Aprendizaje automático

Redes neuronales (parte 1)

FACULTAD DE INGENIERÍA
UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Agenda

- Introducción
- Perceptrón
- Perceptrón multicapa (MLP)
- Propagación hacia atrás
- Funciones de activación
- Regresor y clasificador MLP
- Implementación en Keras y TensorFlow
- Hyperparámetros

Introducción

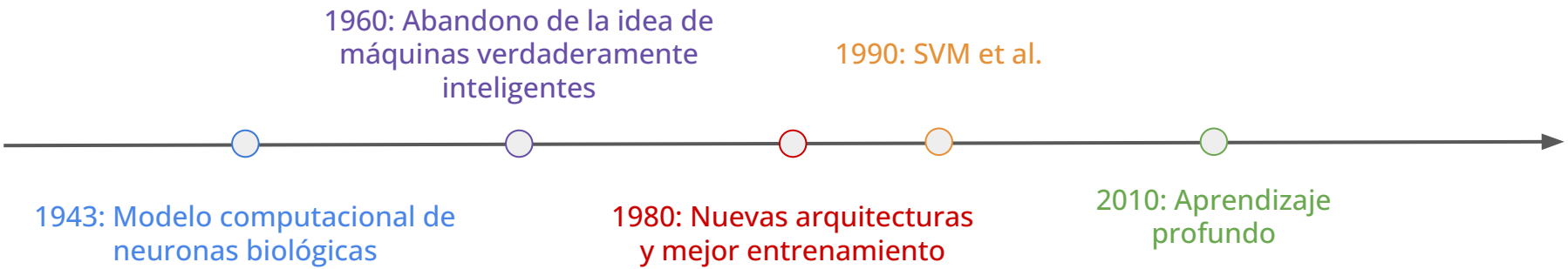
Introducción

- Las redes neuronales artificiales (ANN, *Artificial Neural Network*): modelo de aprendizaje automático inspirado en las redes de neuronas biológicas de nuestro cerebro.

- Base del aprendizaje profundo

- Versátil, potente y ampliable

- Ideal para enfrentarse a grandes y complejas tareas de aprendizaje automático
  - Clasificar billones de imágenes (Google Images)
  - Reconocimiento de voz (Apple’s Siri)
  - Recomendación de videos a millones de personas (YouTube)


1960: Abandono de la idea de máquinas verdaderamente inteligentes
1990: SVM et al.
1943: Modelo computacional de neuronas biológicas
1980: Nuevas arquitecturas y mejor entrenamiento
2010: Aprendizaje profundo

El aprendizaje profundo parece posible

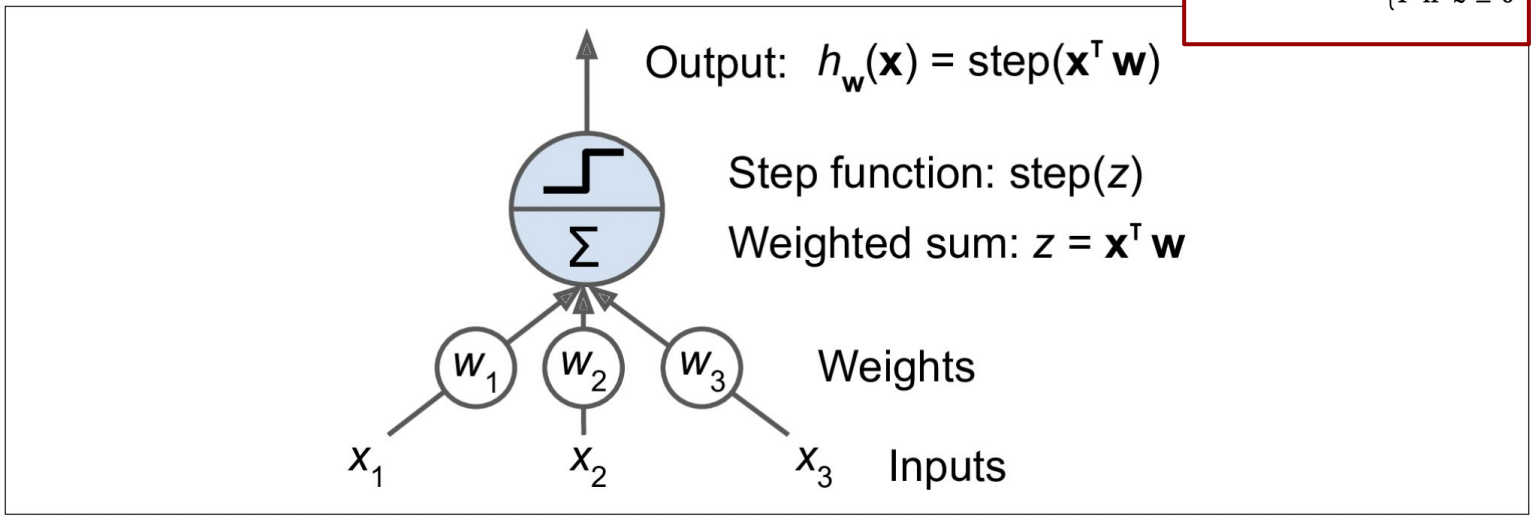
- Enorme cantidad de datos
- Aumento de la potencia informática (ley de Moore, industria del juego)
- Mejores algoritmos de entrenamiento
- Círculo virtuoso de financiación y progreso

