Schedae Informaticae Vol. 27 (2018): 47–57
doi: 10.4467/20838476SI.18.004.10409

LOSSGRAD: Automatic Learning Rate in Gradient Descent

Bartosz Wójcik[1], Łukasz Maziarka[2], Jacek Tabor[2]
[1]Faculty of Physics, Mathematics and Computer Science
Cracow University of Technology
[2]Faculty of Mathematics and Computer Science
Jagiellonian University
e-mail: *bartwojc@gmail.com, l.maziarka@gmail.com, jcktbr@gmail.com*

Abstract. In this paper, we propose a simple, fast and easy to implement algorithm LOSSGRAD (locally optimal step-size in gradient descent), which automatically modifies the step-size in gradient descent during neural networks training. Given a function f, a point x, and the gradient $\nabla_x f$ of f, we aim to find the step-size h which is (locally) optimal, i.e. satisfies:

$$h = \arg\min_{t \geq 0} f(x - t\nabla_x f).$$

Making use of quadratic approximation, we show that the algorithm satisfies the above assumption. We experimentally show that our method is insensitive to the choice of initial learning rate while achieving results comparable to other methods.

Keywords: gradient descent, optimization methods, adaptive step size, dynamic learning rate, neural networks

1. Introduction

Gradient methods, with the stochastic gradient descent at the head, play a basic and crucial role in nonlinear optimization of artificial neural networks. In recent years many efforts have been devoted to better understand and improve existing optimization methods. This led to the widespread use of the Momentum method [10] and learning rate schedulers [17], as well as to creation of new algorithms, such as AdaGrad [1], AdaDelta [18], RMSprop [13], Adam [4].

These methods, however, are not without flaws, as they need an extensive tuning or costly grid search of hyperparameters, together with suitable learning rate scheduler – too large step-size may result in instability, while too small may be slow in convergence and may lead to stuck in the saddle points [12].

As the choice of the proper learning rate is crucial, in this paper we aim to find such step which is locally optimal with respect to the direction of the gradient. Our ideas were motivated by Ł4 algorithm [11], which however apply the idea globally, as it computes the linearization of the loss function at the given point and proceeds to the global root of this linearization. Furthermore, unlike the WNGRAD [15], our algorithm can both increase and decrease the learning rate values.

Our algorithm is similar in principle to the line search methods as its aim is to adjust the learning rate based on function evaluations in the selected direction. Line search methods however, perform well only in a deterministic setting and require a stochastic equivalent to perform well with SGD [8]. In opposite to this, LOSSGRAD is also intended to work well in a stochastic setting. The difference is that our algorithm changes the step-size after taking the step thus never requiring any copying of potentially large amounts of memory for network weights.

2. LOSSGRAD

The method we propose is based on the idea of finding a step-size which is locally optimal, i.e. we follow the direction of the gradient to maximally minimize the cost function. Thus given a function f (which we want to minimize), a point x and the gradient[1] $\nabla_x f$, we aim to find the step-size h which is (locally) optimal, i.e. satisfies:

$$h_{opt} = \arg\min_{t \geq 0} f(x - tv), \text{ where } v = \nabla_x f. \tag{1}$$

A natural problem of how to practically realize the above search emerges. This paper is devoted to the examination of one of the possible solutions.

We assume that we are given a candidate $h > 0$ from the previous step (or some initial choice in the first step). A crucial role is played by investigation of the connection between the value of f after the step size and the value given by its linearized prediction (see Figure 1):

$$r_h = \frac{f(x - hv) - (f(x) - h\langle v, \nabla_x f\rangle)}{h\langle v, \nabla_x f\rangle}.$$

We implicitly assume here that $\langle \nabla f_x, v\rangle > 0$ (if this is not the case, we replace v by $-v$).

