# Inteligência Artificial

Prof. Fabrício Olivetti de França  Prof. Denis Fantinato

3º Quadrimestre de 2019

# Redes Neurais Biológicas

# Rede Neural

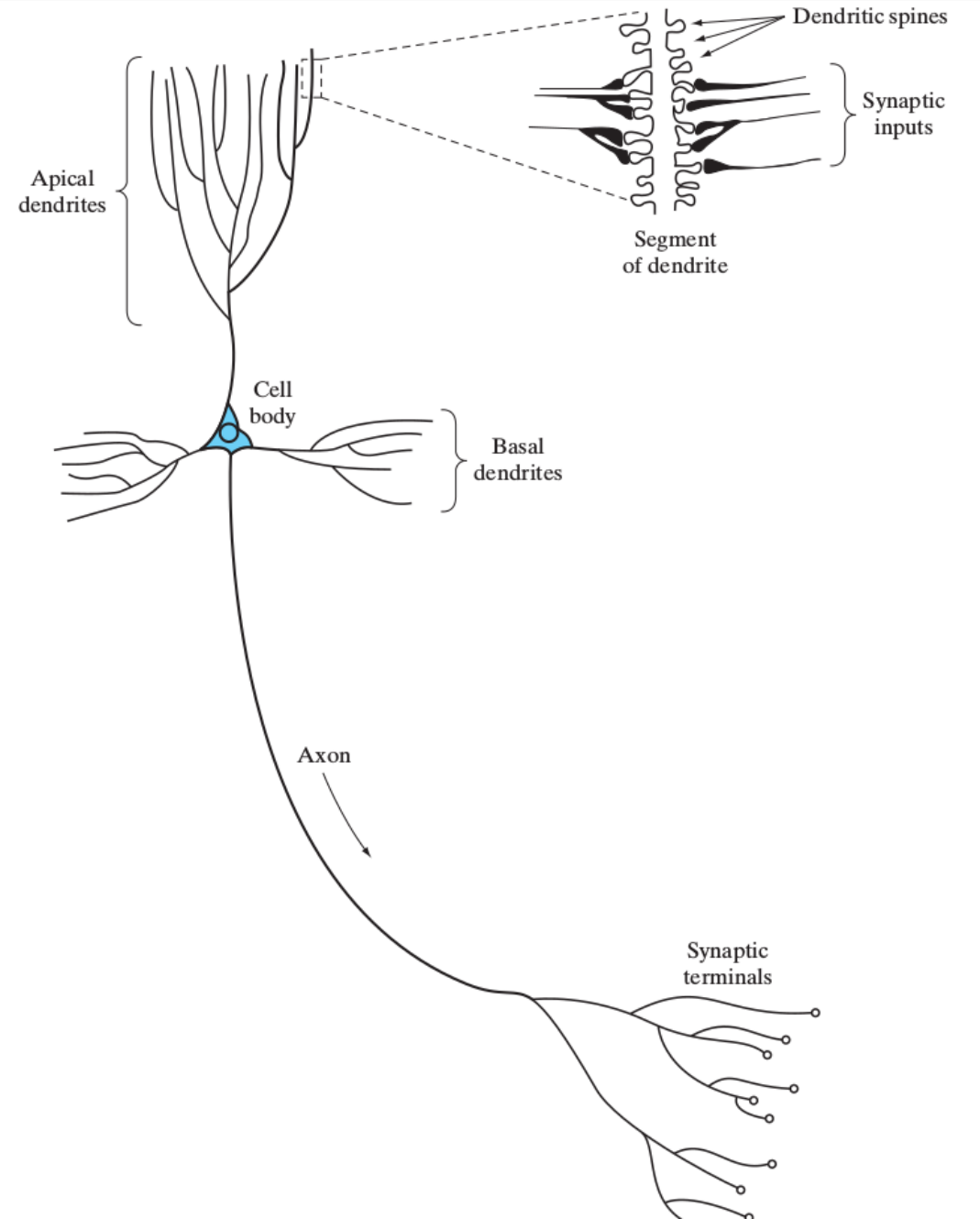
- Área de pesquisa da Neurociência.
- Composta de **neurônios** interconectados formando uma rede de processamento.
- Comunicação via sinais elétricos.

# Rede Neural

A Rede Neural biológica é composta por **neurônios** conectados via axônios.

Essa rede pode ser entendida como um grafo de fluxo de informação.

# Rede Neural


Dendritic spines
Synaptic inputs
Apical dendrites
Segment of dendrite
Cell body
Basal dendrites
Axon
Synaptic terminals

# Rede Neural

De uma forma simplista, o fluxo é determinado pela soma dos sinais elétricos recebidos por um neurônio, se este for maior que um limiar, o neurônio emite um sinal elétrico adiante, caso contrário ele bloqueia o sinal.

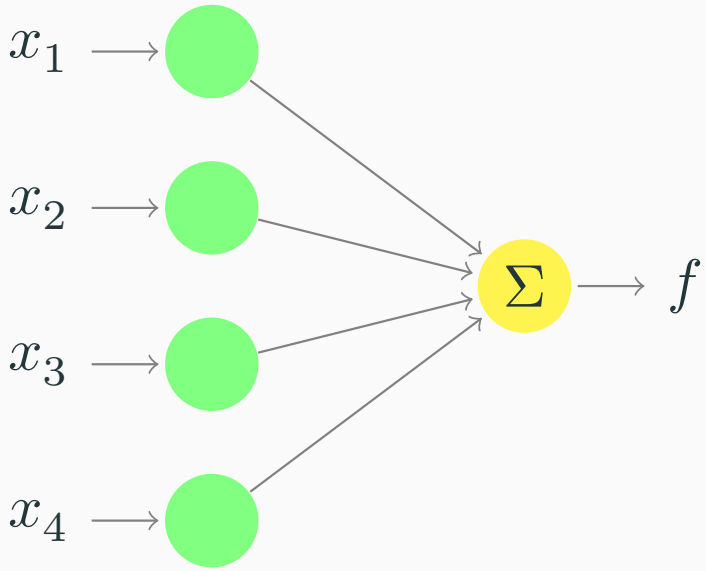
# Neurônio Artificial

# Modelo de Neurônio Artificial

Podemos criar um neurônio artificial como um *nó* de um Grafo que recebe múltiplas entradas e emite uma saída.

A saída é definida como a aplicação de uma função de ativação na soma dos valores de entrada.

