CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 9, Lecture 17

Batch Normalization, Dropout, and Regularization

Lecture Overview

This lecture covers essential regularization techniques that prevent overfitting and improve generalization in neural networks.

Topics Covered:

- Batch normalization theory and implementation
- Dropout mechanism and variants
- L1 and L2 regularization (weight decay)
- Early stopping
- Data augmentation techniques

1. Batch Normalization

1.1 Why Normalize?

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

print("Batch Normalization")
print("="*70)

print("Problem: Internal Covariate Shift")
print("  - As training progresses, layer input distributions change")
print("  - Later layers must continuously adapt to new distributions")
print("  - Slows down training and requires careful initialization")

print("\nSolution: Normalize activations within each mini-batch")
print("  1. Compute batch mean: mu = (1/m) * sum(x)")
print("  2. Compute batch variance: var = (1/m) * sum((x - mu)^2)")
print("  3. Normalize: x_hat = (x - mu) / sqrt(var + epsilon)")
print("  4. Scale and shift: y = gamma * x_hat + beta")

# Manual batch normalization
def batch_norm_manual(x, gamma, beta, eps=1e-5):
    mu = x.mean(dim=0)
    var = x.var(dim=0, unbiased=False)
    x_hat = (x - mu) / torch.sqrt(var + eps)
    return gamma * x_hat + beta

torch.manual_seed(42)
x = torch.randn(32, 64)
gamma = torch.ones(64)
beta = torch.zeros(64)
y = batch_norm_manual(x, gamma, beta)

print(f"\nInput statistics:")
print(f"  Mean: {x.mean():.4f}, Std: {x.std():.4f}")
print(f"\nNormalized statistics:")
print(f"  Mean: {y.mean():.4f}, Std: {y.std():.4f}")
```

```
Batch Normalization
======================================================================
Problem: Internal Covariate Shift
  - As training progresses, layer input distributions change
  - Later layers must continuously adapt to new distributions
  - Slows down training and requires careful initialization

Solution: Normalize activations within each mini-batch
  1. Compute batch mean: mu = (1/m) * sum(x)
  2. Compute batch variance: var = (1/m) * sum((x - mu)^2)
  3. Normalize: x_hat = (x - mu) / sqrt(var + epsilon)
  4. Scale and shift: y = gamma * x_hat + beta

Input statistics:
  Mean: 0.0047, Std: 1.0012

Normalized statistics:
  Mean: 0.0000, Std: 0.9922
```

1.2 PyTorch BatchNorm Layers

```
# PyTorch Batch Normalization
print("PyTorch BatchNorm Layers")
print("="*70)

bn1d = nn.BatchNorm1d(num_features=64)
bn2d = nn.BatchNorm2d(num_features=32)

print("BatchNorm1d parameters:")
print(f"  num_features: 64")
print(f"  weight (gamma) shape: {bn1d.weight.shape}")
print(f"  bias (beta) shape: {bn1d.bias.shape}")
print(f"  running_mean shape: {bn1d.running_mean.shape}")
```

```
print(f"  running_var shape: {bn1d.running_var.shape}")

print("\nTraining vs Evaluation Mode:")
print("  Training: Uses batch statistics")
print("  Eval: Uses running statistics (accumulated during training)")

class MLPWithBatchNorm(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.bn1 = nn.BatchNorm1d(hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.bn2 = nn.BatchNorm1d(hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.relu(self.bn1(self.fc1(x)))
        x = torch.relu(self.bn2(self.fc2(x)))
        return self.fc3(x)

model = MLPWithBatchNorm(784, 256, 10)
print(f"\nModel with BatchNorm:")
print(model)
```

```
PyTorch BatchNorm Layers
======================================================================
BatchNorm1d parameters:
  num_features: 64
  weight (gamma) shape: torch.Size([64])
  bias (beta) shape: torch.Size([64])
  running_mean shape: torch.Size([64])
  running_var shape: torch.Size([64])

Training vs Evaluation Mode:
  Training: Uses batch statistics
  Eval: Uses running statistics (accumulated during training)

Model with BatchNorm:
MLPWithBatchNorm(
  (fc1): Linear(in_features=784, out_features=256, bias=True)
  (bn1): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True)
  (fc2): Linear(in_features=256, out_features=256, bias=True)
  (bn2): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True)
  (fc3): Linear(in_features=256, out_features=10, bias=True)
)
```

2. Dropout

2.1 Dropout Mechanism

