Lecture 04b
Artificial Intelligence

Agenda

- Arquitectura de los perceptrones multicapa
- Funciones de activación no lineales
- Ejemplos de perceptrón multicapa
- *Overfitting* y *Underfitting*
- *Dataloaders* personalizados

Redes neuronales *feedforward* completamente conectadas con una o más capas ocultas

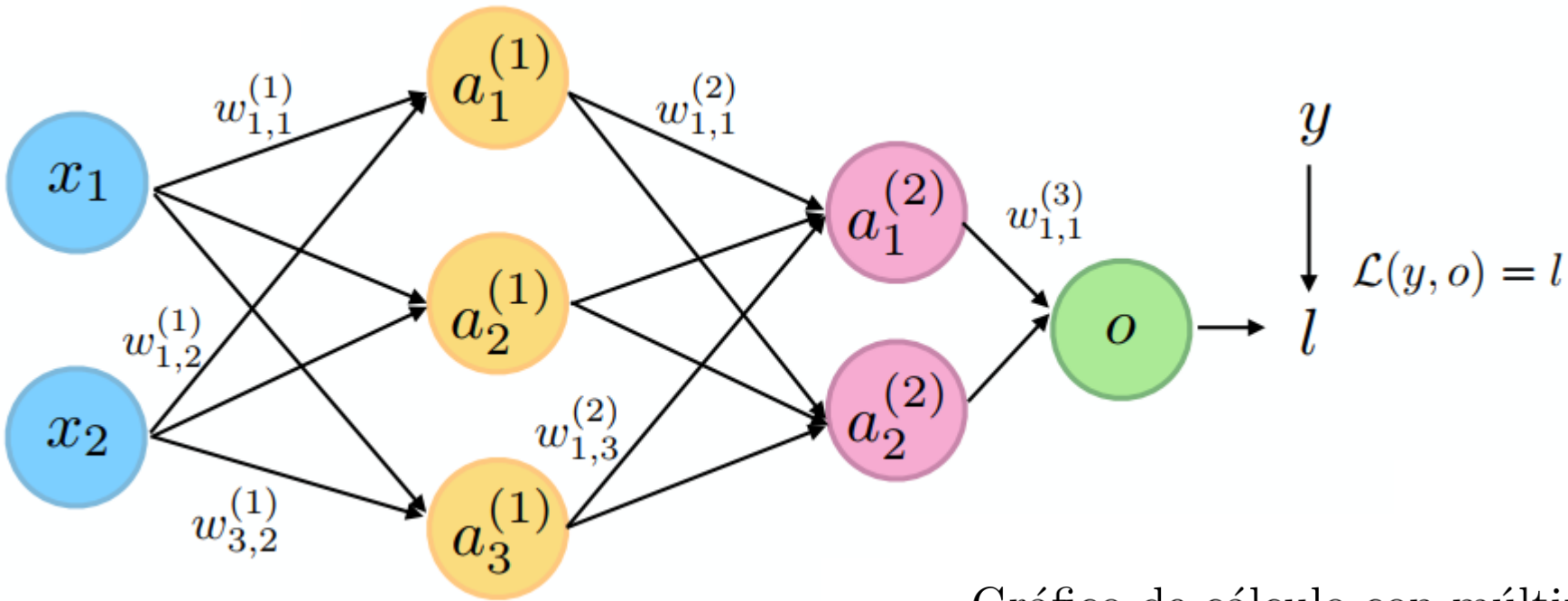
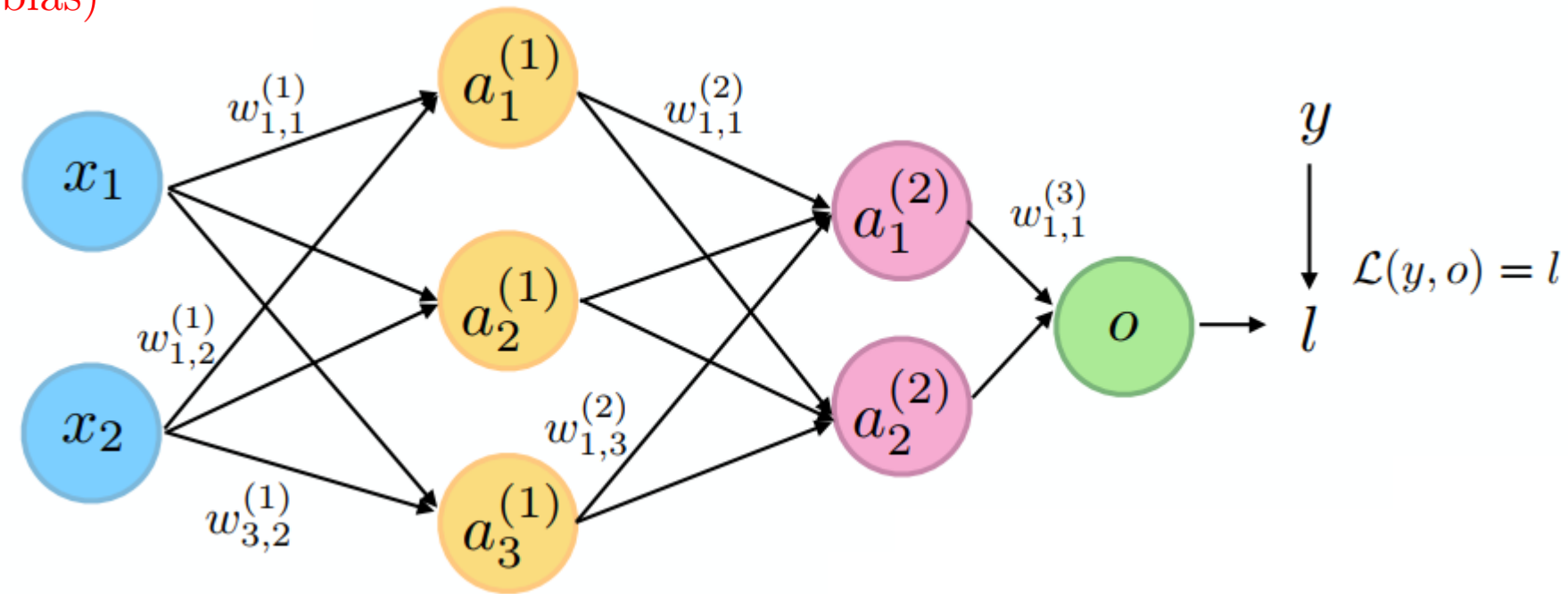


Gráfico de cálculo con múltiples capas completamente conectadas = Percetrón multicapa

Realmente nada nuevo

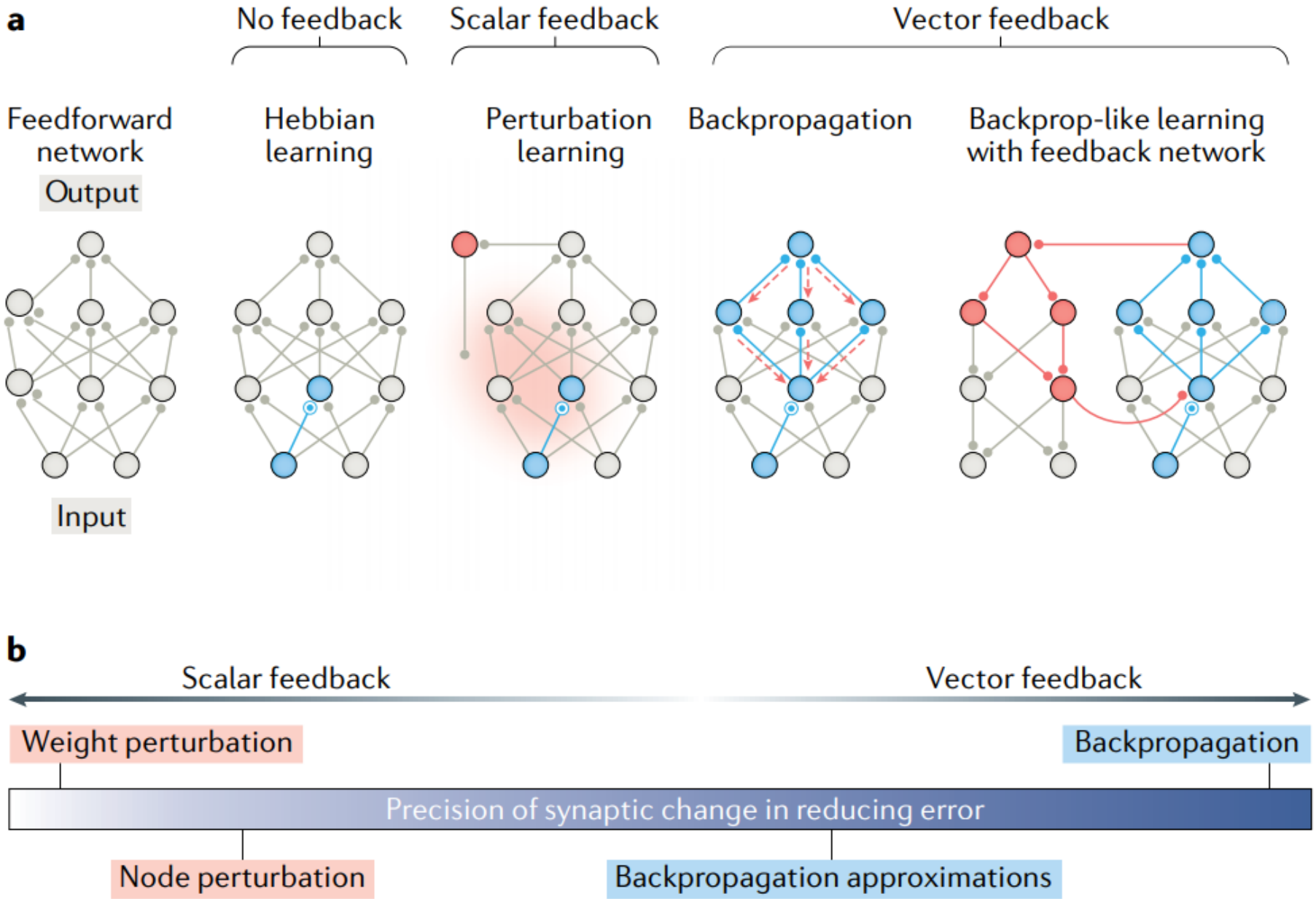
(no mostramos el bias)



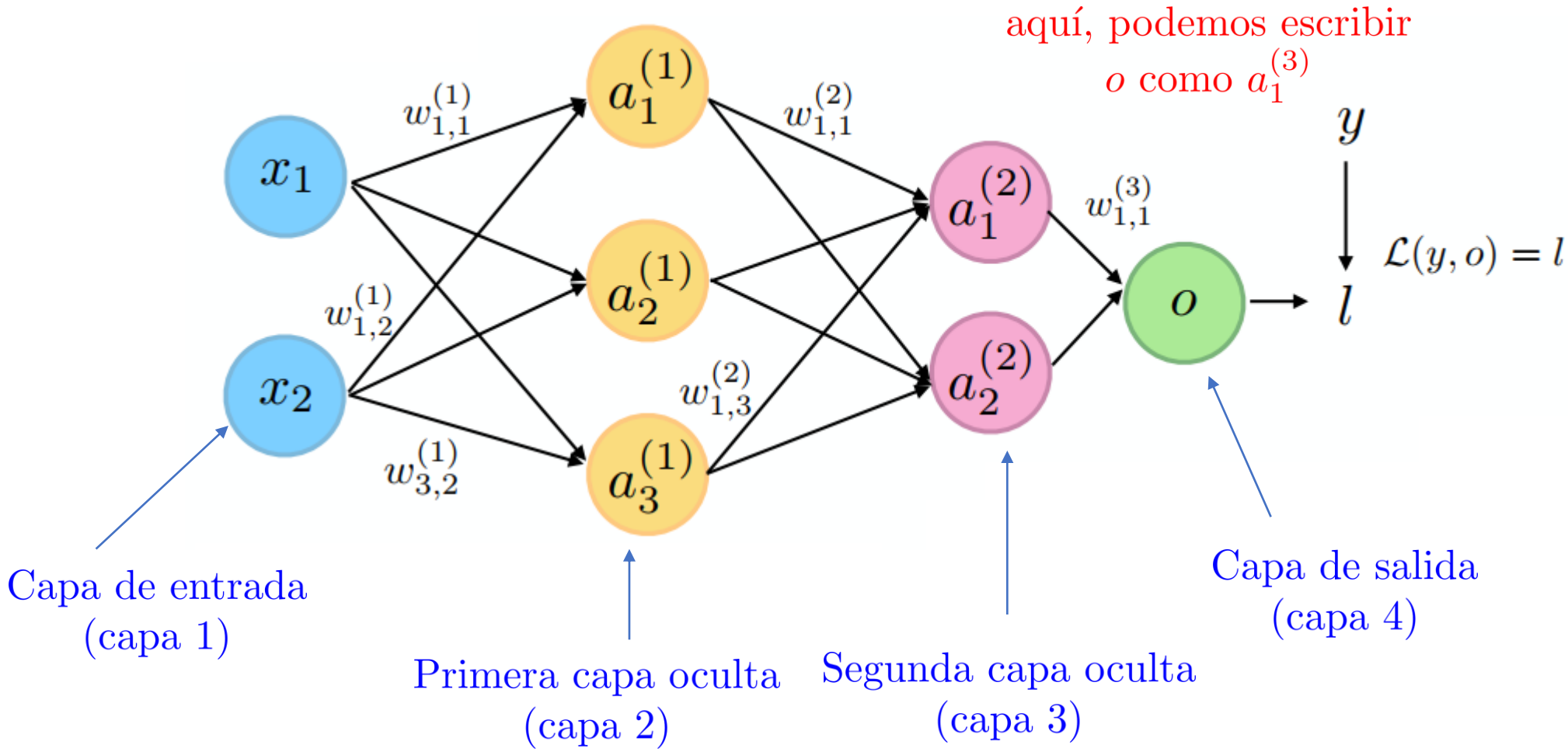
donde $o := \sigma(z) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$

(Asumiendo la red para clasificación binaria)

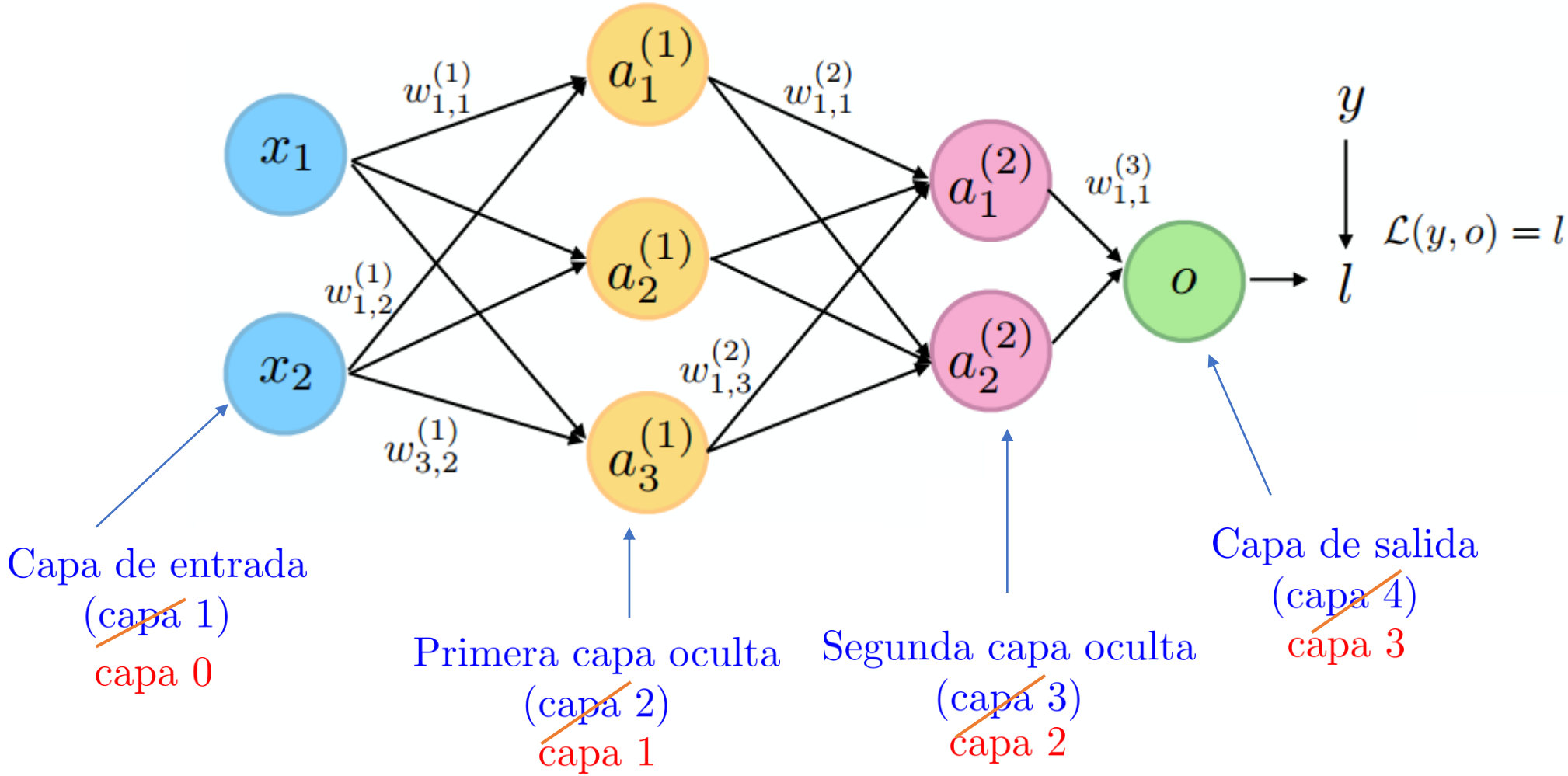
$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

