})$ is approximated by the gradient of a single example: consequently, as the algorithm iterates through the training set, it performs a weight update for each training example.Typically several passes are made over the training set (line 3 of Algorithm 2) until the algorithm converges.
There is an alternative version to the on-line one, called *minibatch* variant: an update of the parameters is performed for every minibatch of m training examples. In particular, the gradient of a minibatch of size m is computed as the average of the m gradients of the given examples in the minibatch, with respect to the considered loss function. Let $L_i(\theta)$ denote the contribution to the loss function of the i-th training example in the minibatch, with respect to the weight vector θ. Then for every

minibatch of size m, the weight update rule at iteration t is as follows:

$$\begin{aligned} \nabla_\theta L(\theta_t) &= \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L_i(\theta_t) \\ v_t &= \gamma\, v_{t-1} + \eta\, \nabla_\theta L(\theta_t) \\ \theta_{t+1} &= \theta_t - v_t \end{aligned} \tag{1.2.1}$$

The above update policy corresponds to lines 5 and 6 of Algorithm 2 and contains a momentum factor as proposed by Qian [21], which we will see in the next section.

1.3 SGD with momentum

SGD has trouble navigating areas where the surface of the loss function curves much more steeply in one dimension than in another, which are quite common around local minima [24]. In these cases SGD oscillates across the slopes of the ravine without making significant progress along the bottom towards the local optimum, as in Figure 1.2a.

(a) SGD without momentum.

(b) SGD with momentum.

Figure 1.2: Optimization paths of SGD with and without momentum.

Momentum [21] is a method that helps accelerating SGD in the relevant direction and reduces oscillations, as we can see in Figure 1.2b. It achieves this by adding a fraction γ of past update vector to the current gradients. The update rule at iteration t is:

$$\begin{aligned} v_t &= \gamma\, v_{t-1} + \eta\, \nabla_\theta L(\theta_t) \\ \theta_{t+1} &= \theta_t - v_t \end{aligned} \tag{1.3.1}$$

where $\nabla_\theta L(\theta_t)$ is the gradient of the loss function with respect to parameters θ_t, η is the learning rate, and γ is the momentum factor, usually set between 0.5 and 0.9. The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. The end result is faster convergence and reduced oscillations.
Nesterov momentum [19] is a slightly different version of the momentum

update that has recently raised much interest (albeit the original proposal is dated back to 1983). For convex functions, it enjoys stronger theoretical converge guarantees if compared to standard momentum. Although the loss function of neural networks is highly non-convex, in practice it consistently delivers fast convergence to good minima.
The core idea behind Nesterov momentum is that, instead of calculating gradient at the current position, the gradient is calculated at the approximated next position. Computing $\theta - \gamma\, v_{t-1}$ gives an approximation of the next position of the parameters, from which the gradients can be calculated. The Nesterov update rule then becomes:

$$\begin{aligned} v_t &= \gamma\, v_{t-1} + \eta\, \nabla_\theta L(\theta_t - \gamma v_{t-1}) \\ \theta_{t+1} &= \theta_t - v_t \end{aligned} \tag{1.3.2}$$

In Figure 1.3 there is a graphical representation of the difference between standard and Nesterov momentum. Momentum first computes the current gradient (small blue vector) and then takes a big jump in the direction of the past update vector (big blue vector). Nesterov Momentum instead first makes a big jump in the direction of the previous accumulated gradient (brown vector), measures the gradient (red vector) and then makes a correction (green vector).

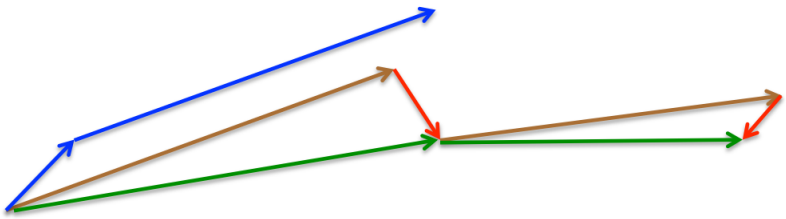


Figure 1.3: Difference between standard and nesterov momentum update vectors.

1.4 Other neural network elements

Later in this thesis, neural networks more advanced than those presented in Section 1.1 will be used. In particular, for the purpose of image classification, there exist structures of neurons and connections between them that have become a standard in terms of effectiveness. In this section we will briefly introduce the most important ones, referring to other authors for a more detailed analysis [23].
Let's start by saying that an image, considered for simplicity in black and white (grayscale), it is nothing but a matrix of n x m pixels in which each pixel takes a value in $\{0, ..., 255\}$ or $[0, 1]$ (respectively from black

to white). To apply such an image as input to the network in Figure 1.1 we have to resize (eg. row-major) the matrix into a vector of length nm.
One of the most important structures is the convolutional layer, which takes its name from the convolution operation between a kernel and an image: the primary purpose of convolution is to extract features from the input image as can be see in the example of Figure 1.4:

(a) Input image.

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

(b) Kernel matrix.

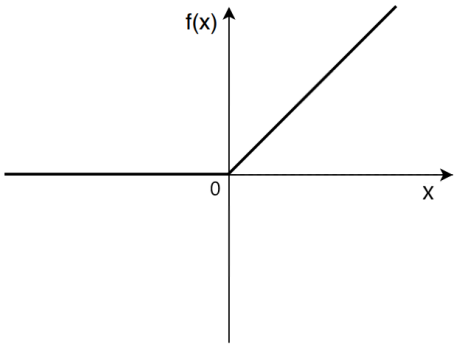
(c) Resulting image.

Figure 1.4: Convolution of an image with a edge detector kernel.

Convolution preserves the spatial relationship between pixels and the type of features extracted vary a lot (e.g., geometric figure detection). The idea is that the more filters we have, the more image features get extracted and the better our network becomes at recognizing patterns in unseen images.
In practice a network that uses convolutional layers (called Convolutional Neural Netowrk - CNN) learns the values of the kernel matrices during the training process (as it happens with weights over the edges), but it is still necessary to specify parameters such as the number of filters and filter size before the training phase. The multiple levels of representation given by the various convolutional layers of a CNN manage to capture increasingly more abstract concepts from the input image.
Another very common element is the Rectified Linear Unit (ReLU), and it is applied to the output of a neuron or to the feature image: if the value of an output is negative it is replaced by zero, otherwise is not changed. The ReLU is a non-linear operation and its output is given

by:

$$ReLU(x) = max(0, x)$$



As found by Krizhevsky et al. [16] the ReLU activation function greatly accelerate the convergence of SGD if compared to other activation functions, such as the sigmoid or tanh functions.
After doing the convolutional step it is appropriate to reduce the dimensionality of the obtained feature map to prevent overfitting. To do so we define a spatial neighborhood (for example, a 2 x 2 window) and take the largest element (Max Pooling) from the feature map within that window; one can also take the average (Average Pooling) or sum of all elements in that window.
In Figure 1.5 in shown the Max Pooling of a feature map using a 2 x 2 window, moved each time by 2 cells (also called *stride*).

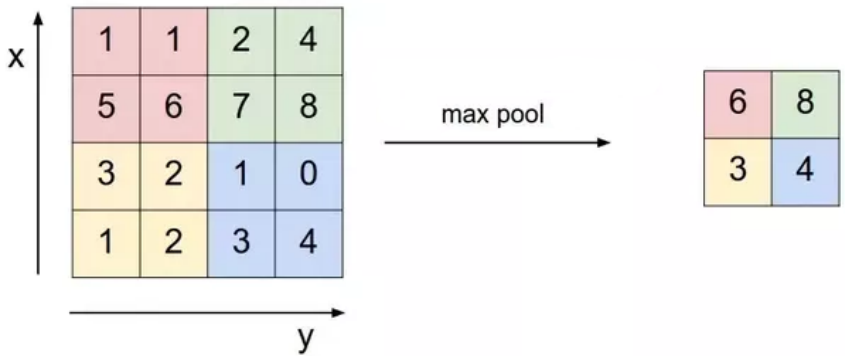


Figure 1.5: Example of Max Pooling.

In addition to controlling overfitting, pooling also makes the network invariant to small transformations, distortions and translations in the input image: this is very useful since we can detect objects in an image no matter where they are located.
The various layers of the network in Figure 1.1 are called fully connected layers, and they are characterized by the fact that each node of a layer t is connected to all the other nodes of layers $t-1$ and $t+1$. Fully

connected layer is a traditional Multi-Layer Perceptron: the output from the convolutional and pooling layers represent high-level features of the input image which can then be classified into various classes by the fully connected layers.

Chapter 2

Minibatch Persistency

In the previous chapter it was presented the on-line version of the gradient descent algorithm, called Stochastic Gradient Descent, in which the network weights are updated for each example in the training dataset: this variant of the algorithm has the advantage of frequently updating the parameter, which can result in faster learning on some problems, and the noisy update process can allow the model to avoid local minima. The main disadvantage of this approach lies in the fact that is more computationally expensive than other configurations of gradient descent, taking significantly longer to train models on large datasets.
In the last few years, the enormous growth in availability and power of parallel architectures (notably, GPUs) made it progressively more convenient the use of larger minibatches: this variant of the gradient descent algorithm, called Mini-Batch Gradient Descent, splits the training dataset into small batches that are used to calculate an approximation of the expected value of the gradient of the loss function and then updates the net parameters accordingly. This version of the algorithm has the advantage of providing a computationally more efficient process than stochastic gradient descent at the price of adding an additional hyperparameter to the learning model.
It has been observed in practice that when using a large minibatch there is a degradation in the generalization properties of the model. Keskar et al. [11] suggested that the reason for poor generalization when using large minibatches is that training in this way tends to converge to sharp minima, which lead to poorer generalization if compared to smaller minibatches. They also observed that the large batch methods lack the explorative properties of small batch methods and tend to converge on minima closest to the initial starting point.
Hoffer et al. [10] have shown empirically that the "generalization gap" stems from the relatively small number of updates rather than the batch size and therefore it is possible to maintain generalization performance with large minibatches by performing the same number of SGD updates. They suggested that there is no inherent "generalization gap": large-batch training can generalize as well as small-batch training by adapting the number of iterations. This implies however a computa-

tional overhead proportional to the mini-batch size, which reduces the positive effect of the increased parallelism.
Based on the previous results, Luschi et al. [18] proposed to scale the learning rate linearly with the batch size to cope with large minibatches: the produced results seem to indicate that increasing the batch size results in both a degradation of the test performance and a progressively smaller range of learning rates that allows stable training.
All the published works however give for granted that a "disposable minibatch" strategy is adopted, namely: within one epoch, the current minibatch is changed at each SGD iteration.
Our goal then is to use large minibatches to exploit the memory and the parallelism of modern GPUs while maintaining a good generalization capability of the model.

2.1 The idea

We investigated a new strategy that reuses a same minibatch for K consecutive SGD iteration, namely we do K successive updates of the net weights using the same minibatch before using another one. We call the parameter K *minibatch persistency*, being $K = 1$ the standard update rule. This approach also has the practical advantage of reducing the computational overhead related to the operation of loading new data into the GPU memory.
The intuition behind this idea is that large minibatches contain a lot of information about the training set, that we do not want to waste by dropping them too early. Of course using consecutively too many times the same minibatch is risky in terms of overfitting, hence one has to computationally evaluate the viability of the approach.
As a result the Algorithm 2 seen in the previous chapter now becomes:

Algorithm 4 Stochastic Gradient Descent for Neural Networks with Minibatch Persistency

parameters: $\eta > 0 \in \mathbb{R}^+, T \in \mathbb{N}$
initialize $\mathbf{w}^{(1)} \in \mathbb{R}^{|E|}$ from a zero mean distribution
for $t = 1, 2, ..., T$ **do**
    sample $(\mathbf{x}^{(bs)}, \mathbf{y}^{(bs)}) \sim D$
    for $i = 1, 2, ..., K$ **do**
        calculate $\mathbf{v}_{t+i-1} = backpropagation(\mathbf{x}^{(bs)}, \mathbf{y}^{(bs)}, \mathbf{w}^{(t)}, (V, E, \sigma))$
        update $\mathbf{w}^{(t+i)} = \mathbf{w}^{(t+i-1)} - \eta\, \mathbf{v}_{t+i-1}$
output: $\mathbf{w}^{(KT+1)}$

where bs is the size of the minibatch and K is the minibatch persis-

tency parameter.
The above update policy contains a momentum factor as proposed by Qian [21]. The choice to include it is therefore arbitrary and motivated by the fact that momentum has now become a standard.

