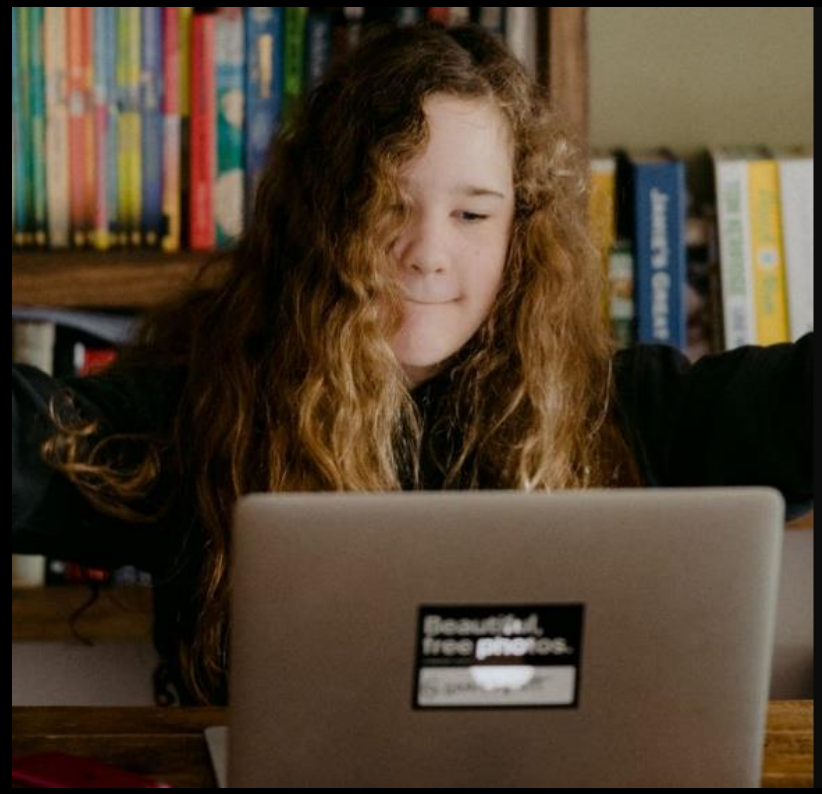
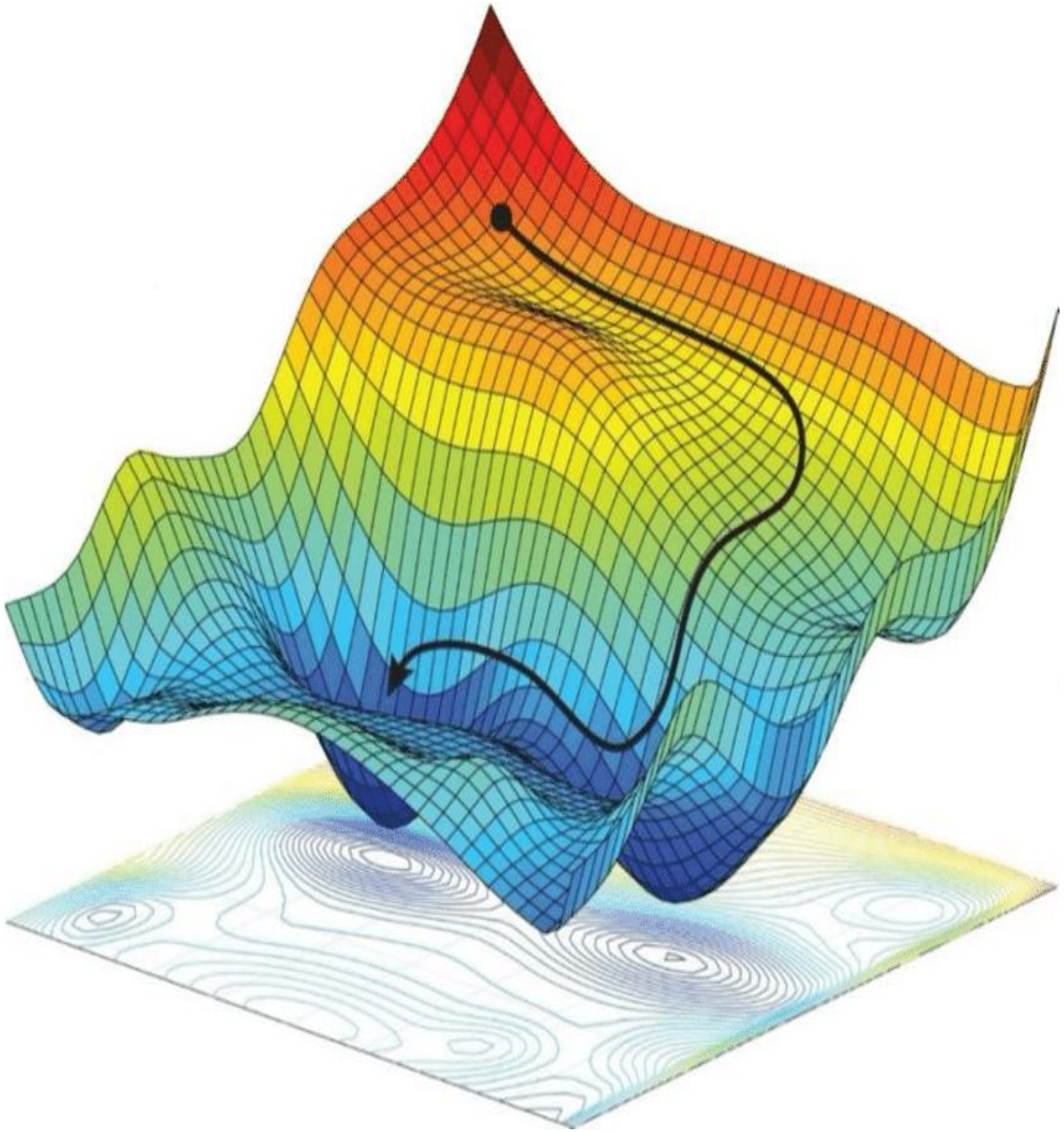
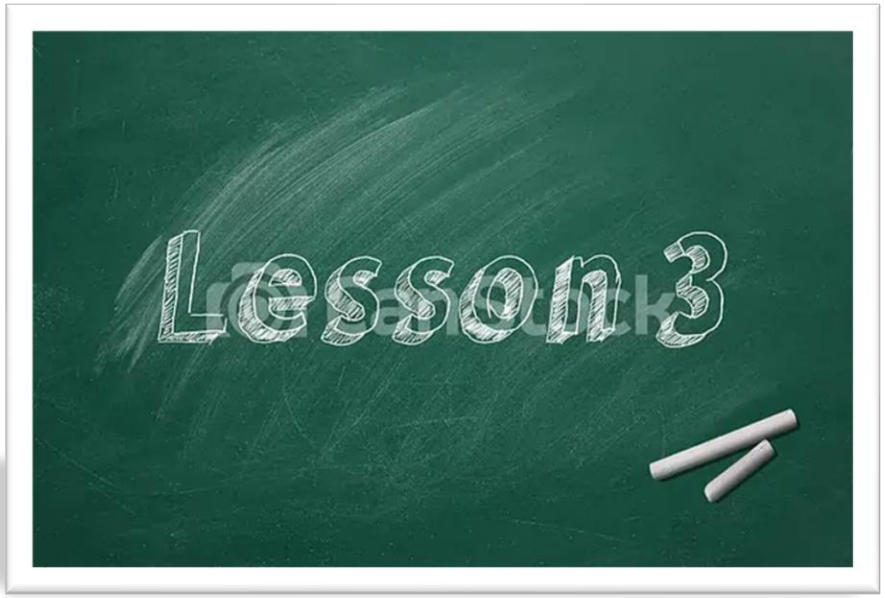
# Ruta de Aprendizaje



