STAT 453: Introduction to Deep Learning and Generative Models

Ben Lengerich

Lecture 12: Optimization / Learning Rates

October 13, 2025

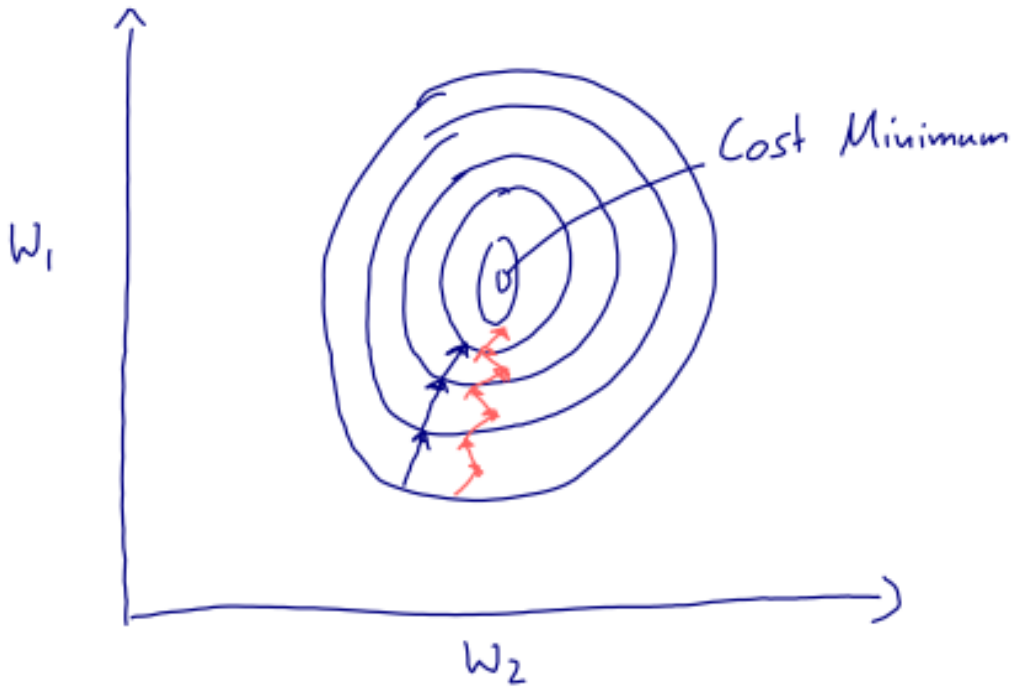
Midterm

- In-class next Wednesday
- Open-note, no electronics/AI
- Review session next Monday
- Study guide up today

Today: Improving Optimization

1. **Learning Rate Decay** → We want a dynamically-changing learning rate instead of the fixed one
2. Learning Rate Schedulers in PyTorch
3. Training with “momentum”
4. ADAM: Adaptive Learning Rates
5. Optimization algorithms in PyTorch

Minibatch Training Recap



- 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**

A **noisy** gradient can be:

- **good**: chance to escape local minima
- **bad**: can lead to extensive oscillation

Minibatch Training: Practical Tip

- Typical minibatch sizes: 32, 64, 128, 256, 512, 1024
  - why powers of 2?
- Usually, pick the biggest that is allowed by your GPU memory
- Practical tip: also want to make the batch size proportional to the number of classes in the dataset

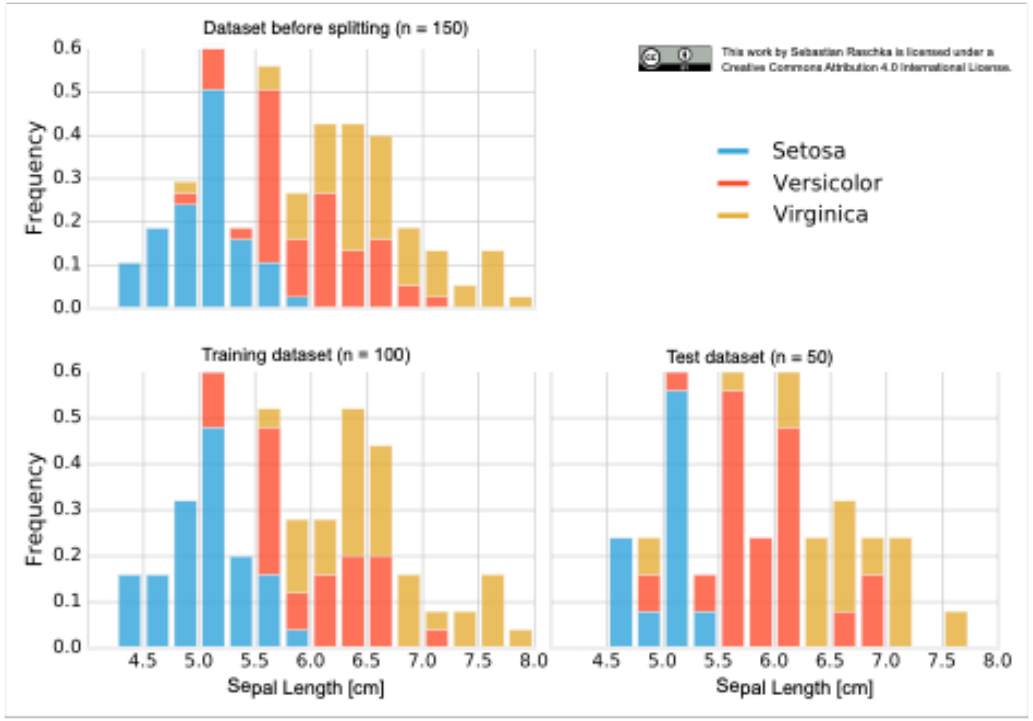
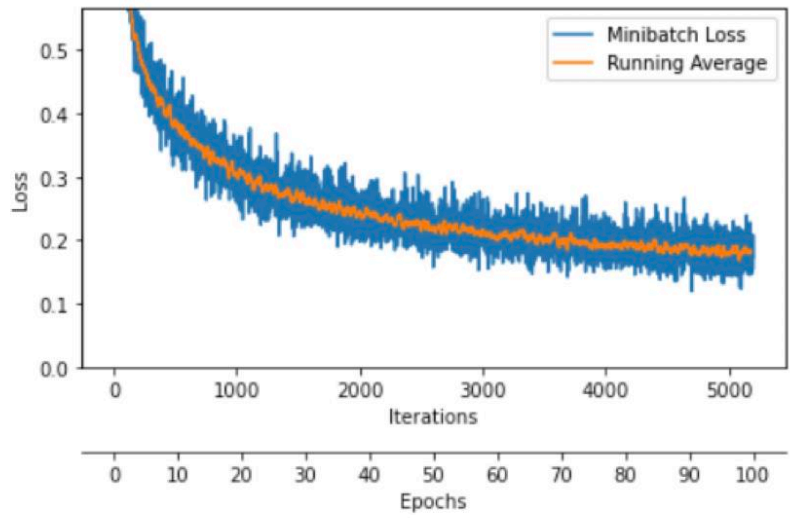


