Lecture 12

Common Optimization Algorithms

STAT 479: Deep Learning, Spring 2019

Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching/stat479-ss2019/

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

<u>Part 1 (before Spring break)</u>

- Input Normalization (BatchNorm, InstanceNorm, GroupNorm, LayerNorm)
- Weight Initialization (Xavier, Kaiming He)

<u>Part 2 (this lecture)</u>

- Learning Rate Decay
- Momentum Learning
- Adaptive Learning

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

Part 1 (before Spring break)

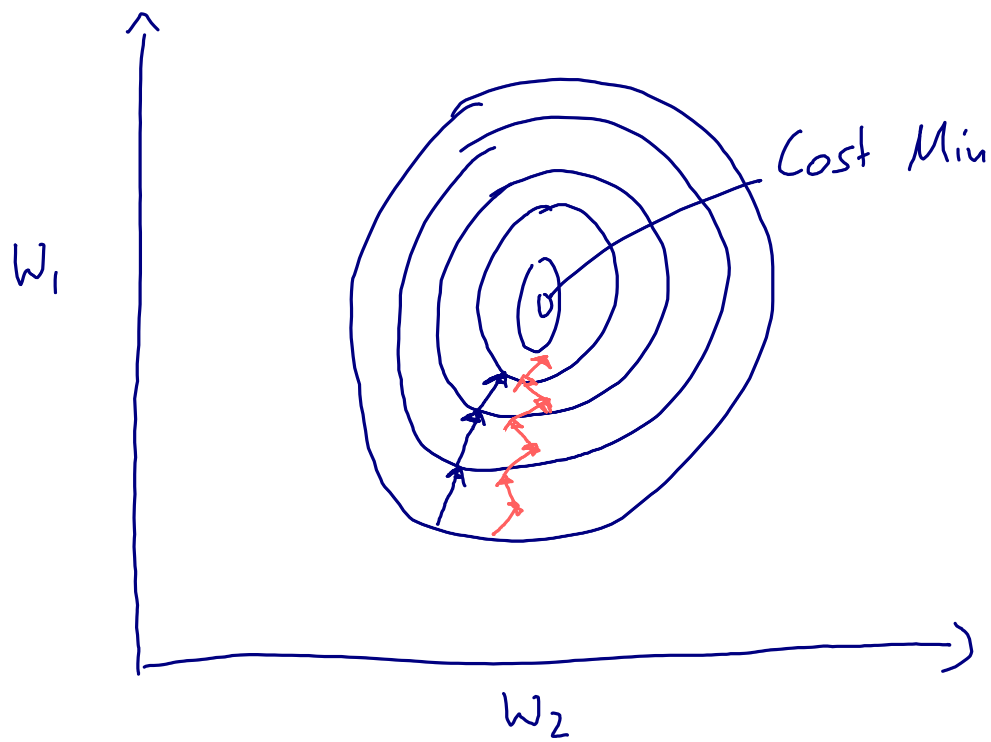
- Input Normalization (BatchNorm, InstanceNorm, GroupNorm, LayerNorm)
- Weight Initialization (Xavier, Kaiming He)

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)

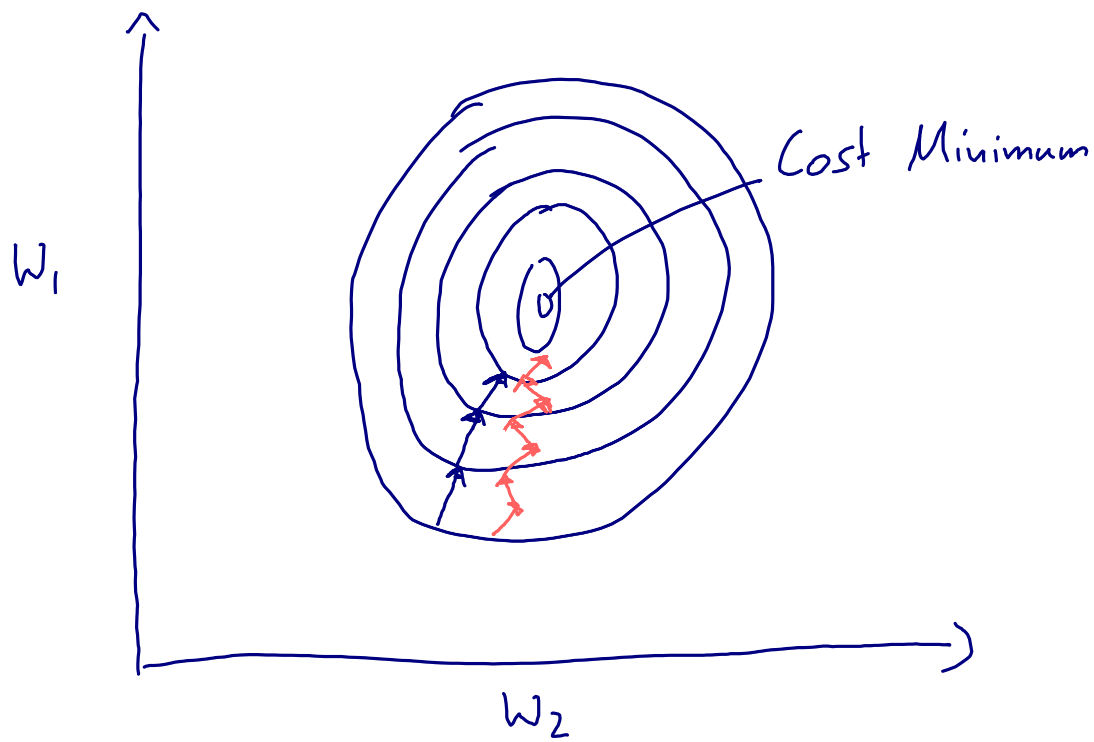
Minibatch Learning Recap



- 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

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

Minibatch Learning Recap



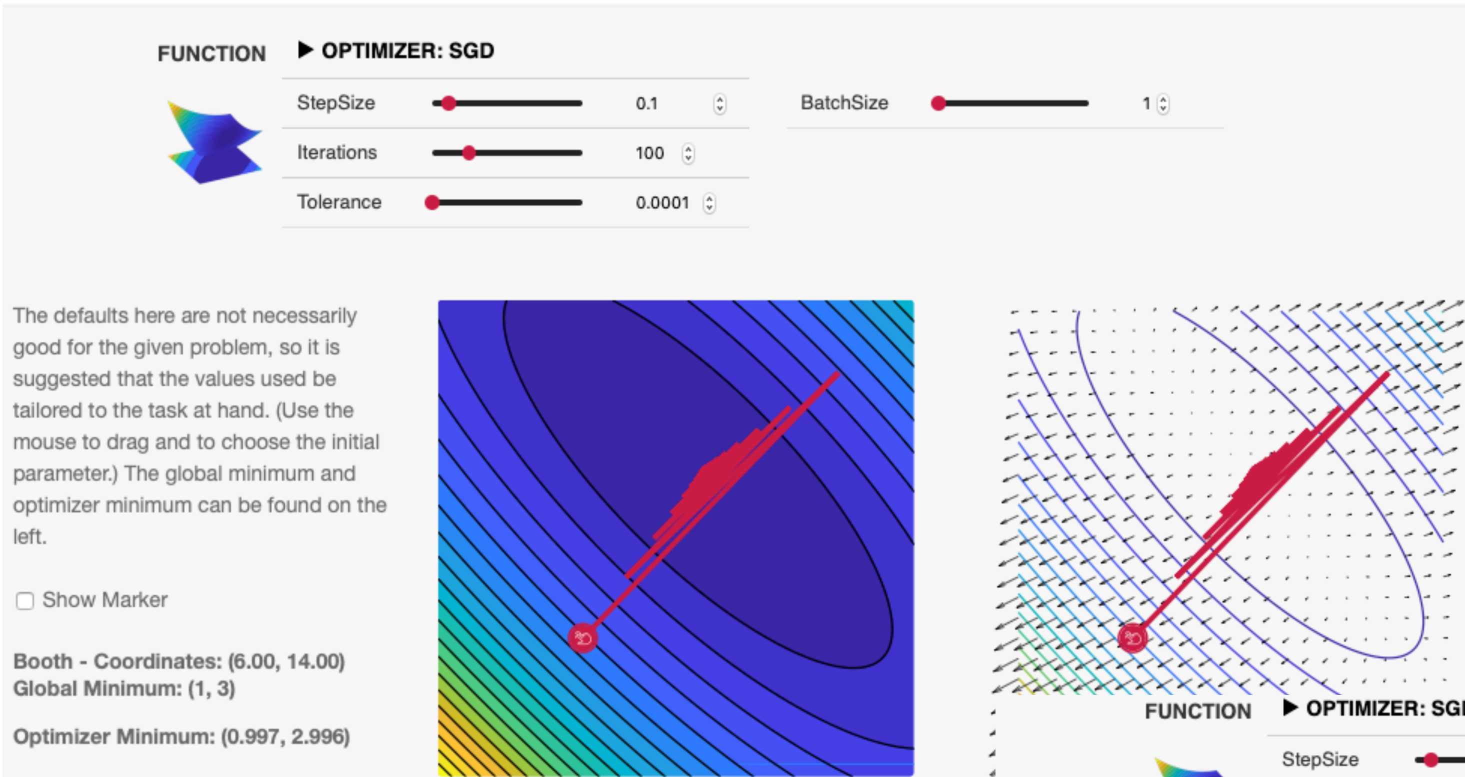
- Minibatch learning is a form of stochastic gradient descent
- Each minibatch can be considered a sample drawn from the training set (where the training set is in turn a sample drawn from the population)
- Hence, the gradient is noisier
- A noisy gradient can be
  - ✦ good: chance to escape local minima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