- **6 lecciones** de Deep Learning



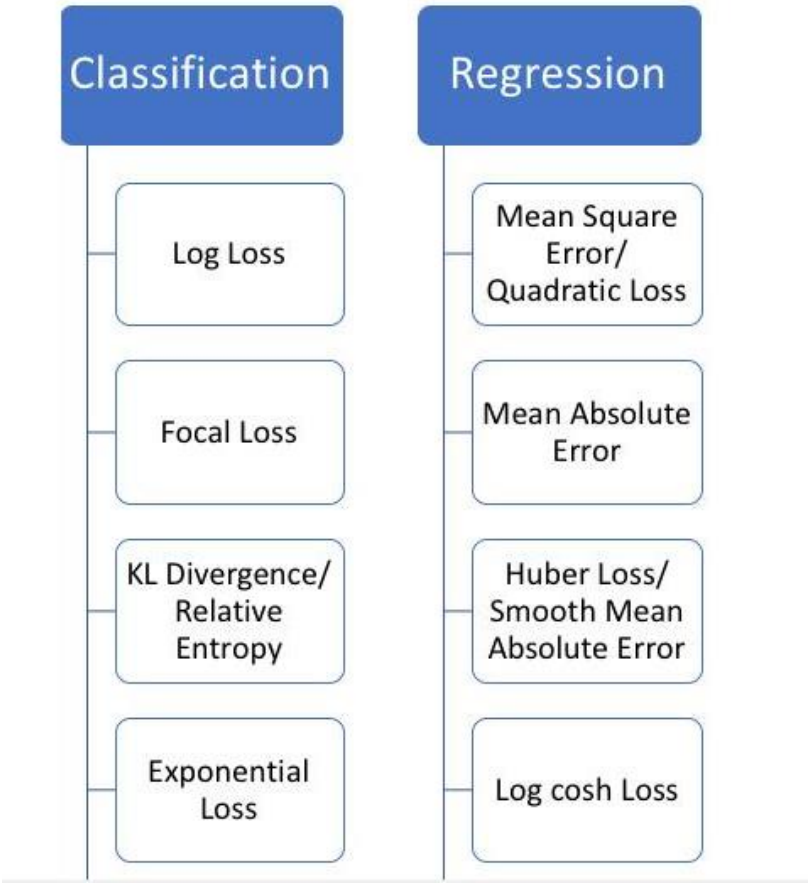



# Stochastic Gradient Descent

# Lesson 3.1: The lost function

La función de pérdida evalúa la desviación entre las predicciones realizadas por la red neuronal y los valores reales de las observaciones utilizadas durante el aprendizaje. Cuanto menor es el resultado de esta función, más eficiente es la red neuronal.

La función de pérdida se puede dividir aproximadamente en dos categorías: pérdida de clasificación y pérdida de regresión.

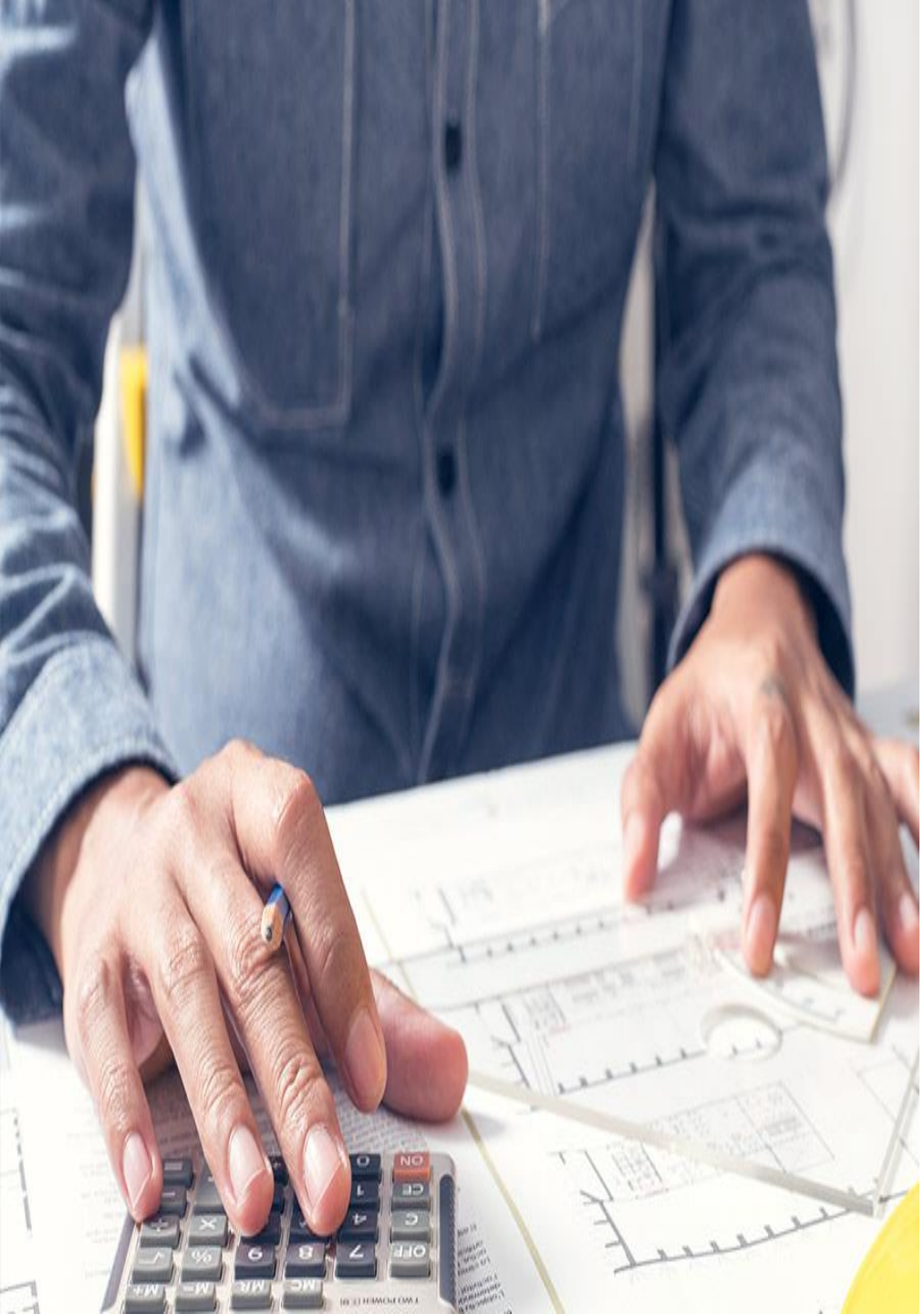



## Classification

La función de clasificación predice la etiqueta.

## Regression

La regresión, por otro lado, trata de predecir un valor continuo, por ejemplo, el área del piso , el número de habitaciones, el tamaño de las habitaciones, predecir el precio de la habitación.



## Lesson 3.1: The lost function

### Regression

**MSE (MEAN SQUARE ERROR)**

El error cuadrático medio se mide como el promedio de la diferencia al cuadrado entre las predicciones y las observaciones reales. Solo le preocupa la magnitud promedio del error, independientemente de su dirección.

- El MSE tiene buena propiedades matemáticas que hacen que sea más fácil calcular gradientes.

