# Optimizing CNNs

Timothy Dozat
Stanford
tdozat@stanford.edu

## Abstract

*This work aims to explore the performance of a popular class of related optimization algorithms in the context of convolutional neural networks. Our goals are to test logically possible but currently unexamined variants of proposed algorithms as well as provide a more thorough comparison of those that have already been written about. In addition to examining the overall performance of the algorithms, we scrutinize how sensitive different algorithms are to their hyperparameters. Finally, we aim to see how algorithm performance on toy datasets carries over to more practical tasks with state-of-the art model structures.*

## 1. Introduction

First-order optimization algorithms are often conceptualized as being distinct and largely unrelated. This conceptualization is useful in that it restricts how much time the researcher must put into finding the right optimizer for their model–that is, they might train a few models with SGD, a few with RMSProp, and a few with NAG, then pick the best from among there–but troubling in that insights from newly proposed optimization algorithms rarely get incorporated into older variants–for example, Kingma & Ba [5] proposed an algorithm that corrects the zero-initialization bias inherent in any algorithm that uses a decaying mean, but few implementations of older algorithms that likewise use decaying means have been updated to reflect this insight. When the relationships between them are examined more closely, it becomes easier to see how insights from one algorithm can be incorporated into other algorithms, but the other algorithms now have twice as many variants as they had before. Obviously, one generally can't claim that one algorithm variant is categorically "better" for non-convex objects, but can we find evidence that favors some algorithm variants over others?

Additionally, as the complexity of the algorithm grows, the number of hyperparameters that need to be hand-tuned grows as well. In addition to the learning rate, most popular algorithms more complex than SGD include at least one other hyperparameter, often two or three more. To make matters worse, most hyperparameters can can be warmed or cooled (and sometimes the algorithm's theoretical bounds depend on it), meaning even more hyperparameters. If one wants to find the optimal optimizer for their model, that amounts to an unmanageably large number of hyperparameters to sift through–so do some hyperparameters make more of a difference than others?

This work aims to examine the performance of different algorithm variants on a relatively simple toy task with a nonetheless difficult objective. Of course, performance on one task with one type of network doesn't guarantee similar performance on all tasks with any network architecture, especially if the original task has few practical applications and the original network leaves out recent insights–so in order to make sure our findings are relevant to a wider class of problems and network structures, we take insights from the first round of experiments and see how they hold up in a slightly more real task with a more recently innovated type of network.

## 2. Background

This section reviews the relationships between a number of popular first-order optimization algorithms.

### 2.1. Momentum

Classical momentum [8] maintains a decaying sum of what the SGD update would be at each iteration and updates the parameters with that moving average paramaterized by $\mu$. This serves to speed up convergeance across dimensions with a small but consistent gradient and slow it down across dimensions with large but inconsistent ones [9]. This can be equivalently represented as a decaying mean by multiplying the learning rate by $(1-\mu)$. Kingma & Ba [5] propose an algorithm with a slightly different kind of momentum–rather than accumulating the gradient scaled by the learning rate, they accumulate only the gradient (we'll call these *scaled gradient* and *unscaled gradient* momentum, respectively). In a stochastic setting, this has the intuitive interpretation as an approximation of the true gradient, rather than the gradient of just the current objective (i.e. it approximates the full batch gradient rather than just the gradient of the current minibatch). When the learning rate is constant, these two variants are equivalent, but the first one uses proportionally less of the current gradient at each update when the learning rate is annealed.

Sutskever et al. [9] advocate for using a schedule that starts with a low value of $\mu$ and then increases it to a very large value of $\mu$ over time, as this allows the model to quickly forget the early gradients that just get it in the right ballpark and remember more of the later gradients needed to getting over saddlepoints near the minimum. Kingma & Ba [5] point out that when using a decaying mean, initializing the accumulation vector to 0 introduces error into the accumulated mean. They propose correcting this error with what what is equivalent to using a schedule of $\mu$ such that $\mu_t = \mu(1 - \mu^{t-1})/(1 - \mu^t)$[1]. With this schedule, the first update will use all of the first gradient, the second will use about half of the second gradient, the third will use about a third of the third gradient, etc.

Lastly, momentum can be thought of as involving two parts to every update–a momentum part, which applies the momentum vector of the previous timestep, and a gradient part, which applies an SGD update. When thought of like this, it seems inefficient not to apply the momentum vector of the current timestep–an algorithm known as Nesterov's Accelerated Gradient (NAG) [7] has been shown to rectify this inefficiency [9].

