Diseño de Redes Neuronales

Alcance de la lectura

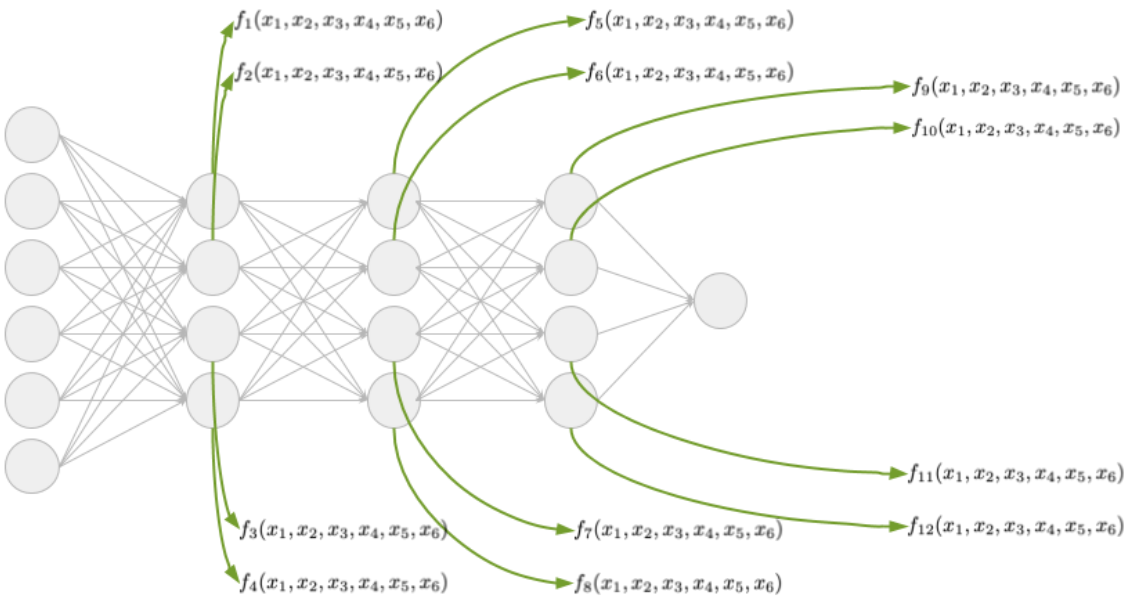
- Implementar un modelo multilayer con Keras.
- Conocer los métodos de regularización.
- Conocer backpropagation y dropout.

Ya teniendo conocimiento sobre los elementos para desarrollar una arquitectura mínima de una red neuronal, estamos en posición de implementar redes neuronales con un mayor grado de profundidad. Por profundidad hacemos referencia a la capacidad de una arquitectura de apilar más capas escondidas de manera secuencia.

En esta lectura aprenderemos sobre el proceso de entrenamiento de una red neuronal profunda, y algunas estrategias para prevenir overfitting del modelo entrenado.

Precis: Teorema de Aproximación Universal

Parte importante de la motivación detrás de la implementación de una red neuronal profunda tiene que ver con su capacidad de representar fenómenos complejos. Mediante la implementación de múltiples capas ocultas, logramos generar representaciones más complejas de los datos.



En su versión más simple, una red neuronal con una capa oculta aprende representaciones estrictamente lineales. El teorema de la aproximación universal establece que independiente de la función que deseamos representar mediante una red neuronal multicapa, una red neuronal con una profundidad substancial será capaz de representar esta función. En la medida que agregamos más capas, las funciones que representan el fenómeno adquieren parte de los impulsos procesados por las capas previas. Esto es un proceso análogo a la implementación de términos polinomiales en los atributos en un modelo de regresión lineal o logística.

Esto no asegura que la red sea capaz de aprender la función siempre. Esto surge en base a dos elementos:

- El algoritmo de optimizacieon puede fallar en cuanto a la búsqueda de los parámetros inferidos.
- La función representada puede ser incurrecta dado overfitting.

Backpropagation, o ¿Cómo aprenden las redes neuronales?

Sabemos que el flujo de una red neuronal implica transmitir los impulsos de una neurona en una capa hacia otra, de manera tal de propagarse hacia adelante. Esto se conoce como *feed forward* propagation. Resulta que mediante este flujo podemos estimar los pesos de manera unidireccional. El problema de ésta fase es la incapacidad de *actualizar* los pesos de cada neurona en la fase de entrenamiento.

El entrenamiento de redes neuronales no fue nada trivial en su tiempo. Gran parte del motivo por el que se dejaron de lado durante mucho tiempo fue debido a que no se tenía claro cómo entrenarlas. Resulta que la clave era una herramienta matemática conocida desde hace bastante tiempo, sin embargo, por alguna razón a nadie se le ocurrió utilizarla. Parte de la revitalización del estudio de las redes neuronales se debio al descubrimiento de utilizar la **regla de la cadena** para determinar el grado de error en la salida de una neurona, condicional al output de las demás neuronas en una red.

La forma general de la regla de la cadena se ve de la siguiente forma para dos una sola función anidada:

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \cdot \frac{\partial g(x)}{\partial x}$$

En la práctica, el output de la red neuronal consiste en una serie de operaciones tensoriales. Para saber qué peso debemos modificar para minimizar la función de error, necesitamos saber cuanto contribuye cada peso en el error obtenido. Para ello implementamos la regla de la cadena sobre la función de salida **desde el final de la red hasta su inicio**.

