

# Lecture 05

## Deep Learning

Objetivo de la regularización:  
reducir *overfitting*

Generalmente se logra reduciendo la capacidad del modelo y/o reduciendo la varianza de las predicciones

# Regularización

En el contexto del aprendizaje profundo, la regularización puede ser entendida como el proceso de agregar información o cambiar la función objetivo para prevenir *overfitting*

## Técnicas de regularización comunes para redes neuronales profundas

- *Early stopping*
- Regularización  $L_1/L_2$  (penalizaciones basadas en normas)
- *Dropout*

# Regularization (mathematics)

From Wikipedia, the free encyclopedia



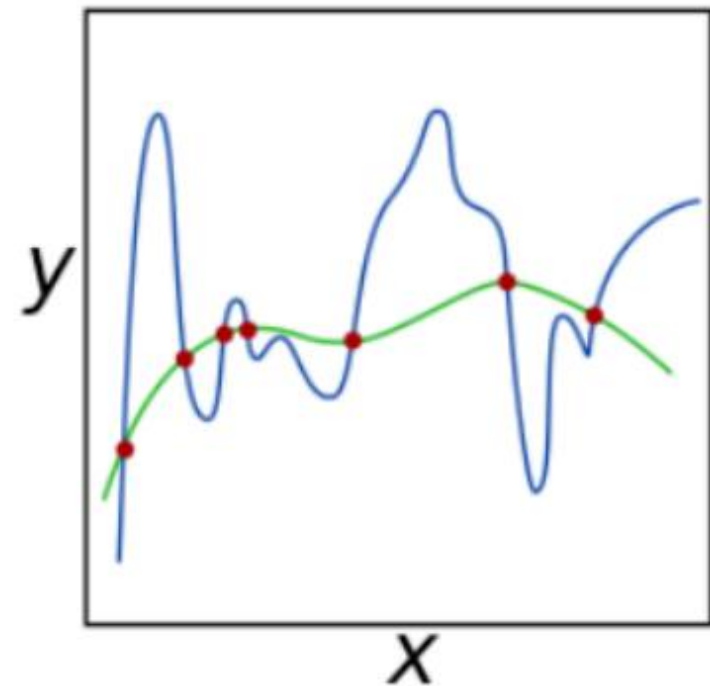
This article **only describes one highly specialized aspect of its associated subject**. Please help [improve this article](#) by adding more general information. The [talk page](#) may contain suggestions. *(November 2020)*

In [mathematics](#), [statistics](#), [finance](#),<sup>[1]</sup> [computer science](#), particularly in [machine learning](#) and [inverse problems](#), **regularization** is the process of adding information in order to solve an [ill-posed problem](#) or to prevent [overfitting](#).<sup>[2]</sup>

Regularization applies to objective functions in ill-posed optimization problems. The regularization term, or penalty, imposes a cost on the optimization function for overfitting the function or to find an optimal solution.

In [machine learning](#), regularization is any [modification](#) one makes to a learning algorithm that is intended to reduce its generalization error but not its training error<sup>[3]</sup>

**Contents** [\[hide\]](#)



The green and blue functions both incur zero loss on the given data points. A learned