# Modelo de Neurônio Artificial



# Modelo de Neurônio Artificial

É possível ponderar a importância dos estímulos de entrada do neurônio através de um vetor de pesos **w**, resultando

$$z = \sum_i w_i x_i$$

Em notação vetorial:

$$z = \mathbf{w}^T \mathbf{x}$$

# Modelo de Neurônio Artificial

Pensando em entradas $x_i$ com valores reais entre $0$ e $1$, o neurônio é ativado sempre que a soma das entradas for maior que um determinado $\tau$, ou seja, a função $f$ é:

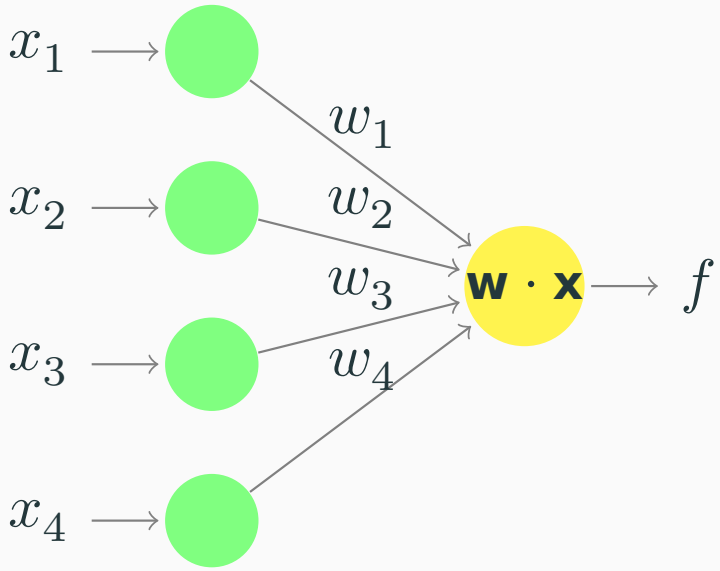
$$y = f(z) = \begin{cases} 0, & \text{se } z < \tau \\ g(z), & \text{c. c.} \end{cases},$$

com $g(z)$ a função de ativação que determinar o pulso a ser enviado e $z$ a soma dos estímulos de entrada.

# Modelo de Neurônio Artificial

Também é possível substituir a somatória pelo produto interno $\mathbf{w} \cdot \mathbf{x}$:

# Modelo de Neurônio Artificial

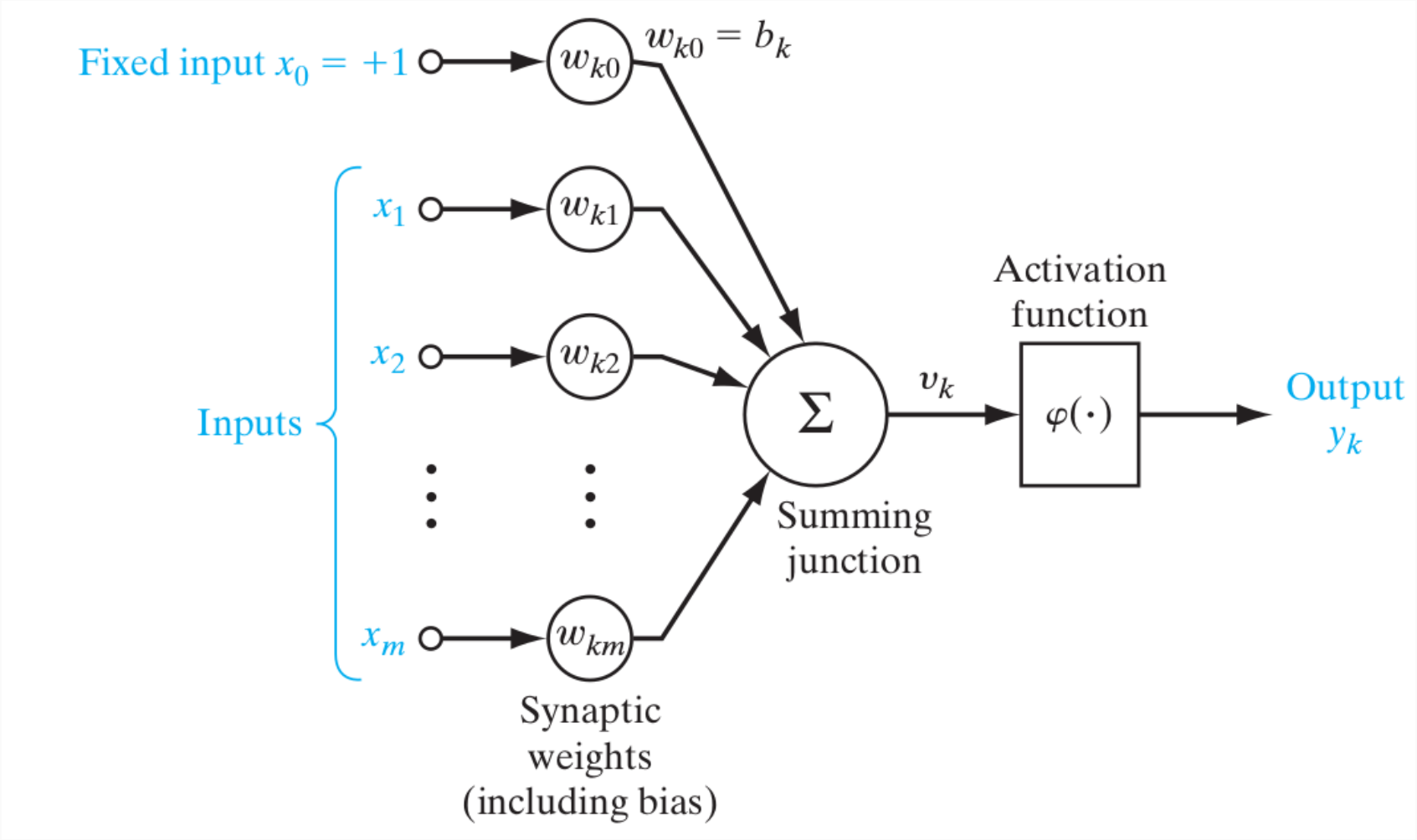


# Modelo de Neurônio Artificial

Para um ajuste mais preciso do limiar de ativação $\tau$, considera-se um termo de viés ou de *bias* queé adiocionado à parte linear do neurônio artificial:

$$z = \sum_i w_i x_i + b$$

# Modelo de Neurônio Artificial

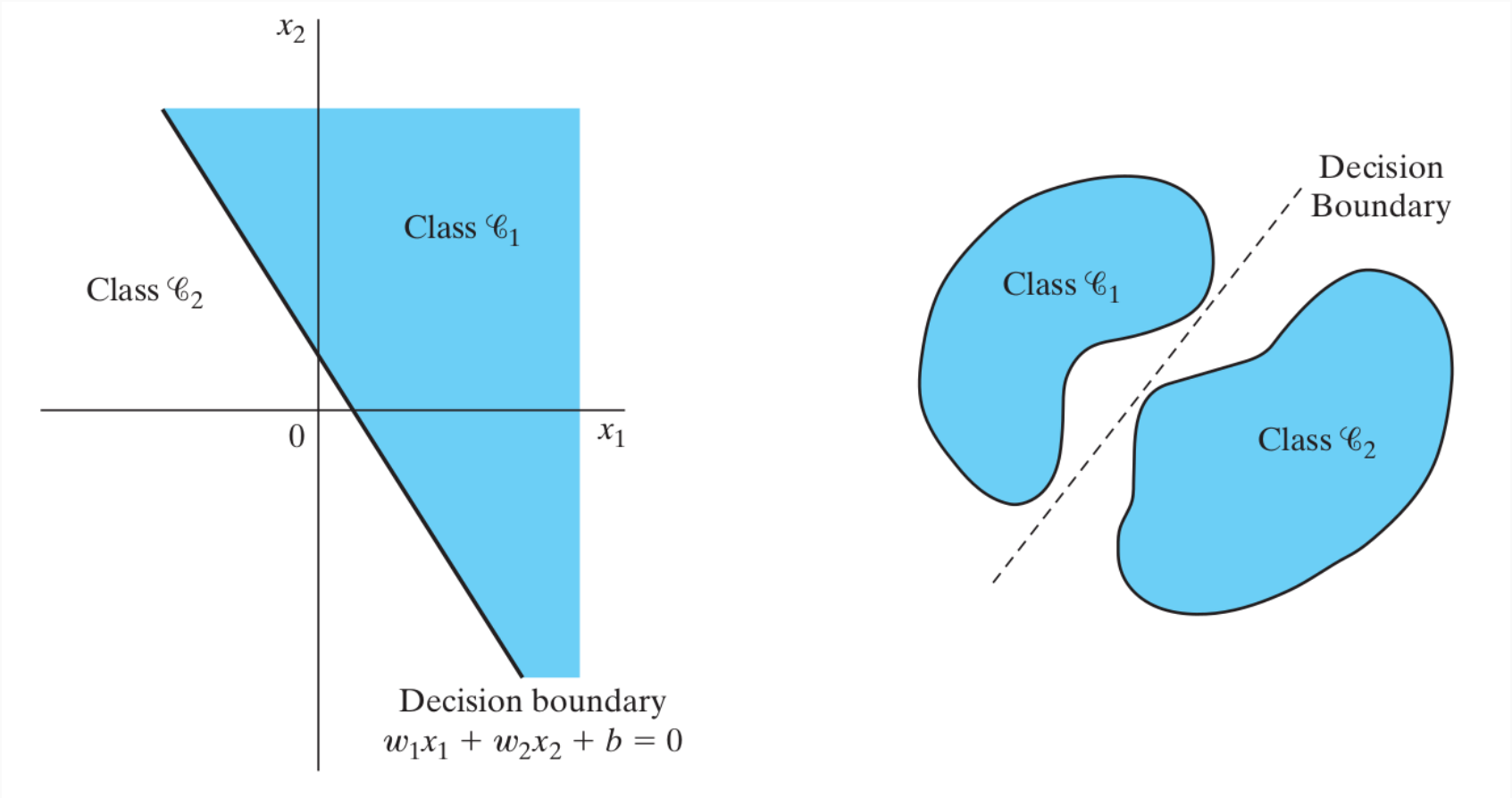


# Modelo de Neurônio Artificial

O Neurônio Artificial é conhecido como Perceptron. Esse tipo de modelo é dito **paramétrico** pois é descrito por um número finito de parâmetros (**w**, $b$).

Esse modelo consegue descrever apenas relações lineares. Em um problema de classificação, o perceptron consegue classificar/distinguir duas classes linearmente separáveis.

# Modelo de Neurônio Artificial



# Rede Neural Artificial

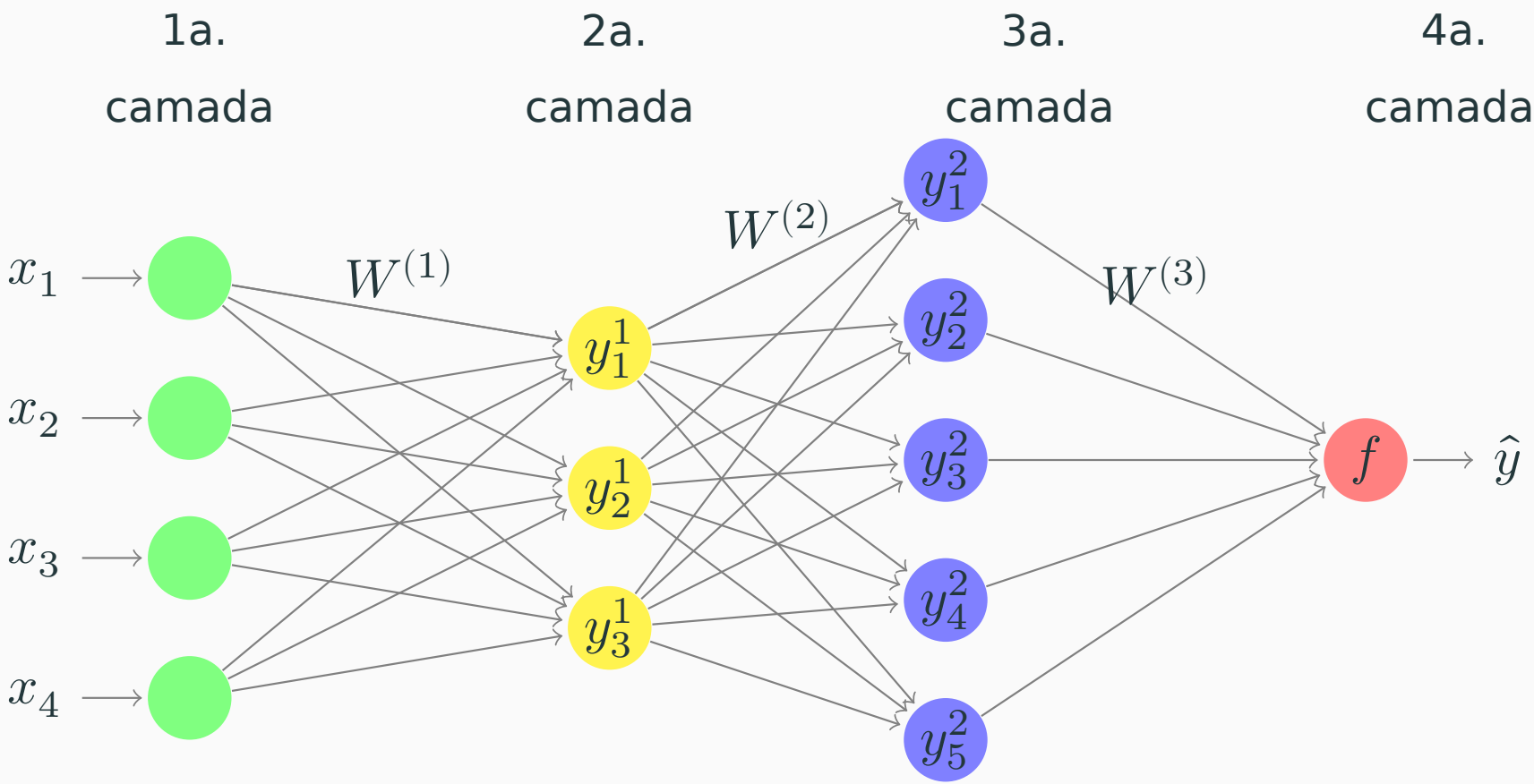
# Percéptron de Múltiplas Camadas

Conhecido como **Multi-Layer Perceptron** (MLP) ou **Feedforward Neural Network**.

Rede composta de vários neurônios conectados por *camadas*.

# Percéptron de Múltiplas Camadas

O uso de camadas permite aproximar funções não-lineares.



