

À seguir, veremos alguns conceitos básicos do **TensorFlow**, da instalação à criação, execução, salvamento e visualização de **grafos computacionais** simples. É importante dominar essas noções básicas antes de criarmos nossa primeira rede neural.

`conda install tensorflow`

ou

`pip3 install --upgrade tensorflow`

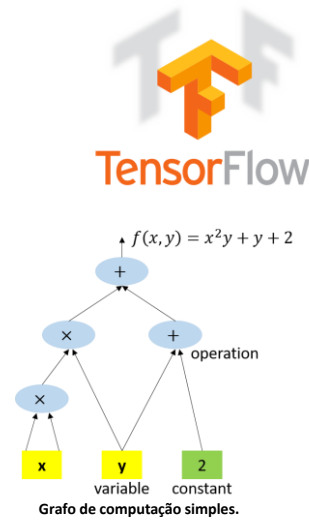
OBS.: Para suporte à GPU, você precisa instalar o `tensorflow-gpu` em vez do `tensorflow`

Para testar sua instalação, digite o seguinte comando. Ele deve gerar a versão do TensorFlow que você instalou.

```
python3 -c 'import tensorflow; print(tensorflow.__version__)'
```

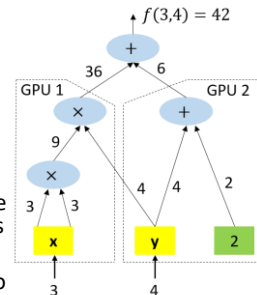
TensorFlow

- 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.
- Seu princípio básico de funcionamento é simples: primeiro, define-se em **Python** um **grafo de computação** a ser executado (como mostrado na figura abaixo) e, em seguida, o **TensorFlow** transforma esse **grafo** em código C++ otimizado e o executa com eficiência.



TensorFlow

- O **TensorFlow** possibilita dividir o **grafo** em vários pedaços e executá-los 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 os cálculos através de centenas de servidores.
- Por exemplo, o **TensorFlow** pode treinar uma **rede neural** com milhões de **parâmetros** (i.e., 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 em vários produtos da empresa, e.g., Google Photos, Google Search, entre outros.



TensorFlow

- O **TensorFlow** foi projetado para ser flexível, escalável e pronto para produção. Alguns destaques do **TensorFlow** são:
 - Roda não apenas no Windows, Linux e macOS, mas também em dispositivos móveis, incluindo iOS e Android.
 - Fornece uma application programming interface (API) em **Python** muito simples chamada **TFLearn** ([tensorflow.contrib.learn](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/learn)) que é compatível com o Scikit-Learn.
 - Também fornece outra API simples chamada **TF-slim** ([tensorflow.contrib.slim](https://github.com/tensorflow/tensorflow/tree/master/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 cálculos, incluindo qualquer arquitetura de **rede neural** que você possa imaginar.

O TensorFlow é fácil de usar, é confiável (estável), de fácil manutenção, escalável e bem documentado.

TF.Learn, como você verá, pode ser usado para treinar vários tipos de redes neurais em apenas algumas linhas de código. Anteriormente, era um projeto independente chamado Scikit Flow (ou skflow).

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 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 excelente ferramenta de visualização chamada **TensorBoard**, que permite navegar pelo **grafo de computação**, visualizar curvas de aprendizado e muito mais.
- Possui uma equipe dedicada de desenvolvedores e uma comunidade crescente que contribui para melhorá-lo.

O TensorFlow não se limita às redes neurais ou mesmo ao aprendizado de máquina; você pode executar simulações de física quântica, se quiser, por exemplo.

Criando e executando um grafo simples

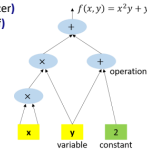
```
import tensorflow as tf
```

```
# 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()
```

- 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 um **grafo de computação**. Na verdade, nem mesmo as variáveis foram inicializadas ainda.
- Para avaliar esse **grafo**, é necessário abrir uma **sessão do TensorFlow** e usá-la para inicializar as **variáveis** e avaliar a função f .
- Uma **sessão do TensorFlow** é responsável por colocar as operações em CPUs e/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).

