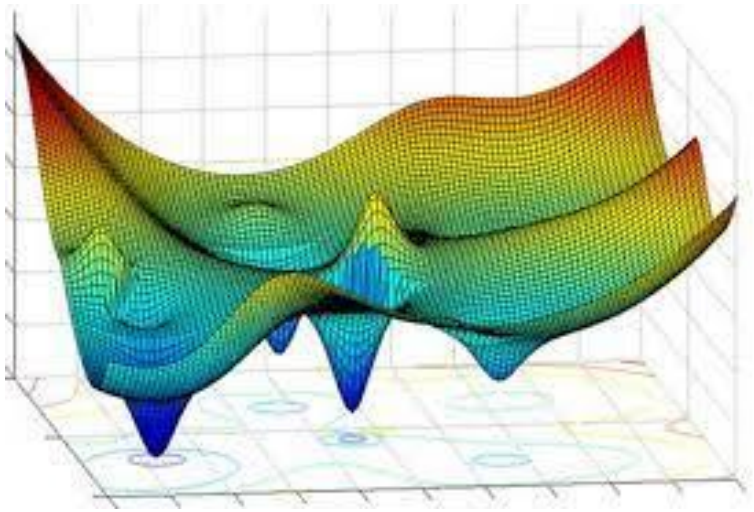
Aprendizaje Profundo

Entrenamiento de la red neuronal

Motivación

- A diferencia de los modelos lineales, la **no linealidad de las redes neuronales** genera funciones de costo (error) más interesantes y principalmente no convexas.
- Calcular una **expresión analítica para el gradiente** puede ser sencillo, pero evaluar esta expresión numéricamente puede ser computacionalmente costoso.
- Con mayores o menores modificaciones, las redes neuronales minimizan la función de costo usando técnicas de **descenso del gradiente** como en métodos clásicos de machine learning.



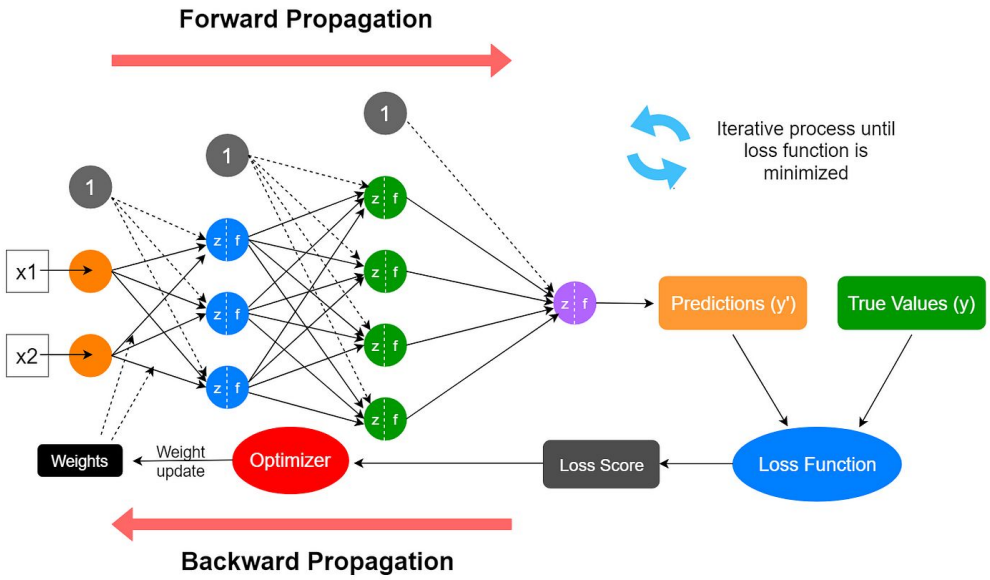
Backpropagation

La *retropropagación* es la esencia del entrenamiento de una red neuronal.

Luego de un paso **hacia adelante**, tenemos el valor de la predicción y podemos calcular el error utilizando la función de costo.

Con este valor, podemos volver **hacia atrás**, ajustar los pesos y volver a hacer el proceso iterativo.

Fine-tunning de los pesos



Conceptos importantes

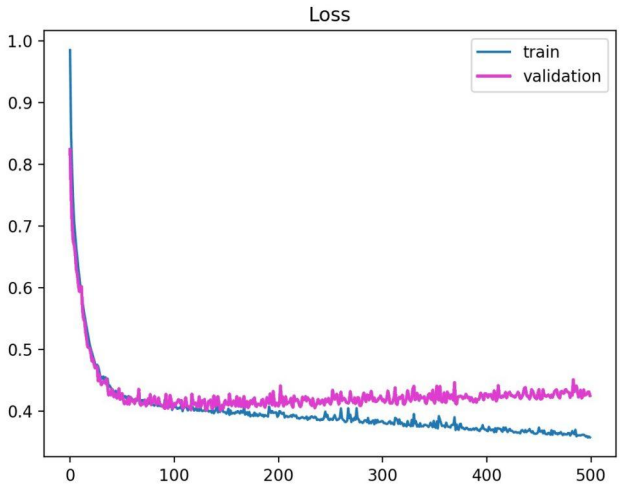
EPOCH

Es una iteración completa sobre el conjunto de entrenamiento que permite calcular y optimizar el gradiente (entrenar el modelo). En otras palabras, **cuando el modelo se ha visto y se ha entrenado con todos los ejemplos de entrenamiento una vez**.

Por su estructura, las redes neuronales necesitan aprender muchos parámetros, por eso las epochs ayudan a hacer un ajuste adecuado más preciso a medida que el modelo "ve" múltiples veces el conjunto de entrenamiento.

