Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Unconventional Training of Deep Neural Networks

Supervisor

Matteo Fischetti

Candidate

Matteo Stringher

Academic Year 2018/19
April 1, 2019

Summary

Deep learning is increasingly drawing attention, since recent discoveries have showed high potential. Researchers from any field contribute every day to a growing number of papers in the field. Major players in the IT field, such as Google, Microsoft and Facebook, publish their ideas and participate to the growth of Deep Learning as well. The overall work has led to great improvements in computer vision, natural language processing, audio recognition, and many other tasks.

In this work we want to address the training problem in an unconventional way to get a different point of view from the traditional one. The loss curve is still poorly understood: theoretical and experimental work is trying to better understand it, but it seems there is still much uncovered and the number of growing publications adds new insights. The highly dimensional space and the deep structure make it harder to understand than traditional machine learning techniques. Therefore, we try unconventional and easier procedures trying to verify some conjectures.

Chapter 2 introduces the problem and studies shape and differences in minima of the loss curve. This works leads to slightly better accuracy. In chapter 3 we try to train deep neural networks with Simulated Annealing, by random moves, without using the derivative. Chapter 4 uses a mixed approach between Stochastic Gradient Descent and Simulated Annealing, trying to converge in a faster way to a mininum. This last part leads to a better accuracy with considerable improvements to standard techniques. Full code is available at `github.com/strmet/thesis`.

Contents

Chapter 1

Deep Learning

Machine learning is one branch of artificial intelligence. It powers many systems such as spam filtering, product recommendations, online fraud detection, battery saving in smartphones, and other tasks. The growth in the research community has been followed by a huge rise in the number of projects in the industry leveraging the new technology.

Deep learning is a subset of Machine Learning, based on learning data representation through the use of neural network architectures, specifically deep neural networks. Inspired by human processing behaviour, deep neural networks have set new state-of-art results in speech recognition, visual object recognition, object detection, and many other domains. In contrast with previous methods in machine learning, deep neural networks allow to extract features and classify sets of data by itself. Before their introduction, constructing a pattern-recognition or machine-learning system required careful engineering and considerable domain expertise to design a feature extractor that transformed the raw data into a suitable internal representation or feature vector [23]. A neural network is made of several layers with different characteristics, mostly convolutional and linear. Each of them recognises different patterns and features in the input set. Zeiler and Fergus [42] have introduced a visualization technique that reveals the input stimuli that excite individual feature maps at any layer in the model, showing the evolution of features during training. Their work showed that very first layers learn basic features such as lines or corners. The information is then combined along with the depth of the network, allowing complex structures, e.g., cat or dog shape. The key aspect of deep learning

is that these layers of features are not designed by human engineers: they are learned from data using a general-purpose learning procedure. This characteristic is bringing a lot of interest in the field.

1.1 Machine Learning foundations

The learning model for a classificaiton problem needs the following information:

- **Domain set**: An arbitrary set called $\mathcal{X}$. This is the input set that must be predicted. The domain is represented by a number of features p. For example in computer vision, when classifying objects, each pixel of the image is a feature.

- **Label set**: $\mathcal{Y}$ the set of possible outputs.

- **The learner's output**: The learner is requested to output a prediction rule, $h : \mathcal{X} \to \mathcal{Y}$. This function is called a predictor, an hypothesis; most likely it is referred to as the classifier.

- **Training data**: $S = ((x_1, y_1) \dots (x_m, y_m))$ is a finite sequence of pairs in $\mathcal{X} \times \mathcal{Y}$. This is the input needed by the learner to train the model.

- **Validation data**: has the same structure of the training data, but it is not used during training to make decisions; instead, its purpose is to assess the predictive capability of the model. Usually the data available are split according to the 80/20% rule, i.e., 80% for training and the remainder for validation.

- **A simple data-generation model**: The examples in the dataset are generated by some probability distribution $\mathcal{D}$ over $\mathcal{X}$. This could be any arbitrary probability distribution. Moreover, a labelling function is needed, such as $f : \mathcal{X} \to \mathcal{Y}$. This function is unknown to the learner: this is what is trying to approximate. In summary, each pair in training data S is generated by first sampling a point x_i according to $\mathcal{D}$ and then labelling it as $y_i = f(x_i)$.

- **Measure of success**: We define the error of a classifier to be the probability that it does not predict the correct label on a random data point generated by the aforementioned underlying distribution. Then we can define the error of a prediction rule h as

$$L_{\mathcal{D},f}(h) \stackrel{\text{def}}{=} \mathop{\mathbb{P}}_{x\sim\mathcal{D}}[h(x) \neq f(x)] \stackrel{\text{def}}{=} \mathcal{D}(\{x : h(x) \neq f(x)\}) \tag{1.1}$$

In other terms: the error of such h is the probability of randomly choosing an example x for which $h(x) \neq f(x)$. $L_{\mathcal{D},f}(h)$ is often called *generalization error*, or *risk*.
Given any set $\mathcal{H}$ and some domain $\mathcal{Z}$, let ℓ be any function $\mathcal{H} \times Z$ to the set of of nonnegative real numbers $\ell : \mathcal{H} \times Z \to \mathbb{R}_+$. For our prediction problems, we have that $Z = \mathcal{X} \times \mathcal{Y}$. We now define again the *risk function* to be the expected loss of a classifier $h \in \mathcal{H}$ w.r.t. a probability distribution D over Z, namely,

$$L_{\mathcal{D}}(h) \stackrel{\text{def}}{=} \mathop{\mathbb{E}}_{z\sim\mathcal{D}}[\ell(h, z)] \tag{1.2}$$

However, we can not access the loss function since $\mathcal{D}$ is unknown. Similarly, we can define the *empirical risk* to be the expected loss over a given sample $S = (z_1, \ldots, z_m) \in Z^m$, such that:

$$L_S(h) \stackrel{\text{def}}{=} \frac{1}{m}\sum_{i=1}^{m} \ell\left(h, z_i\right) \tag{1.3}$$

Many attempts have been done to bound $L_S(h)$ as close as possible to $L_{\mathcal{D}}(h)$, such as in [30] and with a focus in deep learning in [29]. The aim is to guarantee a generalization gap to be small for a given S and/or to approach zero with *a fixed model class* as the size of the training set increases, i.e., $|S|$ gets bigger. Namely, the generalization gap is defined as:

$$\text{generalization gap} \stackrel{\text{def}}{=} L_{\mathcal{D}}(h) - L_{\mathcal{S}}(h) \tag{1.4}$$

1.2 Feedforward neural networks: The structure

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 $t = 1, \ldots, T$.

The input layer, V_0, contains $n + 1$ nodes (or 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, which represents the bias, always outputs 1. We denote by $v_{t,i}$ the i-th neuron of the t-th layer, and by $a_{t,i}(\mathbf{x})$ 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 neuron i in layer $t + 1$ is given by:

$$\begin{aligned} a_{t+1,i}(\mathbf{x}) &= \sum_{r:(v_{t,r},v_{t+1,i})\in E} w((v_{t,r}, v_{t+1,i}))o_{t,r}(\mathbf{x}) \\ o_{t+1,i}(\mathbf{x}) &= \sigma(a_{t+1,i}(\mathbf{x})) \end{aligned} \tag{1.5}$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is the activation function of the neuron, e.g. 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: 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; see 1.1 for an illustration.

1.3 Training a neural network

Once we have specified a neural network by (V, E, σ, w), we obtain a function $h_{V,E,\sigma,w} : \mathbb{R}^{|V_0|-1} \to \mathbb{R}^{|V_T|}$. Usually the hypothesis of a neural network is defined by fixing the graph as well as the activation function σ and letting the hypothesis class be all functions of the form $h_{V,E,\sigma,w}$ for some $w : E \to \mathbb{R}$. The tuple (V, E, σ) is often called the architecture of the network. The hypothesis class by is denoted as

$$\mathcal{H}_{V,E,\sigma} = \{h_{V,E,\sigma,w} : w \text{ is a mapping from } E \text{ to } \mathbb{R}\} \tag{1.6}$$

Understanding the capability of such networks, i.e., what functions hypotheses in $\mathcal{H}_{V,E,\sigma}$ can implement, is a very interesting topic. Hornik [12]

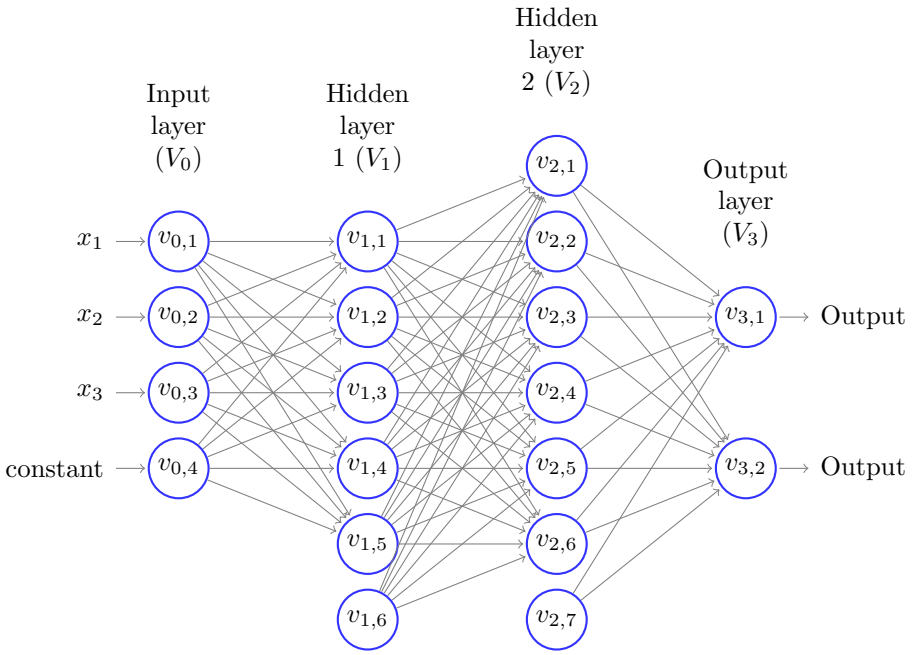


Figure 1.1: A network with two hidden layers

showed that any arbitrary function can be drawn with a single hidden layer network, but an exponential number of neurons is needed. Then, the research community started to focus on deeper networks with multiple layers, which have been proven to be very effective. Such networks are trained using Stochastic Gradient Descent (SGD) and backprogation.

1.3.1 Stochastic Gradient Descent

The problem can be thought in a similar way without focusing on the graph structure of neural networks. Since E is a finite set, we can think of the weight function as a vector $\mathbf{w} \in \mathbb{R}^{|E|}$. Suppose the network has n input neurons and k output neurons, and denote by $h_{\mathbf{w}} : \mathbb{R}^n \to \mathbb{R}^k$ the function calculated by the network if the weight function is defined by w. Then we call $\Delta\left(h_{\mathbf{w}}(\mathbf{x}), \mathbf{y}\right)$ the loss of predicting $h_{\mathbf{w}}(\mathbf{x})$ when the target is $\mathrm{y} \in \mathcal{Y}$. This function must be continuous and differentiable. The aim is to minimize the following value:

$$L_{\mathcal{D}}(\mathbf{w}) = \mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}} \left[\Delta\left(h_{\mathbf{w}}(\mathbf{x}), \mathbf{y}\right)\right] \tag{1.7}$$

The gradient of a differentiable function $f : \mathbb{R}^d \to \mathbb{R}$ at w, denoted by $\nabla f(\mathbf{w})$ is the vector of partial derivates of f, namely, $\nabla f(\mathbf{w}) = \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w_d}\right)$.

Gradient descent is an iterative algorithm: starting from an initial value it updates the parameters until the function approaches a minimum. The update step is

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla f\left(\mathbf{w}^{(t)}\right) \tag{1.8}$$

where η is the so called learning rate.

The Gradient Descent algorithm can be modified in order to be able to train faster complex network structures. Stochastic Gradient Descent (SGD) only needs the expected value of the random vector to be equal to gradient direction. In practice, when using SGD only a subset of the training data (called minibatch) is used in order to perform a step. In the following the number of examples in the minibatch will be called minibatch size. The general procedure is formalized in Algorithm 1. Usually, parameter η is decreased during the computation in order to avoid fluctuations close the minimum (later on referred as scheduled SGD).

Other successful attempts have been done by Smith in [37], where the network has been optimized using a cyclic learning rate (triangular and exponential policy). Such an approach where the learning rate cyclically varies between these bounds is sufficient to obtain near optimal classification results, often with fewer iterations. Moreover, this approach has no additional overhead. Decreasing the learning rate is not the only way to reach the minimum. Smith et al. [38] suggest increasing the batch size, a technique that allows to reduce the noise due to stochasticity. Even if using large batches has been proven to reduce the validation accuracy [18], they have proven that similar results to scheduled SGD can be achieved increasing the batch size, whilst using fewer parameter updates.
SGD is the de facto standard when training neural networks.

1.3.2 Backpropagation

Since no closed form can be found to calculate the overall gradient, an iterative algorithm is needed to compute all the partial derivatives. Backpropagation leverages the chain rule to compute gradients for each layer V_t. The SGD algorithm must be modified as specified in Algorithm 2 and 3. After the forward regular pass where some values are stored, the gradient is backpropagated.

Algorithm 1 SGD for minimizing $L_{\mathcal{D}}(\mathbf{w})$

Parameters: Scalar $\eta > 0$, integer $T > 0$
Initialize: $\mathbf{w}^{(1)} = 0$
for t = 1, 2, . . . ,T **do**
  sample $z \sim \mathcal{D}$
  pick $\mathbf{v}_t \in \partial \ell \left(\mathbf{w}^{(t)}, z\right)$
  update $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \mathbf{v}_t$
end for
Output: $\overline{\mathbf{w}} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{w}^{(t)}$

The proposed algorithms use the global derivative, but other attempts to solve the problems have been tested. In fact, the network can be efficiently trained computing the gradient on local loss functions at each layer. This greedy method allows more parallelizable implementations, since there is no need to backpropagate the gradient.