$$\frac{\partial \varepsilon_{\text{total}}}{\partial w_n^i}$$

Calculamos el error total

$$\varepsilon_{\text{total}}$$

con respecto a todos los pesos

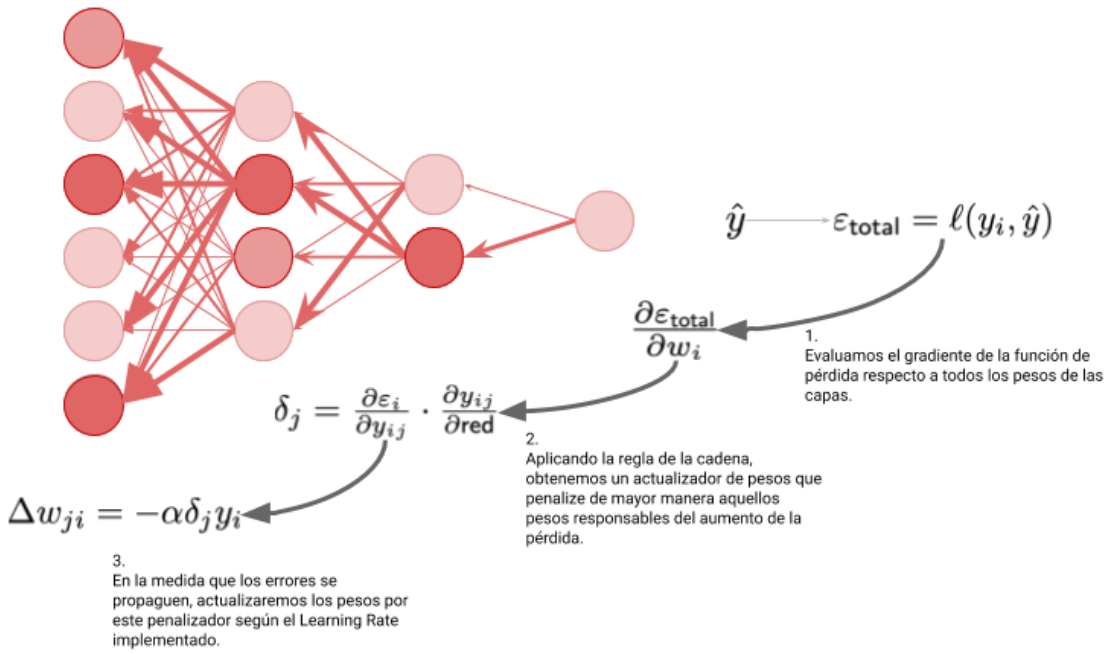
$$w^i$$

de la capa **n** (última capa), sin embargo, el error con respecto a los pesos de la última capa es igual al error de la última capa con respecto al error de la penúltima capa **n-1**, y así hasta la primera capa.

Antes de poder calcular este error, debemos tener un valor predicho por la red y comprarlo con el real. Para esto ingresamos un registro de entrenamiento por la red, lo hacemos pasar por cada una de las capas hasta el final y obtenemos el valor que, según la red, es el que corresponde, a este proceso se le conoce como Forward propagation.

Podemos entonces resumir el proceso general de entrenamiento de una red de la siguiente forma:

1. Inicializar los pesos de la red.
2. Los datos de entrenamiento son alimentados a la red mediante *feed-forward propagation*.
3. El error obtenido al comprar el output de la red con el esperado es propagado hacia atrás mediante *backpropagation*.
4. A medida que el error es propagado hacia atrás, los pesos y sesgos son ajustados según el learning rate escogido por el optimizador.



Lamentablemente, Backpropagation conlleva algunos problemas que aquejan hasta el día de hoy a las redes neuronales, algunos de los cuales repasamos al momento de hablar sobre optimizadores:

- *Network paralysis*: Si los pesos de la red se hacen demasiado grandes, la salida de la red puede ser demasiado grande mientras que la derivada de la función de activación al aplicar la regla de la cadena es muy pequeña, provacando que los pesos casi no se actualicen.

- *Mínimos locales*: Propensión a caer en mínimos locales.

- *Step-size*: Si el tamaño de avance en el espacio de búsqueda (learning rate) es demasiado pequeño, el entrenamiento es demasiado lente, si es muy alto, se puede caer en inestabilidad.

- *Estabilidad*: La red no se desvía a aprender aspectos poco relevantes para el problema, gastando tiempo de entrenamiento.

Digresión: Sobre los temidos mínimos locales

Constantemente en los temas de redes neuronales se mencionan los mínimos locales y lo malo que es caer en estos. Si bien es un escenario a evitar, hay que considerar una virtud de las **redes neuronales profundas**: Si bien existen mínimos locales, éstos comienzan a parecerse entre sí. Por tanto, caer en un mínimo local que sea marginalmente superior al resto es algo bueno, dado que legitima el uso de métodos de optimización estocástica con datos incompletos como Gradiente Descendiente Estocástica.

Redes Neuronales Multicapas

> **Caveat:** Antes de comenzar, hay que considerar que el entrenamiento de estos modelos es estocástico, por lo que puede que los resultados descritos no sean siempre exactamente iguales a los que obtengas.

En la unidad anterior vimos que al parecer, al aumentar el número de neuronas en una capa mejorabamos el rendimiento con respecto a un problema, sin embargo, sucederá lo mismo si aumentamos solamente el número de capas?. Veamos que pasa de forma experimental al aumentar el número de capas y mantener el número de neuronas en cada capa fijo:

Para ejemplificar implementaremos redes neuronales utilizaremos una base de datos sobre una campaña telefónica de Marketing realizada por un banco. El objetivo era ofrecer planes para la subscripción a depósitos a plazo a los clientes. Nuestro objetivo es clasificar si dada una serie de atributos respecto al cliente, se subscribirá a este plan o no. Los datos provienen de *S. Moro, P. Cortez and P. Rita. A Data-Driven Approach to Predict the Success of Bank Telemarketing. Decision Support Systems, Elsevier, 62:22-31, June 2014*.

Comencemos por incorporar los módulos clásicos para la ingesta, división y preprocesamiento de datos.

```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# parámetros gráficos
plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = (10, 6)
```

Importemos los datos contenidos en `bank/bank-full.csv` y recodifiquemos las variables categóricas a dummies, obviando una categoría y eliminado la variable original.

```
df = pd.read_csv('bank/bank-full.csv',sep = ';' )
df = pd.get_dummies(df, columns = ['job', 'marital', 'education',
'default','housing', 'loan', 'contact', 'month', 'poutcome'])
df['y'] = df['y'].map({'yes': 1, 'no': 0})
```

```
print(df.sample(1))
```

```
       age  balance  day  duration  campaign  pdays  previous  y  job_admin.
\
18845   45     1950    4       498         1     -1         0  0           0
job_blue-collar  ...  month_jun  month_mar  month_may  month_nov  \
18845                0  ...          0          0          0          0
month_oct  month_sep  poutcome_failure  poutcome_other  \
18845          0          0                 0               0
poutcome_success  poutcome_unknown
18845                0                1
[1 rows x 52 columns]
```

Antes de comenzar nuestra modelación de datos, debemos tener un atisbo de cuántos registros y columnas tenemos a nuestra diposición. Observamos una cantidad no despreciable (45211) de registros con alrededor de 52 atributos.

```
df.shape
```

```
(45211, 52)
```

Para el preprocesamiento de datos, implementaremos una división entre tres muestras: Entrenamiento, Validación y Holdout. Para ello implementaremos dos segmentaciones:

- Una que divida en mitades, dejando un `df_train` que posteriormente seguiremos segmentando y un `df_test` que servirá como nuestro **holdout sample**.
- Una división entre matrices de atributos en entrenamiento y validación, con sus vectores objetivos correspondientes. Con este subconjunto entrenaremos los datos.

```
from sklearn.model_selection import train_test_split

# Diferenciamos entre muestra a utilizar y muestra holdout
df_train, df_test = train_test_split(df, test_size=0.5, random_state=1313)
# removemos nuestro vector objetivo y lo guardamos en un nuevo objeto
y = df.pop('y')
# Con df_train, dividimos entre entrenamiento y validación
X_train, X_val, y_train, y_val = train_test_split(df, y,
                                                  test_size = 0.50,
random_state = 1313)

y_test = df_test.pop('y')
X_test = df_test
```

Implementando nuestra red neuronal

Siguiendo nuestro conocimiento de la lectura pasada, necesitamos incorporar tres elementos bases: Un **modelo**, una **capa** y un **optimizador**. Para este caso, importaremos los siguientes requerimientos desde `Keras`.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adadelta
from keras.utils import plot_model
```

```
Using TensorFlow backend.
```

De manera adicional, para mantener los registros de los modelos entrenados vamos a importar `Keras.callbacks.History`.

