business innovation through algorithms

Neural Networks

CGnal s.r.l. – Corso Venezia 43 - Milano

19 novembre 2021| Milano

DAY 1

Introduction

- Brief overview of Machine Learning (Supervised, Unsupervised)
- Introduction to Graph, Graph Theory and main metrics for characterizing graphs

DAY 2

Graph Machine Learning

- Community detection on Graphs
- Supervised Machine Learning on Graphs

DAY 3

Explainability & Interpretability

- Introduction to explainability problem
- LIME & SHAP

DAY 4

Simple Neural Networks

- Introduction to Neural Networks, TensorFlow and Computational Graphs
- Implementation and training of simple Neural Networks

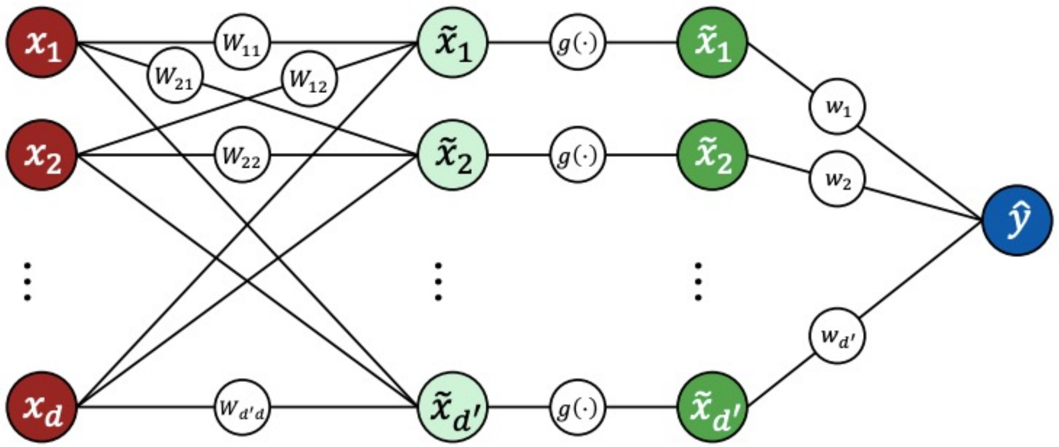
DAY 5

Advanced Neural Networks

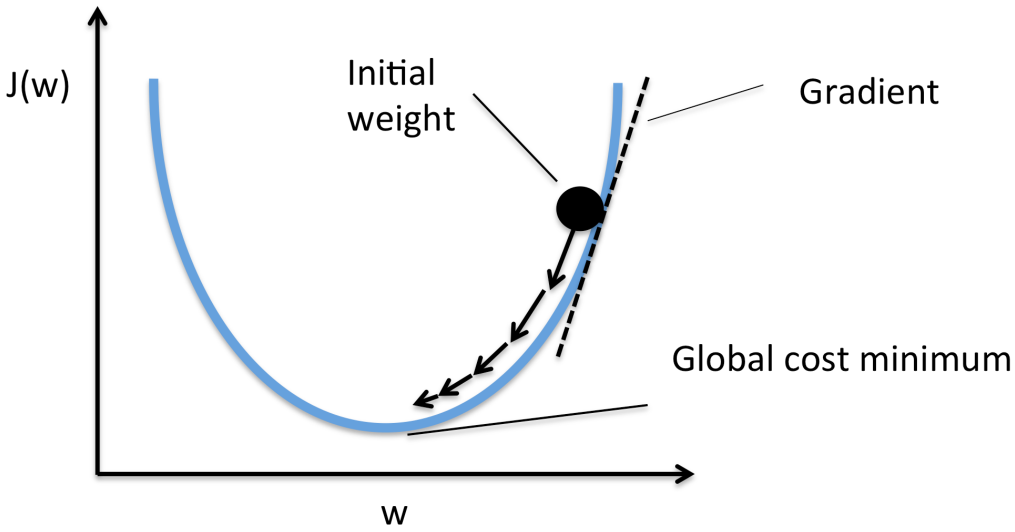
- Convolutional Neural Networks and Recurrent Neural Networks
- Advanced Topics

Let's go back to our training problem…

Neural Network Training



Gradient Descend

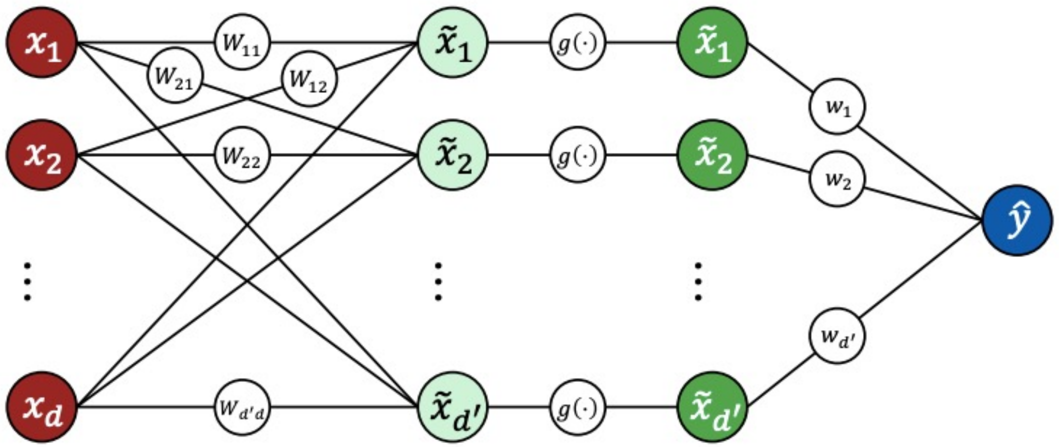


We want to find the weights which minimize the Loss function:

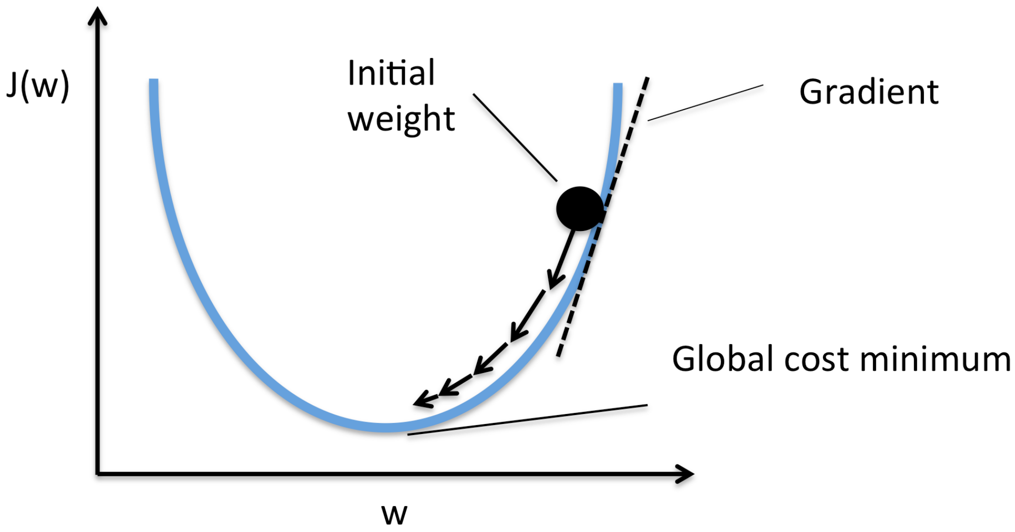
$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma_n \nabla L(\mathbf{w}_n)$$

Let's go back to our training problem…

Neural Network Training



Gradient Descend



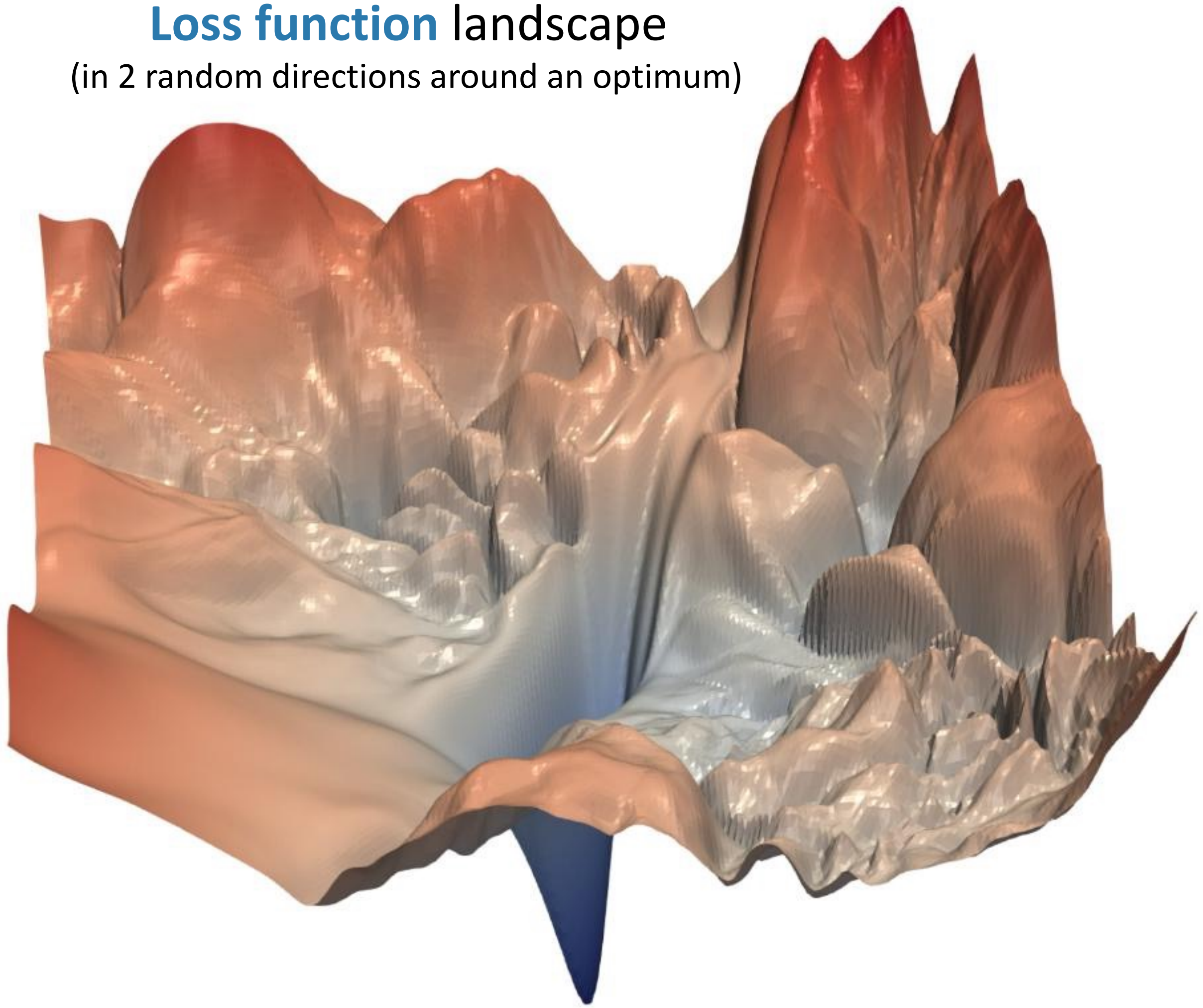
We want to find the weights which minimize the Loss function: $\mathbf{w}_{\mathrm{n+1}} = \mathbf{w}_{\mathrm{n}} - \gamma_{\mathrm{n}} \nabla \mathrm{L}(\mathbf{w}_{\mathrm{n}})$