Algorithm 2 SGD for Neural Networks

Parameters: Scalar $\eta > 0$, integer $T > 0$
Input: Layered graph (V, E), differentiable function $\sigma : \mathbb{R} \to \mathbb{R}$
$\mathbf{w}^{(1)}$ = random_initialization()
for t = 1, 2, . . . , T **do**
  Sample $(\mathbf{x}, \mathbf{y}) \sim \mathcal{D}$
  Calculated gradient $\mathbf{v}_i =$ backpropagation $(\mathbf{x}, \mathbf{y}, \mathbf{w}, (V, E), \sigma)$
  Update $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \eta_i \mathbf{v}_i$
end for
Output: $\overline{\mathbf{w}}$ is the best performing $\mathbf{w}^{(i)}$ on the validation set

Different loss functions are used in deep learning to train the network, such as the L1-Loss or the MSE-Loss, but the most common is the cross-entropy. Let's define ℓ_k as the k-th output of the last layer of the network and recall that y_{ik} is the expected value of x_i. Then the Cross Entropy is:

$$L(\mathbf{w}) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log \ell_k (x_i) \tag{1.9}$$

Algorithm 3 Backpropagation

Input: example (x, y), weight vector $\mathbf{W}$, layered graph (V, E), activation function $\sigma : \mathbb{R} \to \mathbb{R}$

Initialize: denote layers of the graph $V_0, \ldots, V_T$ where $V_t = \{v_{t,1}, \ldots, v_{t,k_t}\}$ define $W_{t,i,j}$ as the weight of $(v_{t,j}, v_{t+1,i})$

Forward pass:
Set $\mathbf{o}_0 = \mathbf{x}$
for $t = 1, \ldots, T$ **do**
    for $i = 1, \ldots, k_t$ **do**
        Set $a_{t,i} = \sum_{j=1}^{k_{t-1}} W_{t-1,i,j} o_{t-1,j}$
        Set $o_{t,i} = \sigma(a_{t,i})$
    end for
end for
Backward pass:
Set $\delta_T = \mathbf{o}_T - \mathbf{y}$
for $i = 1, \ldots, k_t$ **do**
    $\delta_{t,i} = \sum_{j=1}^{k_{t+1}} W_{t,j,i} \delta_{t+1,j} \sigma'(a_{t+1,j})$
end for
$\forall (v_{t-1,j}, v_{t,i}) \in E$ set the partial derivative to $\delta_{t,i} \sigma'(a_{t,i}) o_{t-1,j}$

1.4 Gradient descent variants

SGD is the general method to update weights in a neural network, but many improvements have been proposed in literature. Most of them modify the learning rate and the update rule in order to speed up the training. It has been shown that local minima are not the only trouble during training, in fact, saddle points slow down the computation too. They are surrounded by high error plateaus that can dramatically slow down learning, and give the illusory impression of the existence of a local minimum [3]. Escaping from such saddle points can be accomplished with vanilla SGD but it might be very slow, since steps are very short and not effective.

Momentum, introduced by Polyak as the "heavy ball" method, is widely recognized to accelerate the computation by adding a component of inertia to the optimization process. The general formula is modified into the following:

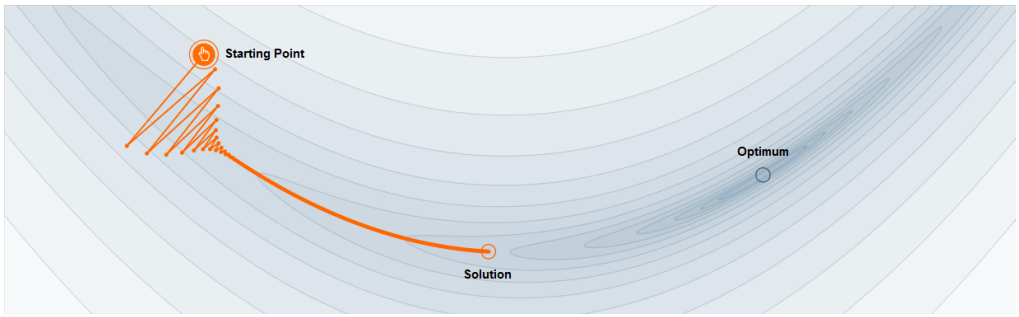
$$\begin{aligned} z^{k+1} &= \beta z^k + \eta \nabla f\left(w^k\right) \\ w^{k+1} &= w^k - \alpha z^{k+1} \end{aligned} \tag{1.10}$$

where $\nabla f(w^k)$ is the gradient of the loss function with respect to the parameters, η is the learning rate, and β is the momentum factor, usually set between 0.5 and 0.9. Momentum smooths and speeds up the computation. In fact, when the function is constrained in a canyon, momentum allows to more rapidly update the steepest direction allowing faster convergence. In such a way saddle points can be overcome and the new update rule can avoid to stop on a false local minimum. Moreover, momentum allows to avoid fluctuations over a local minimum.

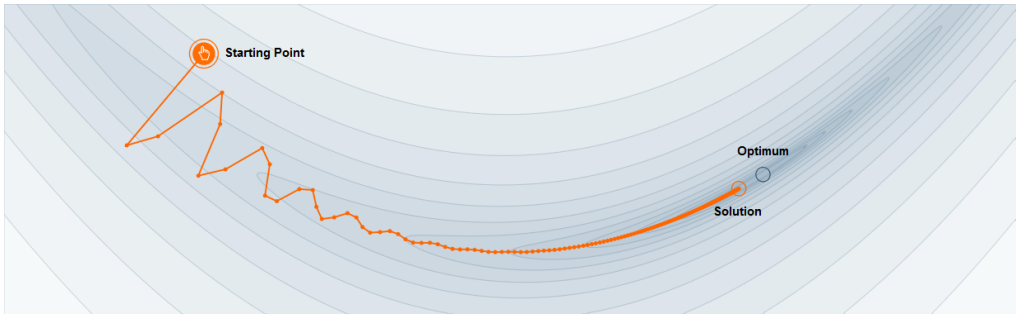
Nesterov Accelerate Gradient (NAG) [28] modifies the equation by adding a correction term to the gradient computation:

$$\begin{aligned} z_{t+1} &= \beta z_t - \eta \nabla f\left(w_t + \beta z_t\right) \\ w_{t+1} &= w_t + z_{t+1} \end{aligned} \tag{1.11}$$

The difference is that the previous update z_t is accounted before evaluating the gradient. Computing $w + \beta z_t$ thus gives us an approximation of the next position of the parameters, a rough idea where our parameters are going to move [33]. In such a way the ball is "less heavy" and slows down before the hill slopes up again, increasing responsiveness. Like momentum,



(a) SGD



(b) Momentum

Figure 1.2: SGD and Momentum (pictures taken from `https://distill.pub/2017/momentum/`)

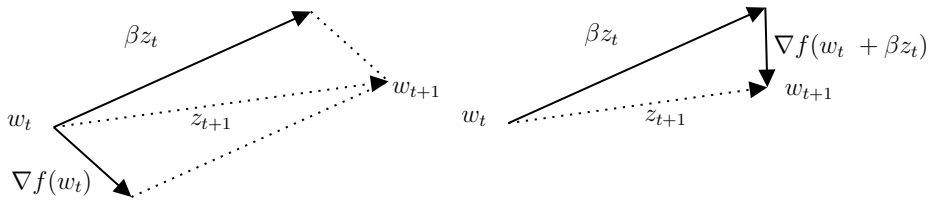


Figure 1.3: Momentum on the left; Nesterov on the right. Nesterov corrects the direction using the previous z_t.

NAG is a first-order optimization method with better convergence rate guarantee than gradient descent in certain situations [35]. However, it must be underlined that the theory predicts that any advantages in terms of asymptotic local rate of convergence will be lost in a noisy environment such as stochastic optimization.

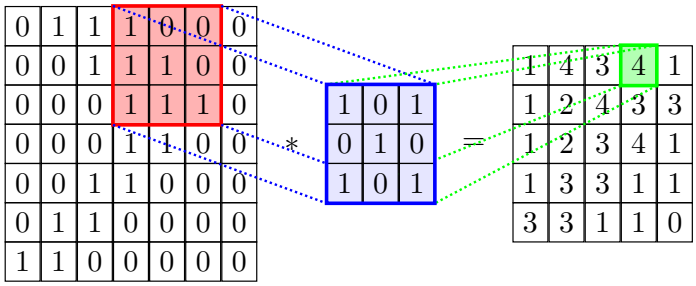
Momentum and NAG modify the direction of the move, while other methods mainly focus on setting a different learning rate at each iteration. Many improvements have been added to SGD to train the network in a faster way (Adam, Adagrad, Adadelta, RMSprop).

1.5 Neural Network Architectures

This section will provide an overview of the most known convolutional architectures, which well suit image classification. Thanks to deep learning, many tasks can be accomplished, such as single and multi object recognition and image segmentation and some others. The former is considered the touchstone for deep learning and usually algorithms are tested on the most famous architectures. The ImageNet Large Scale Visual Recognition Challenge is a benchmark in object category classification and detection on hundreds of object categories and millions of images [34]. The competition is run each year and new test images are collected (from Flickr and other search engines) and labeled especially for this competition among 1000 object categories taken from the ImageNet dataset.

LeNet is a quite old structure, but is very important since it introduces some core concepts as the convolutional filter, already mentioned before, and the pooling operation. Images store the information as multi-dimensional arrays where ordering matters along different channel axes (e.g.

red, green and blue in the RGB notation). Affine transformation cannot exploit all this information since the topological information is not taken into account, while discrete convolution preserves the notion of ordering [6]. A kernel of values, which represents one filter, slides across the input feature map. At each location, the product between each element of the kernel and the input element it overlaps is computed and then, the results are summed up to obtain the output in the current location. In such a way the information about closeness of some pixels is preserved. An example is as follows.



An example of static kernel for edge detection is the following:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{1.12}$$

In a neural network many convolution filters are stacked in a layer. The output size o_j of a convolutional layer along axis j is affected by:

- i_j: input size along axis j
- k_j: kernel size along axis j
- s_j: stride (distance between two consecutive positions of the kernel) along axis j. Strides act as a form of subsampling.
- p_j: zero padding along axis j. In other words the image is surrounded by a number p_j of zeros.

Convolutional filters in neural networks are adapted to the task, modifying the values of the kernel, thanks to the SGD algorithm.

Pooling is an important feature, that plays a key role in the classification. Pooling operations reduce the size of feature maps by using some

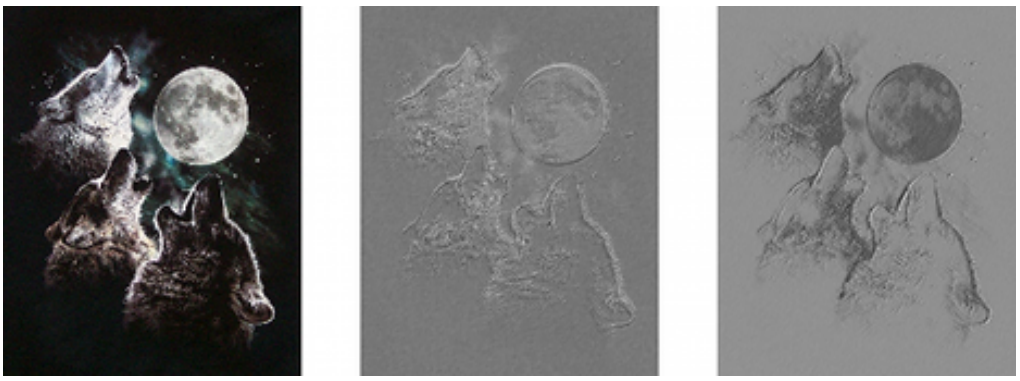


Figure 1.4: Image perception as seen by the neural network

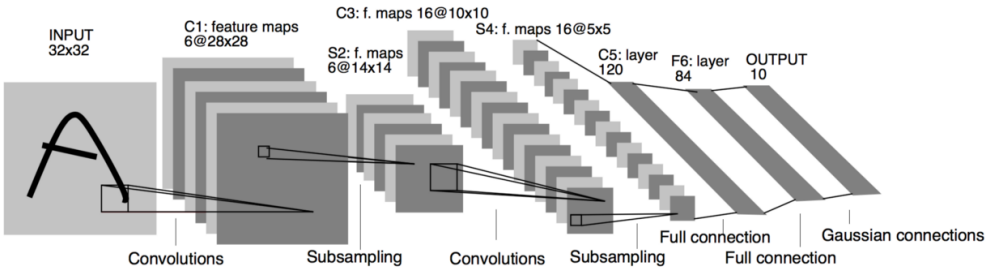


Figure 1.5: LeNet-5 architecture as published in the original paper.

function (e.g max / avg) over a set of input values. Pooling is used to downsample the image and restrict the feature space.

The two operations, convolution and pooling, allow to automatically extract and undersample features, this sets apart neural networks from other machine learning classifiers. In fact, convolution is adapted to the task and static filter, such as (1.12), are not used. At each layer m the features extracted are summed and the activation function is applied, letting to combine the information in layer $m + 1$.

LeNet [22] is one of the first successful network applying the aforementioned concepts. As shown in Figure 1.5, the network has only seven layers. Layer C1 is a convolutional layer with six features map, then S2 is an example of a pooling layer. The image is downsampled to a 14×14 pixels image. The layer C5 filters the images another time and then two classical fully connected layers classify the images.

AlexNet [21] was the ILSVRC 2012 winner; it has a similar framework to LeNet but has a much bigger structure and was run for the first time

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layer	16 weight layers	19 weight layers
Input (224 x 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 1.1: The structure as published in [36].

on a GPU. Most of the computations during the training are in a matrix form, GPUs allow to parallelize most of the computation allowing massive speed ups (50x). Krizhevsky et al. trained on two GPUs the network with 60M+ parameters, but still it has only seven layers.

VGG [36] published in 2014 has shown that very small convolution filters can be effective. This results in a much lower number of parameters, thus allowing to increase the depth of the network. Using smaller convolutional kernels (3×3 vs 7×7) allowed to introduce many more non-linear rectification layers instead of a single one. Table 1.1 shows that some of the convolutional layer are sized 1×1, this lets to increase the non-linearity in the decision function without affecting the receptive fields of the convolutional layers. The structure shown in the table aims at classifying the complex ImageNet dataset, but lighter models can be used for easier tasks.

GoogLeNet [39], published in 2015, won the ILSVRC14, achieving new state-of-art results. VGG is often criticised due to the high number of parameter updates, hence resulting in slowness in modest GPUs; moreover, the high number of parameters may lead to overfitting. It has been shown

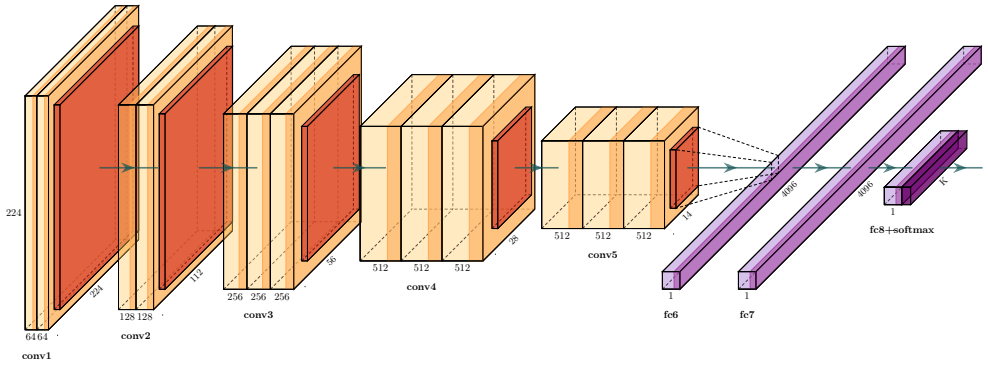


Figure 1.6: VGG16 plot [1]

that most of the parameters are very close to zero, hence previous structures have been tested with more sparse ones, but this did not lead to improvements due to hardware and software implementations. The main idea of the GoogleLeNet architecture is to consider how an optimal local sparse structure of a convolutional vision network can be approximated and covered by readily available dense components [22]. So the GoogLeNet structure added a module called inception module that approximates a sparse CNN with a normal dense construction (Figure 1.7). Another major change that GoogLeNet made, was to replace the fully-connected layers at the end with a simple global average pooling which averages out the channel values across the 2D feature map. In fact, in previous structures the last few layers accounted for most of the parameters.