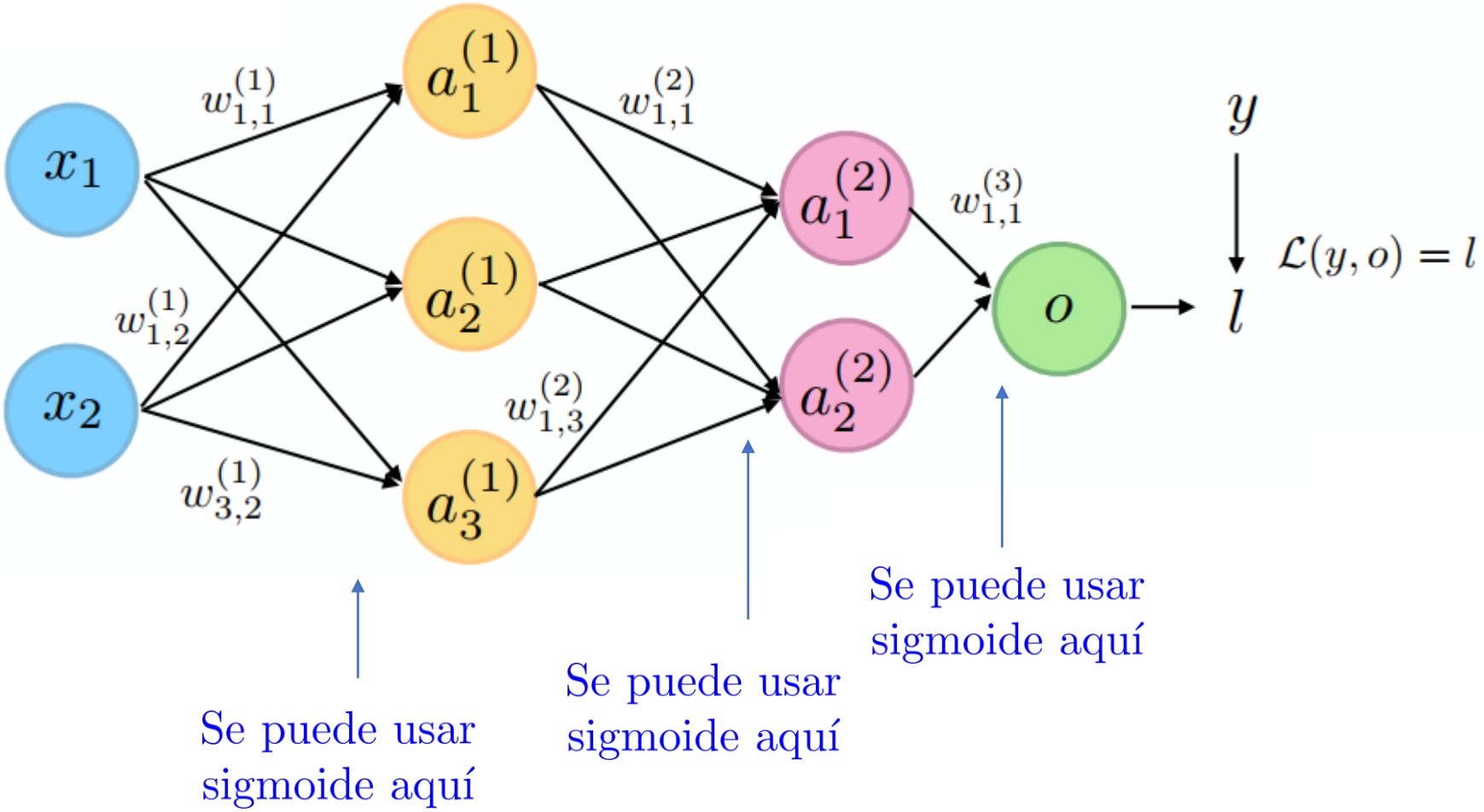


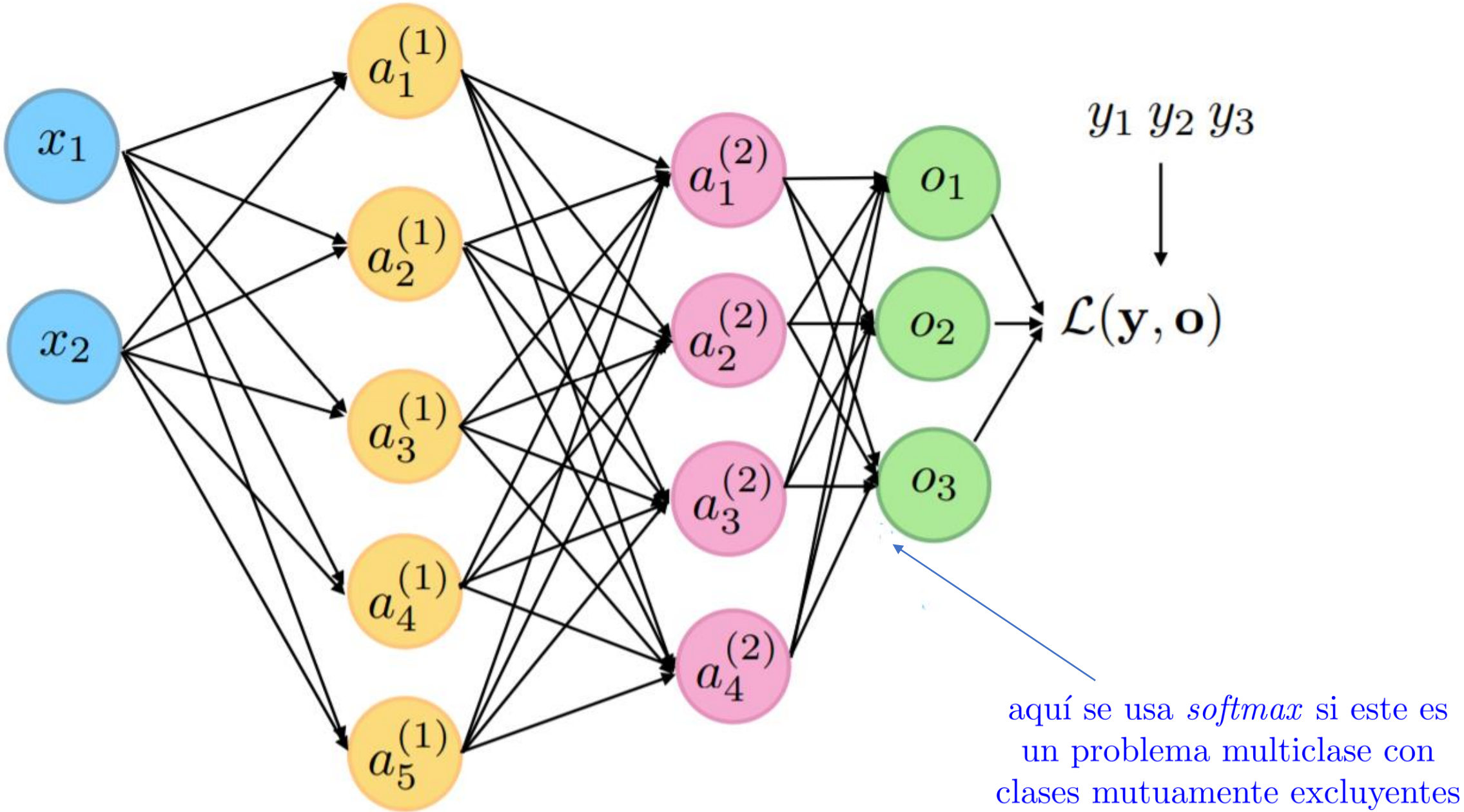
Lillicrap, T. P., Santoro, A., Marris, L., & Akerman, C. (n.d.). Backpropagation and the brain. Nature Reviews Neuroscience, 1–12. https://doi.org/10.1038/s41583-020-0277-3


aquí, podemos escribir
o como $a_1^{(3)}$
x_1
x_2
$a_1^{(1)}$
$a_2^{(1)}$
$a_3^{(1)}$
$a_1^{(2)}$
$a_2^{(2)}$
o
y
l
$\mathcal{L}(y, o) = l$
$w_{1,1}^{(1)}$
$w_{1,2}^{(1)}$
$w_{3,2}^{(1)}$
$w_{1,1}^{(2)}$
$w_{1,3}^{(2)}$
$w_{1,1}^{(3)}$
Capa de entrada
(capa 1)
Primera capa oculta
(capa 2)
Segunda capa oculta
(capa 3)
Capa de salida
(capa 4)

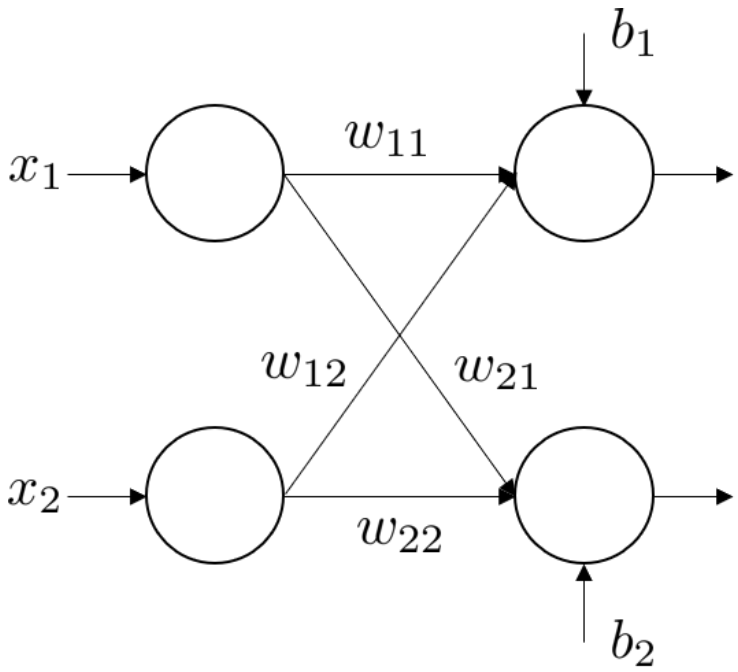
Un esquema de nomenclatura más común, porque entonces un perceptron/Adaline/modelo de regresión lineal puede ser llamado *red neuronal de 1-capa*




x_1
x_2
$a_1^{(1)}$
$a_2^{(1)}$
$a_3^{(1)}$
$a_1^{(2)}$
$a_2^{(2)}$
o
$w_{1,1}^{(1)}$
$w_{1,2}^{(1)}$
$w_{3,2}^{(1)}$
$w_{1,1}^{(2)}$
$w_{1,3}^{(2)}$
$w_{1,1}^{(3)}$
y
l
$\mathcal{L}(y, o) = l$
Se puede usar sigmoide aquí
Se puede usar sigmoide aquí
Se puede usar sigmoide aquí


x_1
x_2
$a_1^{(1)}$
$a_2^{(1)}$
$a_3^{(1)}$
$a_4^{(1)}$
$a_5^{(1)}$
$a_1^{(2)}$
$a_2^{(2)}$
$a_3^{(2)}$
$a_4^{(2)}$
o_1
o_2
o_3
$y_1\ y_2\ y_3$
$\mathcal{L}(\mathbf{y}, \mathbf{o})$
aquí se usa *softmax* si este es un problema multiclase con clases mutuamente excluyentes

Intuición sobre las redes neuronales *FeedForward*



$$z_1 = w_{11}x_1 + w_{12}x_2 + b_1$$

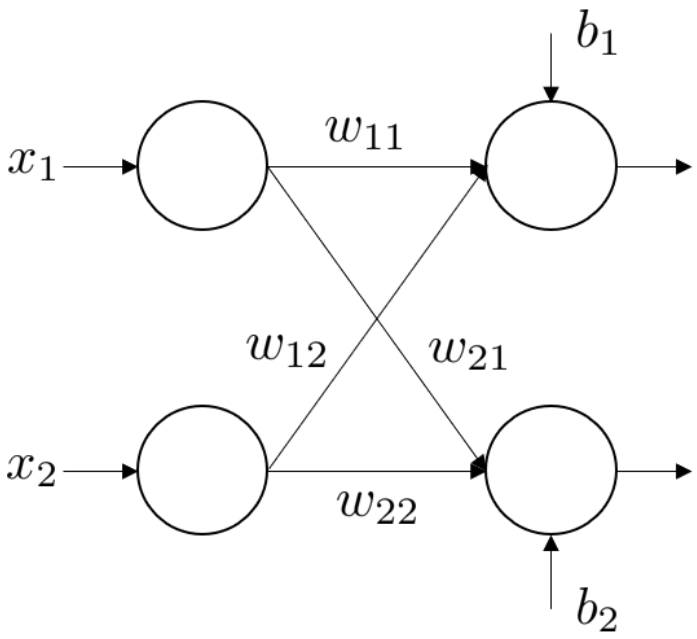
$$z_2 = w_{21}x_1 + w_{22}x_2 + b_2$$

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\mathbf{z} \in \Re^{h_{out} \times 1} \qquad \mathbf{W} \in \Re^{h_{out} \times h_{in}} \qquad \mathbf{x} \in \Re^{h_{in} \times 1} \qquad \mathbf{b} \in \Re^{h_{out} \times 1}$$

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \qquad \mathbf{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Intuición sobre las redes neuronales *FeedForward*



$$\mathbf{X} \in \Re^{n \times h_{in}}$$

$$\mathbf{X} = \begin{bmatrix} x_1^{[1]} & \dots & x_{h_{in}}^{[1]} \\ \vdots & \vdots & \vdots \\ x_1^{[n]} & \dots & x_{h_{in}}^{[n]} \end{bmatrix}$$

$$\mathbf{W} \in \Re^{h_{out} \times h_{in}}$$

$$\mathbf{w}_h \in \Re^{h_{in} \times 1}$$

$$\mathbf{W} = \begin{bmatrix} - - \mathbf{w}_1^\top - - \\ \vdots \\ - - \mathbf{w}_{h_{out}}^\top - - \end{bmatrix}$$

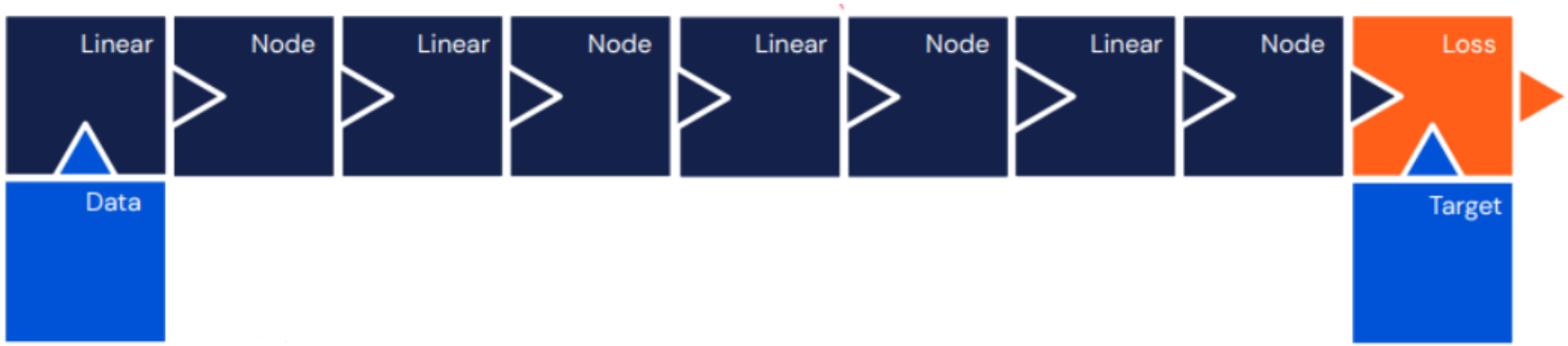
$$\mathbf{b} \in \Re^{h_{out} \times 1}$$

$$\mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_{h_{out}} \end{bmatrix}$$

$$\mathbf{Z} \in \Re^{n \times h_{out}}$$

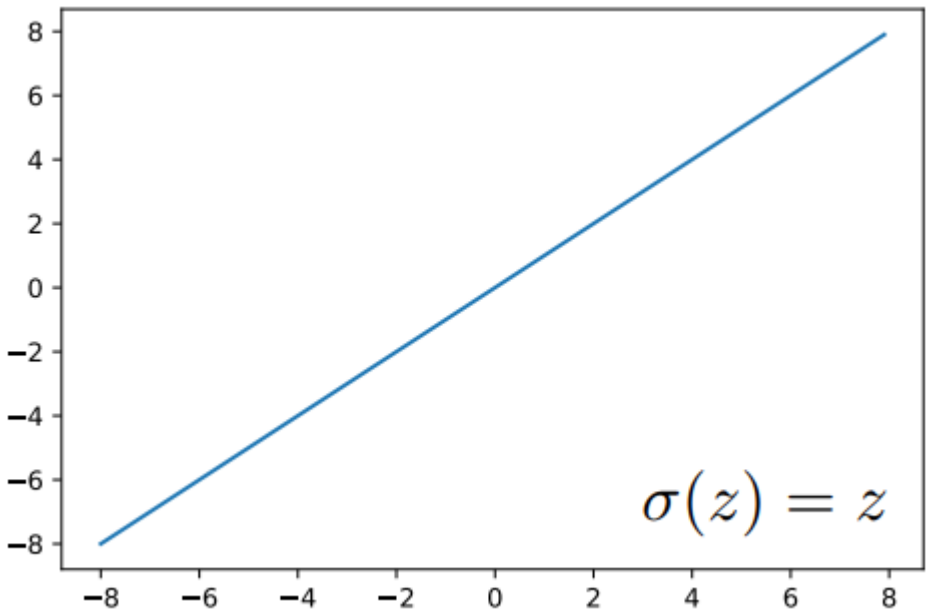
$$\mathbf{Z} = \mathbf{X}\mathbf{W}^\top + \mathbf{b}$$

Intuición sobre las redes neuronales *FeedForward*

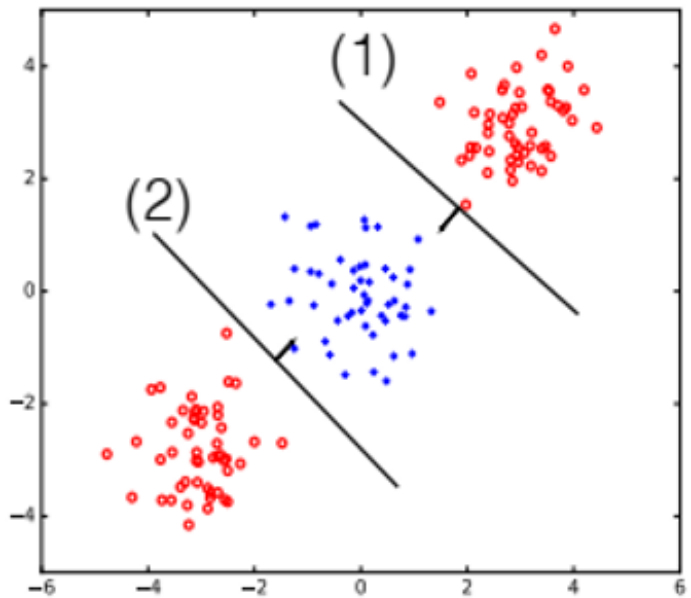


Activaciones lineales

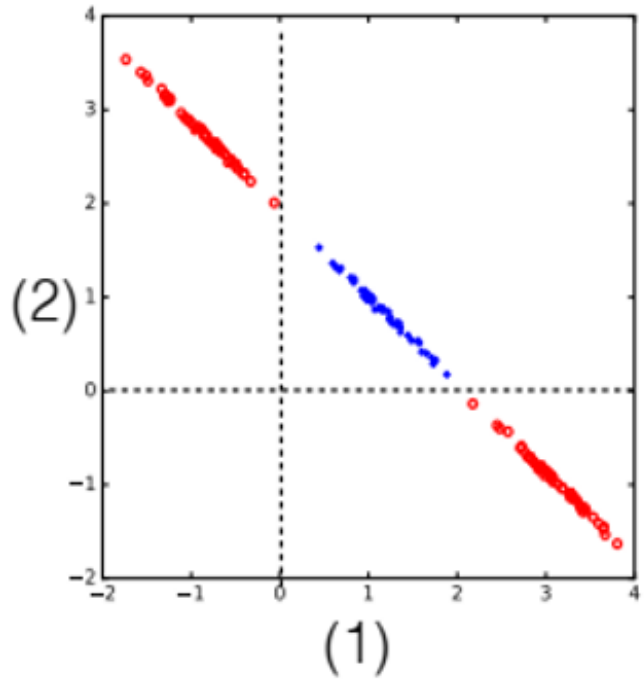
Identidad



Hidden layer units

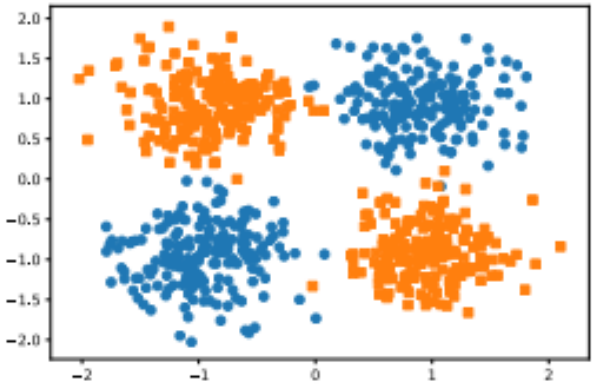


Linear activation

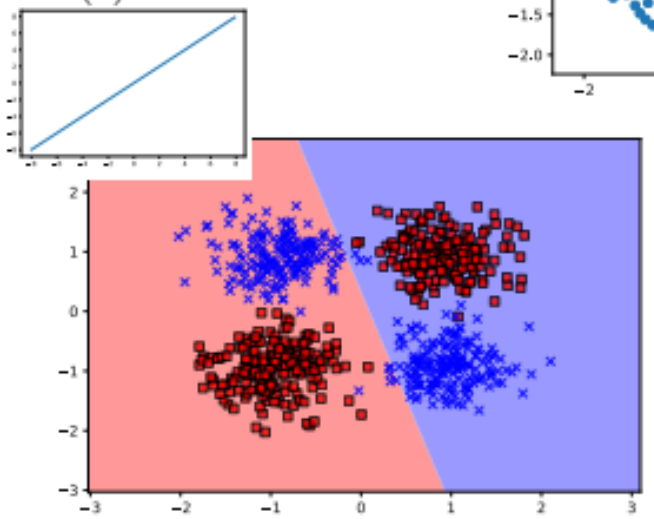


Activaciones no lineales

Podríamos resolver el problema de la XOR!