```
from keras.callbacks import History
```

Nuestra definición del modelo tendrá las siguientes consideraciones:

- **Cantidad de Neuronas**: 25. Considerando que es la mitad de atributos existentes en la matriz de atributos, es un valor relativamente conservador.
- **Función de Activación**: ReLU. Probablemente la función de activación más utilizada entre capas ocultas.
- **Función de Activación del output**: Sigmoidal. Dado que nuestro vector contiene dos clases, es una opción sensata.
- **Optimizador**: Adadelta es un optimizador adaptativo que ajusta la tasa de aprendizaje en la medida que la cantidad de épocas transcurridas aumenta. Es mucho más robusto que otros métodos.

El código subsecuente presenta todas las configuraciones a implementar.

Red Neuronal con 1 capa

```
# Importante: para facilitar la replicación de los modelos, vamos a
implementar una semilla

# configuración en la cantida de neuronas y atributos
n_neurons = X_train.shape[1] // 2
input_dim = (X_train.shape[1], )

#Iniciamos el modelo
model = Sequential()

# Agregamos la capa oculta
model.add(Dense(n_neurons,
                input_shape = input_dim,
                activation = 'relu',
                use_bias = True,
                name = '1st_layer'))
# Agregamos capa de salida
model.add(Dense(1,
                activation='sigmoid',
                name = 'output_layer'))

# Definimos la agenda de entrenamiento
model.compile(optimizer=Adadelta(),
              loss='binary_crossentropy',
              metrics=['accuracy'])
# solicitemos un resumen gráfico
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
1st_layer (Dense)            (None, 25)                1300
_________________________________________________________________
output_layer (Dense)         (None, 1)                 26
=================================================================
Total params: 1,326
Trainable params: 1,326
Non-trainable params: 0
_________________________________________________________________
```

En este caso tenemos un total de 1326 parámetros estimables en una sola capa (considerando pesos y sesgos). Dado que entrenaremos nuestro modelo con conjuntos de entrenamiento, validación y holdout, debemos especificar un poco más los elementos a pasar en el `fit`.

Otro aspecto que debemos considerar cuando entrenemos redes neuronales es que debemos registrar todos sus resultados. Dado que en algunas situaciones se transforman en modelos costosos en términos de cálculo computacional, es una buena idea preservar sus salidas, independiente de la utilidad. Para ello implementaremos un callback con `History`.

```
# Generamos un objeto que va a registrar todas las acciones de nuestro modelo
history_1 = History()

# implementaremos el fit con las siguientes acciones
model.fit(X_train, y_train, epochs=50, batch_size=512, verbose = 0,
          # le decimos al modelo que registre todas las acciones en history_1
          callbacks = [history_1],
          # guardaremos la pérdida y la exactitud del modelo
          # al final de cada época de entrenamiento transcurrida.
          # El modelo NO SE ENTRENA EN ESTOS DATOS
         validation_data = (X_val, y_val))
results_1 = model.evaluate(X_test, y_test)
```

```
22606/22606 [==============================] - 0s 17us/step
```

Red neuronal con 2 capas

Para ese caso, implementaremos una capa escondida posterior a la primera. La configuración de cantidad de neuronas y función de activación será idéntica a la primera capa. Cabe destacar el hecho que no es necesario declarar el `input_shape` de ésta, puesto que se infiere de la capa anterior

```
model = Sequential()

model.add(Dense(n_neurons,
                input_shape = input_dim,
                activation = 'relu',
                use_bias = True,
                name = '1st_layer'))

model.add(Dense(n_neurons,
                activation = 'relu',
                use_bias = True,
                name = '2nd_layer'))

model.add(Dense(1,
                activation='sigmoid',
                name = 'output_layer'))

model.compile(optimizer=Adadelta(),
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
1st_layer (Dense)            (None, 25)                1300
_________________________________________________________________
2nd_layer (Dense)            (None, 25)                650
_________________________________________________________________
output_layer (Dense)         (None, 1)                 26
=================================================================
Total params: 1,976
Trainable params: 1,976
Non-trainable params: 0
_________________________________________________________________
```

Agregar cantidad de elementos y posteriormente guardar el procedimiento de la red neuronal en un objeto generado con `History`

```
history_2 = History()

model.fit(X_train, y_train, epochs=50,
          batch_size=512, verbose = 0,
          callbacks = [history_2],
         validation_data = (X_val, y_val))

results_2 = model.evaluate(X_test, y_test)
```

```
22606/22606 [==============================] - 0s 18us/step
```

Red neuronal con 3 capas

```
model = Sequential()

model.add(Dense(n_neurons,
                input_shape = input_dim,
                activation = 'relu',
                use_bias = True,
                name = '1st_layer'))

model.add(Dense(n_neurons,
                activation = 'relu',
                use_bias = True,
                name = '2nd_layer'))

model.add(Dense(n_neurons,
                activation = 'relu',
                use_bias = True,
                name = '3rd_layer'))

model.add(Dense(1,
                activation='sigmoid',
                name = 'output_layer'))

model.compile(optimizer=Adadelta(),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
1st_layer (Dense)            (None, 25)                1300
_________________________________________________________________
2nd_layer (Dense)            (None, 25)                650
_________________________________________________________________
3rd_layer (Dense)            (None, 25)                650
_________________________________________________________________
output_layer (Dense)         (None, 1)                 26
=================================================================
Total params: 2,626
Trainable params: 2,626
Non-trainable params: 0
_________________________________________________________________
```

```
history_3 = History()

model.fit(X_train, y_train, epochs=50,
          batch_size=512, verbose = 0,
          callbacks = [history_3],
         validation_data = (X_val, y_val))

results_3 = model.evaluate(X_test, y_test)
```

```
22606/22606 [==============================] - 0s 19us/step
```

Red neuronal con 4 capas