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

## Lesson 3.1: The lost function

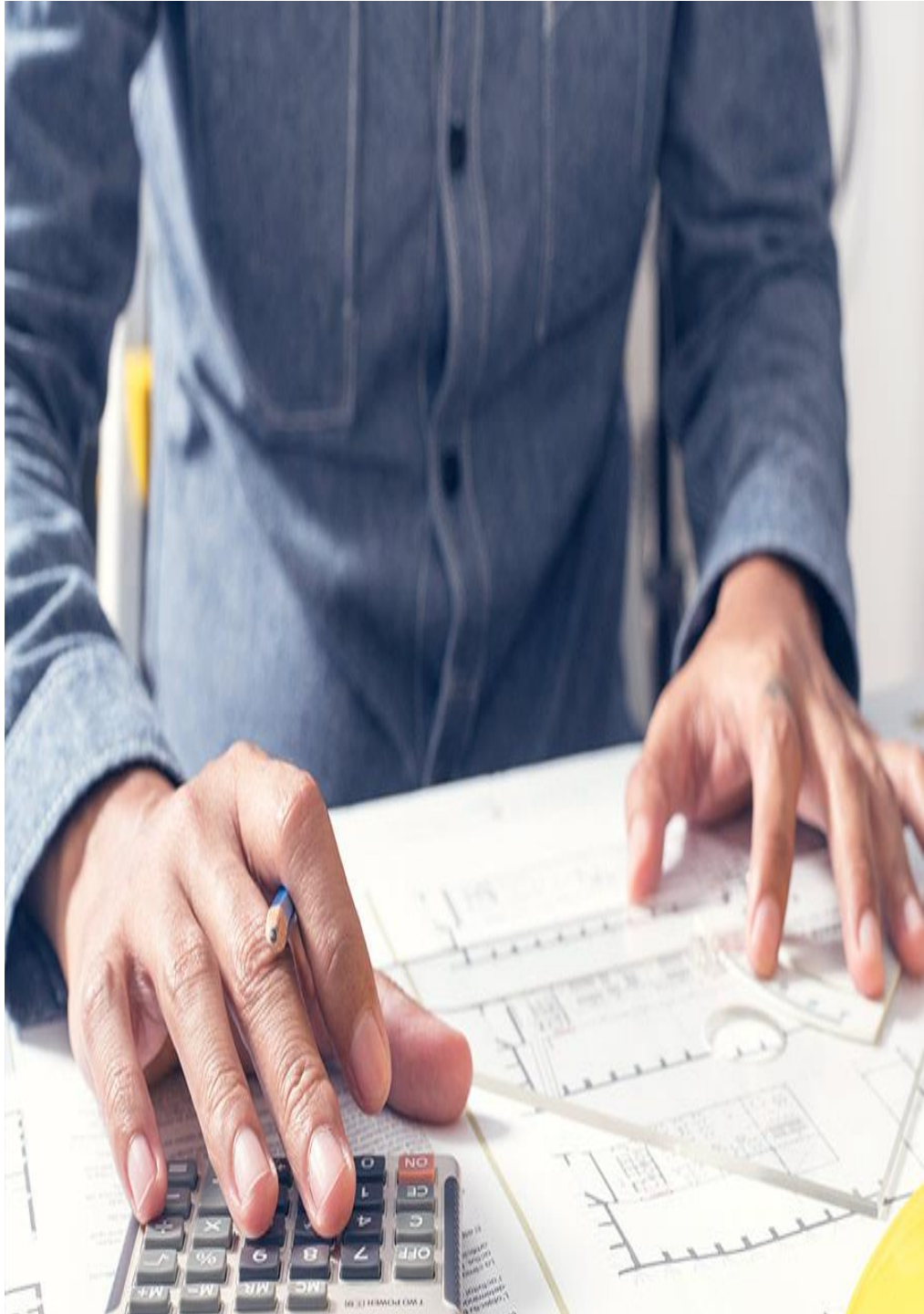
### Regression

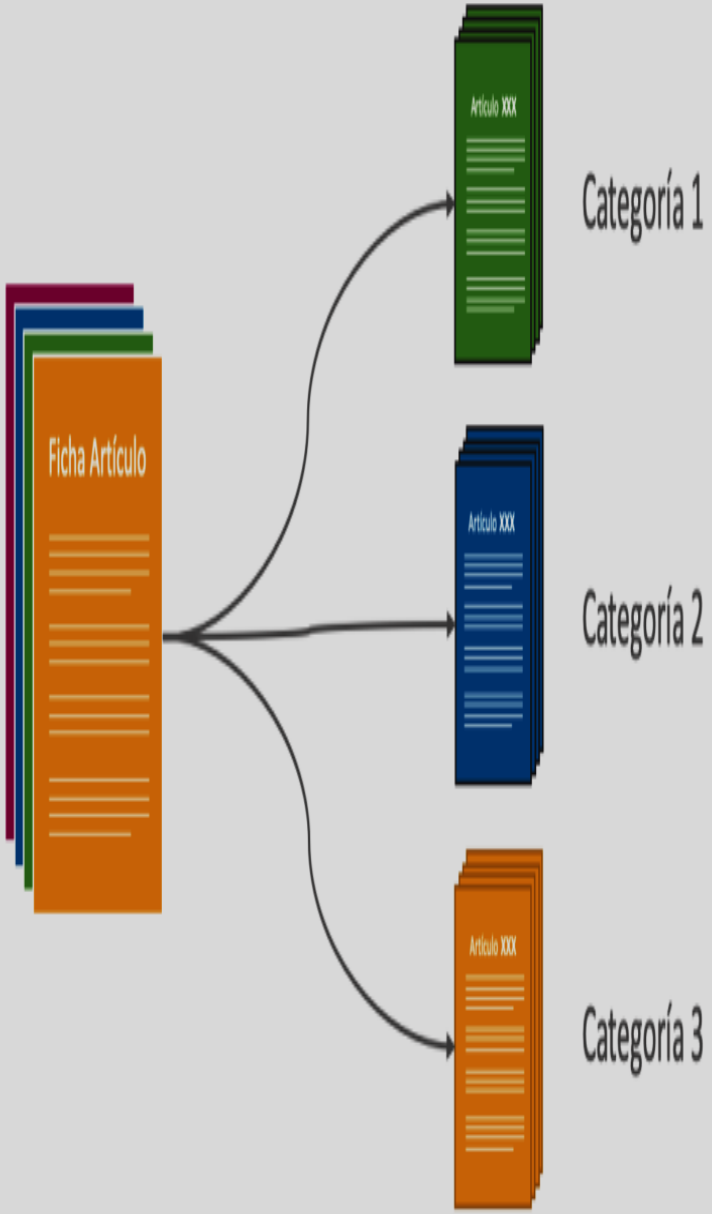
**MAE (MEAN ABSOLUTE ERROR)**

El error absoluto medio, se mide como el promedio de la suma de las diferencias absolutas entre las predicciones y las observaciones reales. También mide la magnitud del error sin considerar su dirección al igual que el MSE.

- Es más robusto para los **outliers** (valores atípicos), ya que no utiliza cuadrado.
- Necesita herramientas más complicadas, como la programación lineal para calcular los gradientes.

$$\mathrm{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$






# Lesson 3.1: The lost function

## Classification

### CrossEntropyLoss

Esta es la más común para los problemas de clasificación. La pérdida de entropía cruzada o la pérdida de registro mide el rendimiento de un modelo de clasificación cuya salida es un valor de probabilidad entre 0 y 1.

- La pérdida de entropía cruzada aumenta a medida que la probabilidad prevista diverge de la etiqueta real.

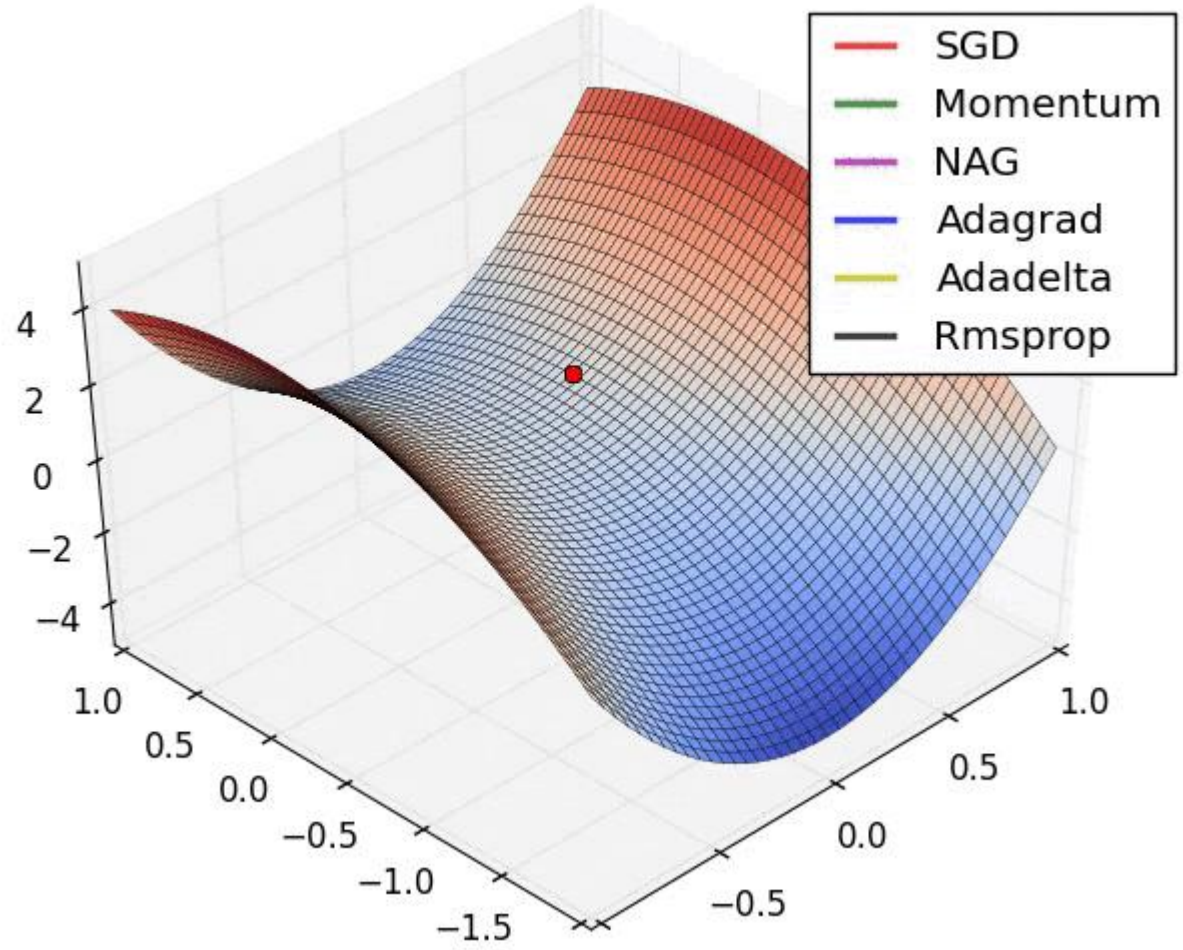
$$L_{\text{cross-entropy}}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i y_i \log(\hat{y}_i)$$

