AUTOLR: A METHOD FOR AUTOMATIC TUNING OF LEARNING RATE

Anonymous authors
Paper under double-blind review

ABSTRACT

One very important hyperparameter for training deep neural networks is the learning rate of the optimizer. The choice of learning rate schedule determines the computational cost of getting close to a minima, how close you actually get to the minima, and most importantly the kind of local minima (wide/narrow) attained. The kind of minima attained has a significant impact on the generalization accuracy of the network. Current systems employ hand tuned learning rate schedules, which are painstakingly tuned for each network and dataset. Given that the state space of schedules is huge, finding a satisfactory learning rate schedule can be very time consuming. In this paper, we present *AutoLR*, a method for auto-tuning the learning rate as training proceeds. Our method works with any optimizer, and we demonstrate results on SGD, Momentum, and Adam optimizers.

We extensively evaluate *AutoLR* on multiple datasets, models, and across multiple optimizers. We compare favorably against state of the art learning rate schedules for the given dataset and models, including for ImageNet on Resnet-50, Cifar-10 on Resnet-18, and SQuAD fine-tuning on BERT. For example, *AutoLR* achieves an EM score of 81.2 on SQuAD v1.1 with $\text{BERT}_{\text{BASE}}$ compared to 80.8 reported in (Devlin et al. (2018)) by just auto-tuning the learning rate schedule. To the best of our knowledge, this is the first automatic learning rate tuning scheme to achieve state of the art generalization accuracy on these datasets with the given models.

1 INTRODUCTION

Learning rate is one of the most important hyperparameters that impact deep neural network (DNN) training performance. It determines how fast we get close to a minima, which in turn determines the computational cost of optimization. It also determines how close we get to the minima, e.g. higher learning rates may get in the neighborhood of a minima much faster, but then just bounce at a height above the minima (Xing et al. (2018); Nar & Sastry (2018)). The learning rate also determines the kind of minima (e.g. wide vs narrow) attained (Keskar et al. (2016)), which has a significant impact on the generalization accuracy.

Therefore, it is not surprising that a lot of effort has gone into automatically tuning the learning rate (Schraudolph (1999); Schaul et al. (2013b); Rolinek & Martius (2018)). However, till date, none of these techniques have been able to deliver state of the art test accuracy on standard benchmarks. Instead, deep learning researchers today rely on a mixture of brute force search, augmented with simple heuristics such as using a staircase, polynomial, or exponential decay-based learning rate schedules. This problem is further exacerbated by the fact that different optimizers such as SGD or Adam work best on different datasets and require very different learning rate schedules.

For example, while training image datasets such as Cifar-10 and ImageNet with SGD or Momentum, a staircase schedule with high starting learning rate typically performs best. On the other hand, with the Adam optimizer, a smaller starting learning rate is generally used. Further, in NLP tasks such as machine translation with Adam, typically a linear warmup is followed by polynomial decay, e.g. the Transformer network (Vaswani et al. (2017)) uses a linear warmup followed by an inverse square root decay. Similar schedules are used for BERT (Devlin et al. (2018)) pre-training, while BERT fine-tuning uses a linear decay.

Our key idea to tackle this problem is driven by the following simple observation. Consider training Cifar-10/ImageNet using the typical staircase schedule. One can observe that the initial high learning

rate results in training loss stagnating after a few epochs. However, if one were to decay the learning rate when loss stagnates, the achieved test accuracy is considerably lower as compared to letting the high learning rate continue for longer, despite no apparent improvement in training loss.

Our insight is that this initial phase of training using a high learning rate, even with no improvement in loss, is crucial for generalization and is one of the key missing pieces from prior attempts at automatic learning rate tuning. We hypothesize that for DNNs, the number of narrow minimas far outnumber the wide minimas. To generalize well, we want the optimizer to land in wide minimas. An interesting intuitive observation is that a large learning rate can escape narrow minimas easily (as the optimizer can jump out of them with large steps), however once it reaches a wide minima, it is likely to get stuck in it (if the "width" of the wide minima is large compared to the step size).

The above hypothesis motivates our *Explore-Exploit* scheme where we force the optimizer to first *explore* the landscape with a high learning rate for sometime in order to land in a wide minima. We should give the *explore* phase enough time so that the probability of landing in a wide minima is high. Once we are confident that the optimizer is stuck in the vicinity of a wide minima, we activate the *Exploit* phase of *AutoLR*.

The basic idea of the *exploit* phase is as follows. We first look at how local perturbations in the current learning rate impact the training loss. Since we only look at local perturbations, we can model the loss as a function of these perturbations via a Taylor series expansion. We then make a quadratic approximation, and solve for the optimal perturbation in the learning rate which will minimize the loss. This is similar to Newton's method, but applied only in the descent direction (section 2). We do extensive validation that the quadratic is a good approximation here. Although the basic idea is simple, it is complicated by the fact that deep learning relies on stochastic optimization methods, such as SGD, which causes the loss values to be noisy across mini-batches, potentially throwing off our estimates. We discuss how we handle stochasticity in section 2.2.

We demonstrate the efficacy of the explore-exploit approach by evaluating *AutoLR* across a wide range of models and datasets, ranging from NLP (SQuAD on BERT-base, Transformer on IWSLT) to CNNs (e.g. ImageNet on ResNet-50, Cifar-10 on ResNet18), and across multiple optimizers: SGD, Momentum and Adam. In all cases, *AutoLR* matches or beats test accuracy of state-of-the-art hand-tuned learning rate schedules. For example, on SQuAD v1.1 fine-tuning with $\text{BERT}_{\text{BASE}}$, *AutoLR* is able achieve an EM score of 81.2, compared to 80.8 reported in Devlin et al. (2018) by just auto-tuning the learning rate schedule (all other parameters were unchanged). To the best of our knowledge, *AutoLR* is the first automatic learning rate scheduler to achieve state of the art results on these datasets for the given models. We show extensive evaluation of our method in section 3.

The main contributions of our work are:

1. The observation that an initial Explore phase with high learning rate is crucial for good generalization in automatic learning rate schemes.
2. Incorporating this observation via an *Explore-Exploit* approach with a novel exploitation scheme for tuning learning rate using a local approximation of the optimization landscape.
3. The first automatic learning rate tuning scheme that beats or achieves generalization of state of the art learning rate schedules in multiple models/datasets including ImageNet.

2 METHOD

Let $L(\theta)$ be the loss of the network as a function of its parameters θ. Note that in practice, since we work with stochastic optimizers such as SGD, the loss computed in each minibatch is only an approximation of the true loss. This distinction is important when we come to the estimation of the best learning rate, but can be ignored for the formulation below.

The loss of the network in the next time step is $L(\theta - \eta\vec{d})$, where $\vec{d}$ is the step direction, and η is the learning rate. If we think of $L(\theta - \eta\vec{d})$ as a function of the learning rate η, our task is to find an η which minimizes this function. Since it is hard to directly estimate L as a function of η, we instead consider what happens if we perturb η by a small amount ϵ, i.e we look at $L(\theta - (\eta + \epsilon)\vec{d})$. Looking

at this as a function of ϵ, and applying Taylor series expansion we get,

$$\begin{aligned}\hat{L}(\epsilon) &= L(\theta - (\eta + \epsilon)\vec{d}) = L(\theta - \eta\vec{d} - \epsilon\vec{d}) \\ &= L(\theta - \eta\vec{d}) - \epsilon\vec{d}^T\vec{g} + \frac{1}{2}\epsilon^2\vec{d}^T H\vec{d} + \mathcal{O}(\epsilon^3),\end{aligned} \tag{1}$$

where $\vec{g}$ is the gradient of L w.r.t θ, and H is the Hessian. Note that computing accurate value of the gradient $\vec{g}$ is expensive, as it requires going over the entire dataset (not just a minibatch), while computing the Hessian H is typically intractable for deep networks which have millions of parameters. However, it turns out that we don't actually need the values of $\vec{g}$ and H. In fact, we can just look at $\hat{L}$ as a quadratic function of ϵ, i.e.

$$\hat{L}(\epsilon) = k_0 + k_1\epsilon + k_2\epsilon^2 + \mathcal{O}(\epsilon^3). \tag{2}$$

To compute $\{k_0, k_1, k_2\}$, we simply evaluate the loss at a few values of ϵ and fit a quadratic polynomial. Note that we are able to get away with having to calculate the high dimensional $\vec{g}$, and H, because the loss function only looks at the effect of these first and second order derivatives in *one single direction* $\vec{d}$, as determined by the terms $\vec{d}^T\vec{g}$ and $\vec{d}^T H\vec{d}$. Once we know the values of $\{k_0, k_1, k_2\}$, we can simply minimize this quadratic to get the optimal value of $\epsilon_{min} = -\frac{k_1}{2k_2}$ to minimize the loss L in the next time step. Also observe that our method only needs the search direction $\vec{d}$ for computing the various loss samples. As a result, our method works for any optimizer, as long as we can access the search direction. We have implemented our method for multiple optimizers including SGD, Momentum, and Adam.

Note that it is very important to use a perturbation ϵ for the Taylor expansion in equation 1, rather than just expanding on η via $L(\theta - \eta\vec{d}) = L(\theta) - \eta\vec{d}^T\vec{g} + \frac{1}{2}\eta^2\vec{d}^T H\vec{d} + \mathcal{O}(\eta^3)$. This is because η may not be small, causing the $\mathcal{O}(\eta^3)$ error to be quite large. See appendix A for empirical validation that a second order approximation captures the loss perturbations accurately.

Epsilon Thresholding. The quadratic approximation in equation 2 yields an optimal value of $\epsilon_{min} = -\frac{k_1}{2k_2}$ for the minimum loss L at the next time step. Note that the quadratic approximation is valid only up to an $\mathcal{O}(\epsilon^3)$ error, while the minimization can yield ϵ_{min} with large absolute values, thereby making huge errors. That is, although our quadratic approximation is valid only for small values of $|\epsilon|$, the quadratic's minima may be beyond this approximation range, as illustrated in Figure 1. Thus, we need to threshold ϵ_{min} to a reasonable value. Since we are interested in approximating the loss to some precision, we use a relative threshold as follows. We approximate $\mathcal{O}(\epsilon^3)$ as $|\epsilon|^3$, and bound the error:

$$|\epsilon|^3 < r * L(\theta), \tag{3}$$

where r is the relative error threshold.

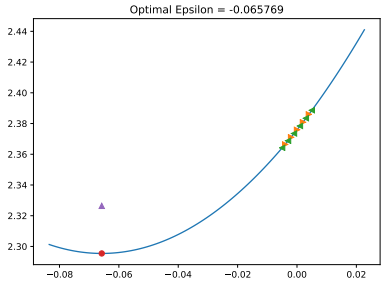


Figure 1: A large $|\epsilon_{min}|$ can give bad approximation. Orange and green triangles show loss samples used in fitting and testing respectively. Red circle shows the minimum loss value of the quadratic, and purple triangle shows the true loss there.

2.1 Tackling Generalization

The choice of learning rate is an important factor in determining generalization quality of the trained network. The above method for tuning the learning rate is a local method, and does not take into account the highly non-linear and non-convex global optimization landscape of deep networks. As a result it can take globally suboptimal decisions which can impact generalization. In this section we discuss the problem in more detail, and our solution.