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

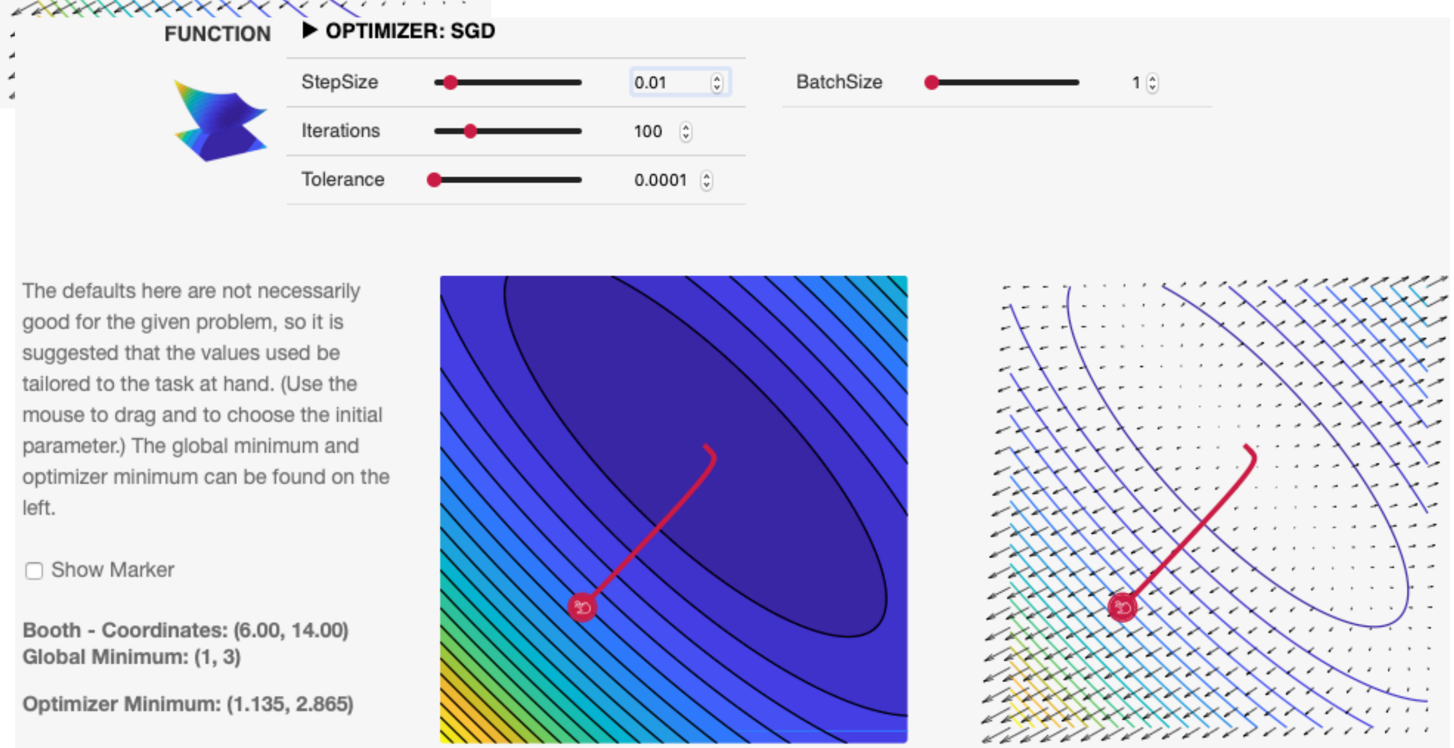
- Note that second order methods that take e.g., gradient curvature into account usually don't work so well in practice and are not often used/recommended in DL

Nice Library & Visualization Tool

https://vis.ensmallen.org



High Learning Rate



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

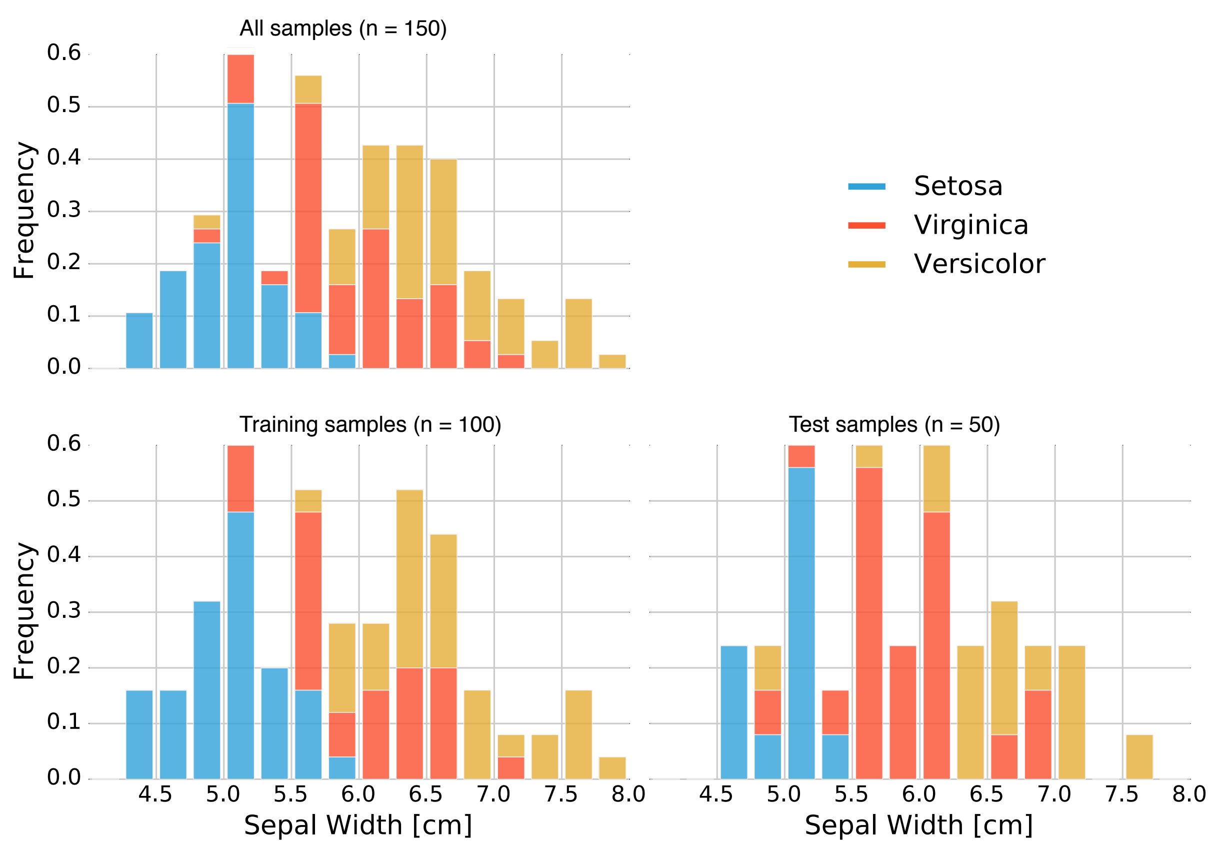
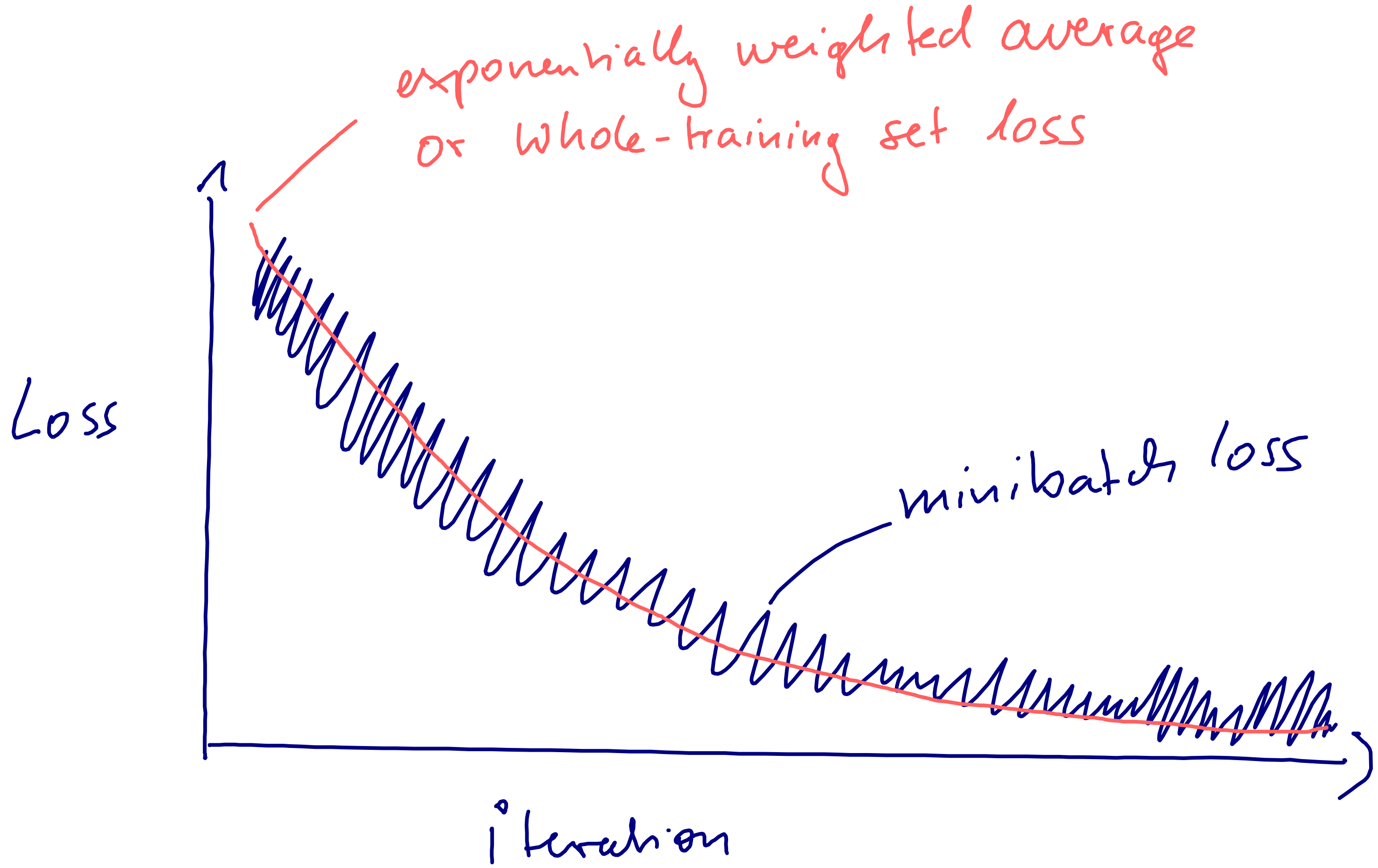