### 2.2. Adaptive learning rates

Duchi et al. [2] motivate an algorithm called AdaGrad that scales the learning rate for each parameter by the inverse $L_2$ norm of all previous gradients for that that parameter. This slows down learning of frequently activated parameters and speeds up learning of infrequently activated parameters (at first). However, because it accumulates this value as a true sum instead of decaying sum or average, in practice the learning rates become too small for the model to achieve fine-grained convergeance. Tieleman & Hinton [10] address this issue by replacing the sum with a decaying mean and dub this variant R(oot)M(ean)S(quare)Prop.

Dauphin et al. [1] propose their own variant of RMSProp that uses second-order information (Equilibrated Gradient Descent). Instead of accumulating an $L_2$ norm of the previous *gradients*, it accumulates an $L_2$ norm of an approximation of the previous *Hessian diagonal*. Computing the full Hessian is obviously intractable for any medium-sized model, but the product of the Hessian with a gaussian random vector can be computed in a reasonable amount of time and approximates the diagonal of the true Hessian when repeated with enough random vectors. Their algorithm as described in the paper uses a true mean rather than a decaying mean, meaning it lacks an additional hyperparameter that RMSProp has[2].

### 2.3. Combination

Some previous researchers [10] found that attempts to combine RMSProp with momentum generally underperformed, but they attempted to combine it with scaled-gradient momentum–however, Kingma & Ba [5] were able to outperform a number of other algorithms by combining unscaled-gradient momentum with RMSProp and then correcting for the zero-initialization bias of the decaying means. Similarly, Dauphin et al. [1] point out that their algorithm can be combined with momentum as well for better results than they present in their paper (which only examines the effects of the adaptiver learning rate component).

## 3. Logical Possibilities

There are four distinct logically possible variants of momentum based on this discussion, differing in whether they use the scaled or unscaled gradients and whether the momentum part of their update uses the previous moment vector (classical momentum) or the current one (NAG). So far the unscaled gradient version of NAG has not been explored in the literature.

Of AdaGrad, there are twelve logical possiblities: variants can use a sum or a mean, an $L_1$, $L_2$, or $L_\infty$ norm, and first- or second- order information. Only those using an $L_2$ norm have been examined previously.

There are also numerous ways to select the moving average hyperparameters (e.g. $\mu$ for momentum). One possibility is to use a constant value that doesn't depend on time $t$; another is to use the initialization-bias correction schedule; one could use a true mean instead of a moving average, equivalent to the limit as $\mu$ approaches 1 of the initialization-bias correction schedule; or one could also hand-craft a schedule that increases slowly over time. Likewise, the global learning rate can (and often should) be decayed according to a schedule, which means yet another hyperparameter that must be set.

In principle, any of the momentum methods can be combined with the adaptive learning rate methods, using any of the four schedules for each moving average, resulting in approximately 750 possible algorithm variants of the "hyper-algorithm" shown below. Time and computing restrictions prevent a complete exploration of this algorithm space, so the ensuing discussion will only examine parts of it and attempt to infer the nature of the unexplored parts.

[1] However, any schedule that gives $\mu_1 = 0$ will avoid zero-initialization bias.

[2] Although one could consider the choice of using a decaying mean versus using a true mean a hyperparameter as well.

