Trabajo Práctico VI - Encodeador de música

Zoe Borrone - 21G245 Luca Mazzarello - 21L720 Ignacio Pardo - 21R1160

2023-11-26

Índice

Introducción 2
1) Encodear la canción en un vector latente 2
Arquitectura Net1 3
Arquitectura Net2 5
Arquitectura Net3 7
Arquitectura Net4 10
Arquitectura Net5 12

Introducción

Para este trabajo práctico se nos puso como objetivo implementar un encodeador de música con redes neuronales para poder obtener un vector representativo de las canciones en un espacio latente.

Luego, estos vectores encontrados deben poder desencodearse para volver a reproducir el sonido original. Una vez aplicados el encoding y decoding, realizamos un análisis exploratorio para comprender y encontrar relaciones de estos nuevos vectores, los cuales serán explicados en el desarrollo del informe.

La base de datos usada para el desarrollo del mismo fue "GTZAN". Esta cuenta con 1000 canciones, cada una con una duración de 30 segundos. Las canciones están divididas y clasificadas en 10 géneros distintos: blues, classical, country, disco, hip hop, jazz, metal, pop, reggae, y rock.

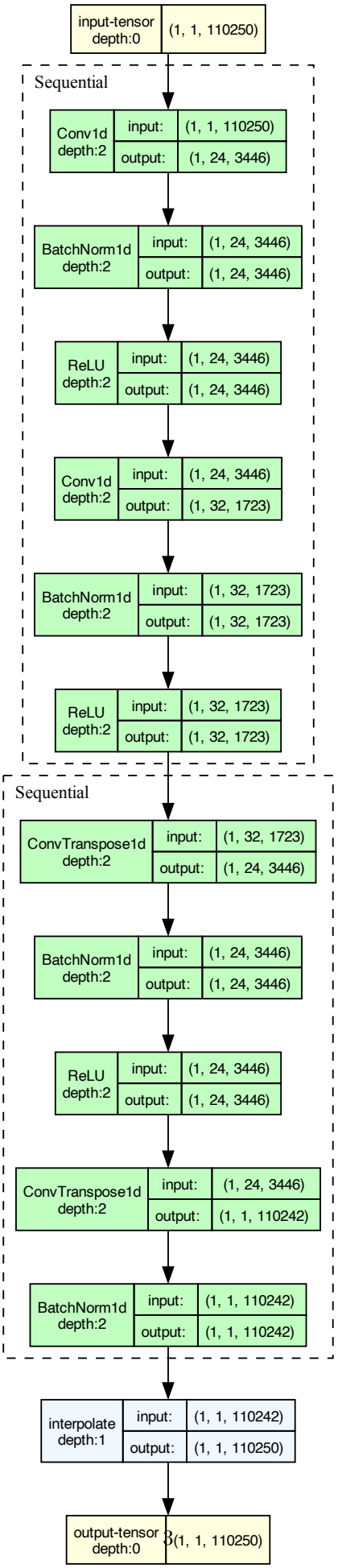
1) Encodear la canción en un vector latente

Comenzamos utilizando la clase brindada en la consigna llamada MusicDataset para cargar y procesar el conjunto de datos de música en formato WAV. Una vez que realizamos esto, separamos los datos en los conjuntos de validación, testeo y entrenamiento. La división que tomamos en cuenta fue fijar los tamaños de los conjuntos de validación y prueba en 100, mientras que el tamaño del conjunto de entrenamiento se calcula para abarcar el resto de los datos. Luego seteamos los DataLoaders respectivos a cada conjunto con un batch_size de 10 y 20.

Con los distintos conjuntos ya creados, comenzamos con la creación e implementación del primer modelo usado. Para la visualización de las canciones, utilizamos los respectivos espectrogramas y formas de onda de cada canción. En primera instancia nos quedamos y basamos nuestro análisis en los espectrogramas. Para esto modificamos el MusicDataset para que por cada canción devuelva tambien un espectograma y un espectograma Mel a partir de las funciones de Torchaudio. Los espectogramas resultantes tenian una resolución de `512x256` por lo que comenzamos utilizando autoencoders cuyas convoluciones eran 2D. Esta idea fue descartada aunque en un comienzo su arquitectura se mantuvo pero con convoluciones 1D.

