

Módulo 2: Una aplicación útil de Deep learning en finanzas

Dra. Patricia Kisbye, FaMAF.

Dr. Gustavo Gianotti, Grupo del Plata S.A.

30 de agosto de 2018

Tabla de contenidos

- 1 Series temporales
- 2 Keras
- 3 Autoencoders y aplicación a series temporales

Series temporales

Series temporales

Una serie temporal $\{S_i\}$ es una serie de datos indexados temporalmente. Durante el curso nos concentraremos en datos discretos.

El orden temporal permite el estudio de predicción y pronóstico de series temporales. Abarcaremos este tema en el siguiente capítulo.

Métricas usuales

En el area de las series temporales las dos métricas usuales son:

- Error cuadrático medio (ECM): Esta métricas mide el promedio de los errores al cuadrado, es decir, la diferencia entre el predictor y lo que se estima. Si tenemos una muestra de m vectores n dimensionales Y , e \hat{Y} sus predicciones, entonces definimos:

$$ECM = \frac{1}{mn} \sum_{i,j=0}^{mn-1} (\hat{Y}(i)_j - Y(i)_j)^2$$

Es muy utilizado en el área de Machine Learning por la simpleza de su cómputo.

- Coeficiente de determinación R^2 :

$$R^2 = 1 - \frac{\sum_{i,j=0}^{mn-1} (Y(i)_j - \hat{Y}(i)_j)^2}{\sum_{i,j=0}^{mn-1} (\bar{Y}_i - Y(i)_j)^2},$$

donde $\bar{Y}_i = \frac{1}{m} \sum_{i=0}^{m-1} Y(i)$ Suele ser mas usado en estadística y Statistical Learning.

Validación cruzada en series temporales

Recordemos que la validación cruzada es una herramienta poderosa en Machine Learning para tunear hiperparámetros de modelos y analizar el poder predictivo de estos, siendo k-fold una de las mas utilizadas. Ej.:

```
X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])  
kf = KFold(n_splits=3)  
for train_index, test_index in kf.split(X):  
    print("TRAIN_INDEX:", train_index, "TEST_INDEX:", test_index)
```

```
TRAIN_INDEX: [2 3] TEST_INDEX: [0 1]  
TRAIN_INDEX: [0 1 3] TEST_INDEX: [2]  
TRAIN_INDEX: [0 1 2] TEST_INDEX: [3]
```

En una serie temporal la estructura temporal es relevante, no suele ser útil entrenar en datos posteriores para predecir datos anteriores. Menos aún entrenar en datos anteriores y posteriores para predecir datos intermedios. Generalmente porque no es como suelen usarse los modelos llevados a la realidad.

En una validación cruzada utilizando k-fold es fácil encontrar casos como los anteriores descriptos. Ej.:

- Entrenar en datos posteriores para predecir datos anteriores:

```
TRAIN_INDEX: [2 3] TEST_INDEX: [0 1]
```

- Entrenar en datos anteriores y posteriores para predecir datos intermedios:

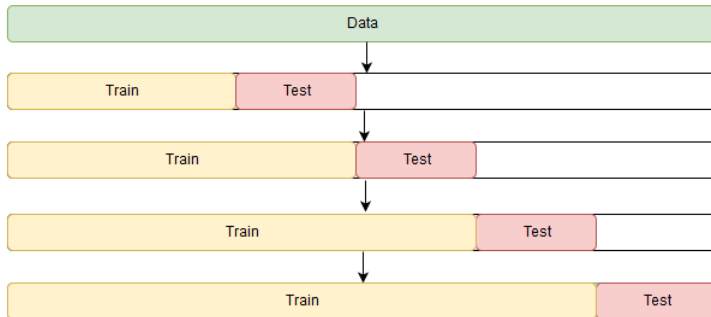
```
TRAIN_INDEX: [0 1 3] TEST_INDEX: [2]
```


Time Series Split es un método de validación cruzada que contempla la estructura temporal de las series temporales.

Se define de la siguiente forma, supongamos que tenemos una muestra de m elementos, y decidimos separar en n segmentos $\{\mathbf{S}_i\}$. Entonces se generaran n segmentos ordenados disjuntos donde todos (salvo el primero) tendrán $\lfloor \frac{m}{n} \rfloor$ elementos, siendo que el primero tendrá los primeros $m - (n - 1) \lfloor \frac{m}{n} \rfloor$ elementos. Luego se prosigue realizar n train y test de la siguiente forma:

$$j\text{-ésimo Train: } \bigcup_{i=0}^j \mathbf{S}_i, j\text{-ésimo Test: } \mathbf{S}_{j+1}$$

Gráficamente sería:



Con un ejemplo:

```
X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [5, 2]])  
tscv = TimeSeriesSplit(n_splits=3)  
for train_index, test_index in tscv.split(X):  
    print("TRAIN:", train_index, "TEST:", test_index)
```

TRAIN: [0 1] TEST: [2]

TRAIN: [0 1 2] TEST: [3]

TRAIN: [0 1 2 3] TEST: [4]

Con un ejemplo práctico:

```
XGBR = XGBRegressor(max_depth=2, learning_rate=1, n_estimators=1)
tscv = TimeSeriesSplit(n_splits=3)
scores = []
for train_index, test_index in tscv.split(df.index):
    XGBR.fit(df[['GOOGL', 'AMZN', 'MSFT']].iloc[train_index], df[['AAPL']].iloc[train_index])
    prediction = XGBR.predict(df[['GOOGL', 'AMZN', 'MSFT']].iloc[test_index])
    scores.append(mean_squared_error(df[['AAPL']].iloc[test_index], prediction))
scores
```

[214.665951163599, 1276.1347327550568, 2209.1440461367065]

Keras

¿Que es y por qué Keras?



¿Que es?:

- Es una librería de Redes Neuronales escrita en Python.
- Diseñada de forma minimalista y directa.
- Construida sobre Theano, TensorFlow y mas recientemente CNTK.

¿Por qué Keras?:

- Simple para empezar y simple para continuar.
- Permite rapidez para crear prototipos.
- Permite cómputo tanto en CPU como GPU
- Soporta modelos con redes convolucionales, recurrentes o ambas.
- Escrita en python, muy modular y fácil de extender (y conseguir extensiones, como un sk-learn wrapper, keras-rl y mucho más...).
- Suficientemente poderoso como para construir modelos serios de redes neuronales y llevar a producción. Incluso Deep Learning.

Diseño general: La idea general se basa en capas, sus inputs y outputs:

- Preparar tus datos de entrada y salida del modelo (X e Y).
- Crear primera capa con las dimensiones correspondientes a los Inputs.
- Crear la última capa con las dimensiones correspondientes a los Outputs.
- Construir cualquier modelo serio de redes neuronales e incluso llevar a producción.

