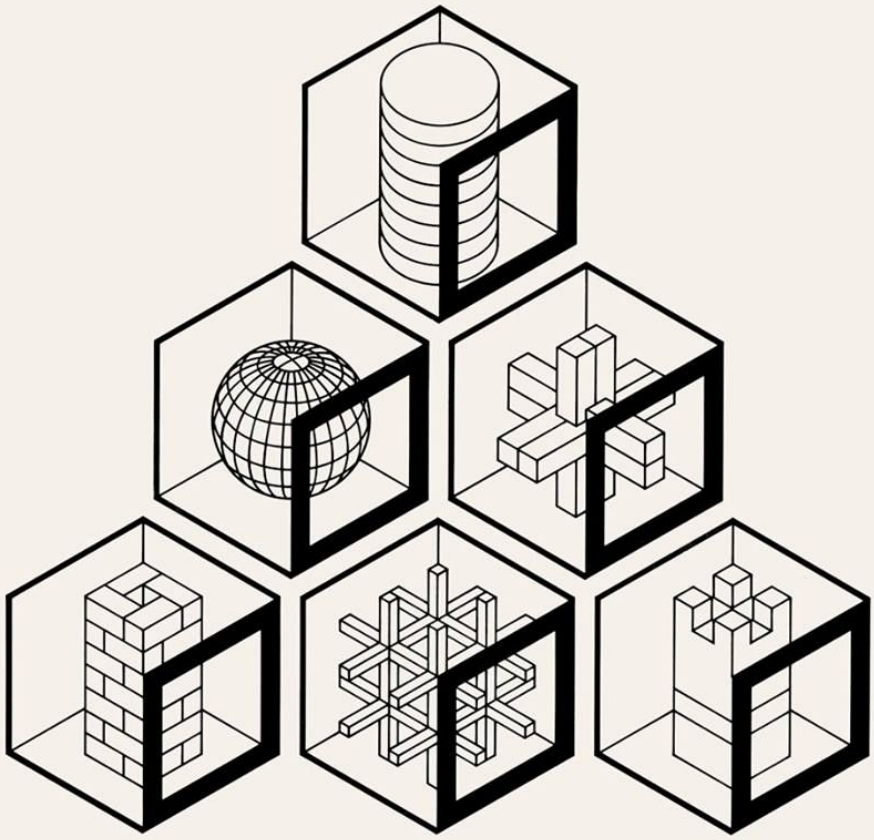
Algoritmos de optimización para redes neuronales



Por Antonio Richaud

Introducción

En el campo del aprendizaje automático, los algoritmos de optimización son herramientas super poderosas que permiten a los modelos aprender de los datos y mejorar su rendimiento con el tiempo. Desde los primeros días de la inteligencia artificial, cuando se intentaba replicar procesos cognitivos humanos, la optimización ha sido un desafío bárbaro. Los investigadores y profesionales del sector han dedicado sudor y lagrimas a perfeccionar métodos que permitan a las máquinas aprender de manera más eficiente y precisa, dando lugar a una variedad de técnicas que han revolucionado la forma en que interactuamos con la tecnología hoy en día.

La optimización, en su esencia, busca minimizar (o maximizar) una función de costo, un proceso fundamental en tareas como la clasificación, la regresión y el reconocimiento de patrones. En términos más simples, es el método mediante el cual un modelo ajusta sus parámetros internos para hacer predicciones mucho más precisas y reducir los errores. Este ajuste se realiza mediante el aprendizaje iterativo, en el cual el modelo evalúa continuamente su rendimiento y modifica sus parámetros para acercarse más a una solución óptima.

La importancia de los algoritmos de optimización en el aprendizaje automático no puede ni debe ser subestimada. Imagínate un modelo de aprendizaje profundo tratando de reconocer objetos en imágenes. Si los parámetros del modelo, como los pesos de una red neuronal, no se optimizan correctamente, el modelo podría ser incapaz de identificar patrones clave, lo que resultaría en predicciones incorrectas o muy poco precisas.

A lo largo de los años, se han desarrollado diversos algoritmos de optimización, cada uno con sus propias ventajas, desventajas y casos de uso específicos. Desde el clásico descenso de gradiente hasta métodos más avanzados como el optimizador Adam, cada técnica ofrece una forma diferente de abordar los desafíos de la optimización. Estos algoritmos no solo difieren en su enfoque teórico, sino también en su implementación práctica y en cómo afectan el rendimiento de los modelos en diferentes escenarios.

En este pequeño documento, platicaremos en profundidad sobre algunos de los algoritmos de optimización más utilizados en el aprendizaje automático. Al final de esta exploración, espero de todo corazón que no solo habrás adquirido un conocimiento profundo sobre cómo funcionan estos algoritmos, sino que también estarás mejor armado para elegir la técnica de optimización más adecuada para tus propios modelos, mejorando así su rendimiento y precisión.

