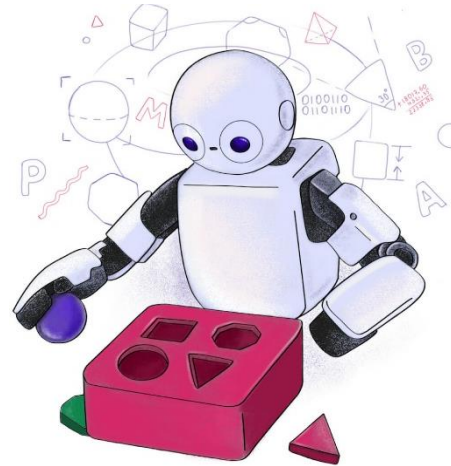


# TP557 - Tópicos avançados em IoT e Machine Learning: *Regressão com DNNs (Parte I)*



***Inatel***

Felipe Augusto Pereira de Figueiredo  
felipe.figueiredo@inatel.br

# O que vamos ver?

- Anteriormente, vimos como minimizar iterativamente a função de erro usando o gradiente descendente.
  - Dá-se um palpite sobre os valores dos pesos (i.e., inicializa-se os pesos com valores aleatórios);
  - Mede-se o erro causado por esse palpite;
  - Usa-se a informação obtida através do erro (i.e., vetor gradiente) para melhorar o palpite;
  - Repete-se o processo até a convergência ou até que um critério de parada seja atingido.
- Em termos gerais, é assim que as redes neurais são treinadas.
- Portanto, neste tópico, nós veremos como criar uma rede neural que atinja o mesmo objetivo do exemplo anterior, encontrar uma **função linear** que aproxime um conjunto de dados.

# Conjunto de dados de treinamento

$$\mathbf{x} = \{-1, 0, 1, 2, 3, 4\}$$

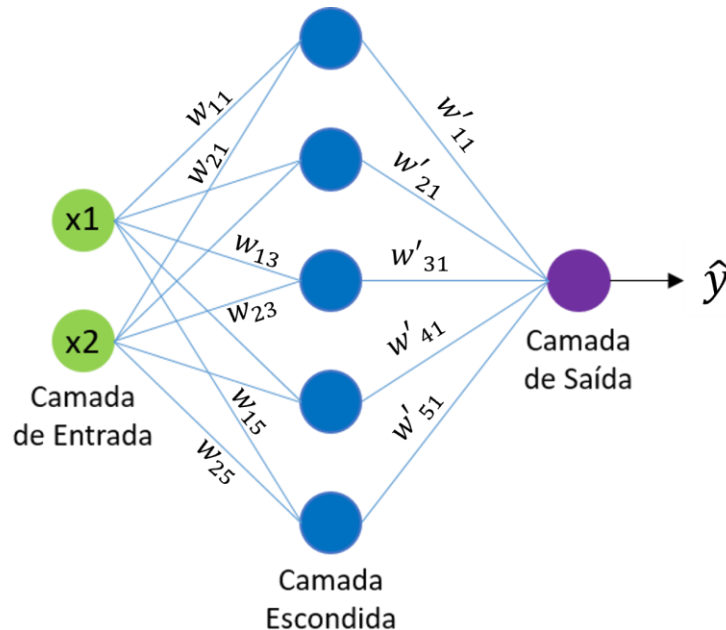
$$\mathbf{y} = \{-3, -1, 1, 3, 5, 7\}$$

- Ao lado temos o conjunto de dados que usamos anteriormente.
- Nosso **objetivo** é encontrar um **modelo que mapeie** os valores de  $\mathbf{x}$  em  $\mathbf{y}$  de forma ótima (no sentido da minimização do erro).
- Antes, nós usamos o **gradiente descendente** para **otimizar os pesos de uma função hipótese** com formato de reta.
- Agora, treinaremos uma rede neural para resolver o problema do mapeamento.

# O modelo

$$\hat{y}(n) = a_0 + a_1 x_1(n) + \dots + a_K x_K(n)$$

$$\hat{y}(n) = a_0 + a_1 x_1(n) + a_2 x_1^2(n) + \dots + a_K x_1^K(n)$$



- Notem que independentemente se estamos usando **hiperplanos**, **polinômios** ou **redes neurais**, em todos os casos temos um **modelo** e o nosso **objetivo é encontrar seus pesos de forma que o erro seja minimizado**.

