



PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA

PUCRS

MALTA

Machine Learning Theory
and Applications Lab

Aprendizado de Máquina

Paradigma baseado em Otimização
Redes Neurais III

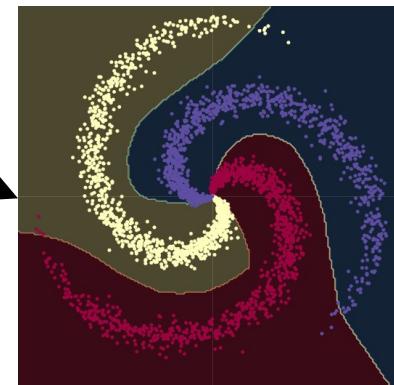
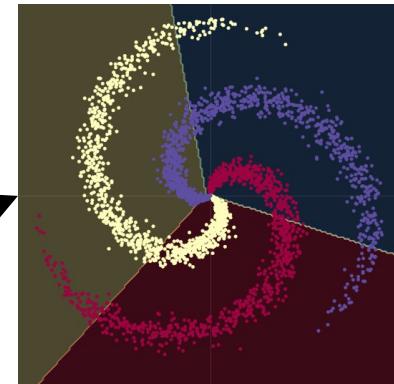
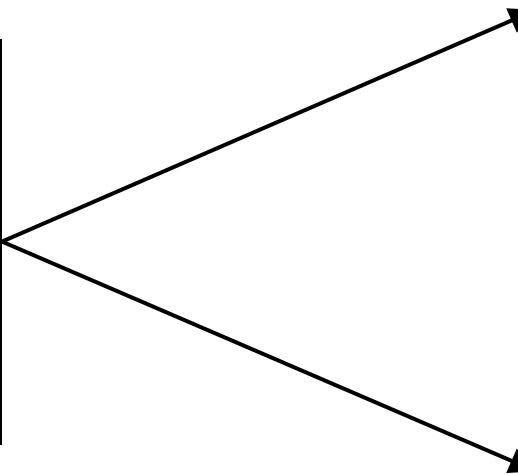
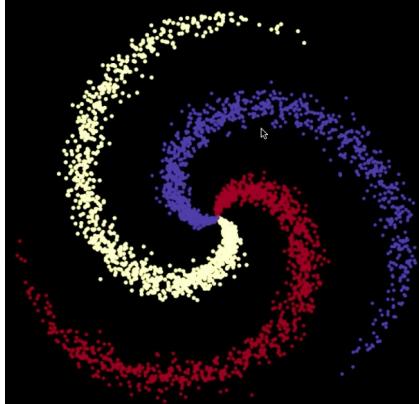
Prof. Me. Otávio Parraga

Hiperparâmetros X Parâmetros

- Parâmetros são aqueles pesos que **aprendemos** ao longo do treinamento
- Hiperparâmetros são as **configurações** que definimos antes de treinar
- **Redes neurais** possuem muitos **hiperparâmetros** para configurarmos

Hiperparâmetros

- Uma mesma rede, hiperparâmetros diferentes



Hiperparâmetros

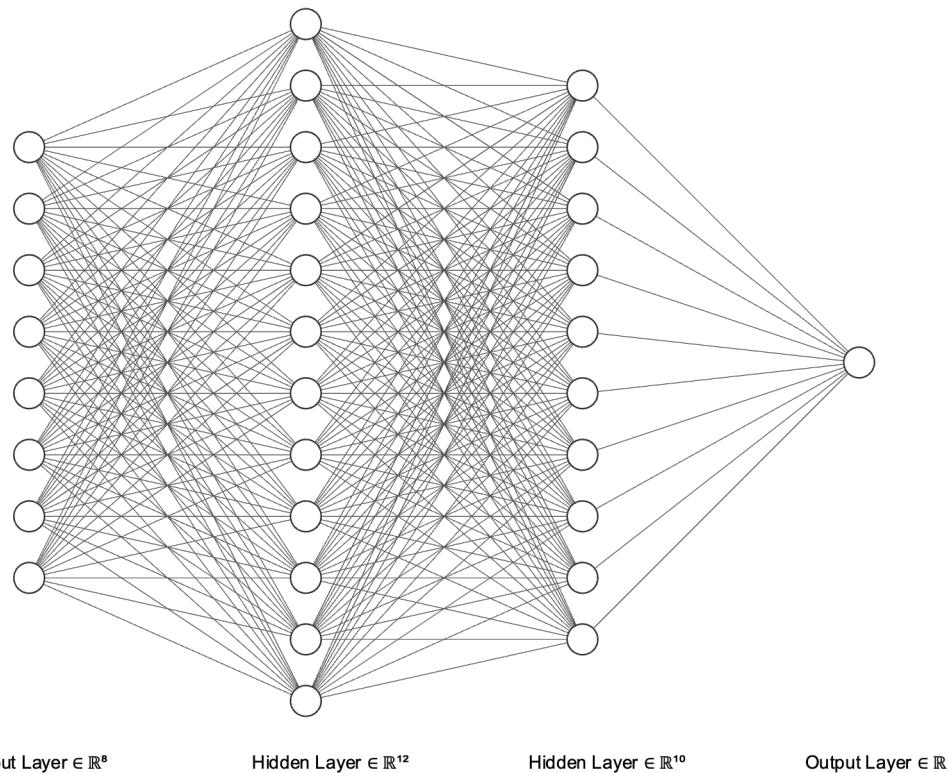
- Architecture
- Loss Functions
- Optimizers
- Activation Functions
- Nº of Neurons
- Regularization
- Early Stop
- Data Augmentation

Hiperparâmetros

- Architecture
- Loss Functions
- Optimizers
- Activation Functions
- Nº of Neurons
- Regularization
- Early Stop
- Data Augmentation

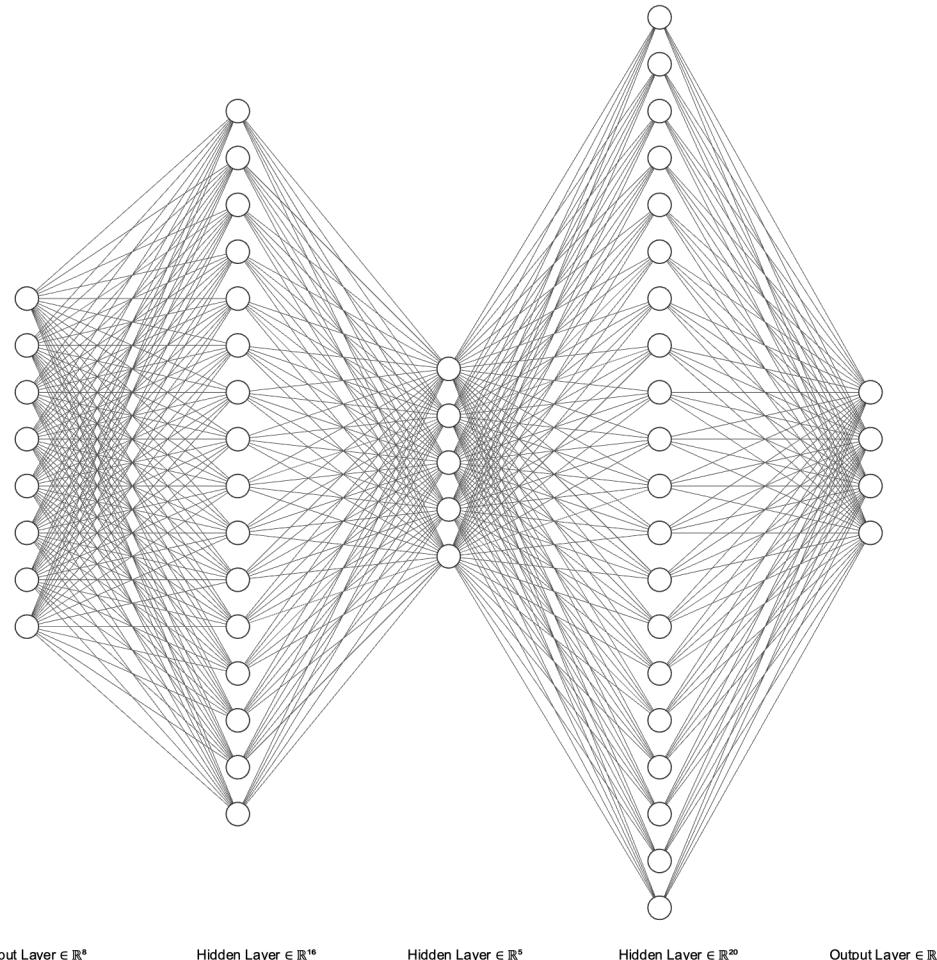
Architecture

- Vimos uma rede **totalmente conectada** (feed forward)