Although understanding generalization of deep neural networks is an open problem, there have been interesting findings recently. Kawaguchi (2016) found that deep neural networks have many local minimas, but all local minimas are also the global minima (also see Goodfellow et al. (2016)). Also, it is widely believed that wide minimas generalize much better than narrow minimas (Hochreiter & Schmidhuber (1997); Keskar et al. (2016); Jastrzebski et al. (2017); Wang et al. (2018)), even though

they have the same training loss. Keskar et al. (2016) found that small batch SGD generalizes better than large batch SGD and also lands in wider minimas, suggesting that noise in SGD acts as an implicit regularizer. Interestingly however, more recent work was been able to generalize quite well even with very large batch sizes (Goyal et al. (2017); McCandlish et al. (2018); Shallue et al. (2018)), by scaling the learning rate linearly as a function of the batch size. This suggests that the lack of noise in large batch SGD can be compensated with high learning rates, and thus the learning rate plays a crucial role in generalization.

Many popular learning rate schedules start the training with high learning rates, and then reduce the learning rate after every few epochs or following some hand tuned fall curve. An interesting observation in many such examples is that even though a high learning rate stagnates after a few epochs, one still needs to run enough iterations at the higher learning rate in order to get good generalization. For example, see Figure 2, which shows the training loss of a Resnet-18 model training on Cifar10 dataset at a fixed LR of 0.1 with SGD. The training loss stagnates after ≈ 50 epochs, however as shown in Table 1, generalization continues to improve if we increase the number of epochs trained at a higher learning rate. This leads us to the following hypothesis.

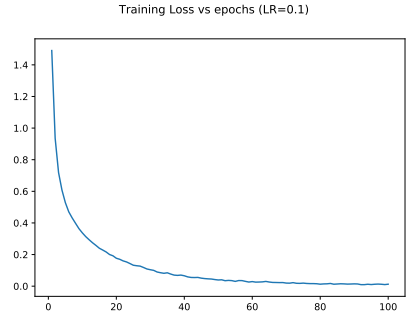


Figure 2: SGD training loss for Cifar-10 at LR of 0.1. Loss stagnates after ≈ 50 epochs.

Explore Epochs	Test Accuracy
40	90.93
50	90.94
60	91.41
80	91.43

Table 1: Cifar-10 on Resnet-18 is trained for 200 epochs with SGD. A LR of 0.1 is used for the explore epochs. Half the remaining epochs are trained at 0.01 and the other half at 0.001. Reported results are average over 4 runs.

We hypothesize that for deep neural networks, narrow minimas far outnumber the wide minimas. To generalize well, we want the optimizer to land in wide minimas. In that respect, an interesting intuitive observation is that a large learning rate can escape narrow minima "valleys" easily (as the optimizer can jump out of them with large steps), however once it reaches a wide minima "valley", it is likely to get stuck in it (if the "width" of the wide valley is large compared to the step size).

Explore-Exploit: This motivates our *Explore-Exploit* scheme where we force the optimizer to first *explore* the landscape with a high learning rate for sometime in order to land in the vicinity of a wide minima. During the explore phase, we only allow a monotonic increase of LR and disallow any decrease even if suggested by the local quadratic fit. We should give the *explore* phase enough time so that the probability of landing in a wide minima is high. Since the ratio of number of narrow vs wide minimas can depend on the architecture, dataset, etc., predicting the right explore phase duration is hard, and is currently a hyperparameter for *AutoLR*. Once we are confident that the optimizer must be now stuck in the vicinity of a wide minima, we start the *exploit* phase and allow *AutoLR* to reduce the learning rate as dictated by the local method to make fast progress and get close to the minima of this hopefully *wide* minima.

Explore-Exploit improves the generalization of *AutoLR* significantly. Without an explore phase, the *AutoLR* scheme will start reducing the learning rate as soon as it reaches close to a local minima, irrespective of whether it is a narrow or wide minima. As shown in Figure 3, locally at epoch 50, it makes sense to reduce the LR as it increases the rate of descent. Without any *explore* phase, *AutoLR* (Eq 2), will make such a decision as well, potentially hurting generalization.

To validate our hypothesis and the effectiveness of the explore-exploit scheme, we first analyze a state-of-the-art learning rate schedule for Cifar-10 on Resnet-18. This learning rate schedule trains with learning rates of $10^{-1}, 10^{-2}, 10^{-3}$ for 100, 50, 50 epochs each. The first 100 epochs at a high learning rate of 0.1, can be though of as an explore phase here, as discussed in Figure 2. We experimented with reducing this "explore" phase, and as shown in Table 1, it is clear that explore

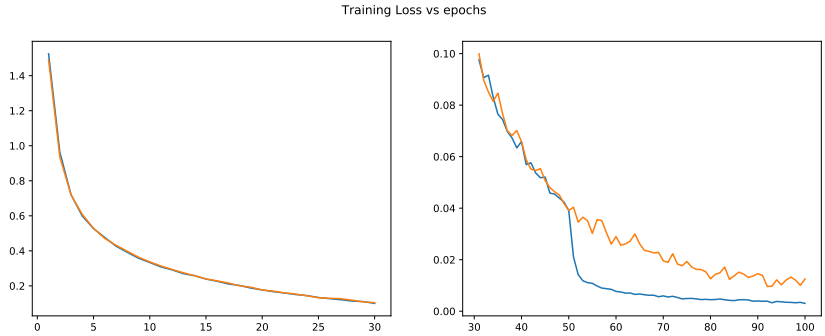


Figure 3: SGD training loss for Cifar-10 on Resnet-18. Plots are split across two to permit higher y-axis fidelity. In the orange plot, a fixed LR of 0.1 is used, while in the blue plot the LR is reduced from 0.1 to 0.01 at epoch 50. Clearly, at epoch 50, it locally makes sense to reduce the LR as it increases the rate of descent significantly.

Explore Epochs	Test Accuracy
0	90.43
10	90.58
25	91.13
50	91.42

Table 2: Cifar-10 on Resnet-18 trained for 200 epochs with SGD. An LR of 0.1 is used for the explore epochs, and is determined by *AutoLR* for rest of the epochs. We report averages over 4 runs.

phase plays an important role in the final generalization accuracy. We now evaluate explore-exploit on the same model and dataset but with learning rate determined by *AutoLR* in the *exploit* phase, and try different duration of the *explore* phase. Table 2 shows the test accuracy for different number of explore epochs. As shown, the explore phase helps with improving generalization accuracy significantly from 90.43 without explore to 91.42 with 50 explore epochs.

Hypothesis validation: To validate our hypothesis of narrow minimas outnumbering wide minimas, we run the following experiment. We train Cifar-10 on Resnet-18 on a high learning rate of 0.1 for only 40 epochs and then 80 epochs each at 10^{-2} and 10^{-3}. We do this training many times with different random initializations. As shown in table 1, on an average this should yield a low generalization accuracy. However, if our hypothesis was true, there is still some chance that, in a few of the many runs, our optimizer lands in the wide minima within 40 epochs. In fact, we find that in 1 of the 20 runs our test accuracy reaches 91.71 as opposed to an average test accuracy of 90.92 over 20 runs. In fact, the best accuracy achieved is even higher than the run with 100 epochs at 0.1, and is likely because the optimizer had more time (160 vs 100 epochs) at lower learning rates to get to the bottom of the wide minima, whose vicinity it had landed in by chance.

Saturation Threshold: The above discussion suggests that overall, higher learning rates are better for generalization and thus one should prefer them even though a local analysis may suggest otherwise. To incorporate this idea into our method we employ what we call a *saturation threshold*. The idea is that we want to continue with the current learning rate and not lower it, unless it has *saturated* in terms of loss drop per iteration. That is, if the training loss drop rate has dropped below a threshold for the current learning rate, it suggests that the current learning rate has served its purpose, and we can move on to a lower learning rate if suggested so by *AutoLR*. We can either use an absolute threshold, or a relative threshold where we use the ratio of loss-drop rate when we started using the current learning rate to the current loss-drop rate. Finding the best absolute threshold for each model/dataset can be tricky, so we use a relative threshold, which we found easy to tune. Also, we noticed that the loss drop rate doesn't change drastically towards the end of the training when the loss has anyways stabilized. This can cause a high relative saturation threshold to be too strict in the later stages of training, while it would perform well early on. To handle this, we used a simple strategy where we choose a relative saturation threshold initially, and first time the saturation threshold is crossed, we switch to an absolute threshold with the current loss drop date as the absolute saturation threshold. This essentially amounts to using the relative threshold to determine a good absolute threshold value for the current model and dataset, which is then used subsequently.

Note that, although both *explore phase* and *saturation threshold* prefer higher learning rates, they serve different purposes. While the explore phase ensures that the optimization reaches neighborhood of a wide minima before it reduces the learning rate (by forcing the optimizer to use a higher learning rate for some minimum time, possibly even after it has saturated); saturation threshold adds an overall preference for a higher learning rate until that learning rate has served its purpose and is not optimizing the loss satisfactorily.

2.2 HANDLING STOCHASTICITY

For computation of the quadratic coefficients in equation 2, we need to compute the loss values $\hat{L}(\epsilon)$ at multiple values of ϵ. Note that in a typical DNN setting we never compute the full loss, but only a stochastic loss based on a given minibatch. This loss, however, can be noisy and throw off our estimate of ϵ_{min}. To handle this we follow two simple strategies. The first is to use a bigger minibatch (called *superbatch*) for computing the loss values, and the second is to use the *same* superbatch for computing all the losses for a particular estimate. The same strategy is used when computing loss drop rates for the saturation threshold check. We use an integer multiple of minibatches to make a superbatch. This allows us to compute the superbatch loss by simply averaging multiple single batch forward passes, and does not increase the peak GPU memory usage. We use a conservative superbatch size of 100 for all our examples. See section B for details on selection of superbatch size.

Finally, as a safeguard we also added a *rollback* policy. In case our system makes a bad call on the learning rate change (most likely because of a bad superbatch sample), which leads to a reduction in loss drop rate, we rollback the decision and revert the state of the network to the time we made the learning rate change. Although, *rollback* triggers rarely, it is helpful in preventing the optimization from going astray because of one bad decision.

3 EXPERIMENTS

We extensively evaluate our method on multiple networks and datasets, as well as multiple optimizers including SGD, Momentum and Adam. We have implemented *AutoLR* as an optimizer in PyTorch (Paszke et al. (2017)), which wraps an existing optimizer, such as SGD, Adam, etc. For our experiments, we used an out of the box policy as in Rolinek & Martius (2018), where we only change the optimizer to *AutoLR*, and don't modify anything else. We evaluate on multiple image datasets – Imagenet on Resnet-50, Cifar-10 on Resnet-18, MNIST, FashionMNIST; as well as NLP datasets – Squad v1.1 for BERT finetuning and IWSLT with transformer networks. (Note: The source code for *AutoLR* will be open sourced, and a link to it has been sent to Reviewers and ACs for this submission.)

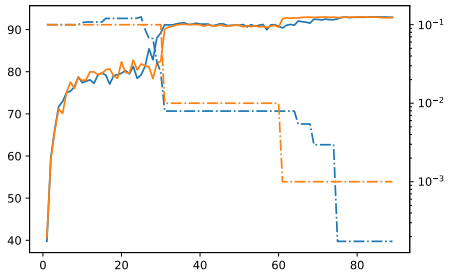
3.1 IMAGENET IMAGE CLASSIFICATION ON RESNET-50

In this experiment we trained the ImageNet dataset (Russakovsky et al. (2015)) on Resnet-50 network [1]. We evaluated our method on SGD with momentum which performs best for this dataset. For baseline runs, we used the standard hand-tuned learning rate schedule of $10^{-1}, 10^{-2}, 10^{-3}$ for 30 epochs each. With *AutoLR*, we trained the network with 25 explore epochs, and used the same seed learning rate as baseline, i.e. 0.1. Table 3 shows the training loss and test accuracies for the various runs. As shown, we comfortably beat the test accuracy of the baseline. Figure 4a shows the learning rate and test top-5 accuracy of *AutoLR* against that of the baseline. See figure 8, for more detailed comparisons of training loss, test accuracy, and learning rate.