El backend se puede cambiar fácilmente accediendo y modificando el .json de configuración:

- windows: `C : /users/(username)/.keras/keras.json`.
- unix: `~ /.keras/keras.json`

Simplemente hay que cambiar el .json en la llave *backend*.

Algunas capas que encontramos:

En Keras encontraremos grandes variedades de capas:







- Densas (MLP): Capa estandar de redes neuronales.
 $output = act_func(input \bullet kernel + bias)$
- Dropout: Consiste en establecer aleatoriamente unidades de entrada en 0 en cada actualización **durante el tiempo de entrenamiento**, lo que ayuda a prevenir el overfitting.
- Noise: Arbitrariamente se suma ruido a las unidades de entrada **durante el tiempo de entrenamiento**, lo que ayuda a prevenir el overfitting.



- MaxPooling: Consiste en tomar solo la unidad de entrada con mayor valor.
- RNN: RNN, LSTM, GRU, etc.: Son redes que toman como inputs a estados anteriores y unidades de entrada. Utilizadas en series temporales, reconocimiento de lenguaje entre otros.
- CNN: Conv1D, Conv2D, etc.: Procesan por sectores cada unidades de entrada. Basadas en la corteza visual. Utilizadas en procesamiento de imágenes, reconocimiento de lenguaje entre otros.
- Y mucho mas.

Algunas funciones de activación:

En keras las funciones de activación se pueden tanto agregar como parámetro de algunas capas (Densas, RNN, CNN), así como capa individual.

Algunas funciones de activación clásicas en Keras:

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$

Name	Plot	Equation	Derivative
Parameteric Rectified Linear Unit (PreLU)		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$

Remarcamos que PreLU y LeakyReLU (una versión menos general de PreLU) solo se pueden utilizar como capas de activación pero no como parámetro de otras capas (Densas, RNN, CNN). Aclaremos mas posteriormente.

Algunas funciones de pérdida, métricas y algunos optimizadores en keras:

- La función de pérdida es la función objetivos a minimizar sobre el conjunto de entrenamiento.
- El algoritmo de optimización es el procedimiento por el cual buscaremos un mínimo local en la función de pérdida.
- La métricas es una función con la cual se evalúa el desempeño del modelo. Puede como no coincidir con la función de pérdida.

Algunas funciones objetivos a minimizar y/o métricas en Keras:

- Error cuadrático medio: Métrica y función de pérdida. Mas es peor.
- Error absoluto medio: Métrica y función de pérdida. Mas es peor.
- hinge: Supongamos que y_p es la predicción de un modelo categorico m en un punto x e y_t su verdadera categoría en ± 1 . Entonces definimos la pérdida del modelo m en el punto x como:

$$l(x) = \max(0, 1 - y_t y_p).$$

La métrica de hinge no es mas que el promedio de dichas perdidas en un conjunto. Métrica y función de pérdida. Mas es peor.

- Y mucho mas.

En cuanto a optimizadores:

- provee: SGD, Adagrad, Adadelata, Rmsprop y Adam.
- Todos estos algoritmos se puede setear con hiperpcarametros.

Modelos Secuenciales:

La forma mas simple de construir modelos en Keras son los modelos secuenciales.

Son una pila de capas (u otros modelos en Keras) donde el output de una capa es el input de la siguiente. Sería un pipeline de capas (layers) o modelos de Keras.

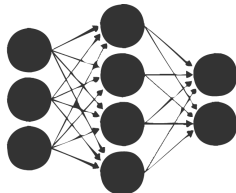
Para modelos con arquitecturas más complejas se debe usar la API funcional de Keras, que permite construir configuraciones arbitrarios de capas (otra sección).

Primero creamos una instancia de la clase de modelo secuencial, después apilamos las capas con `.add()` y para terminar solo hay que compilarlo:

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(units=3, activation='relu', input_dim=10))
model.add(Dense(units=4, activation='sigmoid'))
model.add(Dense(units=2, activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

Gráficamente es un simple Multi Layer perceptron:



La compilación recibe tres argumentos:

- Un optimizador. String que de referencia a un optimizador existente (como *'rmsprop'* o *'adagrad'*) o una instancia de *Optimizer*.
- Una función de pérdida. Recordamos que este es el objetivo que el modelo intentará minimizar. String que de referencia a una función de pérdida (como *'mse'*), o una función objetivo.
- Una lista con métricas (opcional). Las métricas infieren en el entrenamiento, es solo para su evaluación posterior. Nuevamente, puede ser o un sting o una métrica.

Dos formas equivalentes, siendo que la segunda permite cambiar hiper parámetros:

```
model.compile(loss='categorical_crossentropy',  
              optimizer='sgd',  
              metrics=['accuracy'])  
  
model.compile(loss=keras.losses.categorical_crossentropy,  
              optimizer=keras.optimizers.SGD(lr=0.01, momentum=0.9, nesterov=True),  
              metrics=[keras.metrics.categorical_accuracy])
```

Ejemplos de clasificador implementado con modelo secuencial:

```
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.layers import LeakyReLU
from keras.optimizers import SGD

model = Sequential()
model.add(Dense(512, input_shape=(784,)))
model.add(Activation(LeakyReLU(alpha=0.3)))
model.add(Dropout(0.2))
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(10))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy', optimizer=SGD(),
              metrics=['accuracy'])
```

* Recordamos que Dropout consiste en establecer aleatoriamente unidades de entrada en 0 en cada actualización **durante el tiempo de entrenamiento**, lo que ayuda a prevenir el overfitting.

En Keras hay muchas formas de escribir lo mismo.

Por ejemplo remarcamos que agregar a *model* la siguiente capa:

```
model.add(Dense(units=4, activation='relu'))
```

Es equivalente a:

```
model.add(Dense(4))  
model.add(Activation('relu'))
```

Por default omitir la activación en cualquier capa que lo permita, se usará la identidad ($a(x) = x$).

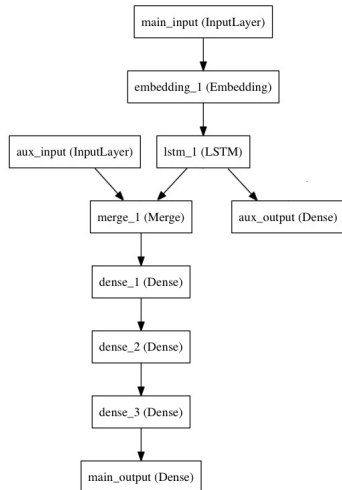
Y se puede volver mas complicado. Este es un ejemplos de clasificador de imágenes implementado con modelo secuencial:

```
model = Sequential()  
# input: 100x100 images with 3 channels -> (100, 100, 3) tensors.  
# this applies 32 convolution filters of size 3x3 each.  
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(100, 100, 3)))  
model.add(Conv2D(32, (3, 3), activation='relu'))  
model.add(MaxPooling2D(pool_size=(2, 2)))  
model.add(Dropout(0.25))  
  
model.add(Conv2D(64, (3, 3), activation='relu'))  
model.add(Conv2D(64, (3, 3), activation='relu'))  
model.add(MaxPooling2D(pool_size=(2, 2)))  
model.add(Dropout(0.25))  
  
model.add(Flatten())  
model.add(Dense(256, activation='relu'))  
model.add(Dropout(0.5))  
model.add(Dense(10, activation='softmax'))  
  
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)  
model.compile(loss='categorical_crossentropy', optimizer=sgd)
```

Modelos Funcionales:

La API funcional de Keras permite definir modelos complejos, como modelos de múltiples salidas, múltiples inputs en distintas capas, modelos con capas compartidas, etc.

