# Exploring Adaptive Learning Methods for Convex Optimization

Bhavdeep Sethi School of Engineering and Applied Sciences Columbia University Email: bas2226@columbia.edu

*Abstract*—This paper explores various adaptive learning techniques as an alternative to the standard gradient descent approach for convex optimization problems. We establish the merits of adaptive learning rates over fixed learning rates by implementing several techniques such as Momentun, Nestorov's Accelerated Gradient, AdaGrad, and AdaDelta on different datasets including Digits (digit image data for OCR task), 20newsgroup (document collection) and Labeled Faces in the Wild (LFW) (face image data). Comparing the accuracies with the baseline Stochastic gradient descent, we conclude that all adaptive learning techniques perform considerably better than SGD for all the datasets experimented on.

*Keywords*—*Adaptive learning, AdaGrad, AdaDelta, Momentum.*

## I. INTRODUCTION

Most machine learning models employ convex optimization methods for training. For instance, the classic artificial neural network uses Stochastic Gradient Descent (SGD) for training. However, it faces a serious demerit that it is highly sensitive to the value of a fixed learning rate. The learning rate is typically set experimentally, using a tuning procedure in which the highest possible learning rate (such that no divergence is seen, and the convergence is also reasonably fast) is hand picked. This renders determining a good learning rate more of an art than science for many problems. [1]

In addition, we know that the eigenvalues of the Hessian matrix, that captures second order differential information about the objective function, describe the steepness of the curve. Higher the eigenvalues, steeper is the curve and we require smaller values of learning rate. In the scenario where we have a fixed learning rate, the optimum value that learns all parameters reasonably well would be the inverse Hessian. However, computing the inverse of the Hessian matrix is computationally intensive. Some adaptive learning techniques provide a good approximation of the Hessian, thereby allowing Stochastic Gradient Descent to exhibit fast convergence as well as a high success rate.

In the following sections we will discuss the most recent work going on in the field, followed by detailed explanation of how some of these techniques work. Next, we present our experiments and share some key findings/insights <sup>1</sup>. Finally, we conclude about the performance of various adaptive learning

Rhea Goel School of Engineering and Applied Sciences Columbia University Email: rg2936@columbia.edu

techniques and discuss some limitations, omissions and future scope of our work.

#### II. RELATED WORK

There is no dearth of alternatives for vanilla Stochastic Gradient Descent that make for faster learning algorithms for neural networks. Back-propagation is itself a slow learning algorithm, and can do worse for a poor parameter selection. Using non-linear optimization techniques helps accelerate the training method. Since authors usually compare their new algorithms with the standard back-propagation, they always report a substantial improvement. [2]

However, different learning methods may perform differently for different learning tasks. It is always possible to fool the best method with a suitable learning task, or make it perform considerably better than the rest. It is a rather surprising fact that standard online back-propagation performs better than many fast learning algorithms as soon as the learning task achieves a realistic level of complexity and when the size of the training set goes beyond a critical threshold. [3]

Among the many algorithms proposed for faster learning methods, there are some that try to adapt the networking topology. However, we restrict our study to the problem of determining the network parameters, for networks with a fixed topology.

There are many modifications to the gradient descent algorithms, and the most powerful one happens to be the application of the Newton method, which requires the computation of the Hessian matrix. Given the cost of that operation for large models, Becker and LecCun [4] proposed a diagonal approximation to the Hessian. Also, a recent method by Schaul et al. [5] incorporating the diagonal Hessian with AdaGradlike terms has been introduced to alleviate the need for hand specified learning rates.

In addition, some heuristic approaches try to slow down progress near local minima in the cost surface. This can be done by manually decreasing the learning rate when the validation accuracy plateaus, or automatically anneal the learning rate based on how many epochs through the data have been done. [6]

To lay out the complete landscape, there are a few very recent methods coming up in the field that we did not explore in our experiments, but are likely to perform better than most

<sup>1</sup>Source Code: https://github.com/BhavdeepSethi/AMLProj.git



Fig. 1: Long Narrow Valley function (a). Stochastic Gradient Descent (b). Momentum

methods in several cases. One of these techniques is AdaSecant [7], which utilizes curvature information estimated from the local statistics of the stochastic first order gradients. They also propose a new variance reduction technique which automatically reduces the variance of noise in the local gradient estimates to speed up the convergence. Another is RMSProp, in which the authors try to exploit the presence of negative eigenvalues of the Hessian to help us design better suited adaptive learning rate schemes, i.e., diagonal preconditioners.[8] They show that the optimal preconditioner is based on taking the absolute value of the Hessian's eigenvalues.

