Notas de Aula - Semana 4

Otimização de Redes Neurais

Curso de Redes Neurais

Eduardo Adame

27 de agosto de 2025

Sumário

1 Introdução

O treinamento de redes neurais é um processo iterativo de otimização que busca encontrar os parâmetros (pesos e bias) que minimizam uma função de perda. Após dominarmos o algoritmo de backpropagation para calcular gradientes, precisamos entender **como** e **quando** atualizar os pesos da rede.

Este documento explora os detalhes práticos do treinamento, incluindo estratégias de atualização de pesos, normalização de dados, e implementação em frameworks modernos.

2 Estratégias de Atualização de Pesos

2.1 O Problema Fundamental

Dado o gradiente $\frac{\partial J}{\partial W}$ para cada peso W na rede, precisamos decidir:

1. **Como atualizar**: Qual algoritmo de otimização usar?
2. **Quanto atualizar**: Qual taxa de aprendizado α escolher?
3. **Quando atualizar**: Após cada exemplo, ou após vários?

Definição 2.1: Regra de Atualização Básica

A atualização de pesos em redes neurais segue a regra geral:

$$W^{(t+1)} = W^{(t)} - \alpha \cdot \frac{\partial J}{\partial W^{(t)}}$$

onde $\alpha > 0$ é a taxa de aprendizado e t indica a iteração.

2.2 Descida de Gradiente em Lote (Batch Gradient Descent)

Definição 2.2: Gradiente em Lote Completo

No gradiente descendente em lote, calculamos o gradiente sobre **todo** o conjunto de treinamento:

$$\frac{\partial J}{\partial W} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial J_i}{\partial W}$$

onde N é o número total de exemplos e J_i é a perda para o exemplo i.

Vantagens:

- Convergência estável e suave
- Gradiente preciso da função de custo real
- Garantias teóricas de convergência

Desvantagens:

- Computacionalmente caro para datasets grandes

- Requer manter todo o dataset em memória
- Pode ficar preso em mínimos locais rasos
- Uma única atualização por época

2.3 Descida de Gradiente Estocástica (SGD)

Definição 2.3: Gradiente Estocástico

No SGD, atualizamos os pesos após **cada exemplo individual**:

$$W^{(t+1)} = W^{(t)} - \alpha \cdot \frac{\partial J_i}{\partial W^{(t)}}$$

onde i é um exemplo escolhido aleatoriamente.

Vantagens:

- Muito mais rápido por iteração
- Pode escapar de mínimos locais devido ao ruído
- Permite aprendizado online (streaming)
- Requer menos memória

Desvantagens:

- Convergência ruidosa e instável
- Pode oscilar ao redor do mínimo
- Dificulta uso de vetorização eficiente
- Requer taxa de aprendizado menor

2.4 Mini-batch Gradient Descent

Definição 2.4: Mini-batch

Mini-batch é um compromisso entre batch e SGD, usando subconjuntos de tamanho B:

$$\frac{\partial J}{\partial W} = \frac{1}{B} \sum_{i \in \mathcal{B}} \frac{\partial J_i}{\partial W}$$

onde $\mathcal{B}$ é um mini-batch de B exemplos, tipicamente $B \in \{16, 32, 64, 128\}$.

Teorema 2.1: Convergência do Mini-batch

Para uma função de perda convexa J com gradiente Lipschitz contínuo, o mini-batch gradient descent com taxa de aprendizado apropriada converge para o mínimo global com taxa $O(1/\sqrt{T})$, onde T é o número de iterações.

Observação 2.1: Tamanho Ótimo do Batch

O tamanho do batch influencia:

- **Batch pequeno** ($B < 32$): Mais ruído, melhor generalização, convergência mais lenta
- **Batch médio** ($B \in [32, 256]$): Equilíbrio entre velocidade e estabilidade
- **Batch grande** ($B > 256$): Convergência suave, mas pode generalizar pior

3 Conceitos Fundamentais de Treinamento

3.1 Época e Iteração

Definição 3.1: Época

Uma **época** é uma passada completa por todo o conjunto de treinamento. Se temos N exemplos e usamos mini-batches de tamanho B, então:

$$\text{Iterações por época} = \left\lceil \frac{N}{B} \right\rceil$$

Exemplo 3.1: Cálculo de Iterações

Considere um dataset com 60.000 exemplos (como o MNIST):

- **Batch completo**: 1 iteração por época
- **SGD** (batch = 1): 60.000 iterações por época
- **Mini-batch** (batch = 32): $\lceil 60.000/32 \rceil = 1.875$ iterações por época

3.2 Embaralhamento de Dados

Observação 3.1: Importância do Embaralhamento

Embaralhar os dados a cada época é crucial porque:

1. Quebra correlações temporais nos dados
2. Garante que cada mini-batch seja representativo
3. Evita ciclos na trajetória de otimização
4. Melhora a convergência em até 30% em alguns casos

4 Normalização de Entrada

4.1 Por que Normalizar?

A normalização dos dados de entrada é fundamental para o treinamento eficiente de redes neurais.

Teorema 4.1: Efeito da Escala no Gradiente

Para uma rede com entrada $\mathbf{x}$ e pesos $\mathbf{W}$, se escalarmos a entrada por λ, o gradiente é escalado por:

$$\frac{\partial J}{\partial \mathbf{W}}(\lambda \mathbf{x}) = \lambda \cdot \frac{\partial J}{\partial \mathbf{W}}(\mathbf{x})$$

Isso pode causar instabilidade numérica quando features têm escalas muito diferentes.

