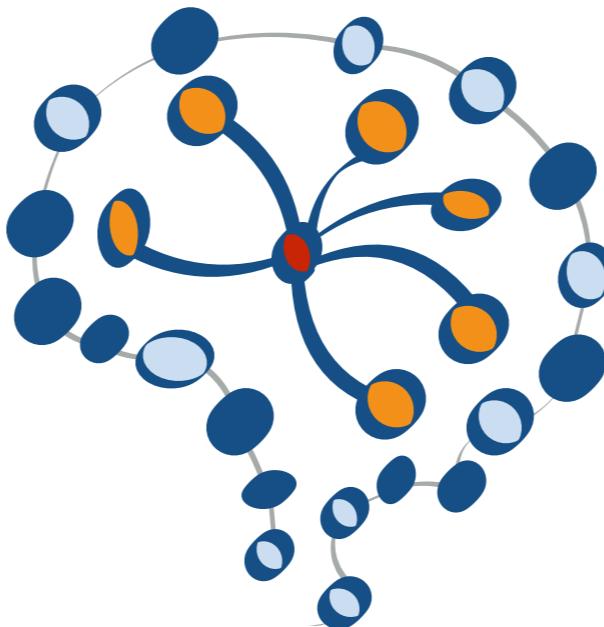


STAT 453: Introduction to Deep Learning and Generative Models

Sebastian Raschka

<http://stat.wisc.edu/~sraschka/teaching>



Lecture 12

Improving Gradient Descent-based Optimization

Overview: Additional Tricks for Neural Network Training (Part 2/2)

Part 1 (Last Lecture, L10)

- Input Normalization & BatchNorm
- Weight Initialization (Xavier Glorot, Kaiming He)

Part 2 (this lecture)

- Learning Rate Decay
- Momentum Learning
- Adaptive Learning



Overview: Additional Tricks for Neural Network Training (Part 2/2)

Part 1 (Last Lecture, L10)

- Input Normalization & BatchNorm
- Weight Initialization (Xavier Glorot, Kaiming He)

Part 2 (this lecture)

- Learning Rate Decay
- Momentum Learning
- Adaptive Learning

(Modifications of the 1st order SGD optimization algorithm; 2nd order methods are rarely used in DL)



Lecture Overview

1. Learning rate decay
2. Learning rate schedulers in PyTorch
3. Training with "momentum"
4. ADAM: Adaptive learning rates & momentum
5. Using optimization algorithms in PyTorch
6. Optimization in deep learning: Additional topics

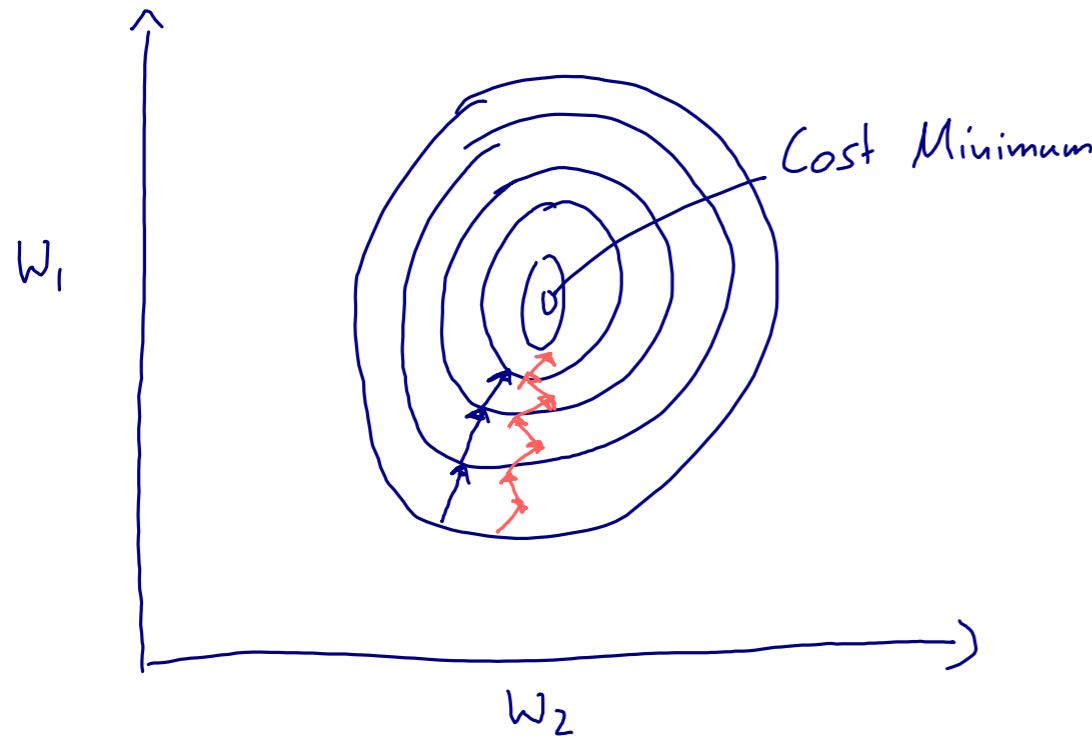


Decreasing the learning rate over the course of training

1. **Learning rate decay**
2. Learning rate schedulers in PyTorch
3. Training with "momentum"
4. ADAM: Adaptive learning rates & momentum
5. Using optimization algorithms in PyTorch
6. Optimization in deep learning: Additional topics



Minibatch Learning Recap

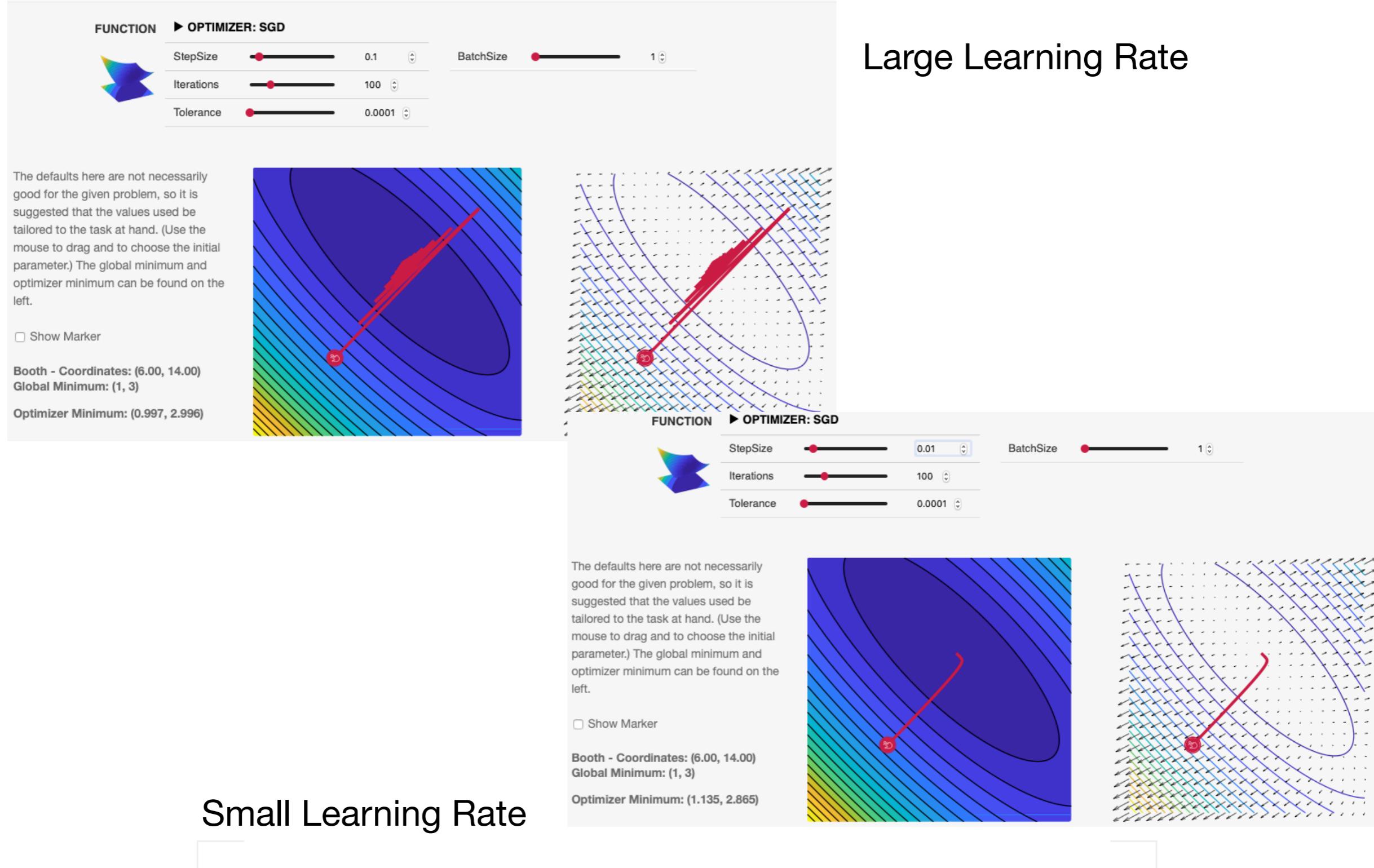


- Main advantage: Convergence speed, because it offers opportunities for parallelism (do you recall what these are?)

- Minibatch learning is a form of stochastic gradient descent
- Each minibatch can be considered a sample drawn from the training set (where the training set is in turn a sample drawn from the population)
- Hence, the gradient is noisier
 - ◆ good: chance to escape local minima
 - ◆ bad: can lead to extensive oscillation

Nice Library & Visualization Tool

<https://vis.ensmallen.org>



Practical Tip for Minibatch Use

- Reasonable minibatch sizes are usually: 32, 64, 128, 256, 512, 1024 (in the last lecture, we discussed why powers of 2 are a common convention)
- Usually, you can choose a batch size that is as large as your GPU memory allows (matrix-multiplication and the size of fully-connected layers are usually the bottleneck)
- Practical tip: usually, it is a good idea to also make the batch size proportional to the number of classes in the dataset