## Lesson 3.2: The optimizer

Hemos descrito el problema que queremos que resuelva la red, pero ahora tenemos que decir cómo resolverlo. Este es el trabajo del optimizador. El optimizador es un algoritmo que ajusta los pesos para minimizar la pérdida.

El optimizador Descenso de Gradiente visto no el único utilizado. De hecho, existe toda una familia de optimizadores que, basados en Descenso de Gradiente, intentan mejorar el rendimiento de éste. Entre otros, tenemos:

- Stochastic Gradient Descent
- Mini-batch Gradient Descent
- Momentum
- AdaGrad
- RMSProp
- Adam

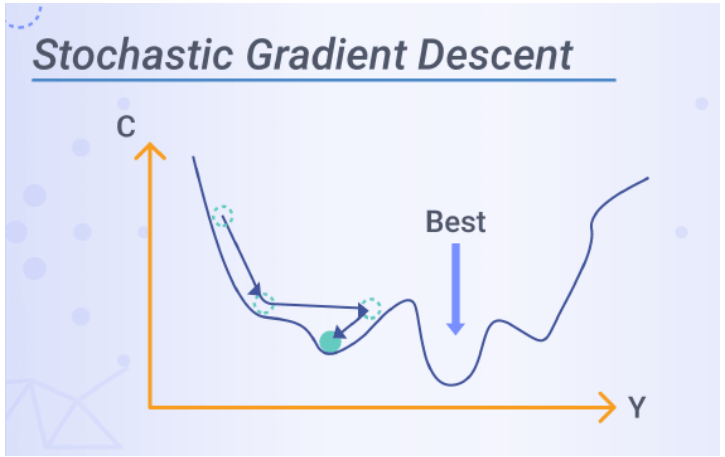


# Lesson 3.2: The optimizer

## Stochastic Gradient Descent

El optimizador Descenso de Gradiente Estocástico simplifica el cálculo considerando solo una muestra escogida de forma aleatoria cada vez que realiza el cálculo del gradiente.

Es decir, pasamos una única muestra, calculamos la función de error asociada y, a partir de esta, el gradiente y los incrementos a aplicar.

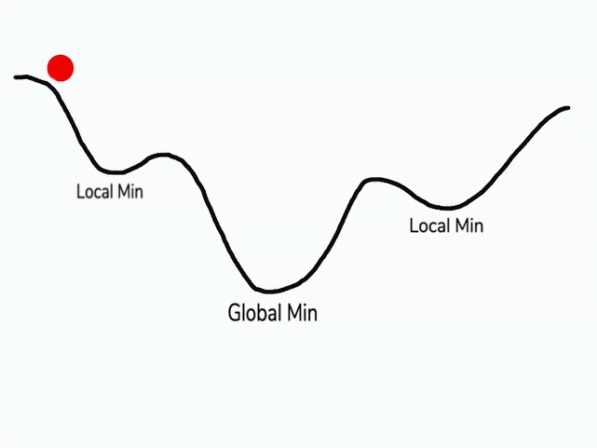



## Adam

El algoritmo de Adam obtiene las ventajas de los algoritmos AdaGrad y RMSProp.

- Utiliza las estimaciones de primer y segundo momento del gradiente para ajustar dinámicamente la tasa de aprendizaje de cada parámetro.

- La principal ventaja de Adam es que después de la corrección de desplazamiento, la tasa de aprendizaje de cada iteración tiene un cierto rango, lo que hace que los parámetros sean relativamente estables.



## Lesson 3.2: Adding the Loss and Optimizer

Después de definir un modelo, puede agregar una función de pérdida y un optimizador con el método de compilación del modelo:

```
model.compile(
    optimizer="adam",
    loss="mae",
)
```

**Conceptos básicos**

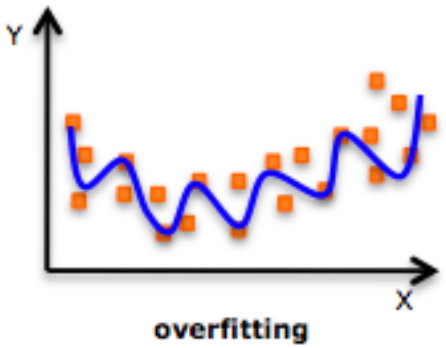
**Minibatch:** La muestra de datos de entrenamiento de cada iteración

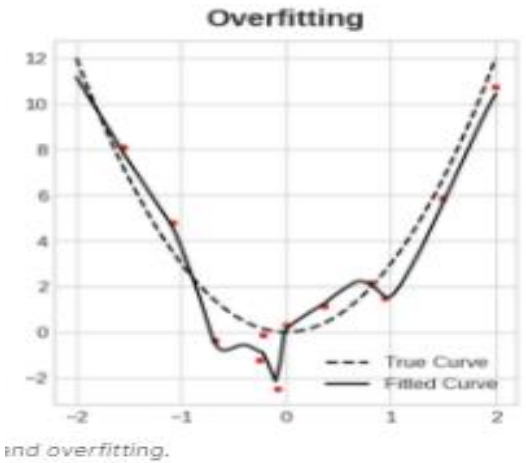
**Epochs:** Es una ronda completa de datos de entrenamiento

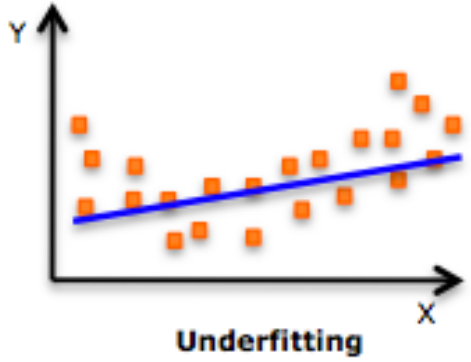
**Learning Rate:** La tasa de aprendizaje es un parámetro de ajuste que determina el tamaño del paso en cada iteración.