```
print("Dropout Regularization")
print("="*70)

print("How Dropout Works:")
print("  Training: Randomly set neurons to zero with probability p")
print("  Testing: Use all neurons, scale outputs by (1-p)")
print("\nInverted dropout (PyTorch default):")
print("  Training: Zero neurons with prob p, scale remaining by 1/(1-p)")
print("  Testing: Use all neurons unchanged")

def dropout_manual(x, p=0.5, training=True):
    if not training or p == 0:
        return x
    mask = (torch.rand_like(x) &gt; p).float()
    return mask * x / (1 - p)

torch.manual_seed(42)
x = torch.ones(2, 10)
print(f"\nInput (all ones):\n{x}")

dropped = dropout_manual(x, p=0.5)
print(f"\nAfter dropout (p=0.5):\n{dropped}")
print(f"Mean preserves: {dropped.mean():.2f} (should be ~1.0)")

dropout = nn.Dropout(p=0.5)
dropout.train()
y_train = dropout(x)
print(f"\nPyTorch Training mode:\n{y_train}")

dropout.eval()
y_eval = dropout(x)
print(f"\nPyTorch Eval mode (unchanged):\n{y_eval}")
```

```
Dropout Regularization
======================================================================
How Dropout Works:
  Training: Randomly set neurons to zero with probability p
  Testing: Use all neurons, scale outputs by (1-p)

Inverted dropout (PyTorch default):
  Training: Zero neurons with prob p, scale remaining by 1/(1-p)
  Testing: Use all neurons unchanged

Input (all ones):
tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])

After dropout (p=0.5):
tensor([[2., 0., 2., 2., 0., 2., 0., 2., 0., 2.],
        [2., 0., 0., 2., 2., 0., 2., 0., 2., 0.]])
Mean preserves: 1.00 (should be ~1.0)

PyTorch Training mode:
tensor([[2., 0., 2., 0., 2., 2., 0., 0., 2., 2.],
        [2., 2., 0., 2., 0., 0., 2., 2., 0., 0.]])

PyTorch Eval mode (unchanged):
tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

2.2 Dropout Best Practices

```
print("Dropout Placement Best Practices")
print("="*70)

class MLPWithDropout(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, dropout_p=0.5):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
```

```
        self.dropout1 = nn.Dropout(dropout_p)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.dropout2 = nn.Dropout(dropout_p)
        self.fc3 = nn.Linear(hidden_size, output_size)
        # Note: No dropout after output layer!

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.dropout1(x)
        x = torch.relu(self.fc2(x))
        x = self.dropout2(x)
        return self.fc3(x)

print("Best Practices:")
print("  1. Apply dropout AFTER activation function")
print("  2. Don't apply dropout to output layer")
print("  3. Common rates: 0.2-0.5 for hidden layers")
print("  4. Use lower rates (0.1-0.2) for input layer if needed")
print("  5. Always remember model.train() vs model.eval()")

print("\nDropout Variants:")
print("  nn.Dropout(p)      - Standard dropout")
print("  nn.Dropout2d(p)    - Spatial dropout for CNNs (drops channels)")
print("  nn.AlphaDropout(p) - For SELU activation")
```

```
Dropout Placement Best Practices
======================================================================
Best Practices:
  1. Apply dropout AFTER activation function
  2. Don't apply dropout to output layer
  3. Common rates: 0.2-0.5 for hidden layers
  4. Use lower rates (0.1-0.2) for input layer if needed
  5. Always remember model.train() vs model.eval()

Dropout Variants:
  nn.Dropout(p)      - Standard dropout
  nn.Dropout2d(p)    - Spatial dropout for CNNs (drops channels)
  nn.AlphaDropout(p) - For SELU activation
```

3. L1 and L2 Regularization

3.1 Weight Decay

```
print("L1 and L2 Regularization")
print("="*70)

print("L2 Regularization (Weight Decay):")
print("  Loss = Original_Loss + (lambda/2) * sum(w^2)")
print("  Effect: Penalizes large weights, encourages small weights")
print("  In PyTorch: Use weight_decay parameter in optimizer")

print("\nL1 Regularization (Lasso):")
print("  Loss = Original_Loss + lambda * sum(|w|)")
print("  Effect: Encourages sparse weights (many zeros)")
print("  Must implement manually in PyTorch")

import torch.optim as optim
model = nn.Linear(10, 5)
optimizer_l2 = optim.SGD(model.parameters(), lr=0.01, weight_decay=0.01)

print("\nL2 Regularization with weight_decay:")
print("  optimizer = SGD(params, lr=0.01, weight_decay=0.01)")

def l1_regularization(model, lambda_l1):
    l1_norm = sum(p.abs().sum() for p in model.parameters())
    return lambda_l1 * l1_norm

# Example training with L1
model = nn.Linear(10, 5)
optimizer = optim.SGD(model.parameters(), lr=0.01)
criterion = nn.MSELoss()

x = torch.randn(32, 10)
y = torch.randn(32, 5)

pred = model(x)
mse_loss = criterion(pred, y)
l1_loss = l1_regularization(model, lambda_l1=0.001)
total_loss = mse_loss + l1_loss

print(f"\nManual L1 Regularization:")
print(f"  MSE Loss: {mse_loss.item():.4f}")
print(f"  L1 Loss: {l1_loss.item():.4f}")
print(f"  Total Loss: {total_loss.item():.4f}")
```

```
L1 and L2 Regularization
======================================================================
L2 Regularization (Weight Decay):
  Loss = Original_Loss + (lambda/2) * sum(w^2)
  Effect: Penalizes large weights, encourages small weights
  In PyTorch: Use weight_decay parameter in optimizer

L1 Regularization (Lasso):
  Loss = Original_Loss + lambda * sum(|w|)
  Effect: Encourages sparse weights (many zeros)
  Must implement manually in PyTorch

L2 Regularization with weight_decay:
  optimizer = SGD(params, lr=0.01, weight_decay=0.01)

Manual L1 Regularization:
  MSE Loss: 1.2847
  L1 Loss: 0.0234
  Total Loss: 1.3081
```

