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 and optimization theory)

1.1 Mathematical Foundations

1.1.1 Universal Approximation Theorem

A feed-forward network with one hidden layer containing finite neurons can approximate any continuous function on compact subsets of $\mathbb{R}^n$ to arbitrary accuracy.

Formally: For any continuous function $f : [0, 1]^n \to \mathbb{R}$ and $\epsilon > 0$, there exists a network g with one hidden layer such that:

$|g(x) - f(x)| < \epsilon$ for all $x \in [0, 1]^n$

This is an existence result. It does not specify how many neurons are needed or guarantee that gradient descent will find the approximation.

1.1.2 Forward Propagation

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

Pre-activation: $z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$

Activation: $a^{(l)} = f(z^{(l)})$

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

For a 3-layer network: $x \to z^{(1)} \to a^{(1)} \to z^{(2)} \to a^{(2)} \to z^{(3)} \to a^{(3)} = \hat{y}$

1.1.3 Backpropagation Derivation

The goal is to compute $\partial L/\partial W^{(l)}$ for all layers l. By chain rule:

Output layer error: $\delta^{(L)} = \frac{\partial L}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} = \nabla_{a^{(L)}} L \cdot f'(z^{(L)})$

For cross-entropy loss with softmax, this simplifies to: $\delta^{(L)} = a^{(L)} - y$

Hidden layer errors (backward recursion): $\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \cdot f'(z^{(l)})$

The term $(W^{(l+1)})^T \delta^{(l+1)}$ propagates error backward; $f'(z^{(l)})$ accounts for local gradient.

Weight gradients: $\partial L/\partial W^{(l)} = \delta^{(l)}(a^{(l-1)})^T$

$\partial L/\partial b^{(l)} = \delta^{(l)}$

Computational complexity: O(number of weights) for both forward and backward pass.

1.2 Activation Functions

1.2.1 ReLU Family

ReLU: $f(x) = \max(0, x)$, $f'(x) = 1$ if $x > 0$ else 0

Advantages: No vanishing gradient for positive inputs, sparse activation. Issue: "Dying ReLU" when neurons stuck at 0.

Leaky ReLU: $f(x) = \max(\alpha \cdot x, x)$, $\alpha \approx 0.01$. $f'(x) = 1$ if $x > 0$ else α

Prevents dying ReLU by allowing small gradients for negative inputs.

PReLU (Parametric): $f(x) = \max(\alpha \cdot x, x)$, α learned per channel

ELU: $f(x) = x$ if $x > 0$ else $\alpha \cdot (e^x - 1)$

Pushes mean activations toward zero, improving gradient flow.

1.2.2 Smooth Activations

GELU (Gaussian Error Linear Unit): $f(x) = x \cdot \Phi(x)$

where Φ is standard Gaussian CDF. Approximation:

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

Used in BERT and GPT. Smooth version of ReLU with probabilistic interpretation.

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

With $\beta = 1$, $\text{swish}(x) = x \cdot \sigma(x)$. Outperforms ReLU on deep networks in practice.

Mish: $f(x) = x \cdot \tanh(\text{softplus}(x)) = x \cdot \tanh(\ln(1 + e^x))$

Self-regularizing, unbounded above, bounded below.

1.2.3 Softmax and Temperature

Standard softmax: $\text{softmax}(z_i) = e^{z_i} / \sum_j e^{z_j}$

With temperature T: $\text{softmax}(z_i/T) = e^{z_i/T} / \sum_j e^{z_j/T}$

- T -> 0: argmax (hard selection)
- T = 1: standard softmax
- T > 1: smoother distribution

Numerical stability: $\text{softmax}(z_i) = e^{z_i - \max(z)} / \sum_j e^{z_j - \max(z)}$

Subtracting $\max(z)$ prevents overflow without changing the result.

1.3 Loss Functions

1.3.1 Cross-Entropy Losses

Binary cross-entropy: $L = -\frac{1}{N} \sum_i [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$

Gradient with respect to logit z (before sigmoid): $\partial L / \partial z = \sigma(z) - y = \hat{y} - y$

Categorical cross-entropy: $L = -\frac{1}{N} \sum_i \sum_c y_{ic} \log(\hat{y}_{ic})$

For one-hot y: $L = -\frac{1}{N} \sum_i \log(\hat{y}_{i,\text{true class}})$

1.3.2 Focal Loss

For highly imbalanced classification: $L_{\text{FL}} = -\alpha_t (1 - p_t)^\gamma \log(p_t)$

where $p_t = p$ if $y = 1$ else $1 - p$, and γ is focusing parameter.