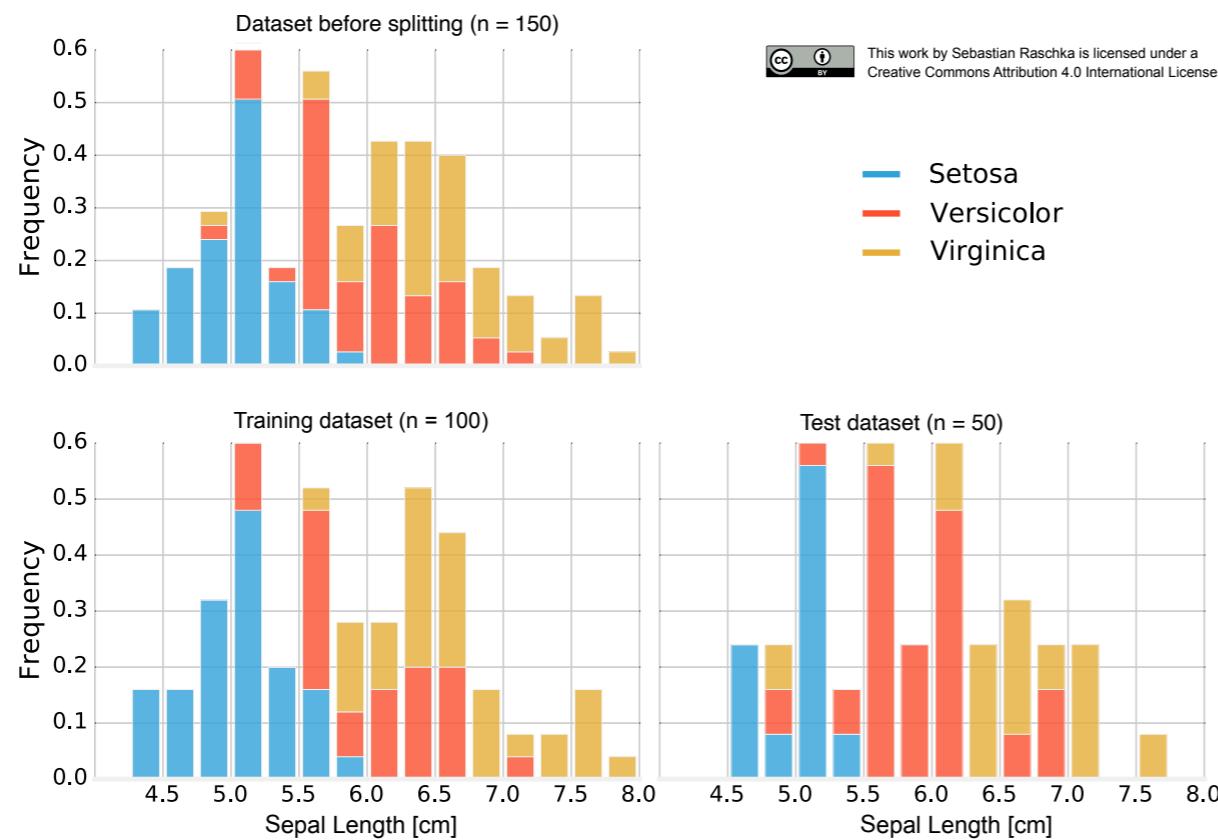
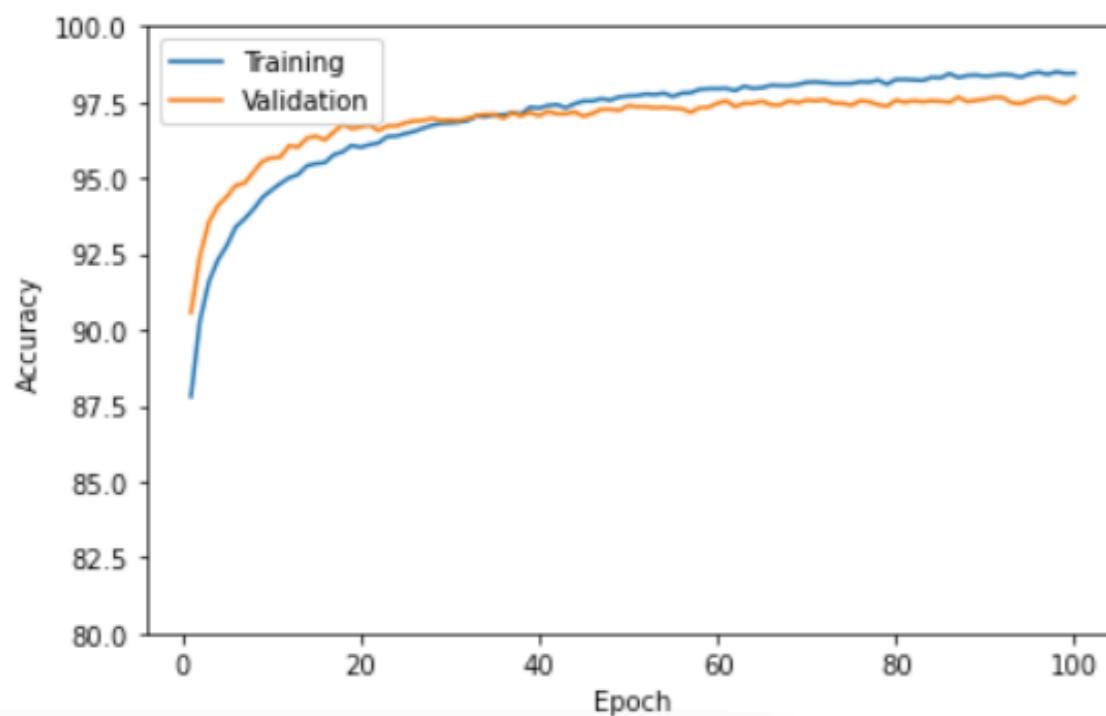
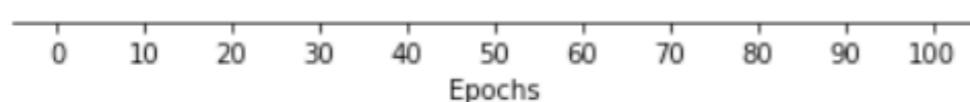
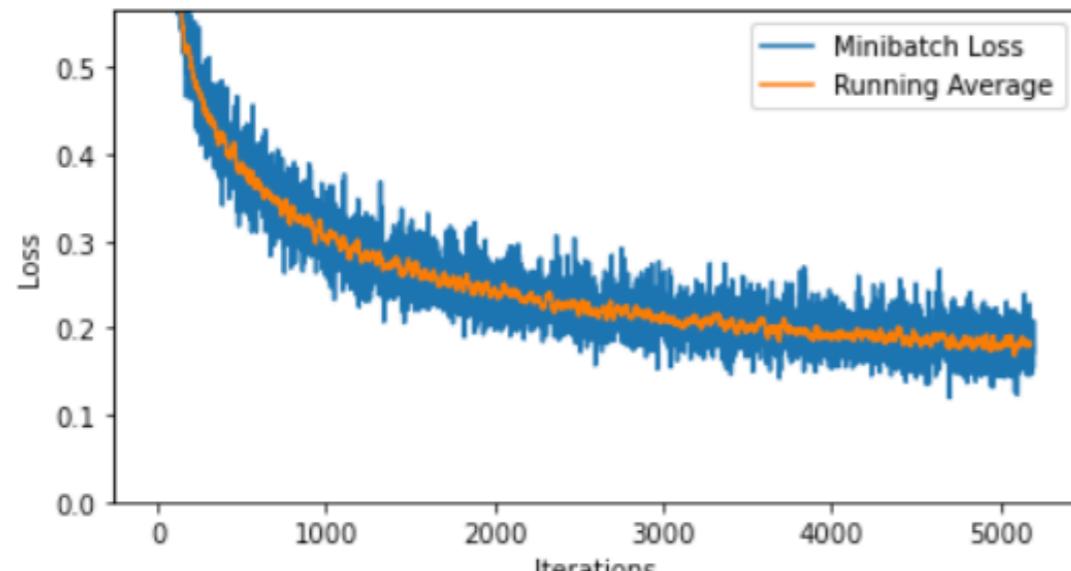


Figure 1: Distribution of *Iris* flower classes upon random subsampling into training and test sets.

Raschka, S. (2018). Model evaluation, model selection, and algorithm selection in machine learning.
<https://arxiv.org/abs/1811.12808>

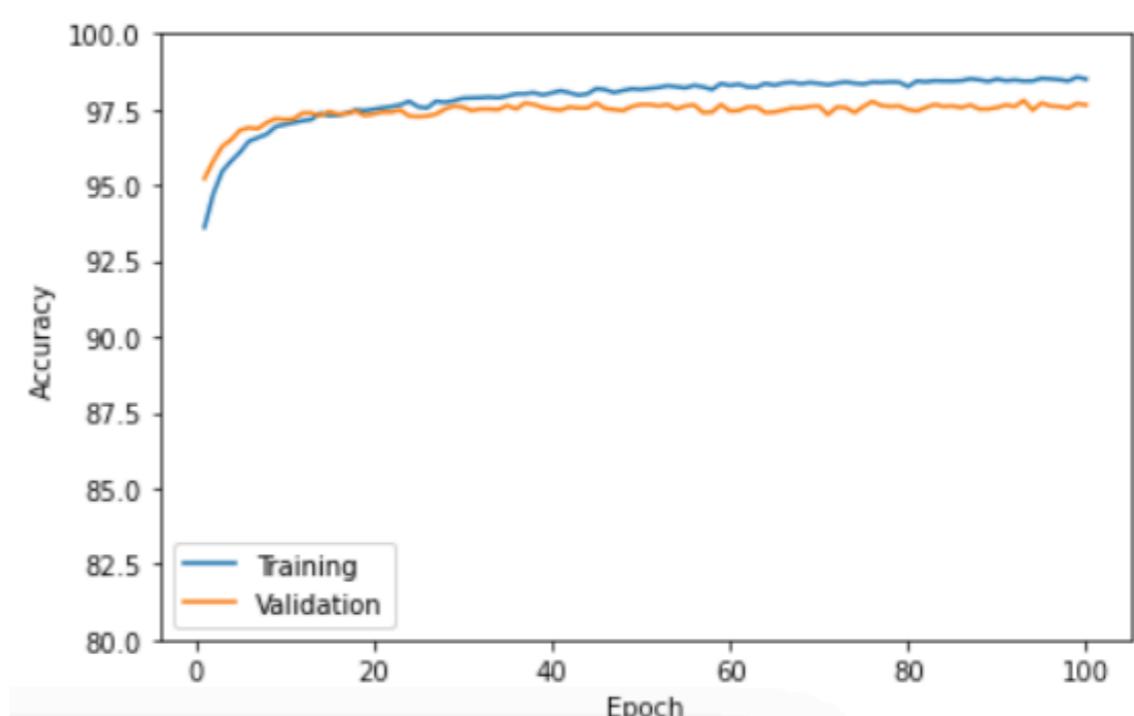
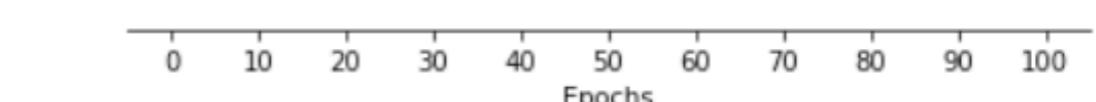
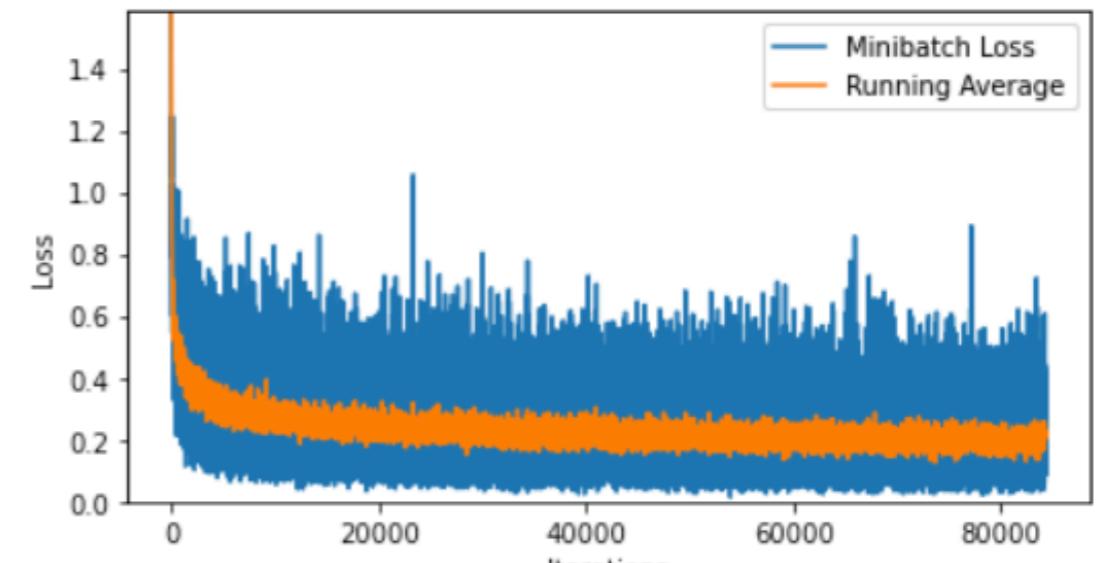
batchsize-1024.ipynb

Epoch: 100/100 | Train: 98.45% | Validation: 97.67%
Time elapsed: 4.38 min
Total Training Time: 4.38 min
Test accuracy 97.08%



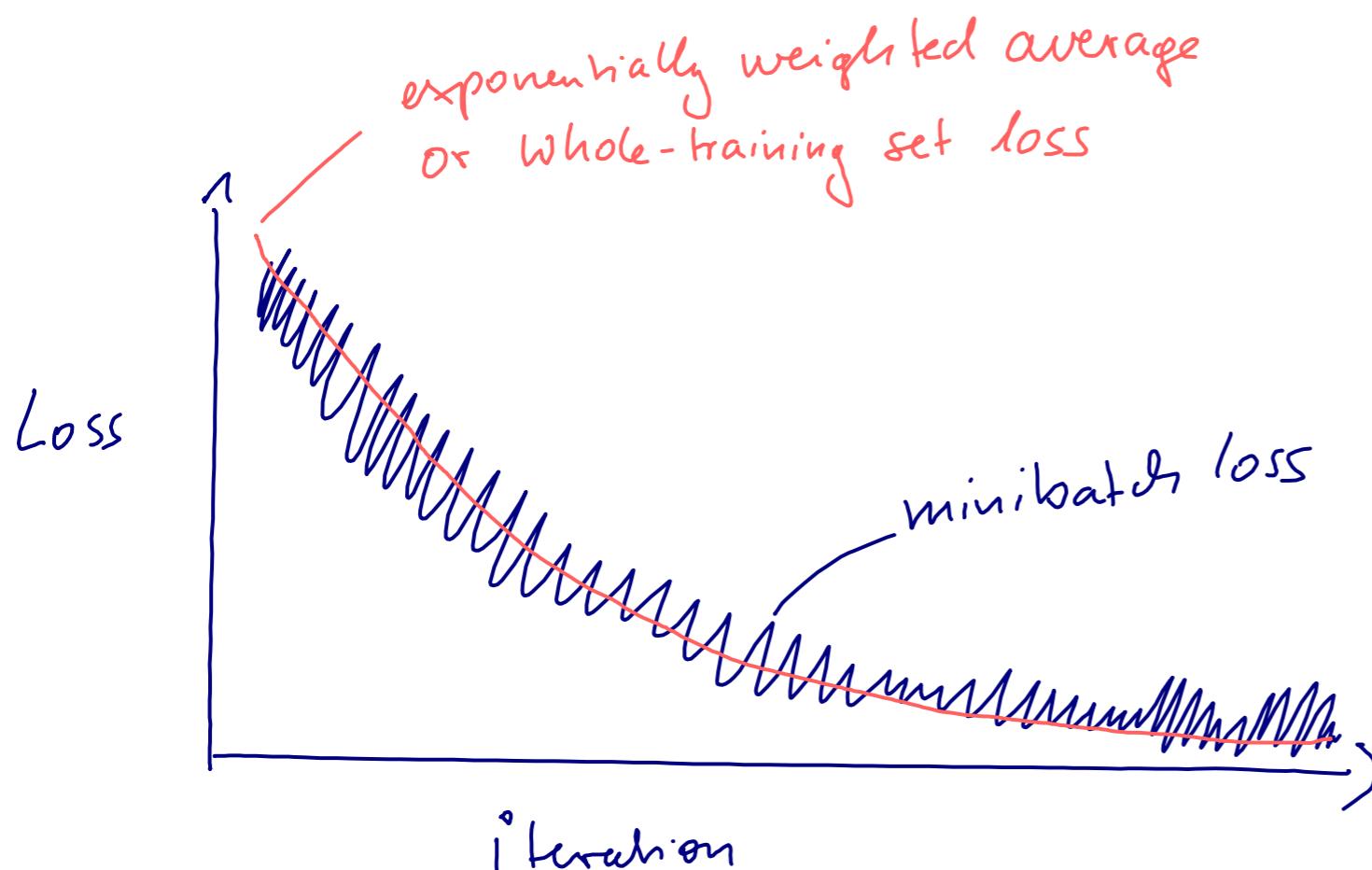
batchsize-64.ipynb

Epoch: 100/100 | Train: 98.50% | Validation: 97.65%
Time elapsed: 5.59 min
Total Training Time: 5.59 min
Test accuracy 97.18%



