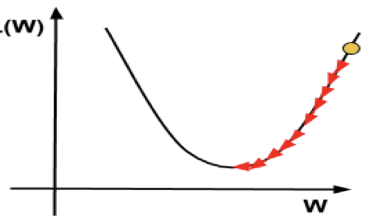
ESE 546, FALL 2020

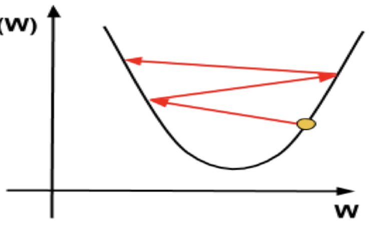
RECITATION 8

TRICKS OF TRADE IN TRAINING NEURAL NETWORKS

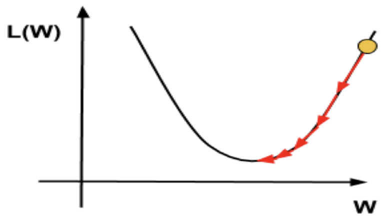
1. Learning Rate Tradeoff.



Small LR

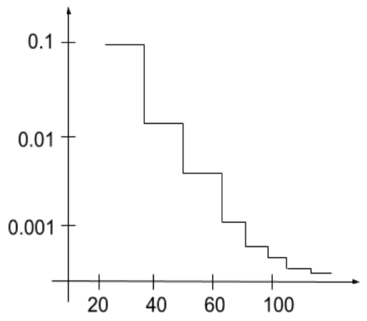


Large LR

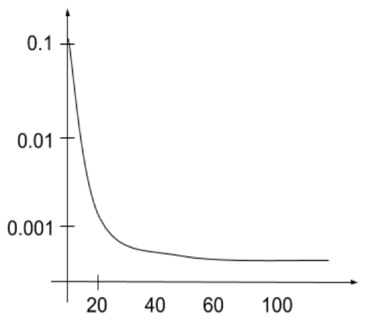


Optimal LR

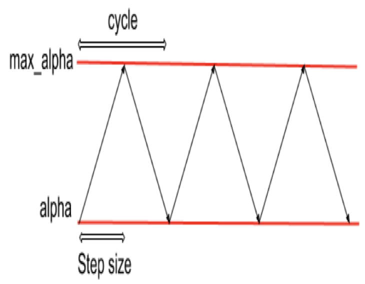
2. Learning Rate Schedulers.



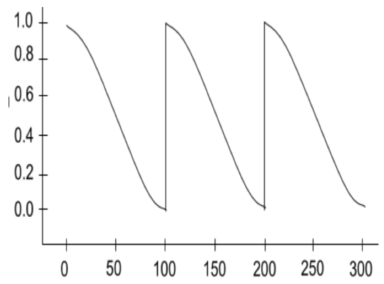
(a) Step decay
$\eta = \eta_0 \cdot \text{drop}^{\frac{\text{epoch}}{\text{interval}}}$



(b) Exponential decay
$\eta = \eta_0 \cdot e^{kt}$



(c) Cyclic decay



(d) Cosine annealing

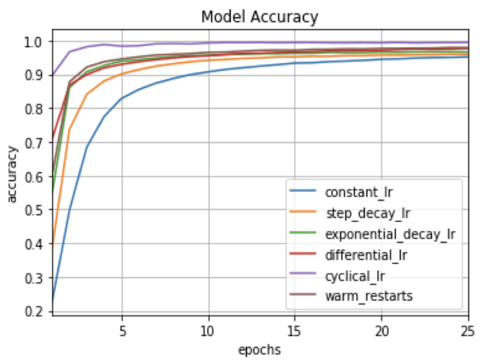
For cyclic decay, we have $\eta = \eta_{\text{min}} + (\eta_{\text{max}} - \eta_{\text{min}})(\max(0, 1-x))$, where $x = |\frac{\text{iterations}}{\text{step-size}} - 2 \cdot \text{cycle} + 1|$ and $\text{cycle} = \text{floor}(\frac{1+\text{iterations}}{2\cdot\text{step-size}})$.

For cosine annealing, we have $\eta = \eta_{\text{min}} + \frac{1}{2}(\eta_{\text{max}} - \eta_{\text{min}})(1 + \cos(\frac{T_{\text{current}}}{T}\pi))$, where T_{current} is the number of epochs since the last restart, and T is the number of epochs in an cycle.

