Spring 2024

CMPE 258-01 Deep Learning

Dr. Kaikai Liu, Ph.D. Associate Professor

Department of Computer Engineering

San Jose State University

Email: kaikai.liu@sjsu.edu

Website: https://www.sjsu.edu/cmpe/faculty/tenure-line/kaikai-liu.php

Homework1

- Design your own neural network and testing various training options for the provided sample code
  - https://github.com/lkk688/DeepDataMiningLearning/blob/main/DeepDataMiningLearning/symboldetection.ipynb
  - Goal: achieve better results (i.e., low BER)
  - Individual work
  - Due: 2/25 11:59PM

Adaptive Learning Rate

- The other big idea of optimization algorithms is adaptive learning rate. The intuition is that we'd like to perform smaller updates for frequent features and bigger ones for infrequent ones.
- Adagrad
  - Uses different learning rates for each iteration
  - It performs smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequently occurring features.
  - When the gradient is changing very fast, the learning rate will be smaller. When the gradient is changing slowly, the learning rate will be bigger.
  - Achieve a different learning rate for each parameter (or an adaptive learning rate).

Other Optimizers

- Adagrad (Adaptive Gradient Descent) Deep Learning Optimizer
  - Adagrad keeps a running sum of the squares of the gradients in each dimension, and in each update, scale the learning rate based on the sum.
  - Using the root of the squared gradients: take into account the magnitude of the gradients and not the sign.
  - Paper: https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf
  - We speed up the updating process along the axis with weak gradients by increasing these gradients along this axis. On the other hand, we slow down the updates of the weights along the axis with large gradients.

$$g_0 = 0$$

$$g_{t+1} \leftarrow g_t + \nabla_\theta \mathcal{L}(\theta)^2$$

$$\theta_j \leftarrow \theta_j - \epsilon \frac{\nabla_\theta \mathcal{L}}{\sqrt{g_{t+1}} + 1e^{-5}}$$

the sum of squared gradients up to the time

divide the current gradient by the root of the term

Other Optimizers

- Adagrad (Adaptive Gradient Descent) Deep Learning Optimizer
  - A big drawback of Adagrad is that as time goes by, the learning rate becomes smaller and smaller due to the monotonic increment of the running squared sum.
  - The sum of squared gradients become big if the training takes too long. When the current gradient is divided by this large number, the update step for the weights becomes very small. It is as if we were using a very low learning rate, which becomes even lower the longer the training takes. In the worst case, we would get stuck at AdaGrad and the training would go on forever.

Other Optimizers

- RMSProp: slight modification of AdaGrad
  - Root Mean Square Propagation (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight
  - This modification is intended to solve the previously described problem that can occur with AdaGrad. In RMSProp, the running sum of squared gradients g_{t+1} is maintained. However, instead of allowing this sum to increase continuously over the training period, we allow the sum to decrease.
  - RMSProp keep moving average of the squared gradients for each weight. And then we divide the gradient by square root the mean square.

AdaGrad	RMS Prop
$g_0 = 0$	$g_0 = 0, \alpha \simeq 0.9$
$g_{t+1} \leftarrow g_t + \nabla_\theta \mathcal{L}(\theta)^2$	$g_{t+1} \leftarrow \alpha \cdot g_t + (1-\alpha)\nabla_\theta \mathcal{L}(\theta)^2$
$\theta_j \leftarrow \theta_j - \epsilon \frac{\nabla_\theta \mathcal{L}}{\sqrt{g_{t+1}} + 1e^{-5}}$	$\theta_j \leftarrow \theta_j - \epsilon \frac{\nabla_\theta \mathcal{L}}{\sqrt{g_{t+1}} + 1e^{-5}}$

Other Optimizers

- Adam is an adaptive learning rate optimization algorithm that utilises both momentum and scaling, combining the benefits of RMSProp and SGD w/th Momentum.
  - Adam paper: https://arxiv.org/pdf/1412.6980.pdf
  - The algorithm updates exponential moving averages of the gradient (m_t) and the squared gradient (v_t) where the hyper-parameters β1 , β2 ∈ [0, 1) control the exponential decay rates of these moving averages.
  - Problem: At the very first time step, m_t v_t are close to 0, This leads to a very large first update step. Solution: ADAM includes a correction clause.