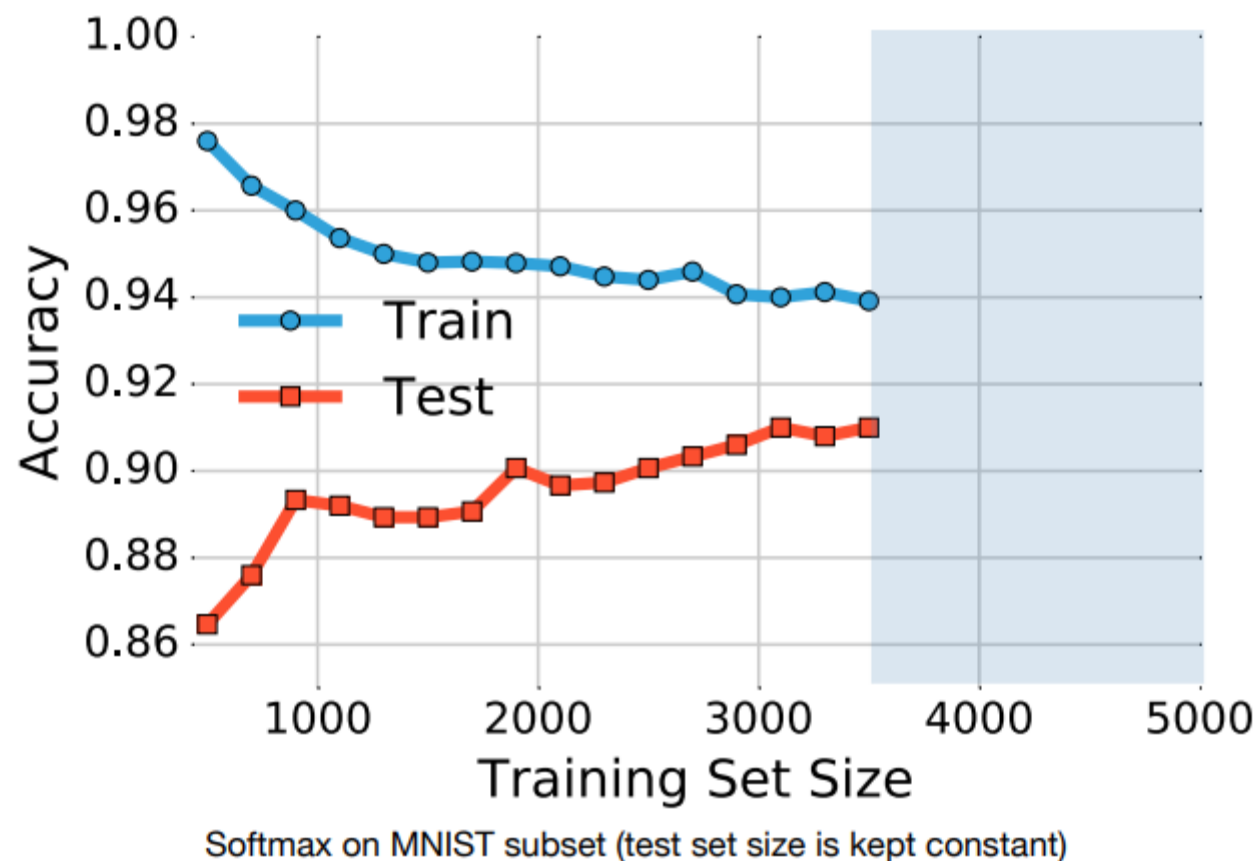
# Agenda

- Mejora del rendimiento de la generalización
- Evitar *overfitting* con (1) más datos y (2) *data augmentation*
- Reducir la capacidad de la red e *early stopping*
- Agregar penalizaciones basadas en normas a la función de costo:  $L_1$  y  $L_2$
- *Dropout*



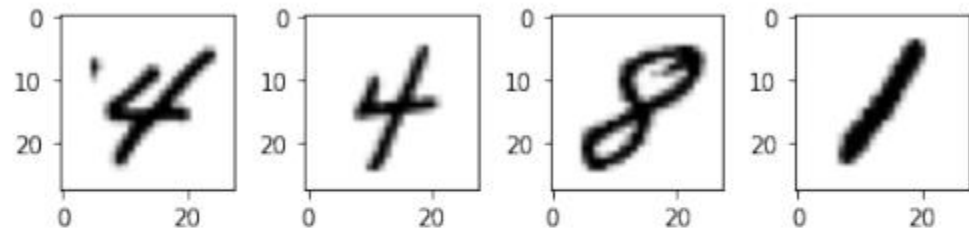
# Primer paso para mejorar el rendimiento: enfocarse en el conjunto de datos en sí

A menudo, la mejor forma de reducir el *overfitting* es obteniendo más datos

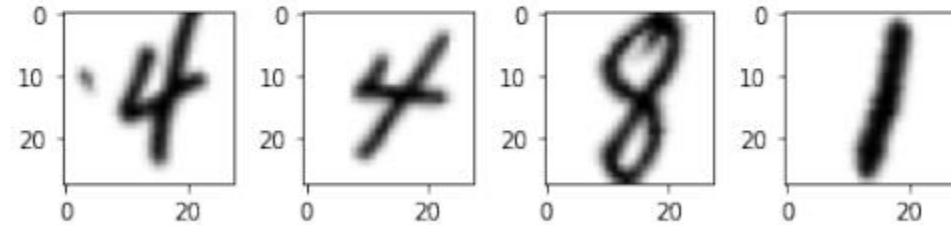


# *Data augmentation* en PyTorch a través de TorchVision

Original



Randomly Augmented







```
# Note transforms.ToTensor() scales input images
# to 0-1 range
```

```
training_transforms = torchvision.transforms.Compose([
    torchvision.transforms.Resize(size=(32, 32)),
    torchvision.transforms.RandomCrop(size=(28, 28)),
    torchvision.transforms.RandomRotation(degrees=30, interpolation=PIL.Image.BILINEAR),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=(0.5,), std=(0.5,)),
    # normalize does (x_i - mean) / std
    # if images are [0, 1], they will be [-1, 1] afterwards
])
```

```
test_transforms = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Resize(size=(32, 32)),
    torchvision.transforms.CenterCrop(size=(28, 28)),
    torchvision.transforms.Normalize(mean=(0.5, ), std=(0.5, )),
])
```

```
# for more see
# https://pytorch.org/docs/stable/torchvision/transforms.html
```

```
train_dataset = datasets.MNIST(root='data',
                                train=True,
                                transform=training_transforms,
                                download=True)
```

```
test_dataset = datasets.MNIST(root='data',
                               train=False,
                               transform=test_transforms)
```

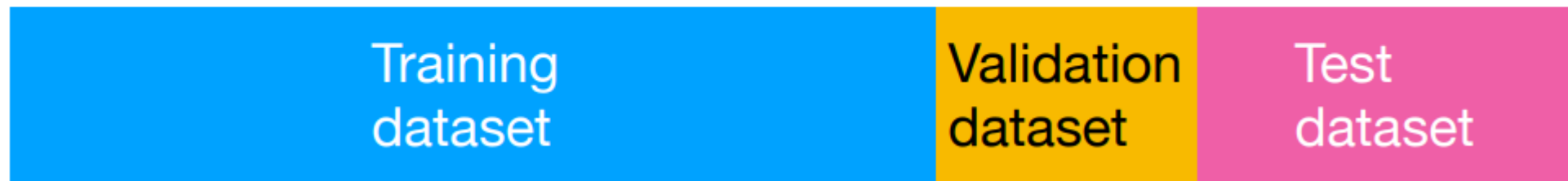
Use (0.5, 0.5, 0.5) for RGB images

# Reducir la capacidad de la red e *Early stopping* o parada anticipada

**Paso 1:** separar el conjunto de datos en 3 partes (siempre recomendado)

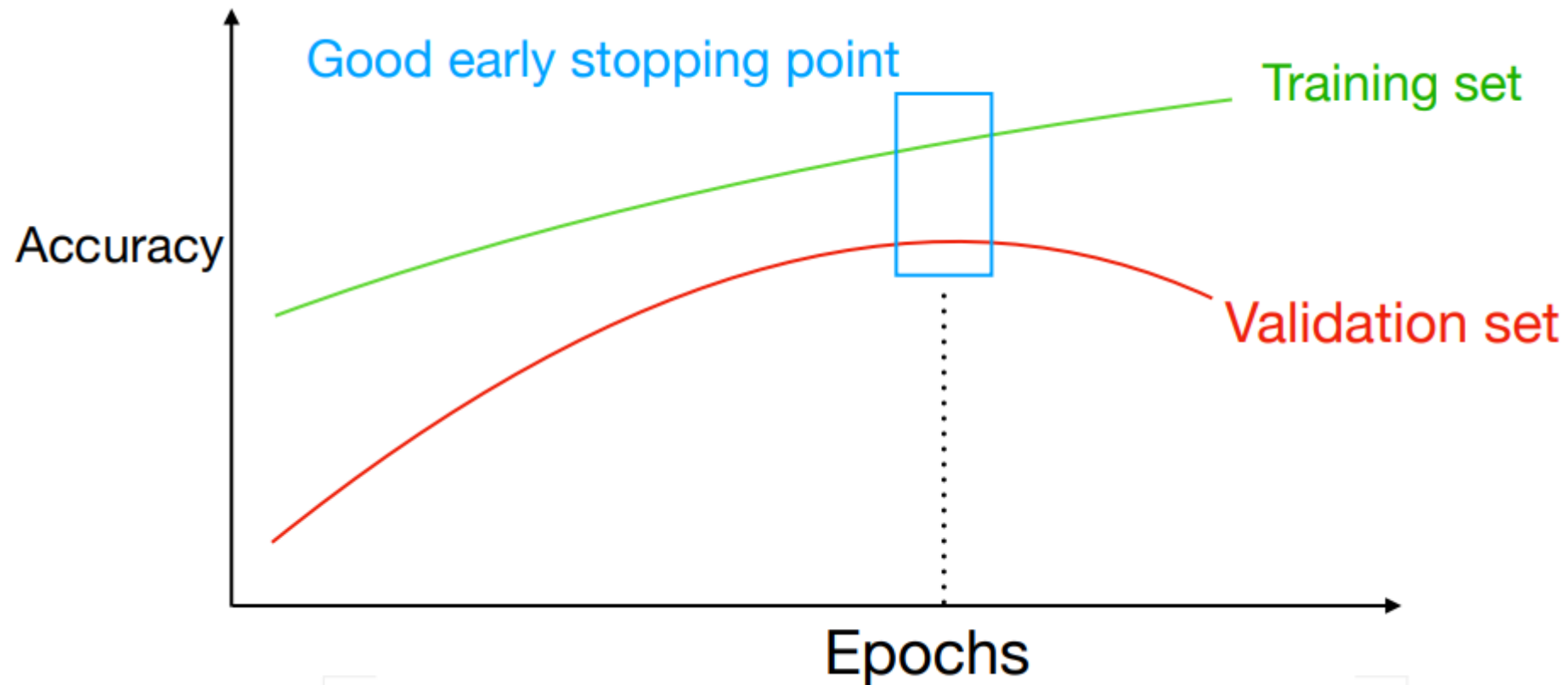
- Usar los datos de test solo una vez al final (para una estimación no sesgada del rendimiento de generalización)
- Usar la precisión de validación para el ajuste

## Dataset



**Paso 2:** parada anticipada (no es muy común actualmente)

- Reducir *overfitting* mediante la observación de la brecha de precisión de entrenamiento/validación y luego parar en el punto “correcto”



# Agregar una penalización contra la complejidad

## Regularización $L_1$ y $L_2$

- Regularización- $L_1 \rightarrow$  Regresión LASSO
- Regularización- $L_2 \rightarrow$  Regresión Ridge (Regularización de Tikhonov)

Básicamente, una “restricción a la magnitud de los pesos” o una “penalización contra la complejidad”

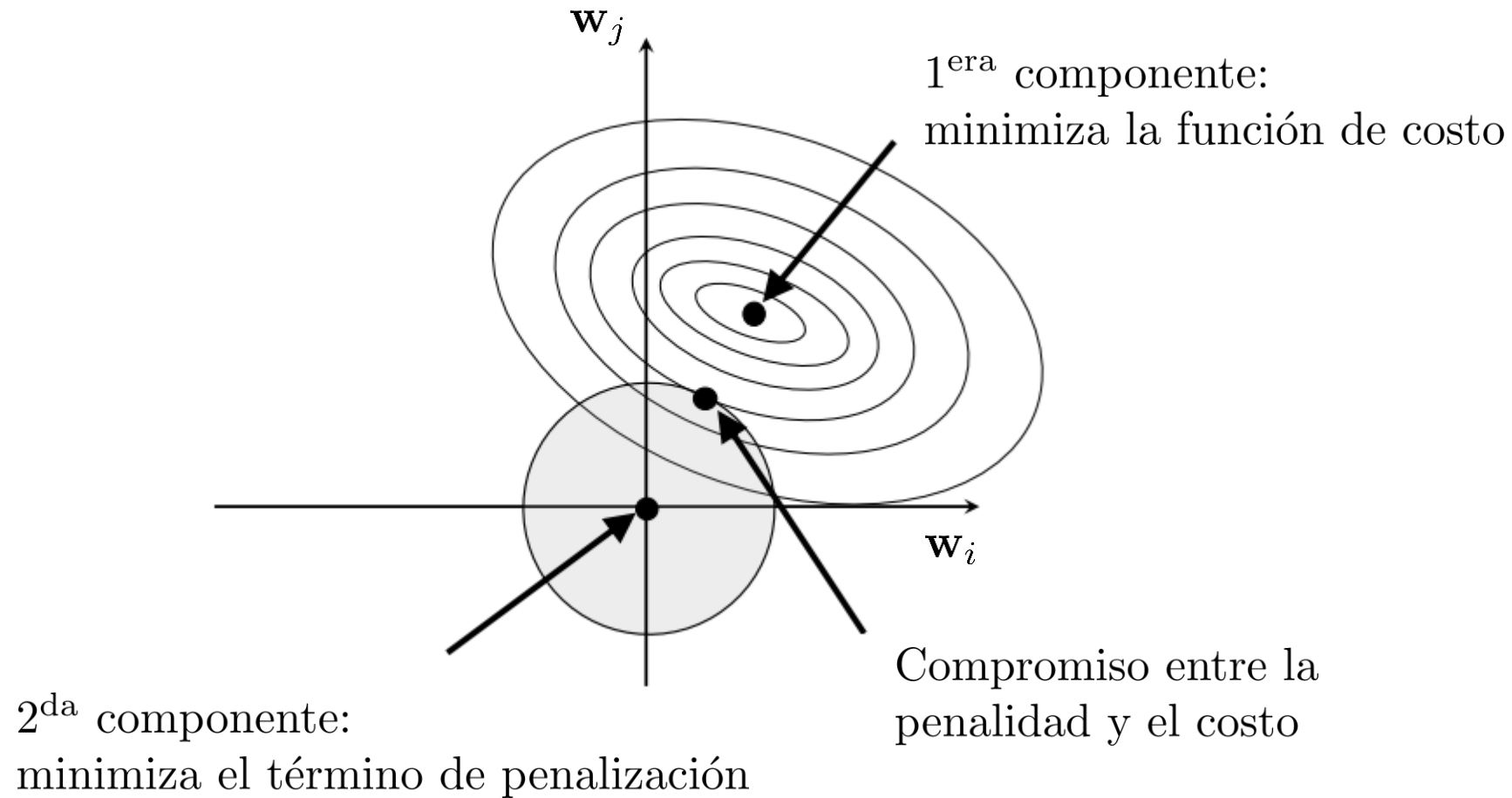
# Regularización $L_2$ para modelos lineales (e.g. regresión logística)

$$\text{Cost}_{\mathbf{w}, \mathbf{b}} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y^{[i]}, \hat{y}^{[i]})$$

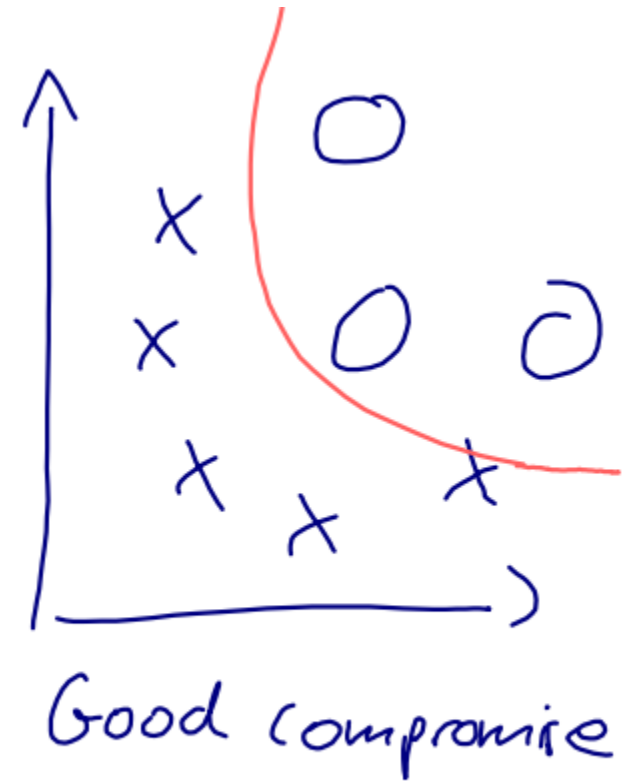
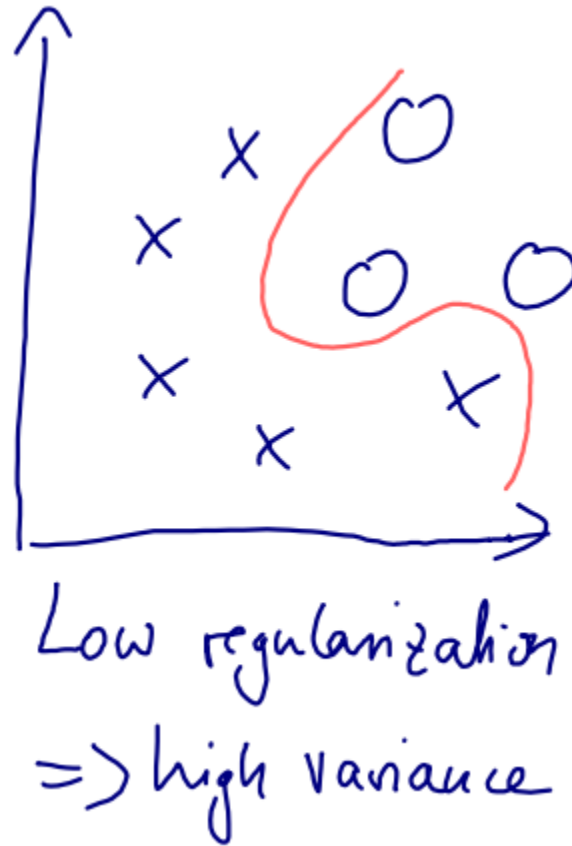
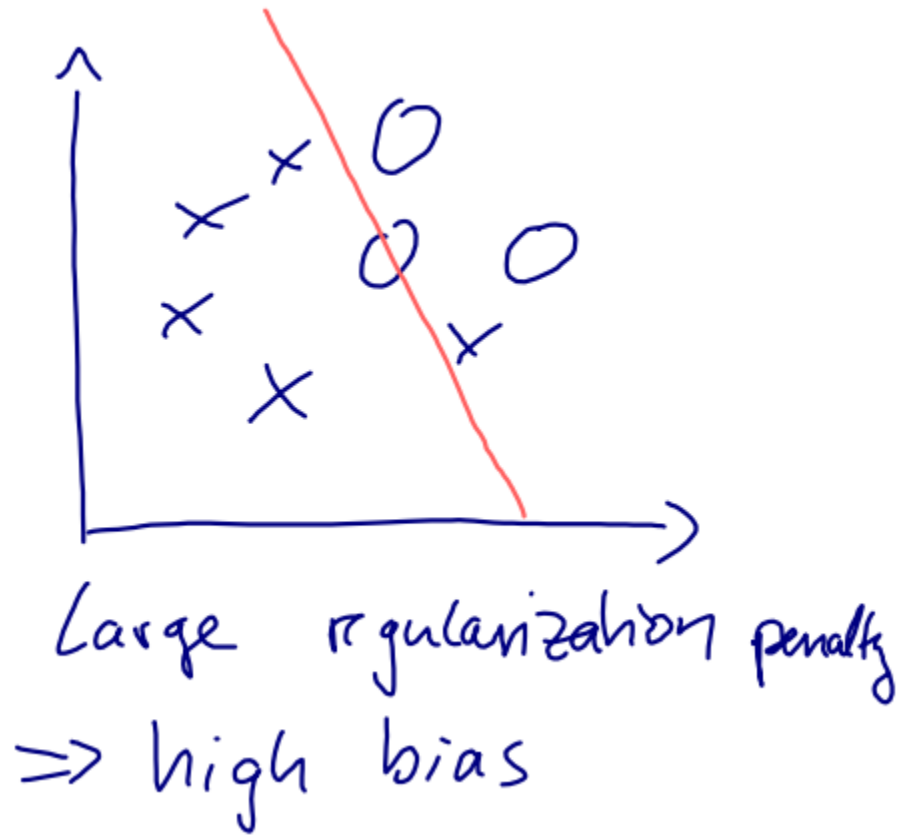
$$\text{L2-Regularized-Cost}_{\mathbf{w}, \mathbf{b}} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n} \sum_j w_j^2$$

Donde:  $\sum_j w_j^2 = \|\mathbf{w}\|_2^2$  y  $\lambda$  es un hiperparámetro.

# Interpretación geométrica de la Regularización $L_2$



## Efectos de las sanciones normativas en la frontera de decisión






# Regularización L<sub>2</sub> para redes neuronales

$$\text{L2-Regularized-Cost}_{\mathbf{w}, \mathbf{b}} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y^{[y]}, \hat{y}^{[i]}) + \frac{\lambda}{n} \sum_l^L \|\mathbf{w}^{(l)}\|_F^2$$

Suma sobre  
las capas



Donde:  $\|\mathbf{w}^{(l)}\|_F^2 = \sum_i \sum_j \left(w_{i,j}^{(l)}\right)^2$  es la norma de Frobenius al cuadrado.

# Regularización L<sub>2</sub> para redes neuronales

Actualización para el gradiente descendente regular:

$$w_{i,j} := w_{i,j} - \eta \frac{\partial \mathcal{L}}{\partial w_{i,j}}$$

Actualización para el gradiente descendente con **regularización L2**:

$$w_{i,j} := w_{i,j} - \eta \left( \frac{\partial \mathcal{L}}{\partial w_{i,j}} + \frac{2\lambda}{n} w_{i,j} \right)$$

# Regularización $L_2$ para redes neuronales en PyTorch

```
# regularize loss
L2 = 0.
for name, p in model.named_parameters():
    if 'weight' in name:
        L2 = L2 + (p**2).sum()

cost = cost + 2./targets.size(0) * LAMBDA * L2

optimizer.zero_grad()
cost.backward()
```

# Regularización $L_2$ para redes neuronales en PyTorch

Automáticamente,

```
#####  
## Apply L2 regularization  
optimizer = torch.optim.SGD(model.parameters(),  
                             lr=0.1,  
                             weight_decay=LAMBDA)  
  
#-----  
  
for epoch in range(num_epochs):  
  
    ### Compute outputs ###  
    out = model(X_train_tensor)  
  
    ### Compute gradients ###  
    cost = F.binary_cross_entropy(out, y_train_tensor)  
    optimizer.zero_grad()  
    cost.backward()
```

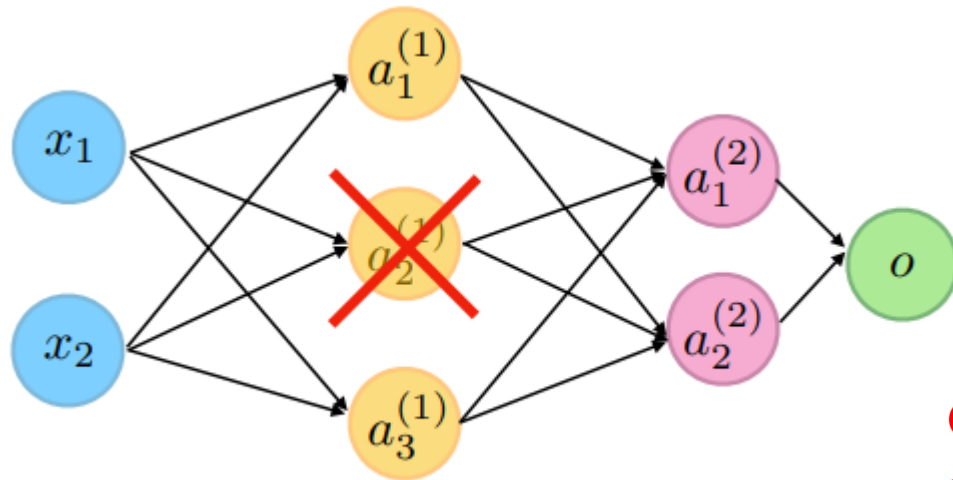
# *Dropout*

Artículos de investigaciones originales:

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, 15(1), 1929-1958.

## *Dropout* en pocas palabras: eliminación de nodos



Originalmente, la probabilidad de eliminación era de 0.5, pero actualmente es común encontrar valores de 0.2 – 0.8

# *Dropout* en pocas palabras: eliminación de nodos

¿Cómo se eliminan nodos eficientemente?

Muestreo de Bernoulli (durante el entrenamiento):

- $p :=$  probabilidad de eliminación
- $v :=$  muestra aleatoria de una distribución uniforme en el rango  $[0, 1]$
- $\forall i \in v : u_i := 0$  si  $u_i < p$  si no 1
- $a := a \odot v$  ( $p \times 100\%$  de las activaciones  $a$  será 0)

Después del entrenamiento, durante la “inferencia”, se deben escalar las activaciones a través de:  $a := a \odot (1 - p)$

¿Por qué es necesario?

¿Preguntas?