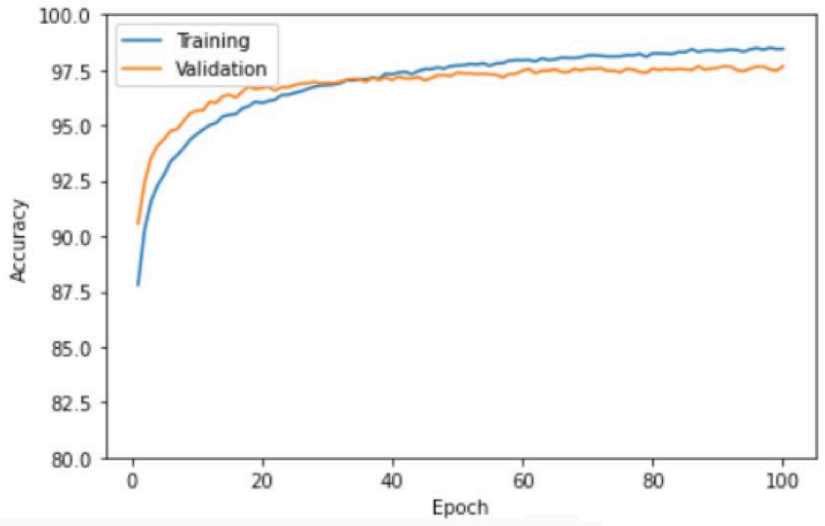
Figure 1: Distribution of *Iris* flower classes upon random subsampling into training and test sets.

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

batchsize-1024.ipynb

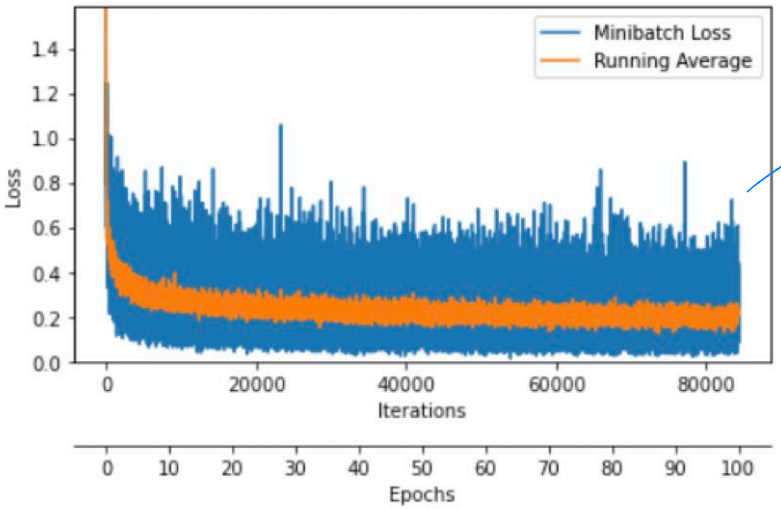
```
Epoch: 100/100 | Train: 98.45% | Validation: 97.67%
Time elapsed: 4.38 min
Total Training Time: 4.38 min
Test accuracy 97.08%
```