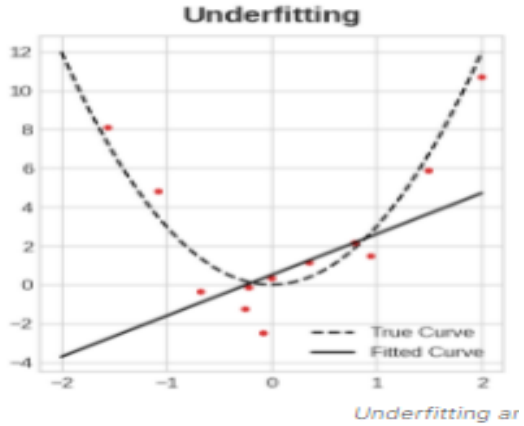

LABORATORIO 3


# Overfitting and underfitting



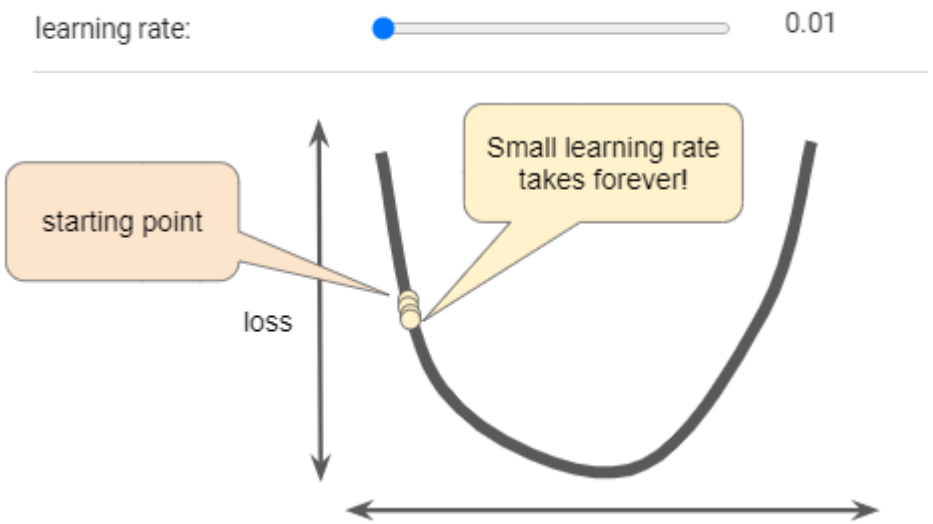
# Lesson 4.1: Learning Rate

**Epochs:** Es una ronda completa de datos de entrenamiento

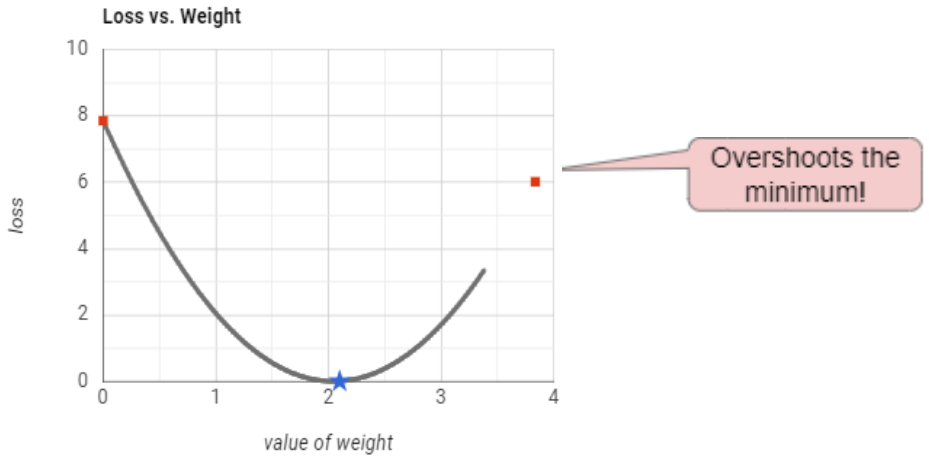
**Learning Rate:** La tasa de aprendizaje es un parámetro de ajuste que determina el tamaño del paso en cada iteración.

Los algoritmos de descenso de gradiente multiplican el gradiente por un escalar conocido como **Learning Rate** (también llamado **step size**) para determinar el siguiente punto.

**Por ejemplo:** si la magnitud del gradiente es 2.5, y Learning Rate es 0.01, entonces determinará que el siguiente punto es 0.025 del punto anterior.

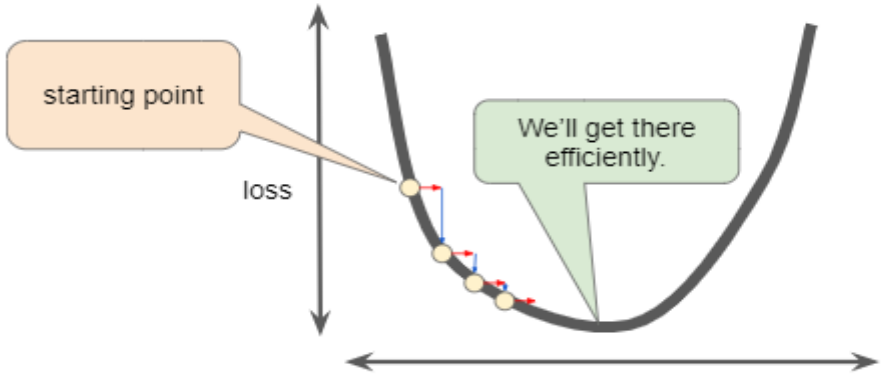



**Por ejemplo:** si Learning Rate es igual a 0.5



# Lesson 4.2: Interpreting the Learning Curves

Figura 8. La tasa de aprendizaje es la correcta.





**Ejercicio 1:** Establecer Learning Rate de 0.03. Siga presionando el botón STEP hasta que el algoritmo de descenso de gradiente alcance el punto mínimo de la curva de pérdida.

¿Cuántos pasos tomó?

**Ejercicio 2:** ¿Puede alcanzar el mínimo más rápidamente con una tasa de aprendizaje más alta? Establecer Learning Rate de 0.1 y siga presionando STEP hasta que el descenso del gradiente alcance el mínimo.

¿Cuántos pasos tomó esta vez?

**Ejercicio 3:** ¿Qué tal una tasa de aprendizaje aún mayor? Restablezca el gráfico, establecer Learning Rate de 1 e intente alcanzar el mínimo de la curva de pérdida.

¿Qué pasó esta vez?

https://developers.google.com/machine-learning/crash-course/fitter/graph?authuser=3

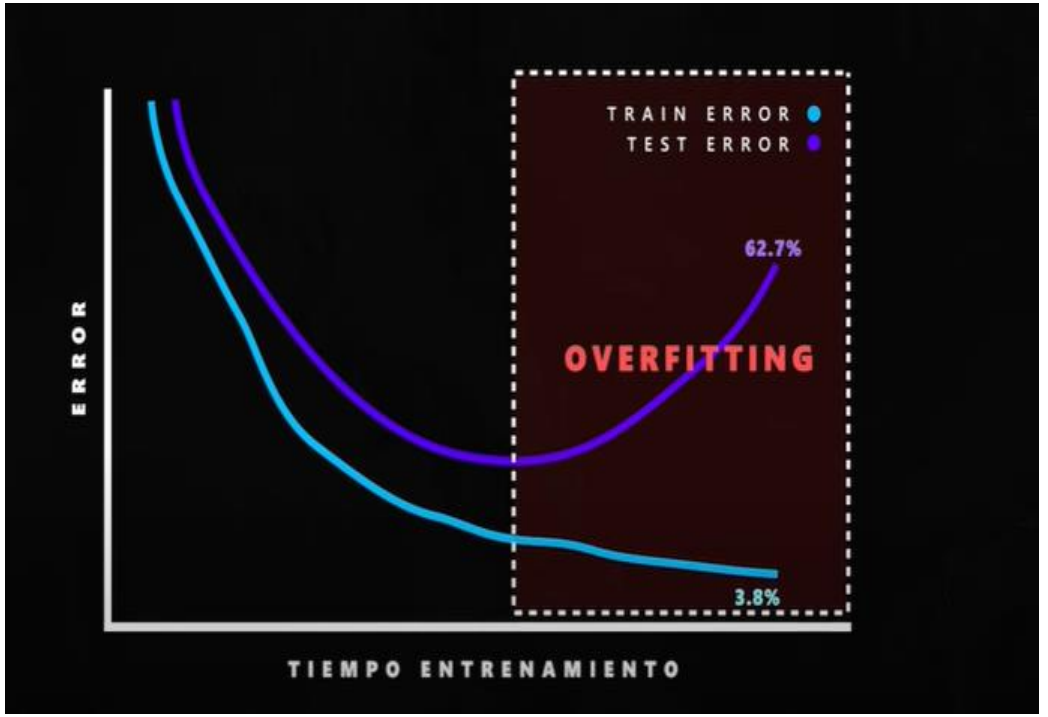
# Lesson 4.2: Overfitting

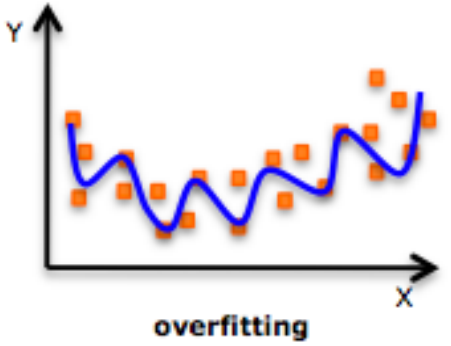
Se denomina sobreajuste **(overfitting)** al hecho de hacer un modelo tan ajustado a los datos de entrenamiento que haga que no generalice bien a los datos de test.