Figure 1: Distribution of *Iris* flower classes upon random subsampling into training and test sets.

Raschka, S. (2018). Model evaluation, model selection, and algorithm selection in machine learning. *arXiv preprint arXiv:1811.12808*.

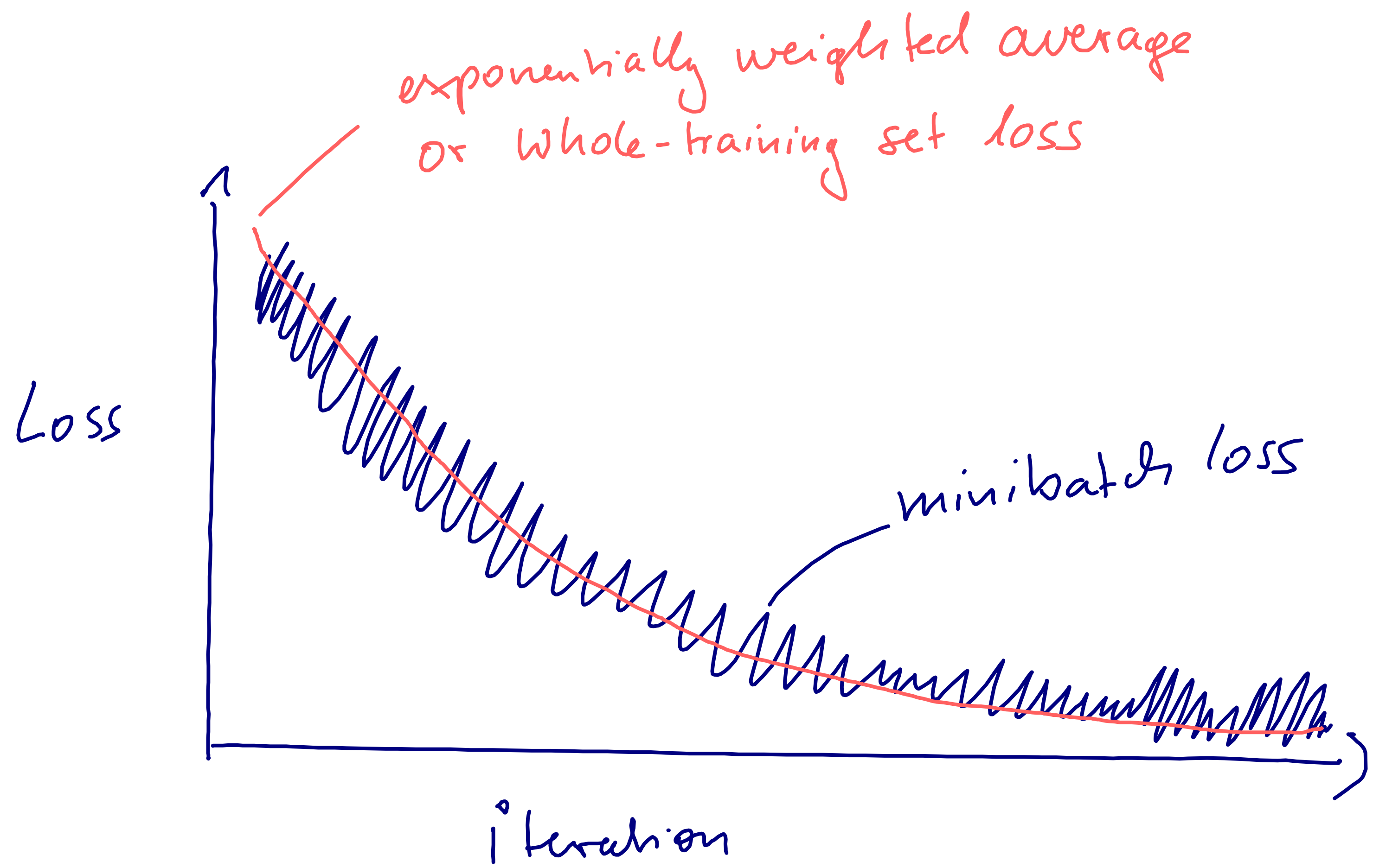
Learning Rate Decay

- 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)