2.2 Tests and experiment setup

The computational tests were performed on the CIFAR-10 dataset [15] using a reduced version of the AlexNet architecture [16], following as a reference what was recently done by Luschi et al. [18].
The other two network architectures used are ResNet-34 and VGG16: the former was considered as described in [8] while the latter was modified on the final fully-connected layers, with a single one with 512 hidden nodes.
In particular the reduced AlexNet implementation uses convolutional layers with stride equal to 1, kernel sizes equal to $[11, 5, 3, 3, 3]$, number of channel per layer equal to $[64, 192, 384, 256, 256]$, max-pool layers with 2 x 2 kernels and stride 2, and 256 hidden nodes for the fully-connected layer.
The CIFAR-10 dataset was shuffled and partitioned into 50,000 examples for the training set and the remaining 10,000 for test set.
Since our experiments are just aimed at evaluating the impact of different values of the minibatch persistency parameter K for a given minibatch size m, we decided to use the basic training algorithm described previously: in particular we did not use dropout [9] or data augmentation. Momentum coefficient γ and learning rate η were fixed, respectively to 0.5 and 0.001/0.01; the optimization method was minibatch SGD with cross entropy loss, a combination of Softmax function and negative log likelihood loss (NLLLoss).
Fixing a priori the momentum coefficient and learning rate (independently of the minibatch size) has some drawbacks as described by Wilson et al. [26]. Therefore the resulting accuracies measured on the test set are certainly not competitive with the state of the art.
The machine used to perform the runs is equipped with an Intel® Xeon® E5v4 CPU @ 3.00 GHz, 64 GB of RAM and a single Nvidia® GTX 1080 Ti GPU, coupled with a custom Linux installation. All the code was implemented in Python v3.6.6 and the reference framework used is PyTorch v0.4.0

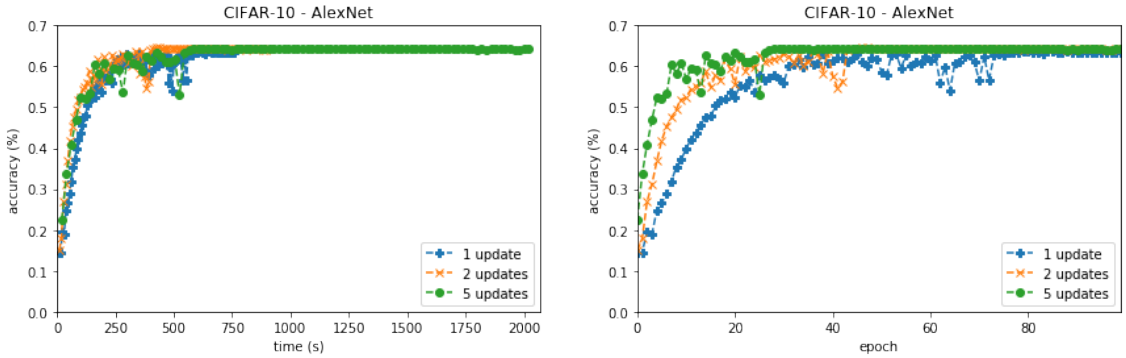
2.3 Results

Each figure in the following sections plots the percentage accuracy and the value of the loss function (on the test dataset) for minibatch persistency parameter $K \in \{1, 2, 5\}$, as a function of the total computing time (subfigure on the left) and of the number of epochs (subfigure on the right). As computing time we considered the wall-clock time, measured in seconds. Since within a single epoch each training example is evaluated K times, one would expect the computing time to perform each epoch be multiplied by K: this is why to have a fair comparison of different values of K it is important to report computing times explicitly.

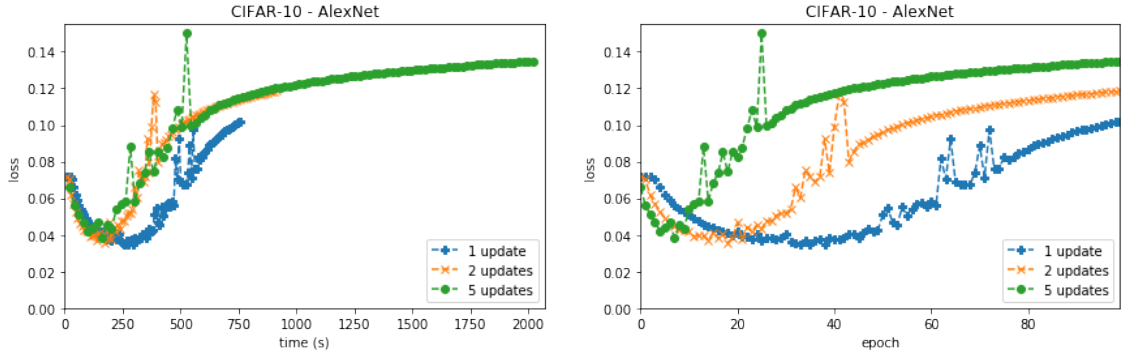
2.3.1 AlexNet

Figures from 2.1 to 2.6 report the results of our experiments for minibatch sized $m = 32$, 256 and 512 and learning rate equal to 0.001 and 0.01. According to [18] the best performance for CIFAR-10 and a reduced AlexNet architecture is achieved for $m \leq 8$, while minibatches of size 256 or 512 are considered too large to produce relevant results: as it is possible to see this is no longer true when minibatch persistency is used.
Figures 2.1 and 2.2 refers to a small minibatch of size $m = 32$: when the learning rate is equal to 0.001, higher values of K improve accuracy measured with respect to epochs, however there is not an equal improvement in computing time. When η is equal to 0.01 the accuracy (measured with respect to epochs or time) seems not to be much influenced by K. It is interesting to note how completing 100 epochs with $K = 5$ took roughly 3 times more time than when $K = 1$, which is less than the factor 5 one would expect.
If we look at the loss graphs for $m = 32$, independently from the learning rate, when K increases the overfitting starts earlier: this behaviour is not unexpected, as mentioned previously the minibatch is probably too small to be representative of the whole training dataset.
When a bigger size minibatch is used, $m = 256$, the results change significantly (Figures 2.3 and 2.4). Here overfitting is always present but it does not lead to a deterioration of accuracy: in fact, the maximum accuracy has a greater value and it is reached earlier when K is large (for both values of η).

We can also see that the additional iterations over the same (large)

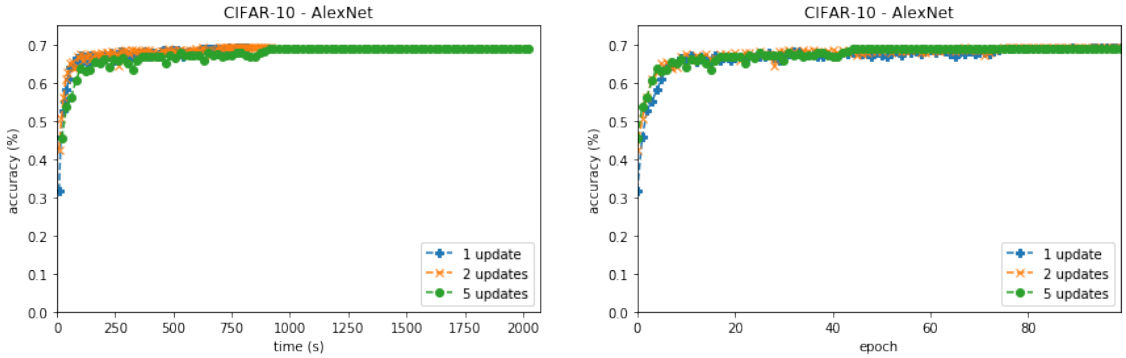


(a) Test accuracy.

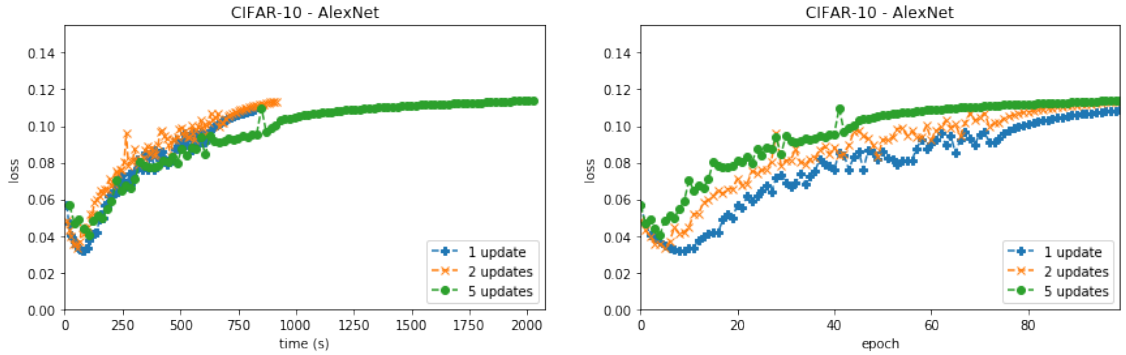


(b) Test loss.

Figure 2.1: Results with m = 32, $\eta =$ to 0.001 and $K = 1$, 2 and 5. AlexNet.

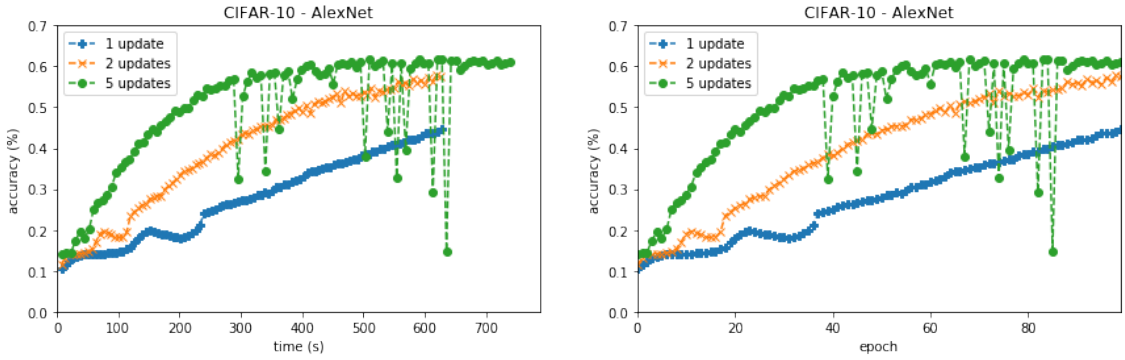


(a) Test accuracy.

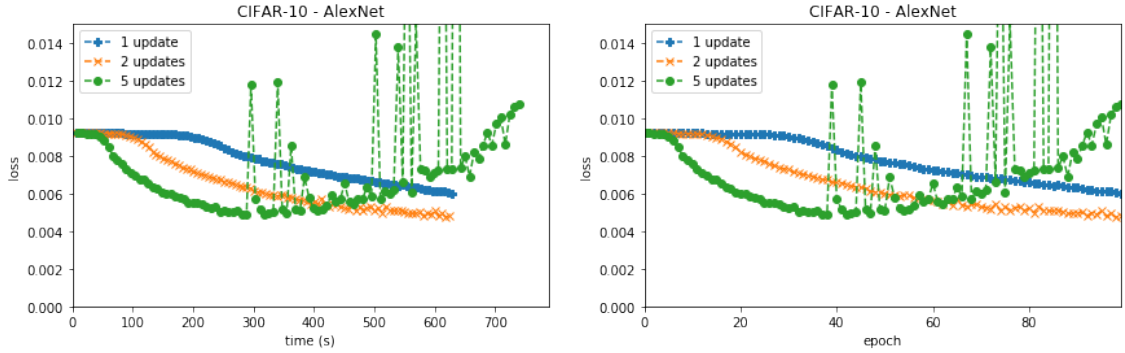


(b) Test loss.

Figure 2.2: Results with m = 32, $\eta =$ to 0.01 and $K = 1$, 2 and 5. AlexNet.

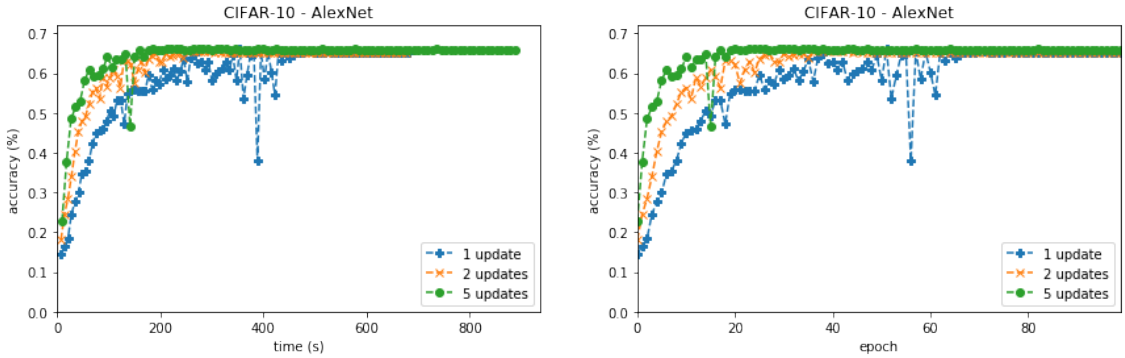


(a) Test accuracy.

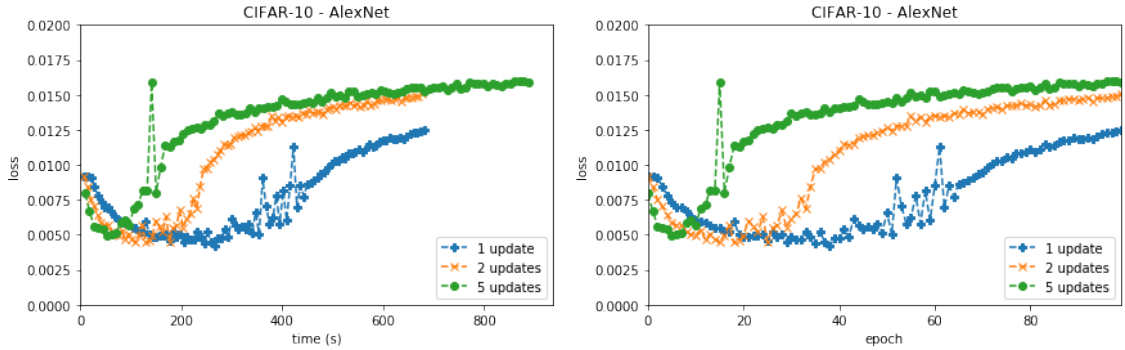


(b) Test loss.

