

TP555 - Inteligência Artificial e Machine Learning: *TensorFlow (v1.x)*



Inatel

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

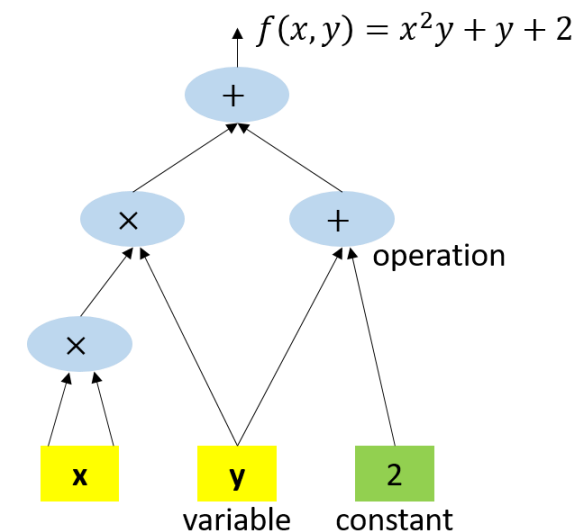
TensorFlow

- Neste tópico, veremos alguns conceitos básicos da biblioteca ***TensorFlow***,
 - Instalação.
 - Criação, execução, armazenamento e visualização de ***grafos computacionais*** simples.
- É importante dominarmos essas noções básicas antes de criarmos nossa primeira rede neural com o TensorFlow.
- **Instalação:**
 - Via interface gráfica do Anaconda
 - Via terminal
 - `conda install tensorflow`
 - `pip3 install --upgrade tensorflow`
- **OBS.:** Para suporte à GPU, você precisa instalar o ***tensorflow-gpu*** ao invés do ***tensorflow***.
- Para testar se a instalação foi bem sucedida, digite o comando abaixo. Ele deve imprimir a versão do TensorFlow que foi instalada.
 - `python3 -c 'import tensorflow; print(tensorflow.__version__)'`

TensorFlow



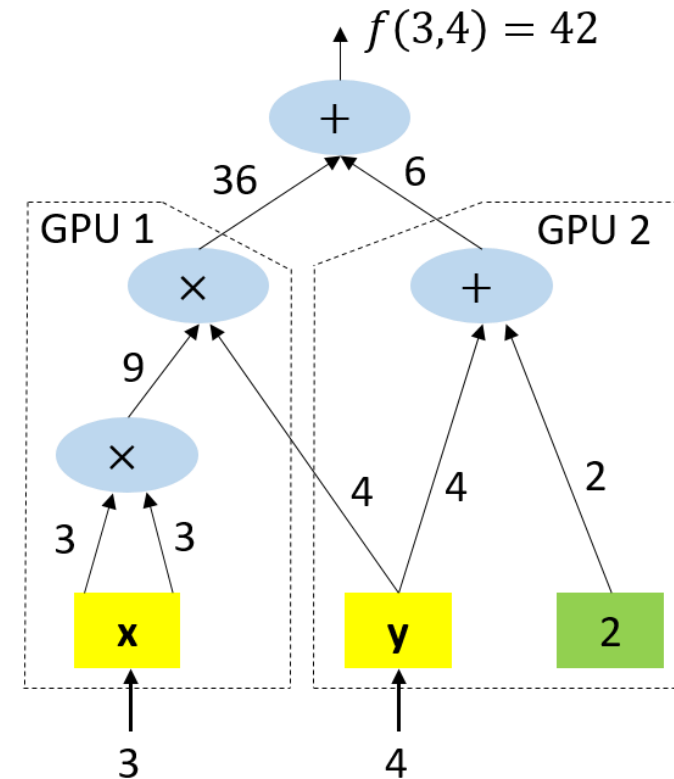
- O **TensorFlow** é uma poderosa **biblioteca de software de código aberto** para computação numérica, adequada e customizada para a execução de algoritmos de aprendizado de máquina em larga escala.
- O nome **TensorFlow** deriva dos tipos de operações que a biblioteca pode realizar em **matrizes de dados multidimensionais**, conhecidas como **tensores**.
- A biblioteca tem 2 grandes releases: 1.x e 2.x.
 - **1.x**: Lançada em Fevereiro de 2017. A escrita do código é dividida em duas partes: construção do **grafo** computacional e posteriormente criação de uma **sessão** para executá-lo. Forma complexa e não Pythonica...
 - **2.x**: Lançada em Setembro 2019. Facilita a criação de modelos através de **eager execution**, não é mais necessário se criar uma **sessão** para executar o **grafo**, pode-se verificar o resultado do código diretamente sem a necessidade de se criar uma **sessão**.
- Seu princípio básico de funcionamento é bem simples: primeiro, define-se em **Python** um **grafo de computação** a ser executado (como mostrado na figura ao lado) e, em seguida, o **TensorFlow** transforma esse **grafo** em código C++ otimizado e o executa com eficiência em uma **sessão**.



Grafo de computação

TensorFlow

- O **TensorFlow** possibilita dividir o **grafo** em várias seções e executá-las em paralelo em várias CPUs ou GPUs, como mostrado na figura ao lado.
- O **TensorFlow** também suporta **computação distribuída**: pode-se treinar **redes neurais** gigantescas com **conjuntos de treinamento** imensos em um período de tempo razoável, dividindo-se os cálculos através de centenas de servidores.
- Por exemplo, o **TensorFlow** pode treinar uma **rede neural** com milhões de **pesos** com um conjunto de treinamento composto por bilhões de exemplos com milhões de **atributos** cada.
- O **TensorFlow** foi desenvolvido pelo time da **Google** chamado de **Google Brain** e é utilizado internamente e em vários produtos da empresa, e.g., Google Photos, Search, Maps, entre outros.



TensorFlow

- O **TensorFlow** foi projetado para ser *flexível, escalável e pronto para produção*. Alguns de seus destaques são:
 - Roda não apenas no Windows, Linux e macOS, mas também em dispositivos móveis e embarcados, incluindo iOS, Android e Raspberry Pi.
 - Fornece uma *application programming interface* (API) em **Python** muito simples chamada **TFLearn** (*tensorflow.contrib.learn*) que é compatível com o Scikit-Learn.
 - Também fornece outra API simples chamada **TF-slim** (*tensorflow.contrib.slim*) para simplificar a criação, o treinamento e a validação de *redes neurais*.
 - Existem várias APIs de alto nível que foram construídas sobre o **TensorFlow**, como **Keras** ou **Pretty Tensor**, que facilitam seu uso em detrimento de uma menor flexibilidade.
 - Entretanto, as APIs originais do **TensorFlow** oferecem muito mais flexibilidade (ao custo de maior complexidade) para criar todos os tipos de grafos de computação, incluindo qualquer arquitetura de *rede neural* que você possa imaginar.

