Chapter 6: Additional things

Mathematical Foundations of Deep Neural Networks

Fall 2021

Department of Mathematical Sciences

Ernest K. Ryu

Seoul National University

Weight sharing

In neural networks, *weight sharing* is a way to reduce the number of parameters by reusing the same parameter in multiple operations. Convolutional layers is the primary example.

$$A_w = \begin{bmatrix} w_1 & \cdots & w_r & 0 & \cdots & & & 0 \\ 0 & w_1 & \cdots & w_r & 0 & \cdots & & 0 \\ 0 & 0 & w_1 & \cdots & w_r & 0 & \cdots & 0 \\ \vdots & & & \ddots & & \ddots & & \vdots \\ 0 & & \cdots & 0 & w_1 & \cdots & w_r & 0 \\ 0 & & \cdots & 0 & 0 & w_1 & \cdots & w_r \end{bmatrix}$$

If we consider convolution with filter w as a linear operator, the components of w appear may times in the matrix representation. This is because the same w is reused for every patch in the convolution. Weight sharing in convolution may now seem obvious, but it was a key contribution back when the LeNet architecture was presented.

Some models (not studied in this course) use weight sharing more explicitly in other ways.

Ensemble learning

Let (X, Y) be a data-label pair. Let $m_1, \dots, m_K$ be models estimating the Y given X.

An *ensemble* is a model

$$M = \theta_1 m_1 + \cdots + \theta_K m_K$$

where $\theta_1, \dots, \theta_K \in \mathbb{R}$. Often $\theta_1 + \cdots + \theta_K = 1$ and $\theta_i \geq 0$ for $i = 1, \dots, K$. (So M is often a nonnegative weighted average $m_1, \dots, m_K$.)

If $\theta_1, \dots, \theta_K$ is chosen well, then

$$\mathbb{E}_{(X,Y)}[\|M(X) - Y\|^2] \leq \min_{i=1,\dots,K} \mathbb{E}_{(X,Y)}[\|m_i(X) - Y\|^2]$$

(The ensemble can be worse if $\theta_1, \dots, \theta_K$ is chosen poorly.)

2016 ImageNet Challenge ensemble

Trimps–Soushen* won the 2016 ImageNet Challenge with an ensemble of

- Inception-v3[1]
- Inception-v4[2]
- Inception-Resnet-v2[2]
- ResNet-200[3]
- WRN-68-3[4]

*J. Shao, X. Zhang, Z. Ding, Y. Zhao, Y. Chen, J. Zhou, W. Wang, L. Mei, and C. Hu, Trimps-Soushen, 2016.
[1]C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, Rethinking the inception architecture for computer vision, *CVPR*, 2016.
[2]C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, Inception-v4, Inception-ResNet and the impact of residual connections on learning, *AAAI*, 2017.
[3]K. He, X. Zhang, S. Ren, and J. Sun, Identity mappings in deep residual networks, *ECCV*, 2016.
[4]S. Zagoruyko and N. Komodakis, Wide residual networks, *BMVC*, 2016.

Dropout ensemble interpretation

Let m be a model with dropout applied to K neurons. The there are 2^K possible configurations, which we label $m_1, \dots, m_{2^K}$. These models share weights.

Dropout can be viewed as randomly selecting one of these models and updating it with an iteration of SGD.

Turning off dropout at test time can be interpreted and making predictions with an ensemble of these 2^K, since each neuron is scaled so that the neuron value has the same expectation as when dropout is applied.

However, this is not a very precise connection, and I am unsure as to how much to trust it.

K. Hara, D. Saitoh, and H. Shouno, Analysis of dropout learning regarded as ensemble learning, *ICANN*, 2016.

Test-time data augmentation

Test-time data augmentation is an ensemble technique to improve the prediction.

Given a single model M and input X, make predictions with

$$\frac{1}{K}\sum_{i=1}^{K} M\big(T_i(X)\big)$$

where $T_1, \ldots, T_K$ are random data augmentations.

The original AlexNet paper uses of test-time data augmentation with random crops and horizontal reflections: “At test time, the network makes a prediction by extracting five … patches … as well as their horizontal reflections …, and averaging the predictions made by the network’s softmax layer on the ten patches.”

Most ImageNet classifiers use similar tricks.

A. Krizhevsky, I. Sutskever, and G. E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, *NeurIPS*, 2012.

Learning rate scheduler

Sometimes, it is helpful to change (usually reduce) the learning rate as the training progresses. PyTorch provides *learning rate* schedulers to do this.

```
optimizer = SGD(model.parameters(), lr=0.1)
scheduler = ExponentialLR(optimizer, gamma=0.9) # lr = 0.9*lr
for _ in range(...):
  for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
  scheduler.step()  # .step() call updates (changes) the learning rate
```

Diminishing learning rate

One common choice is to specify a diminishing learning rate via a function (a lambda expression.) Choices like `C/epoch` or `C/sqrt(iteration)`, where `C` is an appropriately chosen constant, are common.