Apart from modifying the global learning rate (which is same across all dimensions of the parameters) there has been extensive research into per-dimension learning rate as well. We will explore some of these methods in our study.

## III. STOCHASTIC GRADIENT DESCENT

Stochastic Gradient Descent (SGD) is the most commonly used method for training a wide array of machine learning models such as regression models, Support Vector Machines, graphical models etc. It is an optimization technique to minimize an objective function written as a sum of differentiable functions. Used in conjunction with the Back propagation algorithm, SGD is used for training artificial neural networks.

Gradient descent algorithm can be implemented in the batch mode, online mode or using mini-batches. In the batch mode, each update is made by running through all the data samples whereas in the online approach, an update is made for every sample that is misclassified by the current model. In our experiments, we use a compromise between the batch and online mode, employing mini-batches (batches containing more than one sample, but not the entire sample set).

The main idea behind SGD is to update a set of parameters in an iterative manner to minimize an error function. In each update, the parameter is reduced by the gradient of the loss function, scaled by  $\eta$ , the learning rate. The update rule is given by:

$$
\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t)
$$

The learning rate indicates the step size in the search process. If the learning rate is set to a very low value, the algorithm makes only very little progress in each iteration and



Fig. 2: (Top) Classical Momentum (Bottom) Nesterov's Accelerated Gradient

The most unsung birthday in American business and technological history this year may be the 50th anniversary of the Xerox 914 photocopier.<sup>a</sup>

## <sup>a</sup> The Atlantic, July/August 2010.

Fig. 3: Images from Duchi et al. ISMP 2012 slides

hence converges slowly. If it is set too high, each iteration might overshoot the optimum value and we observe divergence. This high sensitivity to the value of the learning rate is one limitation of Stochastic Gradient descent. Next, we will see simple modifications to SGD, using adaptive learning rates.

## IV. ADAPTIVE LEARNING TECHNIQUES

#### *A. Momentum*

The simplest extension to SGD that has been used successfully for many years now is momentum [9]. In SGD, the gradient  $\nabla L(\theta_t)$  may change very rapidly due to the fact that we're computing the gradient at different data samples. Hence, following the gradient direction at every step (as in SGD) can possibly lead to several oscillations in the search process. Momentum partially solves this issue by re-utilizing the previous gradient value, scaled by a momentum hyperparameter. The update rule is given by:

$$
\nu_{t+1} = \mu \nu_t + \eta \nabla L(\theta_t)
$$

$$
\theta_{t+1} = \theta_t + \nu_{t+1}
$$

where  $\mu \in [0, 1]$  is the momentum parameter,  $L(\theta_t)$  is the objective function to minimize and  $\nabla L(\theta_t)$  is the gradient of the function at  $\theta_t$ 

Instead of following the direction indicated by the current gradient, a weighted average of the previous gradient direction and the current gradient direction is computed. The main idea behind momentum is to accelerate progress along dimensions in which gradient consistently points in the same direction and to slow progress along dimensions where the sign of the gradient continues to change. This is done by keeping track of past parameter updates with an exponential decay.

Consider a cost surface like the long narrow valley function. A fixed learning rate might cause the algorithm to oscillate several times before reaching the optimum minimum value. Momentum essentially tamps down these oscillations by correcting the direction of movement chosen at every step (by utilizing historical information) and reaches the bottom of the valley sooner. This is visualized reasonably well in Figure 1.[10]

The desired procedure in this case would be to orient the search towards the center of the valley, but the form of the error function is such that the gradient does not point in this direction. Theoretically, the introduction of the momentum term should provide the search process with a kind of inertia and could help avoid excessive oscillations in narrow valleys of the error function, thereby accelerating convergence to a minimum of the error function.