Una epoch es usualmente un número grande algunos ejemplos van desde 10, 100, 500, 1000 o más. El número de épocas óptimo es cuando el error se fue minimizado lo suficiente.

Una epoch se compone de uno o más lotes (batches).



Conceptos importantes

BATCH

Es un **hiperparámetro** que define la cantidad de observaciones con las que se debe trabajar antes de actualizar los parámetros internos del modelo.

```
1<= batch <= tamaño del dataset de entrenamiento
```

Un conjunto de entrenamiento puede dividirse en uno o más lotes.

Tipos de descenso de gradiente según el batch:

- Descenso de gradiente por lotes (Batch Gradient Descent)

```
Tamaño del lote = Tamaño del conjunto de entrenamiento
```

- Descenso de gradiente estocástico (Stochastic Gradient Descent)

```
Tamaño de lote = 1
```

- Descenso de gradiente por mini lotes (32, 64 y 128 suelen ser valores comúnmente usados..)

```
1 < Tamaño del lote < Tamaño del conjunto de entrenamiento
```


Batch gradient descent
Mini Batch gradient descent
Stochastic gradient descent

Conceptos importantes

BATCH

- El tamaño del lote debe ser lo más pequeño posible para no quedarse sin memoria.
- Modelo simple y dataset pequeño: (por ejemplo, menos de 10.000 ejemplos), puede ser adecuado utilizar un tamaño de lote relativamente grande, como 50 o 100 muestras/lote.
- La tarea de establecer el tamaño del lote, depende de muchos factores, como:
  - la complejidad del modelo
  - la cantidad de datos disponibles
  - el tiempo y recursos que se tienen para entrenar el modelo

Conceptos importantes

Ejemplo

Tenemos un batch de 2 ejemplos de entrenamiento y nuestra red tiene 5 pesos entonces estos dos lotes entran simultáneamente a la red y actualizan sus gradientes individualmente:

- Gradientes del ejemplo 1: (1.5,-2, 1.1, 0.4, -0.9)
- Gradientes del ejemplo 2: (1.2, 2.3, -1.1, -0.8,-0.7)

Una vez que tenemos los gradientes de cada uno de estos ejemplos para hacer el paso de actualización lo que se hace es promediar los gradientes. Entonces el gradiente para ese batch nos quedaría un vector gradiente de (1.35, 0.15, 0, -0.2, -0.8).

Una de las ventajas es que la actualización no depende de un solo ejemplo y ***la variación del gradiente es menor y más consistente*** (reducimos el ruido).

Conceptos importantes

STEP

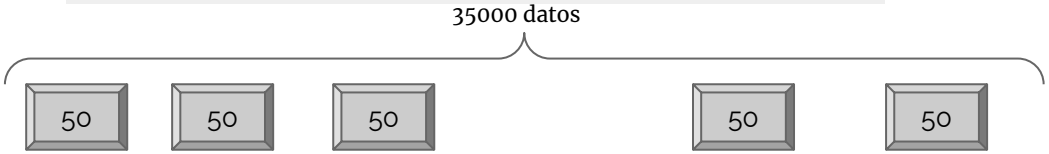
Consiste en realizar una sola actualización del modelo (pesos o conexiones de la red) a partir de 1 lote de datos.

Por ejemplo, dadas
- Dataset de entrenamiento: 35000 ejemplos
- Tamaño del lote= 50

entonces el modelo va a necesitar **700 pasos para completar una época**.

En este caso, el modelo vería cada ejemplo (del lote de 50) una vez durante cada paso.

TAMAÑO DEL PASO:
Número total de ejemplos de entrenamiento / Tamaño del lote




```
Epoch 1
train_loss : 0.6916678399699074 val_loss : 0.6876134955883026
train_accuracy : 52.47428571428572 val_accuracy : 58.053333333333335
Validation loss decreased (inf --> 0.687613).  Saving model ...
Triggers times: 0
================================================
100% 700/700 [00:08<00:00, 81.91it/s]
```

EJEMPLO

TAMAÑO DEL CONJUNTO DE DATOS: **200 muestras (filas de datos)**
TAMAÑO DEL LOTE: **5 lotes**
CANTIDAD DE ÉPOCAS: **1000 épocas.**

- Esto significa que el conjunto de datos se dividirá en 40 lotes, cada uno con cinco muestras. Los pesos del modelo se actualizarán después de cada lote de cinco muestras.
- Esto también significa que una época implica 40 lotes o 40 actualizaciones del modelo.
- Con 1000 épocas, el modelo estará expuesto o pasará por todo el conjunto de datos 1000 veces. Es decir, un total de 40.000 lotes durante todo el proceso de entrenamiento.

Conceptos Importantes:

FUNCIÓN DE COSTO

Es una **función matemática** que cuantifica la diferencia entre la salida predicha y el valor actual del target. El objetivo de la función de costo **es medir qué tan bueno es el desempeño del modelo** y guiar el proceso de aprendizaje proporcionando una medida del error entre los valores predichos y los reales.

