.UBAfiuba
FACULTAD DE INGENIERÍA

Machine Learning en 2024

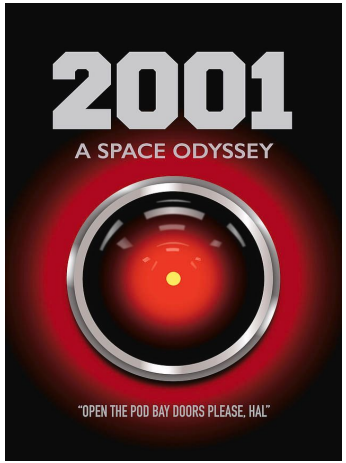
Asignatura: Taller de Procesamiento de Señales

Profesor: Matías Vera

Presentador: Lautaro Estienne

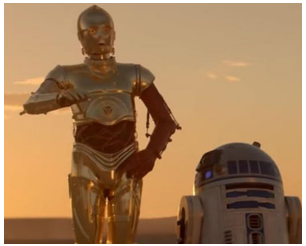
Panorama General

Inteligencia Artificial: Ideas viejas, técnicas nuevas



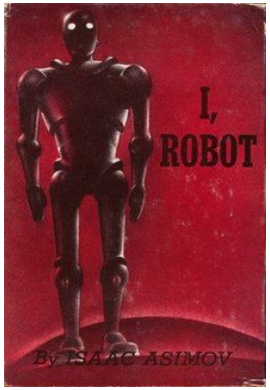
2001: A space Odyssey (1968)

The Terminator (1984)



Star Wars (1977)

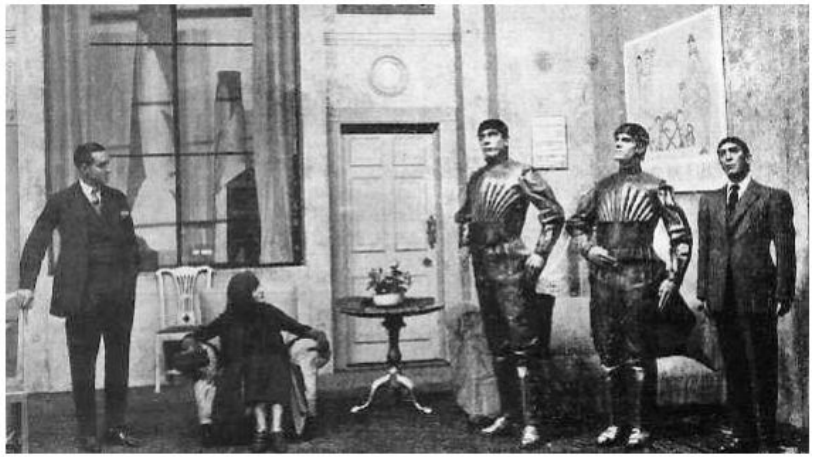
Back to the Future (1985)



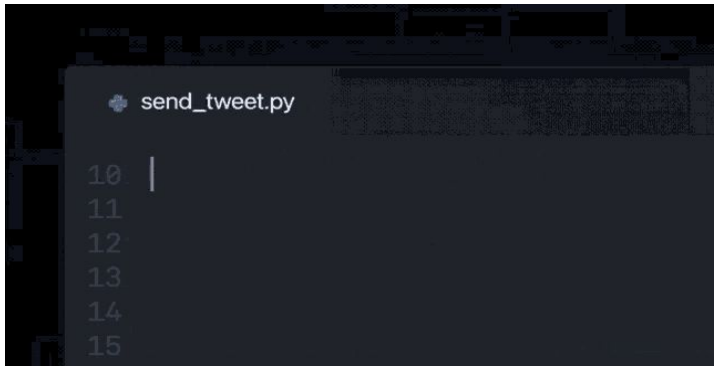
I, Robot (1950)

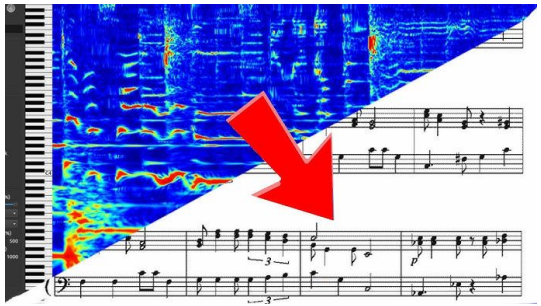
Inteligencia Artificial: Ideas viejas, técnicas nuevas

La palabra "robot" viene del checo, *robota*, que significa "esclavo".



Inteligencia Artificial hoy





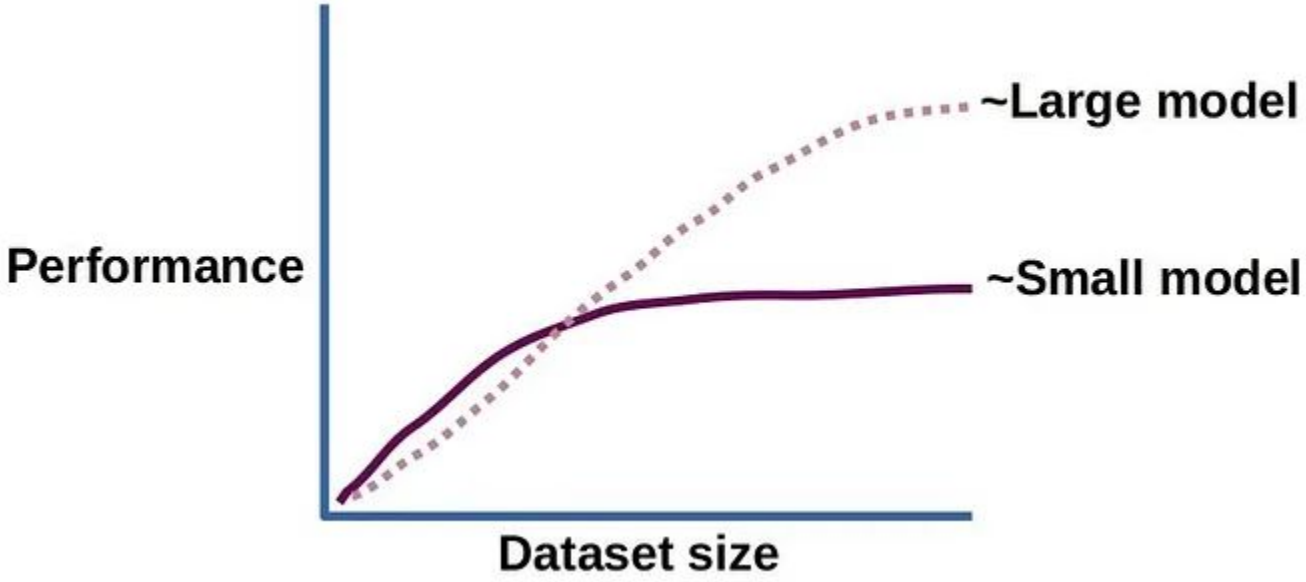
Campos de estudio

- Procesamiento del lenguaje natural (NLP)
- Procesamiento del habla
- Visión por computadora (CV)
- Procesamiento de señales (DSP)
- Robótica
- Neurociencias

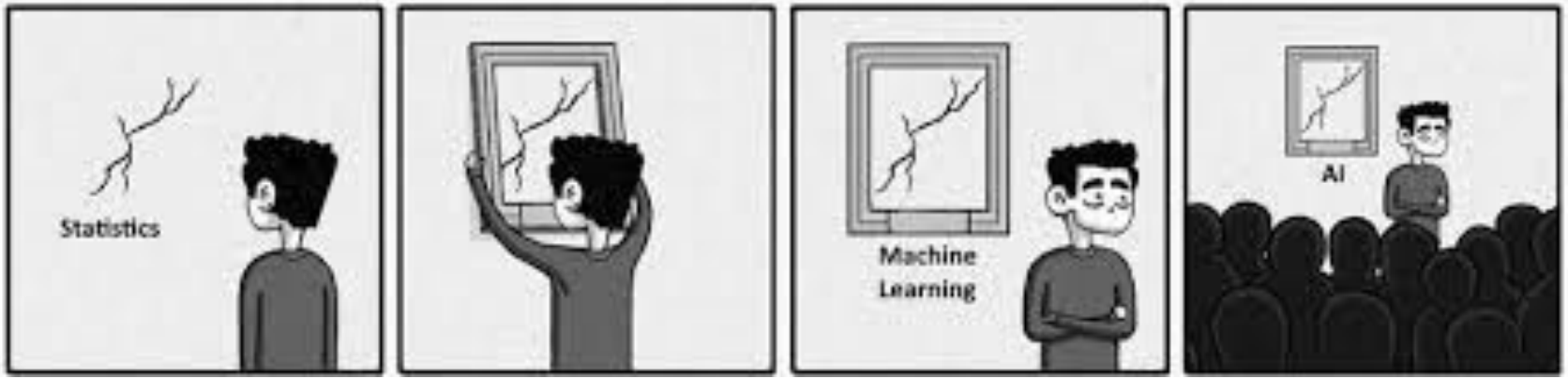
Inteligencia: Algunas características comunes

- Razonamiento y capacidad de resolver tareas
- Representación del conocimiento
- Capacidad de percibir con los sentidos
- Demostración de habilidades sociales
- **Aprendizaje de los datos**

Aprender de los datos



Herramienta común: probabilidad y estadística



Machine Learning: el mismo pipeline desde hace años

1. Qué aplicación quiero resolver.
2. Cómo voy a evaluar mi sistema.
3. Juntar datos para enseñarle a mi sistema.
4. Definir el algoritmo de aprendizaje que voy a usar
5. Entrenar, validar y repetir los pasos 3, 4 y 5 hasta que funcione.
6. Enviar a producción / evaluar en datos nunca vistos.

```
# Model
model = KNNClassifier()

# Data
train_data, test_data = load_data()

# Fit the data to the model
X, y = extract_features(train_data)
X_train, y_train, X_val, y_val = split_data(X, y)
model.fit(X_train, y_train)

# Validate
train_score = evaluate_model(model, X_train, y_train)
val_score = evaluate_model(model, X_val, y_val)
```

```
# Final evaluation
X_test, y_test = extract_features(test_data)
test_score = evaluate_model(model, X_test, y_test)
```

Machine Learning: el mismo pipeline desde hace años

1. Qué aplicación quiero resolver.

```
# Model
model = KNNClassifier()

# Data
train_data, test_data = load_data()

# Fit the data to the model
X, y = extract_features(train_data)
X_train, y_train, X_val, y_val = split_data(X, y)
model.fit(X_train, y_train)

# Validate
train_score = evaluate_model(model, X_train, y_train)
val_score = evaluate_model(model, X_val, y_val)
```

```
# Final evaluation
X_test, y_test = extract_features(test_data)
test_score = evaluate_model(model, X_test, y_test)
```

Machine Learning: el mismo pipeline desde hace años

2. Cómo voy a evaluar mi sistema.

```
# Model
model = KNNClassifier()

# Data
train_data, test_data = load_data()

# Fit the data to the model
X, y = extract_features(train_data)
X_train, y_train, X_val, y_val = split_data(X, y)
model.fit(X_train, y_train)

# Validate
train_score = evaluate_model(model, X_train, y_train)
val_score = evaluate_model(model, X_val, y_val)
```

```
# Final evaluation
X_test, y_test = extract_features(test_data)
test_score = evaluate_model(model, X_test, y_test)
```

Machine Learning: el mismo pipeline desde hace años

3. Juntar datos para enseñarle a mi sistema.

- Filtrar datos basura
- Dividir los datos representativamente de mi problema
- Ver qué características se pueden extraer de los datos

```
# Model
model = KNNClassifier()

# Data
train_data, test_data = load_data()

# Fit the data to the model
X, y = extract_features(train_data)
X_train, y_train, X_val, y_val = split_data(X, y)
model.fit(X_train, y_train)

# Validate
train_score = evaluate_model(model, X_train, y_train)
val_score = evaluate_model(model, X_val, y_val)
```

```
# Final evaluation
X_test, y_test = extract_features(test_data)
test_score = evaluate_model(model, X_test, y_test)
```

Machine Learning: el mismo pipeline desde hace años

4. Definir el algoritmo de aprendizaje que voy a usar.

```
# Model
model = KNNClassifier()

# Data
train_data, test_data = load_data()

# Fit the data to the model
X, y = extract_features(train_data)
X_train, y_train, X_val, y_val = split_data(X, y)
model.fit(X_train, y_train)

# Validate
train_score = evaluate_model(model, X_train, y_train)
val_score = evaluate_model(model, X_val, y_val)
```

```
# Final evaluation
X_test, y_test = extract_features(test_data)
test_score = evaluate_model(model, X_test, y_test)
```

Machine Learning: el mismo pipeline desde hace años

5. Entrenar, validar y repetir los pasos 3, 4 y 5 hasta que funcione.

- ¿En qué falla y por qué?
- ¿Si uso nuevos datos anda igual?
- ¿Es problema del modelo o del procesamiento de los datos?

```
# Model
model = KNNClassifier()

# Data
train_data, test_data = load_data()

# Fit the data to the model
X, y = extract_features(train_data)
X_train, y_train, X_val, y_val = split_data(X, y)
model.fit(X_train, y_train)

# Validate
train_score = evaluate_model(model, X_train, y_train)
val_score = evaluate_model(model, X_val, y_val)
```

```
# Final evaluation
X_test, y_test = extract_features(test_data)
test_score = evaluate_model(model, X_test, y_test)
```

