Documentación Proyecto piano-note-recognizer

Predicción de notas musicales de piano a partir del entrenamiento de una red neuronal alimentada con samples de una octava completa.

Para utilizar este sistema, escribir en la consola el siguiente comando: **‘ go run cmd/main.go ‘**. Aparecerá un link *http://localhost:8080* junto con la salida de la predicción de la nota musical debajo. A continuación se explican detalladamente los archivos principales de este sistema que se encargan de crear y configurar la red neuronal.

Archivo main.go

Organiza el flujo de ejecución del entrenamiento y la validación de la red. Esto incluye llamar a la función **prepareData()** para obtener los datos procesados, configurar la arquitectura de la red neuronal con Gorgonia, entrenar el modelo con los datos de entrenamiento, validar el modelo con los datos de validación, y finalmente evaluar y mostrar los resultados del desempeño del modelo.

prepareData()

La función `prepareData()` tiene los siguientes propósitos:

1. Cargar y Procesar Datos de Audio: La función comienza por cargar archivos de audio que contienen grabaciones de diferentes notas musicales de piano. Estos archivos son procesados para extraer características relevantes que puedan ser utilizadas para entrenar el modelo. Esto incluye la generación de espectrogramas, que transforman la señal de audio en una representación visual que muestra cómo varían las frecuencias del sonido con el tiempo.

2. Normalización de Datos: Tras la generación de espectrogramas, la función se encarga de normalizar estos datos. La normalización es un paso crucial en muchos algoritmos de aprendizaje automático, ya que asegura que las características tengan una escala común, eliminando así posibles sesgos debido a la diferencia en las escalas de los valores originales.

3. División en Conjuntos de Entrenamiento y Validación: La función `prepareData()` implementa una estrategia de división de datos, asignando aproximadamente el 80% de los datos al conjunto de entrenamiento y el 20% al conjunto de validación. Esta división es fundamental para entrenar el modelo en un conjunto de datos y luego validar su desempeño en un conjunto separado que el modelo no ha visto durante el entrenamiento, lo cual es importante para evaluar la capacidad de generalización del modelo.

4. Manejo de Errores y Datos Anómalos: Dentro de la función, se implementan controles para manejar errores, como problemas al cargar los archivos de audio o la detección de valores NaN (Not a Number) en los espectrogramas normalizados. Esto asegura la integridad y calidad de los datos que serán utilizados para el entrenamiento.

5. Preparación de Datos para el Modelo: Finalmente, la función estructura los datos de manera que sean aptos para su uso directo. En particular, los datos de espectrograma se convierten en matrices densas que son compatibles con la librería Gorgonia, que se utiliza para construir y entrenar la red neuronal en Go.

prepareData() es una función crítica que maneja varios pasos preliminares necesarios para asegurar que los datos estén listos y en el formato adecuado para entrenar efectivamente un modelo de red neuronal capaz de reconocer notas musicales a partir de archivos de audio. Esto incluye la carga de datos, su procesamiento, normalización y la adecuada preparación y división del conjunto de datos.

```
func prepareData() ([]*mat.Dense, []*mat.Dense, []string, []string) {
```

¿Qué es un slice?

En Go, un slice es una estructura de datos que representa una secuencia de elementos que pueden ser del mismo tipo. Funciona similar a un array, pero es más flexible porque su tamaño puede cambiar.

¿Para qué sirve un slice?

Los slices son usados ampliamente en Go para manejar y manipular colecciones de datos debido a su flexibilidad y eficiencia en términos de manejo de memoria y rendimiento.

¿Qué significa `[]*mat.Dense`?

`[]*mat.Dense` es un slice donde cada elemento es un puntero a un objeto de tipo `mat.Dense` de la biblioteca Gonum, que es usada para manipulaciones numéricas y representaciones de matrices. `mat.Dense` representa una matriz densa, que es un tipo de estructura de datos utilizada para almacenar datos en forma de tabla, típicamente números, de manera que puedan ser procesados de manera eficiente. Este tipo de estructura es muy común en ciencia de datos y machine learning.

¿Qué indica el asterisco (`*`)?

El asterisco en Go indica que la variable es un puntero al tipo de dato que sigue. En este caso, `*mat.Dense` significa un puntero a una instancia de `mat.Dense`. Esto es utilizado en Go para manipulación eficiente de la memoria, permitiendo que las funciones modifiquen el objeto original sin necesidad de copiarlo completamente, lo cual es crucial en procesamiento de grandes volúmenes de datos.

¿Por qué se coloca dos veces `[]*mat.Dense` y `[]string`?

En la función `prepareData`, se utilizan dos slices de `*mat.Dense` y dos de `string` para diferenciar entre los conjuntos de datos y etiquetas que serán usados en dos fases diferentes del proceso de machine learning:

1. `trainData` y `trainLabels`: Conjunto de datos y etiquetas para el entrenamiento. Se usa para ajustar el modelo.

2. `validData` y `validLabels`: Conjunto de datos y etiquetas para la validación. Se usa para evaluar el rendimiento del modelo y verificar cómo generaliza a datos no vistos anteriormente.

¿De qué sirven los cuatro slices?

Los cuatro slices devueltos por `prepareData` son esenciales para el entrenamiento y la evaluación de modelos en machine learning. Permiten separar los datos en conjuntos de entrenamiento y validación, lo cual es una práctica estándar para evaluar y afinar modelos. No están como parámetros de la función sino que se inicializan dentro de ella en la firma y luego son devueltos al final.

```
notes := []string{"C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"}
```

Se define e inicializa un slice de tipo **string** llamado **notes**. Este slice contiene los nombres de las notas musicales, que corresponden a los nombres de archivos o etiquetas para los datos que se van a procesar. Cada elemento del slice representa una nota musical diferente.

```
filesDir :="C:\\Users\\Pedro\\Documents\\Informática\\Proyectos_Paris\\TP BYMA\\Notas WAV"
```

Se declara una variable **filesDir** de tipo **string**. Esta variable almacena la ruta del directorio donde están ubicados los archivos de audio (en formato WAV, según se deduce de la ruta). Esta ruta se utiliza luego para construir las rutas completas a los archivos individuales de cada nota musical.

```
var trainData, validData []*mat.Dense
```

Se declaran dos variables, **trainData** y **validData**. Ambas son slices de punteros a **mat.Dense**. **mat.Dense** es una estructura de la biblioteca Gonum que representa una matriz densa. Estas matrices se usarán para almacenar los datos de entrenamiento y validación respectivamente, que normalmente incluyen características extraídas de los archivos de audio, como los espectrogramas.

```
var trainLabels, validLabels []string
```

Similar a la línea anterior, pero para las etiquetas. Se declaran dos slices de tipo **string**, **trainLabels** y **validLabels**. Estas variables se usarán para almacenar las etiquetas (en este caso, los nombres de las notas musicales) que corresponden a

cada dato en **trainData** y **validData**. Las etiquetas son cruciales para el entrenamiento supervisado en machine learning, ya que indican la clase o categoría de cada ejemplo de train o test.

```
for i, note := range notes {
```

Propósito: Iterar sobre cada nota musical.

i: Índice de la nota en el slice.

note: Nombre de la nota actual.

Dentro del Primer Bucle

Construcción del Nombre del Archivo

Se construye el nombre del archivo de audio correspondiente a la nota actual.

```
filename := filepath.Join(filesDir, note+".wav")
```

Propósito: Crear la ruta completa del archivo de audio de la nota actual.

Carga de Audio

Se carga el archivo de audio asegurándose de que contenga al menos 1024 muestras.

```
audioData, err := audio.LoadAudio(filename, 1024)
        if err != nil {
            fmt.Printf("Error loading audio from %s: %v\n", filename,
err)
            continue
        }
```

Propósito: Leer el archivo de audio y manejar cualquier error que ocurra durante la carga.

audioData: Datos del audio cargado.

err: Error que ocurrió durante la carga del audio.

Generación y Normalización del Espectrograma

Se genera y normaliza el espectrograma del audio cargado.

```
spectrogram := audio.GenerateSpectrogram(audioData, 1024)
normalizedSpectrogram := audio.NormalizeSpectrogram(spectrogram)
```

Propósito: Transformar la señal de audio en un espectrograma y normalizarlo.

spectrogram: Espectrograma generado a partir de los datos de audio.

normalizedSpectrogram: Espectrograma normalizado.

Verificación de NaN

Se verifica si hay valores NaN en el espectrograma normalizado.

```
        if audio.CheckNaN(normalizedSpectrogram) {
            fmt.Printf("NaN detected in normalized spectrogram for note: %s, skipping...\n", note)
            continue
        }
```

Propósito: Asegurarse de que el espectrograma no contenga valores NaN y manejar casos donde sí los contenga.

Segundo Bucle: Iterar sobre el Espectrograma Normalizado

El segundo bucle for itera sobre cada segmento (slice) del espectrograma normalizado.

```
for _, slice := range normalizedSpectrogram {
```

Propósito: Procesar cada segmento del espectrograma normalizado.

Verificación de la Longitud del Segmento

Se asegura que cada segmento tenga exactamente 1024 elementos.

```
if len(slice) != 1024 {
                fmt.Printf("Invalid slice length for note: %s, expected
1024, got %d\n", note, len(slice))
                continue
            }
```

Propósito: Verificar que cada segmento tenga la longitud correcta y manejar casos donde no la tenga.

Creación de una Matriz Densa

Se crea una matriz densa a partir del segmento, con dimensiones 1024 x 1.

```
matrix := mat.NewDense(1024, 1, slice)
```

Propósito: Convertir el segmento en una matriz densa adecuada para el modelo de red neuronal.