# O código da rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Usaremos as APIs da biblioteca TensorFlow para *criar*, *treinar* e *avaliar* nossas redes neurais.
  - **Application programming interface (API)**: conjunto de regras (e.g., funções, classes, etc.) que um software oferece para que desenvolvedores ou outros programas possam interagir com ele.

# Definindo o conjunto de treinamento

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- As primeiras duas linhas definem o conjunto de treinamento.
- Ou seja, os valores de  $x$  e  $y$  que usaremos para otimizar o modelo durante as iterações e épocas de treinamento.
- Cada valor de  $x$  corresponde a um valor de  $y$ .
- O Tensorflow espera que o conjunto de dados sejam *arrays* NumPy.

# Definindo uma rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

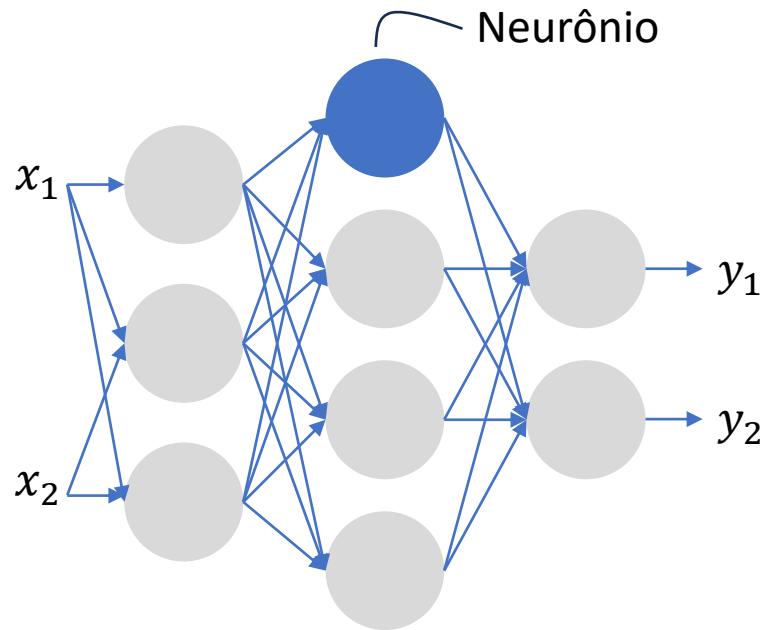
```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Na sequência, temos a definição da rede neural.
- É uma rede neural muito simples, uma das mais simples que veremos.
- Antes de discutirmos o código, vamos relembrar alguns termos para que possamos entendê-lo.

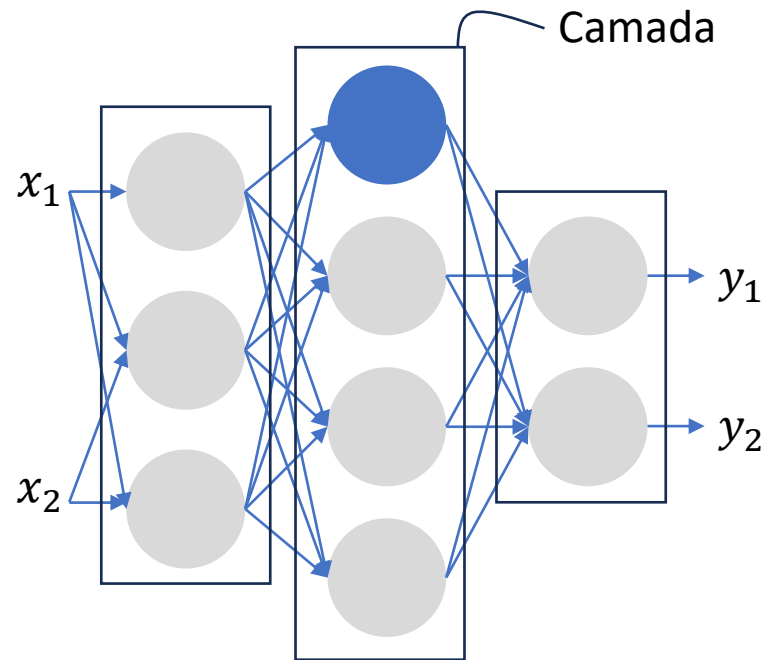
# Relembrando alguns termos

- Cada um dos círculos ao lado é um neurônio ou nó.



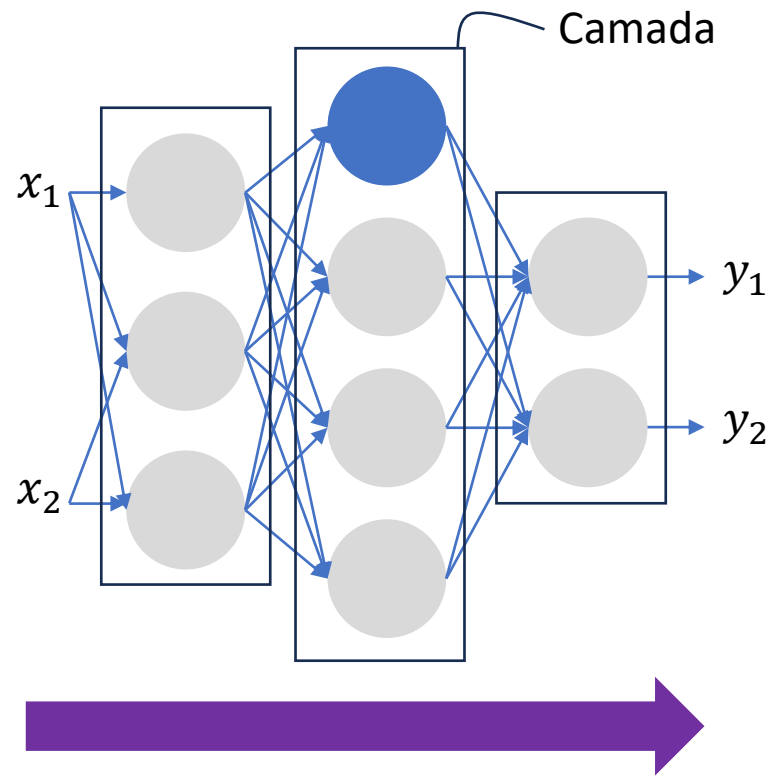


# Relembrando alguns termos



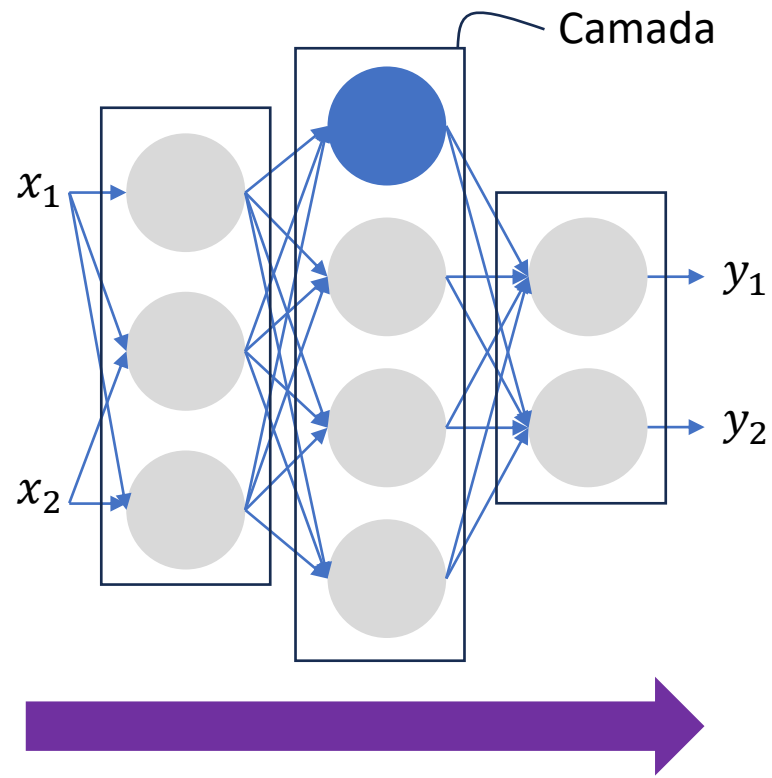
- Nós chamamos cada um dos **conjuntos de neurônios** em um retângulo de **camada**.