#### *B. Nestorov's Accelerated Gradient*

A slight modification to Momentum results in another approach called Nestorov's Accelerated Gradient (NAG). It has been the subject of much recent attention by the convex optimization community. Though seen as different from momentum, the update rules for both are very similar to each other. The update rule of NAG is given by:

$$
\nu_{t+1} = \mu \nu_t + \eta \nabla L(\theta_t + \mu \nu_t)
$$

$$
\theta_{t+1} = \theta_t + \nu_{t+1}
$$

The update rule of NAG differs with that of momentum only in the precise update of the velocity vector  $\nu$ . While momentum computes the gradient update from the current position  $\theta_t$ , NAG first performs a partial update to  $\theta_t$ , computing  $\theta_t + \mu \nu_t$ , which is similar to  $\theta_{t+1}$ , but missing the as yet unknown correction. This benign-looking difference seems to allow NAG to change  $\nu$  in a quicker and more responsive way, letting it behave more stably than classical momentum in many situations, especially for higher values of  $\mu$ 

Consider the scenario in Figure 2.[11], where the an update based on the previous gradient leads to an undesirable increase in the value of the objective function. In NAG, the gradient correction to the velocity vector at the new position  $\theta_t + \mu \nu_t$ is computed, and if it indeed was a poor update, the gradient at the new position will point back towards  $\theta_t$  much more strongly. So, NAG essentially performs a simple gradient descent and 'jitters' away slightly in the direction of the previous gradient for timely correction of updates.



Fig. 4: Cost Function Value for 50 epochs, Momentum

TABLE I: Error vs  $\mu$  (Momentum)

$\mu$	Train Error	Validation Error	<b>Test Error</b>
0 <sub>0</sub>	0.043692	0.075329	0.082342
0.9	0.000000	0.050329	0.038385
0.99	0.000000	0.034079	0.023958
0.999	0.000000	0.036250	0.043419

## *C. AdaGrad*

AdaGrad was introduced by John Duchi, Eden Hazan and Yoram Singer in [12]. The primary motivation behind AdaGrad is whether all features should share the same learning rate.

Consider the Fig. 3. In standard Text Classification tasks, infrequent words like "Xerox" are infrequent yet very informative and discriminative. The informativeness of these rare features is the reason functions like TF-IDF are widely used. In most real life applications, the input instances are of very high dimensions, but many features are irrelevant.[13]

AdaGrad incorporates knowledge of the geometry of the past observations to provide a feature-specific adaptive learning rate. AdaGrad alters Stochastic Gradient Descent update so that frequently occurring features in the gradients get small learning rates and infrequent features get higher ones. As Duchi et al. put it, the learner learns slowly from frequent features but pays attention to rare but informative features.[14] Additionally, AdaGrad is an online learning algorithm with asymptotically sublinear regret.

The update rule of AdaGrad is given by:

$$
g_{t+1} = g_t + \bigtriangledown L(\theta_t)^2
$$

$$
\theta_{t+1} = \theta_t - \frac{\eta \bigtriangledown L(\theta_t)}{\sqrt{g_{t+1} + \epsilon}}
$$

From the update rule, we notice that, each feature dimension has it's own learning rate and it

- Adapts with t
- Considers geometry of past observation
- $\eta$  determines rate first time the feature is encountered.

While there is a global learning rate, each dimension has its own learning rate. Since this dynamic rate is inversely proportional to the gradient magnitudes, large gradients have smaller learning rates and small gradients have large learning



Fig. 5: Train, Validation, Test Error for different techniques for Digit Classification Task

TABLE II: Error vs  $\mu$  (NAG)

$\mu$	Train Error	Validation Error	<b>Test Error</b>		
0.0	0.043692	0.075329	0.082342		
0.9	0.000000	0.050329	0.038940		
0.99	0.000000	0.036579	0.022239		
0.999	0.017915	0.050000	0.047864		

TABLE III: Error vs  $\epsilon$  (AdaGrad)



rates. Thus, the progress along each dimension evens out over time. This is really useful for training deep neural networks since scale of gradient in each layer is different by several orders of magnitude. Also, the accumulation of gradient in the denominator has the same effect as annealing, i.e. having the learning rate decay over time.