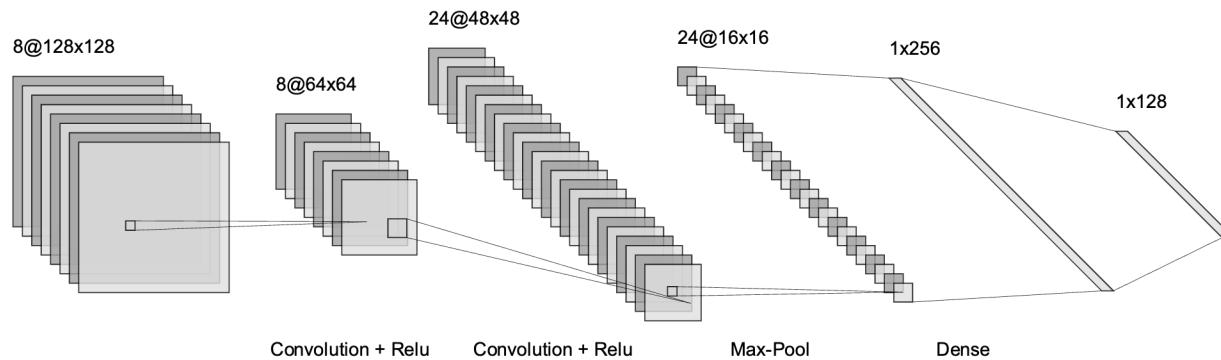
Architecture

- Podemos mudar quantidades de **neurônios** e **camadas**



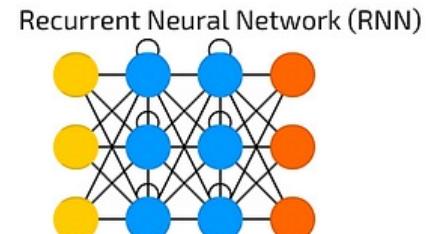
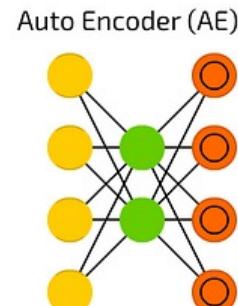
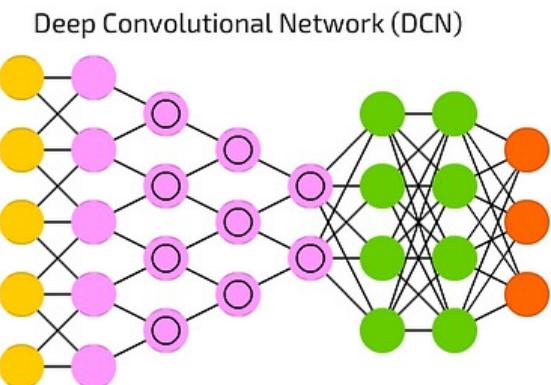
Architecture

- Assim como os **tipos de camadas** (redes convolucionais, por exemplo)



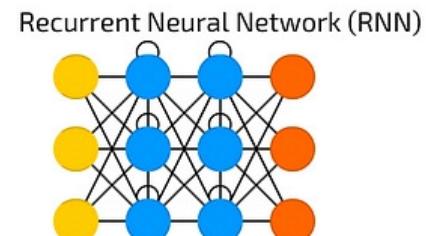
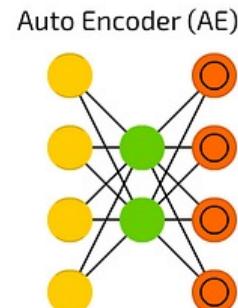
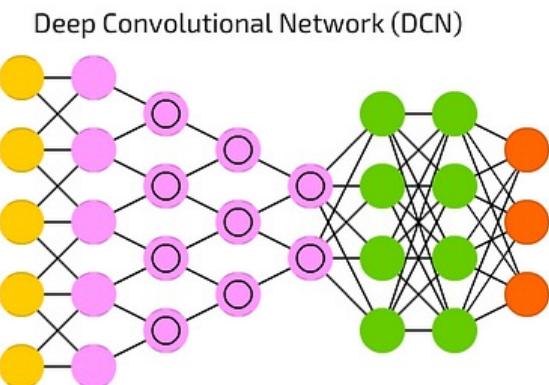
Architecture

- A composição de **tipos e quantidades** de camadas e neurônios chamamos de **arquitetura da rede**
- Variam de acordo com o objetivo



Architecture

- Redes Convolucionais -> Tarefas com **Imagen**
- Auto-Encoders -> Geração de **Representações**
- Redes Recorrentes -> Tarefas com **Séries Temporais**