Perceptrón

Perceptrón

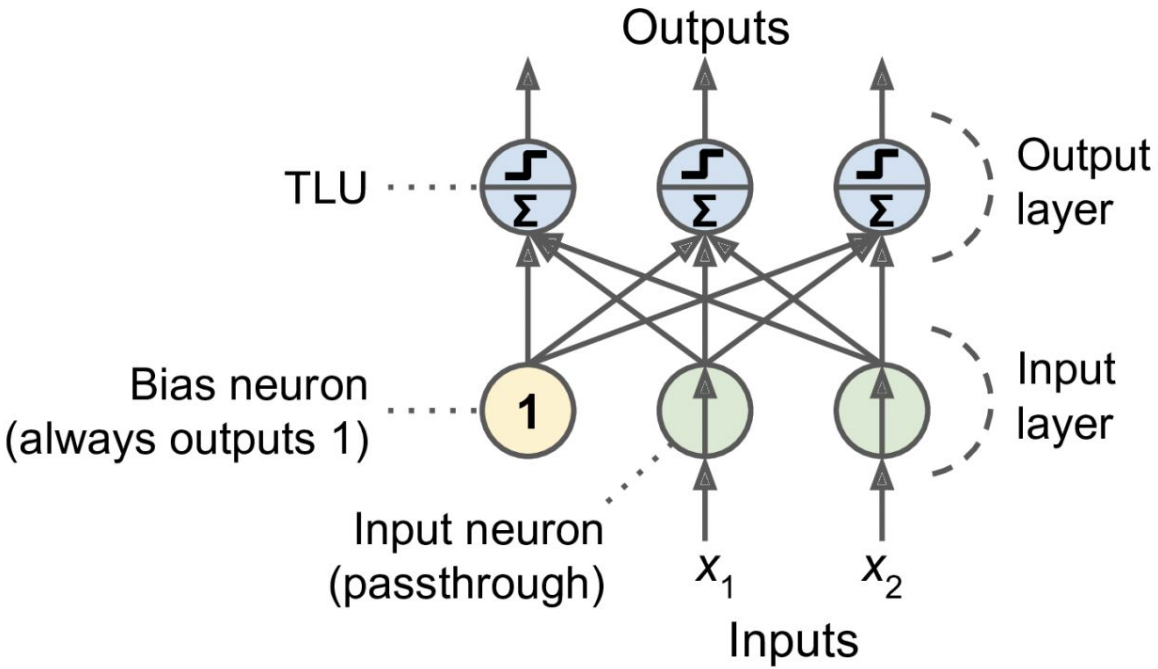
- Una de las arquitecturas más simples de las ANN (Rosenblatt 1957)
- Basado en una **neurona artificial** llamada TLU (Threshold Logic Unit)
- Las entradas y salidas son números
- La conexión a cada entrada está asociada a un **peso w_i**
- La salida es $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$ con $z = \mathbf{x}^T \mathbf{w}$ y step es la función escalón de Heaviside