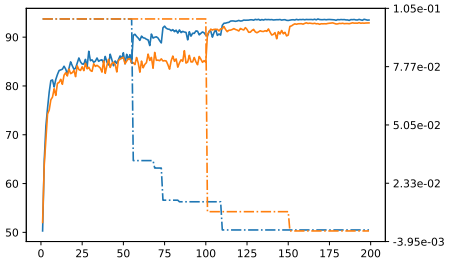
LR Schedule	Training Loss	Test Top 1 Acc.	Test Top 5 Acc.
Baseline	0.74 (0.001)	75.87 (0.035)	92.90 (0.015)
AutoLR	0.74 (0.041)	76.04 (0.098)	93.01 (0.024)

Table 3: Training loss and Test accuracy for ImageNet on Resnet-50. We report the mean and standard deviation over 3 runs.

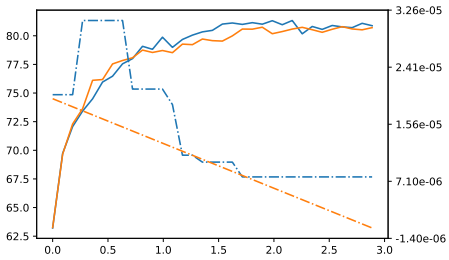
[1] We used the implementation at: https://github.com/cybertronai/imagenet18_old



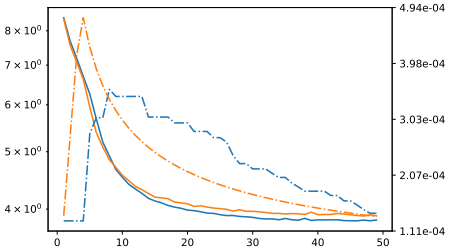
(a) ImageNet



(b) Cifar



(c) SQuAD



(d) IWSLT

Figure 4: Generalization accuracies and learning rates. *AutoLR* plots are show in blue, and baseline in orange. Learning rates are plotted with dashed lines with y-scale on the right, while generalization accuracies are plotted with solid lines with y-scale on the left. The generalization metrics plotted are (a) test top-5 accuracy (b) test accuracy (c) test EM score, and (d) validation perplexity.

3.2 CIFAR-10 IMAGE CLASSIFICATION ON RESNET-18

In this experiment we trained the Cifar-10 dataset (Krizhevsky et al. (2009)) on Resnet-18 network (He et al. (2016)) [2]. We evaluated our method on SGD, SGD with momentum and Adam optimizers. For baseline runs of SGD and Momentum, we used the standard hand-tuned learning rate schedule of $10^{-1}, 10^{-2}, 10^{-3}$ for 100, 50, 50 epochs, respectively; while for Adam we used a fixed learning rate of 1e-3. With *AutoLR*, we trained the network with 50 explore epochs, and used the same seed learning rate as baseline, i.e. 0.1 for SGD, Momentum, and 10^{-3} for Adam. Table 4 shows the training loss and test accuracy for the various runs. As shown, we do better in both training loss and test accuracy in almost all cases, except slightly missing out in SGD. Figure 4b shows the learning rate and test loss of *AutoLR* against that of the baseline for Momentum optimizer. See figures 9, 10, 11 for more detailed comparisons of training loss, test accuracy, and learning rate for SGD, Momentum and Adam optimizers, respectively.

LR Schedule	Optimizer		
	SGD	Momentum	Adam
Training Loss			
Baseline	7.5e-4 (1.0e-4)	0.002 (1.0e-4)	0.006 (9.8e-4)
AutoLR	3.8e-4 (1.8e-4)	0.002 (1.2e-3)	7.1e-4 (5.3e-4)
Test Accuracy			
Baseline	91.49 (0.003)	92.84 (0.003)	91.45 (0.003)
AutoLR	91.57 (0.004)	92.89 (0.004)	91.86 (0.004)

Table 4: Training loss and Test accuracy for Cifar-10 on Resnet-18. We report the mean and standard deviation over 7 runs.

See sections E.1 and section E.2 for results on MNIST and FashionMNIST datasets with SGD, Momentum and Adam optimizers.

[2]We used the implementation at: https://github.com/kuangliu/pytorch-cifar

3.3 SQuAD FINE-TUNING ON BERT

We now evaluate *AutoLR* on a few NLP tasks. In the first task, we fine-tune the $\text{BERT}_{\text{BASE}}$ model (Devlin et al. (2018)) on SQuAD v1.1 (Rajpurkar et al. (2016)) [3]. BERT fine-tuning is prone to ovefitting because of a huge model size compared to the fine-tuning dataset, and is typically run for only a few epochs. We use the standard baseline which trains for 3 epochs with a seed learning rate of $2e-5$ with linear decay. With *AutoLR* we found that the model overfits much earlier, seemingly because *AutoLR* reduces the training loss much more quickly (see figure 13), and we thus needed to fine-tune for only 2 epochs. The *AutoLR* runs were trained with 2500 explore steps ($\approx$ half epoch), and the same seed learning rate of $2e-5$ as baseline. Table 5 shows our results over 3 runs. We achieve a best EM score of 81.2 (after 2 epochs), compared to baseline's best of 80.7 (after 3 epochs). We found it interesting that *AutoLR* was able to tune the learning rate effectively even in the small budget of 2 epochs to beat the baseline number of $\text{BERT}_{\text{BASE}}$ (see Table-2 of Devlin et al. (2018) who reported an EM score of 80.8). Figure 4c shows the learning rate and test EM score of *AutoLR* against that of the baseline. See figure 13 for detailed comparisons.

LR Schedule	Train Loss (av)	EM (best)	EM (av)	F1 (av)
Baseline	0.96 (0.075)	80.7	80.4 (0.18)	88.2 (0.02)
AutoLR	1.05 (0.008)	81.2	80.8 (0.52)	88.5 (0.09)

Table 5: SQuAD fine-tuning on BERT. We report the average training loss, best test EM score, and average test EM, F1 scores over 3 runs.

3.4 MACHINE TRANSLATION ON TRANSFORMER NETWORK WITH IWSLT

In the second NLP task, we train the Transformer network (Vaswani et al. (2017)) on the IWSLT German-to-English (De-En) dataset (Cettolo et al. (2014)) with the Adam optimizer [4]. For baseline, we used the learning rate schedule mentioned in Vaswani et al. (2017). The baseline learning rate starts at $1.25e-7$, and is linearly increased for 4000 steps, followed by an inverse square root decay till 50 epochs. With *AutoLR*, we trained the network with 8 explore epochs, and used a seed learning rate of 5e-5. In both cases we use the model checkpoint with least loss on the validation set for computing BLEU scores on the test set. Table 6 shows the training loss and test accuracy averaged over 3 runs. As shown, *AutoLR* achieves a mean test BLEU score of 34.88, compared to 34.70 for the baseline. Figure 4d shows the learning rate and validation perplexity of *AutoLR* against that of the baseline. See figure 12 for detailed comparisons of training/validation perplexity, learning rate, etc.

LR Schedule	Train ppl	Validation ppl	Test BLEU Score
Baseline	3.55 (0.029)	5.10 (0.033)	34.70 (0.001)
AutoLR	3.46 (0.16)	4.86 (0.014)	34.88 (0.005)

Table 6: Training, validation perplexity and test BLEU scores for IWSLT on Transformer networks. The test BLEU scores are computed on the checkpoint with the best validation perplexity. We report the mean and standard deviation over 3 runs.

4 RELATED WORK

There has been a lot of work recently on understanding the generalization characteristics of DNNs. Kawaguchi (2016) found that DNNs have many local minimas, but all local minimas are also the global minima. Also, it has been observed that wide minimas generalize much better than narrow minimas (Hochreiter & Schmidhuber (1997); Keskar et al. (2016); Jastrzebski et al. (2017); Wang et al. (2018)). Keskar et al. (2016) found that small batch SGD generalizes better and landed in wider minimas than large batch SGD. Interestingly however, more recent work was been able to generalize quite well even with very large batch sizes (Goyal et al. (2017); McCandlish et al. (2018); Shallue

[3] We used the implementation at: https://github.com/huggingface/pytorch-transformers

[4] We used the implementation at: https://github.com/pytorch/fairseq

et al. (2018)), by scaling the learning rate linearly as a function of the batch size. Jastrzebski et al. (2019) analyze how batch size and learning rate influence the curvature of not only the SGD endpoint but also the whole trajectory. They found that small batch or large step SGD have similar characteristics, and yield smaller and earlier peak of spectral norm as well as smaller largest eigenvalue. Nar & Sastry (2018) analyze constant learning rate gradient descent for deep linear networks, and show that step size determines the subset of local optima to which the algorithm can converge.

Many adaptive learning rate methods have been developed which adapt the learning rate on a per parameter basis, including AdaGrad (Duchi et al. (2011)), ADADELTA (Zeiler (2012)), RMSProp (Tieleman & Hinton (2012)), and Adam (Kingma & Ba (2014)). However, most of these methods still require specifying a global learning rate schedule which impacts the performance significantly. Many methods have been proposed to adaptively tune the global learning rate, such as Schaul et al. (2013a) which makes an idealized quadratic loss function assumption near the minima, Baydin et al. (2017) which uses gradient w.r.t the learning rate to update the learning rate in a first order fashion, and Rolinek & Martius (2018) which again uses a first order method along with an estimate of minimum loss achievable in a step. While Schaul et al. (2013a) and Baydin et al. (2017) compare to only constant learning rate baselines, Rolinek & Martius (2018) does compare to state-of-the-art learning rate schedule for Cifar-10, but is unable to match their test accuracy, although does better on train loss.

5 Conclusions

We have presented *AutoLR*, an *Explore-Exploit* method for auto learning rate tuning for training deep neural networks. Our *explore* phase is based on the hypothesis that narrow minimas far outnumber the wide minimas, and thus require some minimum exploration at a high learning rate to land in a wide minima region with high probability. We do multiple validations of this hypothesis, but also plan to further study it both theoretically and empirically in future work. Our *exploit* phase is based on a local quadratic approximation of loss in the search direction, as a function of perturbations in the learning rate. We extensively validate *AutoLR* on both image (ImageNet, Cifar-10, MNIST, FashionMNIST) and NLP (IWSLT, Squad) datasets on as well as multiple optimizers, and achieve or beat generalization accuracy of state of the art hand tuned learning rate schedules.

References

Atilim Gunes Baydin, Robert Cornish, David Martinez Rubio, Mark Schmidt, and Frank Wood. Online learning rate adaptation with hypergradient descent. *arXiv preprint arXiv:1703.04782*, 2017.

Mauro Cettolo, Jan Niehues, Sebastian Stüker, Luisa Bentivogli, and Marcello Federico. Report on the 11th iwslt evaluation campaign, iwslt 2014. In *Proceedings of the International Workshop on Spoken Language Translation, Hanoi, Vietnam*, pp. 57, 2014.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

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.

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT Press, 2016.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

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*, pp. 770–778, 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural Computation*, 9(1):1–42, 1997.

Stanisław Jastrzebski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017.