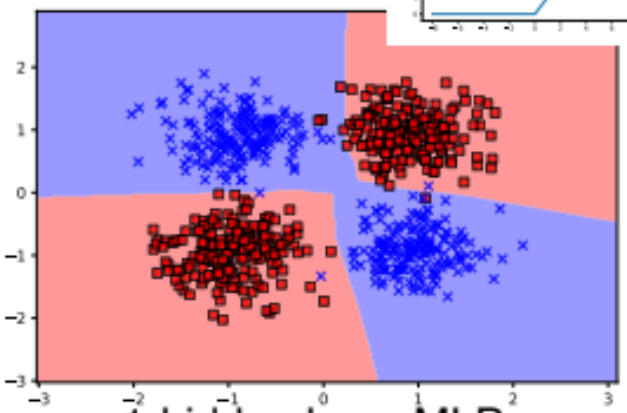


$\sigma(z) = z$



1-hidden layer MLP
with linear activation function

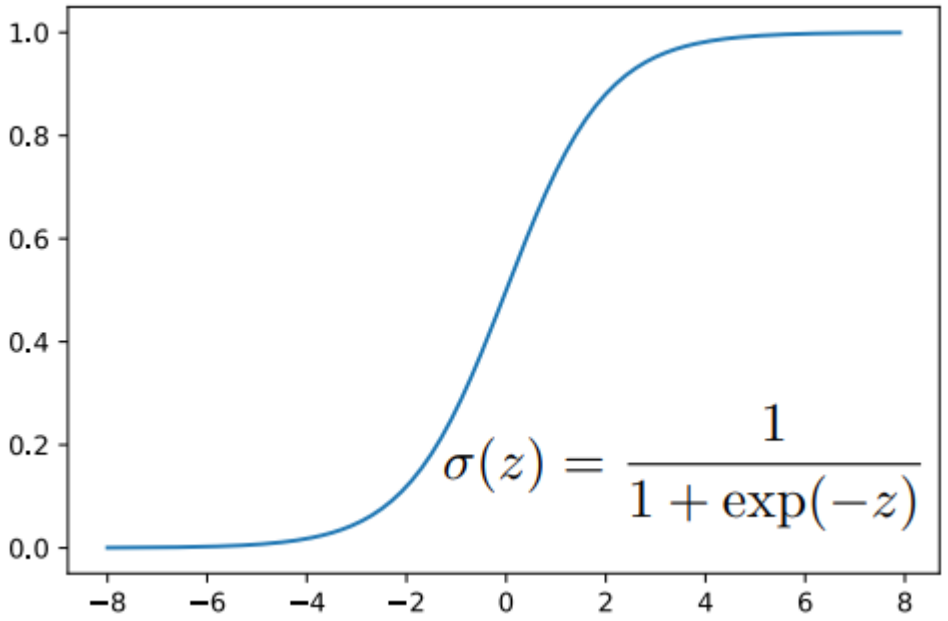
$\sigma(z) = \max(0, z)$



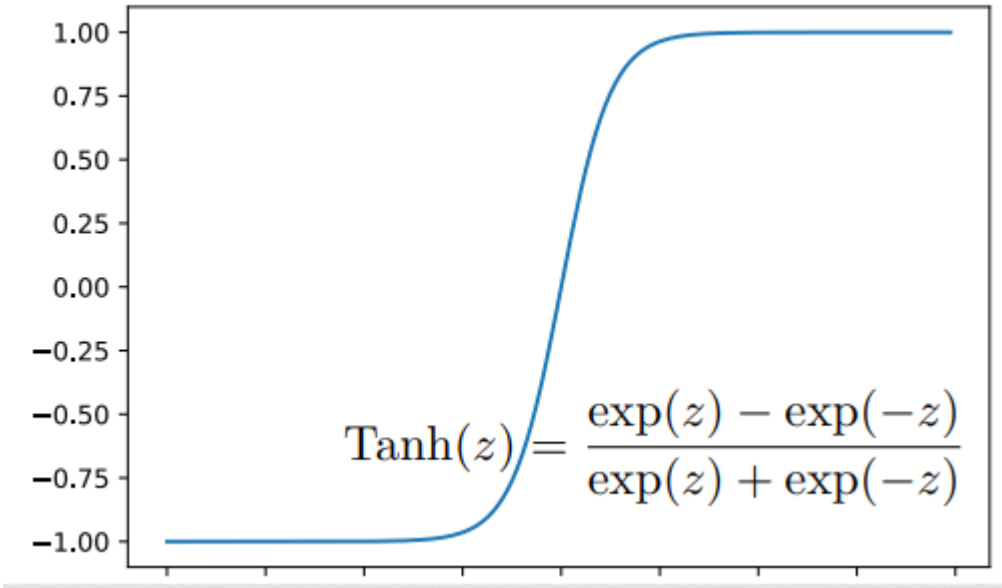
1-hidden layer MLP
with non-linear activation function (ReLU)

Activaciones no lineales

Sigmoide(logistic)



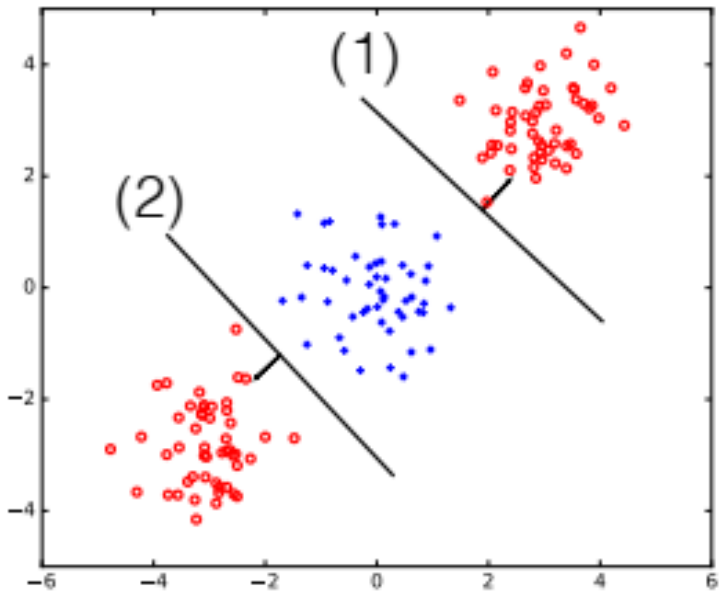
tanh



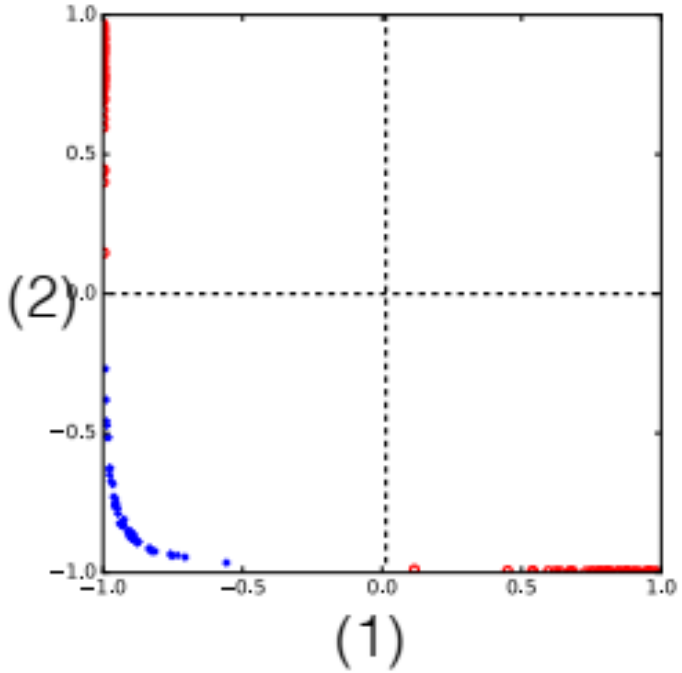
Algunas ventajas de la tangente hiporbólica son:

Media centrada, valores positivos y negativos, grandes gradientes, normaliza las entradas a media cero, derivada simple

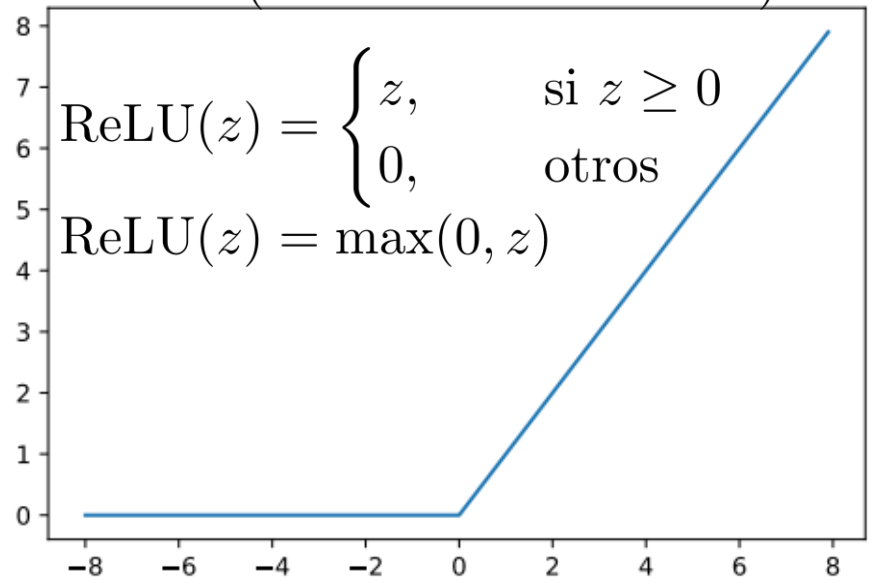
Hidden layer units



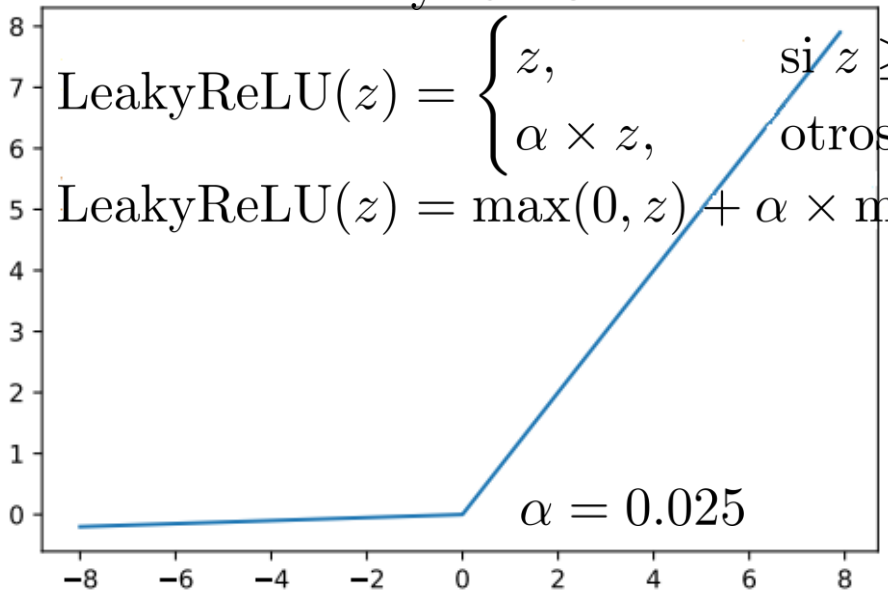
tanh activation



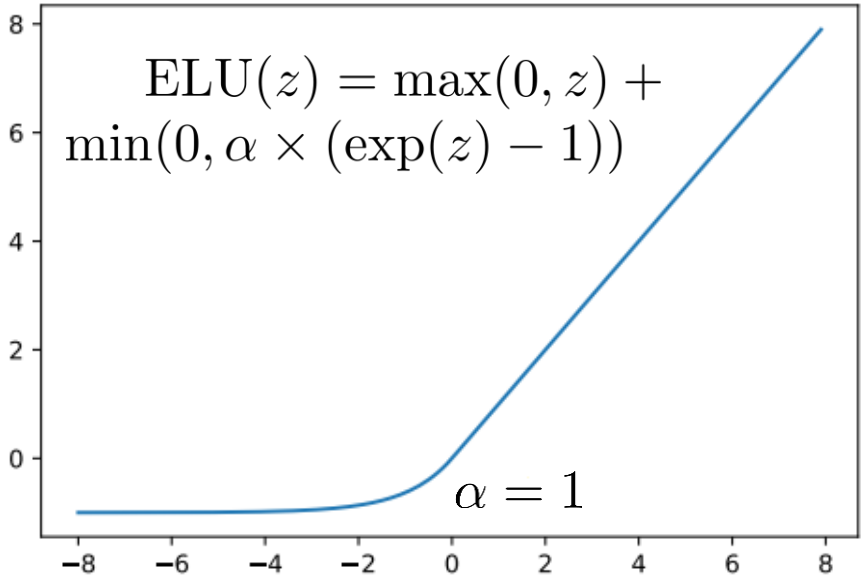
ReLU (Rectified Linear Unit)



Leaky ReLU



ELU (Exponential Linear Unit)



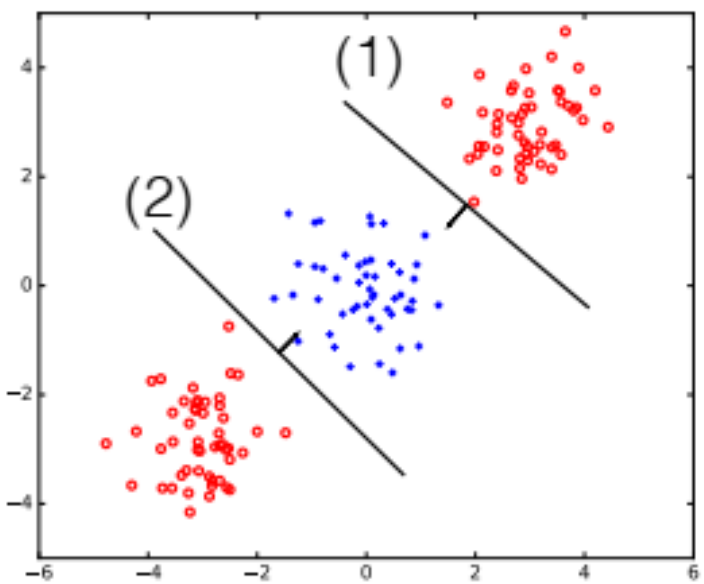
PReLU (Parameterized Rectified Linear Unit)

Aquí α es un parámetro entrenable

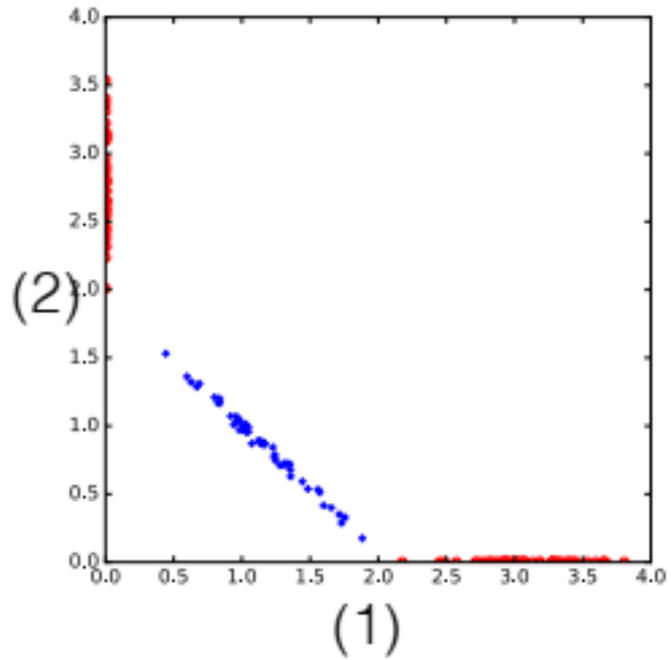
$$\mathrm{PReLU}(z) = \begin{cases} z, & \text{si } z \geq 0 \\ \alpha z, & \text{otros} \end{cases}$$

$$\mathrm{PReLU}(z) = \max(0, z) + \alpha \times \min(0, z)$$

Hidden layer units



ReLU activation



Pérdida no convexa

- Regresión lineal, Adaline, Regresión logística y Regresión *softmax* tienen funciones de pérdida convexas
- Este ya no es el caso; en la práctica, usualmente se obtiene un mínimo local diferente si se repite el entrenamiento (por ejemplo, cambiando la semilla aleatoria para la inicialización de pesos)
- En ocasiones, es deseabke explorar diferentes pesos de inicio, debido a que algunos pueden llevar a mejores soluciones que otros

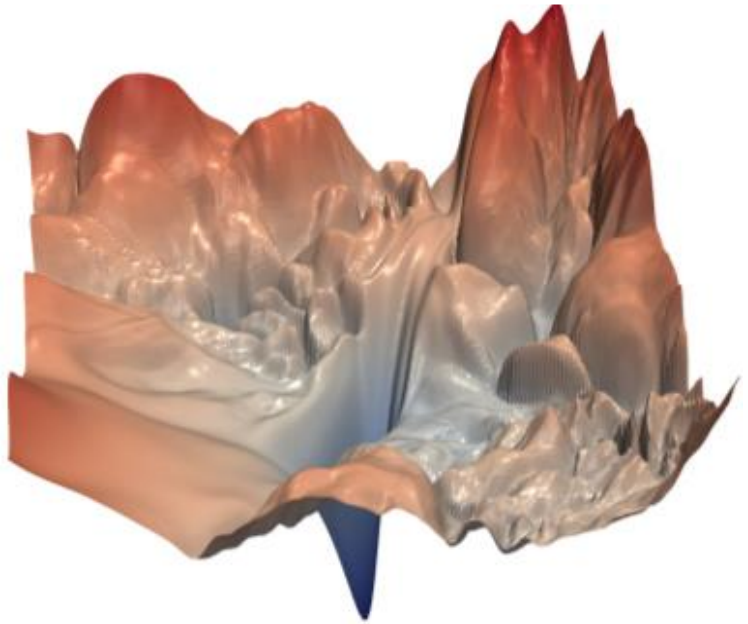


Image Source: Li, H., Xu, Z., Taylor, G., Studer, C. and Goldstein, T., 2018. Visualizing the loss landscape of neural nets. In Advances in Neural Information Processing Systems (pp. 6391-6401).