Asignación a Conjuntos de Train y Test

Se decide si el segmento pertenece al conjunto de train o test, basado en el índice de la nota.

```
            if i%5 == 0 {
                validData = append(validData, matrix)
                validLabels = append(validLabels, note)
            } else {
                trainData = append(trainData, matrix)
                trainLabels = append(trainLabels, note)
            }
```

Propósito: Dividir los datos en conjuntos de Train y Test.

i%5 == 0: Aproximadamente cada 5 datos se agrega uno al Test (0, 5, 10, etc.).

```
    return trainData, validData, trainLabels, validLabels
}
```

Devuelve los datos y etiquetas de entrenamiento y validación.

El encadenamiento de bucles en prepareData() es necesario para iterar sobre cada archivo de audio correspondiente a cada nota musical y luego procesar cada segmento del espectrograma normalizado de esos archivos. Este procesamiento incluye la verificación de errores, normalización y la creación de matrices densas que se asignan a conjuntos de train y test, lo cual es fundamental para preparar los datos que se usarán en el entrenamiento de la red neuronal.

¿Por qué es necesario normalizar el espectrograma de audio?

La normalización ajusta los valores del espectrograma para que todos tengan una escala común, generalmente entre 0 y 1. Esto es importante porque ayuda a que el modelo de red neuronal converja más rápido durante el entrenamiento y evita que ciertos sean dominantes solo por su alto valor numérico.

Si no se normaliza, los datos pueden tener escalas muy diferentes, lo que puede dificultar el aprendizaje del modelo. Algunos valores muy altos podrían dominar la función de pérdida, haciendo que el modelo se entrene de manera poco óptima. Esto podría resultar en un modelo que no generaliza bien y tiene un rendimiento pobre.

¿Por qué es necesario asegurarse de las 1024 muestras de audio?

Garantizar 1024 muestras de audio

Consistencia en el tamaño de entrada: Las redes neuronales requieren entradas de tamaño constante. Al asegurar que cada archivo de audio tenga al menos 1024 muestras, nos aseguramos de que cada entrada a la red tenga la misma dimensión, lo que es crucial para el procesamiento de los datos en la red.

Espectrograma adecuado: La ventana de 1024 muestras es un parámetro para la generación del espectrograma. Al tener una longitud fija, se asegura que la transformación y la normalización del espectrograma sean consistentes. Si los segmentos del espectrograma no tuvieran la misma longitud, no se podrían convertir en matrices densas de dimensiones fijas, que son necesarias para el entrenamiento del modelo.

¿Por qué cada quinto dato se asigna al conjunto de validación?

División de datos en Train y Test

Propósito: Dividir los datos en train y test permite evaluar el rendimiento del modelo en datos no vistos durante el entrenamiento. Esto ayuda a evitar el overfitting y proporciona una métrica para evaluar la capacidad de generalización del modelo.

La decisión ir asignando al test de a cinco datos es una manera simple de lograr una división aproximada de 80/20. En este caso, **i % 5 == 0** asegura que uno de cada cinco datos se usa para validación.

En lugar de agrupar datos de train y test por lotes, esta técnica distribuye los datos de manera que todas las notas y características están representadas de manera más uniforme en ambos conjuntos.

Función TestModel (main.go)

La función **TestModel** está creada para probar un modelo de red neuronal recurrente (RNN) utilizando un archivo de audio. La función carga el archivo de audio, genera un espectrograma del audio, normaliza el espectrograma y usa el modelo RNN para predecir las notas musicales en el archivo de audio. Finalmente, imprime las notas de la predicción en la consola.

```
func TestModel(rnn *neuralnet.RNN, filename string) {
```

Función y parámetros: Define la función TestModel que toma dos parámetros, un puntero a una RNN (rnn) y el nombre del archivo de audio (filename).

La función TestModel recibe un puntero a una estructura RNN del paquete neuralnet, que contiene el modelo de red neuronal recurrente que se va a utilizar para hacer predicciones. El uso de un puntero permite que la función acceda y modifique directamente el estado del modelo RNN.

```
audioData, err := audio.LoadAudio(filename, 1024)
```

Carga de audio: Se llama a la función LoadAudio del paquete audio para cargar el archivo de audio especificado. La función espera que el archivo tenga al menos 1024 muestras. audioData contiene los datos de audio cargados y err contiene cualquier error que ocurra durante la carga.

```
    if err != nil {
        fmt.Printf("Error loading audio from %s: %v\n", filename, err)
        return
    }
```

Manejo de errores: Si hay un error al cargar el audio, imprime un mensaje de error indicando el archivo problemático y detiene la ejecución de la función con return.

```
spectrogram := audio.GenerateSpectrogram(audioData, 1024)
normalizedSpectrogram := audio.NormalizeSpectrogram(spectrogram)
```

Estas líneas se repiten tanto en prepareData como en TestModel, esto asegura que los datos de entrenamiento y los datos de prueba estén en la misma escala. Esto es esencial para que el modelo pueda interpretar correctamente los datos de prueba basados en lo que aprendió durante el entrenamiento.

Generación de espectrograma: Convierte los datos de audio en un espectrograma usando la función GenerateSpectrogram con una ventana de 1024 muestras.

Normalización del espectrograma: Normaliza el espectrograma generado para asegurar que los valores estén en una escala uniforme.

```
if audio.CheckNaN(normalizedSpectrogram) {
        fmt.Println("NaN detected in normalized spectrogram, cannot
proceed with predictions.")
        return
    }
```

Verificación de NaN: Comprueba si hay valores NaN (Not a Number) en el espectrograma normalizado. Si se detectan NaNs, imprime un mensaje de error y detiene la ejecución de la función.

```
var predictions []string
```

Inicialización de predicciones: Declara una variable predictions como un slice de strings para almacenar las notas predichas.

```
for i, segment := range normalizedSpectrogram {
```

Iteración sobre el espectrograma: Itera sobre cada segmento del espectrograma normalizado. i es el índice del segmento y segment es el segmento en sí.

```
if len(segment) != 1024 {
            fmt.Printf("Skipping segment %d due to incorrect length:
%d\n", i, len(segment))
            continue
        }
```

Verificación de longitud de segmento: Comprueba si la longitud del segmento es diferente de 1024. Si es así, imprime un mensaje de error y omite este segmento con continue.

```
segmentMatrix := mat.NewDense(1024, 1, segment)
```

Creación de matriz densa: Convierte el segmento en una matriz densa (segmentMatrix) de dimensiones 1024 x 1.

mat.NewDense: Este método proviene del paquete gonum/mat, que es parte de la biblioteca Gonum, una biblioteca de álgebra lineal en Go.

Crea una nueva matriz densa (Dense) con las dimensiones especificadas y los datos proporcionados.

1024: Número de filas de la matriz.

1: Número de columnas de la matriz.

segment: Slice de float64 que contiene los datos del segmento del espectrograma.

Resultado: segmentMatrix es una matriz densa con dimensiones 1024 x 1, que contiene los datos del segmento del espectrograma.

```
segmentTensor := tensor.New(tensor.WithShape(1, 1024),
tensor.Of(tensor.Float64),
tensor.WithBacking(segmentMatrix.RawMatrix().Data))
```

Creación de tensor: Convierte la matriz densa en un tensor (segmentTensor) con forma 1 x 1024 y tipo de datos Float64.

tensor.WithShape(1, 1024): Especifica la forma del tensor. En este caso el tensor tiene una forma de 1 x 1024.

tensor.Of(tensor.Float64): Especifica que el tensor contiene datos de tipo float64.

tensor.WithBacking(segmentMatrix.RawMatrix().Data): Utiliza los datos de la matriz segmentMatrix como respaldo para el tensor. segmentMatrix.RawMatrix().Data devuelve un slice de float64 con los datos de la matriz.

La red neuronal espera recibir los datos en forma de tensor. Convertir el segmento del espectrograma primero en una matriz densa y luego en un tensor asegura que los datos estén en el formato correcto para ser procesados por la red neuronal.

Tanto la matriz como el tensor tienen dimensiones que corresponden a la forma de los datos del espectrograma. Una matriz de 1024 x 1 significa que cada segmento del espectrograma es una columna con 1024 valores. El tensor de forma 1 x 1024 refleja esta estructura, pero en el formato requerido por Gorgonia.

```
predictedNotes := rnn.Predict(segmentTensor)
```

Predicción de notas: Usa la red neuronal recurrente (rnn) para predecir las notas musicales del segmento de audio. predictedNotes contiene las notas predichas para este segmento.

```
if len(predictedNotes) == 0 {
            fmt.Printf("No predictions for segment %d\n", i)
        }
```

Verificación de predicciones: Si no se han predicho notas para el segmento, imprime un mensaje indicando esto.

```
predictions = append(predictions, predictedNotes...)
```

Almacenamiento de predicciones: Añade las notas predichas para el segmento al slice predictions.

append: Es una función incorporada en Go que se utiliza para agregar elementos a un slice.

predictedNotes... : Desempaqueta el slice predictedNotes y pasa sus elementos individuales a append.

El propósito de predictedNotes... es desempaquetar el slice predictedNotes y agregar cada uno de sus elementos al slice predictions. Esto es necesario porque append normalmente toma elementos individuales como argumentos adicionales, pero en este caso queremos agregar todos los elementos de predictedNotes a predictions en una sola operación.

```
    if len(predictions) == 0 {
        fmt.Println("No notes were predicted.")
    } else {
        fmt.Printf("Notes in file: %s\n", strings.Join(predictions, ",
"))
    }
```

Impresión de resultados: Si no se predijeron notas, imprime un mensaje indicando esto. Si se predijeron notas, imprime las notas predichas en la consola, unidas por comas.

La función TestModel:

- Carga un archivo de audio
- Genera y normaliza un espectrograma del audio
- Verifica que el espectrograma no contenga valores NaN
- Itera sobre los segmentos del espectrograma normalizado
- Convierte cada segmento a un tensor
- Usa una red neuronal recurrente para predecir las notas musicales
- Imprime las notas predichas en la consola

La función es fundamental para probar el modelo RNN con nuevos archivos de audio y verificar su capacidad para reconocer y predecir las notas musicales correctamente.