- $\gamma = 0$: standard cross-entropy
- $\gamma = 2$ (typical): down-weights easy examples

1.3.3 Label Smoothing

Replace one-hot targets with: $y_{\text{smooth}} = (1 - \alpha) \cdot y_{\text{one-hot}} + \alpha/K$

where K is number of classes, α typically 0.1.

Effect: Prevents overconfident predictions, acts as regularization.

1.4 Optimization Algorithms

1.4.1 Stochastic Gradient Descent

Vanilla SGD: $\theta_{t+1} = \theta_t - \eta \cdot g_t$

where $g_t = \nabla_\theta L(\theta_t; x_i, y_i)$ is gradient on mini-batch.

SGD with Momentum: $v_t = \gamma \cdot v_{t-1} + \eta \cdot g_t, \quad \theta_{t+1} = \theta_t - v_t$

Momentum γ typically 0.9. Accelerates convergence in consistent gradient directions.

Nesterov Momentum: $v_t = \gamma \cdot v_{t-1} + \eta \cdot \nabla_\theta L(\theta_t - \gamma \cdot v_{t-1}), \quad \theta_{t+1} = \theta_t - v_t$

Look ahead before computing gradient, providing better correction.

1.4.2 Adaptive Methods

Adam (Adaptive Moment Estimation):

- First moment: $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
- Second moment: $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
- Bias correction: $\hat{m}_t = m_t/(1 - \beta_1^t)$, $\hat{v}_t = v_t/(1 - \beta_2^t)$
- Update: $\theta_{t+1} = \theta_t - \eta \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$

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

AdamW (Decoupled Weight Decay): $\theta_{t+1} = \theta_t - \eta \cdot (\hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon) + \lambda \cdot \theta_t)$

Proper weight decay separate from gradient update. Preferred for transformer training.

RAdam (Rectified Adam): Corrects variance in early training when few samples available for moment estimation.

1.4.3 Learning Rate Schedules

Step decay: $\eta_t = \eta_0 \cdot \gamma^{\lfloor t/s \rfloor}$

Cosine annealing: $\eta_t = \eta_{\min} + 0.5 \cdot (\eta_{\max} - \eta_{\min}) \cdot (1 + \cos(t \cdot \pi/T))$

Warmup + decay: $\eta_t = \eta_{\max} \cdot t/T_{\text{warmup}}$ if $t < T_{\text{warmup}}$ else decay$(t - T_{\text{warmup}})$

One-cycle: Ramp up then ramp down learning rate during training. Often achieves faster convergence.

1.5 Initialization

1.5.1 Xavier/Glorot Initialization

For layer with n_in inputs and n_out outputs:

Uniform: $W \tilde{} U(-sqrt(6/(n_i n + n_o ut)), sqrt(6/(n_i n + n_o ut)))$

Normal: $W \tilde{} N(0, sqrt(2/(n_i n + n_o ut)))$

Designed for sigmoid/tanh activations. Maintains variance of activations through layers.

1.5.2 He/Kaiming Initialization

For ReLU: $W \tilde{} N(0, sqrt(2/n_i n))$

Accounts for ReLU zeroing half the activations. Essential for deep ReLU networks.

For Leaky ReLU: $W \tilde{} N(0, sqrt(2/(1 + alpha2))/n_i n))$

1.6 Normalization Techniques

1.6.1 Batch Normalization

Training:

- Mean: $\mu_B = \frac{1}{m} \sum_i x_i$
- Variance: $\sigma_B^2 = \frac{1}{m} \sum_i (x_i - \mu_B)^2$
- Normalize: $\hat{x}_i = (x_i - \mu_B)/\sqrt{\sigma_B^2 + \epsilon}$
- Scale and shift: $y_i = \gamma \cdot \hat{x}_i + \beta$

Inference: Use running averages: $\mu_{\text{running}} = (1 - \alpha) \cdot \mu_{\text{running}} + \alpha \cdot \mu_B$

Benefits: Reduces internal covariate shift, allows higher learning rates, acts as regularization.

1.6.2 Layer Normalization

Normalize across features instead of batch: $\mu_L = \frac{1}{H} \sum_i x_i$, $\sigma_L^2 = \frac{1}{H} \sum_i (x_i - \mu_L)^2$, $\hat{x} = (x - \mu_L)/\sqrt{\sigma_L^2 + \epsilon}$

Advantages: Batch-size independent, preferred for transformers and RNNs.