TensorFlow

- Inclui implementações em C++ altamente eficientes de muitas operações de aprendizado de máquina, particularmente aquelas necessárias para construir **redes neurais**. Há também uma API em C++ para que os usuários definam suas próprias operações de alto desempenho.
- Fornece vários **nós** de otimização para encontrar os **parâmetros** (i.e., pesos) que minimizam uma **função de custo** (ou de **erro**). Eles são muito fáceis de usar, pois o **TensorFlow** cuida automaticamente do cálculo dos gradientes das funções que você define. Isso é chamado de **diferenciação automática** (ou **autodiff**).
- Oferece uma ferramenta de visualização chamada **TensorBoard**, que permite navegar pelo **grafo de computação** e visualizar curvas de aprendizado.
- Possui uma equipe dedicada de desenvolvedores e uma comunidade crescente que contribui para melhorá-lo cada vez mais.

Criando e executando um grafo simples

```
import tensorflow as tf
```

```
# Creating the graph.
```

```
x = tf.Variable(3, name="x")
```

```
y = tf.Variable(4, name="y")
```

```
f = x*x*y + y + 2
```

```
# Executing the calculation graph.
```

```
sess = tf.Session()
```

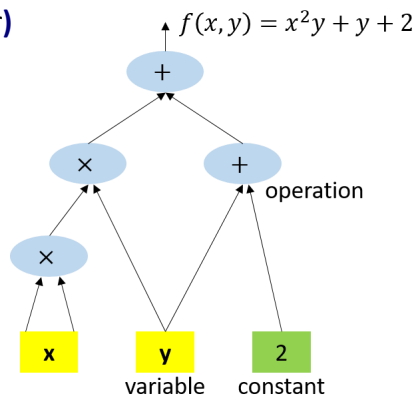
```
sess.run(x.initializer)
```

```
sess.run(y.initializer)
```

```
result = sess.run(f)
```

```
print(result)
```

```
sess.close()
```



```
with tf.Session() as sess:
```

```
    x.initializer.run()
```

```
    y.initializer.run()
```

```
    result = f.eval()
```

Example: [FirstGraph.ipynb](#)

- A primeira parte do código ao lado cria um **grafo de computação** representando a figura abaixo.
- **Importante:** a primeira parte do código, não executa nenhum cálculo, ela apenas cria (ou define) um **grafo de computação**. Na verdade, nem mesmo as variáveis foram inicializadas ainda.
- Para avaliar esse **grafo**, é necessário se criar uma **sessão** do **TensorFlow** e usá-la para inicializar as **variáveis** e avaliar a função f (ou seja, o grafo).
- Uma **sessão** do **TensorFlow** é responsável por carregar as operações em CPUs ou GPUs, executá-las, e manter os valores das variáveis.
- A segunda parte do código ao lado, cria uma sessão, inicializa as variáveis, avalia a função f e finaliza a **sessão** (o que libera recursos: CPUs, GPUs e memória).

Dicas

- Ter que repetir **sess.run()** o tempo todo é um pouco chato, mas felizmente existe uma maneira melhor, que é mostrada no trecho de código ao lado.
- Dentro do bloco da instrução **with**, a **sessão** é definida como a **sessão padrão**. E portanto, executar **x.initializer.run()** é equivalente a executar **tf.get_default_session().run(x.initializer)** e da mesma forma **f.eval()** é equivalente a executar **tf.get_default_session().run(f)**. Isso facilita a leitura do código. Além disso, a **sessão** é finalizada automaticamente ao final do bloco.

Criando e executando um grafo simples

Create an init node

```
init = tf.global_variables_initializer()
```

```
with tf.Session() as sess:
```

```
    # actually initialize all the variables
```

```
    init.run()
```

```
    result = f.eval()
```

Dica


- Ao invés de executar manualmente a inicialização de cada variável, como fizemos no exemplo anterior, nós podemos usar a função ***global_variables_initializer()***.
- Observe que essa função, na verdade, não executa a inicialização imediatamente, mas cria um ***nó*** no ***grafo*** que inicializará todas as variáveis quando for executado, conforme mostrado no trecho de código ao lado.

- Conforme percebemos, um programa utilizando o ***TensorFlow*** normalmente é dividido em duas partes:
 - A primeira parte cria um ***grafo de computação*** (chamada de ***fase de construção***).
 - A segunda parte executa o ***grafo*** (chamada de ***fase de execução***).
- A ***fase de construção*** cria um ***grafo de computação*** representando (declaração) o modelo de aprendizado de máquina e os cálculos necessários para treiná-lo.
- Já a ***fase de execução***, executa o ***grafo de computação***. Essa fase pode executar um laço de repetição que avalia uma etapa de treinamento de um modelo repetidamente (por exemplo, uma época de treinamento por batelada), melhorando gradualmente os pesos do modelo.

Gerenciando grafos

```
x1 = tf.Variable(1)
x1.graph is tf.get_default_graph()
True
```

```
graph = tf.Graph()
with graph.as_default():
    x2 = tf.Variable(2)
```



```
x2.graph is graph
True
```

```
x2.graph is tf.get_default_graph()
False
```

- Qualquer **nó** criado é adicionado automaticamente ao **grafo padrão**. Podem existir vários grafos, mas apenas um é o padrão.

- Na maioria dos casos, isso é bom, mas às vezes nós podemos querer gerenciar vários **grafos** independentes.
- Nós podemos fazer isso criando um novo **grafo** e tornando-o temporariamente o **grafo padrão** dentro de um bloco **with**, como mostrado no trecho de código ao lado.

Dica

- No Jupyter, é comum executarmos os mesmos comandos mais de uma vez enquanto estamos testando um código. Como resultado, podemos acabar com um **grafo padrão** contendo muitos **nós** duplicados. Uma solução é reiniciar o kernel do Jupyter, porém, uma solução mais conveniente é apenas redefinir o **grafo padrão** executando o comando **tf.reset_default_graph()**.

Ciclo de vida do valor de um nó

```
import tensorflow as tf
```

```
w = tf.constant(3)
```

```
x = w + 2
```

```
y = x + 5
```

```
z = x * 3
```

```
with tf.Session() as sess:
```

```
    print(y.eval()) # 10
```

```
    print(z.eval()) # 15
```

- Quando um **nó** é avaliado, o **TensorFlow** determina automaticamente o conjunto de **nós** dos quais ele depende e avalia esses outros **nós** primeiro.
- Por exemplo, no código do **grafo** ao lado, inicialmente se define o **grafo** e em seguida, inicia-se uma sessão para executá-lo e se avaliar o valor de y .
- No caso do y , o **TensorFlow** detecta automaticamente que ele depende de x , que depende de w , então ele avalia primeiro o valor de w , depois o de x , então o de y , que é então retornado.
- Em seguida, o valor de z é avaliado através de nova execução do grafo. Mais uma vez, o **TensorFlow** detecta que ele deve primeiro avaliar os valores de w e x , consecutivamente.
- **OBS.:** É importante ressaltar que o **TensorFlow** não reutilizará o resultado da avaliação anterior de x e w , ou seja, o **TensorFlow** avalia x e w duas vezes.