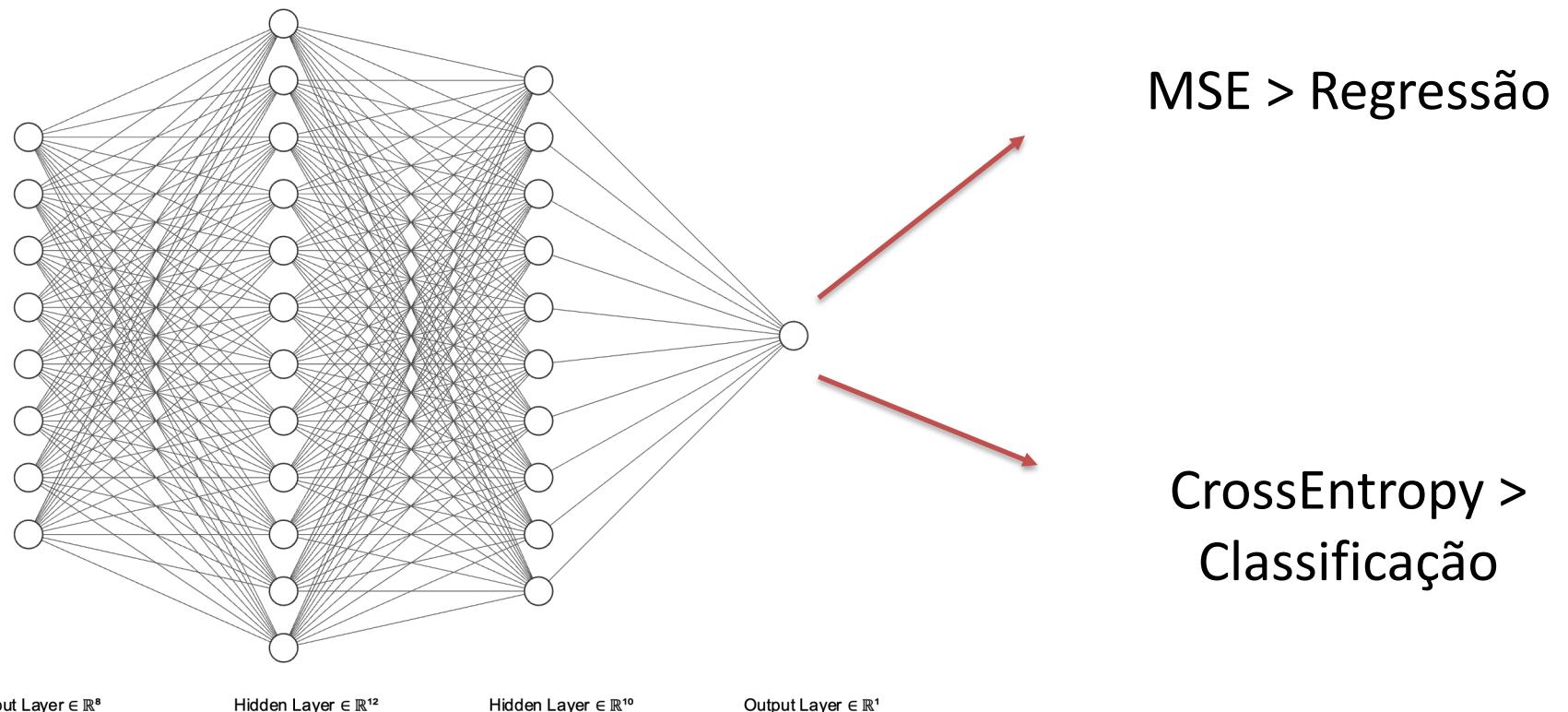


Hiperparâmetros

- Architecture
- Loss Functions
- Optimizers
- Activation Functions
- Nº of Neurons
- Regularization
- Early Stop
- Data Augmentation

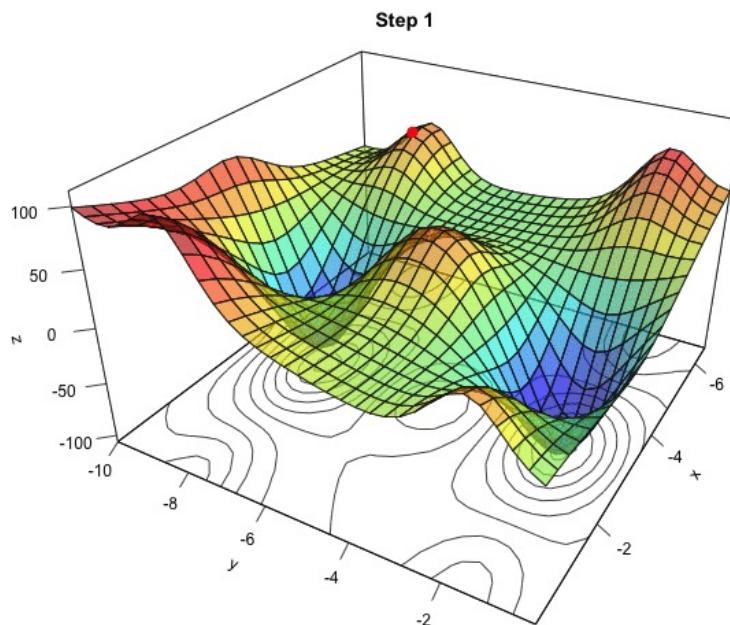
Loss Functions

- Definem o objetivo que será aprendido



Loss Functions

- Podem aplicar restrições durante o treinamento



Loss Functions

`nn.L1Loss`

Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y .

`nn.MSELoss`

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y .

`nn.CrossEntropyLoss`

This criterion computes the cross entropy loss between input logits and target.

`nn.CTCLoss`

The Connectionist Temporal Classification loss.

`nn.NLLLoss`

The negative log likelihood loss.

`nn.PoissonNLLLoss`

Negative log likelihood loss with Poisson distribution of target.

`nn.GaussianNLLLoss`

Gaussian negative log likelihood loss.

`nn.KLDivLoss`

The Kullback-Leibler divergence loss.

`nn.BCELoss`

Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:

Hiperparâmetros

- Architecture
- Loss Functions
- Optimizers
- Activation Functions
- Nº of Neurons
- Regularization
- Early Stop
- Data Augmentation

Optimizers

- Vimos uma regra de **atualização dos pesos**:

$$w_t = w_{t-1} - \alpha \nabla_w Loss$$

- Mas existem várias outras
- Principais:

- SGD + Momentum

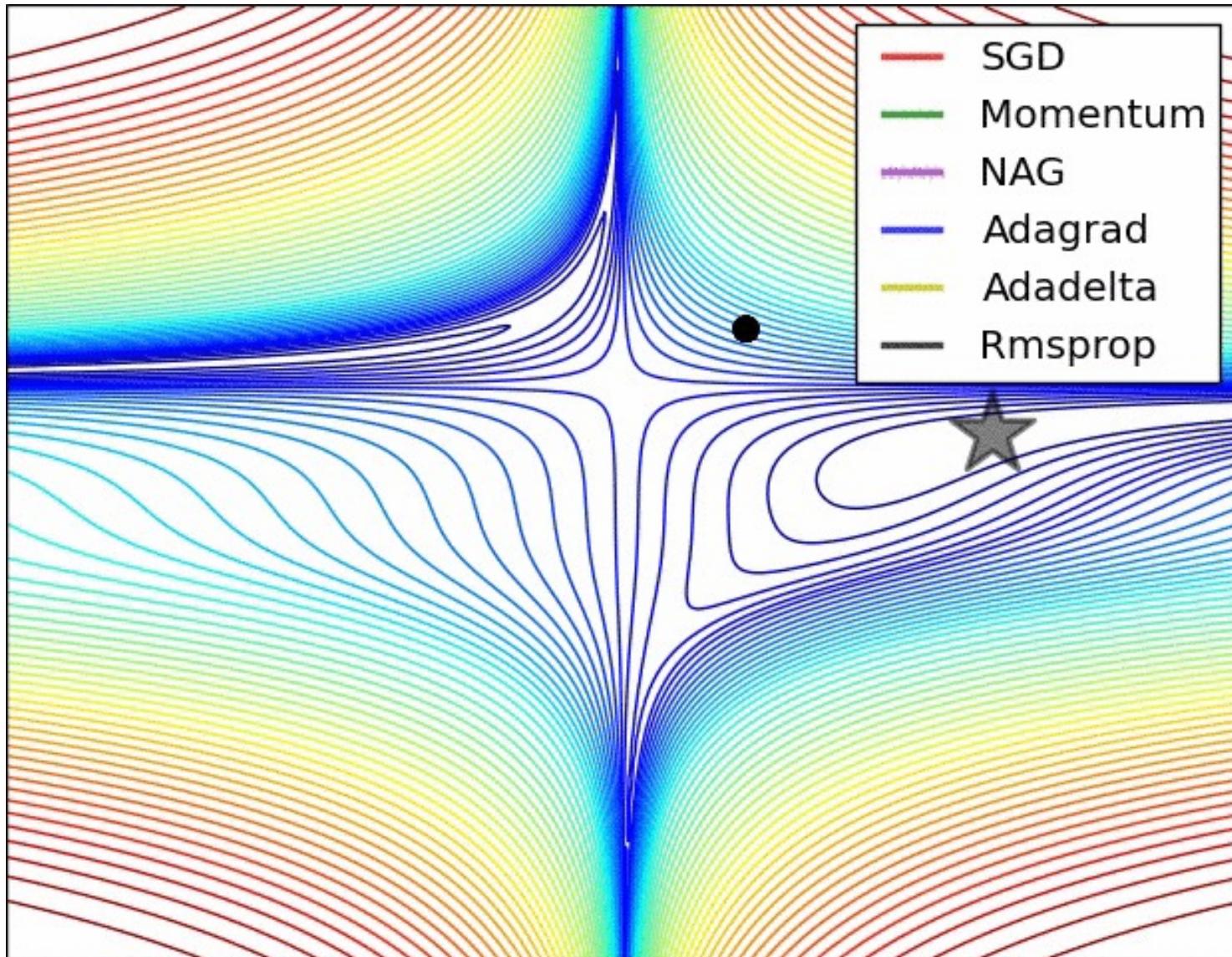
$$v_{dw} = \beta v_{dw} + (1 - \beta) \nabla_w Loss$$