Función main() (main.go)

La función main es el punto de entrada del programa. Su propósito es orquestar el flujo principal de la aplicación, que en este caso implica:

- Preparar los datos de train y test.
- Configurar y crear una red neuronal recurrente (RNN).
- Entrenar la red neuronal con los datos de entrenamiento.
- Validar el modelo entrenado con los datos de validación.
- Probar el modelo entrenado con un archivo de audio específico.

```
trainData, validData, trainLabels, validLabels := prepareData()
```

Llamada a prepareData: Llama a la función prepareData para cargar, procesar y dividir los datos de audio en conjuntos de entrenamiento y validación.

Asignación de variables: Asigna los datos y etiquetas de entrenamiento (trainData, trainLabels) y validación (validData, validLabels).

```
    if len(trainData) == 0 || len(validData) == 0 {
        fmt.Println("Training or validation data is empty, cannot
proceed.")
        return
    }
```

Verificación de datos: Verifica que los conjuntos de datos de entrenamiento y validación no estén vacíos. Si alguno de ellos está vacío, imprime un mensaje de error y termina la ejecución del programa.

```
fmt.Println("Data and labels prepared for training and validation.")
```

Mensaje de estado: Imprime un mensaje indicando que los datos y las etiquetas están listos para el entrenamiento y la validación.

```
    config := neuralnet.NetworkConfig{
        InputSize:  1024,
        HiddenSize: 128,
        OutputSize: 12,
    }
```

Configuración de la red neuronal: Crea una instancia de NetworkConfig con los parámetros para la red neuronal.

InputSize: Tamaño de la entrada (1024).

HiddenSize: Tamaño de la capa oculta (128).

OutputSize: Tamaño de la salida (12, que corresponde al número de notas musicales).

```
rnn := neuralnet.NewRNN(config)
```

Creación de la red neuronal: Usa la configuración para crear una nueva instancia de la red neuronal recurrente (RNN).

```
rnn.Train(trainData, trainLabels, 10)
```

Entrenamiento del modelo: Llama al método Train de la RNN para entrenar el modelo usando los datos y etiquetas de entrenamiento. El número 10 representa el número de épocas o iteraciones de entrenamiento.

```
neuralnet.Validate(rnn, validData, validLabels)
```

Validación del modelo: Llama a la función Validate del paquete neuralnet para validar el modelo entrenado utilizando los datos y etiquetas de validación.

```
TestModel(rnn, testFilename)
```

Prueba del modelo: Llama a la función TestModel para probar el modelo entrenado con el archivo de audio especificado. Esta función cargará el archivo de audio, generará el espectrograma, normalizará los datos y hará predicciones usando la RNN.

La función main en el archivo main.go realiza los siguientes pasos:

- Llama a prepareData para obtener los conjuntos de datos de entrenamiento y validación
- Verifica que los conjuntos de datos no estén vacíos
- Configura los parámetros de la red neuronal
- Crea una nueva instancia de la red neuronal recurrente (RNN)
- Entrena la RNN con los datos de entrenamiento
- Valida el modelo entrenado usando los datos de validación
- Define un archivo de audio para la prueba
- Llama a TestModel para probar el modelo con el archivo de audio especificado y obtiene predicciones

rnn.go

El archivo rnn.go es parte de un programa diseñado para predecir notas musicales a partir de archivos de audio usando una red neuronal recurrente (RNN). El propósito general de este archivo es definir la estructura y las operaciones de la red neuronal, incluyendo cómo se inicializa, cómo realiza las predicciones y cómo se entrena.

El archivo define una red neuronal recurrente (RNN) para el procesamiento y la predicción de secuencias de datos (en este caso, notas musicales). La red neuronal se construye usando la biblioteca Gorgonia, que es una biblioteca de Go para computación numérica que facilita la creación y el entrenamiento de redes neuronales.

Definición de la Estructura de la Red

```
type NetworkConfig struct {
    InputSize  int
    HiddenSize int
    OutputSize int
}
```

- type en Go se usa para definir nuevos tipos de datos.
- struct define una estructura, que es una colección de campos. Es similar a una clase en otros lenguajes de programación.

La estructura NetworkConfig se usa para configurar la red neuronal, especificando el tamaño de la entrada, el tamaño de la capa oculta y el tamaño de la salida.

InputSize: El número de neuronas en la capa de entrada.

HiddenSize: El número de neuronas en la capa oculta.

OutputSize: El número de neuronas en la capa de salida.

```
type RNN struct {
    g      *gorgonia.ExprGraph
    w, h0  *gorgonia.Node
    outW   *gorgonia.Node
    vm     gorgonia.VM
    Config NetworkConfig
}
```

g *gorgonia.ExprGraph: g es un puntero a un grafo de expresiones de Gorgonia. Un grafo de expresiones es una estructura que representa las operaciones matemáticas que realiza la red.

*: El asterisco indica que g es un puntero. En Go, un puntero almacena la dirección de memoria de una variable. *gorgonia.ExprGraph significa que g es un puntero a un ExprGraph de Gorgonia.

w, h0 *gorgonia.Node: w es un nodo que representa los pesos de la red, y h0 es el estado oculto inicial.

w: Representa los pesos de las conexiones entre la capa de entrada y la capa oculta.

h0: Representa el estado inicial de la capa oculta. Se inicializa con ceros.

outW *gorgonia.Node: outW es un nodo que representa los pesos de la capa de salida.

outW: Representa los pesos de las conexiones entre la capa oculta y la capa de salida.

vm gorgonia.VM: vm es una máquina virtual que se usa para ejecutar el grafo de expresiones.

gorgonia.VM: VM significa "Virtual Machine" (Máquina Virtual). Esta máquina virtual ejecuta las operaciones definidas en el grafo de expresiones.

Config NetworkConfig: Config almacena la configuración de la red (tamaños de las capas de entrada, oculta y salida) usando la estructura NetworkConfig.

Explicación de los Nombres Abreviados

g: Es comúnmente usado como abreviatura de "graph" (grafo), representando el grafo de expresiones de la red

w: Es una abreviatura común para "weights" (pesos).

h0: Indica el estado oculto inicial (h de "hidden" y 0 para indicar que es el estado inicial)

outW: Indica los pesos de la capa de salida (out de "output" y W de "weights").

vm: Es una abreviatura para "virtual machine"

```
func NewRNN(config NetworkConfig) *RNN {
```

Esta línea define una nueva función llamada NewRNN que toma un parámetro de tipo NetworkConfig y retorna un puntero a una estructura RNN.

```
g := gorgonia.NewGraph()
```

Se crea un nuevo grafo de expresiones de Gorgonia y se asigna a la variable g.

gorgonia.NewGraph() es una llamada a la función que crea y retorna un nuevo grafo de expresiones.

```
inputSize, hiddenSize, outputSize := config.InputSize, config.HiddenSize, config.OutputSize
```

Se extraen los tamaños de las capas de entrada, oculta y de salida de la configuración proporcionada y se asignan a las variables locales inputSize, hiddenSize y outputSize, respectivamente.

config.InputSize, config.HiddenSize, y config.OutputSize acceden a los valores de inputSize, hiddenSize y outputSize en la estructura NetworkConfig.

```
w := gorgonia.NewMatrix(g, tensor.Float64, gorgonia.WithShape(inputSize+hiddenSize, hiddenSize), gorgonia.WithName("w"), gorgonia.WithInit(gorgonia.GlorotU(1)))
```

Se crea una nueva matriz de Gorgonia para los pesos w con las siguientes características:

- g: El grafo de expresiones al que pertenece esta matriz.
- tensor.Float64: El tipo de datos de los elementos de la matriz (números de punto flotante de 64 bits).
- gorgonia.WithShape(inputSize+hiddenSize, hiddenSize): La forma (dimensiones) de la matriz.
- gorgonia.WithName("w"): El nombre asignado a la matriz.
- gorgonia.WithInit(gorgonia.GlorotU(1)): El método de inicialización de los valores de la matriz (inicialización Glorot Uniforme).

```
h0 := gorgonia.NewMatrix(g, tensor.Float64, gorgonia.WithShape(1, hiddenSize), gorgonia.WithName("h0"), gorgonia.WithInit(gorgonia.Zeroes()))
```

Se crea una nueva matriz de Gorgonia para el estado oculto inicial h0 con las siguientes características:

- g: El grafo de expresiones al que pertenece esta matriz
- tensor.Float64: El tipo de datos de los elementos de la matriz
- gorgonia.WithShape(1, hiddenSize): La forma de la matriz (una fila y hiddenSize columnas)
- gorgonia.WithName("h0"): El nombre asignado a la matriz
- gorgonia.WithInit(gorgonia.Zeroes()): El método de inicialización de los valores de la matriz (inicialización con ceros)

```
outW := gorgonia.NewMatrix(g, tensor.Float64,
gorgonia.WithShape(hiddenSize, outputSize), gorgonia.WithName("outW"),
gorgonia.WithInit(gorgonia.GlorotU(1)))
```

Se crea una nueva matriz de Gorgonia para los pesos de la capa de salida outW con las siguientes características:

- g: El grafo de expresiones al que pertenece esta matriz.
- tensor.Float64: El tipo de datos de los elementos de la matriz.
- gorgonia.WithShape(hiddenSize, outputSize): La forma de la matriz (de hiddenSize filas y outputSize columnas).
- gorgonia.WithName("outW"): El nombre asignado a la matriz.
- gorgonia.WithInit(gorgonia.GlorotU(1)): El método de inicialización de los valores de la matriz.

```
vm := gorgonia.NewTapeMachine(g, gorgonia.BindDualValues(w, h0, outW))
```

Se crea una nueva máquina virtual de Gorgonia (TapeMachine) para ejecutar el grafo de expresiones.