Machine Learning: el mismo pipeline desde hace años

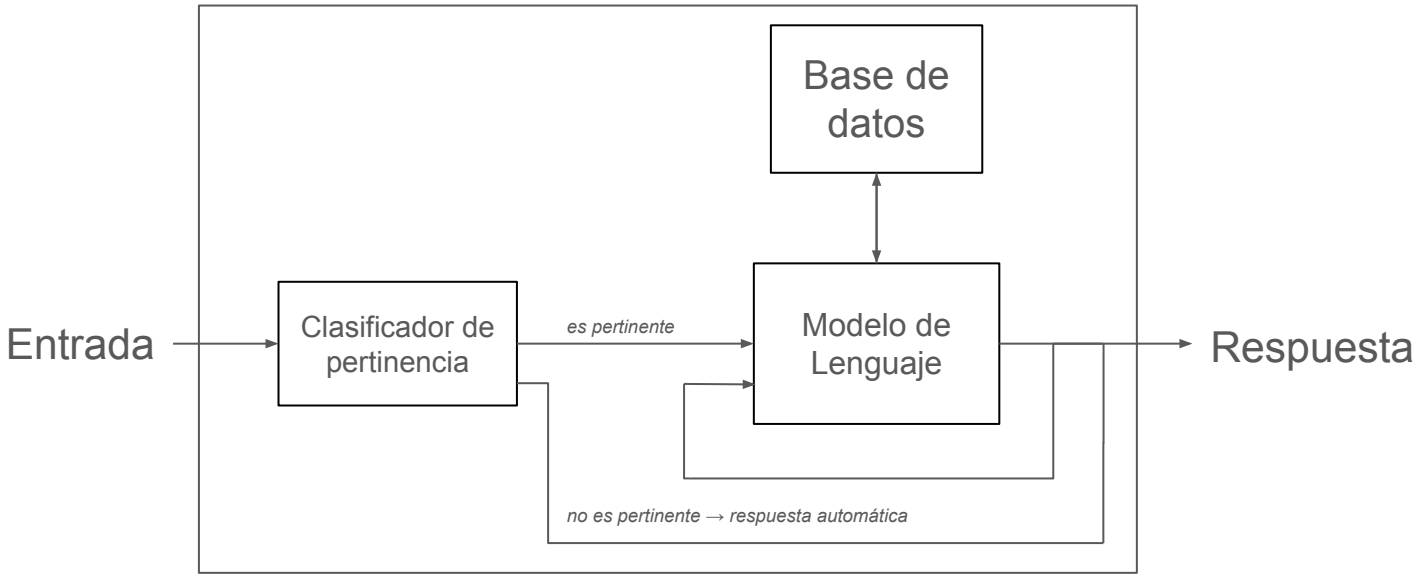
Enviar a producción / evaluar en datos nunca vistos.

```
def evaluate_model(model, X, y):
  output = model.predict(X)
  score = compute_score(output, y)
  return score

# Final evaluation
X_test, y_test = extract_features(test_data)
test_score = evaluate_model(model, X_test, y_test)

# Use in production
model.predict(sample)
```

Aplicaciones y evaluación

Ejemplo de aplicación: Chatbot

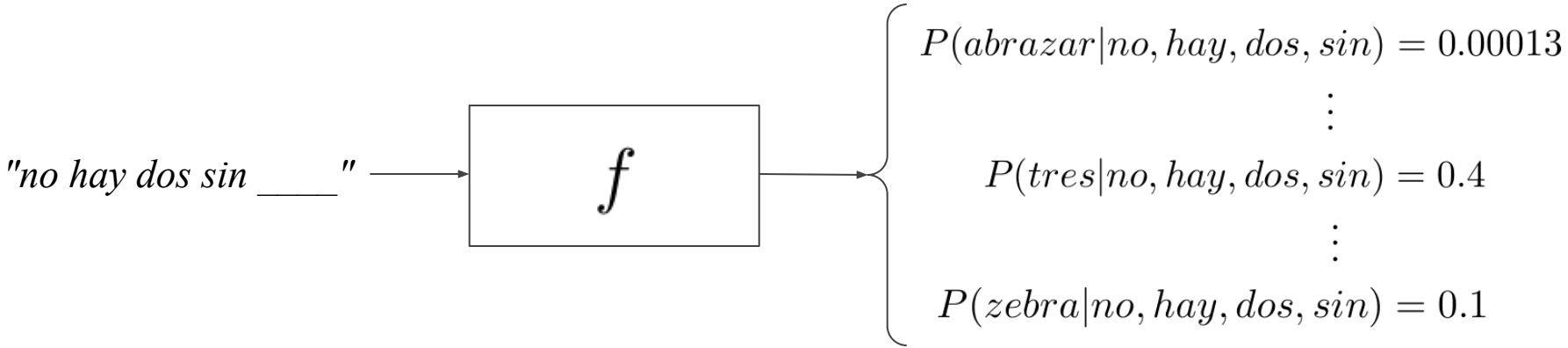
Caso típico en industria: chatbot asistencial en aplicaciones comerciales.



Modelos de Lenguaje

Un modelo de lenguaje autorregresivo predice la probabilidad de la próxima palabra.

$$V = \{a, abrazar, \ldots, hay, \ldots, tres, \ldots, zebra\}$$



Modelos de Lenguaje

Un modelo de lenguaje autorregresivo predice la probabilidad de la próxima palabra.

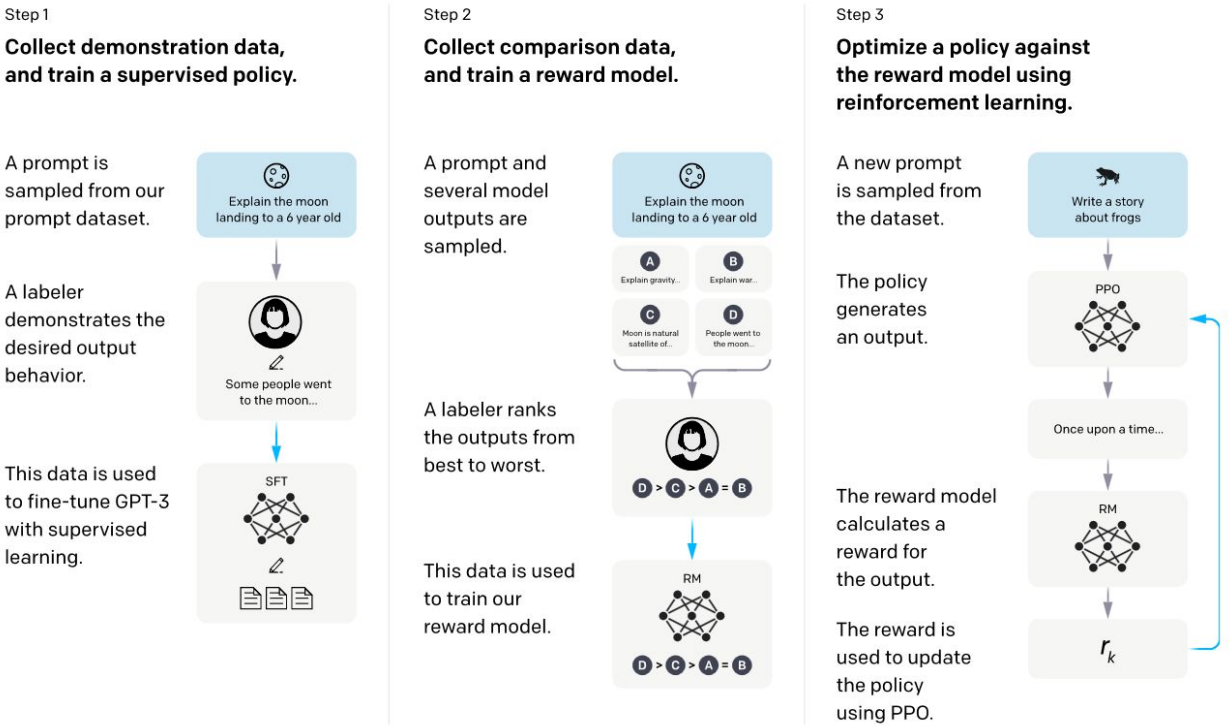
$$V = \{v_1, \ldots, v_n\} \qquad \mathcal{D} = \left\{ w_1^{(1)}, \ldots, w_{T_1}^{(1)}, \ldots, w_1^{(n)}, \ldots, w_{T_n}^{(n)} \right\}$$

$$w_1, \ldots, w_t \longrightarrow \boxed{\mathbf{W} \quad f_{\mathcal{D}}} \longrightarrow P(v|w_1, \ldots, w_t; \mathbf{W}) \quad \forall\, v \in V$$

$$L(\mathbf{W}) = \prod_{i=1}^{n} P(w_1^{(i)}, \ldots, w_{T_i}^{(i)}) = \prod_{i=1}^{n} \prod_{t=1}^{T_i} P(w_t^{(i)}|w_1^{(i)}, \ldots, w_{t-1}^{(i)})$$

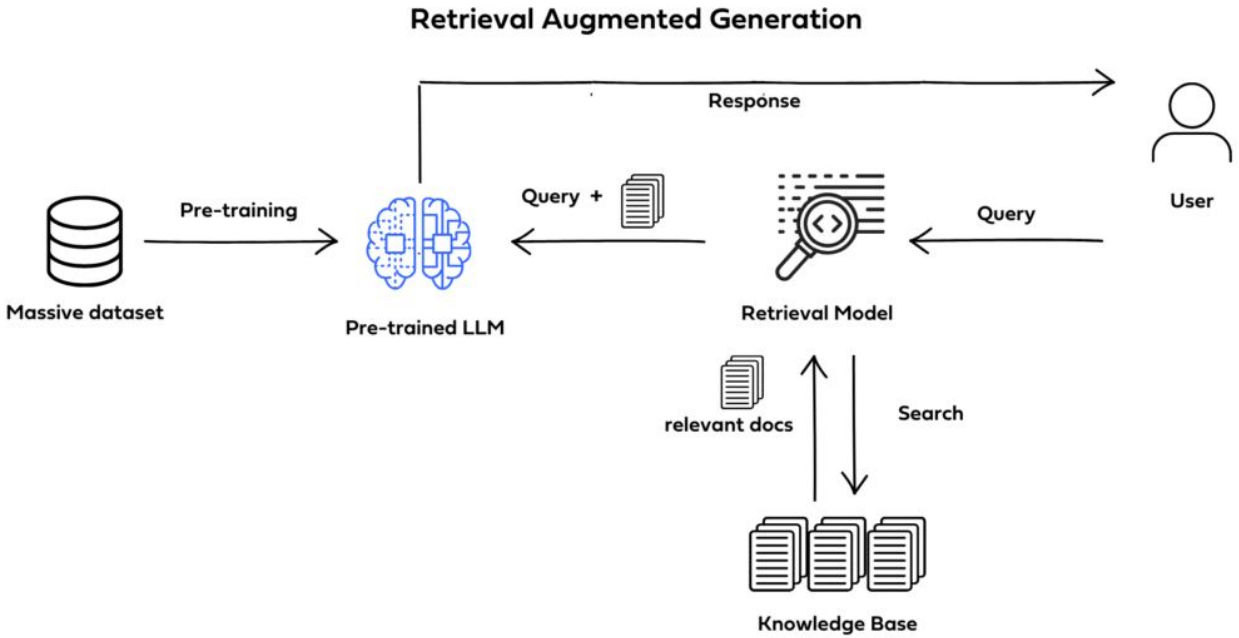
Paréntesis: ChatGPT

Reinforcement Learning from Human Feedback (*"Training language models to follow instructions with human feedback"*, Ouyang et al., 2022)



Paréntesis: Retrieval Augmented Generation (RAG)

Cada vez que el usuario escribe un prompt, ChatGPT agrega información de su base de datos para generar la respuesta.



Otros ejemplos de aplicación

- Detección de fallas en una cinta de producción.
- Seguimiento de trayectoria de un robot.
- Caso real en Salta (2018): predicción de embarazo adolescente.
- Detección temprana de patologías.
- Búsqueda de tópicos en redes sociales.
- Generación de imágenes / video

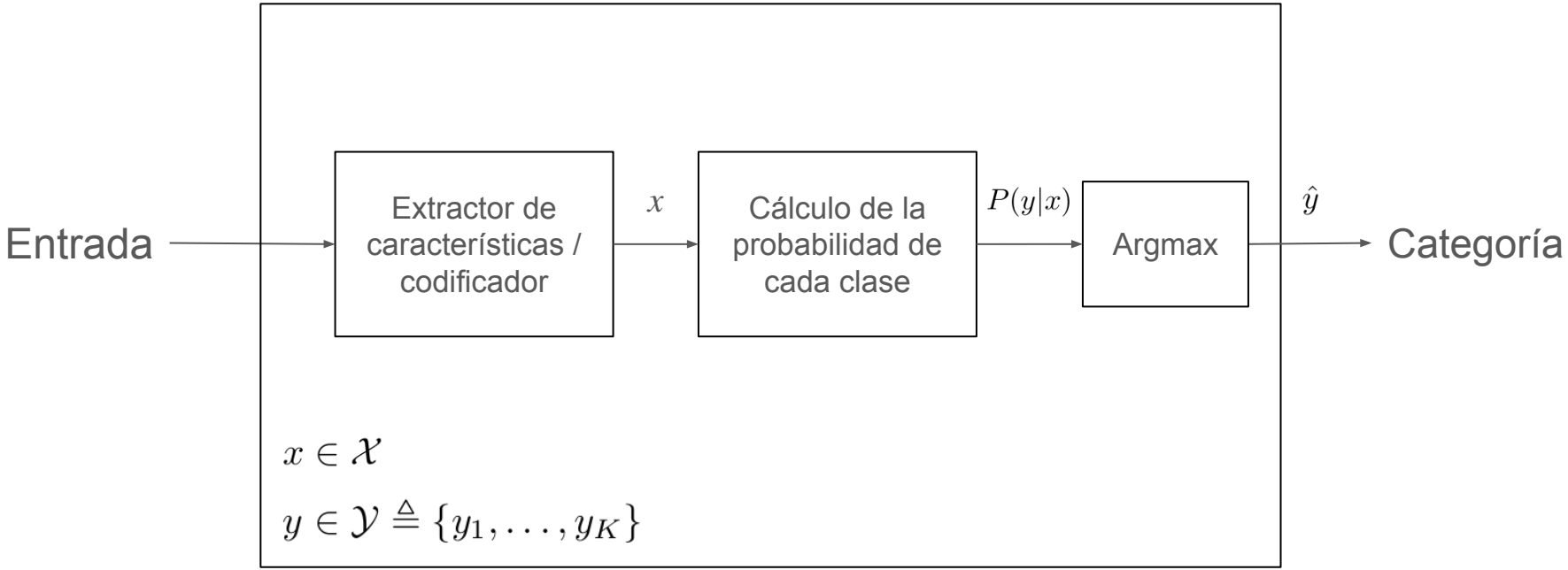
En general tengo que dividir el problema en tareas más pequeñas.