4. Early Stopping and Data Augmentation

4.1 Early Stopping

```
print("Early Stopping")
print("="*70)

class EarlyStopping:
    def __init__(self, patience=5, min_delta=0.001):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss):
        if self.best_loss is None:
            self.best_loss = val_loss
        elif val_loss &gt; self.best_loss - self.min_delta:
            self.counter += 1
            if self.counter &gt;= self.patience:
                self.early_stop = True
        else:
            self.best_loss = val_loss
            self.counter = 0
        return self.early_stop

early_stopping = EarlyStopping(patience=3, min_delta=0.01)
val_losses = [2.5, 2.0, 1.5, 1.2, 1.1, 1.15, 1.18, 1.20, 1.22]

print("Simulated training with early stopping:")
print(f"{'Epoch':&gt;6} {'Val Loss':&gt;10} {'Best':&gt;10} {'Patience':&gt;10}")
print("-" * 40)

for epoch, loss in enumerate(val_losses, 1):
    should_stop = early_stopping(loss)
    print(f"{epoch:&gt;6} {loss:&gt;10.2f} {early_stopping.best_loss:&gt;10.2f} "
          f"{early_stopping.counter:&gt;10}")
    if should_stop:
        print(f"\nStopping early at epoch {epoch}!")
        break
```

```
Early Stopping
======================================================================
Simulated training with early stopping:
 Epoch   Val Loss       Best   Patience
----------------------------------------
     1       2.50       2.50          0
     2       2.00       2.00          0
     3       1.50       1.50          0
     4       1.20       1.20          0
     5       1.10       1.10          0
     6       1.15       1.10          1
     7       1.18       1.10          2
     8       1.20       1.10          3

Stopping early at epoch 8!
```

4.2 Data Augmentation

```
print("Data Augmentation")
print("="*70)

print("Data Augmentation: Artificially increase training set size")
print("  - Create modified versions of training examples")
print("  - Model learns invariances")
print("  - Very effective regularization technique")

print("\nCommon Image Augmentations:")
print("  transforms.RandomHorizontalFlip(p=0.5)")
print("  transforms.RandomRotation(degrees=15)")
print("  transforms.RandomResizedCrop(224, scale=(0.8, 1.0))")
print("  transforms.ColorJitter(brightness=0.2, contrast=0.2)")
```

```
print("  transforms.RandomAffine(degrees=0, translate=(0.1, 0.1))")

print("\nExample training transform pipeline:")
print("""
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=15),
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
""")

print("Important: Only augment training data, not test data!")

print("\nAdvanced Augmentation Techniques:")
print("  - Mixup: Blend two images and labels")
print("  - Cutout: Randomly mask square regions")
print("  - CutMix: Replace regions with patches from other images")
print("  - AutoAugment: Learn augmentation policies")
```

```
Data Augmentation
======================================================================
Data Augmentation: Artificially increase training set size
  - Create modified versions of training examples
  - Model learns invariances
  - Very effective regularization technique

Common Image Augmentations:
  transforms.RandomHorizontalFlip(p=0.5)
  transforms.RandomRotation(degrees=15)
  transforms.RandomResizedCrop(224, scale=(0.8, 1.0))
  transforms.ColorJitter(brightness=0.2, contrast=0.2)
  transforms.RandomAffine(degrees=0, translate=(0.1, 0.1))

Example training transform pipeline:

train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=15),
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

Important: Only augment training data, not test data!

Advanced Augmentation Techniques:
  - Mixup: Blend two images and labels
  - Cutout: Randomly mask square regions
  - CutMix: Replace regions with patches from other images
  - AutoAugment: Learn augmentation policies
```

Summary

Key Takeaways:

- Batch normalization stabilizes training by normalizing layer inputs
- Dropout randomly zeros neurons during training for regularization
- L2 regularization (weight decay) penalizes large weights
- L1 regularization encourages sparse weights
- Early stopping prevents overfitting by monitoring validation loss
- Data augmentation increases effective training set size
- Always use model.train() and model.eval() appropriately

Practice Exercises:

1. Implement batch normalization from scratch
2. Compare training with and without dropout
3. Experiment with different L2 weight decay values
4. Implement early stopping with model checkpointing
5. Create a custom data augmentation pipeline