- A rede ao lado tem **duas camadas ocultas** e **uma camada de saída**.

# Relembrando alguns termos



- Em redes de **alimentação direta**, as camadas estão conectadas em **sequência**.
- As **informações fluem em uma única direção**, ou seja, elas fluem da entrada para as camadas ocultas e, finalmente, para a camada de saída, sem recursões.

# Relembrando alguns termos



- Na terminologia do Tensorflow, nós usamos o termo **sequential** para definir este tipo de rede.
- Além disso, vemos que as **saídas de uma camada estão conectadas a todos os neurônios da próxima camada**, criando conexões **densas**.
- Usaremos o termo **densas** para definir estes tipos de camadas da rede.

# Definindo uma rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Voltando ao código, nós vemos o termo *sequential* e, portanto, temos a definição de uma rede de alimentação direta ou sequencial.
- Dentro dos colchetes, *listamos as camadas* dessa rede neural.
- Nesse exemplo, a *lista* contém apenas um elemento, portanto, temos apenas uma *camada*.

# Definindo uma rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Essa camada é do tipo **densa** (classe *Dense*), então sabemos que é uma rede **densamente conectada**.
- O parâmetro **units** define quantos neurônios a camada possui.
- Podemos ver que essa camada tem **apenas um neurônio**.
- Portanto, esse código define a rede neural mais simples possível.