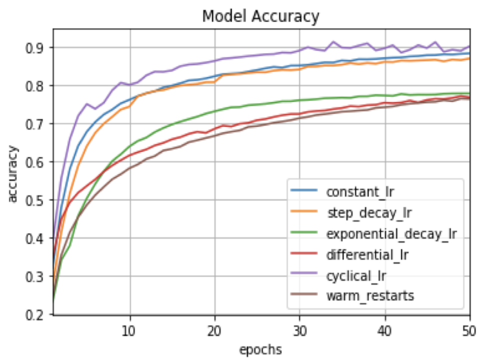
In pytorch, `torch.optim.lr_scheduler` provide `StepLR, ExponentialLR, CyclicLR, CosineAnnealingLR`, etc.

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

The comparison of different learning rate scheduling methods is as:



MNIST



CIFAR-10

3. Adaptive Learning Rate Optimizer.

Adam [3] is an adaptive learning rate method, which means, it computes individual learning rates for different parameters. It uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network.

N-th moment of a random variable is defined as the expected value of that variable to the power of n.

$$m_n = \mathbb{E}[X^n],$$

where m_n is the n-th moment of the random variable X.

Firstly, we compute the gradient of the objective function J with respect to the parameter θ,

$$g_t = \nabla_\theta J(\theta_t)$$

To obtain the unbiased estimators of the first and second moments of gradient g_t, we define m_t to be the first moment (mean) and v_t to be the second moment (variance). Then,

$$\mathbb{E}[m_t] = \mathbb{E}[g_t]$$

$$\mathbb{E}[v_t] = \mathbb{E}[g_t^2]$$

Adam keeps exponentially decaying averages of past gradients m_{t-1} and square gradients v_{t-1}, then, the moving averages of mean and variance are,

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

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

where β_1 and β_2 are hyperparameters, we usually choose $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

Because we initialize averages with zeros, the estimators are biased towards zero,

$$m_1 = \beta_1 m_0 + (1 - \beta_1) g_1 = (1 - \beta_1) g_1$$

$$m_2 = \beta_1 m_1 + (1 - \beta_1) g_2 = \beta_1 (1 - \beta_1) g_1 + (1 - \beta_1) g_2$$

$$\vdots$$

$$m_t = (1 - \beta_1) \sum_{i=0}^{t} \beta_1^{t-i} g_i$$

The expected value of m is,

$$\begin{aligned}\mathbb{E}[m_t] &= \mathbb{E}[(1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}g_i] \\ &= \mathbb{E}[g_t](1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i} + \zeta \\ &= \mathbb{E}[g_t](1-\beta_1^t) + \zeta\end{aligned}$$

Note that the estimator m_t is biased since $\mathbb{E}[m_t] \neq \mathbb{E}[g_t]$. Similarly, we have a biased estimator v_t. We need to correct the bias,

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

The only thing left to do is to use those moving averages to scale learning rate individually for each parameter,

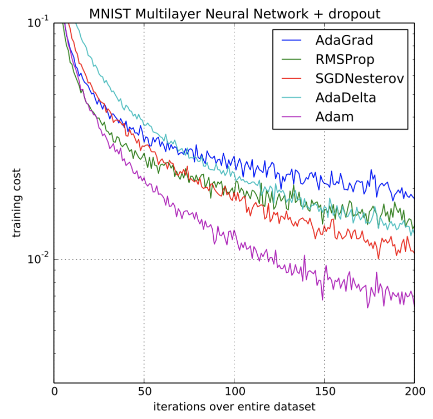
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$

where ϵ is usually $1e-8$.

An Adam implementation is as follows:

```
g = compute_gradient(x, y)
m = beta_1 * m + (1 - beta_1) * g
v = beta_2 * v + (1 - beta_2) * np.power(g, 2)
m_hat = m / (1 - np.power(beta_1, t))
v_hat = v / (1 - np.power(beta_2, t))
w = w - step_size * m_hat / (np.sqrt(v_hat) + epsilon)
```

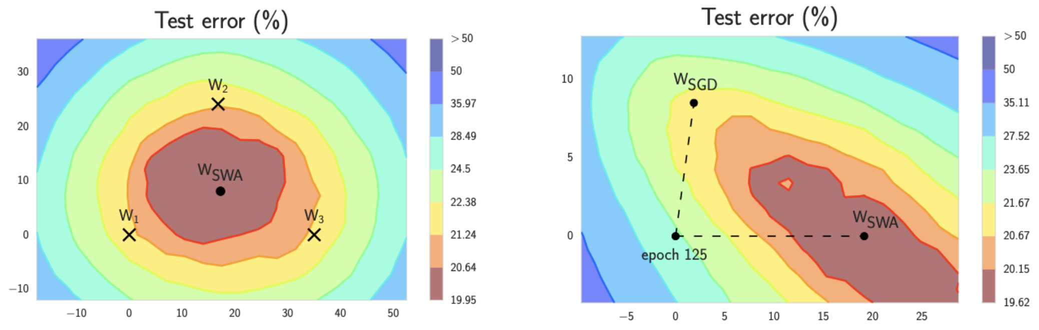
The comparison of different adaptive learning rate optimizers is as:



An interactive visualization tool: `https://bl.ocks.org/EmilienDupont/raw/aaf429be5705b219aaaf8d691e27ca87/`.