Tareas que se modelan probabilísticamente

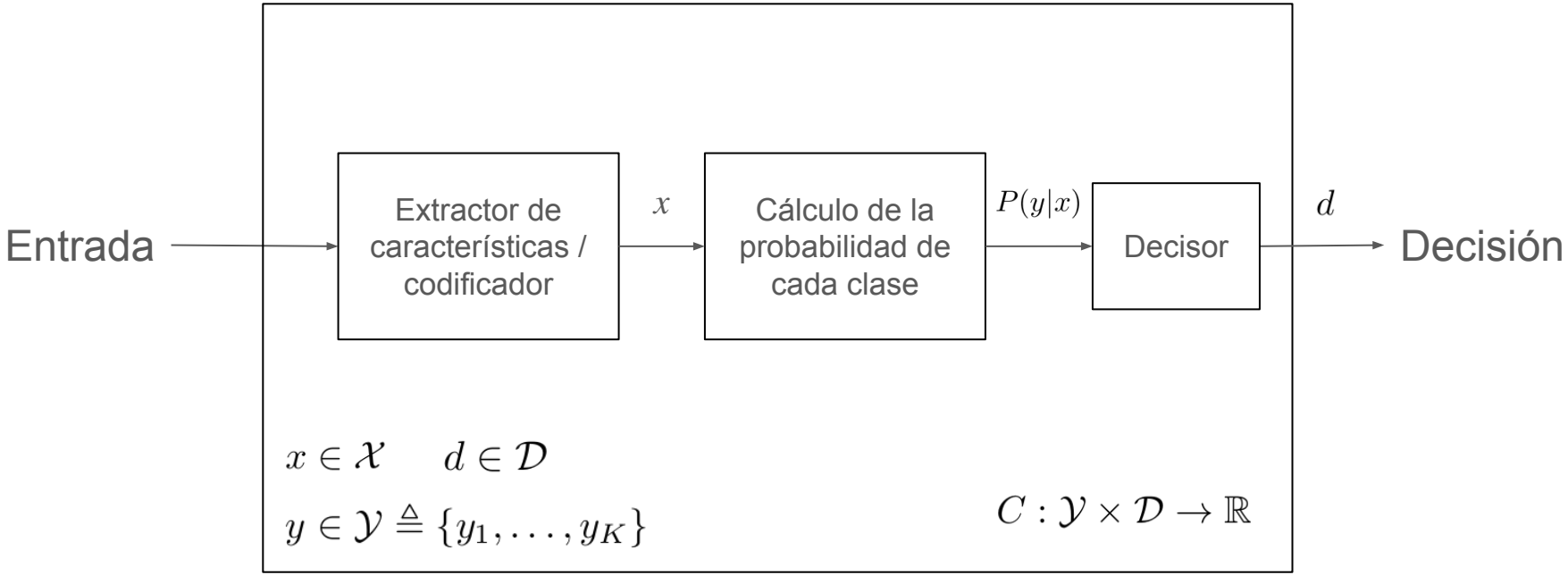
- Clasificación
- Toma de decisiones categóricas
- Regresión
- Clustering
- Reducción de la dimensionalidad
- Filtrado estocástico
- Generación secuencial

Clasificación

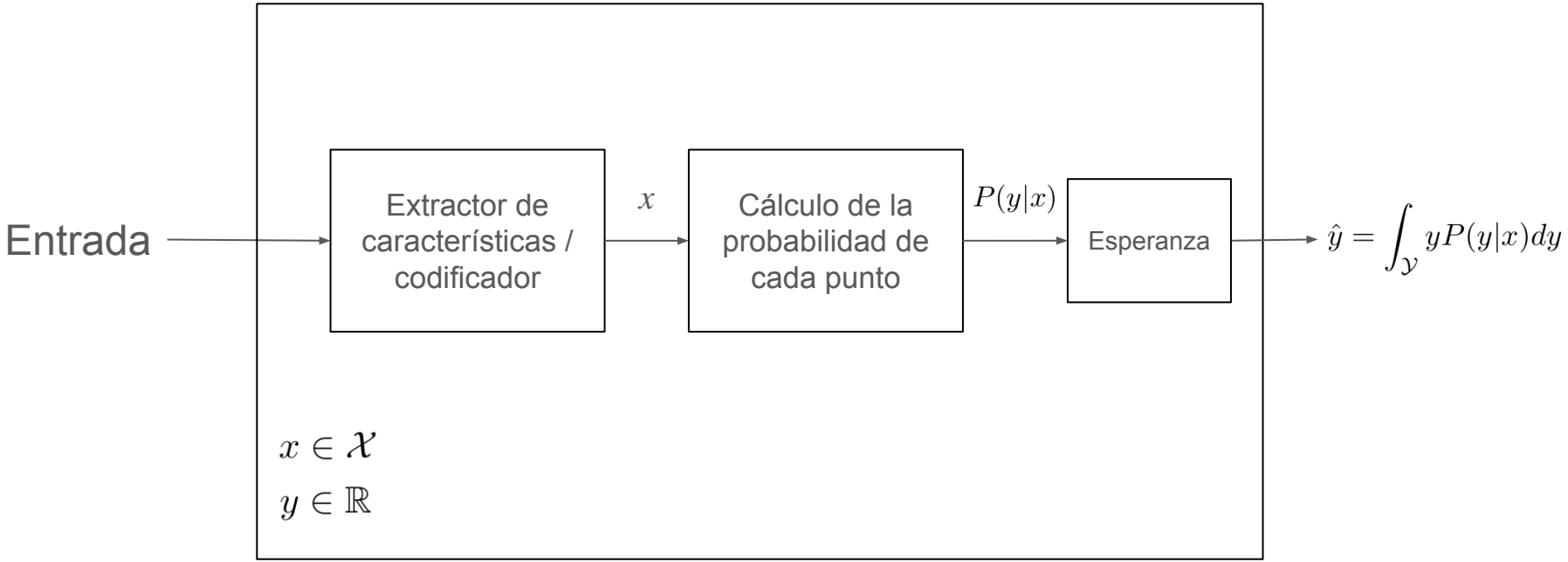
Clasificación Probabilística



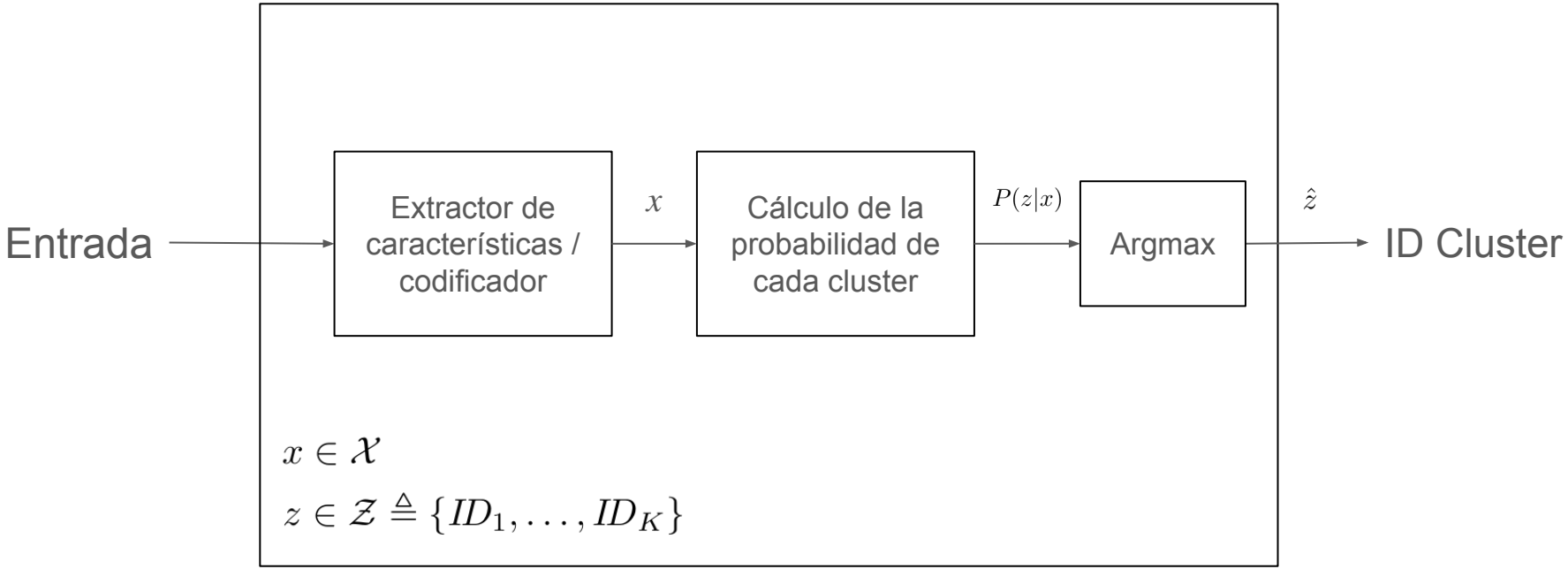
Toma de decisiones categóricas



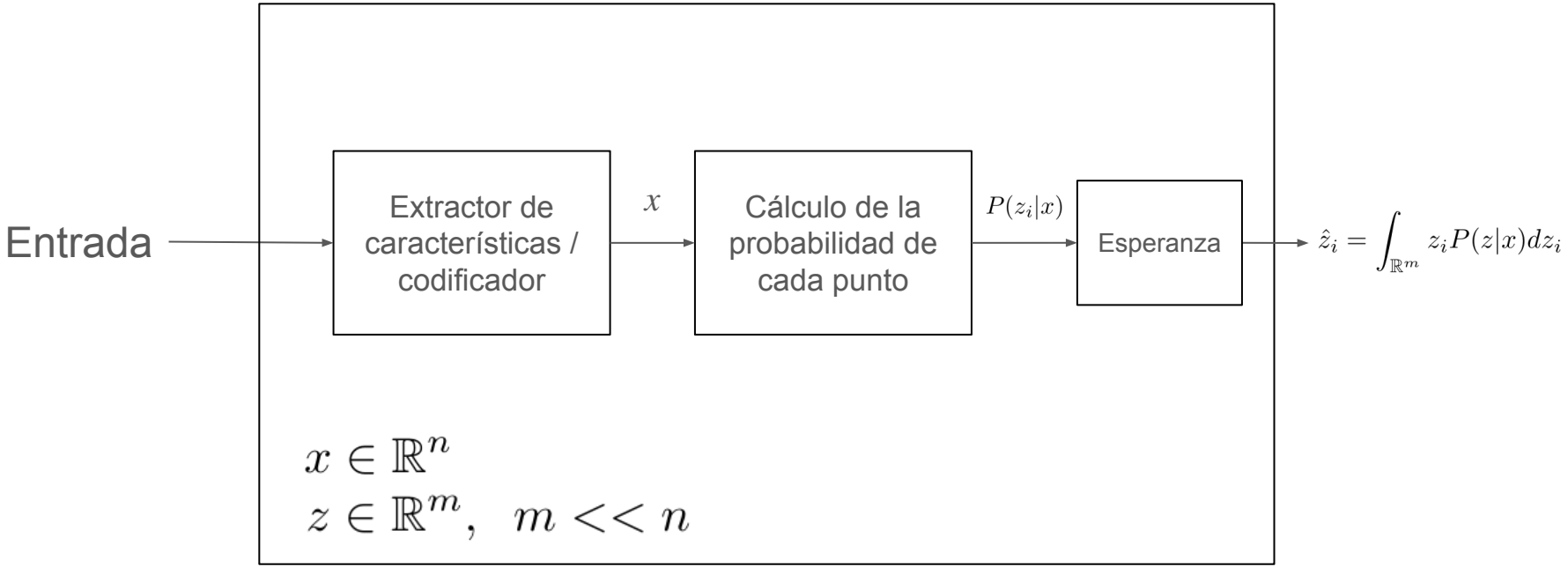
Regresión cuadrática



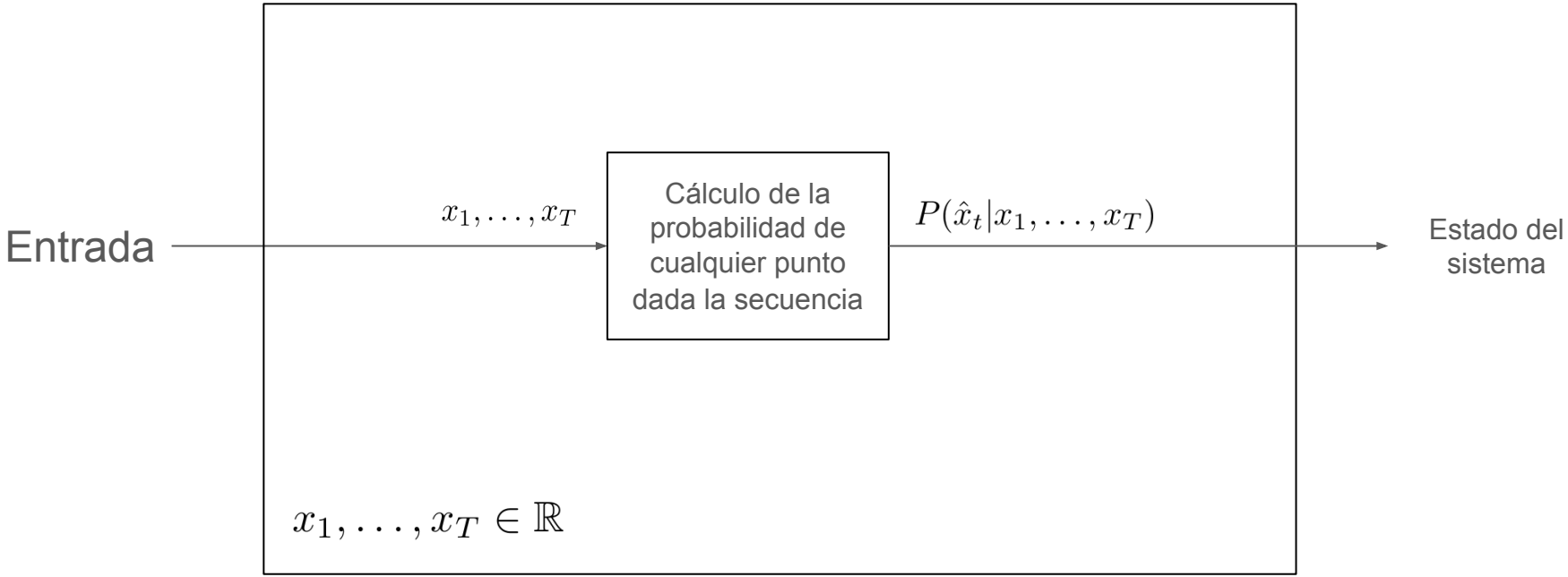
Clustering


Entrada
Extractor de características / codificador
x
Cálculo de la probabilidad de cada cluster
$P(z|x)$
Argmax
$\hat{z}$
ID Cluster
$x \in \mathcal{X}$
$z \in \mathcal{Z} \triangleq \{ID_1, \ldots, ID_K\}$

Reducción de la dimensionalidad



Filtrado Estocástico



La evaluación

- Es muy importante saber cómo evaluar un sistema de ML
- Tengo que pensar qué me interesa evaluar y para qué lo voy a usar.
- ¿En qué no funciona mi sistema cuando no funciona?
- ¿Cuánto me importan los casos en que no funcionan?
- ¿Los datos de evaluación son los mismos que voy a ver en la práctica?