Dica

- 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 **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 (ou encerrada) 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 o inicializador para cada variável, você pode 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 vocês devem ter percebido, um programa utilizando o **TensorFlow** normalmente é dividido em duas partes:
 - A primeira parte cria um **grafo de computação** (isso é chamado de **fase de construção**)
 - A segunda parte executa o **grafo** (esta é a **fase de execução**).
- A **fase de construção** cria um **grafo de computação** representando o modelo de aprendizado de máquina e os cálculos necessários para treiná-lo.
- Já a **fase de execução**, executa um loop que avalia uma etapa de treinamento repetidamente (por exemplo, uma etapa de treinamento por por mini-batch), melhorando gradualmente os parâmetros 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***.

- Na maioria dos casos, isso é bom, mas às vezes você pode querer gerenciar vários ***grafos*** independentes.
- Você pode fazer isso criando um novo ***grafo*** e tornando-o temporariamente o ***grafo padrão*** dentro de um bloco ***with***, como mostrado no trecho 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 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 e executa-se o **grafo** para se avaliar o valor de y .
- Nesse caso, o **TensorFlow** detecta automaticamente que y depende de x , que depende de w , então ele avalia primeiro o valor de w , depois o de x , então o de y e retorna o valor final de y .
- Por fim, o código executa o **grafo** para avaliar o valor de z . Mais uma vez, o **TensorFlow** detecta que ele deve primeiro avaliar os valores de x e w .
- É importante ressaltar que o **TensorFlow** não reutilizará o resultado da avaliação anterior de x e w . Em resumo, o código anterior avalia x e w duas vezes.

Ciclo de vida do valor de nó

```
with tf.Session() as sess:  
    y_val, z_val = sess.run([y, z])  
    print(y_val) # 10  
    print(z_val) # 15
```

- 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 o inicializador é executado e termina quando a **sessão** é encerrada.
- Se você deseja avaliar y e z eficientemente, sem avaliar w e x duas vezes como no código anterior, você deve orientar o **TensorFlow** para avaliar y e z em apenas uma execução do **grafo**, conforme mostrado no código ao lado.

No single-process TensorFlow, várias sessões não compartilham nenhum estado, mesmo que reutilizem o mesmo grafo (cada sessão teria sua própria cópia de cada variável). No TensorFlow distribuído, o estado variável é armazenado nos servidores, não nas sessões, portanto, várias sessões podem compartilhar as mesmas variáveis

Regressão Linear com TensorFlow

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

Regressão Linear com TensorFlow

```
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()
```

OBS.: O principal benefício desse código em comparação ao cálculo direto da **equação normal** usando o **NumPy** é que o **TensorFlow** o executará automaticamente em sua placa de vídeo GPU, caso você tenha uma e que você tenha instalado o **TensorFlow** com suporte a GPUs.

- Nos exemplos que vimos até agora, os **tensores** continham apenas um valor **escalar**, mas também é possível executar cálculos em matrizes de qualquer formato.
- Por exemplo, o código ao lado manipula matrizes 2D para realizar **regressão linear** com o conjunto de dados de preços de casas no estado da Califórnia.
- O exemplo 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), depois, cria dois **nós constantes** do **TensorFlow**, X e y , para armazenar esses dados e os rótulos, e usa algumas das operações de matriz fornecidas pelo **TensorFlow** para definir **theta**.
- 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 executará quando o **grafo** for executado.
- Nós podemos reconhecer que a definição de **teta** 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**.