**Algorithm 1** The "hyperalgorithm" of which all the aforementioned algorithms are variants. True sums/means can be represented as the limit as a decay parameter approaches 1.

```
Require: $\theta_0$
Require: $\alpha_i$    ▷ The (annealed) learning rate at timetsep $i$
Require: $\mu_i \sim [0, 1)$    ▷ The momentum parameter
Require: $\nu_i \sim [0, 1)$    ▷ The variance parameter
Require: $p \sim [1, \infty)$    ▷ The norm number
Require: $\epsilon \approx 1e^{-8}$
Require: $f_i(\theta)$
  $\mathbf{m}_0, \mathbf{v}_0 \leftarrow 0$
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $\mathbf{h}_t \leftarrow \{\nabla_\theta f_t(\theta_{t-1})$ or $\mathcal{R}_{\mathbf{u}\sim\mathcal{N}(0,1)}(\nabla_\theta f_t(\theta_{t-1}))\}$
    $\mathbf{v}_t \leftarrow \nu_t \mathbf{v}_{t-1} + \{(1 - \nu_t)$ or $1\}\mathbf{h}_t^p$
    $\bar{\mathbf{v}}_t \leftarrow \sqrt[p]{\mathbf{v}_t} + \epsilon$
    $\mathbf{g}_t \leftarrow \{1$ or $\frac{\alpha_t}{\bar{\mathbf{v}}_t}\}\nabla_\theta f_t(\theta_{t-1})$
    $\mathbf{m}_t \leftarrow \mu_t \mathbf{m}_{t-1} + \{(1 - \mu_t)$ or $1\}\mathbf{g}_t$
    $\bar{\mathbf{m}}_t \leftarrow \{\mu_{t+1}\mathbf{m}_t + \{(1 - \mu_t)$ or $1\}\mathbf{g}_t$ or $\mathbf{m}_t\}$
    $\Delta_t \leftarrow \{\frac{\alpha_t}{\bar{\mathbf{v}}_t}$ or $1\}\bar{\mathbf{m}}_t$
    $\theta_t = \theta_{t-1} - \Delta_t$
  end while
  return $\theta_T$
```

## 4. Experimental Setup

### 4.1. Convolutional autoencoder

We use a convolutional autoencoder[3] trained on the MNIST dataset of $28 \times 28$ grayscale handwritten digits [6]. Each image was scaled so that each pixel was between 0 and 1 (therefore the dataset was not mean centered–every input value was positive). The autoencoder used two $7\times7\times1\times16$ convolutional filters, followed by a $2 \times 2$ max pooling filter, followed by a $7 \times 7 \times 1 \times 32$ convolutional filter, another $2 \times 2$ max pooling filter, an $800 \times 128$ FC layer, and finally a $128\times16$ FC layer on the encoder side, then the conversely on the decoder side, for a total of 224177 learnable parameters. All models were trained for 50 epochs. Because we're primarily concerned with optimization, in this task we only report training error; however, the patterns reported generally hold for validation error as well.

Using vanilla ReLUs seemed to categorically hurt performance, as large learning rates consistently "killed" every single node in the network–using smaller learning rates allowed the model to retain a larger share of its initial capacity but significantly slowed down learning. The solution to this was to replace the vanilla ReLUs with "leaky" ReLUs–doing so consistently resulted in better performance, as no node was ever permanently deactivated and the models could reach better solutions within 50 epochs. Also of note is that decaying the learning rate by $.94$ every two epochs similarly very frequently improved performance, irrespective of the other hyperparameters.

[3]The code is adapted from [4], which uses the Lasagne extension of Theano.

### 4.2. Momentum

In this section we examine the effect of classical momentum (CM) versus NAG, the momentum comstant, the schedule, and to a lesser extent the choice of accumulating scaled or unscaled gradients. The results are shown in Table 4.2. The optimal learning rate for both CM and NAG, selected from $\{.5, 1, 2\}$, was found to be 2 in all of the best models, whereas vanilla SGD diverged with rates higher than 1. The models' relative robustness to larger learning rates supports the intuition that momentum helps by slowing down learning along turbulent dimensions, preventing diverging at the beginning of training.

We compared using no schedule, using the initialization-bias correction schedule, and using a handcrafted schedule adapted from Sutskever et al [9] (that also avoided initialization bias). The handcrafted schedule performed extremely poorly compared to the constant schedule and the bias-correction schedule–this is likely because $\mu$ started out far too small at the beginning of training, and consequently the model couldn't take advantage of the properties of momentum that prevent divergeance when using a large learning rate. Using an initialization bias correction schedule resulted in underperforming models–this may be because the models gave too much weight to large, noisy gradients early on in training

The choice of storing the scaled gradients (SG) or the unscaled gradients (UG) in $\mathbf{m}_t$ was generally of little consequence. However, the choice of using CM or NAG did make a difference, but not in the expected direction–one would expect NAG to categorically outperform CM, but here NAG generally performed noticeably worse when using the larger learning rates that led to the best performance. Why might this be, given that NAG has been generally seen as an improved form of momentum? It seems likely that the large learning rate and leaky ReLUs (possibly weight initialization as well) may have played a part. The update rule for scaled-gradient NAG, written in terms of $\mathbf{g}_t$ and $\mathbf{m}_{t-1}$ and using a constant momentum schedule, can be simplified to the following:[4]

$$\theta_t \leftarrow \theta_{t-1} - (\mu^2\mathbf{m}_{t-1} + (1 - \mu^2)\mathbf{g}_t)$$