$$m_0 = 0, v_0 = 0$$

$$m_{t+1} \leftarrow \beta_1 m_t + (1 - \beta_1)\nabla_\theta \mathcal{L}(\theta)$$

$$v_{t+1} \leftarrow \beta_2 v_t + (1 - \beta_2)\nabla_\theta \mathcal{L}(\theta)^2$$

$$\theta_j \leftarrow \theta_j - \frac{\epsilon}{\sqrt{v_{t+1}} + 1e^{-5}} m_{t+1}$$

m_t: first momentum

v_t: second momentum

RMSProp + momentum

$$m_{t+1} \leftarrow \beta_1 m_t + (1 - \beta_1)\nabla_\theta \mathcal{L}(\theta)$$

$$v_{t+1} \leftarrow \beta_2 v_t + (1 - \beta_2)\nabla_\theta \mathcal{L}(\theta)^2$$

$$\hat{m}_{t+1} \leftarrow \frac{m_{t+1}}{1 - \beta_1^t}$$

$$\hat{v}_{t+1} \leftarrow \frac{v_{t+1}}{1 - \beta_2^t}$$

$$\theta_j \leftarrow \theta_j - \frac{\epsilon}{\sqrt{v_{t+1}} + 1e^{-5}} m_{t+1}$$

add biases in our moments in order to force our algorithm to take smaller steps in the beginning

Other Optimizers

- AdaMax
  - Adam scales the second moment according to the L2 norm values of the gradient. However, we can extend this principle to use the infinity norm L_∞

The infinity norm L_∞ for a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is defined as $|\mathbf{x}|_\infty := \max(|x_1|, \ldots, |x_n|)$.

  - It has been shown that L∞ also provides stable behavior and AdaMax can sometimes have better performance than Adam (especially in models with embeddings).
  - AdaMax calculates the velocity moment as:

$$v_t = \max(\delta_2 v_{t-1}, |\nabla_w L(x, y, w_t)|)$$

Other Optimizers

- Nadam
  - The Nadam (Nesterov-accelerated Adaptive Moment Estimation) algorithm is a slight modification of Adam where vanilla momentum is replaced by Nesterov Momentum.
  - Nadam generally performs well on problems with very noisy gradients or for gradients with high curvature. It usually provides a little faster training time as well.
  - the velocity vector and the update rule remain intact. The new momentum (after adding the bias) is then shaped as:

$$m_t = \frac{\delta_{t+1} m_t}{1 - \delta_1^{t+1}} + \frac{(1 - \delta_t)(\nabla_w L(x, y, w_t))}{1 - \delta_1^t}$$

Other Optimizers

- AdaBelief
  - Adabelief is a new Optimization algorithm proposed in 2020 that promises :
    - Faster training convergence
    - Greater training stability
    - Better model generalization
  - The key idea is to change the step size according to the “belief” in the current gradient direction.
  - We enhance Adam by **computing the variance of the gradient over time instead of the momentum squared**.
  - The variance of the gradient is the distance from the expected (believed) gradient.
  - That way the optimizer now considers the curvature of the loss function. If the observed gradient greatly deviates from the belief, we distrust the current observation and take a small step.

$$v_t = \delta_2 v_{t-1} + (1-\delta_2)(\nabla_w L(x, y, w_t)^2) \text{ becomes } s_t = \delta_2 s_{t-1} + (1-\delta_2)(\nabla_w L(x, y, w_t) - m_t)^2 + \epsilon.$$

Other Optimizers

- Weight Decay
  - Weight Decay, or L2 Regularization, is a regularization technique applied to the weights of a neural network. We minimize a loss function compromising both the primary loss function and a penalty on the L2 Norm of the weights:

$$L_{new}(w) = L_{original}(w) + \lambda w^T w$$

https://paperswithcode.com/method/weight-decay

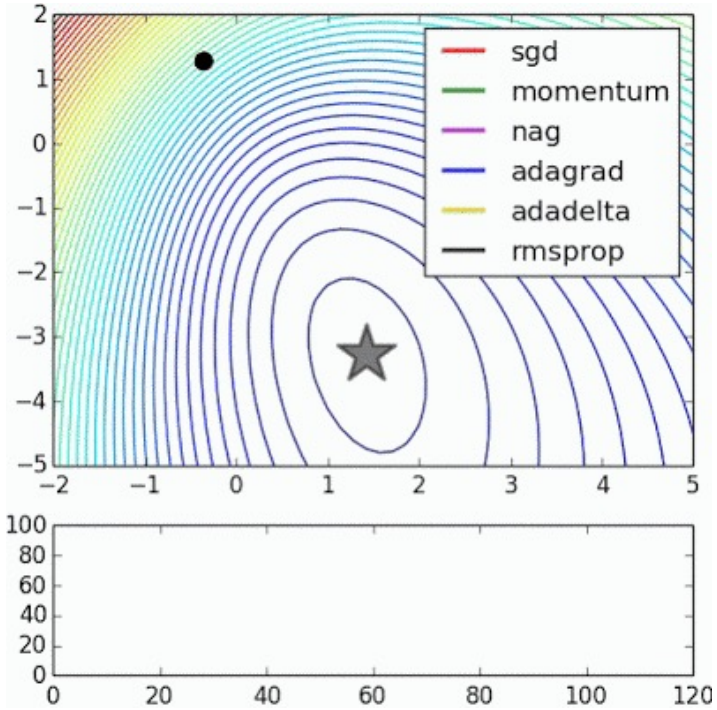
- AdamW:
  - Paper: Decoupled Weight Decay Regularization
    - http://arxiv.org/abs/1711.05101v3
  - AdamW is a stochastic optimization method that modifies the typical implementation of weight decay in Adam, by decoupling weight decay from the gradient update.

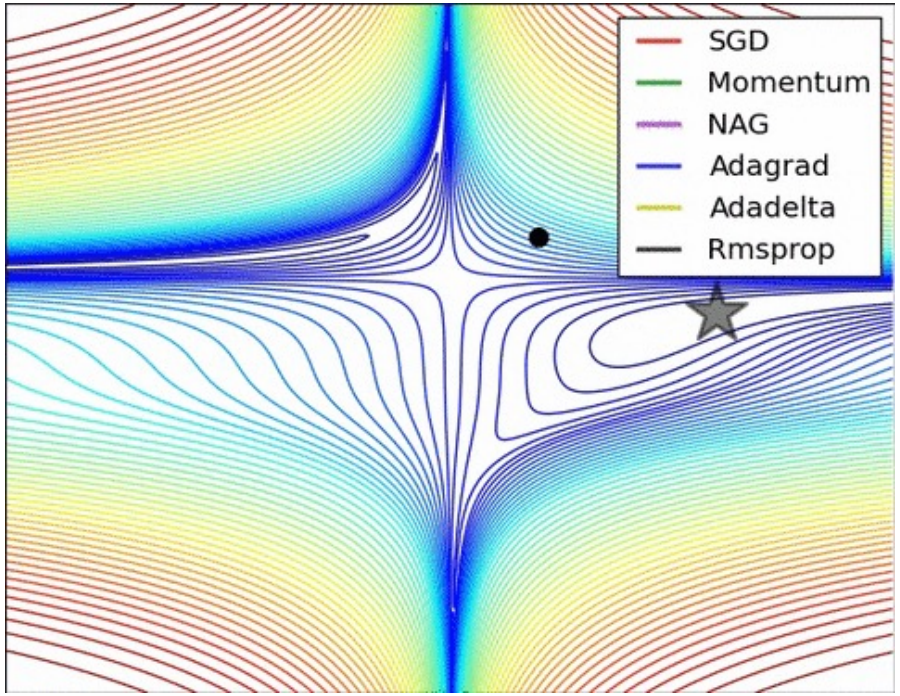
$$\theta_{t+1,i} = \theta_{t,i} - \eta\left(\frac{1}{\sqrt{\hat{v}_t + \epsilon}} \cdot \hat{m}_t + w_{t,i}\theta_{t,i}\right), \forall t$$