- Adam

$$w_t = w_{t-1} - \alpha v_{dw}$$

- E demais variações

Optimizers

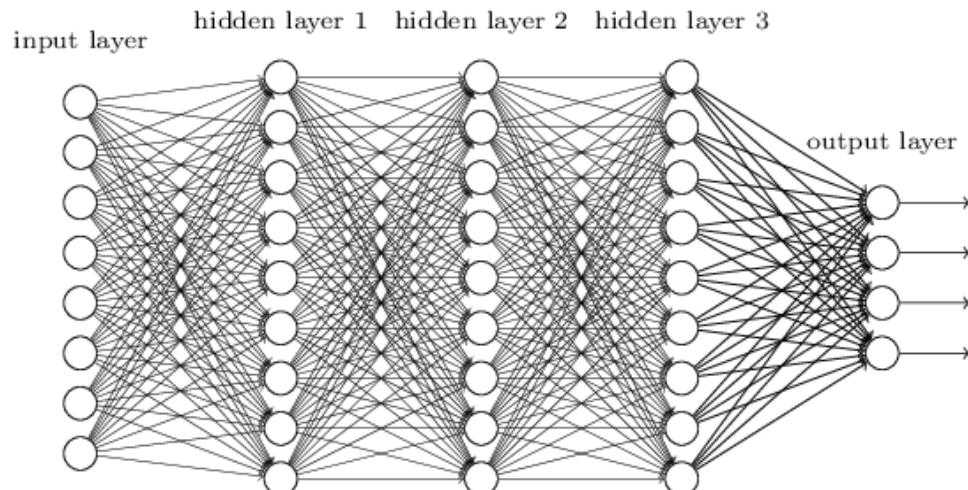


Hiperparâmetros

- Architecture
- Loss Functions
- Optimizers
- Activation Functions
- Nº of Neurons
- Regularization
- Early Stop
- Data Augmentation

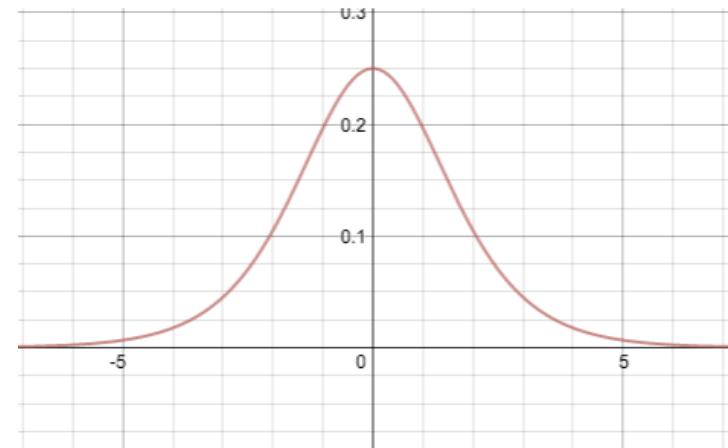
Activation Functions

- O problema do gradiente que desaparece
(Vanishing Gradient)



Gradientes ficam cada vez menores,
fazendo com que camadas mais rasas não
sejam sequer atualizadas!!

Derivada da Síntese



Activation Functions

- Solução: outras funções de ativação
- Na prática, optamos por **ReLU** ou derivadas

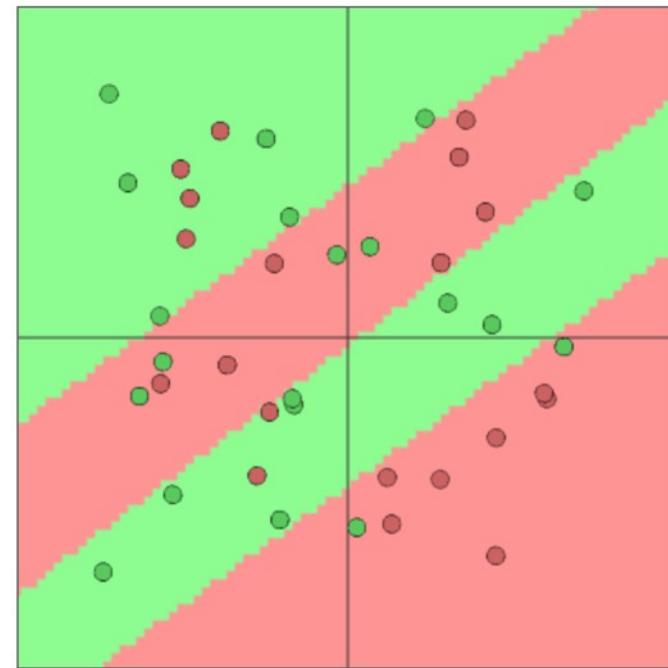
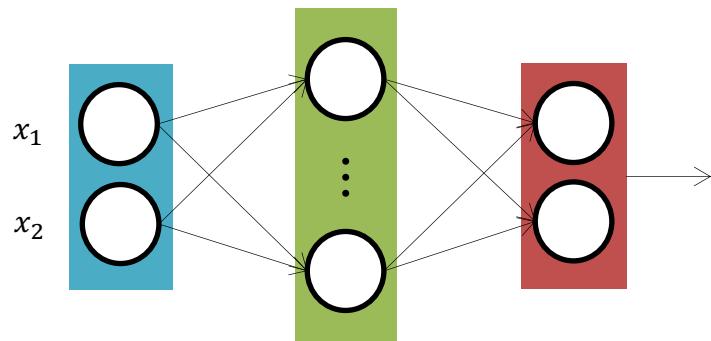
Identity	Sigmoid	TanH	ArcTan
ReLU	Leaky ReLU	Randomized ReLU	Parameteric ReLU
Binary	Exponential Linear Unit	Soft Sign	Inverse Square Root Unit (ISRU)
Inverse Square Root Linear	Square Non-Linearity	Bipolar ReLU	Soft Plus

Hiperparâmetros

- Architecture
- Loss Functions
- Optimizers
- Activation Functions
- Nº of Neurons
- Regularization
- Early Stop
- Data Augmentation

