**Bitácora de Desarrollo: Redes Neuronales para MNIST**

**Implementación Inicial y Primeros Desafíos**

Comencé implementando las estructuras básicas para las redes neuronales feedforward con el objetivo de clasificar dígitos manuscritos del dataset MNIST. Durante este proceso, me enfrenté a varios desafíos técnicos que fui resolviendo progresivamente.

**Estabilización de Softmax**

Uno de los primeros problemas que encontré fue la inestabilidad numérica en la implementación de Softmax al procesar lotes grandes de datos. Para solucionarlo, modifiqué la función para que manejara los datos en lotes cuando fuera necesario:

# Verificar el tamaño para decidir si usar batches

if Z.shape[0] > 1000: # Umbral para usar batches

return ActivationFunctions.batched\_softmax(Z)

else:

# Implementación estable para batches pequeños

Z\_shifted = Z - numpy.max(Z, axis=1, keepdims=True)

exp\_Z = numpy.exp(Z\_shifted)

return exp\_Z / numpy.sum(exp\_Z, axis=1, keepdims=True)

Esta implementación resuelve el problema de desbordamiento numérico al restar el valor máximo antes de aplicar la función exponencial, lo que mantiene los valores en un rango manejable.

**Experimentación con Arquitecturas**

Al intentar entrenar la red con diferentes configuraciones, me encontré con problemas para procesar modelos con muchas neuronas. Después de varias pruebas reduciendo el tamaño de las capas, logré una estructura que funcionaba correctamente:

Entradas: 784 (28x28 píxeles)

Capa oculta 1: 30 neuronas (Sigmoid)

Capa oculta 2: 30 neuronas (Sigmoid)

Capa de salida: 10 neuronas (Softmax)

Inicialización: Xavier

**Corrección de Funciones de Activación**

Durante las pruebas con el modelo, identifiqué que la función Sigmoid no estaba correctamente implementada en la función active. Al corregir este error, el rendimiento de la red mejoró notablemente.

También identifiqué problemas al usar ReLU como función de activación. La principal dificultad fueron las "neuronas muertas", que ocurren cuando una neurona con activación ReLU recibe consistentemente entradas negativas, causando que su gradiente se vuelva cero y dejándola permanentemente desactivada.

**Mejoras en la Depuración y Visualización**

Para entender mejor el comportamiento de la red durante el entrenamiento, implementé mejoras en el sistema de depuración:

* Añadí visualización detallada de la matriz de confusión
* Incorporé gráficos que muestran cómo el modelo va aprendiendo a lo largo de las épocas
* Implementé métricas adicionales para evaluar el rendimiento

**Optimización del FeedForward y Resolución de Problemas con Optimizadores**

Mejoré la implementación del método de propagación hacia adelante (feedforward) para hacerlo más eficiente y preciso:

def \_evaluate(self, X, threshold=None):

"""

Forward pass through the network

Parameters:

X : Input data matrix

threshold : Whether to apply threshold for binary classification

Returns:

numpy.ndarray: Network output after forward pass

"""

L = self.number\_of\_layers()

if L > 0:

a = X

for l in range(L):

z = (a @ self.m\_W[l]) + self.m\_B[l]

a = self.m\_A[l][1](z)

# end for

if threshold is not None and threshold:

# Only apply threshold for binary classification

if a.shape[1] == 1:

return (a >= 0.5).astype(float)

else:

return a # For multi-class, return raw probabilities

else:

return a

else:

return None

# end if

# end def

También perfeccioné la implementación del optimizador Adam para reducir problemas de memoria y mejorar la convergencia:

* Inicialización optimizada de los momentos mt y vt
* Implementación más eficiente para almacenar el último gradiente
* Actualización de momentos con mejor manejo de memoria
* Aplicación de actualizaciones en partes para reducir el uso de memoria
* Liberación explícita de memoria para evitar problemas en conjuntos de datos grandes

**Métodos de Inicialización Mejorados**

Una parte crucial para el buen funcionamiento de las redes neuronales es la inicialización de los pesos. Implementé varios métodos:

* **Xavier/Glorot**: Especialmente útil para funciones de activación Sigmoid y Tanh, ajusta la varianza de la inicialización según el número de conexiones entrantes y salientes.
* **He**: Diseñado específicamente para ReLU, inicializa los pesos con una varianza de 2/n, donde n es el número de conexiones entrantes.

Estos métodos de inicialización ayudaron significativamente a mejorar la convergencia de la red y evitar problemas como la saturación de neuronas o la explosión/desvanecimiento de gradientes.

**Éxito con el Modelo Inicial**

Gracias a las mejoras implementadas en el feedforward, el optimizador Adam y el descenso de gradiente, finalmente logré que el modelo inicial funcionara correctamente sin problemas:

Entradas: 784 (28x28 píxeles)

Capa oculta 1: 100 neuronas (ReLU)

Capa oculta 2: 50 neuronas (ReLU)

Capa de salida: 10 neuronas (Softmax)

Inicialización: Random

**Por qué ReLU Fallaba y Ahora Funciona**

Inicialmente, ReLU presentaba varios problemas que impedían el correcto funcionamiento de la red:

1. **Problema de las neuronas muertas**: Cuando una neurona con activación ReLU recibe valores negativos, su salida es cero y su gradiente también es cero, lo que significa que la neurona deja de aprender. En mi implementación original, muchas neuronas "morían" durante el entrenamiento y no se recuperaban.
2. **Inicialización inadecuada**: La inicialización aleatoria estándar que estaba usando no era adecuada para ReLU. Si los pesos se inicializan con valores muy pequeños, las entradas a la función ReLU tienden a ser negativas, lo que agrava el problema de las neuronas muertas.
3. **Tasas de aprendizaje inapropiadas**: Con tasas de aprendizaje muy altas, los pesos podían actualizarse de manera tan drástica que llevaban a más neuronas a la región negativa.

Las soluciones que implementé y que permitieron que ReLU funcionara correctamente fueron:

1. **Inicialización He**: Este método inicializa los pesos considerando la naturaleza de ReLU, evitando que muchas neuronas empiecen en estado "muerto".
2. **Optimización en el algoritmo Adam**: Mi implementación mejorada de Adam gestiona mejor los momentos y gradientes, lo que permite una actualización más estable de los pesos.
3. **Mejor gestión de memoria y cálculos numéricos**: Las optimizaciones en el método feedforward y en el manejo de la memoria permitieron trabajar con capas más grandes sin problemas de estabilidad numérica.

**Conclusiones y Aprendizajes**

A través de este proceso de implementación y depuración, aprendí:

1. La importancia de una implementación numéricamente estable de las funciones de activación
2. El impacto que tienen los métodos de inicialización en el rendimiento de la red
3. La necesidad de manejar eficientemente la memoria al trabajar con conjuntos de datos grandes
4. Las ventajas y desventajas de diferentes funciones de activación (Sigmoid vs ReLU)
5. La importancia del ajuste fino de hiperparámetros como el tamaño de las capas y la tasa de aprendizaje