Things are not so easy…

CGnal

How to optimize such functions?

Loss function landscape
(in 2 random directions around an optimum)



https://papers.nips.cc/paper/7875-visualizing-the-loss-landscape-of-neural-nets

The **loss function** of a neural network is **not a convex function**: huge number of local minima

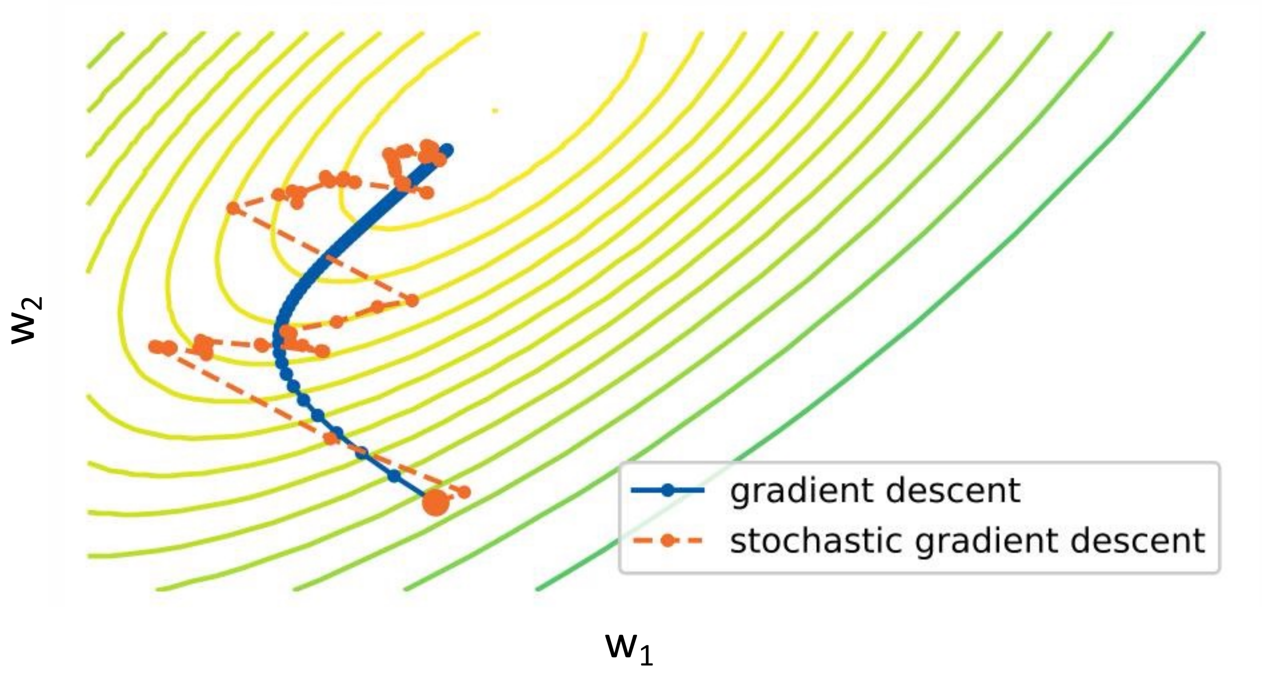
Calculating the **gradient** is **computationally expensive** (sometimes unfeasible) for complex networks and/or large datasets