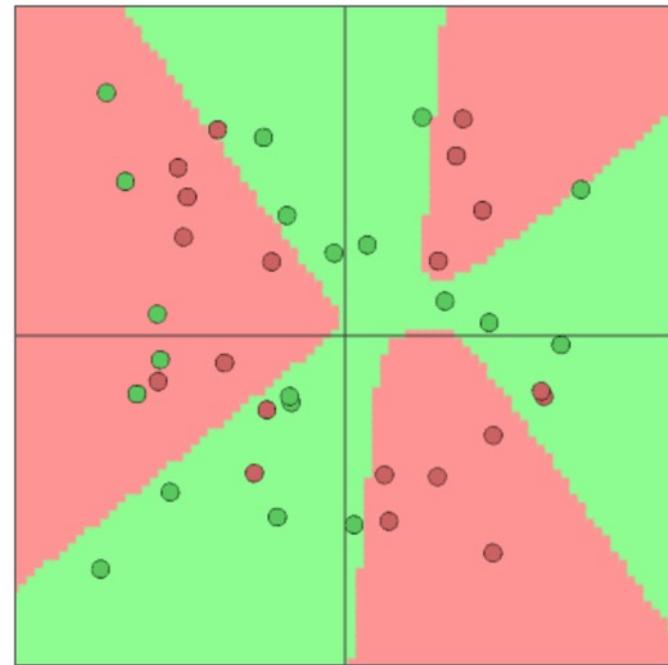
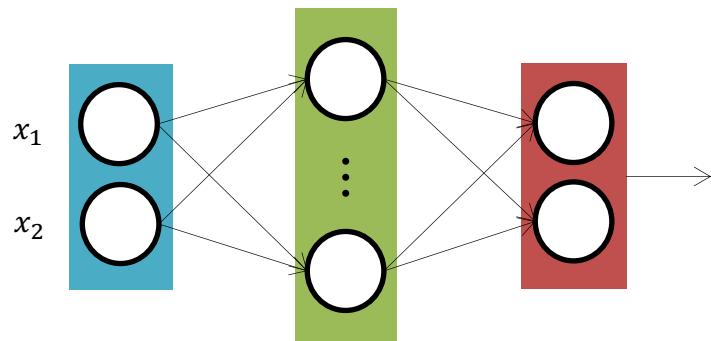
Nº of Neurons

- 2 neurônios na camada escondida



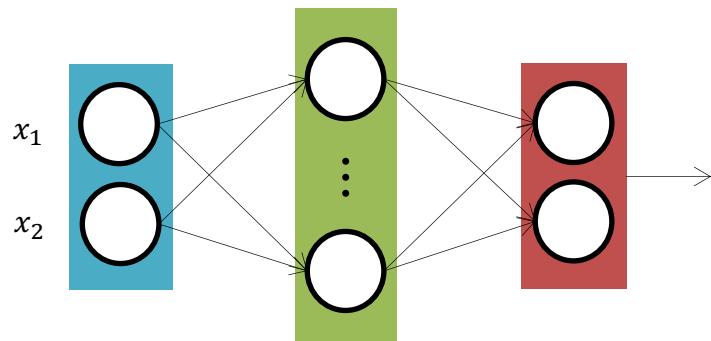
Nº of Neurons

- 4 neurônios na camada escondida



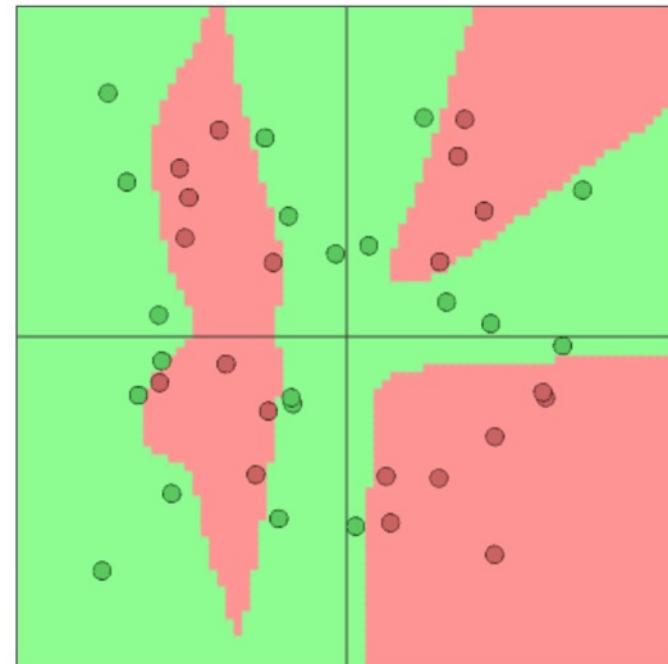
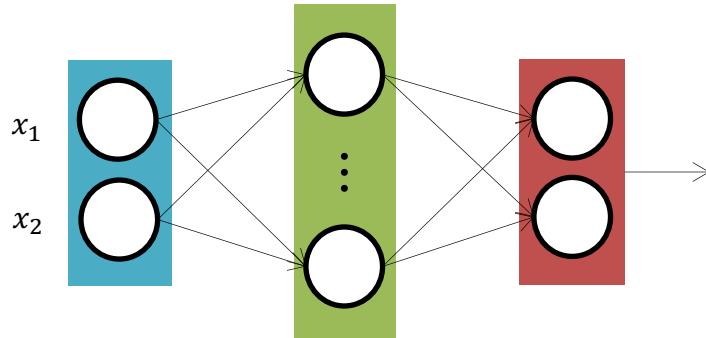
Nº of Neurons

- 8 neurônios na camada escondida



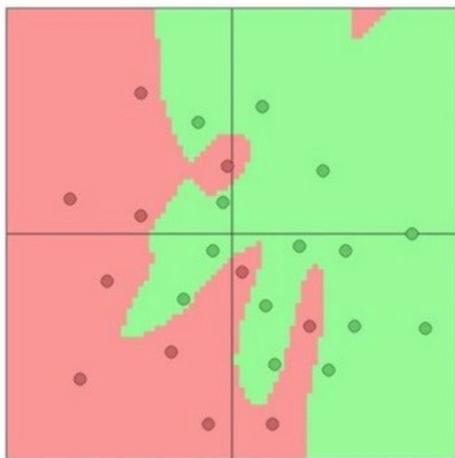
Nº of Neurons

- 128 neurônios na camada escondida
- + neurônios -> + capacidade da rede
- Logo: mais chance de *overfitting*

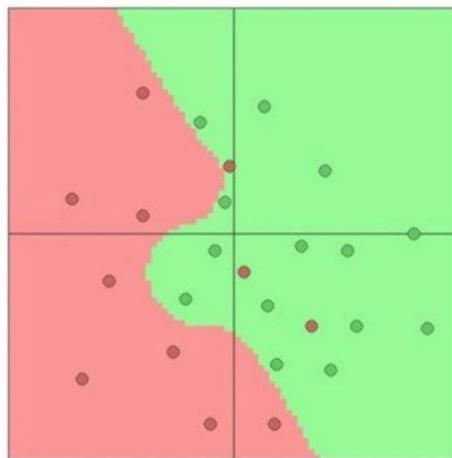


Nº of Neurons

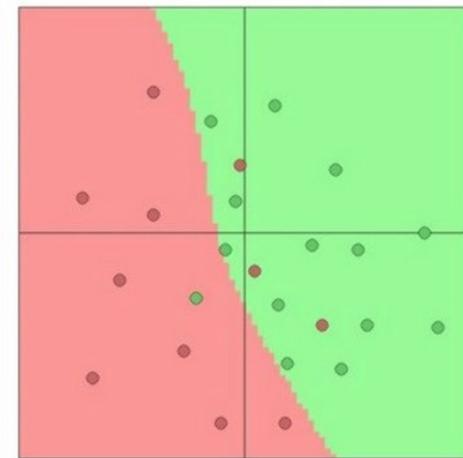
- Número de neurônios não deve ser usado para evitar *overfitting*
- Focar em regularizar uma rede grande!