La optimización es (en términos más poéticos), un arte y una ciencia, y como tal, requiere tanto comprensión teórica como habilidad práctica. Te invito a acompañarme a través de los algoritmos de optimización, donde cada paso nos acercará un poquito más a la creación de modelos de aprendizaje automático más inteligentes y eficaces. :)

1. Descenso de gradiente estocástico (SGD)

Explicación

El Descenso de Gradiente Estocástico (SGD, por sus siglas en inglés) es uno de los métodos de optimización más basicos y ampliamente utilizados en el aprendizaje automático. A diferencia del descenso de gradiente tradicional, que calcula el gradiente de la función de costo usando todo el conjunto de datos, el SGD realiza una actualización de los parámetros para cada ejemplo de entrenamiento de manera individual. Este enfoque, aunque introduce un grado de ruido en el proceso de optimización, puede ser bastante eficiente, especialmente cuando se trabaja con grandes conjuntos de datos.

El funcionamiento básico del SGD es sencillito: en lugar de esperar a calcular el gradiente en todo el conjunto de datos (lo que puede ser costoso en términos de tiempo y recursos), se actualizan los parámetros del modelo después de evaluar cada punto de datos de manera individual. Esto permite que el modelo realice ajustes más rápidamente y que pueda aprender patrones de los datos de forma incremental. Sin embargo, esta rapidez también puede ser un arma de doble filo, ya que puede causar que el modelo oscile alrededor del mínimo óptimo y, en algunos casos, que nunca alcance una convergencia estable.

Fórmula matemática

La actualización de los parámetros en el SGD se realiza de la siguiente manera:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

Vamos a desmenuzar esta fórmula:

• θ representa los parámetros del modelo que se están optimizando.

• η es la tasa de aprendizaje, un hiperparámetro que controla el tamaño del paso que da el modelo en la dirección del gradiente.

• $\nabla_\theta J(\theta; x^{(i)}; y^{(i)})$ es el gradiente de la función de costo $J(\theta)$ con respecto a los parámetros \theta, evaluado en el i-ésimo ejemplo de entrenamiento $(x^{(i)}, y^{(i)})$.

Este proceso se repite para cada ejemplo en el conjunto de datos, lo que permite que los parámetros se actualicen continuamente durante todo el entrenamiento.

Aplicación práctica

El Descenso de Gradiente Estocástico es sencillo de implementar y está soportado por la mayoría de los frameworks de aprendizaje automático. Vamos a checar un ejemplo muy sencillo de cómo se podría implementar SGD en un modelo de regresión lineal utilizando TensorFlow y PyTorch.

Implementación en TensorFlow

```
import tensorflow as tf

# Crear un modelo simple de regresión lineal
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=[1])
])

# Compilar el modelo con el optimizador SGD
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
              loss='mean_squared_error')

# Datos de ejemplo
X_train = [1, 2, 3, 4, 5]
y_train = [1, 2, 3, 4, 5]

# Entrenar el modelo
model.fit(X_train, y_train, epochs=100)
```

Implementación en PyTorch

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

# Modelo simple de regresión lineal
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super(LinearRegressionModel, self).__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)

# Instanciar el modelo, definir la función de pérdida y el optimizador
model = LinearRegressionModel()
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Datos de ejemplo
X_train = torch.tensor([[1.0], [2.0], [3.0], [4.0], [5.0]])
y_train = torch.tensor([[1.0], [2.0], [3.0], [4.0], [5.0]])

# Entrenamiento
for epoch in range(100):
    # Forward pass
    outputs = model(X_train)
    loss = criterion(outputs, y_train)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 10 == 0:
        print(f'Epoch [{epoch+1}/100], Loss: {loss.item():.4f}')
```

Ventajas y desventajas de este algoritmo

Ventajas:

- **Rapidez**: Debido a que SGD actualiza los parámetros para cada ejemplo de entrenamiento, puede converger más rápidamente que el descenso de gradiente tradicional, especialmente en grandes conjuntos de datos.
- **Menor costo computacional**: Al procesar un solo ejemplo de datos en lugar de todo el conjunto, se reduce significativamente la memoria y el tiempo de cálculo necesario en cada iteración.

Desventajas:

- **Oscilaciones y ruido**: Debido a que cada actualización se basa en un solo ejemplo de entrenamiento, el SGD puede introducir ruido en el proceso de optimización, causando que el modelo oscile alrededor del mínimo óptimo.
- **Convergencia no estable**: En algunos casos, especialmente cuando la tasa de aprendizaje no está bien ajustada, el modelo puede no converger de manera efectiva y puede quedarse atrapado en mínimos locales.

El SGD es una herramienta poderosa en el inventario de cualquier científico de datos, pero debe usarse con mucho ojito y, a menudo, se combina con otras técnicas como el momentum o la disminución de la tasa de aprendizaje para mitigar sus errores, aguas ahí.

2. Descenso de gradiente por lotes (Batch gradient descent)

Explicación

El Descenso de Gradiente por Lotes, conocido en inglés como *Batch Gradient Descent*, es el enfoque más tradicional y básico dentro de los métodos de optimización. En este método, se calcula el gradiente de la función de costo utilizando todo el conjunto de datos de entrenamiento antes de actualizar los parámetros del modelo. A diferencia del Descenso de Gradiente Estocástico (SGD), que realiza una actualización para cada ejemplo de entrenamiento, el Batch Gradient Descent actualiza los parámetros solo una vez por cada ciclo a través del conjunto completo de datos, conocido como *época*.

Este enfoque puede ser particularmente efectivo cuando se tiene un conjunto de datos relativamente pequeñito y que cabe en memoria, ya que la actualización basada en todos los datos proporciona un gradiente “más suave” y menos ruidoso. Sin embargo, debido a que se debe esperar a que todos los datos sean procesados antes de realizar una actualización, este método puede ser lento e ineficiente para conjuntos de datos grandes, tanto en términos de tiempo como de recursos computacionales.

Fórmula matemática

La actualización de los parámetros en el Batch Gradient Descent se realiza mediante la siguiente fórmula:

$$\theta = \theta - \eta \nabla_\theta J(\theta)$$

Fórmula desmenuzada:

• θ son los parámetros del modelo.

• η es la tasa de aprendizaje, que determina el tamaño del paso que el modelo dará en la dirección del gradiente.

• $\nabla_\theta J(\theta)$ es el gradiente de la función de costo $J(\theta)$ con respecto a los parámetros θ, calculado usando todo el conjunto de datos.

El gradiente se calcula sumando las derivadas parciales de la función de costo con respecto a cada parámetro, considerando todos los ejemplos del conjunto de datos. Esto da como resultado un gradiente mucho más preciso, pero también significa que la actualización de los parámetros ocurre con menos frecuencia.

Aplicación práctica

Implementar el Batch Gradient Descent es bastante directo y como para el SGD es bien soportado por los frameworks de aprendizaje automático como TensorFlow y PyTorch (Bendito Dios). A continuación, vamos a ver cómo se puede implementar el Batch Gradient Descent en un modelo de regresión lineal sencillito.

Implementación en TensorFlow

