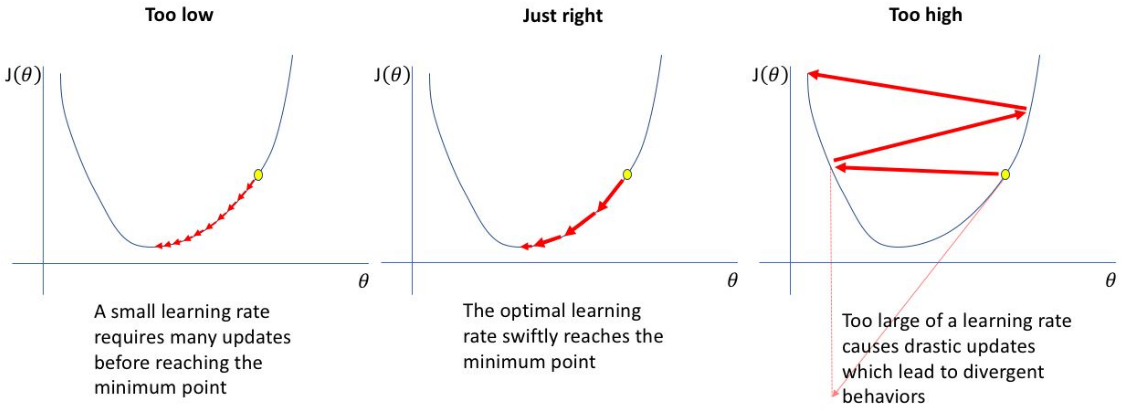
PyTorch

Deep Learning - Spring 2020

Learning Rate

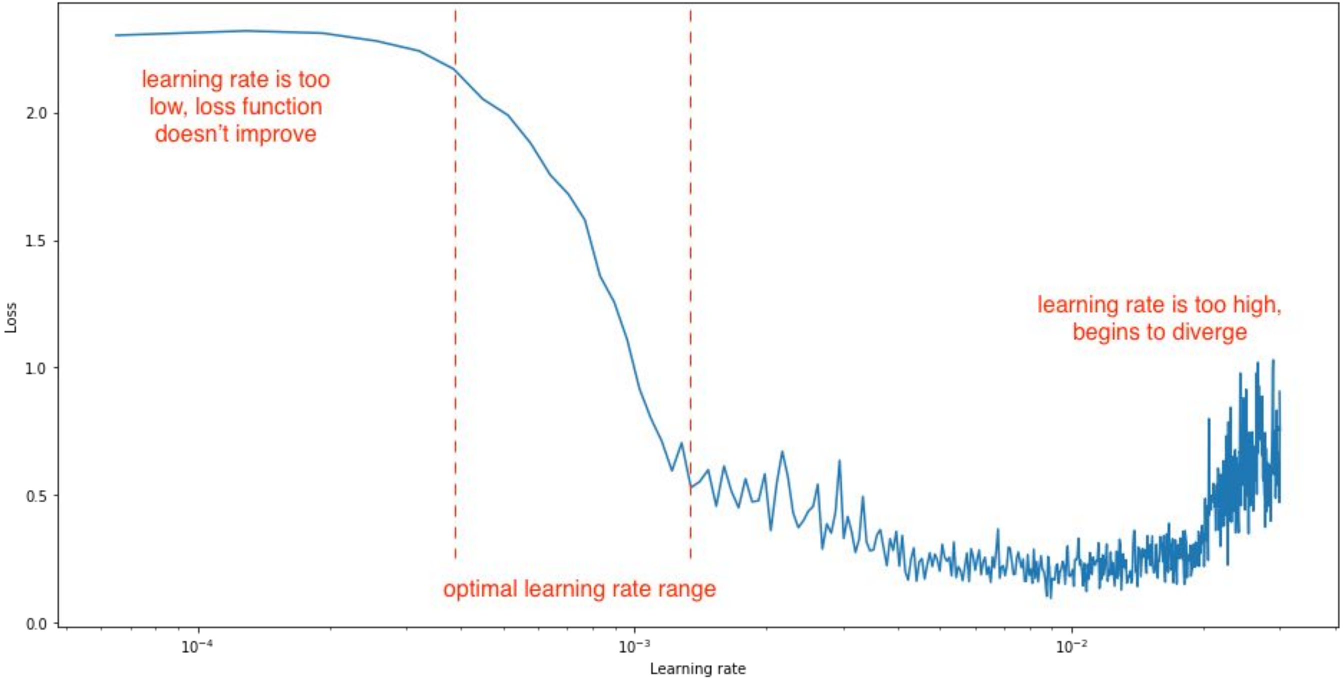
- Learning rate scales the magnitude of our weight updates in order to minimize the network's loss function