¿Cómo saber si hay overfitting?

**Si** el modelo entrenado con el conjunto de train tiene un 90% de aciertos y con el conjunto de test tiene un porcentaje muy bajo, esto señala claramente un problema de **overfitting**.

```
#dividir el dataset
from sklearn.model_selection import train_test_split

train_X, val_X, train_y, val_y = train_test_split(X, y, test_size=0.2, random_state=3)
```



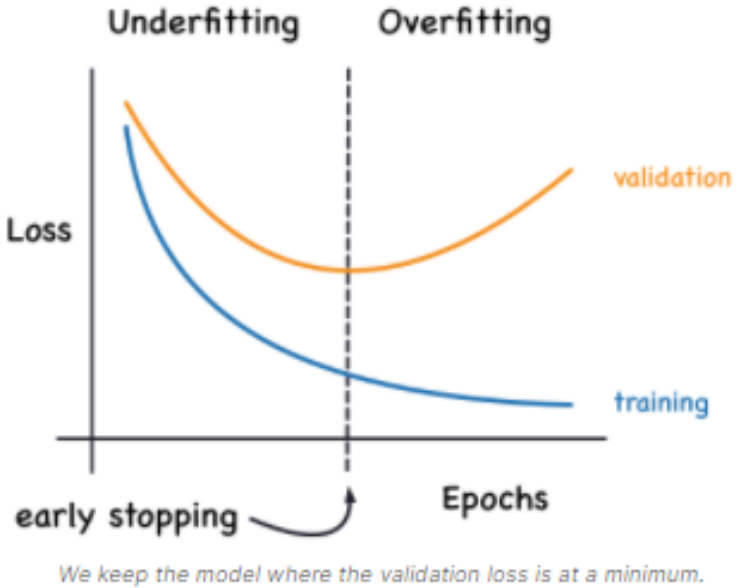

**Note:** El truco para entrenar modelos de aprendizaje profundo es encontrar el mejor equilibrio entre los dos.

https://www.youtube.com/watch?v=ZmLKqZYlYUI&ab_channel=DotCSV

# Lesson 4.3: Early Stopping

**Early Stopping** es una técnica de optimización que se utiliza para reducir el sobreajuste **(overfitting)** sin comprometer la precisión del modelo.

La idea principal de Early Stopping es detener el entrenamiento antes de que un modelo comienze a sobreajustarse.



*We keep the model where the validation loss is at a minimum.*

```
from tensorflow.keras import layers, callbacks
```

```
[ ]  #early stopping para evitar el overfitting

    early_stopping = callbacks.EarlyStopping(
        min_delta=0.01,   #cantidad mínima de cambio para contar como una mejora
        patience=30,      #cuantas epocas esperar para detener el entrenamiento
        restore_best_weights=True,
        )
```

---

**Señal:** Es la parte que se generaliza, la parte que puede ayudar a nuestro modelo a realizar predicciones a partir de nuevos datos.

**Ruido:** El ruido es esa parte que solo es cierta en los datos de entrenamiento, el ruido es toda la fluctuación aleatoria que proviene de los datos que no pueden ayudar al modelo a hacer predicciones.

# Lesson 4.4: Adding Early Stopping

En Keras, añadimos **Early Stopping** en nuestro entrenamiento a través de callbacks. Callbacks se ejecutará después de cada época.

Los siguientes parámetros dice:

- "Si no ha habido al menos una mejora de 0.001 en la pérdida de validación durante las 20 épocas anteriores, detenga el entrenamiento y conserve el mejor modelo que encontró.

```
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(
    min_delta=0.001, # minimum amount of change to count as an improvement
    patience=20, # how many epochs to wait before stopping
    restore_best_weights=True,
)
```

```
#definir tamaño de lote, épocas, etc.
history = model.fit(
        train_X, train_y,
        validation_data=(val_X, val_y),
        batch_size=15,
        epochs=200,
        callbacks=[early_stopping]
        )
```

# Lesson 4.5: Train a Model with Early Stopping

| | fixed acidity | volatile acidity | citric acid | residual sugar | chlorides | free sulfur dioxide | total sulfur dioxide | density | pH | sulphates | alcohol | quality |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1109 | 10.8 | 0.470 | 0.43 | 2.10 | 0.171 | 27.0 | 66.0 | 0.99820 | 3.17 | 0.76 | 10.8 | 6 |
| 1032 | 8.1 | 0.820 | 0.00 | 4.10 | 0.095 | 5.0 | 14.0 | 0.99854 | 3.36 | 0.53 | 9.6 | 5 |
| 1002 | 9.1 | 0.290 | 0.33 | 2.05 | 0.063 | 13.0 | 27.0 | 0.99516 | 3.26 | 0.84 | 11.7 | 7 |
| 487 | 10.2 | 0.645 | 0.36 | 1.80 | 0.053 | 5.0 | 14.0 | 0.99820 | 3.17 | 0.42 | 10.0 | 6 |

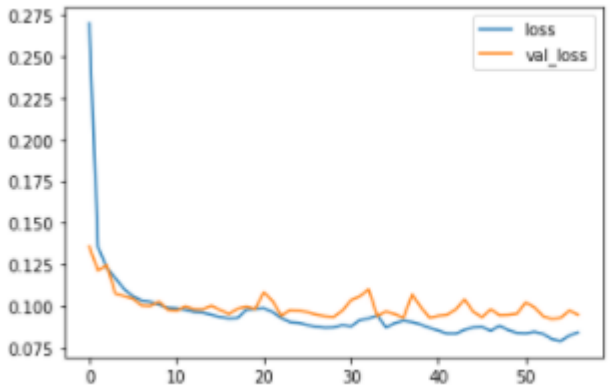
```
from tensorflow import keras
from tensorflow.keras import layers, callbacks

early_stopping = callbacks.EarlyStopping(
    min_delta=0.001, # minimium amount of change to count as an improvement
    patience=20, # how many epochs to wait before stopping
    restore_best_weights=True,
)

model = keras.Sequential([
    layers.Dense(512, activation='relu', input_shape=[11]),
    layers.Dense(512, activation='relu'),
    layers.Dense(512, activation='relu'),
    layers.Dense(1),
])
model.compile(
    optimizer='adam',
    loss='mae',
)
```

# Lesson 4.5: Train a Model with Early Stopping

In [4]:

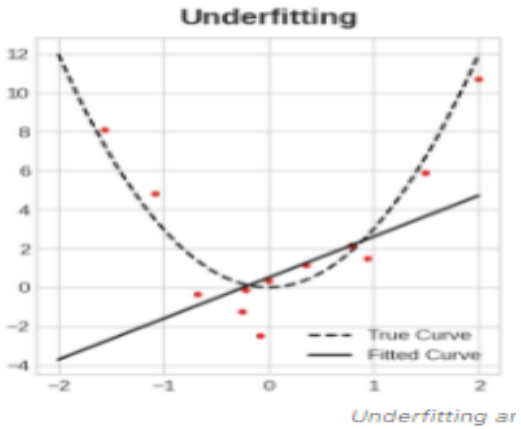
```
history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=256,
    epochs=500,
    callbacks=[early_stopping], # put your callbacks in a list
    verbose=0,  # turn off training log
)

history_df = pd.DataFrame(history.history)
history_df.loc[:, ['loss', 'val_loss']].plot();
print("Minimum validation loss: {}".format(history_df['val_loss'].min()))
```

```
2021-09-13 20:12:24.558514: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:116]
None of the MLIR optimization passes are enabled (registered 2)
2021-09-13 20:12:24.572115: I tensorflow/core/platform/profile_utils/cpu_utils.cc:112] CPU F
requency: 2000175000 Hz


Minimum validation loss: 0.09210038185119629
```



# Lesson 4.6: Underfitting

**Underfitting** (desajuste) es un escenario en Data Science en el que un modelo de datos no puede capturar la relación entre las variables del input y output con precisión, lo que genera una alta tasa de error tanto en el conjunto de entrenamiento como en los datos no vistos.

