Learning process for Deep Neural Networks

Special Topic report for the *B6.2 Optimisation for Data Science* course taught by Prof. Raphael Hauser and Prof. Coralia Cartis
Hillary term 2021-2022

Candidate Number 1064551

Contents

1 Introduction **2**

2 Representative problem **2**
2.1 The Deep Neural Network Setup 2
2.2 Characterization of optimization problem 3

3 Optimization tools **7**
3.1 Stochastic gradient descent 8
3.2 Nesterov Acceleration . . . 11

4 Acceleration methods **12**
4.1 RMSProp 13
4.2 Adam 13
4.3 Final performance comparison 16

5 Conclusion **17**

1 Introduction

In this special topic we move towards an appropriate learning algorithm. Starting from gradient descent, bolt on some new features (Nesterov and SGD), then introduce RMSProp to motive Adam. We will do in the context of a machine learning problem: recognition of handwritten digits using the MNIST dataset. That will also enable us to motivate some differences between pure optimization and machine learning problems. This report assumes a basic understanding of the structure and forward pass of a deep neural network and gradient descent.

2 Representative problem

To bring context to the optimization techniques discussed in this special topic, and to introduce the difference between a learning process and pure optimization, we shall consider a specific machine learning problem: the recognition of handwritten digits.

2.1 The Deep Neural Network Setup

Handwritten digit recognition, using the MNIST dataset, is one of the introductory problems in deep learning, [9]. It is primarily used as a benchmark to compare different machine learning algorithms, which is what we too will do.

The dataset consists of 70 000 images (called features $x_i \in X$), each with a label $y_i \in Y$ corresponding to the number they represent. An image is an array of 28 by 28 pixels, each with a grayscale value, [9]. The data is subdivided into training, validation and test datasets, as explained in Subsection 2.2.

The Neural Network consists of 3 fully connected layers, mapping from the $28 \cdot 28$ dimensional input vector to a 10 dimensional, corresponding to a one-hot encoding (highest output is network prediction) of the possible digits, Figure 1. The first 2 layers use relu, and the final layer has a softmax activation function. The parameters are randomly initialized using a standard normal distribution, scaled with 0.01.

Implementation. The neural network and optimisation algorithm are implemented in Python using the Jax [1] library. The code (especially the implementations of vectorization, just-in-time compilation and gpu acceleration) were inspired by an article

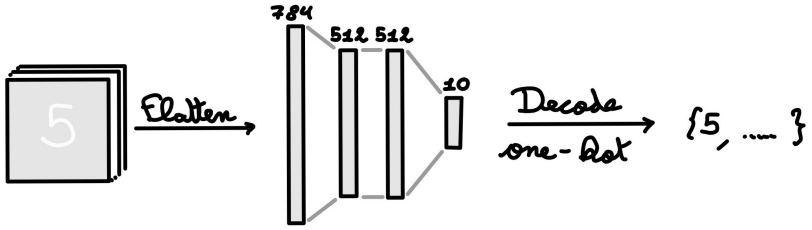


Figure 1: Architecture of the neural network. First the image is flattened into a $28 \cdot 28$ sized vector, then passed through the three layers. Finally the output vector is trained to be a one-hot encoding of the 10 classes, such that $argmax$ provides the network's prediction.

written by Robert Lange [8]. Jax was chosen for its lightweight yet powerful automatic differentiation system and acceleration tools. The optimization process, over the full dataset, uses just under 300MB of GPU memory. The entire repository can found at `https://gitlab.com/th3et3rnalz/optimization_special_topic`.

2.2 Characterization of optimization problem

Learning the optimal parameters θ, differs from pure optimization in a few key ways. These mostly stem from the fact that we have but a finite sample of example mappings (from feature x_i to label y_i), but want to obtain a model that performs well over the entire space (from feature space $\hat{\mathcal{X}}$ to label space $\hat{\mathcal{Y}}$). Another large difference is that the learning problem consists of a nested optimization problem. Beyond optimizing for the weights and biases, we are optimizing hyperparameters (in an outer optimization loop) such that the trained model better approximates the desired mapping.

2.2.1 Objective function

The objective of the learning process is to find θ such that:

$$\min_{\theta} \; f(\theta) := \hat{\mathcal{L}}(\theta; \hat{\mathcal{X}}, \hat{\mathcal{Y}})$$

where $\hat{\mathcal{L}}$ is the loss function over the entire space. In practice, we have but a finite set of samples of this space, $(\mathcal{X}, \mathcal{Y})$. A subset of this, the training dataset $(\mathcal{X}_t, \mathcal{Y}_t)$, is used for optimization. Hence, as a proxy metric to the true objective, we minimize

$$\mathcal{L}(\theta, \mathcal{X}_t, \mathcal{Y}_t) := \frac{1}{m} \sum_{i=1}^{m} l(\theta, x_i, y_i) \tag{1}$$

where m is the cardinality of the training dataset, $x_i \in \mathcal{X}_t$, $y_i \in \mathcal{Y}_t$, and l is the (currently unspecified) loss function.

Accuracy We want to train the model to achieve the highest accuracy possible, where accuracy is defined as the number of correct classifications over the total number of predictions.

$$A := \frac{\#\text{ correctly classified images}}{\#\text{ of images}} \tag{2}$$

However, due to the discrete nature of this metric, it cannot be used as a loss function.