When rewritten like this, it becomes clear that this variant of NAG is reweighting the update to give more weight to the current gradient than the momentum vector, which it depends on for offsetting large values of $\mathbf{g}_t$ that might result in significant overshooting. Thus when large learning

[4]Note that while $\mu^2$ gets used in the update of $\theta$, only $\mu$ gets used in the update of $\mathbf{m}_t$, so NAG is not equivalent to simply using a smaller momentum parameter.

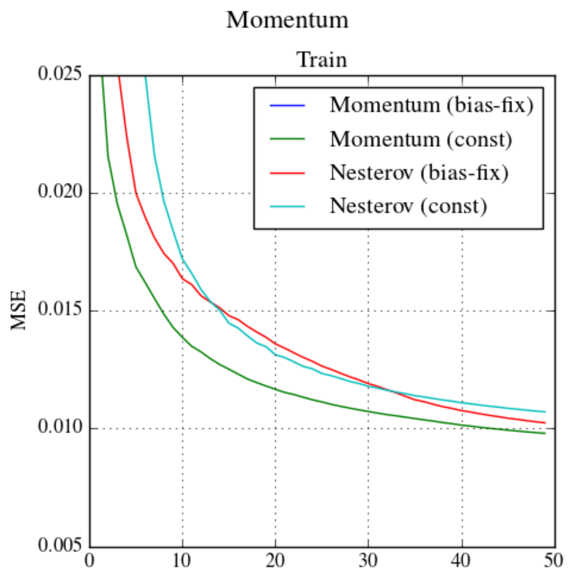

Figure 1. Graph of Momentum and Nesterov ($\mu = .9$) with a constant schedule and an initialization-bias correction schedule with learning rate 2 and scaled gradients. The bias-fixed momentum diverged, whereas the bias-fixed Nesterov appears slated to overtake momentum within a few more training epochs.

rates collide with the large, unreliable gradients that begin training, it is easy for the optimizer to take large steps in the wrong direction relatively early on, and these steps may take a while to recover from (especially when using leaky ReLUs that have a very small gradient when nearly dead). So while NAG will probably eventually overtake CM and converge to a better solution, it may take more training time for this to happen.

| CM | None | Bias-fix | NAG | None | Bias-fix |
|---|---|---|---|---|---|
| $\mu = .9$ | **9.30** | 9.98 | $\mu = .9$ | 11.92 | 11.09 |
| $\mu = .95$ | 9.49 | 10.70 | $\mu = .95$ | 9.65 | 11.58 |

Table 1. MSE ($\times 1000$) on the autoencoder task after 50 epochs with $\alpha_0$ set to 2 and storing the unscaled gradients in $\mathbf{m}_t$, comparing the performance of classical momentum and Nesterov's accelerated gradient with two values of $\mu_{\text{max}}$ and two kinds of schedules (constant $\mu$ and bias correction).

## 4.3. Adaptive Learning Rates

In this section we examine how different adaptive learning rate optimizers perform. In particular, we focus on the choice of the uncentered variance parameter $\nu$ and its schedule, as well as the norm parameter $p$. The results are shown in Table 4.3.

The first interesting observation–other than that they all perform significantly better than even NAG–is that most variants perform reasonably well no matter what the hyperparameter setting, and the $L_1$ and $L_2$ norm variants generally outperform the max norm ones. Interestingly, however, each of the three tested norm variants "prefers" a different initial learning rate–the $L_1$ norm worked best with $\alpha_0 = .001$, the $L_2$ variant liked $\alpha_0 = .002$, and the $L_\infty$ norm preferred $\alpha_0 = .005$. This makes sense given the nature of the norms–the $L_1$ norm of a vector whose values are all less than one will be smaller that the $L_2$ norm of the same vector, and the $L_\infty$ norm will obviously be the largest. So when we invert that in the learning algorithm, we have that the algorithm using the $L_1$ norm will make larger steps than the other two (given the same learning rate) and the max norm will take smaller ones. In order to ensure that they all take steps of roughly the same magnitude, we need to give them different learning rates.