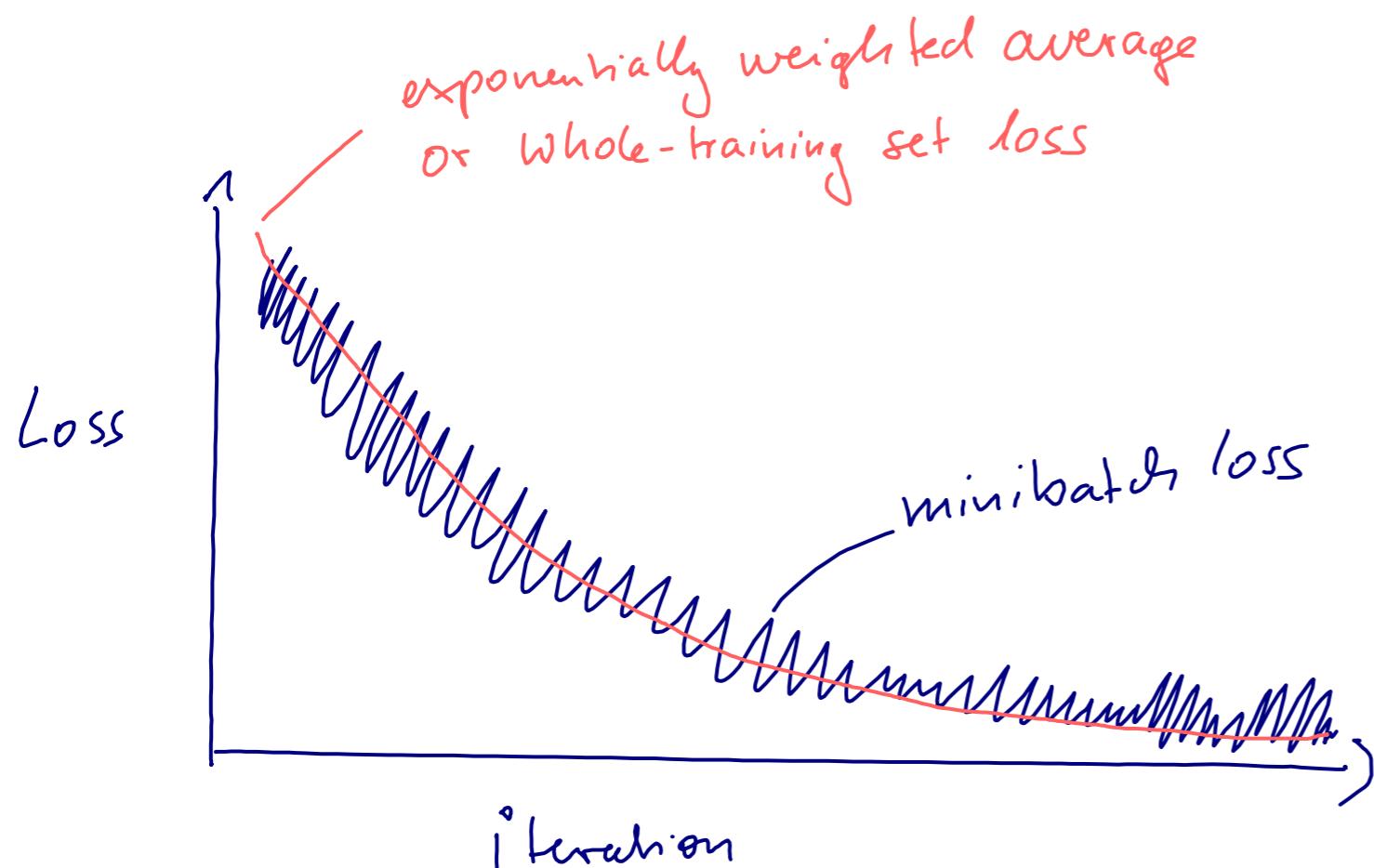
Learning Rate Decay

- Batch effects -- minibatches are samples of the training set, hence minibatch loss and gradients are approximations
- Hence, we usually get oscillations
- To dampen oscillations towards the end of the training, we can decay the learning rate



Learning Rate Decay

- Batch effects -- minibatches are samples of the training set, hence minibatch loss and gradients are approximations
- Hence, we usually get oscillations
- To dampen oscillations towards the end of the training, we can decay the learning rate



Danger of learning rate is to decrease the learning rate too early

Practical tip: try to train the model without learning rate decay first, then add it later

You can also use the validation performance (e.g., accuracy) to judge whether lr decay is useful (as opposed to using the training loss)

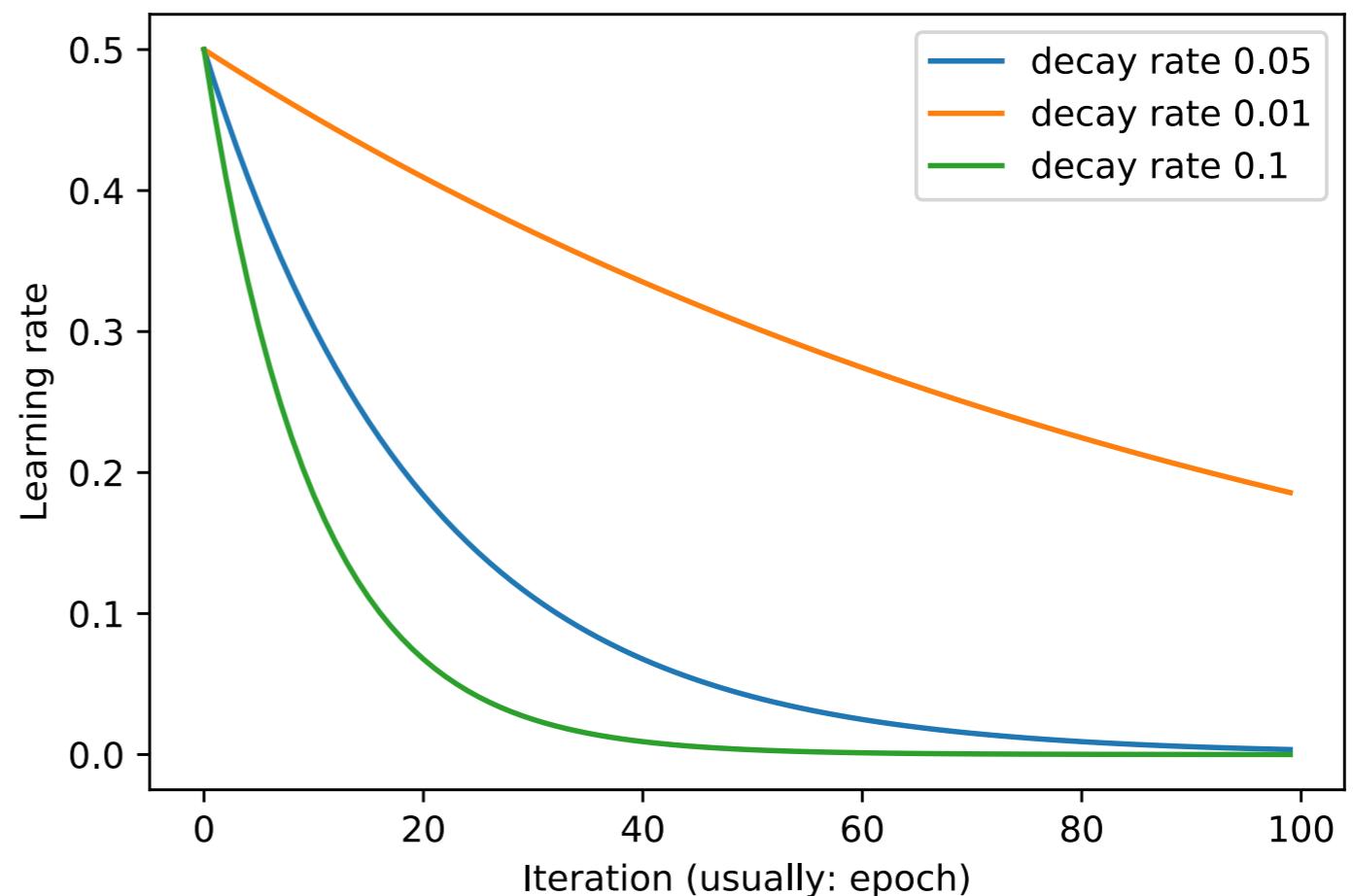
Learning Rate Decay

Most common variants for learning rate decay:

1) Exponential Decay:

$$\eta_t := \eta_0 \cdot e^{-k \cdot t}$$

where k is the decay rate



Learning Rate Decay

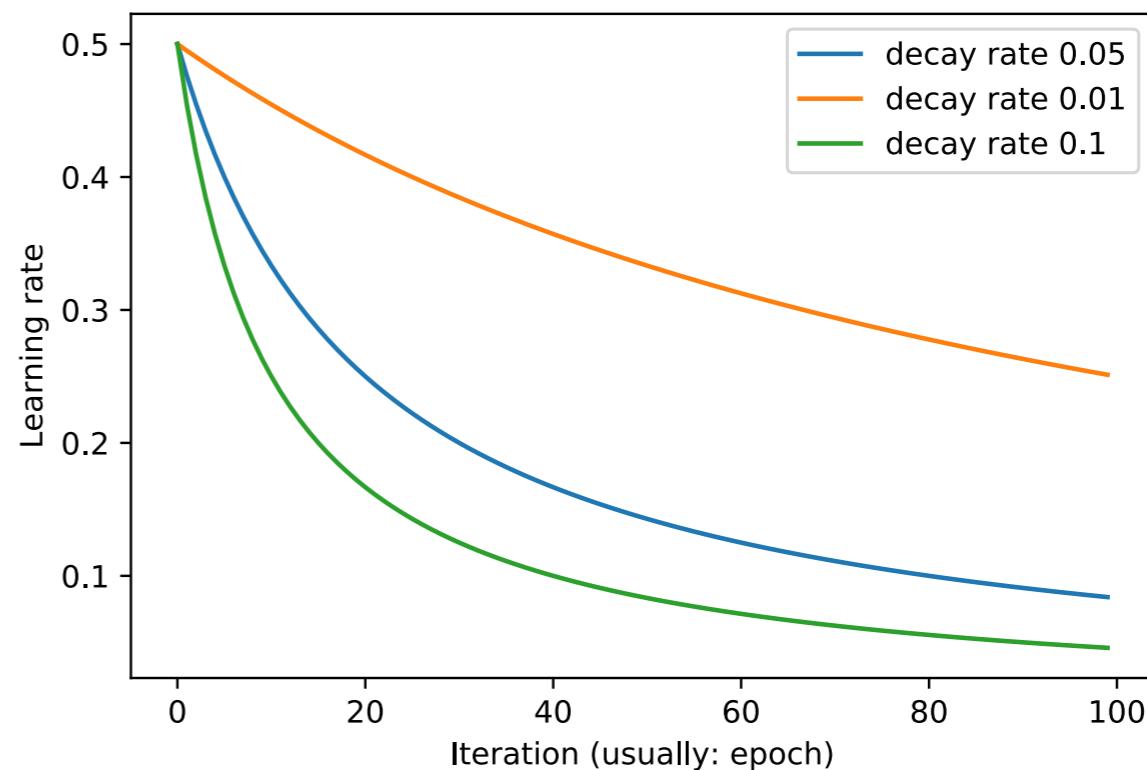
Most common variants for learning rate decay:

2) Halving the learning rate:

$$\eta_t := \eta_{t-1}/2$$

3) Inverse decay:

$$\eta_t := \frac{\eta_0}{1 + k \cdot t}$$



Learning Rate Decay

There are many, many more

E.g., Cyclical Learning Rate

Smith, Leslie N. “[Cyclical learning rates for training neural networks](#).” Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on. IEEE, 2017.

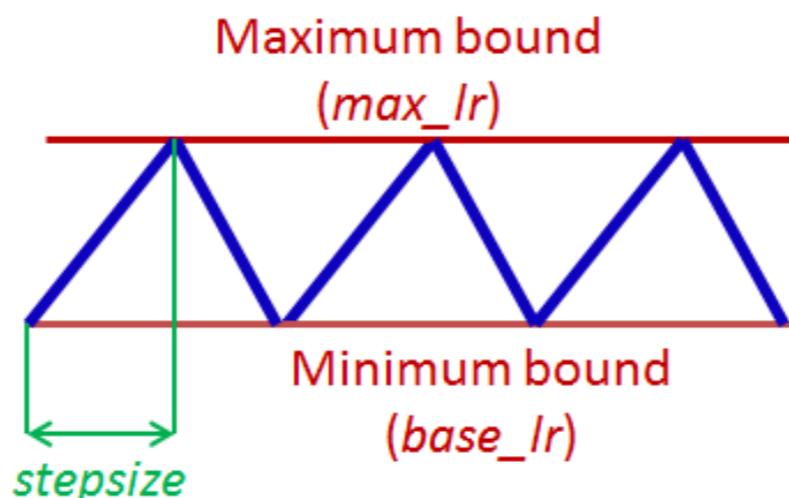


Figure 2. Triangular learning rate policy. The blue lines represent learning rate values changing between bounds. The input parameter *stepsize* is the number of iterations in half a cycle.

