In [146…
```
import os
import torch
import torch.nn as nn
from torch.nn.parameter import Parameter
```

Modelos Neuronales IV

> Autor: Jorge García González (Universidad de Málaga)

> Última Actualización: 9/10/2025

> Asignatura: Programación para la Inteligencia Artificial

En el cuaderno anterior hemos hablado de cómo definir en PyTorch un modelo neuronal de varias capas usando **torch.nn.Sequential()**. Antes de continuar con más cuestiones teóricas es importante que hablemos de algunas utilidades relacionadas con los modelos neuronales en PyTorch. **torch.nn.Sequential()** es una manera muy útil de definir modelos a partir de capas predefinidas, pero en ocasiones necesitamos definir ajustables más complicados que simplemente aplicar capas y activaciones de manera secuencial. PyTorch ofrece una manera más flexible de definir nuestros modelos: simplemente definirlos como una clase nueva con el comportamiento que queramos. Vamos a ver un ejemplo en el que creamos un modelo de PyTorch para modelar una recta como hemos hecho en cuadernos anteriores.

In [147…
```
class Line_2D_Model_PyTorch(torch.nn.Module):
  def __init__(self):
    super().__init__()              # Llamamos al constructor de la clase de la que heredamos por si tuviera alg
    self.m = Parameter(torch.rand(1))
    self.b = Parameter(torch.tensor(0).float())

  def forward(self,x):
    return self.m*x+self.b

  def extra_repr(self):
    return f"inclinación={float(self.m.clone().detach())}, ordenada en el origen={float(self.b.clone().detach())
```

Como podemos ver, definimos una clase que hereda de **torch.nn.Module** y definimos sus parámetros optimizables como atributos de la clase indicados con **torch.nn.parameter.Parameter**. Luego definimos cómo opera este módulo su paso hacia delante con el método **forward**. El método **extra_repr** es simplemente una utilidad para definir qué mostrará la clase en un **print**.

In [148…
```
line_2d_model = Line_2D_Model_PyTorch()
print(line_2d_model)
print([p for p in line_2d_model.parameters()])
x=10
print(line_2d_model(x))
```

```
Line_2D_Model_PyTorch(inclinación=0.07441467046737671, ordenada en el origen=0.0)
[Parameter containing:
tensor([0.0744], requires_grad=True), Parameter containing:
tensor(0., requires_grad=True)]
tensor([0.7441], grad_fn=<AddBackward0>)
```

Poder definir el constructor y el *forward* nos da libertad prácticamente absoluta para definir nuestros modelos siempre que durante el *forward* utilicemos operaciones que sean diferenciables para PyTorch (o no podrá realizar el *backward*).

Ahora vamos a ver un ejemplo un poco más complejo en el que definimos un MLP, pero de una manera más personalizada que usando el modelo secuencial de PYTorch. Asumiremos que somos demasiado perezosos para definir las capas línea por línea y queremos automatizarlo.

In [149…
```
class My_MLP(torch.nn.Module):
  def __init__(self, list_of_neurons, list_of_activations, device='cpu'):
    super().__init__()          # Llamamos al constructor de la clase de la que heredamos por si tuviera algo que
    self.list_of_layers = []
    self.list_of_activations = list_of_activations

    assert len(list_of_neurons)==len(list_of_activations)           # Sanity Check.

    for n in list_of_neurons:
      if len(self.list_of_layers)==0:
        self.list_of_layers.append(nn.LazyLinear(n).to(device))
      else:
        self.list_of_layers.append(nn.Linear(previous_n, n).to(device))
      previous_n=n

  def forward(self, inputs):
    outputs = None
    for layer, activation in zip(self.list_of_layers, self.list_of_activations):
```

```
      outputs = activation(layer(inputs)) if outputs is None else activation(layer(outputs))
    return outputs

  def extra_repr(self):
    return f"{[p for p in self.parameters()]}"
```

Es una lógica sencilla en la que inicializamos la instancia comprobando que la longitud de la lista con las neuronas por capa y el número de funciones de activación coincide (**assert** comprueba que una condición se cumple y para la ejecución con un error en caso contrario) y luego inicializamos todas las capas lineales de PyTorch (recordemos que no incluyen activación) con el número indicado. A la hora de hacer el paso hacia adelante simplemente vamos ejecutando capa a capa con su correspondiente función de activación.