- g: El grafo de expresiones que se ejecutará
- gorgonia.BindDualValues(w, h0, outW): Enlaza los valores duales (valores y gradientes) de w, h0 y outW para el cálculo de gradientes durante el entrenamiento

```
    return &RNN{
        g:      g,
        w:      w,
        h0:     h0,
        outW:   outW,
        vm:     vm,
        Config: config,
    }
```

Retorna un puntero a una nueva instancia de la estructura RNN con los campos inicializados:

- g: El grafo de expresiones.
- w: La matriz de pesos w.
- h0: La matriz del estado oculto inicial h0.
- outW: La matriz de pesos de la capa de salida outW.
- vm: La máquina virtual vm.
- Config: La configuración config proporcionada.

```
func (r *RNN) forward(x *tensor.Dense) (*gorgonia.Node, error) {
```

- Esta línea define el método forward que pertenece al tipo RNN.
- r *RNN indica que el método es parte de la estructura RNN.
- x *tensor.Dense es el parámetro que recibe el método, que es un tensor denso de entrada.
- (*gorgonia.Node, error) indica que el método retorna un puntero a un nodo de Gorgonia y un posible error.

```
    if x.Shape()[0] != 1 || x.Shape()[1] != r.Config.InputSize {
        return nil, fmt.Errorf("input tensor has incorrect shape,
expected [1, %d], got %v", r.Config.InputSize, x.Shape())
    }
```

- Verifica que la forma (shape) del tensor de entrada x sea correcta.
- x.Shape() obtiene la forma del tensor x.
- x.Shape()[0] debe ser 1, lo que indica que el tensor tiene una sola fila.
- x.Shape()[1] debe coincidir con r.Config.InputSize, asegurando que el número de columnas coincida con el tamaño de la entrada configurado.
- Si la forma no es la esperada, retorna nil y un error formateado con un mensaje descriptivo.

```
    xNode := gorgonia.NewTensor(r.g, tensor.Float64, 2,
gorgonia.WithShape(x.Shape()...), gorgonia.WithValue(x))
    if xNode.Value() == nil {
        return nil, fmt.Errorf("xNode value is nil")
    }
```

- Crea un nuevo nodo de tensor en el grafo r.g con el valor y la forma del tensor x.
- gorgonia.NewTensor(r.g, tensor.Float64, 2, gorgonia.WithShape(x.Shape()...), gorgonia.WithValue(x)) crea el tensor en el grafo con el tipo de datos Float64, dimensión 2 (matriz), y la forma y valores del tensor x.
- Verifica si xNode.Value() es nil, lo que indicaría un error en la creación del tensor, y si es así, retorna nil y un error.

```
    if r.h0 == nil {
        r.h0 = gorgonia.NewMatrix(r.g, tensor.Float64,
gorgonia.WithShape(1, r.Config.HiddenSize),
gorgonia.WithInit(gorgonia.Zeroes()))
    }
```

- Asegura que r.h0 (el estado oculto inicial) esté correctamente inicializado.
- Si r.h0 es nil, crea una nueva matriz con forma [1, r.Config.HiddenSize] e inicializa sus valores a ceros.

```
    fmt.Printf("Shape of xNode: %v\n", xNode.Shape())
    fmt.Printf("Shape of h0: %v\n", r.h0.Shape())
```

- Imprime en la consola las formas del nodo xNode y de r.h0 para fines de depuración.

```
    concatenated, err := gorgonia.Concat(1, xNode, r.h0)
    if err != nil {
        return nil, fmt.Errorf("failed to concatenate input and hidden
state: %v", err)
    }
```

- Concatena el tensor de entrada xNode y el estado oculto r.h0 a lo largo del eje de las columnas (eje 1)
- gorgonia.Concat(1, xNode, r.h0) concatena los dos tensores
- Si ocurre un error en la concatenación, retorna nil y un error con un mensaje descriptivo

```
r.w = gorgonia.NewMatrix(r.g, tensor.Float64,
gorgonia.WithShape(r.Config.InputSize+r.Config.HiddenSize,
r.Config.HiddenSize), gorgonia.WithInit(gorgonia.GlorotU(1)))
```

- Asegura que r.w esté correctamente inicializado con la forma adecuada y con valores inicializados usando la inicialización Glorot Uniforme.
- Crea una nueva matriz con forma [r.Config.InputSize + r.Config.HiddenSize, r.Config.HiddenSize].

```
  mulResult, err := gorgonia.Mul(concatenated, r.w)
    if err != nil {
        return nil, fmt.Errorf("failed matrix multiplication: %v", err)
    }
```

- Realiza la multiplicación de matrices entre el tensor concatenado y los pesos r.w
- gorgonia.Mul(concatenated, r.w) realiza la multiplicación de matrices
- Si ocurre un error en la multiplicación, retorna nil y un error con un mensaje descriptivo

```
r.h0 = mulResult
```

- Actualiza r.h0 con el nuevo estado oculto calculado en la multiplicación de matrices

```
r.outW = gorgonia.NewMatrix(r.g, tensor.Float64,
gorgonia.WithShape(r.Config.HiddenSize, r.Config.OutputSize),
gorgonia.WithInit(gorgonia.GlorotU(1)))
```

- Asegura que r.outW esté correctamente inicializado con la forma adecuada y con valores inicializados usando la inicialización Glorot Uniforme
- Crea una nueva matriz con forma [r.Config.HiddenSize, r.Config.OutputSize]

```
    output, err := gorgonia.Mul(r.h0, r.outW)
    if err != nil {
        return nil, fmt.Errorf("failed to add output weights: %v", err)
    }
```

- Realiza la transformación de la capa de salida multiplicando el estado oculto r.h0 con los pesos de salida r.outW

- gorgonia.Mul(r.h0, r.outW) realiza la multiplicación de matrices
- Si ocurre un error en la multiplicación, retorna nil y un error con un mensaje descriptivo

```
return output, nil
```

- Retorna el nodo de salida output y nil indicando que no hubo errores

```
func outputHasNaNOrInf(output *gorgonia.Node) bool {
```

- Define una función llamada outputHasNaNOrInf que toma como parámetro un puntero a un nodo de Gorgonia (output) y retorna un valor booleano (bool)
- output *gorgonia.Node es el nodo de salida de la red neuronal que se va a verificar
- bool indica que la función retorna true si se detecta NaN o Inf en el nodo de salida, y false en caso contrario

```
    if output.Value() == nil {
        fmt.Println("Output value is nil")
        return true
    }
```

- Verifica si el valor del nodo output es nil
- output.Value() obtiene el valor almacenado en el nodo output
- Si el valor es nil, imprime un mensaje en la consola indicando que el valor de salida es nil
- Retorna true porque un valor nil es considerado un problema

```
data, ok := output.Value().Data().([]float64)
if !ok {
    fmt.Println("Output data type is not []float64")
    return true
}
```

- Intenta convertir los datos del valor del nodo output a un slice de tipo []float64
- output.Value().Data() obtiene los datos subyacentes del valor del nodo
- data, ok := output.Value().Data().([]float64) intenta convertir estos datos a un slice de []float64 y asigna el resultado a data. ok será true si la conversión es exitosa
- Si la conversión no es exitosa (!ok), imprime un mensaje indicando que el tipo de datos de salida no es []float64
- Retorna true porque la conversión fallida es considerada un problema

```
for i, v := range data {
    if math.IsNaN(v) {
        fmt.Printf("NaN detected at index %d\n", i)
        hasNaN = true
    }
    if math.IsInf(v, 0) {
        fmt.Printf("Inf detected at index %d\n", i)
        hasInf = true
    }
}
```

- Itera sobre los datos (data) usando un bucle for
- for i, v := range data itera sobre cada elemento v en data y su índice correspondiente i
- if math.IsNaN(v) verifica si el valor v es NaN usando la función IsNaN del paquete math

- Si v es NaN, imprime un mensaje indicando que se detectó NaN en el índice i
- Establece hasNaN en true
- if math.IsInf(v, 0) verifica si el valor v es infinito (positivo o negativo) usando la función IsInf del paquete math
- Si v es infinito, imprime un mensaje indicando que se detectó infinito en el índice i
- Establece hasInf en true

La función outputHasNaNOrInf es crucial para garantizar la integridad de los cálculos en la red neuronal. Al verificar y detectar valores anómalos como NaN o Inf, se puede evitar que estos valores causen problemas más adelante en el proceso de entrenamiento o inferencia.

```
func lossHasNaN(loss *gorgonia.Node) bool {
```

- Define una función llamada lossHasNaN que toma como parámetro un puntero a un nodo de Gorgonia (loss) y retorna un valor booleano (bool)
- loss *gorgonia.Node es el nodo que contiene el valor de la pérdida que se va a verificar
- bool indica que la función retorna true si se detecta NaN en el valor de pérdida, y false en caso contrario

```
lossValue := loss.Value().Data().(float64)
```

- Obtiene el valor de pérdida del nodo loss y lo convierte a tipo float64
- loss.Value() obtiene el valor almacenado en el nodo loss
- loss.Value().Data() obtiene los datos subyacentes del valor del nodo
- (float64) convierte los datos obtenidos a tipo float64 y los asigna a la variable lossValue

```
return math.IsNaN(lossValue)
```

- Verifica si el valor de pérdida lossValue es NaN usando la función IsNaN del paquete math
- math.IsNaN(lossValue) retorna true si lossValue es NaN, y false en caso contrario
- Retorna el resultado de esta verificación

```
func (r *RNN) Train(data []*mat.Dense, labels []string, epochs int) {
```