```python
import tensorflow as tf

# Crear un modelo simple de regresión lineal
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=[1])
])

# Compilar el modelo con el optimizador SGD (modo por lotes se activa automáticamente)
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
              loss='mean_squared_error')

# Datos de ejemplo
X_train = [1, 2, 3, 4, 5]
y_train = [1, 2, 3, 4, 5]

# Entrenar el modelo usando todo el conjunto de datos como un lote

# Aquí batch_size indica el tamaño del lote
model.fit(X_train, y_train, epochs=100, batch_size=len(X_train))
```

Implementación en PyTorch

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

# Modelo simple de regresión lineal
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super(LinearRegressionModel, self).__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)

# Instanciar el modelo, definir la función de pérdida y el optimizador
model = LinearRegressionModel()
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Datos de ejemplo
X_train = torch.tensor([[1.0], [2.0], [3.0], [4.0], [5.0]])
y_train = torch.tensor([[1.0], [2.0], [3.0], [4.0], [5.0]])

# Entrenamiento usando todo el conjunto de datos como un lote
for epoch in range(100):
    # Forward pass
    outputs = model(X_train)
    loss = criterion(outputs, y_train)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 10 == 0:
        print(f'Epoch [{epoch+1}/100], Loss: {loss.item():.4f}')
```

Ventajas y desventajas

Ventajas:

- **Estabilidad**: Debido a que se calcula el gradiente utilizando todo el conjunto de datos, la actualización de los parámetros es más estable y menos propensa a oscilaciones ruidosas, lo que facilita la convergencia hacia el mínimo global.
- **Precisión del gradiente**: Al utilizar todos los datos disponibles, el gradiente calculado es más preciso, lo que puede resultar en un proceso de optimización más robusto.

Desventajas:

- **Ineficiencia computacional**: Procesar todo el conjunto de datos para cada actualización puede ser costoso en términos de tiempo y recursos computacionales, especialmente para conjuntos de datos grandes.
- **Convergencia lenta**: Dado que las actualizaciones se realizan solo después de evaluar todo el conjunto de datos, el proceso de optimización puede ser más lento en comparación con el SGD.
- **No escalable para grandes conjuntos de datos**: En escenarios con grandes volúmenes de datos, el Batch Gradient Descent puede ser impráctico debido a las limitaciones de memoria y tiempo.

El Descenso de Gradiente por Lotes es buena opción para conjuntos de datos pequeñitos donde la estabilidad y la precisión son importantes pero su uso para aplicaciones que manejan grandes cantidades de datos va a estar limitado debido a los problemas de escalabilidad y eficiencia. :c

3. Descenso de gradiente con Mini-Batch

Explicación

El Descenso de Gradiente con Mini-Batch es una técnica de optimización que combina las ventajas del Descenso de Gradiente Estocástico (SGD) y el Descenso de Gradiente por Lotes (Batch Gradient Descent). En lugar de actualizar los parámetros del modelo después de cada ejemplo individual (como en SGD) o después de todo el conjunto de datos (como en el Batch Gradient Descent), el Mini-Batch Gradient Descent realiza actualizaciones después de procesar un pequeño lote de ejemplos de entrenamiento.

Este enfoque le permite al modelo beneficiarse de la velocidad y la eficiencia del SGD, al mismo tiempo que reduce las oscilaciones erráticas al suavizar las actualizaciones con el promedio de un pequeño conjunto de ejemplos. Además, el uso de mini-batches hace que el entrenamiento sea mucho más eficiente en términos de tiempo computacional, ya que permite un uso optimizado del hardware, especialmente en arquitecturas paralelas como GPUs.

