Ejemplo de Autoencoder

Stacking Autoencoders

Declaracion de los import

```
from keras.layers import Input, Dense
from keras.models import Model
from keras.datasets import mnist
import tensorflow as tf
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()
```

La ultima linea del import se encarga de descargar el conjunto de datos Mnist, este es devuelto en dos pares un conjunto para el entrenamiento y el otro conjunto de pruebas, en este caso los labels no son necesarios por ende se cargan de manera anonima "_" pero es necesaria para usar la funcion.

Preprocesar Datos

```
x_train = x_train.astype(np.float32) / 255.0
x_test = x_test.astype(np.float32) / 255.0
noise_rate = 0.05
```

Estandarizamos los datos en valores en 0 y 1, ademas se crear una variable de ruido

```
x_train_noisy = x_train + noise_rate * np.random.normal(loc=0.0, scale=1.0, size=x_train.shap
x_test_noisy = x_test + noise_rate * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)
x_train_noisy = np.clip(x_train_noisy, 0.0, 1.0)
x_test_noisy = np.clip(x_test_noisy, 0.0, 1.0)
```

Se indtroduce el ruido en una copia del conjunto de datos, luego de esto se verifica con las ultimas dos lineas que quede estandarizado con valores entre 0 y 1.

Ahora nuestras matrices posee los siguientes valores: (60000, 28, 28) y (10000, 28, 28) en (60000, 784) y (10000, 784) respectivamente.

```
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
```

```
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
x_train_noisy = x_train_noisy.reshape((len(x_train_noisy), np.prod(x_train_noisy.shape[1:])))
x_test_noisy = x_test_noisy.reshape((len(x_test_noisy), np.prod(x_test_noisy.shape[1:])))
assert x_train_noisy.shape[1] == x_test_noisy.shape[1]
```

Se remoldean los conjuntos de datos en las primeras 4 filas, luego se verifica que los conjuntos de datos de "X_train_noisy" y X_test_noisy" poseen el mismo tamaño, ya que de no ser asi la funcion va a crashear. Una vez prerpocesados los datos pasamos a crear el autoencoder.

Creamos el Autoencoder

```
inputs = Input(shape=(x_train_noisy.shape[1],))
encode1 = Dense(128, activation=tf.keras.activations.relu)(inputs)
encode2 = Dense(64, activation=tf.keras.activations.tanh)(encode1)
encode3 = Dense(32, activation=tf.keras.activations.relu)(encode2)
decode3 = Dense(64, activation=tf.keras.activations.relu)(encode3)
decode2 = Dense(128, activation=tf.keras.activations.sigmoid)(decode3)
decode1 = Dense(x_train_noisy.shape[1], activation=tf.keras.activations.relu)(decode2)
```

Se crea la entrada y lka salidad con el "x_train_noisy", y se han creado vairas capas de diferente tamaño ademas de varios activadores, con estos se puede jugar y variarlos segun se desee.

Contruimos el Modelo

```
autoencoder = Model(inputs, decode1)
autoencoder.compile(optimizer="adam", loss='mean_squared_error' , metrics=['accuracy'])
autoencoder.fit(x_train_noisy,x_train_noisy,batch_size=256,epochs=50,shuffle=True)
```

```
Epoch 1/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0547 - accuracy: 0.0
Epoch 2/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0293 - accuracy: 0.0
Epoch 3/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0241 - accuracy: 0.0
Epoch 4/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0216 - accuracy: 0.0
Epoch 5/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0200 - accuracy: 0.0
Epoch 6/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0187 - accuracy: 0.0
Epoch 7/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0178 - accuracy: 0.0
Epoch 8/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0170 - accuracy: 0.0
Epoch 9/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0165 - accuracy: 0.0
```

```
Epoch 10/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0161 - accuracy: 0.0
Epoch 11/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0157 - accuracy: 0.0
Epoch 12/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0153 - accuracy: 0.0
Epoch 13/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0147 - accuracy: 0.0
Epoch 14/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0144 - accuracy: 0.0
Epoch 15/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0142 - accuracy: 0.0
Epoch 16/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0140 - accuracy: 0.0
Epoch 17/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0138 - accuracy: 0.0
Epoch 18/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0136 - accuracy: 0.0
Epoch 19/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0134 - accuracy: 0.0
Epoch 20/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0133 - accuracy: 0.0
Epoch 21/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0132 - accuracy: 0.0
Epoch 22/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0130 - accuracy: 0.0
Epoch 23/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0129 - accuracy: 0.0
Epoch 24/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0128 - accuracy: 0.0
Epoch 25/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0127 - accuracy: 0.0
Epoch 26/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0126 - accuracy: 0.0
Epoch 27/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0125 - accuracy: 0.0
Epoch 28/50
235/235 [==============================] - 3s 11ms/step - loss: 0.0124 - accuracy: 0.0
Epoch 29/50
```

Se entreno el modelo.Ahora vamos a evaluar y predecir el modelo.

Evaluacion del Modelo

```
metrics = autoencoder.evaluate(x_test_noisy, x_test, verbose=1)
print()
print("%s:%.2f%%" % (autoencoder.metrics_names[1], metrics[1]*100))
print()
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0102 - accuracy: 0.0128
```

```
accuracy:1.28%
```

Predecimos el Modelo

```
results = autoencoder.predict(x_test)
all_AE_weights_shapes = [x.shape for x in autoencoder.get_weights()]
print(all_AE_weights_shapes)
ww=len(all_AE_weights_shapes)
deeply_encoded_MNIST_weight_matrix = autoencoder.get_weights()[int((ww/2))]
print(deeply_encoded_MNIST_weight_matrix.shape)
autoencoder.save_weights("all_AE_weights.h5")
```

```
[(784, 128), (128,), (128, 64), (64,), (64, 32), (32,), (32, 64), (64,), (64, 128), (128
(32, 64)
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

La matriz resultante se almacena en la variable "deep_encoded_MNIST _weight_matrix", la cual contiene los pesos entrenados para la capa intermedia del codificador automático apilado, y esto luego debe alimentarse a un codificador completamente conectado red neuronal junto con las etiquetas (las que descartamos). Esta matriz de peso es una representación distribuida del conjunto de datos original. También se incluye una copia de todos los pesos. guardado para su uso posterior en un archivo H5. También se ha añadido una variable resultados para hacer predicciones con el codificador automático.

✓ 0s completed at 10:56 AM