# Percéptron de Múltiplas Camadas

Essa é uma forma de Aprendizagem Profunda (*Deep Learning*).

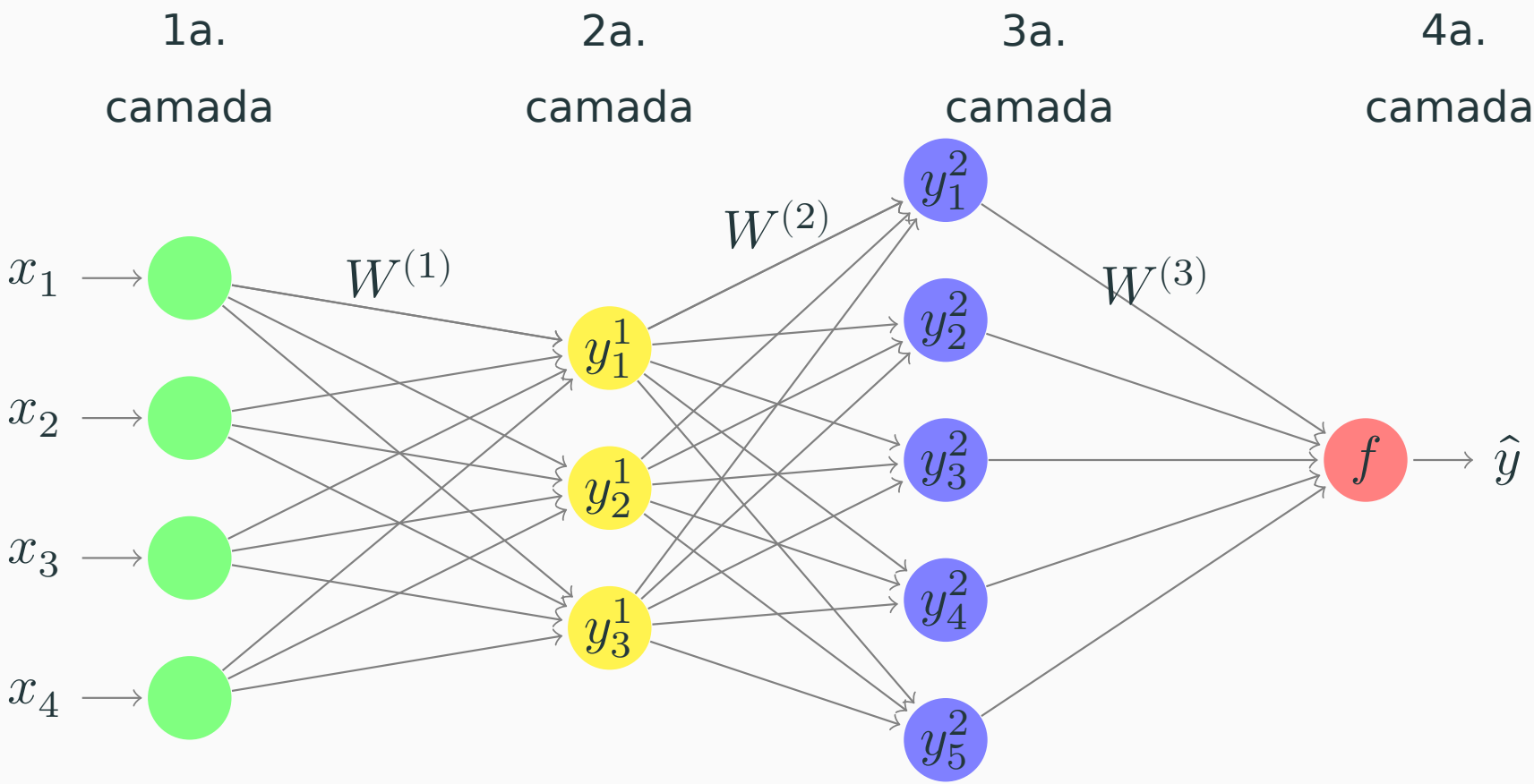
Quanto mais camadas, mais profunda é a rede.

Cada camada tem a função de criar novos atributos mais complexos.

Essa rede define um modelo computacional.

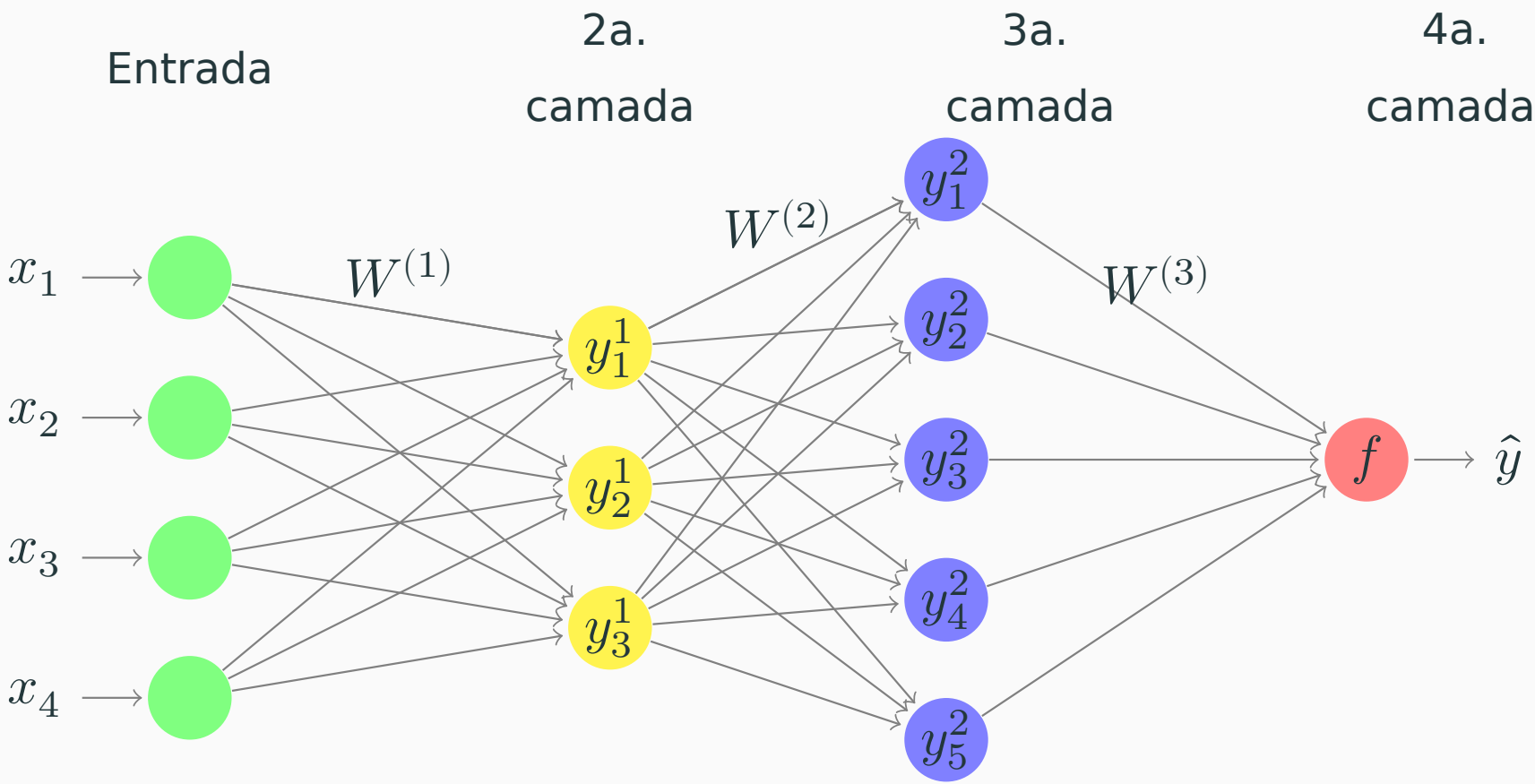
# Definições

Cada conjunto de nós é denominado como uma *camada*.