Our idea relies on considering the loss function in a direction v:

$$\phi : t \to f(x - tv),$$

[1] Clearly, we can use an arbitrary direction provided by some minimization method instead of the gradient in place of v

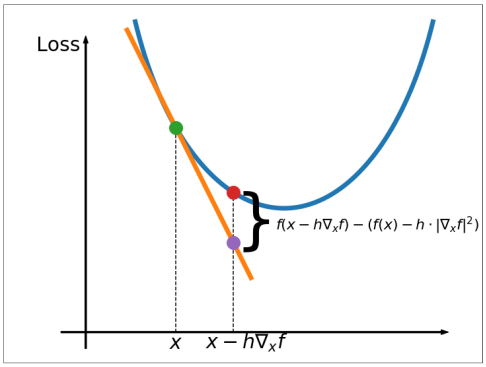

Figure 1. r_h measures the relation between the true value of the loss function f in point $x - h\langle v, \nabla_x f\rangle$ and its linearized prediction given by the gradient.

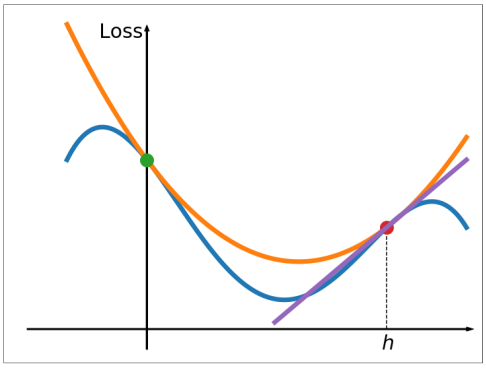

Figure 2. Function ϕ with its corresponding quadratic approximation W. If the derivative of W at point h is positive, we should decrease the step-size to obtain the minima of W.

and fitting a quadratic function $W(t)$, which coincides with ϕ at the end points and has the same derivative at zero (see Figure 2), i.e. such that:

$$\phi(0) = W(0),\ \phi'(0) = W'(0),\ \phi(h) = W(h).$$

Then we get:

$$\phi(t) \approx W(t) = f(x) - t\langle \nabla_x f, v\rangle + r_h \frac{\langle \nabla_x f, v\rangle}{h} t^2. \tag{2}$$

Remark 1. *To compute r_h we need the knowledge of the gradient $\nabla_x f$ and the evaluation of f at $x - hv$ (the predicted next point in which we will arrive according to the current value of the step-size). Consequently, in the case when $v = \nabla_x f$ (i.e. in the case of gradient methods), we need to additionally compute $f(x - h\nabla_x f)$. Then the value r_h simplifies to:*

$$r_h = \frac{f(x - h\nabla_x f) - (f(x) - h \cdot \|\nabla_x f\|^2)}{h \cdot \|\nabla_x f\|^2}.$$

The derivative of function W at point h is given by:

$$W'(h) = -\langle \nabla_x f, v \rangle + r_h \frac{\langle \nabla_x f, v \rangle}{h} 2h = \langle \nabla_x f, v \rangle \cdot (-1 + 2r_h). \tag{3}$$

This means that if:

$$-1 + 2r_h \leq 0 \text{ i.e. } r_h \leq 1/2,$$

the derivative of function W at point h is negative, and therefore we should increase the step-size to further minimize f (see Figure 2). On the other hand, if :

$$-1 + 2r_h > 0 \text{ i.e. } r_h > 1/2,$$

the derivative of function W at point h is positive, and therefore we should decrease the step-size to further minimize f.

In our method, increasing (decreasing) the step-size takes place by multiplication of the current learning rate h by the learning rate adjustment factor c ($\frac{1}{c}$). One can find the LOSSGRAD algorithm pseudocode in algorithm 1. Notice that using our method does not involve almost any additional calculations.