Relationship between Learning Rate and Batch Size

DON'T DECAY THE LEARNING RATE,
INCREASE THE BATCH SIZE

Samuel L. Smith*, Pieter-Jan Kindermans*, Chris Ying & Quoc V. Le

Google Brain

{slsmith, pikinder, chrisying, qvl}@google.com

ABSTRACT

It is common practice to decay the learning rate. Here we show one can usually obtain the same learning curve on both training and test sets by instead increasing the batch size during training. This procedure is successful for stochastic gradient descent (SGD), SGD with momentum, Nesterov momentum, and Adam. It reaches equivalent test accuracies after the same number of training epochs, but with fewer parameter updates, leading to greater parallelism and shorter training times. We can further reduce the number of parameter updates by increasing the learning rate ϵ and scaling the batch size $B \propto \epsilon$. Finally, one can increase the momentum coefficient m and scale $B \propto 1/(1 - m)$, although this tends to slightly reduce the test accuracy. Crucially, our techniques allow us to repurpose existing training schedules for large batch training with no hyper-parameter tuning. We train ResNet-50 on ImageNet to 76.1% validation accuracy in under 30 minutes.

Smith, S. L., Kindermans, P. J., Ying, C., & Le, Q. V. (2017). Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*.

Relationship between Learning Rate and Batch Size

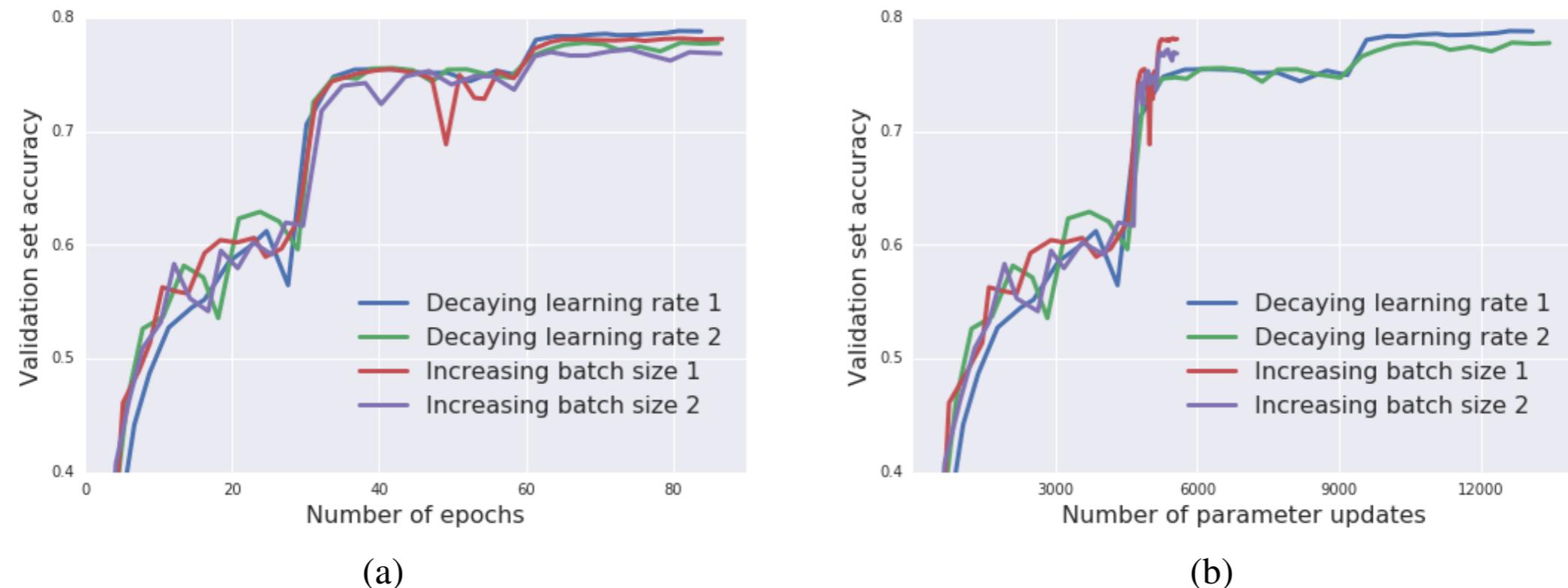
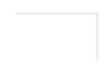


Figure 6: Inception-ResNet-V2 on ImageNet. Increasing the batch size during training achieves similar results to decaying the learning rate, but it reduces the number of parameter updates from just over 14000 to below 6000. We run each experiment twice to illustrate the variance.

Smith, S. L., Kindermans, P. J., Ying, C., & Le, Q. V. (2017). Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*.

Decreasing the learning rate over the course of training

1. Learning rate decay
2. **Learning rate schedulers in PyTorch**
3. Training with "momentum"
4. ADAM: Adaptive learning rates & momentum
5. Using optimization algorithms in PyTorch
6. Optimization in deep learning: Additional topics



Learning Rate Decay in PyTorch

Option 1. Just call your own function at the end of each epoch:

```
def adjust_learning_rate(optimizer, epoch, initial_lr, decay_rate):
    """Exponential decay every 10 epochs"""
    if not epoch % 10:
        lr = initial_lr * torch.exp(-decay_rate*epoch)
        for param_group in optimizer.param_groups:
            param_group['lr'] = lr
```

Learning Rate Decay in PyTorch

Option 2. Use one of the built-in tools in PyTorch:
(many more available)

(Here, the most generic version.)

CLASS `torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda, last_epoch=-1)`

[SOURCE]

Sets the learning rate of each parameter group to the initial lr times a given function. When last_epoch=-1, sets initial lr as lr.

Parameters:

- **optimizer** ([Optimizer](#)) – Wrapped optimizer.
- **lr_lambda** ([function or list](#)) – A function which computes a multiplicative factor given an integer parameter epoch, or a list of such functions, one for each group in optimizer.param_groups.
- **last_epoch** ([int](#)) – The index of last epoch. Default: -1.

Example

```
>>> # Assuming optimizer has two groups.
>>> lambda1 = lambda epoch: epoch // 30
>>> lambda2 = lambda epoch: 0.95 ** epoch
>>> scheduler = LambdaLR(optimizer, lr_lambda=[lambda1, lambda2])
>>> for epoch in range(100):
>>>     scheduler.step()
>>>     train(...)
>>>     validate(...)
```

Source: <https://pytorch.org/docs/stable/optim.html>

Learning Rate Decay in PyTorch

Example, part 1/2

```
#####
### Model Initialization
#####

torch.manual_seed(RANDOM_SEED)
model = MLP(num_features=28*28,
            num_hidden=100,
            num_classes=10)

model = model.to(DEVICE)

optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

#####
### LEARNING RATE SCHEDULER
#####

scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer,
                                                    gamma=0.1)

...
```



Learning Rate Decay in PyTorch

```
for epoch in range(5):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(DEVICE)
        targets = targets.to(DEVICE)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)

        #cost = F.nll_loss(torch.log(probas), targets)
        cost = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        cost.backward()
        minibatch_cost.append(cost)
        ### UPDATE MODEL PARAMETERS

        optimizer.step()

        ### LOGGING
        if not batch_idx % 50:
            print ('Epoch: %03d/%03d | Batch %03d/%03d | Cost: %.4f'
                  %(epoch+1, NUM_EPOCHS, batch_idx,
                    len(train_loader), cost))

#####
### Update Learning Rate
scheduler.step() # don't have to do it every epoch!
#####

model.eval()
```

Example, part 2/2

standard color augmentation in [21] is used. We adopt batch normalization (BN) [16] right after each convolution and before activation, following [16]. We initialize the weights as in [12] and train all plain/residual nets from scratch. We use SGD with a mini-batch size of 256. The learning rate starts from 0.1 and is divided by 10 when the error plateaus, and the models are trained for up to 60×10^4 iterations. We use a weight decay of 0.0001 and a momentum of 0.9. We do not use dropout [13], following the practice in [16].

http://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html

He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition 2016 (pp. 770-778).

scheduler.ipynb:

```
model = MultilayerPerceptron(num_features=28*28,
                             num_hidden_1=NUM_HIDDEN_1,
                             num_hidden_2=NUM_HIDDEN_2,
                             drop_proba=0.5,
                             num_classes=10)
model = model.to(DEVICE)

optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
                                                       factor=0.1,
                                                       mode='max',
                                                       verbose=True)

minibatch_loss_list, train_acc_list, valid_acc_list = train_model(
    model=model,
    num_epochs=NUM_EPOCHS,
    train_loader=train_loader,
    valid_loader=valid_loader,
    test_loader=test_loader,
    optimizer=optimizer,
    device=DEVICE,
    logging_interval=800,
    scheduler=scheduler,
    scheduler_on='valid_acc')
```

scheduler.ipynb:

```
def train_model(model, num_epochs, train_loader,
                valid_loader, test_loader, optimizer,
                device, logging_interval=50,
                scheduler=None,
                scheduler_on='valid_acc'):

    start_time = time.time()
    minibatch_loss_list, train_acc_list, valid_acc_list = [], [], []

    for epoch in range(num_epochs):

        model.train()
        for batch_idx, (features, targets) in enumerate(train_loader):

            ...

        if scheduler is not None:

            if scheduler_on == 'valid_acc':
                scheduler.step(valid_acc_list[-1])
            elif scheduler_on == 'minibatch_loss':
                scheduler.step(minibatch_loss_list[-1])
            else:
                raise ValueError(f'Invalid `scheduler_on` choice.'
```

Saving Models in PyTorch

Save Model

```
model.to(torch.device('cpu'))
torch.save(model.state_dict(), './my_model_2epochs.pt')
torch.save(optimizer.state_dict(), './my_optimizer_2epochs.pt')
torch.save(scheduler.state_dict(), './my_scheduler_2epochs.pt')
```

Load Model

```
model = MLP(num_features=28*28,
            num_hidden=100,
            num_classes=10)

model.load_state_dict(torch.load('./my_model_2epochs.pt'))
model = model.to(DEVICE)

# for this particular optimizer not necessary, as it doesn't have a state
# but good practice, so you don't forget it when using other optimizers
# later
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
optimizer.load_state_dict(torch.load('./my_optimizer_2epochs.pt'))

scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer,
                                                    gamma=0.1,
                                                    last_epoch=-1)
scheduler.load_state_dict(torch.load('./my_scheduler_2epochs.pt'))

model.train()
```

Learning rate schedulers have the advantage that we can also simply save their state for reuse (e.g., saving and continuing training later)

Nudging SGD into the right direction

1. Learning rate decay
2. Learning rate schedulers in PyTorch
- 3. Training with "momentum"**
4. ADAM: Adaptive learning rates & momentum
5. Using optimization algorithms in PyTorch
6. Optimization in deep learning: Additional topics



Training with "Momentum"

Momentum

From Wikipedia, the free encyclopedia

This article is about linear momentum. It is not to be confused with angular momentum.

This article is about momentum in physics. For other uses, see Momentum (disambiguation).

In [Newtonian mechanics](#), **linear momentum**, **translational momentum**, or simply **momentum** (pl. momenta) is the product of the [mass](#) and [velocity](#) of an object. It is a [vector](#) quantity, possessing a magnitude and a direction in three-dimensional space. If m is an object's mass and \mathbf{v} is the velocity (also a vector), then the momentum is

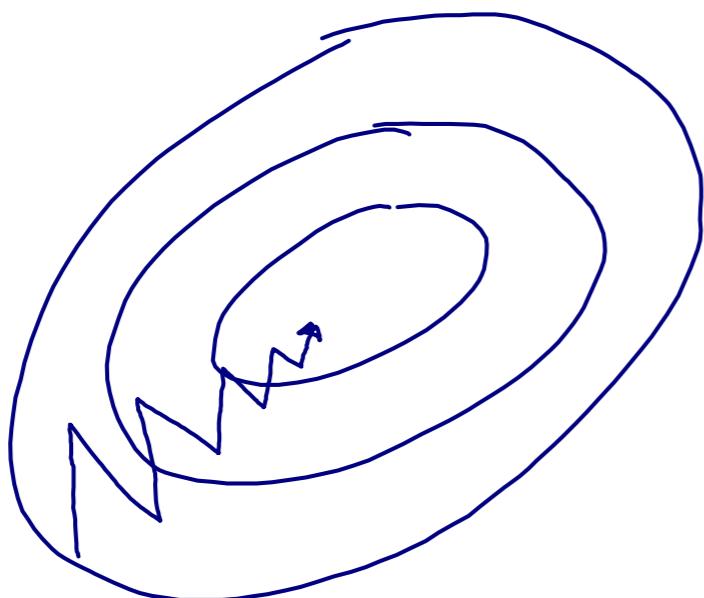
Source: <https://en.wikipedia.org/wiki/Momentum>

- Concept: In momentum learning, we try to accelerate convergence by dampening oscillations using "velocity" (the speed of the "movement" from previous updates)

