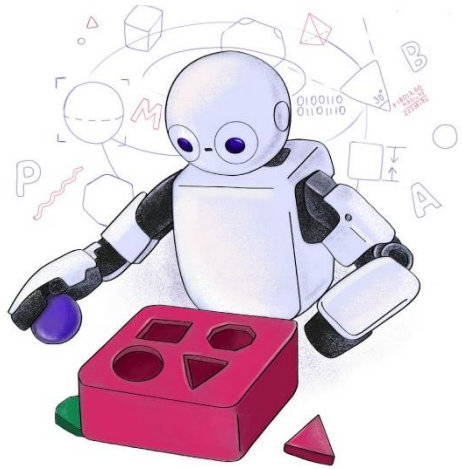
TP557 - Tópicos avançados em IoT e Machine Learning:

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 (valores aleatórios);
  - Mede-se o erro com esse palpite;
  - Usa-se a informação obtida através do erro (vetor gradiente) para melhorar o palpite;
  - Repete-se o processo 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 codificar uma rede neural que atinge o mesmo objetivo anterior, encontrar uma função que aproxime um conjunto de dados.

Conjunto de dados de treinamento

$$\boldsymbol{x} = \{-1, 0, 1, 2, 3, 4\}$$
$$\boldsymbol{y} = \{-3, -1, 1, 3, 5, 7\}$$

- Ao lado temos o conjunto de dados que usamos anteriormente.
- Nosso objetivo é encontrar um modelo que mapeie os valores de $\boldsymbol{x}$ em $\boldsymbol{y}$ de forma ótima (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 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.

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]))
```

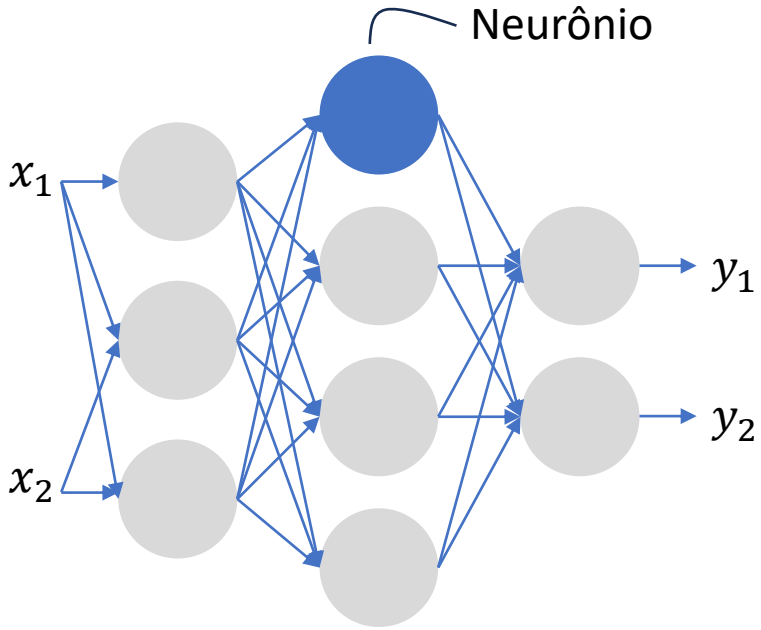
- As primeiras duas linhas definem o conjunto de treinamento.
- Ou seja, os valores de x e y que usaremos para otimizar o modelo durante as iterações e épocas de treinamento.
- Cada valor de x corresponde a um valor de y.
- O Tensorflow espera que o conjunto de dados sejam *arrays* NumPy.

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]))
```