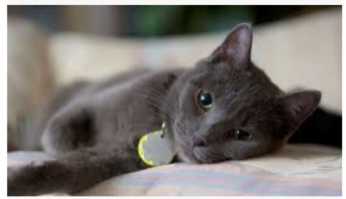
Aprendiendo de los datos

Aprendiendo a clasificar perros y gatos







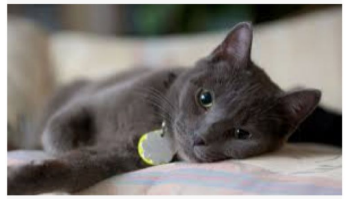


Aprendiendo a clasificar perros y gatos



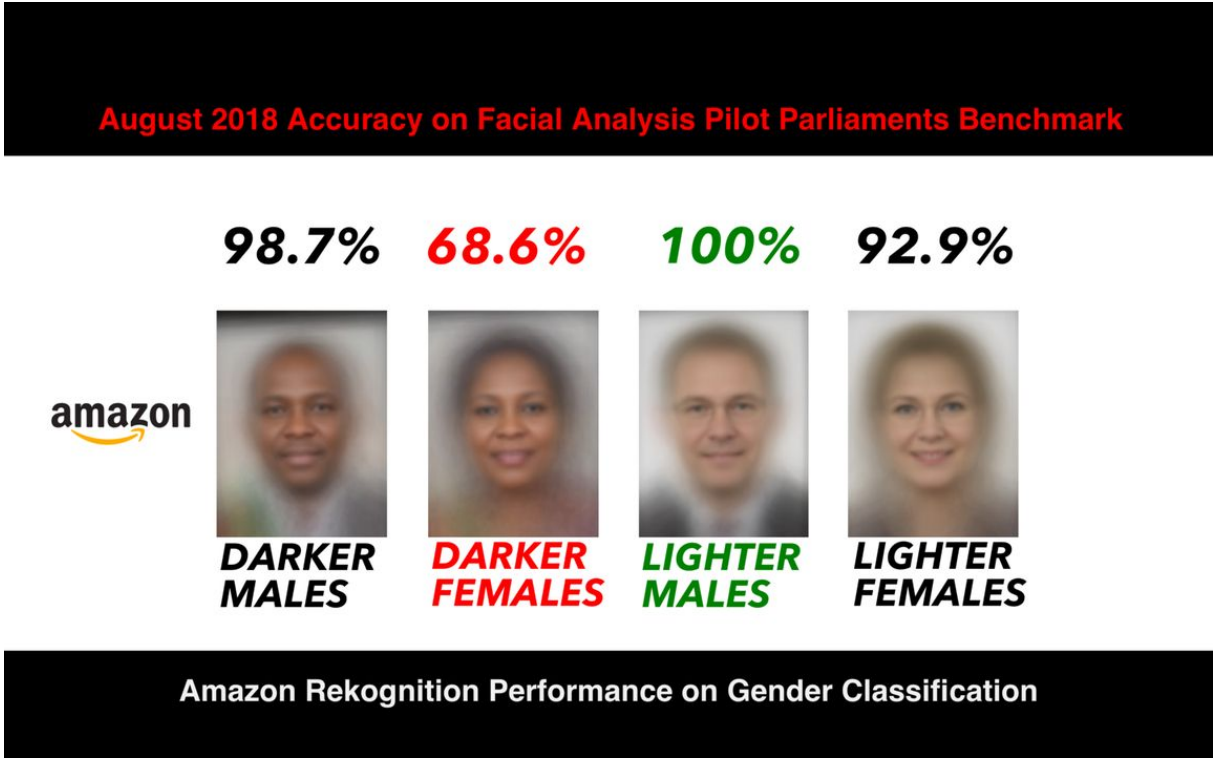






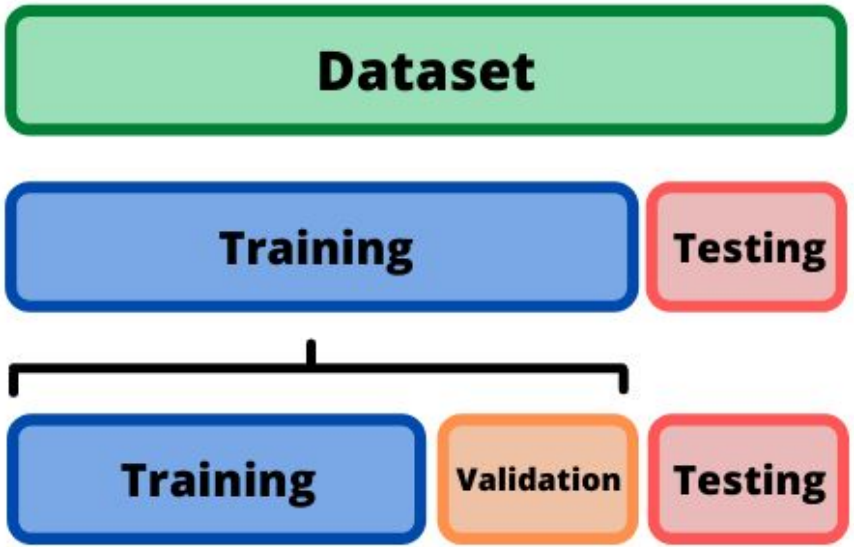


Todos los datos tienen sesgos



Los datos son limitados

- Si queremos medir cómo se va a comportar mi sistema en la práctica, dejamos una porción de test y no la veo hasta el final.
- El resto, dividimos los datos dejando una porción para ajustar los parámetros del modelo y otra para entrenar.

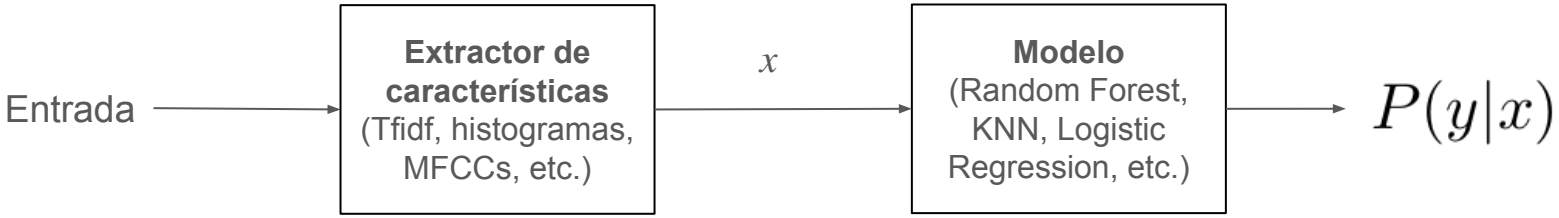


- Podemos dividir de diferentes maneras los datos disponibles pero **siempre teniendo en cuenta las condiciones que voy a tener en cada set.**

Diseñando el modelo

La tendencia antes de ~2010

La mayor parte del tiempo del desarrollo se usaba buscando la mejor combinación de "extracción de características + modelo" que resolvía una tarea.

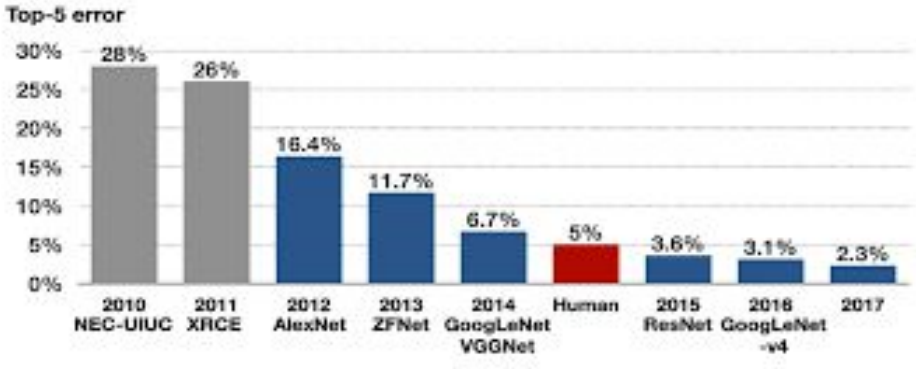


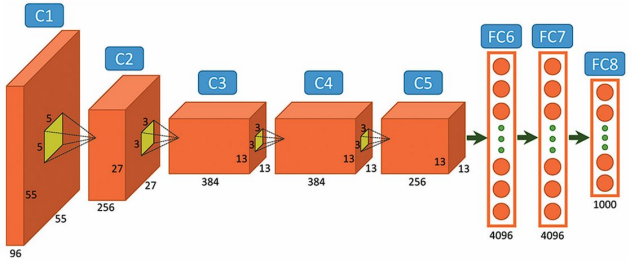
Esta forma de trabajo todavía se usa mucho en dos situaciones muy comunes:

- Cuando quiero extraer características interpretables.
- Cuando tengo tareas muy específicas.

Aparición de las redes neuronales profundas

- La primera aparición de las NN como tales es en 1960.
- En 2012, AlexNet gana la competencia de clasificación de imágenes usando una sucesión de redes convolucionales (que ya existían desde hace rato).





Repaso de Logistic/Softmax Regression

- Estamos en clasificación: Dada una entrada $\mathbf{x} \in \mathbb{R}^n$, quiero calcular $P(y_k|\mathbf{x})$ con $y \in \mathcal{Y} \triangleq \{y_1, \ldots, y_K\}$
- Definimos un conjunto de parámetros $\{\mathbf{w}_1, \ldots, \mathbf{w}_K, \mathbf{b}\}$ con $\mathbf{w}_i \in \mathbb{R}^K$, $\mathbf{b} \in \mathbb{R}^K$ y la probabilidad

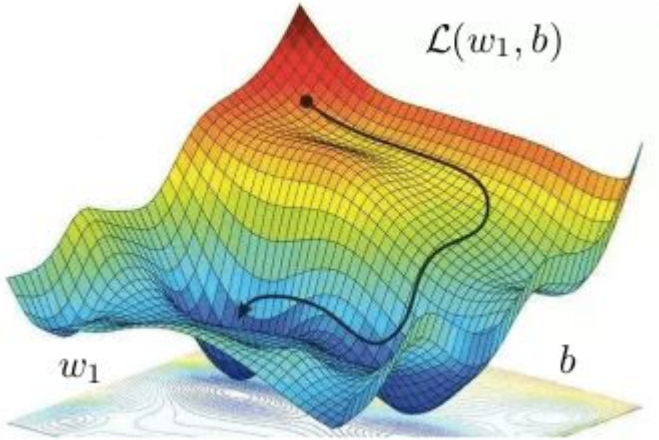
$$P(y_k|x) = \frac{e^{\mathbf{w}_k^T \mathbf{x} + b_k}}{\sum_{j=1}^{K} e^{\mathbf{w}_j^T \mathbf{x} + b_j}}$$

- Dado un conjunto de entrenamiento $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$ definimos la función de costo en función de los parámetros

$$\mathcal{L}(\mathbf{w}_1, \ldots, \mathbf{w}_K, \mathbf{b}) = -\frac{1}{N} \sum_{i=1}^{N} \log \left(P(y^{(i)}|\mathbf{x}^{(i)}) \right)$$

Repaso de Logistic/Softmax Regression

- Minimizamos la función usando alguna variante de gradiente descendente.
- Podemos regular la forma de descender con el parámetro μ, "*learning rate*".