- Ultimately, we'd like a learning rate which results is a *steep decrease* in the network's loss



❖ Figure borrowed from [3]

Learning Rate



Learning Rate

- The optimal learning rate will be dependent on the topology of your loss landscape
- dependent on both your model architecture and your dataset
- using a default learning rate may provide decent results
- often improve the performance or speed up training by searching for an optimal learning rate

Andrej Karpathy @karpathy · Nov 24, 2016

3e-4 is the best learning rate for Adam, hands down.

Andrej Karpathy

@karpathy

(i just wanted to make sure that people understand that this is a joke...)

119 11:21 AM - Nov 24, 2016

Learning Rate Decay (Annealing)

Traverse quickly from the initial parameters to a range of "good" parameter values but then we'd like a learning rate small enough to explore the "deeper, but narrower parts of the loss function"

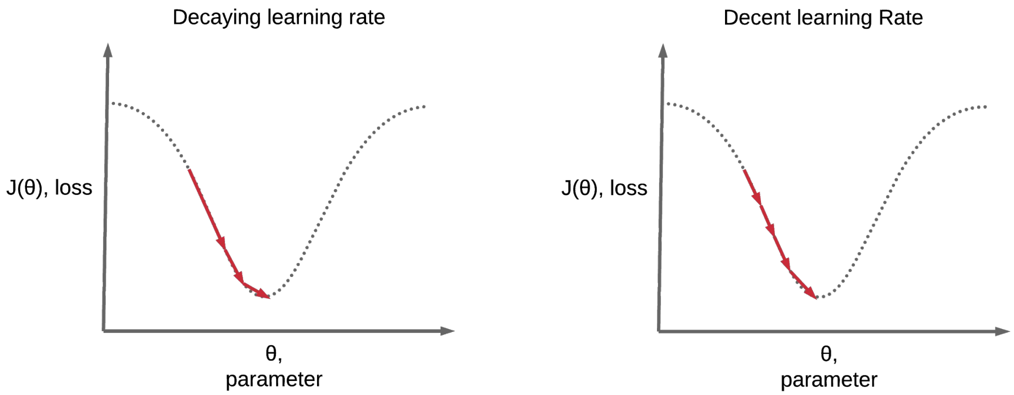


Figure borrowed from [3]

Can Learning Rate Decay Affect Results?

