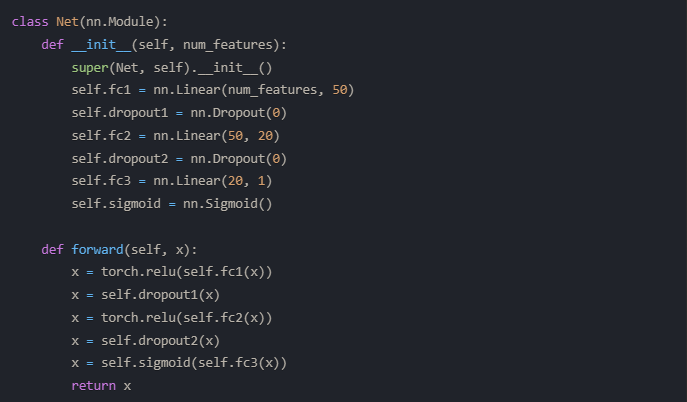
Explicando el codigo de ML



La clase `Net` hereda de la clase `nn.Module` de PyTorch, que es la clase base para todos los módulos de redes neuronales en PyTorch.

En el método `\_\_init\_\_`, se inicializan los componentes de la red neuronal:

1. `self.fc1 = nn.Linear(num\_features, 50)`: Esta línea crea una capa lineal (fully connected) llamada `fc1`. Toma `num\_features` como entradas (dimensión de la entrada) y produce 50 salidas (dimensión oculta).

2. `self.dropout1 = nn.Dropout(0)`: Esta línea crea una capa de dropout llamada `dropout1`. El dropout es una técnica de regularización que ayuda a prevenir el sobreajuste (overfitting) al desactivar aleatoriamente algunas neuronas durante el entrenamiento. En este caso, el valor `0` significa que no se desactiva ninguna neurona (0% de dropout).

3. `self.fc2 = nn.Linear(50, 20)`: Esta línea crea otra capa lineal llamada `fc2`. Toma las 50 salidas de la capa `fc1` como entrada y produce 20 salidas.

4. `self.dropout2 = nn.Dropout(0)`: Esta línea crea otra capa de dropout llamada `dropout2`, con 0% de dropout.

5. `self.fc3 = nn.Linear(20, 1)`: Esta línea crea la última capa lineal llamada `fc3`. Toma las 20 salidas de la capa `fc2` como entrada y produce una sola salida.

6. `self.sigmoid = nn.Sigmoid()`: Esta línea crea una capa de activación Sigmoid, que se aplicará a la salida de la capa `fc3`.

En el método `forward`, se define el flujo de datos a través de la red neuronal:

1. `x = torch.relu(self.fc1(x))`: La entrada `x` se pasa a través de la capa `fc1`, y se aplica la función de activación ReLU (Rectified Linear Unit) a la salida.

2. `x = self.dropout1(x)`: La salida de la capa anterior se pasa a través de la capa de dropout `dropout1`.

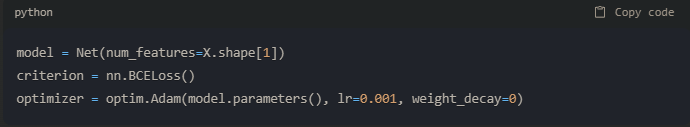
3. `x = torch.relu(self.fc2(x))`: La salida de la capa anterior se pasa a través de la capa `fc2`, y se aplica la función de activación ReLU a la salida.

4. `x = self.dropout2(x)`: La salida de la capa anterior se pasa a través de la capa de dropout `dropout2`.

5. `x = self.sigmoid(self.fc3(x))`: La salida de la capa anterior se pasa a través de la capa `fc3`, y se aplica la función de activación Sigmoid a la salida final.

6. `return x`: Se devuelve la salida final de la red neuronal.

En resumen, esta clase define una red neuronal feedforward con tres capas lineales, dos capas de dropout y una función de activación Sigmoid en la capa de salida. La arquitectura de la red es: `num\_features` -> 50 -> 20 -> 1, donde las capas ocultas tienen 50 y 20 neuronas, respectivamente. La capa de salida tiene una sola neurona, lo que sugiere que este modelo se utiliza para una tarea de clasificación binaria (por ejemplo, predicción de 0 o 1).



Claro, permíteme explicar esta parte del código:

```python

model = Net(num\_features=X.shape[1])

```

Esta línea crea una instancia del modelo `Net` definido anteriormente. El argumento `num\_features` se establece como `X.shape[1]`, que representa el número de columnas (características) en el tensor `X`. Esto se hace porque la capa de entrada de la red neuronal (`fc1`) debe tener el mismo número de entradas que las características de los datos.

```python

criterion = nn.BCELoss()

```

Esta línea define la función de pérdida (loss function) que se utilizará durante el entrenamiento del modelo. `nn.BCELoss` es la función de pérdida de Entropía Cruzada Binaria (Binary Cross Entropy Loss), que es comúnmente utilizada en problemas de clasificación binaria, donde la salida esperada es 0 o 1.

```python

optimizer = optim.Adam(model.parameters(), lr=0.001, weight\_decay=0)

```

Esta línea crea un optimizador llamado Adam, que se utilizará para actualizar los pesos de la red neuronal durante el entrenamiento. `model.parameters()` devuelve una lista de tensores que contienen los pesos y sesgos (biases) de la red neuronal.

Los argumentos utilizados son:

- `lr=0.001`: Este es el valor de la tasa de aprendizaje (learning rate). Determina cuán grandes o pequeños serán los pasos que el optimizador dará para ajustar los pesos durante el entrenamiento.

- `weight\_decay=0`: Este parámetro es una forma de regularización L2 que agrega una penalización proporcional al cuadrado de la norma de los pesos. Un valor de 0 significa que no se aplica esta regularización.

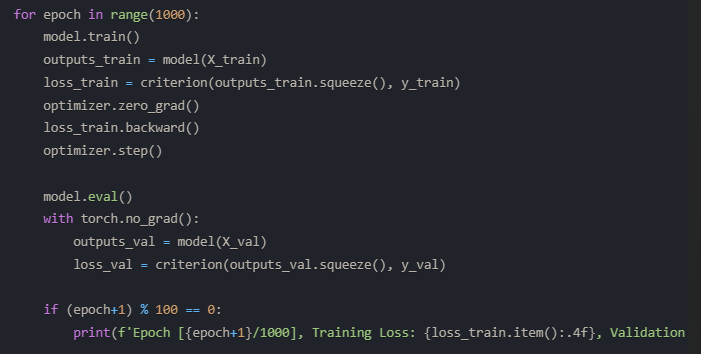
En resumen, estas líneas:

1. Crean una instancia del modelo `Net` con el número correcto de entradas (`num\_features`).

2. Definen la función de pérdida `BCELoss` para un problema de clasificación binaria.

3. Crean un optimizador `Adam` que se utilizará para actualizar los pesos de la red neuronal durante el entrenamiento, con una tasa de aprendizaje de 0.001 y sin regularización L2.

Estas líneas preparan los componentes necesarios para el entrenamiento del modelo de red neuronal.



Claro, permíteme explicar detalladamente este bloque de código:

```python

for epoch in range(1000):

```

Este es un bucle que se ejecutará 1000 veces, donde cada iteración representa una época de entrenamiento.

```python

model.train()

```

Esta línea pone el modelo en modo de entrenamiento. Esto es necesario para habilitar ciertas operaciones, como el cálculo del gradiente y la aplicación de capas como el dropout.

```python

outputs\_train = model(X\_train)

```

Esta línea pasa los datos de entrenamiento `X\_train` a través del modelo y obtiene las salidas predichas.

```python

loss\_train = criterion(outputs\_train.squeeze(), y\_train)

```

Esta línea calcula la pérdida (loss) de entrenamiento comparando las salidas predichas `outputs\_train` con las etiquetas reales `y\_train` utilizando la función de pérdida `criterion` definida anteriormente (`BCELoss`). El método `squeeze()` se utiliza para eliminar dimensiones adicionales innecesarias en el tensor `outputs\_train`.

```python

optimizer.zero\_grad()

```

Esta línea reinicia los gradientes acumulados a cero antes de realizar la retropropagación (backpropagation).

```python

loss\_train.backward()

```

Esta línea calcula los gradientes de la pérdida de entrenamiento con respecto a los pesos del modelo utilizando la retropropagación.

```python

optimizer.step()

```

Esta línea actualiza los pesos del modelo utilizando los gradientes calculados y el optimizador `Adam` definido anteriormente.

```python

model.eval()

```

Esta línea pone el modelo en modo de evaluación. Esto desactiva las capas como el dropout y la normalización por lotes.

```python

with torch.no\_grad():

outputs\_val = model(X\_val)

loss\_val = criterion(outputs\_val.squeeze(), y\_val)