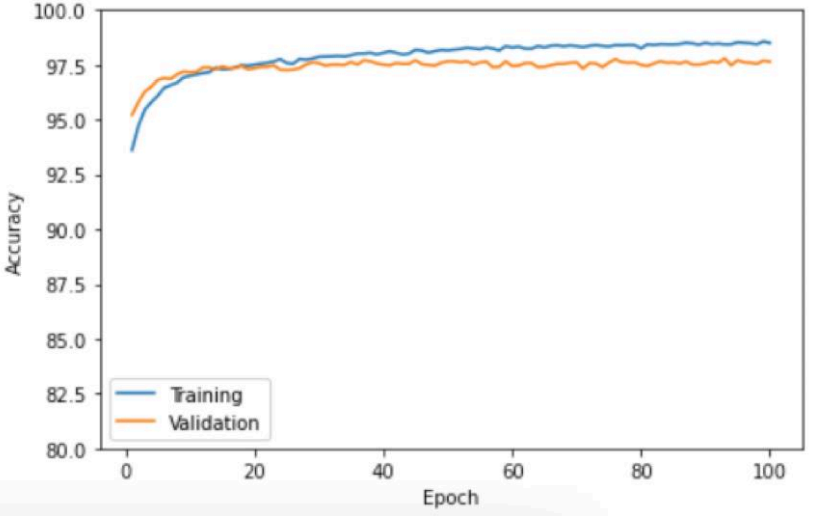


batchsize-64.ipynb

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



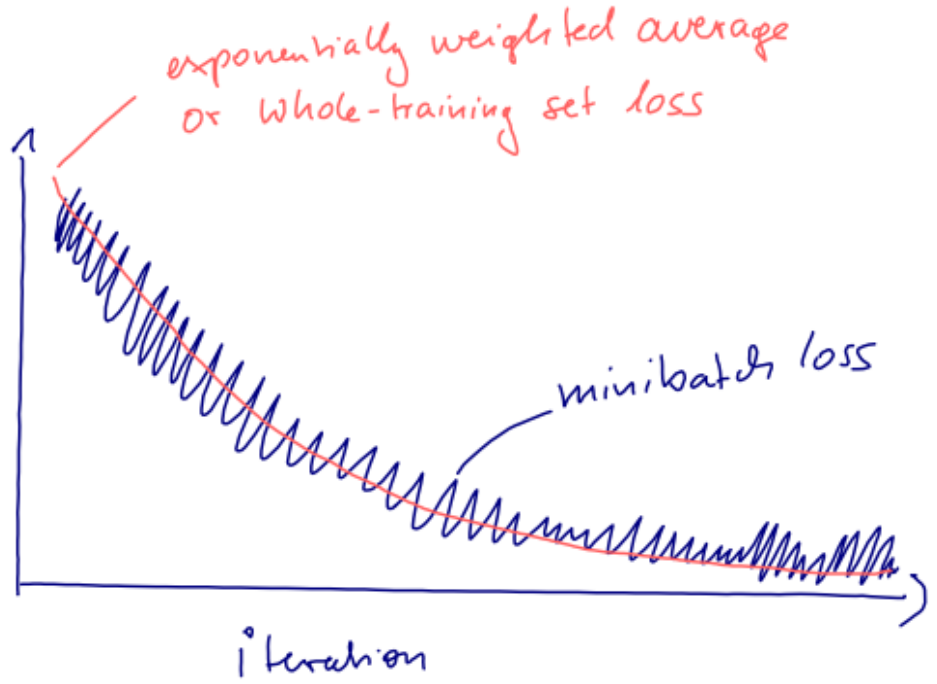
Way more oscillating



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 lr decay:

1. Exponential Decay:

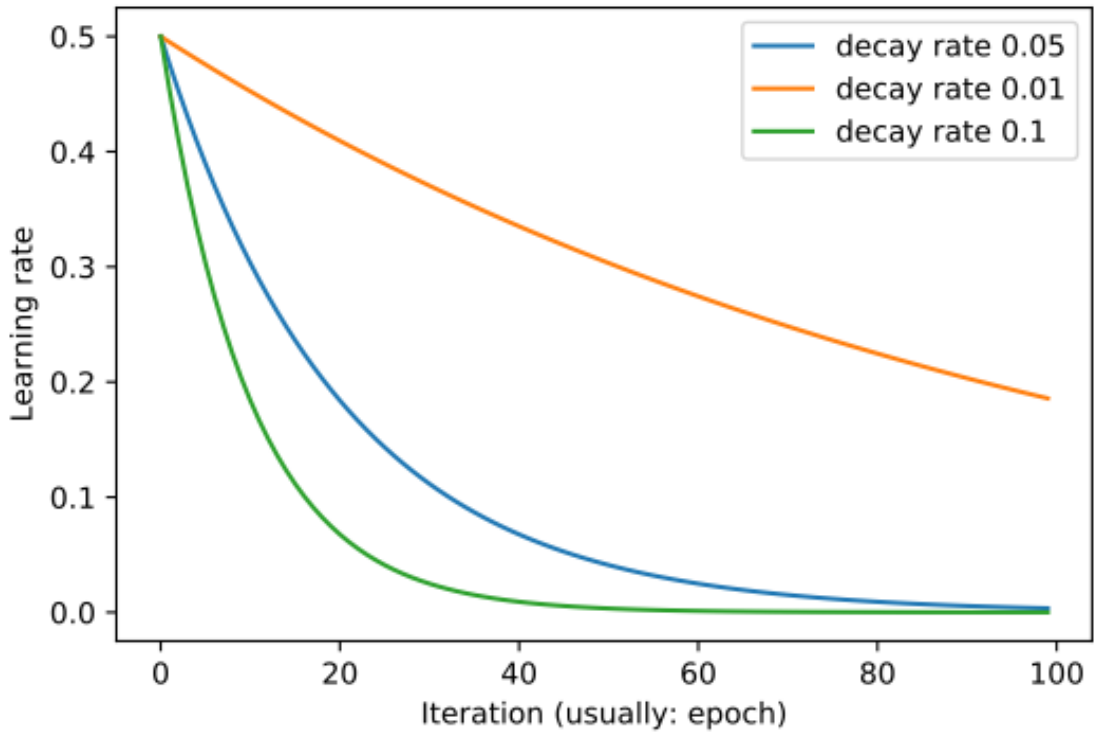
$$\eta_t := \eta_0 e^{-k \cdot t}$$

Initial learning rate set at first

Hyper-parameter

the current epoch

where k is the decay rate



Learning Rate Decay

Most common variants for lr decay:

1. Exponential Decay:

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

where k is the decay rate

2. Halving the learning rate:

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

where t is a multiple of T_0 (e.g. $T_0 = 100$)

阶梯式，只有在 t 是 T_0 的倍数时执行

3. Inverse decay:

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

Cubically decay

Learning Rate Decay

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.

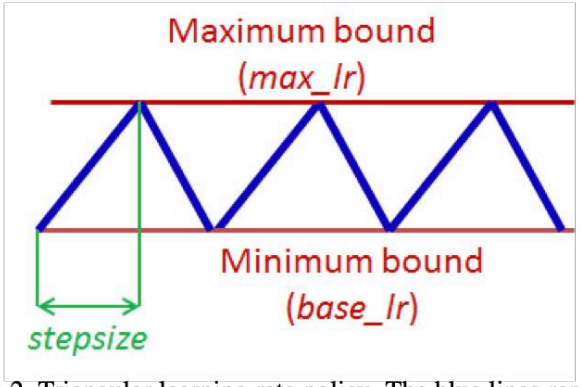
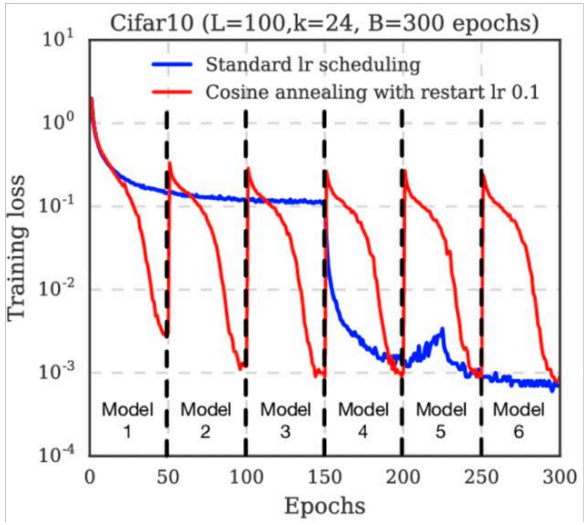


Figure 2. Triangular learning rate policy. The blue lines represent learning rate values changing between bounds. The input parameter *stepsize* is the number of iterations in half a cycle.

Cosine Learning rate decay

SGDR: STOCHASTIC GRADIENT DESCENT WITH WARM RESTARTS, ICLR 2017



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.

Increasing the batch size can achieve the similar performance of learning rate decay