Se calcula una sola vez en un ciclo de entrenamiento. Las más comunes son:

- Mean squared error (regresión).
- Mean absolute error (regresión)
- Binary Cross Entropy (clasificación).
- Hinge (clasificación)

Mean squared error

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(y_{\text{pred}_i} - y_i)^2$$

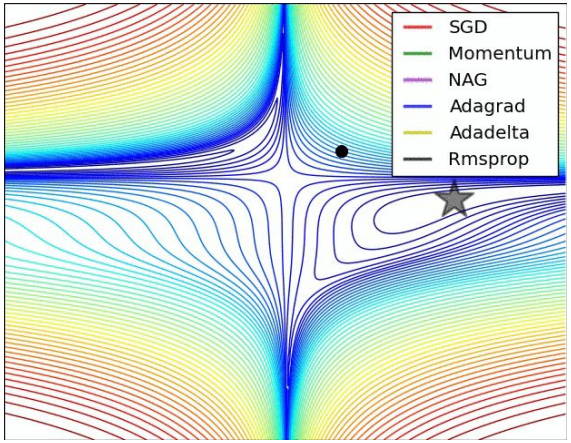
Binary Cross Entropy

$$J(\theta) = -\sum_{i=1}^{n}\underbrace{\left[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right]}_{\text{cost}(h_\theta(x^{(i)}),y^{(i)})}$$

Conceptos importantes

OPTIMIZADORES

Es un algoritmo o método usado para **actualizar los parámetros o pesos del modelo** con el objetivo de **minimizar la función de costo** y mejorar el desempeño durante el entrenamiento (en general, son variaciones del algoritmo del descenso del gradiente).

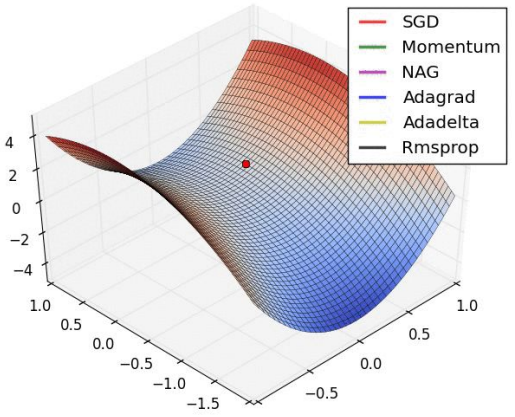


Algunos ejemplos:

- Descenso del Gradiente Estocástico (SGD)
- Adam
- RMSprop
- Adagrad
- etc

¿En qué cambian?

- Reglas de actualización
- Tasas de aprendizajes
- Momento
- etc



Para explorar el tema en profundidad

Conceptos Importantes:

TASA DE APRENDIZAJE (Learning rate)

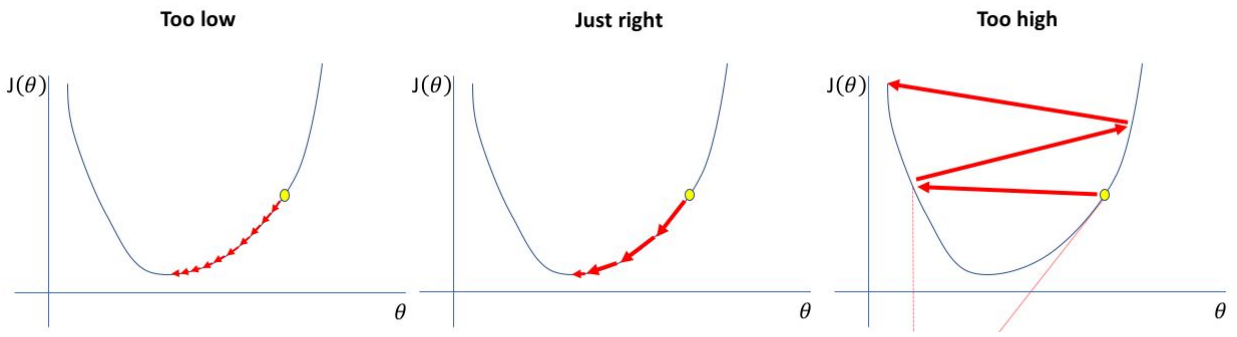
Es un hiper parámetro que controla **cuánto ajustamos** los pesos de nuestra red con respecto al gradiente al minimizar la función de costo.

Cuanto menor sea el valor, más lento avanzaremos en la búsqueda del mínimo local.

$$W_x = W_x - \varepsilon \frac{\delta \text{Error}}{\delta W_x}$$

Los valores típicos de learning rate suelen estar en el rango de 0.1 a 0.0001, pero esto puede variar según el problema y la arquitectura de la red.

Valores como 0.1 ó 0.0.1 pueden ser un buen punto de entrada.



Una tasa de aprendizaje pequeña requiere muchas actualizaciones para alcanzar el mínimo

La tasa de aprendizaje óptima alcanza rápidamente el punto mínimo

Una tasa de aprendizaje muy alta hace actualizaciones drásticas que terminan divergiendo

Conceptos importantes: Algunos optimizadores

DESCENSO DEL GRADIENTE

GD calcula el gradiente de la función de pérdida con respecto a los parámetros del modelo y los actualiza restando una fracción del gradiente (tasa de aprendizaje). Converge al mínimo si la tasa de aprendizaje se elige cuidadosamente. Sin embargo, ***puede quedarse estancado en mínimos locales.***

DESCENSO DEL GRADIENTE ESTOCÁSTICO

SGD actualiza los parámetros con el gradiente de pérdida calculado para un mini lote de datos aleatorio. La convergencia es más rápida en comparación con el GD , ***escapa a los mínimos locales y es bueno para conjuntos de datos grandes.*** Puede requerir un mayor ajuste de la tasa de aprendizaje

ADAM

La idea básica detrás de la optimización de Adam es ajustar la ***tasa de aprendizaje de forma adaptativa*** para cada parámetro del modelo en función del historial de gradientes calculados para ese parámetro. Esto ayuda al optimizador a converger más rápido y con mayor precisión que los métodos de tasa de aprendizaje fija.

ADAGRAD

Adapta las tasas de aprendizaje individualmente para cada parámetro en función de su información de gradiente histórico. Sin embargo, las tasas de aprendizaje pueden llegar a ser muy pequeñas y es posible que no converjan para todos los problemas.