Stanisław Jastrzebski, Zachary Kenton, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amost Storkey. On the relation between the sharpest directions of DNN loss and the SGD step length. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=SkgEaj05t7.

Kenji Kawaguchi. Deep learning without poor local minima. In *Advances in neural information processing systems*, pp. 586–594, 2016.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

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

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

Kamil Nar and Shankar Sastry. Step size matters in deep learning. In *Advances in Neural Information Processing Systems*, pp. 3436–3444, 2018.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

Michal Rolinek and Georg Martius. L4: Practical loss-based stepsize adaptation for deep learning. In *Advances in Neural Information Processing Systems*, pp. 6433–6443, 2018.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In *International Conference on Machine Learning*, pp. 343–351, 2013a.

Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In *International Conference on Machine Learning*, pp. 343–351, 2013b.

Nicol N Schraudolph. Local gain adaptation in stochastic gradient descent. 1999.

Christopher J Shallue, Jaehoon Lee, Joe Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.

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.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Huan Wang, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. Identifying generalization properties in neural networks. *arXiv preprint arXiv:1809.07402*, 2018.

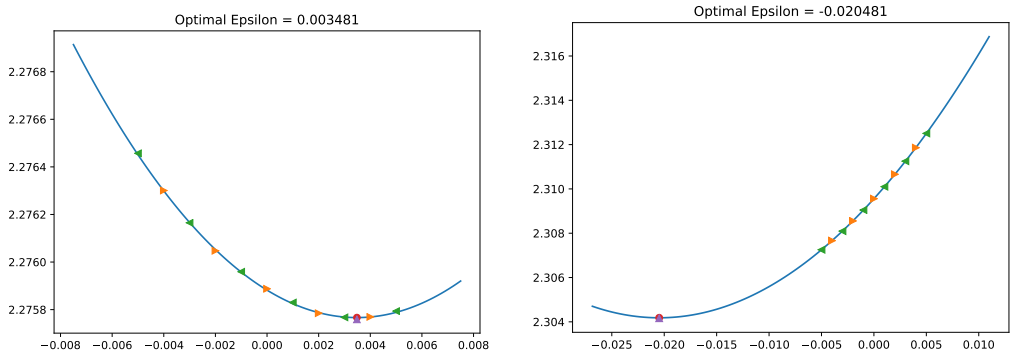
Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

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

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

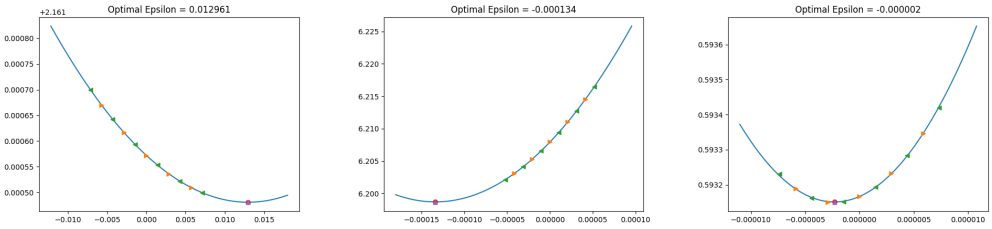
A VALIDATION OF QUADRATIC APPROXIMATION

We want to validate whether our quadratic approximation is a good approximation for modelling the loss as a function of the perturbation ϵ. Figure 5 shows a few examples demonstrating the effectiveness of second order approximation. As shown, the loss values at various samples overlap with the quadratic curve almost perfectly, including those not used in estimation of the quadratic (orange triangles). Also, the estimated loss value at the quadratic minima matches the true loss value there quite well. Figure 6 shows quadratic plots for more datasets/models.



(a) Minimum within range (b) Minimum outside range

Figure 5: Quadratic approximation of loss as a function of ϵ. Shown are two examples from Cifar-10 on Resnet-18 runs where the minimum is (a) within and (b) outside the range of loss samples. The orange triangles show loss samples used in fitting the quadratic, the blue line shows our quadratic approximation, and the green triangles show more loss samples which were not used for fitting. The red circle shows the minimum loss value as per the quadratic, while the purple triangle show the true value at that ϵ.



(a) ImageNet on Resnet-50 (b) IWSLT on Transformer (c) SQuAD fine-tuning on BERT

Figure 6: Quadratic approximation curve samples from (a) ImageNet on Resnet-50, (b) IWSLT on Transformer and (c) SQuAD fine-tuning on BERT. The legend is same as figure 5. It can be seen that the quadratic approximation works pretty well.

B SUPERBATCH SIZE SELECTION

To choose an appropriate superbatch size, we measure the standard deviation of loss as a function of superbatch size. We compute this by evaluating the loss with 10 different randomly sampled superbatches and calculate the standard deviation. Figure 7 shows the measurements. We used a conservative superbatch size of 100 in all our examples, as it corresponded to low variance. Note that higher superbatch sizes add an increased computational overhead, but since we recompute our learning rate infrequently the total overhead is not very high.

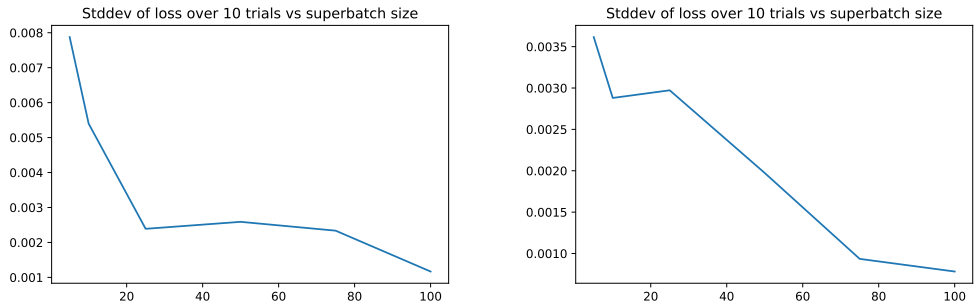


Figure 7: Standard deviation of loss as a function of superbatch size for (a) Cifar10 on Resnet-18, and (b) MNIST.

C Computational Cost

The primary computational cost of our method comes from computing the samples for quadratic approximation. We use a superbatch size of 100 and pick 5 samples for approximating our quadratic, thus incurring a cost of 500 forward passes each time we need to recompute the learning rate. Note that we recompute the learning rate only once in *recompute_window* steps. In most of our examples we keep the *recompute_window* such that our learning rate recomputation occurs 2 times each epoch, except in BERT fine-tuning where we only have 2-3 epochs to train, and thus want more frequent tuning of the learning rate. In Cifar-10 on Resnet-18 for example, we used a *recompute_window* of 1000 minibatch steps. A backward pass is typically 2x more computationally expensive than forward pass, and 1000 minibatch steps thus cost 3000 forward pass worth of compute. Our recomputation costs 500 forward passes of compute, and thus the computational overhead is $500/3000 \approx 16.6\%$. Note, however that in *exploit* phase, we do a recomputation of learning rate only when saturation threshold is crossed. So the computational cost comes down even further and 16.6% overhead for Cifar-10 is an upper bound.

Note that since the main focus of this work was to develop an automatic learning rate tuning scheme which generalizes as well as state of the art hand tuned learning rate schedules, we have not invested much effort on reducing the computational cost. For example, a 100 superbatch size is an overkill and a size of 50 should be suitable for most examples. Similarly, 3 samples for estimating the quadratic are enough most of the times instead of the 5 used, as the quadratic fit is very close mostly. Also, we can experiment with higher *recompute_window* sizes to reduce the cost further.

D Hyperparameters

Table 7 shows the hyperparameters used for all our examples. As shown, the saturation threshold and epsilon thresholds are pretty stable across all image examples, and the main parameter to tune is the explore epochs, which we typically set to around $20 - 30\%$ of the total budget.

The reason for different scale of epsilon thresholds in NLP datasets compared to image datasets, is because they typically run on much lower learning rates (of the order of $1e-5$) compared to image datasets (of the order of $1e-1$ to $1e-3$), which suggests that the optimization landscapes are more sensitive to smaller perturbations and thus need more aggressive clipping. We have some initial ideas on computing this threshold dynamically by looking at change in loss magnitude as we increase the ϵ, and choosing a threshold which limits this change magnitude to some amount. This can be computationally more expensive, as we need to search for the right epsilon, but can be done only few times in the each run.

We also noticed that the loss drop rate changes much more drastically in image datasets than NLP datasets, again most likely because of the very low learning rates used in NLP datasets. Thus we need to use a lower saturation threshold in NLP experiments compared to image ones. We found this hyperparameter very easy to set, by simply eyeballing at the loss drop rate changes of a trial run with fixed LR.

We are also looking at ways to automate the explore epochs hyperparameter, by looking at second order information about the loss landscape, etc. to determine if we have reached a wider minima region. See Jastrzebski et al. (2019) for some interesting analysis on this front.

Experiments	Explore Epochs / Total Epochs	Saturation Threshold	Epsilon Threshold
MNIST	10 / 50	100	1e-3
Fashion-MNIST	10 / 50	100	1e-3
Cifar-10	50 / 200	100	1e-3
Imagenet	25 / 90	500	1e-3
IWSLT'14 (De-En)	8 / 50	5	1e-9
SQuADv1.1 (Bert-Base)	0.45 / 2	2	1e-12

Table 7: Hyperparameters used for all experiments.

E More Results

E.1 MNIST

In this experiment we trained the MNIST dataset LeCun et al. (1998) on a 4 layer network with two convolution layers and two fully connected layers [5]. We evaluated our method on SGD, SGD with momentum and Adam optimizers. For baseline runs, we used a fixed learning rate of 0.1 for SGD, Momentum, and 1e-3 for Adam; and trained for 50 epochs. With *AutoLR*, we trained the network with 10 explore epochs, and used the same seed learning rate as baseline, i.e. 0.1 for SGD, Momentum, and 1e-3 for Adam. Table 8 shows the training loss and test accuracy for the various runs. See figures 14, 15, 16 for more detailed comparisons of training loss, test accuracy, and learning rate for SGD, Momentum and Adam optimizers, respectively.

LR Schedule	Optimizer		
	SGD	Momentum	Adam
Training Loss			
Baseline	2.3e-5 (9.4e-7)	7.2e-6 (4.2e-7)	4.5e-3 (1.3e-3)
AutoLR	2.4e-5 (5.5e-6)	7.3e-6 (7.7e-7)	3.9e-4 (9.5e-4)
Test Accuracy			
Baseline	99.32 (3.4e-4)	99.34 (6.0e-4)	99.21 (1.2e-3)
AutoLR	99.32 (3.7e-4)	99.38 (5.1e-4)	99.35 (7.2e-4)

Table 8: Training loss and Test accuracy for MNIST. We report the mean and standard deviation over 7 runs.

E.2 Fashion MNIST

In this experiment we trained the Fashion MNIST dataset Xiao et al. (2017) on the same 4 layer network as for the MNIST dataset. We evaluated our method on SGD, SGD with momentum and Adam optimizers. For baseline runs, we used a fixed learning rate of 0.1 for SGD, Momentum, and 1e-3 for Adam; and trained for 50 epochs. With *AutoLR*, we trained the network with 10 explore epochs, and used the same seed learning rate as baseline, i.e. 0.1 for SGD, Momentum, and 1e-3 for Adam. Table 9 shows the training loss and test accuracy for the various runs. See figures 17, 18, 19 for more detailed comparisons of training loss, test accuracy, and learning rate for SGD, Momentum and Adam optimizers, respectively.