Implementando o Gradiente Descendente

- Agora vamos usar o ***gradiente descendente em lote*** em vez da ***equação normal*** para encontrar os parâmetros.
- Inicialmente, faremos isso calculando os gradientes manualmente, em seguida, usaremos o recurso do ***autodiff*** disponibilizado pelo ***TensorFlow***, o qual permite que o ***TensorFlow*** calcule os gradientes automaticamente e, finalmente, usaremos alguns ***otimizadores*** prontos disponibilizados pelo ***TensorFlow***.

Ao usar o gradiente descendente, lembre-se de que é importante normalizar primeiro os vetores de atributos de entrada, ou o treinamento pode se tornar muito lento. Você pode fazer isso usando o TensorFlow, NumPy, StandardScaler da ScikitLearn ou qualquer outra solução que você preferir.

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()
```

- O código ao lado é bastante auto-explicativo, exceto por alguns detalhes:
 - A função ***random_uniform()*** cria um ***nó*** no ***grafo*** que cria um ***tensor*** contendo valores aleatórios, dada sua forma e faixa de valores, bem como a função ***rand()*** da biblioteca NumPy.
 - A função ***assign()*** cria um ***nó*** que atribui um novo valor a uma ***variável***. Nesse caso, ela implementa o passo do ***gradiente descendente em lote*** $\theta^{(\text{next step})} = \theta - \alpha \nabla_{\theta} \text{MSE}(\theta)$.
 - O loop principal executa o passo de treinamento acima repetidamente (***n_epochs*** vezes) e a cada 100 iterações imprime o erro quadrático médio (MSE) atual.
 - O MSE deve diminuir a cada iteração.

Usando *autodiff* para cálculo dos 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 razoavelmente fácil, mas se você tivesse que fazer isso para **redes neurais** com várias camadas você teria muita dor de cabeça: seria tedioso e 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**, que calcula de forma automática e eficiente os gradientes.
- Para utilizá-lo, 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 **mse**) e uma lista de variáveis (nesse caso, apenas **theta**), e 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**.
- 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 otimizadores prontos

- Como vimos, o **TensorFlow** calcula os gradientes automaticamente. Além disso, ele também fornece vários **otimizadores** prontos para uso, incluindo um **otimizador de gradiente descendente**.
- Para usar o **otimizador de gradiente descendente** do **TensorFlow**, 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 de momento** (que geralmente converge muito mais rápido que **otimizador de gradiente descendente**) definindo o **otimizador** da seguinte maneira:

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

Outros otimizadores podem ser encontrados na documentação do TensorFlow:
https://www.tensorflow.org/api_docs/python/tf/compat/v1/train#classes

Suprindo dados aos grafos em tempo de execução

- Vamos modificar o código anterior para implementar o ***gradiente descendente em mini-batches***.
- Para isso, precisamos de uma maneira de substituir X e y a cada iteração pelo próximo mini-batch.
- A maneira mais simples de 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.
- Eles são usados para passar os dados de treinamento para o ***TensorFlow*** durante o treinamento.

OBS.: Caso você não especifique um valor em tempo de execução para um nó placeholder, você receberá uma exceção.

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

OBS.: Você pode alimentar a saída de qualquer operação, não apenas de placeholders. Nesse caso, o TensorFlow não tenta avaliar essas operações; Ele usa os valores que você passa.

Suprindo 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 de X e y na fase de construção do **grafo** para torná-los **nós de placeholder**.
- Em seguida, definimos o tamanho de um batch e calculamos seu número total.
- Por fim, na fase de execução, lemos os mini-batches um por um e fornecemos os valores de X e y através do parâmetro **feed_dict** ao avaliar um **nó** que depende deles.

OBS.: Não precisamos passar os valores de X e y ao avaliar θ , pois ele não depende de nenhum 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, pode-se salvar seus parâmetros em disco para poder utilizá-los sempre que se quiser.
- Você pode usá-los em outro programa, compará-lo com outros modelos e assim por diante.
- Além disso, você vai provavelmente querer salvar os parâmetros em intervalos regulares durante o treinamento do modelo, para que, se o computador travar durante o treinamento, você possa 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 *nós* do tipo *Variável* tiverem sido criados.
- Em seguida, na fase de execução, chame o método *save()* sempre que desejar salvar o modelo, passando a *sessão* e o caminho do arquivo onde se deseja salvar o ponto de verificação.

Salvando e restaurando modelos

- Restaurar um modelo também é fácil: você cria um **nó** do tipo **Saver** no final da **fase de construção** como anteriormente, mas no início da **fase de execução**, em vez de inicializar as variáveis usando o **nó de inicialização**, você chama o método **restore()** do objeto **Saver**, 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 **Saver** salva e restaura todas as variáveis com seu próprio nome, mas se você precisar de mais controle, poderá especificar quais variáveis salvar ou restaurar e quais nomes usar.
- Por exemplo, o objeto **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 algumas estatísticas de treinamento, o **TensorBoard** exibe visualizações interativas dessas estatísticas no seu 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 é ajustar o programa para gravar a definição do **grafo** e algumas estatísticas de treinamento, e.g., o erro de treinamento, em um diretório de log ao qual o **TensorBoard** terá acesso.
- 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.
- 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 = "{}-{}-{}".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**.
- A primeira linha cria um **nó** no **grafo** que avaliará o valor do **erro quadrático médio** (MSE) e o gravará em um arquivo de log compatível com **TensorBoard** chamado de **summary**.
- A segunda linha cria um objeto **FileWriter** que é usado para escrever os resultados no arquivo de log.
- O primeiro parâmetro indica o caminho do diretório de log. O segundo parâmetro, que é opcional, é o **grafo** que você deseja visualizar.
- Após a criação, o objeto **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**.

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

Para visualizar o gráfico no Jupyter, usaremos um servidor TensorBoard disponível on-line em <https://tensorboard.appspot.com/>. Essa abordagem não funcionará caso você não tenha acesso à Internet. Como alternativa ao servidor TensorBoard, você pode usar a biblioteca tfgraphviz.

A biblioteca tfgraphviz pode ser instalada executando-se os seguintes comandos