Relationship between Learning rate and Batch Size

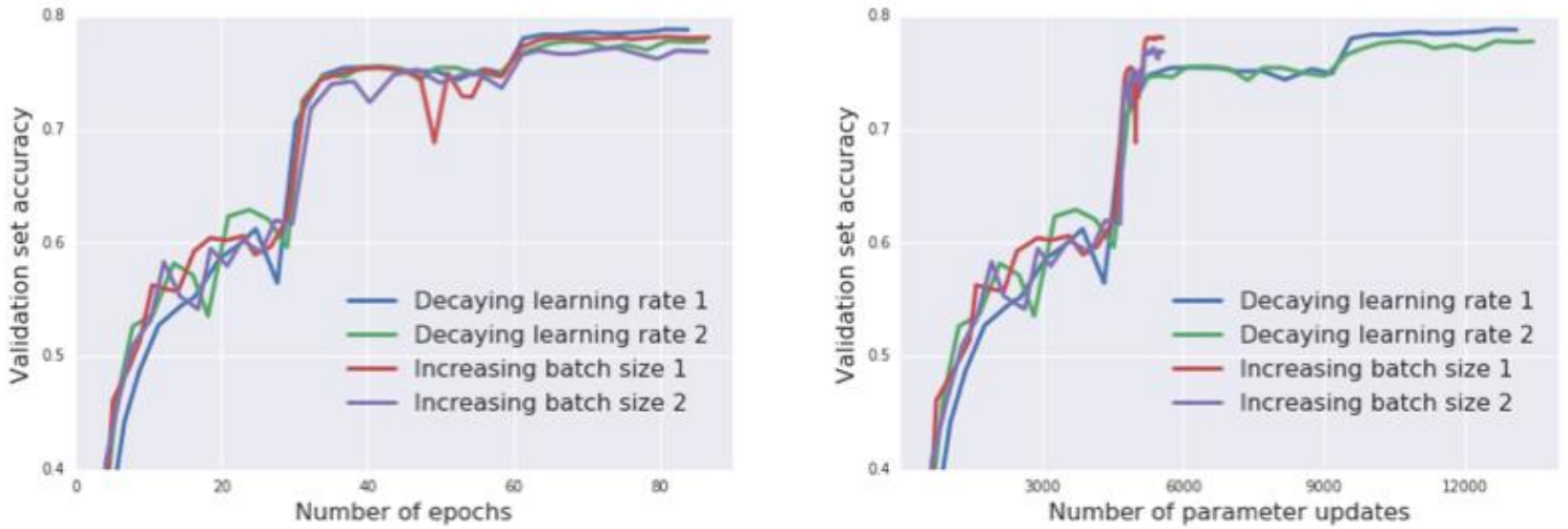


Figure 6: Inception-ResNet-V2 on ImageNet. Increasing the batch size during training achieves similar results to decaying the learning rate, but it reduces the number of parameter updates from just over 14000 to below 6000. We run each experiment twice to illustrate the variance.

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.

Today: Improving Optimization

1. Learning Rate Decay
2. **Learning Rate Schedulers in PyTorch**
3. Training with “momentum”
4. ADAM: Adaptive Learning Rates
5. Optimization algorithms in PyTorch

Learning Rate Schedulers 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 Schedulers 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 Schedulers 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

Reproducibility in DL is tough

Need to report all training details – they all matter.

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. Proceedings of the IEEE conference on computer vision and pattern recognition 2016 (pp. 770-778).

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)

Today: Improving Optimization

1. Learning Rate Decay
2. Learning Rate Schedulers in PyTorch
3. **Training with “momentum”**
4. ADAM: Adaptive Learning Rates
5. Optimization algorithms in PyTorch

Training with “Momentum”

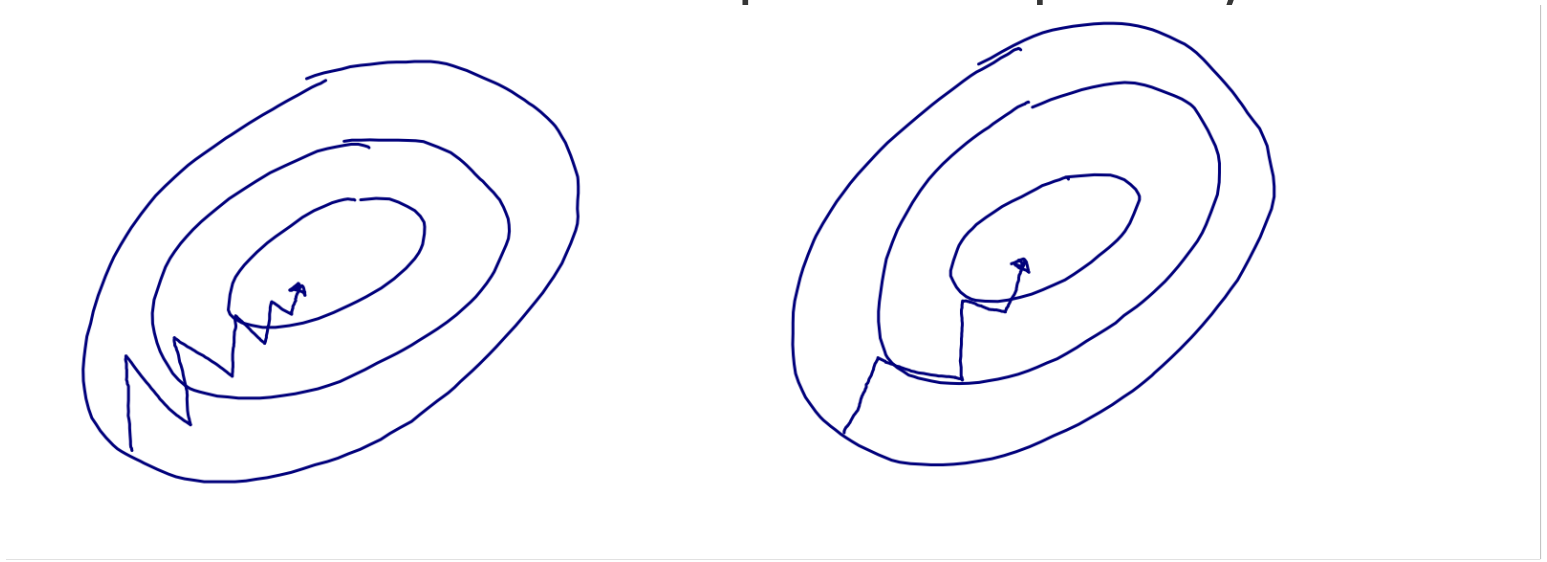
更新时参考历史更新方向，可以平滑掉由噪音梯度引起的震荡，还能在正确的方向上加速

- Main idea: Let’s dampen oscillations by using “velocity” (the speed of the “movement” from previous updates)

https://www.asherworldturns.com/zorbing-new-zealand/

Training with “Momentum”

- Main idea: Let’s dampen oscillations by using “velocity” (the speed of the “movement” from previous updates)



Key take-away: Not only move in the (opposite) direction of the gradient, but also move in the “**weighted averaged**" direction of the last few updates

Training with “Momentum”