F Detailed Plots

[5] We used the implementation at: https://github.com/pytorch/examples/tree/master/mnist

LR Schedule	Optimizer		
	SGD	Momentum	Adam
Training Loss			
Baseline	0.005 (0.006)	0.073 (0.007)	0.018 (0.003)
AutoLR	0.001 (0.001)	4.3e-4 (7.2e-4)	0.002 (0.006)
Test Accuracy			
Baseline	91.33 (0.005)	89.31 (0.003)	90.56 (0.003)
AutoLR	91.88 (0.001)	91.66 (0.002)	91.65 (0.003)

Table 9: Training loss and Test accuracy for FashionMNIST. We report the mean and standard deviation over 7 runs.

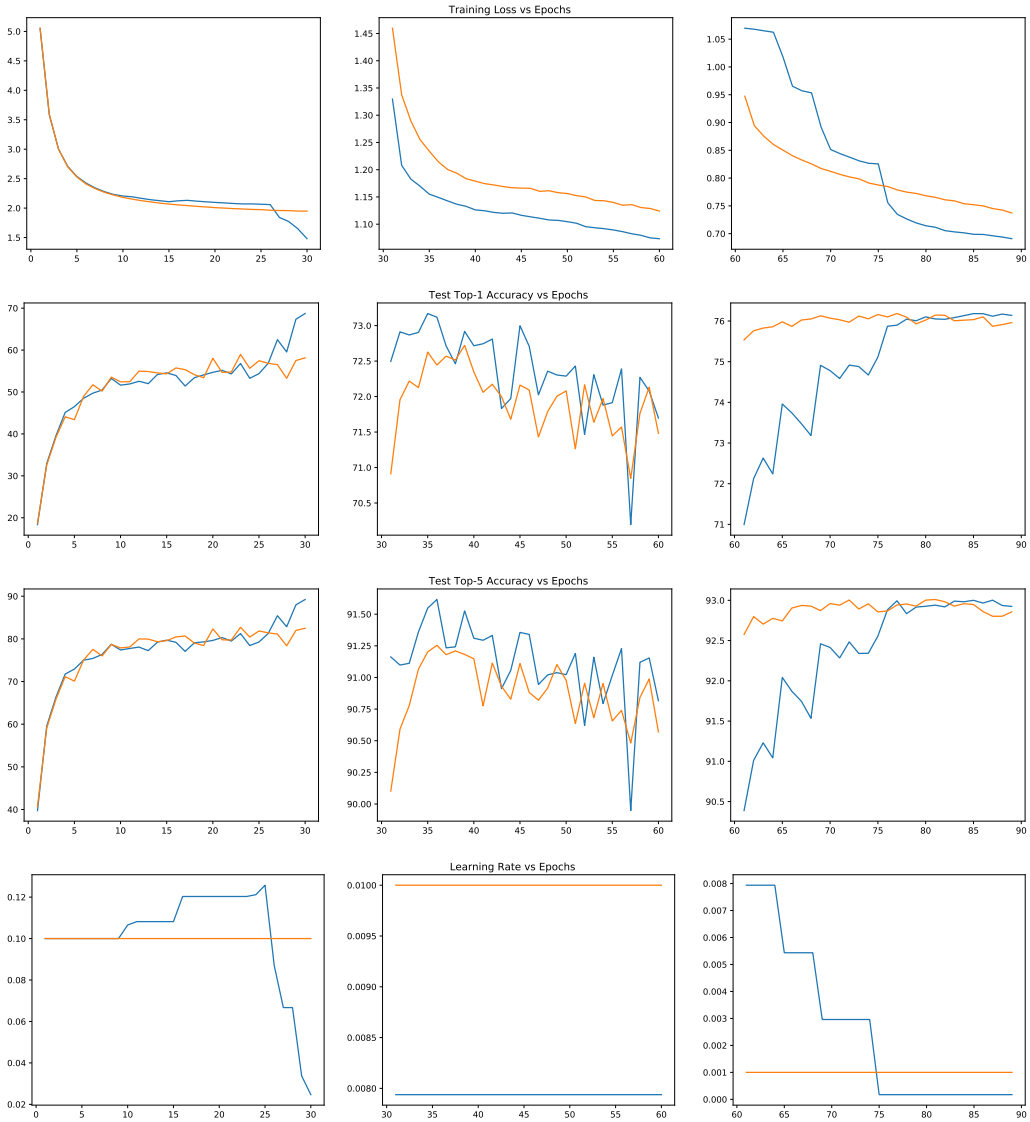


Figure 8: ImageNet on Resnet-50 trained with Momentum. Shown are the training loss, top-1/top-5 test accuracy and learning rate as a function of epochs, for the baseline scheme (orange) vs the *AutoLR* scheme (blue). The plot is split into 3 parts to permit higher fidelity in the y-axis range.

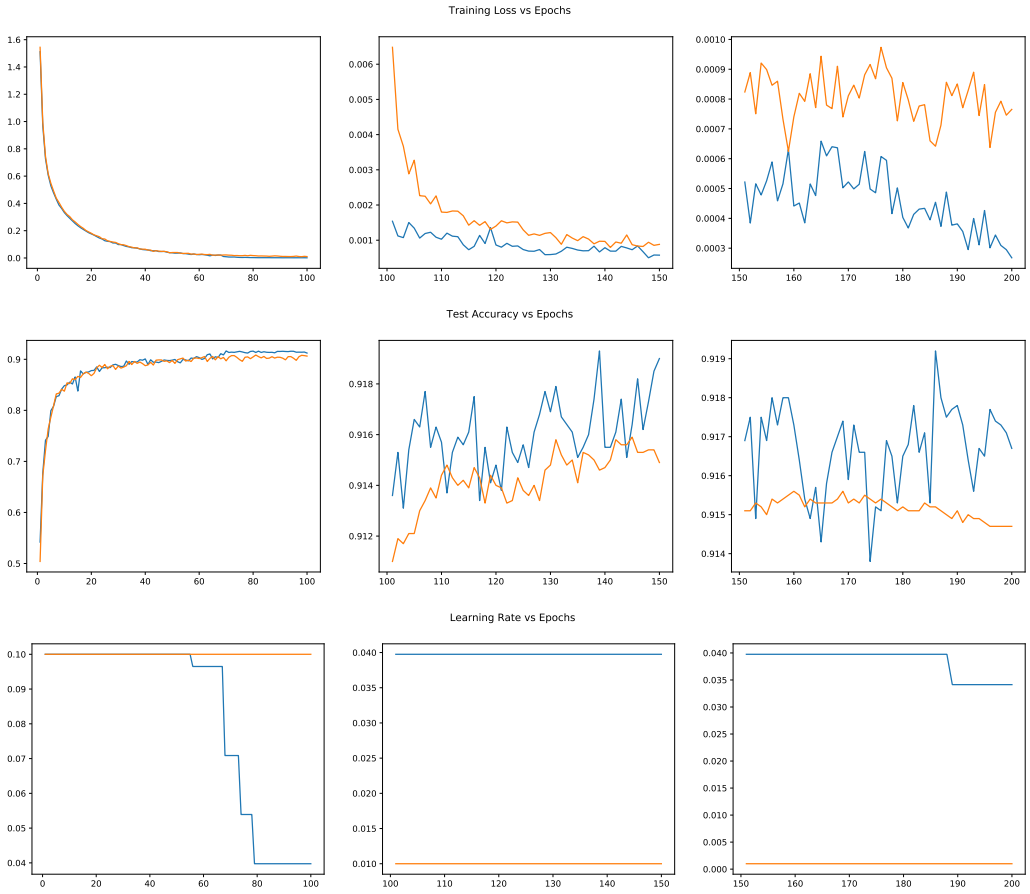


Figure 9: Cifar-10 on Resnet-18 trained with SGD. Shown are the training loss, test accuracy and learning rate as a function of epochs, for the baseline scheme (orange) vs the *AutoLR* scheme (blue). The plot is split into 3 parts to permit higher fidelity in the y-axis range.

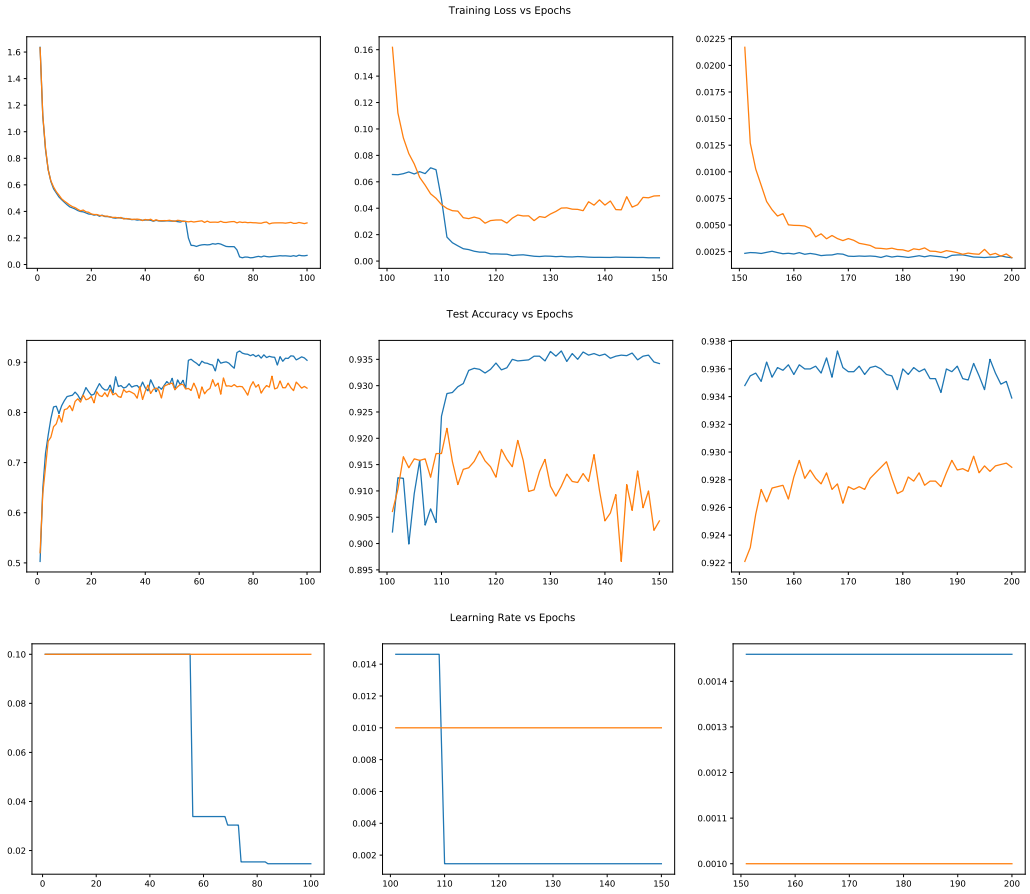


Figure 10: Cifar-10 on Resnet-18 trained with Momentum. Shown are the training loss, test accuracy and learning rate as a function of epochs, for the baseline scheme (orange) vs the *AutoLR* scheme (blue). The plot is split into 3 parts to permit higher fidelity in the y-axis range.