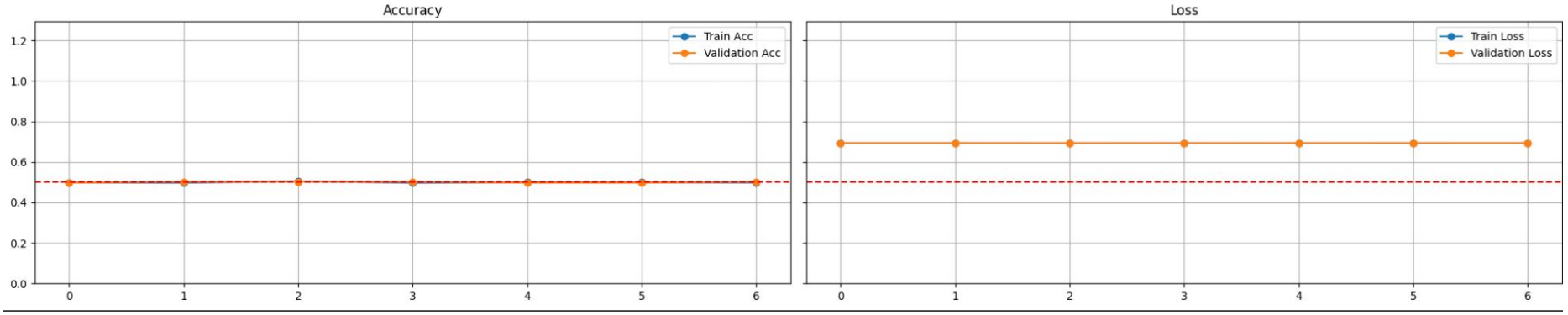
Conceptos importantes: Curvas de entrenamiento

Son representaciones gráficas que muestran cómo cambia el rendimiento de un modelo durante el proceso de entrenamiento (a lo largo de las epochs) tanto en el conjunto de entrenamiento como en el conjunto de validación.

Usualmente se usan para mostrar la loss y alguna métrica específica del problema (por ejemplo el accuracy). Se espera que la loss disminuya a lo largo de las épocas y el accuracy aumente.

Curvas “no saludables”

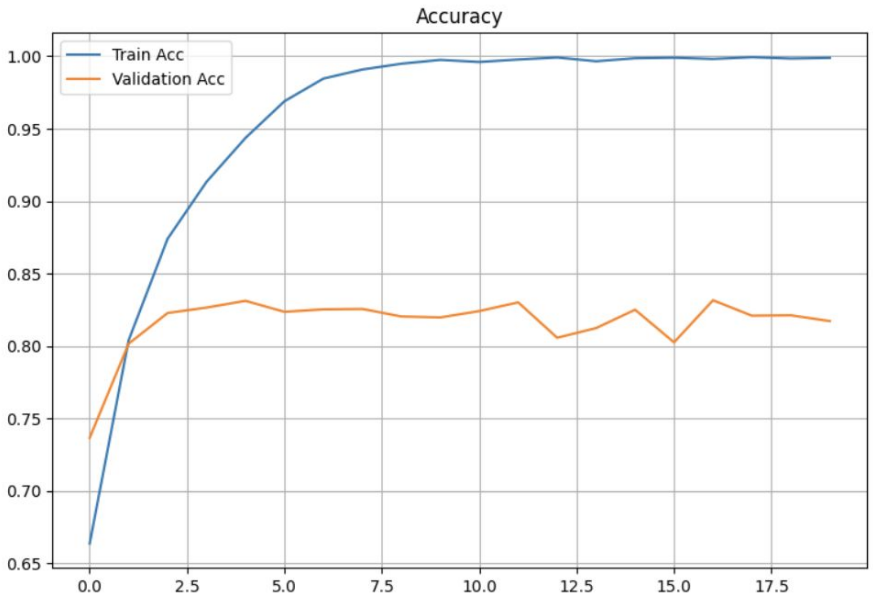
El modelo no está aprendiendo (tanto el accuracy como la loss se mantiene constante a lo largo de las épocas)

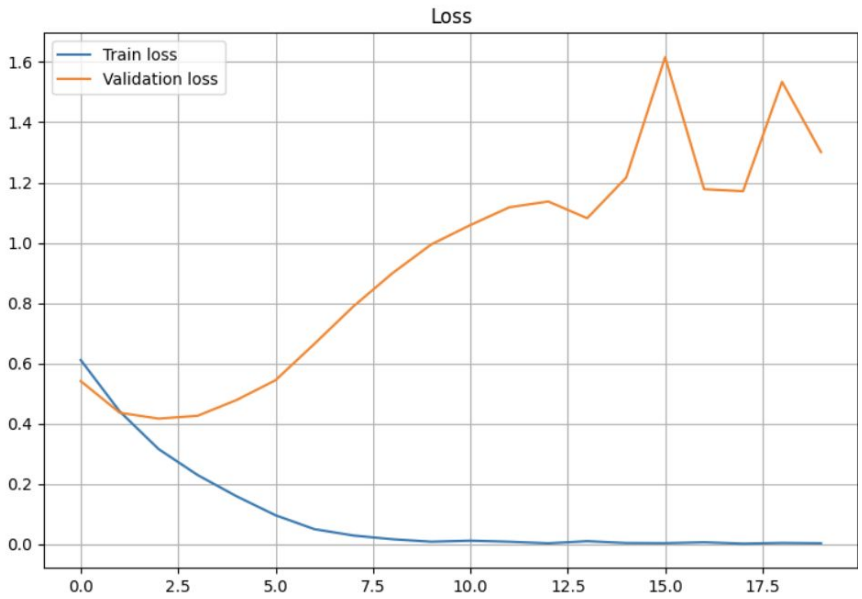


Conceptos importantes: Curvas de entrenamiento

Curvas “no saludables”

Hay un claro signo de **overfitting o sobreajuste**: el modelo funciona muy bien en los datos de entrenamiento pero en los datos de validación el accuracy es bajo y la loss empieza a crecer.