w is the rate of weight decay

Other Optimizers

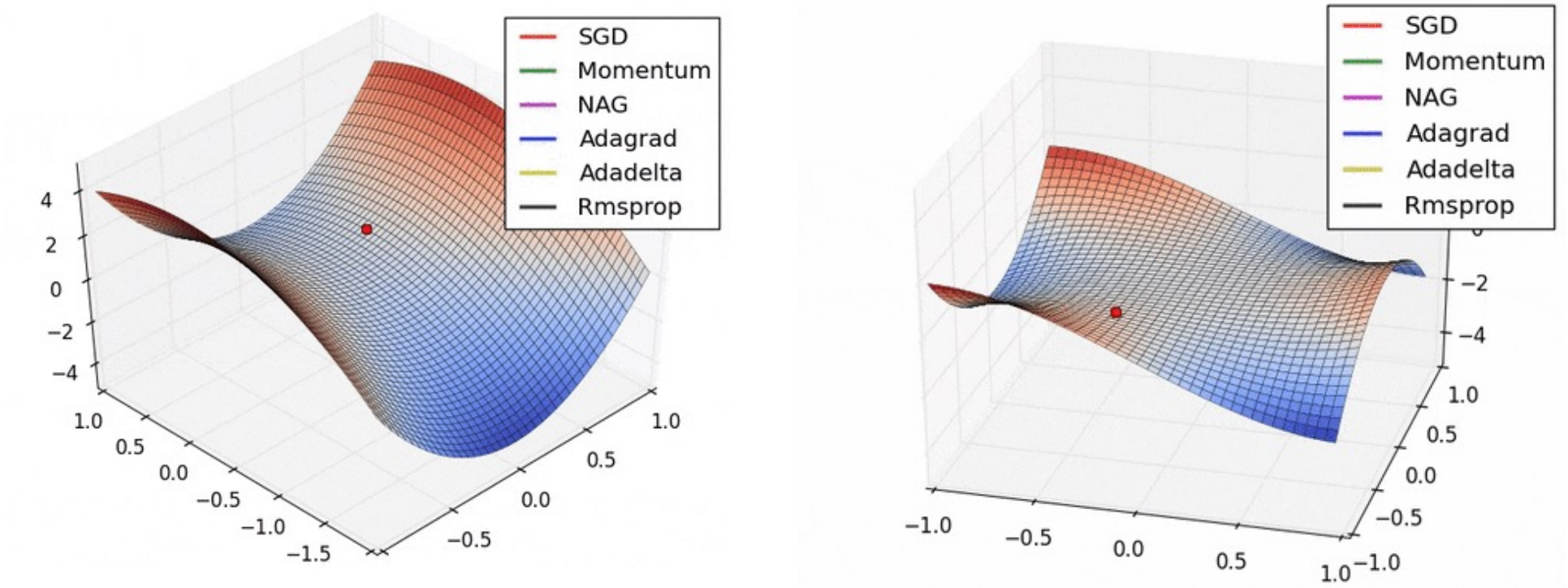
- Visualizing optimizers and observations
  - Algorithms with momentum have a smoother trajectory than non-momentum based but this may result in overshooting.
  - Methods with an adaptive learning rate have a faster convergence rate, better stability, and less jittering.





Other Optimizers

- Visualizing optimizers and observations
  - Algorithms that do not scale the step size (adaptive learning rate) have a harder time to escape local minimums and break the symmetry of the loss function
  - Saddle points cause momentum-based methods to oscillate before finding the correct downhill path



Torch Optim

- torch.optim is a package implementing various optimization algorithms.

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam(model.parameters(), lr=0.05)
```

- Optimizer s also support specifying per-parameter options.
  - contain a params key, containing a list of parameters belonging to it.