Ciclo de vida do valor de um nó

```
import tensorflow as tf
```

```
w = tf.constant(3)
```

```
x = w + 2
```

```
y = x + 5
```

```
z = x * 3
```

```
with tf.Session() as sess:
```

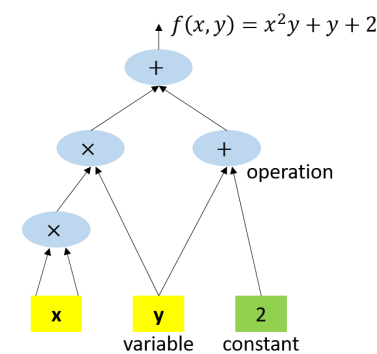
```
    y_val, z_val = sess.run([y, z])
```

```
    print(y_val) # 10
```

```
    print(z_val) # 15
```

- Se desejarmos avaliar y e z eficientemente, sem avaliar w e x duas vezes como no código anterior, nós devemos orientar o **TensorFlow** para avaliar y e z em apenas uma execução do **grafo**, conforme mostrado no código ao lado.
- **Observações:**
 - Todos os valores de um **nó** são eliminados entre as execuções do **grafo**, exceto os valores de **variáveis**, os quais são mantidos pela **sessão** entre as execuções do **grafo**.
 - Uma **variável** inicia sua vida útil quando seu inicializador é executado e termina quando a **sessão** é encerrada.

Informações Importantes



- As **operações** do **TensorFlow** (abreviadas como **ops**) podem receber qualquer número de entradas (i.e., atributos) e produzir qualquer número de saídas (i.e., rótulos ou valores desejados).
- Por exemplo, as **operações** de adição e multiplicação do **grafo** acima recebem duas entradas e produzem uma saída.
- **Constantes** e **variáveis** não recebem entradas, sendo então, chamadas de operações de **origem** (ou do Inglês **source**).
- As entradas e saídas das operações são arrays multidimensionais, denominadas **tensores** (do Inglês **tensors**) (daí o nome “**tensor flow**”).
- Assim como as arrays da biblioteca **NumPy**, os **tensores** têm um **tipo** e uma **forma** (i.e., dimensão). Na verdade, em Python, os **tensores** são simplesmente representados por arrays do tipo **ndarrays** da biblioteca **NumPy**.
- Essas arrays geralmente contêm **floats**, mas podemos também usá-las para armazenar **strings**, **integers**, etc.

[Exemplo: InputsAndOutputs.ipynb](#)

Regressão Linear com TensorFlow: Eq. Normal

```
import numpy as np
from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing()
m, n = housing.data.shape
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]

X = tf.constant(housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
XT = tf.transpose(X)
theta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)

with tf.Session() as sess:
    theta_value = theta.eval()
```

Exemplo: [NLinearRegressionTF.ipynb](#)

- Nos exemplos anteriores, os **tensores** continham apenas um valor **escalar**, mas também é possível executar cálculos em arrays de qualquer formato (i.e., arrays multidimensionais).
- Por exemplo, o código ao lado manipula arrays 2D (i.e., matrizes) para realizar **regressão linear** com o conjunto de dados de valores de casas no estado da Califórnia.
- O código começa baixando o conjunto de dados. Em seguida, adiciona um **atributo** extra de entrada, o **bias** ($x_0 = 1$), a todos os exemplos de treinamento (faz-se isso usando a biblioteca **NumPy** e, portanto, esse trecho é executado imediatamente).
- Em seguida, dois **nós constantes**, X e y , são criados para armazenar os atributos e os respectivos rótulos.
- Finalmente, o código usa algumas operações de matrizes implementadas pelo **TensorFlow** para definir o cálculo de **theta**.
- Lembre-se que até aqui apenas se definiu o **grafo**.

Regressão Linear com TensorFlow: Eq. Normal

```
import numpy as np
from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing()
m, n = housing.data.shape
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]

X = tf.constant(housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
XT = tf.transpose(X)
theta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)

with tf.Session() as sess:
    theta_value = theta.eval()
```

- As funções matriciais: ***transpose()***, ***matmul()*** e ***matrix_inverse()***, são autoexplicativas, mas como discutido antes, elas não realizam cálculos imediatamente, em vez disso, o **TensorFlow** cria **nós** no **grafo** que as avaliarão quando o **grafo** for executado.
- Nós podemos reconhecer que a definição de ***theta*** corresponde à **equação normal**:
$$\hat{\theta} = (X^T X)^{-1} X^T y.$$
- Finalmente, o código cria uma **sessão** e a utiliza para avaliar o valor de ***theta***.
- **OBS.:** O principal benefício desse código em comparação ao cálculo direto da **equação normal** usando a biblioteca **NumPy** é que o **TensorFlow** o executará automaticamente em sua placa de vídeo GPU, caso você tenha uma e que tenha instalado o **TensorFlow** com suporte a GPUs.

Regressão Linear com TensorFlow: GD

- No próximo exemplo nós vamos usar o ***gradiente descendente em batelada*** ao invés da ***equação normal*** para encontrar os pesos do modelo.
- Inicialmente, faremos isso calculando os gradientes manualmente, em seguida, usaremos um recurso poderosíssimo do ***TensorFlow*** conhecido como ***autodiff***, o qual permite que o ***TensorFlow*** calcule os ***gradientes automaticamente*** e, finalmente, usaremos alguns ***otimizadores*** prontos disponibilizados pelo ***TensorFlow***.

Calculando os gradientes manualmente

```
n_epochs = 1000
learning_rate = 0.01
```

```
X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)
```

```
init = tf.global_variables_initializer()
```

```
with tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
            sess.run(training_op)
        best_theta = theta.eval()
```

```
Epoch 0 MSE = 12.34403
Epoch 100 MSE = 0.84830046
Epoch 200 MSE = 0.6261495
Epoch 300 MSE = 0.60210127
Epoch 400 MSE = 0.5865569
Epoch 500 MSE = 0.5744065
```

Exemplo: [GDLinearRegressionTF.ipynb](#)

- O código ao lado é bastante auto-explicativo, exceto por alguns detalhes:
 - A função ***random_uniform()*** cria um ***tensor*** contendo valores aleatórios uniformemente distribuídos.
 - A função recebe a dimensão e a faixa de valores desejados, similarmente ao que é feito com a função ***rand()*** da biblioteca NumPy.
 - Em seguida, cria-se um ***nó*** do tipo variável no ***grafo*** com este ***tensor***.
 - A função ***tf.reduce_mean()*** calcula a média dos elementos de uma array.
 - A função ***assign()*** cria um ***nó*** que atribui um novo valor a uma ***variável***. Nesse caso, ela implementa a regra de atualização dos pesos do ***gradiente descendente em batelada***:
 - $\theta^{(\text{next step})} = \theta - \alpha \nabla_{\theta} \text{MSE}(\theta)$.
 - O laço de repetição principal, dentro da sessão, executa a regra de atualização dos pesos repetidamente (por ***n_epochs*** vezes) e a cada 100 iterações imprime o erro quadrático médio (MSE) atual.
 - Como é esperado, o MSE diminui a cada iteração.