- Define el método **Train** que pertenece al tipo RNN
- **r *RNN** indica que el método es parte de la estructura **RNN**
- **data []*mat.Dense** es el conjunto de datos de entrenamiento, una lista de matrices densas
- **labels []string** son las etiquetas correspondientes a los datos de entrenamiento
- **epochs int** es el número de épocas (iteraciones) que se realizarán durante el entrenamiento

```
    for _, d := range data {
        rawData := d.RawMatrix().Data
        t := tensor.New(tensor.WithShape(d.Dims()),
tensor.Of(tensor.Float64), tensor.WithBacking(rawData))
        tensorData = append(tensorData, t)
    }
```

- Declara una variable **tensorData** que almacenará los datos convertidos a tensores
- Itera sobre cada elemento **d** en **data**
- **d.RawMatrix().Data** obtiene los datos crudos de la matriz **d**

- **tensor.New(tensor.WithShape(d.Dims()), tensor.Of(tensor.Float64), tensor.WithBacking(rawData))** crea un nuevo tensor con la misma forma y datos que **d**
- Añade el tensor creado a la lista **tensorData**

```
labelTensors, err := prepareLabels(labels, r.Config.OutputSize, r.g)
   if err != nil {
       log.Println("Error preparing labels:", err)
        return
    }
```

- Prepara las etiquetas labels en forma de tensores llamando a la función prepareLabels
- prepareLabels(labels, r.Config.OutputSize, r.g) convierte las etiquetas en tensores adecuados para la red
- Si ocurre un error al preparar las etiquetas, lo registra y termina la ejecución del método

```
for epoch := 0; epoch < epochs; epoch++ {
    totalLoss := 0.0
```

- Inicia un bucle que se ejecuta durante el número especificado de épocas
- totalLoss se inicializa en 0.0 para acumular la pérdida total en cada época

```
      for i, t := range tensorData {
          reshapedTensor := tensor.New(tensor.WithShape(1, 1024),
tensor.Of(tensor.Float64), tensor.WithBacking(t.Data()))
```

- Inicia un bucle que itera sobre cada tensor **t** en **tensorData**
- **reshapedTensor := tensor.New(tensor.WithShape(1, 1024), tensor.Of(tensor.Float64), tensor.WithBacking(t.Data()))** crea un nuevo tensor con forma **[1, 1024]** usando los datos de **t**

```
            output, err := r.forward(reshapedTensor)
            if err != nil {
                log.Println("Error in forward pass:", err)
                continue
            }
```

- Realiza la pasada hacia adelante llamando al método **forward** con **reshapedTensor**
- Si ocurre un error durante la pasada hacia adelante, lo registra y salta a la siguiente iteración del bucle

```
            if outputHasNaNOrInf(output) {
                fmt.Printf("NaN or Inf detected in output at epoch %d,
index %d\n", epoch, i)
                continue
            }
```

- Verifica si el resultado de la pasada hacia adelante contiene NaN o Inf usando outputHasNaNOrInf
- Si se detecta NaN o Inf, imprime un mensaje y salta a la siguiente iteración del bucle

```
diff := gorgonia.Must(gorgonia.Sub(output, labelTensors[i]))
sqDiff := gorgonia.Must(gorgonia.Square(diff))
loss := gorgonia.Must(gorgonia.Mean(sqDiff))
```

- Calcula la diferencia entre el resultado de la red output y la etiqueta correspondiente labelTensors[i]
- diff := gorgonia.Must(gorgonia.Sub(output, labelTensors[i])) resta la etiqueta del resultado
- sqDiff := gorgonia.Must(gorgonia.Square(diff)) eleva al cuadrado la diferencia
- loss := gorgonia.Must(gorgonia.Mean(sqDiff)) calcula la pérdida promedio

```
            if lossHasNaN(loss) {
                fmt.Printf("NaN detected in loss at epoch %d, index
%d\n", epoch, i)
                continue
            }
```

- Verifica si la pérdida calculada contiene NaN usando lossHasNaN
- Si se detecta NaN, imprime un mensaje y salta a la siguiente iteración del bucle

```
if _, err := gorgonia.Grad(loss, r.w, r.h0, r.outW); err != nil {
            log.Println("Error computing gradients:", err)
            continue
        }
```

- Calcula los gradientes de la pérdida respecto a los pesos de la red (r.w, r.h0, r.outW) usando gorgonia.Grad
- Si ocurre un error al calcular los gradientes, lo registra y salta a la siguiente iteración del bucle

```
r.clipGradients(5.0)
```

- Llama al método clipGradients para recortar los gradientes y evitar problemas de desbordamiento, asumiendo que el método existe y maneja el recorte de gradientes de manera eficiente

```
        if err = r.vm.RunAll(); err != nil {
            log.Println("Error running the computation graph:", err)
            continue
        }
        r.vm.Reset()
```

- Ejecuta el grafo de computación con la máquina virtual **r.vm**
- Si ocurre un error al ejecutar el grafo, lo registra y salta a la siguiente iteración del bucle
- Resetea la máquina virtual **r.vm** para la próxima iteración

```
        totalLoss += loss.Value().Data().(float64)
}
```

- Acumula la pérdida total en **totalLoss** para cada tensor en **tensorData**

```
        averageLoss := totalLoss / float64(len(tensorData))
        fmt.Printf("Epoch %d: Average Loss: %.4f\n", epoch, averageLoss)
```

- Calcula la pérdida promedio dividiendo totalLoss por el número de tensores en tensorData
- Imprime la pérdida promedio para la época actual

```
func (r *RNN) clipGradients(maxNorm float64) {
```

- Define un método llamado clipGradients que pertenece al tipo RNN
- r *RNN indica que el método es parte de la estructura RNN
- maxNorm float64 es un parámetro que especifica la norma máxima permitida para los gradientes

```
for _, node := range []*gorgonia.Node{r.w, r.h0, r.outW} {
```

- Inicia un bucle for que itera sobre una lista de nodos de Gorgonia que representan los pesos de la red: **r.w**, **r.h0**, y **r.outW**
- node tomará el valor de cada uno de estos nodos en cada iteración del bucle

```
gradVal, _ := node.Grad()
```

- Obtiene el valor del gradiente asociado con el nodo actual
- node.Grad() retorna el gradiente del nodo
- gradVal almacena el valor del gradiente

```
gradTensor, _ := gradVal.(*tensor.Dense)
```

- Convierte el valor del gradiente a un tensor denso
- **gradVal.(*tensor.Dense)** realiza una aserción de tipo para convertir **gradVal** a un tensor denso
- **gradTensor** almacena el tensor denso del gradiente

```
data := gradTensor.Data().([]float64)
```

- Obtiene los datos subyacentes del tensor denso en forma de un slice de **float64**
- **gradTensor.Data().([]float64)** accede a los datos del tensor y los convierte a un slice de **float64**
- **data** almacena los datos del gradiente

```
norm := 0.0
```

- Inicializa una variable norm a 0.0 para acumular la norma del gradiente

```
for _, v := range data {
    norm += v * v
}
```

- Inicia un bucle **for** que itera sobre cada valor **v** en **data**

- **norm += v * v** suma el cuadrado de cada valor **v** a **norm**, acumulando la suma de los cuadrados de los gradientes

```
norm = math.Sqrt(norm)
```

- Calcula la raíz cuadrada de la suma de los cuadrados de los gradientes para obtener la norma euclidiana (L2) del gradiente
- **norm = math.Sqrt(norm)** actualiza norm con la raíz cuadrada de su valor actual

```
if norm > maxNorm {
            scale := maxNorm / norm
```

- Verifica si la norma del gradiente excede **maxNorm**
- **if norm > maxNorm** comprueba si **norm** es mayor que **maxNorm**
- Si es así, calcula un factor de escala para reducir los gradientes
- **scale := maxNorm / norm** calcula el factor de escala como la proporción entre **maxNorm** y **norm**

```
        for i := range data {
            data[i] *= scale
        }
```

- Inicia un bucle **for** que itera sobre cada índice **i** en **data**
- **data[i] *= scale** escala cada valor en **data** multiplicándolo por **scale**
- Esto reduce los gradientes para que su norma no exceda **maxNorm**

```
func prepareLabels(labels []string, numClasses int, g
*gorgonia.ExprGraph) ([]*gorgonia.Node, error) {
```

- **labels [ ]string**: una lista de etiquetas en formato string
- **numClasses int**: el número total de clases posibles
- **g *gorgonia.ExprGraph**: un grafo de expresiones de Gorgonia donde se agregarán los nodos de las etiquetas
- La función retorna un slice de punteros a nodos de Gorgonia (**[]*gorgonia.Node**) y un posible error (**error**)

```
    if numClasses <= 0 {
        return nil, fmt.Errorf("invalid number of classes: %d",
numClasses)
    }
```

- Verifica que el número de clases (**numClasses**) sea mayor que cero
- Si **numClasses** es menor o igual a cero, retorna **nil** y un error indicando que el número de clases es inválido

```
    var result []*gorgonia.Node
```

- Declara una variable **result** que almacenará los nodos de Gorgonia correspondientes a las etiquetas en formato one-hot

```
    for _, label := range labels {
        index, err := labelToIndex(label)
        if err != nil {
            return nil, err // Propagar el error hacia arriba
        }
```

- Inicia un bucle que itera sobre cada etiqueta en **labels**
- **labelToIndex(label)** convierte la etiqueta **label** en un índice entero (**index**) correspondiente a la clase
- Si ocurre un error durante la conversión, retorna **nil** y el error

```
        if index < 0 || index >= numClasses {
            return nil, fmt.Errorf("index out of range [%d] with
numClasses %d", index, numClasses)
        }
```

- Verifica que el índice (index) esté dentro del rango válido [0, numClasses)
- Si index está fuera de este rango, retorna nil y un error indicando que el índice está fuera de rango

```
   oneHot := make([]float64, numClasses)
   oneHot[index] = 1.0
```