Hay un detalle interesante en este código: el uso de LazyLinear para la primera capa. LazyLinear simplemente es una capa Lineal "vaga" que no requiere indicar el tamaño de la entrada (más información: https://docs.pytorch.org/docs/stable/generated/torch.nn.LazyLinear.html#torch.nn.LazyLinear)

Vamos a probarlo...

In [150…
```
list_of_neurons = [3,5,1]
list_of_activations = 3*[torch.nn.ReLU()]
mlp = My_MLP(list_of_neurons, list_of_activations)
print(list_of_neurons)
print(list_of_activations)
print(mlp)
```

```
[3, 5, 1]
[ReLU(), ReLU(), ReLU()]
My_MLP([])
```

Ups. Hemos definido que muestre los parámetros del MLP cuando hacemos un print, sin embargo, la lista de parámetros está vacía. Esto indcia que PyTorch no reconoce ningún parámetro optimizable. Problemático. ¿No podemos entonces usar otros objetos **torch.nn.Module** (que es lo que son los **torch.nn.Linear**) al crear nuestra propia **torch.nn.Module** clase? Hagamos otra prueba suponiendo una clase que solo funciona para MLPs con dos capas ocultas más la capa de salida.

In [151…
```
class My_MLP(torch.nn.Module):
  def __init__(self, list_of_neurons, list_of_activations, device='cpu'):
    super().__init__()         # Llamamos al constructor de la clase de la que heredamos por si tuviera algo que
    self.layer1 = nn.LazyLinear(list_of_neurons[0]).to(device)
    self.layer2 = nn.Linear(list_of_neurons[0], list_of_neurons[1]).to(device)
    self.layer3 = nn.Linear(list_of_neurons[1], list_of_neurons[2]).to(device)
    self.activation1 = list_of_activations[0]
    self.activation2 = list_of_activations[1]
    self.activation3 = list_of_activations[2]

  def forward(self, inputs):
    outputs = self.activation1(self.layer1(inputs))
    outputs = self.activation2(self.layer2(outputs))
    outputs = self.activation3(self.layer3(outputs))
    return outputs

  def extra_repr(self):
    return f"{[p for p in self.parameters()]}"
```

In [152…
```
list_of_neurons = [3,5,1]
list_of_activations = 3*[torch.nn.ReLU()]
mlp = My_MLP(list_of_neurons, list_of_activations)
print(list_of_neurons)
print(list_of_activations)
print(mlp)
```

```
[3, 5, 1]
[ReLU(), ReLU(), ReLU()]
My_MLP(
  [<UninitializedParameter>, <UninitializedParameter>, Parameter containing:
  tensor([[ 0.2323, -0.3665, -0.1739],
          [-0.0948,  0.0778,  0.3858],
          [-0.0061, -0.1340, -0.2615],
          [-0.3173,  0.1205, -0.5644],
          [-0.5141, -0.2040, -0.4174]], requires_grad=True), Parameter containing:
  tensor([ 0.2320, -0.3404,  0.1516,  0.0970, -0.2009], requires_grad=True), Parameter containing:
  tensor([[-0.3354,  0.1948,  0.2246,  0.4458,  0.2878]], requires_grad=True), Parameter containing:
  tensor([-0.2756], requires_grad=True)]
  (layer1): LazyLinear(in_features=0, out_features=3, bias=True)
  (layer2): Linear(in_features=3, out_features=5, bias=True)
  (layer3): Linear(in_features=5, out_features=1, bias=True)
  (activation1): ReLU()
  (activation2): ReLU()
  (activation3): ReLU()
)
```

Interesante. Esta vez sí obtenemos información al hacer un **print** y parece que, además de los parámetros como le hemos indicado, noes

muestra de manera ordenada las capas y las funciones de activación que hay en nuestro modelo. ¿Cuál era el problema con un modelo con un número de capas flexible? Pues un tecnicismo de PyTorch. Si usamos un modelo como sub-modelo, PyTorch espera que sea en un atributo de orden superior (que no esté dentro de una estructura de datos como una lista o un diccionario). Para poder usar listas o diccionarios sin que PyTorch pase por alto esos sub-modulos hay que usar las clases especiales **torch.nn.ModuleList** y **torch.nn.ModuleDict**. Vamos a verlo:

In [153…
```
class My_MLP(torch.nn.Module):
  def __init__(self, list_of_neurons, list_of_activations, device='cpu'):
    super().__init__()         # Llamamos al constructor de la clase de la que heredamos por si tuviera algo que
    self.list_of_layers = nn.ModuleList()
    self.list_of_activations = nn.ModuleList(list_of_activations)

    assert len(list_of_neurons)==len(list_of_activations)          # Sanity Check.

    for n in list_of_neurons:
      if len(self.list_of_layers)==0:
        self.list_of_layers.append(nn.LazyLinear(n).to(device))
      else:
        self.list_of_layers.append(nn.Linear(previous_n, n).to(device))
      previous_n=n

  def forward(self, inputs):
    print(inputs.shape)
    outputs = None
    for layer, activation in zip(self.list_of_layers, self.list_of_activations):
      outputs = activation(layer(inputs)) if outputs is None else activation(layer(outputs))
    return outputs
```

In [154…
```
list_of_neurons = [3,5,1]
list_of_activations = 3*[torch.nn.ReLU()]
mlp = My_MLP(list_of_neurons, list_of_activations)
print(list_of_neurons)
print(list_of_activations)
print(mlp)
```

```
[3, 5, 1]
[ReLU(), ReLU(), ReLU()]
My_MLP(
  (list_of_layers): ModuleList(
    (0): LazyLinear(in_features=0, out_features=3, bias=True)
    (1): Linear(in_features=3, out_features=5, bias=True)
    (2): Linear(in_features=5, out_features=1, bias=True)
  )
  (list_of_activations): ModuleList(
    (0-2): 3 x ReLU()
  )
)
```

Hagamos una inferencia rápida.

In [155…
```
y = mlp(torch.tensor([[10]]).float())
```

```
torch.Size([1, 1])
```

El resultado de la inferencia da igual, lo importante es que la podemos hacer y que si observamos ahora mlp...

In [156…
```
print(mlp)
```

```
My_MLP(
  (list_of_layers): ModuleList(
    (0): Linear(in_features=1, out_features=3, bias=True)
    (1): Linear(in_features=3, out_features=5, bias=True)
    (2): Linear(in_features=5, out_features=1, bias=True)
  )
  (list_of_activations): ModuleList(
    (0-2): 3 x ReLU()
  )
)
```

¡Ya no hay capa **LazyLinear**! En su lugar hay una capa **Linear** con una única entrada. Deduce el tamaño de la entrada durante el primer paso de **forward**.

Ya funciona como debe. Es importantísimo que PyTorch detecte cuáles son los parámetros optimizables del modelo o el método **parameters()** no podrá devolverlos y hay que porporcionarselos al optimizador para que pueda realizarse el ajuste!

Antes de pasar al siguiente tema, es importante señalar un detalle. Hasta la fecha, siempre que hemos usado un modelo de PyTorch para hacer el paso hacia delante (*forward*), lo hemos llamado con una instrucción del tipo **model(x)**. Como si fuera un método, aunque el modelo es un objeto. Por defecto, esa instrucción lo que hace es llamar al método **__call__()**, un método heredado que no hace falta definir en la subclase. Sería lo mismo hacer **model.__call__(x)**. Uno podría pensar que en vez de hacerlo así, se podría hacer **model.forward(x)**, ya que definimos el método **forward()** para la subclase. Sin embargo, aunque **__call__()** hace una llamada a **forward()**,

no es lo único que hace. **__call__()** realiza cálculos extra que PyTorch necesita para su buen funcionamiento y llamar directamente a **forward()** podría llevar a comportamientos indeseados. Aunque definamos el **forward()**, la llamada para hacer un *forward* siempre debe ser a través del **__call__()**. Además es más sencillo.

Ahora toca hablar de otra cosa puramente práctica, pero importante. ¿Cómo guardamos un modelo una vez entrenado?

PyTorch ofrece distintas alternaetivas para guardar diversos elementos según lo que se desea hacer después con esos modelos. El uso más intuitivo de un modelo una vez entrenado es usarlo para hacer inferencia (predecir respuestas para datos nuevos). Sin embargo, uno también puede querer guardar un modelo para continuar con el entrenamiento más tarde o para permitir hacer *fine-tuning* (de eso ya hablaremos más adelante).