1.6.3 RMSNorm

Simplified normalization: $\hat{x} = x/\text{RMS}(x)$ where $\text{RMS}(x) = \sqrt{\frac{1}{n} \sum_i x_i^2}$

Removes mean centering, computationally cheaper. Used in LLaMA.

1.7 Regularization

1.7.1 Dropout

Training: $\tilde{a} = a \cdot m/(1 - p)$

where $m_i \sim \text{Bernoulli}(1 - p)$ and p is dropout probability.

Interpretation: Trains ensemble of 2^n subnetworks. At test time, uses full network (equivalent to ensemble average).

Spatial Dropout: Drop entire feature maps in CNNs. **DropConnect:** Drop weights instead of activations.

1.7.2 Weight Decay

L2 regularization: $L_{\text{total}} = L_{\text{data}} + \frac{\lambda}{2}\|W\|_2^2$

Gradient: $\partial L_{\text{total}}/\partial W = \partial L_{\text{data}}/\partial W + \lambda W$

Equivalent weight update: $W \leftarrow (1 - \eta \cdot \lambda)W - \eta \cdot \partial L_{\text{data}}/\partial W$

Shrinks weights toward zero, preventing large weight magnitudes.

1.7.3 Data Augmentation

Implicit regularization through training data expansion:

- Images: rotation, flip, crop, color jitter
- Text: synonym replacement, back-translation
- Mixup: $x = \lambda \cdot x_i + (1 - \lambda) \cdot x_j$

1.8 Implementation

1.8.1 Setup

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import numpy as np
from typing import List, Tuple, Optional
import warnings
warnings.filterwarnings('ignore')

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

1.8.2 Custom Layers

```
class ResidualBlock(nn.Module):
    """Residual block with skip connection."""

    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, 3, stride, 1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Skip connection
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, 1, stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)  # Skip connection
        out = F.relu(out)
        return out


class TransformerBlock(nn.Module):
    """Transformer encoder block."""
```

```
    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)
        )
        self.dropout = nn.Dropout(dropout)

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

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

        return x
```

1.8.3 Production Training Loop

```
class NeuralNetworkTrainer:
    """Production training with best practices."""

    def __init__(self, model, criterion, optimizer, scheduler=None):
        self.model = model.to(device)
        self.criterion = criterion
        self.optimizer = optimizer
        self.scheduler = scheduler
        self.scaler = torch.cuda.amp.GradScaler()  # Mixed precision

        self.train_losses = []
        self.val_losses = []
        self.best_val_loss = float('inf')

    def train_epoch(self, train_loader, use_amp=True):
        """Train one epoch with optional mixed precision."""
        self.model.train()
        total_loss = 0

        for batch_idx, (data, target) in enumerate(train_loader):
            data, target = data.to(device), target.to(device)
            self.optimizer.zero_grad()

            if use_amp:
                with torch.cuda.amp.autocast():
                    output = self.model(data)
                    loss = self.criterion(output, target)
                self.scaler.scale(loss).backward()
                self.scaler.unscale_(self.optimizer)
                torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)
                self.scaler.step(self.optimizer)
                self.scaler.update()
            else:
                output = self.model(data)
                loss = self.criterion(output, target)
                loss.backward()
                torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)
                self.optimizer.step()

            total_loss += loss.item()

        if self.scheduler:
            self.scheduler.step()

        return total_loss / len(train_loader)
```

```
    @torch.no_grad()
    def evaluate(self, val_loader):
        """Evaluate on validation set."""
        self.model.eval()
        total_loss = 0
        correct = 0
        total = 0

        for data, target in val_loader:
            data, target = data.to(device), target.to(device)
            output = self.model(data)
            loss = self.criterion(output, target)
            total_loss += loss.item()

            pred = output.argmax(dim=1)
            correct += (pred == target).sum().item()
            total += target.size(0)

        return total_loss / len(val_loader), correct / total

    def fit(self, train_loader, val_loader, epochs, patience=10):
        """Full training with early stopping."""
        no_improve = 0

        for epoch in range(epochs):
            train_loss = self.train_epoch(train_loader)
            val_loss, val_acc = self.evaluate(val_loader)

            self.train_losses.append(train_loss)
            self.val_losses.append(val_loss)

            print(f"Epoch {epoch+1}: Train Loss={train_loss:.4f}, "
                  f"Val Loss={val_loss:.4f}, Val Acc={val_acc:.4f}")

            if val_loss < self.best_val_loss:
                self.best_val_loss = val_loss
                torch.save(self.model.state_dict(), 'best_model.pt')
                no_improve = 0
            else:
                no_improve += 1

            if no_improve >= patience:
                print(f"Early stopping at epoch {epoch+1}")
                break

        # Load best model
        self.model.load_state_dict(torch.load('best_model.pt'))
        return self.train_losses, self.val_losses
```