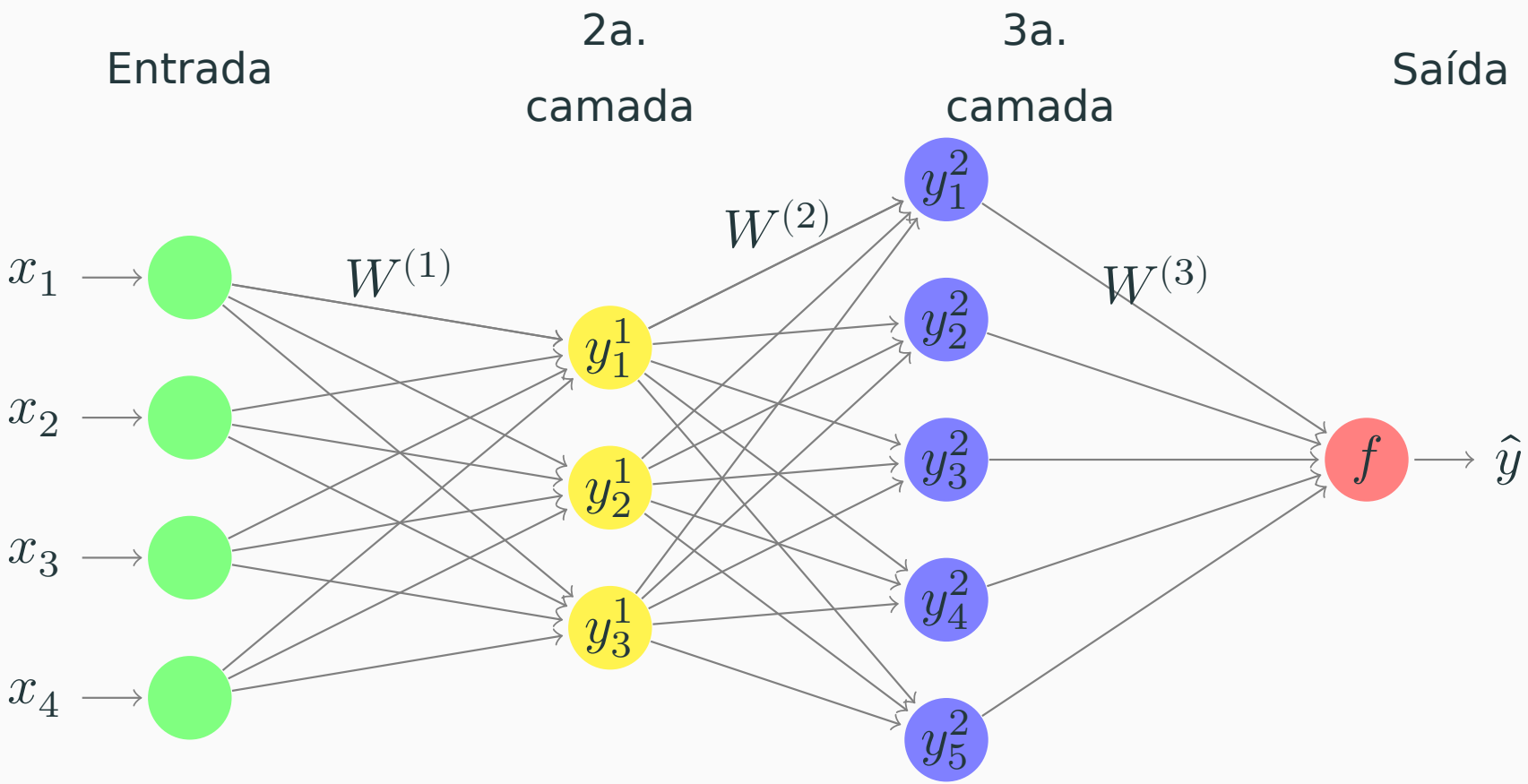
# Definições

A primeira camada é conhecida como *entrada*.