Algorithm 1 LOSSGRAD step

Require: X - inputs for current batch, y - labels for current batch
Require: θ - model weights, α - learning rate, c - learning rate adjustment factor

function LOSSGRAD_STEP(X, y)
  $\hat{y} \leftarrow \text{predict}(X; \theta)$
  $f \leftarrow \text{loss\_function}(\hat{y}, y)$
  $\text{approx} \leftarrow f - h||\nabla_\theta f||^2$
  $\theta \leftarrow \theta - h\nabla_\theta f$
  $\hat{\hat{y}} \leftarrow \text{predict}(X; \theta)$
  $\text{actual} \leftarrow \text{loss\_function}(\hat{\hat{y}}, y)$
  $r_h \leftarrow \frac{\text{actual} - \text{approx}}{h||\nabla_\theta f||^2}$
  if $r_h > 0.5$ **then**
    $\alpha \leftarrow \frac{\alpha}{c}$
  else
    $\alpha \leftarrow c\alpha$

3. LOSSGRAD asymptotic analysis in two dimensions

In the following section we show, how this process behaves for the quadratic form $F(x) = x^T Ax$, where $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ and A is a symmetric positive matrix. Observe that in this case LOSSGRAD can be seen as the approximation of the exact solution to equation (1). Therefore in this section, we study how the minimization process given in (1) works for quadratic functions.

To obtain exact formula we now apply the orthonormal change of coordinates in which F has the simple form:

$$F(x_1, x_2, \ldots) = \lambda_1 x_1^2 + \lambda_2 x_2^2 + \ldots + \lambda_n x_n^2,$$

where $\lambda_1 \geq \lambda_2 \geq \ldots \lambda_n \geq 0$ are the eigenvalues of A.

Starting from the random point $x^0 = (x_1^0, x_2^0, \ldots, x_n^0)$, which gradient is equal to $\nabla^0 = 2(\lambda_1 x_1^0, \lambda_2 x_2^0, \ldots)$, we have:

$$f(t) = F(x^0 - t\nabla^0) = F((1-2t\lambda_1)x_1^0, (1-2t\lambda_2)x_2^0, \ldots, (1-2t\lambda_n)x_n^0) =$$

$$= \lambda_1[(1-2t\lambda_1)x_1^0]^2 + \lambda_2[(1-2t\lambda_2)x_2^0]^2 + \ldots + \lambda_n[(1-2t\lambda_2)x_n^0]^2.$$

After taking the derivative we have:

$$f'(t) = -4\lambda_1^2(1-2t\lambda_1)(x_1^0)^2 - 4\lambda_2^2(1-2t\lambda_2)(x_2^0)^2 - \ldots - 4\lambda_n^2(1-2t\lambda_n)(x_n^0)^2$$

$$= -4[\lambda_1^2(x_1^0)^2 + \lambda_2^2(x_2^0)^2 + \ldots + \lambda_n^2(x_n^0)^2] + 8t[\lambda_1^3(x_1^0)^2 + \lambda_2^3(x_2^0)^2 + \ldots + \lambda_n^3(x_n^0)^2].$$

As our goal is to find $t_0 = \arg\min_t f(t)$, we equate the derivative to zero:

$$t_0 = \frac{1}{2}\frac{\lambda_1^2(x_1^0)^2 + \lambda_2^2(x_2^0)^2 + \ldots + \lambda_n^2(x_n^0)^2}{\lambda_1^3(x_1^0)^2 + \lambda_2^3(x_2^0)^2 + \ldots + \lambda_n^3(x_n^0)^2}.$$

Since $x^1 = x^0 - t_0\nabla^0$, we have:

$$x^1 = \left(\frac{(\lambda_1-\lambda_1)\lambda_1^2(x_1^0)^2 + (\lambda_2-\lambda_1)\lambda_2^2(x_2^0)^2 + \ldots}{\lambda_1^3(x_1^0)^2 + \lambda_2^3(x_2^0)^2 + \ldots} \cdot x_1^0, \ldots\right)$$

To see how this process behaves after a greater number of steps, we assume that $x \in \mathbb{R}^2$. We consider the function which transports the point to its next iteration:

$$g(x) = \frac{\lambda_2 - \lambda_1}{\lambda_1^3(x_1)^2 + \lambda_2^3(x_2)^2} x_1 x_2 (\lambda_2^2 x_2, -\lambda_1^2 x_1).$$