ResNet [10] had set new state-of-art results in 2015, showing the effectiveness of very deep networks for the first time. Previously the increase of the network had lead to worse results. The degradation was due to the fact that not all systems are easy to optimize. The problem had been addressed with the use of deep residual learning. The idea introduced by He et al. is that blocks such as the one depicted in Figure 1.8 are easier to optimize. The shortcut do only backprogate the signal a few layers before, so there is no need to modify the backproagation algorithm. The work has brought interest in much more deep networks, as in [41]. Figure 1.9 shows how research in deep learning has affected and remarkably improved traditional techniques.

[1] Credits to PlotNeuralNet (github.com/HarisIqbal88/PlotNeuralNet)

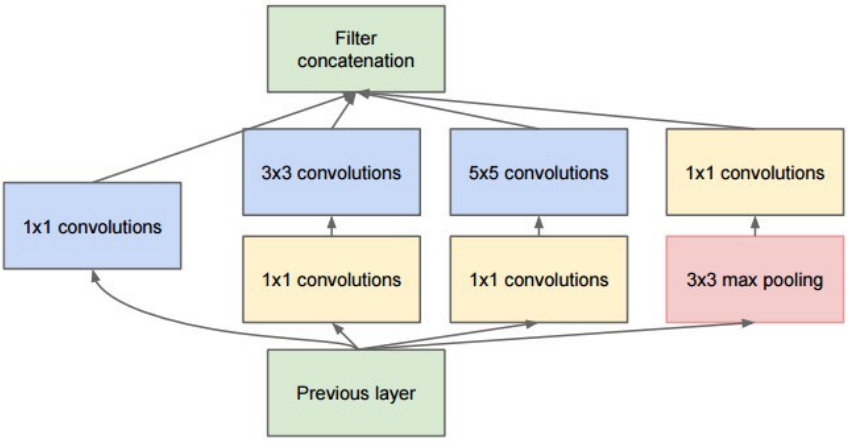


Figure 1.7: Inception module

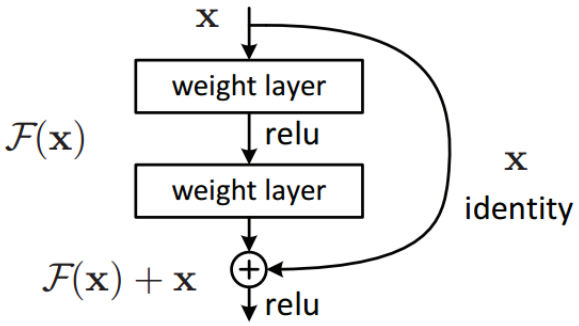


Figure 1.8: A building block

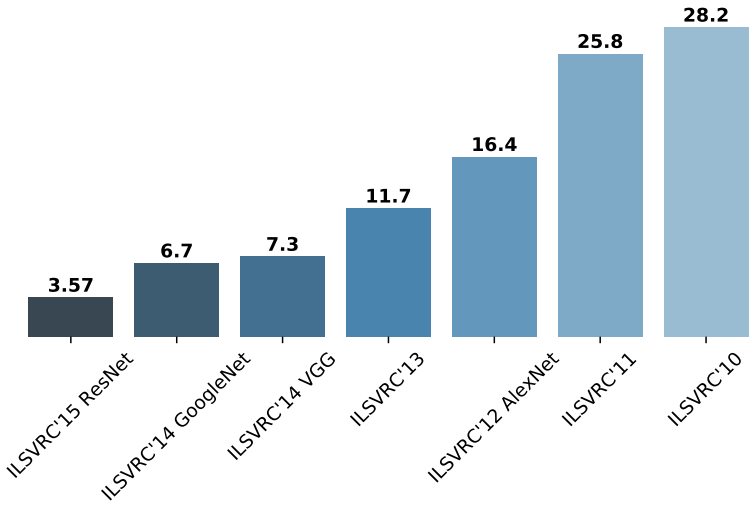


Figure 1.9: Improvements starting from 2010 (Top-5 accuracy on ImageNet)

SqueezeNet [15], published with the title 'SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and $<$ 0.5MB model size', had reduced the model complexity. Some applications may be run with very low computational resources, such as mobile phones. Complex structures, such as the ones explained so far, may easily drain batteries and resources. SqueezeNet:

- Replaces 3x3 filters with 1x1 filter
- Decreases the number of input channels to 3x3 filters
- Downsamples late in the network so that convolution layers have large activation maps

SqueezeNet is useful when dealing with easier tasks, where maximum accuracy is not needed.

1.6 Other meaningful techniques

Apart from the structures presented above, other procedures have remarkably improved state-of-art results and are here presented.

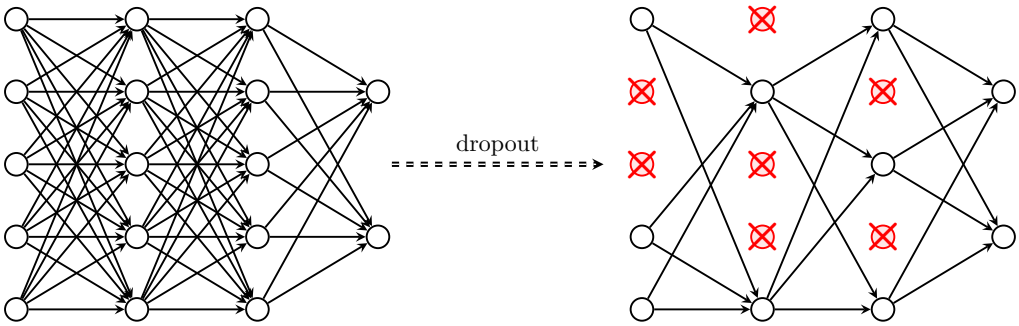
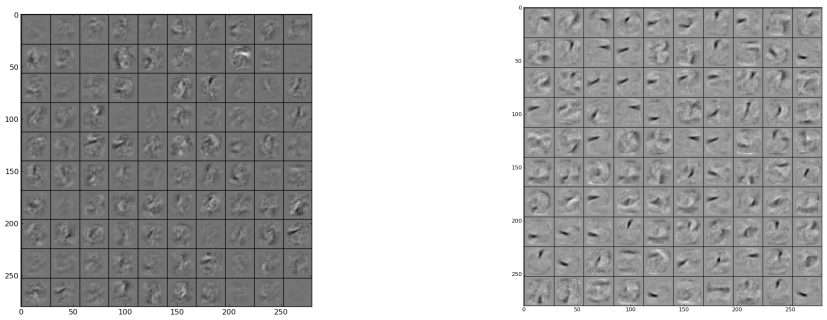


Figure 1.10: Dropout

1.6.1 Dropout

Dropout, published along with [21], showed how to improve test accuracy by restricting the training phase. Overfitting is greatly reduced by randomly omitting half of the feature detectors on each training case [11]. In every training batch some neurons' connections are temporarily removed, leading to a simpler version of the complete neural network. As stated by the authors, dropout prevents complex co-adaptations in which a feature detector is only helpful in the context of several other specific feature detectors. The easiest way to accomplish this is to drop each neuron with probability p independent of the others (usually p is set to 0.5). Thus it follows some information is not backpropagated along the network.

Dropout is a relatively cheap form of regularization compared to more traditional techniques in machine learning, such as Ridge or Lasso, where computing the penalty term introduces significant overhead. One reason why dropout gives major improvements over standard training is that it encourages each individual hidden unit to learn a useful feature without relying on specific other hidden units to correct its mistakes. An example is shown in Figure 1.11, where features learned by first layer hidden units for (1.11a) backprop and (1.11b) dropout on the MNIST dataset are visualized: it can be seen that the use of dropout empowers feature detection.



(a) Without dropout (b) With dropout

Figure 1.11: Dropout visualization

Another interesting regularizing technique, called Cutout, has set new state-of-art results, by masking part of the images [4]. Instead of dropping values during the training process, some information is dropped on purpose.

1.6.2 Batch Normalization

Batch normalization allows to improve the speed of training. Ioffe and Szegedy [16] suggested that the distribution of each layer's input changes during training, thus slowing due to the so-called internal covariate shift. Previously, batch normalization was applied only to the input values: batch normalization applies it at each layer. Moreover, it allows to regularize the model and to avoid the need for Dropout. The algorithm adds some overhead due to normalization at each layer, but allows to speed up the training anyways. The procedure is formalized in Algorithm 4

Algorithm 4 Batch Normalization

Input: Values of x over a mini-batch $\mathcal{B} = \{x_{1...m}\}$

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

1: Mini-batch mean: $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$

2: Mini-batch variance: $\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$

3: Normalize: $\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$

4: Scale and shift: $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$

1.6.3 Data augmentation

A big issue in Deep Learning, that makes sometimes impractical its implementation in the real world, is the high demand for (labeled) data, which can be hardly to retrieve. Data augmentation lets to enlarge the size of the dataset and generally reduce the test error. Traditional transformations consist of using a combination of affine transformation to manipulate the training data, typical examples for images are: shifting, zooming in/out, rotation, flipping, distortion. For each image a duplicate is created, thus enlarging the training set. More recent work uses Generative Adversarial Networks (GANs) to create new input sets [24] [26]. Roh et al. [32] have tested the efficacy of different techniques on a subset of the ImageNet dataset.

Chapter 2

Training by restarts

2.1 The idea

In the first stage of the thesis, we focused on training with multiple restarts. The idea comes to mind after realizing that most of the training time is spent close to the minimum 0, i.e., when all the images are correctly classified. Such a behaviour is shown in Figure 2.1: the cost function easily drops to zero and varies multiple times very close to the minimum.

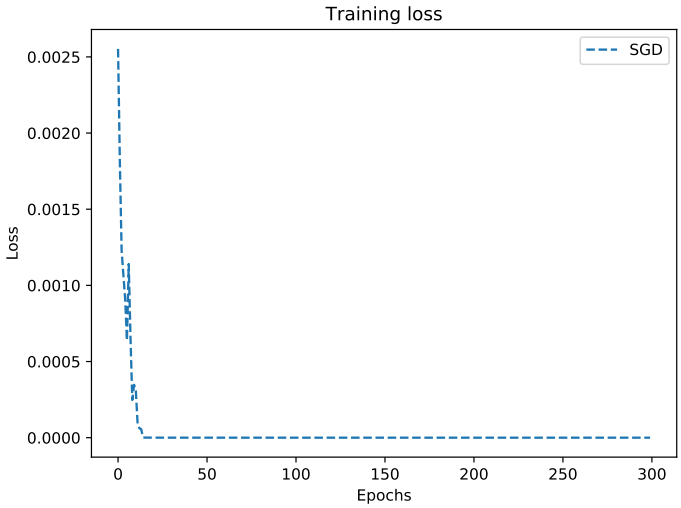


Figure 2.1: Training loss on VGG16

The loss curve of neural networks is still unclear: the highly dimensional space makes it difficult to perform analyses. Kawaguchi [17] have stated

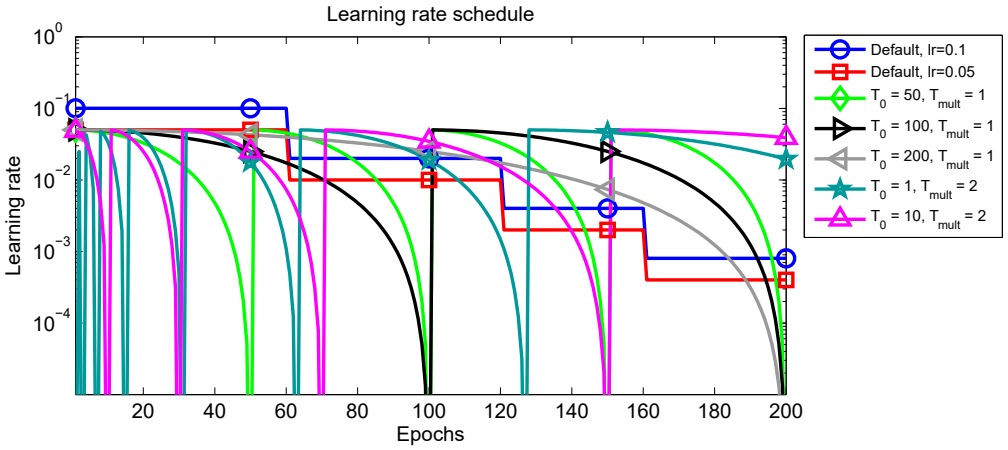


Figure 2.2: Learning rate schedules (image taken from [25])

that the number of possible local minima grows exponentially with the number of parameters. Our work aims at exploring the loss function.

A similar idea has been already covered in a different way by Loshchilov and Hutter [25]. They proposed to periodically simulate warm restarts of SGD, where in each restart the learning rate is initialized to some value and is scheduled to decrease. In their work the cosine function has been exploited and adapted in order to decrease the learning rate at each epoch. The learning rate is selected according to the following formula:

$$\eta_t = \eta_{min} + \frac{1}{2}\left(\eta_{\max} - \eta_{\min}\right)\left(1 + \cos\left(\frac{T_{cur}}{T_i}\pi\right)\right) \tag{2.1}$$

where η_{min} and η_{max} bound the learning rate, and T_{cur} accounts for how many epochs have been performed since the last restart. An example of some schemes is shown in Figure 2.2. Warm restarts allows to escape from sharp minima and to reach flat minima, which are believed to better generalize [18]. However, Dinh et al. [5] have argued that flat minima in practical deep learning hypothesis spaces can be turned into sharp minima via re-parameterization without affecting the generalization gap.

An approach similar has been studied by Smith [37], but his work was not focused on restarts. Instead of monotonically decreasing the learning rate, his method lets the learning rate cyclically vary between reasonable boundary values. Smith's work aims at improving the speed of convergence, not escaping from bad local minima.

Loshchilov and Hutter's work inspired another idea. In the original work each run was to used to ensemble the results and better predict the

output values. In [14] the authors suggest to take M snapshots of the models obtained by SGDR right before the restart and to use those to build an ensemble (Figure 2.3 shows the different scheme).