Multi-class cross entropy A common loss function for a classification problem such as this one is the Multi-class Cross-Entropy (MCE). First we pass the output of the neural network through a softmax function:

$$\hat{y} = \text{softmax}(\bar{y}) = \frac{e^{\bar{y}}}{\sum_{j=1}^{n} e^{\bar{y}_i}} \tag{3}$$

which maps the vector $\bar{y} \in \mathbb{R}^n$ to non-negative values that sum to 1, which can be interpreted as probability. A higher value thus corresponds to higher certainty of the feature belonging to a given class. The MCE is defined as:

$$l(\theta, x) = -\sum_{j=1}^{n} y_j \log(\hat{y}_j) \tag{4}$$

where $\hat{y} = N(x; \theta) : \mathbb{R}^{28 \cdot 28} \to \mathbb{R}^{10}$ is the neural network output and y is the label, with a one-hot encoding. A one hot encoding of a label is a vector with one nonzero entry corresponding to which of the classes the feature belongs to.

This loss function was chosen as it emphasizes poor performance (the gradient of the log is much larger when the prediction accuracy is close to zero). In a classification problem, we just need the model to correctly predict the class, not necessarily have the corresponding probability be 1, such that this an appropriate choice.

2.2.2 Capacity: underfitting and overfitting

The learning process will optimize the model over the training dataset, but we actually want the model to perform well over the entire subspace [2], over all possible pictures of handwritten digits. The learning process then has a dual aim: both to reduce the training error, and to keep the gap between training loss $\mathcal{L}$ and true loss $\hat{\mathcal{L}}$ small.

As a proxy metric to true error, we measure the accuracy over a dataset disjoint from the training dataset: the test dataset. Both datasets are assumed i.i.d.: idependent (each sample is independent from the others) and identically distributed (both datasets are drawn from the same underlying distribution).

Underfitting and overfitting. Comparing both the training and test error during the training process reveals two competing pitfalls that we aim to avoid: underfitting and overfitting, visualized in Figure 2. Underfitting occurs when the training error doesn't get small enough. Overfitting occurs when the gap between training and test errors is too large. We control which is more likely to happen by controlling the model's capacity: its ability to fit a wide variety of functions.

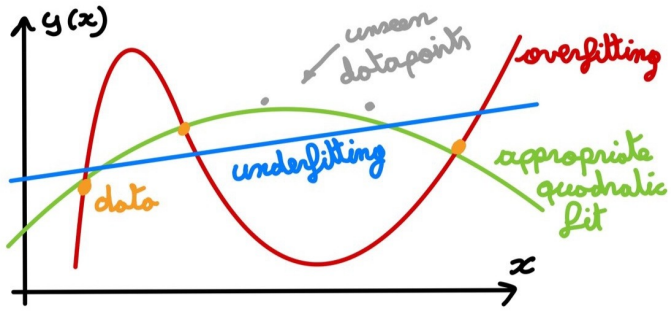


Figure 2: Comparison of underfitting and overfitting for a linear regression problem. If a linear model is used, it cannot accurately represent the data (capacity of the model is too low). On the other hand, a fifth order model does not generalize well to the unseen points.

Model capacity. One way to control the model's capacity is to modify the hypothesis space. For a linear regression, that would mean increasing the order of the polynomial. For a deep neural network this means increasing the number of layers or nodes in a layer. However, capacity is not only determined by the choice of model. We have merely considered the model's representational capacity: the family of functions that the learning algorithm can choose from. There are additional limitations, such that the effective capacity may be less than the model's representation capacity.

There are theoretical results that bound the training and generalization error, but they are generally not used in deep learning, partly because the model capacity is hard to precisely determine. This is due to the additional limitations imposed by the learning algorithm, which has to optimize a strongly non-convex objective.

It is for these reasons that we consider the loss function progression as an indicator of the optimization procedure, and consider the test accuracy as an indicator of model performance.

2.2.3 Hyperparameters and validation sets

There are certain parameters, called hyperparameters, that while unknown, are not to be optimized by the learning algorithm. That may be because they are hard to optimize for, or because it is not appropriate to learn them on the training set.

Certain hyperparameters control the carrying capacity of the model (e.g. the order of the polynomial in linear regression). If we let the learning model modify the order of the polynomial, it will grow without bound. Similarly, for a neural network this would just increase the number of layers and nodes.

Nevertheless, we need to set them, which is where the validation set comes in. While the training dataset is used to train the model, the validation dataset is used to guide the selection of hyperparameters. We then have three datasets: the training, validation and testing datasets, of which the training dataset is usually the largest. After training the θ on the training dataset, the performance is evaluated on the validation dataset, which is then used to guide the selection of new hyperparameters.

2.2.4 Characterization of the optimization landscape

To choose an optimization algorithm, it helps to understand the nature of the to-be-optimized function. Even though neural networks are highly non-convex, we present results indicating that the training process may be tractable.