The only other noteworthy pattern to be seen here is that when $\nu$ is larger than .9, the variants that use no schedule (and therefore aren't corrected for initialization bias) significantly underperform, either completely diverging or simply failing to reach anything resembling a good solution. This has a fairly intuitive explanation: the update rule for RMSProp using constant $\nu$ can be rewritten in terms of the squared gradients (or hessian-vector products if using EGD):

$$\theta_t = \frac{\alpha_t}{\sqrt{(1-\nu)\sum_{i=1}^{t} \nu^{t-i}\mathbf{h}_i^2 + \epsilon}}\mathbf{g_t}$$

The factor of $\frac{1}{\sqrt{1-\nu}}$ evaluates to about 31.6 for $\nu = .999$–that is, not using initialization bias correction is equivalent to multiplying the learning rate by 31.6. Obviously, one can correct for this by using a smaller learning rate–however, as training proceeds, the sum in the denominator gets larger, meaning the effective learning rate gets even smaller. While I don't have the data to scrutinize the interaction between the learning rate, schedule, and variance parameter, I suspect that starting with a learning rate small enough to compensate for a large $\nu$ early on may hinder later training, and this seems to be consistent with what Kingma & Ba found [5].

While equilibrated gradient descent (EGD) shows promise as a powerful learning algorithm[5], for this task it appeared to fail miserably–even the best optimizer using EGD vastly underperformed compared to optimizers that only use first-order information. This is particularly surprising because the researchers who proposed the algorithm use a convolutional autoencoder trained on MNIST and show that it outperforms some of the baselines discussed here. There are a number of possible reasons for this discrepancy. The first is that EGD approximates the diagonal of the Hes-

[5]Anecdotally, it is very good at avoiding attractive local minima in other tasks

sian by repeatedly sampling random vectors and performing an efficient computation of the Hessian-vector product on these–it seems plausible that models with many parameters would need many samples before gains from the more powerful second-order information could be seen. Since these models presented here were only trained for 50 epochs, perhaps it simply needed more passes through the dataset to build up its variance vector $\mathbf{v}_t$. Another possible explanation is that Theano's L-op, which is used for efficient computation of the Hessian-vector product, is not suited to the particular model architecture employed in this research. Specifically, it produced `nan` values for every parameter below the highest max-pooling layer in the encoder.[6] In an attempt to accommodate the algorithm anyway, the implementation employed here backed off to the original gradient wherever `nan`'s occurred in the Hessian-vector product (slightly scaled up, since the gradient was on average a few times smaller)–however, it's possible that combining the algorithms in this way–with some layers using EGD and some using RMSProp–simply doesn't work. In any case, the fact that many deep learning packages don't yet fully support the tools needed to implement EGD (Theno's R-op and L-op seem to be not fully incorporated into the framework, and I believe TensorFlow lacks them altogether) is certainly a blow against the algorithm's current viability.

| $L_1$ | None | Bias-fix | $L_2$ | None | Bias-fix |
|---|---|---|---|---|---|
| $\nu = .9$ | 6.98 | **6.91** | $\nu = .9$ | 7.08 | 7.10 |
| $\nu = .999$ | `nan` | 7.22 | $\nu = .999$ | 44.35 | 7.15 |

| $L_\infty$ | None | Bias-fix |
|---|---|---|
| $\nu = .9$ | 7.63 | - |
| $\nu = .999$ | 10.61 | - |

Table 2. MSE ($\times 1000$) on the autoencoder task after 50 epochs with $\alpha_0$ set to .001, .002, and .005 for the $L_1$, $L_2$, and $L_\infty$ norms, respectively, comparing the performance of of the three kinds of norms with two values of $\nu_{\text{max}}$ and two kinds of schedules (constant $\nu$ and bias correction). Other results involving the true mean and hand-crafted schedule are comparable to $\nu = .999$ and the bias correction schedule.