En este tutorial se comentan las diversas maneras de guardar y cargar un modelo según lo que se desea poder hacer después: https://docs.pytorch.org/tutorials/beginner/saving_loading_models.html

Nosotros vamos a comentar aquí la más básica y directa, pero para casi todas hay que entender el concepto de **state_dict**. ¿Qué es? Como su nombre indica, es un diccionario que define el estado de algo en PyTorch. Ese algo puede ser un modelo, pero también hay otros elementos con **state_dict** (como los optimizadores). Veamos el del modelo anterior.

In [157…
```
print(mlp.state_dict())
```

```
OrderedDict({'list_of_layers.0.weight': tensor([[ 0.0145],
        [ 0.9210],
        [-0.0457]]), 'list_of_layers.0.bias': tensor([ 0.5404, -0.4528,  0.3808]), 'list_of_layers.1.weight': te
nsor([[-0.4683, -0.2743,  0.0049],
        [ 0.3161, -0.1513, -0.2963],
        [ 0.0941,  0.3087,  0.1365],
        [ 0.2028, -0.2521, -0.1802],
        [-0.3079,  0.5393, -0.1272]]), 'list_of_layers.1.bias': tensor([ 0.4349, -0.0700,  0.3118, -0.0064, -0.3
699]), 'list_of_layers.2.weight': tensor([[ 0.4012, -0.1140,  0.2599, -0.2850,  0.1847]]), 'list_of_layers.2.bia
s': tensor([0.3774])})
```

Como se puede ver, el **state_dict** no es más que una forma estructurada de devolver los parámetros optimizables asociándolos a las capas. Si queremos guardar un modelo, lo que hacemos es guardar ese state dict. Vamos a hacer una pequeña prueba guardando el modelo anterior y cargándolo.

In [158…
```
from google.colab import drive
drive.mount('/content/drive')
workpath = '/content/drive/MyDrive/Work/Docencia UMA/2025-2026/Programacion para la IA/data'
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", forc
e_remount=True).
```

In [159…
```
torch.save(mlp.state_dict(), os.path.join(workpath, 'mlp.pt'))
mlp2 = My_MLP(list_of_neurons, list_of_activations)
mlp2.load_state_dict(torch.load(os.path.join(workpath, 'mlp.pt'), weights_only=True))
```

Out[159… `<All keys matched successfully>`

Todo bien. Vamos a echar un vistazo a la estructura de capas de ambos y a sus **state_dict**.

In [160…
```
print(mlp)
print(mlp.state_dict())
print(mlp2)
print(mlp2.state_dict())
```

```
My_MLP(
  (list_of_layers): ModuleList(
    (0): Linear(in_features=1, out_features=3, bias=True)
    (1): Linear(in_features=3, out_features=5, bias=True)
    (2): Linear(in_features=5, out_features=1, bias=True)
  )
  (list_of_activations): ModuleList(
    (0-2): 3 x ReLU()
  )
)
OrderedDict({'list_of_layers.0.weight': tensor([[ 0.0145],
        [ 0.9210],
        [-0.0457]]), 'list_of_layers.0.bias': tensor([ 0.5404, -0.4528,  0.3808]), 'list_of_layers.1.weight': te
nsor([[-0.4683, -0.2743,  0.0049],
        [ 0.3161, -0.1513, -0.2963],
        [ 0.0941,  0.3087,  0.1365],
        [ 0.2028, -0.2521, -0.1802],
        [-0.3079,  0.5393, -0.1272]]), 'list_of_layers.1.bias': tensor([ 0.4349, -0.0700,  0.3118, -0.0064, -0.3
699]), 'list_of_layers.2.weight': tensor([[ 0.4012, -0.1140,  0.2599, -0.2850,  0.1847]]), 'list_of_layers.2.bia
s': tensor([0.3774])})
My_MLP(
  (list_of_layers): ModuleList(
    (0): LazyLinear(in_features=0, out_features=3, bias=True)
    (1): Linear(in_features=3, out_features=5, bias=True)
    (2): Linear(in_features=5, out_features=1, bias=True)
  )
  (list_of_activations): ModuleList(
    (0-2): 3 x ReLU()
  )
)
OrderedDict({'list_of_layers.0.weight': tensor([[ 0.0145],
        [ 0.9210],
        [-0.0457]]), 'list_of_layers.0.bias': tensor([ 0.5404, -0.4528,  0.3808]), 'list_of_layers.1.weight': te
nsor([[-0.4683, -0.2743,  0.0049],
        [ 0.3161, -0.1513, -0.2963],
        [ 0.0941,  0.3087,  0.1365],
        [ 0.2028, -0.2521, -0.1802],
        [-0.3079,  0.5393, -0.1272]]), 'list_of_layers.1.bias': tensor([ 0.4349, -0.0700,  0.3118, -0.0064, -0.3
699]), 'list_of_layers.2.weight': tensor([[ 0.4012, -0.1140,  0.2599, -0.2850,  0.1847]]), 'list_of_layers.2.bia
s': tensor([0.3774])})
```