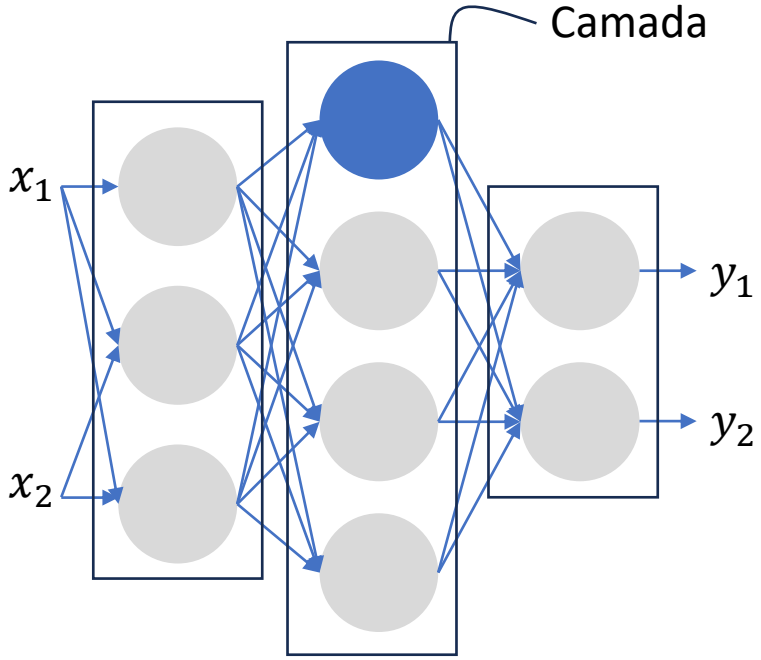
- Na sequência, temos a definição da rede neural.
- É uma rede neural muito simples, uma das mais simples que veremos.
- Antes de discutirmos o código, vamos relembrar alguns termos para que possamos entendê-lo.

Relembrando alguns termos



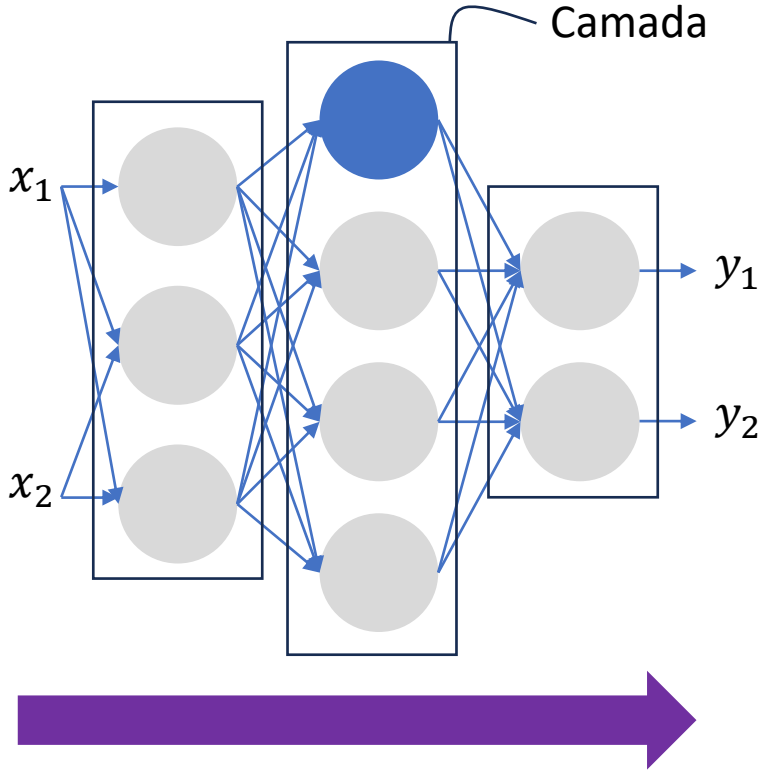
- Cada um dos círculos ao lado é um neurônio ou nó.

Relembrando alguns termos



- Cada um desses conjuntos de neurônios é uma camada.
- A rede ao lado tem duas camadas ocultas e uma camada de saída.