Se puede concatenar fácilmente redes independientes así como capas.



Los dos modelos siguientes son equivalentes, uno en forma secuencial:

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=784))
model.add(Dense(32, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

y otro en forma funcional:

```
from keras.layers import Input, Dense
from keras.models import Model

inputs = Input(shape=(784,))
x = Dense(32, activation='relu')(inputs)
x = Dense(32, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)
model = Model(inputs=inputs, outputs=predictions)
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```


A la hora de definir un modelo funcional hay que:

- Primero definir la (o las) capa *Input(shape = ...)* que transforman los inputs en tensores.
- Se señala entre paréntesis los inputs de cada capa al final de estas mismas, mientras que los outputs se anteponen a las capas con un igual.
- Se termina definiendo el modelo con *Model()* donde se señala los inputs y los outputs.

```
from keras.layers import Input, Dense
from keras.models import Model

inputs = Input(shape=(784,))
x = Dense(32, activation='relu')(inputs)
x = Dense(32, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)
model = Model(inputs=inputs, outputs=predictions)
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

Y la gran flexibilidad permite definir modelos muy complejos. El siguiente modelo trata de predecir la cantidad de retweets y likes que puede tener una noticia en Twitter:

```
main_input = Input(shape=(100,), dtype='int32', name='main_input')
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)
lstm_out = LSTM(32)(x)
auxiliary_output = Dense(1, activation='sigmoid', name='aux_output')(lstm_out)

auxiliary_input = Input(shape=(5,), name='aux_input')

x = keras.layers.concatenate([lstm_out, auxiliary_input])
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)

main_output = Dense(1, activation='sigmoid', name='main_output')(x)
model = Model(inputs=[main_input, auxiliary_input], outputs=[main_output, auxiliary_output])
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
              loss_weights=[1., 0.2])
```

Remarcamos que, como en el ejemplo, a cada output se le puede dar pesos distintos en la función de pérdida. En este ejemplo el *errpr* de *main_output* se encuentra multiplicado por 1, mientras que el de *auxiliary output* por 0.2.

También se puede dar distintas funciones de pérdida o combinaciones de pesos y funciones de pérdida.

En el siguiente ejemplo, con solo ver la compilación nos damos cuenta que es un modelo funcional con dos capas generando output, las cuales se concatenan en la definición del output del modelo funcional *model()*. Al primer output se le aplica la función *categorical_crossentropy*, al segundo *center_loss*.

```
model.compile(optimizer='sgd',  
              loss=['categorical_crossentropy', 'center_loss'],  
              metrics=['accuracy'], loss_weights=[1., 0.2])
```

Por último nos queda remarcar las operaciones para convinar capas, que son muy útiles. Concatenar:

```
encoded_video = LSTM(256)(encoded_frame_sequence)
encoded_video_question = Dense(64, activation='relu')(video_question_input)
merged = keras.layers.concatenate([encoded_video, encoded_video_question])
output = Dense(1000, activation='softmax')(merged)
```

A esta podemos agregar otras capas que mezclan:

- *add*([...]): suma capas.
- *subtract*([...]): resta capas.
- *multiply*([...]): multiplica capas.
- *average*(*input*): promedia de una capa.
- *maximum*(*input*): máximo valor de una capa.
- *dot*([...]): producto escalar entre dos tensores.

Entrenamientos de modelos:

Una vez definido nuestro modelo (secuencial o funcional) se puede proseguir a entrenarlo, simplemente haciendo:

```
model.fit(X_train, Y_train)
```

Pero... las cosas se pueden complicar:

```
network_history = model.fit(X_train, Y_train, batch_size=128,  
                             epochs=4, verbose=1, validation_data=(X_val, Y_val),  
                             callbacks=[Early_Stop])
```

Train on 45000 samples, validate on 15000 samples

Epoch 1/4

45000/45000 [=====] - 2s - loss: 1.3746 - acc: 0.6348 - val_loss: 0.6917 - val_acc: 0.8418

Epoch 2/4

45000/45000 [=====] - 2s - loss: 0.6235 - acc: 0.8268 - val_loss: 0.4541 - val_acc: 0.8795

Epoch 3/4

45000/45000 [=====] - 1s - loss: 0.4827 - acc: 0.8607 - val_loss: 0.3795 - val_acc: 0.8974

Epoch 4/4

45000/45000 [=====] - 1s - loss: 0.4218 - acc: 0.8781 - val_loss: 0.3402 - val_acc: 0.9055

Respecto a las opciones en fit:

- epochs y batch_size son hiper parámetros a configurar.
- Para validación tenemos los siguientes campos: validation_split, validation_data. Remarcamos que la validación solo sirve para evaluar con métricas después de cada epoch, pero no en el entrenamiento. Las métricas y la función de pérdida aparecerán en el historial con el antefijo *val_*, ejemplo: *loss* es la función de pérdida en train, mientras que *val_loss* en el conjunto de validación.
- Pesos en clases y en muestras: class_weight, sample_weight.
- shuffle: importante que se encuentre en False para time series.
- callbacks: Lista de funciones que se llaman durante el entrenamiento de la red.

WARNING: cada vez que se entrena un modelo se retoma desde donde se dejó. Es decir que el modelo no se inicializa como nuevo antes de hacer un *model.fit(...)*. Esto claramente influye cuando se entrena varias veces un modelo. Este punto es importante a la hora de buscar hiper parámetros.

