Transfer Learning Optimization: Normalization Techniques and Gradient Dynamics

Dr. Mahdi Eftekhari
Shahid Bahonar University of Kerman

March 8, 2025

Deadline: Three weeks from today

Assignment Overview

In this assignment, you will explore how different normalization techniques and gradient clipping affect the fine-tuning of pre-trained convolutional neural networks. You will use a pre-trained model (MobileNetV2) and adapt it to classify images from the CIFAR-10 dataset. Through systematic experimentation and analysis, you will gain insights into transfer learning optimization strategies.

Learning Objectives

By completing this assignment, you will:

- Understand how different normalization techniques affect transfer learning dynamics
- Implement custom adaptation layers for pre-trained models
- Analyze gradient flow patterns during fine-tuning
- Evaluate the impact of gradient clipping on training stability
- Visualize and interpret loss landscapes in transfer learning scenarios
- Develop skills in experimental design and analysis for deep learning

Prerequisites

- Basic understanding of convolutional neural networks
- Familiarity with PyTorch
- Knowledge of backpropagation and gradient-based optimization
- Understanding of basic transfer learning concepts

Detailed Step-by-Step Guide

Step 1: Environment Setup and Data Preparation

1. Set up the Colab environment with the required libraries:
    - Pytorch
    - Torchvision
    - Matplotlib
    - Numpy
    - Pandas
2. Load and explore the CIFAR-10 dataset:
    - Examine sample images
    - Understand class distribution
    - Calculate mean and standard deviation for normalization
3. Implement data preprocessing pipeline:
    - Image normalization
    - Data augmentation (random crops, flips)
    - Create training, validation, and test data loaders
4. Implement a function to visualize sample images from each class

Step 2: Base Model Setup

1. Load a pre-trained MobileNetV2 model from torchvision.models:
    - Examine its architecture
    - Configure it to preserve gradient information for later analysis
2. Modify the model for CIFAR-10:
    - Remove the original classification head
    - Add appropriate resizing for CIFAR-10 images (32x32) to match MobileNetV2 input size (224x224)
    - Implement a function that freezes the base model layers
3. Create a baseline adaptation head:
    - Global average pooling
    - A fully connected layer with BatchNorm and ReLU
    - Output layer with 10 units (for CIFAR-10 classes)
4. Test the complete pipeline with a small batch of data:
    - Verify forward pass works
    - Check that gradients flow correctly
    - Ensure output dimensions match expectations

Step 3: Implementing Normalization Variants

1. Implement three different adaptation heads:

   - **Head A**: with Batch Normalization

```python
class BatchNormHead(nn.Module):
    def __init__(self, input_features, dropout_rate=0.5):
        super(BatchNormHead, self).__init__()
        self.global_pool = nn.AdaptiveAvgPool2d(1)
        self.fc1 = nn.Linear(input_features, 256)
        self.bn1 = nn.BatchNorm1d(256)
        self.dropout = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        # Implementation details...
```

   - **Head B**: with Layer Normalization

```python
class LayerNormHead(nn.Module):
    def __init__(self, input_features, dropout_rate=0.5):
        super(LayerNormHead, self).__init__()
        # Implementation details...
```

   - **Head C**: with Filter Response Normalization

```python
class FilterResponseNorm(nn.Module):
    def __init__(self, num_features, epsilon=1e-6):
        super(FilterResponseNorm, self).__init__()
        # Implement FRN from scratch
        # Implementation details...

class FRNHead(nn.Module):
    def __init__(self, input_features, dropout_rate=0.5):
        super(FRNHead, self).__init__()
        # Implementation details...
```

2. Create a factory function that instantiates the appropriate head based on a parameter:

```python
def create_adaptation_head(norm_type, input_features, dropout_rate
    =0.5):
    if norm_type == 'batch':
        return BatchNormHead(input_features, dropout_rate)
    elif norm_type == 'layer':
        return LayerNormHead(input_features, dropout_rate)
    elif norm_type == 'frn':
        return FRNHead(input_features, dropout_rate)
    else:
        raise ValueError(f"Unsupported normalization type: {norm_type
            }")
```