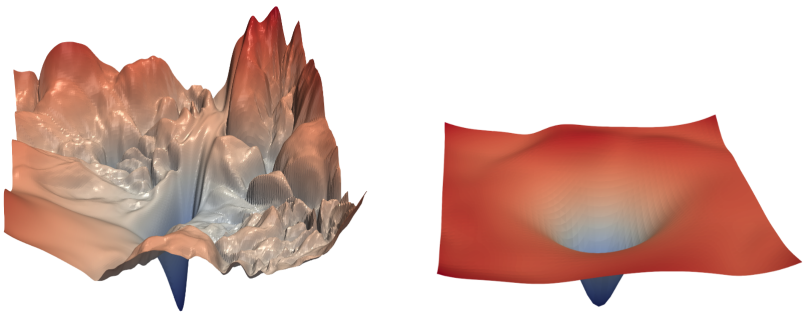
Wide neural networks Wide neural networks, with linear activation functions have no confined points [10], where a confined point is defined as a point from where there is no non-increasing path towards a global minimum, 2.1.

Theorem 2.1: Confined points for Linear Wide NNs [10, Theorem 3]

Consider a feedforward, fully connected neural network, used to model a mapping $\mathcal{D}_x \subset \mathbb{R}^d \to \mathcal{D}_y \subset \mathbb{R}^m$, with parameters θ. The network has l layers with width w is the minimal width of all layers, and the training set has n points. We assume the loss function to be convex in its first argument.
Then, if $w > 2m(n+1)$ and the activations are linear, the objective has no fixed points.

This implies we can reach a global minimum from any starting point, assuming the optimization algorithm does not get stuck on a saddle point. Even though this has been proven for a very restrictive case, it does correlate with empirical findings showing that the training process for non-linear activation functions is also feasible to near-zero loss, [10].

Deep neural networks are usually preferred as they approximate functions with fewer parameters, as shown by Telgarsky [14]. However, Li et al. [11] show, using dimensionality reduction to visualize the loss landscape, that the landscape becomes much less convex, Figure 3a. However, by adding skip connections, the landscape becomes much more convex and training to a global minimum becomes feasible, Figure 3b. Skip connections add layers outputs to other that are further down the forward pass.



(a) without skip connections (b) with skip connections

Figure 3: Visualization of the loss surfaces of ResNet-56 (A deep neural network with 56 layers) with/without skip connections [11]

Lederer [10] implies that if a (linear) network is wide enough, it is feasible to find a global minimum, and Li et al. show that this behaviour can be recovered for deep neural networks using well a chosen architecture. This provides an indication that it may be possible to train general neural networks to an acceptable minimum.

3 Optimization tools

We shall consider modifications we can make to the steepest descent algorithm, to create an appropriate optimization algorithm for this problem. We'll consider approximating the gradient and adding a numerical analog to momentum.

3.1 Stochastic gradient descent

When the training dataset becomes very large, it may no longer be feasible to compute the full gradient. An alternative is then to subdivide the training set, and compute an approximation to the true gradient:

$$\theta^{k+1} = \theta^k - \alpha^k g^k \quad \text{where} \quad g^k = \frac{1}{|S_k|}\nabla_\theta \sum_{\mu \in S_k} l(\theta; x_\mu, y_\mu) \tag{5}$$

where $S_k \subset \{1, 2, \ldots, m\}$ is chosen randomly from a uniform distribution. The cardinality of S_k, $|S_k|$, is called the batch size. $|S_k| = 1$ corresponds to on-line training, and for $1 < |S_k| < m$ the method is called mini-bath. This reduces the computational cost (and hence time) of a single update step. It does however, introduce a stochastic nature to the optimization process, such that the approximation of the gradient no longer guarantees descent, only in expectation, [5, Chapter 7.3].

$$\mathbb{E}_{S_k}[G^k] = \mathbb{E}[G^k|S_k] = \sum_{j=1}^{m} E[G^k|S_k = j] \cdot \mathbb{P}[S_k = j] = \sum_{j=1}^{m} \nabla f_j(\theta^k) \cdot \frac{1}{m} = \nabla f(\theta^k) \tag{6}$$

where G^k is the underlying distribution from which g^k is drawn. As SGD is similar to GD working on a different dataset at every iteration, we can no longer expect monotonically decreasing loss. In our case, the training set will once be randomly subdivided into batches of size $|S_k|$, which will be iterated over. We then define one epoch as an iteration over the entire training dataset. Though computationally convenient, the convergence behaviour is also modified, Theorem 3.1.

Theorem 3.1: SGD Convergence [5, Chapter 7.3, Theorem 9]

Let f be bounded below by f_{low}, $\nabla f_j(\theta)$ be L-smooth (and hence ∇f is L-smooth too, same L), and that there exists and $M > 0$, such that:

$$Var(G^k|S_k) := \mathbb{E}[||G^k - \nabla f(\theta^k)||_2^2|S_k] \leq M. \tag{7}$$

Apply SGD with $|S_k| = 1$ and fixed stepsize $\alpha = \eta/L$, where $\eta \in (0, 1]$. Then for $k \geq 1$:

$$\min_{0 \geq 1 \geq k-1} \mathbb{E}[||\nabla f(\theta^i)||_2^2] \leq \eta M + \frac{2L(f(\theta^0) - f_{low})}{k\eta} \tag{8}$$

Through additional smoothness assumptions, one can show that under the same assumptions as Theorem 3.1:

$$\lim_{k \to \infty} \mathbb{E}[||\nabla f(\theta^k)||_2^2] \leq \eta M \tag{9}$$

Hence, for the SGD method, we can only achieve convergence up to a certain bound, called the noise floor. To achieve convergence, we can either modify our assumption of constant step size (letting $\alpha_k \to 0$ with $k \to \infty$) or we can asymptotically increase the batch size to the full dataset ($|S_k| \to m$ with $k \to \infty$).