Qué pasa si inicializamos todos los pesos en 0 en el perceptrón multicapa?

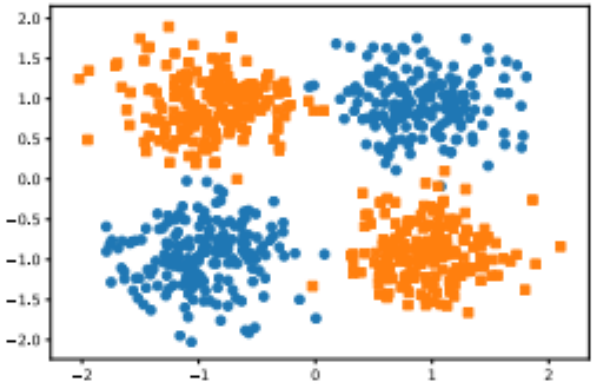
Resumen de PyTorch

```
import torch.nn.functional as F


class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features,
                                         num_hidden_1)

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_hidden_1,
                                         num_hidden_2)

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden_2,
                                          num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = self.linear_2(out)
        out = F.relu(out)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas
```

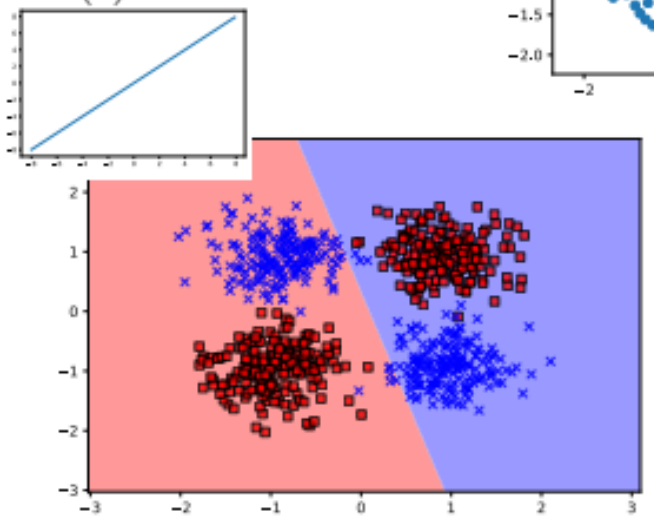
```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        self.my_network = torch.nn.Sequential(
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_2, num_classes)
        )

    def forward(self, x):
        logits = self.my_network(x)
        probas = F.softmax(logits, dim=1)
        return logits, probas
```

Activaciones no lineales

Resolvamos el problema!

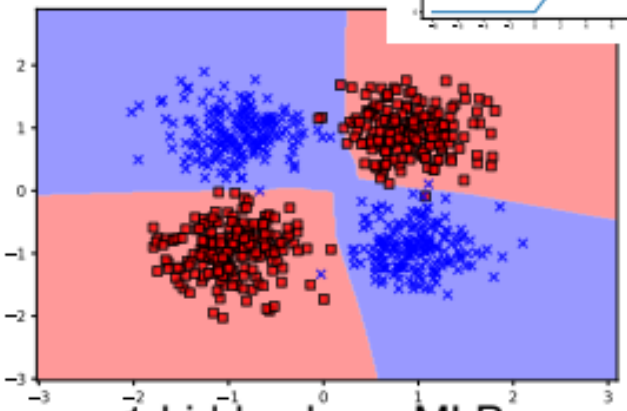


$\sigma(z) = z$



1-hidden layer MLP
with linear activation function

$\sigma(z) = \max(0, z)$



1-hidden layer MLP
with non-linear activation function (ReLU)

arXiv.org > cs > arXiv:2006.14536

Computer Science > Machine Learning

[Submitted on 25 Jun 2020]

Smooth Adversarial Training

Cihang Xie, Mingxing Tan, Boqing Gong, Alan Yuille, Quoc V. Le

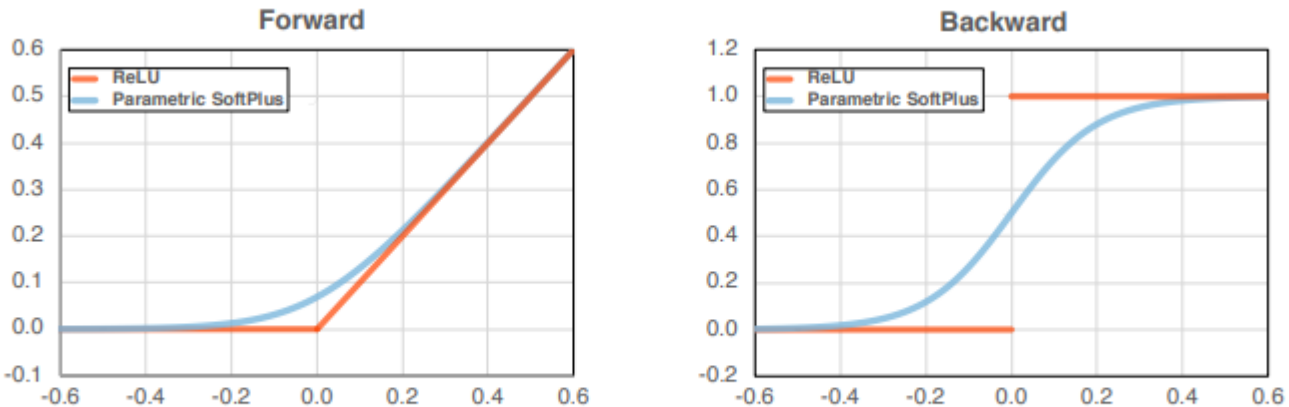


Figure 1: Left Panel: ReLU and Parametric SoftPlus. Right Panel: the first derivatives for ReLU and Parametric SoftPlus. Compared to ReLU, Parametric Softplus is smooth with continuous derivatives.

Comúnmente se cree que las redes no pueden ser precisas y robustas, que ganar robustez significa perder precisión. [...] Nuestra observación clave es que la función de activación ampliamente utilizada ReLU debilita significativamente el entrenamiento adverso debido a su naturaleza de no suavizado. Por eso proponemos *Entrenamiento adverso suavizado (EAS o SAT por sus siglas en ingles)*, en el cual reemplazamos ReLU por sus aproximaciones suavizadas para fortalecer el entrenamiento adverso.

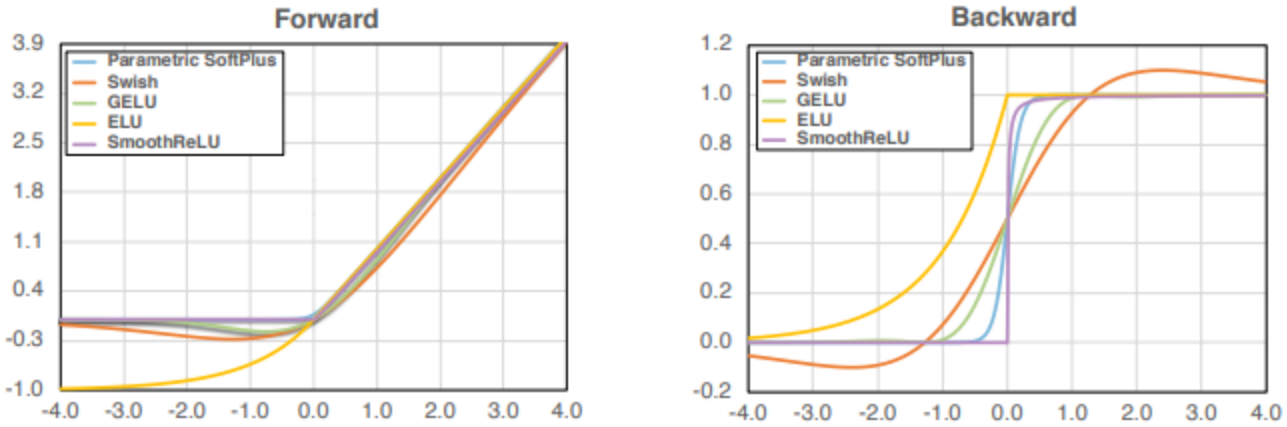


Figure 2: Visualizations of smooth activation functions and their derivatives.

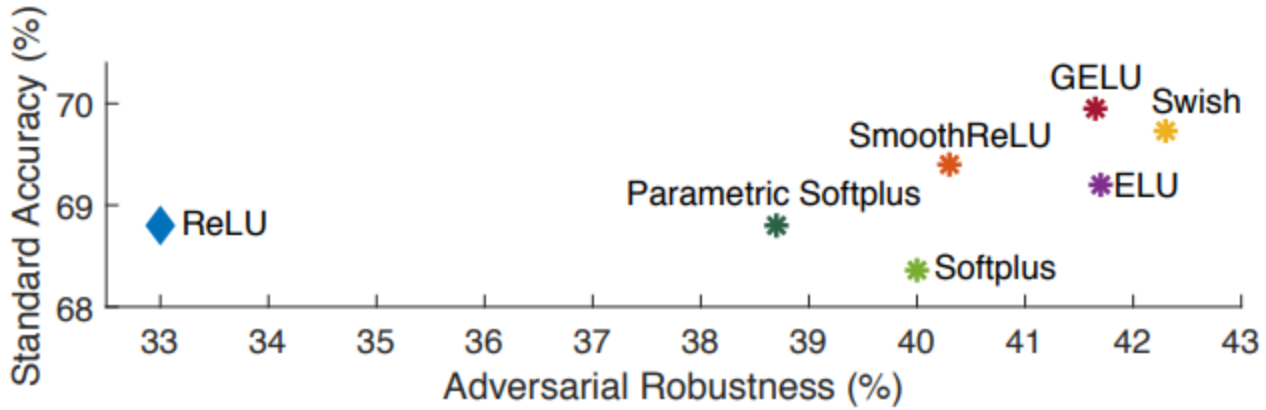
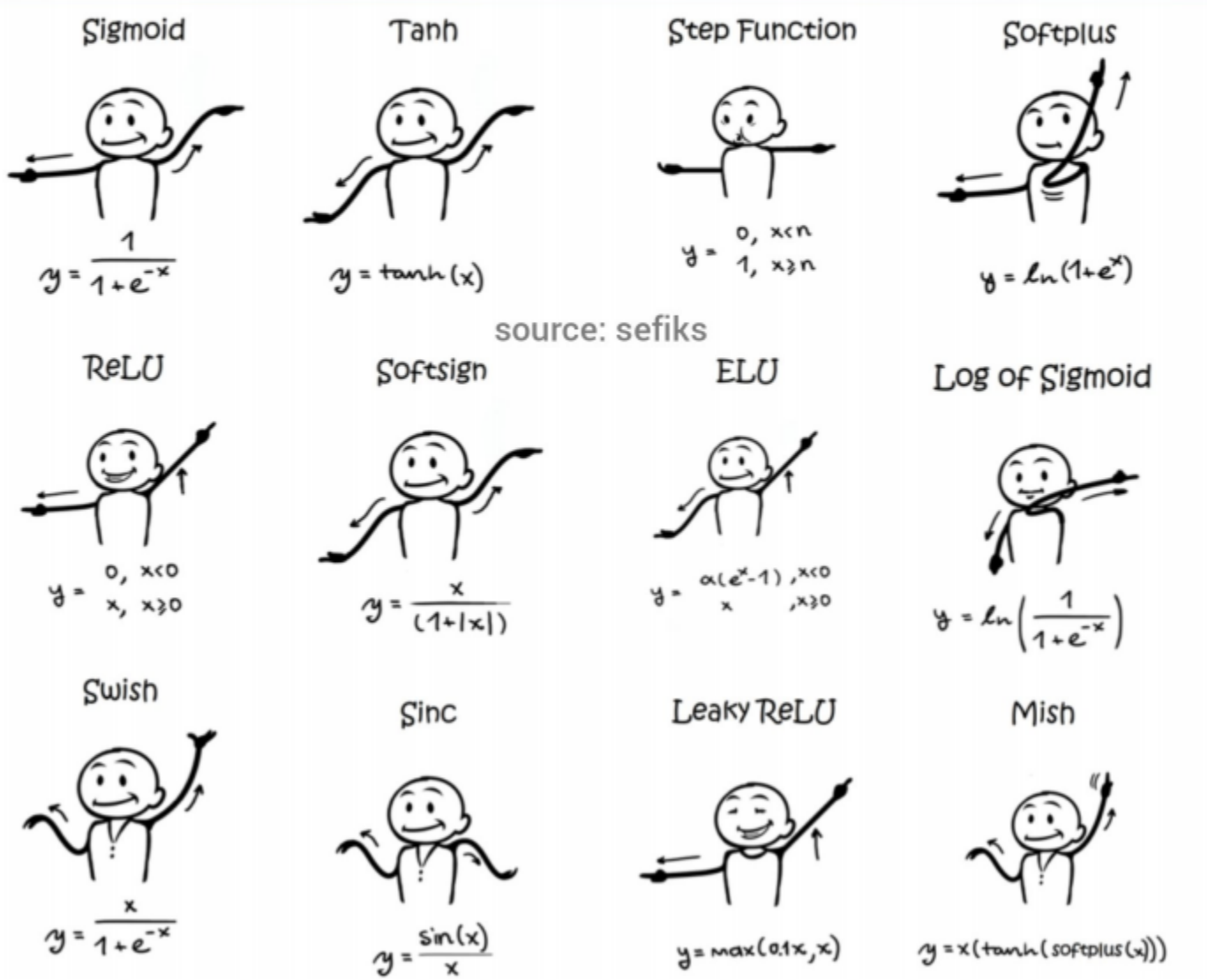


Figure 3: Smooth activation functions improve adversarial training. Compared to ReLU, all smooth activation functions significantly boost robustness, while keeping accuracy almost the same.

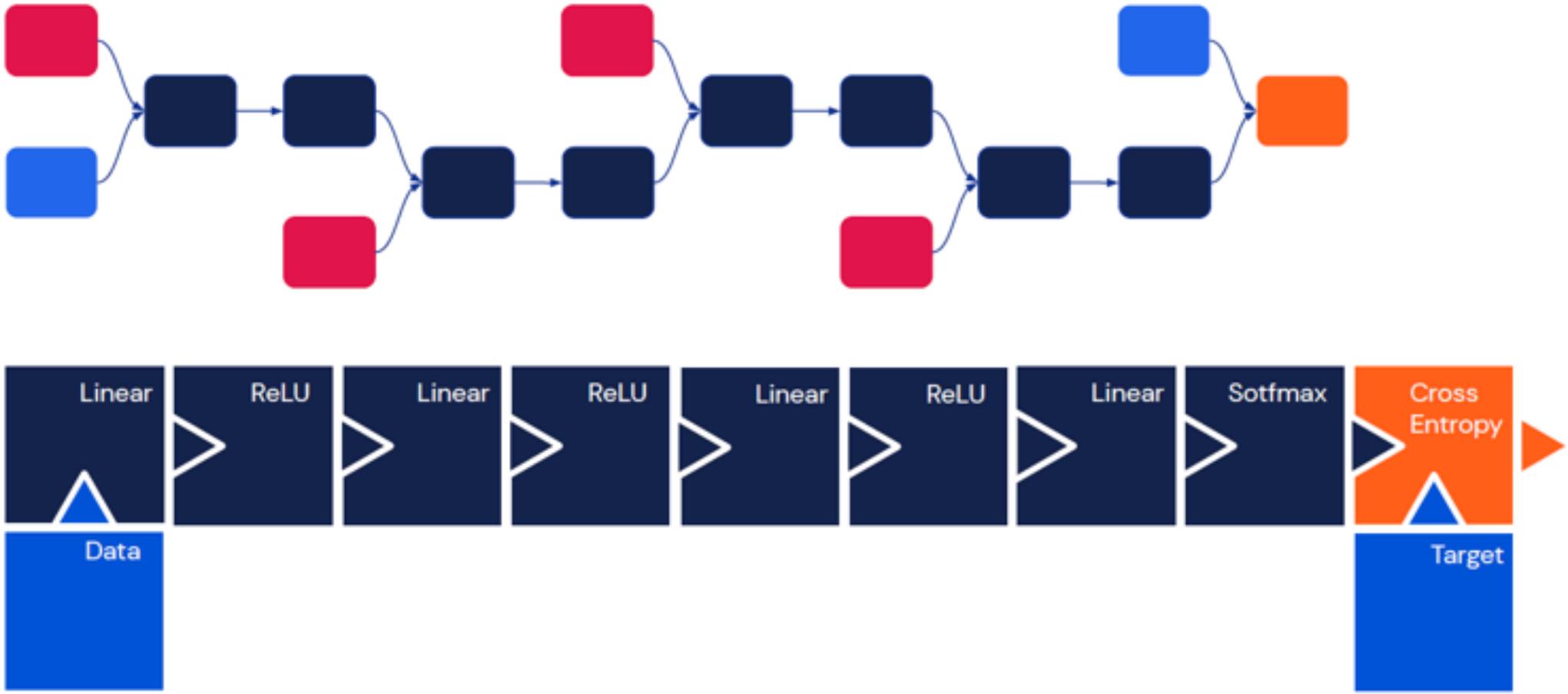
https://arxiv.org/abs/2006.14536

https://twitter.com/TheInsaneApp/status/1366324846976659461?s=20

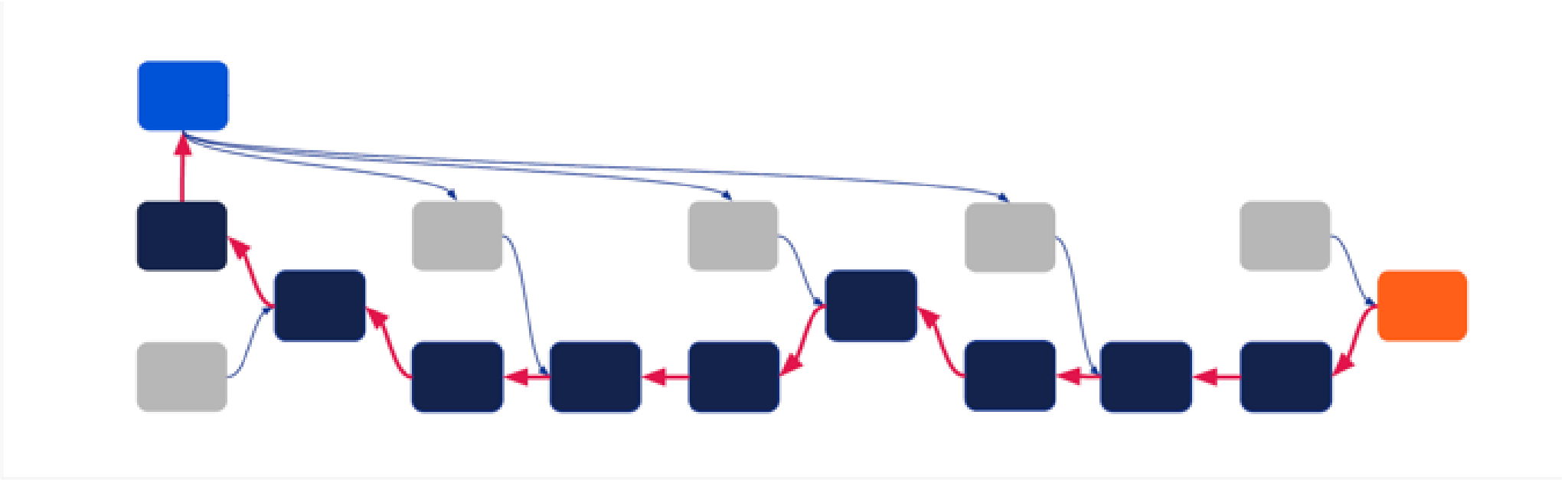
Dance Moves of Deep Learning Activation Functions