## *D. AdaDelta*

AdaDelta was proposed by Matthew Zeiler in [1] in order to improve upon two main drawbacks of AdaGrad:

Since each term in the update is positive, the accumulation of squared gradient in the denominator continues to decrease the learning rate on each dimension throughout training, eventually becoming infinitesimally small and thus stops training completely after running for a long time.

The magnitudes of gradients are factored out in Ada-Grad. Thus, it can be sensitive to initial conditions of the parameters and the corresponding gradients. The learning rates will be low if the initial gradients are large. This can be avoided by increasing the global learning rate. This makes AdaGrad method sensitive to the choice of global learning rate.

AdaDelta dynamically adapts over time using only first order information and has minimal computational overhead beyond vanilla stochastic gradient descent.

The AdaDelta update rule is given by:

$$
g_{t+1} = \gamma g_t + (1 - \gamma) \nabla L(\theta_t)^2
$$

$$
\nu_{t+1} = -\frac{\sqrt{x_t + \epsilon}}{\sqrt{g_{t+1} + \epsilon}} \nabla L(\theta_t)
$$

$$
x_{t+1} = \gamma x_t + (1 - \gamma)\nu_{t+1}^2
$$

$$
\theta_{t+1} = \theta_t + \nu_{t+1}
$$

We notice that,

• Just like Stochastic Gradient Descent, the negative gradient direction for the current iteration  $g_t$  is followed.

- Just like Momentum, the numerator acts like an acceleration term, accumulating previous gradients over a window of time.
- Just like AdaGrad, the denominator helps to even out the progress made in each dimension.
- Finally, AdaDelta relates to Schaul et al.s in that some approximation to the Hessian is made, but by leveraging information from past updates, it costs only one gradient computation per iteration.

In AdaDelta, we use a window w (instead of size t, where t is the current iteration as in AdaGrad) to ensure that the denominator cannot accumulate to infinity and instead becomes a local estimate using recent gradients. This ensures learning continue even after many iterations. To avoid the inefficiency of storing w previous squared gradients, AdaDelta implements the accumulation as an exponentially decaying average of the squared gradients.

Thus, some of the benefits of AdaDelta are:

- Manual setting of a global learning rate is not required
- It is Insensitive to hyper-parameters
- We get dynamic learning rate for each dimension
- It requires minimal computation over standard gradient descent
- It is quite robust to gradient magnitude, noise and choice of architecture
- It is applicable in both local and distributed environments

Thus, AdaDelta is a robust learning method that can be applied to variety of domains.

## V. EXPERIMENTAL SETUP

## *A. Data Sets*

We evaluate the various learning methods on the following data sets:

- The first data set is sample of 1797 8x8 images of digits. There are about 180 samples for each class (0-9). The dataset is nudged to produces a dataset 5 times bigger than the original one, by moving the 8x8 images around by 1px to left, right, down, up.
- The second set is Labeled Faces in the Wild (LFW) people dataset. The dataset is collection of JPEG pictures of famous people collected on the internet. Each image has name of the person as label in the training data. We perform a Face Recognition task here. Were filtering the data so that each class has at least 50 instances. We run our experiments on 1560 samples, with 12 classes.
- The third set is 20newsGroup data which is a standard dataset for document classification. It is the biggest dataset that we're running our experiments on. The training data consists of 11,314 documents with total of 20 distinct classes. We use TF-IDF representation

TABLE IV: Test error for  $\epsilon$  and  $\gamma$  (AdaDelta)

	$\gamma=0.9$	$\gamma=0.99$	$\gamma = 0.999$
$\epsilon = 1e^{-2}$	0.028368	0.026718	0.028368
$\epsilon = 1e^{-4}$	0.030034	0.031701	0.026128
$\epsilon = 1e^{-6}$	0.026701	0.027257	0.026701
$\epsilon = 1e^{-8}$	0.037274	0.027257	0.025590

TABLE V: Test error after 50 epochs (Digit)



of the text as features and limit the number to 10000 features for our experiments. The test data consists of 7,532 documents

The validation set for all the above sets is always 10% of the training data. All the above data sets are available from the scikit-learn python module.