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$



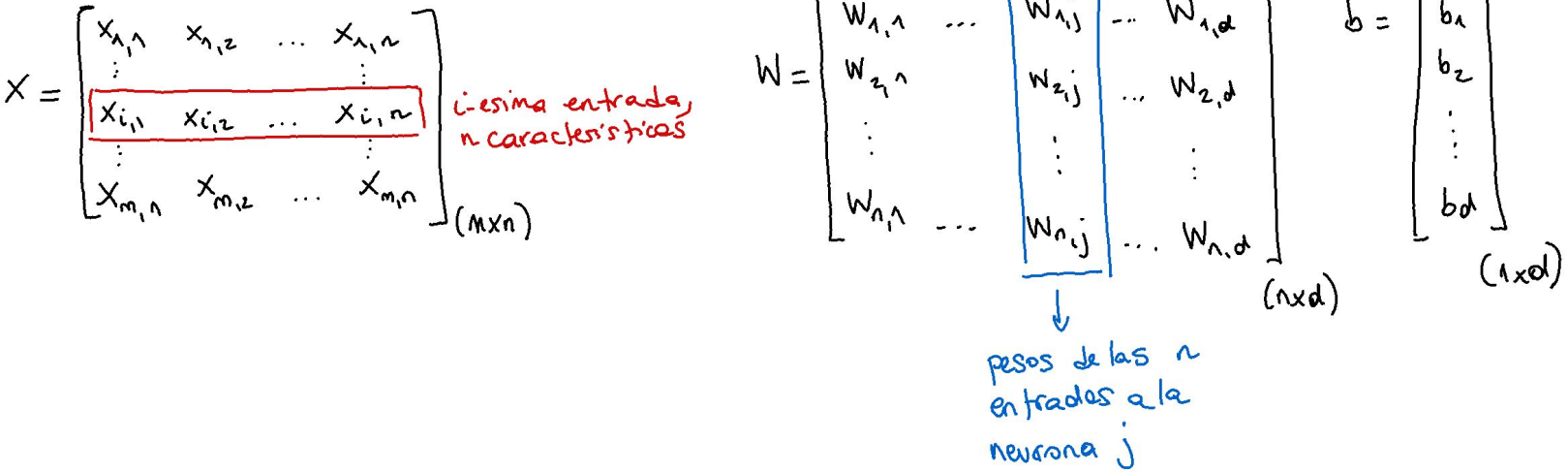
Perceptrón a salida multiple

- Capa con varias TLU
- Cada TLU se conecta a cada entrada (capa *fully connected* o *dense layer*)



Salida del perceptrón

- La salida es $h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$
- X matriz de características en entrada (m x n)
- W matriz de pesos (n x cantidad de neuronas)
- b vector de bias (1 x cantidad de neuronas)
- Φ es la **función de activación** (en este caso la función step)



Como se entrena

- La regla de aprendizaje del perceptrón refuerza las conexiones $\mathbf{w}_{ij}$ que ayudan a reducir el error

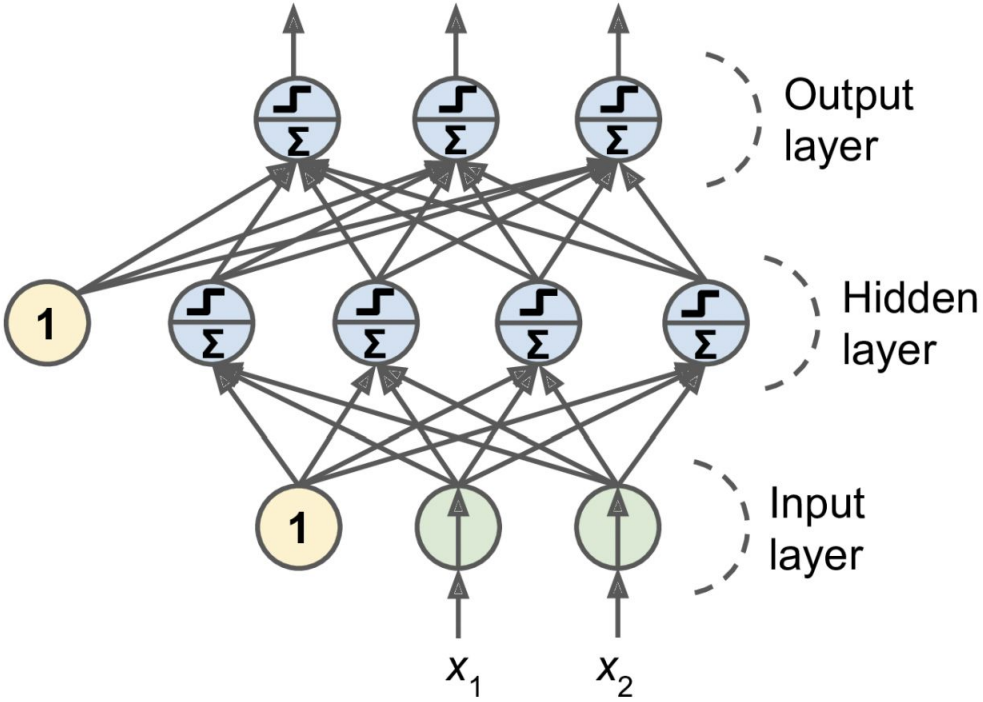
$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

- η es el *learning rate*
- La frontera de decisión de cada neurona el lineal
- Varias limitaciones entre ellas problemas triviales (como el XOR) no se puede resolver con un perceptrón
- Apilar varias capas de perceptrones: perceptrón multicapa (MLP)