## 4.4. Combinations

Kingma & Ba [5] showed that momentum and RMSProp-based methods can be combined. Here we examine the performance of different variant combinations as well as their sensitivity to hyperparameters, focusing on the values of $\mu$ and $\nu$ as well as the choice of momentum type (CM vs NAG) and norm ($L_1$, $L_2$, $L_\infty$). Unfortunately, practical constraints have prevented an exploration of how scaled-gradient momentum interacts with RMSProp, and all experiments here use bias-initialization schedules for both

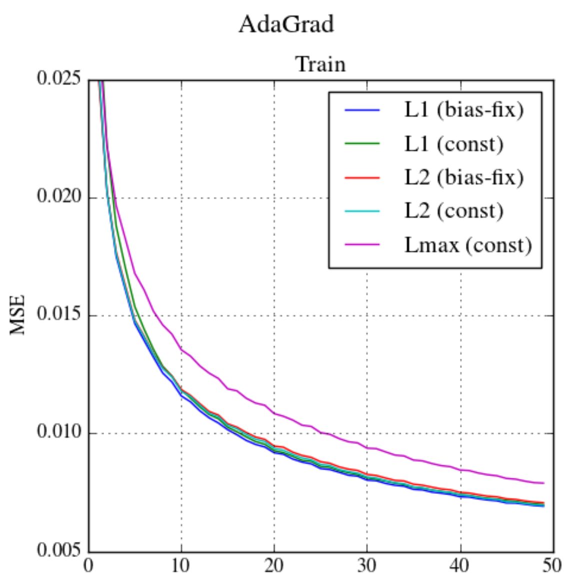

Figure 2. Graph of decaying mean variants of AdaGrad with different schedules for $\nu$. Most achieve comparable performance, but the max-norm version is clearly behind.

decaying mean hyperparameters. The results are given in 4.4.

Firstly, when unscaled-gradient momentum (either CM or NAG) with a decaying mean (as opposed to a decaying sum) is switched in for $\mathbf{g}_t$ in an AdaGrad variant, the preferred learning rate stays the same–that is, $L_1$ the variant still prefers .001, the $L_2$ variant still prefers .002, etc. In one sense, this is what we would expect if the momentum vector $\mathbf{m}_t$ represents an estimate of the true gradient of the loss function–presumably the gradient of the true cost function $f(\theta)$ (the batch gradient) has approximately the same magnitude as the gradient of any of the stochastically generated cost functions (the minibatch gradient), so we would want the model to travel approximately the same distance no matter which gradient it used. However, as we saw above, the momentum methods tolerate higher learning rates than SGD, so we might expect Momentum+AdaGrad combinations to tolerate higher learning rates as well. One explanation for this not apparently holding is that the AdaGrad variants actually do do some of the same things the momentum variants do, namely help prevent divergeance–while infrequent activation patterns can still undergo extreme changes with AdaGrad, common ones that are more at rist of triggering oscillations that ultimately lead to divergeance are automatically slowed down after their first few large jumps. So because AdaGrad variants and momentum variants both prevent early-stage learning oscillations and divergeance, it would make sense that one could swap out $\mathbf{g}_t$ in the AdaGrad variants for a decaying mean version of $\mathbf{m}_t$ or $\bar{\mathbf{m}}_t$

[6]It may be that the Hessian for max pooling layers actually is zero or undefined; I haven't worked out the math myself yet.

| CM, $L_1$ | $\mu = .9$ | $\mu = .95$ | NAG, $L_1$ | $\mu = .9$ | $\mu = .95$ |
|---|---|---|---|---|---|
| $\nu = .9$ | $6.65$ | $6.95$ | $\nu = .9$ | $6.80$ | $7.12$ |
| $\nu = .999$ | $7.26$ | $7.36$ | $\nu = .999$ | $7.42$ | $7.09$ |
| CM, $L_2$ | $\mu = .9$ | $\mu = .95$ | NAG, $L_2$ | $\mu = .9$ | $\mu = .95$ |
| $\nu = .9$ | $6.51$ | $6.31$ | $\nu = .9$ | $6.56$ | $7.00$ |
| $\nu = .999$ | $6.75$ | $6.95$ | $\nu = .999$ | $6.36$ | $7.48$ |
| CM, $L_\infty$ | $\mu = .9$ | $\mu = .95$ | NAG, $L_\infty$ | $\mu = .9$ | $\mu = .95$ |
| $\nu = .9$ | $6.95$ | $8.35$ | $\nu = .9$ | $\mathbf{5.90}$ | $7.58$ |
| $\nu = .999$ | $7.65$ | $7.30$ | $\nu = .999$ | $7.80$ | $7.46$ |

Table 3. MSE ($\times 1000$) on the autoencoder task after 50 epochs. The learning rates are the same as in Table 4.3, both $\mu$ and $\nu$ use the initialization-bias correction schedule, and all variants use unscaled-gradient momentum (as in Adam).

without having to re-tune the learning rate.