CGnal

Optimization techniques

Stochastic Gradient Descent (SGD)

At each step approximate the gradient using a single example or a n < N examples (SGD on mini-batches) -> computationally much faster



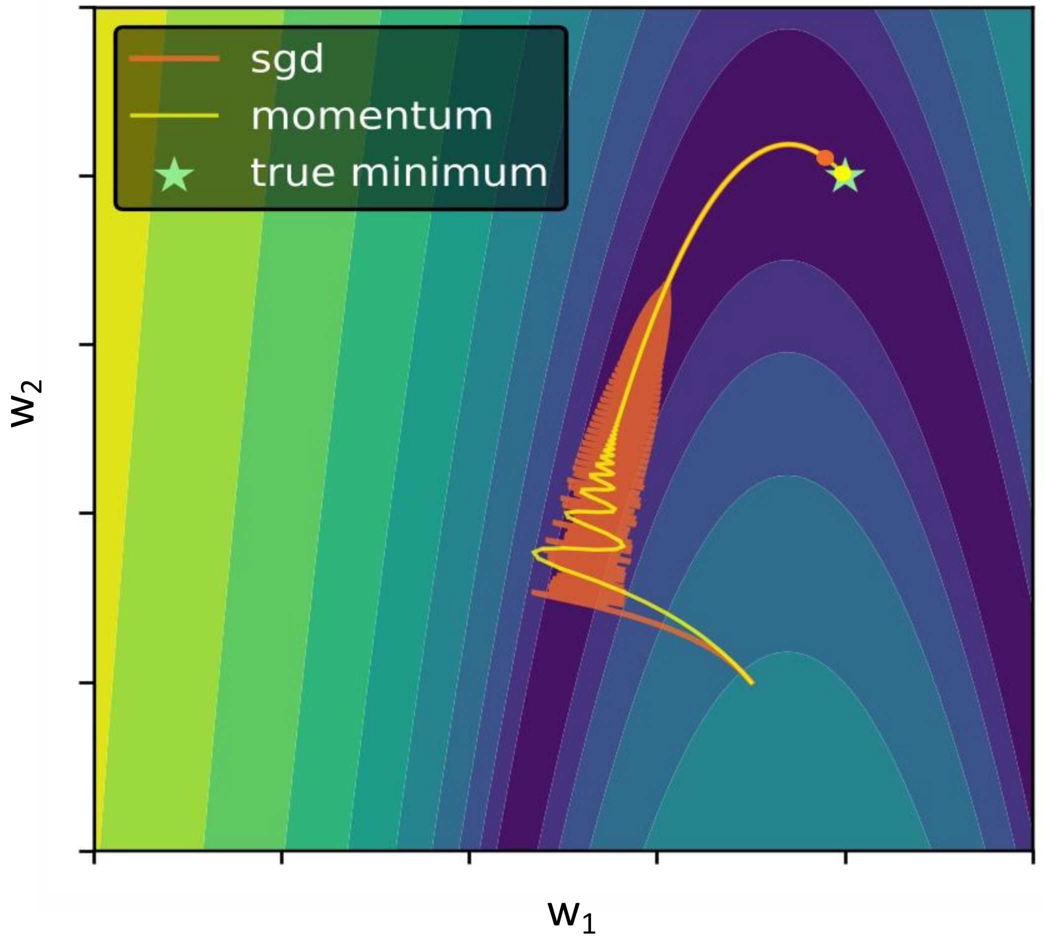
Optimization techniques

Stochastic Gradient Descent (SGD)
At each step approximate the gradient using a single example or a n < N examples (SGD on mini-batches) -> computationally much faster

Momentum SGD
Introduce "inertia": at each step use a linear combination of the previous weight update with the new gradient -> avoids fast oscillations

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \alpha\delta\mathbf{w}_n - \gamma_n \nabla J(\mathbf{w}_n)$$

previous weight update
$\alpha \in [0, 1]$



CGnal

Optimization techniques

Stochastic Gradient Descent (SGD)

At each step approximate the gradient using a single example or a n < N examples (SGD on mini-batches) -> computationally much faster