## *B. Architecture & Framework*

We're running the experiment using Deep Neural Nets with one hidden layer. The hidden layer consists of 200 neurons and uses Rectified Linear Unit as the activation function. The output layer uses Logistic Regression with Negative Log Likelihood as the loss function. The prediction is made using arg max over the Softmax function. L1 regularizer co-efficient used is 0.0, while the L2 regularizer r co-efficient is inverse of the size of the training data. We've written the code in Theano so that it can be run on GPU as well.

## *C. Machine*

We're running the experiments on Amazon EC2 instances. The configuration of the machine is: g2.2xlarge (26 ECUs, 8 vCPUs, 2.6 GHz, Intel Xeon E5-2670, 15 GiB memory, 1 x 60 GiB Storage Capacity)

TABLE VI: Test error after 50 epochs (LFW dataset)

	Train Error	Validation Error	<b>Test Error</b>
SGD	0.017500	0.270000	0.259167
Momentum	0.000000	0.240000	0.231667
<b>NAG</b>	0.000000	0.265000	0.229167
AdaGrad	0.000000	0.225000	0.179167
AdaDelta	0.000000	0.220000	0.184167

TABLE VII: Test error after 50 epochs (20 Newsgroup)





Fig. 6: Train, Validation, Test Error for different techniques for Labeled Faces Dataset

## VI. RESULTS

This section will summarize the key findings of all methods that we implement on different datasets. We present the results of SGD, Momentum, NAG, AdaGrad and AdaDelta for each of the three datasets described above. The typical steps followed for our experiments with each dataset are as follows: First, we empirically determine the optimal value of any hyperparameters used by a technique. We then use the best hyperparameter setting of each method to compare their performance on the dataset in question.

## *A. Digit Classification*

Finding best value of  $\mu$  for Momentum

We perform the classification task using Momentum alone, for 50 epochs,  $\eta = 0.01$  and vary the value of  $\mu$  within the range 0.900 to 0.999. (See Table I) We see that  $\mu = 0.99$  achieves the lowest test error for  $\eta = 0.01$ . However, we see in Figure 4 that for  $\mu = 0.99$ , the value of the objective function varies wildly in the initial epochs, whereas  $\mu = 0.9$  performs in a much more stable manner. Hence, we use the latter parameter setting to compare momentum with other techniques (even though the former setting gives slightly better test error).

Finding best value of  $\mu$  for NAG

We identify the best hyper-parameter value for NAG as for momentum, by varying  $\mu$  from 0.9 to 0.999 and learning over 50 epochs with a step size of 0.01. (See Table II)

We see that  $\mu = 0.99$  achieves the lowest test error for NAG (similar to Momentum, as expected)  $\eta = 0.01$ . As in the case of momentum, severe oscillations in the search process were observed for  $\mu = 0.99$ , and hence,  $T\mu = 0.9$  will be used in further eperiments.

Finding best  $\epsilon$  for AdaGrad

Fixing the learning rate  $\eta = 0.01$ , we vary  $\epsilon$  from 0.1 to 0.001 and obtain the results as in Table III. We find that  $\epsilon = 1e^{-8}$  achieves the lowest test error and hence, will be used for experiments hereon.

Finding best combination of  $\epsilon$  and  $\gamma$  for AdaDelta

We employ grid search to find the least error-inducing combination of the hyper-parameters  $\epsilon$  and  $\gamma$  using AdaDelta. Test error results obtained are presented in Table IV.

The best combination of hyper-parameters that gives the least error is  $\epsilon = 1e^{-8}$  and  $\gamma = 0.999$ .

We now implement all five techniques for the digit classification task for 50 epochs each, at  $\eta = 0.01$  and the best hyper-parameter settings as inferred above. Training, validation and test error have been plotted in Figure 5 and the results have been summarized in Table V.



Fig. 7: Train, Validation, Test Error for different techniques for 20 Newsgroup data

It is clear from the plot that SGD performs considerably poor as compared to all the adaptive learning techniques, followed by Momentum and NAG, where they behave similarly except that NAG tends to be more stable. Next, we see that AdaGrad attains an even lower value of the objective function, thereby giving lower test error rate. However, AdaDelta outperforms AdaGrad as well and hence proves to be the best adaptive learning technique (for the selected parameter settings) for the digit classification task performed on the chosen dataset.