One might therefore assume that the larger the batch size, the better the convergence rate (as we have a more complete estimate of the gradient) and we'll be able to achieve a lower final error bound.

Comparing batch size Let us compare the training process for batch sizes 64 and 256 (multiples of 2 are chosen as this is beneficial for GPU acceleration). Following the scaling rule proposed by Goyal et al. [3] ("When the minibatch size is multiplied by k, multiply the learning rate by k"), we compared the loss progression, epoch time and test accuracy over 10 epochs, Figure 4, from which a few observations can be made:

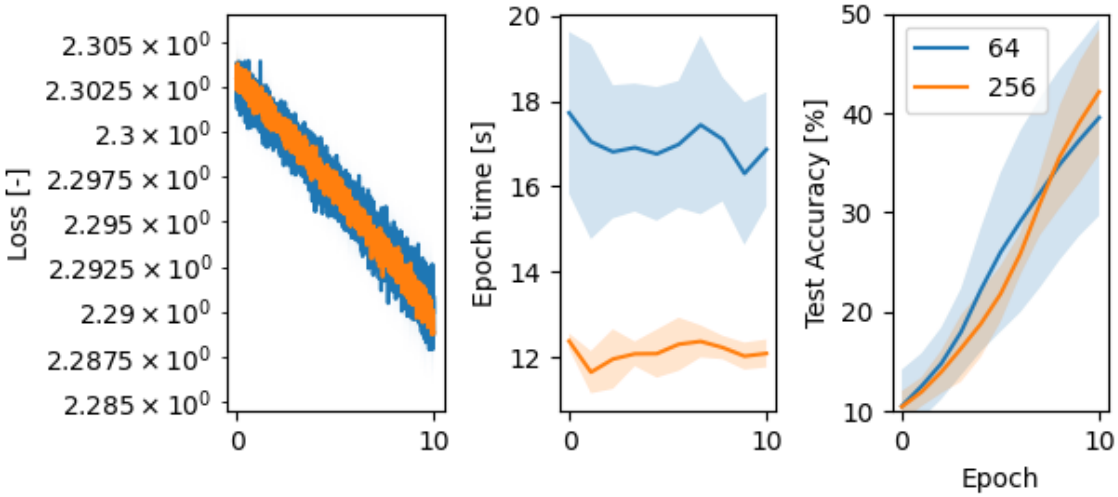


Figure 4: Comparison of convergence behaviour for training deep neural network via SGD with different batch sizes ($b = 64$ and $b = 256$), and $\alpha = 10^{-3} \cdot b/64$. The lines show the values averaged over 5 runs, and the shaded area adds a confidence band of one standard deviation (very small for loss plot). The final accuracy is $A_{b=64} = 39.6\% \pm 9.9$ and $A_{b=256} = 42.16\% \pm 6.3$ (mean $\pm$ standard deviation).

1. The variance in the loss indeed seems smaller for a larger batch size

2. The general loss progression for both batch sizes is quite similar, and the difference between the final test accuracies is statically insignificant.

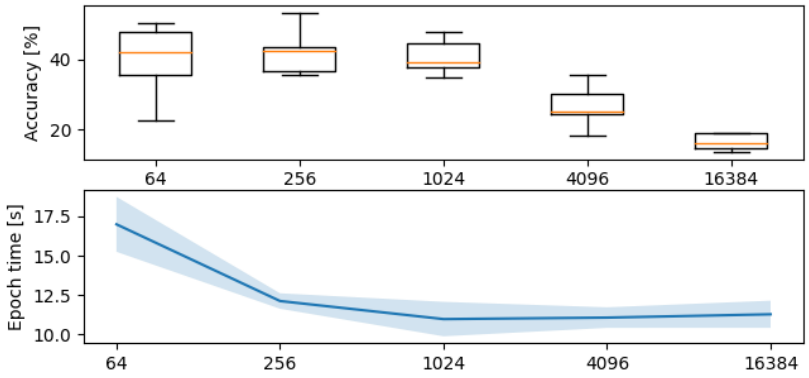


Figure 5: Comparison of accuracy and epoch time for different batch sizes, averaged over 5 runs. Note the statistically significant performance decay for batch sizes above 1024.

3. The just-in-time compilation of the coefficient update function (which needs to be redone for different batch sizes) has a significant effect on the training time for the first epoch

4. The computational cost (visible through epoch time) is drastically reduced for a larger batch, which is due to the optimizations possible when accelerating with a GPU.

This supports the initial guess that larger batch sizes are indeed beneficial, but not because of the reduction in noise, rather due to the reduced epoch time. However, Goyal et al. show numerically that there is a critical batch size above which the final test accuracy is lower [3, Figure 1]. We repeat the training process for larger batch sizes, noting that the linear scaling rule breaks down for batch sizes of 8192 and above, there the step size at $0.001 \cdot 4096/64$.