Relembrando alguns termos



- As camadas estão conectadas em sequência.
- As informações fluem em uma única direção, ou seja, da entrada para as camadas ocultas e, finalmente, para a camada de saída, sem recursões.
- Na terminologia do Tensorflow, nós usamos o termo ***sequential*** para definir esta rede.
- Além disso, vemos que as saídas de uma camada estão conectadas a todos os neurônios da próxima camada, criando conexões ***densas***.
- Usaremos esse termo para definir o nome e tipos das 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/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***, então sabemos que é uma rede ***densamente conectada***.
- O parâmetro ***units*** nos diz 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.

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 precisamos definir apenas para a primeira (neste caso única) camada 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 se sinais de entrada do neurônio.
- Isso significa que o neurônio tem apenas uma entrada, o valor de 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]))
```

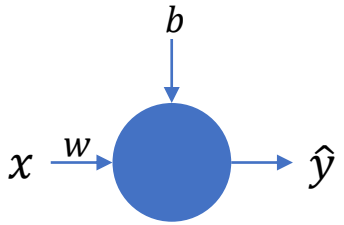
- Quando vimos o funcionamento dos neurônios, aprendemos que eles possuem uma ***função de ativação***, que têm como entrada a combinação ponderada das entradas pelos pesos sinápticos mais o peso de bias e que faz um ***mapeamento não linear*** de sua entrada na saída.
- Porém, como queremos encontrar um ***mapeamento linear entre entrada e 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 que o que queremos é encontrar os pesos de uma função hipótese do tipo $\hat{y} = b + wx$.
- A função de ativação é 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



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

- Visualmente, nossa rede neural se parece com a figura ao lado.
- Nós temos apenas uma camada e um único neurônio nela.
- Ele tem como entrada um único valor, que chamamos de x.
  - Por isso a dimensão da entrada é igual a 1.
- Nenhuma função de ativação é definida.
- Ele irá aprender os pesos 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 (perda)*** e um ***otimizador***.
- A função de erro é o ***erro quadrático médio***, como usamos anteriormente.
- O Tensorflow se encarrega de realizar os cálculos do erro, não precisamos nos preocupar.

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 SGD, que significa *Stochastic Gradient Descent*.
- Ele segue o processo que vimos anteriormente, o qual usa o vetor gradiente para caminhar (descer a) pela curva de erro até atingir um mínimo.
- Percebam que não estamos fazendo, nem faremos, nenhum cálculo nós mesmos. Nós deixaremos o TensorFlow fazer isso por nós.

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 encontrar um mapeamento dos valores de x em y.
- 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 o número de atualizações de pesos que ocorrem durante o treinamento do modelo no conjunto de treinamento.
- 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.
- 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, todo os exemplos são usados para se atualizar os pesos.

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 executa as etapas que discutimos antes:
  - dá-se um palpite a respeito dos valores dos pesos,
  - mede-se o erro causado por aquele palpite,
  - atualiza-se os pesos,
  - 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]))
```

- Ao final do treinamento, o método *fit()* retorna um objeto do tipo *History* que contém, além de outros parâmetros, um dicionário chamado de *history* com 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 predições

```
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 predizer o valor de y para um determinado x.
- A predição também é chamada de ***inferência***.
- Predições são feitas com o método *predict()*.

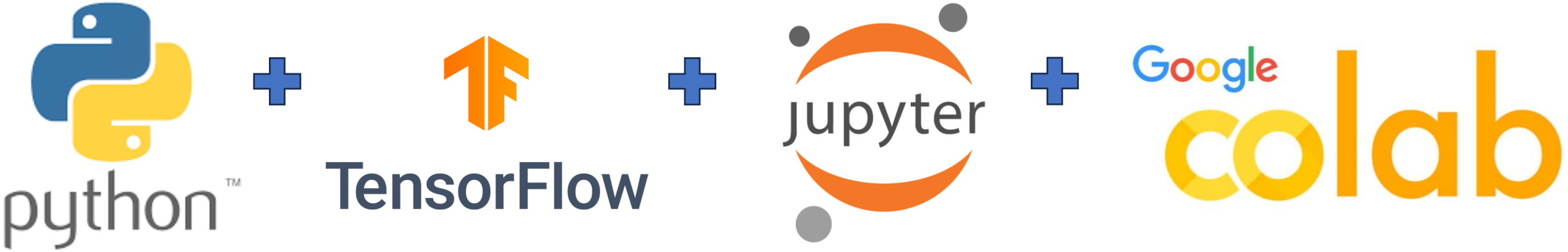
Realizando predições