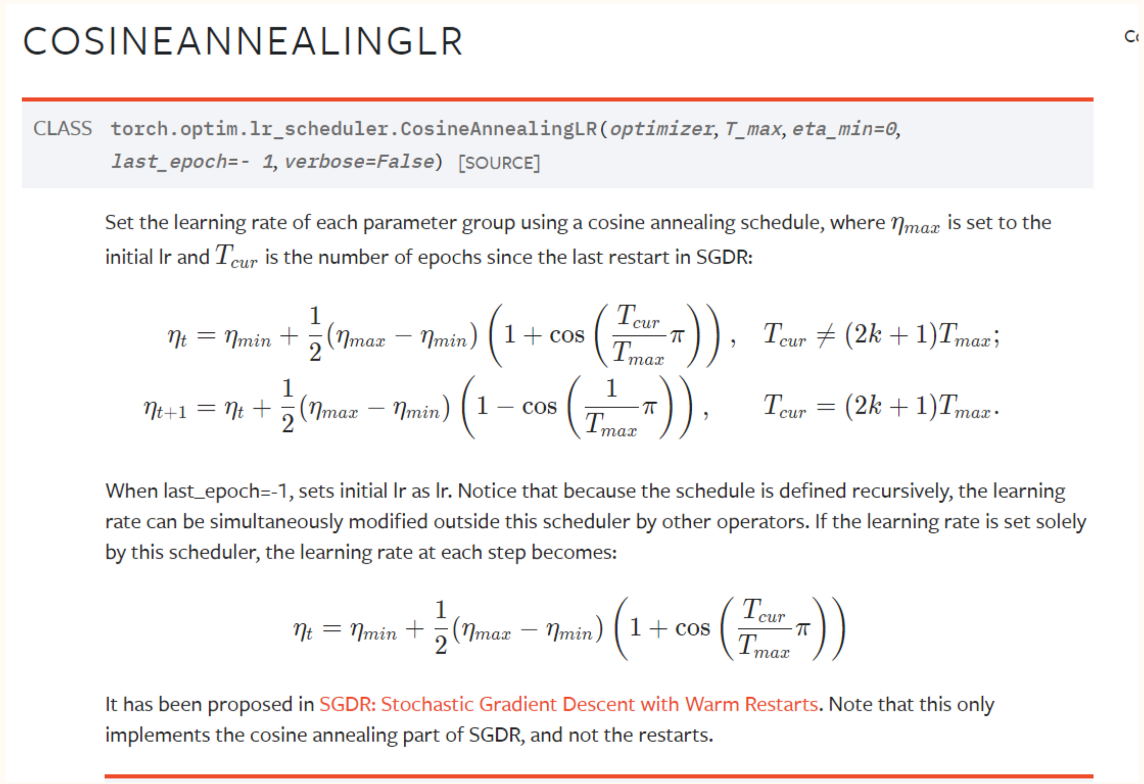
```
# lr_lambda allows us to set lr with a function
scheduler = LambdaLR(optimizer, lr_lambda= lambda ep: 1e-2/ep )
for epoch in range(...):
  for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
  scheduler.step()  # lr=0.01/epoch
```

Cosine learning rate

The cosine learning rate scheduler, which sets the learning rate with the cosine function, is also commonly used.

It is also common to use only a half-period of the cosine rather than having the learning rate oscillate.

The 2nd case in the specification means k and its purpose is to prevent the learning rate from becoming 0.



COSINEANNEALINGLR

CLASS `torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max, eta_min=0, last_epoch=- 1, verbose=False)` [SOURCE]

Set the learning rate of each parameter group using a cosine annealing schedule, where η_{max} is set to the initial lr and T_{cur} is the number of epochs since the last restart in SGDR:

$$\begin{aligned} \eta_t &= \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right), & T_{cur} \neq (2k+1)T_{max}; \\ \eta_{t+1} &= \eta_t + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\frac{1}{T_{max}}\pi\right)\right), & T_{cur} = (2k+1)T_{max}. \end{aligned}$$

When last_epoch=-1, sets initial lr as lr. Notice that because the schedule is defined recursively, the learning rate can be simultaneously modified outside this scheduler by other operators. If the learning rate is set solely by this scheduler, the learning rate at each step becomes:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right)$$

It has been proposed in SGDR: Stochastic Gradient Descent with Warm Restarts. Note that this only implements the cosine annealing part of SGDR, and not the restarts.

Wide vs. sharp minima

As alluded to in hw1:

- Large step makes large and rough progress towards regions with small loss.
- Small steps refines the model by finding sharper minima.

Also small steps better suppress the effect of noise. Mathematically, one can show that SGD with small steps becomes very similar to GD with small steps.[#]

However, using small steps to converge to sharp minima may not always be optimal. There is some empirical evidence that wide minima have better test error than sharp minima.[*]

[#]D. Davis, D. Drusvyatskiy, S. Kakade and J. D. Lee, Stochastic subgradient method converges on tame functions, *Found. Comput. Math.*, 2020.
[*]Y. Jiang, B. Neyshabur, H. Mobahi, D. Krishnan, and S. Bengio, Fantastic generalization measures and where to find them, *ICLR*, 2020.