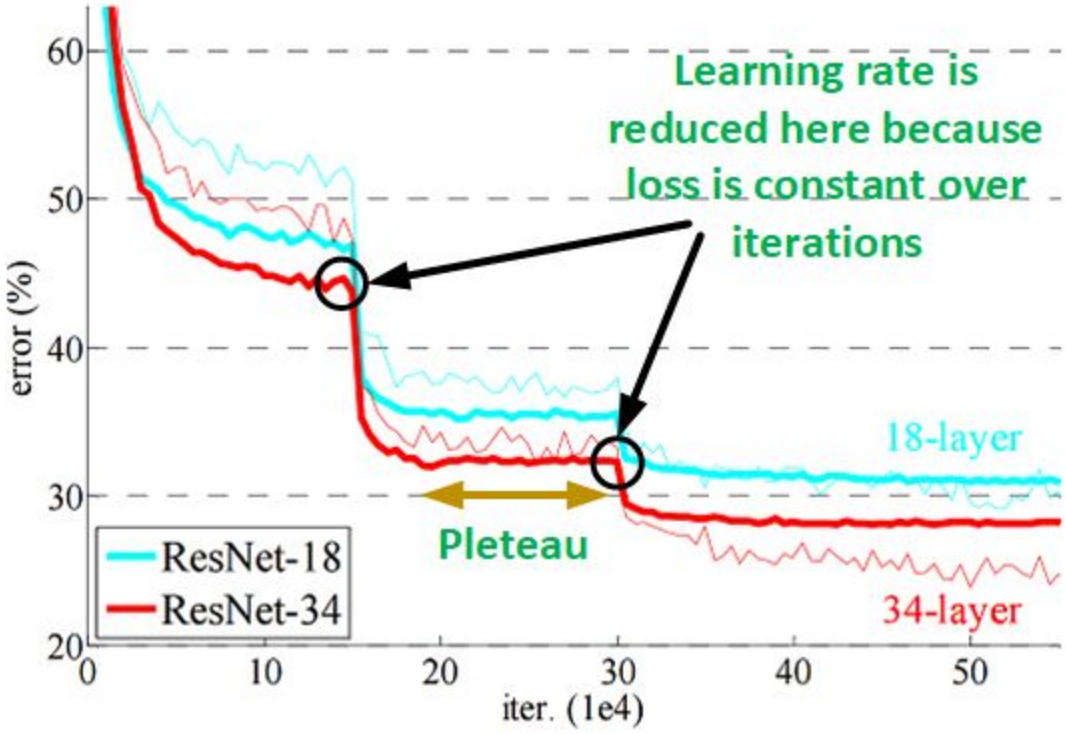


Figure borrowed from http://arxiv.org/abs/1512.03385

Learning Rate Decay Methods

- **Step-based Schedule** changes the learning rate according to some predefined steps

Iteration step

Learning rate at iteration n

$$\eta_n = \eta_0 d^{floor(\frac{1+n}{r})}$$

Decay parameter

Initial learning rate

- **Time-based Schedule** alter the learning rate depending on the learning rate of the previous time iteration

$$\eta_{n+1} = \frac{\eta_n}{1 + dn}$$

Learning Rate Methods

- Exponential schedule are similar to step-based but instead of steps a decreasing exponential function is used

$$\eta_n = \eta_0 e^{-dn}$$

Learning Rate Decay Impl.

```python
def __init__(self, params, defaults):
    torch._C._log_api_usage_once("python.optimizer")
    self.defaults = defaults

    if isinstance(params, torch.Tensor):
        raise TypeError("params argument given to the optimizer should be "
                        "an iterable of Tensors or dicts, but got " +
                        torch.typename(params))

    self.state = defaultdict(dict)
    self.param_groups = []

    param_groups = list(params)
    if len(param_groups) == 0:
        raise ValueError("optimizer got an empty parameter list")
    if not isinstance(param_groups[0], dict):
        param_groups = [{'params': param_groups}]

    for param_group in param_groups:
        self.add_param_group(param_group)
```

Learning Rate Decay Impl.

```
>>> layer = nn.Linear(2, 3)
>>>
>>> layer = nn.Linear(2, 3, bias=True)
>>> optimizer = optim.SGD(layer.parameters(), 1e-2)
>>> optimizer.param_groups[0]
{'params': [Parameter containing:
tensor([[0.0937, 0.6321],
        [0.6721, 0.4741],
        [0.5799, 0.2953]], requires_grad=True), Parameter containing:
tensor([-0.2376, -0.0882, -0.2366], requires_grad=True)], 'lr': 0.01, 'momentum': 0,
'dampening': 0, 'weight_decay': 0, 'nesterov': False}

>>> print(optimizer.param_groups[0].keys())
dict_keys(['params', 'lr', 'momentum', 'dampening', 'weight_decay', 'nesterov'])
```

Learning Rate Decay Impl.

Interlude: See the Updated_Mnist notebook!

Learning Rate Decay (torch.optim.lr_scheduler)

- Learning rate scheduling should be applied after optimizer's update; e.g., you should write your code this way:

```
>>> scheduler = ...
>>> for epoch in range(100):
>>>     train(...)
>>>     validate(...)
>>>     scheduler.step()
```

Learning Rate Decay (torch.optim.lr_scheduler)

torch.optim.lr_scheduler.StepLR(*optimizer*, *step_size*, *gamma=0.1*, *last_epoch=-1*)

Example

```
>>> # Assuming optimizer uses lr = 0.05 for all groups
>>> # lr = 0.05     if epoch < 30
>>> # lr = 0.005    if 30 <= epoch < 60
>>> # lr = 0.0005   if 60 <= epoch < 90
>>> # ...
>>> scheduler = StepLR(optimizer, step_size=30, gamma=0.1)
>>> for epoch in range(100):
>>>     train(...)
>>>     validate(...)
>>>     scheduler.step()
```

Learning Rate Decay (torch.optim.lr_scheduler)

`torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones, gamma=0.1, last_epoch=-1)`