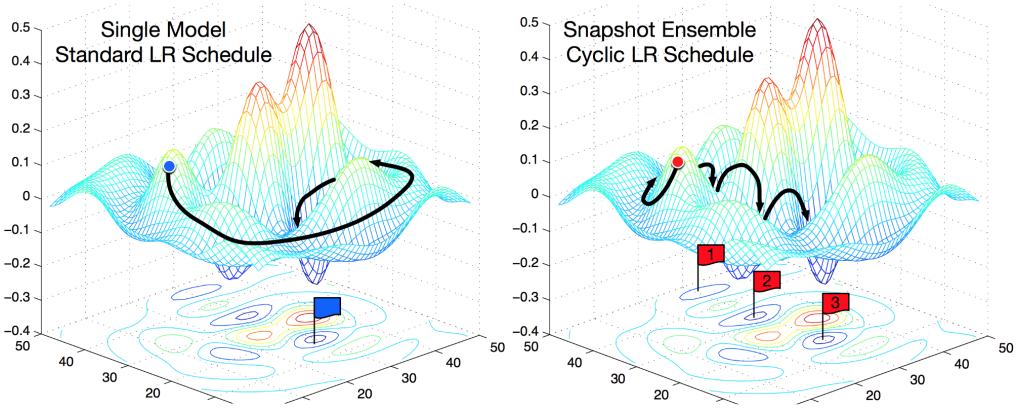


Figure 2.3: On the left a run without snapshots. The ensemble can be obtained by saving multiple models during the training as shown on the right (picture taken from [14]).

2.2 Our algorithm

The algorithm presented in chapter 1 has been modified in order to restart according to the parameter ϵ: when the loss on the training set goes below that threshold, the model is restarted. Our work focuses on "cold restarts", we do not vary the learning rate according to the cosine or similar function. The modified algorithm is shown in Algorithm 5. At step 5 the cost function on the training set is calculated over a sample of minibatches randomly chosen, otherwise another epoch would be necessary; this has been proven to be enough effective.

Number of minibatches	RMSE	Number of minibatches	RMSE
1	4.470e-05	6	2.302e-05
2	3.627e-05	7	1.949e-05
3	2.909e-05	8	2.150e-05
4	2.368e-05	9	1.519e-05
5	2.063e-05	10	1.536e-05

Table 2.1: Estimating the training cost with a sample of minibatches. RMSE with respect to the true value.

Algorithm 5 SGD for Neural Networks with restarts

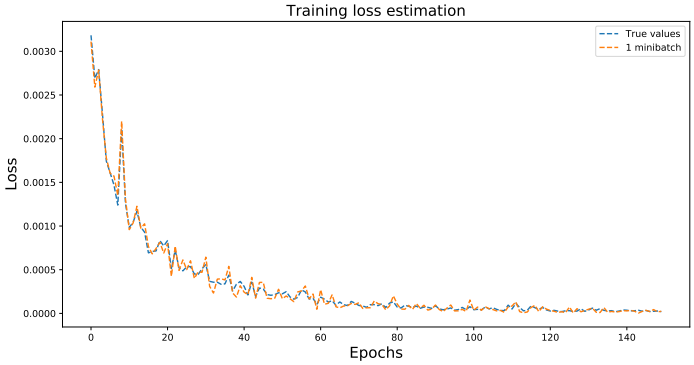
Parameters: Scalar $\eta > 0$, integer $T > 0$, $\epsilon \in \mathbb{R}^+$, $n\_batches > 0$ number of minibatches used to estimate the training loss

Input: Layered graph (V, E), differentiable function L

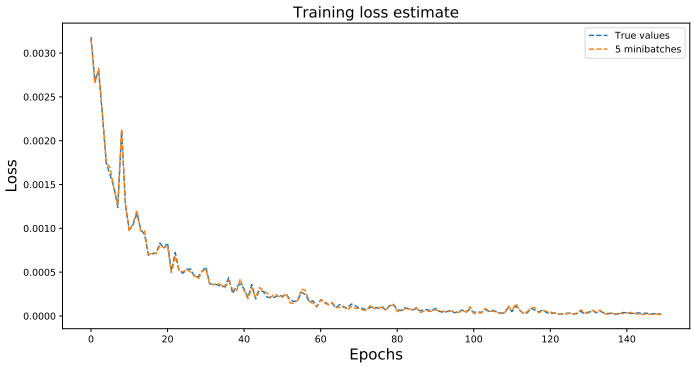
$\mathbf{w}^{(1)}$ = random_initialization()
for i = 1, 2, . . . , T **do**
  Extract a minibatch $(\mathbf{x}, \mathbf{y})$
  Calculate gradient $\mathbf{v}_i =$ backpropagation $(\mathbf{x}, \mathbf{y}, \mathbf{w}, (V, E), L)$
  Update $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \eta \mathbf{v}_i$
  training_cost = train_cost_estimate($n\_batches$)
  if training_cost $< \epsilon$ **then**
    $\mathbf{w}^{(i+1)}$ = random_initialization()
  end if
end for
Output: $\overline{\mathbf{w}}$ is the best performing $\mathbf{w}^{(i)}$ on the validation set

2.3 Results with CIFAR10

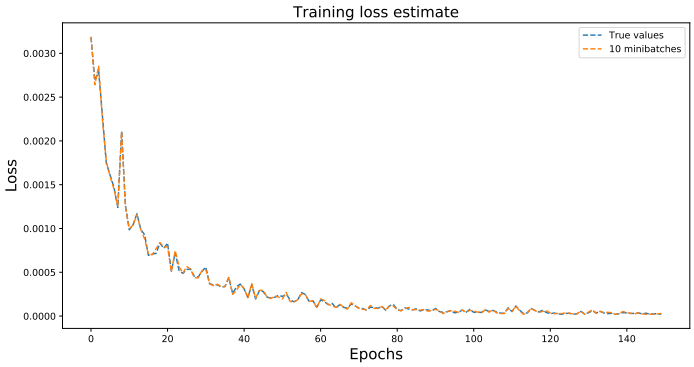
Before looking at the results, it is useful to ensure that estimates of the loss on the training set, based on a sample of minibatches, are meaningful. Figure 2.4 shows how using an increasing number of minibatches leads to better estimates; ten minibatches seem to be enough affordable. Table 2.1 shows how the RMSE easily drops by increasing the number of minibatches. Such results confirm that a sample of 5-10 minibatches can effectively estimate the true value over the whole training set.



(a) One minibatch



(b) Five minibatches



(c) 10 minibatches

Figure 2.4: Training loss estimates

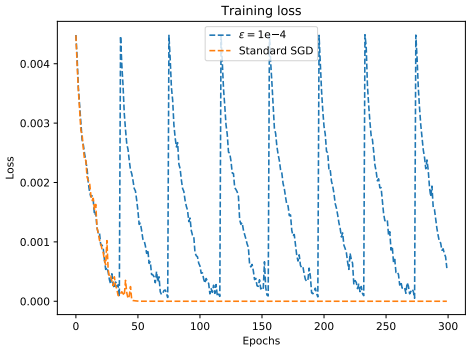
	VGG16		ResNet34	
	Loss	Accuracy	Loss	Accuracy
SGD without restart	0.001432	84.92	0.001589	83.07
Restart $\epsilon = 1e{-}6$	**0.001318**	84.43	**0.001468**	82.96
Restart $\epsilon = 1e{-}7$	**0.001333**	84.88	**0.001429**	**83.47**

Table 2.2: Restart applied to VGG16 and ResNet34

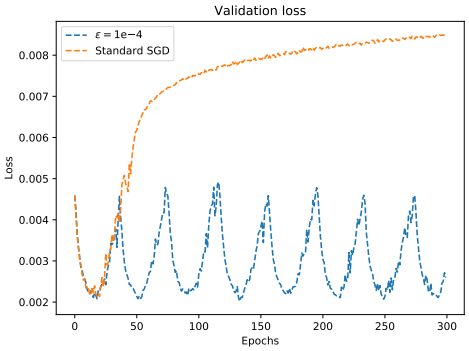
CIFAR-10 [20] has been used to benchmark our technique. It is made of $60,000$ 28×28px images: $50,000$ are used for training, and $10,000$ for testing. Figure 2.5 shows the first results obtained by testing with the AlexNet structure. SGD with momentum ($\eta = 0.001$ and $\beta = 0.9$) has been used, which has been proven to obtain the best results in terms of speed and accuracy. Figure 2.5a clearly shows how the model restarts under a threshold, in this case set to 0.0001. Figure 2.5b plots the validation loss, which easily overfits. Restarts let to train multiple models and obtain different loss values over the validation dataset. The minimum ranges between $[0.0021, 0.0023]$. Overfitting does not seem to appear on the accuracy curve in Figure 2.5d, which seems to be stable after epoch 50. These early results show that the validation accuracy reached is similar, even though most of the times the accuracy obtained with restarts is lower due to the fact that the training process is not complete, thus suggesting that a lower ϵ should be used. In the following, only the validation accuracy plot will be shown.

Figure 2.6 shows the same analysis on VGG16 changing the hyper-parameter. In this case it is clear that at each restart the optimization process is over using ϵ set to $1e{-}7$. The values of the validation loss are bouncing multiple times, making it harder to evaluate. Figure 2.7 shows that a value of $1e{-}6$ is still insufficient, leading to worse results.

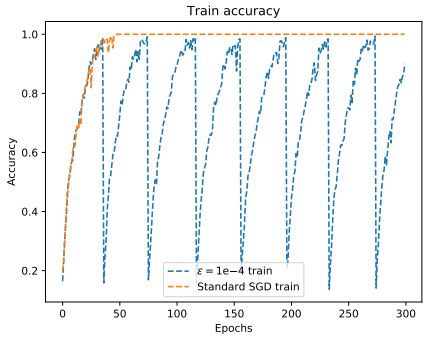
Other tests have shown different results as the complexity and depth of the network increases. ResNet34 in Figure 2.9 has shown that models can significantly differ. In fact, the models obtain different results at each initialization. In the first phase the restart technique has led to improved results w.r.t. to the standard SGD.



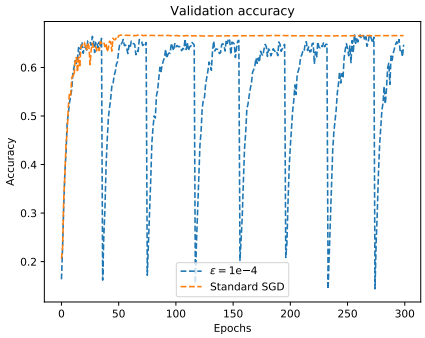
(a) Training loss



(b) Validation loss

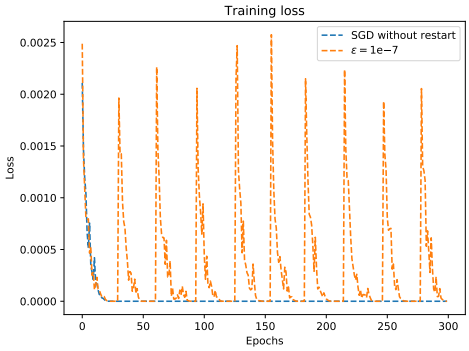


(c) Training accuracy

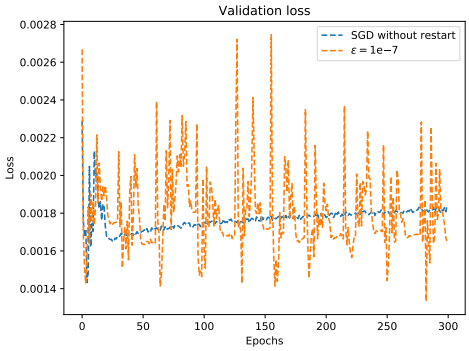


(d) Validation accuracy

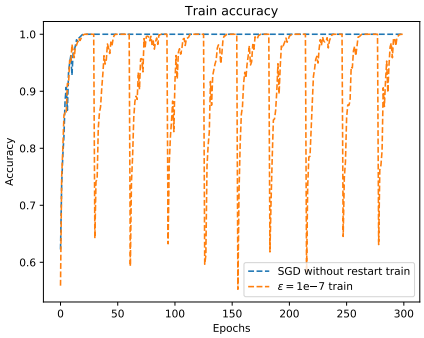
Figure 2.5: Restart applied to AlexNet with $\epsilon = 1e{-}4$, $\eta = 0.001$ and $\beta = 0.9$



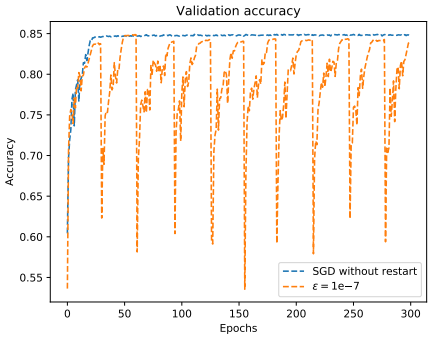
(a) Training loss



(b) Validation loss



(c) Training accuracy



(d) Validation accuracy

Figure 2.6: Restart applied to VGG16 with $\epsilon = 1e{-}7$, $\eta = 0.001$ and $\beta = 0.9$

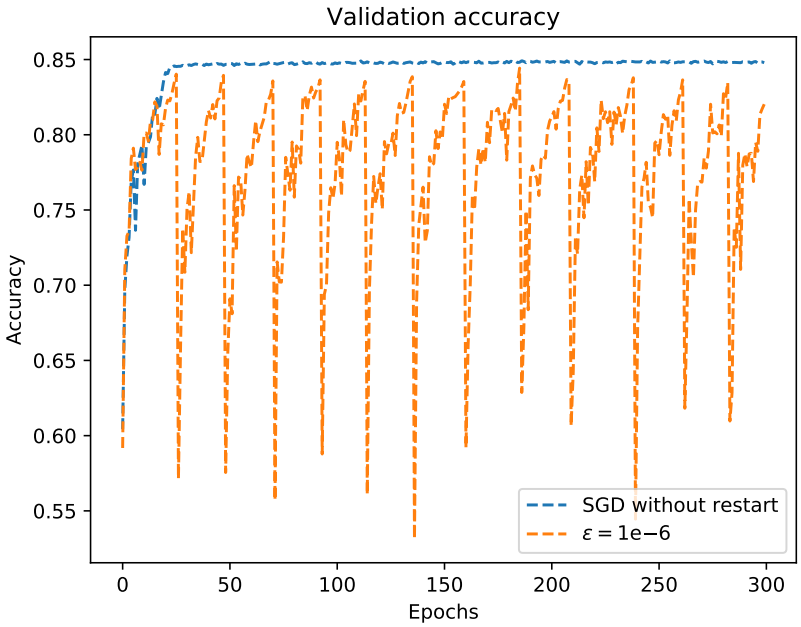


Figure 2.7: Restart applied to VGG16 with $\epsilon = 1e{-}6$, $\eta = 0.001$ and $\beta = 0.9$