El primer modelo que utilizamos fue un autoencoder definido en la clase `autoencoder` cuya arquitectura es la siguiente:

Arquitectura Net1



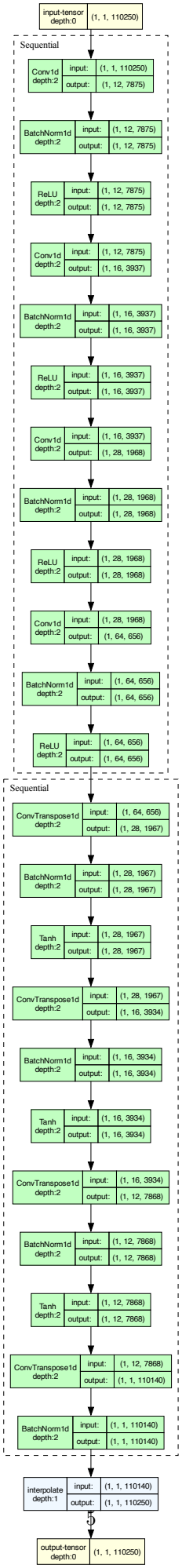
Un problema que tenía este modelo era que al decodear la canción, el tamaño obtenido no era exactamente el mismo que el original, por lo que luego de pasar por el bloque de deconvoluciones, le aplicamos una interpolación al tamaño original.

```
Input Shape      : 110250      torch.Size([1, 1, 110250])
Encoded Shape    : 55136      torch.Size([1, 32, 1723])
Decoded Shape    : 110242      torch.Size([1, 1, 110242])
Output Shape     : 110250      torch.Size([1, 1, 110250])
Encoding Scale: 1/ 1.9996009866511897
```

Este modelo no nos obtuvo buenos resultados, ya que el audio generado salia con un ruido constante de fondo. Inicialmente consideramos podía ser por la interpolación por lo que decidimos cambiar la arquitectura del modelo para que el tamaño de salida del decoder sea el mismo que el de entrada. Para esto, cambiamos el tamaño de los kernels de las convoluciones y deconvoluciones, y agregamos una capa de pooling en el encoder y una de unpooling en el decoder.

El modelo resultante, `autoencoder_alt2`, tiene la siguiente arquitectura:

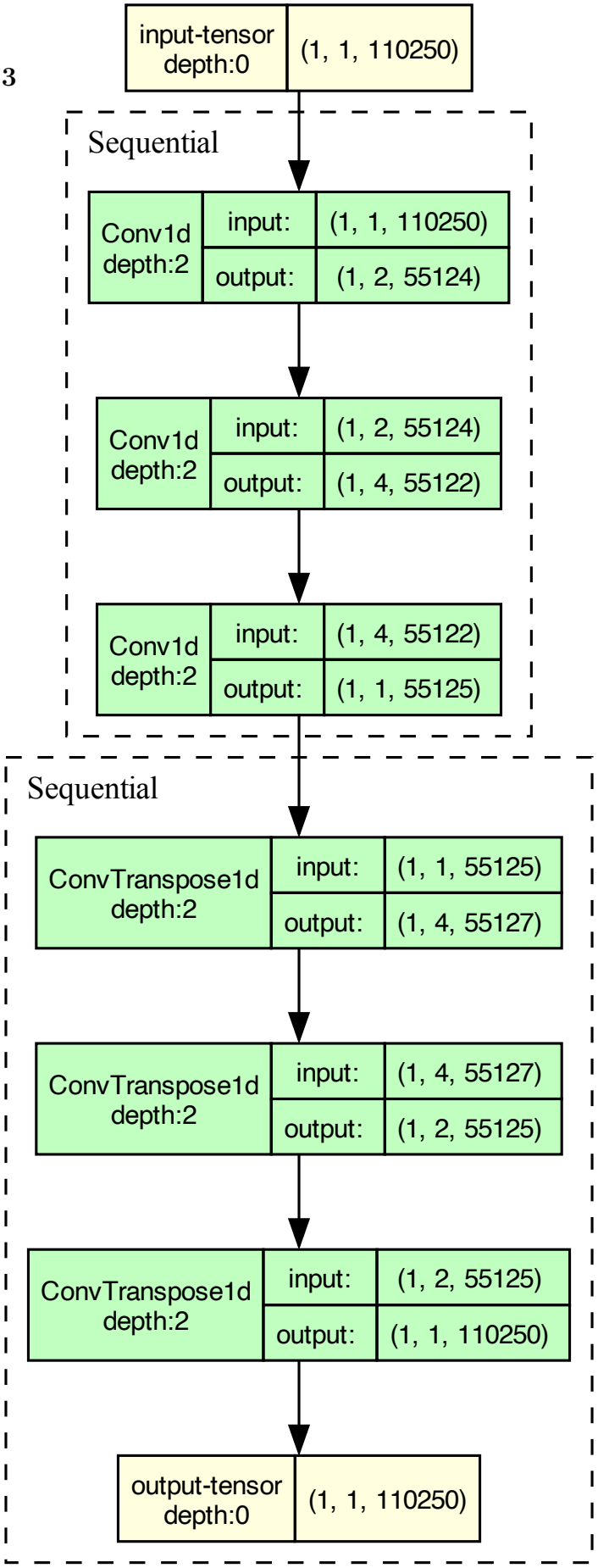
Arquitectura Net2



```
Input Shape      : 110250      torch.Size([1, 1, 110250])
Encoded Shape    : 55216      torch.Size([1, 28, 1972])
Decoded Shape    : 110250      torch.Size([1, 1, 110250])
Output Shape     : 110250      torch.Size([1, 1, 110250])
Encoding Scale: 1/ 1.9967038539553752
```