Usando *autodiff* para calcular os gradientes

- O código anterior funciona bem, mas requer que os gradientes da ***função de custo*** sejam derivados manualmente.
- No caso da ***regressão linear***, isso é bem fácil, mas se tivéssemos que fazer isso para ***redes neurais*** com várias camadas nós teríamos muita dor de cabeça: seria tedioso e bastante propenso a erros.
- Uma solução seria o uso de ***diferenciação simbólica*** para encontrar automaticamente as equações das derivadas parciais, mas o código resultante não seria eficiente.
- Felizmente, o ***TensorFlow*** disponibiliza um recurso muito útil, o ***autodiff*** (*automatic differentiation*), que calcula os gradientes de forma automática, eficiente e precisa.
- ***Autodiff*** é um conjunto de técnicas para avaliar numericamente a derivada de uma função especificada por um programa de computador.
- Entre as várias abordagens para se calcular gradientes automaticamente, o ***TensorFlow*** adota o ***reverse-mode autodiff***, que calcula os gradientes de forma eficiente e precisa quando há muitas entradas e poucas saídas, como costuma ocorrer com ***redes neurais***.

Usando *autodiff* para calcular os gradientes

- Para utilizar o *autodiff*, simplesmente substitua a linha:

```
gradients = 2/m * tf.matmul(tf.transpose(X), error)
```

no código anterior pela linha a seguir, o código continuará funcionando perfeitamente

```
gradients = tf.gradients(mse, [theta])[0]
```

- A função *gradients()* usa uma *op* (neste caso a operação *mse*) e uma lista de variáveis (nesse caso, apenas a variável *theta*). Na sequência, ela cria uma lista de *ops* (uma por variável) para calcular os gradientes da *op* em relação a cada variável.
- Portanto, o *nó gradients* calculará o *vetor gradiente* do MSE em relação ao vetor *theta*.

Usando otimizadores prontos

- Como vimos, o **TensorFlow** calcula os gradientes automaticamente. Além disso, ele também fornece vários **otimizadores** prontos para uso, incluindo um **otimizador** que implementa o algoritmo do **gradiente descendente**.
- Para usar o **otimizador** do tipo **gradiente descendente**, basta substituir as linhas

```
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)
```

pelo código

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

- Para usar um tipo diferente de **otimizador**, basta alterar uma linha. Por exemplo, podemos usar um **otimizador** do tipo **momentum** (que geralmente converge muito mais rápido que **otimizador** do tipo **gradiente descendente**) definindo o **otimizador** da seguinte maneira:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate, momentum=0.9)
```

Fornecendo dados aos grafos em tempo de execução

- Vamos modificar o código anterior para implementar o ***gradiente descendente em mini-batch***.
- Para isso, precisamos de uma maneira de substituir os valores das variáveis X e y a cada iteração pelo próximo mini-batch.
- A maneira mais simples de se fazer isso é usar ***nós*** conhecidos como ***placeholders***.
- Esses ***nós*** são especiais porque, na verdade, eles não realizam nenhum tipo de cálculo, eles apenas transferem os dados que você define em ***tempo de execução*** para o ***grafo*** sendo executado.
- Ou seja, eles são usados para passar os dados de treinamento (o mini-batch) para o ***TensorFlow*** durante o treinamento.

Fornecendo dados aos grafos em tempo de execução

```
A = tf.placeholder(tf.float32, shape=(None, 3))
B = A + 5
with tf.Session() as sess:
    B_val_1 = B.eval(feed_dict={A: [[1, 2, 3]]})
    B_val_2 = B.eval(feed_dict={A: [[4, 5, 6], [7, 8, 9]]})

print(B_val_1)
[[ 6.  7.  8.]]

print(B_val_2)
[[ 9. 10. 11.]
 [12. 13. 14.]]
```

- Para criar um **nó** do tipo **placeholder**, usamos a função **placeholder()** e especificar o tipo de dados do **tensor** de saída.
- Opcionalmente, também podemos especificar sua dimensão.
- Se especificarmos **None** para uma dimensão, isso significa "*qualquer tamanho*".
- Por exemplo, o código ao lado cria um **nó** A do tipo **placeholder** e também um **nó** B, que recebe o valor de A + 5.
- Quando avaliamos o valor do **nó** B, passamos um **feed_dict** para o método **eval()** do tensor B, o qual especifica os valores do **nó** A.
- Observem que o **nó** A deve ter 2 dimensões (ou seja, ele deve ser uma array bidimensional) e deve haver três colunas obrigatoriamente, mas ele pode ter qualquer número de linhas.

Fornecendo dados aos grafos em tempo de execução

```
X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")

batch_size = 100
n_batches = int(np.ceil(m / batch_size))

def fetch_batch(epoch, batch_index, batch_size):
    np.random.seed(epoch * n_batches + batch_index)
    indices = np.random.randint(m, size=batch_size)
    X_batch = scaled_housing_data_plus_bias[indices]
    y_batch = housing.target.reshape(-1, 1)[indices]
    return X_batch, y_batch

with tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        best_theta = theta.eval()
```

- Para implementar o ***gradiente descendente em mini-batch***, precisamos apenas modificar um pouco o código anterior.
- Primeiro devemos mudar a definição das variáveis X e y na fase de construção do ***grafo*** para torná-las ***nós*** do tipo ***placeholder***.
- Em seguida, definimos o tamanho de um ***mini-batch*** e calculamos o número total de batches.
- Por fim, na fase de execução, lemos os mini-batches um por um e os fornecemos aos nós do tipo placeholder, X e y , através do parâmetro ***feed_dict*** ao avaliar um ***nó*** que depende deles.

Salvando e restaurando modelos

```
reset_graph()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
            save_path = saver.save(sess, "/tmp/my_model.ckpt")
            sess.run(training_op)

    best_theta = theta.eval()
    save_path = saver.save(sess, "/tmp/my_model_final.ckpt")
```

- Depois de treinar um modelo, podemos salvar seus parâmetros em disco para poder utilizá-los sempre que quisermos.
- Nós podemos usá-los em outro programa, compará-los com outros modelos e assim por diante.
- Além disso, nós podemos querer salvar os parâmetros em intervalos regulares durante o treinamento do modelo, para que, se o computador travar durante o treinamento, nós possamos continuar do último ponto de verificação salvo em vez de começar do zero.
- O **TensorFlow** possibilita que se salve e restaure um modelo. Para isto, basta criar um **nó** do tipo **Saver** no final da **fase de construção**, ou seja, depois que todos os outros **nós** tiverem sido criados.
- Em seguida, na fase de execução, chama-se o método **save()** sempre que se desejar salvar o modelo, passando a **sessão** e o caminho para o arquivo onde se deseja salvar o ponto de verificação.

Salvando e restaurando modelos

- Restaurar um modelo salvo também é fácil: para isso deve-se criar um **nó** do tipo **Saver** no final da **fase de construção** como anteriormente, mas no início da **fase de execução**, ao invés de inicializar as variáveis usando o **nó de inicialização**, executa-se o método **restore()** do objeto do tipo **Saver** com a sessão corrente e o caminho para o arquivo com o modelo salvo, conforme mostrado no código abaixo.

```
with tf.Session() as sess:  
    saver.restore(sess, "/tmp/my_model_final.ckpt")  
    best_theta_restored = theta.eval()
```

- Por padrão, um objeto do tipo **Saver** salva e restaura todas as variáveis do grafo, mas se precisarmos de mais controle, nós podemos especificar quais variáveis salvar ou restaurar e quais nomes usar.
- Por exemplo, o objeto do tipo **Saver** no código abaixo salva ou restaura apenas a variável **theta** com o nome **weights**.