Then

$$g^2(x) = \frac{\lambda_2 - \lambda_1}{\lambda_1^3(x_1)^2 + \lambda_2^3(x_2)^2} x_1 x_2 \cdot g(\lambda_2^2 x_2, -\lambda_1^2 x_1)$$

$$= \frac{(\lambda_2-\lambda_1)}{\lambda_1^3(x_1)^2 + \lambda_2^3(x_2)^2} x_1 x_2 \frac{(\lambda_2-\lambda_1)}{\lambda_1^3(\lambda_2^2 x_2)^2 + \lambda_2^3(\lambda_1^2 x_1)^2} \lambda_1^2\lambda_2^2 x_1 x_2 \lambda_1^2\lambda_2^2(x_1, x_2)$$

$$= \frac{(\lambda_2-\lambda_1)^2\lambda_1\lambda_2}{\lambda_1^3(x_1)^2 + \lambda_2^3(x_2)^2} \frac{x_1^2 x_2^2}{\lambda_1 x_1^2 + \lambda_2 x_2^2}(x_1, x_2) = \frac{(\lambda_2-\lambda_1)^2\lambda_1\lambda_2}{(\lambda_1^2+\lambda_2^2)\lambda_1\lambda_2 + \lambda_1^4 a_x + \lambda_2^4 a_x^{-1}}(x_1, x_2),$$

where $a_x = x_1^2/x_2^2$, and thus $g^{2n}(x) = K_x^n x$. By taking the worst possible case we obtain that the above minimizes for $a_x = \lambda_2^2/\lambda_1^2$, so we obtain an estimate for the convergence:

$$\|g^{2n}(x)\| \leq \left(\tfrac{\lambda_1-\lambda_2}{\lambda_1+\lambda_2}\right)^{2n}\|x\|. \tag{4}$$

Notice that this is invariant to data and function scaling (i.e. $g(Cx) = Cg(x)$).

Remark 2. *One can easily observe that the estimation (4) gives the upper bound for a decrease rate of the solution to any standard gradient descent method with a fixed learning rate. Whether the same holds for the method in $\mathbb{R}^n$ is an open problem.*

4. Experiments

We tested LOSSGRAD in multiple different setups. First, we explore the algorithm's resilience on the choice of initial learning rate and its behavior for a range of different LR adjustment factor values. Then we proceed to test the algorithm's performance on fully connected networks on MNIST classification and Fashion-MNIST [16] autoencoder tasks. Convolutional neural networks are tested on CIFAR-10 [5] classification task using ResNet-56 architecture [2]. We also evaluate our optimizer for an LSTM model [3] trained on Large Movie Review Dataset (IMDb) [7]. Finally, we test LOSSGRAD on Wasserstein Auto-Encoder with Maximum Mean Discrepancy-based penalty [14] on CelebA dataset [6]. Network architectures and all hyperparameters used in experiments are listed in the appendix A.

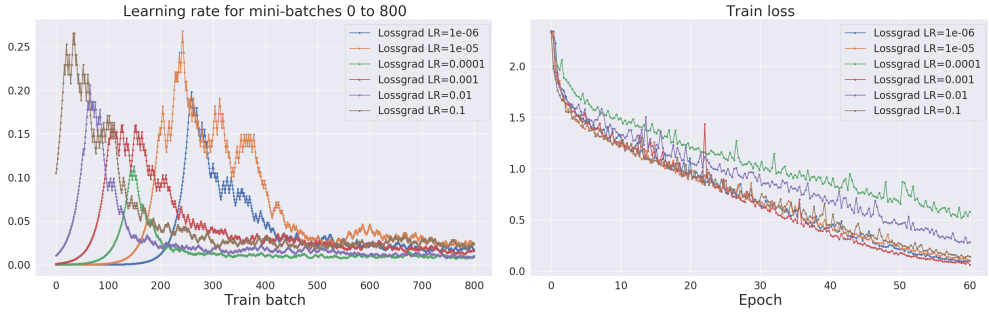

Figure 3. LOSSGRAD with different initial learning rate values trained on CIFAR-10 dataset.