Perceptrón multicapa

Perceptrón multicapa

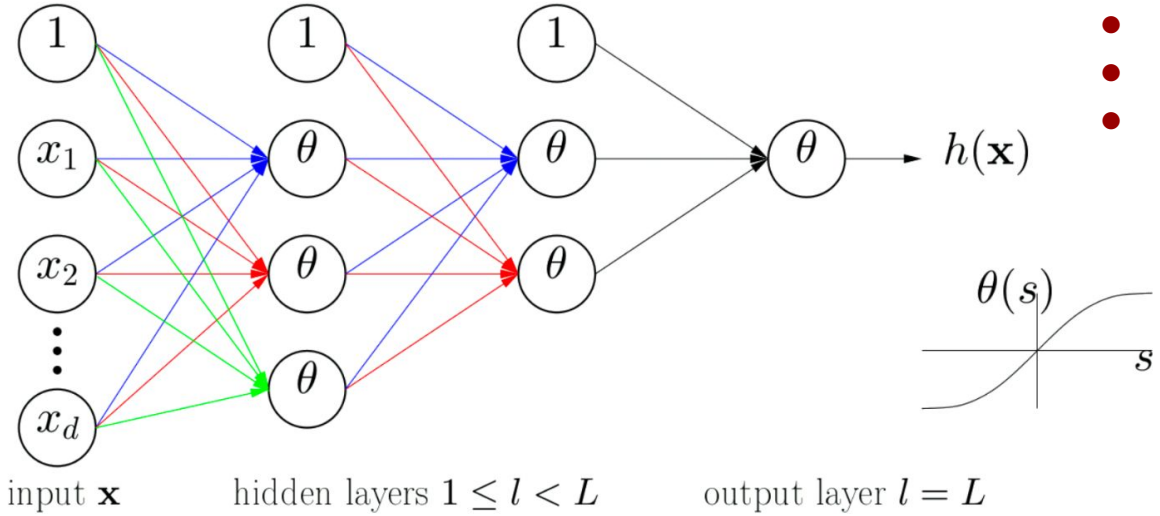
- Varias capas de perceptrones
- Una capa de entrada
- Una o varias capas ocultas (*hidden layers*)
- Una capa de salida
- Todas menos la capa de salida tienen una neurona para el bias
- Todas son capas densas
- Muchas capas ocultas → DNN (*Deep Neural Network*)



Perceptrón multicapa

Cuantos parámetros tiene esta red si $d^{(0)} = 10$?

- $3 \times (d^{(0)}+1)$
- $2 \times (d^{(1)}+1)$
- $d^{(2)}+1$
- Para $d^{(0)} = 10$, $d^{(1)} = 3$ y $d^{(2)} = 2 \rightarrow 44$ parámetros



$$x_j^{(l)} = \theta(s_j^{(l)}) = \theta\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}\right)$$

$$x_1^{(0)} \dots x_{d^{(0)}}^{(0)} \rightarrow \cdots \rightarrow x_1^L = h(\mathbf{x})$$

$$w_{ij}^{(l)} = \begin{cases} 1 \leq l \leq L & \text{layers} \\ 0 \leq i \leq d^{(l-1)} & \text{inputs} \\ 1 \leq j \leq d^{(l)} & \text{outputs} \end{cases}$$

¿Como se entrena un MLP?

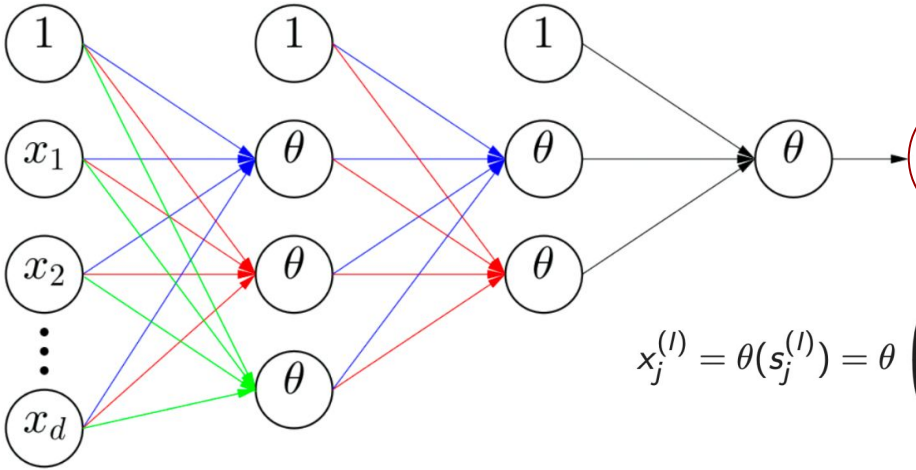
Propagación hacia atrás

Propagación hacia atrás (o *backpropagation* en inglés)

- *Backpropagation* (BP) es un método de GD que usa una forma eficiente de calcular el gradiente automáticamente
- BP calcula el gradiente de la función de error respecto a **todos los parámetros** de la red (w_{ij} y b_j) en dos pasadas: *forward pass* y *backward pass*
- Con el gradiente de la función de error se hace una etapa de bajada por gradiente **actualizando** w_{ij} y b_j
- Se itera hasta convergencia