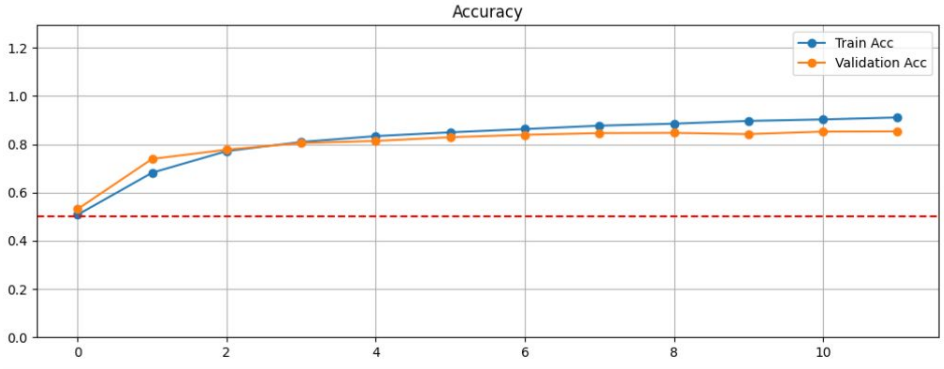


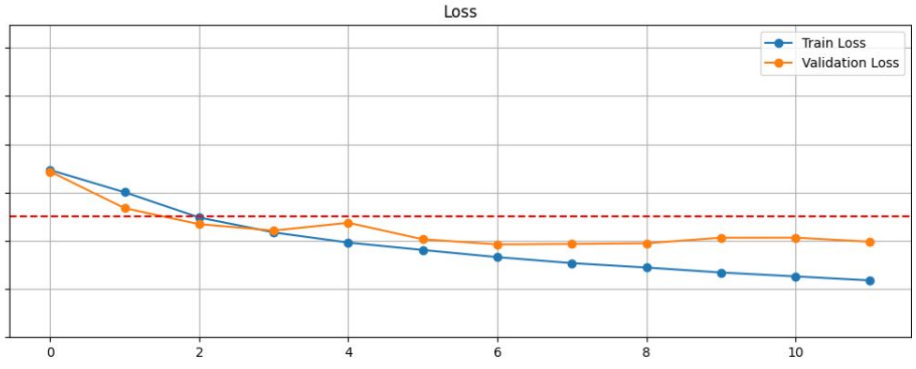
Conceptos importantes: Curvas de entrenamiento

Curvas “saludables”

Las curvas en entrenamiento y validación son parecidas a lo largo de las épocas:

- El accuracy crece (o la métrica del problema que estemos considerando)
- La loss baja



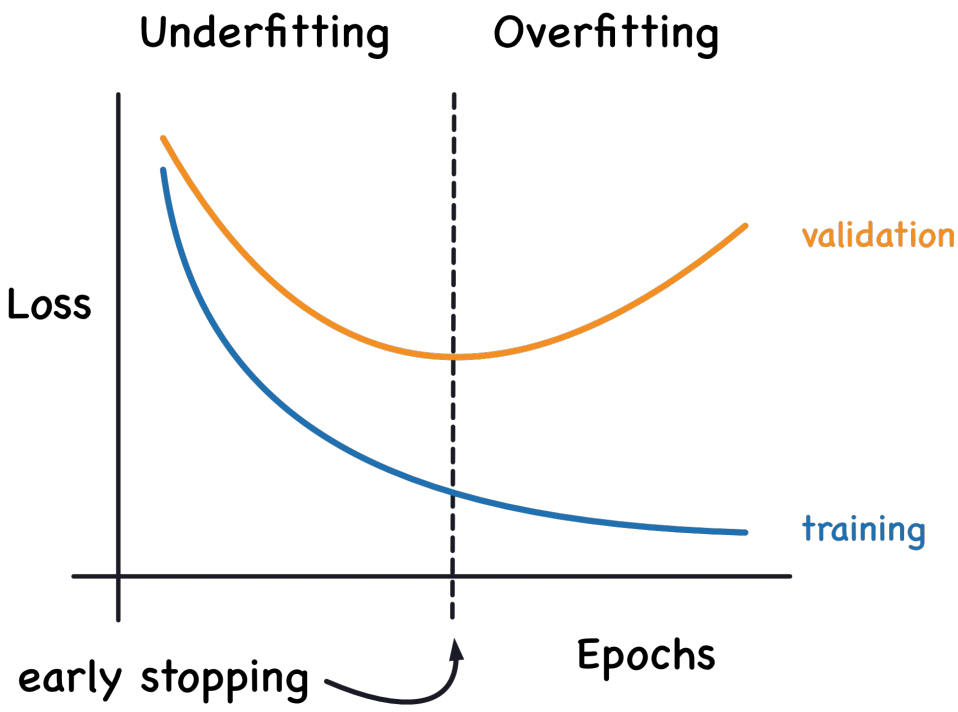


Conceptos importantes: Curvas de entrenamiento

Early stopping

Es una técnica que se utiliza para prevenir el sobreajuste u overfitting.