3. Implement a complete model class that combines the base model with an adaptation head:

```python
class TransferModel(nn.Module):
    def __init__(self, base_model, adaptation_head):
        super(TransferModel, self).__init__()
        self.base_model = base_model
        self.adaptation_head = adaptation_head

    def forward(self, x):
        # Implementation details...
```

4. Validate all three normalization variants with a small batch of data

Step 4: Gradient Analysis Infrastructure

1. Implement a gradient tracking hook:

```python
class GradientTracker:
    def __init__(self, model, tracked_layers=None):
        self.model = model
        self.gradients = {}
        self.handles = []
        self.setup_hooks(tracked_layers)

    def setup_hooks(self, tracked_layers):
        # Implementation details...

    def reset_gradients(self):
        # Implementation details...
```

2. Create visualization functions for gradient analysis:
    - Histogram of gradient magnitudes
    - Layer-wise gradient norm tracking
    - Temporal evolution of gradients during training
3. Implement gradient clipping functionality:

```python
def train_with_gradient_clipping(model, train_loader, optimizer,
                                criterion, clip_value=1.0):
    # Implementation details...
```

4. Test gradient tracking with a small training run

Step 5: Training Framework

1. Implement a complete training function with comprehensive logging:

```python
def train_model(model, train_loader, val_loader, optimizer, criterion
    ,
                num_epochs=15, device='cuda', use_grad_clip=False,
                clip_value=1.0, gradient_tracker=None):
    # Implementation details...
```

2. Create a validation function:

```python
def validate_model(model, val_loader, criterion, device='cuda'):
    # Implementation details...
```

3. Implement an experiment manager to organize multiple training runs:

```python
class ExperimentManager:
    def __init__(self, save_dir='./experiments'):
        self.save_dir = save_dir
        os.makedirs(save_dir, exist_ok=True)
        self.experiments = {}

    def run_experiment(self, name, model, train_loader, val_loader,
        **kwargs):
        # Implementation details...

    def save_results(self):
        # Implementation details...

    def load_results(self, path):
        # Implementation details...
```

4. Set up experiment configurations:

```python
experiment_configs = [
    {'name': 'batchnorm_no_clip', 'norm_type': 'batch', '
        use_grad_clip': False},
    {'name': 'batchnorm_clip', 'norm_type': 'batch', 'use_grad_clip':
        True},
    {'name': 'layernorm_no_clip', 'norm_type': 'layer', '
        use_grad_clip': False},
    {'name': 'layernorm_clip', 'norm_type': 'layer', 'use_grad_clip':
        True},
    {'name': 'frn_no_clip', 'norm_type': 'frn', 'use_grad_clip':
        False},
    {'name': 'frn_clip', 'norm_type': 'frn', 'use_grad_clip': True},
]
```

Step 6: Loss Landscape Visualization

1. Implement a 2D loss landscape visualization:

```python
def compute_loss_landscape(model, dataloader, criterion, device,
                          alpha_range=(-1, 1), beta_range=(-1, 1),
                          steps=10, direction1=None, direction2=None)
                              :
    # Implementation details...
```

2. Create functions to generate random perturbation directions:

```python
def get_random_directions(model):
    # Implementation details...
```

3. Implement visualization for loss landscapes:

```python
def plot_loss_landscape(landscape_data, title='Loss Landscape'):
    # Implementation details...
```

4. Focus specifically on visualizing the adaptation head parameters:

```python
def compute_head_loss_landscape(model, dataloader, criterion,
                                device, steps=10):
    # Implementation details specifically for adaptation head
       parameters
```

Step 7: Running Experiments

1. Run baseline experiments:
   - Train a model with BatchNorm head without gradient clipping
   - Save checkpoints and training logs
2. Run normalization variant experiments:
   - Train models with all three normalization techniques
   - Save checkpoints, training logs, and gradient statistics
3. Run gradient clipping experiments:
   - Train models with all three normalization techniques plus gradient clipping
   - Save checkpoints, training logs, and gradient statistics
4. Compute loss landscapes for all trained models:
   - Generate and save 2D visualizations
   - Focus on adaptation head parameters