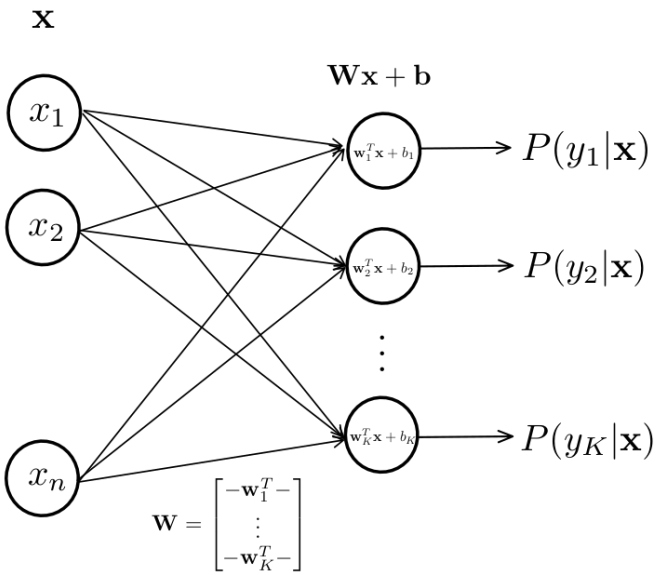


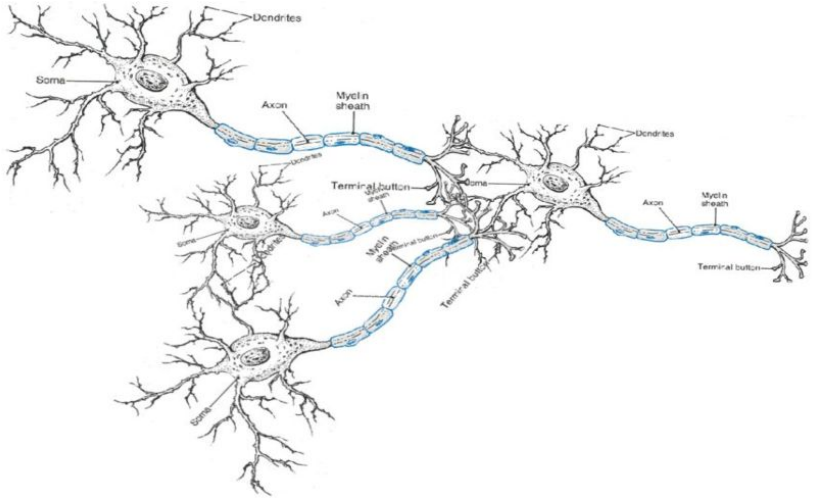
$$\mathbf{w}_i^{\text{new}} := \mathbf{w}_i^{\text{old}} - \mu \nabla_{\mathbf{w}_i} \mathcal{L}(\mathbf{w}_1, \ldots, \mathbf{w}_K, \mathbf{b})$$

$$\mathbf{b}^{\text{new}} := \mathbf{b}^{\text{old}} - \mu \frac{\partial}{\partial b} \mathcal{L}(\mathbf{w}_1, \ldots, \mathbf{w}_K, \mathbf{b})$$

Similitud con una Red Neuronal

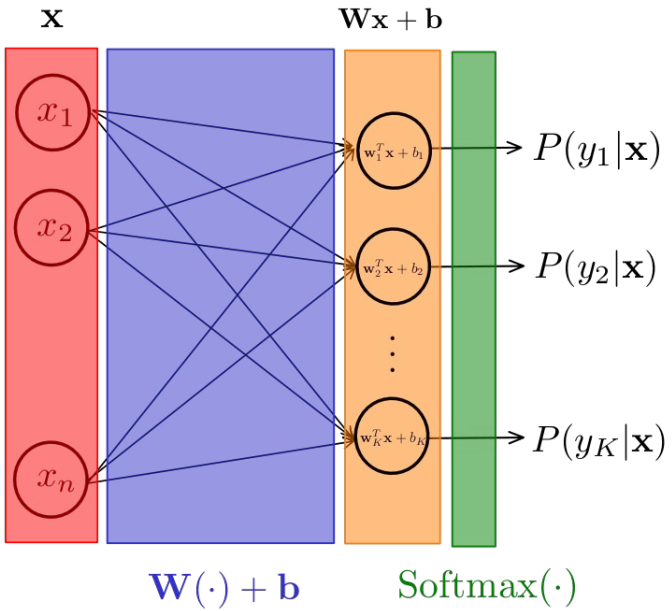
Este modelo, conocido como Perceptron, está inspirado en el funcionamiento de las neuronas.

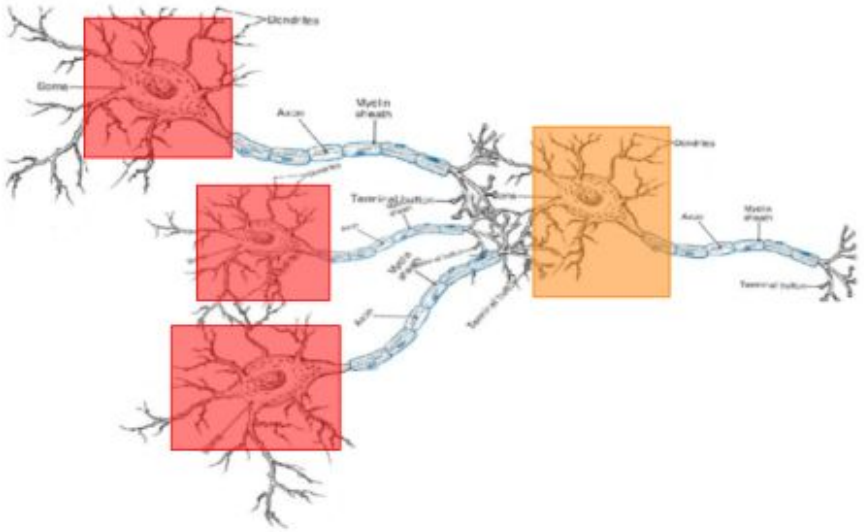




Similitud con una Red Neuronal

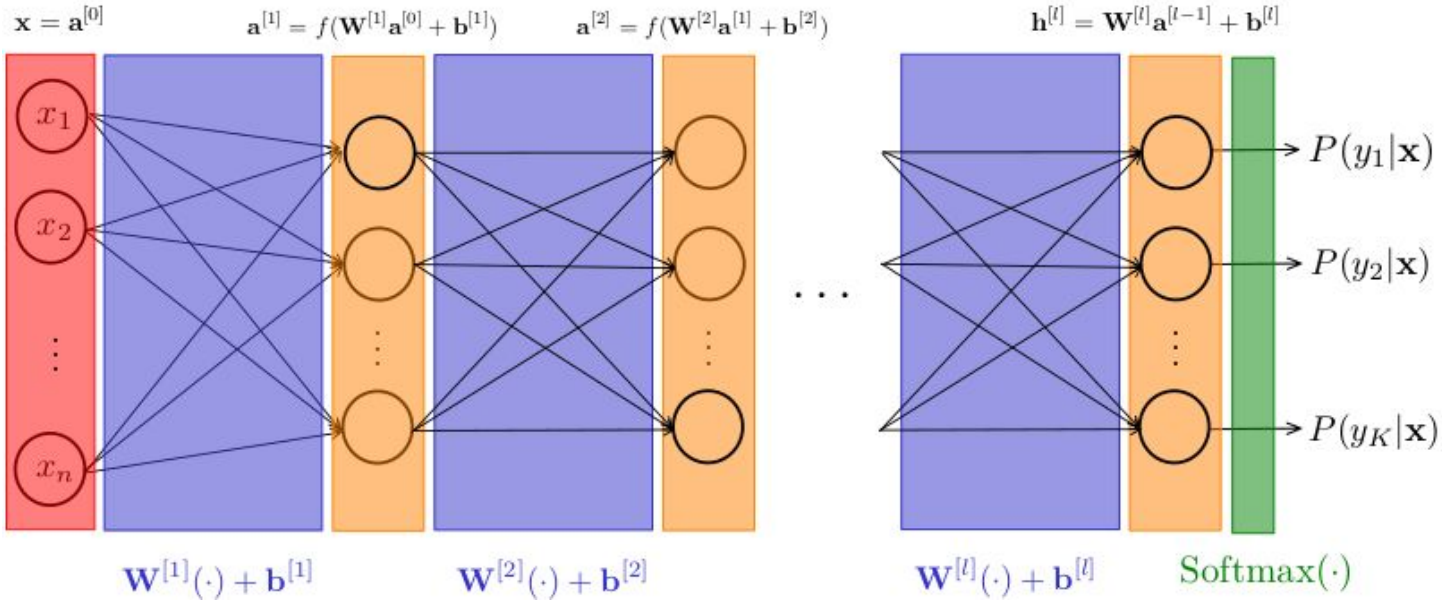
Este modelo, conocido como Perceptron, está inspirado en el funcionamiento de las neuronas.





Perceptron Multicapa (MLP)

En 2012 empieza a dar resultado la idea de "apilar" capas de perceptrones para distribuir el trabajo de cada una de ellas en tareas más simples.

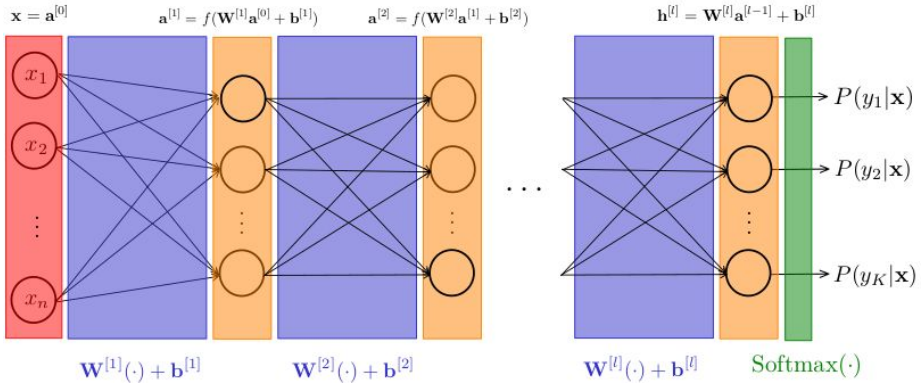


$$\mathbf{a}^{[l]} = f(\mathbf{h}^{[l]})$$

$$\mathbf{h}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$

Backpropagation

Podemos calcular el gradiente automáticamente usando la regla de la cadena



$$\mathcal{L} = f^{[l]} \circ f^{[l]} \circ \ldots \circ f^{[1]}$$

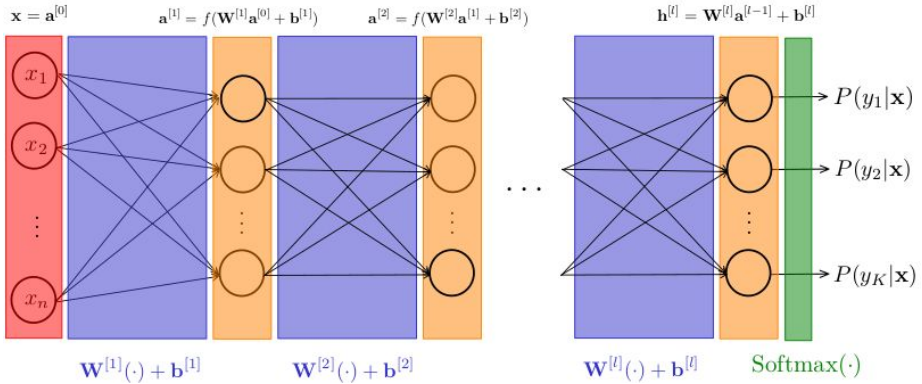
Forward pass

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial f^{[l]}} \frac{\partial f^{[l]}}{\partial f^{[l-1]}} \cdots \frac{\partial f^{[1]}}{\partial w}$$

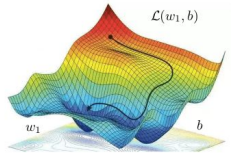
Backward pass

Backpropagation

Podemos calcular el gradiente automáticamente usando la regla de la cadena



Sigue siendo el mismo algoritmo!



$$\mathbf{w}_i^{\text{new}} := \mathbf{w}_i^{\text{old}} - \mu \nabla_{\mathbf{w}_i} \mathcal{L}(\mathbf{w}_1, \ldots, \mathbf{w}_K, \mathbf{b})$$

$$\mathbf{b}^{\text{new}} := \mathbf{b}^{\text{old}} - \mu \frac{\partial}{\partial b} \mathcal{L}(\mathbf{w}_1, \ldots, \mathbf{w}_K, \mathbf{b})$$

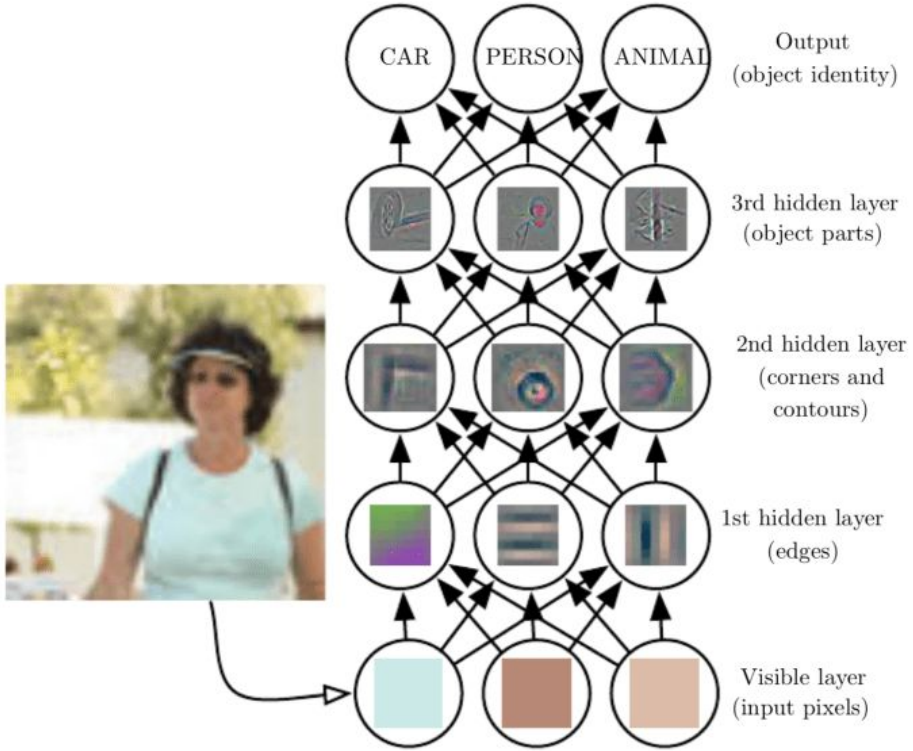
$$\mathcal{L} = f^{[l]} \circ f^{[l]} \circ \ldots \circ f^{[1]}$$

Forward pass

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial f^{[l]}} \frac{\partial f^{[l]}}{\partial f^{[l-1]}} \cdots \frac{\partial f^{[1]}}{\partial w}$$