#### *B. Labeled Faces in the Wild*

As for digit classification, we empirically determine the best parameter settings for each of the techniques and used those values for performing further experiments. We obtained similar results as for the digit classification task; we omit the details for brevity. As per our experiments, we will use  $\mu =$ 0.9 for momentum and NAG,  $\epsilon = 1e^{-8}$  for AdaGrad and AdaDelta, and  $\gamma = 0.999$  for AdaDelta.

We now implement all five techniques for the labeled faces classification task for 50 epochs each, at  $\eta = 0.01$  and the best hyper-parameter settings as inferred above. Training, validation and test error have been plotted in Figure 6 and the results have been summarized in Table VI.

#### *C. Document Classification*

As for digit and labelled faces dataset, we empirically determine the best parameter settings for each of the techniques and used those values for performing further experiments. Again, we omit the details for brevity. As per our experiments, we will use  $\mu = 0.9$  for momentum and NAG,  $\epsilon = 1e^{-8}$  for AdaGrad and AdaDelta, and  $\gamma = 0.999$  for AdaDelta.

We now implement all five techniques for the document classification task for 50 epochs each, at  $\eta = 0.01$  and the best hyper-parameter settings as inferred above. Training, validation and test error have been plotted in Figure 7 and the results have been summarized in Table VII.

## VII. CONCLUSION

In this study we explore some adaptive learning techniques namely Momentum, Nestorov's Accelerated Gradient, AdaGrad and AdaDelta and appreciate their possible advantages over the vanilla Stochastic Gradient Descent in various scenarios. We compare their performance on three different datasets, and conclude that Momentum and NAG perform considerably better than SGD. Furthermore, AdaDelta proves better than AdaGrad for Digit classification task whereas AdaGrad outperforms it for LFW and 20 Newsgroup data. Nevertheless, the two techniques exhibit close behaviour and each may prove very beneficial over all the other competitors.

#### VIII. ACKNOWLEDGEMENTS

The authors would like to thank Prof. Tony Jebara for his valuable advice and continuous guidance during our study. We would also like to express our gratitude towards the TAs Enze Li, Kui Tang, Sami Mourad for helping us all through the project.

## IX. DISCUSSION AND SCOPE

The experiments we ran were limited to the adaptive learning methods Momentum, NAG, AdaGrad and AdaDelta. There has been a lot of research work in this field. Some of the new methods to be explored are AdaSecant (Dec, 2014) , RMSProp & ESGD (Feb, 2015).

We have restricted the experiments to a fixed network topology. One could also experiment with different architectures to see if it affects the performance of these learning techniques.

Additionally, the experiments can be performed on different domains like speech recognition, time series analysis and so on.

#### **REFERENCES**

- [1] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.
- [2] Lutz Prechelt. A quantitative study of experimental evaluations of neural network learning algorithms: Current research practice. *Neural Networks*, 9(3):457–462, 1996.
- [3] W. Schiffmann, M. Joost, and R. Werner. Comparison of optimized backpropagation algorithms. In *Proc. of ESANN'93, Brussels*, pages 97–104, 1993.
- [4] Sue Becker and Yann Le Cun. Improving the convergence of backpropagation learning with second order methods. In *Proceedings of the 1988 connectionist models summer school*, pages 29–37. San Matteo, CA: Morgan Kaufmann, 1988.
- [5] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. *arXiv preprint arXiv:1206.1106*, 2012.
- [6] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.
- [7] Caglar Gülçehre and Yoshua Bengio. ADASECANT: robust adaptive secant method for stochastic gradient. *CoRR*, abs/1412.7419, 2014.
- [8] Yann N. Dauphin, Harm de Vries, Junyoung Chung, and Yoshua Bengio. Rmsprop and equilibrated adaptive learning rates for nonconvex optimization. *CoRR*, abs/1502.04390, 2015.
- [9] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5, 1988.
- [10] Raúl Rojas. Neural Networks: A Systematic Introduction. Springer-Verlag New York, Inc., New York, NY, USA, 1996.
- [11] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147, 2013.
- [12] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.
- [13] Emily Fox. Logistic SGD and AdaGrad, 2013. Class lecture for CSE 547: Machine Learning for Big Data, University of Washington.
- [14] Chris Dyer. Notes on adagrad. Class lecture notes, Carnegie Mellon University.