Add to the previous one

Often referred to as "velocity" ***V***

"velocity" from the previous iteration

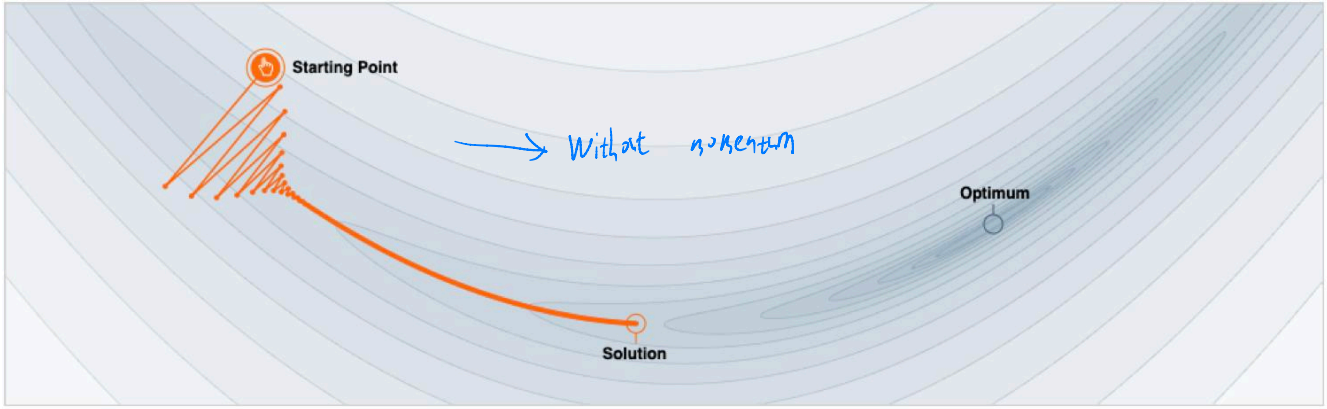
$$\Delta w_{i,j}(t) := \alpha \cdot \Delta w_{i,j}(t-1) + \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

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*

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

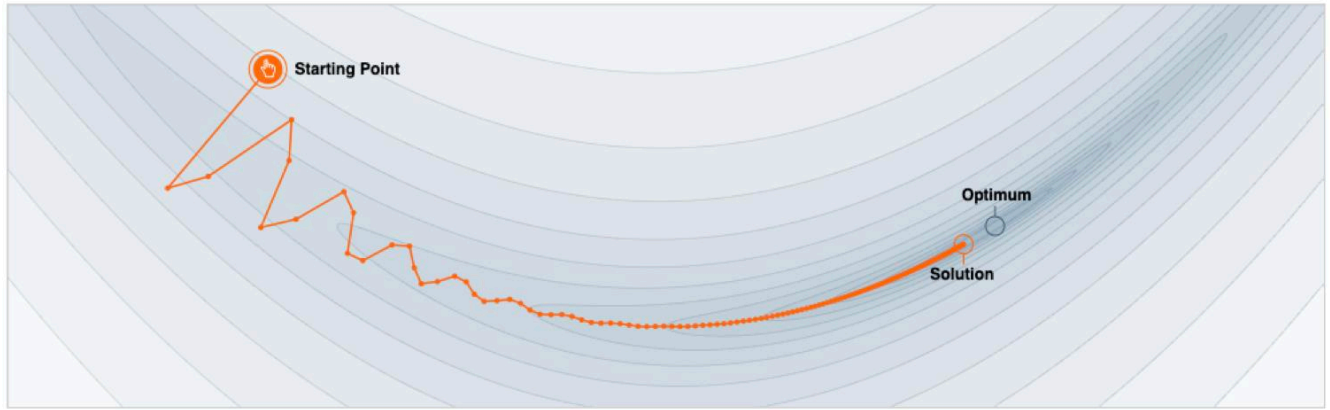
Training with “Momentum”



Step-size α = 0.02 (0, 0.003, 0.006)

Momentum β = 0.0 (0.00, 0.500, 0.990)

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 α = 0.02 (0, 0.003, 0.006)

Momentum β = 0.99 (0.00, 0.500, 0.990)

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/

Momentum in PyTorch

CLASS `torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False)` [SOURCE]

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)

Example

$1 - \eta$

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

Nesterov: A Better Momentum

We already know where the momentum part will push us in this step. Let's calculate the **new gradient** with that update in mind:

Before:

$$\Delta \mathbf{w}_t := \alpha \cdot \Delta \mathbf{w}_{t-1} + \eta \cdot \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}_t)$$
$$\mathbf{w}_{t+1} := \mathbf{w}_t - \Delta \mathbf{w}_t$$