```
pip install graphviz
pip install tfgraphviz
```


Visualizando grafos e curvas de treinamento com o TensorBoard

- Em seguida, é necessário atualizar o código da **fase de execução** para avaliar o ***mse_summary*** regularmente durante o treinamento (por exemplo, a cada 10 mini-batches).
- Isso produzirá um **resumo** (i.e., o log) que pode ser gravado no arquivo de eventos usando o ***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})
[...]
```

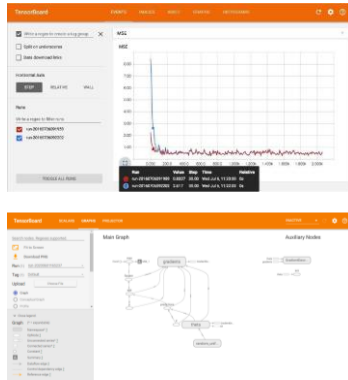
OBS.: Evite logar estatísticas de treinamento em cada etapa do treinamento, pois isso aumentará significativamente o tempo de treinamento.

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, ele 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, inicia-se o servidor executando o comando **`tensorboard`**, apontando-o para o diretório de logs.
- Esse comando inicia um servidor web do **TensorBoard**, que fica *escutando* 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)
```

Visualizando grafos e curvas de treinamento com o TensorBoard



- Em seguida, abra um navegador e acesse **<http://0.0.0.0:6006/>** (ou **<http://localhost:6006/>**).
- Na guia **Events**, você deverá ver o MSE à direita. Se você clicar nele, verá o gráfico do MSE durante o treinamento.
- Você pode marcar ou desmarcar as execuções que deseja 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**.

Para reduzir a desorganização, os nós que possuem muitas arestas (por exemplo, conexões com outros nós) são separados para uma área auxiliar à direita (você pode mover um nó para frente e para trás entre o grafo principal e a área auxiliar clicando com o botão direito do mouse em isto). Algumas partes do grafo também são recolhidas por padrão.

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")
```

- Suponha que você queira criar um **grafo** que adicione a saída de duas **unidades lineares retificadas (ReLU)**.