Grafo computacional



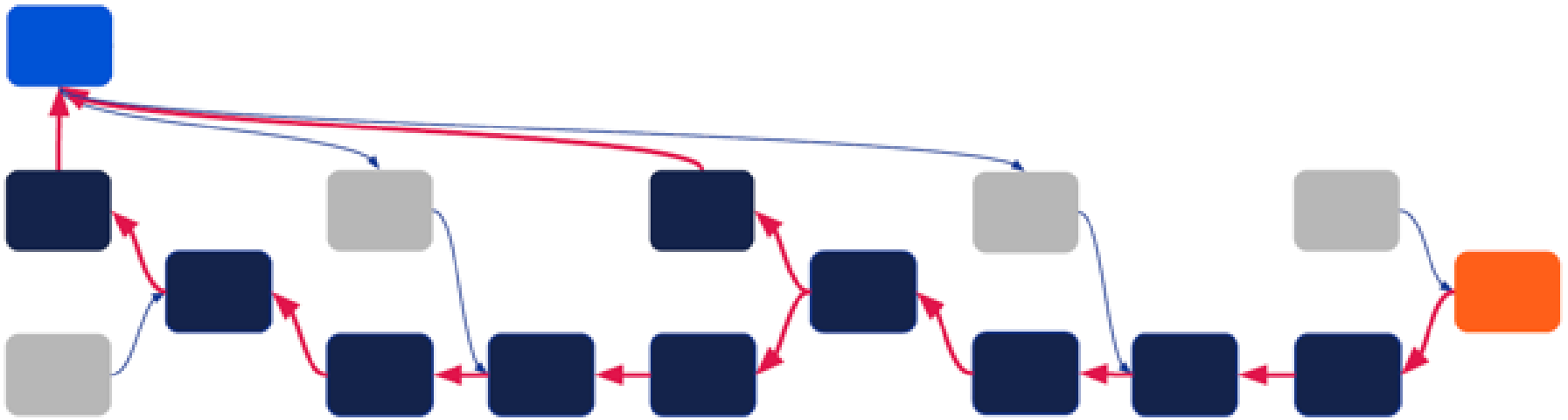
Grafo computacional



$$y = f(g(x)) \quad \frac{\partial y}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

$$y = f(\mathbf{g}(\mathbf{x})) \quad \frac{\partial y}{\partial \mathbf{x}} = \sum_{i=1}^{m} \frac{\partial f}{\partial \mathbf{g}^{(i)}} \frac{\partial \mathbf{g}^{(i)}}{\partial \mathbf{x}}$$

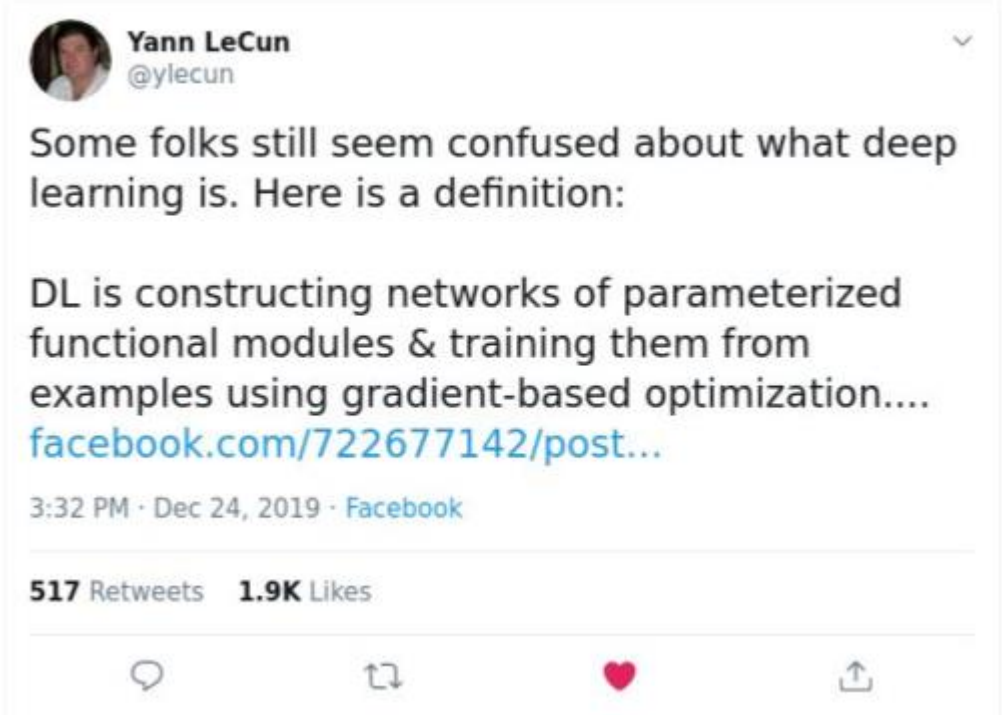
Grafo computacional

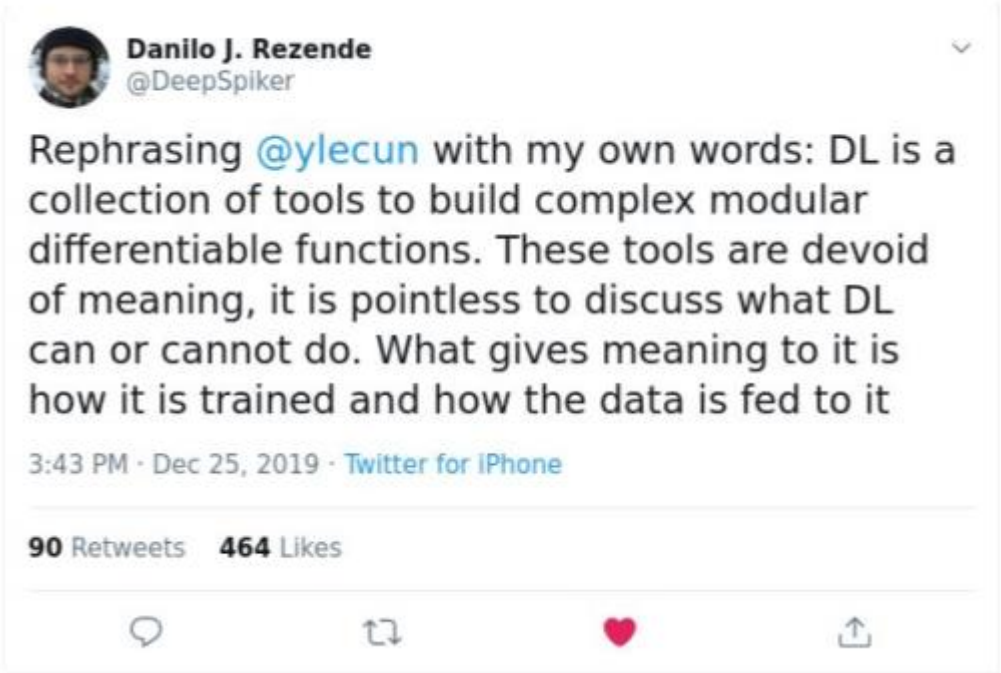


$$y = f(g(x)) \quad \frac{\partial y}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

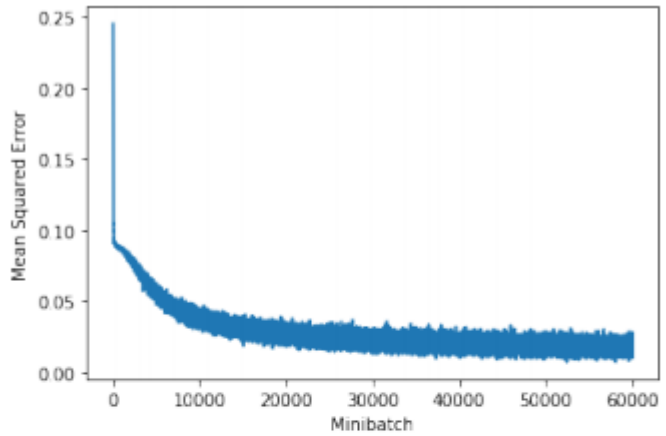
$$y = f(\mathbf{g}(\mathbf{x})) \quad \frac{\partial y}{\partial \mathbf{x}} = \sum_{i=1}^{m} \frac{\partial f}{\partial \mathbf{g}^{(i)}} \frac{\partial \mathbf{g}^{(i)}}{\partial \mathbf{x}}$$

Definición de DeepLearning





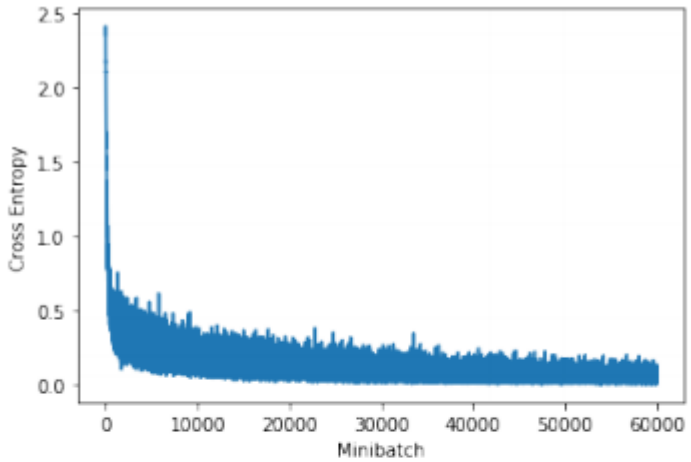
Imprementación de un perceptrón multicapa en PyTorch

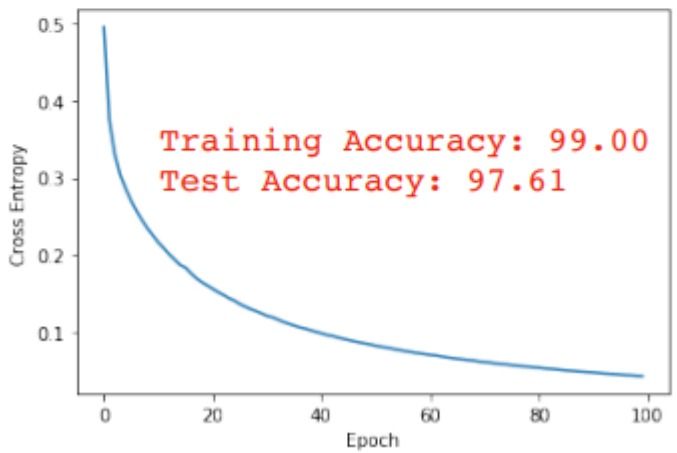


Perceptrón multicapa con función de activacion sigmoide y pérdida MSE

https://github.com/rasbt/stat453-deep-learning-ss21/blob/main/L09/code/mlp-pytorch_sigmoid-mse.ipynb

Imprementación de un perceptrón multicapa en PyTorch





Perceptrón multicapa con función de activacion *softmax* y pérdida cross-entropía

Neuronas muertas

- ReLU es probablemnte la función de activación más popular(fácil de calcular, rápida, buenos resultados)
- Pero especialmente las neuronas de ReLU pueden "morir" durante el entrenamiento
- Puede pasar si, por ejmeplo, la entrada es tan grande/pequeña que la entrada neta es tan pequeña que las ReLUs nunca se recuperan (gradiente 0 en $x < 0$)
- No es necesariamente malo, puede ser considerado como una forma de regularización
- Comparada con sigmoide/Tanh, ReLU sufre menos del problema del gradiente que se desvanece, pero puede ser "explotado" más facilmente

Arquitecturas anchas vs profundas (Ancho vs Profundidad)

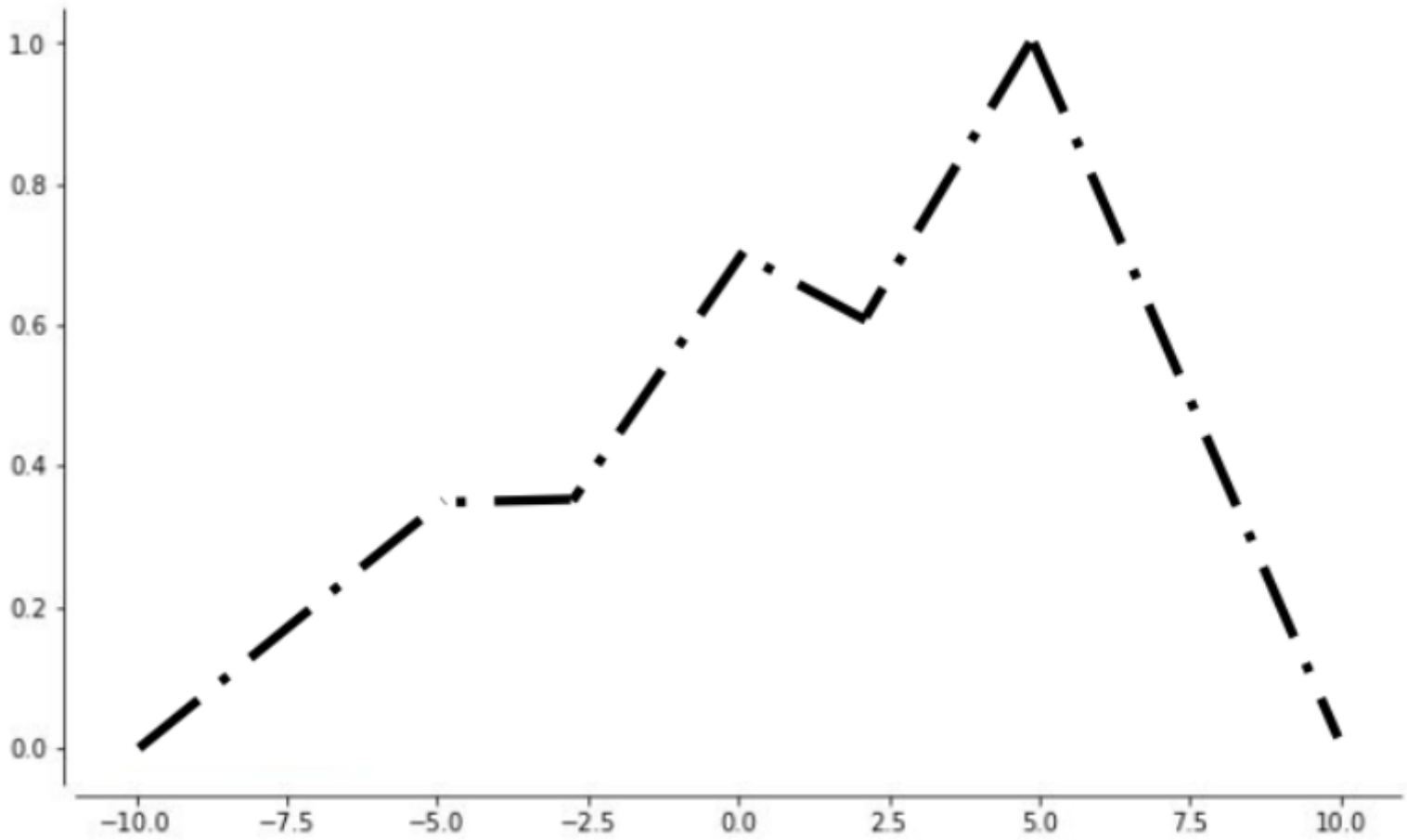
MLP's con una unidad oculta con muchas neuronas son funciones universales de aproximación [1-3] de por si, ¿Por qué querríamos utilizar arquitecturas más profundas?

[1] Balázs Csanád Csáji (2001) Approximation with Artificial Neural Networks; Faculty of Sciences; Eötvös Loránd University, Hungary [2] Cybenko, G. (1989) "Approximations by superpositions of sigmoidal functions" , Mathematics of Control, Signals, and Systems, 2(4), 303{314. doi:10.1007/BF02551274 [3] Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. Neural networks, 2(5), 359-366.

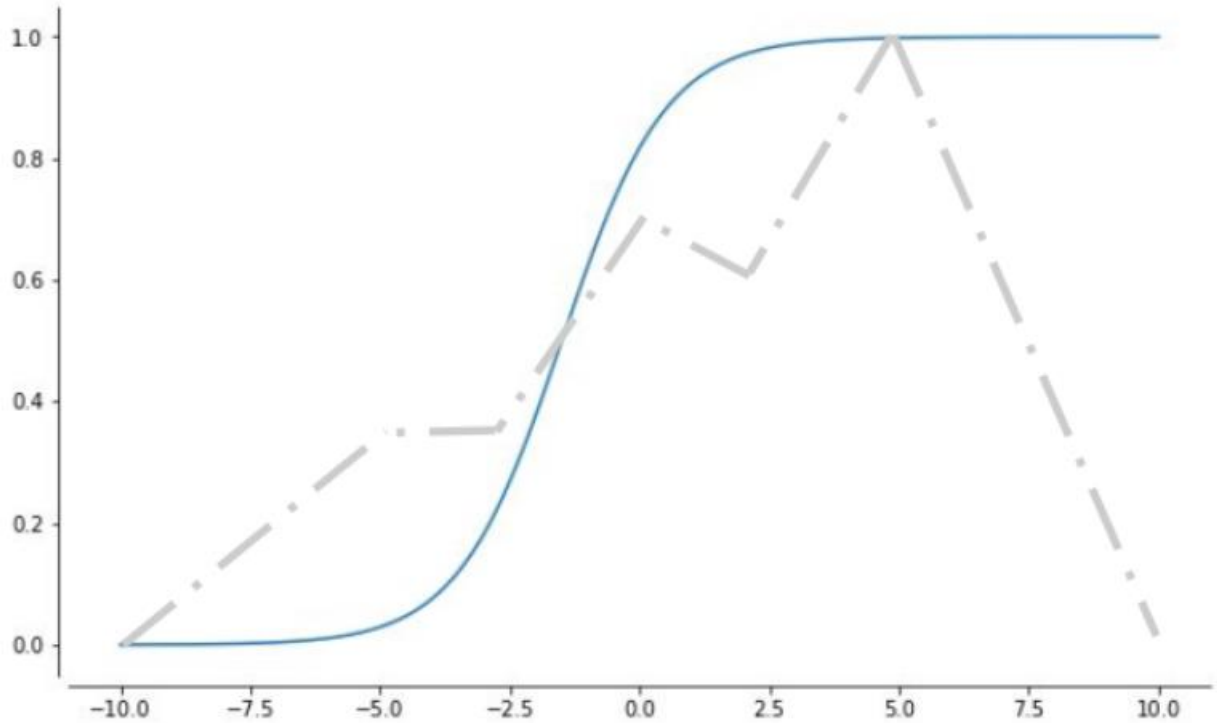
Teorema de aproximación universal

Para cualquier función continua desde un hipercubo $[0, 1]^d$ a números reales, una función de activación f que sea no constante, acotada y continua, y para cada ϵ positivo, existe una red neuronal con una capa oculta usando f que obtiene como máximo un error ϵ en el espacio funcional.