A possible justification, proposed by Wilson and Martinez [4] is related to the relationship between the sampling frequency of the loss landscape. For a large batch size, there will be little noise around the true gradient, but it will be sampled only relatively few times, while for a small batch size, while more noisy, yields more frequent sampling of the loss landscape. This more frequent sampling leads to a faster convergence rate (until we approach the noise floor). By increasing the step size, with batch size, we (apparently) don't loose much until we get to a critical size. As larger batch sizes have smaller epoch times, there is a sweet spot batch size for neural network training.

3.2 Nesterov Acceleration

Momentum based methods incorporate information from the previous step to bias the current step. A pure momentum step is defined as

$$\theta^{k+1} = \theta^k - \alpha^k \nabla f(\theta^k) + \beta(\theta^k - \theta^{k-1}). \tag{10}$$

where $\beta \in (0, 1)$ is the momentum decay constant. This embodies the physics concept of momentum: if an object a moving at a certain velocity, it takes time to change direction or slow down. Nesterov accelerated gradient (NAG) takes this one step further and samples the gradient at a momentum estimated point:

$$\theta^{x+1} = \theta^k - \alpha^k \nabla f\left(\theta^k + \beta(\theta^k - \theta^{k-1})\right) + \beta(\theta^k - \theta^{k-1}) \tag{11}$$

Theorem 3.2: Convergence rate NAG [5, Section 6.4, Theorem 8]

Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be a convex L-smooth function that has at least one finite minimizer $\theta^* \in \operatorname{argmin} f(\theta)$ and a finite minimum $f^* = \min f(\theta)$. Then the sequence $(\theta^k)_{k\in\mathbb{N}}$ of iterates produced by the nesterov acceleration algorithm for L-smooth functions satisfies

$$f(\theta^k) \leq f(\theta^*) + \frac{4L||\theta^0 - \theta^k||^2}{(k+2)^2} \tag{12}$$

for all $k \in \mathbb{N}$.

Theorem 9 of [5] shows that, for an L-smooth, γ convex function, with constant step size, the convergence rate is of NAG is faster than steepest descent:

$$1 - \sqrt{\frac{\gamma}{L}} < 1 - \frac{\gamma}{L}, \tag{13}$$

especially for ill-conditioned problems (where $\gamma \ll L$).

One might expect that to also have acceleration for the non-convex optimisation problem that we have. However, when combined with SGD, it does not, in general, provide an acceleration over SGD, in contrast to the deterministic, strongly convex scenario.

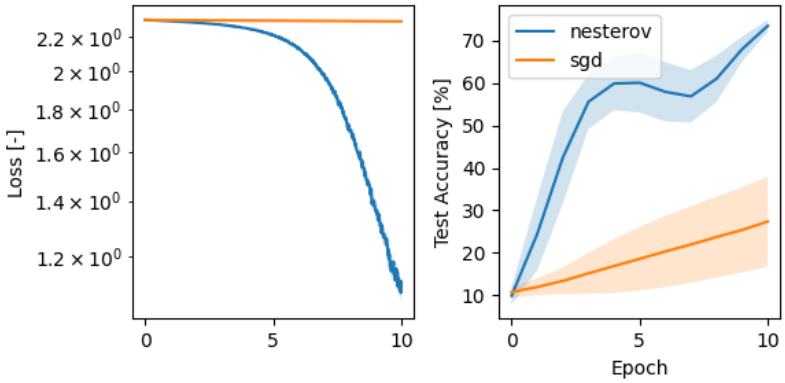


Figure 6: Comparison between SGD+Nesterov and SGD for a batch size of 1024, learning rate of $\alpha = 0.001$ and $\beta = 0.9$. The lines indicate the mean, and the confidence bands one standard deviation, for 5 runs.

Theorem 3.3: Non-acceleration of SGD+Nesterov [12]

Let $(x_i, y_i)_{i=1}^n$ be a dataset generated according to the component decoupled model [WTF IS THAT?]. Consider the optimization problem of minimizing the quadratic function $\frac{1}{2n}\sum_i (x_i^T w - y_i)^2$. For any step size $\eta > 0$ and momentum parameter $\gamma \in (0, 1)$ of SGD+Nesterov with random initialization, with probability one, there exists a $T \in \mathbb{N}$ such that $\forall t > T$,

$$\mathbb{E}[f(w_t)] - f(w^*) \geq \left(1 - C\frac{\gamma}{L}\right)^t [f(w_0) - f(w^*)] \tag{14}$$

where $C > 0$ is a constant.

Theorem 3.3 shows an example of a specific objective function, for which adding Nesterov to SGD does not provide acceleration. Hence, in the general case, we cannot conclude that SGD+Nesterov provides acceleration over pure SGD

In this specific situation, for the same learning rate, Nesterov+SGD does provide an improvement over pure SGD, as can be seen in Figure 6. This improvement is both in terms of the value of the loss, and the test accuracy.

4 Acceleration methods

Having introduced SGD and momentum based methods, and their pitfalls, we can now better understand optimization algorithms commonly used in deep learning. For both methods presented, we can arbitrarily choose to use SGD or the full gradient.