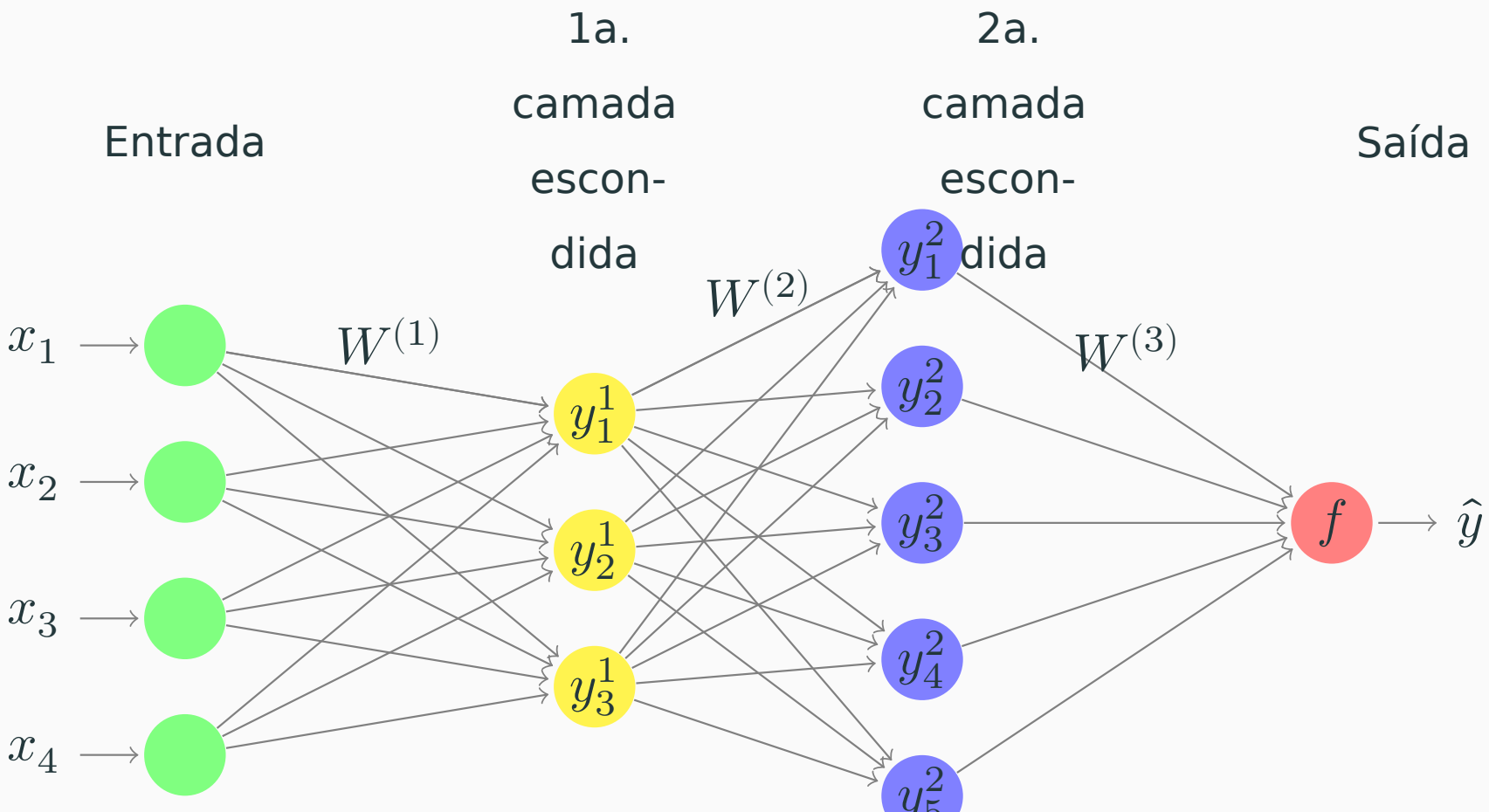
# Definições

A última camada é a *saída* ou *resposta* do sistema.



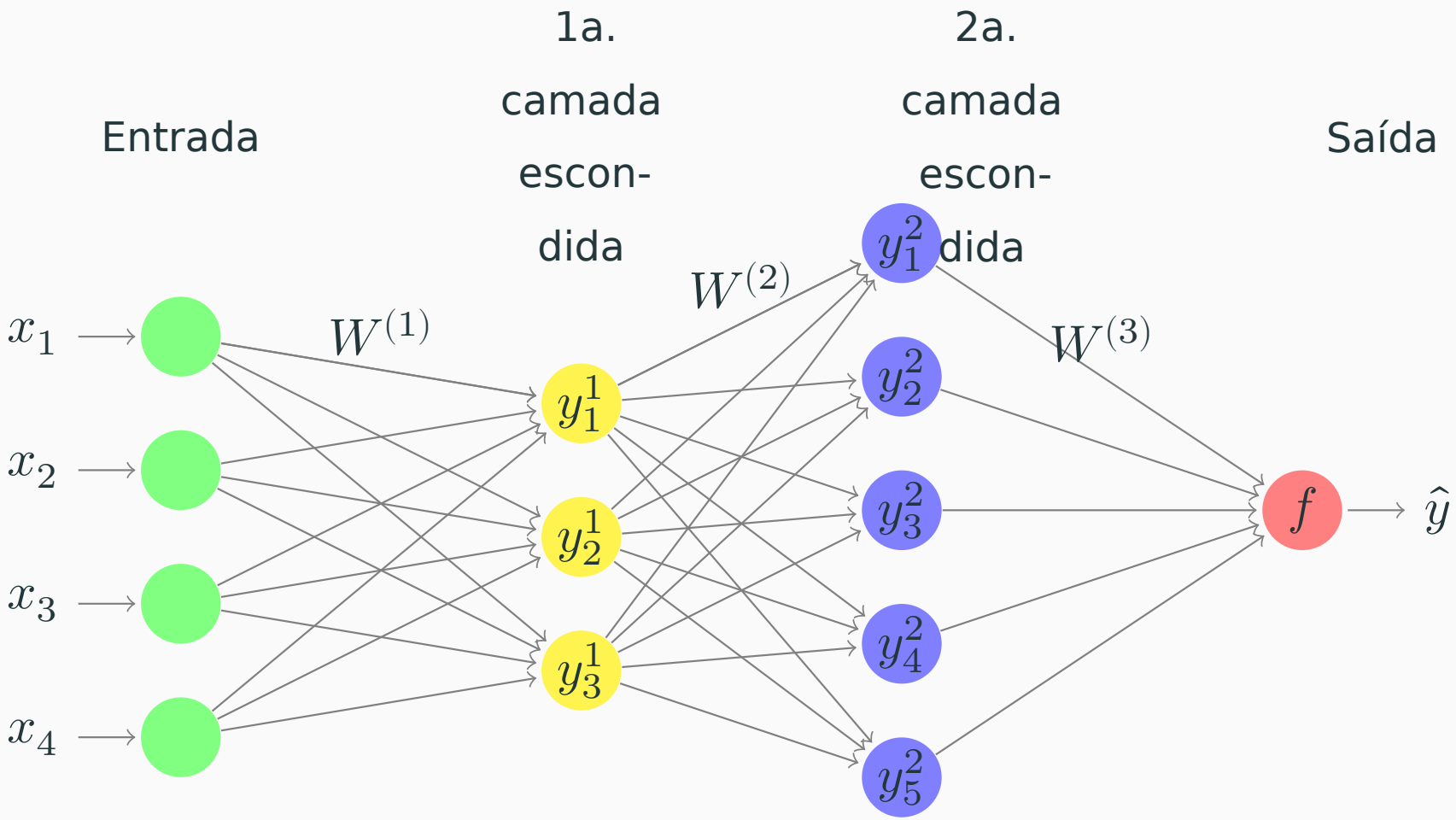
# Definições

As camadas restantes são numeradas de $1$ a $m$ e são conhecidas como *camadas escondidas/intermediárias* ou *hidden layers*. Esse nome vem do fato de que elas não apresentam um significado explícito de nosso problema.



# Definições

Esse grafo é ponderado e os pesos são definidos por uma matriz $W$ de dimensão $n_o \times n_d$, com $n_o$ sendo o número de neurônios da camada de origem e $n_d$ da de destino.



# Definições

Assumindo que as entradas são representadas por um vetor linha **x**, o processamento é definido como:

$$\mathbf{y^1} = f^{(1)}(W^{(1)}\mathbf{x})$$

# Exercício

Dado que a camada $i$ tem $m$ neurônios e a camada $j$ tem $n$ neurônios.

Determine as dimensões de $\mathbf{y}^i$, $W^{(j)}$, e $\mathbf{y}^j$.

# Definições

A camada seguinte calcula a próxima saída como

$$\mathbf{y^2} = f^{(2)}(W^{(2)}\mathbf{y^1})$$

e assim por diante. Regra geral (camada $j$ depois da camada $i$):

$$\mathbf{y^j} = f^{(j)}(W^{(j)}\mathbf{y^i})$$

# Processamento em Lote

Se temos uma matriz de dados $X \in \mathbb{R}^{d \times n}$ e quiséssemos obter uma saída $\mathbf{y} \in \mathbb{R}^{1 \times n}$ em uma Rede Neural com duas camadas escondidas contendo $h1$ e $h2$ neurônios, qual seria a sequência de processamento?