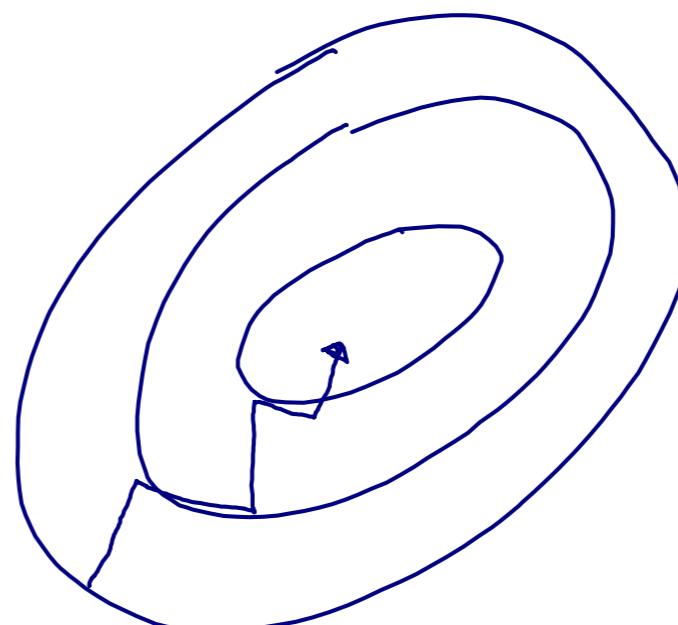
Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks : The Official Journal of the International Neural Network Society*, 12(1), 145–151. [http://doi.org/10.1016/S0893-6080\(98\)00116-6](http://doi.org/10.1016/S0893-6080(98)00116-6)

Training with "Momentum"

- Concept: In momentum learning, we try to accelerate convergence by dampening oscillations using "velocity" (the speed of the "movement" from previous updates)

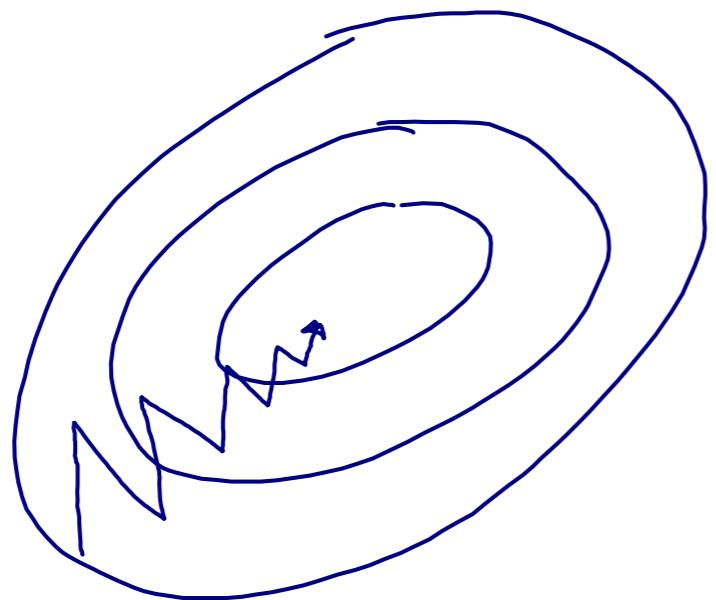


Without momentum

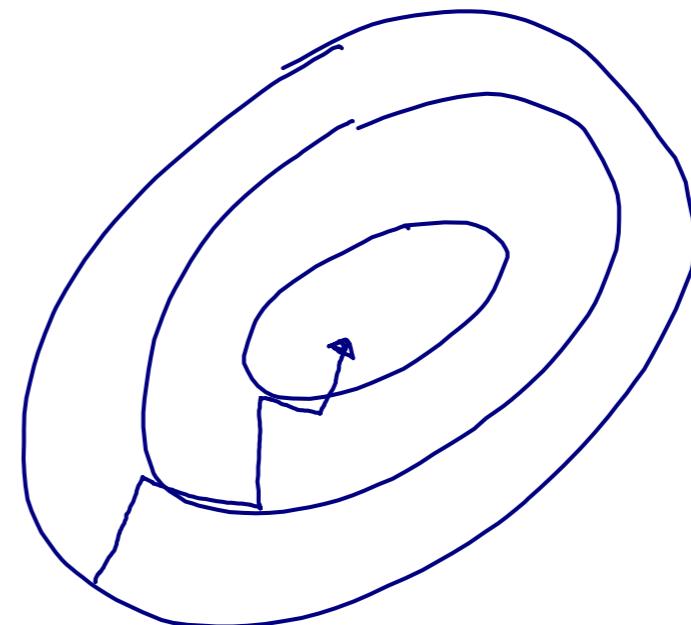


With momentum

Training with "Momentum"



Without momentum



With momentum

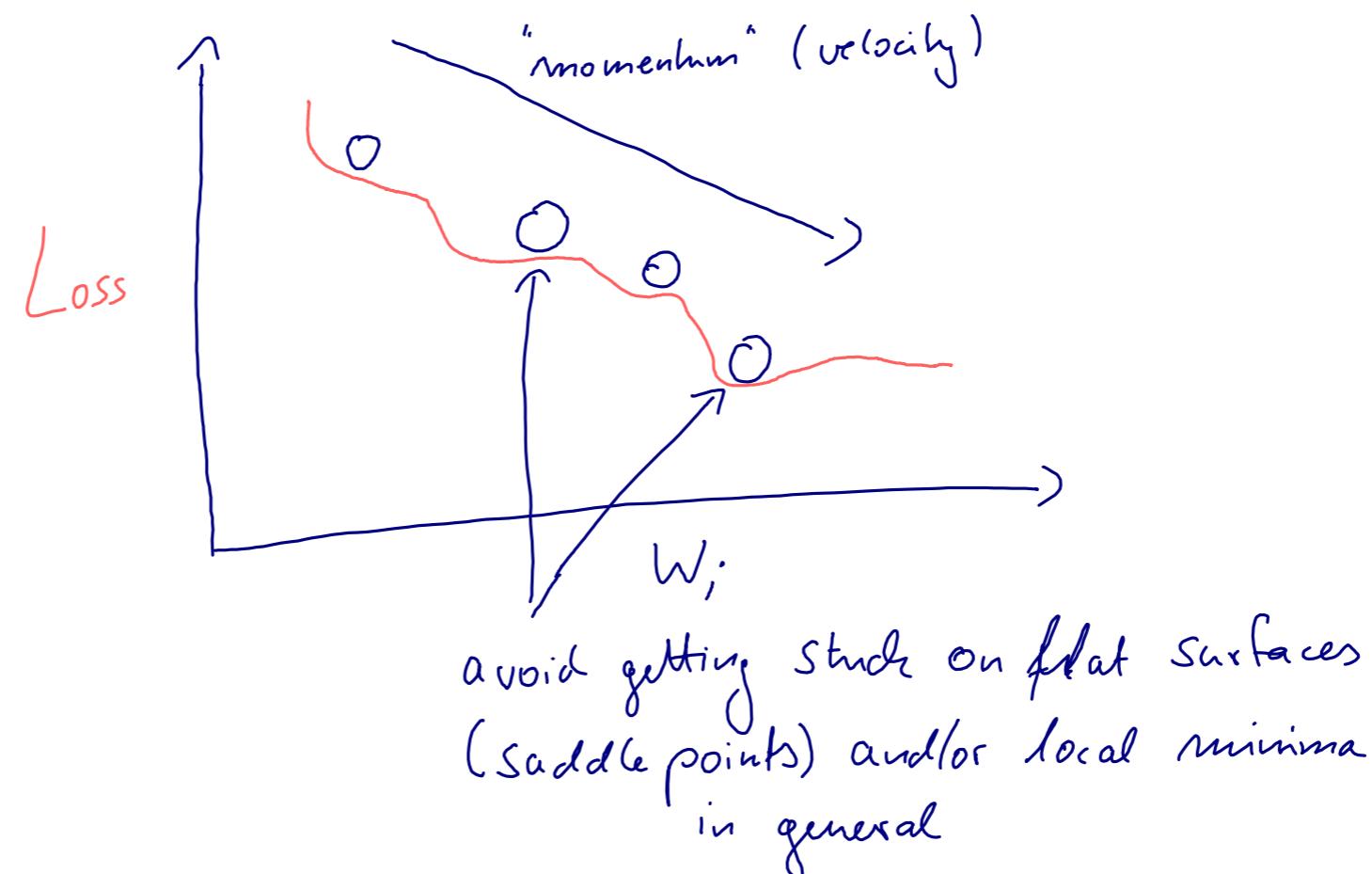
Key take-away:

Not only move in the (opposite) direction of the gradient, but also move in the "averaged" direction of the last few updates



Training with "Momentum"

Helps with dampening oscillations, but also helps with escaping local minima traps



Training with "Momentum"

$$\Delta w_{i,j}(t) := \alpha \cdot \Delta w_{i,j}(t - 1) + \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

Often referred to as "velocity" v

"velocity" from the previous iteration

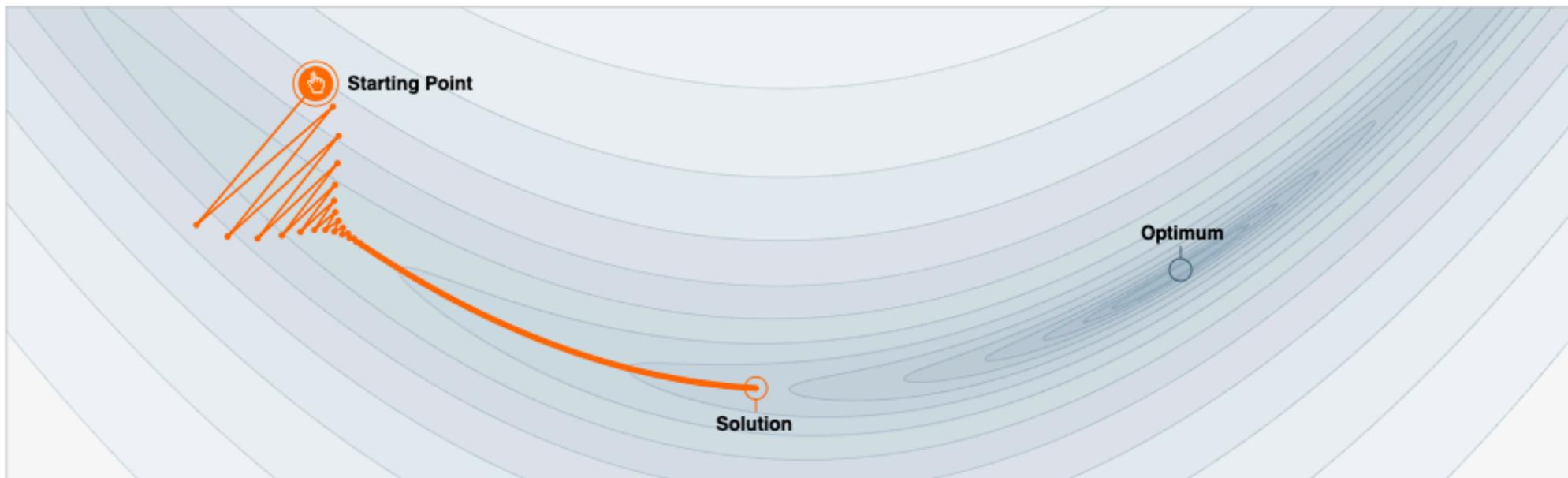
Usually, we choose a momentum rate between 0.9 and 0.999; you can think of it as a "friction" or "dampening" parameter

Regular partial derivative/gradient multiplied by learning rate at current time step t

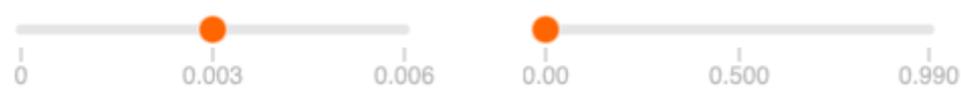
Weight update using the velocity vector:

$$w_{i,j}(t + 1) := w_{i,j}(t) - \Delta w_{i,j}(t)$$

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks : The Official Journal of the International Neural Network Society*, 12(1), 145–151. [http://doi.org/10.1016/S0893-6080\(98\)00116-6](http://doi.org/10.1016/S0893-6080(98)00116-6)

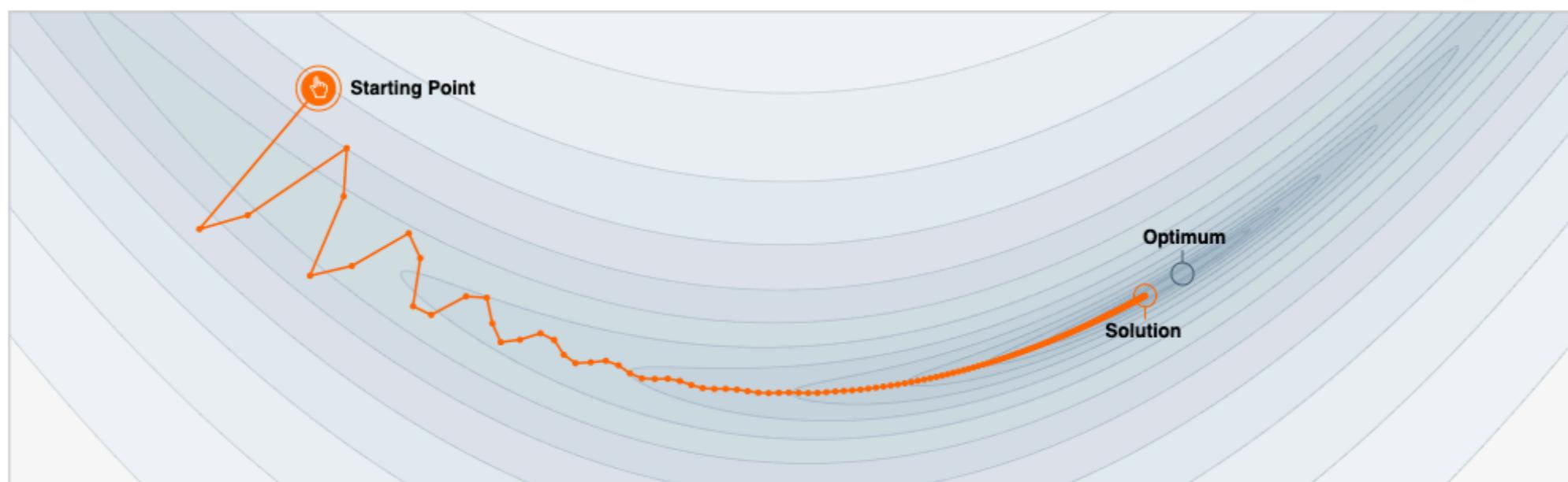


Step-size $\alpha = 0.02$

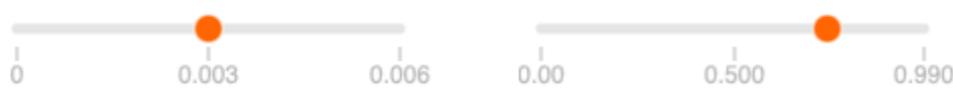


Momentum $\beta = 0.0$

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?