Momentum SGD

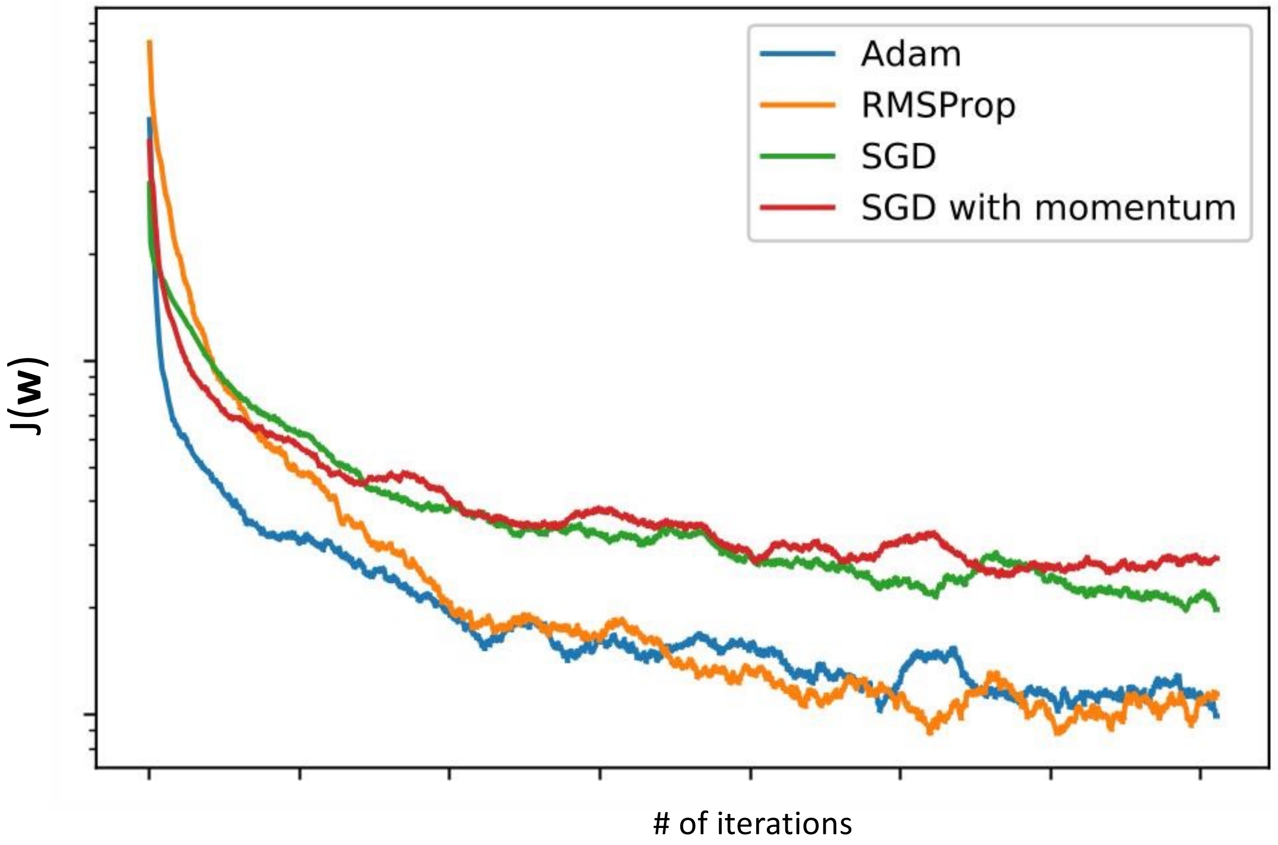
Introduce "inertia": at each step use a linear combination of the previous weight update with the new gradient -> avoids fast oscillations

RMSprop

Learning rate is divided by the average magnitude of the gradient in recent steps -> avoids plateaus with vanishing gradients