(quais cálculos temos que fazer?)

# Processamento em Lote

Se temos uma matriz de dados $X \in \mathbb{R}^{d \times n}$ e quiséssemos obter uma saída **y** $\in \mathbb{R}^{1 \times n}$ em uma Rede Neural com duas camadas escondidas contendo $h1$ e $h2$ neurônios, qual seria a sequência de processamento?

$$Y^{(1)} = f^{(1)}(W^{(1)}X)$$

$$Y^{(2)} = f^{(2)}(W^{(2)}Y^{(1)})$$

$$y = f^{(3)}(W^{(3)}Y^{(2)})$$

# Funções de Ativação

As funções de ativação comumente utilizadas em Redes Neurais são:

- **Linear:** $f(z) = z$, função identidade.
- **Logística:** $f(z) = \frac{1}{1+e^{-z}}$, cria uma variável em um tipo sinal, com valores entre $0$ e $1$.
- **Tangente Hiperbólica:** $f(z) = \tanh(z)$, idem ao anterior, mas variando entre $-1$ e $1$.
- **Rectified Linear Units:** $f(z) = \max(0, z)$, elimina os valores negativos.
- **Softmax:** $f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$, faz com que a soma dos valores de $z$ seja igual a $1$.

# Ajustando os Parâmetros

Para determinar os valores corretos dos pesos, utilizamos o Gradiente Descendente, assim como nos algoritmos de Regressão Linear e Logística.

# Ajustando os Parâmetros

Note porém que o cálculo da derivada da função de erro quadrático é igual aos casos já estudados apenas na última camada.

# Ajustando os Parâmetros

Para as outras camadas precisamos aplicar o algoritmo de **Retropropagação** que aplica a regra da cadeia.

# Retropropagação

O algoritmo segue os seguintes passos:

- Calcula a saída para uma certa entrada.
- Calcula o erro quadrático.
- Calcula o gradiente do erro quadrático em relação a cada peso.
- Atualiza pesos na direção oposta do gradiente.
- Repita.

# Retropropagação

Assumindo a função de ativação logística $\sigma(z)$, cuja derivada é $\sigma(z)(1-\sigma(z)) = y(1-y)$, e erro quadrático

$$e^2 = (\hat{y} - y)^2$$

o gradiente da camada de saída é

$$\frac{\partial e^2}{\partial W^{(3)}} = 2(\hat{y} - y)y(1-y)\mathbf{x}$$

# Retropropagação

O gradiente da camada anterior é calculado como:

$$\frac{\partial e^2}{\partial W^{(2)}} = 2(\hat{y} - y)(y(1-y))W^{(3)}\sigma'(W^{(2)}\mathbf{x})\mathbf{x}$$

E assim por diante até a camada de entrada.

Regra de Adaptação:

$$W^{(i)} = W^{(i)} - \mu\frac{\partial e^2}{\partial W^{(i)}}$$

em que $\mu$ é a taxa de aprendizagem.

# Vanishing Gradient

Determine os valores mínimos e máximos das derivadas das seguintes funções de ativação:

- Logistica: $\sigma(z) = \frac{1}{1+e^{-z}}$ cuja derivada é $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.
- Tangente Hiperbólica: $\tanh(z)$ cuja derivada é $\tanh'(z) = 1 - \tanh^2(z)$.
- RELU: $relu(z) = max(0, z)$ cuja derivada é $relu'(z) = 1, 0$, se $z > 0$ ou caso contrário, respectivamente.

# Vanishing Gradient

Como $0 \leq \sigma(z) \leq 1$, temos que o valor mínimo da derivada é quando $\sigma(z) = \{0, 1\}$ em que $\sigma'(z) = 0$.

O valor máximo ocorre quando $\sigma(z) = 0.5$ e $\sigma'(z) = 0.25$.

# Vanishing Gradient

Como $-1 \leq \tanh(z) \leq 1$, temos que o valor mínimo da derivada é quando $\tanh(z) = \{-1, 1\}$ em que $\tanh'(z) = 0$.

O valor máximo ocorre quando $\tanh(z) = 0$ e $\tanh'(z) = 1$.

# Vanishing Gradient

Como $relu'$ assume apenas dois valores, temos diretamente que o mínimo e máximo são $0$ e $1$, respectivamente.

# Vanishing Gradient

O gradiente de cada camada quando utilizamos a função logística, terá o valor de no máximo $25\%$ da camada seguinte, ou seja, quanto mais camadas, menores os valores de gradientes das primeiras camadas.

# Vanishing Gradient

Isso é conhecido como *Vanishing Gradient* e é possível remediar utilizando outras funções de ativação como $tanh$ e RELU.

# Dicas para Melhorar o desempenho do ajuste

- Utilize tanh como função sigmoidal.
- Utilize *softmax* para multi-classes.
- Escale as variáveis de saída para a mesma faixa de valores da segunda derivada da função de ativação (ex.: para tanh deixe as variáveis entre $-1$ e $1$).

# Dicas para Melhorar o desempenho do ajuste

- Ajuste os parâmetros utilizando mini-batches dos dados de treinamento.
- Inicialize os pesos como valores aleatórios uniformes com média zero e desvio-padrão igual a $\frac{1}{\sqrt{m}}$, com $m$ sendo o número de nós da camada anterior.

# Keras

# Biblioteca Keras

Uma biblioteca intuitiva de Redes Neurais em Python:

- Permite construir, treinar e executar Redes Neurais diversas (Deep Learning).
- Escrita em Python e permite modelos de configurações avançados.
- Usa Tensorflow ou Theano, serve apenas como *frontend*.
- Permite o uso de CPU ou GPU para processar.
- Usa estrutura de dados do *numpy* e estrutura de comandos similar ao *scikit-learn*.

# Keras

Vamos criar a seguinte rede de exemplo no Keras:

Primeiro criamos o modelo do tipo *Sequential* que cria cada camada da rede em sequência:

```
from keras.models import Sequential
model = Sequential()
```

# Keras

Em seguida importamos as funções *Dense* para criação de camadas densas e *Activation* que define a função de ativação nos neurônios:

```
from keras.layers import Dense, Activation
```

# Keras

Finalmente, criamos as camadas da primeira até a última na sequência. Note que para a primeira camada devemos especificar a dimensão de entrada:

```
model.add(Dense(units=3, input_dim=4))
model.add(Activation('tanh'))

model.add(Dense(units=5))
model.add(Activation('tanh'))
```

# Keras

A camada de saída para problemas de regressão deve ter um neurônio e ativação linear:

```
model.add(Dense(units=1))
model.add(Activation('linear'))
```

# Keras

A camada de saída para problemas de classificação binária deve ter um neurônio e ativação sigmoid:

```
model.add(Dense(units=1))
model.add(Activation('sigmoid'))
```

# Keras

A camada de saída para problemas de classificação multi-classes deve ter um neurônio e ativação softmax:

```
model.add(Dense(units=1))
model.add(Activation('softmax'))
```

# Keras

Uma vez que a rede está construída, podemos *compilá-la* e definir a função de erro e o algoritmo de treinamento:

```
model.compile(loss='mean_squared_error',
              optimizer='sgd'
              metrics=['accuracy']
             )
```

# Keras

Possíveis funções de erro:

- mean_squared_error: erro quadrático médio.
- squared_hinge: maximiza a margem de separação.
- categorical_crossentropy: minimiza a entropia, para multiclasses.