  - Há apenas uma camada com um único neurônio nela.

# Definindo uma rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Um parâmetro que definimos ***apenas para a primeira camada*** (neste caso a única) de uma rede neural é o formato (i.e., ***dimensões***) das ***entradas***.
- No exemplo, o parâmetro ***input\_shape*** tem o valor 1, que indica a dimensão da entrada, ou seja, a ***quantidade de atributos de entrada*** do neurônio.
- Isso significa que o neurônio tem apenas uma entrada, o atributo  $x$ .

# Definindo uma rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Quando vimos o funcionamento dos neurônios, aprendemos que eles possuem uma ***função de ativação***, que tem como entrada a ***combinação ponderada das entradas pelos pesos sinápticos mais o peso de bias*** e que faz um ***mapeamento não linear*** de sua entrada na saída.
- Porém, como queremos encontrar um ***mapeamento linear entre a entrada e a saída, não usaremos*** nenhuma ***função de ativação***.

# Definindo uma rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

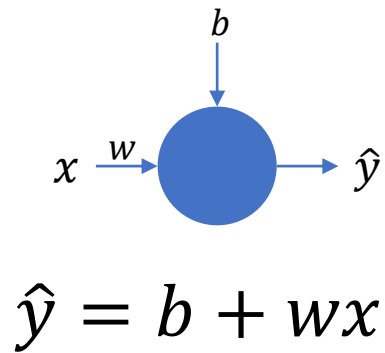
```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Lembrem-se nosso objetivo é encontrar os pesos de uma função hipótese, do tipo  $\hat{y} = b + wx$ , modelada pela rede neural.
- A função de ativação pode ser definida através do parâmetro *activation* da classe **Dense**.
- Por padrão, a ativação é definida como **None**, ou seja, não se tem ativação.