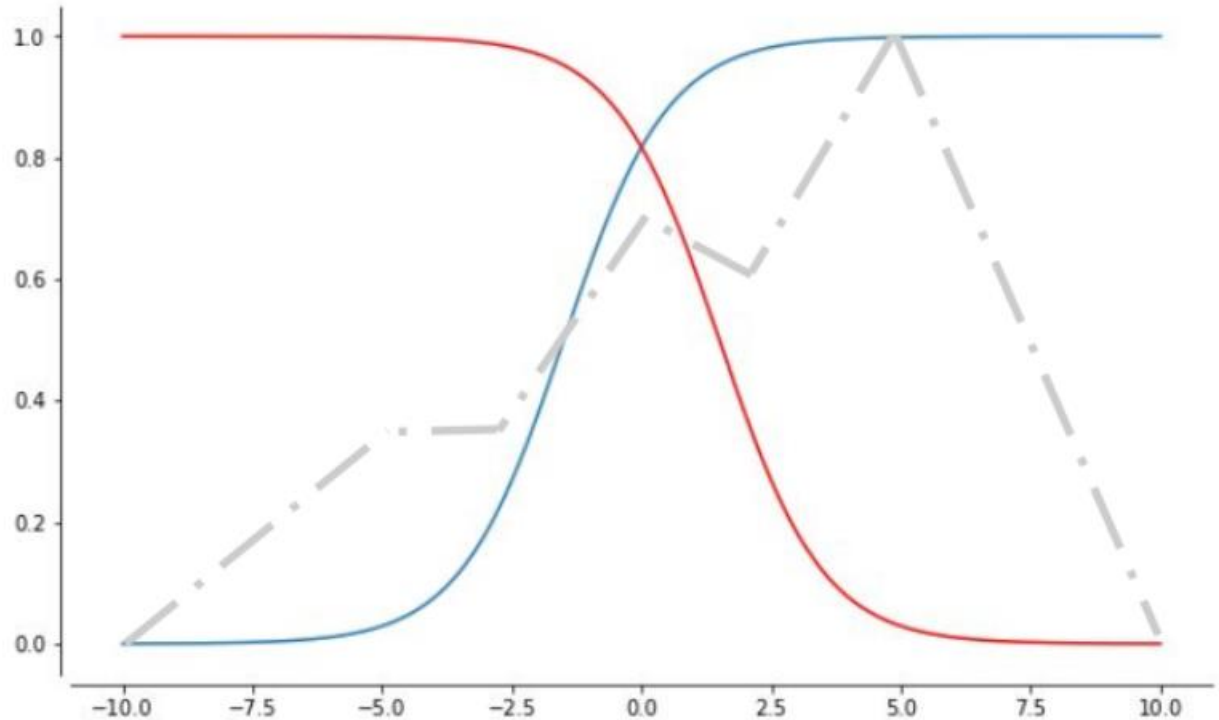
Teorema de aproximación universal - intuición



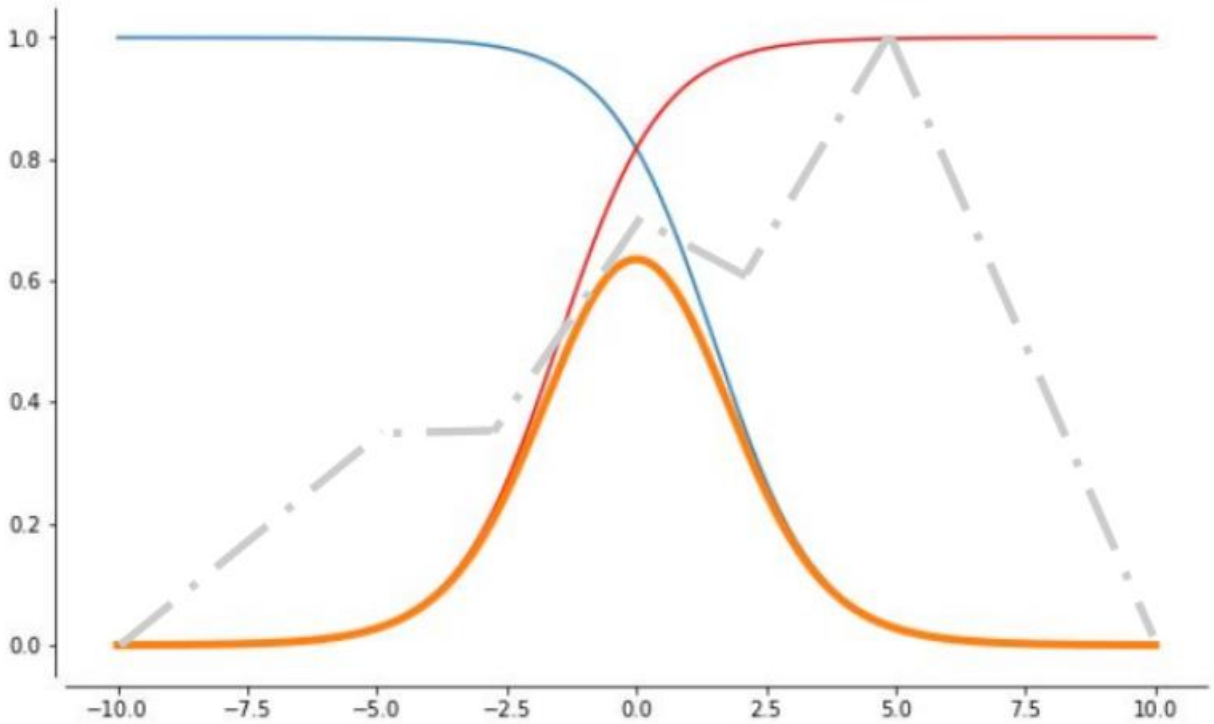
Teorema de aproximación universal - intuición



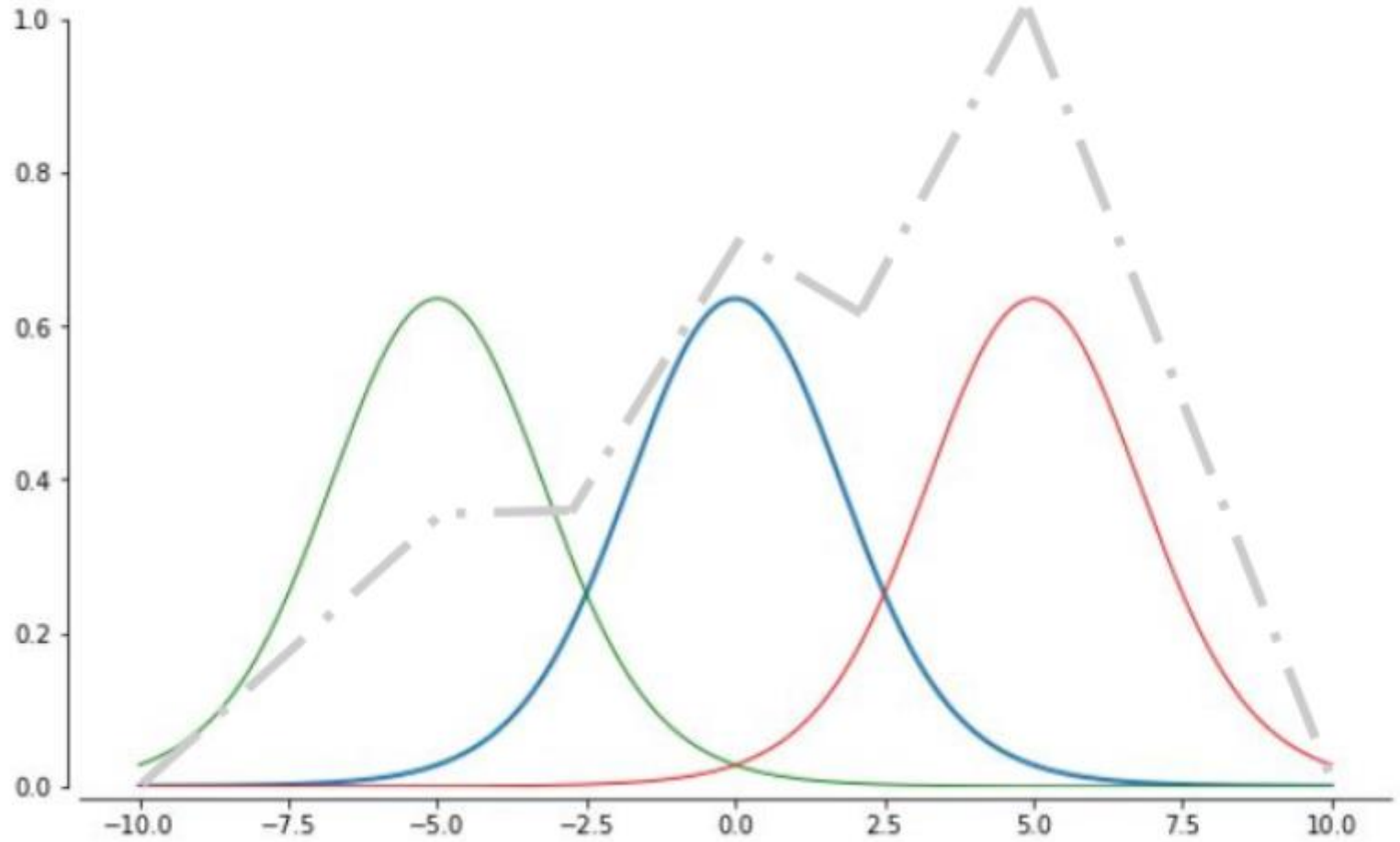
Teorema de aproximación universal - intuición



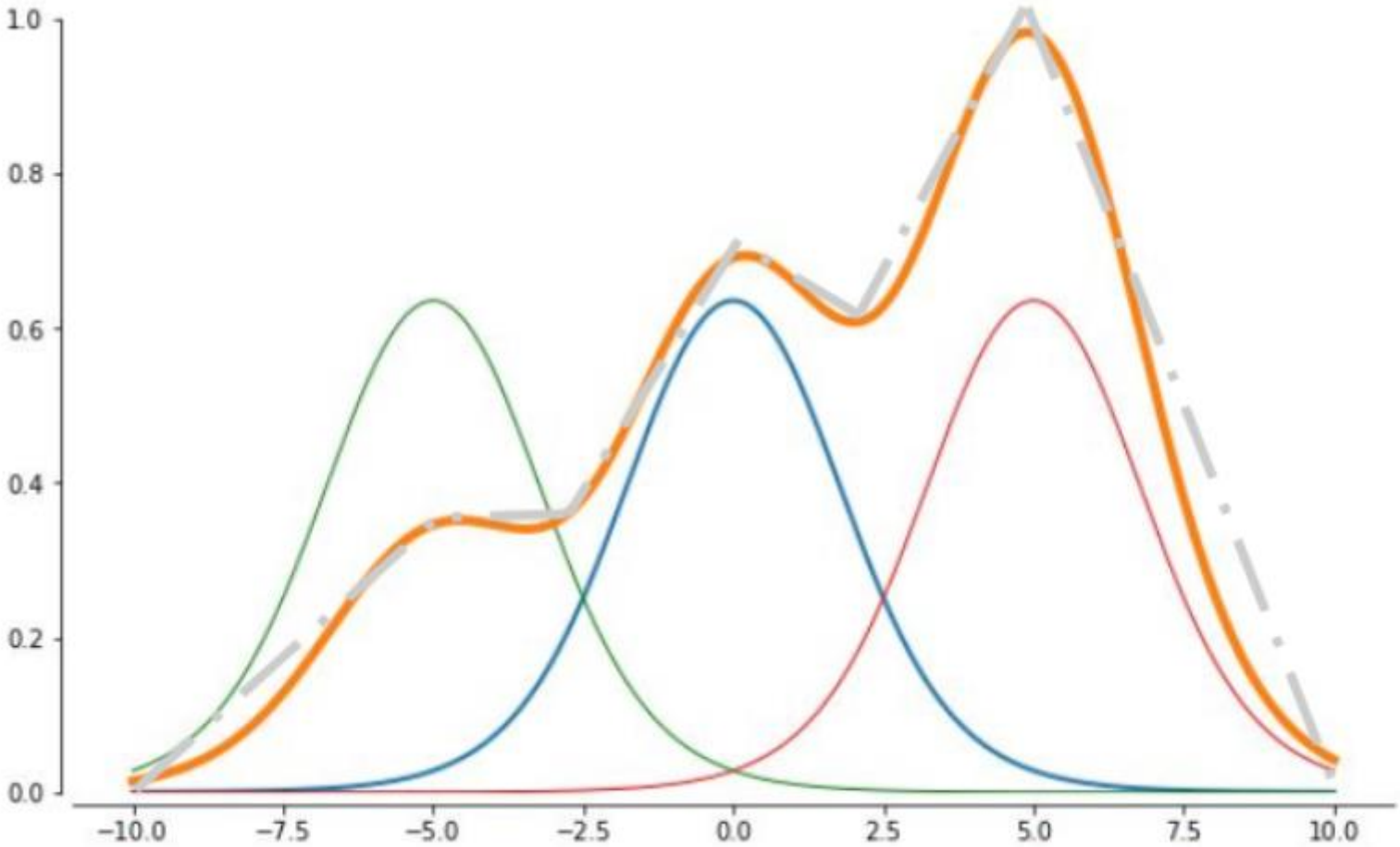
Teorema de aproximación universal - intuición



Teorema de aproximación universal - intuición



Teorema de aproximación universal - intuición



Arquitecturas anchas vs profundas (Ancho vs Profundidad)

- Pueden alcanzar la misma expresividad con más capas pero menos parámetros (combinatorias); menos parámetros $\rightarrow$ menos sobreajuste
- Además, teniendo más capas provee cierta forma de regularización: capas posteriores están restringidas por el comportamiento de capas anteriores
- Sin embargo, más capas $\rightarrow$ gradientes que explotan/desaparecen
- Después: diferentes capas para diferentes niveles de abstracción (DL es en realidad más sobre aprendizaje de funciones más que simplemente apilar multiples capas)

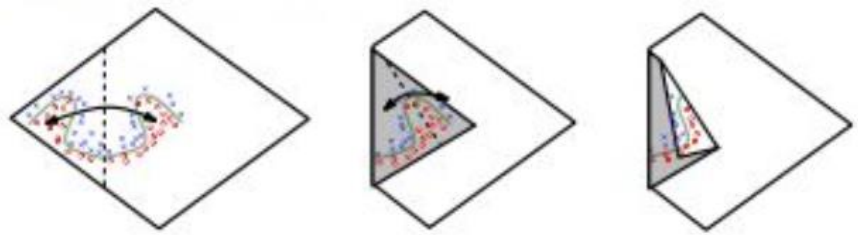


Figure 3: Space folding of 2-D space in a non-trivial way. Note how the folding can potentially identify symmetries in the boundary that it needs to learn.

Number of **linear regions** grows **exponentially** with **depth**, and **polynomially** with **width**.

[Submitted on 8 Feb 2014 (v1), last revised 7 Jun 2014 (this version, v2)]

On the Number of Linear Regions of Deep Neural Networks

Guido Montúfar, Razvan Pascanu, Kyunghyun Cho, Yoshua Bengio

We study the complexity of functions computable by deep feedforward neural networks with piecewise linear activations in terms of the symmetries and the number of linear regions that they have. Deep networks are able to sequentially map portions of each layer's input-space to the same output. In this way, deep models compute functions that react equally to complicated patterns of different inputs. The compositional structure of these functions enables them to re-use pieces of computation exponentially often in terms of the network's depth. This paper investigates the complexity of such compositional maps and contributes new theoretical results regarding the advantage of depth for neural networks with piecewise linear activation functions. In particular, our analysis is not specific to a single family of models, and as an example, we employ it for rectifier and maxout networks. We improve complexity bounds from pre-existing work and investigate the behavior of units in higher layers.

El problema con los modelos que son muy simples
y con los modelos que se acoplan ”demasiado bien”
a los datos de entrenamiento