Figure 2.3: Results with m = 256, $\eta =$ to 0.001 and $K = 1$, 2 and 5. AlexNet.

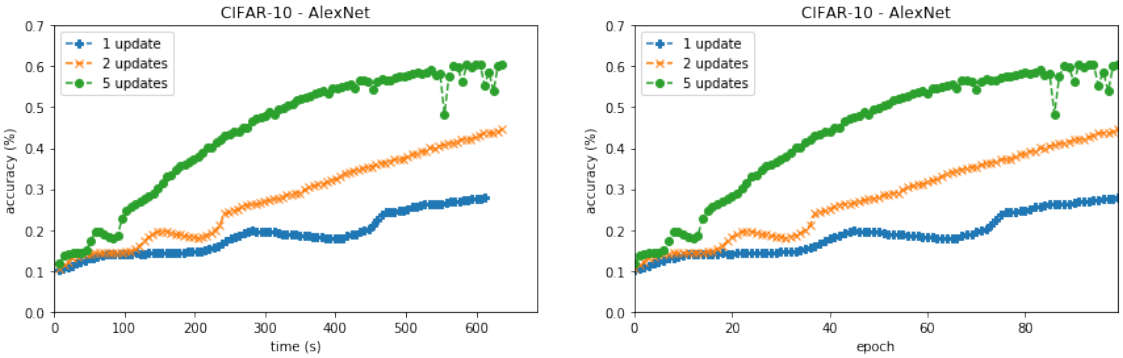


(a) Test accuracy.

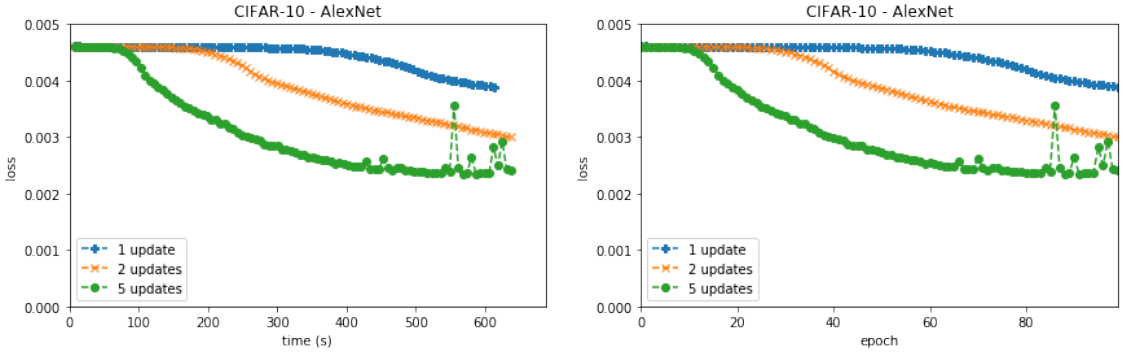


(b) Test loss.

Figure 2.4: Results with m = 256, $\eta =$ to 0.01 and $K = 1$, 2 and 5. AlexNet.

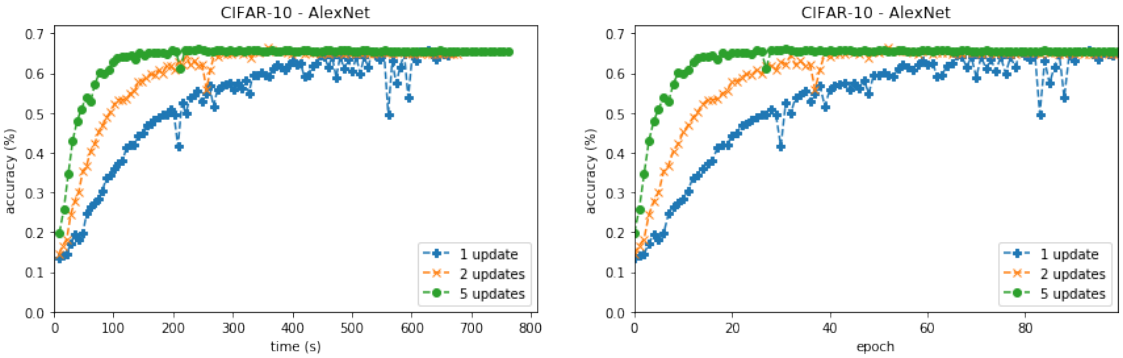


(a) Test accuracy.

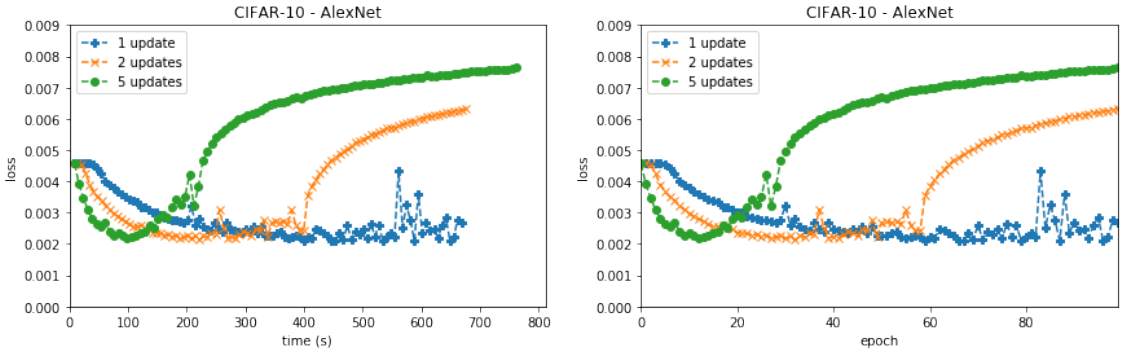


(b) Test loss.

Figure 2.5: Results with m = 512, $\eta =$ to 0.001 and $K = 1$, 2 and 5. AlexNet.



(a) Test accuracy.



(b) Test loss.

Figure 2.6: Results with m = 512, $\eta =$ to 0.01 and $K = 1$, 2 and 5. AlexNet.

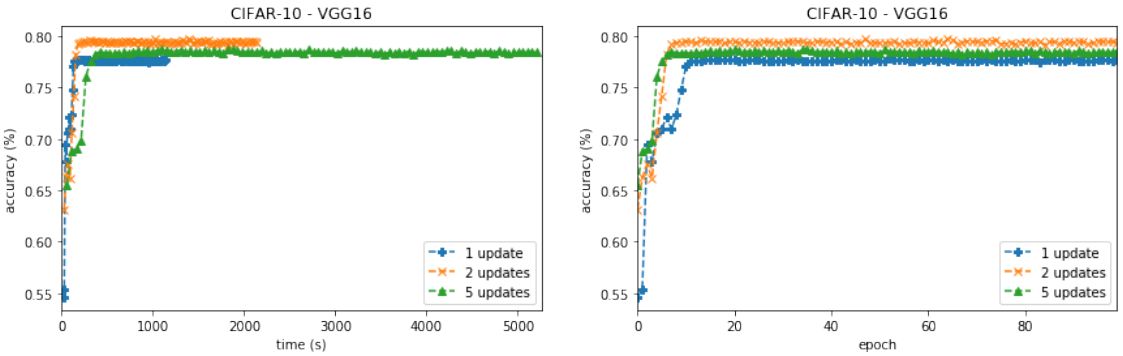
		accuracy			loss		
configuration	K	max-value	time(s)	epoch	min-value	time(s)	epoch
	1	0.6364	607.4	80	0.03472	273.5	36
32_001	2	0.6449	429.1	47	0.03557	174.0	19
	5	0.6445	606.5	30	0.03869	162.0	8
	1	0.6913	777.8	95	0.03185	82.7	10
32_01	2	0.6938	870.5	95	0.03353	55.7	6
	5	0.6891	1741.8	86	0.04040	101.9	5
	1	0.4453	627.5	100	0.00601	627.5	100
256_001	2	0.5773	618.7	99	0.00473	618.7	99
	5	0.6176	620.7	84	0.00486	288.3	39
	1	0.6602	354.5	52	0.00423	266.0	39
256_01	2	0.6562	264.6	39	0.00444	129.8	19
	5	0.6617	230.1	26	0.00495	53.8	6
	1	0.2781	612.6	100	0.00389	612.6	100
512_001	2	0.4454	636.6	100	0.00300	636.6	100
	5	0.6040	605.6	95	0.00235	567.6	89
	1	0.6576	628.3	94	0.00210	587.9	88
512_01	2	0.6639	358.4	53	0.00212	222.1	33
	5	0.6600	242.8	32	0.00219	99.0	13

Table 2.1: Best accuracy and loss for each configuration of m, η and K with the respective times and epochs. AlexNet architecture.

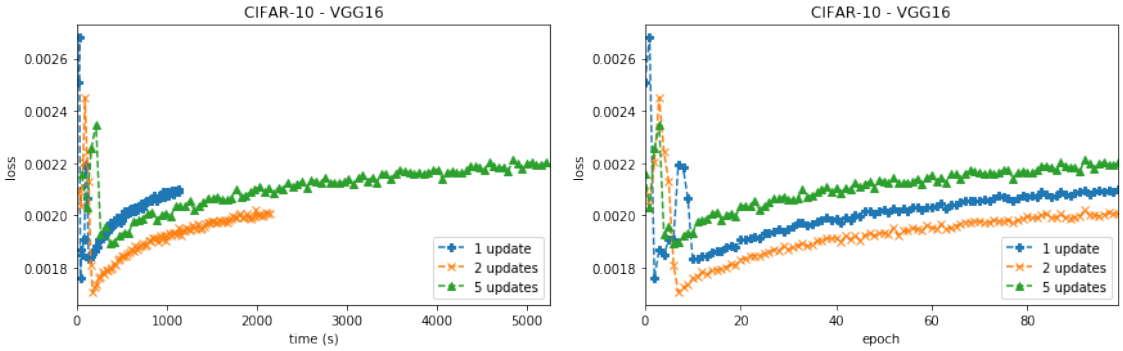
minibatch have a smaller temporal impact compared to using a small minibatch: this confirms that minibatch persistency has a positive effect in terms of GPU exploitation when the minibatch is big enough.
The same considerations can be made for even a larger minibatch of size 512 (figures 2.5 and 2.6): repeating multiple iterations on the same minibatch produces clearly positive results. It should also be said that the value of $\eta = 0.001$ it is probably too low and therefore we are only observing the initial part of the optimization process.
The above results are rather encouraging and show that the use of large minibatches becomes more appealing when combined with the minibatch persistency technique.
Table 1 shows the values of best accuracies and losses for each combination of minibatch size, learning rate and minibatch persistency parameter, with the respective times and values of the epoch. The numerical values refer to the same runs as the graphs just presented.

2.3.2 VGG16 and ResNet34

Additional runs have been performed on VGG16 and ResNet34 (the architecture specifications are present in Section 2.2), considering as minibatch size the value 512 and as learning rate the values 0.01 and 0.1. All other parameters remained unchanged from the previous runs. Figures 2.7 and 2.8 refers to VGG16: with a bigger and more complex network the benefits of using the minibatch persistency changes slightly. While for AlexNet the minibatch persistency technique produces almost no increase in maximum accuracy but an improvement of computing time (figure 2.6) it produces almost opposite results with VGG16.
In particular, looking at Figure 2.7a we can see how the maximum accuracy reached increases significantly, even if using high values of K is more subject to overfitting. Contrary to what happened for AlexNet, reusing the same minibatch K times makes the execution time roughly K times longer: this could be due to the fact that data loading on the GPU is no longer the bottleneck of executions. A more performing GPU (or multiple GPUs) could change this effect. The same considerations, albeit mitigated, remain valid even when η is equal to 0.1 (Figure 2.8).

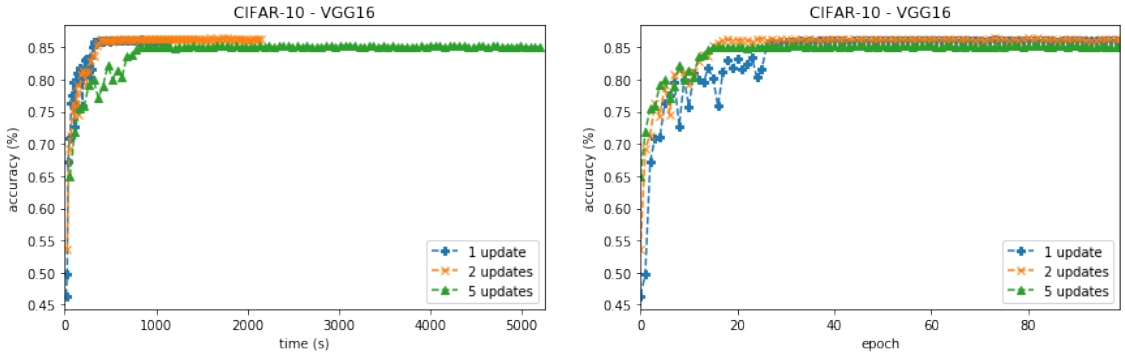


(a) Test accuracy.