$$\lambda = 0.001$$



$$\lambda = 0.01$$



$$\lambda = 0.1$$

Hiperparâmetros

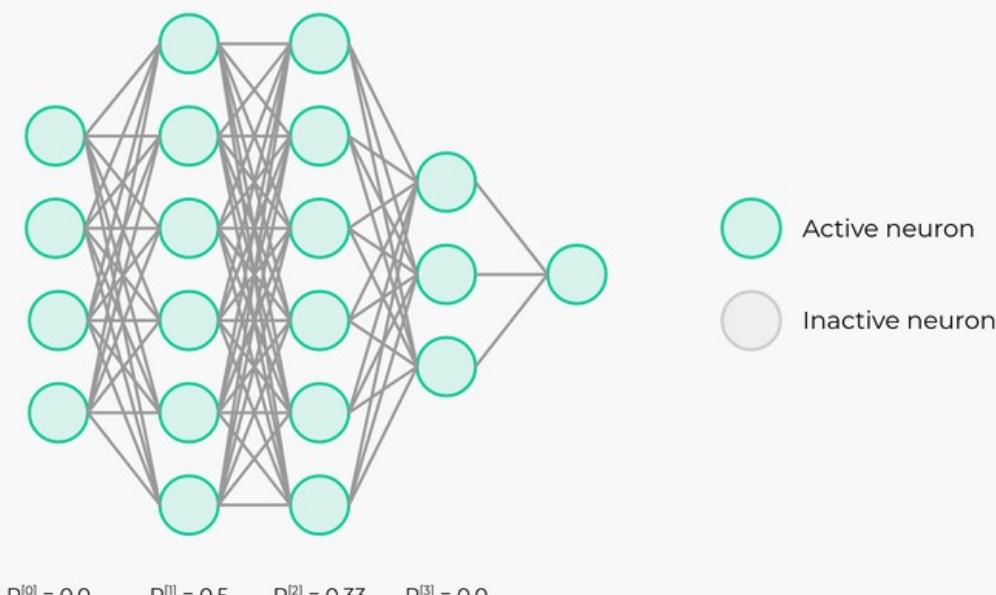
- Architecture
- Loss Functions
- Optimizers
- Activation Functions
- Nº of Neurons
- **Regularization**
- Early Stop
- Data Augmentation

Regularization

- Vimos duas formas de aplicar regularização
 - L1 (Lasso): $J(w) = \sum_{i=1}^N \left(\hat{f}(x^{(i)}) - f(x^{(i)}) \right)^2 + \lambda \sum_{j=0}^m |w_j|$
 - L2 (Ridge): $J(w) = \sum_{i=1}^N \left(\hat{f}(x^{(i)}) - f(x^{(i)}) \right)^2 + \lambda \sum_{j=0}^m w_j^2$
- Funcionam do mesmo jeito para redes neurais!

Regularization

- Dropout



```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            # input = (256, 256, 3)
            nn.Flatten(), # (196608)
            nn.Linear(256*256*3, 1024),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.3, inplace=True),
            nn.Linear(1024, 64),
            nn.ReLU(inplace=True),
            nn.Linear(64, 2)
        )

    def forward(self, x):
        x = self.layers(x)
        return x
```

Hiperparâmetros

- Architecture
- Loss Functions
- Optimizers
- Activation Functions
- Nº of Neurons
- Regularization
- Early Stop
- Data Augmentation

Early Stop

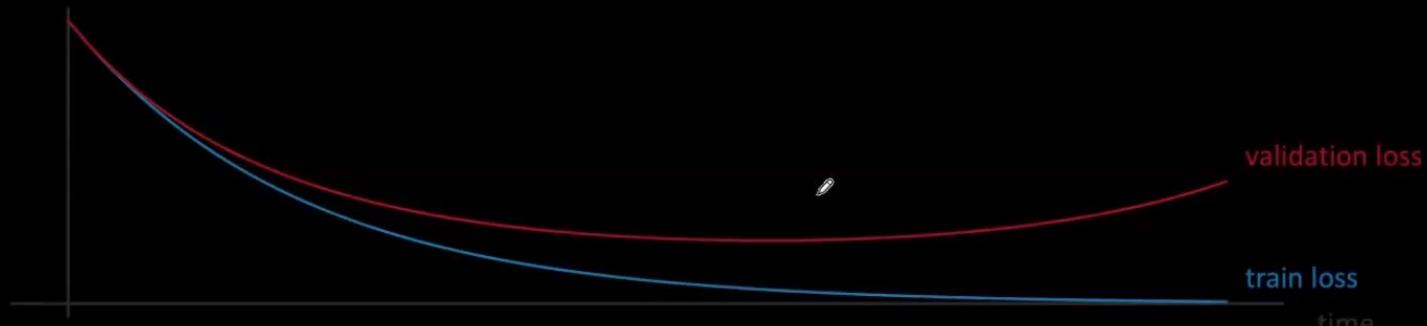
- O processo de otimização de uma rede neural é **estocástico**
 - Vários mínimos locais
 - Alguns melhores que outros
- Temos dois **problemas**:
 - Melhor modelo pode **não ser o da última iteração**
 - Modelo pode ficar **treinando infinitamente** para procurar a melhor solução

Early Stop

- Solução: Early Stop
 - Salvamos modelos de acordo com alguma **métrica**
 - Se ela não aumentar em x épocas, **encerra o treinamento**

- Early-stopping

- `if acc > best_acc: torch.save(model, 'model.pth')`

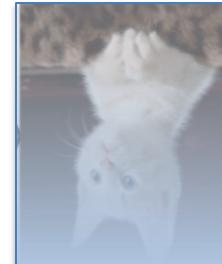


Hiperparâmetros

- Architecture
- Loss Functions
- Optimizers
- Activation Functions
- Nº of Neurons
- Regularization
- Early Stop
- Batch Norm
- Data Augmentation