- BP busca cómo ajustar cada peso de conexión w_{ij} y cada bias b_j para reducir el error de la red

Propagación hacia atrás



Composición de funciones de los parámetros de la red

$$x_j^{(l)} = \theta(s_j^{(l)}) = \theta\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}\right)$$

$e(w)$: función error de la red

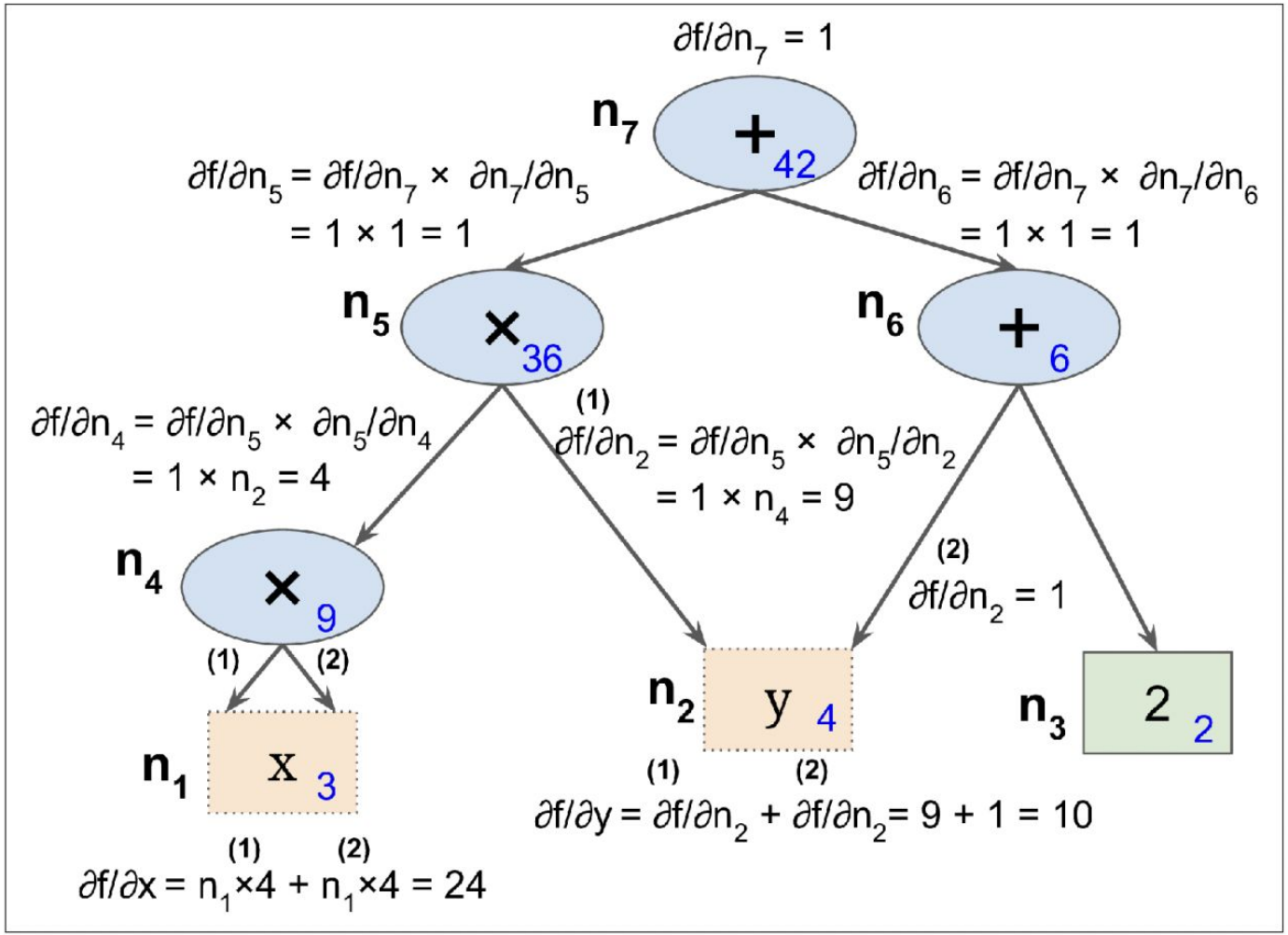
$$\nabla e(w): \frac{\partial e(w)}{\partial w_{ij}^{(l)}} \quad \forall i,j,l$$

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \eta \frac{\partial e(w)}{\partial w_{ij}^{(l)}} \quad \forall i,j,l$$