Step 8: Analysis and Visualization

1. Create comparative training curves:
   - Plot training and validation loss
   - Plot training and validation accuracy
   - Compare convergence rates
2. Analyze gradient statistics:
   - Compare gradient magnitude distributions
   - Examine layer-wise gradient norms
   - Analyze temporal gradient behavior
3. Compare loss landscapes:
   - Create side-by-side visualizations
   - Analyze landscape smoothness and convexity
   - Identify patterns related to generalization
4. Create a comprehensive analysis notebook:
   - Organized sections for each experiment
   - Clear visualizations with interpretations
   - Thoughtful discussion of findings

Step 9: Final Report

1. Write a comprehensive report including:
   - Introduction to transfer learning and normalization techniques
   - Experimental setup and methodology
   - Results and analysis
   - Discussion of findings
   - Conclusions and practical recommendations
   - Limitations and future directions
2. Create an executive summary with key findings:
   - Which normalization technique performed best?
   - How did gradient clipping affect each technique?
   - What practical guidelines can be derived?

Deliverables

- Python code implementing all components described above
- Trained model checkpoints for each experiment
- Visualization notebook with all plots and analyses
- Final report (PDF, 5-8 pages)
- Presentation slides summarizing findings (optional, extra points!)

Evaluation Criteria

Your assignment will be evaluated based on:

1. Implementation correctness (30%)
   - Correct implementation of normalization techniques
   - Proper transfer learning setup
   - Functional gradient analysis tools
2. Experimental design (20%)
   - Systematic approach to experiments
   - Appropriate hyperparameter choices
   - Control of confounding variables
3. Analysis depth (30%)
   - Thoroughness of result analysis
   - Quality of visualizations
   - Insights derived from experiments
4. Report quality (20%)
   - Clarity of explanations
   - Depth of discussion
   - Quality of recommendations
   - Presentation of findings

Resources

1. **Papers**:
   - "Filter Response Normalization Layer: Eliminating Batch Dependence in the Training of Deep Neural Networks" (Singh & Krishnan, 2020)
   - "Visualizing the Loss Landscape of Neural Nets" (Li et al., 2018)
   - "Delving Deep into Rectifiers" (He et al., 2015)
2. **PyTorch Documentation**:
   - MobileNetV2: `https://pytorch.org/vision/stable/models/mobilenetv2.html`
   - Normalization Layers: `https://pytorch.org/docs/stable/nn.html#normalization-layers`
3. **Tutorials**:
   - PyTorch Transfer Learning Tutorial: `https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html`

FAQ

Q: How much of the pre-trained model should I freeze?
A: You should freeze all layers of the base MobileNetV2 model except the last convolutional block. This allows some adaptation while preserving most of the pre-trained features.

Q: What if my model doesn't fit in Colab's GPU memory?
A: You can reduce the batch size or use a smaller pre-trained model like MobileNetV2 which is designed to be memory-efficient.

Q: How do I handle the input size mismatch between CIFAR-10 (32x32) and MobileNetV2 (224x224)?
A: You can either resize the CIFAR-10 images to 224x224 or modify the first layers of MobileNetV2. For simplicity, resizing the input images is recommended.

Q: How many epochs should I train for?
A: 15-20 epochs should be sufficient to observe differences between normalization techniques. Since you're fine-tuning rather than training from scratch, convergence should be relatively quick.

Q: What value should I use for gradient clipping?
A: Start with a clip value of 1.0 and experiment if needed. The goal is to see its effect on training stability rather than optimizing for performance.

Tips for Success

1. **Start small:** Test your implementation on a subset of data before running full experiments.
2. **Save frequently:** Colab sessions can disconnect, so save checkpoints regularly.
3. **Monitor resources:** Keep an eye on GPU memory usage to avoid out-of-memory errors.
4. **Visualize early:** Create visualizations as you go rather than waiting until the end.
5. **Focus on analysis:** The quality of your analysis is more important than achieving the highest accuracy.
6. **Be systematic:** Keep careful track of experimental conditions and results.
7. **Compare thoughtfully:** Look beyond accuracy to convergence rate, stability, and generalization.

Good Luck!