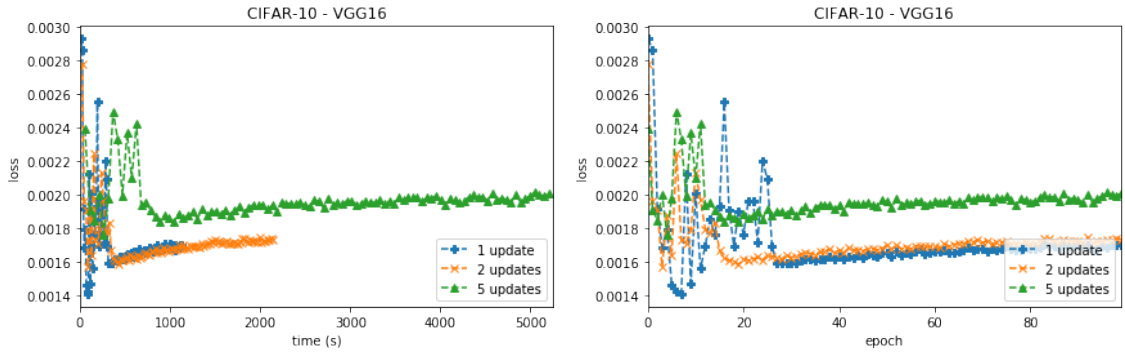


(b) Test loss.

Figure 2.7: Results with m = 512, $\eta =$ to 0.01 and $K = 1, 2$ and 5. VGG16.



(a) Test accuracy.



(b) Test loss.

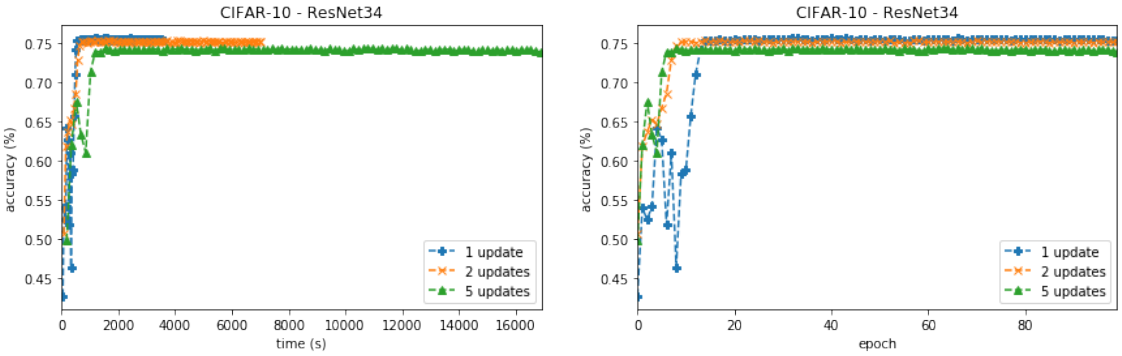
Figure 2.8: Results with m = 512, $\eta =$ to 0.1 and $K = 1, 2$ and 5. VGG16.

Looking at the accuracy obtained by ResNet34 on CIFAR-10, Figures 2.9 and 2.10, there is a marked deterioration in performance compared to what happened for VGG16. The larger value of η (0.1) highlights this behaviour even more.
An important thing to observe is that even the standard SGD method, $K = 1$, performs worse on ResNet34 than on VGG16. If we look at the number of parameters (weights on the edges) of ResNet34, which determines the complexity of the model, we can explain the results obtained: ResNet34 with 21.2 million parameters, compared to 2.5 of AlexNet and 14.7 of VGG16, is much more complex than these two. Probably the complexity of the dataset is too low compared to that of the network, which ends up overfitting it without producing good results.

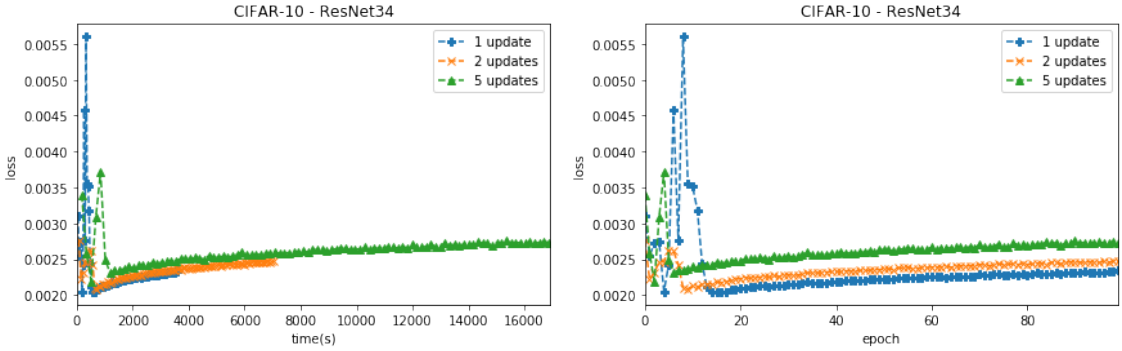
Table 2 summarizes the values of best accuracies and losses for each combination of minibatch size, learning rate and minibatch persistency parameter, with the respective times and values of the epoch.

2.4 Improving the performance

After testing the effectiveness of our method under controlled conditions (i.e., by removing or minimizing the effect of other variables such

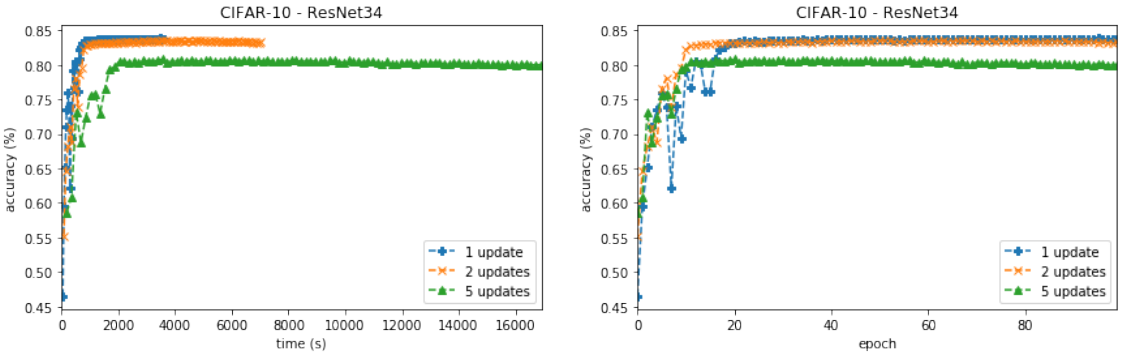


(a) Test accuracy.

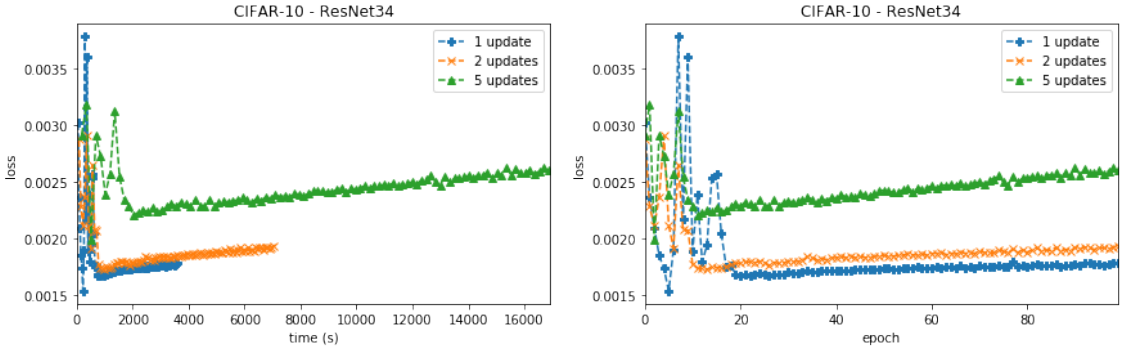


(b) Test loss.

Figure 2.9: Results with m = 512, $\eta =$ to 0.01 and $K = 1$, 2 and 5. ResNet34.



(a) Test accuracy.



(b) Test loss.

Figure 2.10: Results with m = 512, $\eta =$ to 0.1 and $K = 1$, 2 and 5. ResNet34.

		accuracy			loss		
configuration	K	max-value	time(s)	epoch	min-value	time(s)	epoch
VGG16_ 512_1	1	0.8621	835.6	74	0.00141	93.2	8
	2	0.8644	1737.5	81	0.00157	90.0	4
	5	0.8535	3016.6	58	0.00177	263.8	5
VGG16_512_01	1	0.7783	603.9	53	0.00176	38.5	3
	2	0.7974	1392.5	65	0.00171	175.0	8
	5	0.7872	2708.9	52	0.00189	368.4	7
ResNet34_512_1	1	0.8390	3452.9	96	0.00154	215.9	6
	2	0.8355	3583.1	51	0.00173	985.3	14
	5	0.8085	3548.8	21	0.00199	508.7	3
ResNet34_512_01	1	0.7561	1479.3	41	0.00205	578.9	16
	2	0.7534	1266.2	18	0.00209	704.7	10
	5	0.7439	5228.1	31	0.00219	509.3	3

Table 2.2: Best accuracy and loss for each configuration of m, η and K with the respective times and epochs. VGG16 and AlexNet34 architectures.

as learning rate, optimization method, and dropout), we decided to test the *minibatch persistency* method in conditions closer to the real ones in which one wants to maximize the performance of a neural network.
In particular, of the three architectures seen in the previous section we considered AlexNet because it is the one that scales better with our machine. We have taken as reference two different implementations [28, 17] that use a reduced version of AlexNet and try to reach the maximum possible accuracy. To do this, in addition to using a particular parameter configuration, they introduce a technique called Data Augmentation. Since a typical convolutional neural network enjoies the invariance property, namely it can successfully recognize images in which the objects represented are translated or rotated, it is possible to increase the quality of the results produced by the network.
Data Augmentation refers to any method that artificially increases the original training set with label-preserving transformations and can be represented as the mapping:

$$\phi : S \to T$$

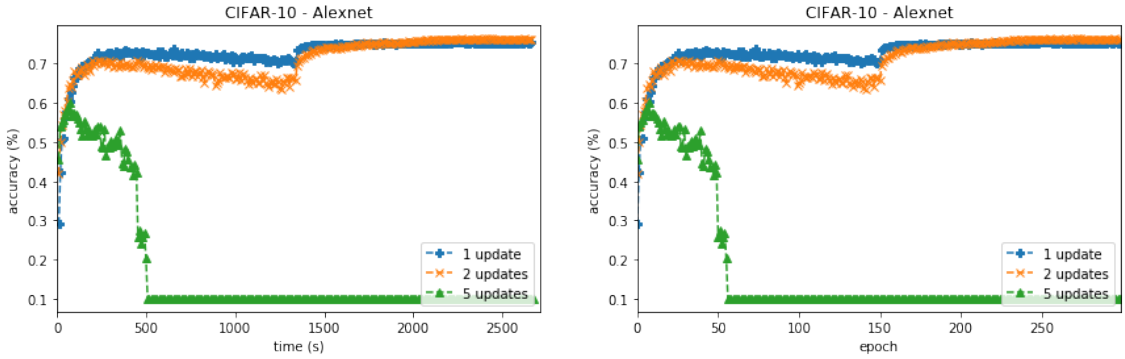
where, S is the original training set and T is the augmented set of S. The artificially inflated training set is thus represented as:

$$S' = S \cup T$$

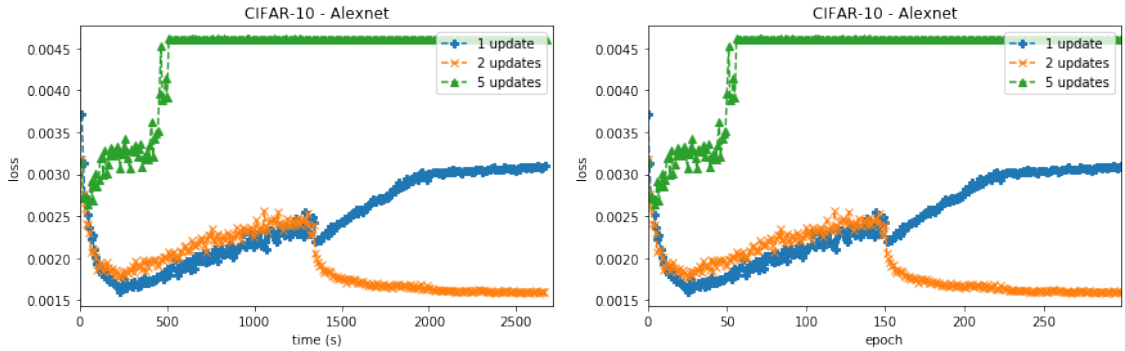
Note that the term "label-preserving transformations" refers to the fact that if image x is an element of class y then $\phi(x)$ is also an element of

class y. The are two main types of transformations, geometric transformations which alter the geometry of the image and the photometric transformations which modify the colour channels. Common examples of the former are flip, rotation, scaling, cropping, translation, and of the latter are colour jittering and edge enhancement.
Besides augmenting the size of the dataset, Data Augmentation is also a form of regularization, as observed by Yaeger et al. [27]. These two properties make Data Augmentation a technique that greatly improves deep learning algorithms performances in most cases [25].
In particular we used a random horizontal flip (with probability equal to 0.5) and random translation of 4 pixels (with padding equal to 0, $p = 0.5$) as geometric transformations to the CIFAR10 dataset. The use of data augmentation is the single improvement that has mostly increased the performance of the network. With respect to the setup previously presented, we incremented the epochs to 300 and we scheduled a decreasing learning rate value during the training: lr=0.1 for the first 150 epochs, lr=0.01 from 150 to 225 epochs, and then lr=0.001 for the last 75 epochs. The momentum coefficient and the other parameters remained unchanged. It should be noted that this particular scheduling of the learning rate is the one that gave us the best results, considering that we want the best performing configuration against which to compare our minibatch persistency method.
Figure 2.11 reports the results of three runs with $K = 1, 2, 5$ with the configuration just presented. We can see immediately that something went wrong for $K = 5$, most likely due to severe overfitting which has made the parameters of the network diverge.
From the run with $K = 1$ (that is the standard SGD method) we get results in line with our expectations, reaching a maximum accuracy of 0.7544, very similar to what obtained by other implementations from the literature. The behaviour of the loss function when $K = 2$ is very different from what has been observed so far: after reaching a minima ($\sim$25 epochs) the loss increases but then it suddenly drops ($\sim$150 epochs) when the learning rate is reduced. As a result in the final epochs, when the learning rate is decreased, the accuracy obtained by the *minibatch persistency* method outperforms that obtained from the standard SGD method, reaching a value of 0.7646.
It can also be noted that the total computing time seems not to be influenced by the parameter K: this could be due to a good exploitation of our GPU with the AlexNet architecture and consequently the total

running time is dominated by the data transfer to the GPU.