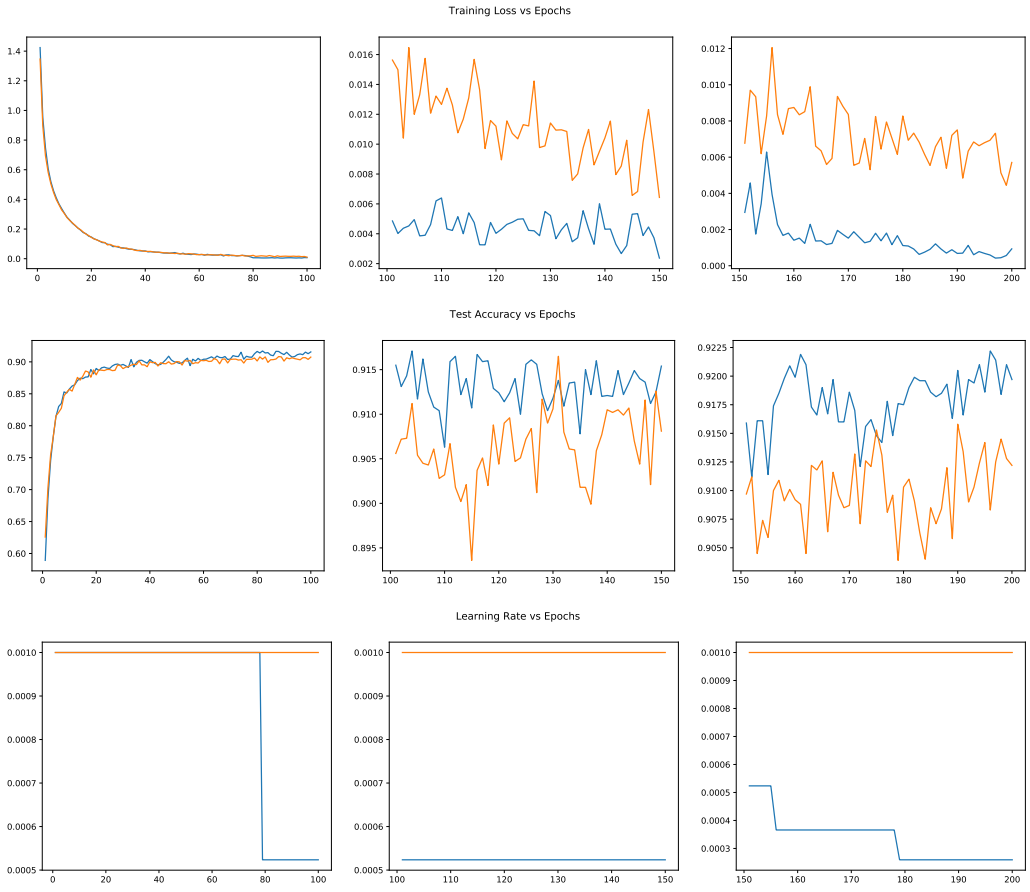


Figure 11: Cifar-10 on Resnet-18 trained with Adam. Shown are the training loss, test accuracy and learning rate as a function of epochs, for the baseline scheme (orange) vs the *AutoLR* scheme (blue). The plot is split into 3 parts to permit higher fidelity in the y-axis range.

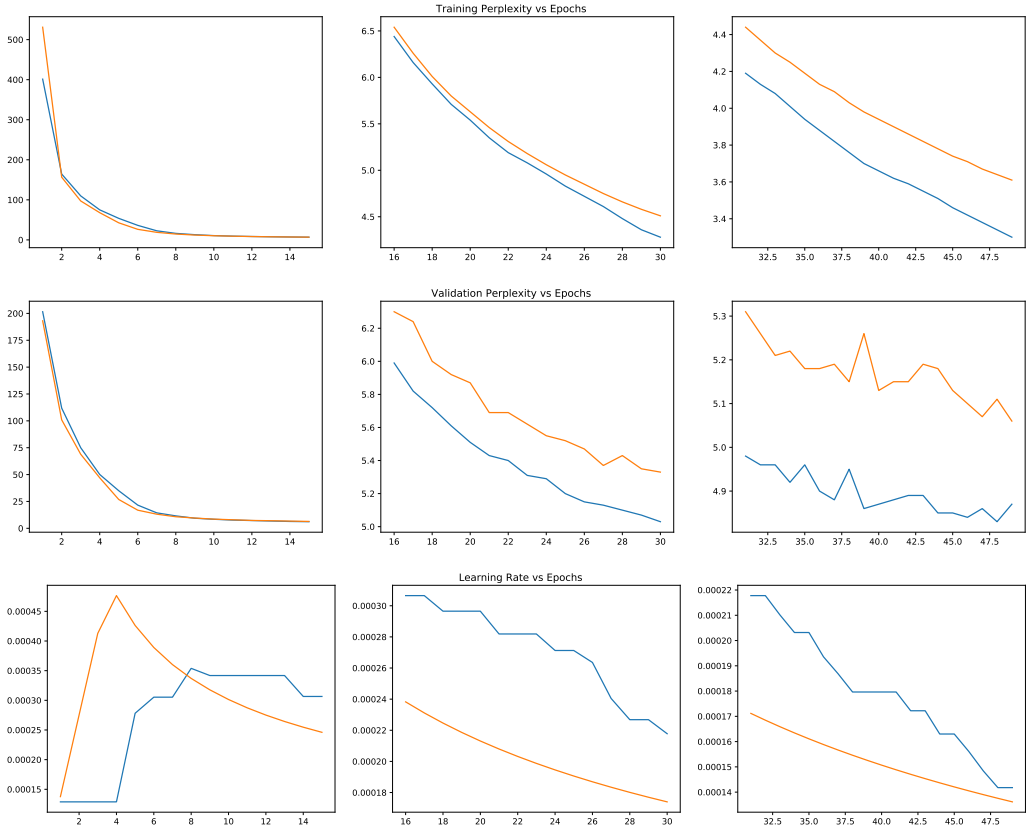


Figure 12: IWSLT on Transformer network trained with Adam. Shown are the training perplexity, validation perplexity and learning rate as a function of epochs, for the baseline scheme (orange) vs the *AutoLR* scheme (blue). The plot is split into 3 parts to permit higher fidelity in the y-axis range.

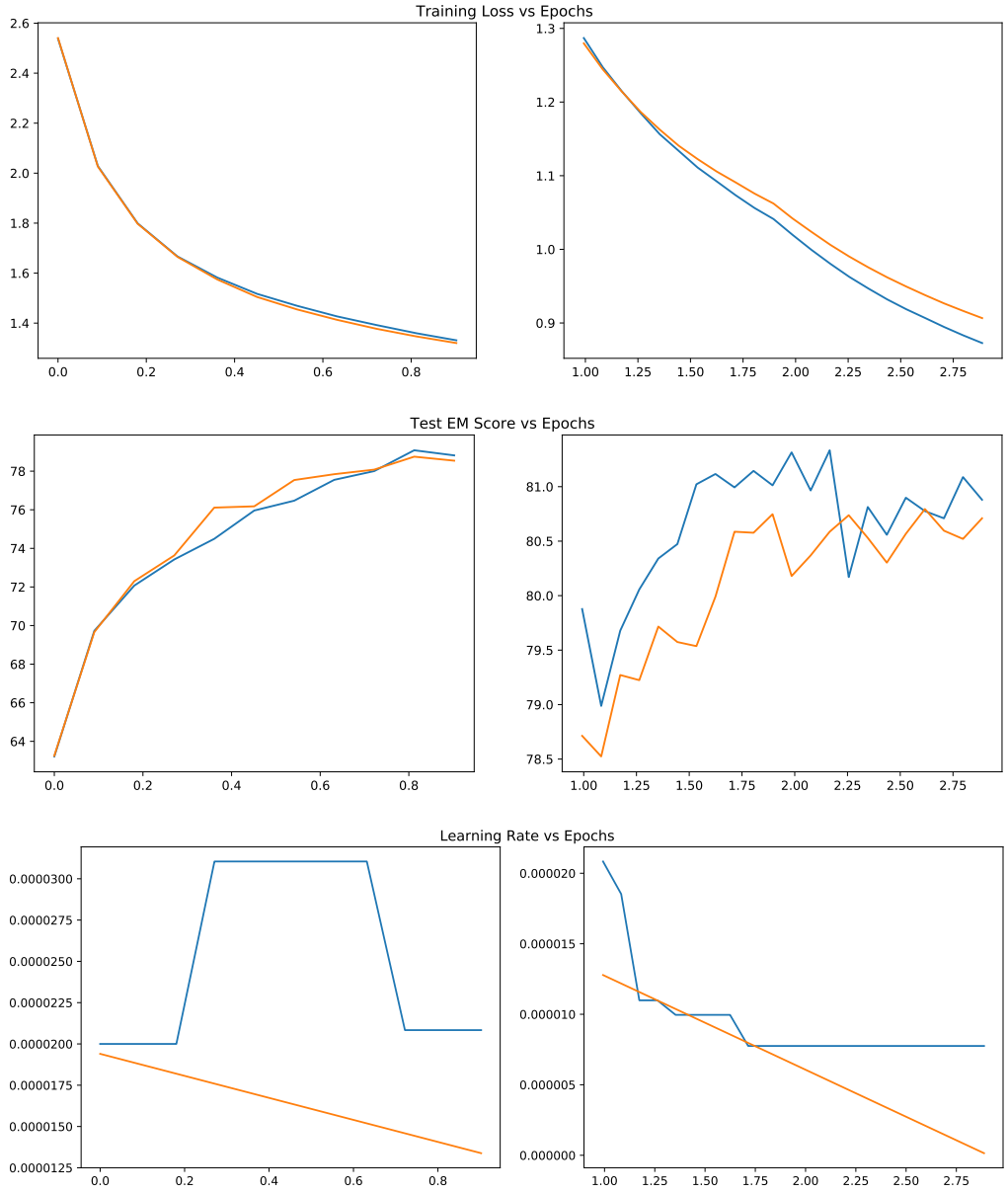


Figure 13: SQuAD fine-tuning on BERT trained with Adam. Shown are the training loss, test EM score, and learning rate as a function of epochs, for the baseline scheme (orange) vs the *AutoLR* scheme (blue). The plot is split into 2 parts to permit higher fidelity in the y-axis range. It is clear that with *AutoLR* the network starts to overfit after the 2nd epoch, where the testing loss continues to go down, but generalization suffers. We saw similar behavior with different seeds, and thus need to train with *AutoLR* for only 2 epochs.