In our experiments we tried to compare LOSSGRAD with WNGRAD [15] and ł4 applied to SGD [11]. We found out that ł4 based on vanilla SGD is extremely unstable both on standard and tuned parameters on almost all datasets and network architectures, so we do not present the results here. For each experiment we also tested standard SGD optimizer with a range of learning rate hyperparameters, including a highly tuned one. For comparison, we also included SGD with scheduled learning rate if that enhanced the results. Because dropout heavily affects LOSSGRAD's stability, we decided not to use it in our experiments.

We test the initial learning rate with values ranging between 10^{-1} and 10^{-6} for a convolutional neural network on CIFAR-10 with the rest of the settings staying the same. Resulting test loss values are presented in Fig. 3 on the right, while the first 800 batches' step size values are presented on the left. Irrespectively of the initial learning rate chosen, the step size always converges to values around 0.025 for this experiment setup. Thus, the need for tuning is practically eliminated, and this property makes the algorithm noticeably attractive.

As LOSSGRAD requires one hyperparameter, we explore which values are appropriate. This is tested by training a convolutional neural network on CIFAR-10 dataset, using our optimizer parameterized with a different value each time, with the rest of the settings staying the same. We evaluated the following hyperparameter values:

	Test loss	Test acc.
Lossgrad c=1.001	2.737	0.655
Lossgrad c=1.005	2.870	**0.673**
Lossgrad c=1.01	1.995	0.658
Lossgrad c=1.05	**1.165**	0.665
Lossgrad c=1.1	1.862	0.661
Lossgrad c=1.2	2.830	0.643

Table 1. LOSSGRAD results for different c hyperparameter trained on CIFAR-10 dataset.

	Test loss	Test acc.
SGD LR=0.01	0.101	0.971
SGD LR=0.07	0.085	0.980
SGD LR=0.45	0.087	**0.985**
WNGrad	**0.073**	0.978
Lossgrad	0.094	0.980

Table 2. LOSSGRAD results for classification on MNIST dataset.

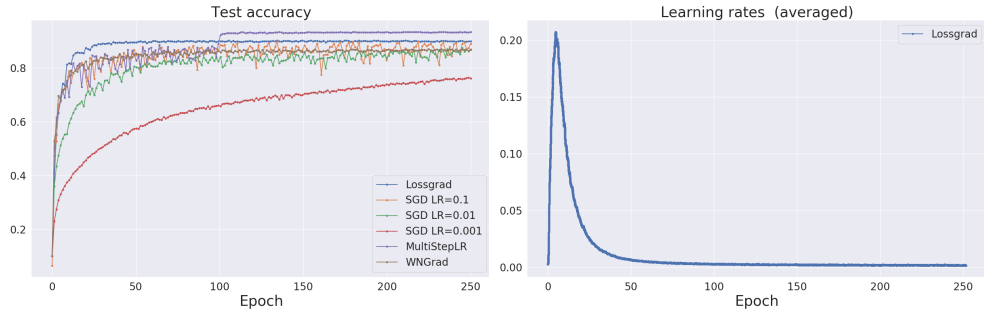

Figure 4. Training ResNet-56 with LOSSGRAD.

$1.001, 1.005, 1.01, 1.05, 1.1, 1.2$. We found that low and high values tend to cause unstable behavior. According to these results, the rest of the experiments in this paper use $c = 1.05$ and initial learning rate set to 1^{-4}.

	Train loss	Test loss
SGD LR=2.0	0.018	0.019
SGD LR=4.0	0.013	0.013
SGD LR=16.0	**0.010**	**0.010**
StepLR	0.015	0.016
Trapezoid	0.012	0.012
WNGrad	0.023	0.023
Lossgrad	0.022	0.022

Table 3. LOSSGRAD results for autoencoder on Fashion-MNIST dataset.