Este modelo tampoco nos obtuvo buenos resultados. Como no estabamos logrando remover el ruido decidimos comenzar con un modelo desde cero implementado en la clase `autoencoder_alt3`. La arquitectura de este modelo es la siguiente:

Arquitectura Net3



```
Input Shape       : 110250      torch.Size([1, 1, 110250])
 Encoded Shape    : 55125      torch.Size([1, 1, 55125])
 Decoded Shape    : 110250      torch.Size([1, 1, 110250])
 Output Shape     : 110250      torch.Size([1, 1, 110250])
 Encoding Scale: 1/ 2.0
```

Para probar la eficiencia del modelo definimos la función de pérdida a usar, y optamos por "Mean Squared Error". A su vez usamos optimizadores en el proceso de entrenamiento para ajustar los pesos y los sesgos de la red para minimizar la función de pérdida. Para ellos usamos el optimizador Adam. En un comienzo cuando entrenamos el primer autoencoder obtuvimos mejores resultados con Adam que con SGD, por lo que continuamos usandolo para los próximos modelos. Para finetunear la taza de aprendizaje y los epochs corrimos un GridSearch sobre el modelo `autoencoder_alt3`, y a su vez sobre los modelos `autoencoder_alt4` y `autoencoder_alt5` que se explicarán más adelante. El espacio de busqueda que utilizamos es el siguiente:

```
lrs = [0.001, 0.005, 0.01, 0.05, 0.0001, 0.0005]
nums_epochs = [5, 10, 20, 30]
models = [
    autoencoder_alt3,
    autoencoder_alt4,
    autoencoder_alt5
]
```

A su vez se corrió con un scheduler con `step=4` y `gamma=0.5`.

```
Mejor autoencoder_alt3:
    Por Mejor Train Loss: lr=0.01, epochs=30
    Por último Train Loss: lr=0.01, epochs=30
    Por Mejor Validation Loss: lr=0.01, epochs=30
    Por último Validation Loss: lr=0.01, epochs=30
Mejor autoencoder_alt4:
    Por Mejor Train Loss: lr=0.01, epochs=30
    Por último Train Loss: lr=0.01, epochs=30
    Por Mejor Validation Loss: lr=0.01, epochs=30
    Por último Validation Loss: lr=0.01, epochs=30
Mejor autoencoder_alt5:
    Por Mejor Train Loss: lr=0.005, epochs=30
    Por último Train Loss: lr=0.005, epochs=30
    Por Mejor Validation Loss: lr=0.005, epochs=30
    Por último Validation Loss: lr=0.005, epochs=30
```

Después de cada epoch, evalúa el modelo en un conjunto de validación y guarda las pérdidas. También guarda las reconstrucciones de un sample de Test en archivos de Numpy a medida que el modelo se va entrenando para poder visualizar su progresión. Además, utilizamos operaciones como `torch.cuda.empty_cache()`, `torch.mps.empty_cache()`, y `gc.collect()` para gestionar eficientemente la memoria RAM y GPU durante el proceso de entrenamiento.