```
saver = tf.train.Saver({"weights": theta})
```


Visualizando grafos e curvas de treinamento com o TensorBoard

- Agora nós temos um **grafo de computação** que treina um modelo de **regressão linear** usando o algoritmo do **gradiente descendente em mini-batches** e que salva pontos de verificação em intervalos regulares.
- Parece bem avançado, não é? No entanto, ainda estamos confiando na função **print()** para visualizar o progresso do modelo durante o treinamento.
- Entretanto, existe uma maneira muito melhor para se avaliar o progresso do modelo: o **TensorBoard**.
- De posse de estatísticas de treinamento, o **TensorBoard** exibe visualizações interativas dessas estatísticas no navegador web (por exemplo, as **curvas de aprendizado**).
- Pode-se também fornecer a definição do **grafo de computação** e o **TensorBoard** apresentará uma interface para navegarmos pelo **grafo**.
- Isso é muito útil para identificar erros no **grafo**, encontrar gargalos de computação entre outras coisas.

Visualizando grafos e curvas de treinamento com o TensorBoard

- O primeiro passo para utilizar o **TensorBoard** é modificar o código para gravar a definição do **grafo** e algumas estatísticas de treinamento, como por exemplo, o erro de treinamento, em um diretório de log ao qual o **TensorBoard** terá acesso.
- **OBS.:** É necessário usar um diretório de log diferente toda vez que se executar o programa ou o **TensorBoard** irá misturar estatísticas de diferentes execuções, o que atrapalhará as visualizações e análises dos resultados.
 - A solução mais simples para isso é incluir um **timestamp** (i.e., data e hora) ao nome do diretório de log. Isso pode ser feito com o trecho de código abaixo.

```
from datetime import datetime
now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "tf_logs"
logdir = "{} /run-{} /".format(root_logdir, now)
```

Visualizando grafos e curvas de treinamento com o TensorBoard

- Em seguida, adicionamos o código abaixo ao final da **fase de construção** do **grafo**.

```
mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
```

- A primeira linha cria um **nó** no **grafo** que avalia o valor do **erro quadrático médio** (MSE) e o grava em um arquivo de log compatível com **TensorBoard** chamado de **summary**.
- A segunda linha cria um objeto do tipo **FileWriter** que é usado para escrever os resultados no arquivo de log.
 - O primeiro parâmetro indica o caminho do diretório de log e o segundo, que é opcional, é o **grafo** que você deseja visualizar.
 - Após a criação, o objeto do tipo **FileWriter** cria o diretório de log se ele ainda não existir e grava a definição do **grafo** em um arquivo de log chamado **arquivo de eventos**.

Visualizando grafos e curvas de treinamento com o TensorBoard

- Em seguida, é necessário atualizar o código da **fase de execução** para avaliar o **nó *mse_summary*** regularmente durante o treinamento (por exemplo, a cada 10 mini-batches).
- Isso produzirá um **summary** (i.e., o log) que pode ser gravado no arquivo de eventos usando o objeto **file_writer**.
- O código atualizado da **fase de execução** é mostrado abaixo.

```
[...]
for batch_index in range(n_batches):
    X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
    if batch_index % 10 == 0:
        summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
        step = epoch * n_batches + batch_index
        file_writer.add_summary(summary_str, step)
    sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
[...]
```

Visualizando grafos e curvas de treinamento com o TensorBoard

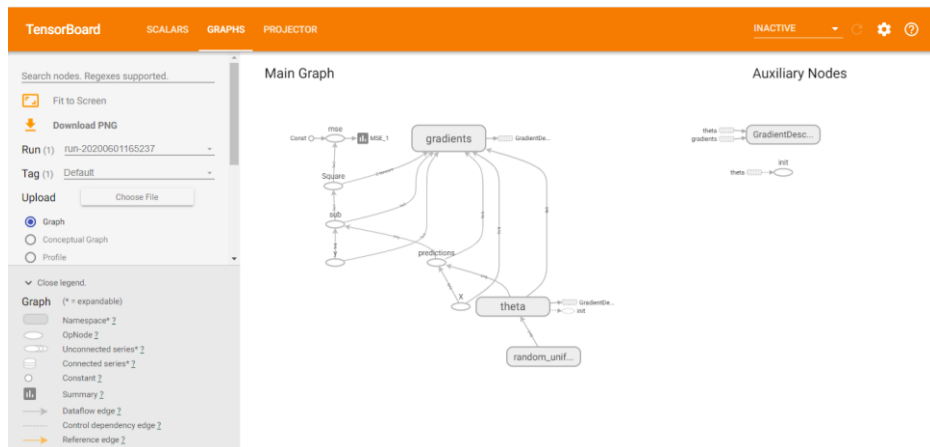
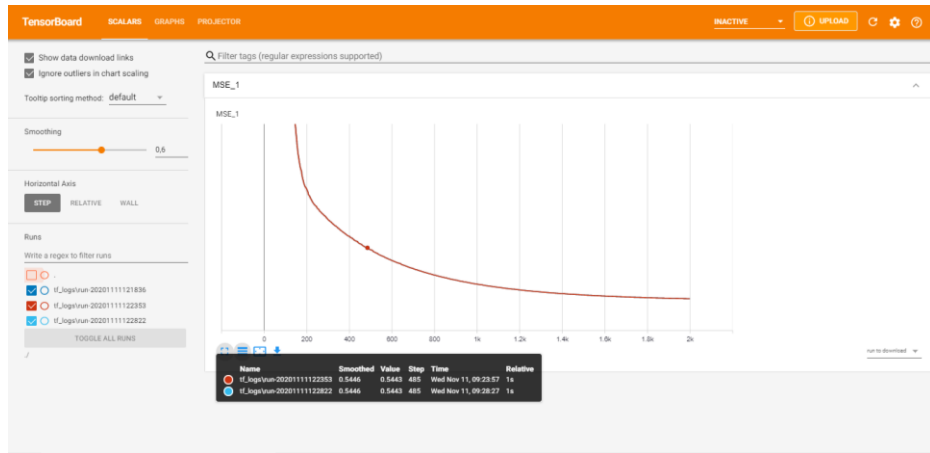
- Por fim, encerra-se o **FileWriter** no final do programa com ***file_writer.close()***.
- Ao executar o programa, o objeto do tipo **FileWriter** criará o diretório de log e gravará um **arquivo de eventos** nesse diretório, contendo a definição do **grafo** e os valores de MSE.
- Agora podemos iniciar o servidor do **TensorBoard**. Para isso, é necessário ativar o **ambiente virtual**, caso você tenha criado um e, em seguida, iniciar o servidor executando o comando **tensorboard**, apontando-o para o diretório de logs.
- Esse comando inicia o servidor web do **TensorBoard**, que fica *escutando* (i.e., esperando) por conexões na porta 6006.