```
model = Sequential()

model.add(Dense(n_neurons,
                input_shape = input_dim,
                activation = 'relu',
                use_bias = True,
                name = '1st_layer'))

model.add(Dense(n_neurons,
                activation = 'relu',
                use_bias = True,
                name = '2nd_layer'))

model.add(Dense(n_neurons,
                activation = 'relu',
                use_bias = True,
                name = '3rd_layer'))

model.add(Dense(n_neurons,
                activation = 'relu',
                use_bias = True,
                name = '4th_layer'))

model.add(Dense(1,
                activation='sigmoid',
                name = 'output_layer'))

model.compile(optimizer=Adadelta(),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
1st_layer (Dense)            (None, 25)                1300
_________________________________________________________________
2nd_layer (Dense)            (None, 25)                650
_________________________________________________________________
3rd_layer (Dense)            (None, 25)                650
_________________________________________________________________
4th_layer (Dense)            (None, 25)                650
_________________________________________________________________
output_layer (Dense)         (None, 1)                 26
=================================================================
Total params: 3,276
Trainable params: 3,276
Non-trainable params: 0
_________________________________________________________________
```

```
history_4 = History()

model.fit(X_train, y_train, epochs=50,
          batch_size=512, verbose = 0,
          callbacks = [history_4],
         validation_data = (X_val, y_val))

results_4 = model.evaluate(X_test, y_test)
```

```
22606/22606 [==============================] - 1s 29us/step
```

Comparación de métricas a lo largo de cada modelo

Partamos por verificar el comportamiento promedio de las métricas de pérdida y exactitud tanto en la muestra de holdout como en el entrenamiento del modelo. Para ello haremos uso de list comprehensions para extraer las cuatro métricas en cada modelo. Posteriormente las pasaremos en un DataFrame.

```
# generamos una función anónima para list comprehension
preserve_history_values = lambda x: [np.mean(x.history[i]).round(3) for i in
x.history.keys()]

values = pd.DataFrame(
    {'1 Capa': preserve_history_values(history_1),
     '2 Capas': preserve_history_values(history_2),
     '3 Capas': preserve_history_values(history_3),
     '4 Capas': preserve_history_values(history_4),
    'metric': list(history_1.history.keys())}
).set_index('metric')

values
```

	1 Capa	2 Capas	3 Capas	4 Capas
metric				
val_loss	0.955	1.145	0.883	0.505
val_acc	0.846	0.841	0.830	0.858
loss	0.763	0.956	0.757	0.430
acc	0.871	0.863	0.864	0.877

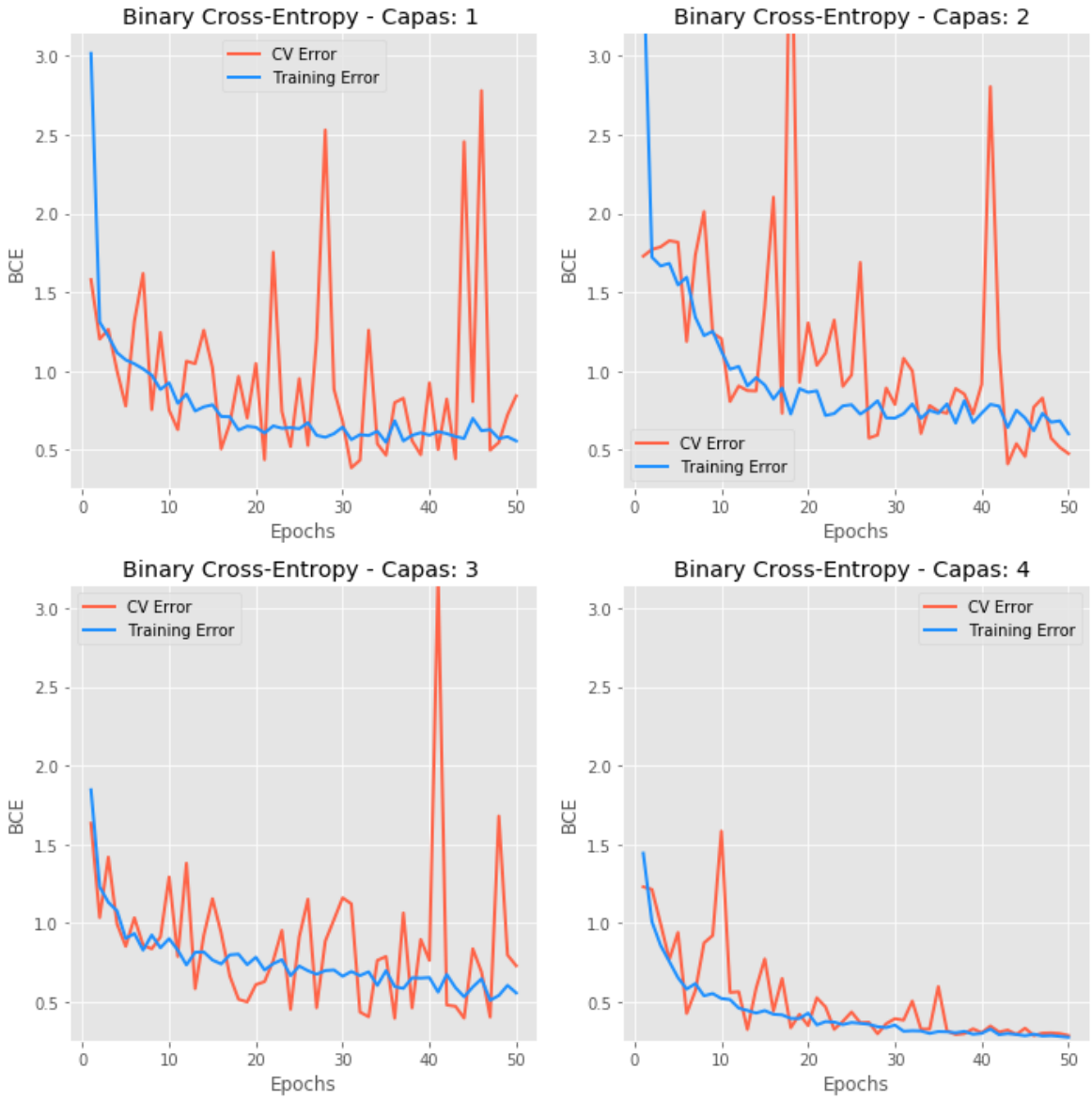
Observamos un comportamiento particular: si bien sabemos que una de las virtudes de las redes neuronales es la idea de la **aproximación universal**, el promedio de las métricas de la función de pérdida y la exactitud promedio de los modelos aparentemente no mejora. Tomemos como ejemplo el comportamiento de la exactitud tanto en la muestra holdout como en el modelo de entrenamiento: *Las métricas se mantienen relativamente constantes sin presentar fluctuaciones grandes en su magnitud*. De hecho, si optimizamos nuestro modelo en consideración a cuál de todos entrega un mejor desempeño general, optaríamos por el de 2 capas ocultas.

Inspeccionemos con un mayor detalle el comportamiento de manera visual.