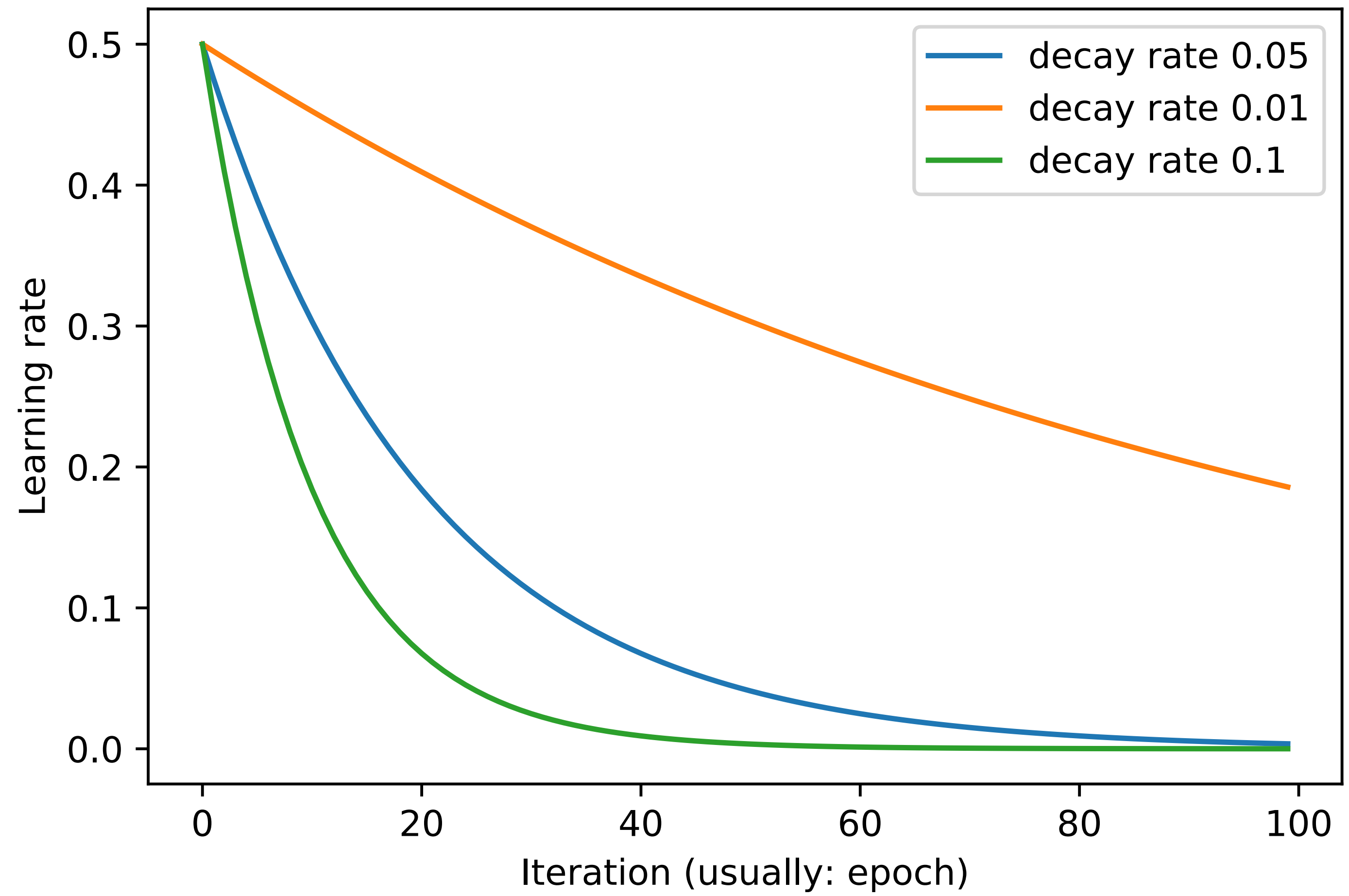
Learning Rate Decay

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

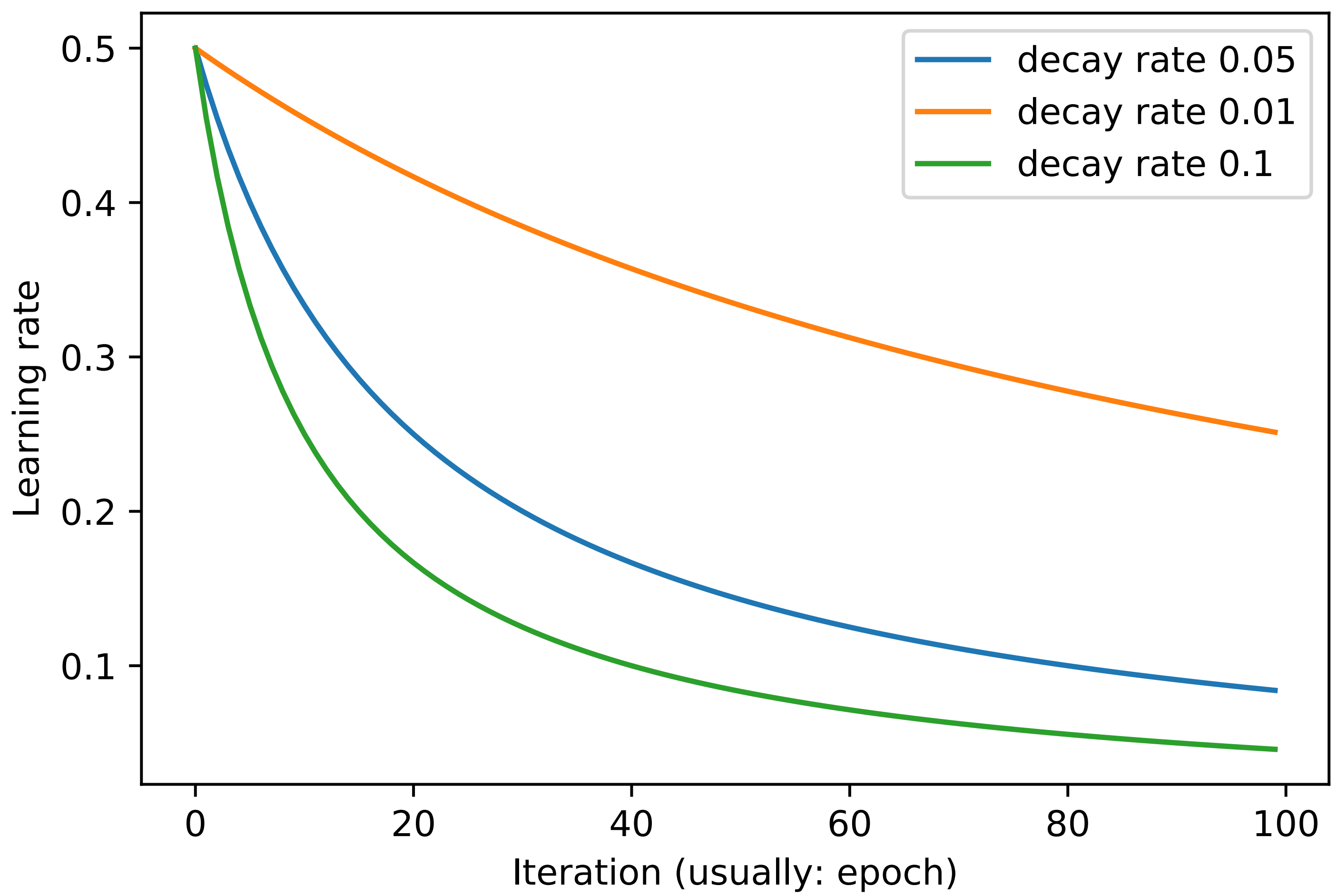
Most common variants for learning rate decay:

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.

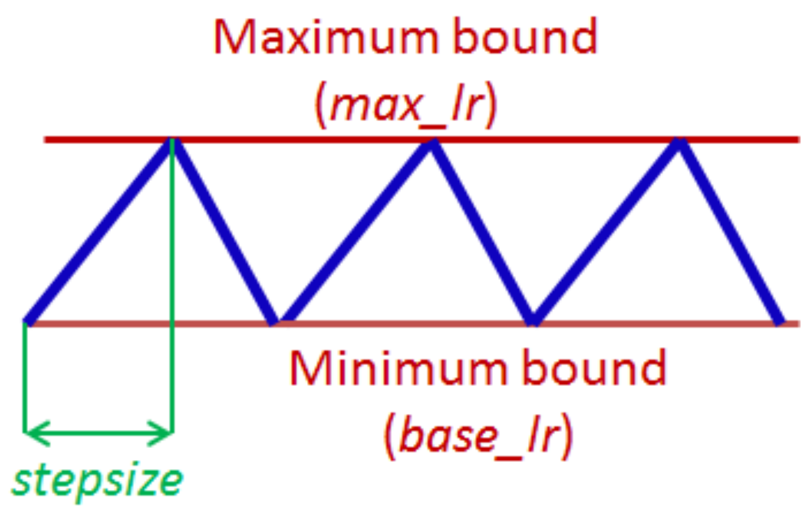


Figure 2. Triangular learning rate policy. The blue lines represent learning rate values changing between bounds. The input parameter *stepsize* is the number of iterations in half a cycle.

(which, I found, didn't work well at all in practice, unfortunately -- at least in my case)

Learning Rate Decay in PyTorch

<u>Option 1.</u> 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,
                                                    last_epoch=-1)

...
```

https://github.com/rasbt/stat479-deep-learning-ss19/tree/master/L12_optim/lr_scheduler_and_saving_models.ipynb

Learning Rate Decay in PyTorch

Example, part 2/2

```
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()
```

https://github.com/rasbt/stat479-deep-learning-ss19/tree/master/L12_optim/lr_scheduler_and_saving_models.ipynb

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)

https://github.com/rasbt/stat479-deep-learning-ss19/tree/master/L12_optim/lr_scheduler_and_saving_models.ipynb

Weight Initialization Experiments (Last-lecture-follow-up)

https://github.com/rasbt/stat479-deep-learning-ss19/tree/master/L13_intro-cnn/code/cnn-with-diff-init

Uniform: Test accuracy 97.63%

```
def weights_init(m):
    if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        torch.nn.init.uniform_(m.weight.detach(), -0.1, 0.1)
        torch.zero_(m.bias.detach())

model.apply(weights_init)
```

Normal: Test accuracy 97.76%

```
def weights_init(m):
    if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        torch.nn.init.normal_(m.weight.detach(), mean=0, std=0.1)
        torch.zero_(m.bias.detach())

model.apply(weights_init)
```

Default: Test accuracy 97.77%

Xavier Normal: Test accuracy 97.69%