Backward pass

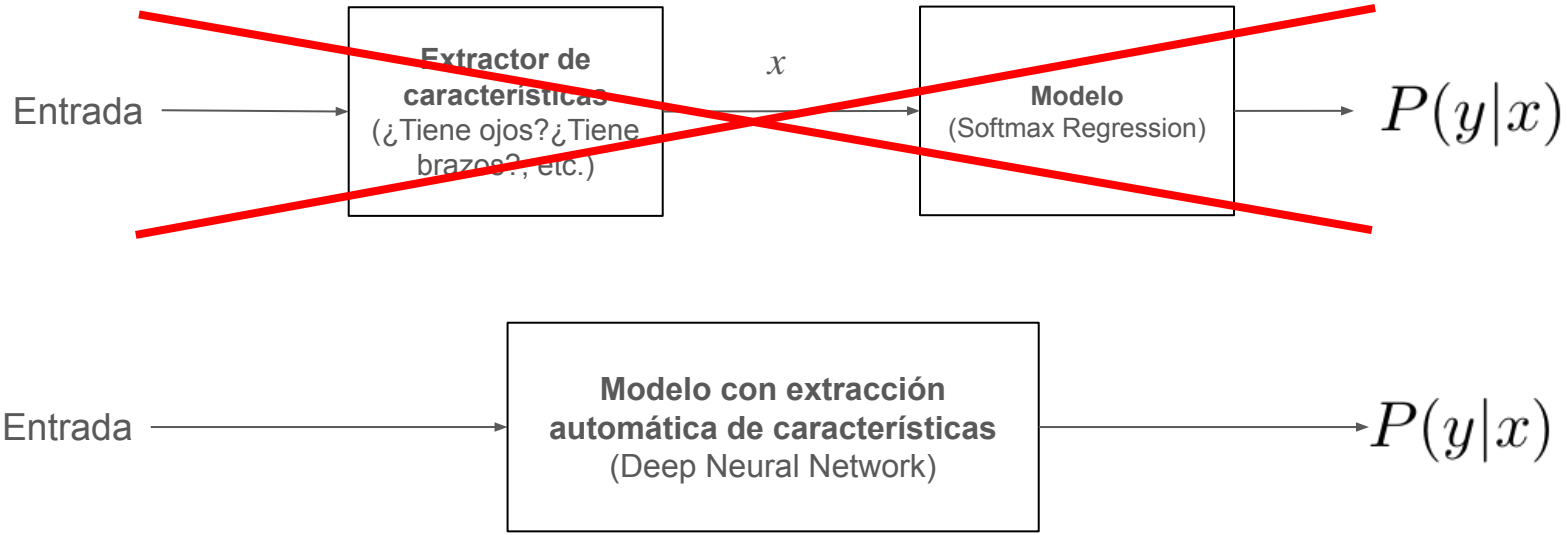
Representaciones Distribuidas



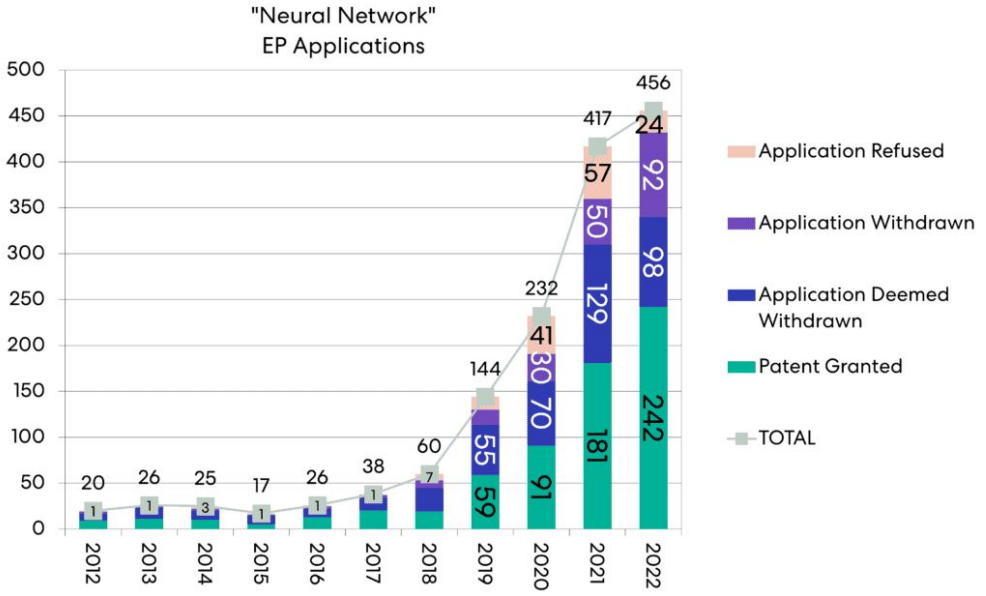
- Cada vez que ingresa un estímulo a una capa, esta capa responde específicamente a ciertos patrones de la entrada y les pasa esa respuesta a las siguientes capas.
- Entreno los parámetros para extraer patrones que sirvan para mi tarea.

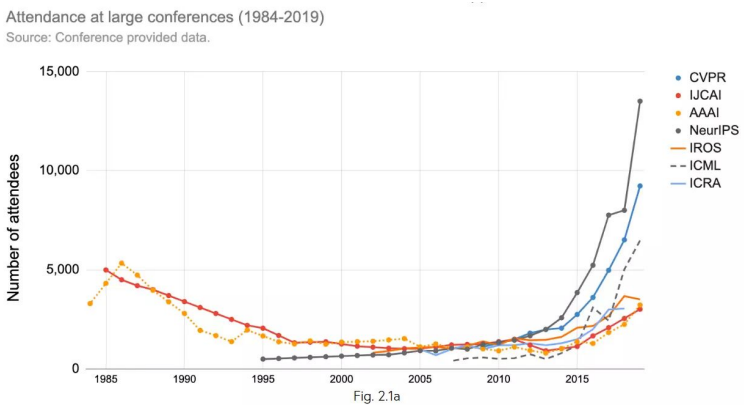
La tendencia entre ~2012 y ~2018

En lugar de hacer el esfuerzo en extraer las características importantes para nuestra tarea, hacemos el esfuerzo en buscar una arquitectura que las extraiga automáticamente y la alimentamos con muchos datos de entrenamiento.



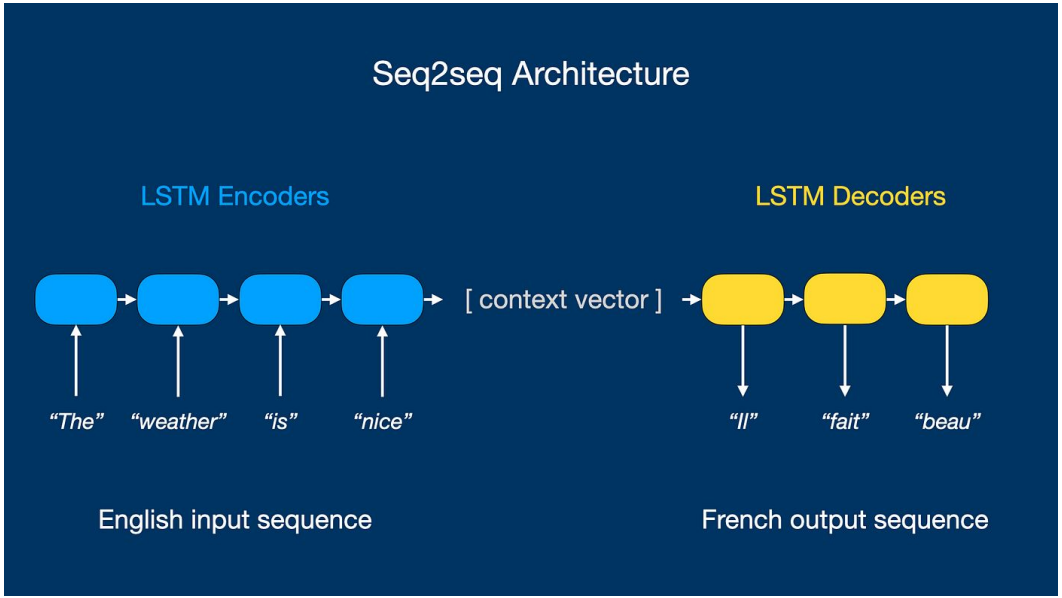
La tendencia entre ~2012 y ~2018



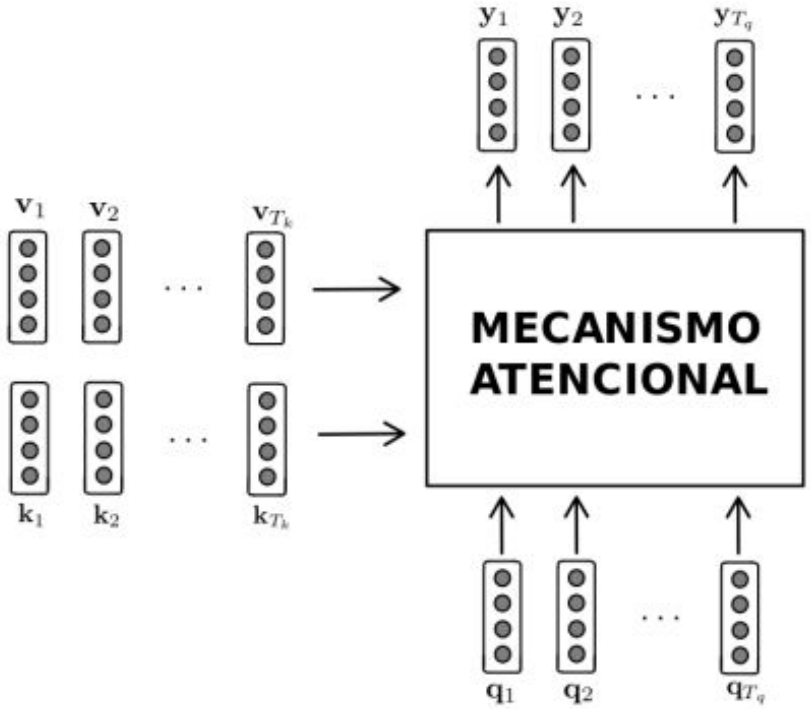


Redes Atencionales para secuencias

Las arquitecturas existentes antes de 2017 tenían limitaciones prácticas a la hora de entrenar, sobre todo las RNN.



Redes Atencionales para secuencias



Aparecen los mecanismos atencionales, que permiten procesar secuencias enteras en paralelo.

Se calcula una secuencia de salida como un promedio ponderado de la secuencia de valores:

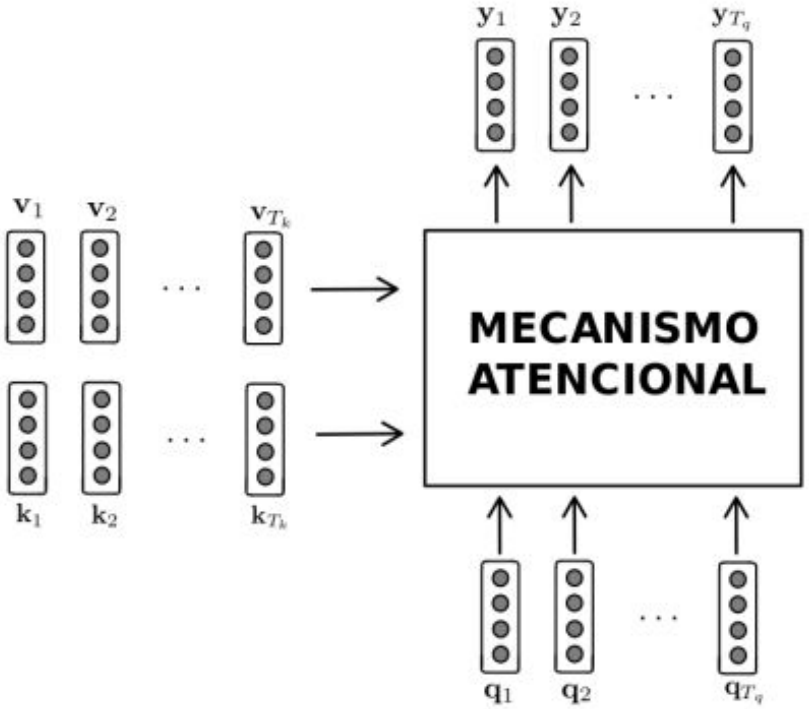
$$\mathbf{y}_t = \sum_{i=1}^{T_k} s_i \mathbf{v}_i$$

El valor s_i que pondera la suma se conoce como "puntaje de alineamiento" y está en función del pedido y de la secuencia de keys.

$$\mathbf{y}_t = \sum_{i=1}^{T_k} \text{align}(\mathbf{k}_1, \ldots, \mathbf{k}_{T_k}, \mathbf{q}_t)_i \ \mathbf{v}_i$$

$$= \sum_{i=1}^{T_k} \frac{e^{\text{score}(\mathbf{q}_t, \mathbf{k}_i)}}{\sum_{j=1}^{T_k} e^{\text{score}(\mathbf{q}_t, \mathbf{k}_j)}} \mathbf{v}_i$$

Redes Atencionales para secuencias



$$\mathbf{y}_t = \sum_{i=1}^{T_k} \text{align}(\mathbf{k}_1, \ldots, \mathbf{k}_{T_k}, \mathbf{q}_t)_i \; \mathbf{v}_i$$

$$= \sum_{i=1}^{T_k} \frac{e^{\text{score}(\mathbf{q}_t, \mathbf{k}_i)}}{\sum_{j=1}^{T_k} e^{\text{score}(\mathbf{q}_t, \mathbf{k}_j)}} \; \mathbf{v}_i$$

La forma de calcular el score puede variar:

$$\text{score}(\mathbf{q}, \mathbf{k}) = \begin{cases} \mathbf{q}^T \mathbf{k} \\ \mathbf{q}^T \mathbf{W} \mathbf{k} \\ \mathbf{w}^T \tanh\left(\mathbf{W}_q \mathbf{q} + \mathbf{W}_k \mathbf{k}\right) \end{cases}$$