```
$ source env/bin/activate
$ tensorboard --logdir tf_logs/
Starting TensorBoard on port 6006
(You can navigate to http://0.0.0.0:6006)
```

Podemos carregar também através do notebook:

```
%load_ext tensorboard
%tensorboard --logdir tf_logs/
```

Visualizando grafos e curvas de treinamento com o TensorBoard

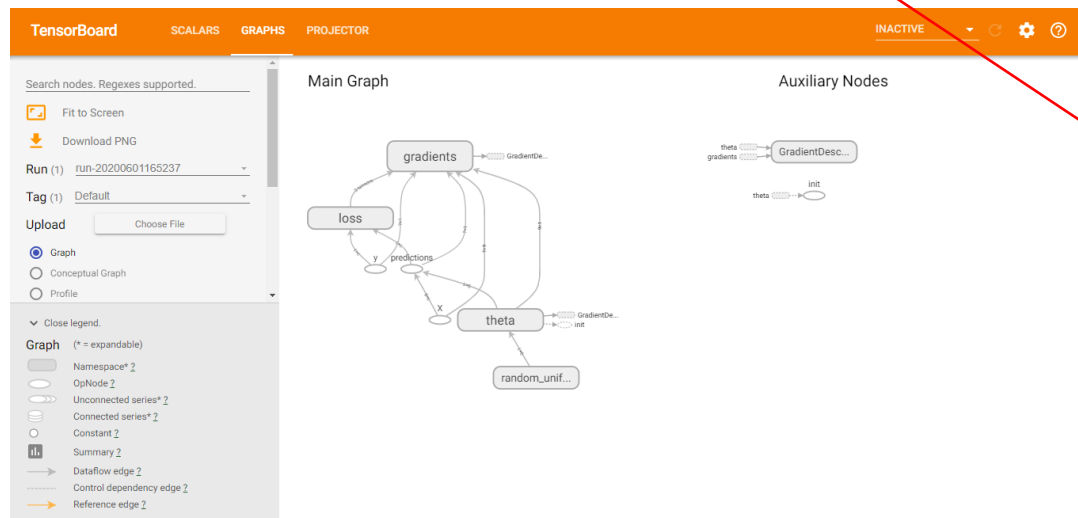


- Em seguida, abrimos um navegador web e acessamos ***http://0.0.0.0:6006/*** (ou ***http://localhost:6006/***).
- Na guia ***Scalars***, vemos o gráfico do MSE, o qual mostra sua evolução durante o treinamento.
- Nós podemos marcar ou desmarcar as execuções que desejamos ver, aumentar ou diminuir o zoom, passar o mouse sobre a curva para obter detalhes e assim por diante.
- Para visualizar o ***grafo***, basta clicar na guia ***Graphs***.

Escopos de nome

```
with tf.name_scope("loss") as scope:  
    error = y_pred - y  
    mse = tf.reduce_mean(tf.square(error), name="mse")
```

- Ao lidar com modelos mais complexos, como **redes neurais**, por exemplo, o **grafo** pode ficar muito complexo, com milhares de **nós**, dificultando sua visualização.
- Para evitar isso, podemos criar **escopos de nome** para agrupar **nós** relacionados.
- Por exemplo, vamos modificar o código anterior para definir as operações **error** e **mse** dentro de um **escopo de nome** chamado **loss** conforme mostrado no trecho de código ao lado.
- No **TensorBoard**, os **nós mse** e **error** agora aparecem dentro do **namespace loss**.



Modularidade

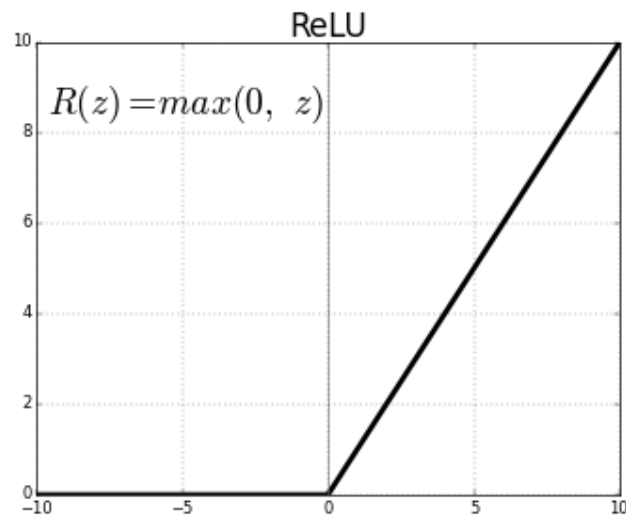
```
n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")

w1 = tf.Variable(tf.random_normal((n_features, 1)), name="weights1")
w2 = tf.Variable(tf.random_normal((n_features, 1)), name="weights2")
b1 = tf.Variable(0.0, name="bias1")
b2 = tf.Variable(0.0, name="bias2")

z1 = tf.add(tf.matmul(X, w1), b1, name="z1")
z2 = tf.add(tf.matmul(X, w2), b2, name="z2")

relu1 = tf.maximum(z1, 0., name="relu1")
relu2 = tf.maximum(z2, 0., name="relu2")

output = tf.add(relu1, relu2, name="output")
```



- Agora suponha que nós queiramos criar um **grafo** que tenha como resultado final a soma da saída de duas **unidades lineares retificadas (ReLU)**.
- **OBS.:** Uma **ReLU** calcula uma função linear das entradas e gera como saída o resultado da função linear caso este seja positiva, e 0 caso contrário. A equação da **ReLU** é mostrada abaixo e ilustrada pela figura ao lado.

$$h_a(X) = \max(Xa, 0).$$

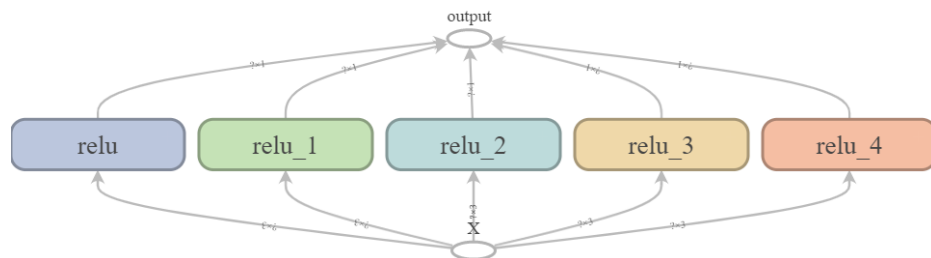
- O trecho de código ao lado realiza a tarefa, mas como podemos ver, é bastante repetitivo.
- Além disso, é difícil manter (i.e., dar manutenção) esse código repetitivo e, além disso, ele é bastante propenso a erros.
- Ficaria ainda pior se quiséssemos adicionar mais algumas **ReLU**s ao **grafo**.

Modularidade

```
def relu(X):  
    w_shape = (int(X.get_shape()[1]), 1)  
    w = tf.Variable(tf.random_normal(w_shape), name="weights")  
    b = tf.Variable(0.0, name="bias")  
    z = tf.add(tf.matmul(X, w), b, name="z")  
    return tf.maximum(z, 0., name="relu")
```

```
n_features = 3  
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")  
relus = [relu(X) for i in range(5)]  
output = tf.add_n(relus, name="output")
```