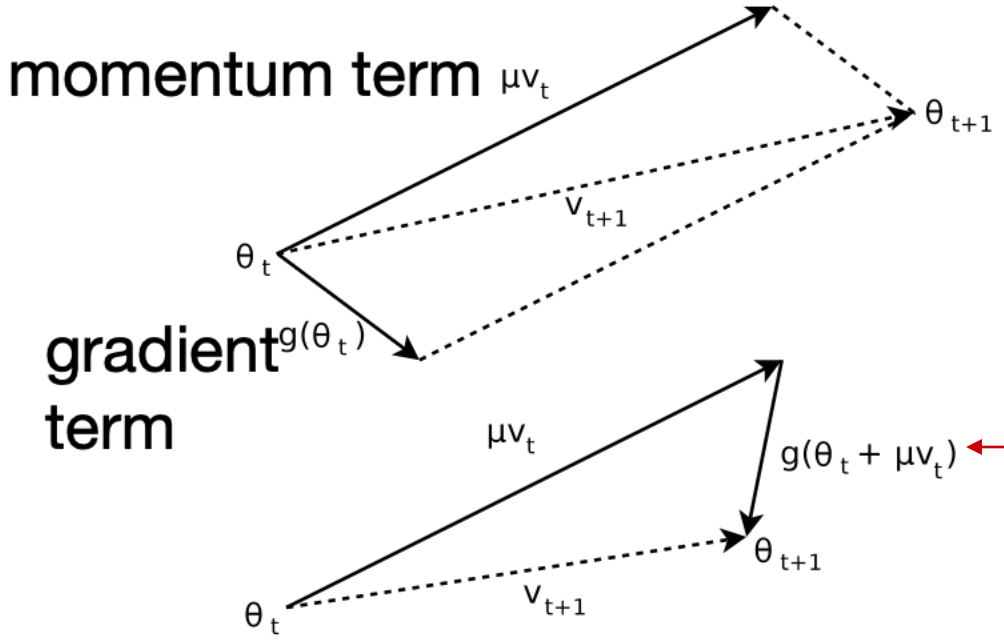
Nesterov:

$$\Delta \mathbf{w}_t := \alpha \cdot \Delta \mathbf{w}_{t-1} + \eta \cdot \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}_t - \alpha \cdot \Delta \mathbf{w}_{t-1})$$
$$\mathbf{w}_{t+1} := \mathbf{w}_t - \Delta \mathbf{w}_t$$

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547.

Sutskever, I., Martens, J., Dahl, G. E., & Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. ICML (3), 28(1139-1147), 5.

Nesterov: A Better Momentum



correction term (gradient of the point where you would have ended up via the standard momentum method)

Figure 1. **(Top)** Classical Momentum **(Bottom)** Nesterov Accelerated Gradient

Sutskever, I., Martens, J., Dahl, G. E., & Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. ICML (3), 28(1139-1147), 5.

Today: Improving Optimization

1. Learning Rate Decay
2. Learning Rate Schedulers in PyTorch
3. Training with “momentum”
4. **ADAM: Adaptive Learning Rates**
5. Optimization algorithms in PyTorch

Adaptive Learning Rates

Many different flavors of adapting the learning rate

Rule of thumb:

1. decrease learning if the gradient changes its direction
2. increase learning if the gradient stays consistent

Adaptive Learning Rates

Rule of thumb:

1. decrease learning if the gradient changes its direction
2. 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

Rule of thumb:

1. decrease learning if the gradient changes its direction
2. increase learning if the gradient stays consistent

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

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

Step 2: Define a local gain (g) for each weight (initialized with g=1)

If gradient is consistent:

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

Else:

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

RMSProp

- Unpublished (but very popular) algorithm by Geoff Hinton
- Based on Rprop [1]
- Very similar to another concept called AdaDelta
- **Main idea:** divide learning rate by an exponentially decreasing moving average of the squared gradients
  - **RMS = "Root Mean Squared"**
  - Takes into account that gradients can vary widely in magnitude
  - Damps oscillations like momentum (in practice, works better)

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

ADAM (Adaptive Moment Estimation)

- Probably the most widely used optimization algorithm in DL
- Combination of momentum + 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-1}

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

RMSProp term:

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

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.

ADAM (Adaptive Moment Estimation)

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: Stepsize
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)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (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.

Many more variants

- Adam with weight decay

  https://arxiv.org/pdf/1711.05101.pdf

- Layer-wise Adaptive Rate Scaling (LARS)

  https://arxiv.org/pdf/1708.03888.pdf

Today: Improving Optimization

1. Learning Rate Decay
2. Learning Rate Schedulers in PyTorch
3. Training with “momentum”
4. ADAM: Adaptive Learning Rates
5. **Optimization algorithms in PyTorch**

