Optimización en Redes Neuronales Profundas

Contenido para la asignatura de Aprendizaje Automático

4 de febrero de 2025

Definición

La **optimización en redes neuronales profundas** se centra en la tarea de ajustar los parámetros (*weights* y *biases*) para minimizar la *función de costo* o *loss function*. El objetivo es que la red extraiga representaciones útiles de los datos de entrada para así resolver tareas de clasificación, regresión u otras aplicaciones. El procedimiento de *backpropagation* calcula el gradiente de la función de costo con respecto a cada parámetro y, mediante distintos algoritmos de optimización (por ejemplo, `SGD`, `Adam`, `RMSProp`), se determinan los pasos de actualización. La elección del optimizador, así como de sus hiperparámetros (*learning rate*, *beta*, etc.), es un factor crucial para el rendimiento y la velocidad de convergencia de la red neuronal.

- **Tasa de Aprendizaje (*learning rate*, α)**: Es un número que determina la magnitud de los ajustes en los pesos durante cada actualización. La regla de actualización típica para un peso w es:

  $$w \leftarrow w - \alpha \frac{\partial \mathcal{L}}{\partial w},$$

  donde $\mathcal{L}$ es la función de costo. Si α es demasiado grande, los pasos son muy bruscos y el entrenamiento puede "descontrolarse"(divergencia). Si es muy pequeño, el proceso es muy lento. Basta con entender que α regula *cuánto* cambiamos los parámetros en cada paso para acercarnos a la solución.

- **Retropropagación (*backpropagation*)**: Es el algoritmo que calcula el gradiente de la función de costo hacia atrás, capa por capa. Básicamente, si $\mathcal{L}$ es el error de la red, la retropropagación encuentra derivadas como:

  $$\frac{\partial \mathcal{L}}{\partial w_i},$$

  para cada peso w_i. Luego, con esas derivadas, se actualizan los pesos. Para entenderlo de forma sencilla, *backpropagation* nos indica la dirección en la que tenemos que mover cada parámetro para disminuir el error.

- **Regularización**: Son técnicas que ayudan a evitar que la red "memorice"los datos de entrenamiento. Por ejemplo, el *weight decay* (L2) agrega un término $\lambda \sum w_i^2$ a la función de costo para que los pesos no crezcan demasiado:

  $$\mathcal{L}_{\text{total}} = \mathcal{L} + \lambda \sum w_i^2.$$

Así, se limita la complejidad del modelo. Otra técnica común es *dropout*, donde se .ªpagan.ªleatoriamente ciertas neuronas en cada paso, forzando a la red a no depender en exceso de conexiones específicas. Para un estudiante de primer año, la idea básica es que la regularización nos protege de la trampa de aprender demasiado bien el entrenamiento y fallar al generalizar.

- **Función de Costo (*loss function*)**: Es la forma de medir *cuánto* se equivoca la red. Por ejemplo, para un problema de clasificación binaria se usa la entropía cruzada binaria:

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

donde y_i es la etiqueta verdadera y $\hat{y}_i$ es la predicción de la red. En palabras sencillas, la función de costo es la forma de indicar a la red *qué tan mal lo está haciendo* y por tanto cómo corregir sus pesos para mejorar.

Algoritmos de Optimización

Algoritmo	Actualización Base	Ventajas	Desventajas
`SGD`	$w \leftarrow w - \alpha \nabla_w \mathcal{L}$	Simplicidad y bajo costo computacional	Sensible a la escala de los datos y a la tasa de aprendizaje
`Adam`	$\begin{cases} m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)\nabla_w \mathcal{L}, \\ v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)\nabla_w^2 \mathcal{L}, \\ w \leftarrow w - \alpha \frac{\frac{m_t}{1-\beta_1^t}}{\sqrt{\frac{v_t}{1-\beta_2^t}}+\epsilon} \end{cases}$	Convergencia rápida y adaptativa	Sensible a la elección de hiperparámetros y al *weight decay*
`RMSProp`	$\begin{cases} r_t \leftarrow \beta r_{t-1} + (1-\beta)\nabla_w^2 \mathcal{L}, \\ w \leftarrow w - \alpha \frac{\nabla_w \mathcal{L}}{\sqrt{r_t}+\epsilon} \end{cases}$	Control adaptativo del paso de aprendizaje	Posible oscilación si β no se elige adecuadamente

Cuadro 1: Comparación resumida de tres métodos de optimización comunes para redes neuronales profundas.