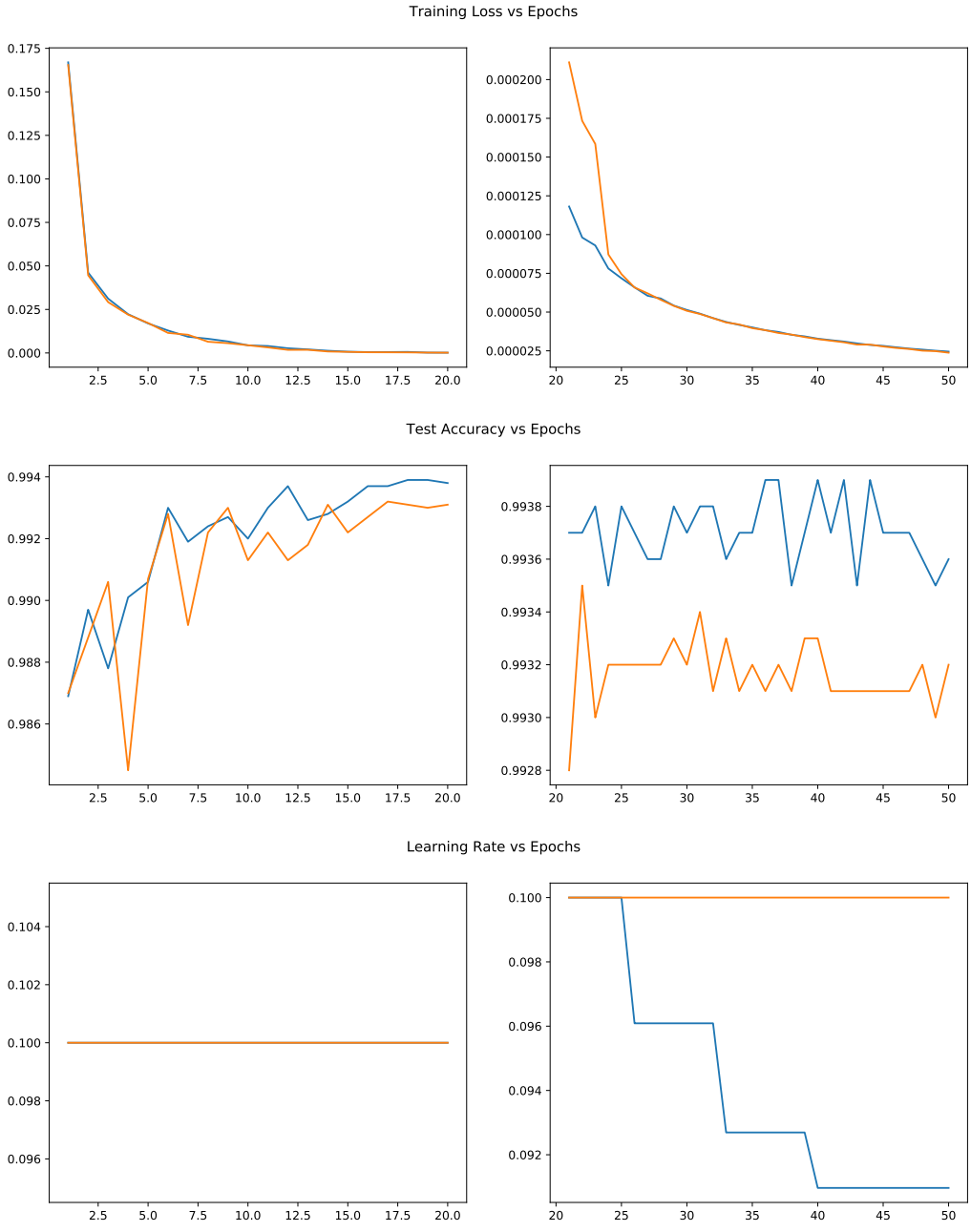


Figure 14: MNIST on Resnet-18 trained with SGD. Shown are the training loss, test accuracy and learning rate as a function of epochs, for the baseline scheme (orange) vs the *AutoLR* scheme (blue). The plot is split into 2 parts to permit higher fidelity in the y-axis range.

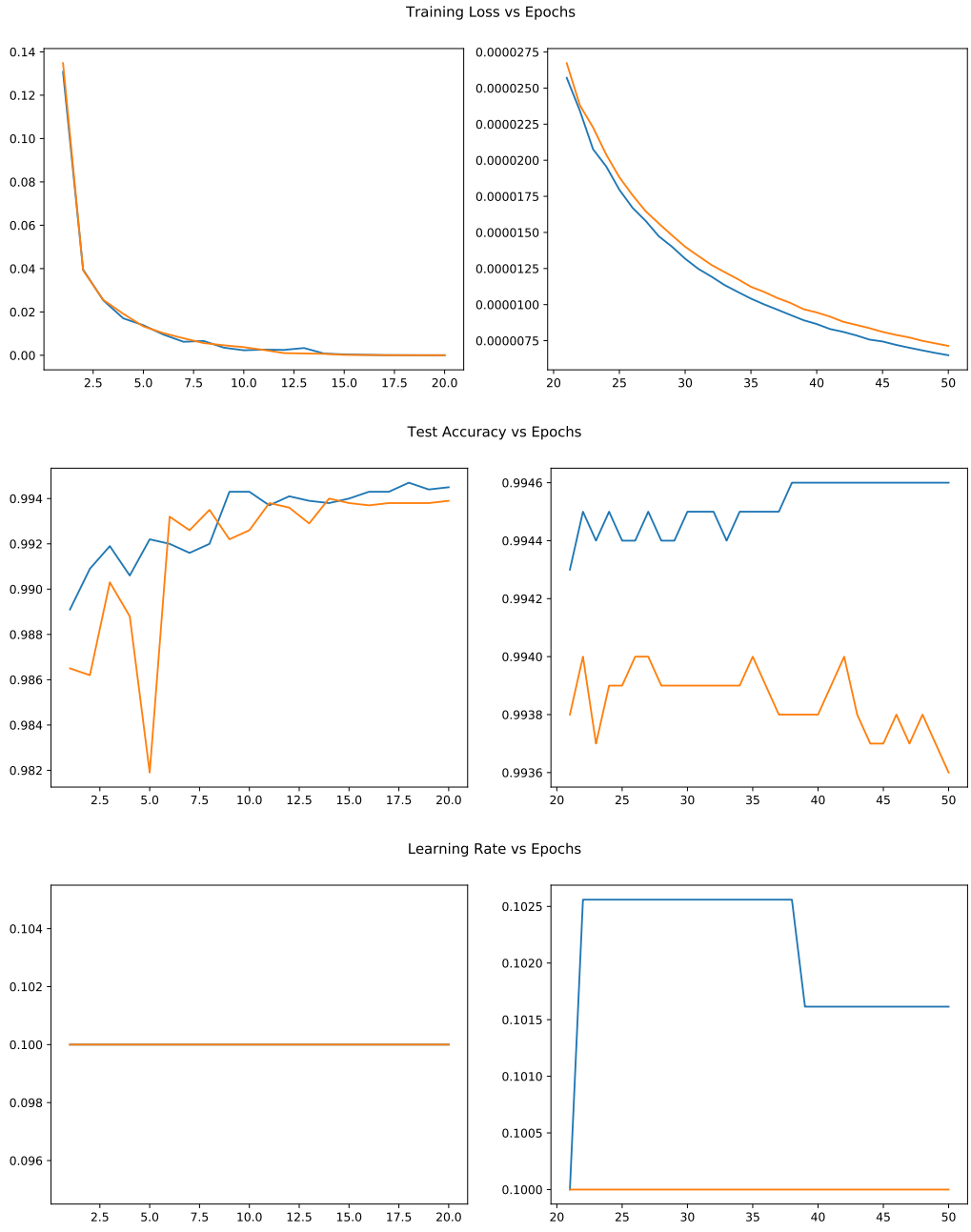


Figure 15: MNIST on Resnet-18 trained with Momentum. Shown are the training loss, test accuracy and learning rate as a function of epochs, for the baseline scheme (orange) vs the *AutoLR* scheme (blue). The plot is split into 2 parts to permit higher fidelity in the y-axis range.

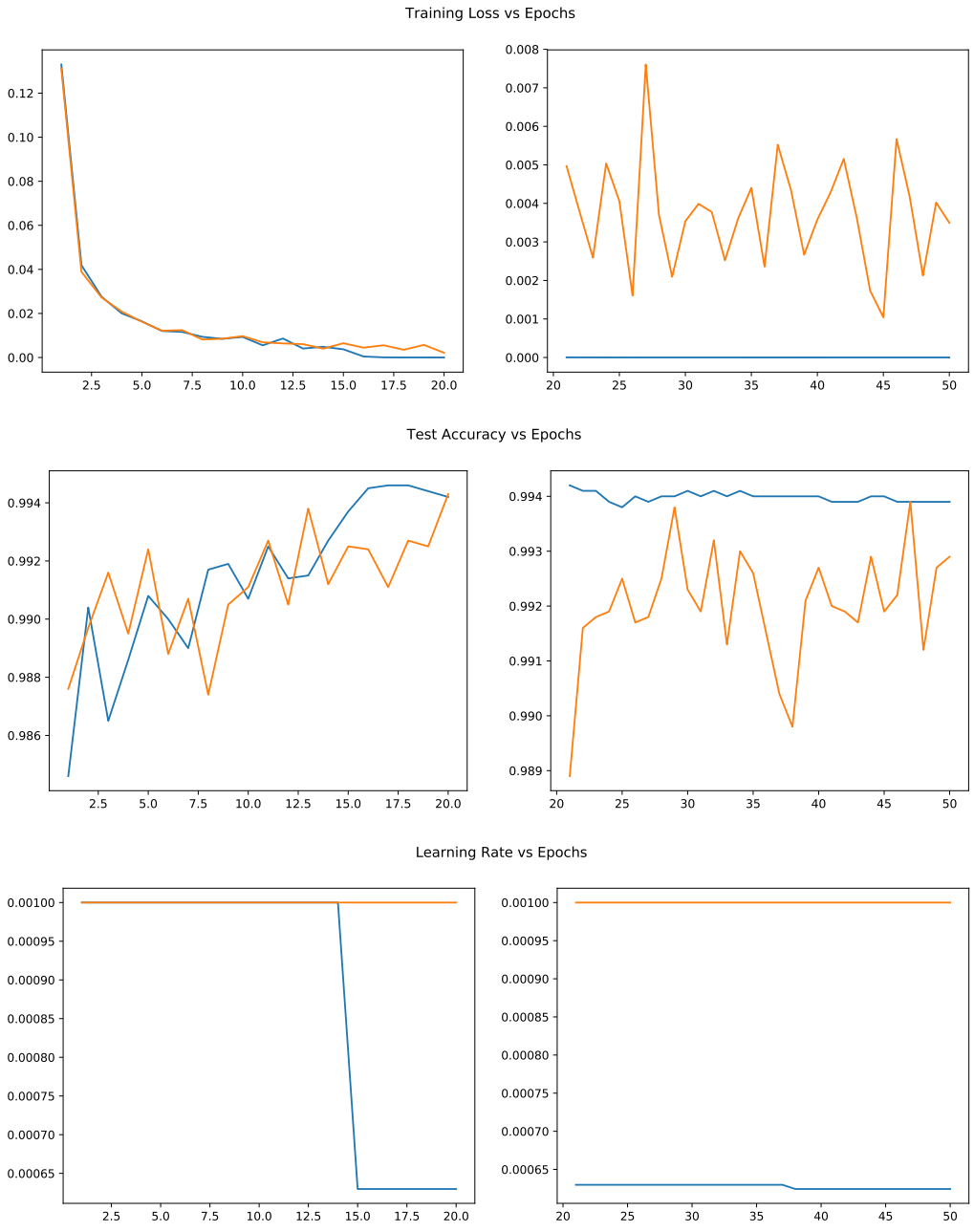


Figure 16: MNIST on Resnet-18 trained with Adam. Shown are the training loss, test accuracy and learning rate as a function of epochs, for the baseline scheme (orange) vs the *AutoLR* scheme (blue). The plot is split into 2 parts to permit higher fidelity in the y-axis range.