In [161]:
```
y = mlp2(torch.tensor([[10,10]]).float())
```

```
torch.Size([1, 2])
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipython-input-683350750.py in <cell line: 0>()
----> 1 y = mlp2(torch.tensor([[10,10]]).float())

/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py in _wrapped_call_impl(self, *args, **kwargs)
   1771             return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1772         else:
-> 1773             return self._call_impl(*args, **kwargs)
   1774
   1775     # torchrec tests the code consistency with the following code

/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py in _call_impl(self, *args, **kwargs)
   1782                 or _global_backward_pre_hooks or _global_backward_hooks
   1783                 or _global_forward_hooks or _global_forward_pre_hooks):
-> 1784             return forward_call(*args, **kwargs)
   1785
   1786         result = None

/tmp/ipython-input-2035331528.py in forward(self, inputs)
     18     outputs = None
     19     for layer, activation in zip(self.list_of_layers, self.list_of_activations):
---> 20       outputs = activation(layer(inputs)) if outputs is None else activation(layer(outputs))
     21     return outputs

/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py in _wrapped_call_impl(self, *args, **kwargs)
   1771             return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1772         else:
-> 1773             return self._call_impl(*args, **kwargs)
   1774
   1775     # torchrec tests the code consistency with the following code

/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py in _call_impl(self, *args, **kwargs)
   1877
   1878         try:
-> 1879             return inner()
   1880         except Exception:
   1881             # run always called hooks if they have not already been run


/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py in inner()
   1804                 ):
   1805                     if hook_id in self._forward_pre_hooks_with_kwargs:
-> 1806                         args_kwargs_result = hook(self, args, kwargs)  # type: ignore[misc]
   1807                         if args_kwargs_result is not None:
   1808                             if isinstance(args_kwargs_result, tuple) and len(args_kwargs_result) == 2:

/usr/local/lib/python3.12/dist-packages/torch/nn/modules/lazy.py in _infer_parameters(self, module, args, kwargs
)
    259         """
    260         kwargs = kwargs if kwargs else {}
--> 261         module.initialize_parameters(*args, **kwargs)
    262         if module.has_uninitialized_params():
    263             raise RuntimeError(f'module {self._get_name()} has not been fully initialized')

/usr/local/lib/python3.12/dist-packages/torch/nn/modules/linear.py in initialize_parameters(self, input)
    292                 self.reset_parameters()
    293         if self.in_features == 0:
--> 294             assert input.shape[-1] == self.weight.shape[-1], (
    295                 f"The in_features inferred from input: {input.shape[-1]} "
    296                 f"is not equal to in_features from self.weight: "

AssertionError: The in_features inferred from input: 2 is not equal to in_features from self.weight: 1
```

Y... error. Vale, sí, hemos ido a buscar el error. Pero es un buen momento para reflexionar sobre lo que estamos haciendo. Solo hemos cargado el **state_dict** en otra instancia del modelo, por lo tanto necesitamos que esa isntancia exista. ¡Necesitamos tener ya un código que monte la misma estructura de capas y lo único que hemos guardado son los parámetros en esa estructura de capas.