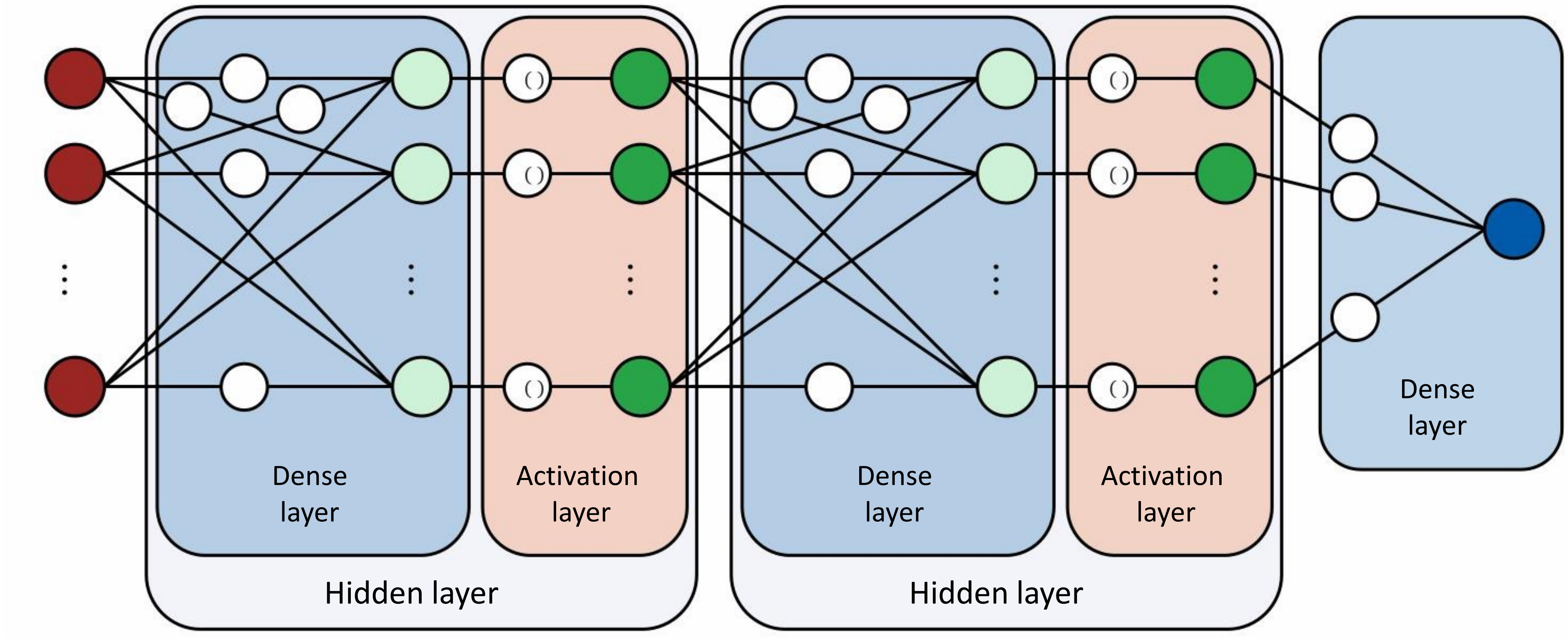
Adam

Combine Momentum SGD + RMSprop



CGnal

Weight initialization



Gradient descent requires a **starting point in the weight space**

What happens if we initialize all weights **with the same value**?

Within each layer, the gradients for each of the weights will be the same as well ⇒ **updates will be the same** ⇒ **network degrades!**

Symmetry breaking with random initialization

Not all random initizializations are good choices

Symmetry breaking with random initialization

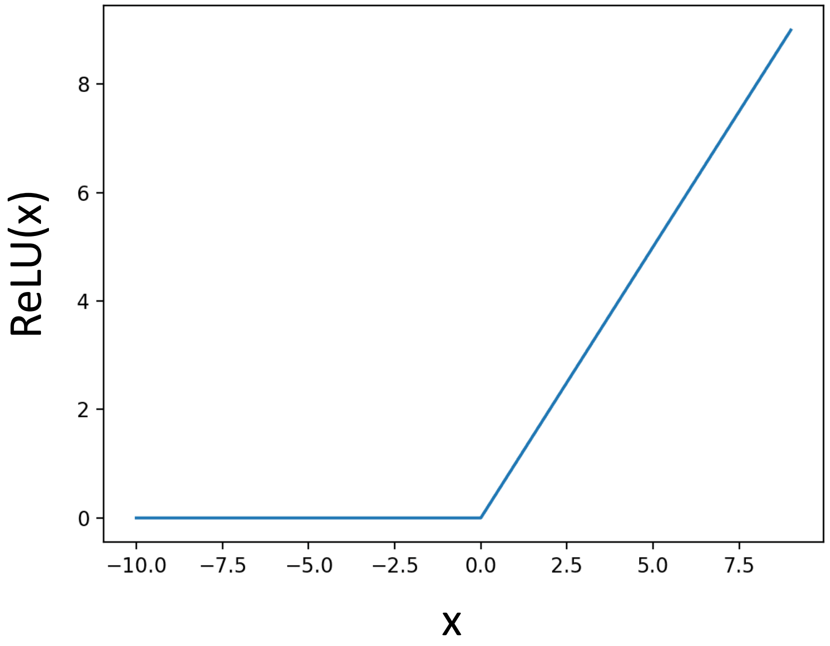
Not all random initizializations are good choices

Some intuition

(for ReLU activation, but similar arguments hold for other activation functions)

Large positive values → large output values in the forward pass → multiply partial derivatives in the backward pass
→ **exploding gradients**

Large negative values → partial derivatives are zero
→ **vanishing gradients**



See animations:
https://www.deeplearning.ai/ai-notes/initialization/index.html

CGnal

Weight random initializers

Historically, **random initialization** from a uniform or normal distribution in a small range around 0, e.g. [-1, 1]

In the last decade, the topic has been studied in detail

n = number of input neurons
m = number of output neurons

Xavier/Glorot initialization: $w = Uniform\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$

Normalized Xavier/Glorot initialization: $w = Uniform\left[-\sqrt{\frac{6}{n+m}}, \sqrt{\frac{6}{n+m}}\right]$

Sigmoid or tanh activation

He initialization: $w = Gaussian\left[0, \sqrt{\frac{2}{n}}\right]$ → ReLU activation

