Advanced Optimizers and Learning Rate Scheduling

Lecture Overview

This lecture covers advanced optimization algorithms and learning rate scheduling strategies for training deep neural networks effectively.

Topics Covered:

- SGD with momentum
- Adaptive optimizers: RMSprop, Adam, AdamW
- Learning rate scheduling strategies
- Warmup techniques
- Choosing the right optimizer

1. Momentum and SGD Variants

1.1 SGD with Momentum

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

print("SGD with Momentum")
print("="*70)

print("Basic SGD Update:")
print("  w = w - lr * gradient")

print("\nProblems with basic SGD:")
print("  - Slow convergence in ravines")
print("  - Oscillates perpendicular to optimal direction")

print("\nMomentum Update:")
print("  v = momentum * v - lr * gradient")
print("  w = w + v")

print("\nBenefits:")
print("  - Accelerates in consistent gradient directions")
print("  - Dampens oscillations")
print("  - Typical momentum: 0.9")

def sgd_step(w, grad, lr):
    return w - lr * grad

def momentum_step(w, v, grad, lr, momentum=0.9):
    v_new = momentum * v - lr * grad
    w_new = w + v_new
    return w_new, v_new

# Example: f(x,y) = x^2 + 10*y^2
print("\nExample: f(x,y) = x^2 + 10*y^2, starting at (10, 1):")

w_sgd = np.array([10.0, 1.0])
lr = 0.1
print(f"\nSGD (lr={lr}):")
for i in range(5):
    grad = np.array([2*w_sgd[0], 20*w_sgd[1]])
    w_sgd = sgd_step(w_sgd, grad, lr)
    print(f"  Step {i+1}: ({w_sgd[0]:.4f}, {w_sgd[1]:.4f})")

w_mom = np.array([10.0, 1.0])
v = np.array([0.0, 0.0])
print(f"\nSGD with Momentum (momentum=0.9):")
for i in range(5):
    grad = np.array([2*w_mom[0], 20*w_mom[1]])
    w_mom, v = momentum_step(w_mom, v, grad, lr, 0.9)
    print(f"  Step {i+1}: ({w_mom[0]:.4f}, {w_mom[1]:.4f})")
```

```
SGD with Momentum
======================================================================
Basic SGD Update:
  w = w - lr * gradient

Problems with basic SGD:
  - Slow convergence in ravines
  - Oscillates perpendicular to optimal direction

Momentum Update:
  v = momentum * v - lr * gradient
  w = w + v

Benefits:
  - Accelerates in consistent gradient directions
  - Dampens oscillations
  - Typical momentum: 0.9

Example: f(x,y) = x^2 + 10*y^2, starting at (10, 1):
```

```
SGD (lr=0.1):
  Step 1: (8.0000, -1.0000)
  Step 2: (6.4000, 1.0000)
  Step 3: (5.1200, -1.0000)
  Step 4: (4.0960, 1.0000)
  Step 5: (3.2768, -1.0000)

SGD with Momentum (momentum=0.9):
  Step 1: (8.0000, -1.0000)
  Step 2: (4.6000, 0.1000)
  Step 3: (2.0600, 0.7100)
  Step 4: (0.5340, -0.0810)
  Step 5: (-0.2794, -0.4771)
```

2. Adaptive Learning Rate Optimizers

2.1 RMSprop

```
print("Adaptive Learning Rate Optimizers")
print("="*70)

print("AdaGrad:")
print("  cache = cache + gradient^2")
print("  w = w - lr * gradient / (sqrt(cache) + eps)")
print("  Problem: LR monotonically decreases")

print("\nRMSprop:")
print("  cache = decay * cache + (1-decay) * gradient^2")
print("  w = w - lr * gradient / (sqrt(cache) + eps)")
print("  Typical decay: 0.9 or 0.99")

model = nn.Linear(10, 5)
optimizer_rmsprop = optim.RMSprop(model.parameters(), lr=0.01, alpha=0.99)

print("\nPyTorch RMSprop:")
print("  optimizer = RMSprop(params, lr=0.01, alpha=0.99)")
```

```
Adaptive Learning Rate Optimizers
======================================================================
AdaGrad:
  cache = cache + gradient^2
  w = w - lr * gradient / (sqrt(cache) + eps)
  Problem: LR monotonically decreases