Pero claro, teníamos una capa **LazyLinear**. No nos ha dado problema al cargarle la información de una capa **Linear**, pero al intentar hacer un paso de *forward* con un tamaño de entrada distinto al de los parámetros que le hemos dado, no ha podido hacer coincidir las matrices y ha fallado. Si lo hacemos sin mala fe debería funcionar:

In [162…
```
mlp2 = My_MLP(list_of_neurons, list_of_activations)
mlp2.load_state_dict(torch.load(os.path.join(workpath, 'mlp.pt'), weights_only=True))
```

Out[162… <All keys matched successfully>

In [163…
```
y = mlp2(torch.tensor([[10]]).float())
```

```
torch.Size([1, 1])
```

In [164…
```
print(mlp)
```

```
My_MLP(
  (list_of_layers): ModuleList(
    (0): Linear(in_features=1, out_features=3, bias=True)
    (1): Linear(in_features=3, out_features=5, bias=True)
    (2): Linear(in_features=5, out_features=1, bias=True)
  )
  (list_of_activations): ModuleList(
    (0-2): 3 x ReLU()
  )
)
```

Todo se ha cargado como debía y nuestra **LazyLinear** se ha convertido satisfactoriamente en una capa **Linear**.

Dada una neurona lineal, sabemos que tiene n+1 parámetros (un parámetro para cada entrada más el bias). Si una capa tiene tiene k neuronas lineales, esa capa tendrá k(n+1)=kn+k parámetros.

Supongamos un perceptrón multicapa con una capa oculta con k_1 neuronas y una neurona en la capa de salida. Para n entradas, la red tendrá $k_1(n+1)+k_1+1 = k_1(n+2)+1$ parámetros.

Ahora supongamos un segundo perceptrón con dos capas ocultas de k_1 y k_2 neuronas y una capa de salida de una única neurona. La red tendrá $k_1(n+1)+k_2(k_1+1)+k_2+1 = k_1(n+1)+k_2k_1+2k_2+1$ parámetros.

Supongamos que queremos disponer 10 neuronas en las capas ocultas y la entrada es un único valor. En el primer caso, tendríamos 10(1+2)+1=31 parámetros. Si tenemos dos capas ocultas de 5 neuronas, en el segundo caso tendremos 5(2)+5(5)+2(5)+1=10+25+10+1=46 parámetros.

In [165…
```
model = torch.nn.Sequential(
    torch.nn.Linear(1,10),
    torch.nn.Linear(10,1),
)
print(torch.tensor([p.numel() for p in model.parameters()]).sum())

model = torch.nn.Sequential(
    torch.nn.Linear(1,5),
    torch.nn.Linear(5,5),
    torch.nn.Linear(5,1),
)
print(torch.tensor([p.numel() for p in model.parameters()]).sum())
```

```
tensor(31)
tensor(46)
```

¿Qué pasaría si en vez de 1 valor de entrada hubiera 10? En el primer caso los números cambiarían a 10(12)+1 = 121 parámetros, mientras que en el segundo se elevaría a 5(11)+5(6)+5+1=91.

In [166…
```
model = torch.nn.Sequential(
    torch.nn.Linear(10,10),
    torch.nn.Linear(10,1),
)
print(torch.tensor([p.numel() for p in model.parameters()]).sum())

model = torch.nn.Sequential(
    torch.nn.Linear(10,5),
    torch.nn.Linear(5,5),
    torch.nn.Linear(5,1),
)
print(torch.tensor([p.numel() for p in model.parameters()]).sum())
```

```
tensor(121)
tensor(91)
```

Tener una única capa oculta hace que los parámetros crezcan más rápido en relación al tamaño de la entrada que si hacemos redes más profundas. Las redes más profundas además dotan al modelo de mayor flexibilidad a menor coste. Intuitivamente añadir capas es componer funciones. Incluso con el mismo número de neuronas.

$y=f_1(f_2(f_3(f_4(x))))$

Cada función aplica una transformación que la siguiente puede "aprovechar". Si hablaramos de neuronas convolucionales diríamos que cada capa aprende características que el siguiente nivel puede componer enc aracterísticas de más alto nivel.

En la práctica, elegir la arquitectura de la red atiende a unos criterios estrictos de limitación física relativos a la memoria: más parámetros requieren más espacio en memoria y provocan entrenamientos más lentos. Además, redes más profundas también provocan entrenamientos más lentos de por sí.

Recordemos que, aunque hasta ahora no lo hayamos hecho, las redes se entrenan típicamente en GPUs en las que se busca maximizar el paralelismo, pero cada capa secuencial tiene que esperar al resultado de la capa anterior para poder computarse, con lo que se está forzando un cómputo en secuencia en lugar de en paralelo.

Hay un equilibrio que al final responde a la prueba y el error, el hardware disponible y el problema.