- Uma **ReLU** calcula uma função linear das entradas e gera como saída o resultado da função linear caso este seja positivo, e 0 caso contrário. A equação da **ReLU** é mostrada abaixo.

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

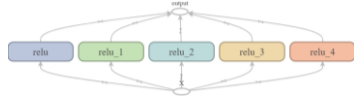
- O trecho de código ao lado realiza a tarefa, mas é bastante repetitivo.
- Além disso, é difícil manter esse código repetitivo e ele é propenso a erros.
- Ficaria ainda pior se quiséssemos adicionar mais algumas **ReLU**s.

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** permite que você fique DRY (Don't Repeat Yourself): simplesmente crie uma função para criar uma **ReLU**.
- 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()** dentro de um **escopo de nome**, chamado de **relu**.
- A figura ao lado mostra o **grafo** resultante. Observe que o **TensorFlow** também fornece nomes distintos aos **escopos de nome**, acrescentando **_1**, **_2** aos nomes do escopo e assim por diante.

Observe que, quando você cria um nó, o TensorFlow verifica se o nome já existe e, se existir, acrescenta um sublinhado seguido por um índice para tornar o nome exclusivo. Portanto, a primeira ReLU contém nós denominados "weights", "bias", "z" e "relu" (além de muitos outros nós com o nome padrão, como "MatMul"); a segunda ReLU contém nós denominados "weights_1", "bias_1" e assim por diante; o terceiro ReLU contém nós denominados "weights_2", "bias_2" e assim por diante. O TensorBoard identifica essas séries e as junta para reduzir a desordem.

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")
```

- 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, suponha que você queira 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. Você pode 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, imagine se houverem muitos parâmetros compartilhados como este, será tedioso ter que passá-los como parâmetros o tempo todo.

Muitas pessoas criam um dicionário Python contendo todas as variáveis em seu modelo e o passam para todas as funções. Outras criam uma classe para cada módulo (por exemplo, uma classe ReLU usando variáveis de classe para manipular o parâmetro compartilhado). Ainda outra opção é definir a variável compartilhada como um atributo da função *relu()* na primeira chamada, assim:

```
def relu(X):
    with tf.name_scope("relu"):
        if not hasattr(relu, "threshold"):
            relu.threshold = tf.Variable(0.0, name="threshold")
        [...]
        return tf.maximum(z, relu.threshold, name="max")
```

Entretanto, o TensorFlow oferece outra opção, que pode levar a um código ligeiramente mais limpo e mais modular do que as soluções anteriores.

Compartilhando variáveis


- O **TensorFlow** oferece uma opção, que pode levar a um código mais limpo e mais modular do que a solução anterior.
- A idéia é usar a função **get_variable()** para criar a variável compartilhada se ela ainda não existir, ou reutilizá-la se ela já existir.
- 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 você realmente deseja reutilizar uma variável, é necessário dizê-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")
```



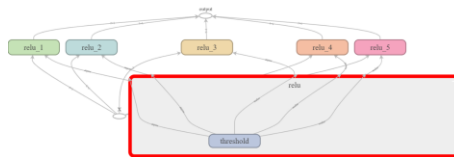
- Este código buscará a variável "relu/threshold" existente ou gerará uma exceção se ela não existir ou se não foi criada usando a função ***get_variable()***.

OBS.: Depois que a reutilização estiver configurada como True, ela não poderá ser configurada novamente como False dentro do bloco. Além disso, se você definir outros escopos de variáveis dentro deste, eles herdarão automaticamente reuse=True. Por fim, apenas variáveis criadas pela função `get_variable()` podem ser reutilizadas dessa maneira.