Example

```
>>> # Assuming optimizer uses lr = 0.05 for all groups
>>> # lr = 0.05     if epoch < 30
>>> # lr = 0.005    if 30 <= epoch < 80
>>> # lr = 0.0005   if epoch >= 80
>>> scheduler = MultiStepLR(optimizer, milestones=[30,80], gamma=0.1)
>>> for epoch in range(100):
>>>     train(...)
>>>     validate(...)
>>>     scheduler.step()
```

Learning Rate Decay (torch.optim.lr_scheduler)

- **torch.optim.lr_scheduler.ExponentialLR**(*optimizer*, *gamma*, *last_epoch=-1*)

- **torch.optim.lr_scheduler.ReduceLROnPlateau**(***optimizer***, ***mode='min'***, ***factor=0.1***, ***patience=10***, ***verbose=False***, ***threshold=0.0001***, ***threshold_mode='rel'***, ***cooldown=0***, ***min_lr=0***, ***eps=1e-08***)

Example

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> scheduler = ReduceLROnPlateau(optimizer, 'min')
>>> for epoch in range(10):
>>>     train(...)
>>>     val_loss = validate(...)
>>>     # Note that step should be called after validate()
>>>     scheduler.step(val_loss)
```

Learning Rate Decay (torch.optim.lr_scheduler)

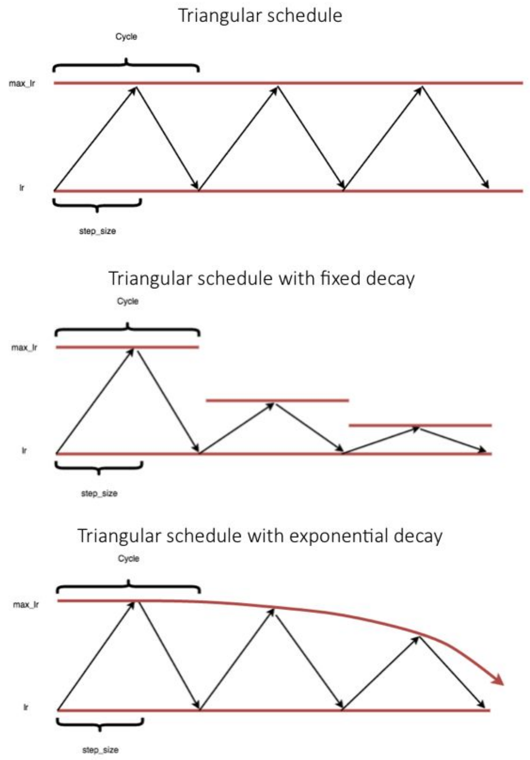
- **torch.optim.lr_scheduler.MultiplicativeLR**(*optimizer*, *lr_lambda*, *last_epoch=-1*)
  - Multiply the learning rate of each parameter group by the factor given in the specified function.

- **torch.optim.lr_scheduler.CyclicLR**(*optimizer*, *base_lr*, *max_lr*, *step_size_up=2000*, *step_size_down=None*, *mode='triangular'*, *gamma=1.0*, *scale_fn=None*, *scale_mode='cycle'*, *cycle_momentum=True*, *base_momentum=0.8*, *max_momentum=0.9*, *last_epoch=-1*)
  - The policy cycles the learning rate between two boundaries with a constant frequency, as detailed in the paper Cyclical Learning Rates for Training Neural Networks. The distance between the two boundaries can be scaled on a per-iteration or per-cycle basis

Learning Rate Decay (torch.optim.lr_scheduler)

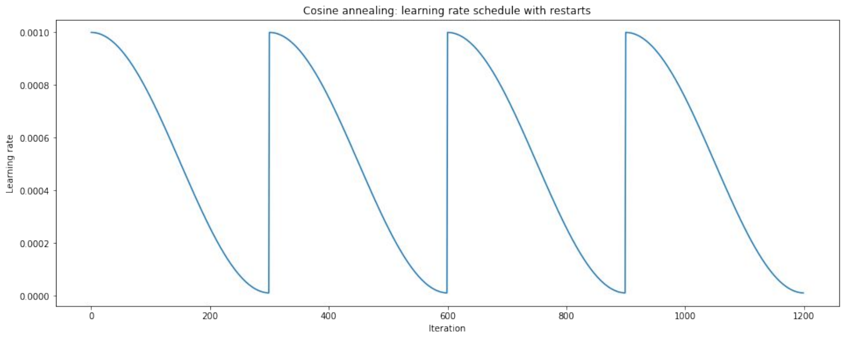
- `torch.optim.lr_scheduler.CyclicLR(...)`

- `torch.optim.lr_scheduler.OneCycleLR(...)`