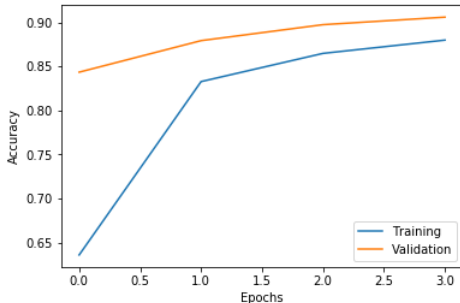
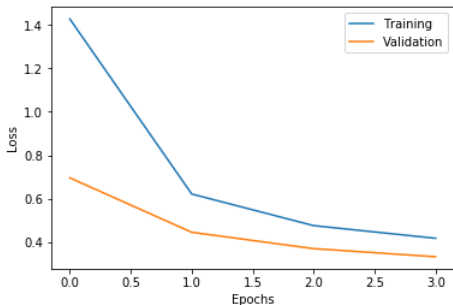
`model.fit(...)` devuelve un objeto de la clase *History*. Si hacemos:

```
network_history = model.fit(X_train, Y_train, batch_size=128,  
                             epochs=4, verbose=1, validation_data=(X_val, Y_val), shuffle=False)
```

El objeto `network_history` (instancia de *History*) tendrá, entre otros, los siguientes atributos:

- `network_history.history`: Diccionario con la historia del entrenamiento y de evaluación con las métricas y la función de pérdida en el conjunto de validación. Recordamos que los resultados de las métricas y la función de pérdida sobre el conjunto de validación se diferencian de los de entrenamiento por tener el prefijo `val_`
- `network_history.epoch`: lista de epochs
- `network_history.model`: modelo.
- `network_history.params`: Parámetros de entrenamiento.
- `network_history.validation_data`: Conjunto de datos utilizados para la validación.

Puede resultar útil graficar la información disponible en el atributo history:



Uso de modelos:

Para usar el modelo simplemente hay que llamar dos métodos:

- **predict:** Usa el modelo para predecir valores.

```
y_ = model.predict(X_val)
```

- **evaluate:** Evalúa en las métricas y la función de pérdida.

```
model.evaluate(X_val, Y_val)
```

```
15000/15000 [=====] - 2s 105us/step
```

```
[0.34249946967760719, 0.90380000003178917]
```

Inspección de modelos:

Se puede realizar una primera inspección a un modelo en Keras con:

```
from keras.models import Sequential
from keras.layers.core import Dense
from keras.optimizers import SGD

model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer=SGD(lr=0.001),
              metrics=['accuracy'])
```

```
model.summary()
```

Layer (type)	Output Shape	Param #
dense_22 (Dense)	(None, 512)	401920
dense_23 (Dense)	(None, 10)	5130

=====
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0

Las capas de los modelos son iterables:

```
print('Model Input Tensors: ', model.input, end='\n\n')
print('Layers - Network Configuration:', end='\n\n')
for layer in model.layers:
    print(layer.name, layer.trainable)
    print('Layer Configuration:')
    print(layer.get_config(), end='\n{}\n'.format('----'*10))
print('Model Output Tensors: ', model.output)
```

Model Input Tensors: Tensor("dense_22_input:0", shape=(?, 784), dtype=float32)

Layers - Network Configuration:

dense_22 True

Layer Configuration:

```
{'name': 'dense_22', 'trainable': True, 'batch_input_shape': (None, 784), 'dtype': 'float32', 'units': 512, 'activation': 'relu', 'use_bias': True, 'kernel_initializer': {'class_name': 'VarianceScaling', 'config': {'scale': 1.0, 'mode': 'fan_avg', 'distribution': 'uniform', 'seed': None}}, 'bias_initializer': {'class_name': 'Zeros', 'config': {}}, 'kernel_regularizer': None, 'bias_regularizer': None, 'activity_regularizer': None, 'kernel_constraint': None, 'bias_constraint': None}
```

dense_23 True

Layer Configuration:

```
{'name': 'dense_23', 'trainable': True, 'units': 10, 'activation': 'softmax', 'use_bias': True, 'kernel_initializer': {'class_name': 'VarianceScaling', 'config': {'scale': 1.0, 'mode': 'fan_avg', 'distribution': 'uniform', 'seed': None}}, 'bias_initializer': {'class_name': 'Zeros', 'config': {}}, 'kernel_regularizer': None, 'bias_regularizer': None, 'activity_regularizer': None, 'kernel_constraint': None, 'bias_constraint': None}
```

Model Output Tensors: Tensor("dense_23/Softmax:0", shape=(?, 10), dtype=float32)

Además se pueden extraer los pesos (kernes y bias):

```
model.get_layer('dense_22').get_weights()  
  
[array([[ 0.02572416, -0.02885175,  0.02570447, ...,  0.03771176,  
         0.03179846, -0.04697915],  
       [ 0.00162984, -0.02235272, -0.0122257 , ..., -0.00062346,
```

y settear:

```
model.get_layer('dense_22').set_weights(array_weights)
```

También se puede grabar y cargar configuraciones (sin pesos generados por entrenamiento):

```
import json
json_to_save = model.to_json()
with open('model.json', 'w') as outfile:
    json.dump(json_to_save, outfile)
```

```
import json
from keras.models import model_from_json

with open('model.json') as json_file:
    json_from_file = json.load(json_file)
model = model_from_json(json_from_file)
```

y pesos del entrenamiento:

```
model.save_weights('model_weights.h5')
```

```
model.load_weights('model_weights.h5')
```

Capas escondidas y reentrenamiento de capas:

La mayoría de las capas de Keras se pueden dar nombre específico, tanto en los modelos secuenciales como en los funcionales:

```
from keras.models import Sequential
from keras.layers.core import Dense
from keras.optimizers import SGD

model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,), name = 'Capa_Densa_1'))
model.add(Dense(512, activation='relu', name = 'Capa_Densa_2'))
model.add(Dense(10, activation='softmax', name = 'Capa_Densa_3'))
model.compile(loss='categorical_crossentropy', optimizer=SGD(lr=0.001),
              metrics=['accuracy'])

model.summary()
```

Layer (type)	Output Shape	Param #
Capa_Densa_1 (Dense)	(None, 512)	401920
Capa_Densa_2 (Dense)	(None, 512)	262656
Capa_Densa_3 (Dense)	(None, 10)	5130
Total params: 669,706		
Trainable params: 669,706		
Non-trainable params: 0		

Eso nos permite extraer una capa puntual:

```
model.get_layer('Capa_Densa_2')
```

También se puede renombrar a una ya existente:

```
model.get_layer('Capa_Densa_2').name = 'capa_2'  
model.summary()
```

Layer (type)	Output Shape	Param #
Capa_Densa_1 (Dense)	(None, 512)	401920
capa_2 (Dense)	(None, 512)	262656
Capa_Densa_3 (Dense)	(None, 10)	5130

=====
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
=====

Dado un modelo ya definido y entrenado, se puede crear uno nuevo que culmine en una capa escondida:

```
model.summary()
```

Layer (type)	Output Shape	Param #
Capa_Densa_1 (Dense)	(None, 512)	401920
capa_2 (Dense)	(None, 512)	262656
Capa_Densa_3 (Dense)	(None, 10)	5130
Total params: 669,706		
Trainable params: 669,706		
Non-trainable params: 0		

```
from keras.models import Model
intermediate_layer_model = Model(model.layers[0].input, model.get_layer('capa_2').output)
intermediate_layer_model.summary()
```