4.1 RMSProp

RMSProp is an (unpublished) adaptive learning rate optimizer, proposed by Hinton [6]. It adapts the learning rate depending on the history of the squared gradient in a given direction:

$$v_t = \beta \cdot v_{t-1} + (1-\beta) g_t^2 \tag{15}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} \cdot g_t \tag{16}$$

where g_t is either the full gradient or a stochastic estimate (for SGD), α is the learning rate, $\epsilon = 10^{-8}$ is added to prevent division by zero, and all multiplications and divisions are performed element-wise. The velocity v_t is an exponentially average of the squared gradient.

This modification of the learning rate specific per direction/variable, increases the step size when the gradient is large relative to the history, and decreases it for a direction where the gradient is small. This potentially enables us to choose a larger α without encountering divergence.

To analyze the effect of this coordinate specific learning rate multiplication factor, set $\epsilon = 0$ and rewrite in one equation:

$$\theta_{t+1} = \theta_t - \alpha \operatorname{sign}(g_t) \sqrt{\frac{g_t^2}{\beta \cdot v_{t-1} + (1-\beta) g_t^2}}. \tag{17}$$

v_t can be understood to be a (biased) historical average of g_t^2, thus the fraction g_t^2 / v_t indicates whether the gradient in that parameter is increasing or decreasing. If g_t^2 is increasing (steeper gradient), the stepsize is reduced, and if g_t^2 is decreasing (flattening) the stepsize is increased. In effect, for every parameter, it tries to "jump over" flat planes and not jump over cliffs (where a local minimum may lie).

4.2 Adam

Adam is an optimization technique that combines RMSProp and momentum, and adds bias correction, [7]. On this specific problem, it has similar performance to RMSProp, though seems to be more stable, Figure 8.

4.2.1 The algorithm

Let $f(\theta)$ be a noisy objective function (in our case the loss function made noisy due to SGD), differentiable w.r.t. θ, whose expected value $\mathbb{E}[f(\theta)]$ we are interested in minimizing.

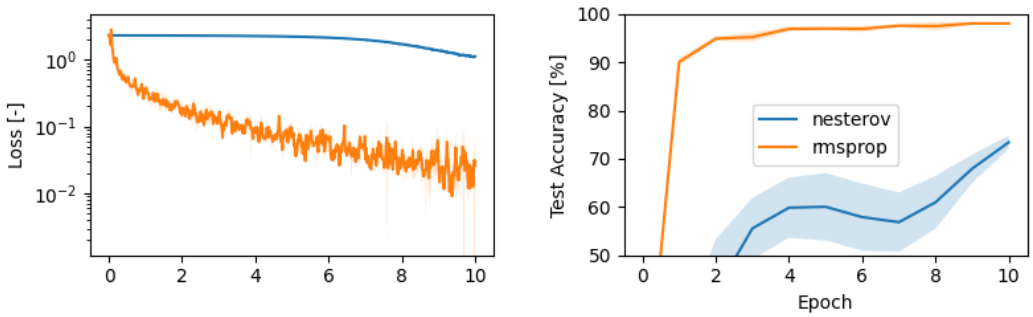


Figure 7: Comparison of Nesterov and RMSProp over 5 runs with shaded error bounds of one strandard deviation.

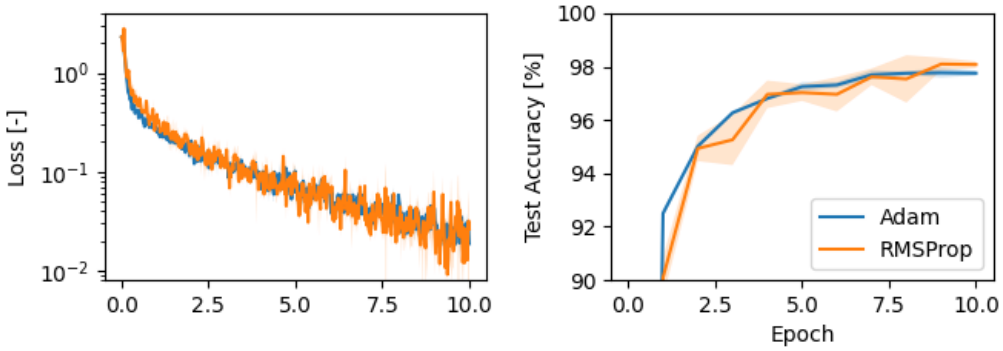


Figure 8: Comparison of Adam and RMSProp for 5 runs, with a batch size of 1024, $\alpha = 0.001$, $\beta = \beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. The final accuracy is $A_{Adam} = 97.7 \pm 0.10$ and $A_{RMSProp} = 98.1 \pm 0.13$ (in format mean % $\pm$ standard deviation).

The algorithm iteratively updates exponential moving averages of the gradient (m_t) and the squared gradient (v_t), as controlled by $\beta_1, \beta_2 \in [0, 1)$. These are initialized as zero vectors, and thus biased towards zero, especially for the initial steps and for small β_1, β_2. Correcting this bias yields $\hat{v}_t$ and $\hat{m}_t$. Here too ϵ (usually chosen 10^{-8}) is added to prevent division by zero errors.