```
def weights_init(m):
    if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        torch.nn.init.xavier_normal_(m.weight)
        torch.zero_(m.bias.detach())

model.apply(weights_init)
```

Xavier Uniform: Test accuracy 97.36%

```
def weights_init(m):
    if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        torch.nn.init.xavier_uniform_(m.weight)
        torch.zero_(m.bias.detach())

model.apply(weights_init)
```

He Normal: Test accuracy 97.67%

```
def weights_init(m):
    if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        torch.nn.init.kaiming_normal_(m.weight)
        torch.zero_(m.bias.detach())

model.apply(weights_init)
```

He Uniform: Test accuracy 97.54%

```
def weights_init(m):
    if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        torch.nn.init.kaiming_uniform_(m.weight)
        torch.zero_(m.bias.detach())

model.apply(weights_init)
```

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 (disambigua

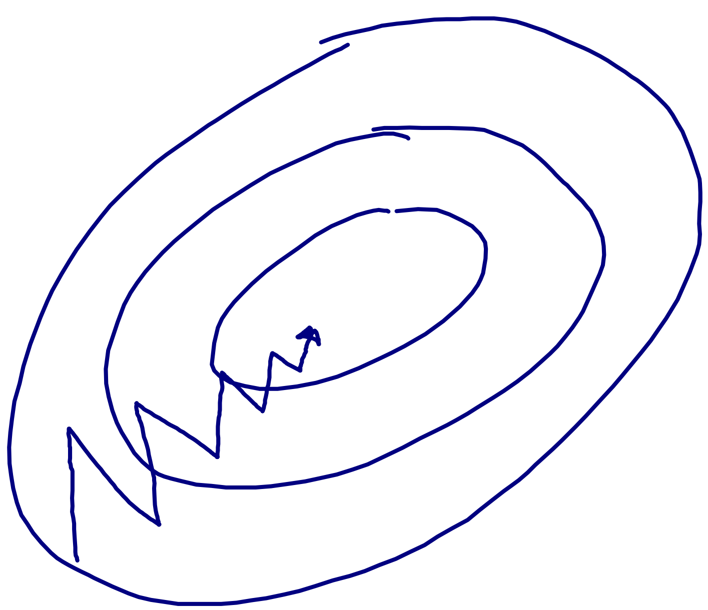
In Newtonian mechanics, **linear momentum**, **translational momentum**, or simply **momentum** (pl. momenta) is the product of the mass and velocity of an object. It is a vector quantity, possessing a magnitude and a direction in three-dimensional space. If m is an object's mass and $\mathbf{v}$ is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

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

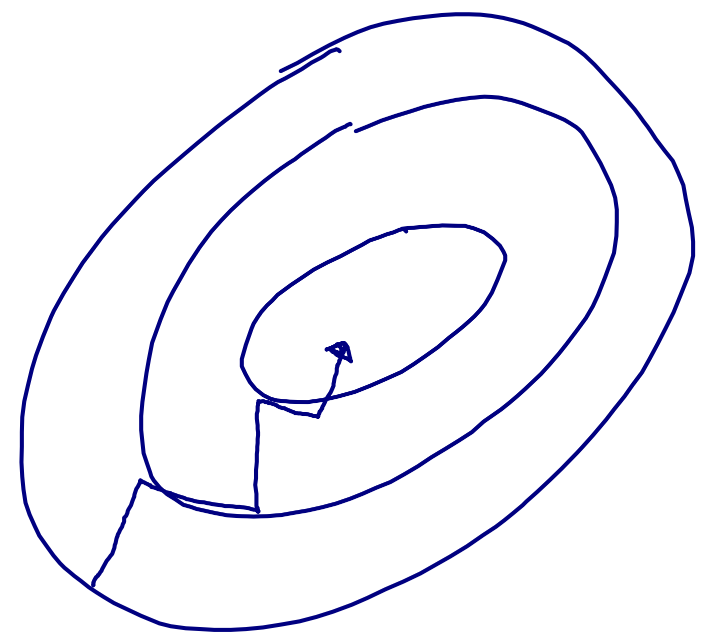
- Momentum is a jargon term in DL and is probably a misnomer in this context