Veja mais em:
https://github.com/keras-team/keras/blob/master/keras/activations.py.

# Keras

Possíveis algoritmos de otimização:

- sgd: Stochastic Gradient Descent.
- rmsprop: ajusta automaticamente a taxa de aprendizado.
- adam: determina a taxa de aprendizado ótima de cada iteração.

Veja mais em:
https://github.com/keras-team/keras/blob/master/keras/optimizers.py.

# Keras

Possíveis métricas de avaliação:

- sgdmean_squared_error: Erro quadrático médio.
- mean_absolute_error: Erro absoluto médio.
- binary_accuracy: Acurácia para classes binárias.
- categorical_accuracy: Acurácia para multi-classes.

Veja mais em:
https://github.com/keras-team/keras/blob/master/keras/metrics.py.

# Keras

Finalmente, o ajuste da rede é feita com:

```
model.fit(x_train, y_train, batch_size=n, epochs=max_it)
score = model.evaluate(x_val, y_val)
y_pred = model.predict(x_test)
```

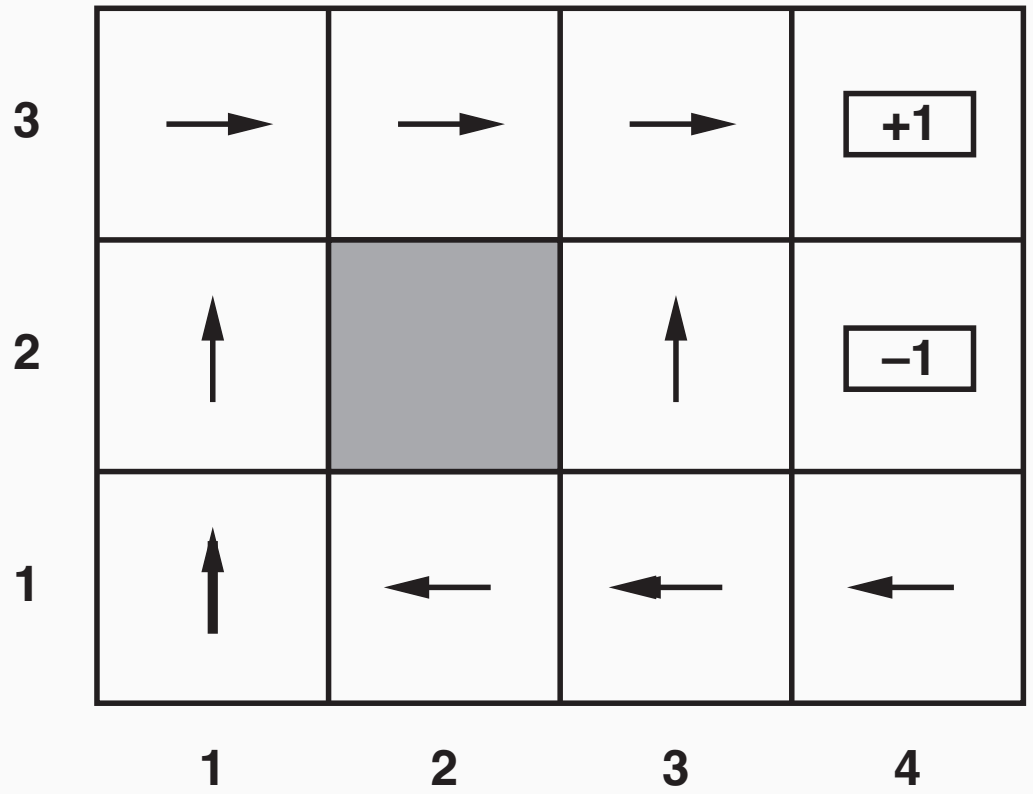
# Parte II

# Redes Neurais em problemas de IA

# Grid World

Relembrando o nosso Grid World vimos que existem duas formas de pensar na solução do problema:

- Encontrando uma política ótima.
- Estimando a função Utilidade para definir a política ótima.



# Grid World

As Redes Neurais podem resolver ambos, dado que temos um conjunto de amostras de entrada.

Obter amostras dos valores da função utilidade pode ser mais fácil do que amostras da política ótima.

# Grid World

Um procedimento simples é seguir uma certa política, gerar $n$ amostras, estimar a função utilidade e colocar em uma tabela:

| s | U |
|---|---|
| (1,1) | 0.72 |
| (1,2) | 0.76 |
| (1,3) | 0.80 |
| (1,2) | 0.84 |
| (1,3) | 0.88 |
| (2,3) | 0.92 |
| (3,3) | 0.96 |
| (4,3) | 1.00 |

# Grid World

Em seguida cria-se uma Rede Neural especificando a topologia e os parâmetros tal que a entrada seja a coordenada dos estados e a saída o valor estimado de $U(s)$.

Aplica-se o procedimento de retropropagação até convergência e usa-se a Rede Neural para determinar uma nova política $\pi$. O processo pode ser repetido até encontrar uma política satisfatória.

# Grid World

Uma alternativa similar é acrescentar a informação da ação na entrada da Rede Neural e criar estimativas de $Q(s, a)$ do algoritmo Q-Learning.

O procedimento se torna parecido com a alternância entre o algoritmo Q-Learning e a estimação de $Q$ através das Redes Neurais.

# Grid World

A estratégia de fazer com que a Rede Neural descubra a política ótima requer apenas algumas alterações. Uma delas é que a Rede Neural terá múltiplas saídas, uma para cada possível ação.

# Neuroevolução

Uma alternativa para o ajuste dos pesos se dá pela utilização de heurísticas de busca e otimização que não precisam de informação do gradiente.

Vimos na aula passada alguns exemplos como a Estratégia Evolutiva.

# Neuroevolução

Dada uma Rede Neural com a topologia definida, basta fazer:

- Construa uma população de pesos aleatórios, cada um representando um conjunto de pesos distintos.
- A função-objetivo é a média do resultado final de aplicar cada um dos conjuntos de pesos até o final de um episódio.
- A mutação pode ser uma estimativa do gradiente.

# Mutação

Dado um vetor **w** representando os pesos de uma Rede Neural:

- Crie uma matriz $G$ contendo $n$ vetores aleatórios com distribuição Gaussiana de mesmas dimensões de **w**.
- Gere $n$ novos indivíduos com pesos $\mathbf{w} + \alpha \cdot \mathbf{g}_i$.
- Avalie esses indivíduos gerando um vetor **f** dos valores avaliados.
- Atualize **w** com:

$$\mathbf{w} = \mathbf{w} + \alpha \frac{1}{n} \sum_i f_i \cdot \mathbf{g}_i$$

# Neuroevolução

O algoritmo de Neuroevolução é indicado quando temos pouca informação sobre o problema e/ou estamos lidando com um espaço de estados intratável para ser capturado por amostragens.