6) t: 8 p: 4

8) t: 2 p: 7

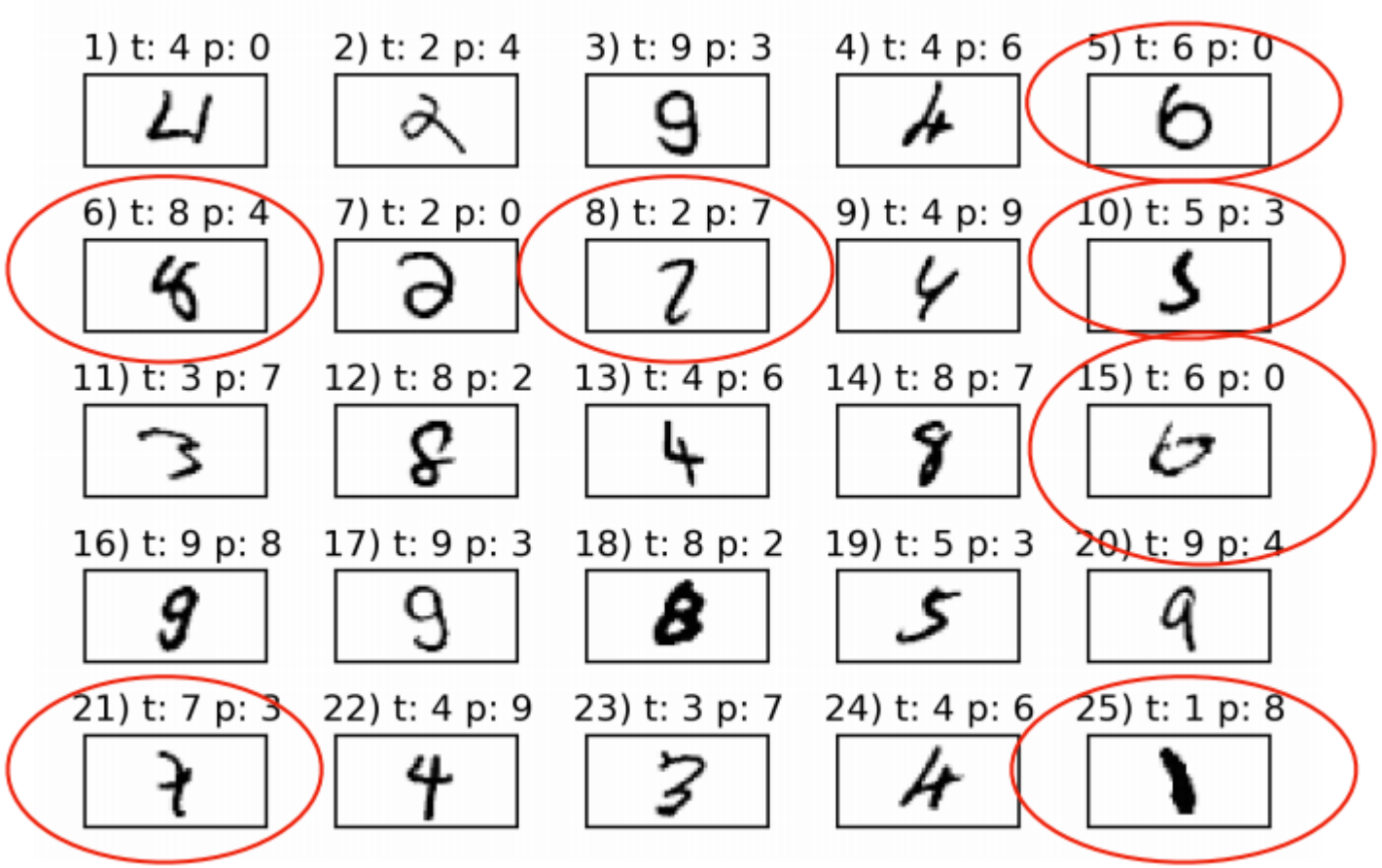
10) t: 5 p: 3

15) t: 6 p: 0

7) t: 2 p: 0

8) t: 2 p: 7

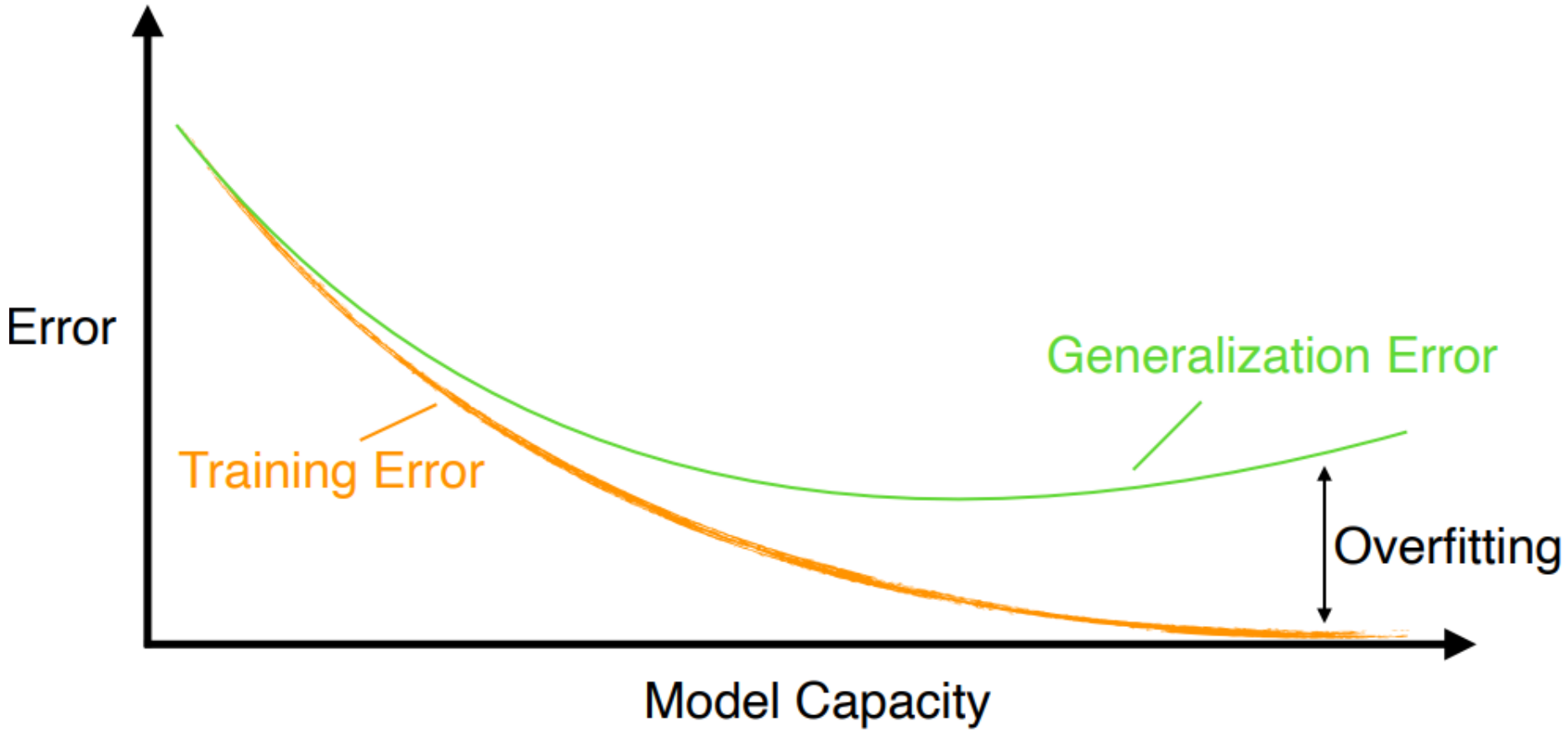
Práctica recomendada: mirar algunos casos fallidos



Casos de fallo de un $\approx 93\%$ de precisión
2-Capas (1-Capa oculta) MLP en MNIST (donde t=clase objetivo(target) y p=clase predicho(predicted))

Overfitting / Underfitting

Usualmente se usa el error del conjunto de prueba como estimador del error de generalización



Descomposición por sesgo-varianza

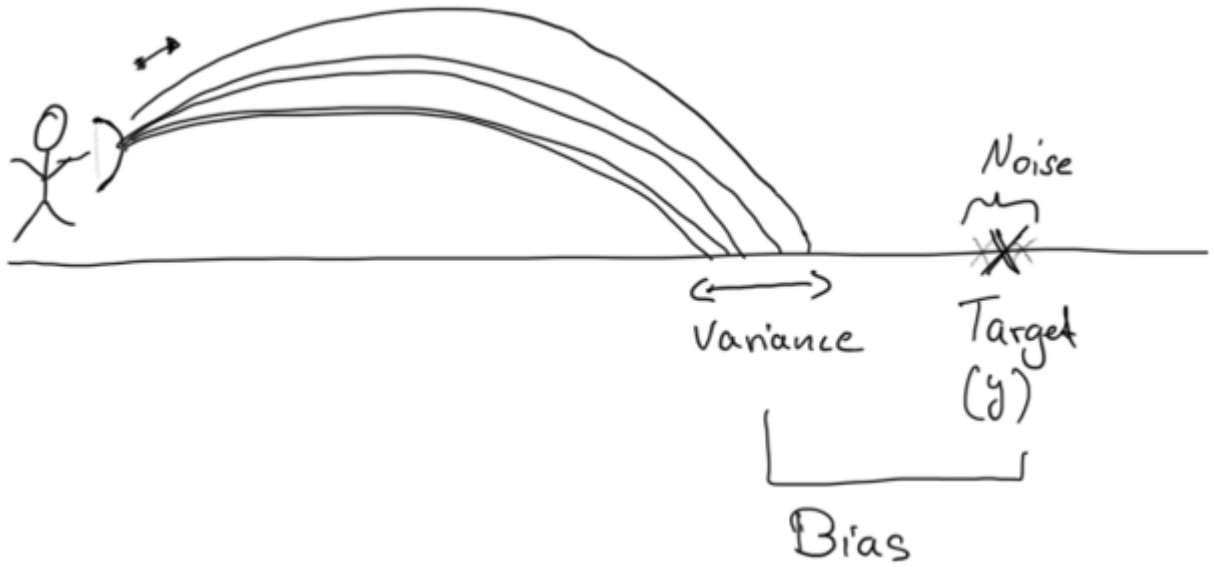
Definición general:

$$\text{Bias}_\theta[\hat{\theta}] = E[\hat{\theta}] - \theta$$

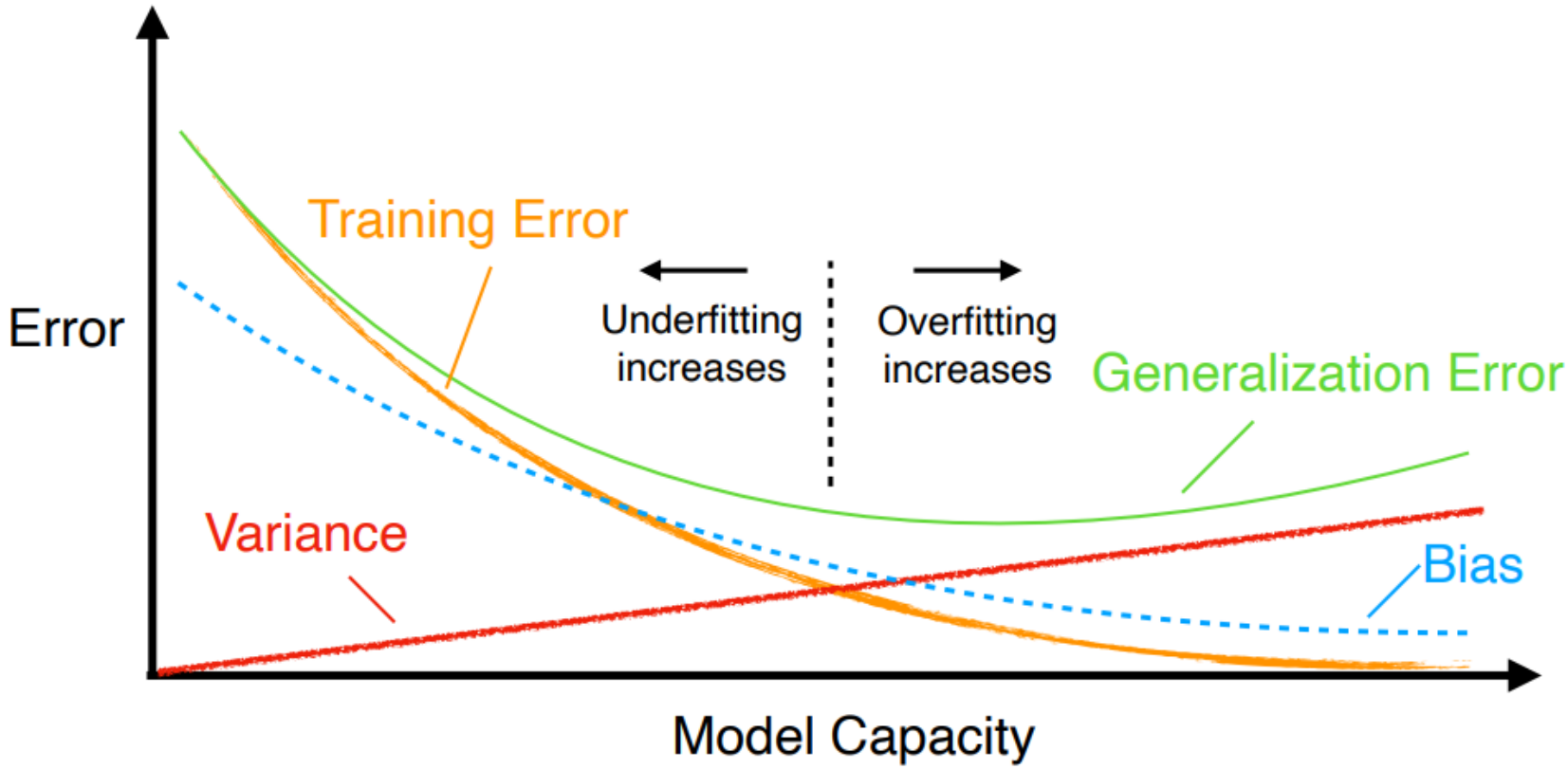
$$\text{Var}_\theta[\hat{\theta}] = E[\hat{\theta}^2] - (E[\hat{\theta}])^2$$

$$\text{Var}_\theta[\hat{\theta}] = E\left[(E[\hat{\theta}] - \hat{\theta})^2\right]$$

Intuición:



Sesgo y Varianza vs Overfitting y Underfitting



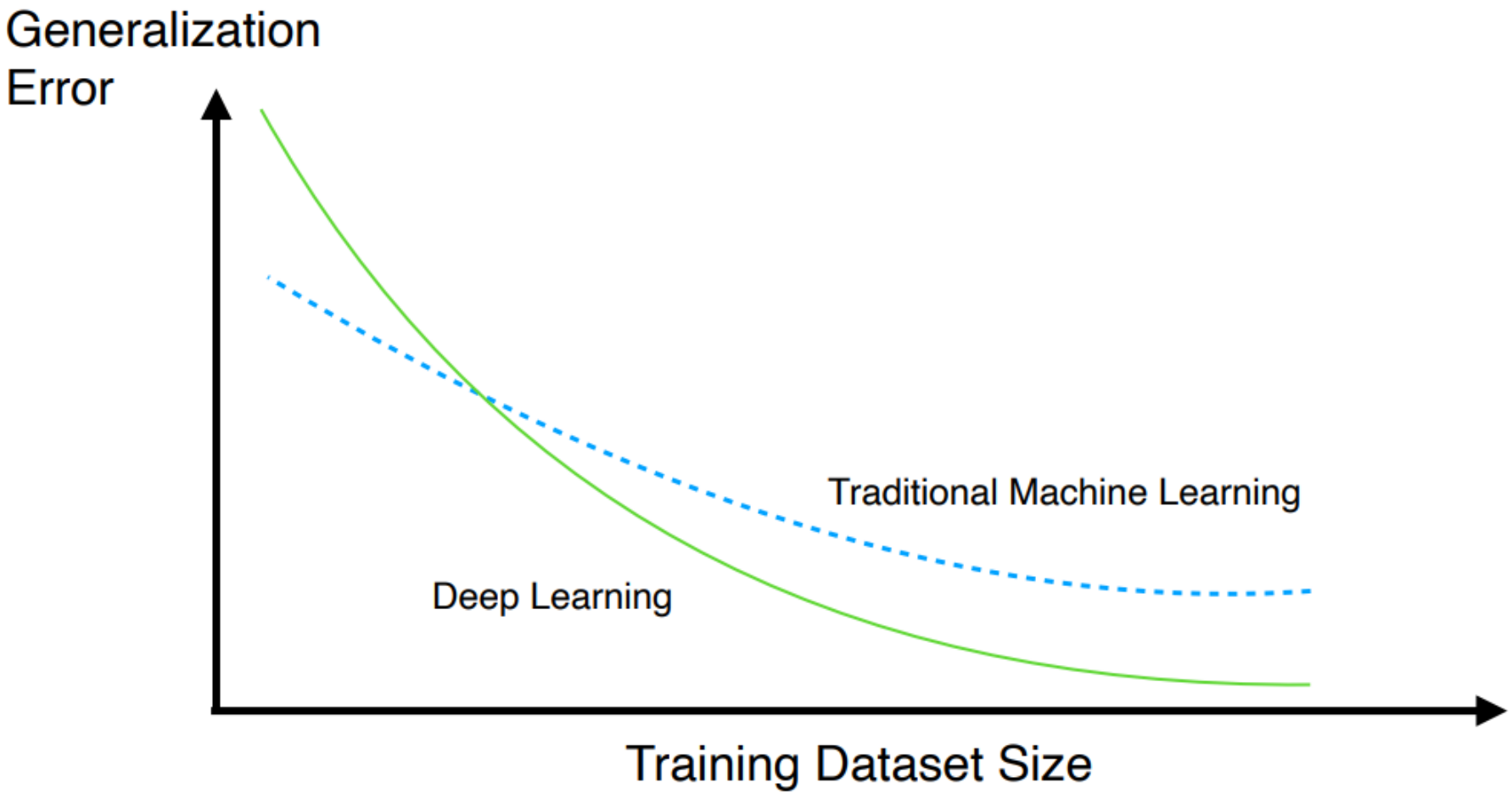
Capacidad: concepto abstracto que significa aproximadamente el número de parámetros del modelo multiplicado por la eficiencia con la que se utilizan los parámetros

Material de lectura adicional

Raschka, S. (2018). Model evaluation, model selection, and algorithm selection in machine learning. arXiv preprint arXiv:1811.12808

`https://arxiv.org/pdf/1811.12808.pdf`

El aprendizaje profundo funciona mejor con grandes conjuntos de datos



arXiv.org > cs > arXiv:1912.02292

Computer Science > Machine Learning

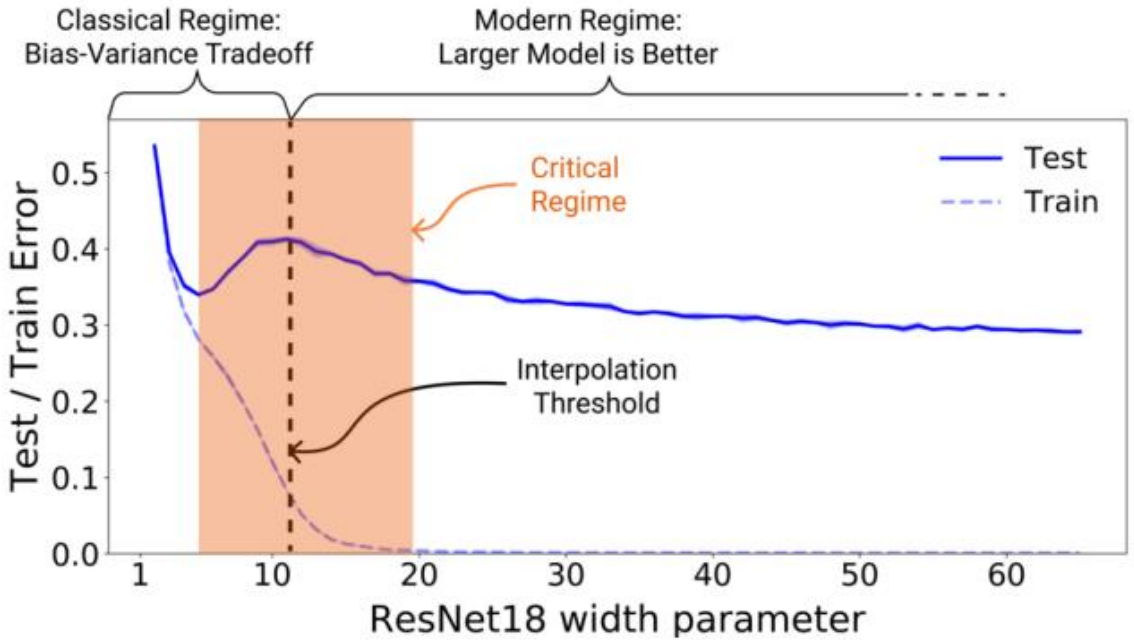
[Submitted on 4 Dec 2019]

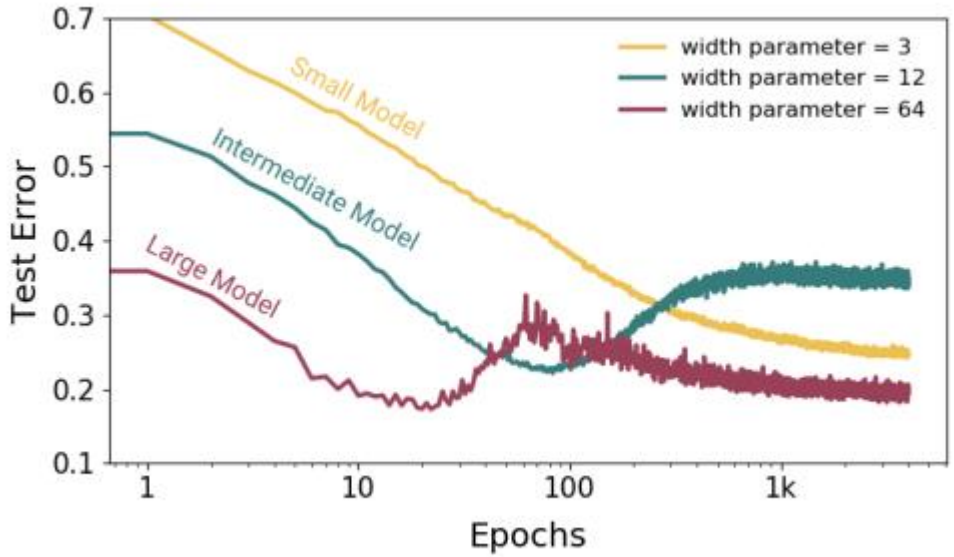
Deep Double Descent: Where Bigger Models and More Data Hurt