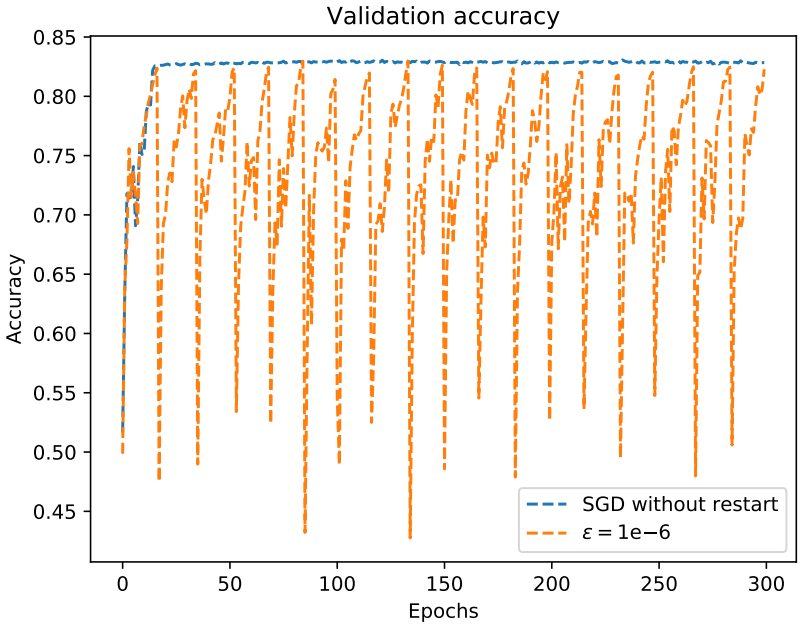


Figure 2.8: Restart applied to ResNet34 with $\epsilon = 1e{-}6$, $\eta = 0.001$ and $\beta = 0.9$

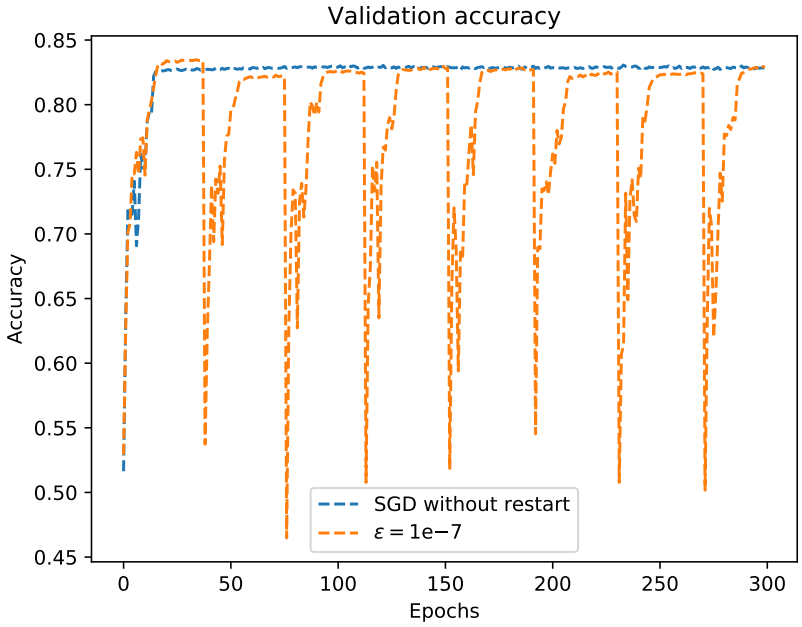


Figure 2.9: Restart applied to ResNet34 with $\epsilon = 1e{-}7$, $\eta = 0.001$ and $\beta = 0.9$

Chapter 3

Training Deep Neural Networks with Simulated Annealing

3.1 The idea

SGD is the de facto standard when training neural networks. Leveraging the gradient, SGD allows to rapidly find a good set of weights in such an high dimensional space; moreover, the use of minibatches leads to achieve a considerable speed up. We now want to try new algorithms with the hope to trade computing time with improved accuracy.

SGD has still unsolved problems, such as gradient vanishing/explosion, party solved by ResNets [10], improved initialization [43], ReLu [8]. Moreover, the high interest and research to better understand how the loss function is shaped suggests that the optimization process should be possible with other procedures as well. We want to tackle this problem in a different way. Achieving state-of-art results with a derivative-free approach would probably mean that the loss function generated by the dataset and the network structure is quite easy and flat.

As already mentioned, it is well known that small minibatches allow to reach a better test error, thus suggesting that sub-optimal procedures can better generalize. In the context of local learning, Nøkland and Hiller Eidnes [31] have recently stated that local learning appears to add an inductive bias that reduces overfitting. The well known dropout, which acts as regularizer, modifies the gradient, leading to a sub-optimal gradient. Neelakantan et al. [27] have proved that adding noise to the gradient helps

to avoid overfitting, but also can result in lower training loss. These examples suggest that other algorithmic procedures should be tested in order to better investigate the training cost function. Moreover, other neural structures, discarded from SGD, could be successfully trained by other algorithms. Motivated by what explained so far, we have decided to adapt the simulated annealing algorithm coming from discrete optimization to the continuous space.

3.2 Simulated Annealing

Simulated Annealing (SA), published by Kirkpatrick et al. [19], is a well-known algorithm in discrete optimization. Leveraging a physical analogy, it allows to escape from local minima and more effectively search for the global optimum than hill climbing. It is one of the oldest metaheuristics and has been adapted to solve many combinatorial optimization problems. SA is a stochastic local search algorithm that, starting from some initial solution, iteratively explores the neighbourhood of the current solution [7]. At high temperatures the particles are free to move, and the structure is subject to substantial changes. The temperature decreses over time, and so does the probability for a particle to move, until the system reaches its ground state, the one of lowest energy.

Let us introduce the formal notation for the discrete space; in the next section it will be adapted to our needs. Let $s \in S$ be a solution in the set S of all possible candididate solutions and $f : S \to \mathbb{R}$ be the objective function. An optimal solution s^* is a candidate solution for which holds $f(s^*) \leq f(s) \quad \forall s \in S$. $\Delta(s, s')$ is the objective function difference of two candidate solutions s, s'. Let T be the temperature, with initial value T_0 and final T_f. SA allows to accept worsening moves. The most commonly used acceptance criterion is the so called Metropolis condition where the probability acceptance is

$$p = \begin{cases} e^{-\frac{\Delta(s,s')}{T}} & \text{if} \quad \Delta(s, s') > 0 \\ 1 & \text{if} \quad \Delta(s, s') \leq 0 \end{cases} \tag{3.1}$$

At each iteration the temperature is decreased according to the cooling factor α:

$$T_{i+1} = \alpha \times T_i \tag{3.2}$$

Usually the temperature is "annealed" multiple times; it means, that the temperature is restored multiple times to the initial value. With an equal worsening of the objective function value, a solution is more likely to be accepted when the temperature is high, while when the temperature is low, typically towards the end the search, improving candidate solutions are prioritized.

3.3 Our algorithm

We have adapted SA to work in the continuous space. Let us start by recalling that for an objective function $L(\mathbf{w})$ differentiable, there exists a gradient

$$\nabla(\mathbf{w}) \tag{3.3}$$

We are not going to exploit it, our method will completely be derivative-free. Given the set of parameters $\mathbf{w}$, we generate a random move $\Delta(\mathbf{w})$ and then we evaluate the new loss in a nearby point $\mathbf{w}'$ defined as:

$$\mathbf{w}' := \mathbf{w} - \epsilon\Delta(\mathbf{w}) \tag{3.4}$$

with $\epsilon > 0$. Now, if the norm of $\epsilon\Delta(w)$ is little enough, we know, thanks to the Taylor approximation, that

$$L(\mathbf{w}') \simeq L(\mathbf{w}) - \epsilon\nabla^T(\mathbf{w})\Delta(\mathbf{w}) \tag{3.5}$$

Then the objective function improves if:

$$\nabla(\mathbf{w})^T\Delta(\mathbf{w}) > 0 \tag{3.6}$$

We are in the continuous space, so we can try the opposite direction, hoping to improve the objective function.

$$\mathbf{w}'' := \mathbf{w} + \epsilon\Delta(\mathbf{w}) \tag{3.7}$$

To summarise, we always choose among two possible moves:

$$\mathbf{w}' := \mathbf{w} - \epsilon\Delta(\mathbf{w})$$
$$\mathbf{w}'' := \mathbf{w} + \epsilon\Delta(\mathbf{w})$$

If one of the two improves $L(\mathbf{w})$, then we surely accept the best move, otherwise we accept according to the Metropolis formula explained before.

The algorithm is formalized in Algorithm 6. In the following, Simulated Annealing in the minibatch version will be refereed as Stochastic Simulated Annealing (SSA).

Our approach is completely derivative-free, thus meaning that there is no need for a continuous loss function to optimize. We can, indeed, *optimize over discrete functions*, such as the accuracy of the model, which is the ultimate goal in classification tasks.

Algorithm 6 Stochastic Simulated Annealing for Neural Networks

Parameters: Scalar $\epsilon > 0$, $\epsilon \in \mathbb{R}^+$, $0 < \alpha < 1$
Input: Layered graph (V, E), loss function L to be minimized
Divide the training dataset in N minibatches
Initialize: $\mathbf{w}^{(1)}$ = random_initialization()
i = 1
for $t = 1, \ldots,$ Nepochs **do**
    for $n = 1, \ldots,$ N **do**
        Extract the n-th minibatch $(\mathbf{x}, \mathbf{y})$
        Generate random move $\Delta(\mathbf{w})$
        $\mathbf{w}_{best} = \arg\min\{L(\mathbf{w}^{(i)} \pm \epsilon\Delta(\mathbf{w}))\}$
        $\mathbf{w}^{(i+1)} = \mathbf{w}_{best}$
        prob $= e^{-\frac{L(\mathbf{w}_{best},\mathbf{x},\mathbf{y}) - L(\mathbf{w}^{(i)},\mathbf{x},\mathbf{y})}{T}}$
        if $L(\mathbf{w}_{best}, \mathbf{x}, \mathbf{y}) > L(\mathbf{w}^{(i)}, \mathbf{x}, \mathbf{y})$ **and** prob $<$ random(0, 1) **then**
            $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)}$
        end if
        i = i + 1
    end for
    $T = \alpha \times T$
end for
Output: $\overline{\mathbf{w}}$ is the best performing $\mathbf{w}^{(i)}$ on the validation set at the end of each epoch

3.4 Results

Before looking at the results, it is useful to compare the distribution of the gradient and our random move, which is drawn by a Gaussian random

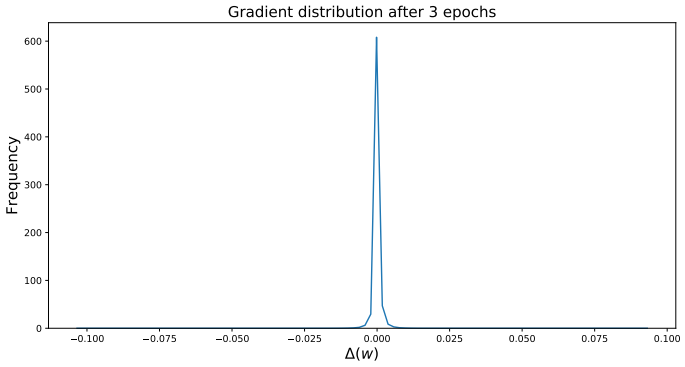


Figure 3.1: Gradient distribution

variable, so that each $\Delta(w)_i \sim \mathcal{N}(0,1)$, where $\Delta(w)_i$ is a generic element in $\Delta(w)$. As Figure 3.1 shows, the gradient has a distribution that looks like to the Gaussian one. However, the parameter ϵ must be properly set to a reasonable value. Figure 3.2 shows a comparison between the distribution of the gradient, multiplied by the typical learning rate 0.01, and our random move. In the last picture the two distributions seem to very similar.

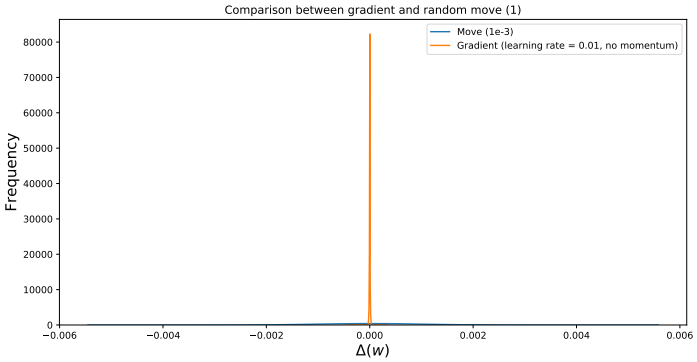
During the analysis of the algorithm a static ϵ has been shown to be highly inefficient, leading either to a very slow convergence or to bad results. We have then decided to adapt this parameter during training to the results obtained. More specifically, if the value of ϵ leads to a worse objective function value, ϵ is decreased by a factor of 10, similarly to what happens with scheduled SGD.

3.4.1 LeNet-like structure

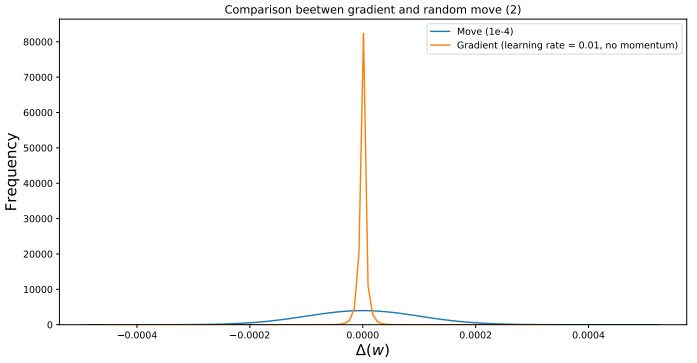
Before analysing the results over complex networks, the algorithm has been tested with the LeNet structure. In our implementation, the number of convolutional filters has been increased to 32 and 64 for the first two layers, whilst the fully connected hidden layer has been set to 1000 neurons.

Information about the training process has been kept. Figure 3.3 shows how the empirical probability of accepting worsening moves decreases during the training phase. Diversification and intensification have been divided almost equally setting T_0 to 1 and α to 0.97. With such parameters the probability slowly decreases to 0 in 200 epochs.