- Concept: In momentum learning, we try to accelerate convergence by dampening oscillations using "velocity" (the speed of the "movement" from previous updates)

Training with "Momentum"

- Momentum is a jargon term in DL and is probably a misnomer in this context
- Concept: In momentum learning, we try to accelerate convergence by dampening oscillations using "velocity" (the speed of the "movement" from previous updates)

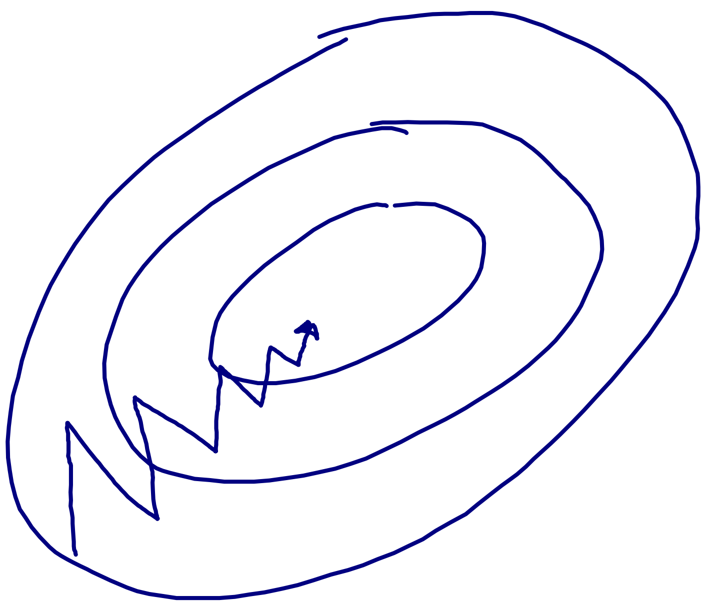


Without momentum

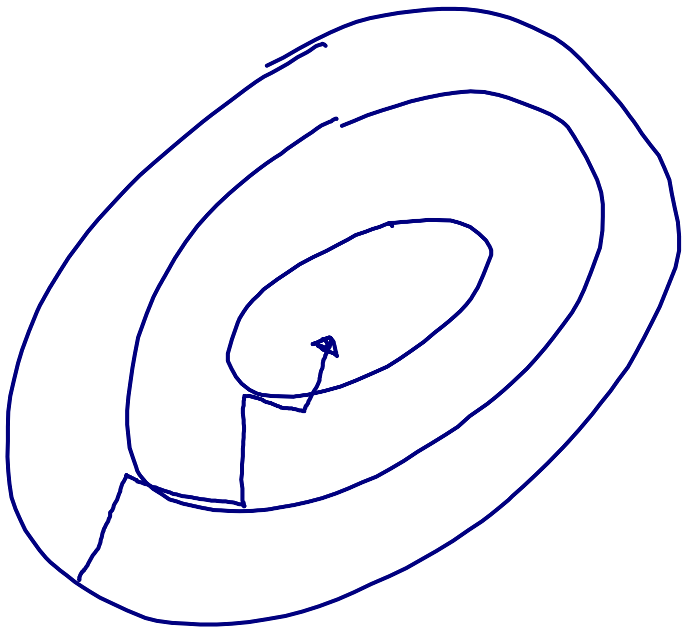


With momentum

Training with "Momentum"



Without momentum



With momentum

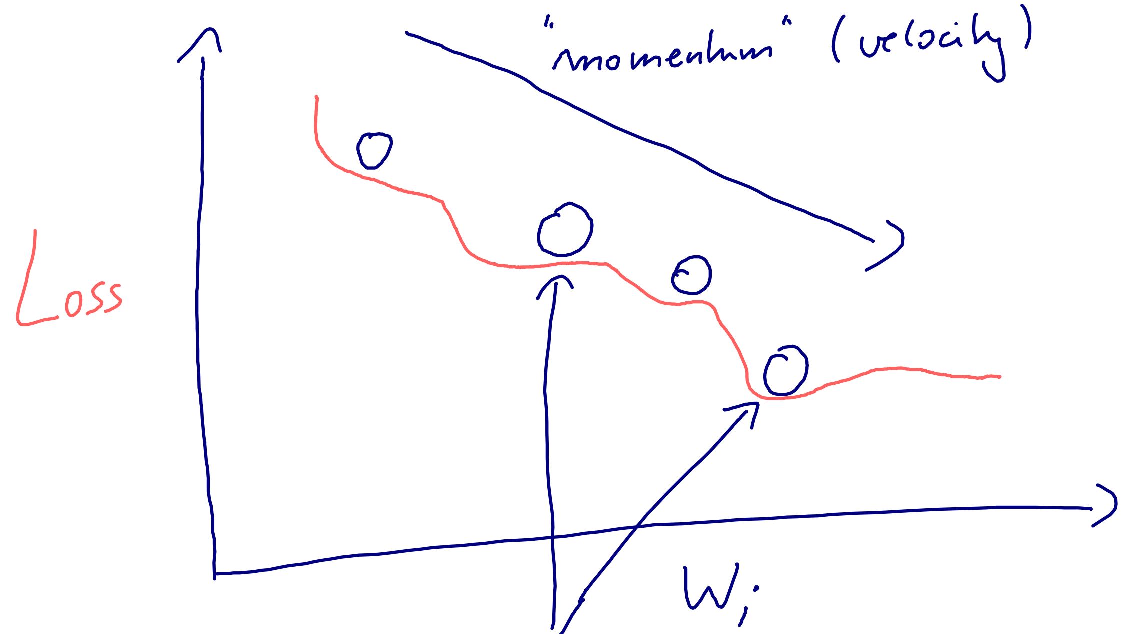
Key take-away:

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"

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

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



avoid getting stuck on flat surfaces (saddle points) and/or local minima in general

Training with "Momentum"

Often referred to as "velocity" v

"velocity" from the previous iteration

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

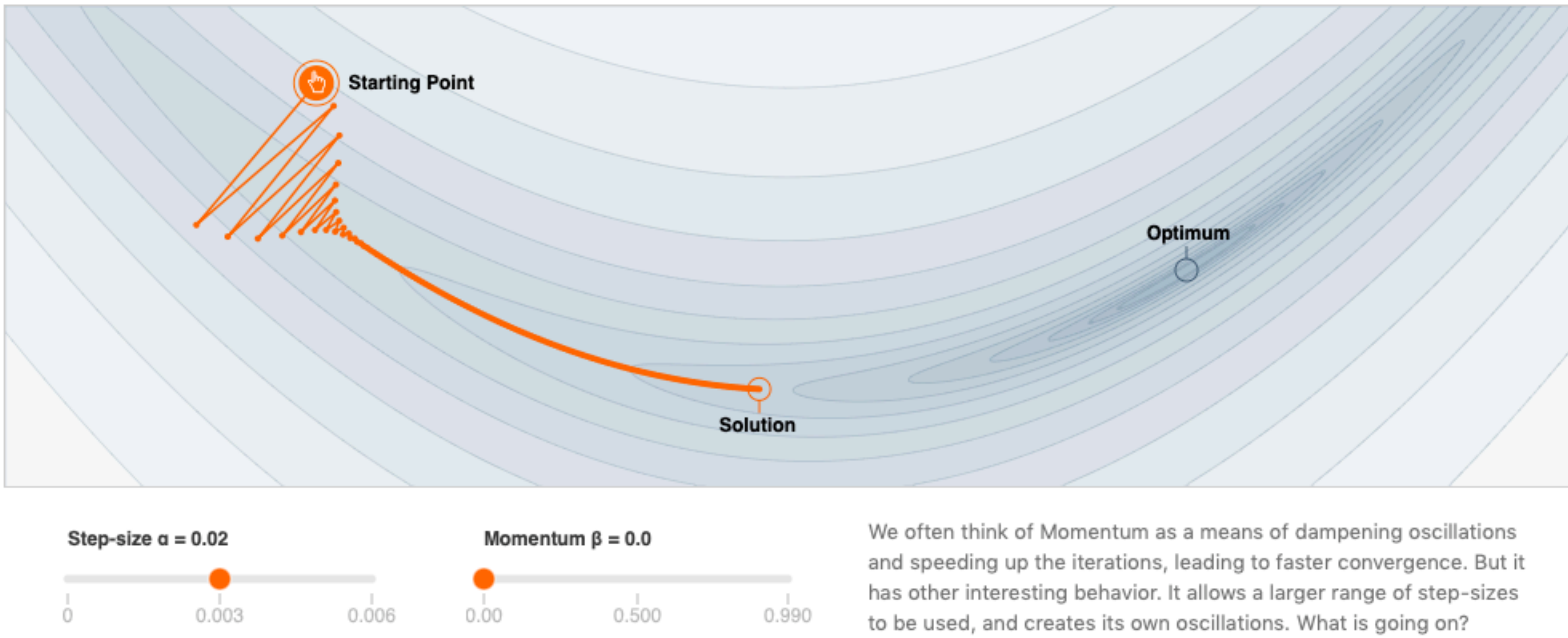
Usually, we choose a momentum rate between 0.9 and 0.999; you can think of it as a "friction" or "dampening" parameter

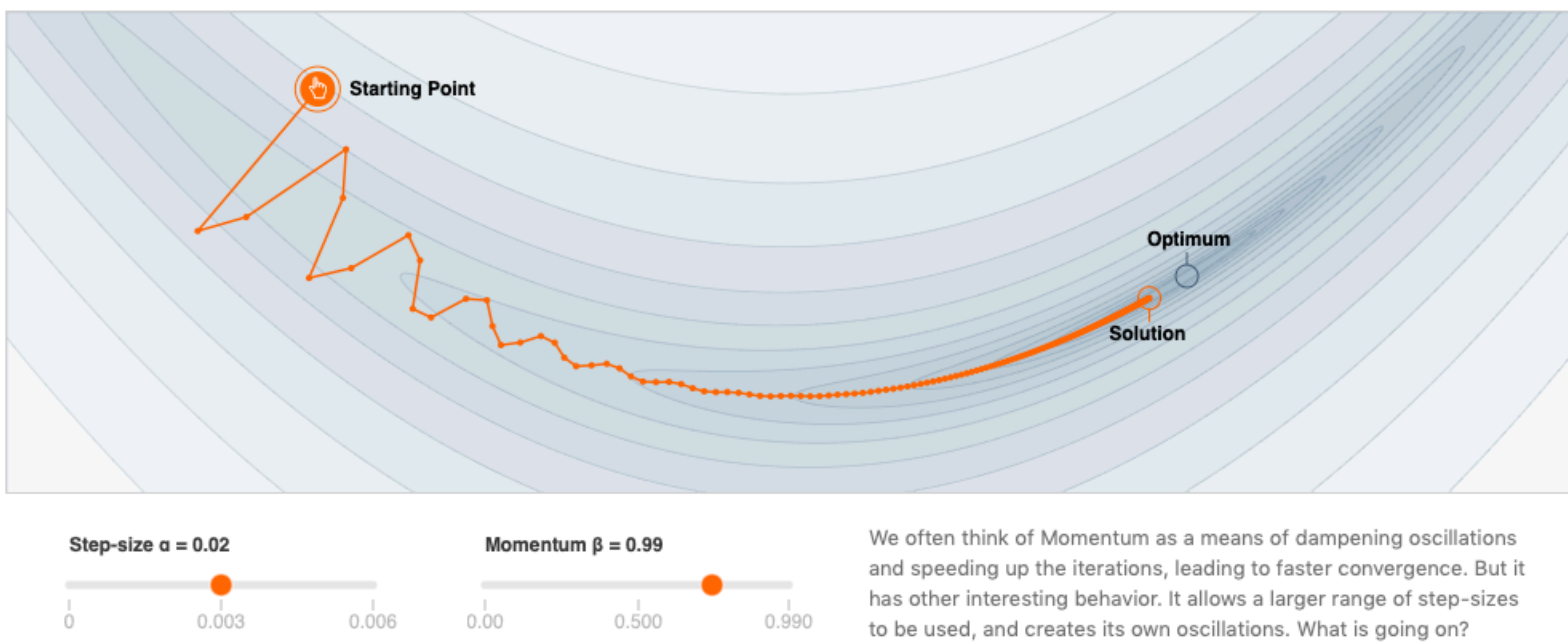
Regular partial derivative/ gradient multiplied by learning rate at current time step t

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





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

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

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

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

Note that the optional "dampening" term is used as follows:

```
v = momentum * v + (1-dampening) * gradientW
W = W - lr * v
```

Also note that in PyTorch, the learning rate is also applied to the momentum terms, instead of the original definition, which would be

```
v = momentum * v + (1-dampening) * lr * gradientW
W = W - v
```

A Better Momentum Method: Nesterov Accelerated Gradient

Similar to momentum learning, but performs a correction after the update (based on where the loss, w.r.t. the weight parameters, is approx. going to be after the update)

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:

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

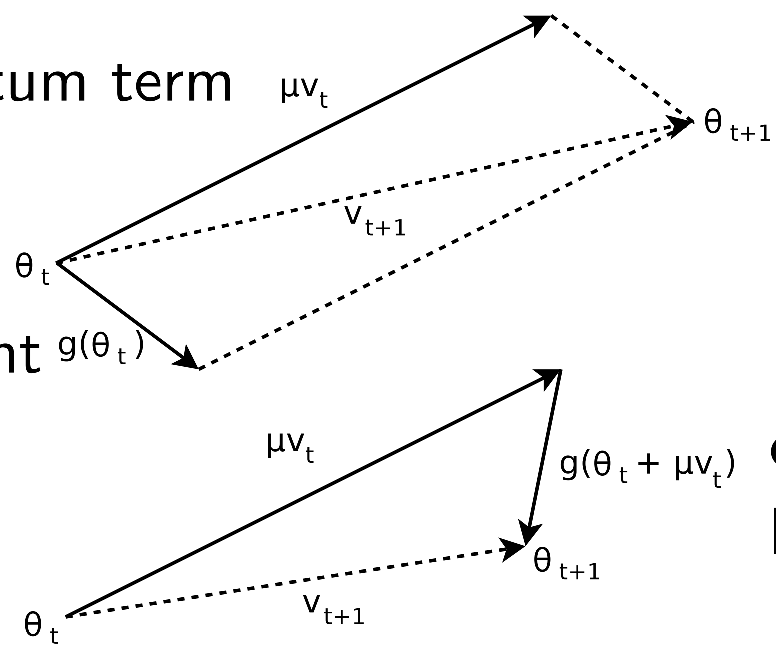
Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547.

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.

A Better Momentum Method: Nesterov Accelerated Gradient

momentum term

gradient term

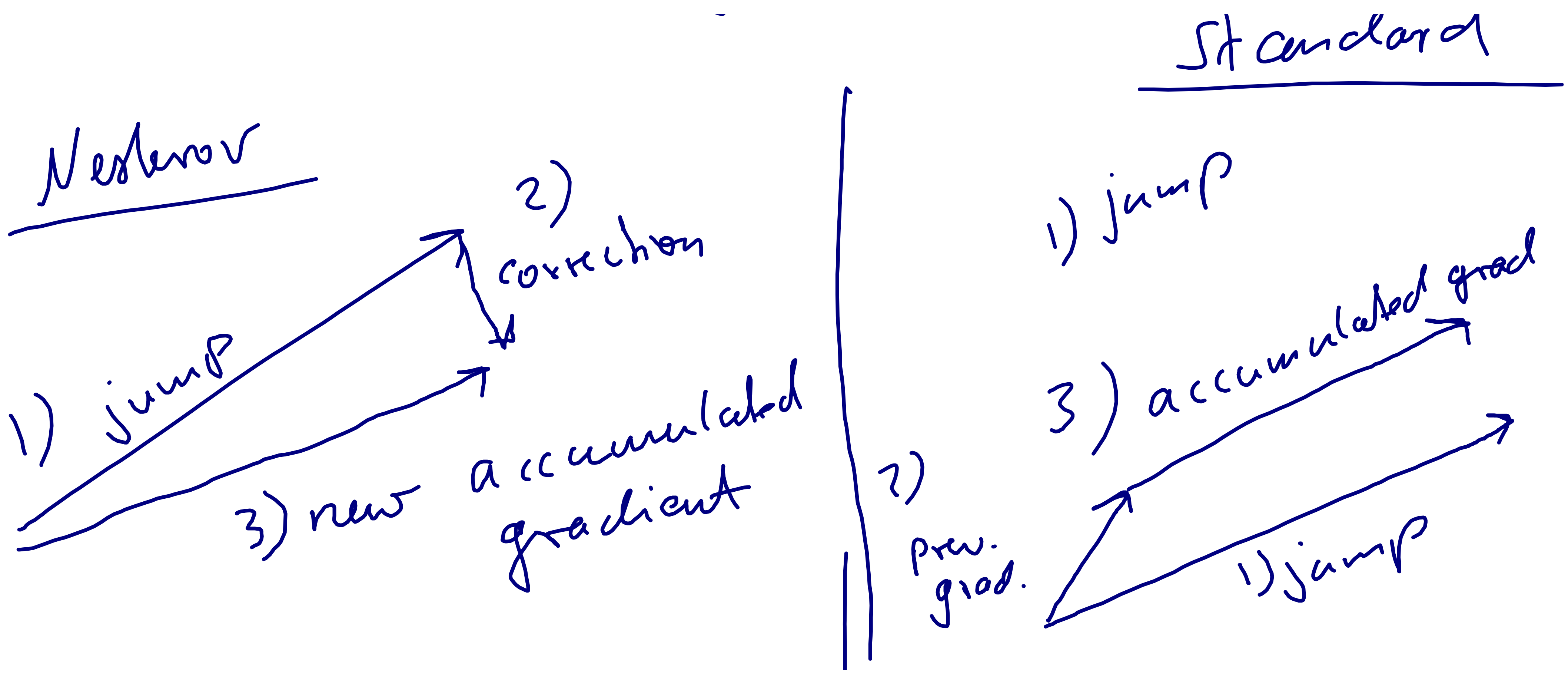


correction term (gradient of the point where you would have ended up via the standard momentum method)

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.

A Better Momentum Method: 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.

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 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 Hüsken. "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

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

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 term:

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

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

RMSProp term:

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

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

Adaptive Learning Rate via ADAM

Momentum term:

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

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

RMSProp term:

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

ADAM update:

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

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: 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*.

Adaptive Learning Rate via ADAM

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

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

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.

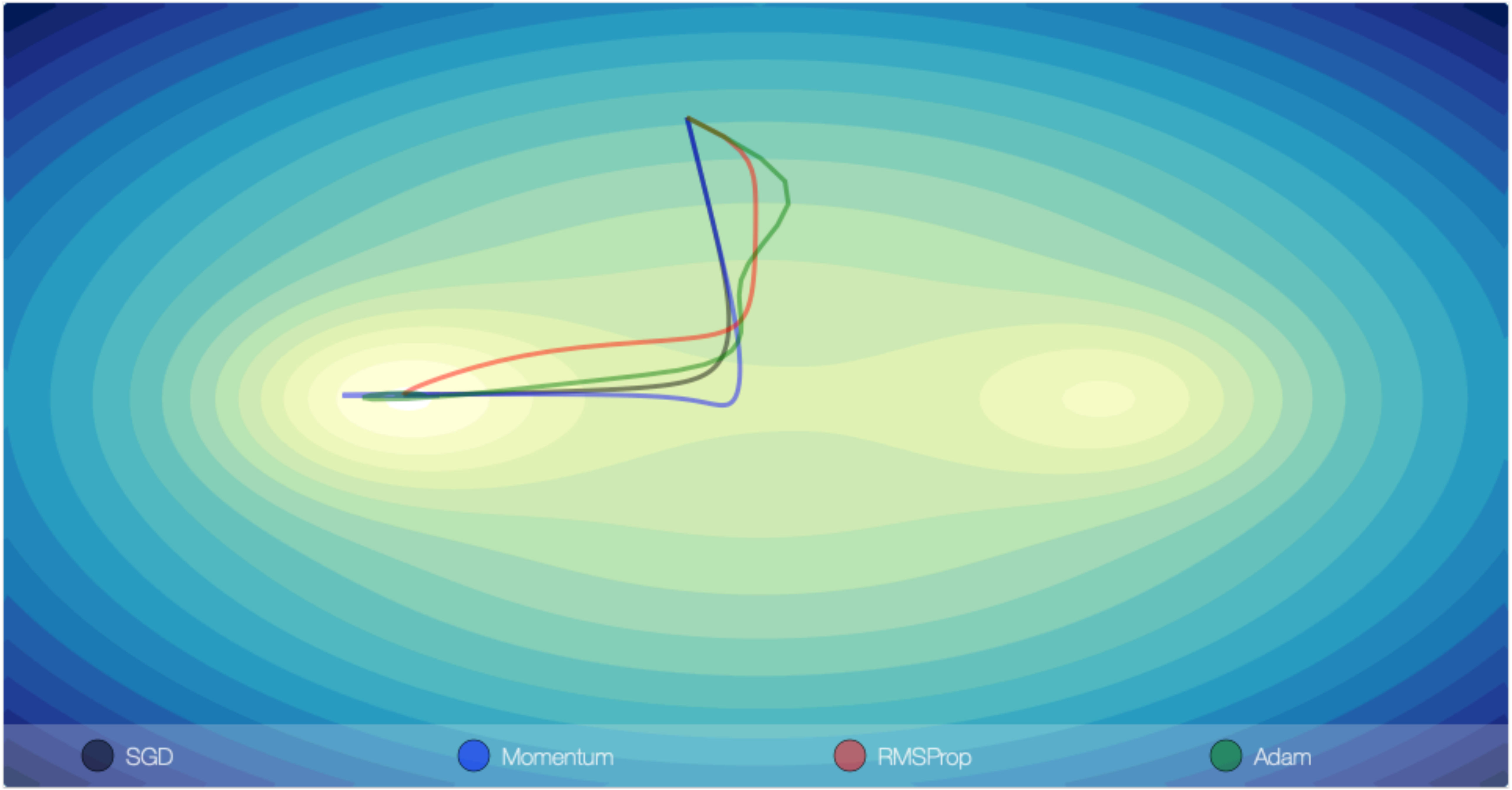
It has been proposed in Adam: A Method for Stochastic Optimization.

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)
- **weight_decay** (*float*, *optional*) – weight decay (L2 penalty) (default: 0)

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

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



https://bl.ocks.org/EmilienDupont/aaf429be5705b219aaaf8d691e27ca87

Using Different Optimizers in PyTorch

Usage is the 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 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)
```

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).