4. Stochastic Weight Averaging Optimizer.

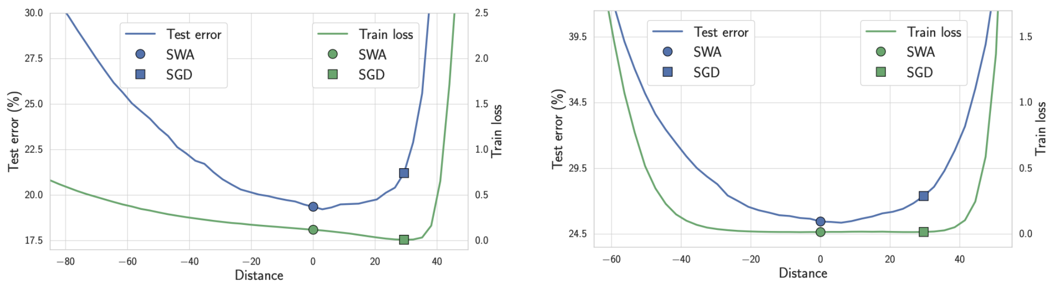
SWA [2] is based on averaging the samples proposed by SGD using a learning rate schedule that allows exploration of the region of weight space corresponding to high-performing networks.



ResNet-164 on CIFAR-100

```
Algorithm 1: Stochastic Weight Averaging
Require: Weight $\hat{w}$, learning rate $\eta$, epochs $n$
1: Initialize with $w \leftarrow \hat{w}$ and $w_{\text{SWA}} \leftarrow w$
2: for $i = 1, 2.., n$ do
3:     $w \leftarrow w - \eta \nabla \mathcal{L}_i(w)$
4:     $n_{\text{models}} \leftarrow i$
5:     $w_{\text{SWA}} \leftarrow \frac{w_{\text{SWA}} \cdot n_{\text{models}} + w}{n_{\text{models}} + 1}$
6: end for
7: Output: $w_{\text{SWA}}$
```

Simple averaging of multiple points along the trajectory of SGD leads to better generalization than conventional training. SWA procedure finds much flatter solutions than SGD.



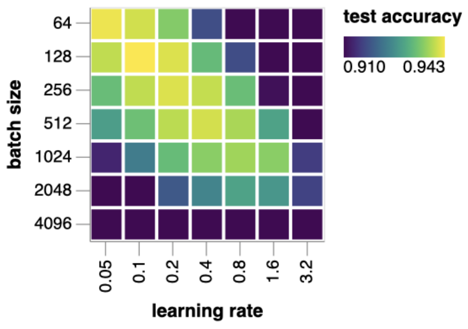
ResNet-164 and VGG-16 on CIFAR-100

SWA is implemented in `torchcontrib` of pytorch:

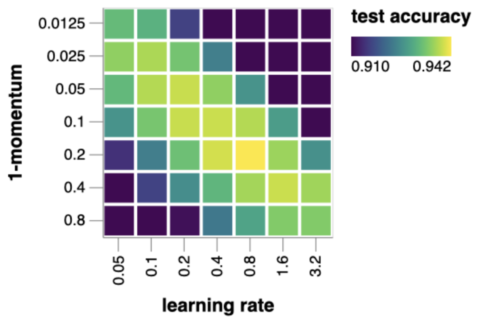
```
from torchcontrib.optim import SWA
...
base_opt = torch.optim.SGD(model.parameters(), lr=0.1)
opt = SWA(base_opt, swa_start=10, swa_freq=5, swa_lr=0.05)
for (input, target) in data_loader:
     opt.zero_grad()
     loss_fn(model(input), target).backward()
     opt.step()
opt.swap_swa_sgd()
```

5. Hyperparameter Tuning.

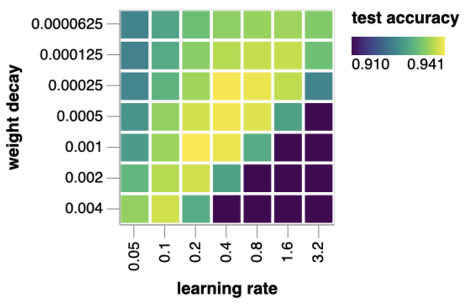
We plot CIFAR-10 test accuracy of ResNet over various two-dimensional slices in hyperparameter space. In each case, the learning rate on the x-axis refers to the maximal learning rate in the schedule. A second hyperparameter (batch size, momentum or weight decay) is varied on the y-axis of each plot. Other hyperparameters are fixed at: batch size $= 512$, momentum $= 0.9$, weight decay $= 5e-4$.