The update equations for Adam are:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{18a}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{18b}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{18c}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t \tag{18d}$$

The effective step at time t, with $\epsilon = 0$ is $\Delta_t = -\alpha \cdot \hat{m}_t/\sqrt{\hat{v}_t}$. If all previous gradients were zero, extreme sparsity, the effective step size is bounded above by $|\Delta_t| \leq \alpha \cdot (1-\beta_1)\sqrt{1-\beta_2}$ in the case $(1-\beta_1) > \sqrt{1-\beta_2}$. Otherwise it is bounded above by $|\Delta_t| \leq \alpha$. Usually, the gradient will be similar to that on the previous steps, such that $\hat{m}_t/\sqrt{\hat{v}_t} \approx \pm 1$, and hence the step size α is an approximate bound on the effective step size. This can interpreted as establishing an approximate trust region.

Signal-to-noise. The value of $|\hat{m}_t/\sqrt{\hat{v}_t}|$ will only deviate from unity when the loss landscape changes significantly, when g_t^2 differs from its exponentially weighted average. We can hence say that $\hat{m}_t/\sqrt{\hat{v}_t}$ resembles a signal to noise ratio (SNR). A smaller SNR means there is uncertainty, and smaller step size is required, and vice-versa for a larger SNR. The SNR typically becomes 0 towards an optimum, reducing the effective step size as a form of automatic annealing.

Initialization bias correction. At all time steps, we wish the expectation of m_t to be equal to the expectation of g_t. First rewrite the definition of m_t in explicit form:

$$m_t = (1-\beta_1)\sum_{i=1}^{t} g_i \beta_1^{t-i} \tag{19}$$

Then the expectation is:

$$\mathbb{E}[v_t] = \mathbb{E}\left[(1-\beta_1)\sum_{i=1}^{t} g_i \beta_1^{t-i}\right] \tag{20}$$

$$= \mathbb{E}[g_t](1-\beta_1)\sum_{i=1}^{t} \beta_1^{t-i} + \zeta \tag{21}$$

$$= \mathbb{E}[g_t](1-\beta_1^t + \zeta \tag{22}$$

where the last step is due to a collapsing sum and ζ is an offset that is zero if the true first moment is stationary. It is kept small due to the exponentially decaying average, placing less weight on past terms. Setting $\zeta = 0$, we can define the exponentially corrected first moment:

$$\hat{v}_t = \frac{v_t}{1-\beta_1^t}. \tag{23}$$

just as in the algorithm. The derivation for $\hat{m}_t$ is analogous.

4.2.2 Convergence analysis

Let $f_1(\theta) + f_2(\theta) + \cdots + f_T(\theta)$ be a series of loss functions (in our case corresponding to the mini-batches). At each time t, we must predict θ_t so as to minimize $f_t(\theta)$.

However, since the sequence is unknown, we evaluate the $f_t(\theta_t) - f_t(\theta^*)$ where $\theta^* = \text{argmin}_\theta \sum_{t=i}^{T} f_t(\theta)$, defining regret:

$$R(T) = \sum_{i=1}^{T} [f_t(\theta_t) - f_t(\theta^*)] \tag{24}$$

Theorem 4.1 implies that, if the gradients g_i are bounded and the data features are sparse, then the regret is bounded. Specifically, it is bounded by the sum of by a constant and a term growing with $\sqrt{T}$. This implies:

$$\frac{R(T)}{T} = \mathcal{O}\left(\frac{1}{\sqrt{T}}\right). \tag{25}$$

Under the assumptions of Theorem 4.1, we thus conclude:

$$\lim_{t\to\infty} f_t(\theta_t) - f_t(\theta^*) = 0 \tag{26}$$

Theorem 4.1: Adam Regret bound [7, Theorem 4.1]

Assume that the function f_t has bounded gradients, $||\nabla f_t(\theta)||_2 \leq G$, $||\nabla f_t(\theta)||_\infty \leq G_\infty$ for all $\theta \in R^d$ and distance between any θ_t generated by Adam is bounded, $||\theta_n - \theta_m||_2 \leq D$, $||\theta_n - \theta_m||_\infty \leq D_\infty$ for any $m, n \in \{1, \ldots, T\}$ and $\beta_1, \beta_2 \in [0, 1)$ satisfying $\beta_1^2/\sqrt{\beta_2} < 1$. Let $\alpha_t = \alpha/\sqrt{t}$ and $\beta_{1,t} = \beta_1 \gamma^{t-1}$ with $\gamma \in (0, 1)$. Then Adam achieves the following guarantee, for all $T \geq 1$:

$$\begin{aligned} R(t) \leq & \frac{D^2}{2\alpha(1-\beta_1)} \sum_{i=1}^{d} \sqrt{T\hat{v}_{T,i}} + \frac{\alpha(1+\beta_1)G_\infty}{(1-\beta_1)\sqrt{1-\beta_2}(1-\gamma)^2} \sum_{i=1}^{d} ||g_{1:T,i}||_2 \\ & + \sum_{i=1}^{d} \frac{D_\infty^2 G_\infty \sqrt{1-\beta_2}}{2\alpha(1-\beta_1)(1-\gamma)^2} \end{aligned} \tag{27}$$

4.3 Final performance comparison

Comparing the results of all algorithms, it's easy to come jump to conclusions. Considering Figure 9, Adam and RMSProp seem to be optimal. From the close-up in Figure 8, we can see that RMSProp finds a better solution, though it's not clear whether it is statistically significant. One might be tempted to say that Adam, thanks to the stable performance and high test accuracy is the optimal algorithm, but it has been shown to not converge asymptotically in some scenarios [13], especially in reinforcement learning settings [15].