(a) Test accuracy.



(b) Test loss.

Figure 2.11: Results with m = 512, scheduled η and $K = 1$, 2 and 5. AlexNet.

Chapter 3

Role of learnig rate on neural networks

The learning rate is a parameter that controls how much network weights are adjusted with respect to the loss gradient, as already seen in Section 1.2 and Section 2.1.
Bengio [2] underlines how the learning rate is often the single most important hyper-parameter: too small a learning rate will make a training algorithm converge slowly, while too large a learning rate will make the algorithm diverge. Typical values of the learning rate for a neural network trained with normalized inputs range between 1 and 10^{-6}, although the exact values greatly depend on the parametrization of the model. A good starting value could be 0.01, which typically works for standard multi-layer neural networks.
In common practice the learning rate is decreased monotonically during the training, following a particular scheduling: Bergstra and Bengio [3] proposed a value inversely proportional to the number of iterations, $\mu_t \propto \frac{\mu_0}{t}$, or alternatively, a step decay method could be used. In order to find the best range of values for the learning rate, one could set up a small validation batch and observe the trend of the loss function.

3.1 Cyclical Learning Rate

Smith [22] proposed a simple yet innovative approach: the global learning rate varies cyclically within a range of values (instead of setting it to a fixed value). This method is motivated by the observation that a varying learning rate during training is beneficial overall and achieves remarkable classification accuracy. Furthermore, unlike adaptive learning rates, the Cyclical Learning Rate (CLR) method requires very little additional computation. Another observation is that increasing the learning rate might have a short term negative effect and yet achieve a long term positive effect.
The operation of the CLR method is straightforward: one sets minimum and maximum boundaries and the learning rate cyclically varies between these bounds. Several functional forms are possible, such as triangular window (figure 3.1), parabolic or sinusoidal window: all seem to produce equivalent results.

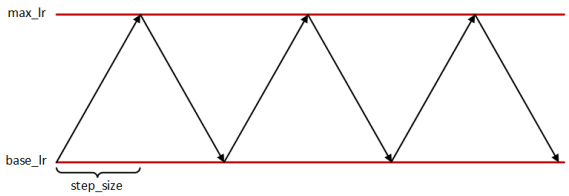
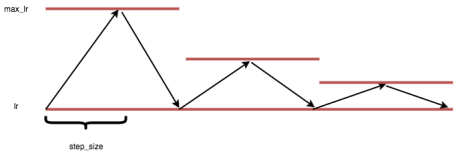
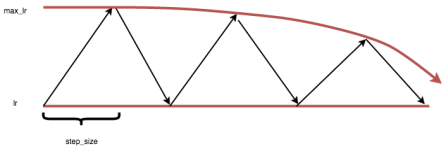


Figure 3.1: Triangular, or linear, learning rate policy. The parameter *stepsize* corresponds to the number of iterations in half a cycle.

It is also possible to change the maximum or the minimum bound as the iterations go on, thus adding another layer of complexity. Figure shows some possibilities.



(a) The amplitude of the cycle decreases by half after each period.



(b) The amplitude of the cycle decreases exponentially as the iterations go on.

Figure 3.2: Examples of different cyclical policies.

When considering the loss function topology we can understand why CLR methods work. Bengio et al. [6] argue that the difficulty in minimizing the loss arises from saddle points rather than sharp local minima. These saddle points can considerably slow down training, mostly because the objective function tends to be flat in many direction and give the illusory impression of the existence of a local minimum. It is also worth noting that the ratio of the number of saddle points to local minima increases exponentially with the dimensionality of the parameters. Consequently, increasing the learning rate has the beneficial effect of allowing a more rapid traversal (or escape) of those saddle points.
An even more empirical reason as to why CLR works is that it is likely that the optimum learning rate will be between the bounds, and near optimal learning rates will be used throughout training.
From tests it results that the final accuracy is quite robust w.r.t. the *stepsize* parameter, but a good choice for it should be between 2 and 10 times the number of iterations in an epoch. Regarding the values of the boundaries, a reasonable estimate can be obtained with one training run of the network for a few epochs. Simply running the training process for several epochs while letting the learning rate increase linearly from a very low value, thus obtaining a plot of the accuracy over learning rate: then note the values that correspond to an increase and a fall in

accuracy as minimum and maximum boundary values respectively.

3.2 Adaptive learning rates

The challenge of using learning rate schedules is that their hyperparameters have to be defined in advance and they depend heavily on the type of model and dataset. Another problem is that the same learning rate is applied to all parameter updates. If we have sparse data, we may want to update the parameters in different way.
Adaptive gradient descent algorithms such as Adagrad [7], Adadelta [29], and Adam [12], provide an alternative to classical SGD. These methods provide heuristic approach without requiring tuning hyperparameters for the learning rate schedule manually. The drawback is that calculating different learning rates at each iteration has a significant computational cost that the vanilla SGD does not have.
Adagrad [7] is an algorithm for gradient-based optimization that adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. Often, when the input instances are of very high dimension, in any any particular instance only a few features are non-zero and consequently these infrequently occurring features are highly informative. Thanks to this characteristic Adagrad has good performance with sparse data.
Adagrad modifies the learning rate η at each step t for every parameter, based on the past gradients that have been computed for that specific parameter. Using the same notation already presented in formula 1.2.1, the update rule at each iteration is:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \nabla_\theta L(\theta_t) \tag{3.2.1}$$

where η is the usual learning rate, G_t is a diagonal matrix where each diagonal element is the sum of the squares of the past gradients up to iteration t, $\odot$ in an element-wise matrix-vector multiplication, and $\nabla_\theta L(\theta_t)$ is the gradient of the loss function. Most implementations use a default value for η (eg. 0.01) and then they leave it as is. The accumulation of past squared gradients in the denominator also represents its main weakness, as the accumulated sum keeps growing during training. Consequently the learning rate, intended as the fraction, becomes smaller and smaller, eventually becoming infinitesimally small at which point the optimization process stops.

Adadelta [29] is derived from Adagrad, but it has the purpose of fixing the monotonically decreasing learning rate. Instead of accumulating all past squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients; the update rule now becomes:

$$\begin{aligned} E[\nabla_\theta L(\theta_t)^2]_t &= \gamma E[\nabla_\theta L(\theta_{t-1})^2]_{t-1} + (1-\gamma)\nabla_\theta L(\theta_t)^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[\nabla_\theta L(\theta_t)^2]_t + \epsilon}} \odot \nabla_\theta L(\theta_t) \end{aligned} \tag{3.2.2}$$

It is also possible to remove completely the learning rate from the equation and substitute it with the root mean squared error of parameter updates: η at the numerator can be replaced with $\sqrt{E[\nabla_\theta L(\theta_{t-1})^2]_{t-1} + \epsilon}$. In this way we do not even need to set a default learning rate, as it has been eliminated from the update rule.
Adam [12], as stated by Kingma and Ba, is designed to combine the advantages of Adadelta and momentum: it stores an exponentially decaying average of past squared gradients and also keeps an exponentially decaying average of past gradients. The first and second moment of the gradients, respectively m_t and v_t, are calculated as:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta L(\theta_t) \\ v_t &= \beta_2 m_{t-1} + (1-\beta_2)\nabla_\theta L(\theta_t)^2 \end{aligned} \tag{3.2.3}$$

Due to the initialization of m_t and v_t as vectors of 0's, the bias-corrected first and second moment estimates $\hat{m}_t$ and $\hat{v}_t$ are computed:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{3.2.4}$$

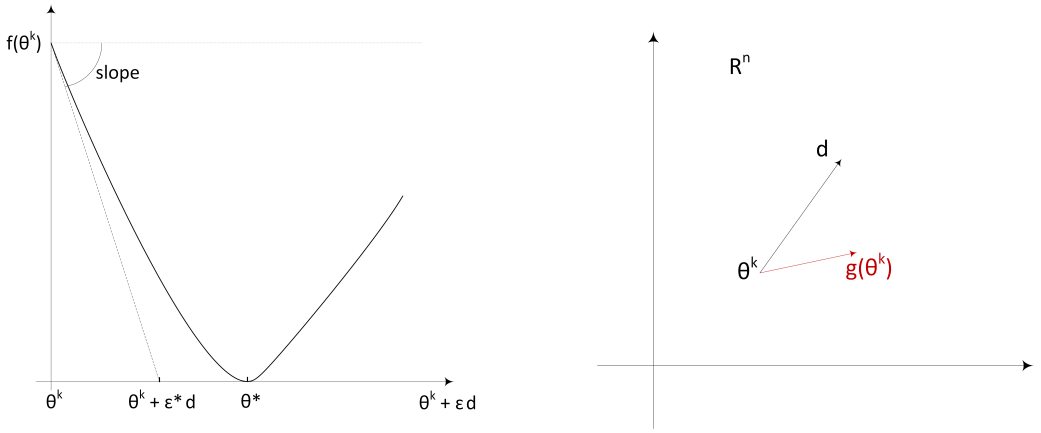
with β^t is denoted β to the power t.
At the end the Adam update rule is given by:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t \tag{3.2.5}$$

The authors suggest default values for η, β_1^t, β_2^t and ϵ which should work well for machine learning problems. These values, together with the optimization algorithm, are now present in the most important machine learning libraries and frameworks.

3.3 Alternative computation of learning rate: Adaptive Nesterov

The idea we tried to develop is halfway between doing a scheduling of the learning rate and computing the learning rate with an adaptive method. Of the first method it tries to maintain the low computational complexity (overhead) by not calculating a different learning rate for each parameter but using one that is the same for all, while from the second method it keeps the automatic calculation of the learning rate at each iteration.
The approach we are going to present is largely inspired from the so-called Polyak's stepsize [20], a subgradient method applied to a convex optimization problem. Without conducting a complete examination of the Polyak method, let's have a look at the general operation.
Suppose we have a convex function $f(\theta)$, $f : \mathbb{R}^n \to \mathbb{R}$ and we know the value of the function in a specific point $\theta^k \subseteq \mathbb{R}^n$ in the parameter space, as shown in Figure 3.3a.



(a) Convex function, the x axis represents a direction d in the parameter space parametrized by ϵ.

(b) Vectors d and $g(\theta^k)$ in the parameter space $\theta \subseteq \mathbb{R}^n$.

Figure 3.3: Convex function f and relative parameter space $\theta \subseteq \mathbb{R}^n$.