Layer (type)	Output Shape	Param #
Capa_Densa_1_input (InputLayer)	(None, 784)	0
Capa_Densa_1 (Dense)	(None, 512)	401920
capa_2 (Dense)	(None, 512)	262656
Total params: 664,576		
Trainable params: 664,576		
Non-trainable params: 0		

Muchas veces es útil congelar ciertas capas de un modelo y reentrenar algunas otras. a continuación damos un ejemplo de como definir un nuevo modelo donde solo algunas capas sean reentrenables y otras estén congeladas:

```
model.summary()
```

Layer (type)	Output Shape	Param #
Dense_1 (Dense)	(None, 512)	401920
Dense_2 (Dense)	(None, 512)	262656
Total params: 664,576		
Trainable params: 664,576		
Non-trainable params: 0		

```
from keras.models import Model
model.get_layer('Dense_1').trainable = False
part_frozen_model = Model(model.layers[0].input, model.layers[-1].output)
part_frozen_model.compile(loss='categorical_crossentropy', optimizer=SGD(lr=0.001),
                          metrics=['accuracy'])
part_frozen_model.summary()
```

Layer (type)	Output Shape	Param #
Dense_1_input (InputLayer)	(None, 784)	0
Dense_1 (Dense)	(None, 512)	401920
Dense_2 (Dense)	(None, 512)	262656
Total params: 664,576		
Trainable params: 262,656		
Non-trainable params: 401,920		

Scikit-Learn Wrappers:

Incluyeron en Keras un complemento que permite portar un modelo secuencial a Scikit-Learn, tanto regresores como clasificadores. Esto permite convinarlo en un pipeline, o hacer un gridsearch. De usar el Scikit-Learn Wrappers, es más común usar KerasRegressor para series temporales en finanzas.

Los hiper parámetros del modelo así como el epoch y batch_size pasan a ser hiper parámetros del modelo resultante del Scikit-Learn Wrappers.

Ejemplo 1:

```
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV

def create_model(ne_dim=8, optimizer='rmsprop', init='glorot_uniform'):
    model = Sequential()
    model.add(Dense(ne_dim, input_dim=8, kernel_initializer=init, activation='relu'))
    model.add(Dense(8, kernel_initializer=init, activation='relu'))
    model.add(Dense(1, kernel_initializer=init, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model

model = KerasClassifier(build_fn=create_model, verbose=0)
ne_dim=[8, 12, 15]
optimizers = ['rmsprop', 'adam']
init = ['glorot_uniform', 'normal', 'uniform']
epochs = [50, 100, 150]
batches = [5, 10, 20]
param_grid = dict(optimizer=optimizers, epochs=epochs, batch_size=batches, init=init)
grid = GridSearchCV(estimator=model, param_grid=param_grid)
grid_result = grid.fit(X, Y)
```

Si vamos a definir un modelo para entrenar directamente, se pasan los argumentos a la hora de definirlo. Ejemplo 2:

```
from keras.models import Sequential
from keras.layers.core import Dense
from keras.optimizers import SGD
def create_model(ne_dim=512):
    model = Sequential()
    model.add(Dense(512, activation='relu', input_shape=(784,), name = 'Capa_Densa_1'))
    model.add(Dense(512, activation='relu', name = 'Capa_Densa_2'))
    model.add(Dense(10, activation='softmax', name = 'Capa_Densa_3'))
    model.compile(loss='categorical_crossentropy', optimizer=SGD(lr=0.001),
                  metrics=[categorical_accuracy])

    return model

model_comp = KerasClassifier(create_model, ne_dim = 256, epochs=2, batch_size=128, verbose=0)
model_comp.fit(X_train, Y_train)
```

Importante:

- El `.fit(...)` de un modelo tipo `KerasRegressor` o `KerasClassifier` (Scikit-Learn Wrappers) siempre entrena un modelo inicializado nuevo, a diferencia de un `.fit(...)` de `keras` que entrena desde donde quedo el último `.fit`.
- El `.fit(...)` de un modelo de Scikit-Learn Wrappers retorna un objeto de la clase `History` al igual que el `.fit(...)` de `keras`.

WARNING:

Algunas versiones de Keras en conjunto con tensorflow tienen un meamory leak (como Keras 2.2.2 y Tensorflow 1.8), lo que provoca que no se puede hacer un GridSearchCV por ejemplo. Algunas opciones para solucionar este problema:

- Usar Theano o CTNK como backend o una versión de Keras y Tensorflow sin ese problema.
- Usar loops anidados en lugar de GridSearchCV (o similares) y al final de cada iteración agregar:

```
from keras import backend as K
import gc
import tensorflow as tf

for ....:
    for ...:
        for ...:
            .
            .
            .
            .
            del model
            K.clear_session()
            tf.reset_default_graph()
            tf.contrib.keras.backend.clear_session()
            sess = tf.Session()
            K.set_session(sess)
            gc.collect()
```





hiper parametrización:

Como sabemos lo usual en ML es hacer alguna búsqueda en cuadrícula en conjunto con un kfold para definir los hiper parámetros. Como ya remarcamos, en una serie termpporal el kfold deberá ser con Time Series Split.

Respecto a la función de activación:

Hoy en día el estándar es usar ReLU o alguna variante (como LeakyReLU o PReLU) para todas las capas ocultas y las capas de output usan softmax para un clasificador y lineal para regresión. Esto significa que a priori las activaciones ya deberían estar definidas. Hay dos artículos que son de referencia en el tema:

- Rectified Linear Units Improve Restricted Boltzmann Machines
- ImageNet Classification with Deep Convolutional Neural Networks

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parametric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Respecto al `batch_size`:

En el estudio **On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima** recomiendan un batch size entre 32 y 512.

Remarcamos el el default de Keras en 32 en caso de omitir este parámetro.

Respecto a los epochs:

Se puede utilizar *EarlyStopping*, el cual es un callback que frena el entrenamiento cuando no hay mejora. Donde tomas los siguientes argumentos:

- **monitor:** String que indica la función que se desea monitorear para frenar en caso de no haber mejoras. ej: 'loss', 'val_loss' (si .fit cuenta con conjunto de validación), 'acc' (solo en caso de ser una métrica presente durante la compilación), 'val_acc', etc.
- **min_delta:** Si la mejora del monitor es menor que min_delta frena el entrenamiento.
- **patience:** Cantidad de epoch para que se frene el entrenamiento si no ha habido mejoras.

El uso de *EarlyStopping* puede ser delicado, a priori no es buena práctica usarlo para parar según el el conjunto de validación.