De facto standard, available ready-to-use in TF

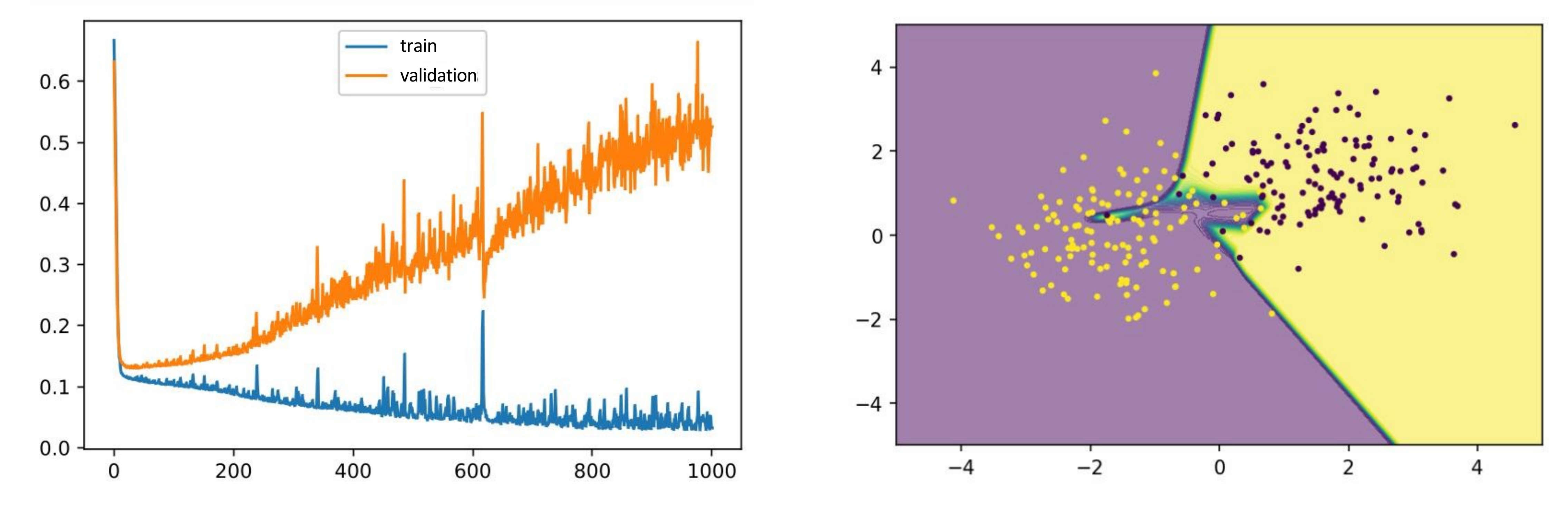
```
tf.keras.initializers.HeNormal
tf.keras.initializers.GlorotUniform
```

Xavier Glorot, Yoshua Bengio, Understanding the difficulty of training deep feedforward neural networks (2010)

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification (2015)

The problem of overfitting

Being highly complex models, neural networks are prone to overfitting



Regularization techniques like **L1/L2 regularization** used for linear models are also possible choices for neural networks

Another possibility is **early stopping**, i.e. stop the training before validation error grows

CGnal

Dropout

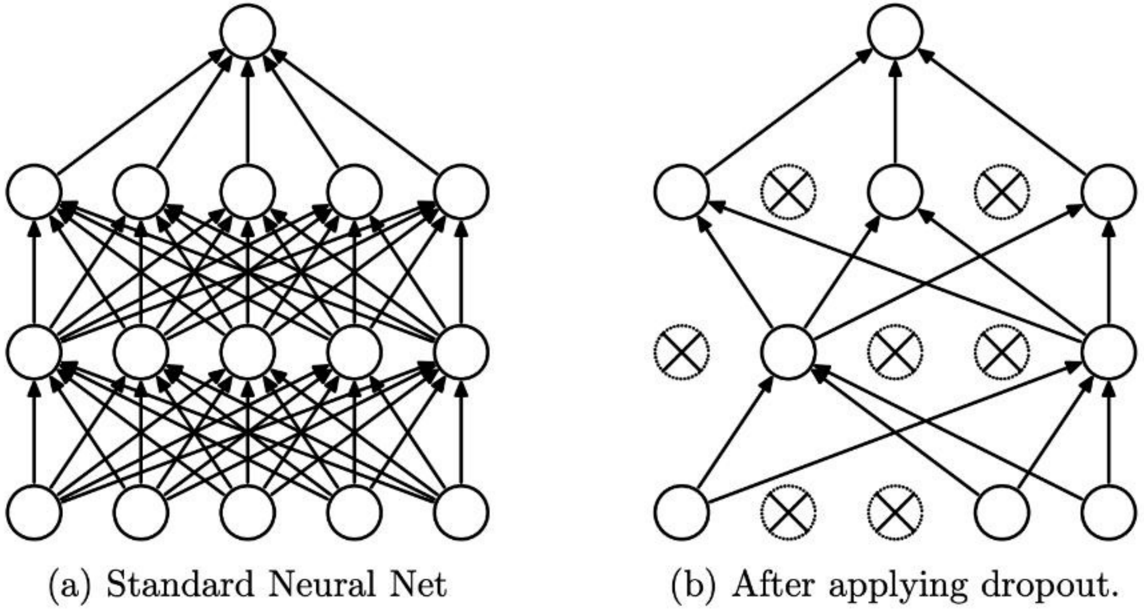
At train time – sets neuron activations to 0 with a given probability *p*