Detiene el entrenamiento cuando el rendimiento en los datos de validación deja de mejorar después de cierto número de épocas

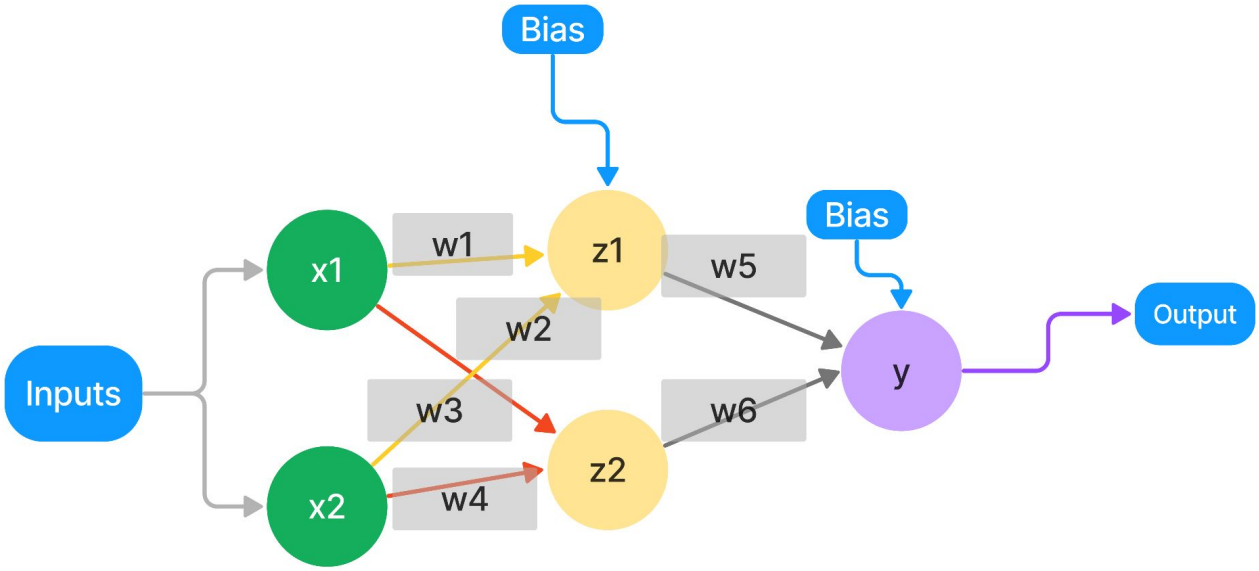


Parametros claves

Paciencia: El número de épocas que el entrenamiento continuará después de que la métrica de validación deje de mejorar. Si el modelo no mejora después de este número de épocas, el entrenamiento se detiene.

Métrica de evaluación: Puede ser la pérdida (loss) o alguna otra métrica, como la precisión o el F1-score, según el problema.

Ejemplo Práctico



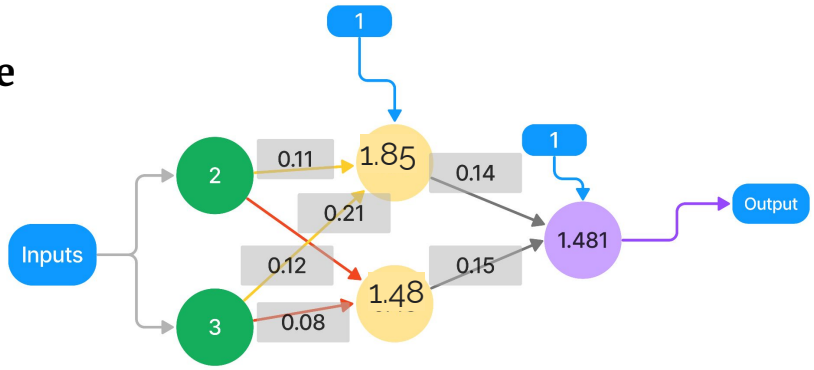
Capa de Entrada Capa Oculta Capa de Salida

Ejemplo Práctico

Inicializamos los pesos aleatoriamente y hacemos una iteración hacia adelante

Supongamos que la activación es lineal por simplicidad

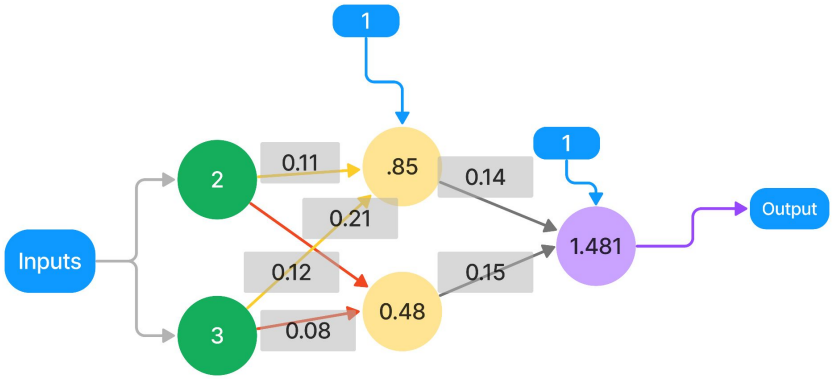
Supongamos que estamos resolviendo un problema de regresión...



$$\begin{bmatrix} 2 & 3 \end{bmatrix} * \begin{bmatrix} 0.11 & 0.12 \\ 0.21 & 0.08 \end{bmatrix} + \begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{bmatrix} 1.85 & 1.48 \end{bmatrix} * \begin{bmatrix} 0.14 \\ 0.15 \end{bmatrix} + \begin{bmatrix} 1 \end{bmatrix} = \begin{bmatrix} 1.481 \end{bmatrix}$$

Ejemplo Práctico

Calculamos el error



Capa de Entrada Capa Oculta Capa de Salida

Valor real

2

1

3

$$\text{Error} = \frac{1}{2}(1.481 - 1)^2 = 0.11$$

$$\text{Error} = \frac{1}{2}(\text{predicción} - \text{valor real})^2$$

Ejemplo Práctico

¿Cómo podemos reducir el error?