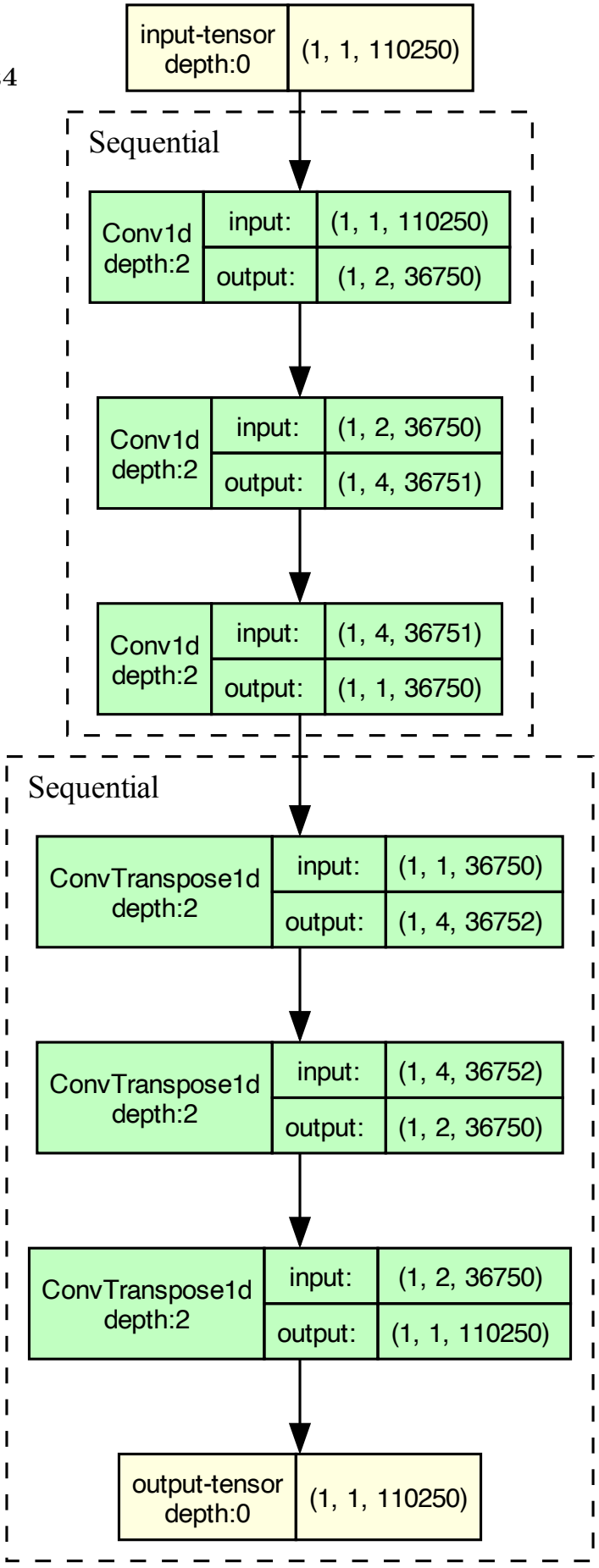
Con el objetivo de encontrar un tamaño de vector óptimo para el análisis, probamos con diferentes configuraciones de autoencoders, que generan salidas de igual tamaño, pero diferentes salidas del autoencoder y del decoder. Estas variaciones se pueden dar por la definición de la arquitectura del autoencoder, ya que la cantidad y tipo de capas puede influir en las variaciones del tamaño de salida. También se pueden dar variaciones por la elección de la dimensionalidad del espacio latente, por las capas de pooling en el encoder usadas o por la definición de los hiper parámetros como stride y padding.

Los 3 modelos con los que obtuvimos mejores rendimientos no cuentan con funciones de activación en el proceso de encodear y desencodear ya que observamos que agregando las mismas, no mejoraba el rendimiento de los modelos. Para llegar a estos partimos de modelos más complejos, con funciones de activación como "ReLU" y "Tahn" y con Batch Normalization, pero fuimos disminuyendo la complejidad de los mismos,

obteniendo ademas del “autoencoer_alt_3” los autoencoders “autoencoder_alt_4” y “autoencoderalt_5” que logran menores tamaños de salida del encoder.