```
def relu(X):  
    with tf.name_scope("relu"):  
        [...]
```



- Felizmente, o **TensorFlow** nos permite ficar DRY (Don't Repeat Yourself), ou seja, nós podemos simplesmente implementar uma função para criar uma **ReLU** e não ficar repetindo código.
- O trecho de código ao lado cria cinco **ReLU**s e gera sua soma.
- Observe que a função **add_n()** cria uma operação que computa a soma de uma lista de **tensores**.
- Usando **escopos de nome**, podemos tornar o **grafo** mais claro.
 - Isso é feito simplesmente movendo todo o conteúdo da função **relu()** para dentro de um **escopo de nome**, que no código ao lado foi chamado de **relu**.
- A figura ao lado mostra o **grafo** resultante.
- Nós podemos observar que o **TensorFlow** cria automaticamente nomes distintos aos **escopos de nome**, acrescentando **_1**, **_2** aos nomes do escopo e assim por diante.

Compartilhando variáveis

```
def relu(X, threshold):  
    with tf.name_scope("relu"):  
        [...]  
        return tf.maximum(z, threshold, name="max")  
  
threshold = tf.Variable(0.0, name="threshold")  
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")  
relus = [relu(X, threshold) for i in range(5)]  
output = tf.add_n(relus, name="output")
```

Exemplo: [SharingVariablesExample3.ipynb](#)

- Para compartilhar uma variável entre vários componentes do **grafo**, uma opção simples é criá-la primeiro e depois passá-la como parâmetro para as funções que a utilizam.
- Por exemplo, vamos supor que queiramos controlar o **limiar** (i.e., **threshold**) da **ReLU** (normalmente o limiar é igual 0) usando uma variável de limiar compartilhada com todas as **ReLU**s.
- Para isso, nós poderíamos criar essa variável primeiro e depois passá-la para a função **relu()**, conforme mostrado no trecho de código ao lado.
- Essa abordagem funciona bem para poucas variáveis compartilhadas, porém, imaginem se houvessem muitos parâmetros compartilhados como este, seria tedioso ter que passá-los como parâmetros o tempo todo para as funções.


Compartilhando variáveis

- Para resolver isso, o **TensorFlow** oferece uma opção que pode levar a um código mais limpo e mais modular do que a solução anterior.
- A ideia é usar a função ***get_variable()*** para criar a variável compartilhada se ela ainda não existir, ou reutilizá-la caso ela já exista.
- O comportamento desejado (criação ou reutilização) é controlado por um atributo da função ***variable_scope()***.
- Por exemplo, o trecho de código abaixo ***cria*** uma variável chamada ***relu/threshold***, que é uma variável escalar, pois ***shape=()*** e usando 0.0 como valor inicial.

```
with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(), initializer=tf.constant_initializer(0.0))
```

Compartilhando variáveis

- Observe que se a variável já tiver sido criada por uma chamada anterior à função ***get_variable()***, esse código gerará uma exceção.
- Esse comportamento evita a reutilização de variáveis por engano.
- Se nós realmente desejamos reutilizar uma variável, é necessário fazê-lo explicitamente, definindo o **atributo de reutilização do escopo da variável** como ***True***. Nesse caso, não é necessário se especificar a forma ou o inicializador.

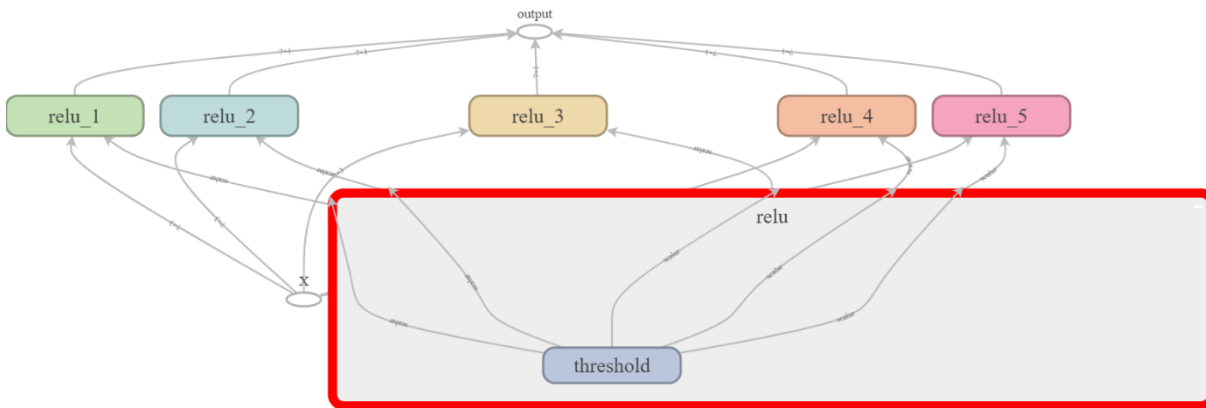


```
with tf.variable_scope("relu", reuse=True):  
    threshold = tf.get_variable("threshold")
```

- Este código buscará a variável ***relu/threshold*** existente ou gerará uma exceção se ela não existir ou se não foi criada usando a função ***get_variable()***.

Compartilhando variáveis

```
def relu(X):  
    with tf.variable_scope("relu", reuse=True):  
        threshold = tf.get_variable("threshold") # reuse existing variable  
        [...]  
        return tf.maximum(z, threshold, name="max")  
  
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")  
with tf.variable_scope("relu"): # create the variable  
    threshold = tf.get_variable("threshold", shape=(), initializer=tf.constant_initializer(0.0))  
    relus = [relu(X) for relu_index in range(5)]  
    output = tf.add_n(relus, name="output")
```

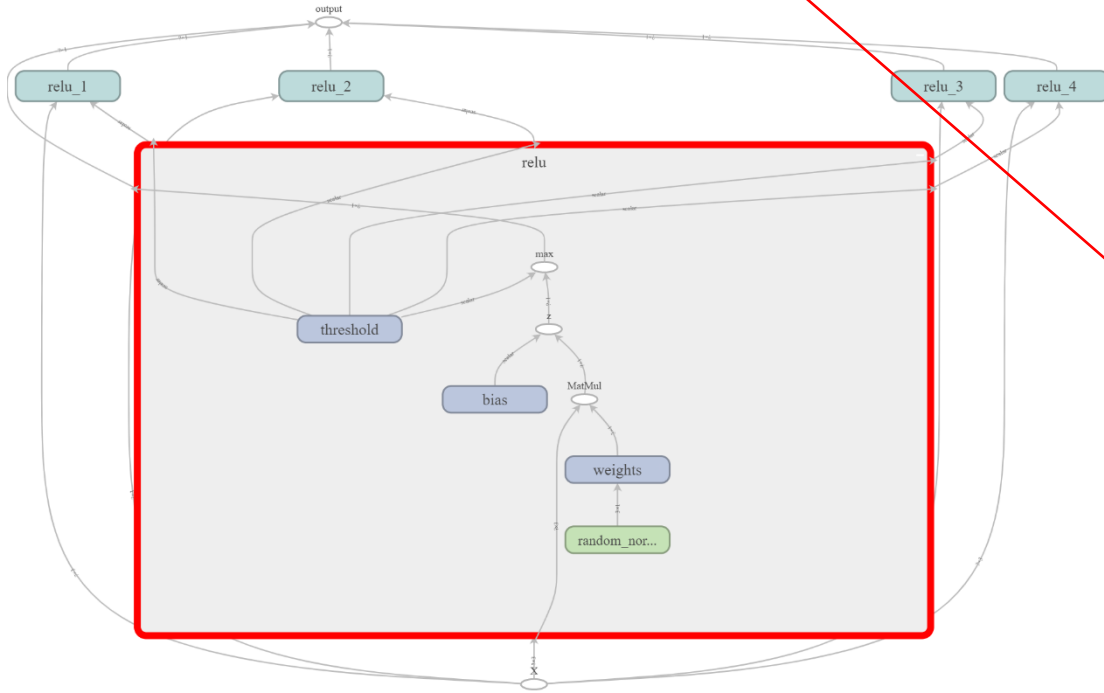


- Agora temos todas as peças necessárias para que a função ***relu()*** acesse a variável ***threshold*** sem ter que passá-la como parâmetro.
- O trecho de código ao lado define primeiro a função ***relu()***, depois cria a variável ***relu/threshold*** (como um escalar que posteriormente será inicializado com o valor 0.0) e cria cinco ***ReLU***s chamando a função ***relu()***.
- A função ***relu()*** reutiliza a variável ***relu/threshold*** e cria os outros ***nós*** pertencentes a função ***ReLU***.

Compartilhando variáveis