  anneals the learning rate from an initial learning rate to some maximum learning rate and then from that maximum learning rate to some minimum learning rate much lower than the initial learning rate (Special case of the above scheduler).



Learning Rate Decay (torch.optim.lr_scheduler)

`torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer, T_0, T_mult=1, eta_min=0, last_epoch=-1)`



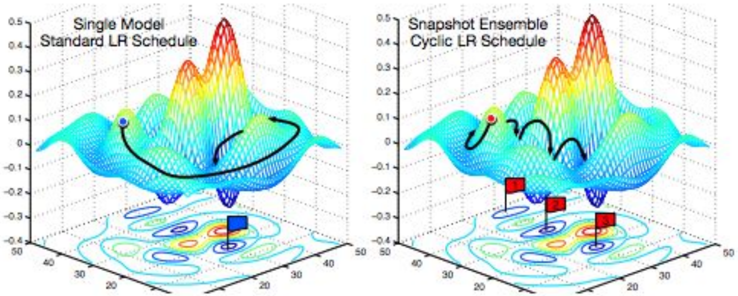


Figure 1: Left: Illustration of SGD optimization with a typical learning rate schedule. The model converges to a minimum at the end of training. **Right:** Illustration of Snapshot Ensembling. The model undergoes several learning rate annealing cycles, converging to and escaping from multiple local minima. We take a snapshot at each minimum for test-time ensembling.

Figure borrowed from [6]

Adaptive Learning Algorithms

Adagrad, Adadelta, RMSProp, Adam do not require the effort of hyper-parameter tuning for scheduling

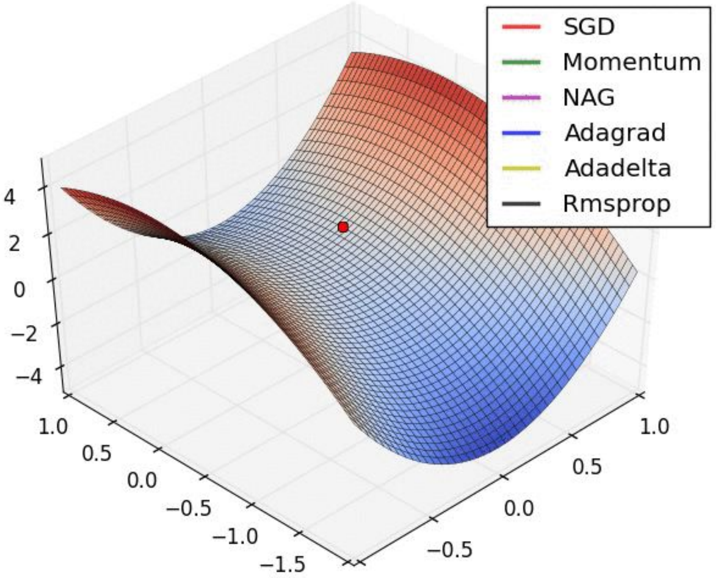


Figure borrowed from [5]

Some Tips & Notes!

- Start with a baseline (usually using Adam), if time permits explore whether improvements can be achieved
- Typical learning rates for standardized input (between 0 and 1): from 1e-6 to 1
- Searching for the optimal learning rate: start with a wide range (e.g. logarithmic scale: 0.1, 1e-2, 1e-3, …, 1e-6) and then make it fine-grained
- LR-decay is a second order parameter search (pick a good learning rate with no decay, see if you need decay)
- Smaller batch-sizes are better suited to smaller learning rates
- **Learning rate warm-up:** mitigate the effect of “early-overfitting”

References

[1] Deep Learning Book, 2016

[2] Torch.optim documentation

[3] https://www.jeremyjordan.me/nn-learning-rate/

[4] Practical recommendations for gradient-based training of deep architectures http://arxiv.org/abs/1206.5533

[5] http://cs231n.github.io/neural-networks-3/

[6] Cyclical Learning Rates for Training Neural Networks https://arxiv.org/abs/1506.01186