	Test loss	Test accuracy
SGD LR=0.001	0.719	0.762
SGD LR=0.01	0.561	0.868
SGD LR=0.1	0.384	0.890
MultiStepLR	**0.278**	**0.934**
WNGrad	0.678	0.870
Lossgrad	0.492	0.900

Table 4. LOSSGRAD results for ResNet-56 on CIFAR-10 dataset.

Tab. 2 and Tab. 3 presents the results for fully connected network trained on MNIST and an autoencoder trained on Fashion-MNIST, respectively. We noticed the occurrence of sudden spikes in loss (classification accuracy drops and subsequent recoveries) in case of classification on MNIST, but not when training an autoencoder on Fashion-MNIST. The spikes correspond to learning rate peaks, which suggests that temporarily too high step size causes the learning process to diverge.

Fig. 4 presents test accuracy and averaged step size (learning rate) when training a ResNet-56 network on CIFAR-10, while Tab. 4 presents the results summary. Even

with low initial learning rate, LOSSGRAD still manages to achieve results better than SGD, being beaten only by optimized scheduled SGD. Note the step size spike at the beginning of the training process. This spike consistently appears at the beginning of the training in nearly every setup tested in this paper. This result is in line with many learning rate schedulers used in the training of neural networks, that increase the step-size at the beginning of the training and then, after few epochs, they decrease the step-size value [17].

	Test loss	Test acc.
SGD LR=0.05	0.66	0.623
SGD LR=0.1	0.359	0.845
SGD LR=0.5	**0.297**	**0.875**
scheduled	0.299	0.874
WNGrad	0.567	0.726
Lossgrad	0.583	0.708

Table 5. LOSSGRAD result on IMDb dataset.

	Train loss	Test loss
SGD LR=0.0001	11742.473	12859.591
SGD LR=1e-05	12704.917	12881.991
original (Adam)	**8598.712**	**11082.079**
WNGrad	14321.215	14304.257
Lossgrad	25225.673	25196.921

Table 6. LOSSGRAD results for WAE on CelebA dataset.

Results for LSTM trained on IMDb dataset are presented in Tab. 5. Here, for the vanilla SGD, a higher learning rate is preferred. LOSSGRAD selects a very low step-size instead (below 0.01 after the initial peak) and manages only to achieve better results than untuned SGD.

Finally, the results for WAE-MMD are presented in Tab. 6. The originally used optimizer (Adam) and scheduler combination from [14] is marked as „original". Properly tuned SGD, as well as WNGrad, yield better results than LOSSGRAD, which chooses a very low step size for this problem.

We provide an implementation of the algorithm with basic examples of usage on a git repository: `https://github.com/bartwojcik/lossgrad`.

5. Conclusion

We proposed LOSSGRAD, a simple optimization method for approximating locally optimal step-size. We analyzed the algorithm behavior in two dimensions quadratic form example and tested it on a broad range of experiments. Resilience on the choice of initial learning rate and the lack of additional hyperparameters are the most attractive properties of our algorithm.

In future work, we aim to investigate and possibly mitigate the loss spikes encountered in the experiments, as well as work on increasing the algorithm's effectiveness. A version for momentum SGD and Adam is also an interesting topic for exploration that we intend to pursue.

6. References

[1] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[4] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[5] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[6] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3730–3738, 2015.

[7] Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1*, pages 142–150. Association for Computational Linguistics, 2011.

[8] Maren Mahsereci and Philipp Hennig. Probabilistic line searches for stochastic optimization. In *Advances in Neural Information Processing Systems*, pages 181–189, 2015.

[9] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

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

[11] Michal Rolinek and Georg Martius. L4: Practical loss-based stepsize adaptation for deep learning. In *Advances in Neural Information Processing Systems*, pages 6434–6444, 2018.

[12] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[13] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[14] Ilya Tolstikhin, Olivier Bousquet, Sylvain Gelly, and Bernhard Schoelkopf. Wasserstein auto-encoders. *arXiv preprint arXiv:1711.01558*, 2017.