If we consider a general direction d in the parameter space parametrized by ϵ, then we can write the linear interpolation:

$$f(\theta^k + \epsilon_k\, d) \approx f(\theta^k) + g^T(\theta^k)\, \epsilon_k\, d \tag{3.3.1}$$

where $g(\theta^k) = \frac{\partial f}{\partial \theta}\Big|_{\theta^k}$, $d \in \mathbb{R}^n$ and $\epsilon \in \mathbb{R}$.
We know that there is an optimal value $f^* = f(\theta^*) = f(\theta^k + \epsilon^*\, d)$. We

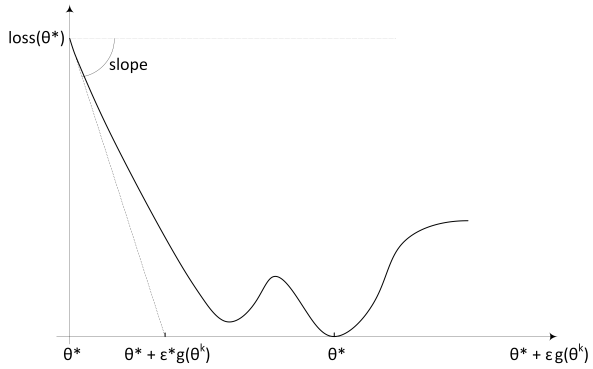
would like to calculate the value of ϵ^* which brings close to f^*:

$$\epsilon_k^* = \frac{|f(\theta^k) - f^*|}{g^T(\theta^k)\, d} \tag{3.3.2}$$

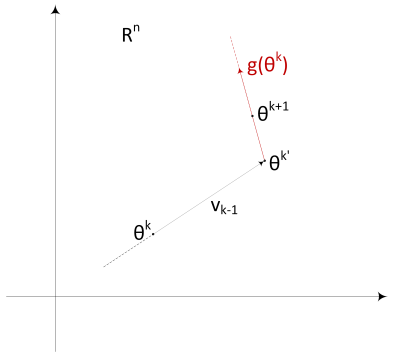
Actually, ϵ^* does not get us exactly to f^* since we have done a linear interpolation of a convex function, but for small values of $|\theta^k - \theta^*|$ it is a good approximation. If we consider $g(\theta^k)$ as the direction d, we obtain the Polyak's stepsize:

$$\epsilon_k^* = \gamma_k \frac{|f(\theta^k) - f^*|}{||g(\theta^k)||^2} \tag{3.3.3}$$

where $0 < \gamma_k < 2$ in the original formulation. This term is introduced since the approximation in formula 3.3.1 could be more or less accurate, depending from the topology of the function f. Polyak's stepsize works best when the optimal function value f^* is known, but the same method can also be applied to an estimate of f^*.
Our hope is that the same idea works with a highly non-convex function, such as the loss function of deep neural network. It should be noted that without the convexity requirement we loose all the convergence guarantees of the Polyak method [20].
Suppose we have a non-convex loss function $L(\theta)$, $L : \mathbb{R}^n \to \mathbb{R}$ and we are at the iteration k, where the parameter vector is $\theta^k \subseteq \mathbb{R}^n$, as in the figure below:



(a) Non-convex loss function, the x axis represents a direction $g(\theta^k)$ in the parameter space parametrized by ϵ.



(b) Vectors v_{t-1} and $g(\theta^k)$ in the parameter space $\theta \subseteq \mathbb{R}^n$.

Figure 3.4: Non-convex function L and relative parameter space $\theta \subseteq \mathbb{R}^n$.

The proposed algorithm is a hybrid between Nesterov momentum and Polyak's stepsize, hence the name *Adaptive Nesterov*. We maintain the past update vector as it is done in the momentum method (Section 1.3) and we include the computation of η at each iteration.

More in detail, suppose we are at the iteration k and we have the parameter vector θ^k and the past update vector $-v_{k-1}$: we move in the parameter space by a fraction α of the vector $-v_{k-1}$. In this temporary point $\theta^{k'} = \theta^k - \alpha v_{k-1}$ we compute the gradient of the loss function $g(\theta^{k'}) = \nabla_\theta L(\theta_{k'}) = \frac{\partial L}{\partial \theta}\big|_{\theta^{k'}}$, as it would happen in Nesterov momentum. At this point we have to decide how much to follow the direction along the vector $g(\theta^{k'})$, namely we have to choose a value for η.
At this point we use the equation 3.3.3 for computing the value of η, setting f^* to 0 since we suppose that there exists a configuration of parameters that correctly classifies all the data in the training set. This is true under mean squared error or cross-entropy loss functions [14], the latter used in our implementation. We are aware that the approximation could be very rough due to the non-convexity of the loss function and the assumption on f^*. Thus, the parameter η at each iteration k is computed as:

$$\eta_k = \gamma_k \frac{|L(\theta^{k'})|}{||g(\theta^{k'})||^2} \tag{3.3.4}$$

where γ_k expresses the degree of confidence in following the gradient, similarly to what happens in equation 3.3.3. In our tests, this value was actually fixed to 1.
At this point we have done a step η_k along the direction $g(\theta^{k'})$, reaching the point θ^{k+1}. Finally we update the vector v_k to the current iteration. The update rule for the algorithm just presented is:

$$\begin{aligned} \eta_k &= \gamma_k \frac{L(\theta^k - \alpha v_{k-1})}{||g(\theta^k - \alpha v_{k-1})||^2} \\ v_k &= \alpha\, v_{k-1} + \eta_k\, g(\theta^k - \alpha v_{k-1}) \\ \theta_{t+1} &= \theta_t - v_t \end{aligned} \tag{3.3.5}$$

where we set α equal to 0.9 and γ_k equal to 1.
The relative pseudocode follows: Note that the learning rate hyperparameter disappears from the parameters needed by the algorithm because it is dynamically calculated at each iteration over a new minibatch.

3.4 Tests of Adaptive Nesterov

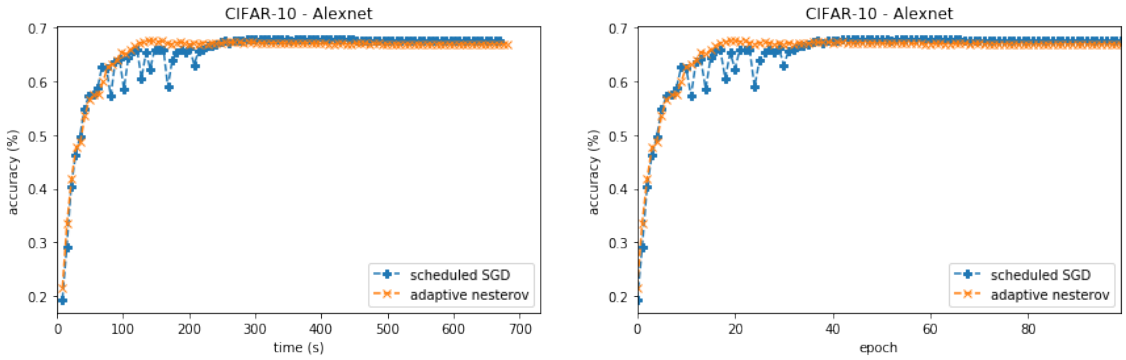
The setup used to perform the tests of the *adaptive Nesterov* method is the same to the one previously presented in Section 2.2. In particular, the three network architectures (reduced AlexNet, VGG16 and

Algorithm 5 Stochastic Gradient Descent with Nesterov momentum and adaptive stepsize

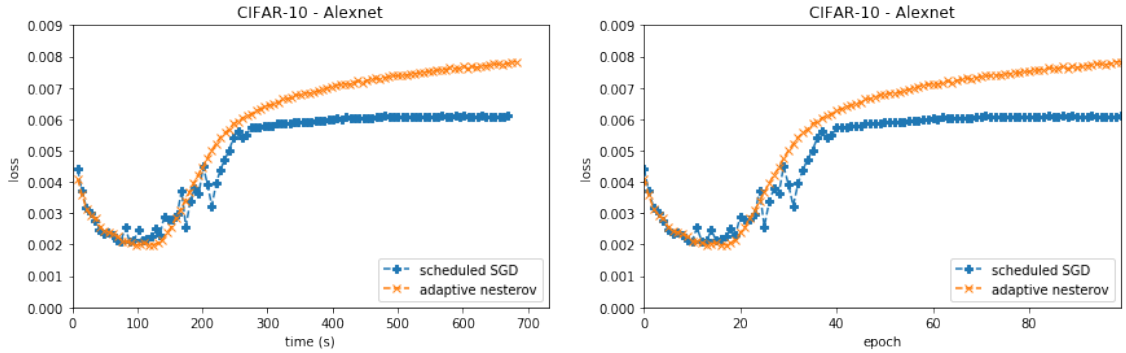
parameters: $\eta > 0 \in \mathbb{R}^+$, $T \in \mathbb{N}$
initialize $\mathbf{w}^{(1)} \in \mathbb{R}^{|E|}$ from a zero mean distribution
for $t = 1, 2, ..., T$ **do**
    sample $(\mathbf{x}^{(bs)}, \mathbf{y}^{(bs)}) \sim D$
    $\mathbf{w}^{(t')} = \mathbf{w}^{(t)} - \alpha\, \mathbf{v}_{t-1}$
    calculate $\mathbf{g}(\mathbf{w}^{(t')}) = backpropagation(\mathbf{x}^{(bs)}, \mathbf{y}^{(bs)}, \mathbf{w}^{(t')}, (V, E, \sigma))$
    $\eta_t = \gamma_t \frac{Loss(\mathbf{w}^{(t')})}{||\mathbf{g}(\mathbf{w}^{(t')})||^2}$
    $\mathbf{v}_t = \alpha\, \mathbf{v}_{t-1} + \eta_t\, \mathbf{g}(\mathbf{w}^{(t')})$
    update $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \mathbf{v}_t$
output: $\mathbf{w}^{(T+1)}$

ResNet34) and the dataset on which they were trained remained unchanged.
Regarding the standard *minibatch* SGD with Nesterov momentum used as reference, a learning rate schedule was executed according to recent implementations: $\eta = 0.1$ for the first 40 epochs, then it was set to 0.01 for the following 30 epochs and to 0.001 for the remaining 30 epochs. The momentum term was set to 0.5. Compared to what was done in Chapter 2, the minibatch size was set to 512 for all configurations, since we are only interested on the impact that the learning rate has on the different methods.
The only change made compared to the pseudocode just presented (Algorithm 5) is the clipping of $||g(\theta^{k'})||^2$ in the range $[10^{-4}, 10^{10}]$ in order to prevent the divergence of network parameters, and the clipping of η_t in the range $[0.001, 0.1]$ to have a fair comparison with the standard SGD with scheduled η. The following figures refer to the test dataset, a fraction of 10,000 samples taken from the CIFAR-10 dataset.
Figure 3.5 reports the results of our experiments when comparing momentum SGD with scheduled learning rate (blue curve) against the adaptive Nesterov method (orange curve), both trained on AlexNet architecture. First we can observe that the accuracy achieved by the two methods is very similar: 67.82% and 67.65% respectively. But what changes is when these values are reached: the standard SGD method reaches it after about 320 seconds while the adaptive method after only 140 seconds. Another interesting thing to observe is that the computation of η_t at each iteration does not seem to have much influence on the computing time, since the difference to execute 100 epochs is only 17 seconds. This is positive since we wanted an adaptive method with low overhead.

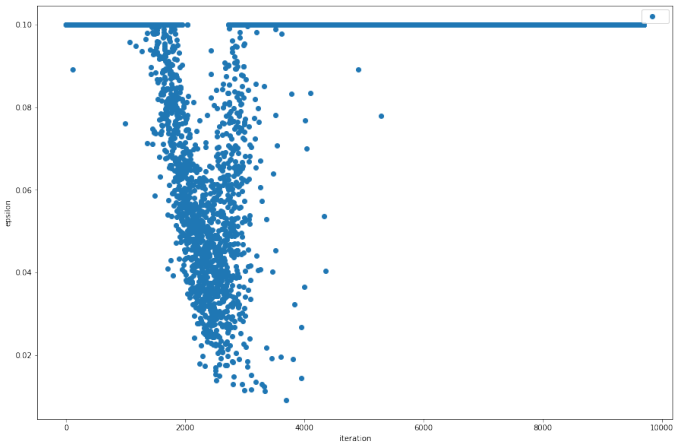
When looking at the loss function trend (Figure 3.5b) the two methods seem rather similar in a first phase, but they differ in the final training phase. The higher values of the loss assumed by the adaptive Nesterov method can be justified by looking at the ϵ values: the higher (clipped) values of ϵ make the optimization process less conservative, thus worsening the objective function. However, this negative effect does not influence the general accuracy.



(a) Test accuracy.



(b) Test loss.

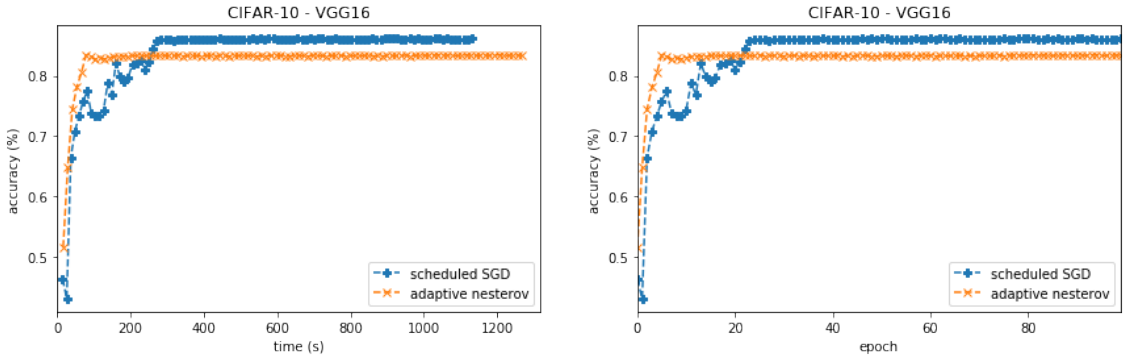


(c) Value of ϵ as the number of iterations increases.

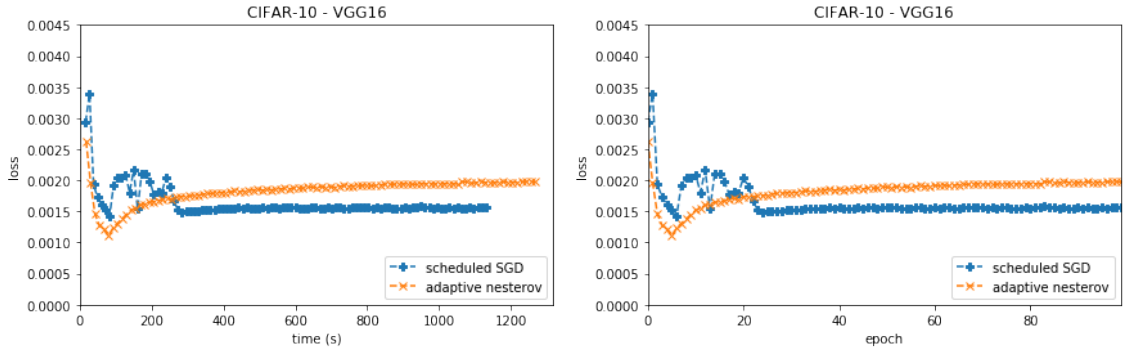
Figure 3.5: Performance difference between SGD with scheduled learning rate and the adaptive nesterov method. AlexNet.