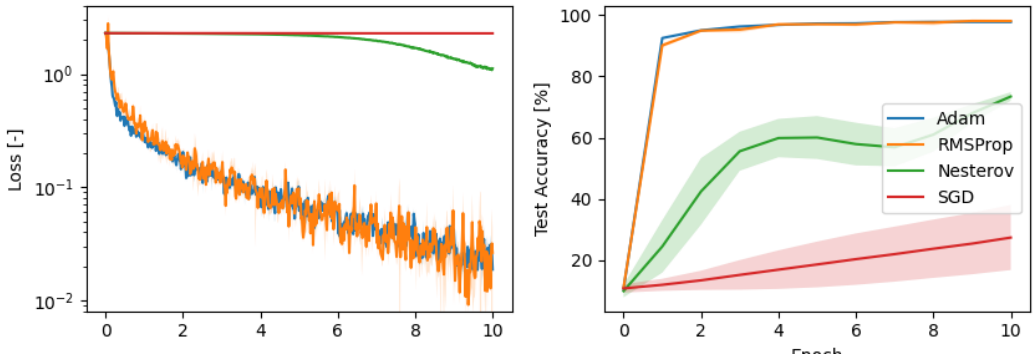


Figure 9: Final comparison of all optimization algorithms presented. Optimization was performed over 10 epochs, with $\alpha = 0.001$, $\beta = \beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, batch size of 1024

More importantly, these are the results without any hyperparameter optimization. Especially the learning rate α has a large effect on performance in this epoch-constrained setting. In practice SGD+Nesterov, RMSProp and Adam are all frequently used algorithms, with the optimal one differing between problems.

5 Conclusion

We have motivated the introduction of Adam, by first introducing SGD, Nesterov and RMSProp. Each of the three steps to Adam can be applied arbitrarily to form an optimization algorithm that is appropriate for a given learning problem. The optimal algorithm is strongly dependent on the type of problem, as even Adam (which seemed optimal in this scenario) can have non-convergence asymptotically.

To find out which of the presented algorithms is optimal, hyperparameter tuning is needed. The model seems to have an appropriate capacity as we don't get a reduction in testing accuracy, when training beyond the peak accuracy. Maybe this means we should increase the capacity to be able to attain a lower test error, which also part of hyperparameter tuning.

One of the main conclusions from this report is that, in the optimization of deep learning problems, we don't approach the asymptotic bounds, neither the noise floor of SGD (though we do see noise in the loss function) or converge towards machine error. Empirical tests are crucial for discovering optimal performance in a computationally constrained environment.

References

[1] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: http://github.com/google/jax.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[3] Priya Goyal, Piotr Dollar, and Ross Girshick. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: *Data @ Scale by Facebook* (June 2017). URL: https://research.facebook.com/publications/accurate-large-minibatch-sgd-training-imagenet-in-1-hour/.

[4] Priya Goyal, Piotr Dollar, and Ross Girshick. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: *Data @ Scale by Facebook* (June 2017).

[5] Raphael Hauzer and Coralia Cartis. *B6.2 Optimisation for Data Science, Lecture Notes*. Mar. 2022. URL: https://courses.maths.ox.ac.uk/course/view.php?id=112.

[6] Geoffrey Hinton. *RMSProp: Divide the gradient by a running average of its recent magnitude*. URL: https://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf.

[7] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[8] Robert Tjarko Lange. "Getting started with JAX (MLPs, CNNs and RNNs)". In: *roberttlange.github.io* (2020). URL: https://roberttlange.github.io/posts/2020/03/blog-post-10/.

[9] Yann LeCun. *The mnist database*. URL: http://yann.lecun.com/exdb/mnist/.

[10] Johannes Lederer. *Optimization Landscapes of Wide Deep Neural Networks Are Benign*. 2020. DOI: 10.48550/ARXIV.2010.00885. URL: https://arxiv.org/abs/2010.00885.

[11] Hao Li et al. “Visualizing the Loss Landscape of Neural Nets”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/a41b3bb3e6b050b6c9067c67f663b915-Paper.pdf.

[12] Chaoyue Liu and Mikhail Belkin. “Accelerating SGD with momentum for over-parameterized learning”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=r1gixp4FPH.

[13] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. “On the Convergence of Adam and Beyond”. In: *CoRR* abs/1904.09237 (2019). arXiv: 1904.09237. URL: http://arxiv.org/abs/1904.09237.

[14] Matus Telgarsky. *Representation Benefits of Deep Feedforward Networks*. 2015. DOI: 10.48550/ARXIV.1509.08101. URL: https://arxiv.org/abs/1509.08101.

[15] Huaqing Xiong et al. “Non-asymptotic Convergence of Adam-type Reinforcement Learning Algorithms under Markovian Sampling”. In: *CoRR* abs/2002.06286 (2020). arXiv: 2002.06286. URL: https://arxiv.org/abs/2002.06286.