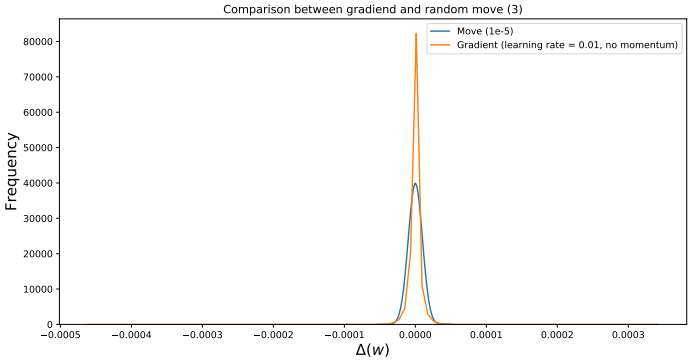
Figure 3.4 plots the number of worsening moves accepted vs not ac-



(a) Random move with ϵ =1e−3



(b) Random move with ϵ = 1e−4



(c) Random move with ϵ =1e−5

Figure 3.2: Gradient comparison with random move

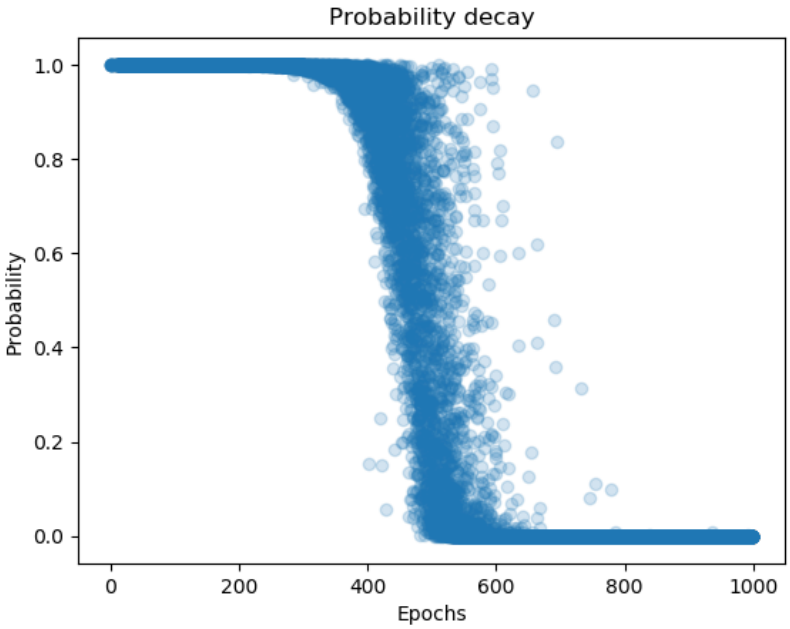


Figure 3.3: Probability decay

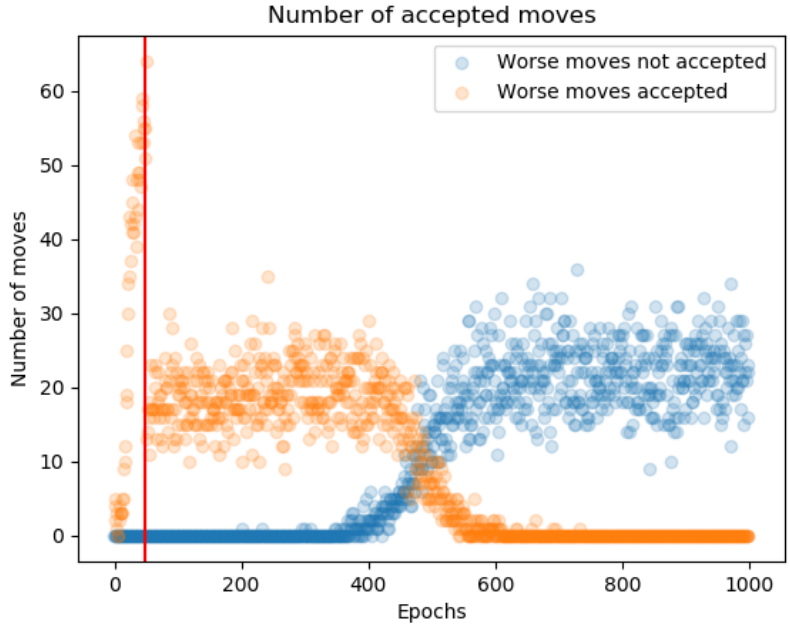
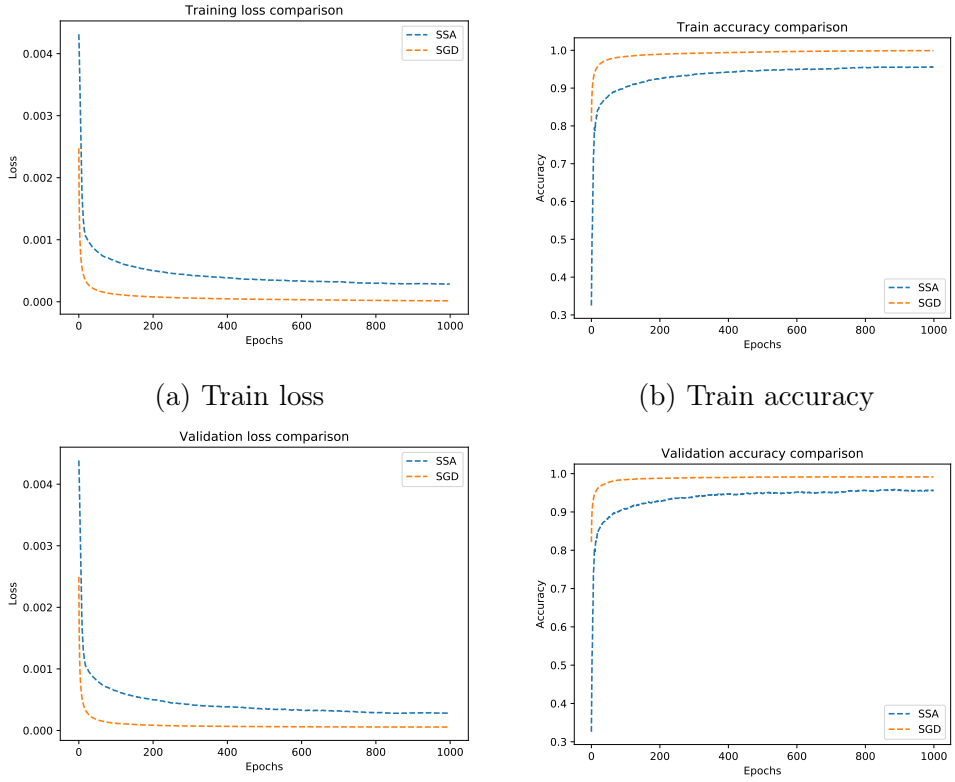


Figure 3.4: The number of worsening moves accepted and not accepted for each epoch.



(a) Train loss

(b) Train accuracy

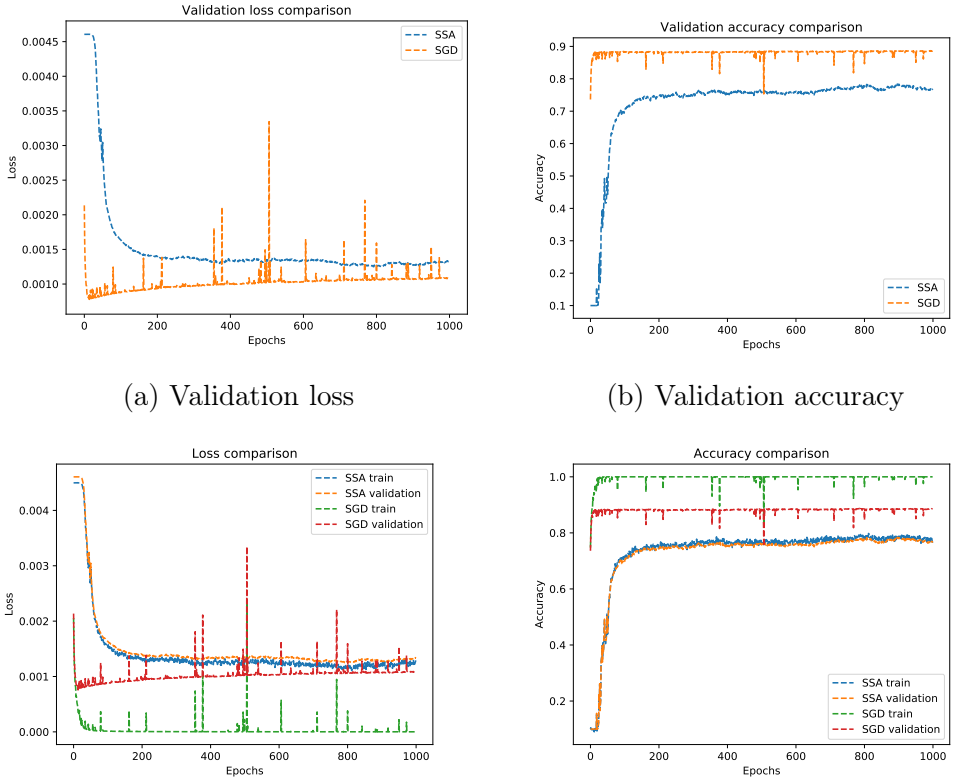
(c) Validation loss

(d) Validation accuracy

Figure 3.5: SSA on LeNet-like structure and MNIST. SGD: $\eta = 0.001$, $\beta = 0$. SSA: starting $\epsilon = 0.01$, $\alpha = 0.97$, $T_0 = 1$

cepted for each epoch. The red bar indicates a drop by a factor of 10 of ϵ. As explained before when the number of worsening moves is over a threshold the ϵ value is reduced. Most of the worsening moves are accepted during the diversification phase but, starting from epoch 400, they tend to be rejected. The plot allows to understand that, as the optimization process goes further, the random moves become less effective. In fact, apart from what is shown to the left of the red bar, the trend of worsening moves increases using the same ϵ.

The results obtained with the LeNet structure show that optimization in deep learning using SA is possible. Even though SGD performs significantly better, SA can find a good model in a reasonable time. In Figure 3.5d the accuracy still improves in the final phase, but the process has been stopped to epoch 1000. Conclusions cannot be drawn on overfitting, since the validation loss with SGD does not increase and the SSA's optimization



(a) Validation loss

(b) Validation accuracy

(c) Loss comparison

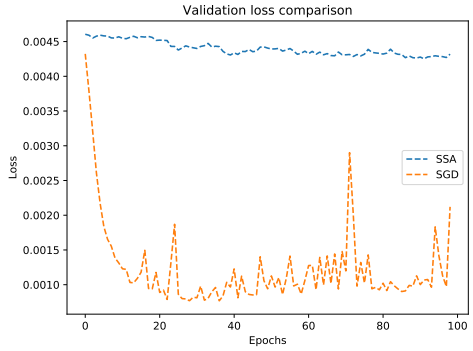
(d) Accuracy comparison

Figure 3.6: Comparison on VGG16 with Fashion-MNIST. SGD: $\eta = 0.001$, $\beta = 0$. SSA: starting $\epsilon = 0.01$, $\alpha = 0.97$, $T_0 = 1$

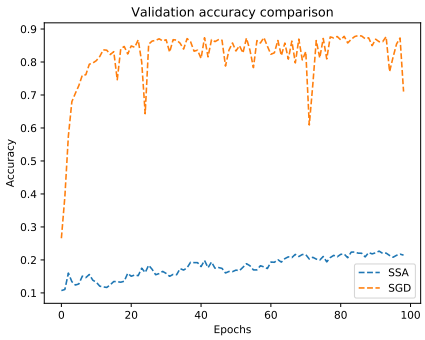
process is not over (the minimum of 0 is not reached by the training set loss). In any case, the validation and training curves for SSA are almost overlapping.

3.4.2 VGG16 and MobileNet

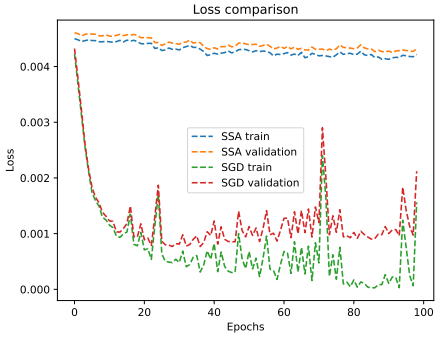
More complex networks and datasets have been tested. In the research community MNIST is being discarded in favour of Fashion-MNIST [40], which lets to use the same input parameters (28×28px grayscale images). The results on the LeNet structure are very similar to the ones presented before. Instead the tests on VGG16 (Figure 3.6), show that SSA does not scale to more complex problems. While at the end the two validation loss curves are very close, the gap in accuracy is increased. Moreover, in Figure 3.6c the gap between the SGD and SSA's training loss is increased.



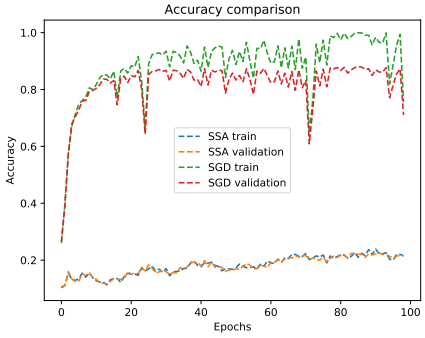
(a) Validation loss



(b) Validation accuracy



(c) Loss comparison



(d) Accuracy comparison

Figure 3.7: Comparison on MobileNet with Fashion-MNIST. SGD: $\eta = 0.001$, $\beta = 0$. SSA: starting $\epsilon = 0.01$, $\alpha = 0.97, T_0 = 1$

MobileNet [13], similar in the core concept to SqueezeNet, has been tested as well, but worse results have been achieved. It must be underlined that even SGD does not achieve zero on the training loss in a steady way as in VGG or ResNet.

Even tough the first tests have achieved acceptable results, models trained with SSA can hardly overcome SGD, due to the fact that the random move is not effective enough. In the next chapter a combination of SGD and SA will be studied, showing that the derivative multiplied with high learning rates can be more effective than random moves with low ϵ.

Chapter 4

Hyper-parameter tuning by Simulated Annealing

4.1 The idea

Hyper-parameters are usually very hard to optimize, as they depend on the algorithm and on the underlying dataset. Hyper-parameter search is commonly performed manually, via rules-of-thumb or by testing sets of hyper-parameters on a predefined grid [2]. In SGD, momentum is widely recognized to increase the speed of convergence. Instead, the learning rate is highly dependent on the depth of the network (generally on the model complexity) and on dataset difficulty. Usually they are set on a best-practice basis, but methods such as CLR let to reduce the number of choices [37]. Bergstra and Bengio [1] showed empirically and theoretically that randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid. We now introduce a new method, which allows to set only a few bounds and then the optimization process runs independently. In this way there is no need to set other parameters. This process could be extended to other techniques, such as momentum or Nesterov, creating a large pool of choices.

Leveraging the core concepts of Simulated Annealing, we use the derivative to set the direction. Rather than using conventional learning rates, we explore higher learning rates hoping to accelerate the optimization process. Then the move is accepted according to the Metropolis criterion. The approach is formalized in Algorithm 7, and will be later referred as SGD-SA.