Tomemos este modelo:

```
def create_model(dim_neuron):  
  
    input_layer = Input(shape=(8, ))  
    x = Dense(dim_neuron)(input_layer)  
    xx = LeakyReLU(alpha=0.3)(x)  
    output_model = Dense(1)(xx)  
  
    model = Model(input_layer, outputs=output_model)  
  
    # Compile model  
    rms = RMSprop(decay=0.0001)  
    model.compile(loss='mean_squared_error', optimizer=rms)  
    return model
```

Podríamos hacer búsqueda en cuadrícula con el Scikit-Learn Wrapper para series temporales (posible memory leak):

```
sk_model = KerasRegressor(create_model)  
param_grid = {'dim_neuron':[2, 4, 8, 16, 32], 'epochs':[20, 25, 30, 35]}  
my_cv = TimeSeriesSplit(n_splits=3).split(df.index)  
grid_model = GridSearchCV(estimator=sk_model, cv=my_cv, param_grid=param_grid)  
grid_result = grid_model.fit(X.values, Y.values)
```

O haciendo la búsqueda manual:

```
neurons = [2, 4, 8, 16, 32]
Epochs = [20, 25, 30, 35]
best_score = 1000000
print('neuron amm', ' ', 'epochs', ':', 'max_score')
for n in neurons:
    for epoch in Epochs:
        tscv = TimeSeriesSplit(n_splits=3)
        scores = []
        for train_index, test_index in tscv.split(X.index):
            model = create_model(dim_neuron=n)
            h = model.fit(
                X.iloc[train_index].values, Y.iloc[train_index].values, epochs=epoch, verbose=0,
                validation_data=(X.iloc[test_index].values, Y.iloc[test_index].values)
            )
            scores.append(h.history['val_loss'][-1])
        del model
        K.clear_session()
        tf.reset_default_graph()
        tf.contrib.keras.backend.clear_session()
        sess = tf.Session()
        K.set_session(sess)
        gc.collect()
        # hopefully the momery is realesed
        print(scores)
        if np.mean(scores) < best_score:
            best_score=np.mean(scores)
            print(n, epoch, ':', np.mean(scores))
```

Con TimeSeriesSplit también podemos seleccionar hacer CV en los últimos elementos. Ejemplo:

```
sk_model = KerasRegressor(create_model)
param_grid = {'dim_neuron':[2, 4, 8, 16, 32], 'epochs':[20, 25, 30, 35]}
my_cv = deque(TimeSeriesSplit(n_splits=100).split(df.index), maxlen=10)
grid_model = GridSearchCV(estimator=sk_model, cv=my_cv, param_grid=param_grid)
grid_result = grid_model.fit(X.values, Y.values)
```

Autoencoders y aplicación a series temporales

Autoencoders:

FEED FORWARD

Feed Forward Network sometimes Referred to as MLP, is a fully connected dense model used as a simple classifier.



Convolutional Network assume that highly correlated features located close to each other in the input matrix and can be pooled and treated as one in the next layer.



Known for superior Image classification capabilities.

SUPERVISED

RECURRENT

Simple Recurrent Neural Network is a class of artificial neural network where connections between units form a directed cycle.



Hopfield Recurrent Neural Network It is a RNN in which all connections are symmetric. it requires stationary inputs.



Long Short Term Memory Network contains gates that determine if the input is significant enough to remember, when it should continue to remember or forget the value, and when it should output



UNSUPERVISED

Auto Encoder aims to learn a representation (encoding) for a set of data, typically for the purpose of dimensionality reduction.

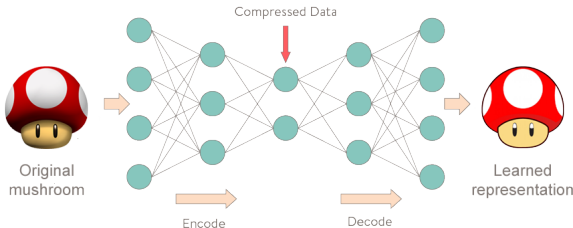


Restricted Boltzmann Machine can learn a probability distribution over its set of inputs..



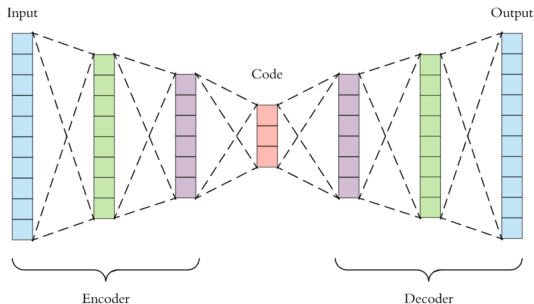
Deep Belief Net is a composition of simple, unsupervised networks such as restricted Boltzmann machines ,where each sub-network's hidden layer serves as the visible layer for the next.





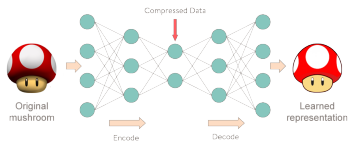
Los autoencoders son redes profundas que buscan aprender de forma no supervisada una representación simplificada de los datos. Durante el proceso de aprendizaje se genera una comprensión en dimensiones menores de los datos.

Dependiendo el problema, lo útil puede ser la representación simplificada de los datos, la comprensión en dimensiones menores de los datos o ambos. Se utiliza para reducción de dimensionalidad, generar features, detectar anomalías, reducción de ruido entre varias otras aplicaciones.



Los autoencoders están compuestos por dos submodelos apilados el encoder Φ y el decoder Ψ .

El objetivo es que $\Psi(\Phi(x)) = x$ con x en a población de interés.



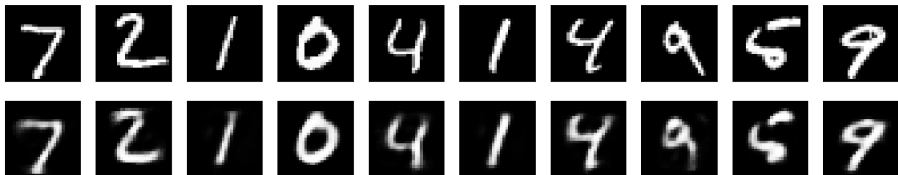
- **encoder:** Modelo encargado de comprimir los datos en dimensiones menores. El output de este modelo se suele llamar espacio código o espacio latente.
- **decoder:** Modelo encargado de reconstruir una versión simplificada de los datos a partir de la representación comprimida.

- Para problemas con datos de clases se utiliza como función de pérdida a `binary_crossentropy` y como función de activación de la última capa una `softmax` o una `sigmoid`. Para valores reales se utiliza `mean_squared_error` como función de pérdida y la función lineal como función de activación de la última capa.
- Suele estar construidas con capas Densas o convolucionales (audio e imágenes), y rara vez con RNN.
- Puede incluir capas de ruido o dropout para regularizar.