Regla de la cadena

$$y = f(u) = f(g(x))$$

$$\frac{dy}{dx} = \frac{du}{dx}\frac{dy}{du}$$


∂f/∂n7 = 1
n7
+ 42
∂f/∂n5 = ∂f/∂n7 × ∂n7/∂n5
= 1 × 1 = 1
∂f/∂n6 = ∂f/∂n7 × ∂n7/∂n6
= 1 × 1 = 1
n5
× 36
n6
+ 6
∂f/∂n4 = ∂f/∂n5 × ∂n5/∂n4
= 1 × n2 = 4
(1)
∂f/∂n2 = ∂f/∂n5 × ∂n5/∂n2
= 1 × n4 = 9
n4
× 9
(2)
∂f/∂n2 = 1
(1)
(2)
n2
y 4
n3
2 2
n1
x 3
(1)
(2)
∂f/∂y = ∂f/∂n2 + ∂f/∂n2= 9 + 1 = 10
(1)
(2)
∂f/∂x = n1×4 + n1×4 = 24

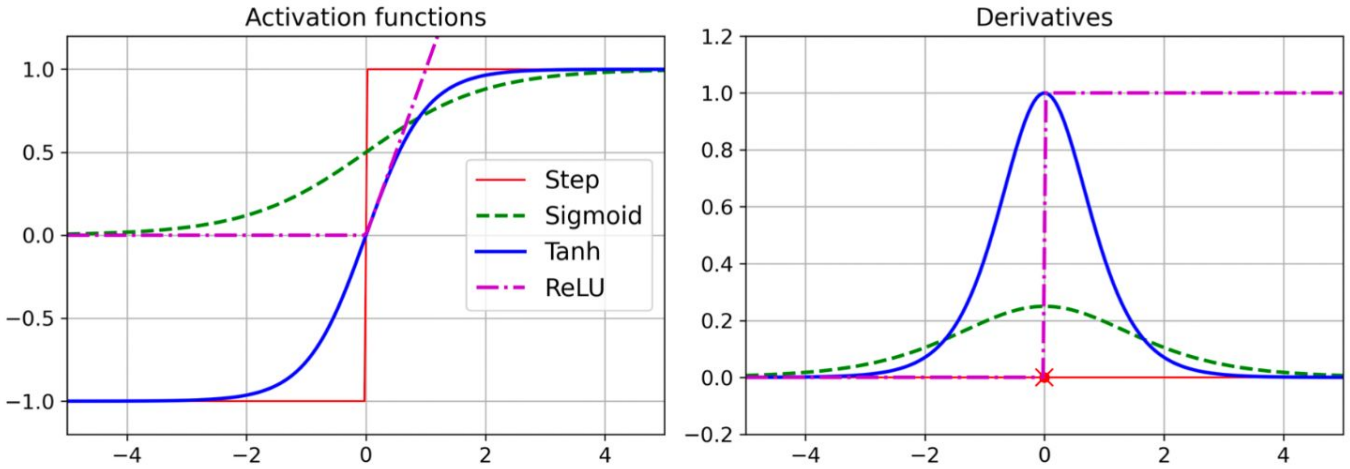
Propagación hacia atrás

- Se hace para cada instancia y varias veces por todo el conjunto de entrenamiento
- Cada pasada completa es lo que se conoce como una **época**
- *Forward pass*: predecir la salida para una instancia y guarda todos los resultados intermediarios
- Calcular el error e(w) (función de costo)
- *Backward pass*: calcula cuánto cada neurona contribuye al error e(w) para todas las capas (usando la regla de la cadena)
- Una vez que tenemos el gradiente se hace una etapa de bajada por gradiente
- Se repite el procedimiento hasta convergencia
- Se inicializan los pesos de manera aleatoria
- Esencial para que el algoritmo funcione correctamente: se cambia la función de activación a una sigmoid

$$\sigma(z) = 1 / (1 + \exp(-z))$$

Funciones de activación

- Sigmoid : $\theta(x) = \frac{1}{1+e^{-s}}$
- Tanh : $\theta(x) = \tanh(x) = \frac{e^{s}-e^{-s}}{e^{s}+e^{-s}}$
- ReLU : $\theta(x) = \max(0, x)$



¿Porque necesitamos funciones de activación?

Regresores y clasificadores con MLP

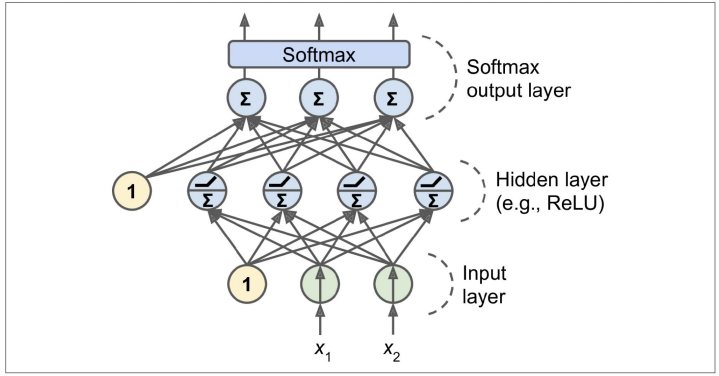
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Regresión