```
optim.SGD([
        {'params': model.base.parameters()},
        {'params': model.classifier.parameters(), 'lr': 1e-3}
    ], lr=1e-2, momentum=0.9)
```

This means that model.base's parameters will use the default learning rate of 1e-2, model.classifier's parameters will use a learning rate of 1e-3, and a momentum of 0.9 will be used for all parameters.

- Taking an optimization step

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

Torch Optim

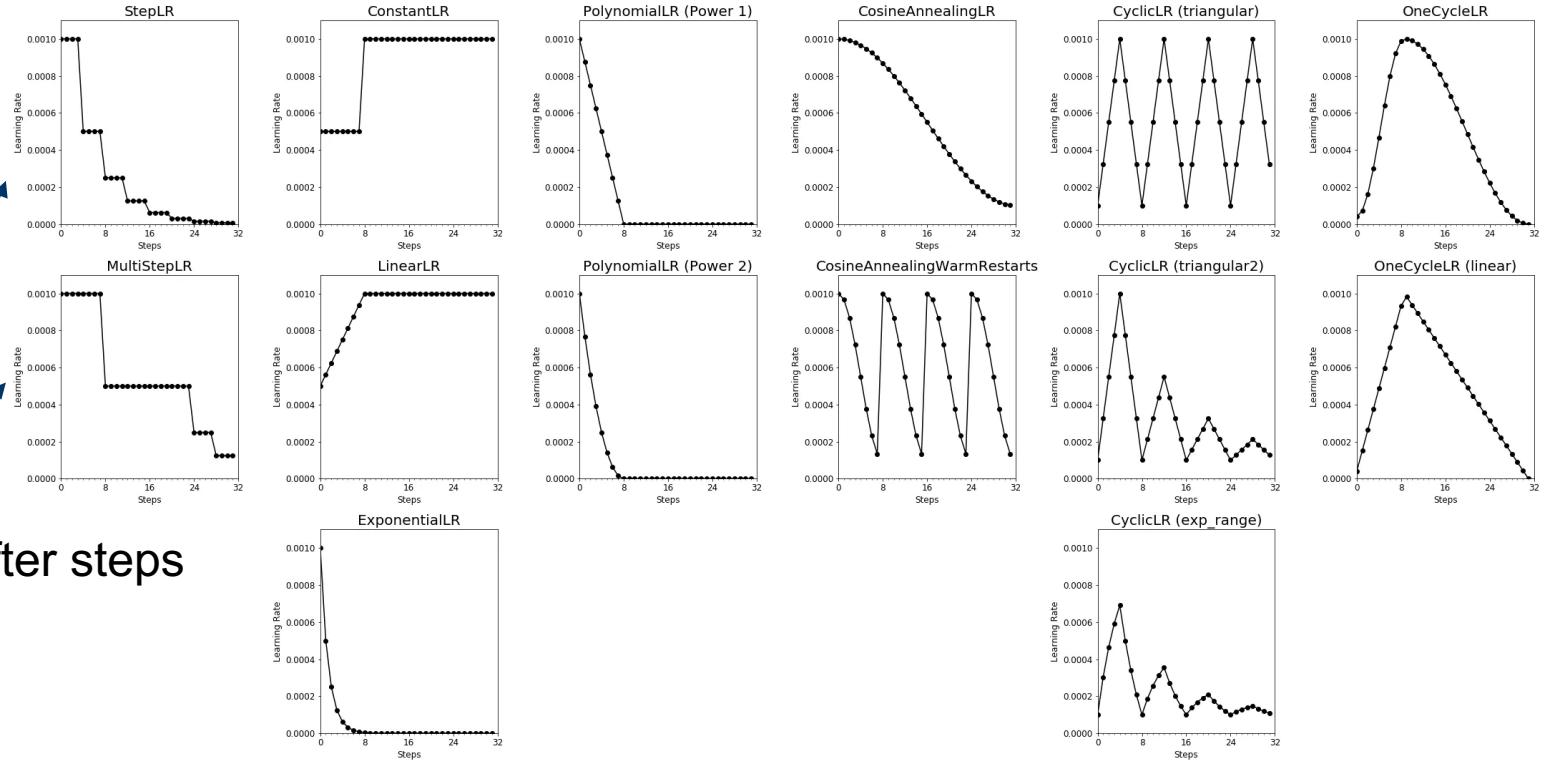
- torch.optim.lr_scheduler provides several methods to adjust the learning rate based on the number of epochs.
  - https://pytorch.org/docs/stable/optim.html#module-torch.optim.lr_scheduler
  - Learning rate scheduling should be applied after optimizer's update

```
optimizer = optim.SGD(model.parameters(), lr=0.01,
momentum=0.9)
scheduler = ExponentialLR(optimizer, gamma=0.9)

for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler.step()
```

Torch Optim

- torch.optim.lr_scheduler provides several methods to adjust the learning rate based on the number of epochs.
  - https://pytorch.org/docs/stable/optim.html#module-torch.optim.lr_scheduler
  - Learning rate scheduling should be applied after optimizer's update



Decay gamma after steps

Thank You

Address:
ENG257, SJSU

Email Address:
Kaikai.liu@sjsu.edu