El modelo `autoencoder_alt4` tiene la siguiente arquitectura:

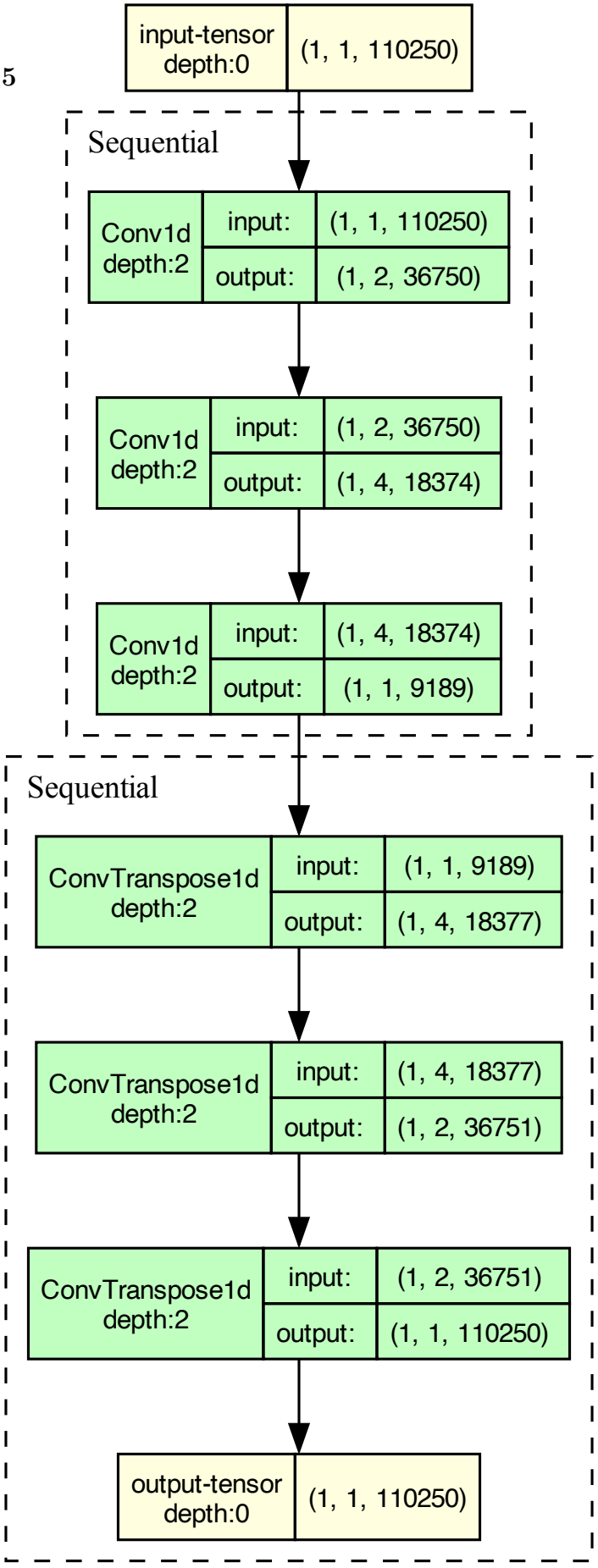
Arquitectura Net4



```
Input Shape      : 110250      torch.Size([1, 1, 110250])
Encoded Shape    : 36750      torch.Size([1, 1, 36750])
Decoded Shape    : 110250      torch.Size([1, 1, 110250])
Output Shape     : 110250      torch.Size([1, 1, 110250])
Encoding Scale: 1/ 3.0
```

El modelo `autoencoder_alt5` tiene la siguiente arquitectura:

Arquitectura Net5



```
Input Shape      : 110250      torch.Size([1, 1, 110250])
Encoded Shape    : 9189      torch.Size([1, 1, 9189])
Decoded Shape    : 110250      torch.Size([1, 1, 110250])
Output Shape     : 110250      torch.Size([1, 1, 110250])
Encoding Scale: 1/ 11.998041136141039
```