Training Loss vs Generalization Error

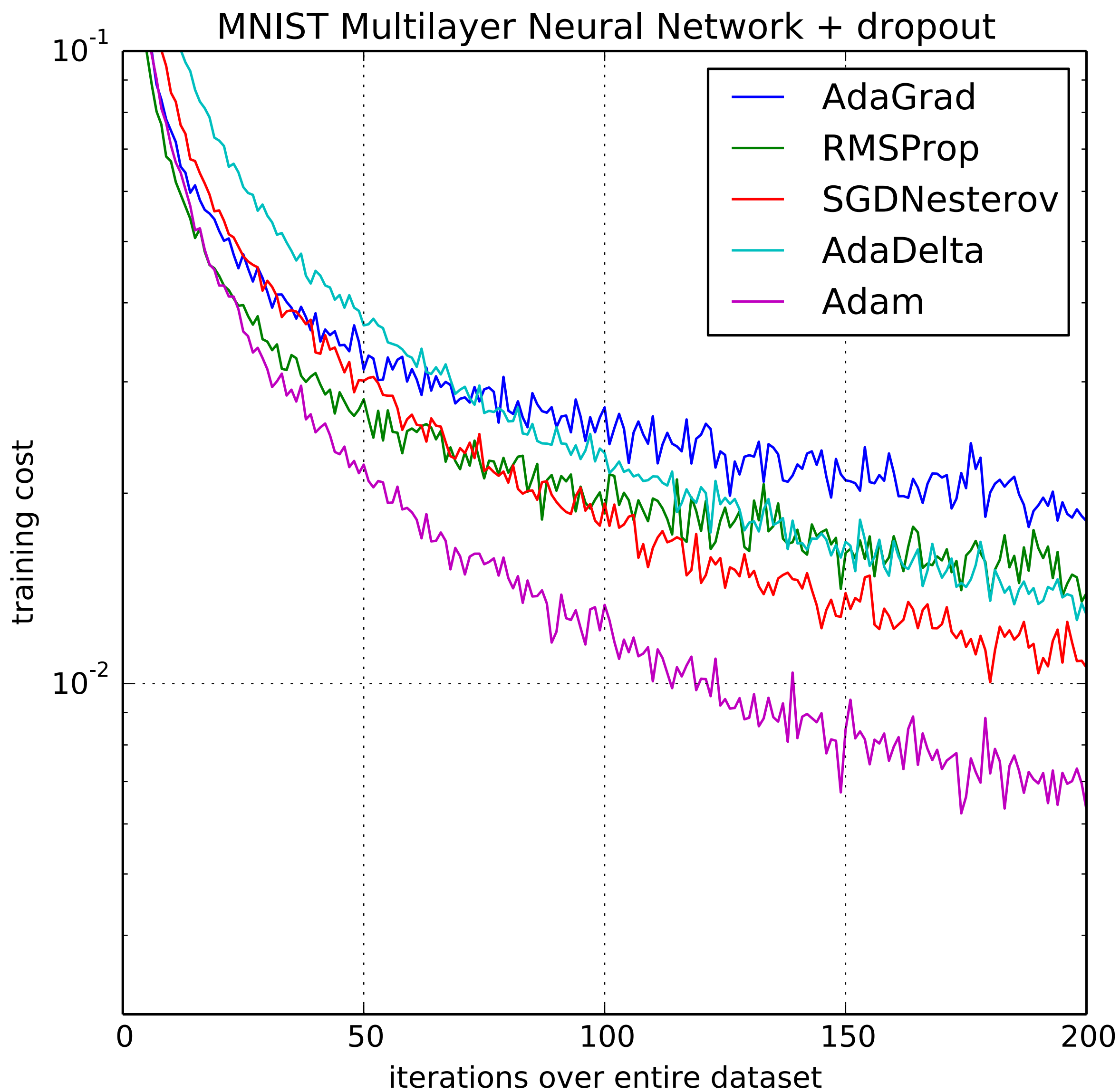
Improving Generalization Performance by Switching from Adam to SGD

Nitish Shirish Keskar, Richard Socher

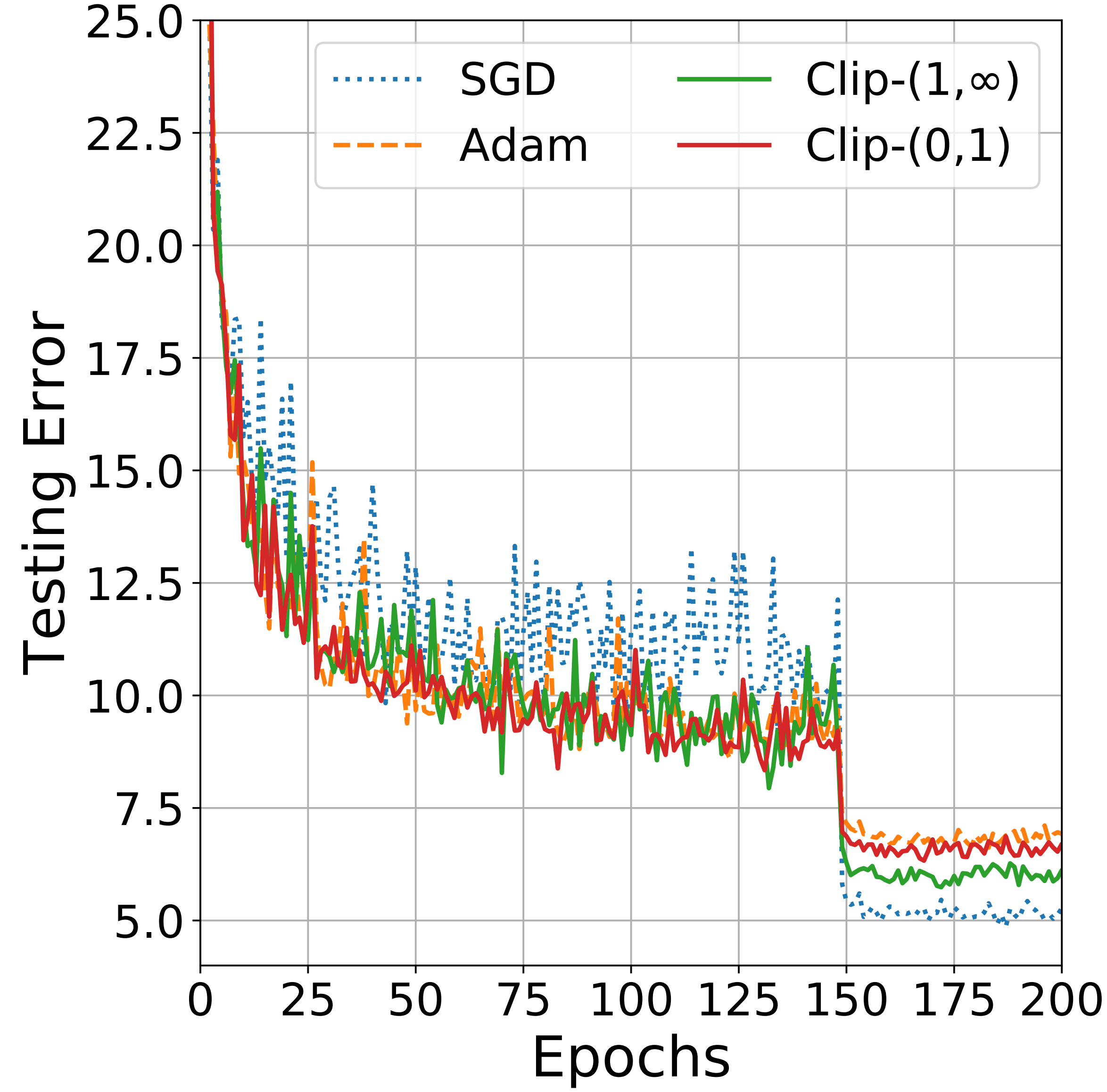
(Submitted on 20 Dec 2017)

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 condition is satisfied. The condition we propose relates to the projection of Adam steps on the gradient subspace. By design, the monitoring process for this condition adds very little overhead and does not increase the number of hyperparameters in the optimizer. We report experiments on several standard benchmarks such as: ResNet, SENet, DenseNet and PyramidNet for the CIFAR-10 and CIFAR-100 data sets, ResNet on the tiny-ImageNet data set and language modeling with recurrent networks on the PTB and WT2 data sets. The results show that our strategy is capable of closing the generalization gap between SGD and Adam on a majority of the tasks.

Training Loss vs Generalization Error



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



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

Reading Assignment

"An overview of gradient descent optimization algorithms" by Sebastian Ruder:
http://ruder.io/optimizing-gradient-descent/index.html