1.8.4 Model Compression

```
class ModelCompressor:
    """Compression techniques for deployment."""

    @staticmethod
    def quantize_dynamic(model):
        """Dynamic quantization for CPU inference."""
        return torch.quantization.quantize_dynamic(
            model, {nn.Linear, nn.LSTM, nn.GRU}, dtype=torch.qint8
        )

    @staticmethod
    def prune_weights(model, amount=0.3):
        """Magnitude-based pruning."""
        import torch.nn.utils.prune as prune

        for name, module in model.named_modules():
            if isinstance(module, nn.Linear):
                prune.l1_unstructured(module, name='weight', amount=amount)
                prune.remove(module, 'weight')

        return model

    @staticmethod
```

```
    def knowledge_distillation_loss(student_logits, teacher_logits, labels,
                                    temperature=4.0, alpha=0.5):
        """Combined loss for knowledge distillation."""
        soft_loss = F.kl_div(
            F.log_softmax(student_logits / temperature, dim=1),
            F.softmax(teacher_logits / temperature, dim=1),
            reduction='batchmean'
        ) * (temperature ** 2)

        hard_loss = F.cross_entropy(student_logits, labels)

        return alpha * soft_loss + (1 - alpha) * hard_loss
```

1.9 Common Parameters

Component	Parameter	Typical Range	Notes
SGD	learning_rate	0.01-0.1	With momentum
Adam	learning_rate	1e-4 to 1e-3	Often requires tuning
Adam	beta_1	0.9	First moment decay
Adam	beta_2	0.999	Second moment decay
Dropout	rate	0.1-0.5	Higher for larger models
BatchNorm	momentum	0.1	Running average update
Weight Decay	lambda	1e-4 to 1e-2	L2 regularization
Gradient Clipping	max_norm	1.0-5.0	Prevents explosion
Label Smoothing	alpha	0.1	Regularization effect
Warmup	steps	1000-10000	Larger for bigger models

1.10 Practice Problems

1. **Gradient Flow Analysis:** Implement a deep network (20+ layers) with and without batch normalization. Plot gradient norms at each layer during training. Verify that normalization prevents vanishing gradients.
2. **Optimizer Comparison:** Train the same architecture using SGD, SGD+momentum, Adam, and AdamW. Plot training curves. Identify which optimizer converges fastest and which achieves best final performance.
3. **Initialization Study:** Initialize a 10-layer ReLU network with Xavier and He initialization. Compare activation distributions at each layer. Verify He maintains better variance for ReLU.
4. **Pruning vs Accuracy:** Progressively prune a trained network (10%, 30%, 50%, 70%, 90% sparsity). Plot accuracy vs sparsity. Identify the pruning threshold where accuracy degrades significantly.
5. **Knowledge Distillation:** Train a large teacher model and distill to a 10x smaller student. Compare student accuracy with and without distillation. Vary temperature and alpha to optimize distillation.

1.11 References

1. Goodfellow, I., Bengio, Y., & Courville, A. (2016). "Deep Learning." MIT Press.
2. Vaswani, A., et al. (2017). "Attention Is All You Need." NeurIPS.
3. He, K., et al. (2016). "Deep Residual Learning for Image Recognition." CVPR.
4. Kingma, D. P., & Ba, J. (2015). "Adam: A Method for Stochastic Optimization." ICLR.

5. Ioffe, S., & Szegedy, C. (2015). "Batch Normalization: Accelerating Deep Network Training." ICML.
6. Srivastava, N., et al. (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." JMLR.
7. Loshchilov, I., & Hutter, F. (2019). "Decoupled Weight Decay Regularization." ICLR.

Neural networks combine theoretical principles with empirical practice. Mathematical understanding of forward/backward propagation, optimization dynamics, and regularization enables building models that train efficiently and generalize reliably.