- Los MLP pueden utilizarse para regresión
- Determinar si se quiere una o varias salidas
- En general la última capa de neuronas no tiene función de activación
- Función de error típica: MSE, MAE, Huber

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Clasificación



- Los MLP pueden utilizarse para clasificación
- En la capa de salida, tantas neuronas como clases
- Clasificación binaria: sigmoid
- Clasificación más de dos clases: softmax
- Salida de la red: probabilidad de pertenencia a cada clase
- Función de error típica: entropía cruzada

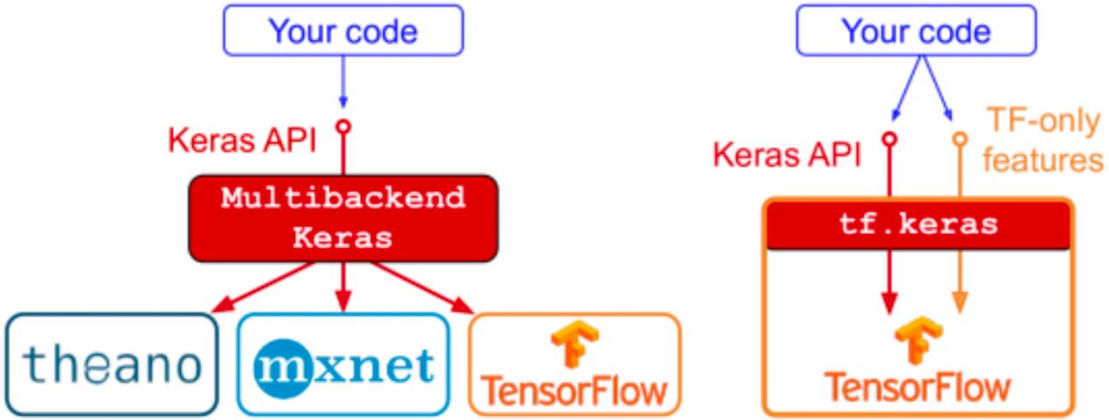
Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

Implementación Keras y TensorFlow

Keras y TensorFlow

Keras API de alto nivel para aprendizaje profundo

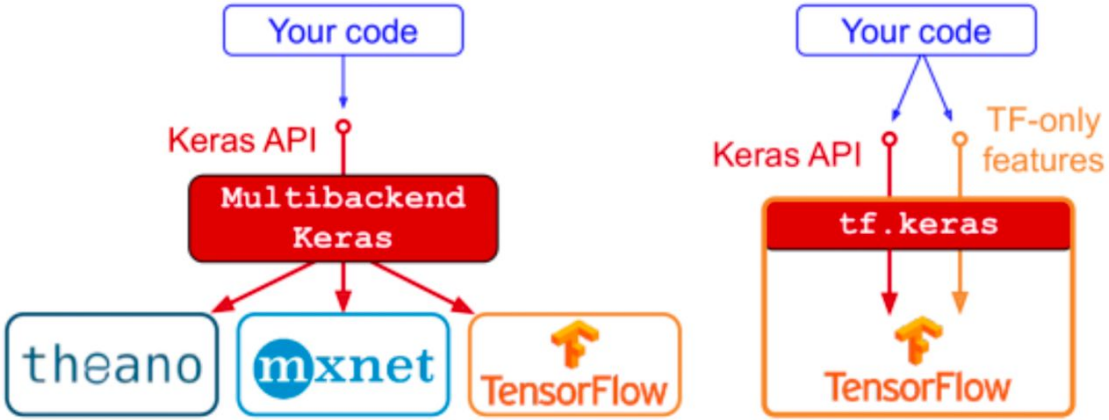
TensorFlow biblioteca para aprendizaje profundo



Keras y TensorFlow

Keras API de alto nivel para aprendizaje profundo

TensorFlow biblioteca para aprendizaje profundo




```
>>> import tensorflow as tf
>>> from tensorflow import keras
```

Clasificación de imágenes

```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
X_test = X_test / 255.
```

Definición del modelo

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
```

- Softmax

$$\sigma(\mathbf{s}(\mathbf{x}))_k = \frac{e^{s_k(\mathbf{x})}}{\sum_{j=1}^{K} e^{s_j(\mathbf{x})}}$$

- k = 10 clases
- Convierte k valores a que sumen 1
- Para clases excluyentes

$$\hat{y} = \underset{k}{\operatorname{argmax}}\, \sigma(\mathbf{s}(\mathbf{x}))_k$$

Inicialización de pesos

```
hidden1 = model.layers[1]
weights, biases = hidden1.get_weights()