En el siguiente ejemplo podemos ver un autoencoder reducido en Keras:

```
input_img = Input(shape=(784,))  
encoded = Dense(32, activation='relu')(input_img)  
decoded = Dense(784, activation='sigmoid')(encoded)  
  
autoencoder = Model(input_img, decoded)  
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

Que al entrenarlo, produce del siguiente input de test (arriba) el siguiente output (abajo):



Una mejor versión para imágenes:

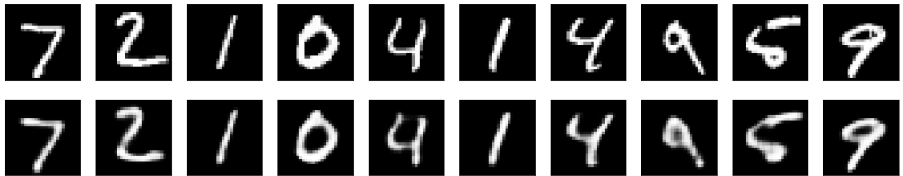
```
input_img = Input(shape=(28, 28, 1))

x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

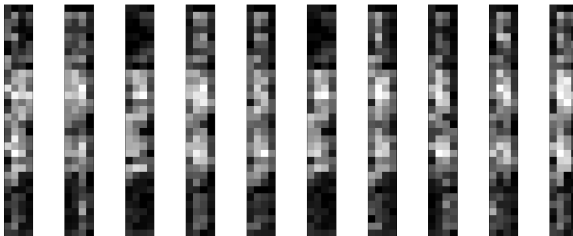
x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

conv_autoencoder = Model(input_img, decoded)
conv_autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

En el mismo conjunto test obtenemos:

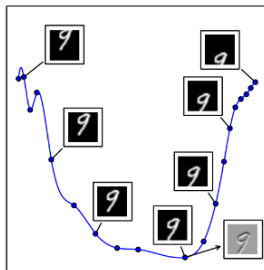


La representación en el espacio latente del conjunto test anterior es:



Los autoencoders se basan en una hipótesis que las distintas clases de los datos de interés forman una superficie en un espacio de dimensión alta. Uno espera que representar los datos de interés en un espacio latente de dimensión inferior sea más fácil distinguir entre las distintas superficies de las distintas clases.

El siguiente ejemplo es una representación de la superficie que forman la clase 9 al utilizar un autoencoder con espacio latente 2:

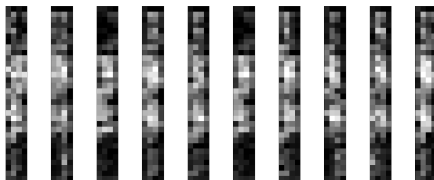


Ejemplo de uso:

si uno pretende entrenar un clasificador que prediga una imagen que numero es, uno podría en lugar de entrenar un modelo (convolucional o no) sobre la imágenes:



Uno puede entrenar sobre el espacio latente:



Uno esperando que sea mas fácil entrenar un modelo (convolucional o no) que se encargue de separar las superficies en el espacio latente.

Es decir los auto encoders pueden usarse para producir features reducidos y de relevancia.

Ejemplo de uso:

Otro uso de autoencoders es en la detección de anomalías. Puesto a que el autoencoder aprende una copia simple de los datos, se supone que una anomalía no debería ser aprendida, puesto a que una anomalía es algo fuera de lo común.

Para ello se puede construir dos algoritmos que descarten anomalías:

- **Con pocos datos anómalos identificados y etiquetados:**
Dado un parámetro a a determinar, se considera x anómalo si $\|x - \Psi(\Phi(x))\| > a$.
- **Con varios datos anómalos identificados y etiquetados:**
Con suficientes datos etiquetados se puede entrenar un algoritmo de ML de clasificación cuyo input es $x - \Psi(\Phi(x))$.

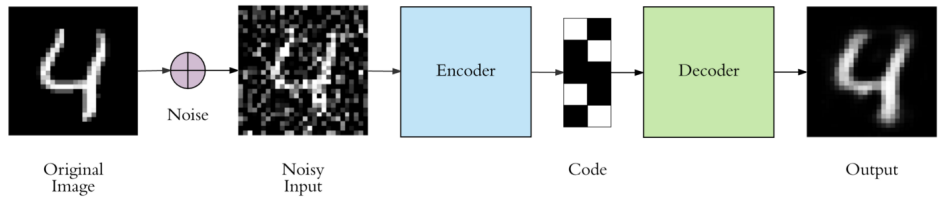
Otro ejemplo importante de autoencoder son los **denoising autoencoder**, cuyo objetivo es aprender a reducir ruido.

Formalmente:

- sea h un proceso que se encarga de introducir ruido a la muestra de entrenamiento.
- Φ un encoder.
- Ψ un decoder.

El objetivo de un denoising autoencoder es lograr que $\Psi(\Phi(h(x))) = x$.

Gráficamente:

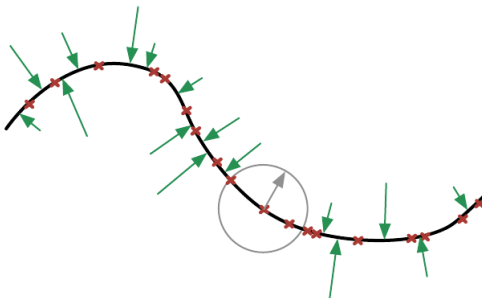


Ejemplo de input y output en test:

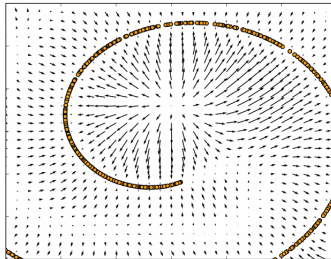


Idea intuitiva:

En un denoising autoencoder el modelo aprende mejor a proyectar datos a las superficies próximas, generando así mejor separación entre las superficies en el espacio latente.

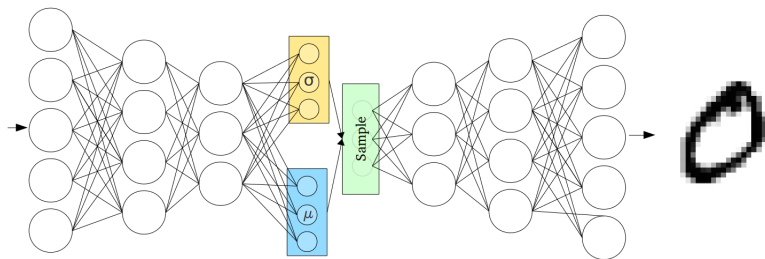


A continuación vemos un campo vectorial en un espacio latente de dimensión dos de un denoising autoencoder:



Otro ejemplo de autoencoder son los variational autoencoder, el cual se encarga de aprender la distribución de las distintas clases en el espacio latente.