```
def plot_cross_entropy(history, ax=None):
    layers = len(history.model.layers)
    extract_val_loss = history.history['val_loss']
    extract_loss = history.history['loss']
    plt.plot(range(1, len(extract_val_loss) + 1),
            extract_val_loss,label = 'CV Error',
            color='tomato', lw=2)
```

```
    plt.plot(range(1, len(extract_loss) + 1),
             extract_loss, label = 'Training Error',
             color='dodgerblue', lw=2)
    plt.legend()

plt.figure(figsize=(10, 10))
for index, value in enumerate([history_1, history_2, history_3, history_4]):
    plt.subplot(2, 2, index + 1)
    plot_cross_entropy(value)
    if index == 0:
        y_lim_min, y_lim_max = plt.ylim()
    plt.ylim(y_lim_min, y_lim_max)
    plt.title("Binary Cross-Entropy - Capas: {} ".format(index + 1))
    plt.tight_layout()
    plt.xlabel('Epochs')
    plt.ylabel('BCE')
```



El aumento de capas parece mejorar bastante la capacidad de la red para determinar la regla de clasificación del problema, sin embargo, si observamos cuidadosamente los valores entregados por las métricas de pérdida y precisión para el conjunto de pruebas notaremos algo extraño: Aunque pequeño, tanto el error de pérdida como el accuracy empeoran al aumentar la cantidad de capas. Si observamos solamente el error de entrenamiento, este muestra una curva considerablemente mejor al aumentar las capas, decae mucho más rápido al usar 3 y 4 capas que al usar 1 o 2.

La intuición nos dice que el modelo debiese ser más preciso. Sin embargo, ocurre lo contrario: en este caso la diferencia es mínima. Resulta que en un conjunto de datos más completo (tanto en dimensiones como registros de observaciones), este efecto se ve magnificado. Este comportamiento se debe a que el modelo se **sobre ajusta** al conjunto de entrenamiento.

Lamentablemente, aumentar descriteriadamente la cantidad de capas de una red neuronal, aunque nos ayuda a encontrar mejores patrones en los datos, también provoca un sobre ajuste importante. Si recordamos nuestras clases y lecturas del comienzo de este módulo, una de las soluciones a este problema es la regularización de los parámetros del modelo.

Métodos de regularización

Las formas de regularización buscan penalizar los coeficientes estimados de manera tal de evitar aprender de manera excesiva parámetros redundantes. Hagamos un pequeño repaso sobre las principales formas de regularización:

1. Regularización Ridge (Norma L2)

Ya la estudiamos previamente para los modelos más comunes de machine learning, en el caso de las redes neuronales agregaremos un término cuadrático a la función objetivo:

$$\text{función de costo} = \text{función de pérdida} + \frac{\lambda}{2m}\Sigma_w^m w^2$$

Donde:

- **w** son los pesos de la red.
- **m** son los ejemplos de entrenamiento.
- λ es un hiperparámetro de regularización que idealmente deberemos ajustar mediante gridsearch.

Cabe recordar que unas de las características principales de la regularización ridge es que los coeficientes no se regularizan linealmente hasta 0.

2. Regularización Lasso (Norma L1)

Similar a la regularización L2:

$$\text{función de costo} = \text{función de pérdida} + \frac{\lambda}{2m}\Sigma_w^m ||w||$$

Dado que el penalizador en L1 evalúa el valor absoluto del regularizador, tenderá a ponderar de manera estricta, guiándolos a cero. Esta es una virtud que permite utilizar un subconjunto disperso de los datos con los principales atributos o neuronas, manteniéndose invariante a los demás inputs.

3. Elastic-net

Una combinación de las regularizaciones L1 y L2:

$$\text{función de costo} = \text{función de pérdida} + \frac{\lambda_1}{2m}\Sigma_w^m ||w|| + \frac{\lambda_2}{2m}\Sigma_w^m w^2$$

¿Cómo implementar regularización en una capa?

La regularización se implementa a nivel de capas, donde se ingresa con el parámetro `kernel_regularizer`. Para importar el regularizador a implementar, es necesario incorporarlo con `from keras.regularizers import l2`