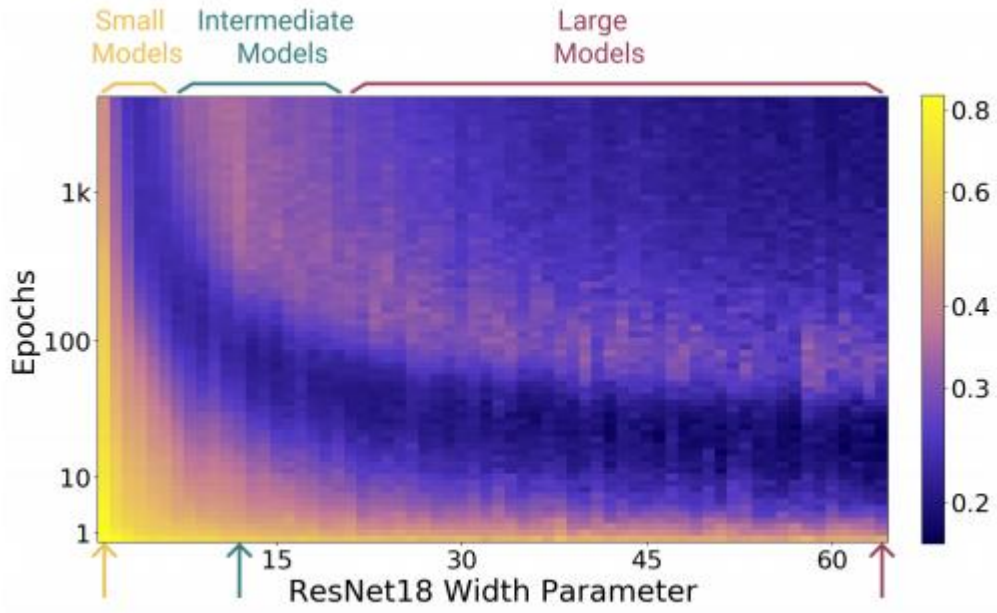
Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, Ilya Sutskever

`https://arxiv.org/abs/1912.02292`

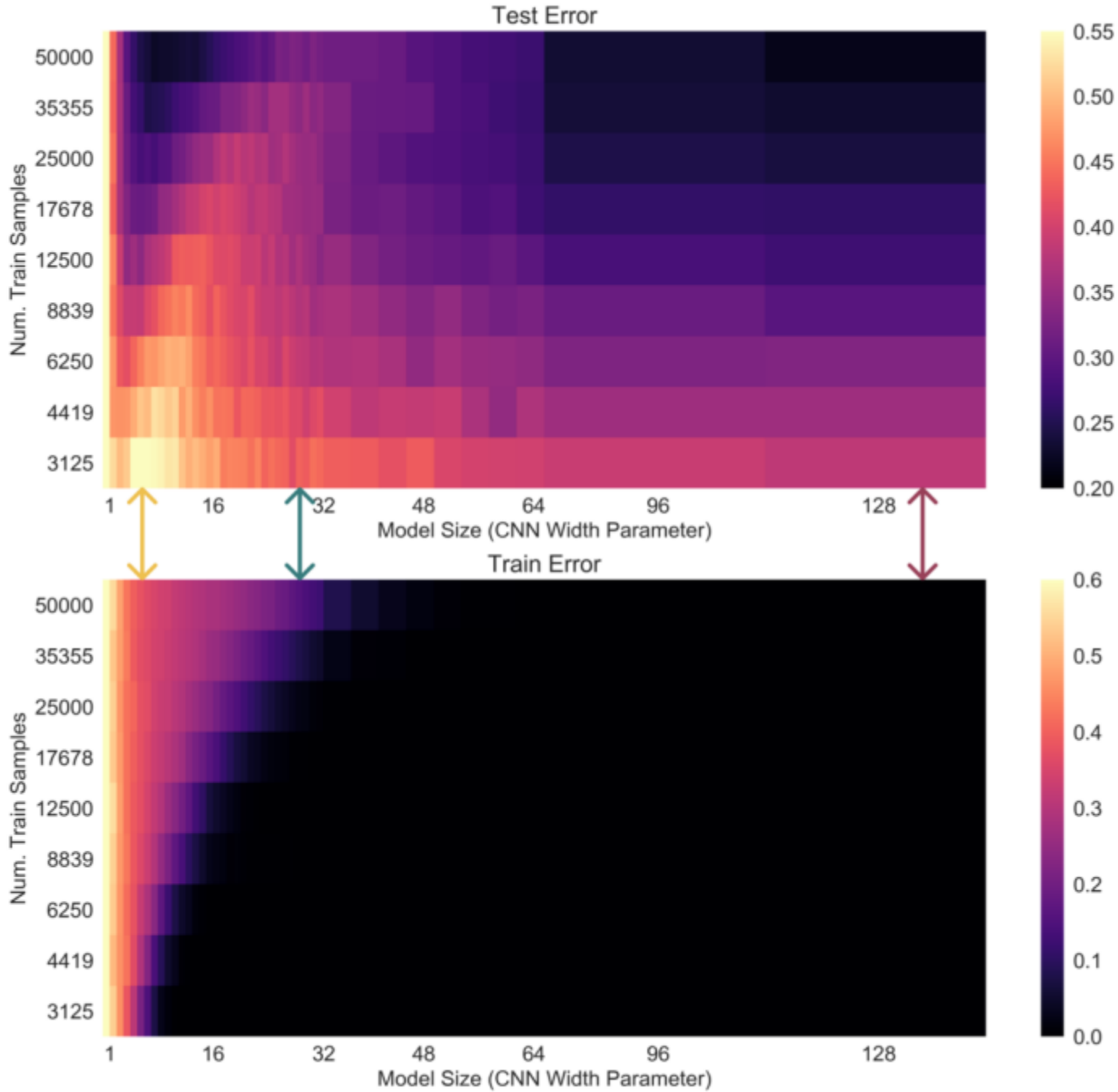
Arquitecturas: CNNS (standar y ResNet) y transformadores entrenados con pérdida cross-entropy







- En la región crítica, solo un modelo se ajusta bien a los datos y es muy sensible al ruido
- Modelos sobre parametrizados: muchos modelos se ajustan bien a los datos, SGD encuentra uno que memoriza los datos de entrenamiento pero que también se desempeña bien en los datos de prueba.


Test Error
Num. Train Samples
50000
35355
25000
17678
12500
8839
6250
4419
3125
1
16
32
48
64
96
128
Model Size (CNN Width Parameter)
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
Train Error
Num. Train Samples
50000
35355
25000
17678
12500
8839
6250
4419
3125
1
16
32
48
64
96
128
Model Size (CNN Width Parameter)
0.6
0.5
0.4
0.3
0.2
0.1
0.0

Conjuntos de entrenamiento/validación/prueba

La proporción depende en el tamaño del conjunto de datos, pero conjuntos de 80/15/5 usalmente es una buena idea

- El conjunto de entrenamiento es usado para entrenamiento, no es necesario graficar la precisión durante el entrenamiento pero puede ser útil
- La precisión del conjunto de validación provee un cálculo aproximado del desempeño de generalización (puede tener un sesgo optimista si se diseña la red para un buen redimiento en el conjunto de validación ("fuga de información")
- El conjunto de prueba solo debería ser usado una vez para obtener un sesgo estimado del desempeño de generalización

```
Epoch: 001/100 | Batch 000/156 | Cost: 1136.9125
Epoch: 001/100 | Batch 120/156 | Cost: 0.6327
Epoch: 001/100 Train Acc.: 63.35% | Validation Acc.: 62.12%
Time elapsed: 3.09 min
Epoch: 002/100 | Batch 000/156 | Cost: 0.6675
Epoch: 002/100 | Batch 120/156 | Cost: 0.6640
Epoch: 002/100 Train Acc.: 66.05% | Validation Acc.: 66.32%
Time elapsed: 6.15 min
Epoch: 003/100 | Batch 000/156 | Cost: 0.6137
Epoch: 003/100 | Batch 120/156 | Cost: 0.6311
Epoch: 003/100 Train Acc.: 65.82% | Validation Acc.: 63.76%
Time elapsed: 9.21 min
Epoch: 004/100 | Batch 000/156 | Cost: 0.5993
Epoch: 004/100 | Batch 120/156 | Cost: 0.5832
Epoch: 004/100 Train Acc.: 66.75% | Validation Acc.: 64.52%
Time elapsed: 12.27 min
Epoch: 005/100 | Batch 000/156 | Cost: 0.5918
Epoch: 005/100 | Batch 120/156 | Cost: 0.5747
Epoch: 005/100 Train Acc.: 68.29% | Validation Acc.: 67.00%
Time elapsed: 15.33 min
...
```

```
model.eval()
with torch.set_grad_enabled(False): # save memory during inference
    test_acc, test_loss = compute_accuracy_and_loss(model, test_loader, DEVICE)
    print(f'Test accuracy: {test_acc:.2f}%')
```

```
Test accuracy: 88.28%
```

Parámetros vs Hiperparámetros

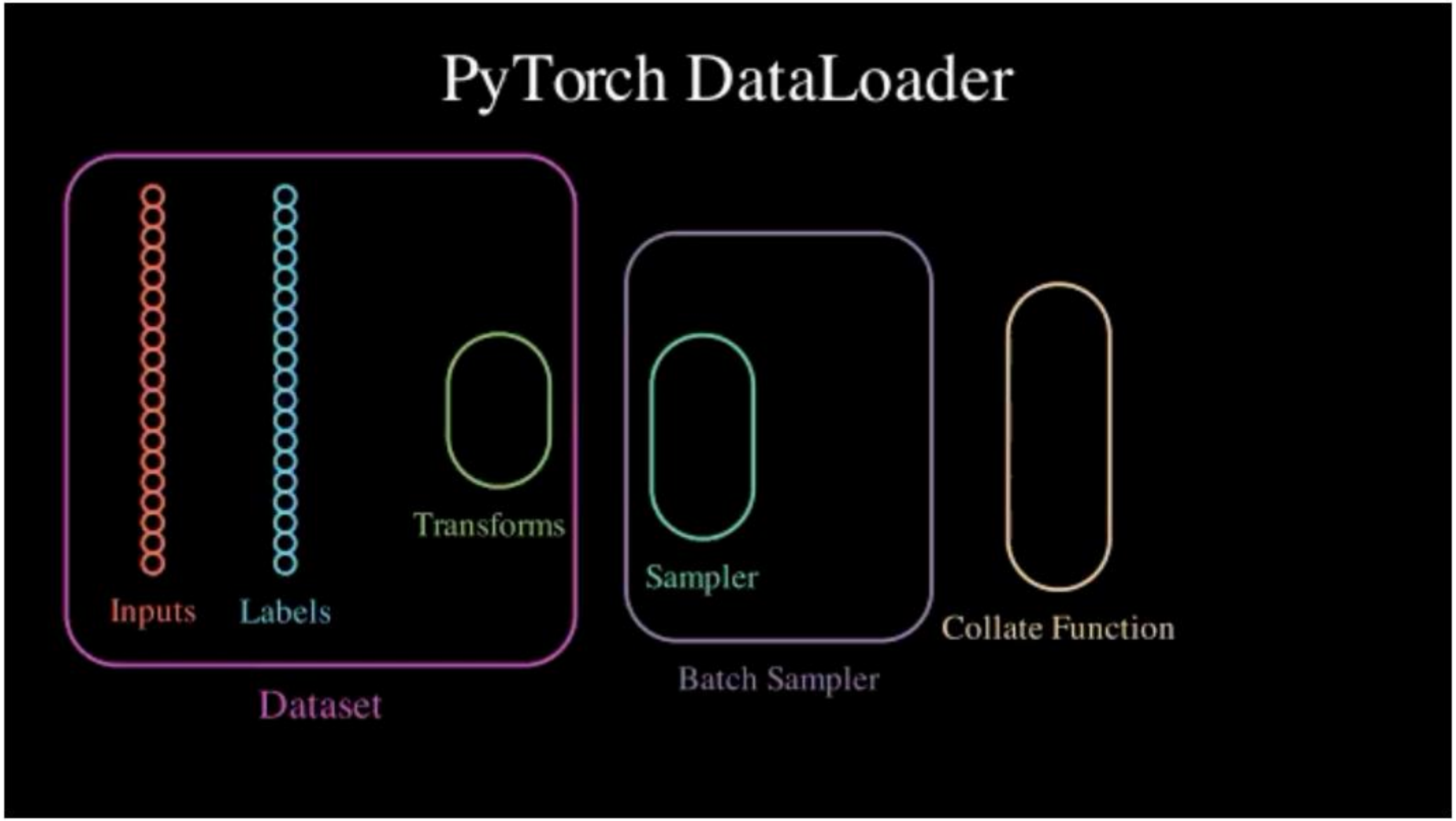
Parámetros

- *weights (weight parameters)*
- *biases (bias units)*

Hiperparámetros

- *minibatch size*
- *data normalization schemes*
- *number of epochs*
- *number of hidden layers*
- *number of hidden units*
- *learning rates*
- *(random seed, **why?**)*
- *loss function*
- *various weights (weighting terms)*
- *activation function types*
- *regularization schemes*
- *weight initialization schemes*
- *optimization algorithm type*

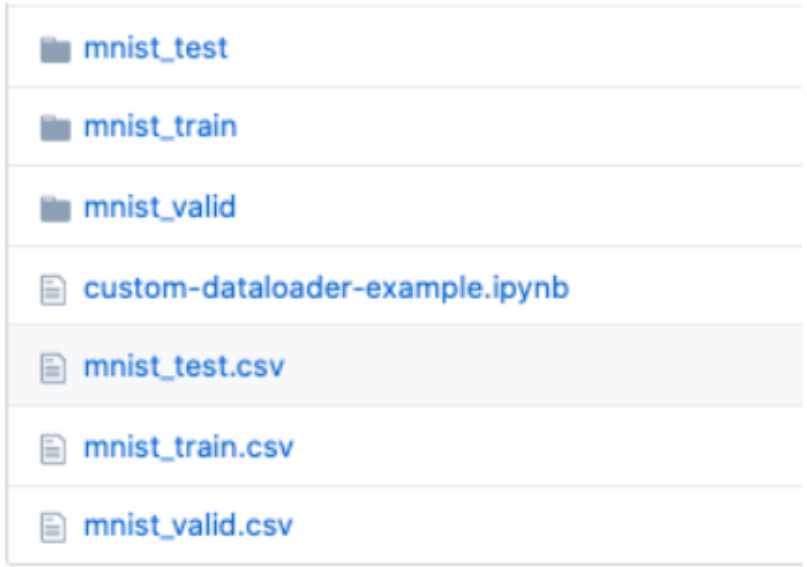
(En su mayoría sin explicación científica, principalmente ingeniería; se necesita probar muchas cosas → *graduate student descent*)



https://twitter.com/_ScottCondron/status/1363494433715552259?s=

Dataloaders personalizados

El ejemplo muestra como se pueden crear cargadores de datos personalizados para iterar de manera eficiente sobre una colección de imágenes propia (pretenda que las imagenes de MNIST que existan son colecciones personalizadas de imágenes)



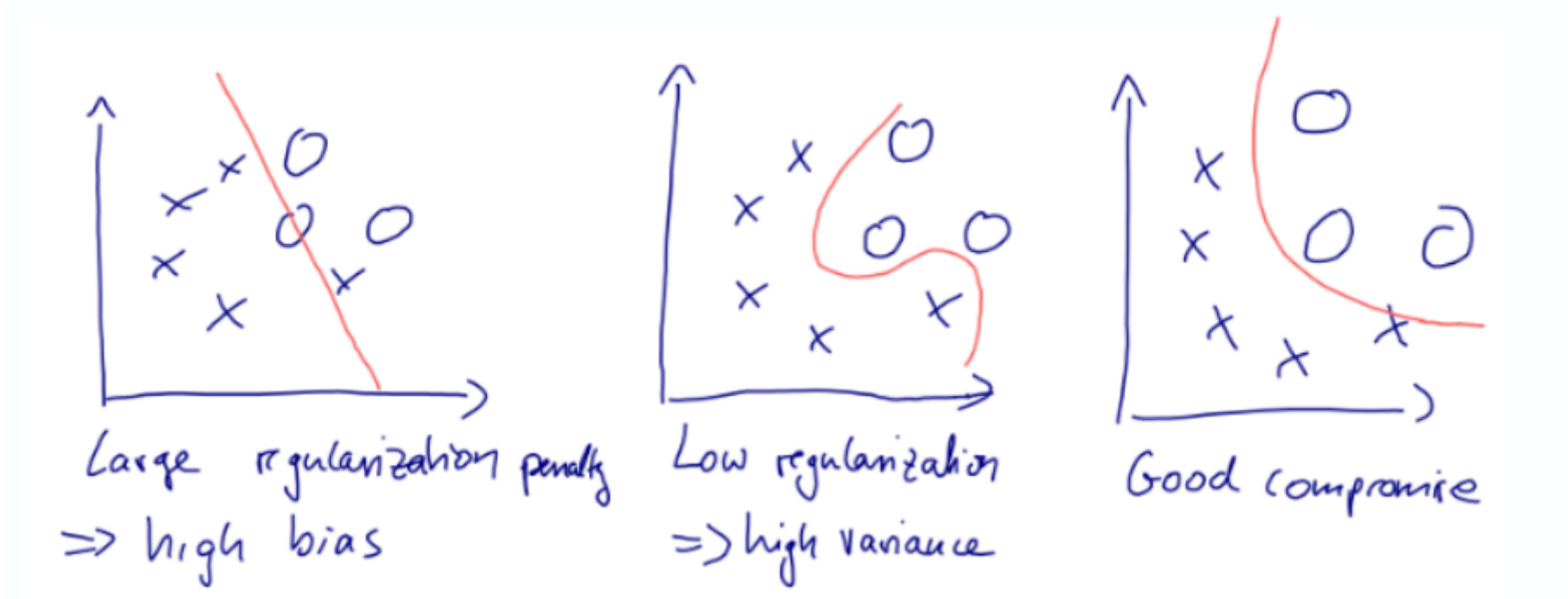

```python
import torch
from PIL import Image
from torch.utils.data import Dataset
import os



class MyDataset(Dataset):

    def __init__(self, csv_path, img_dir, transform=None):

        df = pd.read_csv(csv_path)
        self.img_dir = img_dir
        self.img_names = df['File Name']
        self.y = df['Class Label']
        self.transform = transform

    def __getitem__(self, index):
        img = Image.open(os.path.join(self.img_dir,
                                      self.img_names[index]))

        if self.transform is not None:
            img = self.transform(img)

        label = self.y[index]
        return img, label

    def __len__(self):
        return self.y.shape[0]
```

Buenas noticias: ya podemos resolver problemas no lineales

Malas noticias: nuestras redes multicapas tienen muchos parámetors, y es fácil sobreajustarse a los datos

En la próxima sesión....

Regularización de redes neuronales profundas para prevenir el sobreajuste



¿Preguntas?