- Crea un vector one-hot con el tamaño **numClasses**, inicializado con ceros
- Establece el valor en la posición **index** a 1.0, convirtiendo el vector en una representación one-hot de la etiqueta

```
t := tensor.New(tensor.WithBacking(oneHot), tensor.WithShape(numClasses))
```

- Crea un nuevo tensor de Gorgonia con los valores del vector one-hot y la forma **numClasses**
- **tensor.WithBacking(oneHot)** proporciona los datos del tensor
- **tensor.WithShape(numClasses)** define la forma del tensor

```
node := gorgonia.NewTensor(g, tensor.Float64, 1, gorgonia.WithValue(t))
```

- Crea un nuevo nodo de tensor en el grafo de Gorgonia **g** con el tensor **t**
- **tensor.Float64** especifica que los datos son de tipo **float64**
- 1 indica que el tensor es un vector (una dimensión)

```
result = append(result, node)
```

- Añade el nodo creado al slice result

```
return result, nil
```

- Termina el bucle **for** que itera sobre las etiquetas
- Retorna el slice **result** que contiene los nodos de las etiquetas en formato one-hot y **nil** indicando que no hubo errores

Función labelToIndex

La función labelToIndex convierte una etiqueta de nota musical (en forma de cadena de texto) en un índice entero correspondiente a esa nota. Esta función es crucial para mapear las etiquetas categóricas de notas musicales a índices numéricos que se puedan utilizar en la red neuronal, especialmente en el proceso de one-hot encoding. Si la etiqueta no es válida, la función retorna un error.

```
func labelToIndex(label string) (int, error) {
```

- Define una función llamada **labelToIndex** que toma como parámetro una cadena de texto (**label**) y retorna un entero (**int**) y un posible error (error)
- **label string**: La etiqueta de la nota musical que se va a convertir
- **(int, error)**: Los valores de retorno son el índice correspondiente a la nota y un posible error

```
    noteToIndex := map[string]int{
        "C": 0, "C#": 1, "D": 2, "D#": 3,
        "E": 4, "F": 5, "F#": 6, "G": 7,
        "G#": 8, "A": 9, "A#": 10, "B": 11,
    }
```

- Define un mapa llamado **noteToIndex** que asocia cada nota musical con un índice entero
- El tipo del mapa es **map[string]int**, donde las claves son cadenas de texto que representan las notas musicales y los valores son enteros que representan los índices correspondientes
- El mapa contiene todas las notas musicales de una octava, incluyendo las notas sostenidas (representadas con #)

```
index, exists := noteToIndex[label]
```

- Busca la etiqueta **label** en el mapa **noteToIndex**
- **index** es el valor asociado con la clave **label** en el mapa
- **exists** es un booleano que indica si la clave **label** existe en el mapa

```
 if !exists {
     return -1, fmt.Errorf("label '%s' is not a valid note", label)
 }
```

- Verifica si **label** no existe en el mapa **noteToIndex**
- **if !exists** comprueba si **exists** es **false**
- Si **label** no existe en el mapa, retorna **-1** y un error formateado con un mensaje que indica que la etiqueta no es una nota válida

```
return index, nil
```

- Retorna el índice asociado con la etiqueta label y nil indicando que no hubo errores

```
func Validate(rnn *RNN, validData []*mat.Dense, validLabels []string) {
```

- **rnn *RNN**: un puntero a la instancia de la red neuronal recurrente a validar
- **validData []*mat.Dense**: un slice de punteros a matrices densas que representan los datos de validación
- **validLabels [ ]string**: un slice de cadenas que representan las etiquetas correspondientes a los datos de validación

```
var totalLoss float64
```

- Declara una variable **totalLoss** para acumular la pérdida total calculada sobre el conjunto de datos de validación

```
for i, vData := range validData {
```

- Inicia un bucle **for** que itera sobre cada elemento **vData** en **validData**, junto con su índice **i**

```
vTensor := tensor.New(tensor.WithShape(vData.Dims()),
tensor.Of(tensor.Float64), tensor.WithBacking(vData.RawMatrix().Data))
```

- Convierte cada matriz de datos de validación **vData** en un tensor de Gorgonia
- **tensor.WithShape(vData.Dims())** especifica la forma del tensor basada en las dimensiones de **vData**
- **tensor.Of(tensor.Float64)** indica que los elementos del tensor son de tipo float64
- **tensor.WithBacking(vData.RawMatrix().Data)** proporciona los datos crudos de vData como respaldo del tensor

```
        output, err := rnn.forward(vTensor)
        if err != nil {
            log.Printf("Error during validation forward pass: %v\n", err)
            continue
        }
```

- Realiza una pasada hacia adelante en la red neuronal con el tensor de datos de validación vTensor
- Si ocurre un error durante la pasada hacia adelante, lo registra y continúa con la siguiente iteración del bucle

```
labelTensor, err := prepareLabels([]string{validLabels[i]},
rnn.Config.OutputSize, rnn.g)
        if err != nil {
            log.Printf("Error preparing labels: %v\n", err)
            continue
        }
```

- Prepara las etiquetas de validación en formato tensor llamando a la función **prepareLabels** con la etiqueta correspondiente
- **prepareLabels([ ]string{ validLabels[i] }, rnn.Config.OutputSize, rnn.g)** convierte la etiqueta en un tensor adecuado
- Si ocurre un error al preparar las etiquetas, lo registra y continúa con la siguiente iteración del bucle

```
diff := gorgonia.Must(gorgonia.Sub(output, labelTensor[0]))
sqDiff := gorgonia.Must(gorgonia.Square(diff))
loss := gorgonia.Must(gorgonia.Mean(sqDiff))
```

- Calcula la diferencia entre la salida de la red **output** y la etiqueta de validación **labelTensor[0]**
- **gorgonia.Must(gorgonia.Sub(output, labelTensor[0]))** resta la etiqueta de la salida de la red
- **gorgonia.Must(gorgonia.Square(diff))** eleva al cuadrado la diferencia

- **gorgonia.Must(gorgonia.Mean(sqDiff))** calcula la pérdida promedio (error cuadrático medio)

```
totalLoss += loss.Value().Data().(float64)
```

- Acumula la pérdida total sumando el valor de la pérdida actual a **totalLoss**

```
if len(validData) > 0 {
    averageLoss := totalLoss / float64(len(validData))
    fmt.Printf("Validation Loss: %.4f\n", averageLoss)
} else {
    fmt.Println("No valid data provided for validation.")
}
```

- Verifica si hay datos de validación disponibles
- Si hay datos de validación, calcula la pérdida promedio dividiendo **totalLoss** por el número de elementos en **validData**
- Imprime la pérdida promedio de validación en la consola
- Si no hay datos de validación, imprime un mensaje indicando que no se proporcionaron datos de validación

Función Predict (rnn.go)

La función Predict se utiliza para realizar predicciones con la red neuronal recurrente (RNN) usando un tensor de entrada. Toma el tensor de entrada, realiza una pasada hacia adelante a través de la red para obtener las salidas, convierte las salidas a probabilidades y luego decodifica estas probabilidades en predicciones de notas musicales. Esta función es crucial para aplicar el modelo entrenado a nuevos datos y obtener predicciones significativas.

```
func (r *RNN) Predict(x tensor.Tensor) []string {
```

- Define el método Predict que pertenece al tipo RNN

- **x tensor.Tensor** es el tensor de entrada que contiene los datos para los que se quieren hacer predicciones
- La función retorna un slice de cadenas (**[ ]string**) que representan las notas musicales predichas

```
if r.vm == nil {
   fmt.Println("No VM initialized")
   return nil
}
```

- Verifica si la máquina virtual (**r.vm**) está inicializada
- Si **r.vm** es **nil**, imprime un mensaje indicando que no hay una máquina virtual inicializada y retorna **nil**

```
    if _, ok := x.Data().([]float64); !ok {
        fmt.Println("Invalid tensor data type; expected []float64")
        return nil
    }
    if x.Shape()[0] != 1 || x.Shape()[1] != r.Config.InputSize {
        fmt.Printf("Invalid input shape; expected [1, %d], got %v\n",
r.Config.InputSize, x.Shape())

        return nil
    }
```

- Verifica si el tipo de dato y la forma del tensor de entrada son los esperados
- **x.Data().([]float64)** intenta convertir los datos del tensor a un slice de **float64**. Si falla, imprime un mensaje indicando que el tipo de datos es inválido y retorna **nil**
- **x.Shape()[0] != 1 || x.Shape()[1] != r.Config.InputSize** verifica que la forma del tensor sea **[1, r.Config.InputSize]**. Si la forma no es la esperada, imprime un mensaje indicando la forma inválida y retorna **nil**

```
    if !ok {
        fmt.Println("Tensor is not of type *tensor.Dense as required")
        return nil
```

```
}
```

- Verifica si el tensor **x** es del tipo ***tensor.Dense**, necesario para la función **forward**
- Si **x** no es del tipo ***tensor.Dense**, imprime un mensaje indicando el tipo incorrecto y retorna **nil**

```
output, err := r.forward(denseX)
if err != nil {
    fmt.Printf("Error during prediction: %v\n", err)
    return nil
}
```

- Realiza la pasada hacia adelante en la red neuronal usando el tensor de entrada **denseX**
- Si ocurre un error durante la pasada hacia adelante, imprime un mensaje con el error y retorna **nil**

```
outputData, ok := output.Value().Data().([]float64)
if !ok {
    fmt.Println("Output data type assertion failed")
    return nil
}
```

- Verifica si los datos de salida pueden convertirse a un slice de **float64**
- **output.Value().Data().([]float64)** intenta convertir los datos de salida. Si falla, imprime un mensaje indicando que la conversión falló y retorna **nil**

```
probs := softmax(outputData)
threshold := 0.5
predictedNotes := decodePredictions(probs, threshold)
return predictedNotes
```