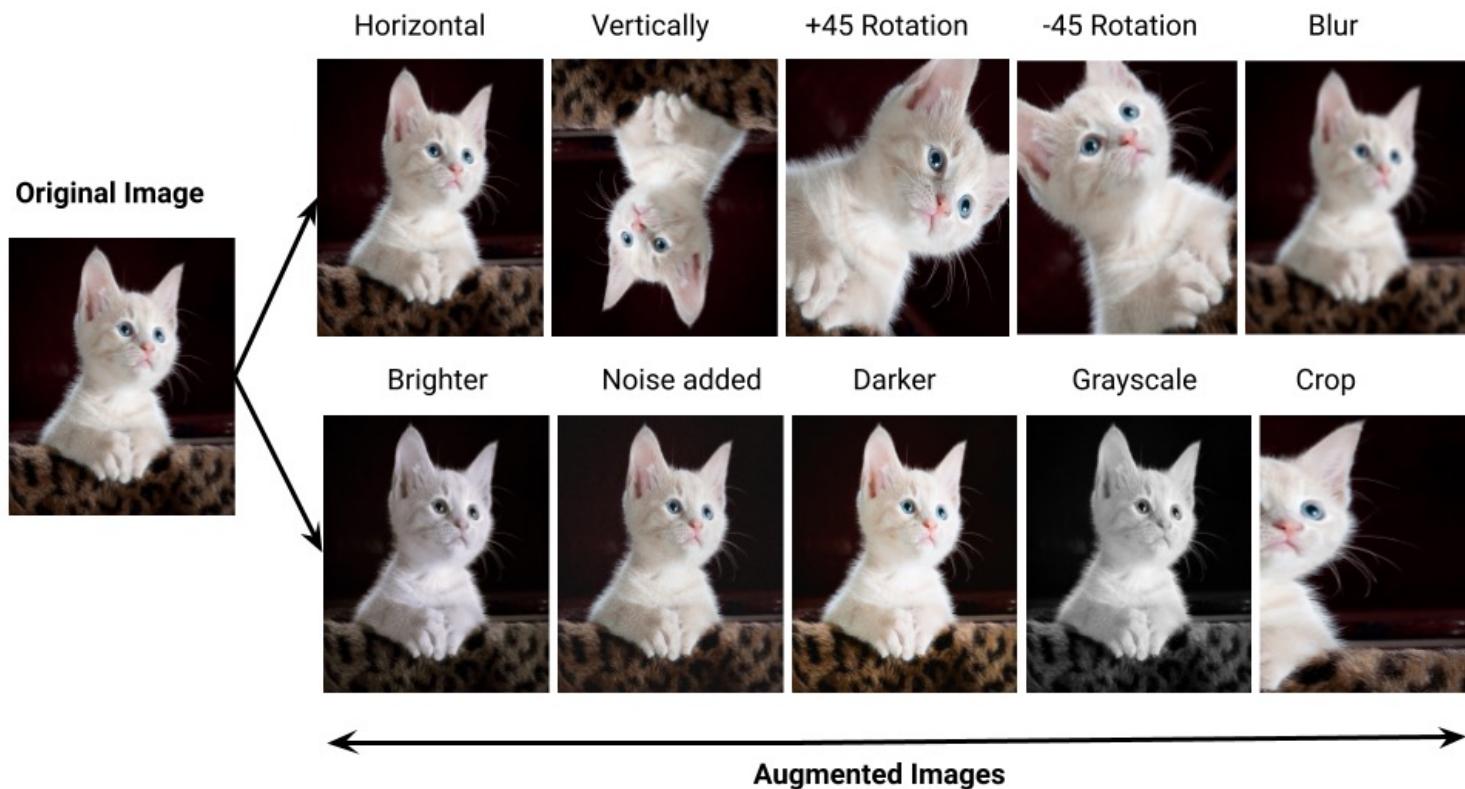
Data Augmentation

- Queremos sempre que o modelo **generalize**
- Mas somos **limitados aos exemplos coletados**
- Como ensinar para rede que todas as imagens abaixo são de gatos?



Data Augmentation

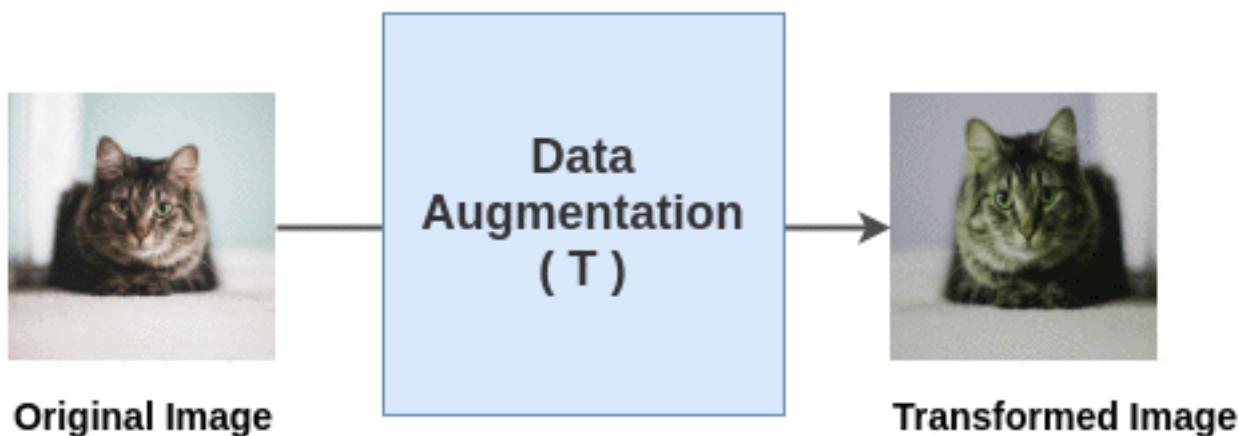
- Aumentando de forma **sintética** o conjunto de dados