```
def relu(X):  
    threshold = tf.get_variable("threshold", shape=(), initializer=tf.constant_initializer(0.0))  
    [...]  
    return tf.maximum(z, threshold, name="max")
```

```
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")  
relus = []  
for relu_index in range(5):  
    with tf.variable_scope("relu", reuse=(relu_index >= 1)) as scope:  
        relus.append(relu(X))  
output = tf.add_n(relus, name="output")
```



- Se analisarmos o código anterior, nós percebemos que é um pouco estranho que a variável **relu/threshold** seja definida fora da função **relu()**, onde todo o restante do código **ReLU** reside.
- Para corrigir isso, o trecho de código ao lado cria a variável **relu/threshold** dentro da função **relu()** na primeira chamada e a reutiliza nas chamadas subsequentes.
- Agora, a função **relu()** não precisa mais se preocupar com **escopos de nome** ou compartilhamento de variáveis: ela apenas chama **get_variable()**, que criará ou reutilizará a variável de **relu/threshold**.
- O restante do código chama a função **relu()** cinco vezes, certificando-se de definir **reuse=False** na primeira chamada e **reuse=True** para as outras chamadas.
- O **grafo** resultante é um pouco diferente do anterior, pois a variável compartilhada está localizada na primeira **ReLU**.

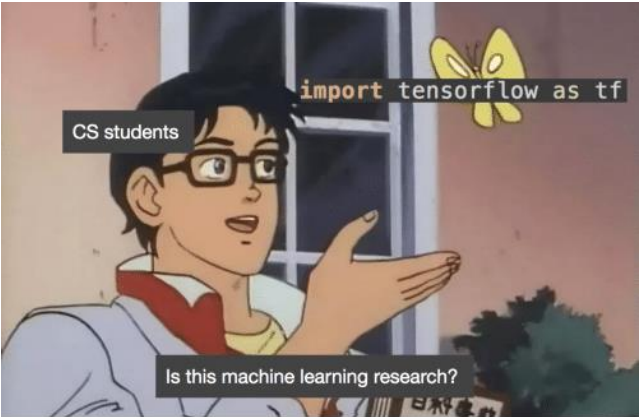
Referências

- "Curso TensorFlow para Iniciantes",
[https://www.youtube.com/watch?v=JHsnHgb9hDo&list=PLyqOvdQmGdT
R_X-BxOJCPIibdjQ_hXycV](https://www.youtube.com/watch?v=JHsnHgb9hDo&list=PLyqOvdQmGdT
R_X-BxOJCPIibdjQ_hXycV)
- "Tutoriais TensorFlow ", <https://www.tensorflow.org/tutorials>

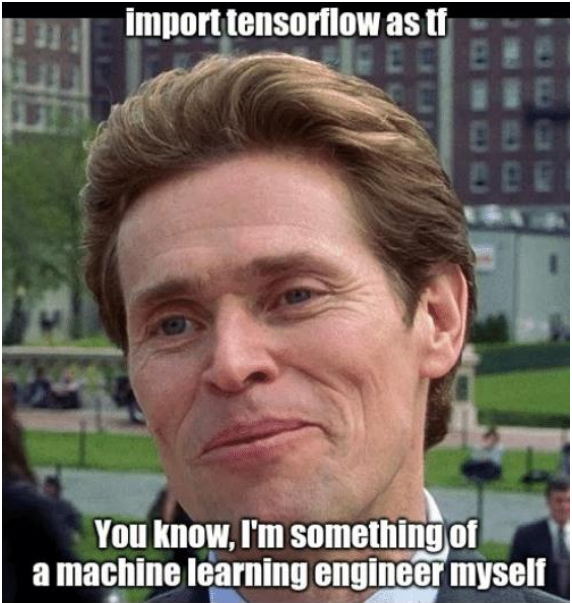
Avisos

- Material, exemplos e lista de exercícios #11 já estão disponíveis.

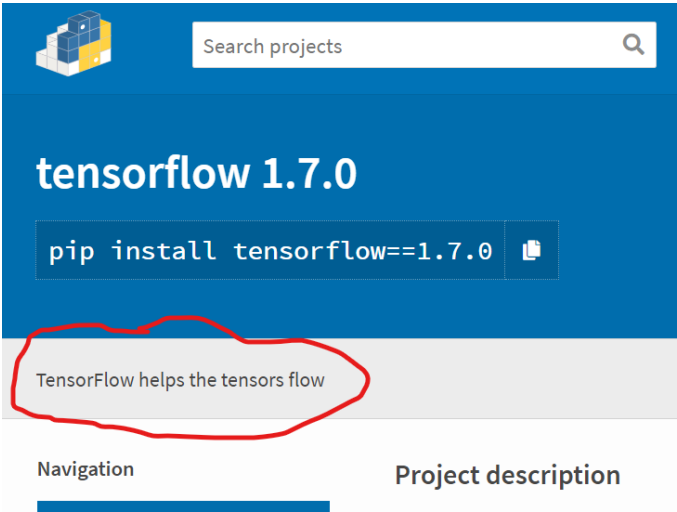
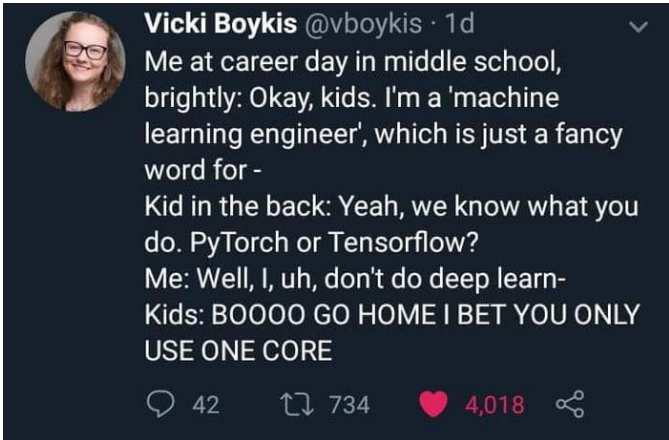
Obrigado!



Programmers Nowadays



You know...



Figuras