The positive results obtained with AlexNet are not confirmed for

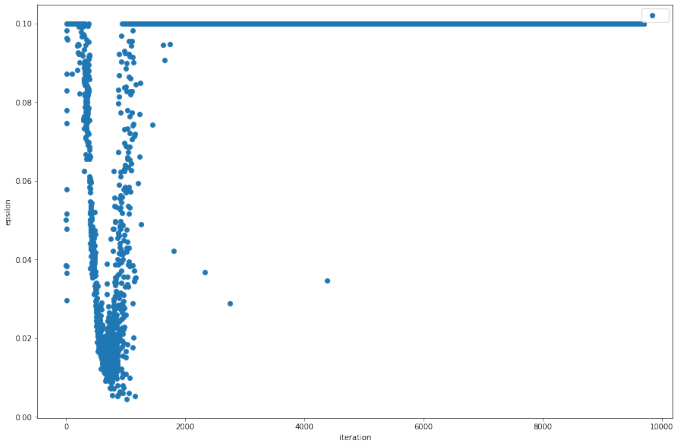
the VGG16 network, as we can see by looking at Figure 3.6. There is a noticeable difference in accuracy between the scheduled SGD and the adaptive method: it seems like the optimization process stopped very early, after a rapid attainment of what could be a local minimum. Looking at the loss function it would seem that the adaptive method has a clear advantage, although this does not translate into better accuracy. The values assumed by ϵ during training are partially unexpected, and are similar to what happens with AlexNet.



(a) Test accuracy.



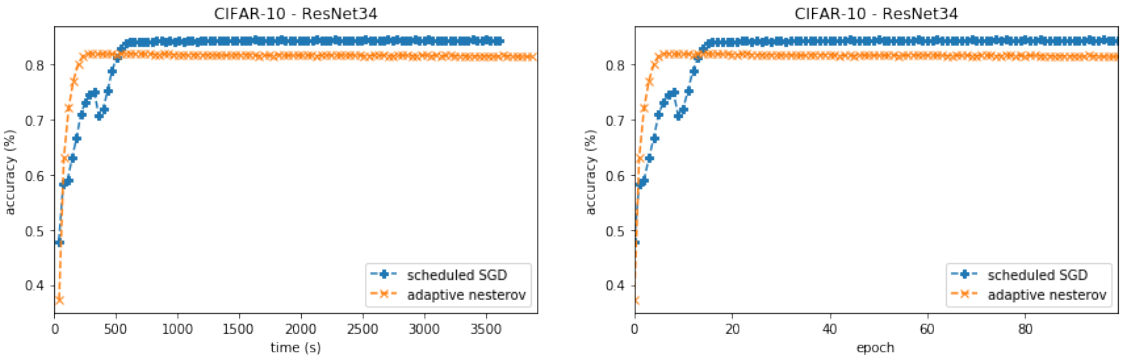
(b) Test loss.



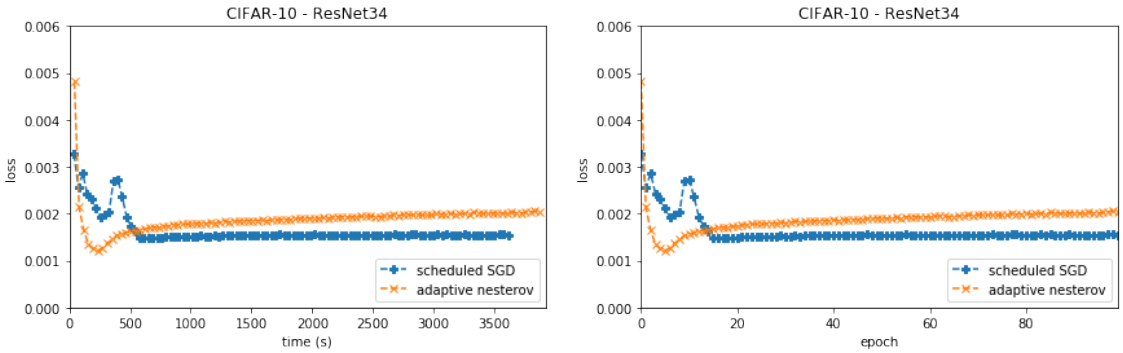
(c) Value of epsilon as the number of iterations increases.

Figure 3.6: Performance difference between SGD with scheduled learning rate and the adaptive nesterov method. VGG16.

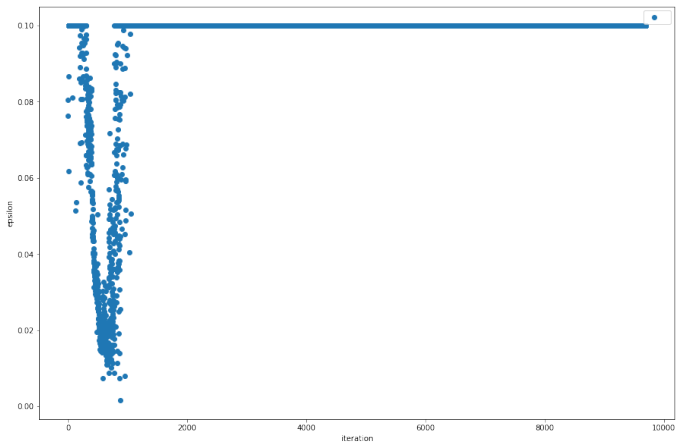
Looking at the results obtained for ResNet34, Figure 3.7, the same considerations made for VGG16 remain valid. A difference between ResNet34 and VGG16 (compared to AlexNet) is execution time, slightly larger for the adaptive method. This can be explained by the fact that the calculation of sums and vector products takes longer as the size of the vectors increase, size which depends on the number of network parameters.



(a) Test accuracy.



(b) Test loss.



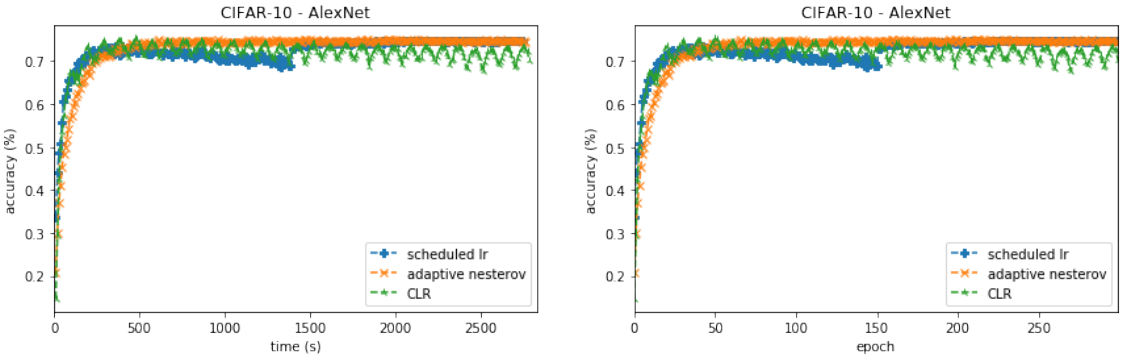
(c) Value of epsilon as the number of iterations increases.

Figure 3.7: Performance difference between SGD with scheduled learning rate and the adaptive Nesterov method. ResNet34.

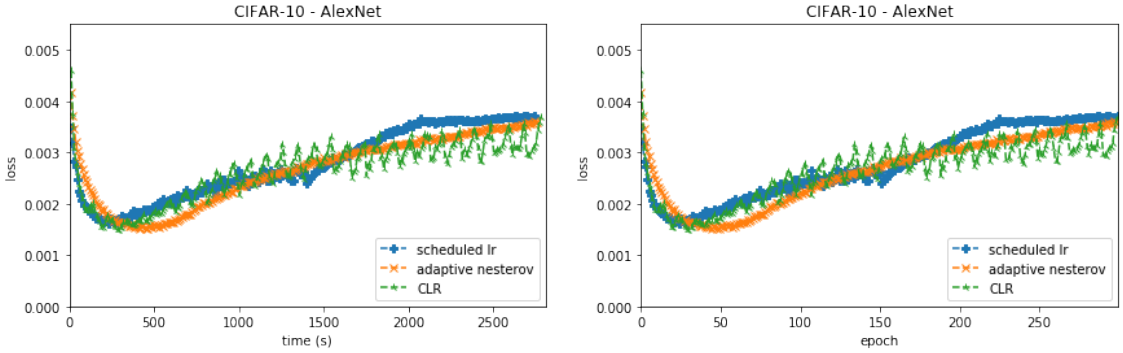
3.5 Improving the performance

Our goal is to see how our adaptive method behaves when the number of epochs increases and data augmentation is added, that is trying to make the network perform better in the classification of data (as already done for the minibatch persistency method).
We also want to see if the disadvantages of this method remain: a maximum accuracy lower than the standard method, despite a better loss function. We want to understand if the behaviour of the learning rate during the training phase (constantly high value in the second half of training) will repeat with different training conditions.
The experimental setup remains the one presented in Section 2.4 except for the use of all three network architectures considered so far (AlexNet, VGG16 and ResNet34). Data augmentation is performed by a random horizontal flip and a random translation of 4 pixels. Scheduling of the learning rate for the standard SGD method was: lr=0.1 for the first 150 epochs, lr=0.01 from 150 to 225 epochs and then lr=0.001 for the last 75 epochs. It should be noted that this particular scheduling of the learning rate is the one that gave us the best results, considering we want the best performing configuration against which to compare our adaptive method. The momentum coefficient and the other parameters remained unchanged.
Since our method is a possible alternative to either scheduling the learning rate or using an adaptive method, we decided to compare it also with the CLR policy (Section 3.1) applied to the standard SGD algorithm with Nesterov momentum. Remember that CLR does not require any additional computation since the learning rate at each iteration is given by a simple triangular function, and from this point of view it has an advantage over our method. The cycle stepsize remains as a parameter to be tuned, although the authors claim that the final accuracy is quite robust to cycle length. As suggested, we used a value of 10 times the number of iterations in an epoch as cycle lenght.
Summarizing, the following results present the behaviour of our designed adaptive Nesterov method against a scheduled and a cyclical learning rate algorithms, both using SGD with momentum.
Figure 3.8 presents the result of our experiments for minibatch sizes $m = 512$ and AlexNet architecture. The accuracy achieved by our adaptive method is very similar but slightly higher in the final phase than the scheduled learning rate approach. The cyclical learning rate produces

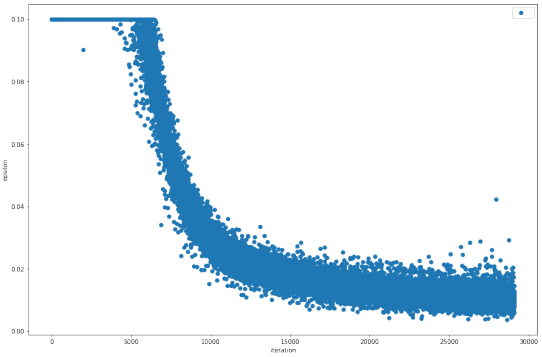
slight fluctuations in the loss function which reflects in broad fluctuations in the measured accuracy. As for the loss function, the adaptive



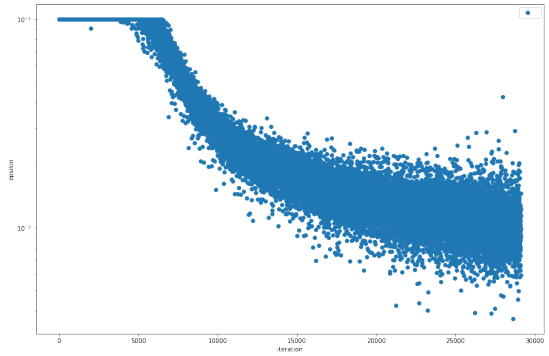
(a) Test accuracy.



(b) Test loss.



(c) Value of ϵ as the number of iterations increases, linear scale.



(d) Value of ϵ as the number of iterations increases, logarithmic scale.