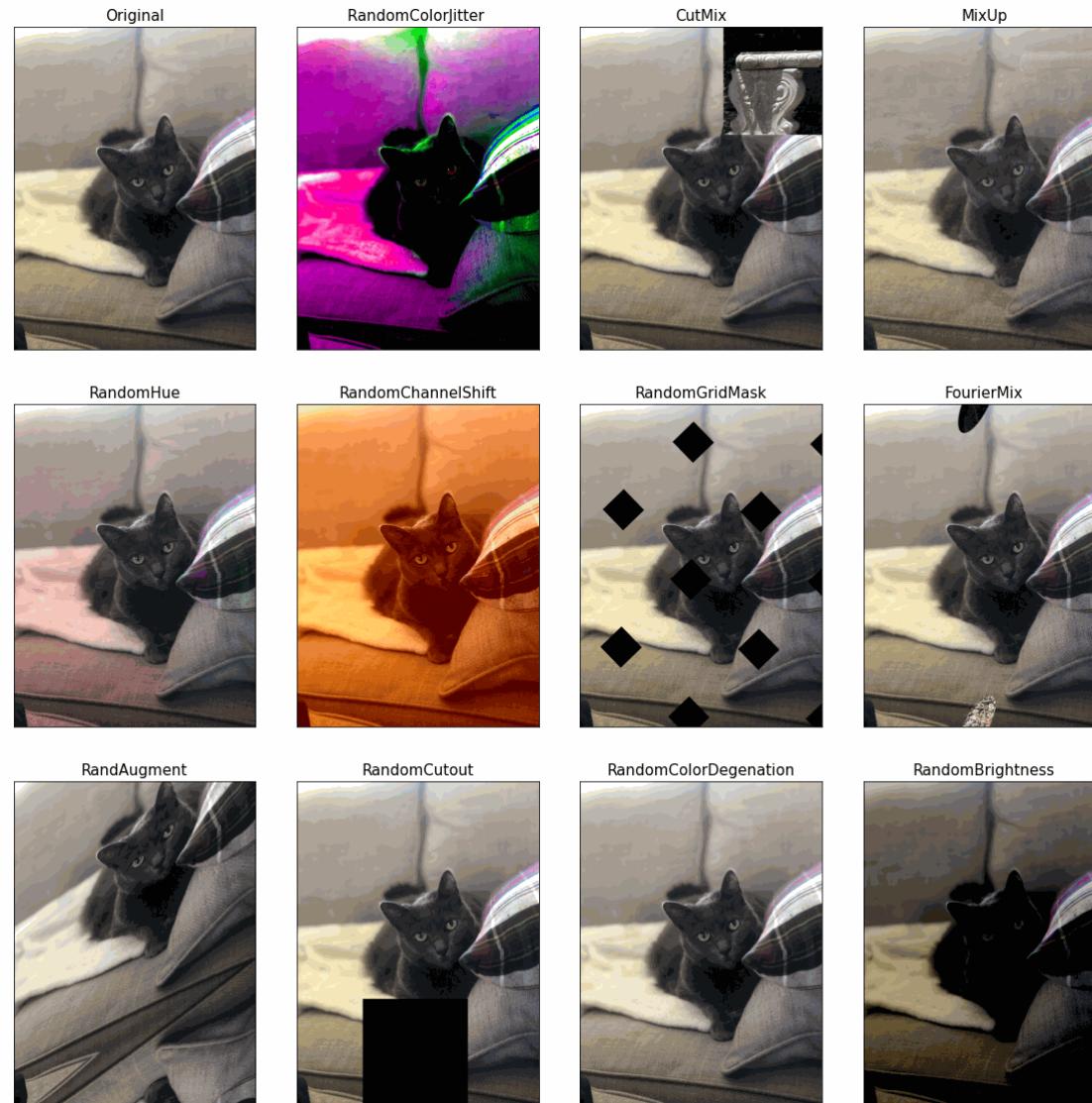
Data Augmentation

- Desta forma garantimos:
 - Melhor generalização
 - Mais dados para treinamento

Random Transformation

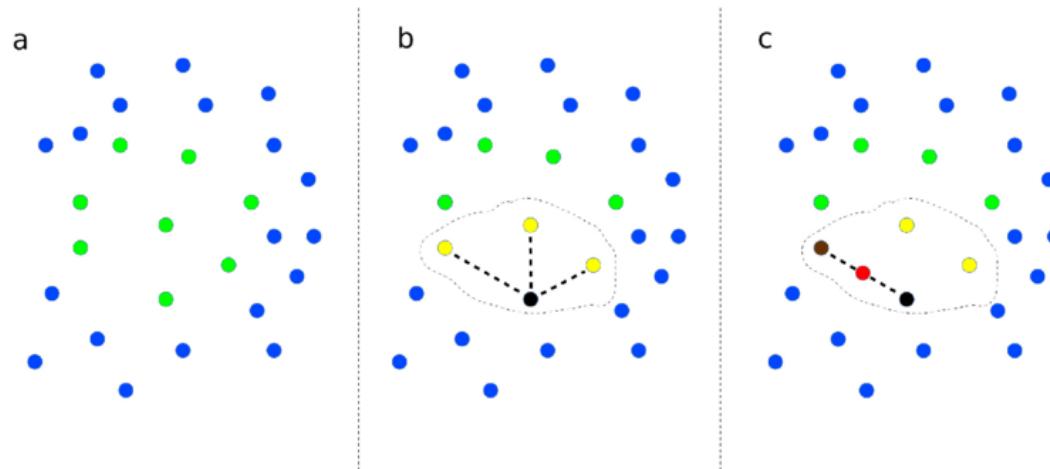


Data Augmentation



Data Augmentation

- Podemos aplicar com **diferentes modalidades**:
 - Dados Tabulares (Ex: SMOTE)
 - Dados Visuais
 - Dados Textuais (NLPAug)

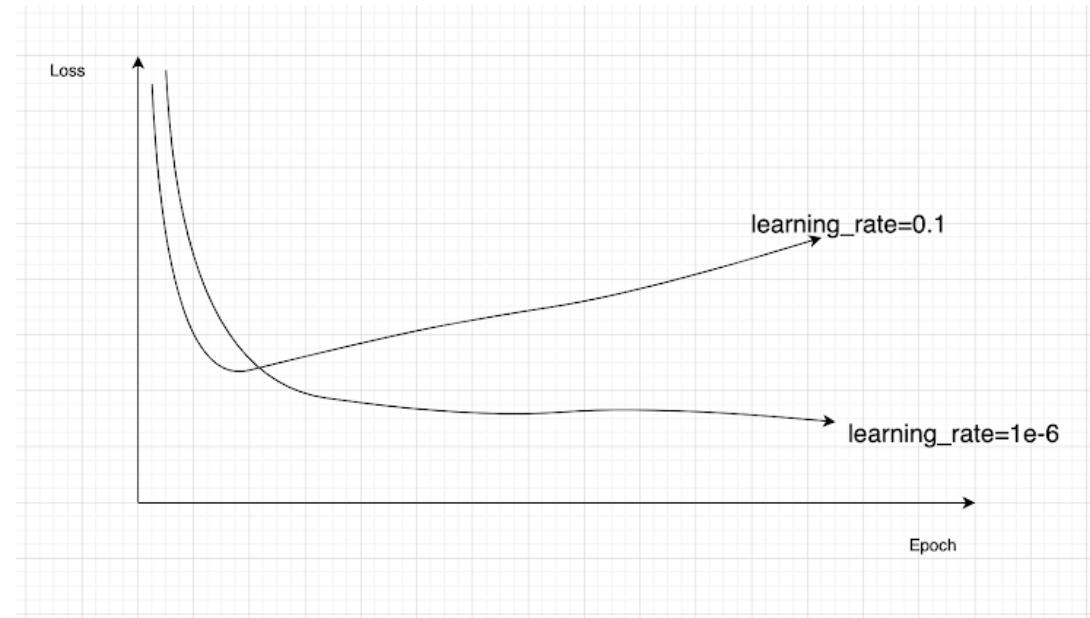


Hiperparâmetros

- Como procurer pelos melhores Hiperparâmetros:
 - Tuning Manual
 - Grid Search
 - Random Search

Tuning Manual

- Definir conjunto de hiperparâmetros
- Analisar resultados da rede e **ajustar conforme as métricas** em validação

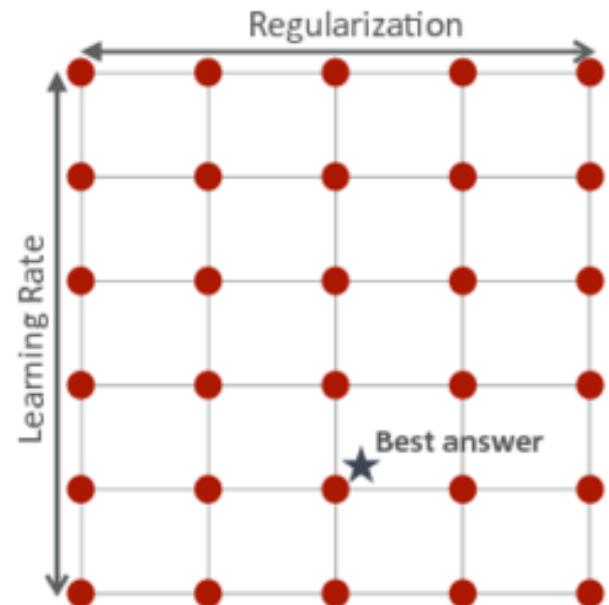


Qual a próxima taxa de LR para testarmos?

- a) 0.2
- b) 10^{-4}
- c) 10^{-8}

Grid Search

- Definir **conjuntos de valores possíveis** para cada Hiperparâmetro
- Treinar uma rede para **cada** combinação
- Ex:
 - Learning rate: {0.1, 0.001, 0.00001}
 - Hidden layers: {(10,10), (10,5), (20,5)}
 - Activation function:
{sigmoid, ReLU, LeakyReLU}



Random Search

- Distribuição de probabilidade para cada hiperparâmetro
- Budget computacional
- Aleatorizar os hiperparâmetros a cada rodada

Melhor Alternativa

- Depende... (Sem almoço grátis em ML);
- Regra de ouro:
 - Random Search seguido de um ajuste-fino manual usualmente traz bons resultados
- Alguns hiperparâmetros possuem valores “padronizados”:
 - Learning Rate: 1e-3, 1e-5,...
 - Função de ativação: ReLU, LeakyReLU,...

Frameworks

- Temos **dois principais frameworks** para trabalhar com redes neurais:
 - Pytorch (Meta)
 - Tensorflow (Google)
- Ambos servem para construir **grafos computacionais autodiferenciáveis**

Frameworks

- Tensorflow: Grafo Estático
 - Execução mais rápida
 - Mais fácil otimizar
 - Difícil debugar e modificar livremente o grafo
- PyTorch: Grafo Dinâmico
 - Execução levemente mais lenta
 - Mais difícil de otimizar
 - Total Liberdade e acesso ao grafo computacional

Resumindo

- Redes Neurais (atuais) são **enormes!!**
 - **Impossível** escrever a fórmula dos gradientes **manualmente** para cada um dos parâmetros
- Backpropagation
 - **Aplicação recursiva da regra da cadeia** por um grafo computacional para calcular os gradientes de todas as entradas/parâmetros/pontos intermediários
- Implementações atuais possuem **estrutura de grafo**
 - Nodos implementam API *backward()* e *forward()*
 - **Forward:** computa os resultados de uma operação e salva valores intermediários necessários para cálculo do gradiente em memória
 - **Backward:** aplica a regra da cadeia para calcular o gradiente da função de custo com relação as entradas

Resumindo

- Neurônios em uma rede são dispostos em **camadas**. Elas podem ser de diferentes tipos, não apenas as totalmente conectadas
- A abstração de **camada** tem a propriedade desejável de permitir a utilização de **código vetorial eficiente** (multiplicação de matrizes!)
- Quanto **maior** a rede, **melhor**!
 - Mas demandará regularização para evitar **overfitting**!