```
network.add(layers.Dense(units=16,activation='relu',
                         kernel_regularizer=l2(0.01),
                         input_shape=(number_of_features,)))
```

Dropout

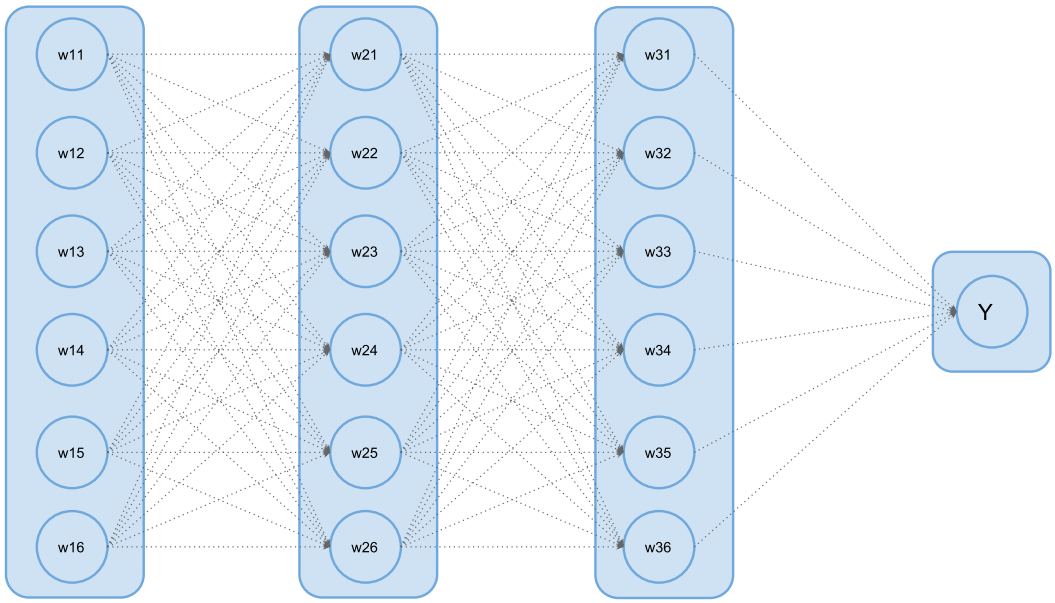
Este tipo de regularización es nuevo y comunmente utilizado en redes neuronales, puede ser utilizado en conjunto con los otros métodos de regularización vistos pues no involucra alterar la función de costo. Dropout modifica la configuración de las neuronas en la red durante el entrenamiento.

Primero, se define una probabilidad de eliminación **p** al comienzo **de cada época de entrenamiento**. El mecanismo conlleva a evaluar la probabilidad de cada neurona de **mantenerse** dentro de la red con una tasa **1-p**. Para todas las neuronas que se ignoran, se modifica su peso a 0.

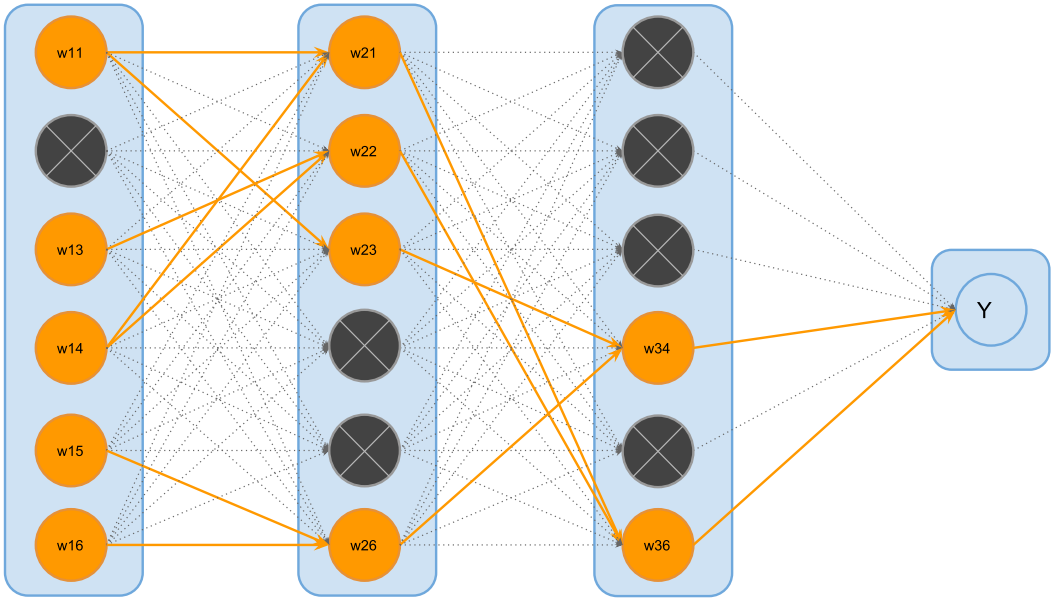
La técnica de Dropout se implementa a nivel de capas **solo en la fase de entrenamiento**. La eliminiación de las neuronas obliga al modelo neuronal evitar apoyarse en demasiadas neuronas para realizar las predicciones.

Una visualización esquemática sobre el efecto de Dropout en las redes neuronales se presenta a continuación:

- Una red neuronal antes de aplicar dropout en cada capa



- Una red neuronal posterior a aplicar dropout en cada capa



Entonces, ¿cuál es un valor razonable en Dropout? Se sugiere que en primera etapa **p = .5**. Dado que éste es un hiperparámetro, podemos modificar su rango de valores durante entrenamiento para mejorar el desempeño del modelo.

Otro aspecto a considerar es el hecho que los autores del método recomiendan restringir la norma de regularización de los pesos a un máximo de 3 (Srivastava et al. 2014). Para implementar ésto en `Keras`, podemos importar la función `Keras.constraints.maxnorm`.

Vemos ahora qué sucede con el modelo con cuatro capas al implementar regularizadores norma-L2 en la red:

Comencemos por incorporar la capa de Dropout, la regularización L2 y el constraint.

```
# Capa de Dropout
from keras.layers import Dropout
# función de regularización
from keras.regularizers import l2
# constraint
from keras.constraints import maxnorm
```

Para facilitar la composición de nuestra red neuronal, guardaremos los parámetros en nuevos objetos.

```
# iniciamos el modelo
model = Sequential()
# Dimensionado en la capa de ingreso
input_dim = X_train.shape[1]
# Norma de regularización con lambda=0.01
ridge_regularizer = l2(0.01)
# restriccion de norma de regularización
constraint = maxnorm(3)
```

La técnica de Dropout se aplica en relación a una capa específica, por lo que ésta debe añadirse al modelo **posterior** a la definición de una capa oculta. Otro aspecto a considerar es el hecho que el regularizador y la restricción de norma asociada se implementan **dentro de la capa oculta**.

```
# generamos una capa oculta con 20 neuronas
model.add(Dense(20,
                # donde ingresan todos los atributos
                input_shape = (input_dim, ),
                # implementamos un regularizador con norma l2 y lambda=0.01
                kernel_regularizer = ridge_regularizer,
                # imponemos restricción
                kernel_constraint = constraint,
                # implementamos una función de activación ReLU
                activation = 'relu',
                # consideramos una neurona extra que representa el sesgo
                use_bias = True,
                # Le asignamos un nombre
                name = '1ra_capa'))
# Aplicaremos Dropout con p=0.2 a esta capa específica
model.add(Dropout(rate=0.2, name = '1st_dropout'))
```

El ejercicio lo repetiremos para las otras tres capas restantes:

```
model.add(Dense(20,
                kernel_regularizer = ridge_regularizer,
                kernel_constraint = constraint,
                activation = 'relu', use_bias = True,
                name = '2da_capa'))

model.add(Dropout(rate=0.2, name = '2nd_dropout'))

model.add(Dense(20,
                kernel_regularizer = ridge_regularizer,
                kernel_constraint = constraint,
                activation = 'relu', use_bias = True,
                name = '3ra_capa'))

model.add(Dropout(rate=0.2, name = '3nd_dropout'))

model.add(Dense(20,
                kernel_regularizer = ridge_regularizer,
                kernel_constraint = constraint,
                activation = 'relu', use_bias = True,
```

```
                name = '4ta_capa'))

model.add(Dropout(rate=0.2, name = '4nd_dropout'))
```

Finalmente generamos la capa de egreso y sus detalles de compilación. La arquitectura de nuestra red queda definida de la siguiente manera