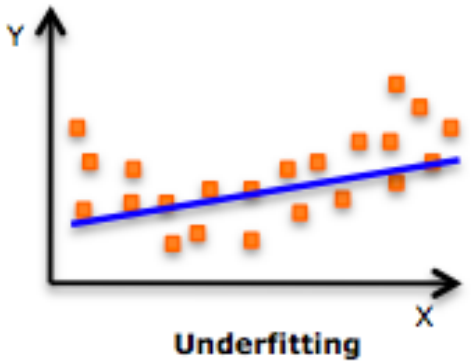



Formas de prevenir el **Underfitting**:

- **Tratar los datos correctamente, eliminando los outliers y variables innecesarias.**

- **Aumentar las iteraciones en algunos casos** (incrementar la duración del entrenamiento)

- **Disminuir la regularización:** Hay algunas técnicas que ayudan a reducir el ruido y los valores atípicos dentro de un modelo.




https://www.ibm.com/cloud/learn/underfitting

# Lesson 4.7: Underfitting & Overfitting

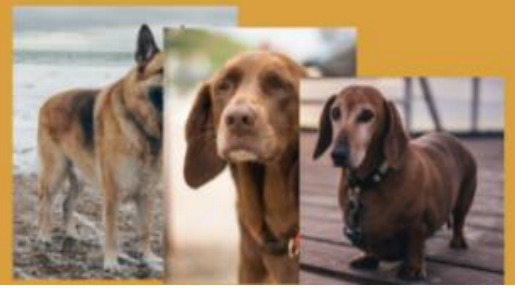

Entreno al modelo con
1 sóla raza de perro

Muestra nueva:
¿Es perro?

La máquina fallará en reconocer al perro por falta de suficientes muestras. No puede generalizar el conocimiento.

Este es el caso en el que le enseñamos sólo una raza de perros y pretendemos que pueda reconocer a otras 10 razas de perros distintas.

Entreno al modelo con
10 razas de perro color marrón

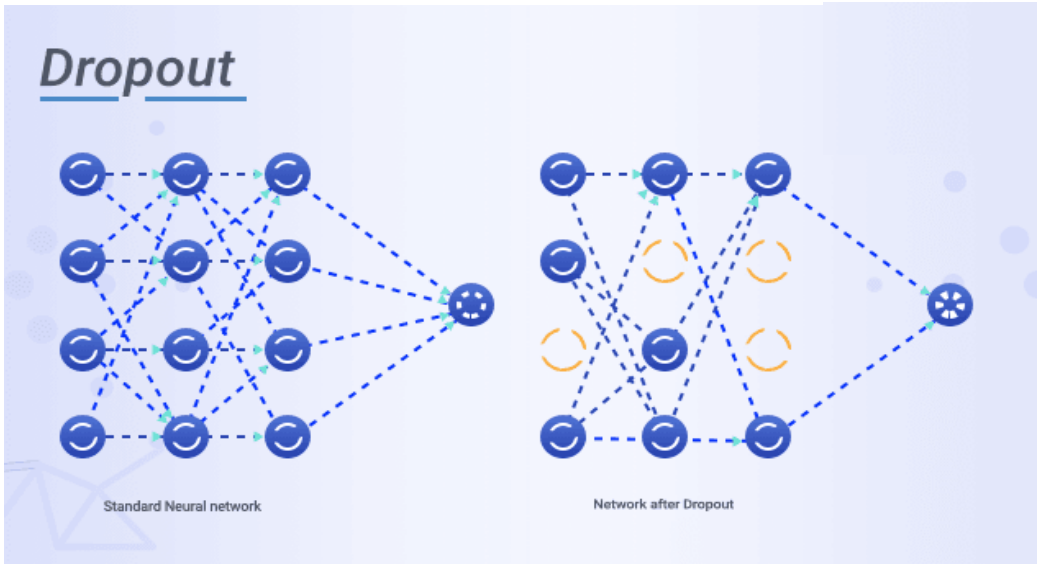


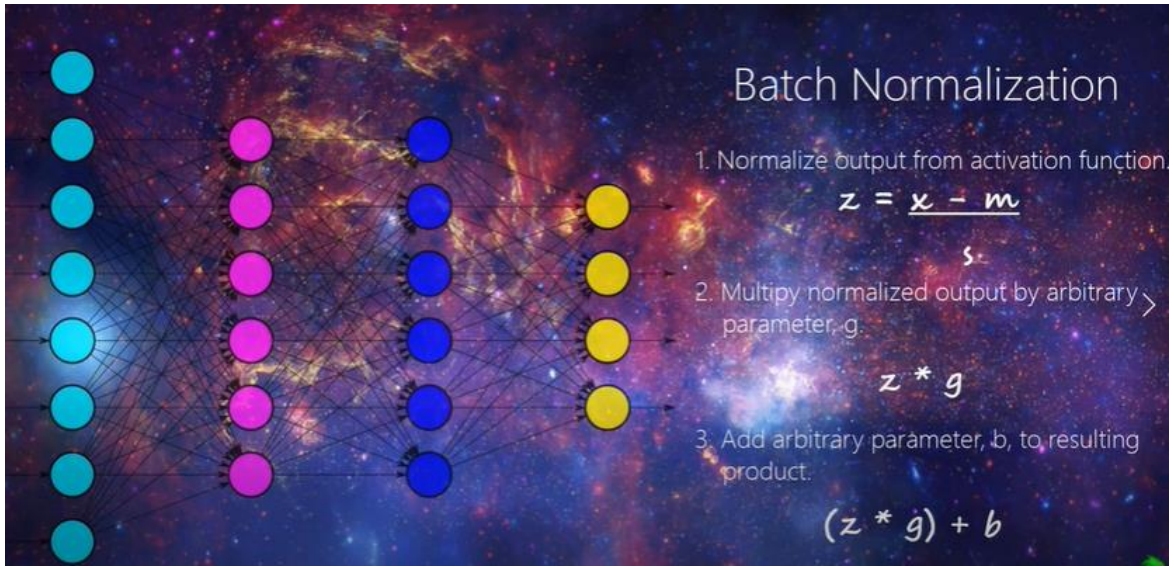
Muestra nueva:
¿Es perro?

La máquina fallará en reconocer un perro nuevo porque no tiene estrictamente los mismos valores de las muestras de entrenamiento.

Si entrenamos a nuestra máquina con 10 razas de perros sólo de color marrón de manera rigurosa y luego enseñamos una foto de un perro blanco, nuestro modelo no podrá reconocerlo cómo perro por no cumplir exactamente con las características que aprendió. Aquí **se trata de un problema de overfitting**.


LABORATORIO 4



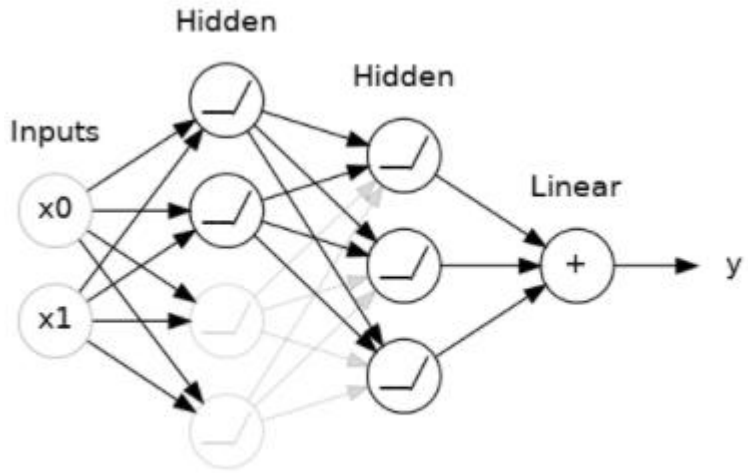


# Dropout & Batch Normalization

# Lesson 5.1: Dropout

El **Dropout** es una técnica de regularización que se basa en la eliminación de neuronas en las capas de la red neuronal.

El procedimiento es: por cada nueva entrada a la red en fase de entrenamiento, se desactivará aleatoriamente un porcentaje de las neuronas en cada capa oculta, acorde a una probabilidad de descarte previamente definida. Dicha probabilidad puede ser igual para toda la red, o distinta en cada capa.




*Here, 50% dropout has been added between the two hidden layers.*

Es una de las técnicas que **más funcionan dentro de una red neuronal** y consiste en dropear nodos temporalmente en cada iteración al momento de entrenar. Estos nodos se dropean con una probabilidad p, que usualmente es en 50%.