Algorithm 7 SGD-SA

Parameters: A set of learning rates H, integer $T > 0$, $\alpha \in [0, 1]$

Input: Layered graph (V, E), loss function L differentiable

$\mathbf{w}^{(1)}$ = random_initialization()
i = 1
Divide the training dataset in N minibatches
for t = 1, ..., Nepochs **do**
    for n = 1, ..., N **do**
        Extract the n-th minibatch ($\mathbf{x}$, $\mathbf{y}$)
        Sample $\eta_i \sim U(H)$
        Calculate gradient $\mathbf{v}_i =$ backpropagation $(\mathbf{x}, \mathbf{y}, \mathbf{w}, (V, E), L)$
        Update $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \eta_i \mathbf{v}_i$
        prob $= e^{-\frac{L(\mathbf{w}_{best}, \mathbf{x}, \mathbf{y}) - L(\mathbf{w}^{(i)}, \mathbf{x}, \mathbf{y})}{T}}$
        if $L(\mathbf{w}_{best}, \mathbf{x}, \mathbf{y}) > L(\mathbf{w}^{(i)}, \mathbf{x}, \mathbf{y})$ **and** prob $<$ random(0, 1) **then**
            $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)}$
        end if
        i = i + 1
    end for
    $T = T \times \alpha$
end for
Output: $\overline{\mathbf{w}}$ is the best performing $\mathbf{w}^{(i)}$ on the validation set at the end of each epoch

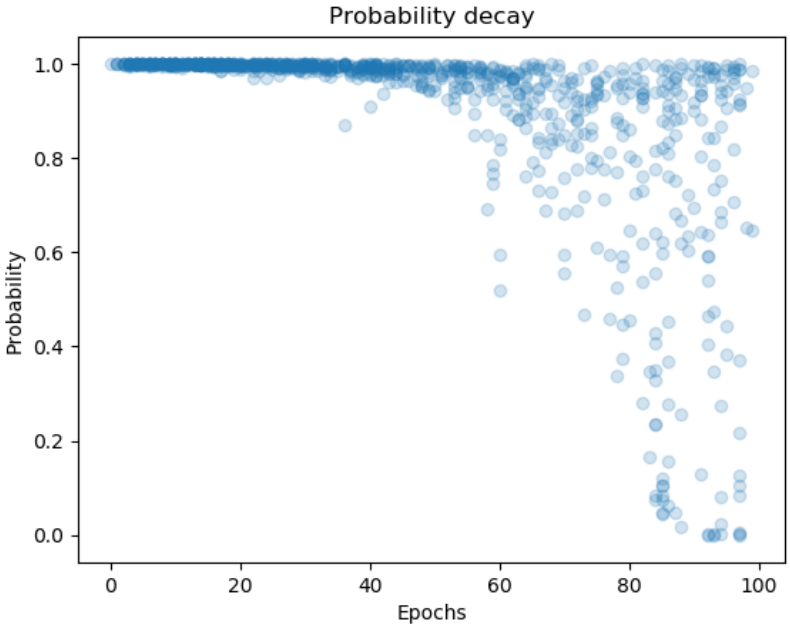


Figure 4.1: Probability decay during training (AlexNet, set H_1, $T_0 = 1$, $\alpha = 0.93$). At each worsening move, the probability has been saved. Most of the training time is spent in the diversification phase.

4.2 Results

The algorithm has been tested with two different set of parameters. The first having higher learning rates than the second one.

$$H_1 = \{0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.09, 0.08, 0.07, 0.06, 0.05\}$$

$$H_2 = \{0.5, 0.4, 0.3, 0.2, 0.1, 0.09, 0.08, 0.07, 0.06, 0.05, 0.04, 0.03, 0.02, 0.01\}$$

The cooling factor α has been reduced to 0.93 for all the following tests, since few epochs are used. As depicted in Figure 4.2, the number of worsening moves is much lower than in the previous tests. Results show that the training process is much slower, in particular with the first set H_1, usually leading to worsening moves, but a higher accuracy is reached multiple times with different structures. The first set seems to perform poorly on AlexNet, while it reaches a better accuracy on both VGG16 and ResNet34. Instead, the second set H_2 improves the results on all the networks. Highlighted results are shown in Tables 4.1, 4.2 and show that a slower learning process generally achieves better results. Usually the training accuracy curve is not

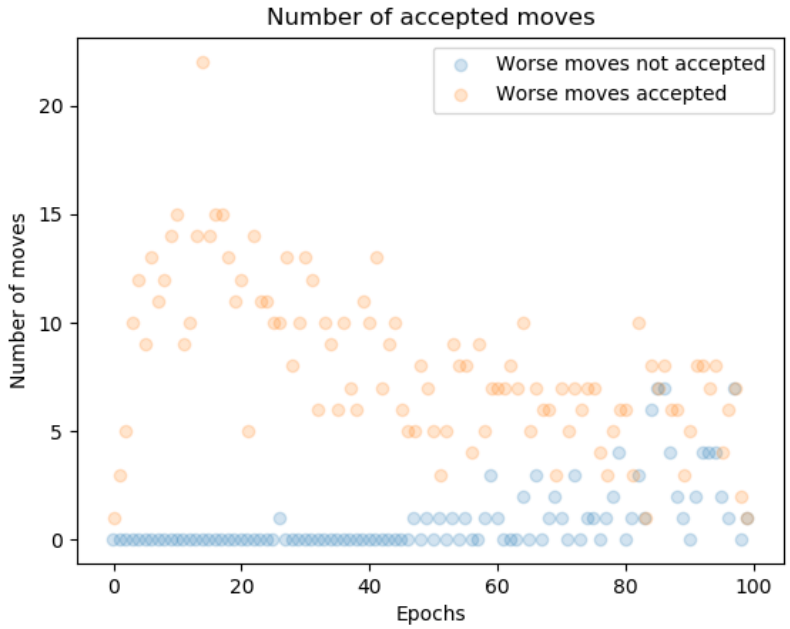
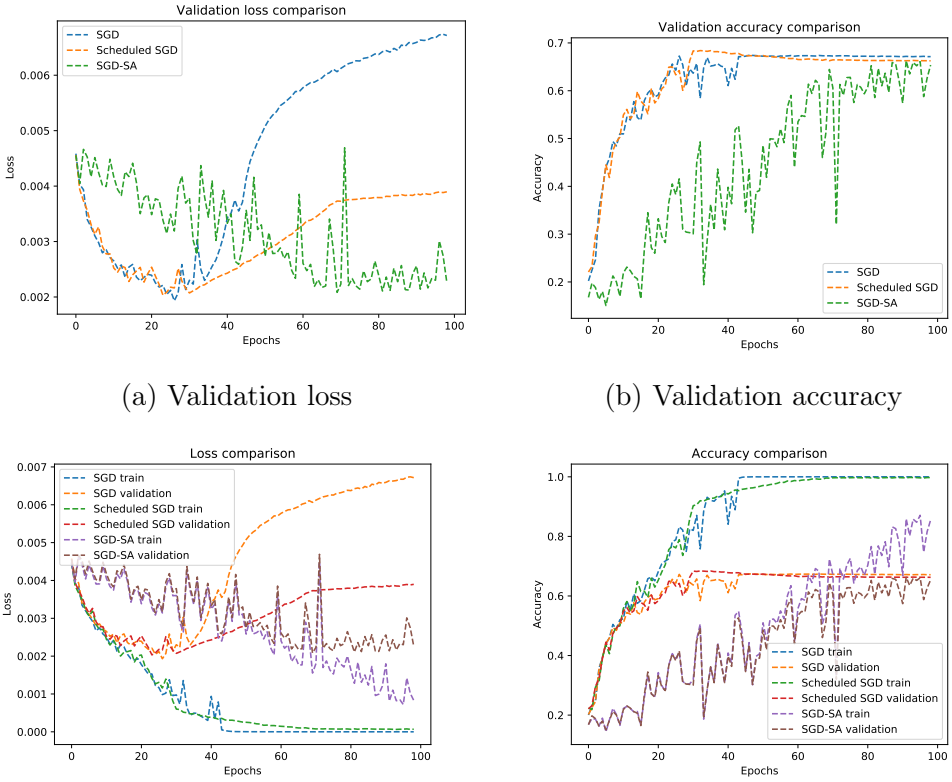


Figure 4.2: The number of worsening moves accepted and not accepted for each epoch (AlexNet, set H_2, $T_0 = 1$, $\alpha = 0.93$). The number of worse moves is two times lower than SSA with random moves.



(a) Validation loss

(b) Validation accuracy

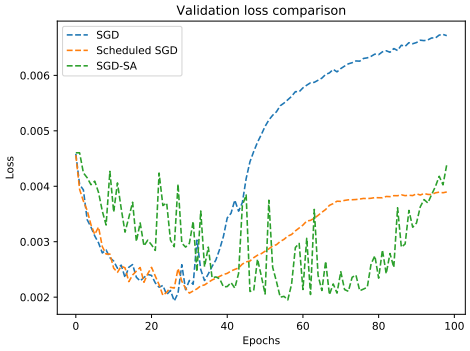
(c) Loss comparison

(d) Accuracy comparison

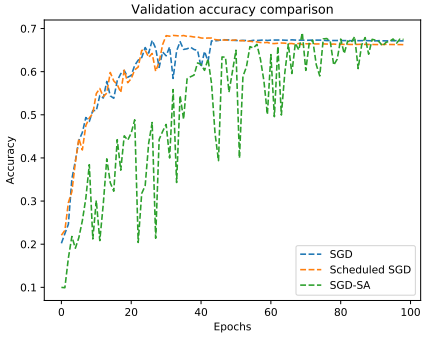
Figure 4.3: AlexNet on CIFAR10. SGD: $\eta = 0.1$, $\beta = 0$. SGD SA: $\alpha = 0.93$, $T_0 = 1$, set H_1. Scheduled SGD: $\eta = 0.1$ first 30 epochs, 0.01 following 40 epochs, 0.001 final 30 epochs.

shown in the same plot with the validation one, but it can seen (Figures 4.4d, 4.5d, 4.7d, 4.8d), that SGD-SA reaches training accuracy equal to 1 in a slower time, but then the validation accuracy is generally higher than SGD. Moreover, the accuracy curve shows another interesting aspect: most of the times scheduled SGD stops the training process with $\eta = 0.1$ in the first 30 epochs and then the curve is steady, resulting in lower validation accuracy.

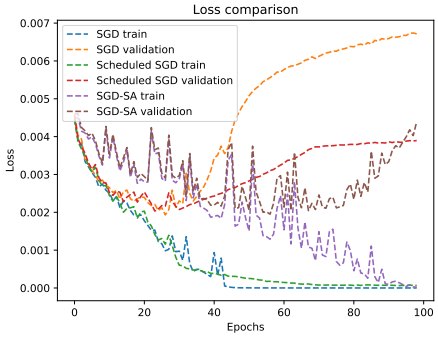
SGD-SA has been proven to be more effective than pure simulated annealing. It allows to reduce overfitting, and similarly to CLR it lets to drop the burden of choosing hyper-parameters: only the set H must be chosen.



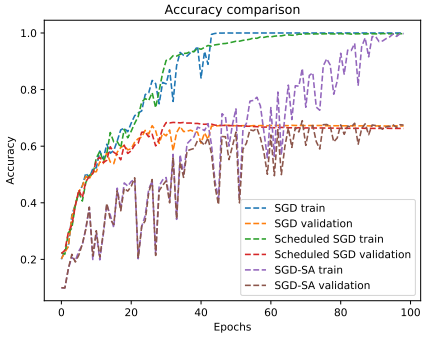
(a) Validation loss



(b) Validation accuracy



(c) Loss comparison

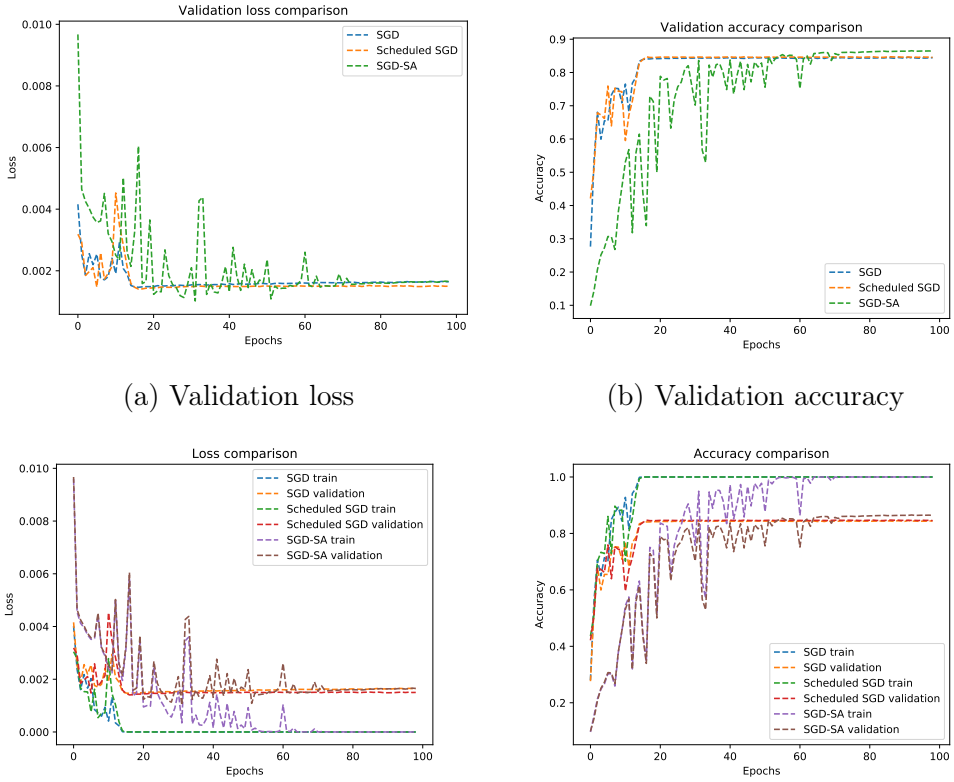


(d) Accuracy comparison

Figure 4.4: AlexNet on CIFAR10. SGD: $\eta = 0.1$, $\beta = 0$. SGD SA: $\alpha = 0.93$, $T_0 = 1$, set H_2. Scheduled SGD: $\eta = 0.1$ first 30 epochs, 0.01 following 40 epochs, 0.001 final 30 epochs.

	Alexnet		VGG16		ResNet34	
	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy
SGD ($\eta = 0.1$)	0.001931	67.41	0.001465	84.6	0.001547	82.28
SGD-SA	0.002083	66.29	**0.001019**	**86.54**	**0.001316**	**83.19**
Scheduled SGD	0.002032	68.4	0.001406	84.73	0.001472	82.58
SGD ($\eta = 0.001$) with momentum	0.002137	66.68	0.001432	84.92	0.001589	83.07

Table 4.1: Set H_1



(a) Validation loss

(b) Validation accuracy

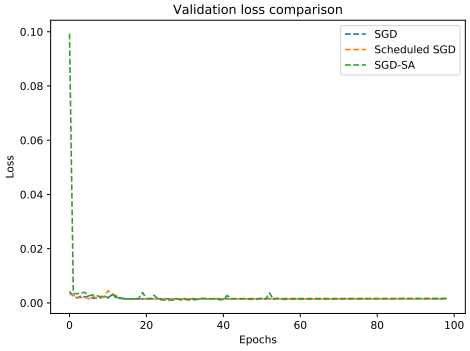
(c) Loss comparison

(d) Accuracy comparison

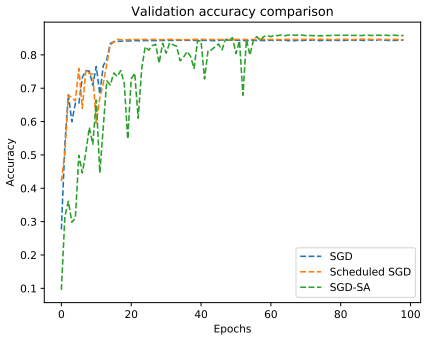
Figure 4.5: VGG16 on CIFAR10. SGD: $\eta = 0.1$, $\beta = 0$. SGD SA: $\alpha = 0.93$, $T_0 = 1$, set H_1. Scheduled SGD: $\eta = 0.1$ first 30 epochs, 0.01 following 40 epochs, 0.001 final 30 epochs.