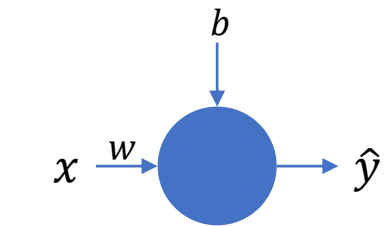


# Nossa rede neural



- Visualmente, nossa rede neural se parece com a figura ao lado.
- Nós temos apenas ***uma camada*** com um ***único neurônio*** nela.
- O neurônio tem apenas uma entrada, o atributo,  $x$ .
  - Por isso a dimensão da entrada é igual a 1.

# Nossa rede neural



$$\hat{y} = b + wx$$

- Nenhuma função de ativação é definida.
  - Isso se deve ao fato de queremos encontrar um **mapeamento linear**.
- Ele irá aprender os pesos ( $w$  e  $b$ ) que mapeiam  $x$  em  $y$  da **melhor forma possível**, baseado no **conjunto de treinamento**.
- Esse modelo é conhecido na literatura como **Perceptron**.

# Compilando a rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Depois de termos definido o modelo, precisamos compilá-lo.
- Ao compilarmos o modelo, devemos definir a ***função de erro (loss)*** e um ***otimizador***.
- Como antes, usaremos o ***erro quadrático médio*** como função de erro.
- **Obs.:** O Tensorflow se encarrega de realizar os cálculos de erro, portanto, não precisamos nos preocupar com isso.

# Compilando a rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- O **otimizador** é o SGD, que significa *Stochastic Gradient Descent*.
- Na verdade, ele implementa o **gradiente descendente em mini-lotes**.
- Ele segue o processo que vimos anteriormente, o qual calcula o vetor gradiente usando um subconjunto de amostras do conjunto de treinamento.
- O SGD usa o vetor gradiente para caminhar pela (descer a) superfície de erro até atingir **um** mínimo.

# Treinando a rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- O treinamento propriamente dito é feito através do método *fit()*.
- O que ele faz é atualizar **iterativamente** os pesos do modelo usando o SGD.
- O treinamento é feito por 500 épocas.
- Uma **época** representa **uma passagem completa pelo conjunto de treinamento** durante o treinamento do modelo.

# Treinando a rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Uma *iteração* representa *uma atualização dos pesos*.
- Assim, uma época é um conjunto de iterações.
- Por exemplo, se o conjunto de treinamento tem 1000 exemplos e está sendo treinado em *mini-batches* de 100 exemplos, então uma época é composta por 10 iterações, i.e., os *pesos são atualizados 10 vezes*.
- Por padrão, o tamanho do *mini-batch* é igual a 32.

# Treinando a rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- O tamanho do *mini-batch* é definido através do parâmetro *batch\_size* do método *fit()*.
- Porém, como no código acima o número de exemplos de treinamento é menor do que o tamanho padrão, *1 iteração se torna igual a 1 época*.
- Ou seja, todos os exemplos são usados para se atualizar os pesos → **GDB**.

# Treinando a rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Cada *iteração* de atualização executa as etapas que discutimos antes:
  - dá-se um palpite a respeito dos valores dos pesos (inicialmente, eles são aleatórios),
  - mede-se o erro causado por esse palpite,
  - atualiza-se os pesos usando-se o gradiente do erro,
  - e repete-se o processo, aqui, até se completar as 500 épocas.



# Treinando a rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Percebamos que não estamos fazendo, nem faremos, nenhum cálculo nós mesmos (i.e., cálculo do erro, vetor gradiente, atualizações).
- Nós deixaremos o TensorFlow fazer isso por nós.
- Essa é uma das grandes vantagens do TensorFlow, principalmente quando temos redes neurais profundas.

# Treinando a rede neural

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Ao final do treinamento, o método *fit()* retorna um objeto do tipo *History*.
- Ele contém, além de outros parâmetros, um **dicionário** chamado de **history** contendo o **erro** e **todas as métricas extras** medidas **ao final de cada época** no conjunto de treinamento e no conjunto de validação (se houver).

# Realizando previsões

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Depois de treinado, podemos usar o modelo para prever o valor de  $y$  para um determinado  $x$ .
- A previsão também é chamada de ***inferência***.
- Previsões são feitas com o método *predict()*.

# Realizando previsões

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])
```

```
model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input_shape=[1])])  
model.compile(optimizer='sgd', loss='mean_squared_error')
```

```
history = model.fit(x, y, epochs=500)
```

```
print(model.predict([10.0]))
```

- Qual valor seria predito e impresso para  $x = 10$ ?
- Sabemos das aulas anteriores que a melhor função hipótese é dada por  $\hat{y} = -1 + 2x$ .
- Assim, quando  $x = 10$ , o valor de  $\hat{y}$  deveria ser 19. Correto?
- Vamos ver no próximo exemplo, que, surpreendentemente, a resposta é não.

# Exemplo

- [Regressão com DNNs](#)



TensorFlow



# Atividades

- Quiz: “***TP557 – Regressão com DNNs (Parte I)***”.
- Exercício: [Regressão com DNNs \(Parte I\)](#).

Perguntas?

Obrigado!