4.2 Métodos de Normalização

Definição 4.1: Normalização Min-Max

Transforma os dados para o intervalo $[0, 1]$:

$$x'_i = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$$

ou para o intervalo $[-1, 1]$:

$$x'_i = 2 \cdot \frac{x_i - x_{\min}}{x_{\max} - x_{\min}} - 1$$

Definição 4.2: Padronização (Z-score)

Transforma os dados para ter média zero e desvio padrão unitário:

$$x'_i = \frac{x_i - \mu}{\sigma}$$

onde $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$ e $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$

Observação 4.1: Quando Usar Cada Método

- **Min-Max**: Quando os dados têm limites conhecidos e distribuição uniforme
- **Z-score**: Quando os dados seguem distribuição aproximadamente normal
- **Normalização L2**: Para dados direcionais ou quando a magnitude não importa

5 Classificação Multiclasse

5.1 Função Softmax

Definição 5.1: Softmax

A função softmax é uma generalização da função sigmoide para múltiplas classes:

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

onde K é o número de classes. Propriedades:

- $0 < \text{softmax}(z_i) < 1$ para todo i
- $\sum_{i=1}^{K} \text{softmax}(z_i) = 1$

5.2 Entropia Cruzada Categórica

Definição 5.2: Cross-Entropy Loss

Para classificação multiclasse com K classes:

$$J = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \log(\hat{y}_{ik})$$

onde y_{ik} é 1 se o exemplo i pertence à classe k (one-hot encoding), e $\hat{y}_{ik}$ é a probabilidade predita.

Teorema 5.1: Gradiente Softmax-CrossEntropy

O gradiente da cross-entropy com relação às logits (entrada do softmax) tem forma elegante:

$$\frac{\partial J}{\partial z_i} = \hat{y}_i - y_i$$

Esta simplicidade é uma das razões para a popularidade desta combinação.

6 Implementação Prática

6.1 Algoritmo Completo de Treinamento

Algoritmo 6.1: Mini-batch SGD com Momentum

Algorithm 1 Treinamento de Rede Neural com Mini-batch SGD

Entrada: Dataset $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$, taxa α, batch size B, épocas E
Inicialização: Pesos $W \sim \mathcal{N}(0, \sigma^2)$
for época $= 1$ até E **do**
  $\mathcal{D}_{shuffled} \leftarrow$ Embaralhar($\mathcal{D}$)
  Dividir $\mathcal{D}_{shuffled}$ em mini-batches de tamanho B
  for cada mini-batch $\mathcal{B}$ **do**
    // Forward Pass
    $\hat{\mathbf{y}} \leftarrow$ ForwardPropagation($\mathcal{B}$, W)
    $J \leftarrow$ CalcularPerda($\mathbf{y}$, $\hat{\mathbf{y}}$)
    // Backward Pass
    $\nabla W \leftarrow$ BackPropagation(J, W)
    // Atualização
    $W \leftarrow W - \alpha \cdot \nabla W$
  end for
  Avaliar no conjunto de validação
end for
Retorna: Pesos treinados W

6.2 Considerações de Implementação

Observação 6.1: Eficiência Computacional

Para maximizar a eficiência:

1. **Vetorização**: Processar todo o mini-batch como uma matriz
2. **GPU**: Batches maiores aproveitam melhor o paralelismo
3. **Pré-processamento**: Normalizar offline quando possível
4. **Cache**: Manter mini-batches frequentes em memória rápida

7 Diagnóstico e Debugging

7.1 Curvas de Aprendizado

Definição 7.1: Curvas de Treinamento

Monitorar durante o treinamento:

- **Loss de treino**: Deve diminuir consistentemente
- **Loss de validação**: Indica generalização
- **Acurácia**: Métrica interpretável do desempenho
- **Norma do gradiente**: $\|\nabla W\|$ indica velocidade de aprendizado

7.2 Problemas Comuns e Soluções

Observação 7.1: Diagnóstico de Problemas

- **Loss explode** ($\to \infty$): Taxa de aprendizado muito alta
- **Loss estagna**: Taxa muito baixa ou saturação
- **Loss oscila**: Batch muito pequeno ou taxa alta
- **Validação piora**: Overfitting, precisa regularização

8 Exemplo Prático: Implementação em Keras

Exemplo 8.1: Código Keras Completo

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

# 1. Preparar dados
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()

# 2. Normalização
X_train = X_train.reshape(-1, 784).astype('float32') / 255.0
X_test = X_test.reshape(-1, 784).astype('float32') / 255.0

# 3. One-hot encoding
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

# 4. Construir modelo
model = keras.Sequential([
    keras.layers.Dense(128, activation='relu', input_shape=(784,)),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])

# 5. Compilar
model.compile(
    optimizer=keras.optimizers.SGD(learning_rate=0.01),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# 6. Treinar
history = model.fit(
    X_train, y_train,
    batch_size=32,
    epochs=20,
    validation_split=0.1,
    shuffle=True  # Embaralhamento automático
)
```

9 Considerações Avançadas

9.1 Taxa de Aprendizado Adaptativa

Observação 9.1: Learning Rate Scheduling

Estratégias para ajustar α durante o treinamento:

- **Step decay**: $\alpha_t = \alpha_0 \cdot \gamma^{\lfloor t/s \rfloor}$
- **Exponential decay**: $\alpha_t = \alpha_0 \cdot e^{-kt}$
- **Cosine annealing**: $\alpha_t = \alpha_0 \cdot \cos(\frac{\pi t}{T})$

9.2 Inicialização de Pesos

Teorema 9.1: Inicialização de Xavier/Glorot

Para manter a variância das ativações constante através das camadas, inicialize os pesos com:

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

onde n_{in} e n_{out} são o número de neurônios na camada anterior e atual.