	Alexnet		VGG16		ResNet34	
	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy
SGD ($\eta = 0.1$)	0.001931	67.41	0.001465	84.6	0.001547	82.28
Scheduled SGD	0.002032	68.4	0.001406	84.73	0.001472	82.58
SGD ($\eta = 0.001$) with momentum	0.002137	66.68	0.001432	84.92	0.001589	83.07
SGD-SA	0.001949	**69.02**	**0.001032**	**85.99**	**0.001119**	**84.06**

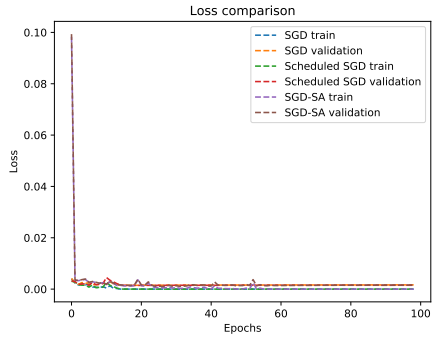
Table 4.2: Set H_2



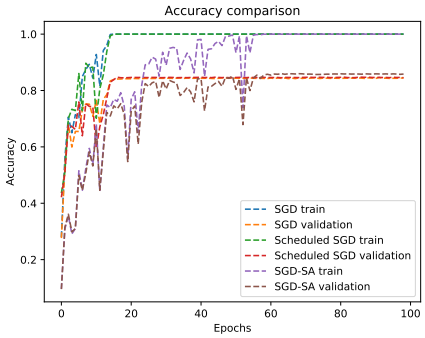
(a) Validation loss



(b) Validation accuracy

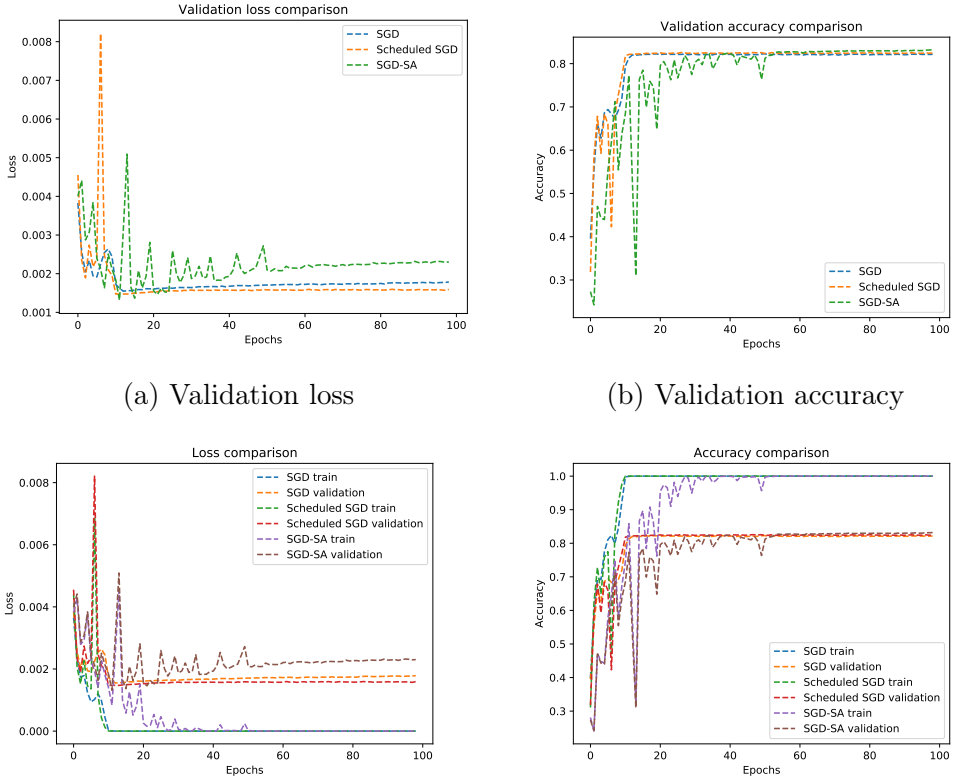


(c) Loss comparison



(d) Accuracy comparison

Figure 4.6: VGG16 on CIFAR10. SGD: $\eta = 0.1$, $\beta = 0$. SGD SA: $\alpha = 0.93$, $T_0 = 1$, set H_2. Scheduled SGD: $\eta = 0.1$ first 30 epochs, 0.01 following 40 epochs, 0.001 final 30 epochs.



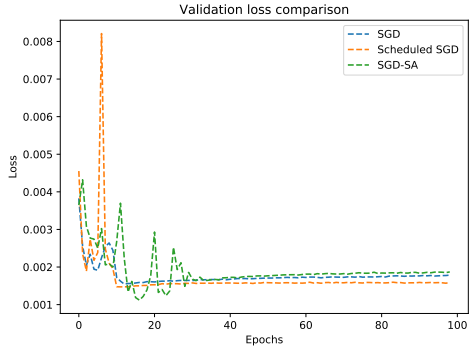
(a) Validation loss

(b) Validation accuracy

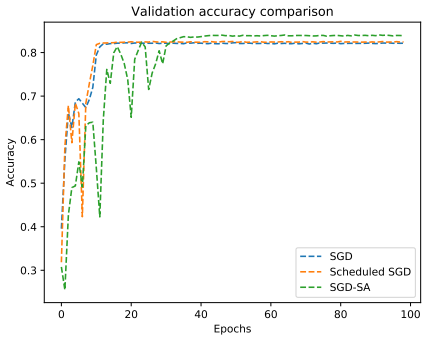
(c) Loss comparison

(d) Accuracy comparison

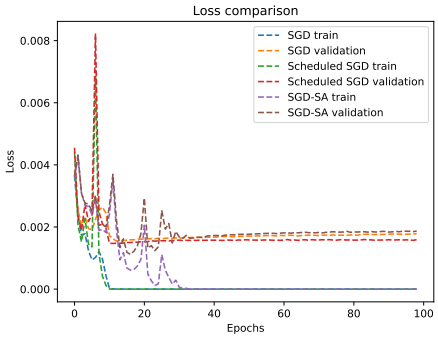
Figure 4.7: ResNet34 on CIFAR10. SGD: $\eta = 0.1$, $\beta = 0$. SGD SA: $\alpha = 0.93$, $T_0 = 1$, set H_1. Scheduled SGD: $\eta = 0.1$ first 30 epochs, 0.01 following 40 epochs, 0.001 final 30 epochs.



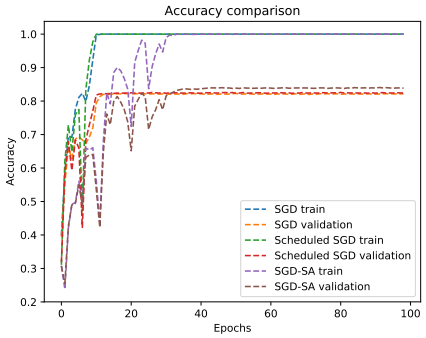
(a) Validation loss



(b) Validation accuracy



(c) Loss comparison



(d) Accuracy comparison

Figure 4.8: ResNet34 on CIFAR10. SGD: $\eta = 0.1$, $\beta = 0$. SGD SA: $\alpha = 0.93$, $T_0 = 1$, set H_2. Scheduled SGD: $\eta = 0.1$ first 30 epochs, 0.01 following 40 epochs, 0.001 final 30 epochs.

Chapter 5

Conclusions

Deep learning growth is constantly pushing new projects in the industry, which recognizes the potential of the new technology.

Our work is aimed at clarifying some points in deep learning. Chapter 2 shows that the learning process and initialization lead to different results. Therefore, each new structure and learning process must be studied in terms of hyper-parameters. Chapter 3 tries to demonstrate that these loss curves are quite easy to train and flat. Then, we have tried to leave the derivative and use, instead, random moves. This work suggests that improvements must be done, since a Gaussian distributed random move is not enough effective. Chapter 4 mixes SGD and SA with the aim to train the network in a faster way using higher learning rates. The new procedure has showed to be not as fast as typical SGD, but gives better results in terms of validation accuracy and loss. These results have been confirmed on all the architectures we tested. Moreover, the new proposed algorithm, allows to have fewer hyper-parameters with respect to scheduled SGD.

Future work could more extensively test Simulated Annealing on neural networks with different generated random moves. The improved results achieved in the last chapter suggest that hyper-parameter tuning by SA should be studied more extensively.

Acknowledgements

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPUs used for this research.

Bibliography

[1] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=2188385.2188395.

[2] Marc Claesen and Bart De Moor. Hyperparameter search in machine learning. *CoRR*, abs/1502.02127, 2015. URL http://arxiv.org/abs/1502.02127.

[3] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2933–2941. Curran Associates, Inc., 2014. URL http://papers.nips.cc/paper/5486-identifying-and-attacking-the-saddle-point-problem-in-high-dimensional-non-convex-optimization.pdf.

[4] Terrance Devries and Graham W. Taylor. Improved regularization of convolutional neural networks with cutout. *CoRR*, abs/1708.04552, 2017. URL http://arxiv.org/abs/1708.04552.

[5] Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. Sharp minima can generalize for deep nets. *CoRR*, abs/1703.04933, 2017. URL http://arxiv.org/abs/1703.04933.

[6] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2016. URL http://arxiv.org/abs/1603.07285. cite arxiv:1603.07285.

[7] Alberto Franzin and Thomas Stützle. Revisiting simulated annealing: A component-based analysis. *Computers & Operations Research*, 104, 12 2018. doi: 10.1016/j.cor.2018.12.015.

[8] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR. URL http://proceedings.mlr.press/v15/glorot11a.html.

[9] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL `http://arxiv.org/abs/1512.03385`.

[11] 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. URL `http://arxiv.org/abs/1207.0580`.

[12] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2):251–257, March 1991. ISSN 0893-6080. doi: 10.1016/0893-6080(91)90009-T. URL `http://dx.doi.org/10.1016/0893-6080(91)90009-T`.

[13] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. URL `http://arxiv.org/abs/1704.04861`.

[14] Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E. Hopcroft, and Kilian Q. Weinberger. Snapshot ensembles: Train 1, get M for free. *CoRR*, abs/1704.00109, 2017. URL `http://arxiv.org/abs/1704.00109`.

[15] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016. URL `http://arxiv.org/abs/1602.07360`.

[16] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`.

[17] Kenji Kawaguchi. Deep learning without poor local minima. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 586–594. Curran Associates, Inc., 2016. URL `http://papers.nips.cc/paper/6112-deep-learning-without-poor-local-minima.pdf`.

[18] 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. URL `http://arxiv.org/abs/1609.04836`.

[19] Scott Kirkpatrick, C. D. Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220 4598:671–80, 1983.

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

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[22] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

[23] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. doi: 10.1038/nature14539. URL https://doi.org/10.1038/nature14539.

[24] Xiaodan Liang, Zhiting Hu, Hao Zhang, Chuang Gan, and Eric P. Xing. Recurrent topic-transition GAN for visual paragraph generation. *CoRR*, abs/1703.07022, 2017. URL http://arxiv.org/abs/1703.07022.

[25] Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with restarts. *CoRR*, abs/1608.03983, 2016. URL http://arxiv.org/abs/1608.03983.

[26] Marco Marchesi. Megapixel size image creation using generative adversarial networks. *CoRR*, abs/1706.00082, 2017. URL http://arxiv.org/abs/1706.00082.

[27] Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *CoRR*, abs/1511.06807, 2015. URL https://arxiv.org/abs/1511.06807.

[28] Yurii Nesterov. A method of solving a convex programming problem with convergence rate O(1/sqr(k)). *Soviet Mathematics Doklady*, 27:372–376, 1983. URL http://www.core.ucl.ac.be/~{}nesterov/Research/Papers/DAN83.pdf.

[29] Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. Norm-based capacity control in neural networks. In Peter Grünwald, Elad Hazan, and Satyen Kale, editors, *Proceedings of The 28th Conference on Learning Theory*, volume 40 of *Proceedings of Machine Learning Research*, pages 1376–1401, Paris, France, 03–06 Jul 2015. PMLR. URL http://proceedings.mlr.press/v40/Neyshabur15.html.

[30] Behnam Neyshabur, Srinadh Bhojanapalli, David Mcallester, and Nati Srebro. Exploring generalization in deep learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5947–5956. Curran Associates, Inc.,

2017. URL http://papers.nips.cc/paper/7176-exploring-generalization-in-deep-learning.pdf.

[31] Arild Nøkland and Lars Hiller Eidnes. Training Neural Networks with Local Error Signals. *arXiv e-prints*, art. arXiv:1901.06656, Jan 2019.

[32] Yuji Roh, Geon Heo, and Steven Euijong Whang. A survey on data collection for machine learning: a big data - AI integration perspective. *CoRR*, abs/1811.03402, 2018. URL http://arxiv.org/abs/1811.03402.

[33] Sebastian Ruder. An overview of gradient descent optimization algorithms., 2016. URL http://arxiv.org/abs/1609.04747. cite arxiv:1609.04747Comment: Added derivations of AdaMax and Nadam.

[34] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014. URL http://arxiv.org/abs/1409.0575.

[35] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, New York, NY, USA, 2014. ISBN 1107057132, 9781107057135.

[36] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL http://arxiv.org/abs/1409.1556.

[37] Leslie N. Smith. Cyclical learning rates for training neural networks, 2015. URL http://arxiv.org/abs/1506.01186. cite arxiv:1506.01186Comment: Presented at WACV 2017; see https://github.com/bckenstler/CLR for instructions to implement CLR in Keras.

[38] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489, 2017. URL http://arxiv.org/abs/1711.00489.

[39] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015. URL http://arxiv.org/abs/1409.4842.

[40] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[41] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *CoRR*, abs/1605.07146, 2016. URL http://arxiv.org/abs/1605.07146.

[42] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 818–833, Cham, 2014. Springer International Publishing. ISBN 978-3-319-10590-1.

[43] Hongyi Zhang, Yann N. Dauphin, and Tengyu Ma. Fixup initialization: Residual learning without normalization. *CoRR*, abs/1901.09321, 2019. URL `http://arxiv.org/abs/1901.09321`.