Neural Networks - Advanced Handout

Machine Learning for Smarter Innovation

1 Neural Networks - Advanced Handout

Target Audience: Data scientists and ML engineers **Duration**: 90 minutes reading **Level**: Advanced (mathematical foundations, optimization theory)

1.1 Mathematical Foundations

1.1.1 Forward Propagation

For layer l with weights $W^{(l)}$, bias $b^{(l)}$, and activation f:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$
$$a^{(l)} = f(z^{(l)})$$

Where: - $a^{(0)} = x$ (input) - $a^{(L)} = \hat{y}$ (output)

1.1.2 Backpropagation

Loss gradient with respect to output:

$$\delta^{(L)} = \nabla_{a^{(L)}} \mathcal{L} \odot f'(z^{(L)})$$

Gradient propagation (for $l = L-1, ..., 1$):

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \odot f'(z^{(l)})$$

Weight gradients:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^T$$
$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)}$$

1.2 Activation Functions

1.2.1 ReLU and Variants

ReLU: $f(x) = \max(0, x)$, $f'(x) = \mathbf{1}_{x>0}$

Leaky ReLU: $f(x) = \max(\alpha x, x)$ where $\alpha \approx 0.01$

GELU (Gaussian Error Linear Unit):

$$f(x) = x \cdot \Phi(x) \approx 0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$$

Swish: $f(x) = x \cdot \sigma(\beta x)$

1.2.2 Softmax for Classification

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Numerical stability:

$$\text{softmax}(z_i) = \frac{e^{z_i - \max(z)}}{\sum_{j=1}^{K} e^{z_j - \max(z)}}$$

1.3 Loss Functions

1.3.1 Cross-Entropy Loss

Binary:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

Multi-class:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic})$$

1.3.2 Focal Loss (for imbalanced data)

$$\mathcal{L}_{FL} = -\alpha_t(1 - p_t)^{\gamma}\log(p_t)$$

Where γ is focusing parameter (typically 2).

1.4 Optimization Algorithms

1.4.1 Stochastic Gradient Descent (SGD)

$$\theta_{t+1} = \theta_t - \eta\nabla_\theta\mathcal{L}(\theta_t; x_i, y_i)$$

1.4.2 SGD with Momentum

$$v_t = \gamma v_{t-1} + \eta\nabla_\theta\mathcal{L}$$
$$\theta_{t+1} = \theta_t - v_t$$

1.4.3 Adam (Adaptive Moment Estimation)

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Default: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

1.4.4 AdamW (Weight Decay Decoupled)

$$\theta_{t+1} = \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \theta_t \right)$$

1.5 Initialization Strategies

1.5.1 Xavier/Glorot Initialization

For layer with n_{in} inputs and n_{out} outputs:

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}} \right)$$

For tanh activation. Maintains variance through layers.

1.5.2 He/Kaiming Initialization

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}}} \right)$$

For ReLU activation. Accounts for ReLU zeroing half the activations.

```
# PyTorch implementation
nn.init.kaiming_normal_(layer.weight, mode='fan_in', nonlinearity='relu')
nn.init.xavier_uniform_(layer.weight)
```

1.6 Batch Normalization

1.6.1 Forward Pass

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$
$$y_i = \gamma \hat{x}_i + \beta$$

Where μ_B, σ_B^2 are batch statistics; γ, β are learned.

1.6.2 Inference

Use running averages instead of batch statistics:

$$\mu_{running} = (1 - \alpha)\mu_{running} + \alpha\mu_B$$

1.6.3 Layer Normalization (for sequences)

Normalize across features instead of batch:

$$\hat{x}_i = \frac{x_i - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}}$$

1.7 Regularization Theory

1.7.1 Dropout

During training, randomly zero activations with probability p:

$$\tilde{a} = \frac{1}{1-p} \cdot a \cdot m, \quad m_i \sim \text{Bernoulli}(1-p)$$

Interpretation: Ensemble of 2^n sub-networks.

1.7.2 L2 Regularization (Weight Decay)

$$\mathcal{L}_{total} = \mathcal{L}_{data} + \frac{\lambda}{2}\|W\|_2^2$$

Gradient becomes:

$$\nabla_W \mathcal{L}_{total} = \nabla_W \mathcal{L}_{data} + \lambda W$$

1.7.3 Label Smoothing

Instead of one-hot targets, use:

$$y_{smooth} = (1 - \alpha)y_{one-hot} + \frac{\alpha}{K}$$

Typically $\alpha = 0.1$.