- El principal objetivo del entrenamiento es reducir la diferencia entre la predicción y el valor real.
- El output actual es siempre constante por lo tanto la única opción posible es cambiar el valor de la predicción.
- Para cambiar la predicción, necesitamos cambiar los pesos.

La **retropropagación** es un mecanismo para actualizar los pesos usando el descenso por el gradiente.

$$W_x = W_x - \varepsilon \frac{\delta \text{Error}}{\delta W_x}$$

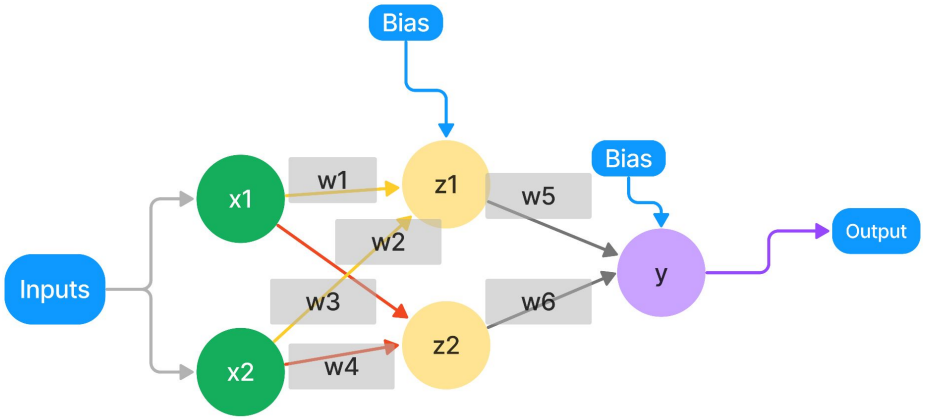
Ejemplo Práctico

Volvamos un poco atrás

$$\text{Error} = \frac{1}{2}(\text{predicción} - \text{valor real})^2$$

Función que depende de los pesos y los biases

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} * \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \end{bmatrix} = \begin{bmatrix} x_1 w_1 + x_2 w_2 + b_1 & x_1 w_3 + x_2 w_4 + b_2 \end{bmatrix} = \text{predicción}(w_1, w_2, w_3, w_4, b_1, b_2)$$



$$w_i = w_i - \varepsilon \frac{\partial \text{predicción}(w_1, w_2, w_3, w_4, b_1, b_2)}{\partial w_i}$$

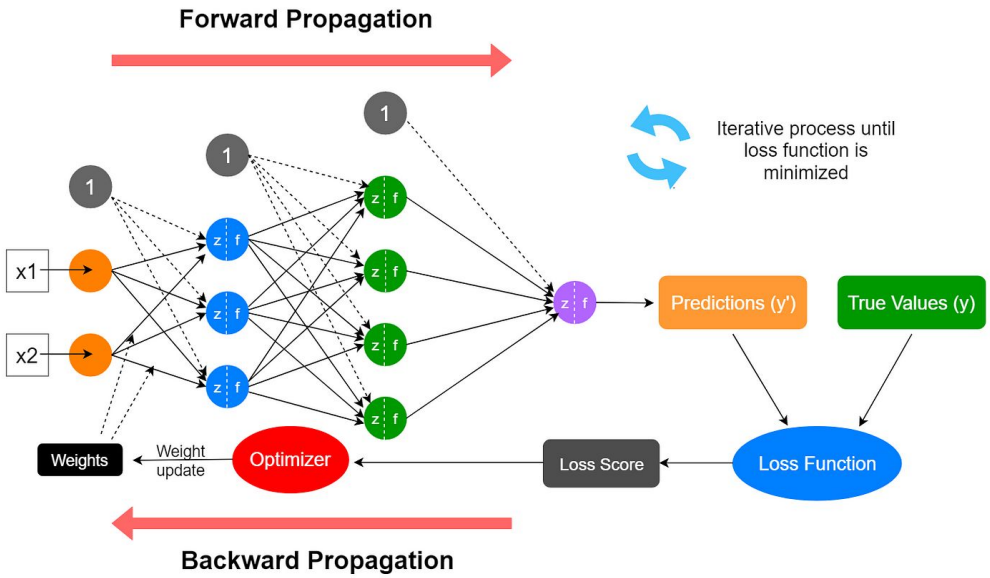
Teniendo la fórmula del error completa ya podemos derivar con respecto a los pesos usando la regla de la cadena 🫠, obtener los nuevos pesos y volver a iterar.

Por suerte pytorch lo hace por nosotros!!