- Calcula las probabilidades usando la función **softmax** en los datos de salida **outputData**

- Define un umbral (**threshold**) para la decodificación de las predicciones
- **decodePredictions(probs, threshold)** decodifica las probabilidades en notas musicales usando el umbral
- Retorna las notas musicales predichas (**predictedNotes**)

Función Softmax (rnn.go)

La función **softmax** convierte un vector de valores (logits) en un vector de probabilidades, donde cada valor está en el rango [0, 1] y la suma de todas las probabilidades es 1. En el caso de esta red neuronal para predecir notas musicales, **softmax** se utiliza para convertir las salidas de la red en probabilidades de cada nota musical.

```
func softmax(x []float64) []float64 {
```

- Se define la función **softmax** que toma como entrada un slice de **float64** (**x**) y retorna otro slice de **float64**
- **x [ ]float64** es el array de logits que necesita ser convertido en probabilidades

```
    if len(x) == 0 {
        return nil
    }
```

- Verifica si la longitud del vector x es 0
- Si x está vacío, la función retorna nil, ya que no hay nada que convertir

```
exps := make([]float64, len(x))
```

- Crea un nuevo slice exps de la misma longitud que x para almacenar los valores exponenciales de los elementos de x

```
maxVal := max(x)
```

- Llama a una función **max** (no mostrada en este fragmento) que encuentra el valor máximo en el array **x**
- **maxVal** almacena este valor máximo
- Encontrar el valor máximo se utiliza para evitar desbordamientos numéricos al calcular las exponenciales

```
var sum float64
```

- Declara una variable sum de tipo float64 para acumular la suma de los valores exponenciales

```
for i, v := range x {
    exps[i] = math.Exp(v - maxVal)
    sum += exps[i]
}
```

- Itera sobre cada índice **i** y valor **v** en **x**
- Calcula la exponencial de **v - maxVal** y la almacena en **exps[i]**
- **math.Exp(v - maxVal)** calcula la exponencial de **v - maxVal**
- Restar **maxVal** de **v** ayuda a evitar desbordamientos numéricos y mejora la estabilidad numérica
- Acumula la suma de los valores exponenciales en **sum**

```
for i := range exps {
    exps[i] /= sum
}
```

- Itera sobre cada índice **i** en **exps**
- Divide cada valor en **exps** por **sum** para normalizar las probabilidades
- **exps[i] /= sum** convierte cada valor en **exps** a una probabilidad, asegurando que la suma de todas las probabilidades sea 1

```
return exps
```

- Retorna el slice **exps**, que ahora contiene las probabilidades normalizadas

Función decodePredictions (rnn.go)

La función **decodePredictions** toma un array de probabilidades y un umbral (threshold) para determinar qué notas musicales se deben predecir. La función compara cada probabilidad con el umbral y, si la probabilidad es mayor que el umbral, incluye la nota correspondiente en la lista de resultados.

```
func decodePredictions(probs []float64, threshold float64) []string {
```

- **probs [ ]float64**: Un slice de probabilidades generadas por la función softmax
- **threshold float64**: Un valor umbral para determinar si una nota debe ser incluida en las predicciones
- La función retorna un slice de string (**[ ]string**) que representa las notas musicales predichas

```
notes := []string{"C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A",
"A#", "B"}
```

- Define un slice **notes** que contiene las representaciones de las notas musicales
- Estas son las clases posibles que se pueden predecir

```
var results []string
```

- Declara una variable **results** que almacenará las notas musicales predichas

```
numClasses := len(notes)
```

- Define la variable **numClasses** como la longitud del slice **notes**
- Esto asegura que el número de clases coincida con la longitud de las notas musicales

```
if len(probs) < numClasses {
  numClasses = len(probs)
}
```

- Verifica si la longitud de **probs** es menor que **numClasses**
- Si es así, ajusta **numClasses** a la longitud de **probs** para evitar accesos fuera de rango

```
 for i := 0; i < numClasses; i++ {
     if probs[i] > threshold {
         results = append(results, notes[i])
     }
 }
```

- Inicia un bucle **for** que itera desde **0** hasta **numClasses**
- **if probs[i] > threshold** verifica si la probabilidad de la clase actual es mayor que el umbral
- Si la probabilidad es mayor que el umbral, añade la nota correspondiente a **results**
- **results = append(results, notes[i])** añade la nota **notes[i]** al slice **results**

```
return results
```

- Retorna el slice results, que contiene las notas musicales predichas

audio.go

Función LoadAudio

La función LoadAudio carga un archivo de audio en formato WAV y retorna un slice de enteros con las muestras de audio. Esta función también asegura que el número de muestras sea al menos igual al tamaño del marco (frameSize), rellenando con ceros si es necesario. A continuación, se explica línea por línea:

```
func LoadAudio(filename string, frameSize int) ([]int, error) {
```

Esta línea declara la función LoadAudio que toma dos parámetros: filename (un string que representa el nombre del archivo de audio) y frameSize (un int que indica el tamaño del marco de las muestras). La función devuelve un slice de int y un posible error.

```
    file, err := os.Open(filename)
    if err != nil {
        return nil, fmt.Errorf("error opening file %s: %w", filename,
err)
    }
    defer file.Close()
```

Estas líneas abren el archivo de audio especificado por **‘ filename ’**. Si ocurre un error al abrir el archivo, la función retorna nil y un mensaje de error. La línea ‘ **defer file.Close() ’** asegura que el archivo se cierre automáticamente cuando la función termine, liberando así los recursos.

```
    decoder := wav.NewDecoder(file)
```

```
    if !decoder.IsValidFile() {
        return nil, fmt.Errorf("invalid WAV file %s", filename)
    }
```

Se crea un nuevo decodificador WAV utilizando el archivo abierto. Se verifica si el archivo es un archivo WAV válido. Si no es válido, la función retorna nil y un mensaje de error.

```
    buff, err := decoder.FullPCMBuffer()
    if err != nil {
        return nil, fmt.Errorf("error decoding WAV file %s: %w",
filename, err)
    }
    if buff.NumFrames() == 0 {
        return nil, fmt.Errorf("empty audio buffer in WAV file %s",
filename)
    }
```

Estas líneas decodifican todo el contenido del archivo WAV en un buffer de audio. Si ocurre un error durante la decodificación, la función retorna nil y un mensaje de error. La siguiente verificación asegura que el buffer no esté vacío; si lo está, retorna nil y un mensaje de error.

El buffer de audio es necesario porque actúa como un contenedor temporal para las muestras de audio que se extraen del archivo WAV. Las muestras de audio son datos digitales que representan la señal de audio en un formato que puede ser procesado por algoritmos y sistemas de análisis.

Almacenamiento Temporal de Muestras de Audio

El buffer de audio almacena todas las muestras de audio leídas del archivo WAV. Esto permite que las muestras estén disponibles en un formato estructurado y accesible para su procesamiento posterior. Al tener todas las muestras en un buffer, se facilita el acceso secuencial o aleatorio a las mismas según sea necesario para diferentes operaciones.

Manipulación y Procesamiento

Una vez que las muestras de audio están en el buffer, pueden ser manipuladas y procesadas de diversas maneras. Por ejemplo, se pueden aplicar algoritmos de procesamiento de señales, como la transformación de Fourier para obtener espectrogramas, o técnicas de análisis para identificar notas musicales.

Verificación de Integridad

Al cargar las muestras en un buffer, es posible verificar su integridad, como se hace en el código con la línea **if buff.NumFrames() == 0**. Esto asegura que el archivo de audio no esté vacío y que contenga datos válidos antes de proceder con operaciones adicionales.

```
    numSamples := buff.NumFrames()
    if numSamples < frameSize {
        extendedData := make([]int, frameSize)
        copy(extendedData, buff.Data)

        for i := numSamples; i < frameSize; i++ {
            extendedData[i] = 0
        }
        fmt.Printf("Audio from %s was padded from %d to %d samples\n",
filename, numSamples, frameSize)
        return extendedData, nil
    }
```

La función verifica si el número de muestras en el buffer es menor que **frameSize**. Si es así, crea un nuevo slice **extendedData** con el tamaño de **frameSize**, copia las muestras de audio en este nuevo slice y rellena el resto con ceros. Luego imprime un mensaje indicando que el audio fue rellenado y retorna el **extendedData**.

```
    fmt.Printf("Loaded %d samples from %s\n", numSamples, filename)
    return buff.Data, nil
}
```

Finalmente, si el número de muestras es igual o mayor que frameSize, la función imprime un mensaje indicando la cantidad de muestras cargadas y retorna las muestras del buffer.

Función GenerateSpectogram

La función **GenerateSpectrogram** toma un slice de enteros que representan datos de audio y un tamaño de marco (**frameSize**) para generar un espectrograma, que es una representación visual de las frecuencias presentes en el audio a lo largo del tiempo.

```
func GenerateSpectrogram(data []int, frameSize int) [][]float64 {
```

Esta línea declara la función GenerateSpectrogram, que recibe dos parámetros: **data** (un slice de enteros que contiene los datos de audio) y **frameSize** (un entero que indica el tamaño de cada segmento de análisis). La función devuelve una matriz de slices de floats (**[ ][ ]float64**), que representa el espectrograma.

```
    if frameSize <= 0 || frameSize > len(data) {
        return nil // Retorna nil para frameSize no válido
    }
```

Aquí se verifica si el **frameSize** es válido. Si **frameSize** es menor o igual a cero, o mayor que la longitud de los datos de audio, la función retorna **nil**.

```
fft := fourier.NewFFT(frameSize)
```

Esta línea crea una nueva instancia de la transformada rápida de Fourier (FFT) con el tamaño de marco especificado. La FFT se utilizará para convertir los datos de audio del dominio del tiempo al dominio de la frecuencia.

```
window := hannWindow(frameSize)
```

Aquí se crea una ventana de Hann de tamaño **frameSize**. La ventana de Hann se utiliza para reducir las discontinuidades al inicio y al final de cada segmento de datos de audio, minimizando los efectos de borde en el análisis de Fourier.

```
spectrogram := make([][]float64, 0)
```

Se inicializa una variable spectrogram como un slice de slices de floats. Esta variable almacenará el espectrograma generado.

```
    for start := 0; start < len(data); start += frameSize {
        end := start + frameSize
        if end > len(data) {
            end = len(data)
            tempData := make([]int, frameSize) // Asegura que el buffer
tenga siempre frameSize elementos
            copy(tempData, data[start:end])
            data = tempData
        } else {
            data = data[start:end]
        }
```

Este bloque de código itera sobre los datos de audio en segmentos de tamaño **frameSize**. Para cada segmento:

Calcula el índice de inicio (**start**) y el índice de fin (**end**) del segmento.