Using different optimizers in PyTorch

- Same as for vanilla SGD (which we've seen before)
- More details 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)
```

Remember to save the optimizer state if you are using momentum / ADAM

ADAM in PyTorch

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

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

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.

Parameters:

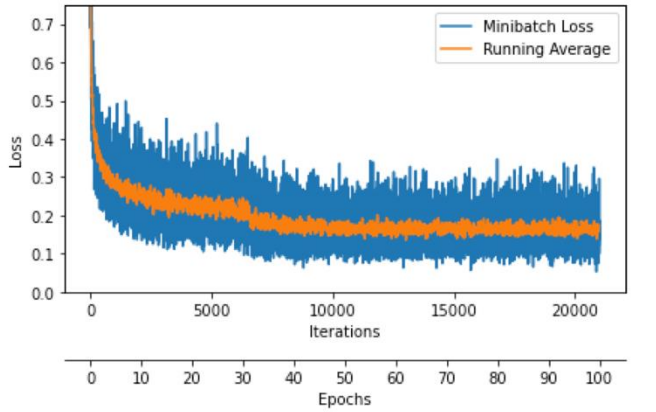
- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)

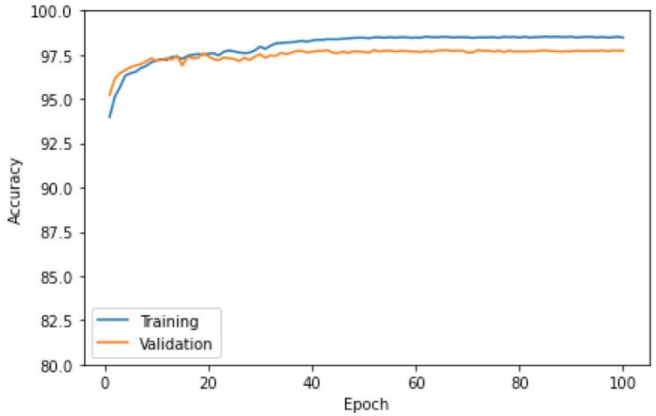
Source: https://pytorch.org/docs/stable/optim.html

ADAM in PyTorch

sgd-scheduler-momentum.ipynb

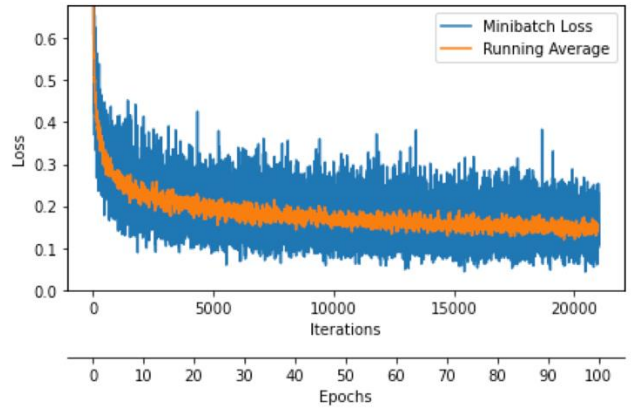
```
Epoch: 100/100 | Train: 98.47% | Validation: 97.73%
Time elapsed: 4.60 min
Total Training Time: 4.60 min
Test accuracy 97.34%
```

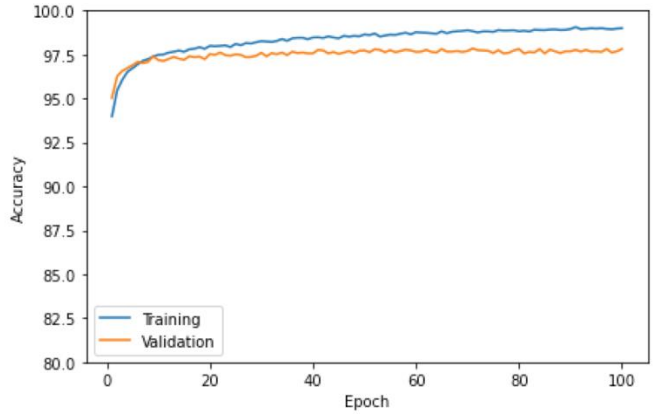




adam.ipynb

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





Optimization is not a settled science

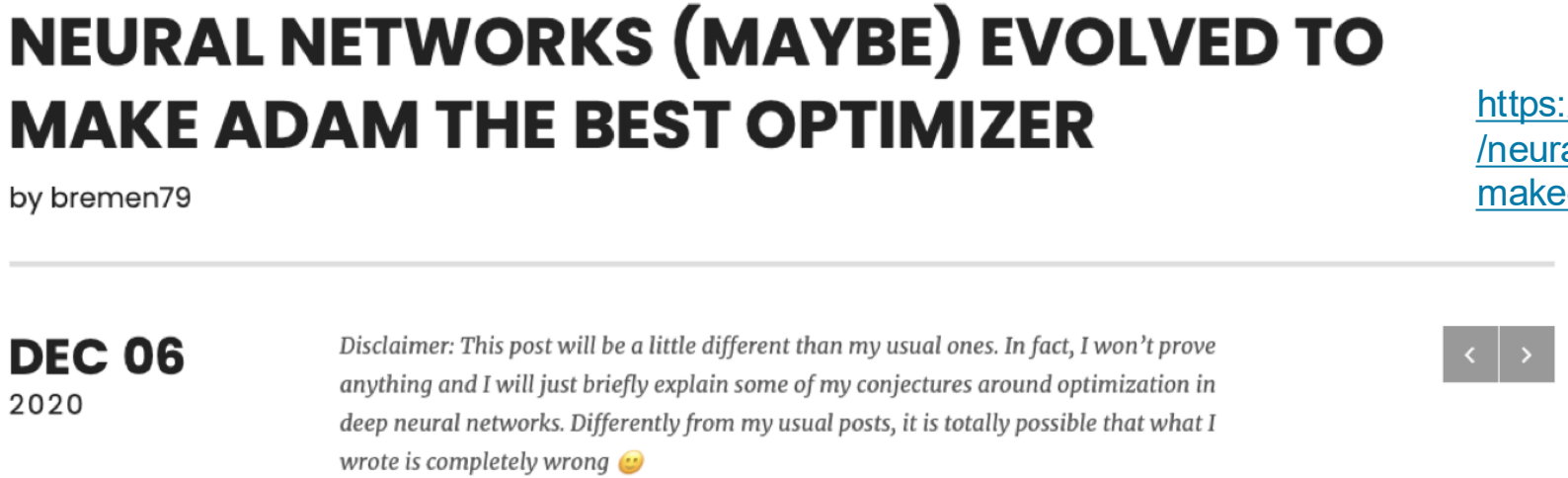
Improving Generalization Performance by Switching from Adam to SGD

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

Optimization is not a settled science



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

by bremen79

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 🙂

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

"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."

Questions?