PyTorch

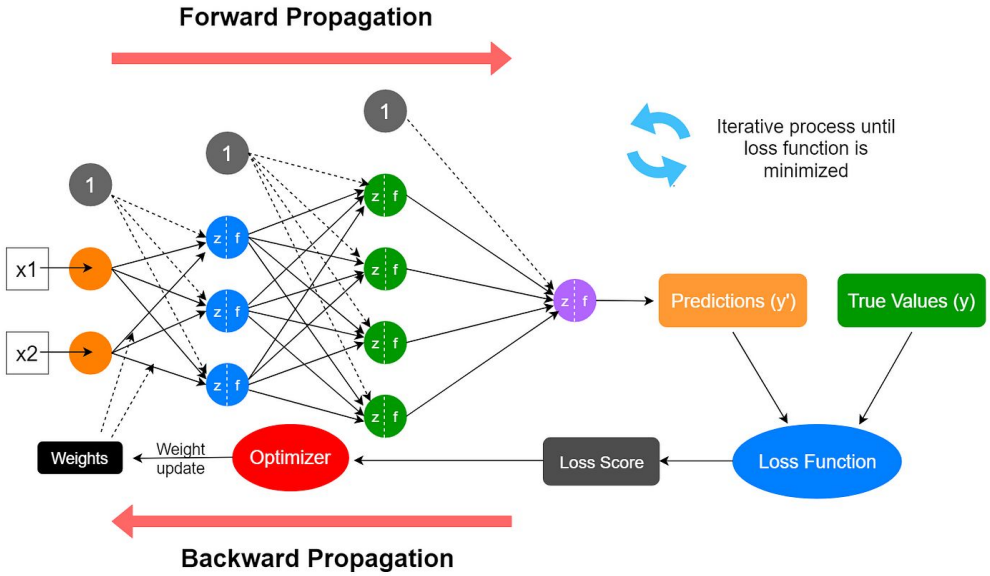
Resumen: Backpropagation

- El algoritmo de **retropropagación** permite calcular el gradiente de la función de costo (con respecto a los parámetros de *pesos* y *bias*) haciendo que la información "se mueva" hacia atrás en la red mediante un procedimiento sencillo y no costoso.
- De esta manera, ajusta los pesos y los valores de bias de la red en cada paso, minimizando la diferencia entre la salida actual y el valor real.
- El objetivo final es determinar cuáles caminos de la red son los más importantes en la predicción correcta del output final.
- El gradiente nos dice cuánto un parámetro necesita cambiar de manera de minimizar la función de costo.



Resumen: Backpropagation

1. Inicializar los pesos aleatoriamente.
2. Hacer un paso hacia adelante y calcular la predicción.
3. Hacer un paso hacia atrás:
   a. Calcular el gradiente de la función de costo con respecto a los pesos y biases
   b. Actualizar a los nuevos valores de pesos y biases.
4. Volver al paso 1 con los nuevos valores



Backpropagation: Ejemplo Práctico

```python
class MyFirstPytorchModel(nn.Module):
    def __init__(self,
                 input_features = 4, # Cantidad de features de entrada
                 hidden_layer_1 = 8, # Cantidad de neuronas de la primera capa oculta
                 hidden_layer_2 = 9, # Cantidad de neuronas de la segunda capa oculta
                 output_features = 3 # Dimensión de la salida: ¿Cuántas clases quiero predecir?
                 ):
        super().__init__() # Llama al método __init__ de la clase nn.Module
        # Generamos una red con 3 capas lineales (la última es de salida)
        self.fully_connected_1 = nn.Linear(input_features, hidden_layer_1)
        self.fully_connected_2 = nn.Linear(hidden_layer_1, hidden_layer_2)
        self.output = nn.Linear(hidden_layer_2, output_features)

    # Foward pass
    def forward(self, x):
      x = F.relu(self.fully_connected_1(x))
      x = F.relu(self.fully_connected_2(x))
      x = self.output(x)
      return x
```

```python
# Para hacer que el entrenamiento sea reproducible
torch.manual_seed(seed)

model     = MyFirstPytorchModel() #Instanciamos la clase
criterion = nn.CrossEntropyLoss() # Función de loss para problemas multiclase
optimizer = torch.optim.Adam(model.parameters(), lr = 0.1) # Definimos el optimizador
model.parameters
```

```
<bound method Module.parameters of MyFirstPytorchModel(
  (fully_connected_1): Linear(in_features=4, out_features=8, bias=True)
  (fully_connected_2): Linear(in_features=8, out_features=9, bias=True)
  (output): Linear(in_features=9, out_features=3, bias=True)
)>
```

Backpropagation: Ejemplo Práctico

1. Inicializar los pesos aleatoriamente.
2. Hacer un paso hacia adelante y calcular la predicción.
3. Hacer un paso hacia atrás:
   a. Calcular el gradiente de la función de costo con respecto a los pesos y biases
   b. Actualizar a los nuevos valores de pesos y biases.
4. Volver al paso 1 con los nuevos valores

```python
#epoch = Un entrenamiento sobre todo el dataset
epochs = 100
losses = []
for epoch in tqdm.trange(epochs):
    epoch+=1

    # Foward pass y obtener la prediccion
    y_pred = model.forward(X_train)

    # Calcular la loss de cada epoca
    loss = criterion(y_pred, y_train)
    losses.append(loss.item())

    # No queremos imprimir los resultados de las 100 epocas asi que podemos poner una condicion
    if epoch%10 == 1:
       print(f'epoch: {epoch:2}  loss: {loss.item():10.8f}')

    optimizer.zero_grad() #Setea todos los gradientes de la epoca anterior, sino se acumulan!
    loss.backward() #Calcula la derivada de la loss respecto de los pesos y biases
    optimizer.step() #Actualiza todos los parametros del modelo (pesos y biases)
```

```
 21%|██        | 21/100 [00:00<00:00, 191.95it/s]epoch:  1  loss: 1.09526443
epoch: 11  loss: 0.30797487
epoch: 21  loss: 0.12414974
epoch: 31  loss: 0.09246645
epoch: 41  loss: 0.06528996
epoch: 51  loss: 0.06579033
100%|██████████| 100/100 [00:00<00:00, 339.20it/s]epoch: 61  loss: 0.06147761
epoch: 71  loss: 0.06083250
epoch: 81  loss: 0.06066041
epoch: 91  loss: 0.06020032
```

LET'S CODE

Notebook 2: Parte II

Material Complementario

P. II.1 2024 - Introducción a la programación orientada a objetos (POO)

Material Adicional

Preprocesamiento de Datos

Optimización de Hiper-parámetros

LET'S CODE

5. Optimización de Hiperparámetros