**Nota:** Es aplicado tanto a los inputs como a los hidden layers, pero no a outputs.

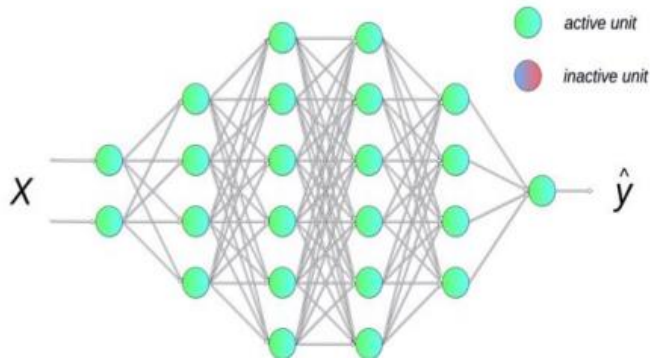
# Lesson 5.2: Adding Dropout

En Keras,

- la tasa de argumento de **dropout rate** (tasa de abandono) define qué porcentaje de las unidades de entrada se apagarán.

- Se coloca **Dropout layer** (capa de abandono) justo antes de la capa a la que desea aplicar el abandono.

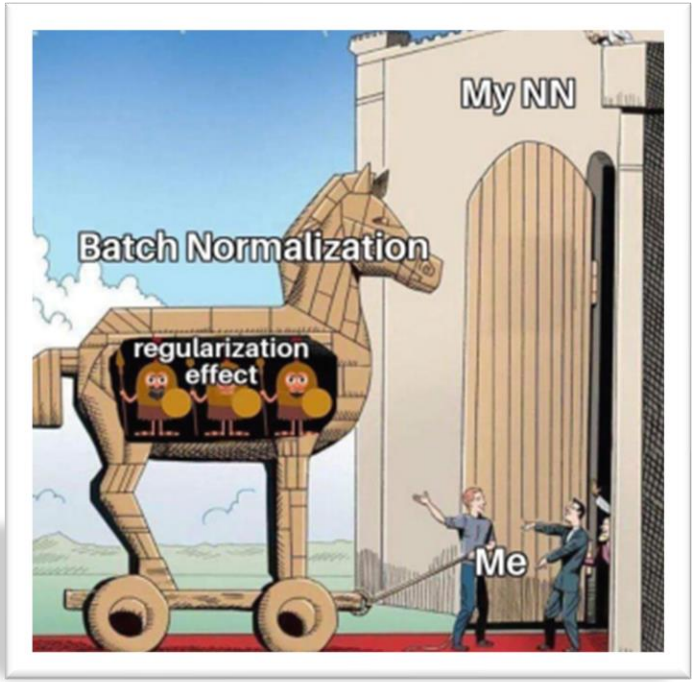
```
keras.Sequential([
    # ...
    layers.Dropout(rate=0.3), # apply 30% dropout to the next layer
    layers.Dense(16),
    # ...
])
```

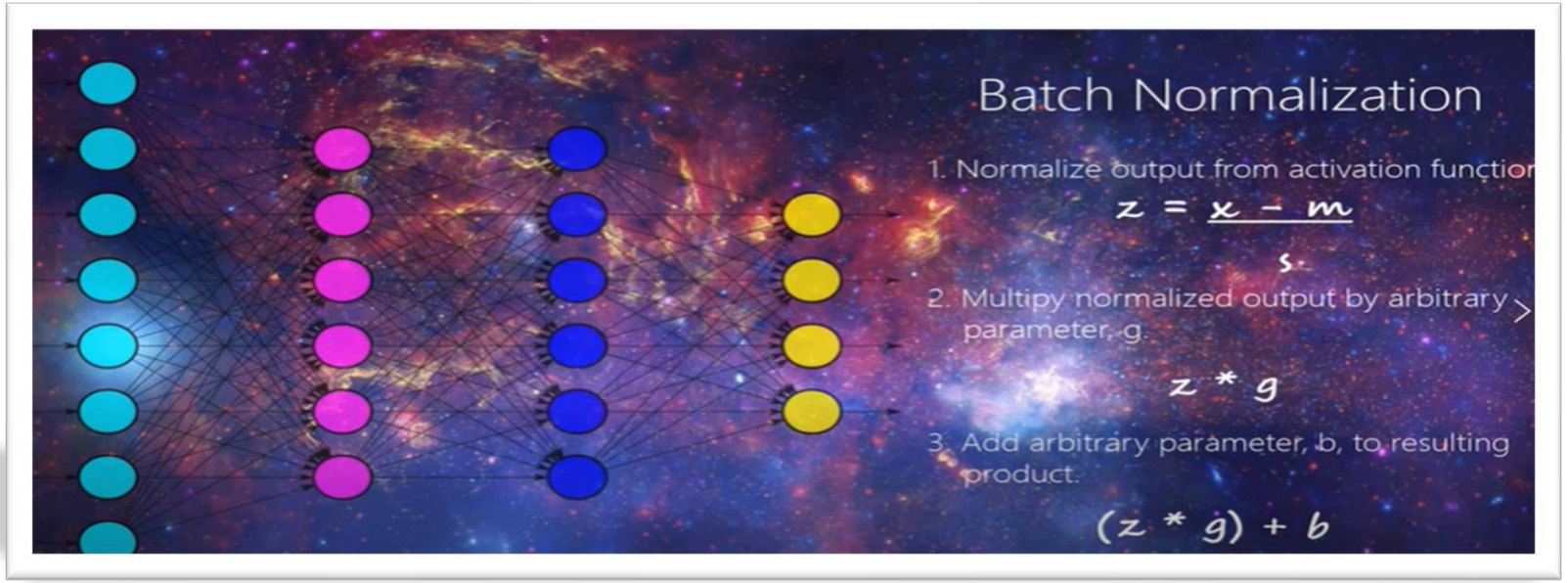


# Lesson 5.3: Batch Normalization

**Batch Normalization** (normalización por lotes) o también llamado **batchnorm**, puede ayudar a corregir el entrenamiento lento o inestable.

La normalización por lotes consiste básicamente en añadir un paso extra, habitualmente entre las neuronas y la función de activación, con la idea de normalizar las activaciones de salida.



# Lesson 5.4: Adding Batch Normalization

**Batch Normalization** (normalización por lotes) se puede utilizar en casi cualquier punto de una red.

- Puedes ponerlo después de una capa

```
layers.Dense(16, activation='relu'),
layers.BatchNormalization(),
```

- Puedes ponerlo entre una capa y su función de activación

```
layers.Dense(16),
layers.BatchNormalization(),
layers.Activation('relu'),
```

Y si lo agrega como la primera capa de su red, puede actuar como una especie de preprocesador adaptativo, reemplazando a algo como Scikit Learn StandardScaler.

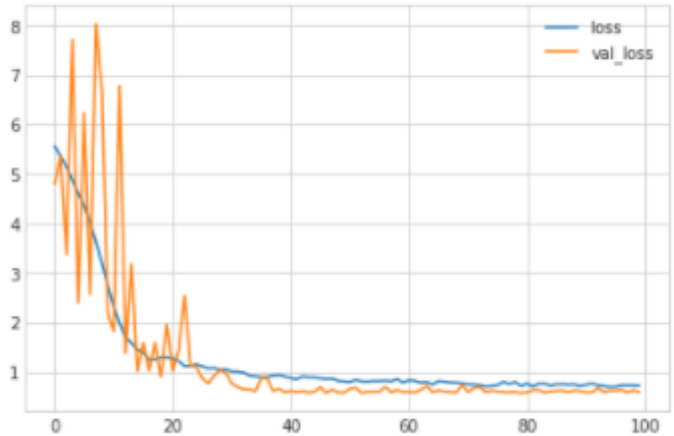
## Lesson 5.5: Using Dropout and Batch Normalization

Al agregar Dropout, es posible que deba aumentar la cantidad de unidades en sus capas densas.

```
]:
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(1024, activation='relu', input_shape=[11]),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(1),
])
```

# Lesson 5.5: Using Dropout and Batch Normalization

```
model.compile(
    optimizer='adam',
    loss='mae',
)

history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=256,
    epochs=100,
    verbose=0,
)


# Show the learning curves
history_df = pd.DataFrame(history.history)
history_df.loc[:, ['loss', 'val_loss']].plot();
```




LABORATORIO 5