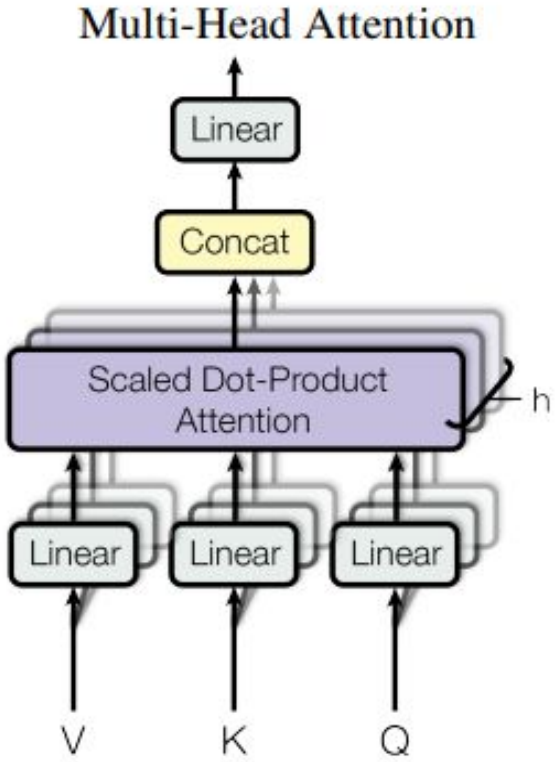
Redes Atencionales para secuencias

$$Q = (\mathbf{q}_1, \ldots, \mathbf{q}_{T_q}) \qquad \mathbf{q}_i \in \mathbb{R}^{d_{model}} \quad \forall\, i = 1, \ldots, T_q$$
$$K = (\mathbf{k}_1, \ldots, \mathbf{k}_{T_k}) \qquad \mathbf{k}_i \in \mathbb{R}^{d_{model}} \quad \forall\, i = 1, \ldots, T_k$$
$$V = (\mathbf{v}_1, \ldots, \mathbf{v}_{T_k}) \qquad \mathbf{v}_i \in \mathbb{R}^{d_{model}} \quad \forall\, i = 1, \ldots, T_k$$

Se suelen proyectar todos los vectores del modelo antes de ingresarlos o devolverlos:

$$\mathrm{Attn}(Q, K, V; \mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V) = \sum_{i=1}^{T_k} \frac{e^{s_{ti}}}{\sum_{j=1}^{T_k} e^{s_{tj}}} \mathbf{W}^V \mathbf{v}_i \qquad s_{ti} = \mathrm{score}(\mathbf{W}^Q \mathbf{q}_t, \mathbf{W}^K \mathbf{k}_i)$$

Multi-Head Attention



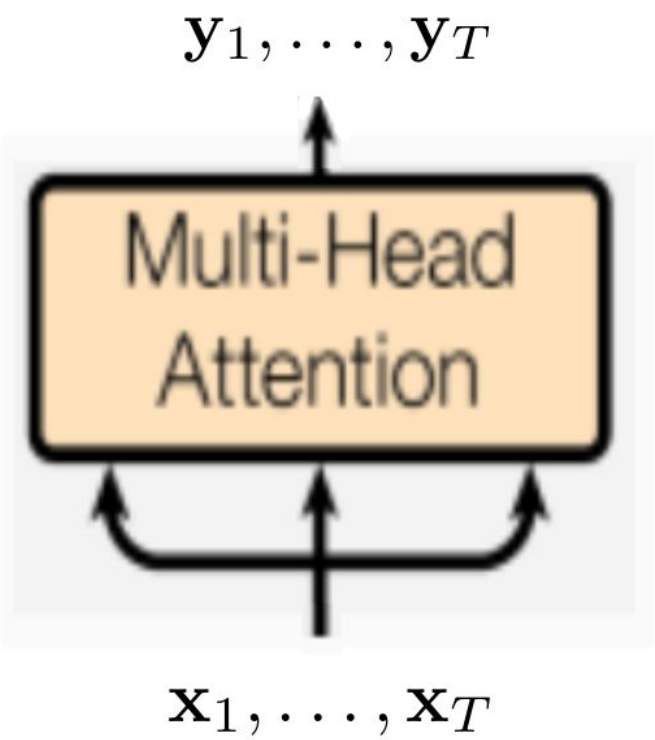
Puedo repetir esto muchas veces en una misma capa y obtener múltiples "cabezas" de atención.

$$\text{MultiHead}(Q, K, V) = \sum_{j=1}^{h} \mathbf{W}_j^O \mathbf{h}_j$$

$$\mathbf{h}_j = \text{Attn}(Q, K, V; \mathbf{W}_j^Q, \mathbf{W}_j^K, \mathbf{W}_j^V)$$

Idealmente, cada cabeza debería prestar atención a diferentes aspectos de la secuencia.

Self-Attention



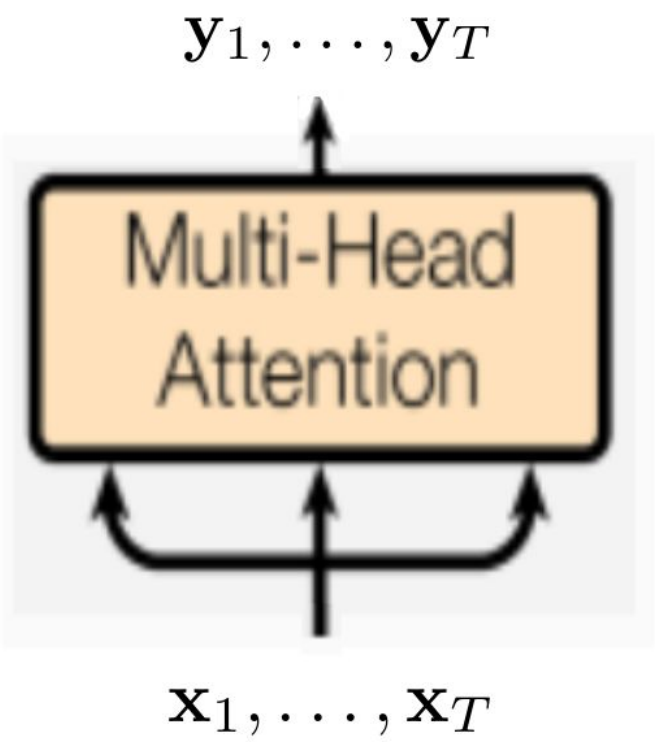
Dado que es posible "prestar atención" a la misma secuencia, se puede definir una función que recibe una entrada y la procesa como una capa *MultiHeadAttention*:

$$\text{MultiHeadSelfAttn}(\mathbf{x}_1, \ldots, \mathbf{x}_T) = \text{MultiHead}(X, X, X)$$

$$X = (\mathbf{x}_1, \ldots, \mathbf{x}_T) \qquad \mathbf{x}_t \in \mathbb{R}^{d_{model}} \;\; \forall\, t = 1, \ldots, T$$

Este bloque se puede tomar como una capa más de la red

Self-Attention



Dado que es posible "prestar atención" a la misma secuencia, se puede definir una función que recibe una entrada y la procesa como una capa *MultiHeadAttention*:

$$\text{MultiHeadSelfAttn}(\mathbf{x}_1, \ldots, \mathbf{x}_T) = \text{MultiHead}(X, X, X)$$

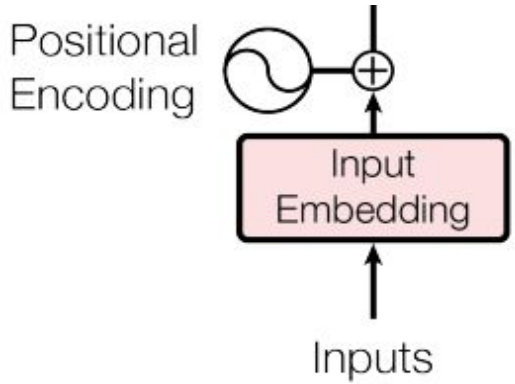
$$X = (\mathbf{x}_1, \ldots, \mathbf{x}_T) \qquad \mathbf{x}_t \in \mathbb{R}^{d_{model}} \;\; \forall\, t = 1, \ldots, T$$

Este bloque se puede tomar como una capa más de la red

¿Hay algún problema con hacer esto? 🤔

Positional Encoding

Sí… Los mecanismos de atención no distinguen el orden de la secuencia. Necesito agregarlo aparte.



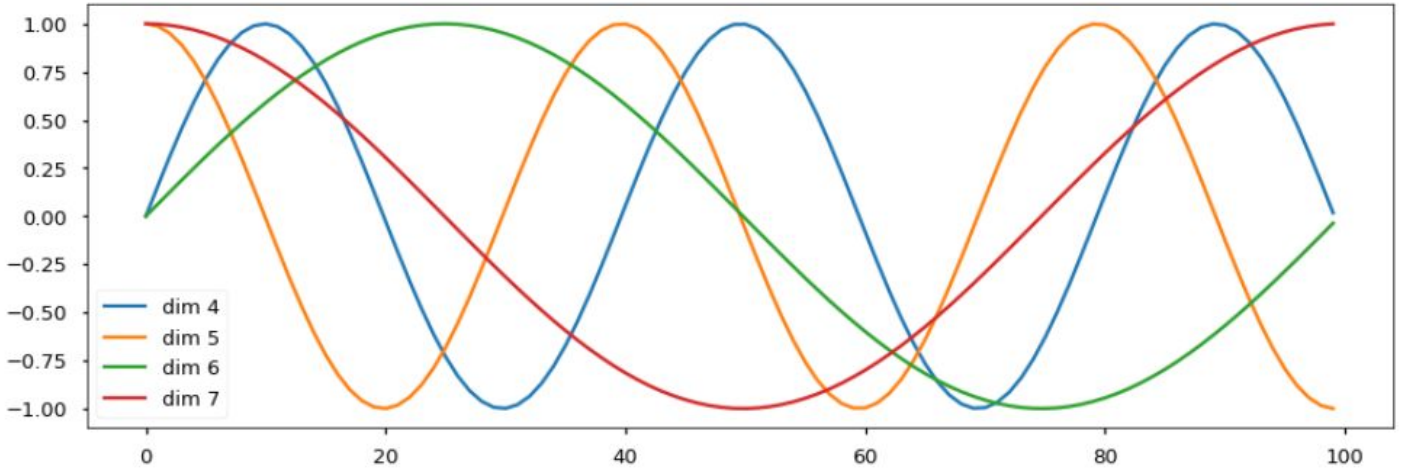
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

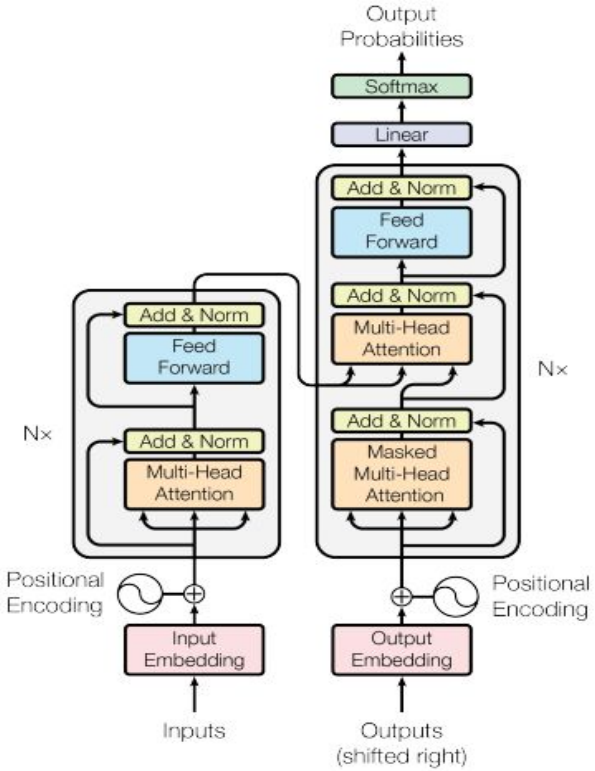
```
class PositionalEncoding(nn.Module):
    "Implement the PE function."
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) *
                             -(math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + Variable(self.pe[:, :x.size(1)],
                         requires_grad=False)
        return self.dropout(x)
```

Positional Encoding

```
plt.figure(figsize=(15, 5))
pe = PositionalEncoding(20, 0)
y = pe.forward(Variable(torch.zeros(1, 100, 20)))
plt.plot(np.arange(100), y[0, :, 4:8].data.numpy())
plt.legend(["dim %d"%p for p in [4,5,6,7]])
None
```



The Transformer



Attention is all you need (Vaswani et al., 2017)

Ideas para mejorar el modelo seq2seq:

- Reemplazar las RNN por Self-Attention
- Agregar conexiones residuales dentro de la misma capa
- Utilizar una codificación posicional antes de ingresar al sistema

Spinoffs: BERT y GPT

Las empresas empiezan a sacar al público modelos de lenguaje súper pre-entrenados.

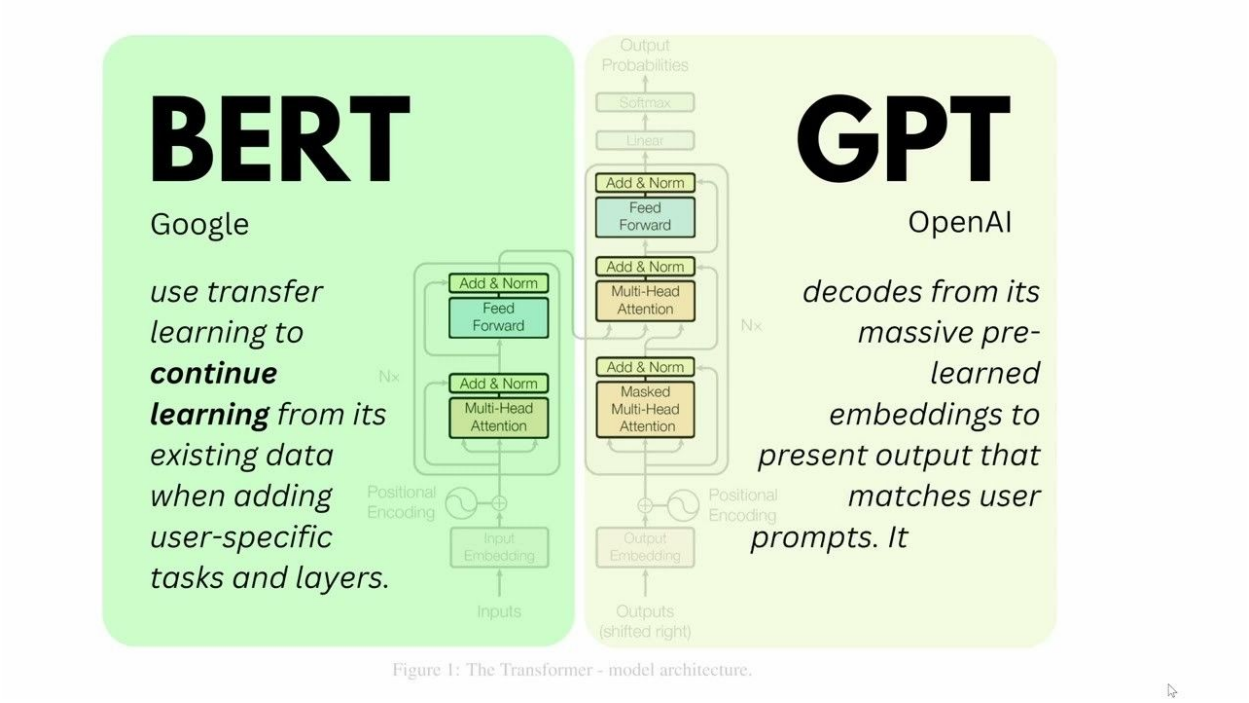


Figure 1: The Transformer - model architecture.