Si el índice de fin supera la longitud de los datos de audio, ajusta **end** a la longitud de los datos y crea un **tempData** de tamaño **frameSize**, copiando los datos del segmento y rellenando el resto con ceros si es necesario.

Si el índice de fin no supera la longitud de los datos, simplemente extrae el segmento de los datos de audio.

```
frame := make([]float64, frameSize)
for i := range frame {
    frame[i] = float64(data[i]) * window[i]
}
```

Se crea un slice **frame** de floats de tamaño **frameSize**. Luego, cada valor del **frame** se multiplica por el valor correspondiente de la ventana de Hann. Esto aplica la ventana de Hann al segmento de datos de audio.

```
coeff := fft.Coefficients(nil, frame)
```

Se calculan los coeficientes de la FFT para el segmento de datos de audio modulado por la ventana de Hann. Estos coeficientes representan las amplitudes de las distintas frecuencias presentes en el segmento.

```
power := make([]float64, frameSize)
for i := range power {
    if i < len(coeff)/2 {
        realPart := real(coeff[i])
        imagPart := imag(coeff[i])
        power[i] = realPart*realPart + imagPart*imagPart
    } else {
        power[i] = 0
    }
    if math.IsNaN(power[i]) || math.IsInf(power[i], 0) {
        power[i] = 0
    }
}
```

Se crea un slice **power** de floats de tamaño **frameSize** para almacenar la potencia espectral del segmento. Para cada coeficiente de la FFT:

Si el índice está en la primera mitad de los coeficientes (ya que los coeficientes son simétricos), se calculan las partes real e imaginaria, y se suma el cuadrado de ambas para obtener la potencia.

Si el índice está en la segunda mitad, se rellena con ceros.

Se verifica si la potencia es **NaN** (Not a Number) o infinita, y se reemplaza con cero si es necesario.

```
        spectrogram = append(spectrogram, power)
    }
```

Se agrega el slice de potencias calculado para el segmento actual al espectrograma.

```
    return spectrogram
}
```

Finalmente, la función retorna el espectrograma generado.

Función hannWindow

La función **hannWindow** genera una ventana de Hann de un tamaño especificado. La ventana de Hann es una función de ponderación que se aplica a una señal de audio antes de realizar la transformada de Fourier, con el fin de reducir las discontinuidades al principio y al final de cada segmento.

```
func hannWindow(size int) []float64 {
```

Esta línea declara la función **hannWindow**, que recibe un parámetro **size** (un entero que representa el tamaño de la ventana) y devuelve un slice de floats (**[ ]float64**), que contiene los coeficientes de la ventana de Hann.

```
if size <= 1 {
    return []float64{1}
}
```

Este bloque de código maneja casos especiales donde el tamaño de la ventana es menor o igual a 1:

Si **size** es menor o igual a 1, se retorna un slice que contiene un único valor **1**. Esto es porque una ventana de tamaño 1 no tiene ningún efecto en la señal de audio.

```
window := make([]float64, size)
```

Se inicializa un slice de floats llamado **window** con el tamaño especificado. Este slice almacenará los coeficientes de la ventana de Hann.

```
piFactor := 2 * math.Pi / float64(size-1)
```

Se calcula un factor de **pi** precomputado (**piFactor**) que se utilizará dentro del bucle para reducir las operaciones repetitivas. Este factor es **2 * π / (size - 1)** y se usa en la fórmula de la ventana de Hann.

```
for i := range window {
    window[i] = 0.5 * (1 - math.Cos(piFactor*float64(i)))
}
```

Este bucle itera sobre cada índice **i** del slice **window**:

- Para cada **i**, se aplica la fórmula de la ventana de Hann: 0.5 × (1 − cos(\piFactor × i))
- Esta fórmula calcula el valor del coeficiente de la ventana de Hann para la posición **i**.

```
    return window
}
```

Finalmente, la función retorna el slice **window** que contiene los coeficientes de la ventana de Hann.

Función NormalizeSpectogram

La función **NormalizeSpectrogram** toma un espectrograma y normaliza cada fila para que sus valores estén en un rango uniforme, usualmente entre 0 y 1. Esto se hace dividiendo cada valor en la fila por el valor máximo absoluto de esa fila. Esta normalización es útil para comparaciones y análisis, asegurando que las variaciones en magnitud no distorsionen los resultados.

```
func NormalizeSpectrogram(spectrogram [][]float64) [][]float64 {
```

Esta línea declara la función **NormalizeSpectrogram**, que recibe un espectrograma representado como una matriz de floats (**[ ][ ]float64**) y devuelve la misma estructura de datos, pero con los valores normalizados.

```
for j := range row {
```

Este bucle **for** itera sobre cada fila del espectrograma. La variable **i** representa el índice de la fila y **row** es la fila actual (un slice de floats).

```
maxVal := findMaxAbs(row)
```

Para cada fila, se llama a la función **findMaxAbs** para encontrar el valor absoluto máximo de esa fila. Este valor se almacena en **maxVal**. Este valor se utilizará para normalizar los valores de la fila.

```
        if maxVal == 0 { // Evitar división por cero
            continue
        }
```

Si **maxVal** es igual a cero, la fila se omite (se continúa con la siguiente iteración del bucle). Esto evita una división por cero, que resultaría en valores **NaN** (Not a Number) o infinitos.

```
for j := range row {
```

Este segundo bucle **for** itera sobre cada valor en la fila actual. La variable j representa el índice del valor dentro de la fila.

```
spectrogram[i][j] /= maxVal
```

Para cada valor en la fila, se realiza la normalización dividiendo el valor por **maxVal**. Esto escala todos los valores de la fila para que estén en un rango relativo al valor máximo de la fila.

```
            if math.IsNaN(spectrogram[i][j]) ||
math.IsInf(spectrogram[i][j], 0) {
                spectrogram[i][j] = 0
            }
```

Después de la normalización, se verifica si el valor resultante es **NaN** o infinito (**Inf**). Si es así, el valor se reemplaza por cero. Esto asegura que el espectrograma no contenga valores que podrían causar problemas en análisis posteriores.

```
    return spectrogram
}
```

Finalmente, la función retorna el espectrograma normalizado.

Función findMaxAbs

La función **findMaxAbs** encuentra el valor absoluto máximo en un slice de números flotantes. Este valor se utiliza para normalizar los datos en otras funciones.

```
func findMaxAbs(slice []float64) float64 {
```

Esta línea declara la función **findMaxAbs**, que recibe un slice de números flotantes (**[ ]float64**) y devuelve un solo valor flotante (**float64**).

```
maxVal := 0.0
```

Se inicializa una variable **maxVal** con el valor 0.0. Esta variable almacenará el valor absoluto máximo encontrado en el slice.

```
for _, v := range slice {
```

Este bucle **for** itera sobre cada valor en el slice. La variable _ se usa para ignorar el índice, y **v** representa el valor actual en la iteración.

```
absV := math.Abs(v)
```

Para cada valor **v**, se calcula su valor absoluto usando la función **math.Abs**, y el resultado se almacena en **absV**.

```
   if absV > maxVal {
            maxVal = absV
        }
```

Se compara **absV** con **maxVal**. Si **absV** es mayor que **maxVal**, se actualiza **maxVal** para que sea igual a **absV**. Esto asegura que **maxVal** siempre contenga el valor absoluto más grande encontrado hasta ese punto en el slice.

```
    return maxVal
}
```

Finalmente, la función retorna **maxVal**, que ahora contiene el valor absoluto máximo del slice.

Función ChackNan

La función **CheckNaN** revisa si hay valores **NaN** (Not a Number) en una matriz de números flotantes (**[ ][ ]float64**). Si encuentra algún valor **NaN**, imprime su posición y retorna t**rue**. Si no encuentra ningún valor **NaN**, retorna **false**.

```
func CheckNaN(data [][]float64) bool {
```

Esta línea declara la función **CheckNaN**, que recibe una matriz de números flotantes (**[ ][ ]float64**) y devuelve un valor booleano (**bool**).

```
for i, series := range data {
```

Este bucle **for** itera sobre cada fila de la matriz **data**. La variable **i** representa el índice de la fila, y **series** es la fila actual (un slice de floats).

```
for j, value := range series {
```

Este segundo bucle **for** itera sobre cada valor en la fila actual **series**. La variable **j** representa el índice del valor dentro de la fila, y **value** es el valor actual.

```
if math.IsNaN(value) {
```

Dentro del bucle interno, se utiliza la función **math.IsNaN** para verificar si el **value** actual es **NaN**.

```
fmt.Printf("NaN found at position [%d][%d]\n", i, j)
```

Si **value** es **NaN**, se imprime un mensaje en la consola que indica la posición **[i] [j]** del **NaN** dentro de la matriz. Esta línea es opcional y sirve como un registro para identificar dónde se encuentra el **NaN**.