Step-size $\alpha = 0.02$



Momentum $\beta = 0.99$

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Source: <https://distill.pub/2017/momentum/>

Combining adaptive learning rates with momentum

1. Learning rate decay
2. Learning rate schedulers in PyTorch
3. Training with "momentum"
- 4. ADAM: Adaptive learning rates & momentum**
5. Using optimization algorithms in PyTorch
6. Optimization in deep learning: Additional topics



Adaptive Learning Rates

There are many different flavors of adapting the learning rate
(bit out of scope for this course to review them all)

Key take-aways:

- decrease learning if the gradient changes its direction
- increase learning if the gradient stays consistent



Adaptive Learning Rates

Key take-aways:

- decrease learning if the gradient changes its direction
- increase learning if the gradient stays consistent

Step 1: Define a local gain (g) for each weight (initialized with $g=1$)

$$\Delta w_{i,j} := \eta \cdot g_{i,j} \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}$$



Adaptive Learning Rates

Step 1: Define a local gain (g) for each weight (initialized with $g=1$)

$$\Delta w_{i,j} := \eta \cdot g_{i,j} \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}$$

Step 2:

If gradient is consistent

$$g_{i,j}(t) := g_{i,j}(t - 1) + \beta$$

else

$$g_{i,j}(t) := g_{i,j}(t - 1) \cdot (1 - \beta)$$

Note that
multiplying by a factor has a larger
impact if gains are large, compared
to adding a term

(dampening effect if updates oscillate
in the wrong direction)

Adaptive Learning Rate via RMSProp

- Unpublished algorithm by Geoff Hinton (but very popular) based on Rprop [1]
- Very similar to another concept called AdaDelta
- Concept: divide learning rate by an exponentially decreasing moving average of the squared gradients
- This takes into account that gradients can vary widely in magnitude
- Here, RMS stands for "Root Mean Squared"
- Also, damps oscillations like momentum (but in practice, works a bit better)

[1] Igel, Christian, and Michael Hüsker. "Improving the Rprop learning algorithm." *Proceedings of the Second International ICSC Symposium on Neural Computation (NC 2000)*. Vol. 2000. ICSC Academic Press, 2000.

Adaptive Learning Rate via RMSProp

$$MeanSquare(w_{i,j}, t) := \beta \cdot MeanSquare(w_{i,j}, t - 1) + (1 - \beta) \left(\frac{\partial \mathcal{L}}{w_{i,j}(t)} \right)^2$$

moving average of the squared gradient for each weight

$$w_{i,j}(t) := w_{i,j}(t) - \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}(t)} / \left(\sqrt{MeanSquare(w_{i,j}, t)} + \epsilon \right)$$

where beta is typically between 0.9 and 0.999

small epsilon term to
avoid division by zero

Adaptive Learning Rate via ADAM

- ADAM (Adaptive Moment Estimation) is probably the most widely used optimization algorithm in DL as of today
- It is a combination of the momentum method and RMSProp

Momentum-like term:

$$\Delta w_{i,j}(t) := \alpha \cdot \Delta w_{i,j}(t-1) + \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

m_t

$$m_t := \alpha \cdot m_{t-1} + (1 - \alpha) \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

original momentum term

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Adaptive Learning Rate via ADAM

Momentum-like term:

$$m_t := \alpha \cdot m_{t-1} + (1 - \alpha) \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

RMSProp term:

$$r := \beta \cdot \text{MeanSquare}(w_{i,j}, t-1) + (1 - \beta) \left(\frac{\partial \mathcal{L}}{\partial w_{i,j}}(t) \right)^2$$

ADAM update:

$$w_{i,j} := w_{i,j} - \eta \frac{m_t}{\sqrt{r} + \epsilon}$$

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Adaptive Learning Rate via ADAM

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Step size
 Require: $\beta_1, \beta_2 \in [0, 1]$: Exponential decay rates for the moment estimates
 Require: $f(\theta)$: Stochastic objective function with parameters θ
 Require: θ_0 : Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector)
 $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
 $t \leftarrow 0$ (Initialize timestep)

while θ_t not converged **do**

- $t \leftarrow t + 1$
- $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
- $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
- $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
- $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
- $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)

end while

return θ_t (Resulting parameters)

Also add a bias correction term
for better conditioning in earlier iterations

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Experimenting with different optimization algorithms

1. Learning rate decay
2. Learning rate schedulers in PyTorch
3. Training with "momentum"
4. ADAM: Adaptive learning rates & momentum
- 5. Using optimization algorithms in PyTorch**
6. Optimization in deep learning: Additional topics



Using Different Optimizers in PyTorch

Usage is the same as for vanilla SGD, which we used before,
you can find an overview at: <https://pytorch.org/docs/stable/optim.html>