Fórmula matemática

La actualización de los parámetros en el Mini-Batch Gradient Descent se realiza con la siguiente fórmula:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

Fórmula desmenuzada:

• θ son los parámetros del modelo.

• η es la tasa de aprendizaje.

• $\nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$ es el gradiente de la función de costo calculado en un mini-batch de ejemplos de entrenamiento, desde el i-ésimo hasta el (i+n)-ésimo ejemplo.

Este gradiente es un promedio de los gradientes de los ejemplos dentro del mini-batch, lo que ayuda a suavizar las actualizaciones y a reducir el ruido.

Aplicación práctica

El Mini-Batch Gradient Descent es ampliamente utilizado en el entrenamiento de redes neuronales, particularmente en el entrenamiento de redes neuronales profundas, como las redes neuronales convolucionales (CNNs). Vamos a hacer un ejemplo sencillo de cómo aplicar mini-batches en el entrenamiento de una red neuronal convolucional usando TensorFlow.

Implementación en TensorFlow

```python
import tensorflow as tf
from tensorflow.keras import datasets, layers, models

# Cargar el conjunto de datos CIFAR-10
(X_train, y_train), (X_test, y_test) = datasets.cifar10.load_data()

# Normalizar los datos
X_train, X_test = X_train / 255.0, X_test / 255.0

# Definir la arquitectura de la red neuronal convolucional
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10)
])

# Compilar el modelo con el optimizador SGD y usar mini-batches
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

# Entrenar el modelo utilizando mini-batches
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))
```

Implementación en PyTorch

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

# Transformaciones para los datos de entrada
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# Descargar y cargar el conjunto de datos CIFAR-10
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=32,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=32,
                                         shuffle=False, num_workers=2)

# Definir la red neuronal convolucional
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.fc1 = nn.Linear(64*6*6, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Instanciar el modelo, definir la función de pérdida y el optimizador
net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01)

# Entrenamiento del modelo usando mini-batches
for epoch in range(10):  # 10 epochs
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # Obtener los inputs y labels del mini-batch
        inputs, labels = data

        # Reiniciar los gradientes
        optimizer.zero_grad()

        # Forward pass
        outputs = net(inputs)
        loss = criterion(outputs, labels)

        # Backward pass y optimización
        loss.backward()
        optimizer.step()

        # Imprimir estadísticas cada 100 mini-batches
        running_loss += loss.item()
        if i % 100 == 99:
            print(f'Epoch {epoch + 1}, Batch {i + 1}, Loss: {running_loss / 100:.4f}')
            running_loss = 0.0

print('Entrenamiento finalizado')
```

Ventajas y desventajas

Ventajas:

- **Equilibrio entre precisión y velocidad**: El Mini-Batch Gradient Descent ofrece un buen compromiso entre la precisión del Batch Gradient Descent y la rapidez del SGD, ya que proporciona actualizaciones más suaves y frecuentes.
- **Eficiencia computacional**: Utilizar mini-batches permite un uso más eficiente de la memoria y los recursos computacionales, especialmente en GPUs, lo que puede acelerar significativamente el proceso de entrenamiento.
- **Mejor generalización**: El ruido introducido por los mini-batches puede ayudar al modelo a salir de mínimos locales y mejorar su capacidad de generalización.

Desventajas:

- **Elección del tamaño del Mini-Batch**: Seleccionar el tamaño del mini-batch adecuado puede ser desafiante. Un tamaño demasiado pequeño puede llevar a un entrenamiento inestable, mientras que un tamaño demasiado grande puede hacer que el modelo se comporte como el Batch Gradient Descent, perdiendo las ventajas de la rapidez y la eficiencia.
- **Oscilaciones residuales**: Aunque el uso de mini-batches reduce el ruido en comparación con el SGD, aún pueden ocurrir oscilaciones alrededor de los mínimos, especialmente si la tasa de aprendizaje es alta.

4. Descenso de gradiente con momento

Explicación

El Descenso de Gradiente con Momento es una técnica de optimización que considera los gradientes pasados para suavizar la actualización de los parámetros del modelo. Este enfoque no solo toma en cuenta el gradiente actual, sino que también acumula un promedio ponderado exponencialmente de los gradientes anteriores. Esto ayuda a que las actualizaciones de los parámetros sean más suaves y a acelerar la convergencia hacia un óptimo.

El momentum funciona como una especie de “memoria” de las actualizaciones anteriores, lo que permite al modelo mantener la inercia en la dirección del gradiente, evitando las oscilaciones que pueden ocurrir con el descenso de gradiente estándar. Esto es especialmente útil en superficies de costo con mesetas o valles, donde el gradiente es pequeño y el avance puede ser lento.

Fórmula matemática

La actualización de los parámetros con Momentum se realiza en dos sencillos pasos:

1. Actualización de la velocidad:

$$v_t = \beta v_{t-1} + (1 - \beta) \nabla_\theta J(\theta)$$

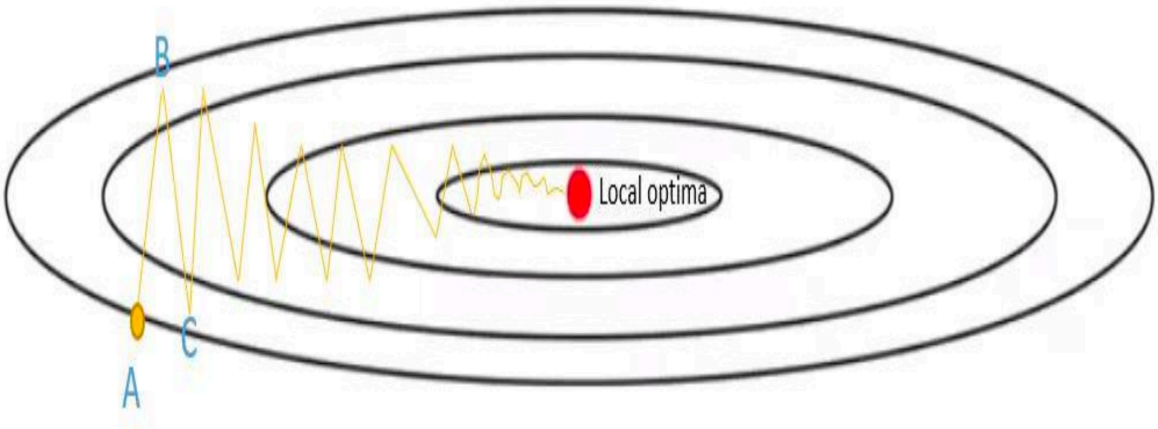
2. Actualización de los parámetros:

$$\theta = \theta - \eta v_t$$

Explicado por partes:

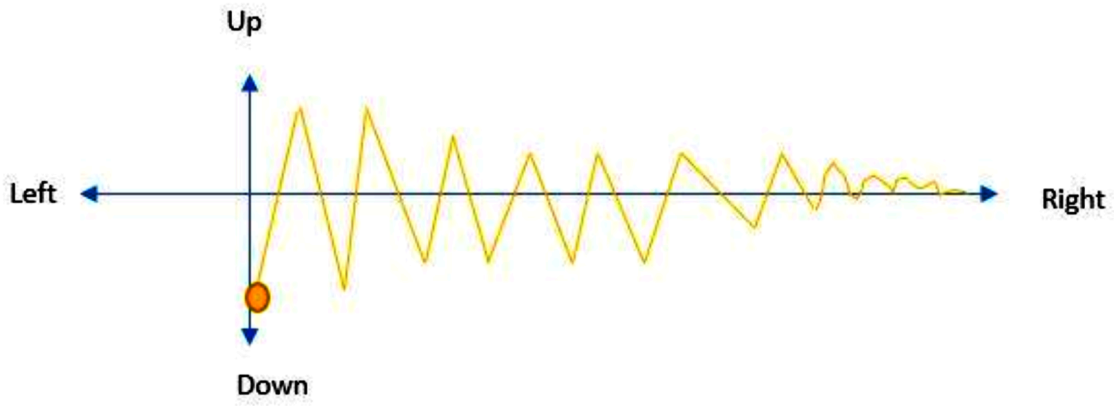
• v_t es la “velocidad” o acumulación del gradiente.
• β es el coeficiente de momentum, que determina cuánto del gradiente anterior se conserva.
• $\nabla_\theta J(\theta)$ es el gradiente actual de la función de costo.
• η es la tasa de aprendizaje.
• θ son los parámetros del modelo.

Visualización



Imaginemos que estamos comenzando el proceso de descenso de gradiente desde un **punto inicial A**. Después de una iteración, el modelo puede moverse a un nuevo punto, **el punto B**. Luego de otra iteración, el modelo podría terminar en **el punto C**. Con cada iteración de descenso de gradiente, el modelo se acerca cada vez más al **óptimo local**, el cual se refiere a un punto en el espacio de parámetros donde la función de costo alcanza un valor mínimo (o máximo, en el caso de maximización) en comparación con los puntos cercanos. Es importante destacar que, a diferencia de un óptimo global, un óptimo local no es necesariamente el punto donde la función de costo tiene su valor mínimo absoluto en todo el espacio de parámetros, sino solo dentro de un vecindario específico alrededor de ese punto.

Aquí te dejo otra grafica la cual representa el comportamiento del descenso de gradiente. Imagina que estás tratando de encontrar el camino más directo hacia un punto óptimo (hacia la derecha en la gráfica), pero en cada paso, en lugar de avanzar suavemente, te encuentras subiendo y bajando continuamente.



- **Movimiento hacia la derecha (Right):** Este es el camino que queremos seguir para acercarnos al mínimo de la función de costo, es decir, al punto donde nuestro modelo tiene el menor error posible.

- **Oscilaciones hacia arriba y abajo (Up y Down):** A medida que el modelo avanza hacia la derecha, también experimenta subidas y bajadas. Estas subidas y bajadas representan cómo los pesos del modelo se ajustan de manera descontrolada, haciendo que el modelo “rebote” mientras trata de encontrar el mejor camino.

Impacto del tasa de aprendizaje

Si se usa una tasa de aprendizaje alta, la oscilación vertical (es decir, las actualizaciones de los pesos) tendrá mayor magnitud. Esto puede llevar a un sobreajuste si no se ajusta correctamente la tasa de aprendizaje junto con el coeficiente de momentum.

Aplicación práctica

El uso del momentum es importante en la optimización de modelos de clasificación complejos, como las redes neuronales. Te dejo un ejemplo de cómo se implementa en un modelo de clasificación usando TensorFlow:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models

# Cargar el conjunto de datos MNIST
(X_train, y_train), (X_test, y_test) = datasets.mnist.load_data()

# Normalizar los datos
X_train, X_test = X_train / 255.0, X_test / 255.0

# Definir la arquitectura de la red neuronal convolucional
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Compilar el modelo usando SGD con momentum
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Entrenar el modelo
model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))
```

Y un ejemplo en PyTorch también:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

# Transformaciones para los datos de entrada
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5))])

# Descargar y cargar el conjunto de datos MNIST
trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                      download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.MNIST(root='./data', train=False,
                                     download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64,
                                         shuffle=False, num_workers=2)

# Definir la red neuronal convolucional
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Instanciar el modelo, definir la función de pérdida y el optimizador
net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)

# Entrenamiento del modelo usando momentum
for epoch in range(10):  # 10 epochs
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data

        optimizer.zero_grad()  # Reiniciar los gradientes

        outputs = net(inputs)  # Forward pass
        loss = criterion(outputs, labels)  # Calcular la pérdida
        loss.backward()  # Backward pass
        optimizer.step()  # Optimización con momentum

        # Imprimir estadísticas cada 100 mini-batches
        running_loss += loss.item()
        if i % 100 == 99:
            print(f'Epoch {epoch + 1}, Batch {i + 1}, Loss: {running_loss / 100:.4f}')
            running_loss = 0.0

print('Entrenamiento finalizado')
```

Ventajas y desventajas

Ventajas:

• **Superar mínimos locales:** El momentum ayuda a superar los mínimos locales al mantener la dirección de las actualizaciones, lo que le permite al modelo salir de áreas planas o valles de la función de costo.
• **Acelerar la convergencia:** El momentum acumula la velocidad de las actualizaciones, lo que resulta en una convergencia más rápida hacia el óptimo.
• **Reducción de oscilaciones:** El momentum reduce las oscilaciones erráticas que pueden ocurrir con el SGD, especialmente cuando la tasa de aprendizaje es alta.

Desventajas:

• **Ajuste del factor de momentum:** Seleccionar el valor adecuado para β es crucial. Si es demasiado alto, puede llevar a saltarse el mínimo; si es demasiado bajo, el efecto del momentum puede ser mínimo.
• **Mayor complejidad computacional:** Aunque el incremento de la complejidad es mínimo, requiere almacenamiento y cálculo adicional para la velocidad acumulada.

El Descenso de Gradiente con Momentum es una herramienta bastante necesaria que mejora la eficacia del entrenamiento de modelos, ayudando a superar obstáculos comunes en la optimización, como mínimos locales y oscilaciones. Es ampliamente utilizado en combinación con otros métodos de optimización en la práctica moderna del aprendizaje profundo.

5. Optimizador ADAM

Explicación

El ADAM (Adaptive Moment Estimation) es uno de los algoritmos de optimización más utilizados y efectivos en el aprendizaje profundo. Es particularmente apreciado por su capacidad para manejar grandes cantidades de datos y modelos con millones de parámetros, lo que lo convierte en una opción preferida para entrenar redes neuronales profundas.
ADAM es una combinación de dos métodos de optimización populares: el Descenso de Gradiente con Momentum y RMSprop. Combina lo mejor de ambos mundos, lo que permite una convergencia rápida y estable, incluso en escenarios con datos ruidosos o que cambian dinámicamente.

¿Cómo funciona? 🤔

El optimizador ADAM utiliza dos conceptos fundamentales:

1. Estimación del momento: Calcula un promedio ponderado exponencialmente de los gradientes pasados, lo que ayuda a suavizar las actualizaciones. Este proceso es similar al momentum y ayuda a evitar oscilaciones bruscas en el proceso de optimización.
2. Estimación del momento cuadrático: Calcula un promedio ponderado exponencialmente de los cuadrados de los gradientes pasados. Esto permite que ADAM ajuste dinámicamente la tasa de aprendizaje para cada parámetro, lo que mejora la estabilidad y la velocidad de convergencia.

Fórmulas:

1. Cálculo del promedio ponderado exponencial de los gradientes:

$$v_{dw} = \beta_1 v_{dw} + (1 - \beta_1) \nabla_\theta J(\theta)$$

$$v_{db} = \beta_1 v_{db} + (1 - \beta_1) \nabla_\theta J(\theta)$$

Explicación:

• $v_{dw} y v_{db}$ son los promedios ponderados exponencialmente de los gradientes para los pesos y el sesgo, respectivamente.

• β_1 es el coeficiente que determina el peso de los gradientes anteriores, con un valor típico de 0.9.

2. Cálculo del promedio ponderado exponencial de los cuadrados de los gradientes:

$$s_{dw} = \beta_2 s_{dw} + (1 - \beta_2)(\nabla_\theta J(\theta))^2$$

$$s_{db} = \beta_2 s_{db} + (1 - \beta_2)(\nabla_\theta J(\theta))^2$$

Explicación:

• $s_{dw} y s_{db}$ son los promedios ponderados exponencialmente de los cuadrados de los gradientes para los pesos y el sesgo, respectivamente.

• β_2 es el coeficiente que determina el peso de los gradientes cuadrados anteriores, con un valor típico de 0.999.

3. Corrección del sesgo:

Dado que los promedios ponderados exponencialmente están sesgados hacia cero, especialmente en las primeras iteraciones, ADAM aplica una corrección de sesgo:

$$\hat{v}dw = \frac{vdw}{1-\beta_1^t}$$

$$\hat{s}dw = \frac{sdw}{1-\beta_2^t}$$

4. Actualización de los parámetros:

Finalmente, los parámetros del modelo se actualizan de la siguiente manera:

$$\theta = \theta - \eta \frac{\hat{v}dw}{\sqrt{\hat{s}dw} + \epsilon}$$

Explicación:

• η es la tasa de aprendizaje.

• ϵ es un término pequeño para evitar la división por cero, con un valor típico de 10^{-8}.

Pasos para implementar ADAM

1. Inicializar $v_{dw}, v_{db}, s_{dw}, s_{db}$ a cero.
2. En cada iteración, calcular los gradientes y aplicar las fórmulas de actualización mencionadas arriba.
3. Aplicar la corrección del sesgo.
4. Actualizar los parámetros del modelo usando la fórmula final.

Implementación en código

Te dejo un pequeño ejemplo de código utilizando TensorFlow:

```python
import tensorflow as tf
from tensorflow.keras import datasets, layers, models

# Cargar y preparar el conjunto de datos MNIST
(X_train, y_train), (X_test, y_test) = datasets.mnist.load_data()

# Normalizar los datos
X_train, X_test = X_train / 255.0, X_test / 255.0

# Definir la arquitectura del modelo
model = models.Sequential([
    # Aplana las imágenes de 28x28 píxeles a un vector de 784 elementos
    layers.Flatten(input_shape=(28, 28)),
    # Capa completamente conectada con 128 neuronas y activación ReLU
    layers.Dense(128, activation='relu'),
    # Capa de salida con 10 neuronas para las 10 clases posibles
    layers.Dense(10, activation='softmax')
])

# Compilar el modelo usando el optimizador ADAM
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Entrenar el modelo
model.fit(X_train, y_train, epochs=5, validation_data=(X_test, y_test))

# Evaluar el modelo
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc:.4f}')
```

Y ¿como no? también un ejemplo en PyTorch para que no se pierda la bonita costumbre:

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

# Definir un modelo simple
model = nn.Sequential(
    nn.Linear(10, 5),
    nn.ReLU(),
    nn.Linear(5, 1)
)

# Definir la función de pérdida y el optimizador ADAM
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Ejemplo de entrenamiento
for epoch in range(100):
    # Datos de ejemplo
    inputs = torch.randn(10)
    target = torch.randn(1)

    # Forward pass
    output = model(inputs)
    loss = criterion(output, target)

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

    if epoch % 10 == 0:
        print(f'Epoch {epoch}, Loss: {loss.item()}')
```

Ventajas y desventajas del ADAM

Ventajas:

• **Rápida convergencia:** ADAM tiende a converger más rápidamente que otros optimizadores, especialmente en problemas complejos.
• **Adaptabilidad:** Ajusta dinámicamente la tasa de aprendizaje de cada parámetro, lo que mejora la estabilidad del entrenamiento.
• **Eficiencia computacional:** Es eficiente en términos de memoria y cálculo, lo que lo hace adecuado para grandes conjuntos de datos y modelos con muchos parámetros.

Desventajas:

• **Sensibilidad a hiperparámetros:** Aunque ADAM es robusto, su rendimiento puede ser sensible a la elección de \beta_1, \beta_2 y \eta.
• **Puede convergir a óptimos subóptimos:** En algunos casos, ADAM puede converger a soluciones subóptimas, especialmente si los hiperparámetros no están bien ajustados.

Para una comprensión más profunda del optimizador ADAM, te recomiendo muchísimo que le des una leída al paper original que te dejo por aquí: https://arxiv.org/pdf/1412.6980.pdf

6. RMSprop (Root Mean Square Propagation)

Explicación:

RMSprop es un optimizador que fue propuesto por Geoffrey Hinton en un curso online y es bastante utilizado en el entrenamiento de redes neuronales. El objetivo principal de RMSprop es mantener una tasa de aprendizaje adaptativa que ajuste de manera adecuada la magnitud de las actualizaciones de los pesos. Esto ayuda a evitar que los pasos sean demasiado grandes, lo que podría causar que el modelo se salte el mínimo, o demasiado pequeños, lo que ralentizaría el proceso de convergencia.

RMSprop se diseñó específicamente para resolver problemas donde el descenso de gradiente puede oscilar o converger lentamente debido a la variabilidad en la magnitud de los gradientes.

¿Cómo funciona? 🤔

Funciona manteniendo un promedio ponderado exponencialmente de los cuadrados de los gradientes pasados. Esto significa que en lugar de utilizar un valor constante para la tasa de aprendizaje, RMSprop ajusta dinámicamente la tasa de aprendizaje para cada parámetro del modelo. Al hacer esto, se asegura de que los pasos dados en la dirección del gradiente sean lo suficientemente grandes para que el modelo avance, pero no tan grandes como para causar inestabilidad.

Fórmulas:

1. Cálculo del promedio ponderado exponencial de los cuadrados de los gradientes:

$$s_{dw} = \beta s_{dw} + (1 - \beta)(\nabla_\theta J(\theta))^2$$

$$s_{db} = \beta s_{db} + (1 - \beta)(\nabla_\theta J(\theta))^2$$

Explicación:

• $s_{dw} y s_{db}$ son los promedios ponderados exponencialmente de los cuadrados de los gradientes para los pesos y el sesgo, respectivamente.

• β es el coeficiente que controla cuánto peso se da a los gradientes anteriores versus los nuevos, con un valor típico de 0.9.

2. Actualización de los parámetros:

$$\theta = \theta - \frac{\eta}{\sqrt{s_{dw} + \epsilon}} \nabla_\theta J(\theta)$$

$$b = b - \frac{\eta}{\sqrt{s_{db} + \epsilon}} \nabla_\theta J(\theta)$$

Donde:

• η es la tasa de aprendizaje.

• ϵ es un valor muy pequeño (por ejemplo, 10^{-8}) que se añade para evitar la división por cero.

Cómo Implementar RMSprop

Pasos para Implementar RMSprop:

1. Inicialización:

• Inicializar $s_{dw} y s_{db}$ a cero.

2. Cálculo de gradientes:

• Durante la retropropagación, calcular los gradientes para w y b.

3. Cálculo del promedio ponderado:

• Actualizar $s_{dw} y s_{db}$ utilizando las fórmulas mencionadas anteriormente.

4. Actualización de parámetros:

• Actualizar los parámetros w y b utilizando los valores ajustados de la tasa de aprendizaje.

Implementación en TensorFlow

Aquí tienes un ejemplo de cómo implementar RMSprop en TensorFlow:

```python
import tensorflow as tf
from tensorflow.keras import datasets, layers, models

# Cargar y preparar el conjunto de datos MNIST
(X_train, y_train), (X_test, y_test) = datasets.mnist.load_data()

# Normalizar los datos
X_train, X_test = X_train / 255.0, X_test / 255.0

# Definir la arquitectura del modelo
model = models.Sequential([
    layers.Flatten(input_shape=(28, 28)),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Compilar el modelo usando el optimizador RMSprop
model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Entrenar el modelo
model.fit(X_train, y_train, epochs=5, validation_data=(X_test, y_test))

# Evaluar el modelo
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc:.4f}')
```

Implementación de RMSprop en PyTorch

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

# Transformaciones para los datos de entrada
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Descargar y cargar el conjunto de datos MNIST
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

testset = torchvision.datasets.MNIST(root='./data', train=False, download=True,
transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)

# Definir la red neuronal
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 128)  # Capa completamente conectada
        self.fc2 = nn.Linear(128, 10)  # Capa de salida

    def forward(self, x):
        x = x.view(-1, 28 * 28)  # Aplanar la imagen de 28x28
        x = torch.relu(self.fc1(x))  # Activación ReLU
        x = self.fc2(x)  # Capa de salida con 10 neuronas
        return x

# Instanciar el modelo
net = Net()

# Definir la función de pérdida y el optimizador RMSprop
criterion = nn.CrossEntropyLoss()
optimizer = optim.RMSprop(net.parameters(), lr=0.001, alpha=0.9)

# Entrenamiento del modelo
for epoch in range(5):  # Entrenar por 5 épocas
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data

        # Inicializar los gradientes
        optimizer.zero_grad()

        # Forward pass
        outputs = net(inputs)
        loss = criterion(outputs, labels)

        # Backward pass y optimización
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 100 == 99:  # Imprimir cada 100 mini-batches
            print(f'Epoch {epoch + 1}, Batch {i + 1}, Loss: {running_loss / 100:.4f}')
            running_loss = 0.0

print('Entrenamiento finalizado')

# Evaluación del modelo
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy en el conjunto de prueba: {100 * correct / total:.2f}%')
```

Ventajas y desventajas de RMSprop

Ventajas:

• **Tasa de Aprendizaje Adaptativa:** RMSprop ajusta automáticamente la tasa de aprendizaje, lo que lo hace adecuado para problemas con gradientes que cambian rápidamente.
• **Convergencia Estable:** Es menos propenso a oscilaciones y puede converger más rápidamente que el descenso de gradiente estándar.

Desventajas:

• **Sensibilidad a β:** Aunque es menos sensible que otros métodos, la elección de β aún puede influir en el rendimiento del modelo.
• **Menos eficaz en escenarios de largo plazo:** En algunos casos, puede quedarse atrapado en mínimos locales, especialmente si se aplica durante muchas iteraciones.