Gráficamente un variational autoencoder se representaría:



En los ejercicios de Keras encontraran una implementación del mismo.

Autoencoders en Finanzas:

En finanzas los autoencoders se usan para dar precio a una canasta de productos financieros, puesto a que es un excelente modelo para reducir ruido, y encontrar relaciones entre las variables involucradas.

Sea $\{P_i\}$ una serie de productos financieros con vector de precios $[p_{i,t}]$ en tiempo t , el objetivo autoencoder sera aprender $\Psi(\Phi([p_{i,t}])) = [p_{i,t}]$ y para ello utilizará error cuadrático medio como función de pérdida.

Ejemplo:

```
input_layer = Input(shape=(input_dim, ))
encoded = Dense(dim_neuron)(input_layer)
encoded = LeakyReLU(alpha=0.3)(encoded)
encoded = Dropout(0.2)(encoded)
encoded = Dense(3, activation="linear", name="encoder")(encoded)

dencoded = Dense(dim_neuron)(encoded)
dencoded = LeakyReLU(alpha=0.3)(dencoded)
dencoded = Dropout(0.2)(dencoded)
dencoded = Dense(output_dim, activation="linear")(dencoded)

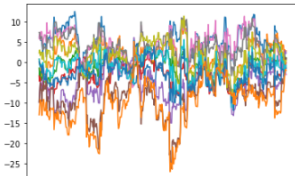
auto_encoder = Model(input_layer, outputs=dencoded)

rms = RMSprop(decay=0.001)
auto_encoder.compile(loss='mean_squared_error', optimizer=rms)
```

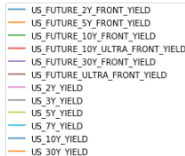
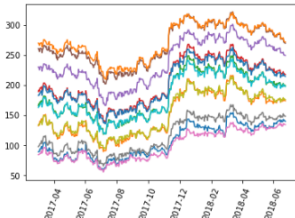
Remarcamos que para cada tiempo t , con $\Phi([p_{i,t}])$ obtendremos una representación de $[p_{i,t}]$ en el espacio latente. Por lo general en finanzas suele ser útil dicha representación en el espacio latente, tanto para analizar riesgo como para obtener un entendimiento cualitativo, por ello a la última capa del encoder le pedimos una función de activación lineal.

Para el siguiente gráfico tomamos la tasa de retorno a la madurez de distintos bonos (y futuros) de EEUU y entrenamos un autoencoder sin normalizar los datos. Los datos graficados corresponder a una parte del conjunto de entrenamiento y un mes fuera de la muestra (test):

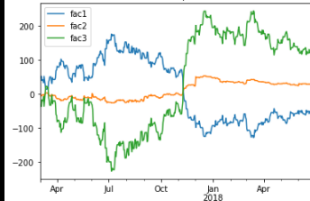
AE error:



Yield:



Latent space:



A la hora de entrenar y utilizar el modelo hay varios detalles que hay que tener en cuenta:

- ¿Cada cuanto reentrenar el modelo?
- ¿Cuánto dividir la muestra para hacer validación cruzada con TimeSeriesSplit?
- ¿Cómo se puede mejorar el modelo?
- ¿normalizar la muestra?
- ¿Cómo armar un portafolio con este modelo?

Muchas de estas preguntas no tienen respuesta exacta, solo algunas prácticas de uso que fueron determinadas por prueba y error.

¿Cada cuanto reentrenar el modelo?

La pregunta no tiene una respuesta directa. Depende del mercado entre otros factores.

En la práctica si no hubo cambios de regímenes (cambios macro o micro del mercado) el decoder lo inicializo y lo entro de nuevo cada 22 días hábiles, mientras que el encoder cada 66 días hábiles. En caso de haber un cambio estructural en el mercado probablemente al modelo haya que inicializo y lo entro nuevamente.

¿Cuánto dividir la muestra para hacer validación cruzada con TimeSeriesSplit?

La idea es que los conjuntos de test tengan la misma cantidad de días que vas a pasar sin reentrenar el modelo.

Dado a que en el uso inicializo y entreno el modelo completo cada 66 días, elijo hacer TimeSeriesSplit tal que los conjunto de test tengan alrededor de 66 días. Más aun, elijo hacer validación cruzada con los últimos 10 periodos del TimeSeriesSplit, donde cada test tiene 66 días hábiles.

¿Cómo se puede mejorar el modelo?

- Agregar sample weights al entrenamiento, dándole mayor importancia a los periodos mas recientes puede mejorar el resultado, pero puede requerir una validación cruzada para definir los pesos.
- Usar un denoise autoencoder también puede generar mejoras en el modelo e incrementar el conjunto de de datos para entrenamiento. Entreno $\Psi(\Phi([\mu_{i,t}])) = [p_{i,t}]$ con $\mu_{i,t}$ en $N(p_{i,t}, v_{i,t}^2)$ y $v_{i,t}^2$ varianza de los últimos 90 días hábiles anteriores a t para t en el conjunto de entrenamiento y testeo $\Psi(\Phi([p_{i,\tau}])) = [p_{i,\tau}]$ con τ en el conjunto de test.

¿normalizar la muestra?

Depende el objetivo del modelo. Si el objetivo es simplemente dar precio, puede que sea mejor que los precios mas caros pesen mas a la hora de entrenar el modelo, por ende no normalizar.

En la siguiente filmina veremos una estrategia para invertir que utiliza datos normalizados.

¿Cómo armar un portafolio con este modelo?

En finanzas no hay fórmula ni modelo cuantitativo mágico.

Toda decisión de inversión y armar un portafolio tiene que venir acompañado de un estudio cualitativo y un serio análisis de riesgo.

Sin embargo hay una forma de armar un portafolio utilizando un autoencoder con datos normalizados.

Las unidades vendidas de un producto se anotarán con cantidades negativas, mientras que las compras con positiva. Ahora, sea $e_{i,t} = p_{i,t} - \Psi(\Phi(p_{i,t}))$ el error del modelo y d_i la desviación estandar de e_i en los últimos 90 días. Supongamos que empezamos con un portafolio vacío.

Entonces:

- En cada producto i marcaremos como umbrales todo múltiplo positivo y negativo de d_i , es decir $\{kd_i\}_{k \in \mathbb{Z}}$.
- En cada producto i tal que $e_{i,t-1} < kd_i \leq e_{i,t}$ venderemos $1/d_i$ cantidades del producto i si teníamos $-(k-1)/d_i$ cantidades del producto i al instante $t-1$.
- En cada producto i tal que $e_{i,t-1} > kd_i \geq e_{i,t}$ compraremos $1/d_i$ cantidades del producto i si ya teníamos $-(k+1)/d_i$ cantidades del producto i al instante $t-1$.