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)
```

Using Different Optimizers in PyTorch

Usage is the same as for vanilla SGD, which we used before,
you can find an overview at: <https://pytorch.org/docs/stable/optim.html>

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

Remember to save the optimizer state if you are using, e.g., Momentum or ADAM, and want to continue training later
(see earlier slides on saving states of the learning rate schedulers).

Adaptive Learning Rate via ADAM

$$m_t := \alpha \cdot m_{t-1} + (1 - \alpha) \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

$$r := \beta \cdot \text{MeanSquare}(w_{i,j}, t-1) + (1 - \beta) \left(\frac{\partial \mathcal{L}}{\partial w_{i,j}}(t) \right)^2$$

CLASS `torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)`

[SOURCE] ⚡

Implements Adam algorithm.

It has been proposed in [Adam: A Method for Stochastic Optimization](#).

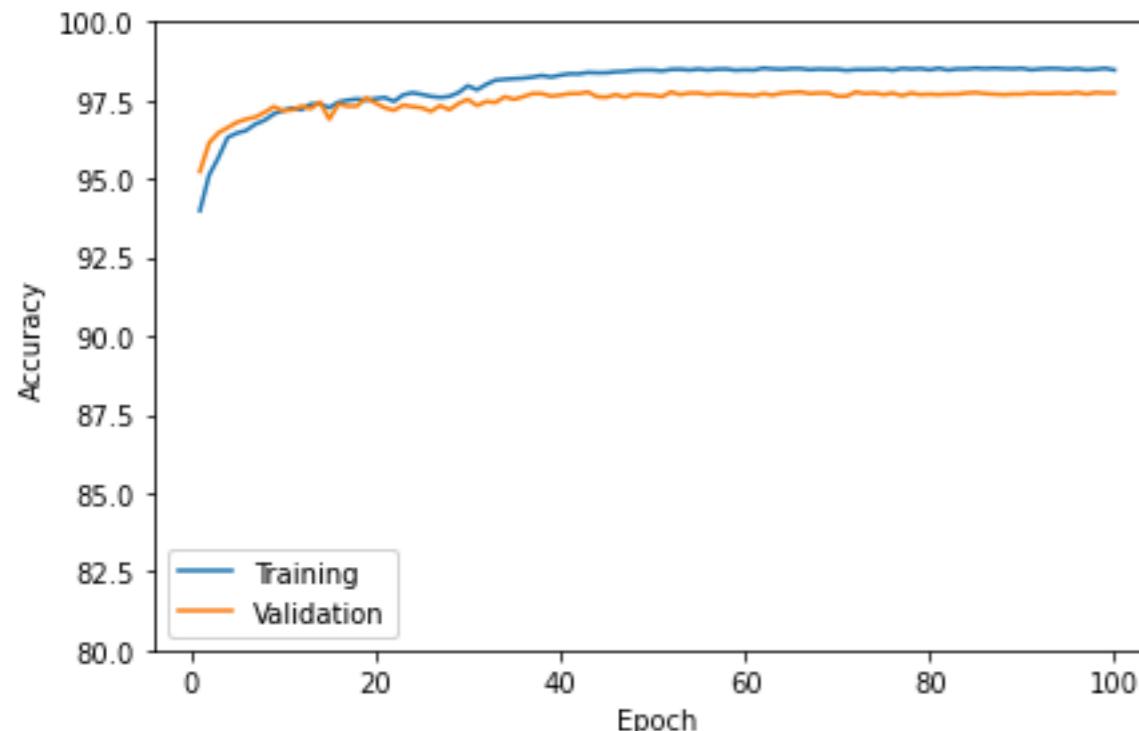
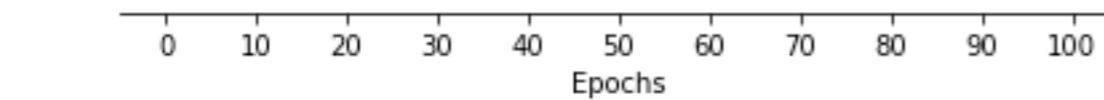
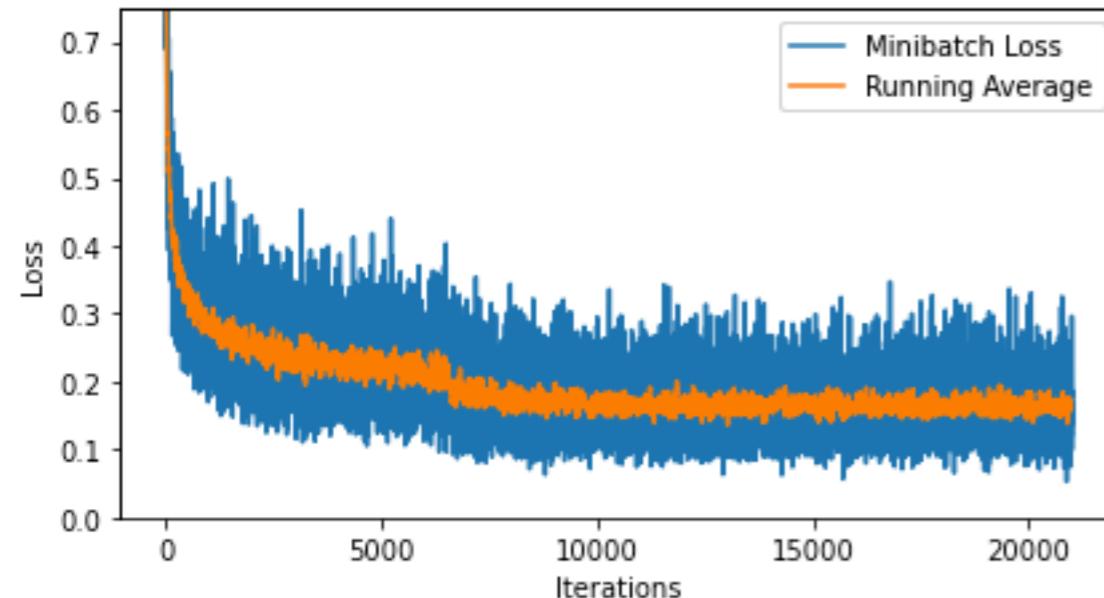
The default settings for the "betas" work usually just fine

- Parameters:**
- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
 - **lr** (*float*, *optional*) – learning rate (default: 1e-3)
 - **betas** (*Tuple*[*float*, *float*], *optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
 - **eps** (*float*, *optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
 - *amsgrad* (*float*, *optional*) – whether to use the AMSGrad variant of this algorithm from the paper [On the Convergence of Adam and Beyond](#) (default: 0)

Source: <https://pytorch.org/docs/stable/optim.html>

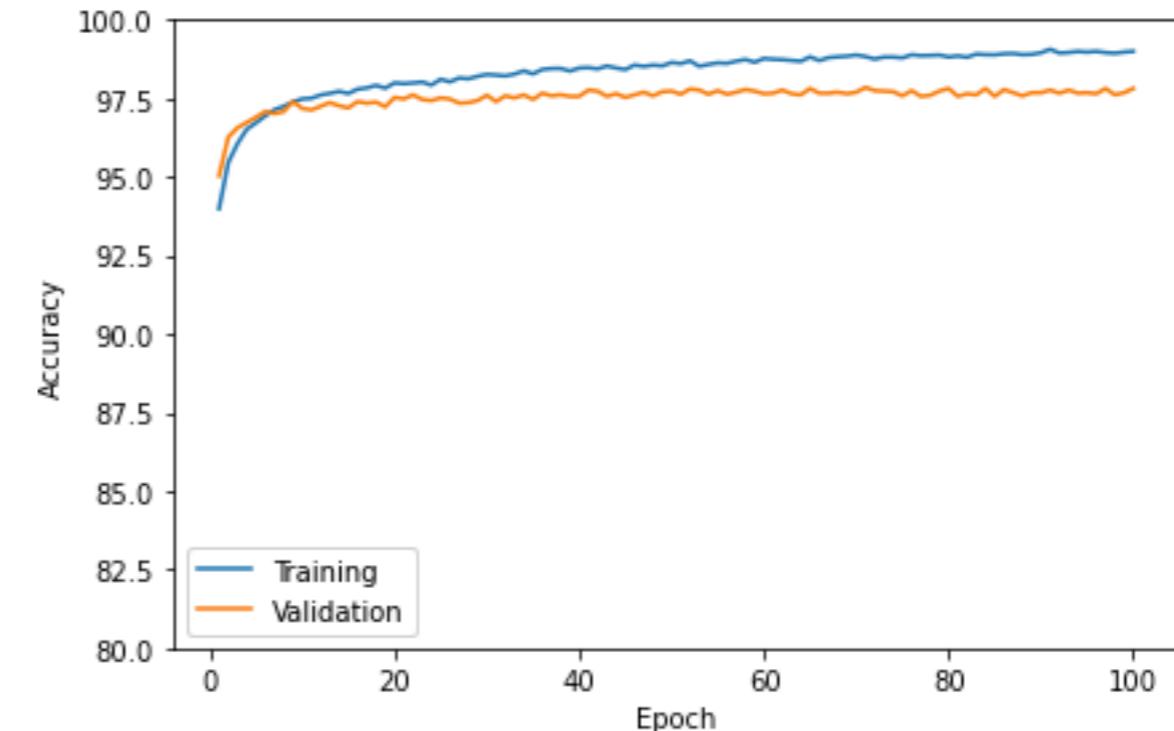
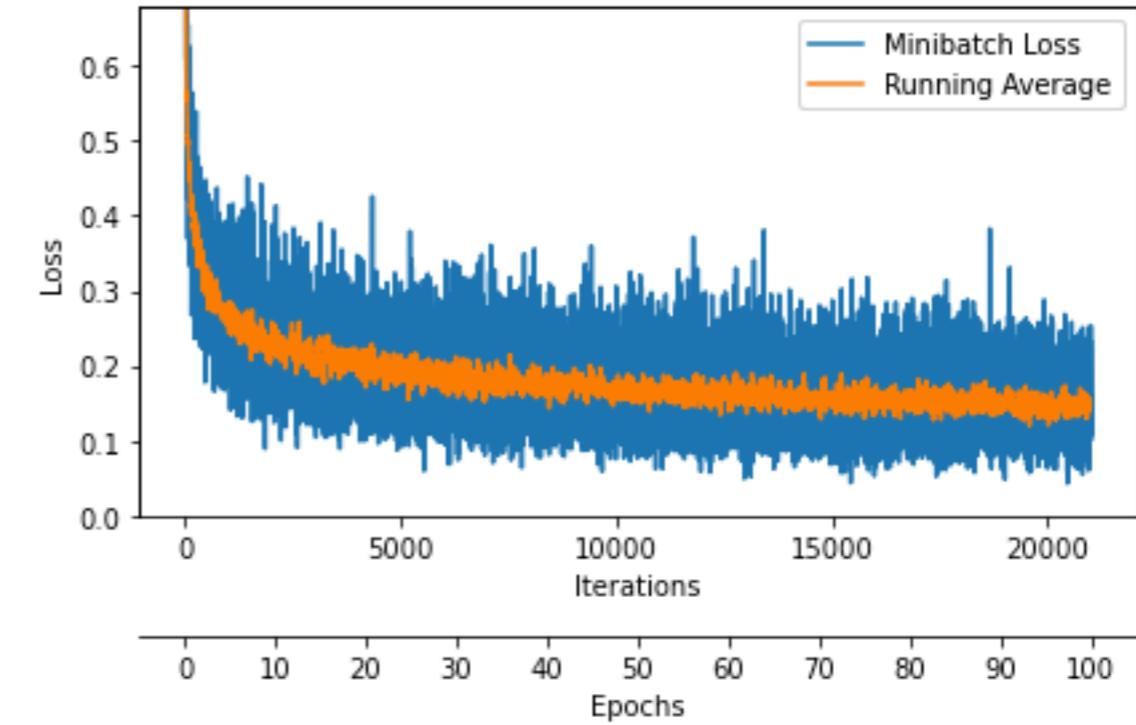
sgd-scheduler-momentum.ipynb

Epoch: 100/100 | Train: 98.47% | Validation: 97.73%
Time elapsed: 4.60 min
Total Training Time: 4.60 min
Test accuracy 97.34%



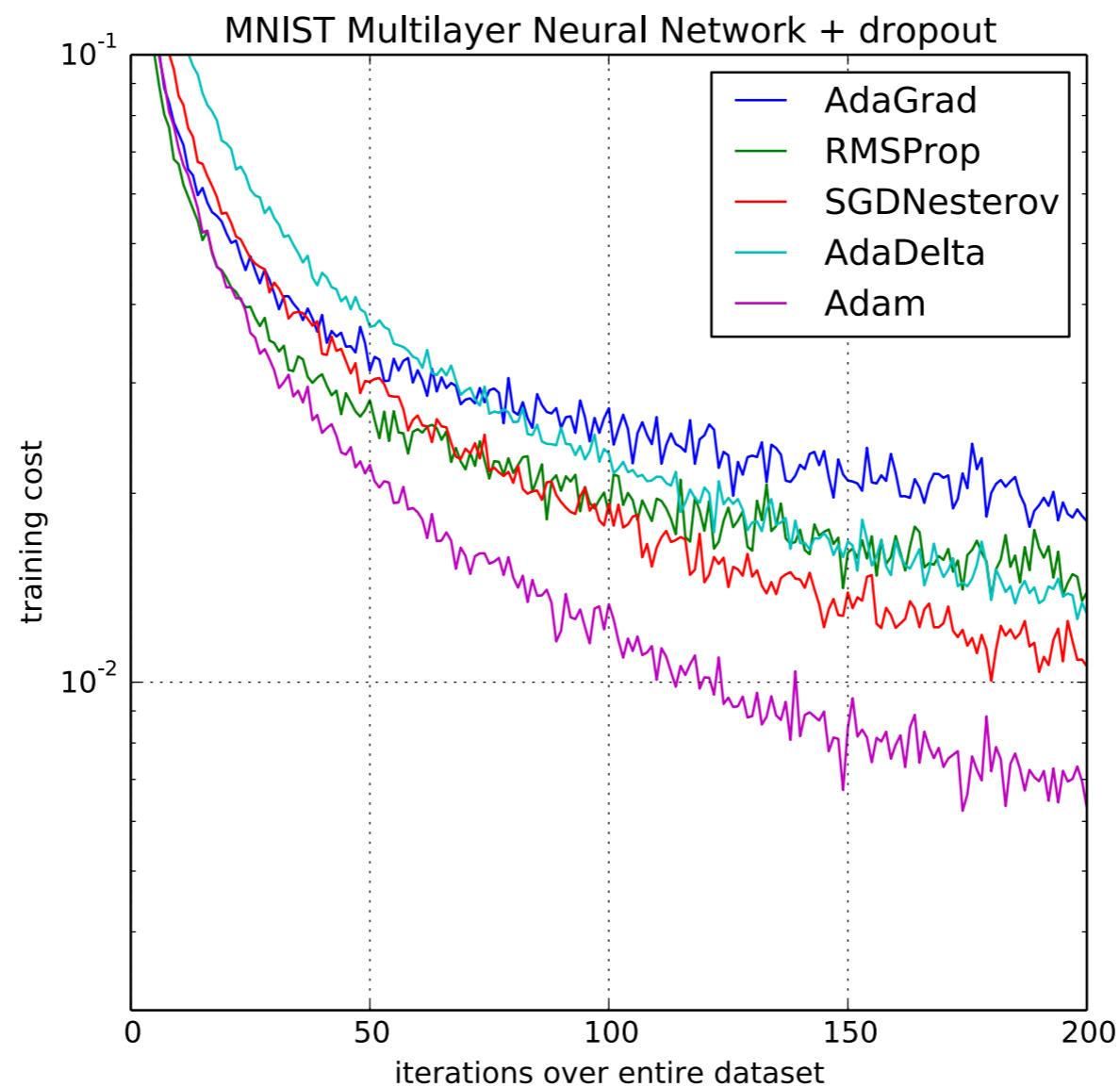
adam.ipynb

Epoch: 100/100 | Train: 99.00% | Validation: 97.82%
Time elapsed: 4.71 min
Total Training Time: 4.71 min
Test accuracy 97.37%



Decreasing the learning rate over the course of training

1. Learning rate decay
2. Learning rate schedulers in PyTorch
3. Training with "momentum"
4. ADAM: Adaptive learning rates & momentum
5. Using optimization algorithms in PyTorch
6. **Optimization in deep learning: Additional topics**



Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. <https://arxiv.org/abs/1412.6980>

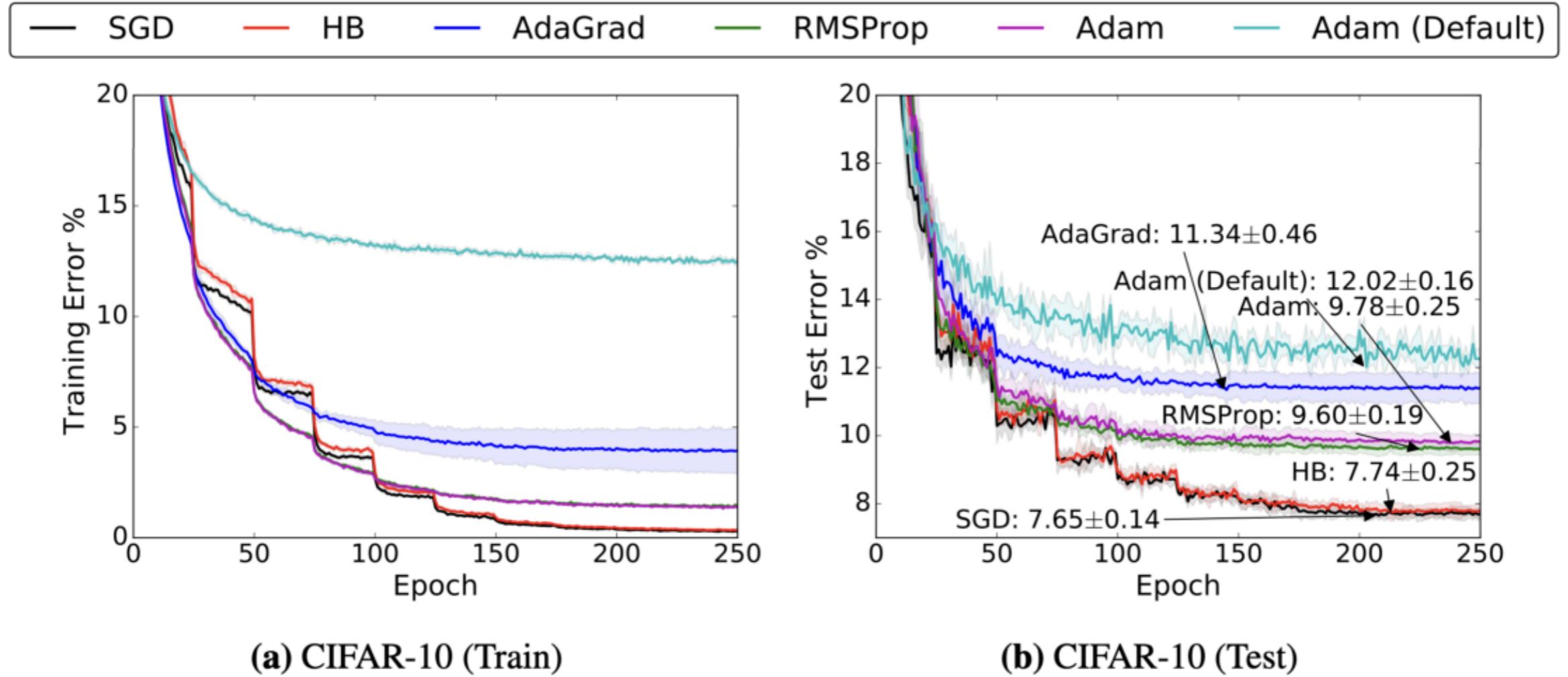


Figure 1: Training (left) and top-1 test error (right) on CIFAR-10. The annotations indicate where the best performance is attained for each method. The shading represents \pm one standard deviation computed across five runs from random initial starting points. In all cases, adaptive methods are performing worse on both train and test than non-adaptive methods.

Wilson AC, Roelofs R, Stern M, Srebro N, Recht B. The marginal value of adaptive gradient methods in machine learning, <https://arxiv.org/abs/1705.08292>

Training Loss vs Generalization Error

Improving Generalization Performance by Switching from Adam to SGD

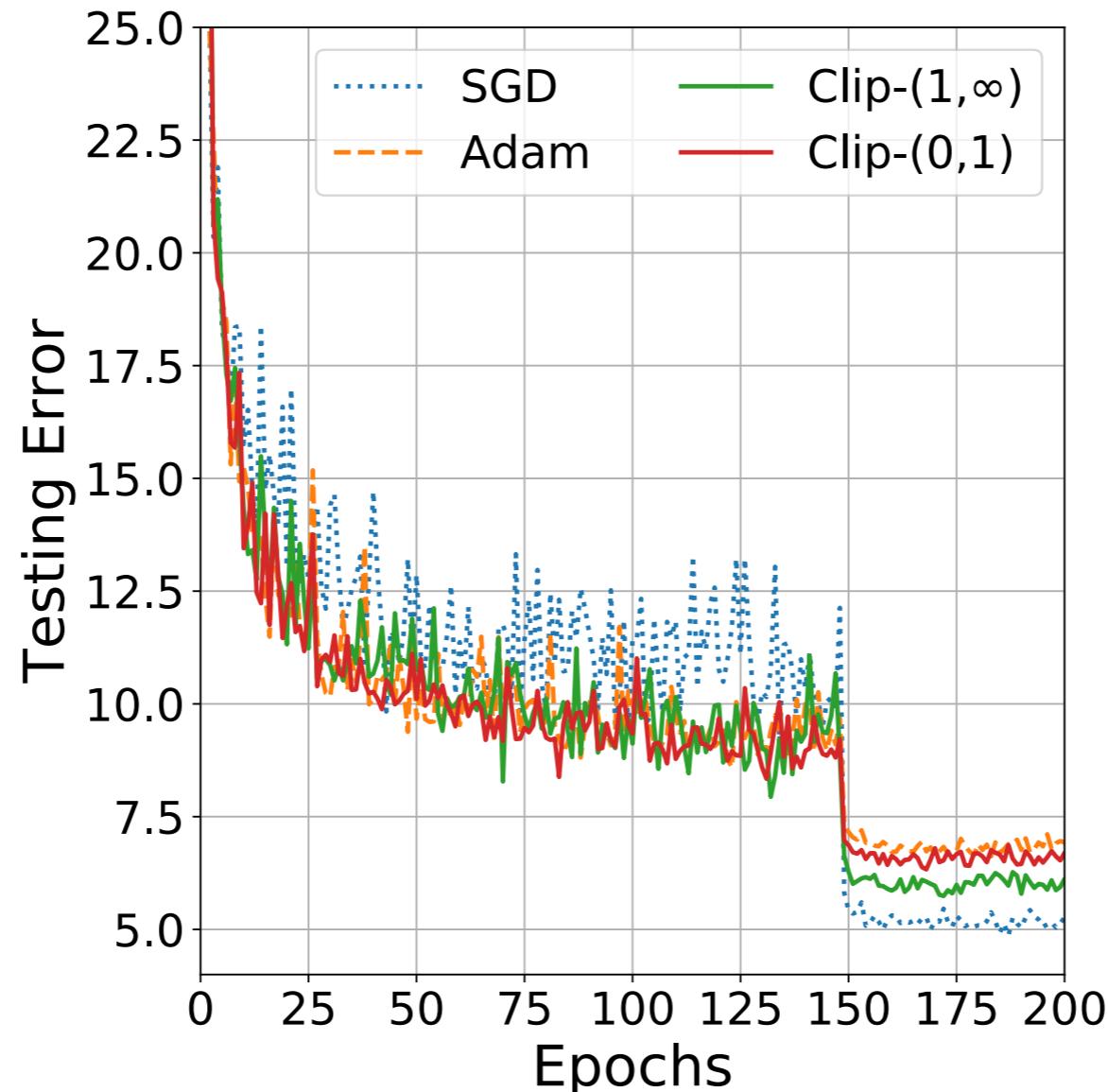
Nitish Shirish Keskar, Richard Socher

(Submitted on 20 Dec 2017)

Despite superior training outcomes, adaptive optimization methods such as Adam, Adagrad or RMSprop have been found to generalize poorly compared to Stochastic gradient descent (SGD). These methods tend to perform well in the initial portion of training but are outperformed by SGD at later stages of training. We investigate a hybrid strategy that begins training with an adaptive method and switches to SGD when appropriate. Concretely, we propose SWATS, a simple strategy which switches from Adam to SGD when a triggering



Training Loss vs Generalization Error



Keskar, N. S., & Socher, R. (2017). Improving generalization performance by switching from adam to sgd. *arXiv preprint arXiv:1712.07628*.

Which Optimizer should I use for my ML Project?

<https://www.lightly.ai/post/which-optimizer-should-i-use-for-my-machine-learning-project>

Optimizer	State Memory [bytes]	# of Tunable Parameters	Strengths	Weaknesses
SGD	0	1	Often best generalization (after extensive training)	Prone to saddle points or local minima Sensitive to initialization and choice of the learning rate α
SGD with Momentum	$4n$	2	Accelerates in directions of steady descent Overcomes weaknesses of simple SGD	Sensitive to initialization of the learning rate α and momentum β