1.8 Attention Mechanism

1.8.1 Scaled Dot-Product Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where: - $Q \in \mathbb{R}^{n \times d_k}$ (queries) - $K \in \mathbb{R}^{m \times d_k}$ (keys) - $V \in \mathbb{R}^{m \times d_v}$ (values)

1.8.2 Multi-Head Attention

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, ..., head_h)W^O$$
$$head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

1.8.3 Self-Attention

When $Q = K = V = X$ (same input):

$$\text{SelfAttention}(X) = \text{softmax}\left(\frac{XX^T}{\sqrt{d}}\right)X$$

1.9 Transformer Architecture

1.9.1 Encoder Block

```
Input -> LayerNorm -> MultiHeadAttention -> Residual -> LayerNorm -> FFN ->
    Residual -> Output
```

1.9.2 Position Encoding

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

1.9.3 Implementation

```
class TransformerBlock(nn.Module):
    def __init__(self, d_model, n_heads, d_ff, dropout=0.1):
        super().__init__()
        self.attention = nn.MultiheadAttention(d_model, n_heads, dropout=
    dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.ff = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model),
            nn.Dropout(dropout)
        )

    def forward(self, x, mask=None):
        # Self-attention with residual
        attn_out, _ = self.attention(x, x, x, attn_mask=mask)
        x = self.norm1(x + attn_out)

        # Feed-forward with residual
        ff_out = self.ff(x)
        x = self.norm2(x + ff_out)

        return x
```

1.10 Gradient Issues

1.10.1 Vanishing Gradients

Cause: Saturating activations (sigmoid, tanh) or deep networks.

Solutions: - ReLU activations - Batch/Layer normalization - Residual connections - LSTM/GRU for sequences

1.10.2 Exploding Gradients

Cause: Large weight magnitudes, especially in RNNs.

Solutions: - Gradient clipping: $g \leftarrow \min(1, \frac{\theta}{\|g\|}) \cdot g$ - Weight initialization - Layer normalization

```
# Gradient clipping in PyTorch
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

1.11 Learning Rate Scheduling

1.11.1 Cosine Annealing

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})(1 + \cos(\frac{t}{T}\pi))$$

1.11.2 Warmup + Decay

$$\eta_t = \begin{cases} \eta_{max} \cdot \frac{t}{T_{warmup}} & t < T_{warmup} \\ \eta_{max} \cdot \text{decay}(t - T_{warmup}) & t \geq T_{warmup} \end{cases}$$

```
# PyTorch scheduler
scheduler = optim.lr_scheduler.CosineAnnealingWarmRestarts(
    optimizer, T_0=10, T_mult=2
)

# OneCycleLR (recommended for training)
scheduler = optim.lr_scheduler.OneCycleLR(
    optimizer, max_lr=0.01, epochs=epochs, steps_per_epoch=len(train_loader)
)
```

1.12 Mixed Precision Training

1.12.1 FP16 Training

```
from torch.cuda.amp import autocast, GradScaler

scaler = GradScaler()

for batch in train_loader:
    optimizer.zero_grad()

    with autocast():
        outputs = model(inputs)
        loss = criterion(outputs, targets)

    scaler.scale(loss).backward()
    scaler.step(optimizer)
```

```
    scaler.update()
```

Benefits: 2x memory reduction, faster training on modern GPUs.

1.13 Distributed Training

1.13.1 Data Parallel

```
model = nn.DataParallel(model)  # Simple, single-machine multi-GPU
```

1.13.2 Distributed Data Parallel

```
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP

dist.init_process_group("nccl")
model = DDP(model, device_ids=[local_rank])
```

1.14 Model Compression

1.14.1 Quantization

```
# Post-training quantization
quantized_model = torch.quantization.quantize_dynamic(
    model, {nn.Linear}, dtype=torch.qint8
)
```

1.14.2 Knowledge Distillation

$$\mathcal{L}_{KD} = \alpha\mathcal{L}_{CE}(y, \hat{y}_{student}) + (1-\alpha)T^2\mathcal{L}_{KL}(\sigma(z_T/T), \sigma(z_S/T))$$

1.14.3 Pruning

```
import torch.nn.utils.prune as prune

# Prune 30% of weights with smallest magnitude
prune.l1_unstructured(module, name='weight', amount=0.3)
```

1.15 References

1. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.
2. Vaswani, A., et al. (2017). “Attention Is All You Need”

3. He, K., et al. (2016). "Deep Residual Learning for Image Recognition"
4. Kingma, D. P., & Ba, J. (2015). "Adam: A Method for Stochastic Optimization"

Deep learning is an empirical science. Theory provides guidance, but experimentation determines success.