```
model.add(Dense(1,
                activation='sigmoid',
                name = 'output_reg'))

model.compile(optimizer=Adadelta(),
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
1ra_capa (Dense)             (None, 20)                1040
_________________________________________________________________
1st_dropout (Dropout)        (None, 20)                0
_________________________________________________________________
2da_capa (Dense)             (None, 20)                420
_________________________________________________________________
2nd_dropout (Dropout)        (None, 20)                0
_________________________________________________________________
3ra_capa (Dense)             (None, 20)                420
_________________________________________________________________
3nd_dropout (Dropout)        (None, 20)                0
_________________________________________________________________
4ta_capa (Dense)             (None, 20)                420
_________________________________________________________________
4nd_dropout (Dropout)        (None, 20)                0
_________________________________________________________________
output_reg (Dense)           (None, 1)                 21
=================================================================
Total params: 2,321
Trainable params: 2,321
Non-trainable params: 0
_________________________________________________________________
```

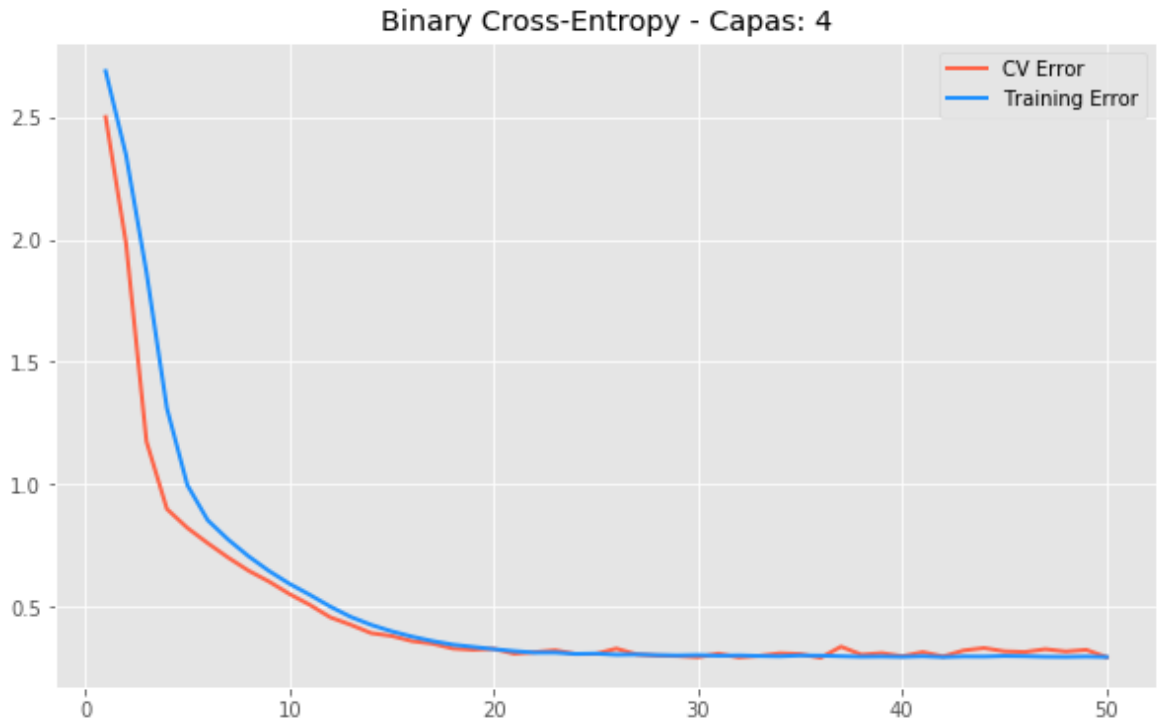
```
history_4_dropout = History()

model.fit(X_train, y_train, epochs=50,
          batch_size=512, verbose = 0,
          callbacks = [history_4_dropout],
         validation_data = (X_val, y_val))

results_4_dropout = model.evaluate(X_test, y_test)
```

```
22606/22606 [==============================] - 1s 26us/step
```

```
plot_cross_entropy(history_4_dropout)
plt.title("Binary Cross-Entropy - Capas: 4");
```




```
print('Error de perdida de prueba final modelo 4.2:
{0}'.format(results_4_dropout[0]))
print('Accuracy de prueba final modelo 4.2:
{0}\n'.format(results_4_dropout[1]))
```

```
Error de perdida de prueba final modelo 4.2: 0.2930468605185445
Accuracy de prueba final modelo 4.2: 0.8874635052693626
```

Inmediatamente podemos notar el gran cambio en la curva del error de validación, es mucho menos errático, eso quiere decir que la red está evitando, en cada epoch de entrenamiento el overfit, al contrarió de los gráficos anteriores donde el error de entrenamiento se disparaba cada ciertos epoch debido a que la red comenzaba a sobreajustarse.

El efecto de dropout y la regularización en general suele ser más visible al entrenar redes profundas, por lo que hay que tenerlo en cuenta a la hora de implementar un modelo grande.

Aspectos adicionales

Efecto del preprocesamiento de datos en las redes neuronales

> **Caveat**: Ésta discusión está inspirada en el capítulo de Redes Neuronales de Hastie et al. (2009).

Dado que el dimensionado de los inputs determina los persos de cada neurona en la primera capa, si existen dimensiones heterogéneas entre sí, podemos incurrir en la calidad de los pesos entrenados por el modelo.

Una solución es estandarizar todos los atributos de nuestra matriz de manera tal que tengan una media de 0 y una desviación estandar de uno. Mediante este paso, aseguramos que todos los atributos sean tratados de igual manera en el proceso de regularización.

Para ejemplificar el impacto de la estandarización en el comportamiento de las redes neuronales, tomemos el siguiente ejemplo: Compararemos el comportamiento de las curvas de validación cruzada y el training set con la red neuronal con dos capas. La única diferencia entre ambas será que una estará entrenada con los datos sin estandarizar y otra si. A continuación se detalla el proceso de subsampling, estandarización y entrenamiento de la red neuronal.