```
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 vocês esperam que seja predito e impresso para uma entrada $x = 10$?
- Seria 19?
- Vamos ver no próximo exemplo, que, surpreendentemente, a resposta é não.

Exemplo

- Regressão com DNNs

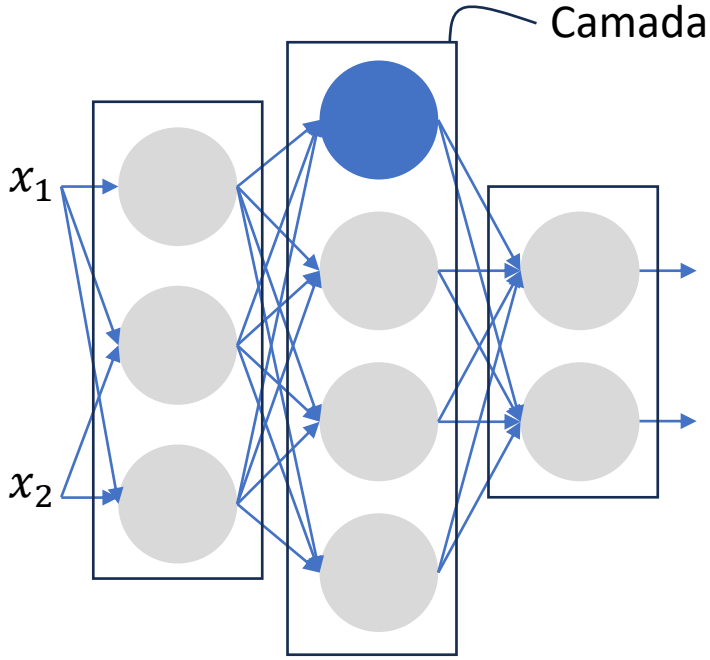


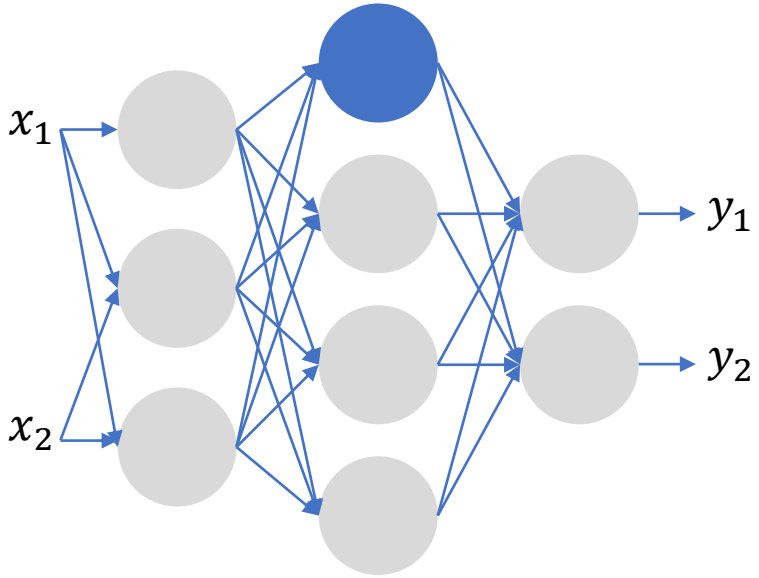
Atividades

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

Perguntas?

Obrigado!


Camada
x_1
x_2


x_1
x_2
y_1
y_2