Batch size N v.s.learning rate η



Momentum ρ v.s. learning rate η



Weight decay α v.s. learning rate η

Each plot has a ridge of maximal test accuracy oriented at 45° to the axes and spanning a wide range in log-parameter space. We observe that almost-flat directions in which $\frac{\eta}{N}$, $\frac{\eta}{1-\rho}$ and $\eta\alpha$ are held constant.

Check this post for more details: `https://myrtle.ai/learn/how-to-train-your-resnet/`.

6. Mix-up Training.

Mixup [4] is an augmentation method. In mixup, each time we randomly sample two examples (x_i, y_i) and (x_j, y_j). Then, we form a new example by a weighted linear interpolation of these two examples:

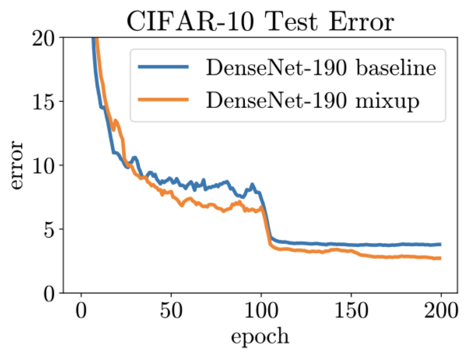
$$\hat{x} = \lambda x_i + (1 - \lambda) x_j$$
$$\hat{y} = \lambda y_i + (1 - \lambda) y_j$$

where $\lambda \in [0, 1]$ is a random number drawn from the Beta(α, α) distribution. In mixup training, we only use the new example $(\hat{x}, \hat{y})$.

An pytorch implementation is as follows:

```
for (x1, y1), (x2, y2) in zip(loader1, loader2):
    lam = numpy.random.beta(alpha, alpha)
    x = Variable(lam * x1 + (1. - lam) * x2)
    y = Variable(lam * y1 + (1. - lam) * y2)
    optimizer.zero_grad()
    loss(net(x), y).backward()
    optimizer.step()
```

The result of mixup training is as:



7. Low Precision Training.

Neural networks are commonly trained with 32-bit floating point (FP32) precision. That is, all numbers are stored in FP32 format and both inputs and outputs of arithmetic operations are FP32 numbers as well. A reduced precision has a narrower range that makes results more likely to be out-of-range and then disturb the training progress.

Run in mixed precision using `torch.cuda.amp.autocast`:

```
with autocast():
    output = model(input)
    loss = loss_fn(output, target)
```

Run in mixed precision using Nvidia `apex.amp` package:

```
model, optimizer = amp.initialize(model, optimizer, opt_level='O1')
...
with amp.scale_loss(loss, optimizer) as scaled_loss:
    scaled_loss.backward()
```

The result of low-precision [1] training is as:

Model	Efficient			Baseline		
	Time/epoch	Top-1	Top-5	Time/epoch	Top-1	Top-5
ResNet-50	**4.4 min**	**76.21**	**92.97**	13.3 min	75.87	92.70
Inception-V3	**8 min**	**77.50**	**93.60**	19.8 min	77.32	93.43
MobileNet	**3.7 min**	**71.90**	**90.47**	6.2 min	69.03	88.71

8. Multiple GPUs and Distributed Training.

We can wrap our model on multiple GPUs using `torch.nn.DataParallel`:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = Model(input_size, output_size)
model = nn.DataParallel(model).to(device)
```

`torch.nn.parallel.DistributedDataParallel` is proven to be significantly faster than `torch.nn.DataParallel` for single-node multi-GPU data parallel training.

```
import torch.distributed as dist
from torch.utils.data.distributed import DistributedSampler

dist.init_process_group(backend="nccl", init_method="env://")
rank = dist.get_rank()
world_size = dist.get_world_size()

# prepare the dataset
dataset = RandomDataset(input_size, data_size)
train_sampler = DistributedSampler(dataset)
rand_loader = DataLoader(dataset, batch_size=batch_size//world_size,
                         shuffle=(train_sampler is None),
                         sampler=train_sampler)
```

```
model = Model(input_size, output_size)

# define distributed model
device = torch.device('cuda', rank)
model = model.to(device)
model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[rank
                                        ], output_device=rank)
```

Pytorch documentation: https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html.

References

[1] He, T., Zhang, Z., Zhang, H., Zhang, Z., Xie, J., and Li, M. (2019). Bag of tricks for image classification with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 558–567.

[2] Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., and Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*.

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

[4] Zhang, H., Cisse, M., Dauphin, Y. N., and Lopez-Paz, D. (2017). mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*.