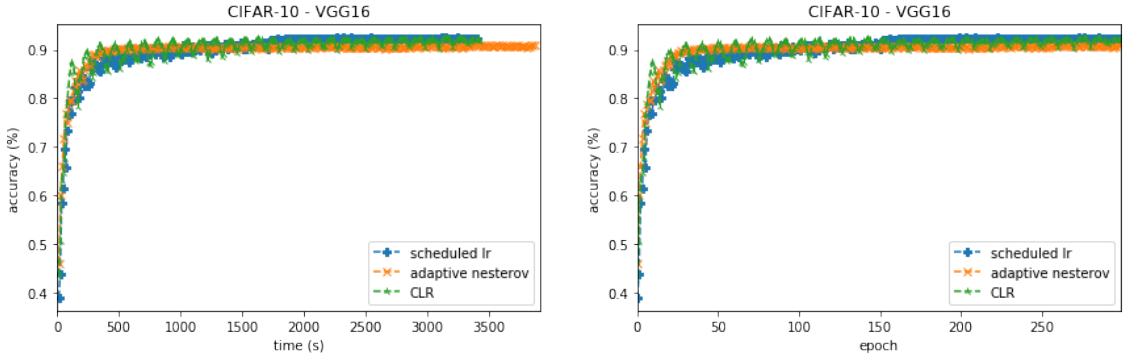
Figure 3.8: Performance difference between scheduled learning rate, adaptive nesterov method and CLR. AlexNet.

method is better than the scheduled η, with a lower minimum value. The behaviour of the stepsize during the training drastically changes with respect to what we observed in the tests of Figure 3.5: a wide range of values are assumed and clipping is very limited. The learning rate has a decreasing trend similar the scheduled one, sign of a more conservative convergence of the optimization process.

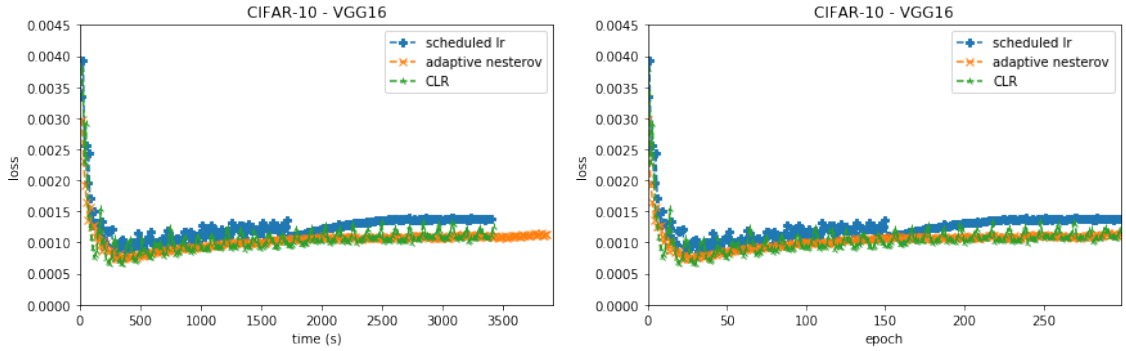
The considerations made above for AlexNet remain valid also for

VGG16 and ResNet34 (Figures 3.9 and 3.10 respectively), except for minor details. Our adaptive method suffers of a greater computational time with respect to the other two approaches, due to the calculation of η_k from equation 3.3.5, which is no longer negligible when the vectors contain millions of elements.

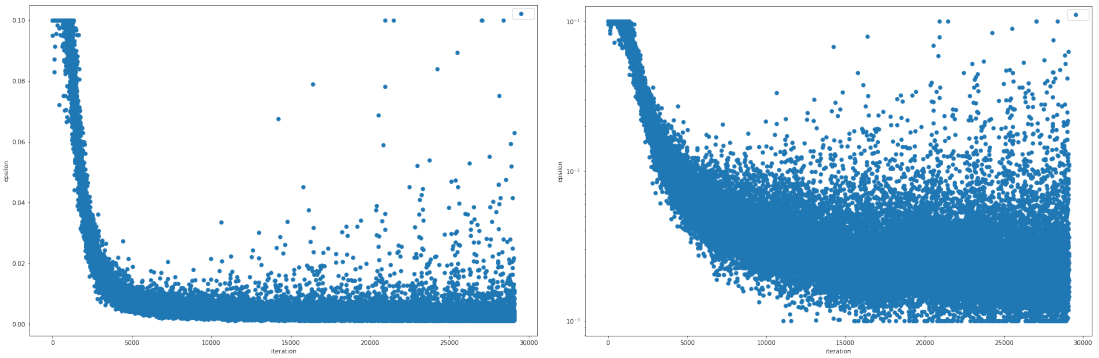
We can also notice that the behaviour of the loss function is very similar



(a) Test accuracy.



(b) Test loss.



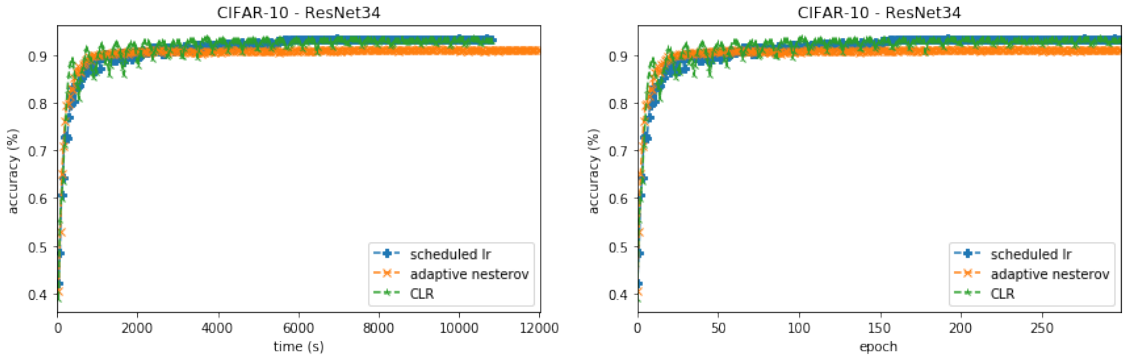
(c) Value of ϵ as the number of iterations increases, linear scale.

(d) Value of ϵ as the number of iterations increases, logarithmic scale.

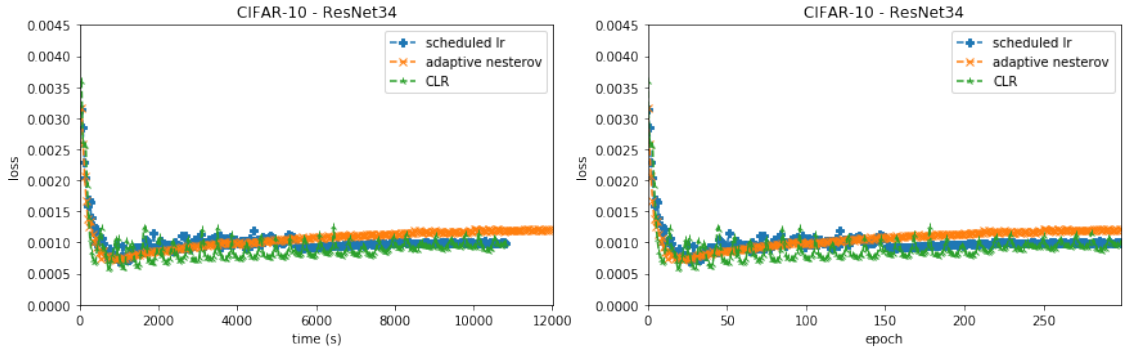
Figure 3.9: Performance difference between scheduled learning rate, adaptive Nesterov method and CLR. VGG16.

in each of the three configurations, but for the undulatory pattern typical of CLR (Figures 3.9b and 3.10b). Figures 3.9d and 3.10d relative to VGG16 and ResNet34, respectively, show the value of ϵ in logarithmic

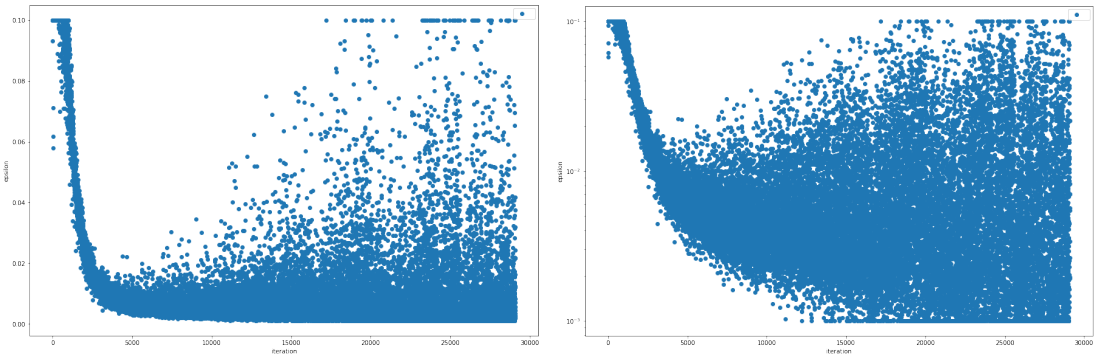
scale as the number of iterations increases: the range of values assumed in the second half of training is very spread out, possible compromising the achievement of a good minimum of the loss function.



(a) Test accuracy.



(b) Test loss.



(c) Value of ϵ as the number of iterations increases, linear scale.

(d) Value of ϵ as the number of iterations increases, logarithmic scale.

Figure 3.10: Performance difference between scheduled learning rate, adaptive Nesterov method and CLR. ResNet34.

Chapter 4

Conclusions

Machine learning and, in particular, deep learning are fields in strong growth in recent years and around which much interest has been concentrated. Important steps have already been taken towards understanding the functioning of the neural networks, the role played by the learning rate during the training, and the link between the loss function topology and the ability to generalize of the network.
However, it is still possible to make important developments towards a deeper understanding of neural networks, or improving accuracy and computational times with relatively simple ideas, as evidenced by recent publications of Smith et al. [22] or Luschi et al. [18]. Our techniques must therefore be framed in this context.
Our first method, called *minibatch persistency*, derives from the idea that large minibatches contain a lot of information about the training set, and is aimed at exploiting the growing power of parallel architectures (GPUs). This technique consists in executing K consecutive SGD iteration using the same minibatch, $K = 1$ being the standard update rule. From the various tests executed we observed that using a value of K greater than 2 often leads to a rapid overfit and often it is not beneficial. When $K = 2$ the positive effect of reusing a minibatch is more consistent across different configurations of parameters and architectures. We can therefore say that it is worth considering using this technique when training a neural network, in addition to those already consolidated.
The second method, called *adaptive Nesterov*, is a transposition of the Polyak's stepsize to the highly non-convex environment typical of the deep neural network loss functions, where the direction of movement is that of the Nesterov method. The resulting algorithm is a hybrid between a scheduled learning rate method and an adaptive one: it tries to maintain low computational complexity while automatically calculating the stepsize at each iteration, thus removing a tunable hyperparameter. The resulting performances are very close to those of the compared established methods, especially with more advanced configurations. This technique can be a valid alternative to the scheduled and the adaptive methods.

From the various tests performed it is clear that two architectures, VGG16 and ResNest34, affect the calculation time more than AlexNet. Probably with a more performing architecture (multiple GPUs instead of one), the total computing time gap between the various methods would be reduced as it happens for AlexNet. Regarding the values assumed by accuracy and loss function, there would be no differences as they are dependent on the complexity of the model class (aside from other parameters).
Future work may concern hybrid techniques using *minibatch persistency*, such as reusing the minibatch only in the initial part of training or changing the stepsize in the subsequent iterations to mitigate overfit. For *adaptive Nesterov* one could improve and develop the formula with which it is calculated the stepsize at iteration k, for example deriving from what was done by Hayashi et al. [14]. It would still be appropriate to conduct more tests on different architectures, datasets and parameters, to fully understand the impact of these methods.

Bibliography

[1] Peter L. Bartlett and Shai Ben-David. Hardness results for neural network approximation problems. *Theoretical Computer Science*, 284(1):53 – 66, 2002. Computing Learining Theory.

[2] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533, 2012.

[3] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012.

[4] Léon Bottou. On-line learning in neural networks. chapter On-line Learning and Stochastic Approximations, pages 9–42. Cambridge University Press, New York, NY, USA, 1998.

[5] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *CoRR*, abs/1606.04838, 2016.

[6] Yann N. Dauphin, Harm de Vries, Junyoung Chung, and Yoshua Bengio. Rmsprop and equilibrated adaptive learning rates for non-convex optimization. *CoRR*, abs/1502.04390, 2015.

[7] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *ArXiv e-prints*, December 2015.

[9] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[10] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *CoRR*, abs/1705.08741, 2017.

[11] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *CoRR*, abs/1609.04836, 2016.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[13] Adam R. Klivans and Alexander A. Sherstov. Cryptographic hardness for learning intersections of halfspaces. In *47th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2006*, pages 553–562, 12 2006.

[14] Jayanth Koushik and Hiroaki Hayashi. Improving stochastic gradient descent with feedback. *CoRR*, abs/1611.01505, 2016.

[15] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[17] Zijin Luo. pytorch-cifar10, 2018.

[18] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *CoRR*, abs/1804.07612, 2018.

[19] Yurii Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Dokl. Akad. Nauk SSSR*, 269:543–547, 1983.

[20] Boris Polyak. Introduction to optimization. *Optimization Software*, 1987.

[21] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Netw.*, 12(1):145–151, January 1999.

[22] L. N. Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472, March 2017.

[23] CS231N Staff. Convolutional neural networks for visual recognition, 2016.

[24] Richard S. Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum, 1986.

[25] Luke Taylor and Geoff Nitschke. Improving deep learning using generic data augmentation. *CoRR*, abs/1708.06020, 2017.

[26] D. Randall Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Netw.*, 16(10):1429–1451, December 2003.

[27] Larry S. Yaeger, Richard F. Lyon, and Brandyn J. Webb. Effective training of a neural network character classifier for word recognition. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems 9*, pages 807–816. MIT Press, 1997.

[28] Wei Yang. pytorch-classification, 2017.

[29] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

List of Figures

List of Tables