Using NAG momentum without any kind of adaptive learning rate was generally worse than CM in this setting–however, when adaptive learning rates are incorporated back into the algorithm, algorithms with NAG instead of CM become competitive again. However, unless the learning rate is reduced (which results in worse performance for both CM and NAG, at least after 50 epochs), they still don't consistently outperform those that only use CM. This suggests that combining NAG variants with AdaGrad variants alleviates some of the issues that arise from assigning more weight to the current gradient during early-stage learning updates, but not all of them. The initial steps that NAG takes still overweight the current gradient, which still means often leaving large swaths of leaky ReLUs nearly dead, but because the nearly dead leaky ReLUs will have consistently small gradients until something in the data manages to trigger them, the denominator of the AdaGrad component will gradually shrink as the large initial steps start to wear off, resulting in larger steps to recovery than NAG by itself sees.

Finally, incorporating momentum–of any kind–into EGD produced few if any noticeable gains.

## 4.5. Summary

There are a number of key takeaways that we can learn from this discussion. First, using leaky ReLUs was integral for achieving the best performance in as few epochs as possible. However, they seemed not to play as nicely with NAG. Regarding the hyperparameters $\mu$ and $\nu$: the best values for both parameters seem to be around .9 for this task, although the algorithms seem more robust to variations in $\nu$ than $\mu$. For the momentum hyperparameter $\mu$, zero-initialization bias correction seems not to help in general (and sometimes even to hurt), but for $\nu$, it's much more important. The max-norm variants of AdaGrad are at best very sensitive to hyperparameters and at worst just not as good–using $L_1$ and $L_2$ norms seems to work better, and when combined with momentum, the $L_2$ norm variants generally achieve lower training error (although it is worth acknowledging that the $L_\infty$ variant with NAG achieved the lowest training error overall).

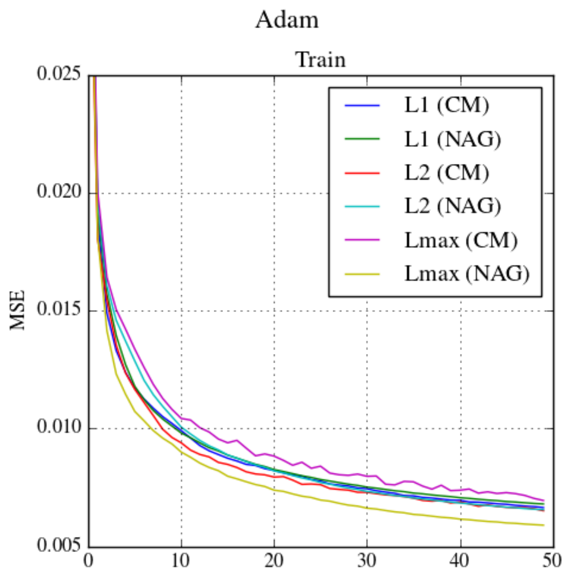

Figure 3. Graph of initialization bias-corrected combinations of unscaled-gradient momentum and decaying mean AdaGrad.

## 5. ResNets

The task above is essentially a toy task–the input dataset is simple, the model is relatively small, and the goals are not useful for practical purposes. So how do we know that the observations above will carry over into tasks involving more complex datasets, with larger models, with real purposes? Residual Networks (ResNets) [3] were originally designed for image recognition and have a relatively simple structure that nonetheless facilitates backpropagation through the network. Because this model is designed to facilitate efficient learning, we might ask whether the model structure renders the choice of optimization algorithm or hyperparameter settings inconsequential.

Limited again by time and computing power, we were only able to train a few fairly shallow models on CIFAR-10 with a representative sample of learning algorithms. Specifically, we examined unscaled-gradient NAG with $\alpha_0 = 1$, $L_2$ and $L_1$ norm variants of decaying-mean AdaGrad, $L_1$ and $L_2$ variants of AdaGrad, and $L_2$ AdaGrad with NAG. EGD again failed to compile. $\mu$ was set to .9, $\nu$ was set to .999[7], and all used initialization-bias correction schedules. In order to provide a more complete analysis we present both training and validation error in Table 5.