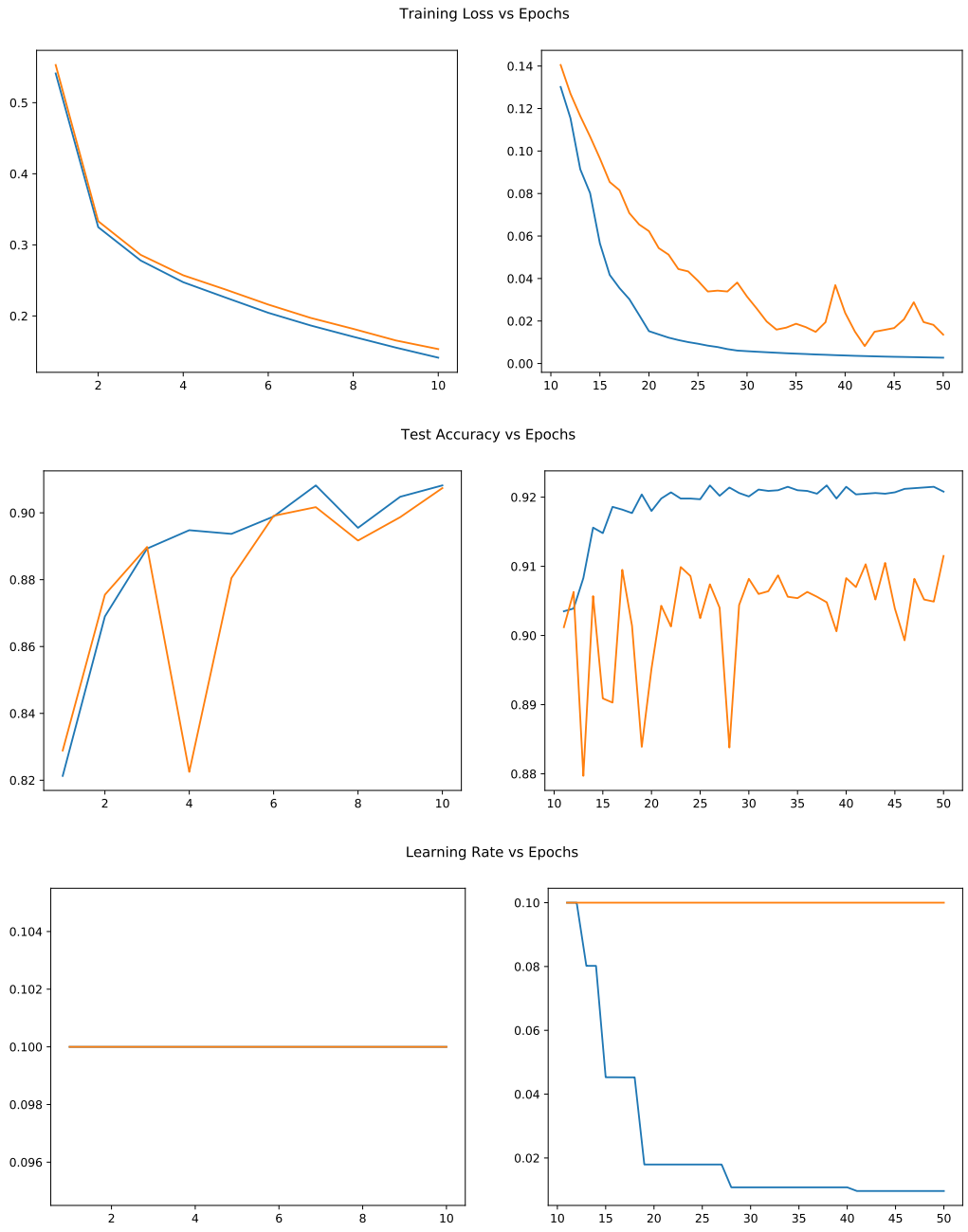


Figure 17: Fashion MNIST trained with SGD. Shown are the training loss, test accuracy and learning rate as a function of epochs, for the baseline scheme (orange) vs the *AutoLR* scheme (blue). The plot is split into 2 parts to permit higher fidelity in the y-axis range.

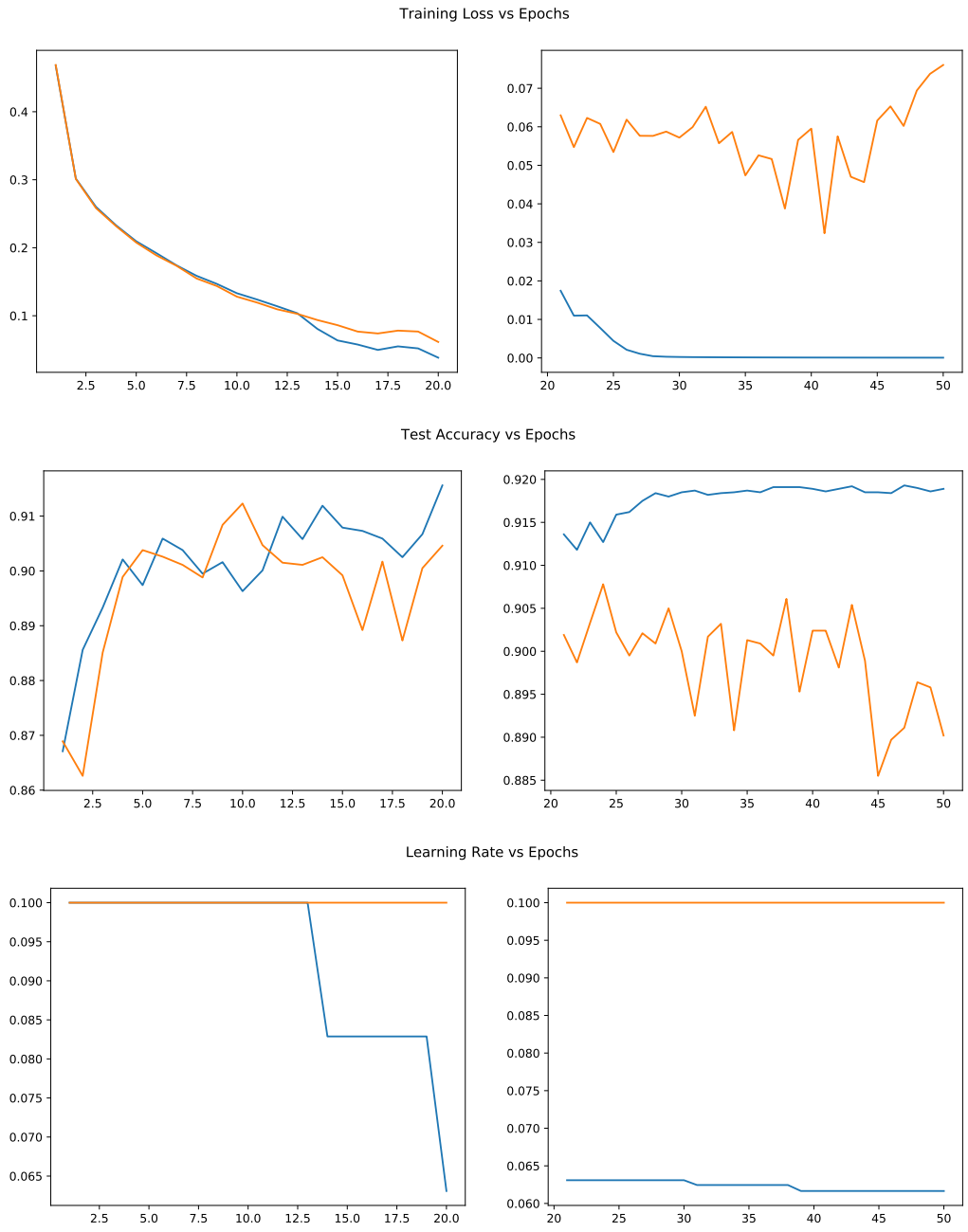


Figure 18: Fashion MNIST trained with Momentum. Shown are the training loss, test accuracy and learning rate as a function of epochs, for the baseline scheme (orange) vs the *AutoLR* scheme (blue). The plot is split into 2 parts to permit higher fidelity in the y-axis range.

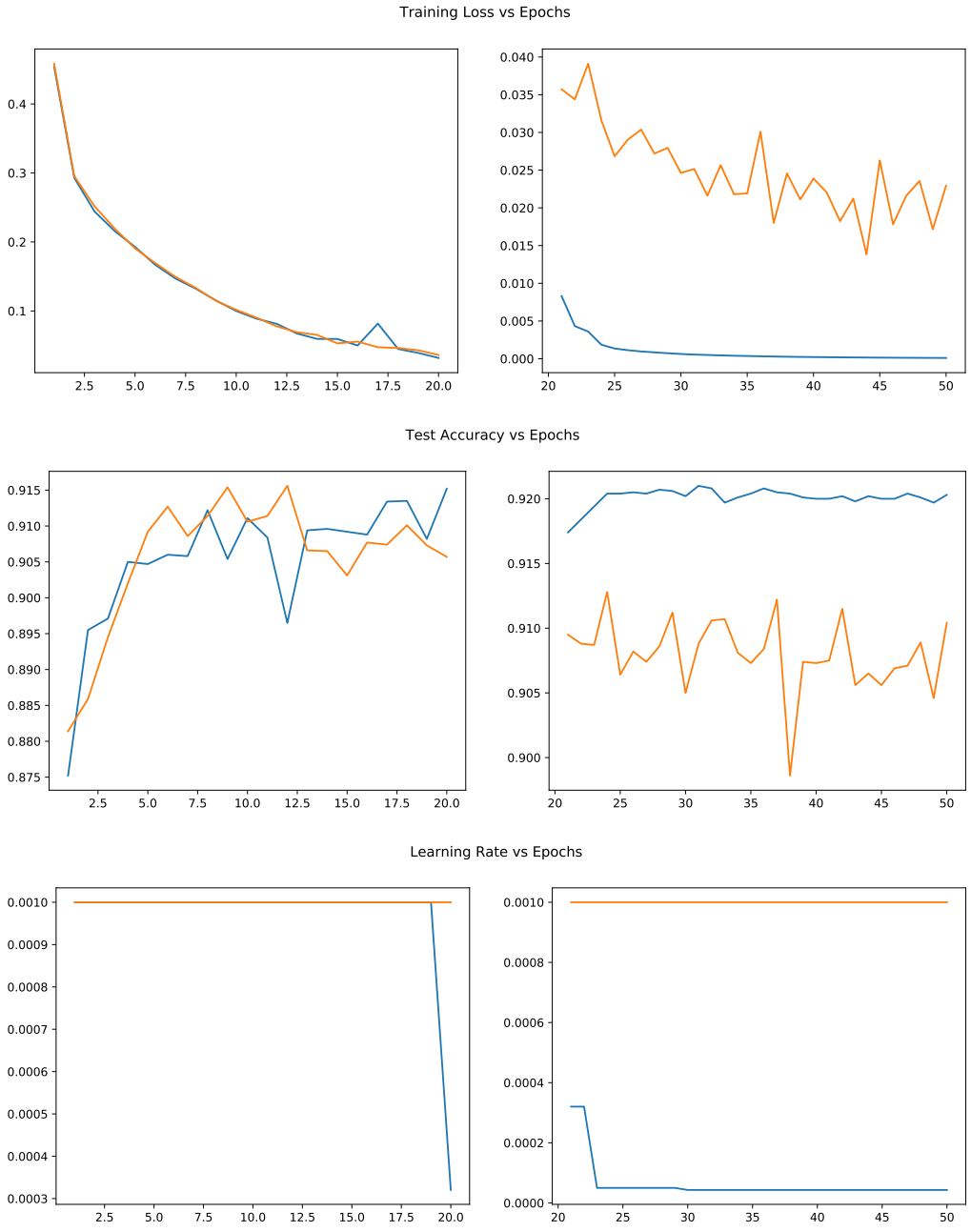


Figure 19: Fashion MNIST trained with Adam. Shown are the training loss, test accuracy and learning rate as a function of epochs, for the baseline scheme (orange) vs the *AutoLR* scheme (blue). The plot is split into 2 parts to permit higher fidelity in the y-axis range.