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")
```



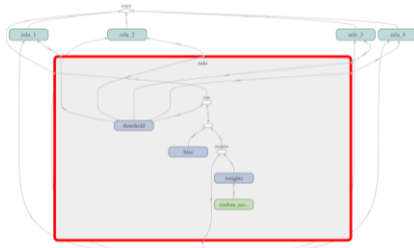
- Agora temos todas as peças necessárias para que a função **relu()** acesse a variável **threshold** sem ter que passá-la como parâmetro.
- O trecho de código ao lado define primeiro a função **relu()**, depois cria a variável **relu/threshold** (como um escalar que posteriormente será inicializado 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 ReLU**.

OBS.: As variáveis criadas usando `get_variable()` são sempre nomeadas usando o nome de seu `variable_scope` como um prefixo (por exemplo, "relu/threshold"), mas para todos os outros nós (incluindo variáveis criadas com `tf.Variable()`), o escopo da variável age como um novo escopo de nome. Em particular, se um escopo de nome com um nome idêntico já foi criado, um sufixo é adicionado para tornar o nome exclusivo. Por exemplo, todos os nós criados no código anterior (exceto a variável de `threshold`) têm um nome prefixado de "relu_1/" a "relu_5/".

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")
```

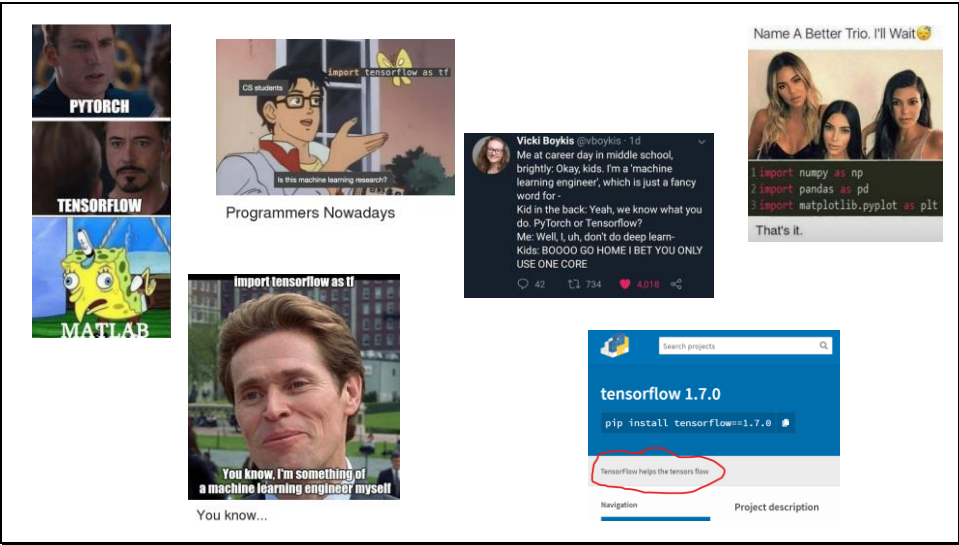


- É um pouco estranho que a variável de threshold (ou limiar) seja definida fora da função **relu()**, onde todo o restante do código **ReLU** reside.
- Para corrigir isso, o trecho de código ao lado cria a variável de **limiar** dentro da função **relu()** na primeira chamada e a reutiliza nas chamadas subsequentes.
- Agora, a função **relu()** não precisa se preocupar com **escopos de nome** ou compartilhamento de variáveis: ela apenas chama **get_variable()**, que criará ou reutilizará a variável de **limiar**.
- 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**.

Avisos

- Material já está disponível no site.
- Todas as listas podem ser entregues, impreterivelmente, até dia 23/06.
- Alguns grupos ainda não definiram o horário de suas apresentações.

Obrigado!



Figuras