Ejemplo

Objetivo: Ilustrar el proceso de entrenamiento de una red neuronal profunda de 2 capas ocultas para un problema de clasificación binaria en un contexto bancario (predicción de contratación de un producto).

1. **Arquitectura de la Red:**

$$\text{Capa Oculta 1: } h^{(1)} = \text{máx}(0, W^{(1)}x + b^{(1)}) \quad \text{(ReLU)}$$

$$\text{Capa Oculta 2: } h^{(2)} = \text{máx}(0, W^{(2)}h^{(1)} + b^{(2)}) \quad \text{(ReLU)}$$

$$\text{Capa de Salida: } \hat{y} = \sigma\big(W^{(3)}h^{(2)} + b^{(3)}\big) \quad \text{(Sigmoide)}$$

2. **Función de Costo:** Para la clasificación binaria, se utiliza normalmente la *entropía cruzada*:

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

3. **Optimizador:** Se elige `Adam` con $\alpha = 0{,}001$, $\beta_1 = 0{,}9$ y $\beta_2 = 0{,}999$.

4. **Entrenamiento:**

   - Inicializar los pesos aleatoriamente.
   - Dividir el conjunto de datos en *mini-batches*.
   - Para cada *mini-batch*, calcular la salida (*forward pass*), la función de costo y propagar gradientes (*backward pass*).
   - Actualizar los parámetros siguiendo las fórmulas de `Adam`.

5. **Validación:**

   - Monitorear la precisión (*accuracy*) o la métrica deseada (AUC, F1, etc.) en el conjunto de validación al terminar cada época.
   - Ajustar hiperparámetros (tasa de aprendizaje, número de neuronas) si es necesario.

Código de Implementación (Ejemplo en Python)

A continuación se muestra un ejemplo simplificado de implementación de una red neuronal de dos capas en `PyTorch` usando `Adam` como optimizador:

```
import torch
import torch.nn as nn
import torch.optim as optim

# Definición de la red neuronal
class TwoLayerNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(TwoLayerNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        out = self.relu(self.fc1(x))
        out = self.relu(self.fc2(out))
        out = self.sigmoid(self.fc3(out))
        return out

# Parámetros principales
input_size = 20   # Ejemplo: 20 características
hidden_size = 64  # 64 neuronas en cada capa oculta
output_size = 1   # Clasificación binaria
```

```
# Creación de la red
model = TwoLayerNet(input_size, hidden_size, output_size)

# Definición de la función de costo y optimizador
criterion = nn.BCELoss()         # Función de costo de entropía cruzada para binario
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Ejemplo de ciclo de entrenamiento
for epoch in range(10):  # 10 épocas
    # Suponiendo que train_loader itera sobre (datos, etiquetas)
    for data, labels in train_loader:
        # 1) Forward pass
        outputs = model(data)
        loss = criterion(outputs, labels)

        # 2) Backward pass y actualización
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch [{epoch+1}/10], Loss: {loss.item():.4f}")

print("Entrenamiento finalizado.")
```

Ejercicios Propuestos

1. **Comparación de Optimizadores:** Utilizando un conjunto de datos de 10,000 ejemplos de ventas en línea, entrene la misma arquitectura de red neuronal profunda con `SGD`, `Adam` y `RMSProp`. Compare la velocidad de convergencia y el error final, analizando cómo el optimizador influye en la estabilidad del entrenamiento.

2. **Ajuste de Hiperparámetros:** Varíe la tasa de aprendizaje (*learning rate*) y el tamaño de *mini-batch* en un problema de clasificación de imágenes simples. Registre el comportamiento de la función de costo y la exactitud en validación. Determine qué combinación ofrece un mejor balance entre velocidad de entrenamiento y calidad de la solución final.

3. **Regularización:** Agregue *dropout* en las capas ocultas de la red anterior. Entrene el modelo y compare el rendimiento en entrenamiento y validación. Analice cómo interacciona la regularización con el método de optimización seleccionado.

4. **Manejo de Datos Desbalanceados:** Considere un escenario donde sólo el 5 % de las muestras pertenecen a la clase positiva (fraude, por ejemplo). Diseñe un experimento para aplicar técnicas de muestreo o modificación de la función de costo. Evalúe el impacto de estas estrategias en el F1-score, AUC y otras métricas relevantes.

Referencias

- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
- Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations (ICLR)*.
- Ruder, S. (2017). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.