```

Este bloque calcula las salidas predichas `outputs\_val` y la pérdida de validación `loss\_val` utilizando los datos de validación `X\_val` y `y\_val`. La cláusula `with torch.no\_grad()` evita que se calculen los gradientes durante la validación, lo cual es innecesario y ahorra memoria.

```python

if (epoch+1) % 100 == 0:

print(f'Epoch [{epoch+1}/1000], Training Loss: {loss\_train.item():.4f}, Validation Loss: {loss\_val.item():.4f}')

```

Esta línea imprime la pérdida de entrenamiento y validación cada 100 épocas. El método `item()` se utiliza para obtener el valor escalar de los tensores `loss\_train` y `loss\_val`.

En resumen, este código realiza lo siguiente en cada época:

1. Pone el modelo en modo de entrenamiento.

2. Calcula las salidas predichas y la pérdida de entrenamiento utilizando los datos de entrenamiento.

3. Reinicia los gradientes a cero.

4. Calcula los gradientes de la pérdida de entrenamiento mediante retropropagación.

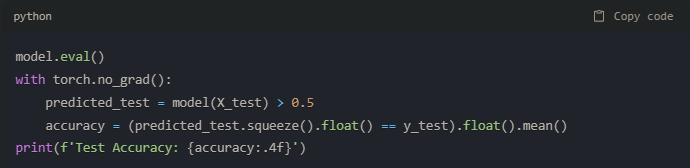
5. Actualiza los pesos del modelo utilizando el optimizador.

6. Pone el modelo en modo de evaluación.

7. Calcula las salidas predichas y la pérdida de validación utilizando los datos de validación.

8. Imprime la pérdida de entrenamiento y validación cada 100 épocas.

Este proceso se repite durante 1000 épocas, permitiendo que el modelo se entrene y ajuste sus pesos gradualmente para minimizar la pérdida en los conjuntos de entrenamiento y validación.



Claro, permíteme explicar este bloque de código:

```python

model.eval()

```

Esta línea pone el modelo en modo de evaluación. Esto es importante porque durante la evaluación, no queremos que las capas como el dropout y la normalización por lotes estén activas, ya que solo queremos obtener las predicciones del modelo con los pesos entrenados.

```python

with torch.no\_grad():

```

Esta línea crea un contexto en el cual no se realizarán operaciones que involucren el cálculo de gradientes. Esto es útil durante la evaluación, ya que no necesitamos calcular gradientes, lo que ahorra memoria y tiempo de cómputo.

```python

predicted\_test = model(X\_test) > 0.5

```

Esta línea pasa los datos de prueba `X\_test` a través del modelo y obtiene las salidas predichas. Como se trata de un problema de clasificación binaria, se comparan las salidas predichas con un umbral de 0.5. Si la salida es mayor que 0.5, se considera como una predicción positiva (1), de lo contrario, se considera una predicción negativa (0).

```python

accuracy = (predicted\_test.squeeze().float() == y\_test).float().mean()

```

Esta línea calcula la precisión (accuracy) del modelo comparando las predicciones `predicted\_test` con las etiquetas reales `y\_test`. Primero, se aplica `squeeze()` a `predicted\_test` para eliminar dimensiones adicionales innecesarias. Luego, se convierte `predicted\_test` a un tensor de tipo `float` para poder compararlo con `y\_test` (que también es un tensor de tipo `float`). La comparación `predicted\_test.float() == y\_test` devuelve un tensor de valores booleanos (True o False) que indica si cada predicción es correcta o no. Finalmente, se convierte este tensor de booleanos a tipo `float` y se calcula la media, lo que da la proporción de predicciones correctas, es decir, la precisión del modelo.

```python

print(f'Test Accuracy: {accuracy:.4f}')

```

Esta línea imprime la precisión del modelo en el conjunto de prueba, formateada con cuatro dígitos decimales.

En resumen, este bloque de código:

1. Pone el modelo en modo de evaluación.

2. Crea un contexto en el que no se realizarán operaciones que involucren el cálculo de gradientes.

3. Obtiene las predicciones del modelo en el conjunto de prueba `X\_test`, utilizando un umbral de 0.5 para convertirlas a predicciones binarias.

4. Calcula la precisión del modelo comparando las predicciones con las etiquetas reales `y\_test`.

5. Imprime la precisión del modelo en el conjunto de prueba.

Este proceso es importante para evaluar el rendimiento final del modelo después del entrenamiento y determinar qué tan bien generaliza el modelo a nuevos datos.