

Lectura I.2: Entrenamiento de modelos

Redes Neuronales Informadas por Física

Especialización en Inteligencia Artificial - B52024

Docentes: Benjamin A. Tourn - Carlos G. Massobrio



Outline

- Entrenamiento vs. optimización
- Inconvenientes principales y estrategias para sortearlos
- Optimizadores de primer y segundo orden
- Learning rate scheduling
- Regularización

Entrenamiento vs. optimización

Aprendizaje: reducir una función costo $J(\theta)$ con la expectativa de reducir una métrica P definida sobre un conjunto de prueba → reducción indirecta de P

Optimización: Reducir función costo $J(\theta)$ en sí misma

$J(\theta)$ se define sobre el conjunto de entrenamiento:

$$J(\theta) = \mathbb{E}_{(x,y) \sim \hat{p}_{\text{data}}} L(f(x; \theta), y),$$

Riesgo empírico Distribución empírica Función de pérdida por muestra

Entrenamiento vs. optimización

En realidad, se pretende minimizar la función costo $J^*(\theta)$ donde la esperanza se compute en toda la distribución generadora en lugar de sólo en un conjunto de entrenamiento finito:

$$J^*(\theta) = \mathbb{E}_{(x,y) \sim p_{\text{data}}} L(f(x; \theta), y).$$

↑
Error de generalización esperado (riesgo)

↑
Distribución generadora (¡desconocida!)
↓

Si se conociera, aprendizaje = optimización

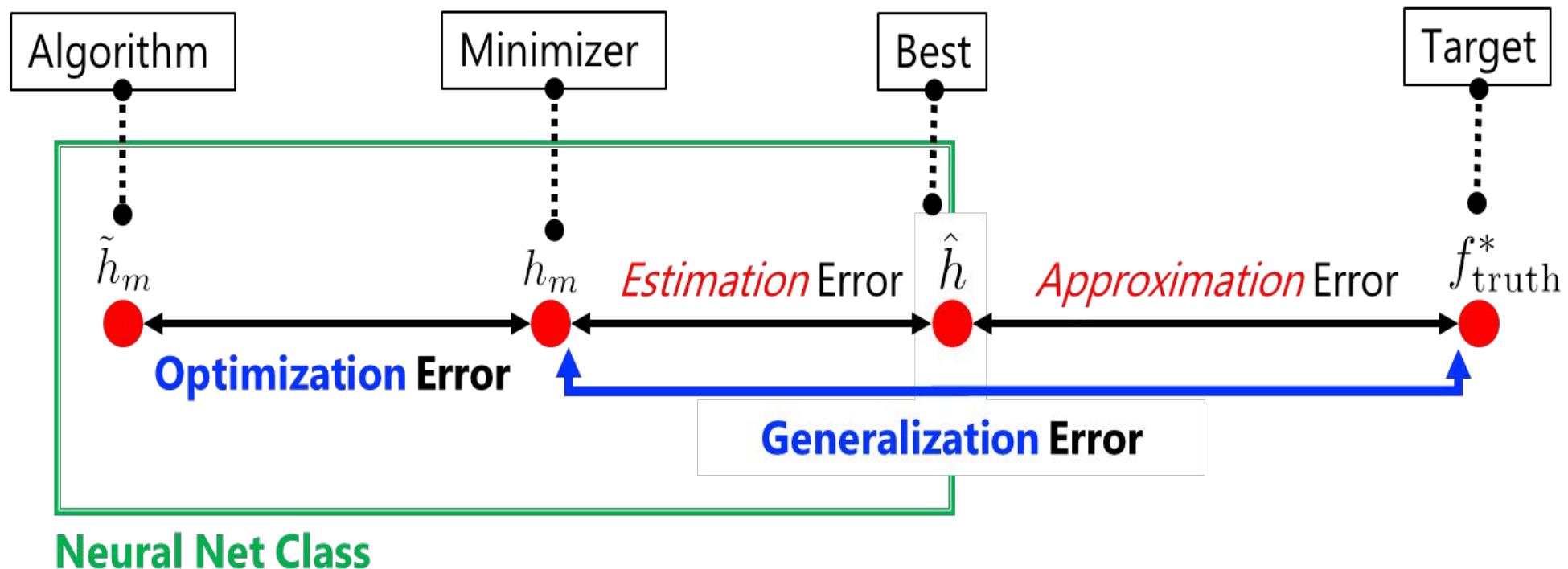
Entrenamiento vs. optimización

$$\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}(\mathbf{x}, \mathbf{y})}[L(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y})] = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}),$$

- **La minimización del riesgo empírico tiende al overfitting (aprender de memoria el conjunto de entrenamiento)**
- **La minimización se lleva a cabo mediante algoritmos basados en gradiente**
- **Aparece el concepto de Early Stopping**

Descomposición del error

$$\underset{\theta}{\text{minimize}} \text{Loss}_m^{\text{Höld}}(\theta) \implies \theta_m^* \implies h_m := h_{\theta_m^*}$$



Características principales de los algoritmos



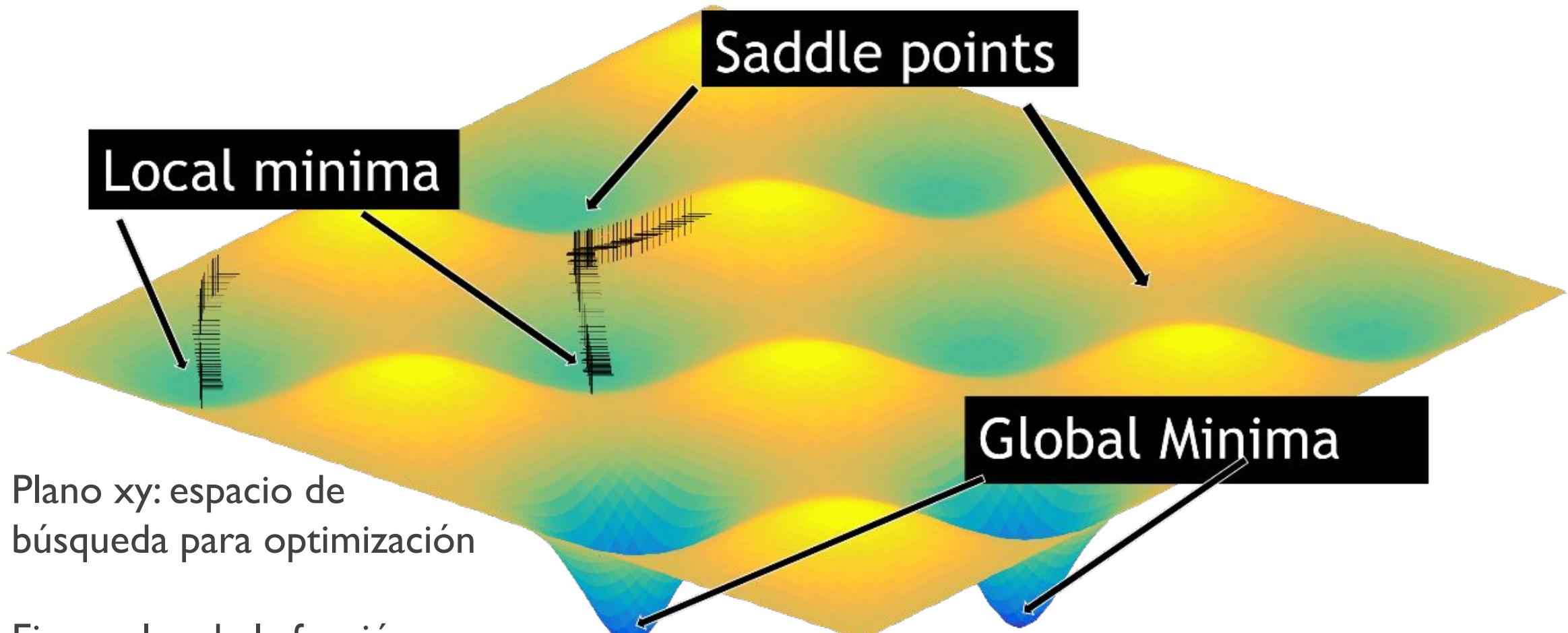
Puntos estacionarios

$$\nabla \mathcal{L}(\theta) = 0 \text{ minimization w.r.t. } \theta$$

Expansión de Taylor: $\mathcal{L}(\theta - \eta \nabla \mathcal{L}(\theta)) = \mathcal{L}(\theta) - \eta \|\nabla \mathcal{L}(\theta)\|_2^2 + \mathcal{O}(\eta^2)$

- Un método basado en gradiente es un método de búsqueda local ávida (“greedy”), capaz de evitar la maldición de la dimensionalidad (CoD)
- Un paso del algoritmo decrece la función de pérdida ($\eta > 0$) si $\nabla \mathcal{L}(\theta) \neq 0$
- Converge a un punto estacionario bajo ciertas condiciones (si no diverge y si $\eta \neq 0$)

Tipos de puntos estacionarios



¿Se pueden evitar
mínimos locales
“malos” en NN?

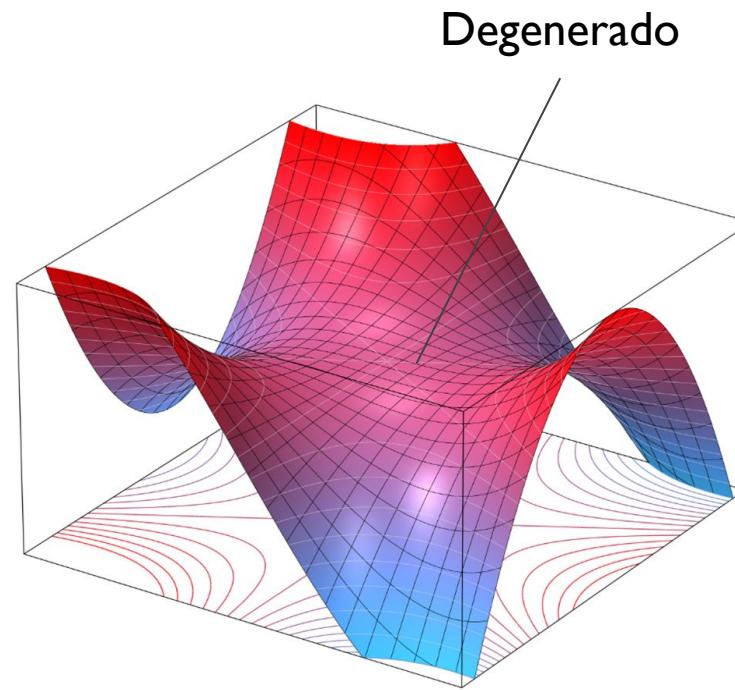
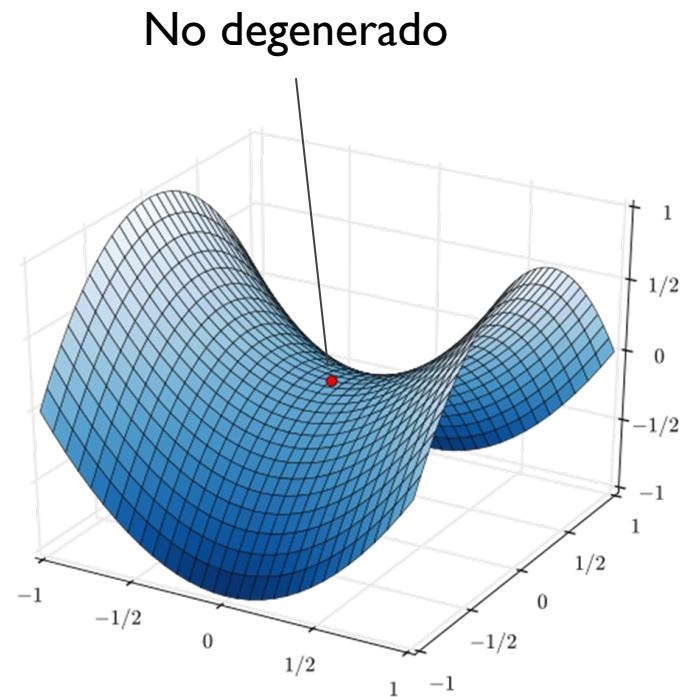
La respuesta
es... SI!!!

Publicaciones (últimos 5 años):

- Deep Learning without Poor Local Minima [NeurIPS 2016: Kenji Kawaguchi]
- Elimination of All Bad Local Minima in Deep Learning [AISTATS 2020: Kenji Kawaguchi, Leslie Kaelbling]
- Depth Learning with Nonlinearity Creates No Bad Local Minima in ResNets [NN 2020: Kenji Kawaguchi and Yoshua Bengio]

¿Que pasa con los puntos silla?

Tipos de puntos silla



No degenerado: Hessiano positivo definido con autovalores $\neq 0$ y signos opuestos.

Degenerado: Hessiano positivo semidefinido con autovalores todos 0.

Es más difícil escapar de un punto silla degenerado, ya que la información de segundo orden no es suficiente.

¿Se pueden evitar
mínimos locales
“malos” y puntos
silla en NN?

La respuesta es
también... SI!!!

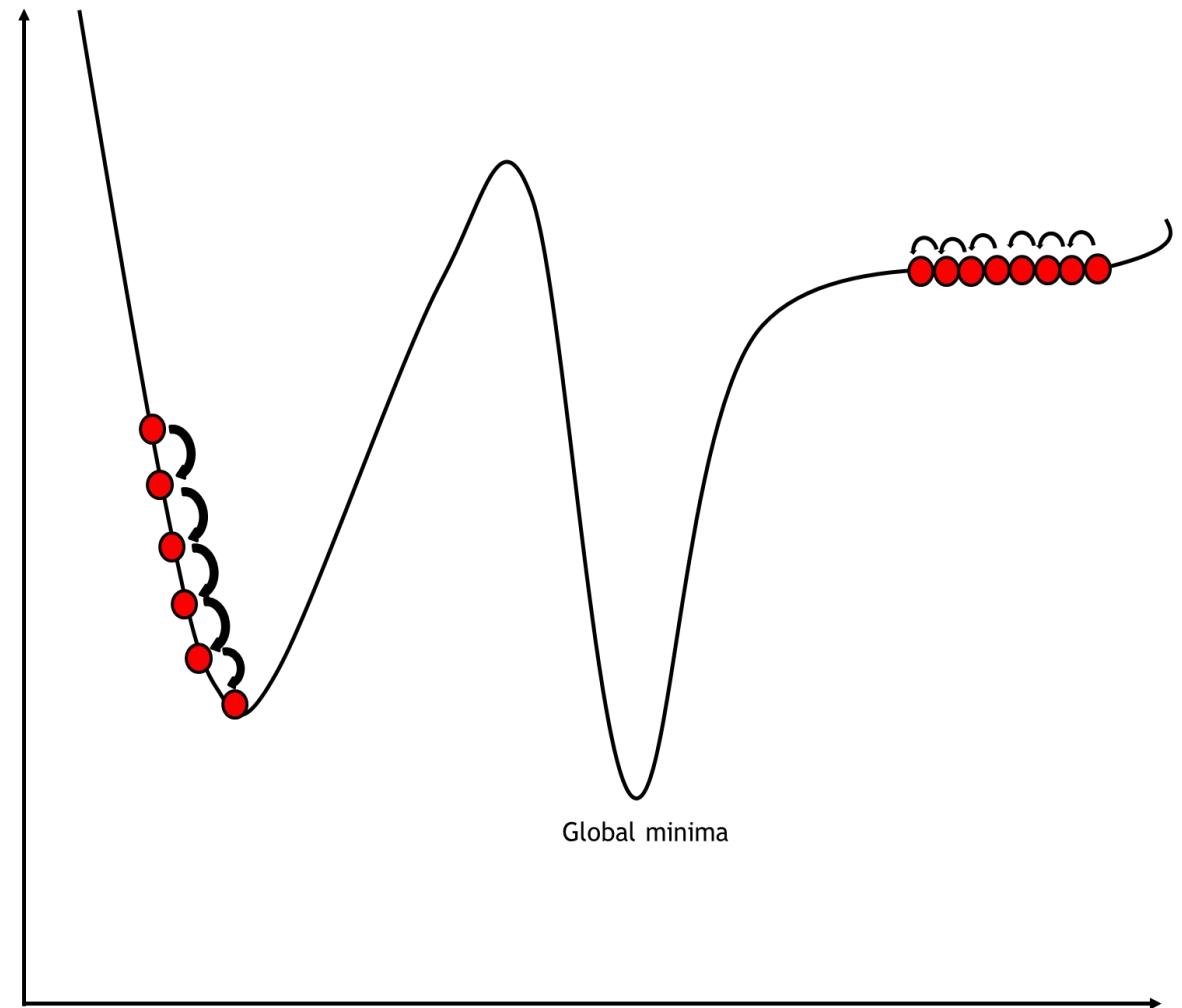
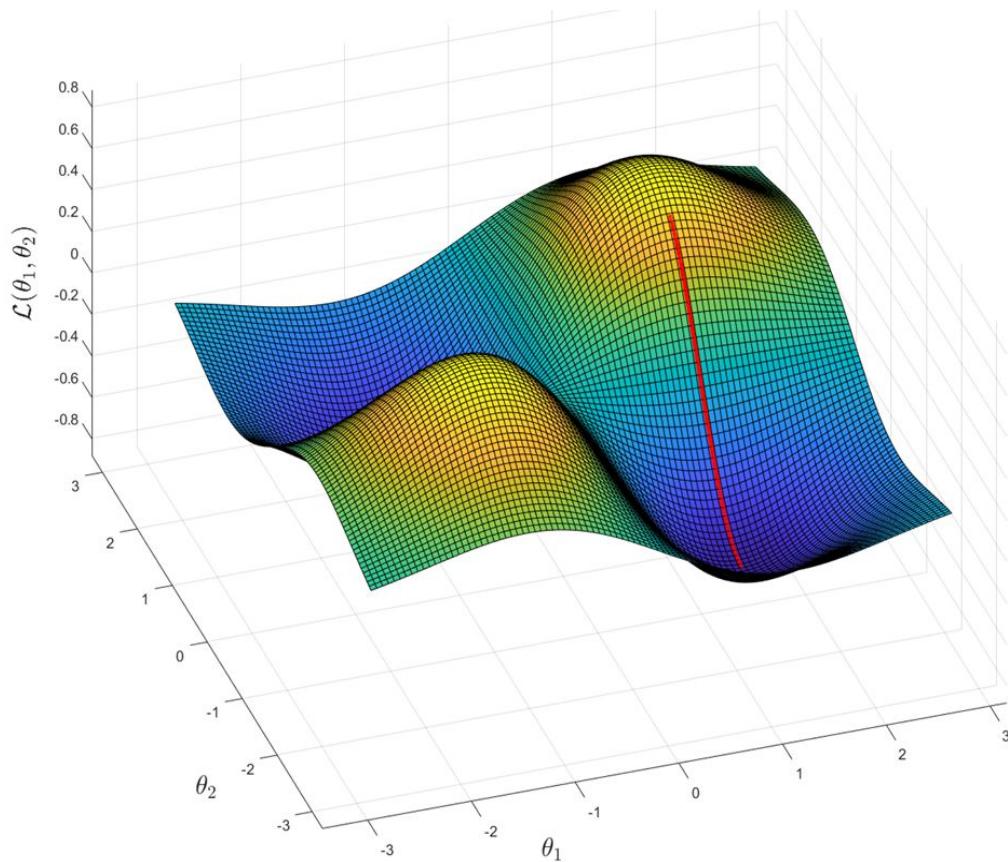
Publicaciones relacionadas:

- Allerton 2019: Kenji Kawaguchi and Jiaoyang Huang
- ICLR 2021: Kenji Kawaguchi
- ICML 2021: Keyulu Xu, Mozhi Zhang, Stefanie Jegelka, Kenji Kawaguchi
- AAAI 2021: Kenji Kawaguchi and Qingyun Sun

Gradient descent

$$\theta^* = \arg \min_{\theta} \mathcal{L}(\theta)$$

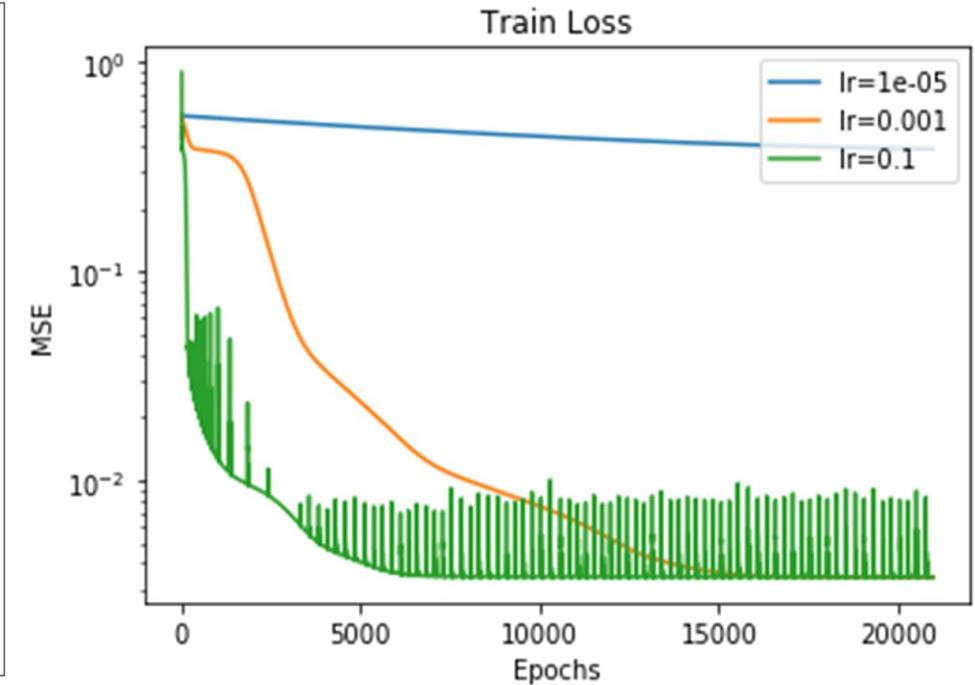
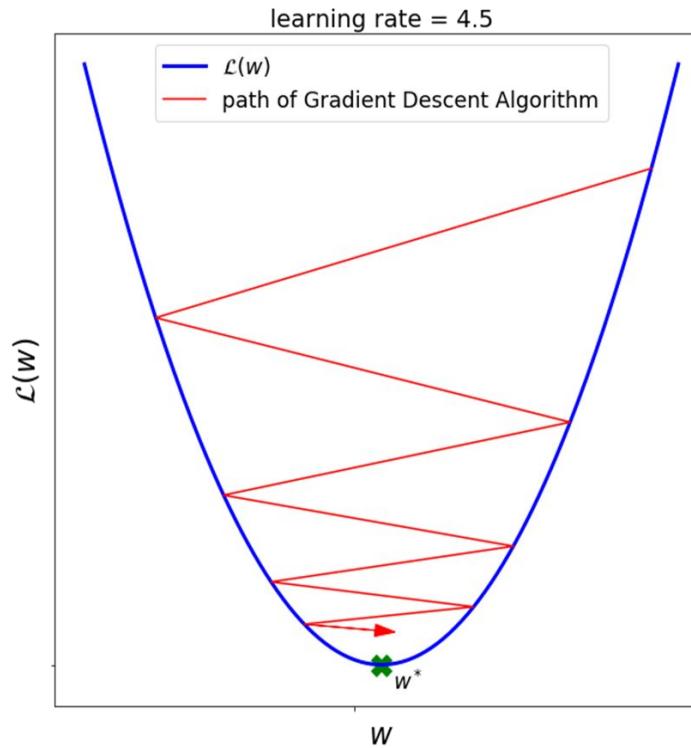
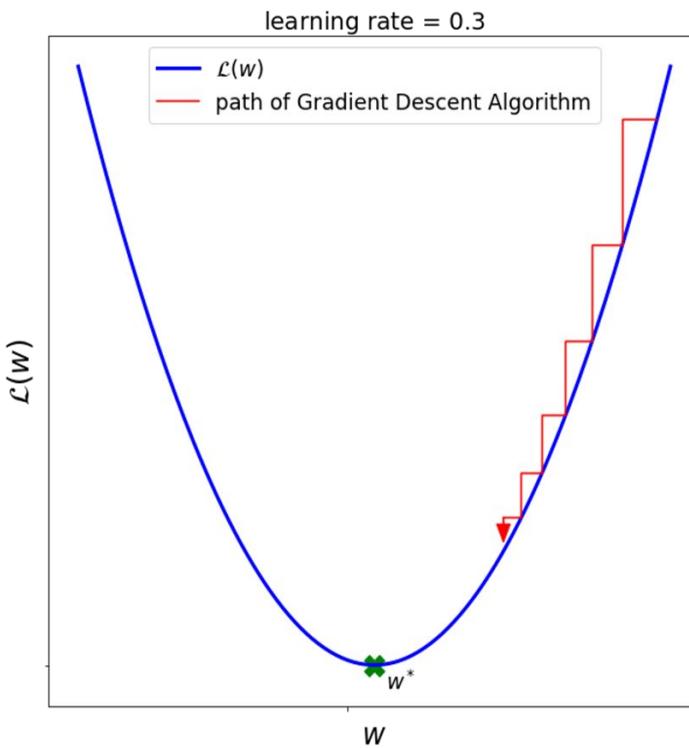
$$\theta_{n+1} = \theta_n - \eta \nabla_{\theta} \mathcal{L}(\theta)$$



Gradient descent (GD) vs. Stochastic gradient descent (SGD)

- Full GD realiza el cómputo del gradiente sobre todo el dataset. SGD lo realiza una instancia a la vez.
- Dichas instancias deben ser i.i.d. para asegurar convergencia al mínimo global: deben elegirse de manera aleatoria (o hacer “shuffle”)
- La aleatoriedad puede ayudar a escapar de mínimos locales malos en funciones muy irregulares o no convexas
- Como contra, la aleatoriedad hace que SGD no pueda converger totalmente
- Soluciones efectivas: introducir un decaimiento del learning rate, y/o realizar entrenamiento en mini-batch → Buena performance en arquitectura de GPUs

Influencia del learning rate

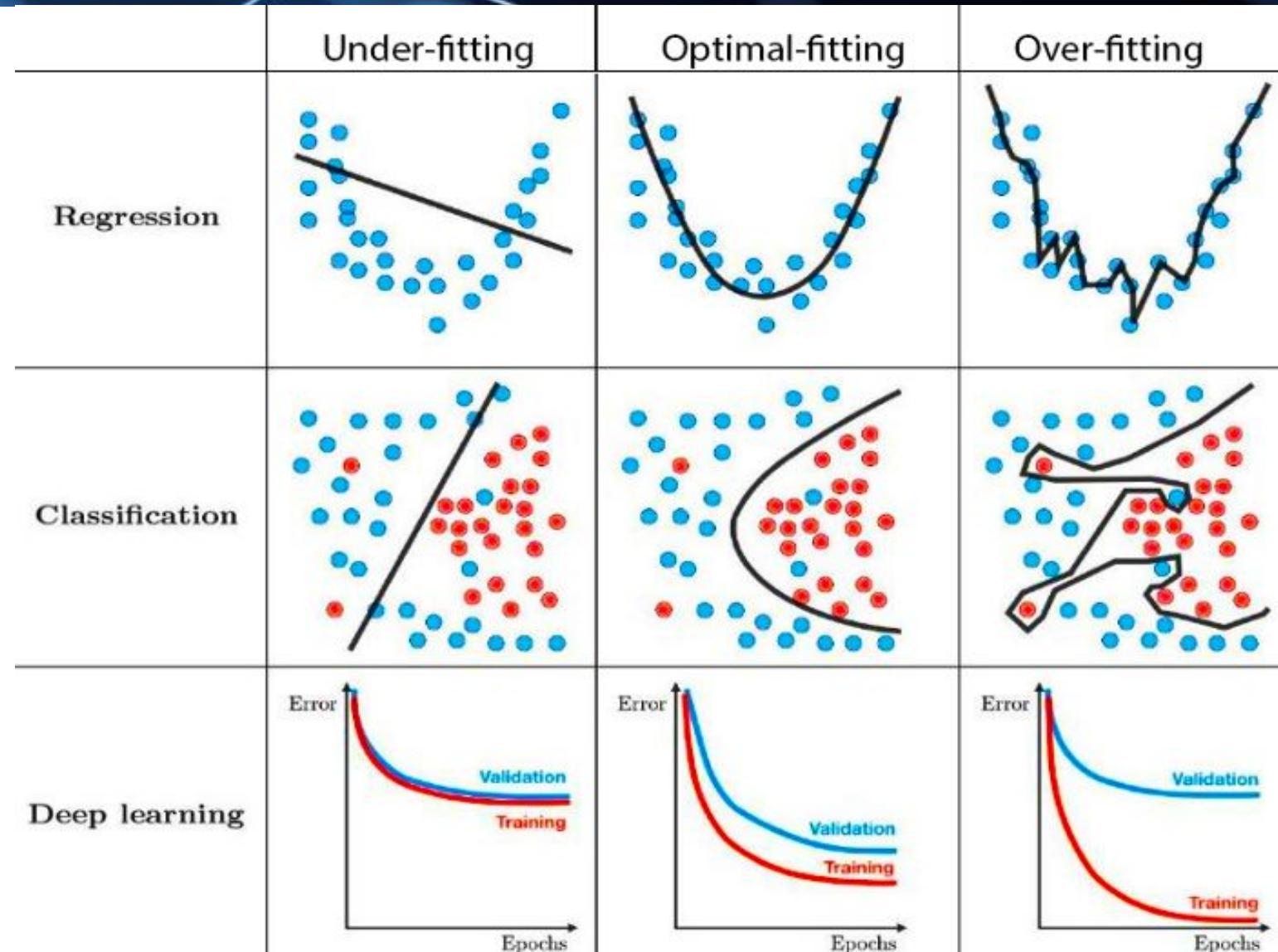


Ir asociado al aprendizaje de la función seno usando datos con ruido, mediante NN

Una estrategia efectiva es utilizar learning adaptativo/variable (más adelante...)

Underfitting y overfitting

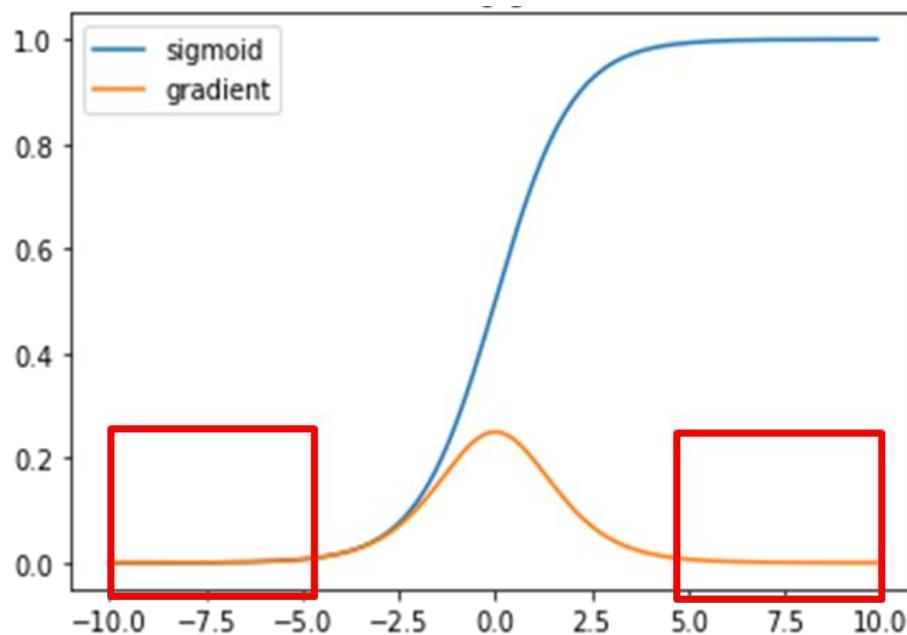
- Modelos de baja capacidad vs. modelos de alta capacidad
- Evaluar el gap entre los errores de entrenamiento y validación



Vanishing y exploding gradients

Diferentes capas de una red aprenden a tasas muy diferentes entre sí. En general las primeras capas son las que sufren estos problemas

Ej: vanishing gradients



Ej: exploding gradients. Multiplicar 100 matrices random (Gaussianas)

A single matrix

```
tensor([[ 0.7948, -0.5345,  2.2011, -1.2147, -0.8642],  
       [-1.0132, -0.3668,  1.3064, -0.3739,  0.3137],  
       [-0.7110,  0.0634, -3.0735, -0.8933, -0.2504],  
       [ 0.2037, -0.6473,  1.2173,  0.6089, -1.1243],  
       [ 0.7627,  0.8086, -0.9196, -1.1723,  0.4238]])
```

After multiplying 100 matrices

```
tensor([[-5.9421e+28,  2.5019e+28, -4.7395e+27,  9.2365e+28, -9.0353e+28],  
       [-4.1826e+28,  1.7611e+28, -3.3361e+27,  6.5015e+28, -6.3598e+28],  
       [ 8.4344e+28, -3.5513e+28,  6.7274e+27, -1.3111e+29,  1.2825e+29],  
       [-2.4234e+28,  1.0204e+28, -1.9328e+27,  3.7670e+28, -3.6850e+28],  
       [ 2.1614e+28, -9.1011e+27,  1.7237e+27, -3.3597e+28,  3.2865e+28]])
```

2010: paper de Xavier Glorot & Youshua Bengio “Understanding the difficulty of training Deep neural networks”, Proc 13th Int. Conf. on AI and Statistics pp. 249-256

Inicializaciones de los pesos

- **Idea:** varianza de las salidas de cada capa = varianza de las entradas de tal capa
- **Objetivo:** evitar vanishing y exploding gradients
- Inicialización Xavier (o Glorot):

$$w \sim \mathcal{N} \left(0, \sqrt{\frac{1}{\text{fan}_{\text{avg}}}} \right)$$

$$\text{fan}_{\text{avg}} = 0.5(\text{fan}_{\text{in}} + \text{fan}_{\text{out}})$$

fan: número de unidades en
entrada y salida

- Inicialización de He:

$$\mathcal{N}(0, \sqrt{\frac{2}{\text{fan}_{\text{in}}}})$$



Glorot Initialization

```
torch.nn.init.xavier_normal_(w)
```

Example:

```
w = torch.empty(5, 5)  
nn.init.xavier_normal_(w)
```

He Initialization

```
torch.nn.init.kaiming_normal_(w)
```

Example:

```
w = torch.empty(5, 5)  
torch.nn.init.kaiming_normal_(w)
```



Glorot Initialization

```
tf.initializers.GlorotNormal()
```

Example:

```
init = tf.initializers.GlorotNormal()  
w = initializer(shape=(4, 4))
```

He Initialization

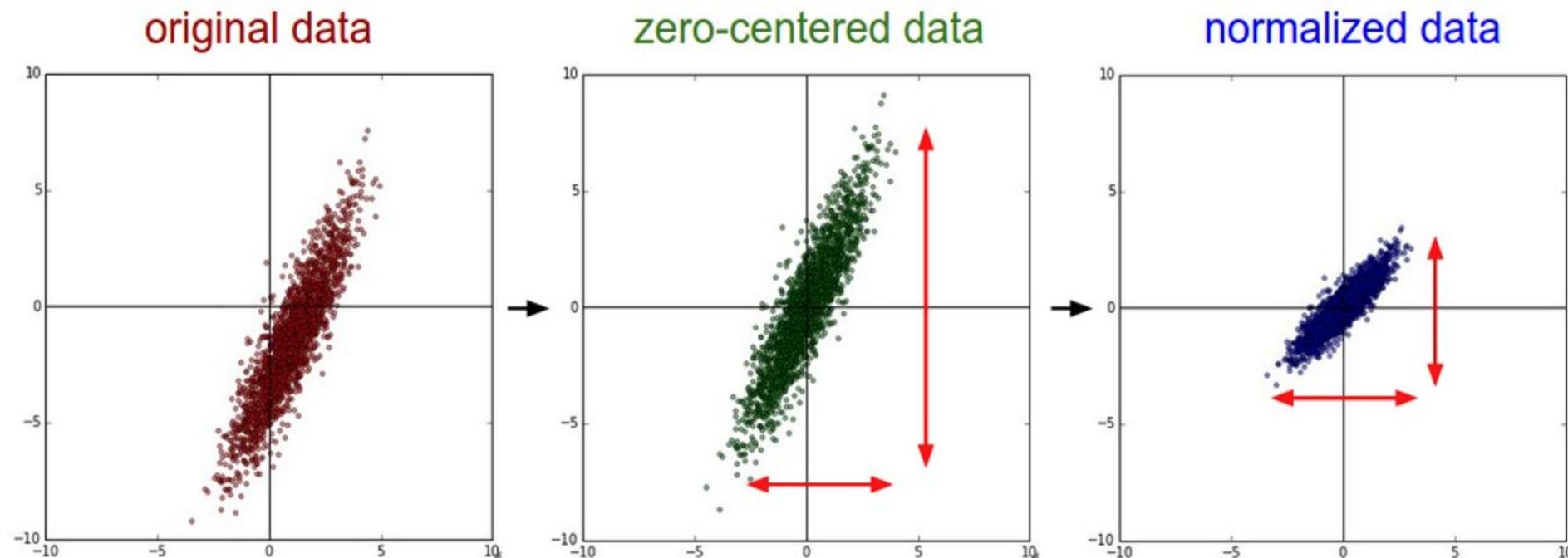
```
tf.initializers.HeNormal()
```

Example:

```
init = tf.initializers.HeNormal()  
w = initializer(shape=(4, 4))
```

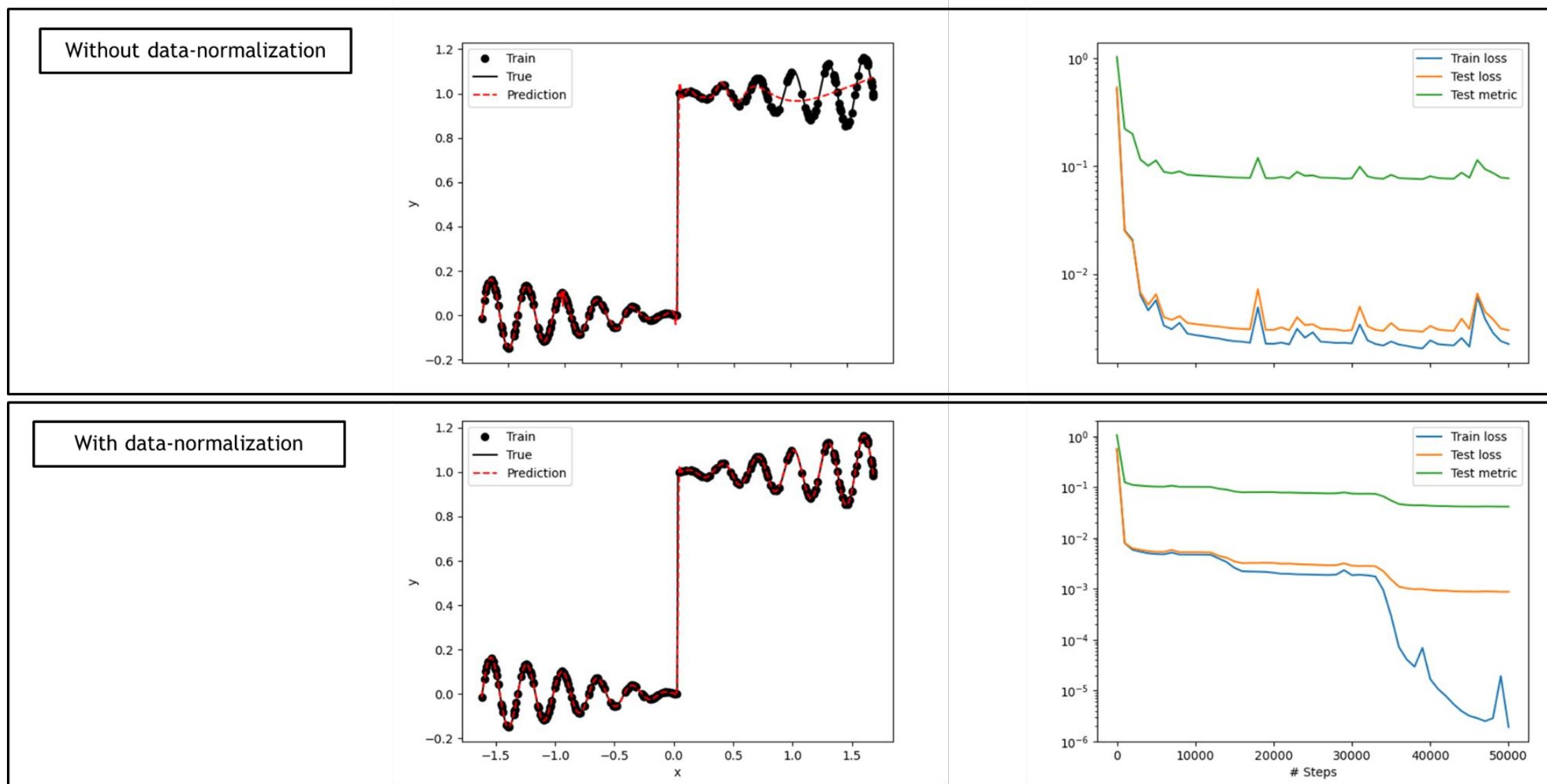
Normalización de los datos

- Normalización min-max
- Normalización z-score



Common data preprocessing pipeline. **Left:** Original toy, 2-dimensional input data. **Middle:** The data is zero-centered by subtracting the mean in each dimension. The data cloud is now centered around the origin. **Right:** Each dimension is additionally scaled by its standard deviation. The red lines indicate the extent of the data - they are of unequal length in the middle, but of equal length on the right.

Normalización de los datos



Batch normalization

Algorithm:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

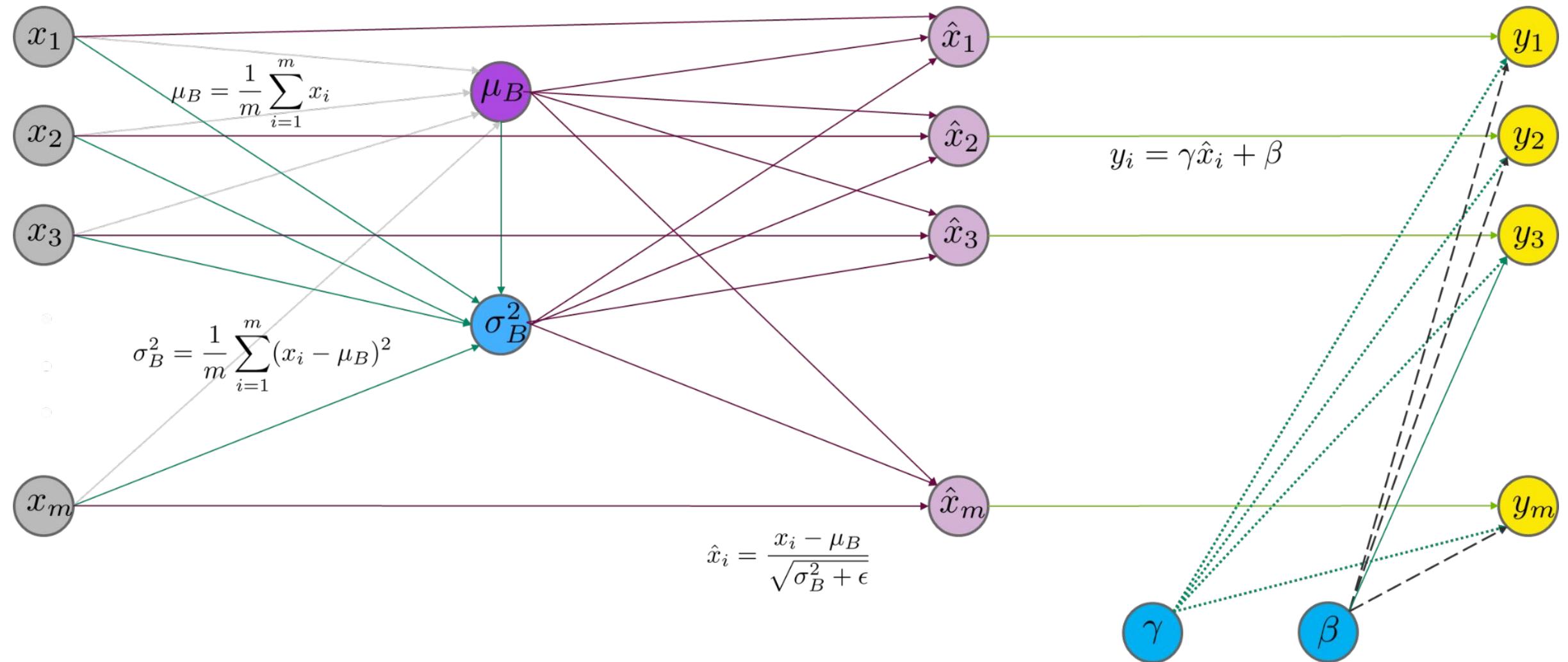
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

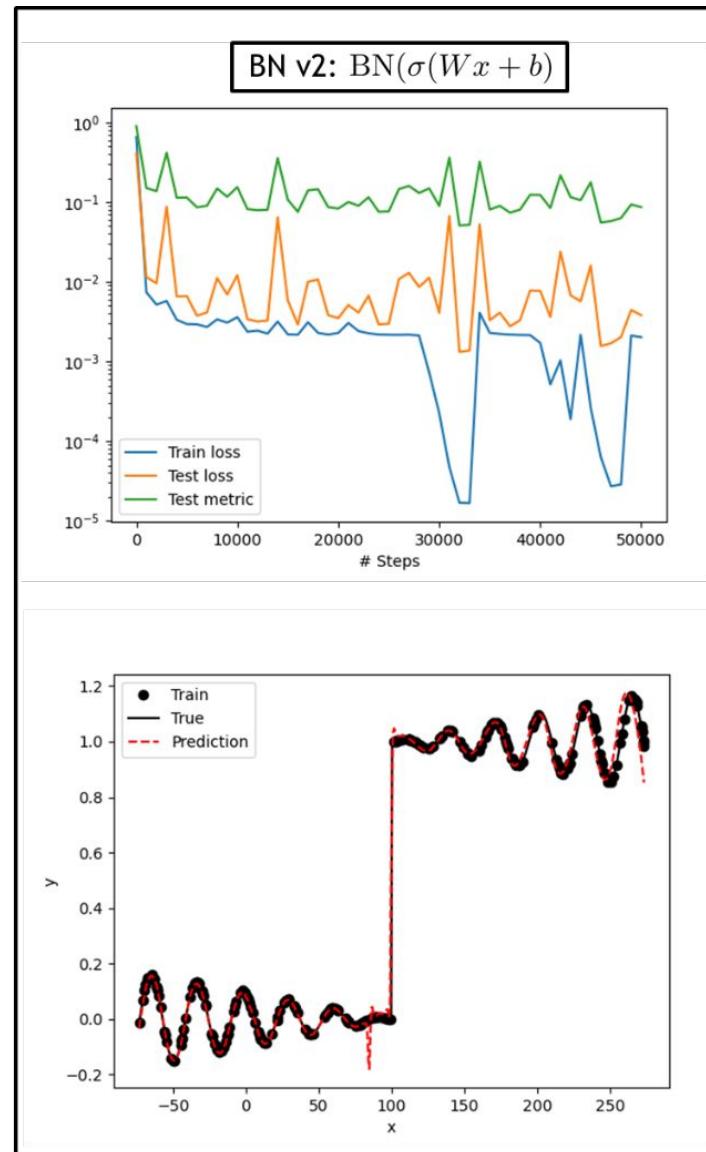
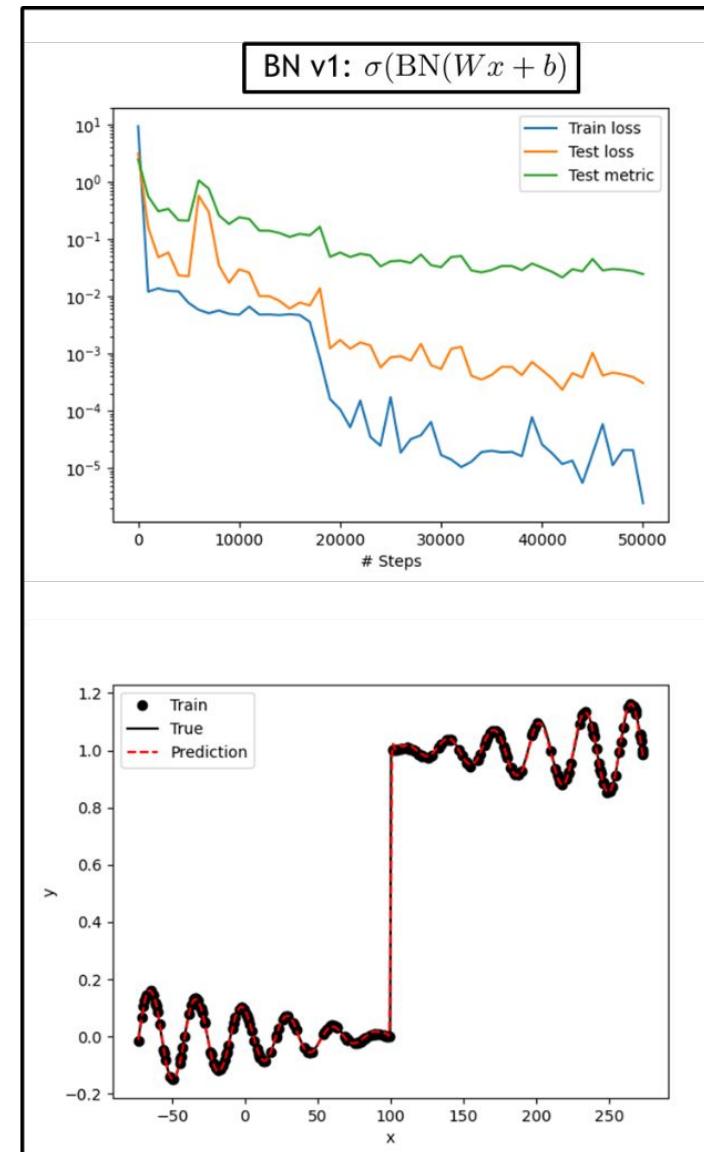
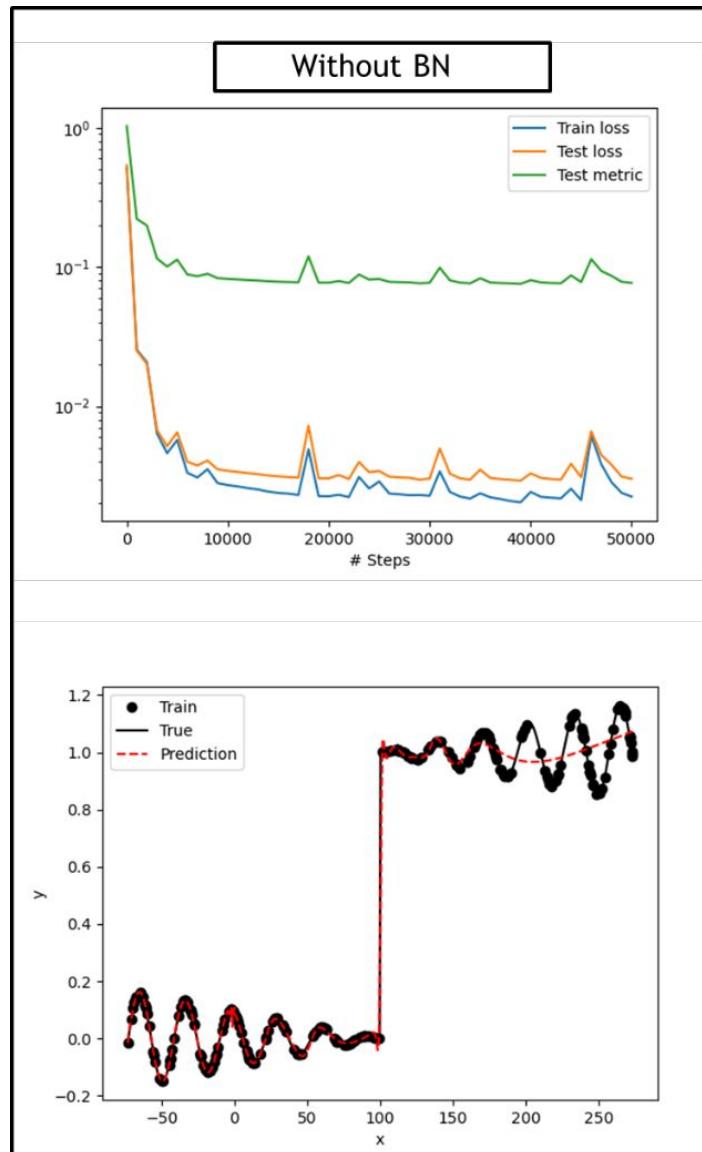
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

- y_i : transformación afín
- Mejora convergencia
- Mejora el flujo del gradiente
- Evita vanishing/exploding grads
- Permite lr mayores
- Reduce dependencia de la inicialización
- Actúa como estrategia de regularización

Batch normalization



Batch normalization



¿Qué optimizador utilizar?



Optimizadores de primer y segundo orden

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha \nabla f(\mathbf{x}_n)$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha \mathbf{H}_n^{-1} \nabla f(\mathbf{x}_n)$$

First Order	Second Order
Momentum	Newton Method
Nesterov	Quasi-Newton methods: BFGS
Adagrad	Quasi-Newton methods: L-BFGS
RMSProp	
Adam	
Nadam	
Rprop	

Una estrategia práctica que ofrece buenos resultados es utilizar un optimizador de segundo orden a continuación de uno de primer orden

Adam: Adaptive moment optimizer (primer orden)

- Método híbrido que combina ideas de optimizadores basados en el concepto de “momento” y del optimizador RMSProp
- Integra en su implementación un learning rate adaptativo
- Ofrece buena convergencia, performance robusta en una gama amplia de problemas, y es computacionalmente eficiente
- Puede conducir al overfitting debido al LR adaptativo

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
3. $\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^T}$
4. $\widehat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^T}$
5. $\theta \leftarrow \theta + \eta \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}}} + \epsilon$

Quasi-Newton (segundo orden)

- El cálculo exacto de la matriz Hessiana y su inversa es prohibitivo en DL
- Se utilizan **métodos iterativos** para computar una **aproximación**: método BFGS y L-BFGS
- En consecuencia, no son estrictamente de segundo orden (convergencia “super-lineal”)

Algorithm 1: BFGS Algorithm

Input : initial $\mathbf{x}_0 \in \mathbb{R}^n$, functions $f(\mathbf{x})$, $\nabla f(\mathbf{x})$. tolerance θ

Output: x

```

1 initialize  $\mathbf{H}^{-1} \leftarrow \mathbb{I}$ 
2 do
3   compute  $s = \mathbf{H}^{-1} \nabla f(\mathbf{x})$ 
4   perform a line search  $\min_{\alpha} f(\mathbf{x} + \alpha d)$ 
5    $s \leftarrow \alpha d$ 
6    $y \leftarrow \nabla f(\mathbf{x} + s) - \nabla f(\mathbf{x})$ 
7    $\mathbf{x} \leftarrow \mathbf{x} + s$ 
8   update  $\mathbf{H}^{-1} \leftarrow \left( \mathbb{I} - \frac{ys^\top}{s^\top y} \right) \mathbf{H}^{-1} \left( \mathbb{I} - \frac{ys^\top}{s^\top y} \right) + \frac{ss^\top}{s^\top y}$ 
9 while until  $\|s\|_\infty < \theta$ ;

```

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha(\mathbf{H}_n^{-1} \mathbf{g}_n)(\mathbf{x}_n)$$

$$\nabla_{\boldsymbol{\theta}}^2 f(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_1^2} & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_1 \partial \theta_2} & \dots & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_1 \partial \theta_n} \\ \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_2^2} & \dots & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_2 \partial \theta_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_d \partial \theta_1} & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_d \partial \theta_2} & \dots & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_d^2} \end{bmatrix}$$

$$\mathbf{H}_{n+1} = \mathbf{H}_n + \frac{\mathbf{y}_n \mathbf{y}_n^\top}{\mathbf{y}_n^\top \mathbf{s}_n} - \frac{\mathbf{H}_n \mathbf{s}_n \mathbf{s}_n^\top \mathbf{H}_n}{\mathbf{s}_n^\top \mathbf{H}_n \mathbf{s}_n}$$

Learning rate scheduling



Learning rate scheduling

- Utilizar un learning rate constante no es la mejor estrategia
- Usualmente se comienza con lr alto y luego se disminuye
- Existen varias estrategias: decaimiento exponencial, polinómico, cíclico, etc.
- Ojo: Adam implementa automáticamente un lr scheduling

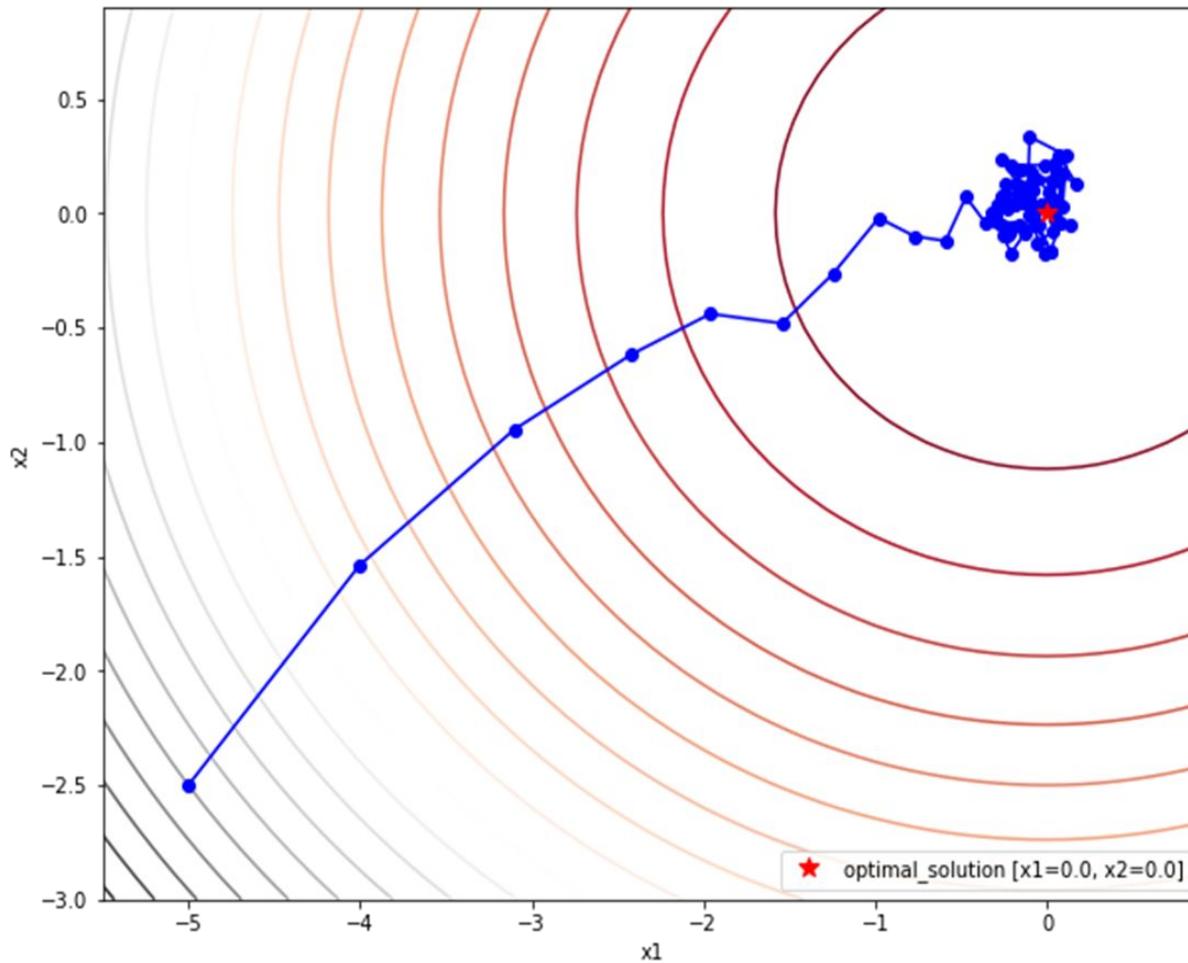
$$\eta(t) = \eta_i \quad \text{if} \quad t_i \leq t \leq t_{i+1} : \text{Piecewise decay}$$

$$\eta(t) = \eta_0 \exp(-\lambda t) : \text{Exponential decay}$$

$$\eta(t) = \eta_0(\beta t + 1)^{-\alpha} : \text{Polynomial or Power decay}$$

Learning rate scheduling: constante

$$f(x) : x_1^2 + 2x_2^2$$



epoch 70, x1: 0.020621, x2: -0.171758

```
def f(x1, x2):
    return x1**2 + 2*x2**2

def sgd(x1, x2, f_grad):
    g1, g2 = f_grad(x1, x2)
    g1 += torch.normal(0.0, 1, (1,)) # Gradient with noise
    g2 += torch.normal(0.0, 1, (1,)) # Gradient with noise
    eta_t = eta * lr()
    return (x1 - eta_t * g1, x2 - eta_t * g2)

def f_grad(x1, x2):
    return 2 * x1, 4 * x2

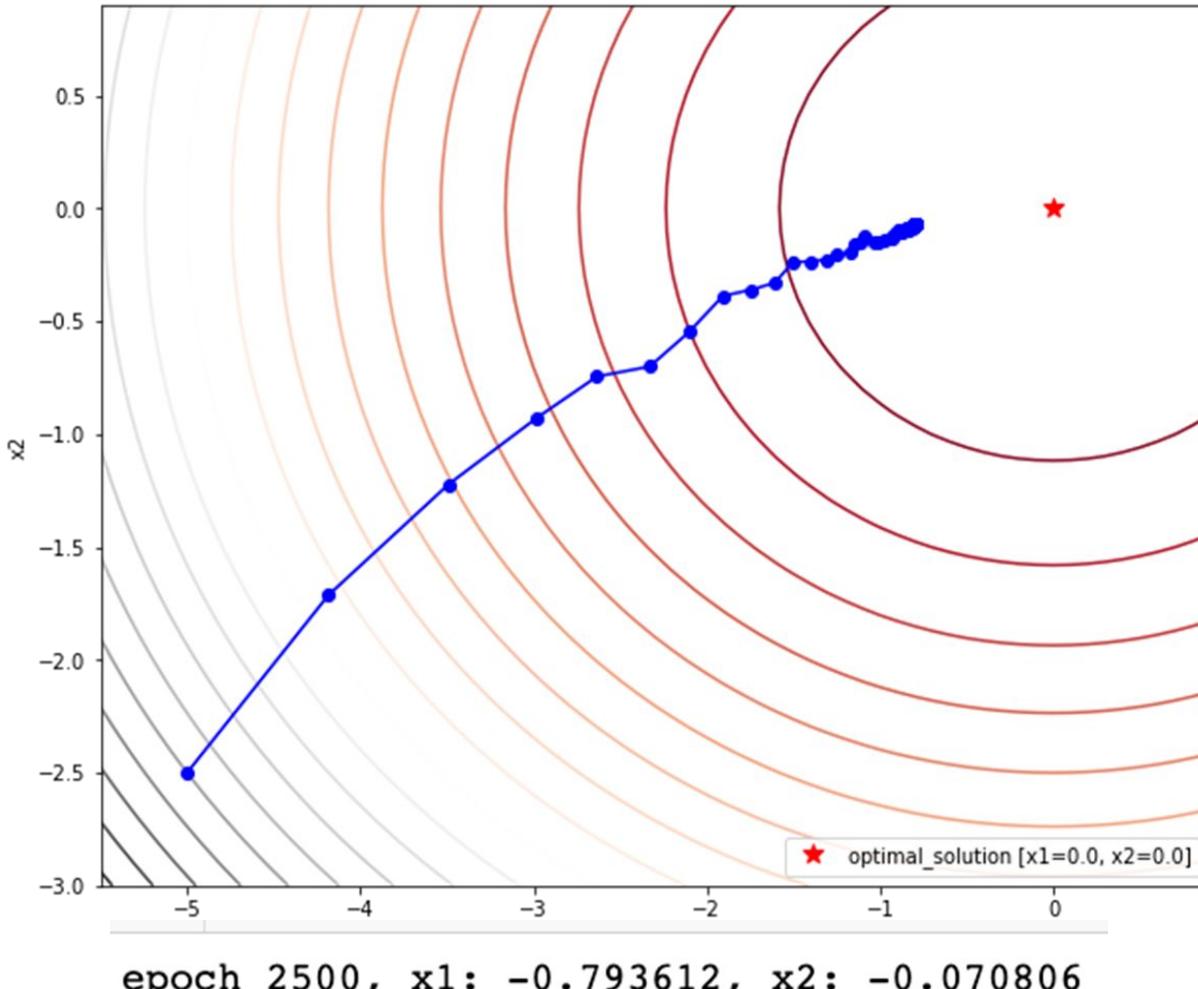
def train(trainer,f_grad, steps=50):
    x1, x2 = -5, -2.5
    results = [(x1, x2)]
    for i in range(steps):
        x1, x2 = trainer(x1, x2, f_grad)
        results.append((x1, x2))
    print(f'epoch {i + 1}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results

def constant_lr():
    return 1

eta = 0.1
lr = constant_lr
results = train(sgd, steps=70, f_grad=f_grad)
```

Learning rate scheduling: exponential

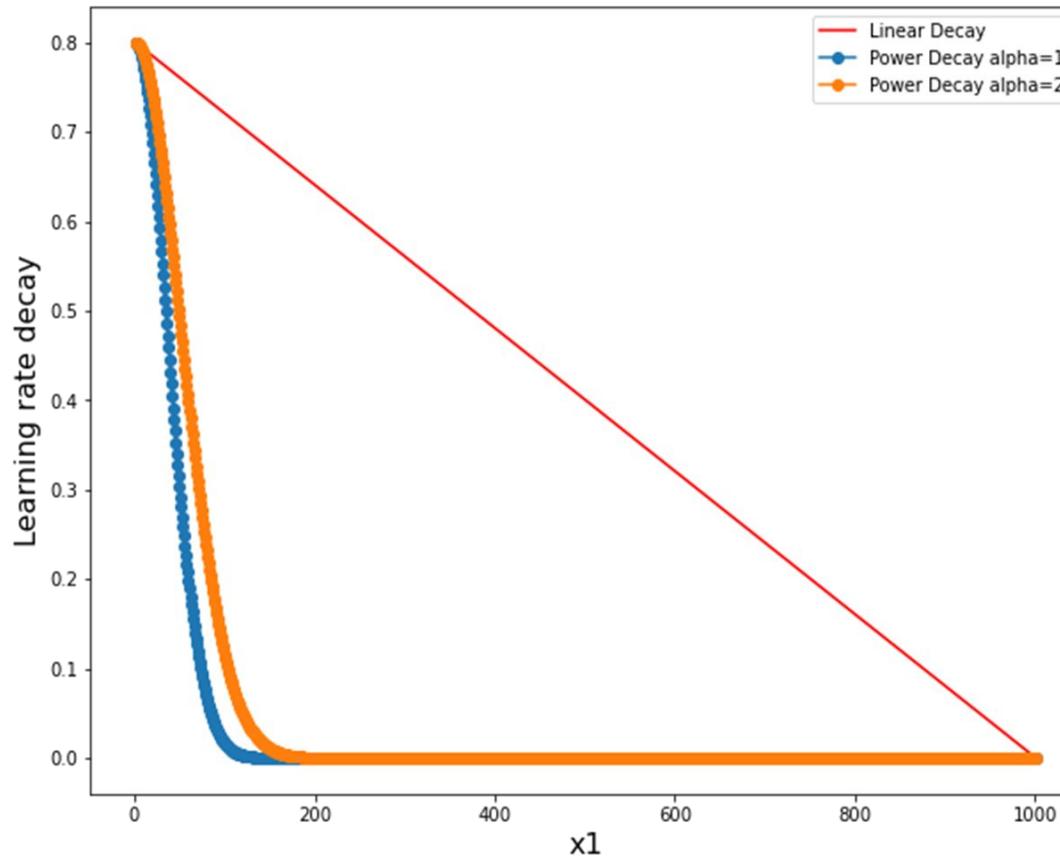
$$f(x) : x_1^2 + 2x_2^2$$



```
--  
13  
14 def f(x1, x2):  
15     return x1**2 + 2*x2**2  
16  
17  
18 def sgd(x1, x2, f_grad):  
19     g1, g2 = f_grad(x1, x2)  
20     # Simulate noisy gradient  
21     g1 += torch.normal(0.0, 1, (1,))  
22     g2 += torch.normal(0.0, 1, (1,))  
23     eta_t = eta * lr()  
24     return (x1 - eta_t * g1, x2 - eta_t * g2)  
25  
26 def f_grad(x1, x2):  
27     return 2 * x1, 4 * x2  
28  
29 def train_2d(trainer,f_grad, steps=50):  
30     x1, x2 = -5, -2.5  
31     results = [(x1, x2)]  
32     for i in range(steps):  
33         x1, x2 = trainer(x1, x2, f_grad)  
34         results.append((x1, x2))  
35     print(f'epoch {i + 1}, x1: {float(x1):f}, x2: {float(x2):f}')  
36     return results  
37  
38 def exponential_lr():  
39     global it  
40     it += 1  
41     return np.exp(-0.1 * it)  
42 it = 1  
43 eta = 0.1  
44 lr = exponential_lr  
45 results = train_2d(sgd, steps=2500, f_grad=f_grad)
```

Learning rate scheduling: potencia

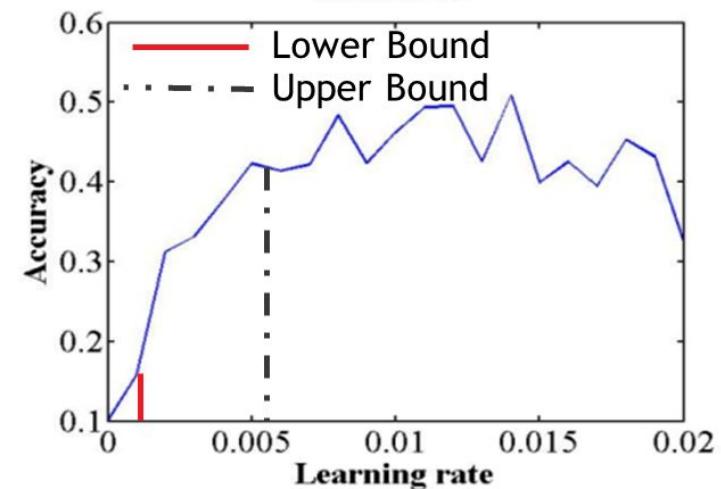
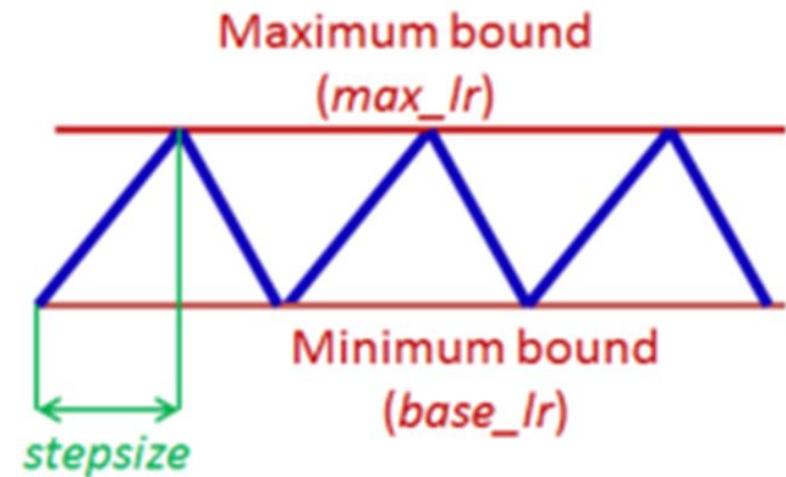
$$\eta(t) = \frac{\eta_0}{(1 + \beta t)^\alpha}$$



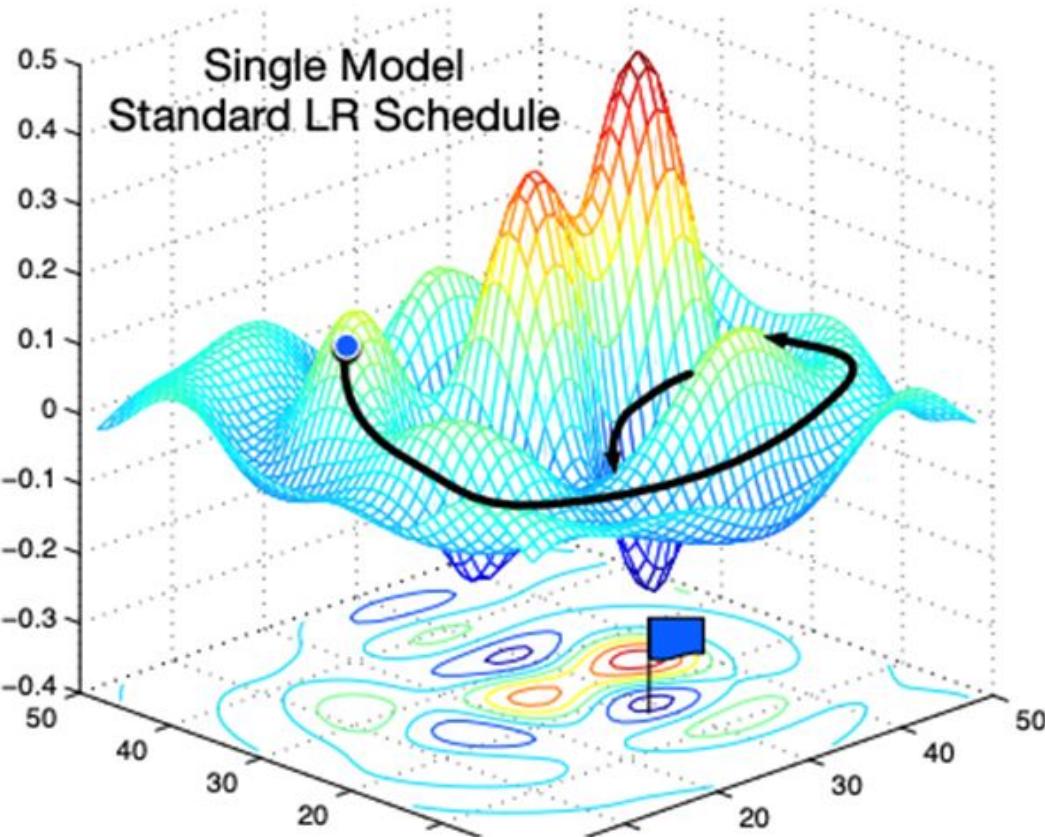
```
1 class lr_decay():
2     def __init__(self, epochs, initial_lr, power):
3         self.epochs = epochs
4         self.initial_lr = initial_lr
5         self.power = power
6
7     def linear_decay(self, epoch):
8         decay = (1 - (epoch / float(self.epochs)))
9         eta_updated = self.initial_lr * decay
10        return float(eta_updated)
11
12    def power_decay(self, epoch, lr, alpha):
13        decay = self.initial_lr/float((self.epochs))
14        return float(lr * 1 / (1 + decay * epoch)**alpha)
15
16    def power_deacay_hist(self, alpha=1):
17        lr = self.initial_lr
18        power_decay_list = []
19        for ep in range(0, self.epochs + 1):
20            lr = self.power_decay(ep, lr, alpha)
21            power_decay_list.append(lr)
22        return power_decay_list
23
24
25
26 initial_lr = 0.8
27 epochs = 1000
28 lr_sch = lr_decay(epochs, initial_lr, power)
29 lr_linear = np.array([lr_sch.linear_decay(ep) for ep in range(0, epochs + 1)])
30 lr_power_alpha_1 = lr_sch.power_deacay_hist(alpha=1)
31 lr_power_alpha_2 = lr_sch.power_deacay_hist(alpha=0.5)
32 --
```

Learning rate scheduling: cíclico

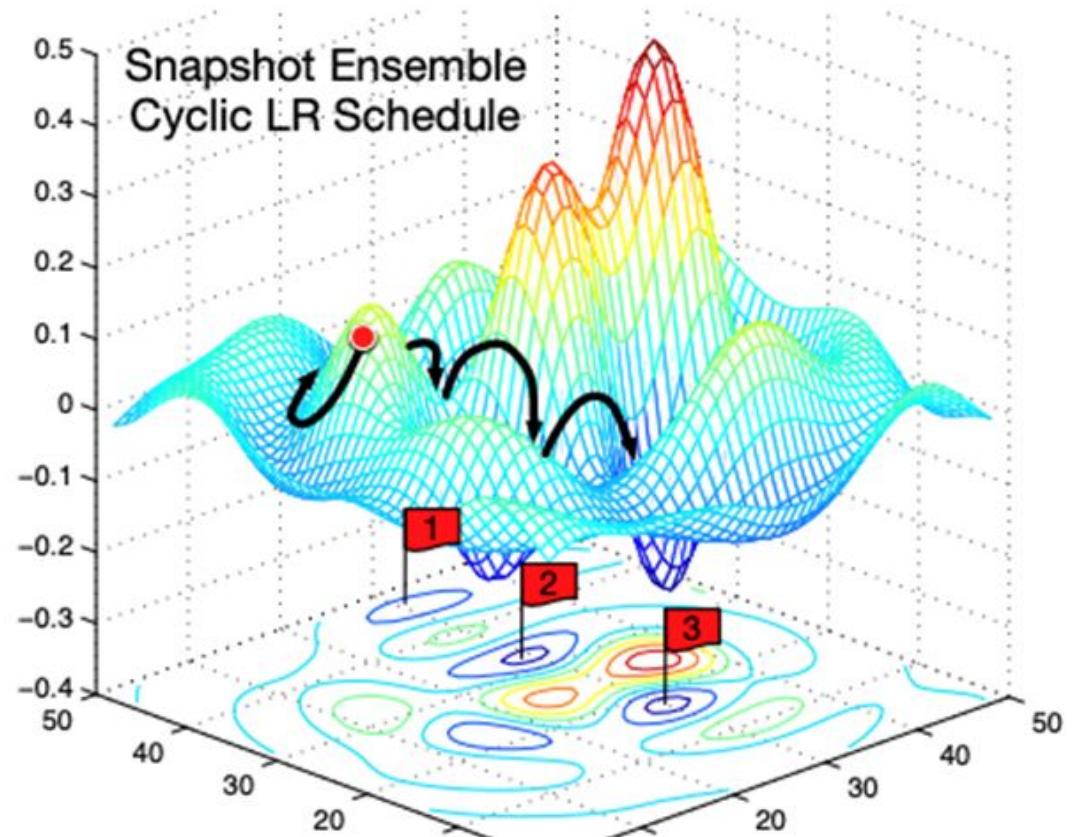
- La idea es sucesivamente converger y escapar de mínimos locales hasta llegar a un buen mínimo local
- Stepsize: 2-10 epochs (suele ser diferente el N° de stepsizes en la parte ascendente que en la descendente)
- Base_Ir: se elige en el punto donde se aprecia un incremento en el accuracy (problema de clasificación)
- Max_Ir: se elige en el punto donde se aprecia un estancamiento en el accuracy



Learning rate scheduling: cíclico



SGD con learning rate schedule típico converge a un mínimo local al final del entrenamiento

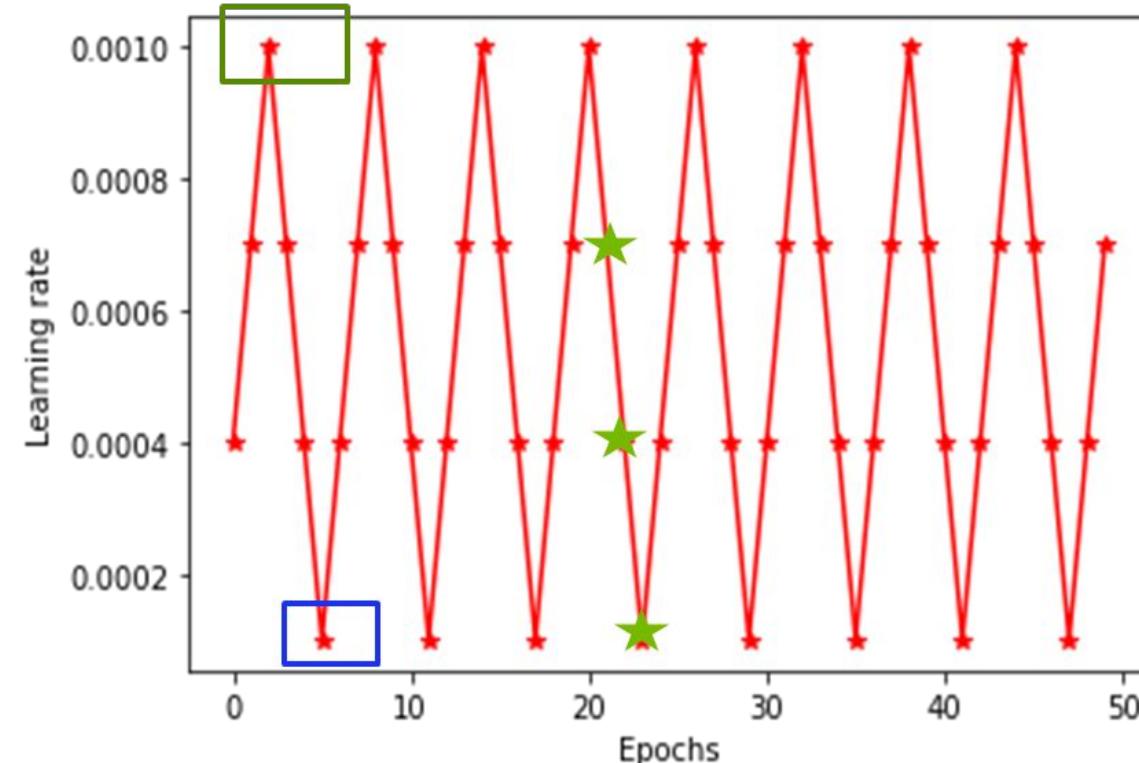


LR scheduling cíclico converge y escapa de múltiples mínimos locales

Learning rate scheduling: cíclico (implementación Pytorch)

```
optimizer = SGD(Net.parameters(), lr = lr_initial)

scheduler = torch.optim.lr_scheduler.CyclicLR(optimizer, base_lr=0.0001,\n                                              max_lr=0.001,step_size_up=3,mode="triangular")
```



Regularización



Regularización L2

- Regresión “ridge” o de Tikhonov (“weight decay” en PyTorch)
- Se utiliza para reducir los errores de test para nuevas entradas. En contrapartida, puede aumentar el error de entrenamiento
- Se practica durante el entrenamiento, pero no durante el testing

$$\mathcal{L}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2} \mathbf{w}^\top \mathbf{w} + \mathcal{L}(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

α : parámetro de regularización

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \mathbf{w} + \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

Gradiente de la función obj. (contribución lineal)

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}$$

Solución cerrada (regresión lineal)

- Las componentes de $\mathbf{X}^\top \mathbf{X}$ en la diagonal corresponden a la varianza de cada una de las entradas
- El algoritmo de aprendizaje percibe una varianza mayor en tales componentes, lo cual contrae los pesos asociados a las demás componentes cuya covarianza sea menor

Regularización L1

$$\mathcal{L}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \mathcal{L}(\mathbf{w}; \mathbf{X}, \mathbf{y}) + \alpha \|\mathbf{w}_1\|_1$$

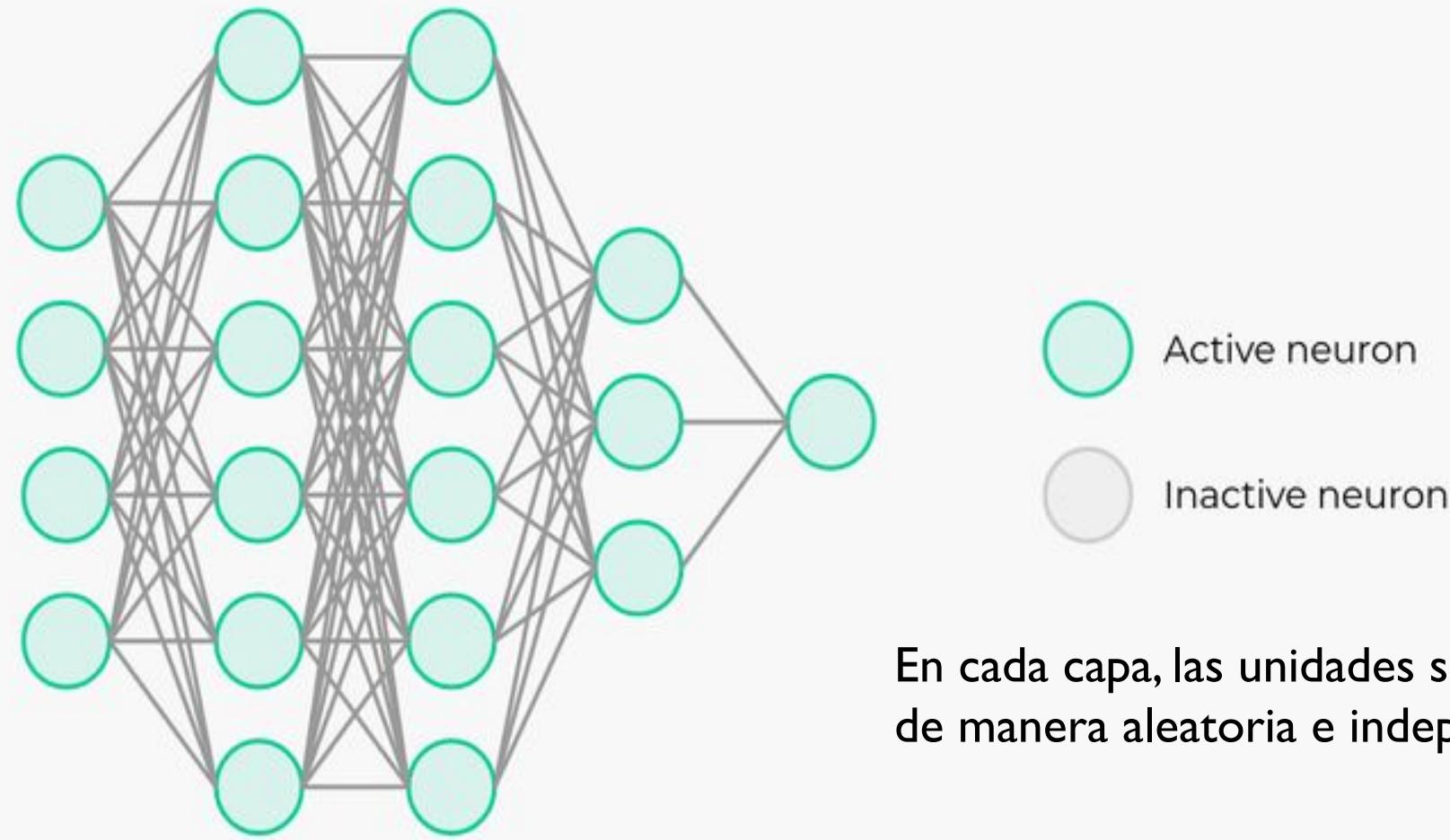
$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \text{sign}(\mathbf{w}) + \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{X}, \mathbf{y}; \mathbf{w})$$

- Regresión “LASSO” (least absolute shrinkage and selection operator)
- La contribución al gradiente es no lineal
- Promueve soluciones “sparse” → Algunos parámetros tienen un valor óptimo de cero
- Este tipo de regularizador se utiliza como selector de features (aquellas cuyos parámetros sean cero pueden ser descartadas del modelo)
- No tiene implementación en PyTorch ni TF, pero podemos definirla fácilmente

Dropout

- Estrategia propuesta por Hinton en 2012
- Permite crear “ensembles” de muchas DNN, es decir que entrena de manera ingeniosa y sin costo adicional, múltiples modelos y los evalúa en cada conjunto de prueba
- Puede ser utilizado junto con estrategias L1 o L2
- Es una estrategia equivalente a regularización L2 en el caso de regresión lineal, pero no en DL
- Dropout rate “ p ”: hiperparámetro definido en $[0, 1]$ en cada capa, que indica la probabilidad de que una unidad sea removida temporalmente, y que actúa como una máscara sobre cada unidad.

Dropout: esquema de funcionamiento



$$P^{[0]} = 0.0$$

$$P^{[1]} = 0.5$$

$$P^{[2]} = 0.33$$

$$P^{[3]} = 0.0$$

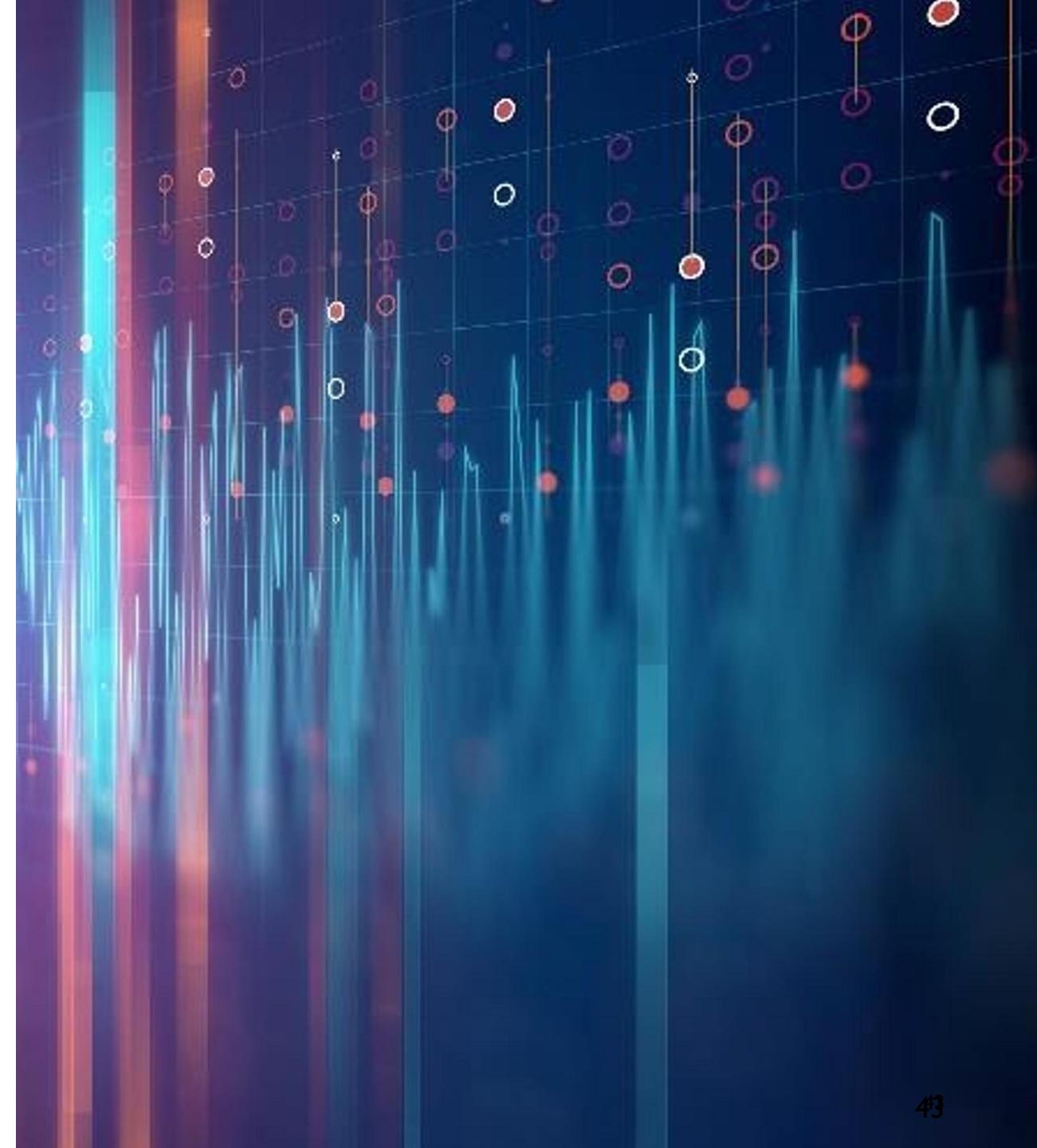
Dropout: características destacables

- Consideremos el caso de entrenar un modelo via SGD (una entrada del dataset a la vez) utilizando dropout
- Dado que cada neurona puede estar presente o ausente debido al dropout rate, hay un total de 2^m (donde “m” es el número de neuronas “droppeables”) redes neuronales posibles por cada entrada del dataset
- **Luego de correr N epochs, habremos entrenado N redes neuronales diferentes, por cada entrada del dataset!**
- La red neuronal resultante puede ser vista como un “ensemble” promediado de todas estas redes similares

Dropout: datos prácticos

- En la práctica, se aplica dropout a las capas 1-3 más alejadas, pero no a la capa de salida
- El dropout se encuentra activo solo durante el entrenamiento. Para comparar el error de entrenamiento con el de validación se debe evaluar con toda la red sin dropout
- Si el modelo aún así overfitea, podemos incrementar el dropout rate
- El dropout tiende a reducir la tasa de convergencia, pero los beneficios son mayores que esta desventaja

Sumario



Sumario

- Las principales dificultades en entrenar NN se relacionan con la estrategia de back propagation, y a la necesidad de minimizar una función no convexa multidimensional.
- Stochastic gradient descent (SGD) y mini-batch GD con decaimiento de learning rate son métodos prácticos y efectivos.
- El underfitting ocurre para modelos de baja capacidad, mientras que el overfitting ocurre para modelos de gran capacidad.
- Tanto las inicializaciones de Xavier y He como la normalización de las entradas son componentes claves y efectivos para entrenar NN.
- Batch normalization puede conducir a una convergencia más rápida, evitar vanishing gradients, permitir learning rates mas elevados, y actuar como regularizador.

Sumario

- Hay muchos optimizadores, pero la combinación de ADAM + L-BFGS es usualmente la que mejores resultados proporciona.
- Las estrategias de decaimiento y cíclicas para el learning rate son importantes para acelerar la convergencia y evitar mínimos “malos”.
- La regularización L2 conduce a una mejor generalización mientras que la L1 promueve la “sparsity”. Cuál emplear depende del problema.
- La regularización mediante dropout es efectiva para regularizar pero depende fuertemente de la tasa de dropout, que debe ser variable a través de las capas.

¿Dudas o
preguntas?