[15] Xiaoxia Wu, Rachel Ward, and Léon Bottou. Wngrad: Learn the learning rate in gradient descent. *arXiv preprint arXiv:1803.02865*, 2018.

[16] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[17] Chen Xing, Devansh Arpit, Christos Tsirigotis, and Yoshua Bengio. A walk with sgd. *arXiv preprint arXiv:1802.08770*, 2018.

[18] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

A. Network architecture, hyperparameters and datasets description

MNIST		Fashion-MNIST		CIFAR		
Type	Outputs	Type	Outputs	Type	Kernel	Outputs
Linear	500	Linear	200	Conv2d	5x5	28x28x30
ReLU		ReLU		MaxPool2d	2x2	14x14x30
Linear	300	Linear	100	ReLU		
ReLU		ReLU		Conv2d	5x5	10x10x40
Linear	100	Linear	50	MaxPool2d	2x2	5x5x40
ReLU		ReLU		ReLU		
Linear	10	Linear	100	Conv2d	3x3	3x3x50
		ReLU		ReLU		
		Linear	200	Linear		250
		ReLU		ReLU		
		Linear	784	Linear		100
		Sigmoid		ReLU		
				Linear		10

Table 7. Architecture summary for experiments presented in Tab. 2 (left), Tab. 3 (middle), Tab. 1 and Fig. 3 (right).

Tab. 7 presents the architecture used for image classification on CIFAR-10 dataset. This architecture was used to obtain the results presented in Fig. 3 and Tab. 1. We initialized the neuron weights using a normal distribution with a 0.05 standard deviation and bias weights with a constant 0.2. The minibatch size was set to 100 and cross entropy was chosen as the loss metric. The model was trained for 60 epochs with c hyperparameter set to 1.05 in the learning rate convergence experiment and for 120 epochs with an initial learning rate of 0.1 in the second experiment. We preprocessed the CelebA dataset by resizing its images to 64x64 and discarding labels.

Tab. 7 also contains the architectures used for image classification on MNIST dataset and for autoencoder training on Fashion-MNIST dataset. We trained both networks for 30 epochs with the rest of the hyperparameters being the same as in CIFAR-10 experiment. For MNIST, we selected the following learning rate values: $0.01, 0.07, 0.45$ for SGD and 1.0 for WNGrad. For Fashion-MNIST, we tested the following learning rate values: $2.0, 4.0, 16.0$ for SGD and 5.0 for WNGrad. We also evaluated two scheduled optimizers. The first one linearly increases the learning rate from 0 to 10 in epoch 10, stays constant until epoch 15 and decreases afterward (trapezoidal). The second one starts with 16.0 and is multiplied by 0.5 after 20 epochs. Mean square error was used as the loss function.

ResNet-56 architecture is described in [2]. We trained the network for 250 epochs, used random cropping and inversion when training and applied weight decay with $\lambda = 0.001$. The learning rate values we used were: $0.1, 0.01, 0.001$ for SGD, 0.2 for WNGrad, 0.1 with 0.1 multiplier after epochs 100 and 150 for step scheduled SGD. We set the mini-batch size to 128 in this experiment.

For IMDb experiments, we used a pretrained embedding layer, trained with GloVe algorithm [9] on 6 billion tokens available from `torchtext` library. Its 100 element output was fed to 1 layer of bidirectional LSTM with 256 hidden units for each direction. The final linear layer transformed that to scalar output. We set 0.1 as the learning rate for WNGRAD , 0.5, 0.1, 0.05 for SGD and also tested a scheduled SGD with learning rate initially set to 0.5 and then decreasing to 0.05 after 10 epochs.

The architecture for WAE-MMD is the same as in [14]. Minibatch size was set to 64, mean square error was selected as a loss function and the network was trained for 80 epochs. We set 1^{-4} as the initial learning rate for WNGrad and $1^{-5}, 1^{-4}$ for SGD.