AdaGrad	$\sim 4n$	1	Works well on data with sparse features Automatically decays learning rate	Generalizes worse, converges to sharp minima Gradients may vanish due to aggressive scaling
RMSprop	$\sim 4n$	3	Works well on data with sparse features Built in Momentum	Generalizes worse, converges to sharp minima
Adam	$\sim 8n$	3	Works well on data with sparse features Good default settings Automatically decays learning rate α	Generalizes worse, converges to sharp minima Requires a lot of memory for the state
AdamW	$\sim 8n$	3	Improves on Adam in terms of generalization Broader basin of optimal hyperparameters	Requires a lot of memory for the state
LARS	$\sim 4n$	3	Works well on large batches (up to 32k) Counteracts vanishing and exploding gradients Built in Momentum	Computing norm of gradient for each layer can be inefficient

NEURAL NETWORKS (MAYBE) EVOLVED TO MAKE ADAM THE BEST OPTIMIZER

by bremen79

[https://parameterfree.com/2020/12/06/
neural-network-maybe-evolved-to-make-
adam-the-best-optimizer/](https://parameterfree.com/2020/12/06/neural-network-maybe-evolved-to-make-adam-the-best-optimizer/)

DEC 06
2020

Disclaimer: This post will be a little different than my usual ones. In fact, I won't prove anything and I will just briefly explain some of my conjectures around optimization in deep neural networks. Differently from my usual posts, it is totally possible that what I wrote is completely wrong 😊



"it is known that Adam will not always give you the best performance, yet most of the time people know that they can use it with its default parameters and get, if not the best performance, at least the second best performance on their particular deep learning problem. "

"Usually people try new architectures *keeping the optimization algorithm fixed*, and most of the time the algorithm of choice is Adam. This happens because, as explained above, Adam is the *default optimizer*."

Understanding Generalization through Visualizations

W. Ronny Huang, Zeyad Emam, Micah Goldblum, Liam Fowl, Justin K. Terry, Furong Huang, Tom Goldstein

<https://arxiv.org/abs/1906.03291>

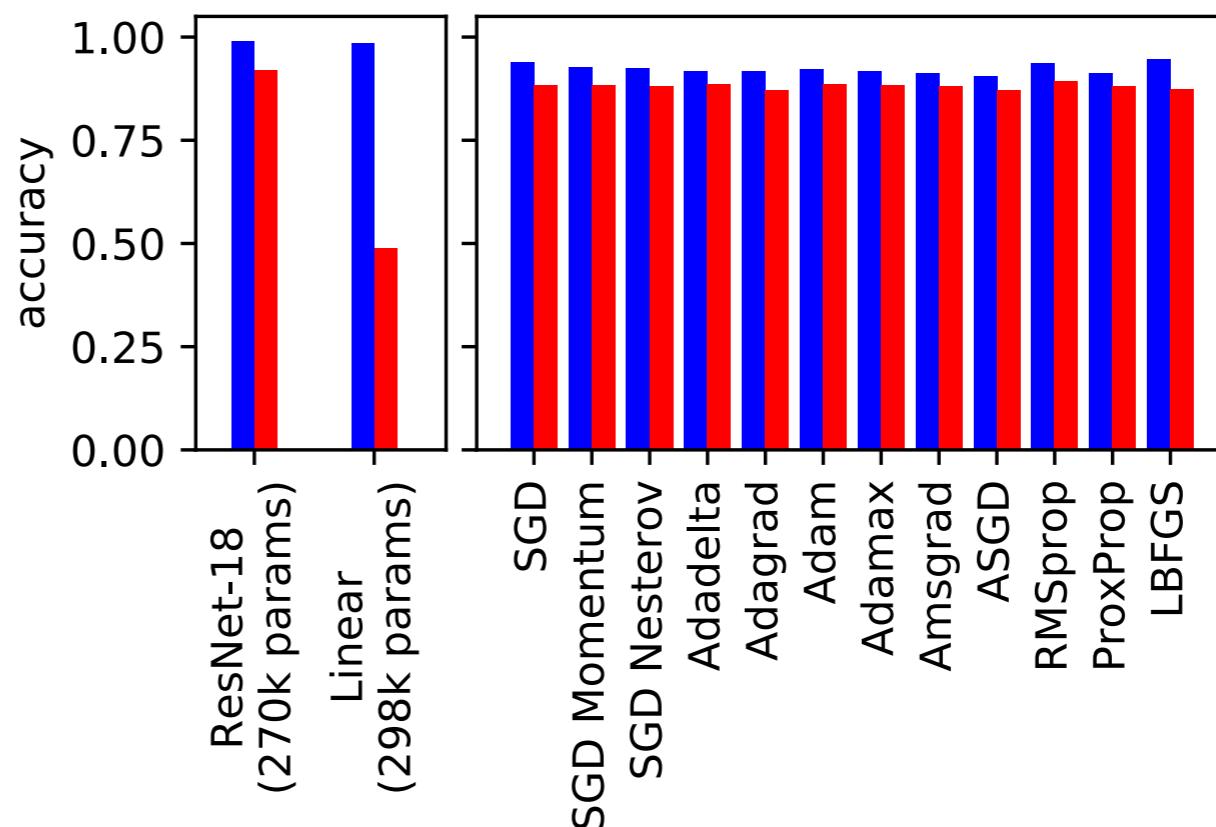


Figure 2: (left) CIFAR10 trained with ResNet-18 and a linear model having comparable number of parameters. Both can fit the training data well, but neural nets are able to generalize to unseen data, while linear models cannot. (right) CIFAR10 trained with various optimizers using VGG13, generalizing well irrespective of the optimizer used.

[Submitted on 15 Oct 2020 (v1), last revised 20 Dec 2020 (this version, v5)]

AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients

Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar Tatikonda, Nicha Dvornek, Xenophon Papademetris, James S. Duncan

<https://arxiv.org/abs/2010.07468><https://github.com/juntang-zhuang/Adabelief-Optimizer>

"uses the exponential moving average of variance of gradient instead of the exponential moving average of square of gradients to calculate the adaptive learning rate"

"trains fast as Adam, generalizes well as SGD, and is stable to train GANs"

Algorithm 1: Adam Optimizer**Initialize** $\theta_0, m_0 \leftarrow 0, v_0 \leftarrow 0, t \leftarrow 0$ **While** θ_t not converged

$$t \leftarrow t + 1$$

$$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$$

$$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Bias Correction

$$\hat{m}_t \leftarrow \frac{m_t}{1 - \beta_1^t}, \hat{v}_t \leftarrow \frac{v_t}{1 - \beta_2^t}$$

Update

$$\theta_t \leftarrow \Pi_{\mathcal{F}, \sqrt{\hat{v}_t}} \left(\theta_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \right)$$

Algorithm 2: AdaBelief Optimizer**Initialize** $\theta_0, m_0 \leftarrow 0, s_0 \leftarrow 0, t \leftarrow 0$ **While** θ_t not converged

$$t \leftarrow t + 1$$

$$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$$

$$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$s_t \leftarrow \beta_2 s_{t-1} + (1 - \beta_2) (g_t - m_t)^2 + \epsilon$$

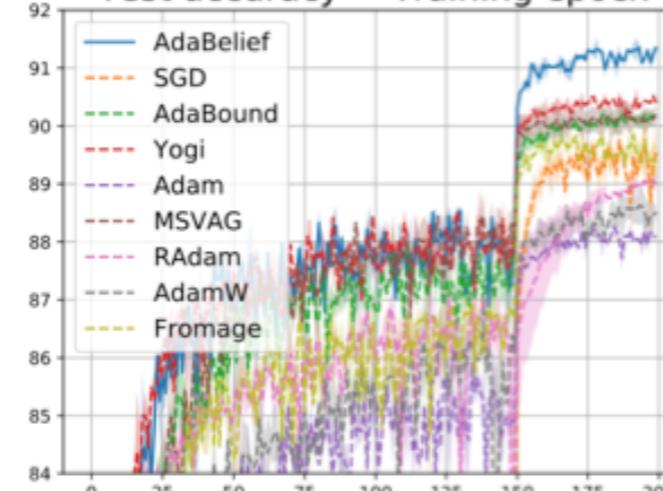
Bias Correction

$$\hat{m}_t \leftarrow \frac{m_t}{1 - \beta_1^t}, \hat{s}_t \leftarrow \frac{s_t}{1 - \beta_2^t}$$

Update

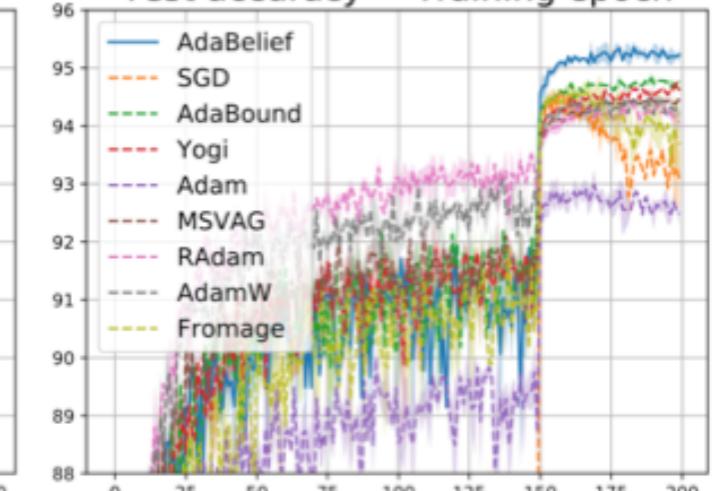
$$\theta_t \leftarrow \Pi_{\mathcal{F}, \sqrt{\hat{s}_t}} \left(\theta_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{s}_t} + \epsilon} \right)$$

Test accuracy ~ Training epoch



(a) VGG11 on Cifar10

Test accuracy ~ Training epoch



(b) ResNet34 on Cifar10