At test time – multiplies the activation of all neurons by *p*

- i.e. sets it to the expected value

Makes neuron learn to work with a randomly chosen sample of other neurons

Image from:
http://jmlr.org/papers/v15/srivastava14a.html



Drives it towards **creating useful features** rather than relying on other neurons to correct its mistakes

Batch normalization

This technique was originally proposed to mitigate the *internal covariate shift*

- updates in one layer change the input distributions of subsequent layers

Normalize the activations subtracting the mean and dividing by the variance

- use mini-batch statistics
- this may reduce significantly the *representation power* of the network

$$\mu_B = \frac{1}{|B|} \sum_{i \in B} x_i \qquad \sigma_B^2 = \frac{1}{|B|} \sum_{i \in B} (x_i - \mu_B)^2$$

→ Add two trainable parameters γ, β that make sure that the BN transformation can represent the identity transformation, thus restoring the full representation power of the original network without BN

$$\hat{x}_i = \gamma \cdot \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

Example: image classification with Inception

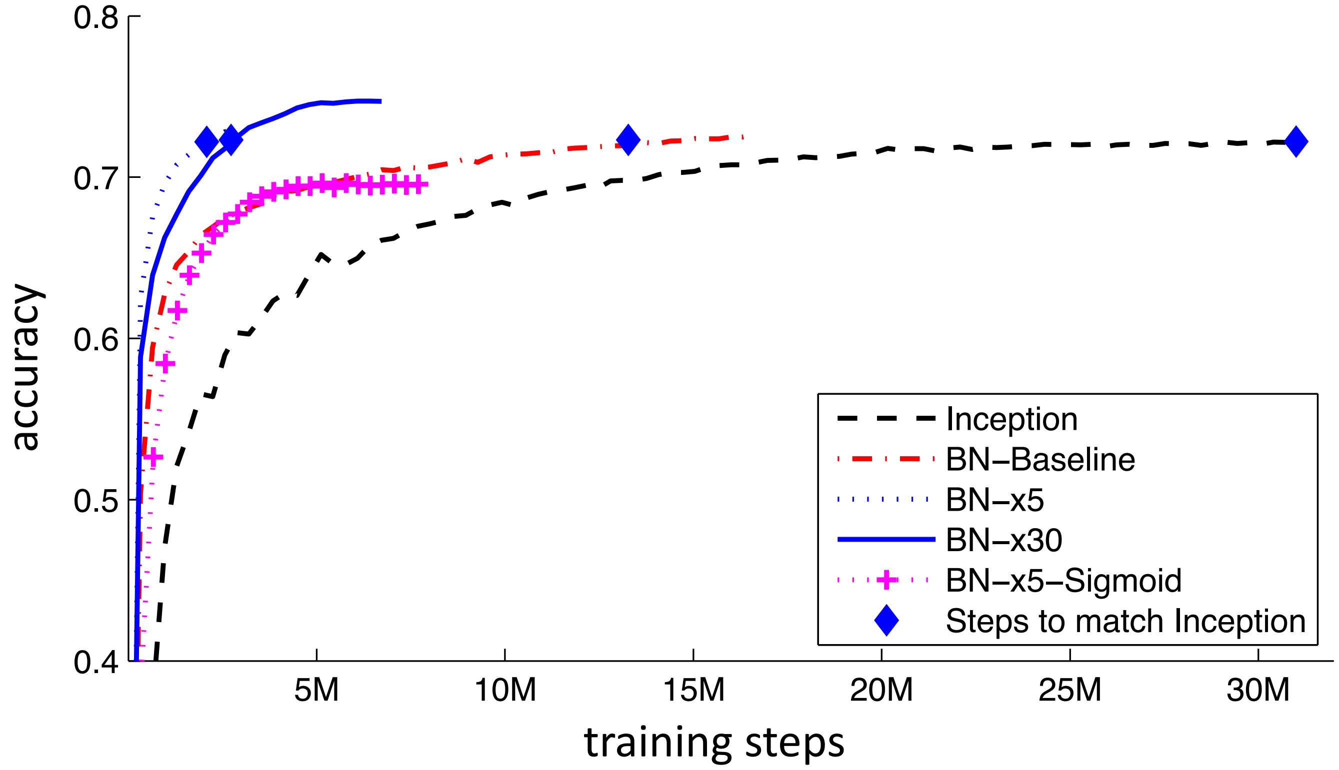


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Very powerful in several applications, particularly image recognition

Later, it was shown to **not** reduce the *internal covariate shift*

Not clear why it works so well...

S. Ioffe, C. Szegedy, [Batch normalization: accelerating deep network training by reducing internal covariate shift](), ICML'15: Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 July 2015 Pages 448–456

Summary

Stochastic Gradient Descent with its variants is used for training neural networks

If done wrong, weight initialization may cause the gradients to vanish or explode

Neural networks can be regularized with **L1/L2** penalties or **early stopping**

Dropout makes neurons create useful features rather than rely on other neurons to correct their mistakes

Batch normalization is extremely powerful for both regularizing the network and speed up training

Hands on

Exercise 1
Multi-Layer Perceptron

Exercise 2
Autoencoder