>>> weights
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
         0.03859074, -0.06889391],
       ...,
       [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
         0.00272203, -0.06793761]], dtype=float32)

>>> weights.shape
(784, 300)

>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       ...,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)

biases.shape
(300,)
```

Compilación del modelo

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

cross-entropy loss

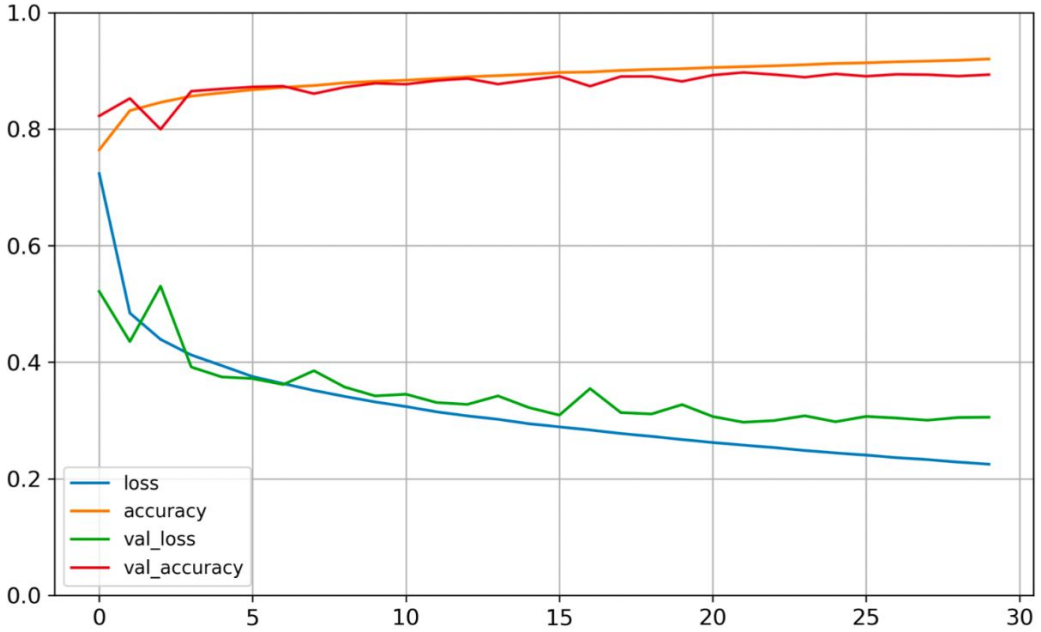
- `binary_crossentropy`
- `categorical_crossentropy`
- `sparse_categorical_crossentropy`

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{n=1}^{N}[y_n \log \hat{y}_n + (1 - y_n)\log(1 - \hat{y}_n)]$$

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_n^k \log \hat{y}_n^k$$

Entrenamiento del modelo

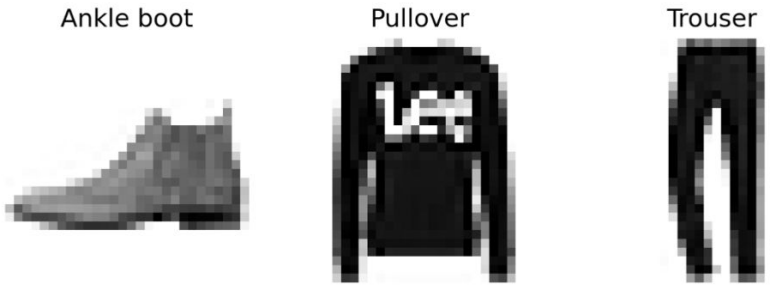
```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```



Evaluación del modelo

```
>>> model.evaluate(X_test, y_test)
313/313 [==============================] - 1s 2ms/step - loss: 0.3382 - accuracy: 0.8822
[0.3381877839565277, 0.8822000026702881]

>>> y_pred = np.argmax(model.predict(X_new), axis=-1)
>>> y_pred
array([9, 2, 1])
```



Guardando y cargando el modelo

```
model = keras.models.Sequential([...])
model.compile(...)
history = model.fit(..)

model.save("my_keras_model.h5")

model = keras.models.load_model("my_keras_model.h5")
model.predict(X_new)
```

Hiperparámetros

Hiperparámetros

- Hay varios hiperparámetros que ajustar
  - Número de capas
  - Número de neuronas
  - Tipo de función de activación
  - Inicialización de pesos
  - Learning rate
  - Optimizador
  - Batch size
  - Número de iteraciones
- Probar varias combinaciones
  - Grid search
  - Randomized search
- Varias librerías de Python
  - Hyperopt, Hyperas, kopt, Keras Tuner, ...
- La búsqueda de hiperparámetros sigue siendo un campo de investigación activo


FACULTAD DE
INGENIERÍA
UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY