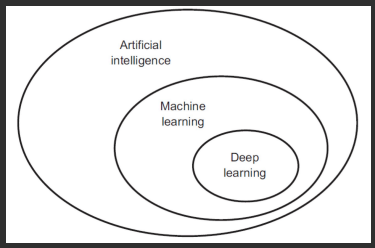
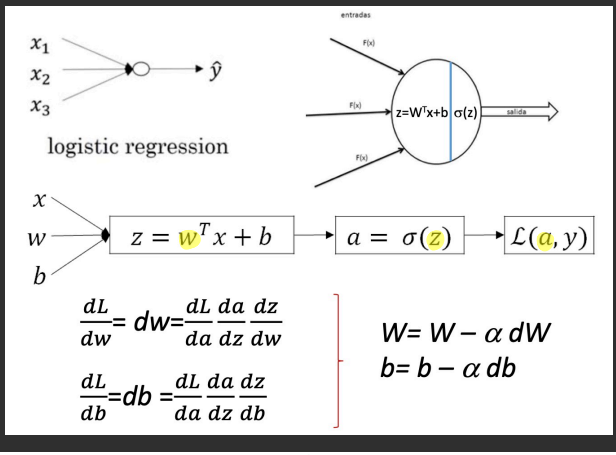
- "Deep" not because it understands deeper, but because the model has more hidden layers
- automatize the "feature engineering"
- it needs powerfull hardware, lots of data



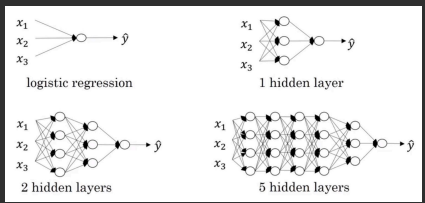


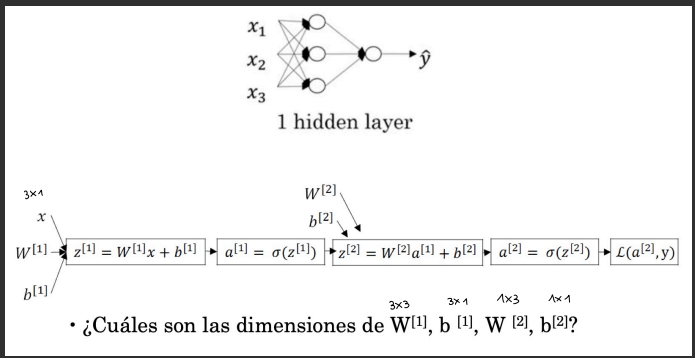
$$\frac{dL}{dw} = dw = \frac{dL}{da}\frac{da}{dz}\frac{dz}{dw}$$

$$\frac{dL}{db} = db = \frac{dL}{da}\frac{da}{dz}\frac{dz}{db}$$

$$W = W - \alpha\, dW$$

$$b = b - \alpha\, db$$





- ¿Cuáles son las dimensiones de W[1], b [1], W [2], b[2]?

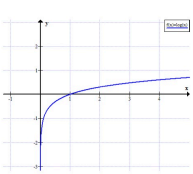
3×3 3×1 1×3 1×1

$(n_l \times n_{l-1})$ without transposing

Loss function logistic classification

$$L(y, \hat{y}) = -(y \log \hat{y} + (1-y) \log (1-\hat{y}))$$

- Si y=1 $L(y, \hat{y}) = -\log \hat{y}$: minimizar $L(y, \hat{y})$ → maximizar $\hat{y}$
- Si y=0 $L(y, \hat{y}) = -\log (1-\hat{y})$: minimizar $L(y, \hat{y})$ → minimizar $\hat{y}$



$y=0 \Rightarrow L(y,\hat{y}) = -\log(1-\hat{y})$

$\hat{y}=0 \Rightarrow L(y,\hat{y}) = 0$

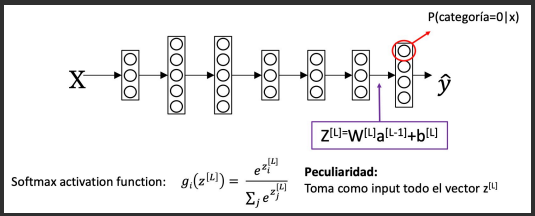
$\hat{y}=1 \Rightarrow L(y,\hat{y}) \approx \infty$

$y=1 \Rightarrow L(y,\hat{y}) = -\log \hat{y}$

$\hat{y}=0 \Rightarrow L(y,\hat{y}) \approx \infty$

$\hat{y}=1 \Rightarrow L(y,\hat{y}) = 0$

Logistic is for binary classification, while softmax for multiple classification.



Generalized Loss function - Cross-Entropy

$$L(y, \hat{y}) = -\sum_j y_j \log \hat{y}_j$$ ← Cross-Entropy

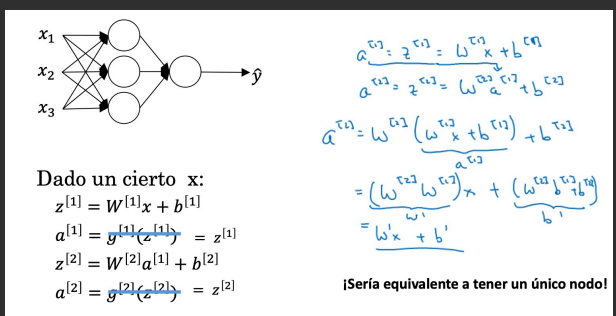
to minimize L we need to maximize the probability of belonging to the correct class.

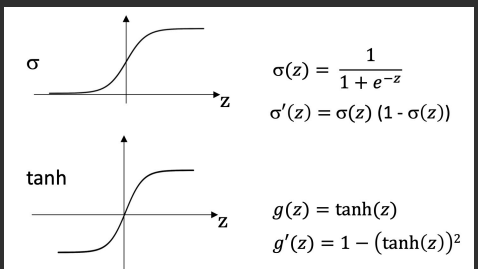
Cost function

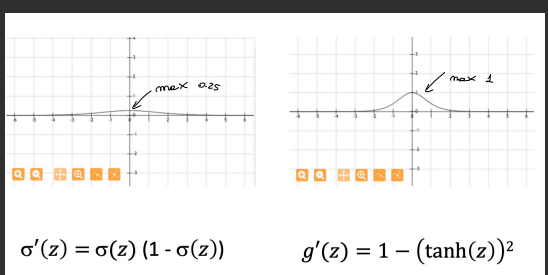
$$\text{Cost} = \frac{1}{N} \sum_{i=1}^{N} L(y_i, \hat{y}_i)$$

Activation function

Without it we could only learn hyperplane decision boundaries.

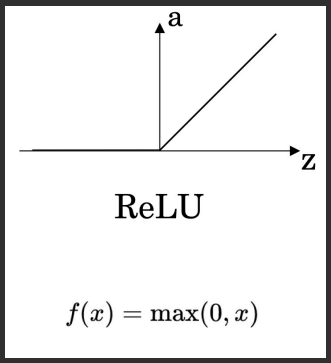






Vanish gradient

- having lots of hidden layers and using sigmoid could bring to not learn anything because $\sigma(x)$ has 0.25 as maximum, so multiplying lots of small numbers let the gradient tends to 0.
- In the hidden layers, sigmoid is not used.

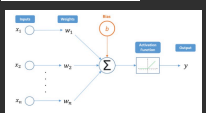


- fast
- no gradient vanishing
- not differentiable in 0. For convention is chosen 0 or 1 in that point.

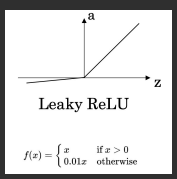
Dying ReLu

- high LR ⟹ weights could become negatives

- high negative bias ⟹ input relu negative



Solve the problem



How to use activation function in deep neural networks

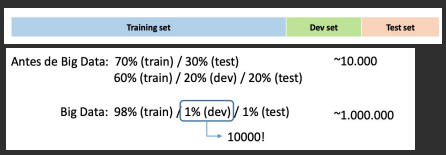
Input layer: tanh, σ

Hidden layers: ReLu, leaky ReLu

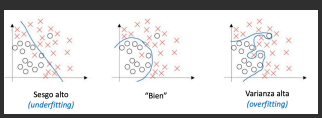
Output layer: binary classification ⟹ tanh, σ

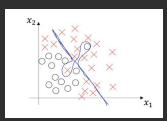
multiclass classification ⟹ softmax

Dataset



Bias \ Variance





high variance
high bias

Error humano ~ 0%					
	Train set error	1%	15%	15%	0.5%
	Dev set error	11%	16%	30%	1%
		overfitting ↑ high variance	underfitting ↑ high bias	overfitting underfitting ↑ high variance high bias	good model ↑ low variance low bias

High Bias?
(training data performance)

yes →
- Larger network
- Train longer
- Search for another NN architecture

no ↓

High Variance?
(dev set performance)

yes →
- More data
- Regularization [redacted]
- Search for another NN architecture

no ↓

Regularization

- If weight are smaller, the network is less specialized

1 L1 Regularization in Neural Networks

- Adds the sum of the absolute values of the weights to the loss function:

$$L = \text{Loss} + \lambda \sum |w_i|$$

- Encourages sparsity by forcing some weights to become exactly **zero**.
- Can help **prune** unnecessary connections in a neural network.

✅ **Effects in NN:**

- Reduces complexity by **eliminating** some weights.
- Leads to a **sparser** network (can be useful for feature selection).
- May cause instability when training deep networks.

2 L2 Regularization in Neural Networks

- Adds the sum of the squared values of the weights to the loss function:

$$L = \text{Loss} + \lambda \sum w_i^2$$

- Forces weights to be **small** but not necessarily zero.
- Helps distribute learning across all neurons.

✅ **Effects in NN:**

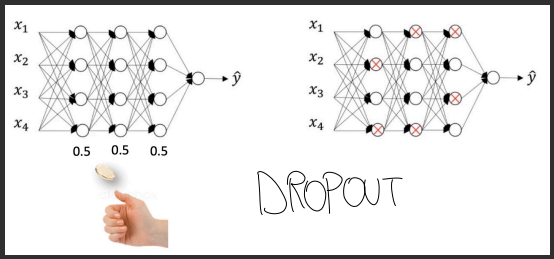
- Prevents overfitting by discouraging large weights.
- Works well in deep networks where all features contribute.
- Doesn't enforce sparsity but smoothens learning.

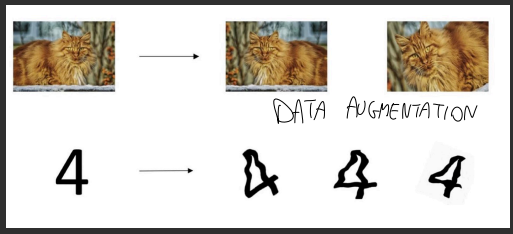
3 Key Differences in Neural Networks

Feature	L1 Regularization (Lasso)	L2 Regularization (Ridge)
Effect on weights	Some weights become **zero** (sparse network)	Weights become **small** but not zero
Sparsity	Yes (reduces unnecessary connections)	No (keeps all connections)
Best use case	When you want to **prune** neurons/connections	When you want to **stabilize** learning
Training effect	Can make training unstable	More stable training

🚀 **Conclusion:**

- L1 **removes** unimportant weights (good for reducing model size).
- L2 **shrinks** weights evenly (good for stabilizing training).
- For deep learning, **L2 (weight decay) is more commonly used**.

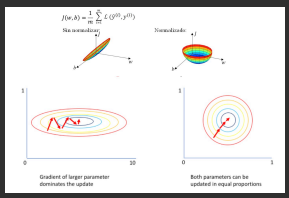




Normalize input

– help the training

$$z = \frac{x - \mu}{\sigma}$$



Weight initialization

- ReLU / Leaky ReLU → He Initialization ✅
- Sigmoid / Tanh → Xavier o Lecun ✅
- Molti layer → Ortogonale ✅
- Bias → Zero ✅

Gradient descent

1. Batch Gradient Descent

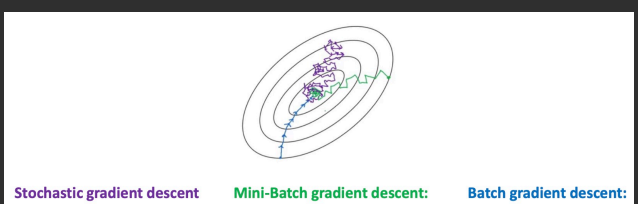
- Updates the model's weights **after processing the entire dataset** (one full epoch).
- **Advantage**: More stable updates since they use the full dataset.
- **Disadvantage**: Computationally expensive and slow for large datasets.
- **Best for**: Small datasets where computational efficiency is not a concern.

2. Stochastic Gradient Descent (SGD)

- Updates the model's weights **after each individual sample**.
- **Advantage**: Very fast updates, does not require processing the entire dataset.
- **Disadvantage**: High variance in updates, may lead to noisy convergence.
- **Possible Solution**: Decrease the learning rate over time using a decay function. $lr = \frac{const\_1}{n_{iteracion} + const\_2}$
- **Best for**: Large datasets where immediate updates are preferred.

3. Mini-Batch Gradient Descent

- Updates the model's weights **after processing a small subset (mini-batch) of the dataset**.
- **Advantage**: Balances speed (like SGD) and stability (like Batch GD).
- **Disadvantage**: Choosing the right batch size is crucial.
- **Best for**: Most deep learning tasks since it leverages parallelization.



Method	Update Frequency	Speed	Stability	Suitability
Batch GD	After full dataset	Slow	Stable	Small datasets
SGD	After each sample	Very fast	Noisy	Large datasets
Mini-Batch GD	After mini-batch	Fast	Balanced	Deep learning

Batch normalization

What is Batch Normalization?

Batch Normalization (BN) is a technique that **makes training faster and more stable** by **normalizing** the values (activations) in a mini-batch.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

MOVE DIVIDE BY 0.

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Momentum / RMSProp / Adam

Optimizer	Key Idea	Strengths ✅	Weaknesses ❌
Momentum	Uses past gradients to accelerate learning	✅ Faster convergence ✅ Reduces oscillations	❌ Struggles with adaptive learning rates
RMSprop	Adapts learning rate for each parameter using a moving average of squared gradients	✅ Good for non-stationary problems ✅ Works well in deep networks	❌ Can become too aggressive in updates
Adam	Combines **Momentum** and **RMSprop**	✅ Best of both worlds ✅ Faster & more stable	❌ Uses more memory

last | 10 | 50 | 2 | _ gradients
B - 0.9 | 0.98 | 0.5

$$v_{dW} = \beta v_{dW} + (1-\beta)dW \qquad W = W - \alpha v_{dW}$$
$$v_{db} = \beta v_{db} + (1-\beta)db \qquad b = b - \alpha v_{db}$$

$$s_{dW} = \beta s_{dW} + (1-\beta)dW^2 \qquad W = W - \alpha \frac{dW}{\sqrt{s_{dW}} + \epsilon}$$
$$s_{db} = \beta s_{db} + (1-\beta)db^2 \qquad b = b - \alpha \frac{db}{\sqrt{s_{db}} + \epsilon}$$

Prevent division by 0

Instead of using the **same learning rate** for all parameters, RMSprop **scales it based on recent gradients**.

- If s_{dW} is **big**, the step is small.
- If s_{dW} is **small**, the step is bigger.

Why Do We Need RMSprop?

Imagine you're walking down a mountain (minimizing loss), but the path is very uneven:

- Some slopes are **steep** (big gradients 🔺).
- Some parts are **flat** (small gradients ⬜).
- If you take **big steps**, you might fall (unstable training).
- If you take **small steps**, it will take forever to reach the bottom.

👉 **RMSprop helps by adjusting your step size dynamically** so that:

- In **steep areas**, you take **smaller steps** (to avoid overshooting).
- In **flat areas**, you take **bigger steps** (to speed up learning).

$$v_{dW} = \beta_1 v_{dW} + (1-\beta_1)dW$$
$$v_{db} = \beta_1 v_{db} + (1-\beta_1)db$$

Momentum

$$s_{dW} = \beta_2 s_{dW} + (1-\beta_2)dW^2$$
$$s_{db} = \beta_2 s_{db} + (1-\beta_2)db^2$$

RMSProp

$$W = W - \alpha \frac{v_{dW}}{\sqrt{s_{dW}} + \epsilon}$$
$$b = b - \alpha \frac{v_{db}}{\sqrt{s_{db}} + \epsilon}$$

🔷 The Problem: Walking Down a Mountain

1. **The terrain is rough** – Some areas are **steep**, some are **flat**, and some have **sharp drops**.
2. **If you take big steps, you might trip** – Standard Gradient Descent can overshoot the minimum.
3. **If you take small steps, it will take forever** – RMSprop can slow down too much in flat areas.
4. **You also don't want to zig-zag too much** – Momentum helps smooth your path.