RMSprop:
  cache = decay * cache + (1-decay) * gradient^2
  w = w - lr * gradient / (sqrt(cache) + eps)
  Typical decay: 0.9 or 0.99

PyTorch RMSprop:
  optimizer = RMSprop(params, lr=0.01, alpha=0.99)
```

2.2 Adam Optimizer

```
print("Adam Optimizer")
print("="*70)

print("Adam = Adaptive Moment Estimation")
print("  Combines momentum + RMSprop")

print("\nUpdate rules:")
print("  m = beta1 * m + (1 - beta1) * gradient       # 1st moment")
print("  v = beta2 * v + (1 - beta2) * gradient^2    # 2nd moment")
print("  m_hat = m / (1 - beta1^t)                   # Bias correction")
print("  v_hat = v / (1 - beta2^t)                   # Bias correction")
print("  w = w - lr * m_hat / (sqrt(v_hat) + eps)")

print("\nDefault hyperparameters:")
print("  lr = 0.001")
print("  beta1 = 0.9")
print("  beta2 = 0.999")
print("  eps = 1e-8")

model = nn.Linear(10, 5)
optimizer_adam = optim.Adam(model.parameters(), lr=0.001, betas=(0.9, 0.999))

print("\nPyTorch Adam:")
print("  optimizer = Adam(params, lr=0.001, betas=(0.9, 0.999))")

print("\nAdamW (Decoupled Weight Decay):")
print("  Better for transformers")
print("  optimizer = AdamW(params, lr=0.001, weight_decay=0.01)")

print("\nOptimizer Recommendations:")
print("  CNN: SGD(lr=0.1, momentum=0.9, weight_decay=1e-4)")
print("  Transformer: AdamW(lr=1e-4, weight_decay=0.01)")
print("  General: Adam(lr=1e-3)")
```

```
Adam Optimizer
======================================================================
Adam = Adaptive Moment Estimation
  Combines momentum + RMSprop

Update rules:
  m = beta1 * m + (1 - beta1) * gradient       # 1st moment
  v = beta2 * v + (1 - beta2) * gradient^2    # 2nd moment
  m_hat = m / (1 - beta1^t)                   # Bias correction
  v_hat = v / (1 - beta2^t)                   # Bias correction
  w = w - lr * m_hat / (sqrt(v_hat) + eps)

Default hyperparameters:
  lr = 0.001
  beta1 = 0.9
  beta2 = 0.999
  eps = 1e-8

PyTorch Adam:
  optimizer = Adam(params, lr=0.001, betas=(0.9, 0.999))

AdamW (Decoupled Weight Decay):
  Better for transformers
  optimizer = AdamW(params, lr=0.001, weight_decay=0.01)

Optimizer Recommendations:
  CNN: SGD(lr=0.1, momentum=0.9, weight_decay=1e-4)
  Transformer: AdamW(lr=1e-4, weight_decay=0.01)
  General: Adam(lr=1e-3)
```

3. Learning Rate Scheduling

3.1 Step and Exponential Decay

```
print("Learning Rate Scheduling")
print("="*70)

print("Why schedule learning rate?")
print("  - Start high for fast initial progress")
print("  - Reduce for fine-tuning and convergence")

model = nn.Linear(10, 5)
optimizer = optim.SGD(model.parameters(), lr=0.1)

# Step LR
step_scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

print("\nStepLR: Reduce by gamma every step_size epochs")
print("  Epoch | LR")
for epoch in range(35):
    if epoch % 10 == 0:
        print(f"  {epoch:&gt;5} | {optimizer.param_groups[0]['lr']:.6f}")
    step_scheduler.step()

# Exponential
optimizer2 = optim.SGD(model.parameters(), lr=0.1)
exp_scheduler = optim.lr_scheduler.ExponentialLR(optimizer2, gamma=0.95)

print("\nExponentialLR: LR = initial_lr * gamma^epoch")
print("  Epoch | LR")
for epoch in range(20):
    if epoch % 5 == 0:
        print(f"  {epoch:&gt;5} | {optimizer2.param_groups[0]['lr']:.6f}")
    exp_scheduler.step()
```

```
Learning Rate Scheduling
======================================================================
Why schedule learning rate?
  - Start high for fast initial progress
  - Reduce for fine-tuning and convergence

StepLR: Reduce by gamma every step_size epochs
  Epoch | LR
      0 | 0.100000
     10 | 0.010000
     20 | 0.001000
     30 | 0.000100

ExponentialLR: LR = initial_lr * gamma^epoch
  Epoch | LR
      0 | 0.100000
      5 | 0.077378
     10 | 0.059874
     15 | 0.046329
```

3.2 Cosine Annealing

```
print("Cosine Annealing Scheduler")
print("="*70)

model = nn.Linear(10, 5)
optimizer = optim.SGD(model.parameters(), lr=0.1)
cosine_scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=50, eta_min=0.001)

print("CosineAnnealingLR: Smooth cosine decay")
print(f"  T_max={cosine_scheduler.T_max}, eta_min={cosine_scheduler.eta_min}")

print("\n  Epoch | LR")
for epoch in range(55):
    if epoch % 10 == 0:
        print(f"  {epoch:&gt;5} | {optimizer.param_groups[0]['lr']:.6f}")
    cosine_scheduler.step()
```

```
print("\nCosineAnnealingWarmRestarts:")
print("  Restarts cosine schedule periodically")
print("  Good for escaping local minima")
```

```
Cosine Annealing Scheduler
======================================================================
CosineAnnealingLR: Smooth cosine decay
  T_max=50, eta_min=0.001

  Epoch | LR
      0 | 0.100000
     10 | 0.090211
     20 | 0.062426
     30 | 0.023675
     40 | 0.002885
     50 | 0.001000

CosineAnnealingWarmRestarts:
  Restarts cosine schedule periodically
  Good for escaping local minima
```

3.3 ReduceLROnPlateau and Warmup

```
print("ReduceLROnPlateau and Warmup")
print("="*70)

model = nn.Linear(10, 5)
optimizer = optim.SGD(model.parameters(), lr=0.1)

plateau_scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=3
)

print("ReduceLROnPlateau:")
print("  Reduce LR when metric stops improving")
print("  mode='min' for loss, 'max' for accuracy")
print("  factor=0.5, patience=3")

val_losses = [2.0, 1.5, 1.2, 1.0, 0.9, 0.88, 0.87, 0.87, 0.5, 0.4]
print("\n  Epoch | Val Loss | LR")
for epoch, val_loss in enumerate(val_losses):
    print(f"  {epoch:&gt;5} | {val_loss:.4f}   | {optimizer.param_groups[0]['lr']:.6f}")
    plateau_scheduler.step(val_loss)

print("\nLearning Rate Warmup:")
print("  Start with very small LR, gradually increase")
print("  Prevents early training instability")

print("\nLinear warmup + cosine decay (common for transformers):")
print("  warmup_epochs = 5")
print("  for epoch in range(warmup_epochs):")
print("      lr = base_lr * (epoch + 1) / warmup_epochs")
print("  # then cosine decay")
```

```
ReduceLROnPlateau and Warmup
======================================================================
ReduceLROnPlateau:
  Reduce LR when metric stops improving
  mode='min' for loss, 'max' for accuracy
  factor=0.5, patience=3

  Epoch | Val Loss | LR
      0 | 2.0000   | 0.100000
      1 | 1.5000   | 0.100000
      2 | 1.2000   | 0.100000
      3 | 1.0000   | 0.100000
      4 | 0.9000   | 0.100000
      5 | 0.8800   | 0.100000
      6 | 0.8700   | 0.100000
      7 | 0.8700   | 0.100000
      8 | 0.5000   | 0.050000
      9 | 0.4000   | 0.050000

Learning Rate Warmup:
  Start with very small LR, gradually increase
  Prevents early training instability
```

```
Linear warmup + cosine decay (common for transformers):
  warmup_epochs = 5
  for epoch in range(warmup_epochs):
      lr = base_lr * (epoch + 1) / warmup_epochs
  # then cosine decay
```

Summary

Key Takeaways:

- Momentum accelerates convergence and reduces oscillation
- Adam combines momentum and adaptive learning rates
- AdamW is preferred for transformers
- Learning rate scheduling improves final model quality
- Cosine annealing provides smooth decay
- ReduceLROnPlateau adapts based on validation performance
- Warmup prevents early training instability

Practice Exercises:

1. Implement momentum SGD from scratch
2. Compare Adam vs SGD on a classification task
3. Experiment with different learning rate schedules
4. Implement linear warmup with cosine decay
5. Visualize optimizer trajectories on a 2D loss surface