There are a few interesting things to note–the first is that

[7]This value was chosen before we realized that .9 may be more robust

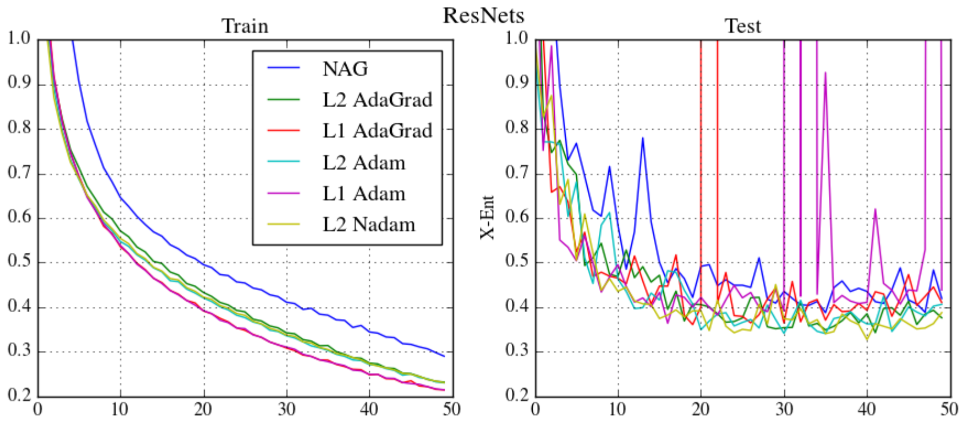

Figure 4. Training/validation cross entropy of resnets trained on CIFAR-10

once again, all adaptive learning rate methods outperform the momentum one. And in fact, the AdaGrad-based approaches without momentum end with slightly lower training and testing error, although based on Figure 4, clearly 50 epochs were insufficient to achieve convergeance–it's very possible that the variants with momentum would eventually overtake them. Additionally, the algorithms that used the $L_1$ norm instead of the $L_2$ norm achieved lower training error but were exceptionally poor at generalizing. In contrast, the three $L_2$ models are fairly consistently lowest in terms of test error, suggesting that either a .001 learning rate for the $L_1$ norm doesn't transfer over to other tasks as well in practice, and is too high here, or that $L_2$ norms are in general more conservative and robust to small perturbations in the input.

| | Train | Test |
|---|---|---|
| NAG | 0.289 | 0.419 |
| $L_2$ AdaGrad | 0.232 | **0.376** |
| $L_1$ AdaGrad | **0.214** | 0.410 |
| $L_2$ Adam | 0.230 | 0.406 |
| $L_1$ Adam | 0.214 | 0.438 |
| $L_2$ Nadam | 0.230 | 0.388 |

Table 4. The results of training resnets with different optimization algorithms.

## 6. Conclusion

This exploration makes a few contributions. First, it finds that the value for $\nu$ for Adam (and which should be just as applicable to RMSProp) suggested by Kingma & Ba [5] may be too high, and that $\nu = .9$ may be better in practice. Similarly, it finds that *not* correcting for initialization bias in momentum methods may be better that the alternative. It explored the effect of the norm used in extensions of AdaGrad, finding that the original $L_2$ norm seems to be the most robust (although it may be the case that using an $L_1$ norm instead sometimes works just as well or better). One must be careful when working with ReLUs, especially when using some kind of momentum, as this work seems to have run into problems with them. Equilibrated gradient descent–while extremely theoretically appealing–seems to have implementational problems that prohibit its use in some models. Finally, even when working with architectures specifically designed to facilitate gradient propagation and speed up training, using a fast and more powerful optimization algorithm still allows for even faster convergeance.

## References

[1] Y. Dauphin, H. de Vries, and Y. Bengio. Equilibrated adaptive learning rates for non-convex optimization. In *Advances in Neural Information Processing Systems*, pages 1504–1512, 2015.

[2] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[3] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[4] M. S. Jones. Convolutional autoencoders in python/theano/lasagne. Blog post (retrieved February 17, 2016, April 2015.

[5] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] Y. LeCun, C. Cortes, and C. J. Burges. The mnist database of handwritten digits, 1998.

[7] Y. Nesterov. A method of solving a convex programming problem with convergence rate $O(1/k^2)$. In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[8] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[9] I. Sutskever, J. Martens, G. Dahl, and G. 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.

[10] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.