La tendencia

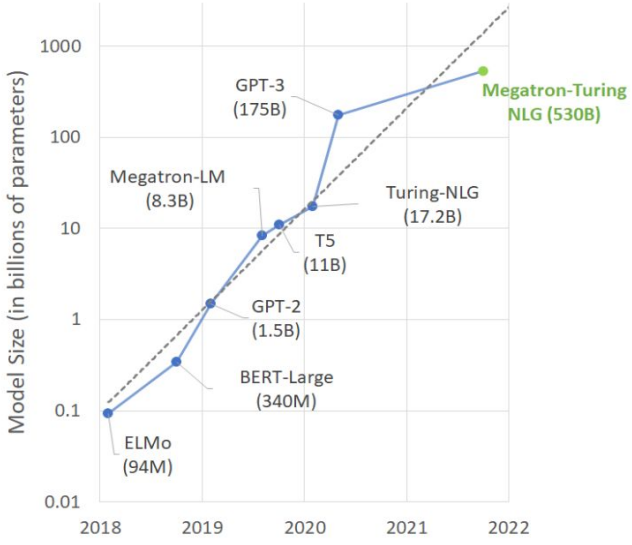
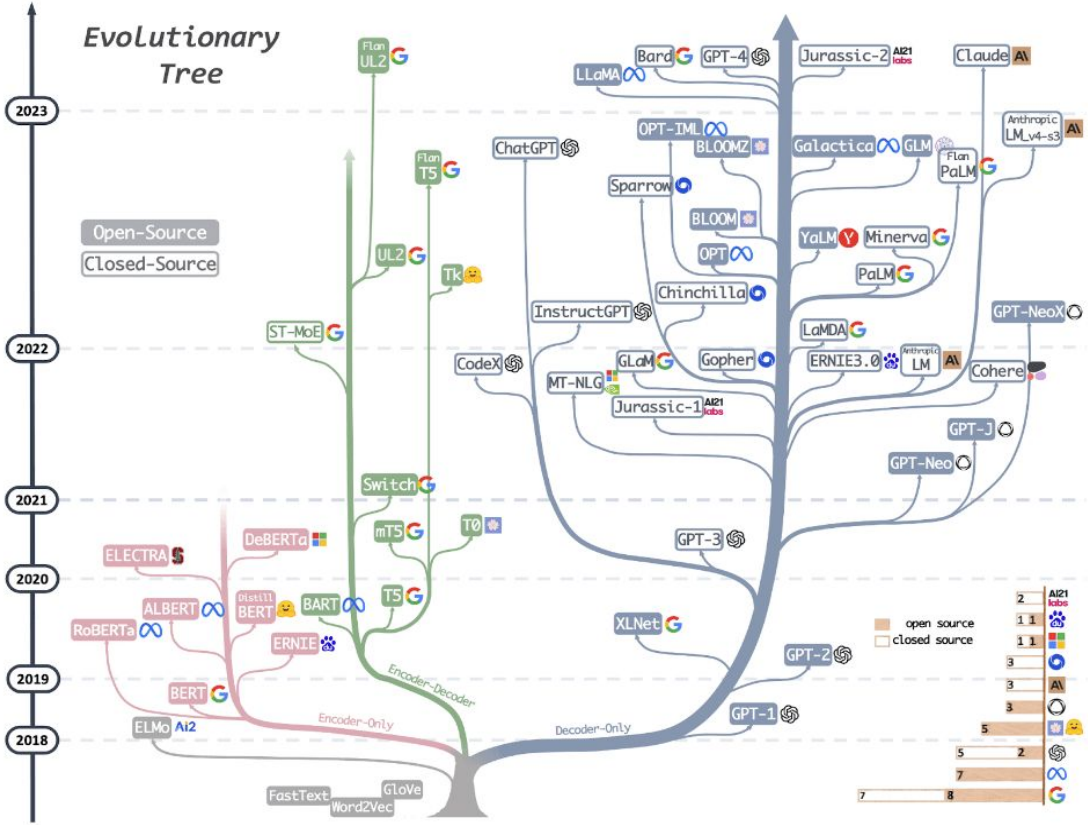
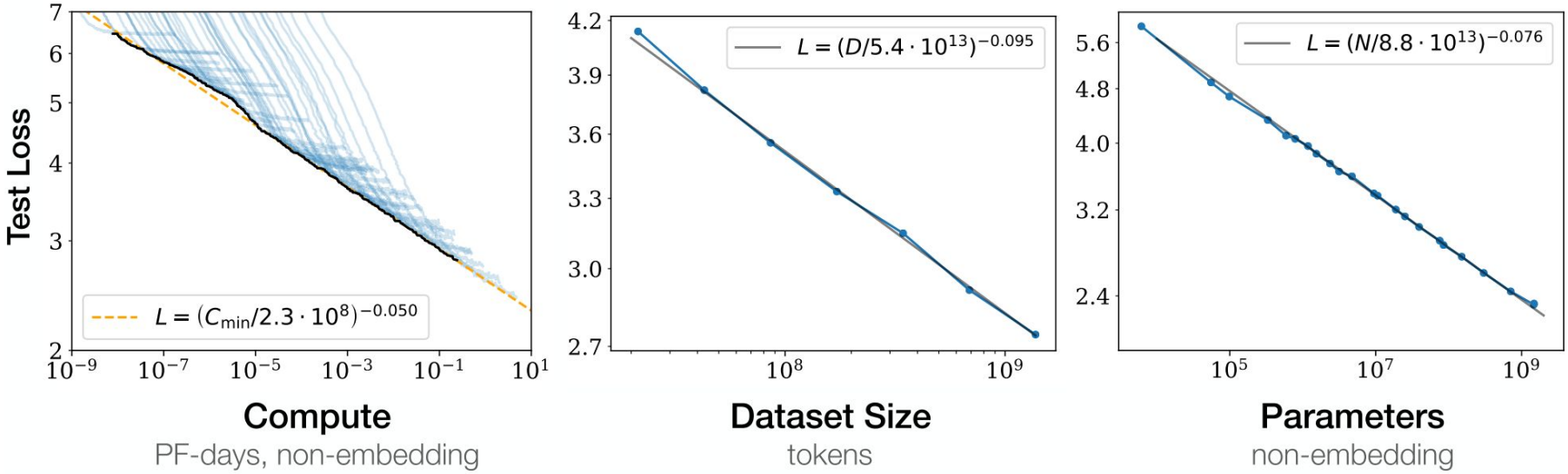


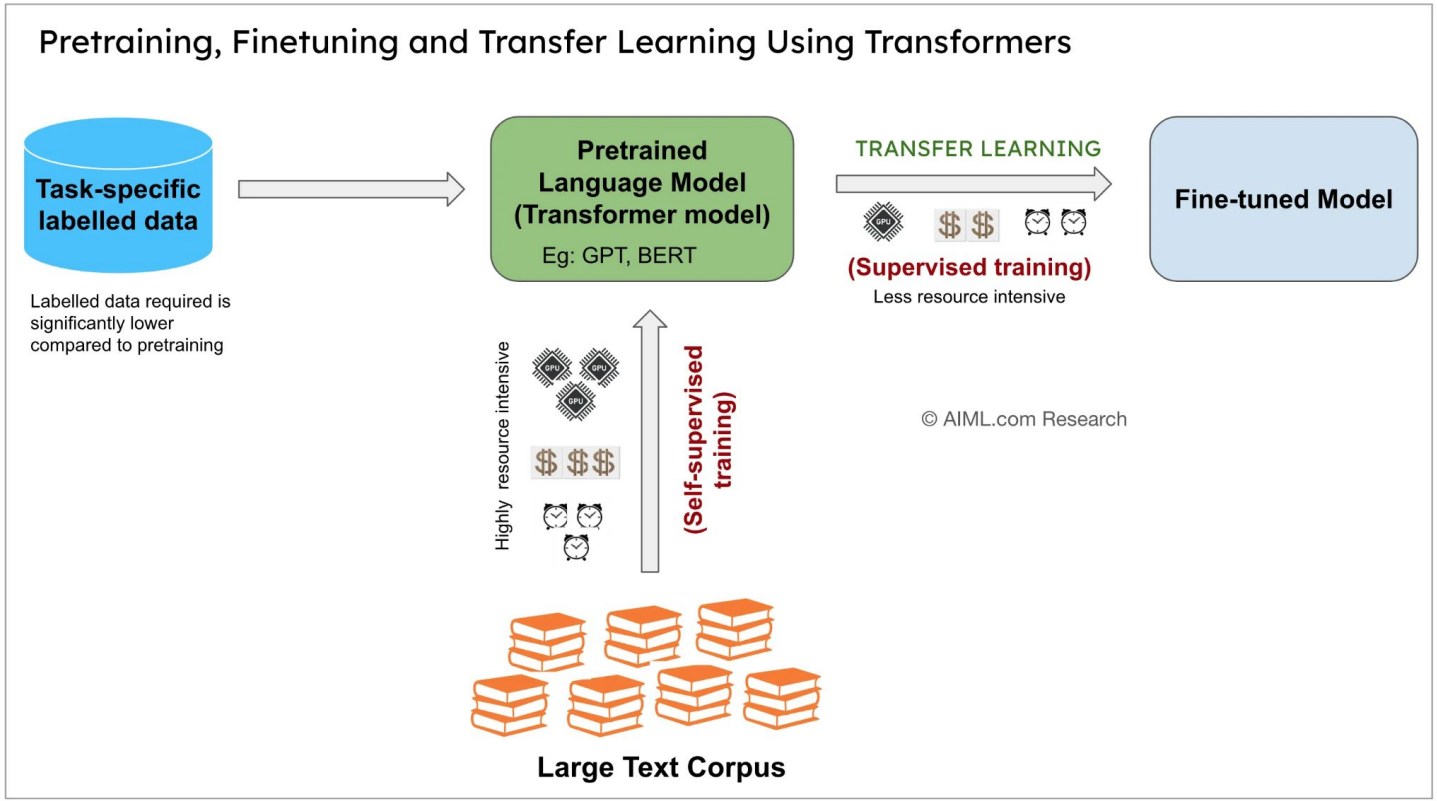
Figure 1. Trend of sizes of state-of-the-art NLP models with time



Scaling laws



La tendencia desde ~2017 hasta ahora



Propiedades emergentes: Tendencia incierta

In-context learning (*"Language Models are Few-Shot Learners"*, Brown et al., 2020)

Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

```
1  Translate English to French:        ← task description
2  sea otter => loutre de mer          ← examples
3  peppermint => menthe poivrée        ←
4  plush girafe => girafe peluche      ←
5  cheese =>                           ← prompt
```

Zero-shot

The model predicts the answer given only a natural language description of the task. No gradient updates are performed.

```
1  Translate English to French:        ← task description
2  cheese =>                           ← prompt
```

Pero siempre es el mismo Pipeline!

1. Qué aplicación quiero resolver.
2. Cómo voy a evaluar mi sistema.
3. Juntar datos para enseñarle a mi sistema.
4. Definir el algoritmo de aprendizaje que voy a usar
5. Entrenar, validar y repetir los pasos 3, 4 y 5 hasta que funcione.
6. Enviar a producción / evaluar en datos nunca vistos.

Investigación y Formación en la UBA

Carreras de posgrado afines:

- Doctorado en Ingeniería (FIUBA)
- Doctorado en Computación (FCEyN)
- Maestría en Ciencia de datos (FCEyN)
- Maestría en IA Embebida (= 2 especializaciones) (FIUBA)
- Maestría en Ingeniería Matemática (FIUBA)
- Ver más: https://www.fi.uba.ar/posgrado

Grupos de Investigación afines:

- Laboratorio de Procesamiento de Señales en Comunicaciones (FIUBA)
- Laboratorio de Inteligencia Artificial Aplicada (FCEyN)
- Grupos y Laboratorios de FIUBA / FIUBA-CONICET: https://www.fi.uba.ar/investigacion
- Grupos y Laboratorios de FCEyN / FCEyN-CONICET: https://icc.fcen.uba.ar
- Centro de Simulaciones Computacionales CONICET: https://csc.conicet.gov.ar

Recursos

- Cursos de Machine Learning:
  - Curso de Machine Learning (Stanford, cs229n)
  - Cursos de Deep Learning / Machine learning de CMU o de MIT
  - Reinforcement Learning (Stanford, cs234)

- Cursos orientados a aplicaciones:
  - Computer Vision (Stanford, cs231n)
  - NLP + Deep Learning (Stanford, cs224n)
  - Natural Language Understanding (Stanford, cs224u)

- Tutoriales:
  - Tutoriales de Pytorch
  - The Annotated Transformer
  - Curso de Hugging Face
  - Videos de Corey Schafer de Python

- Libros:
  - Speech and Language Processing, D. Jurafsky and J. H. Martin
  - Foundations of Statistical Natural Language Processing, C. D. Manning and H. Schütze
  - Natural Language Processing with Transformers, L. von Werra, L.Tunstall, and T. Wolf
  - Deep Learning, A. Courville, I. Goodfellow and Y. Bengio