```
from sklearn.preprocessing import StandardScaler

X_mat = StandardScaler().fit_transform(df)
X_evaluate_std, X_holdout_std, y_evaluate_std, y_holdout_std =
train_test_split(X_mat, y, test_size=0.5, random_state=11238)
X_train_std, X_test_std, y_train_std, y_test_std =
train_test_split(X_evaluate_std, y_evaluate_std, test_size=0.5,
random_state=11238)

model_std = Sequential(name = 'two_layer_model_std')

model_std.add(Dense(20,
                input_shape = (X_mat.shape[1], ),
                activation='relu',
                use_bias=True,
                name = 'first_hidden_layer'))
model_std.add(Dense(20,
                activation='relu',
                use_bias=True,
                name = 'second_hidden_layer'))

model_std.add(Dense(1,
                activation='sigmoid',
                name='output_layer'))

model_std.compile(optimizer=Adadelta(),
             loss = 'binary_crossentropy',
             metrics = ['accuracy'])
```

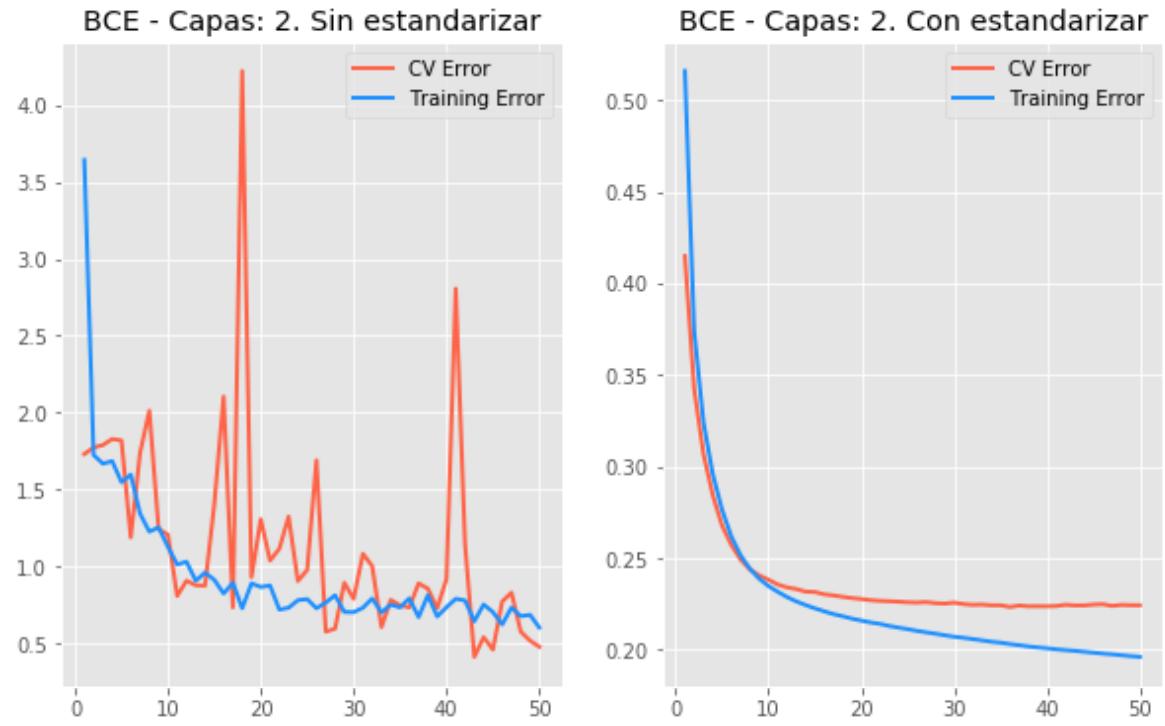
```
history_2_std = History()
model_std.fit(X_train_std, y_train_std,
              epochs=50, batch_size=512,
              verbose=0, callbacks=[history_2_std],
         validation_data = (X_test_std, y_test_std))
store_results_4_std = model.evaluate(X_holdout_std, y_holdout_std)
```

```
22606/22606 [==============================] - 1s 24us/step
```

La figura presentada abajo compara ambas situaciones. Se aprecia que los puntos de partida en la red entrenada con datos estandarizados parte de valores mucho más bajos (.6 para el error CV y .4 para el error en el training set), esto en comparación a los puntos de partida de la red sin estandarizar.

El otro punto a destacar es que el comportamiento a lo largo de las épocas es mucho más estable.

```
plt.figure(figsize=(10, 6))
plt.subplot(1, 2, 1)
plt.title("BCE - Capas: 2. Sin estandarizar")
plot_cross_entropy(history_2)
plt.subplot(1, 2, 2)
plt.title("BCE - Capas: 2. Con estandarizar")
plot_cross_entropy(history_2_std)
```



Implementando Búsqueda de Hiperparámetros por Grilla

De la última parte de la lectura surge la duda sobre cómo modificar esta cantidad de parámetros. Resulta que es posible generar métodos de búsqueda de grilla para ésto. A grandes rasgos, la implementación de búsqueda de grilla en un modelo generado con `Keras` tiene los siguientes pasos:

Importar los módulos necesarios

De manera adicional a los elementos básicos para implementar una red neuronal, debemos importar un wrapper que permitirá comunicar la estructura interna de `Keras` en `scikit-learn`. Éste se puede encontrar en `keras.wrappers_scikit_learn.KerasClassifier`. Si quisieramos implementar una red neuronal para un modelo de regresión, el wrapper es `keras.wrappers_scikit_learn.KerasRegressor`. También hay que importar el módulo de validación cruzada `sklearn.model_selection.GridSearchCV`.

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

Pasar la arquitectura base

Para poder integrar `Keras` con `scikit-learn` debemos generar el primer elemento, que es la estructura de la red neuronal. Las buenas prácticas sugieren que ésta debe declararse en una función. Dentro de la función declararemos todos los pasos hasta el método compile.

```
def create_keras_structure():
    model = Sequential()
    model.add(Dense(12, 8))
    model.add(Dense(8))
    model.add(Dense(1))
    model.compile(loss='binary_crossentropy')
    return model
```

Implementando el wrapper

Posterior a la definición del modelo, debemos integrarlo a un objeto válido para `scikit-learn`. Es en esta parte donde ocupamos `KerasClassifier` y declaramos de forma adicional la cantidad de épocas y el tamaño del batch.

```
sklearn_keras_model = KerasClassifier(build_fn = create_keras_structure,
                                      epochs=50, batch_size=100)
```

Implementar GridSearchCV

En este punto podemos declarar el diccionario con los hiperparámetros, así como implementar el modelo. El análisis de los resultados es idéntico a lo que ya conocemos.

```
param_grid = {'lr': [0.01, 0.1], 'neurons': [1, 2, 3, 4]}

neural_net_grid_search = GridSearchCV(sklearn_keras_model, param_grid =
param_grid)
neural_net_grid_search = neural_net_grid_search.fit(X, y)
```

Bibliografía

- Hastie, T; Tibshirani, T; Friedman, J. 2008. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics. Springer.
- Chollet, F. 2017. Deep Learning with Python. Manning Publications Co.
- Grus, J. 2015. Data science from scratch: first principles with python. " O'Reilly Media, Inc.".
- Karpathy, A. 2017. *cs231n: Convultional Neural Networks for Visual Recognition*. Department of Computer Science, Stanford University.
- "Neural Nets".- Francis Tseng.