

TP555 - Inteligência Artificial e Machine Learning: *Redes Neurais Artificiais com TensorFlow v1.x*



Inatel

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

Motivação

- Neste tópico veremos como criar nossos próprios modelos de redes neurais utilizando a biblioteca **TensorFlow v1.x**.
- Existem basicamente 2 APIs nativas nesta versão para se criar e treinar modelos com o **TensorFlow**:
 - **TensorFlow direto/original**: API de baixo nível extremamente prolixa (não é consisa) e sujeita a bugs sutis e difíceis de serem detectados.
 - **TFLearn**: API de alto nível que facilita a criação, treinamento e validação de modelos, além de ser compatível com a biblioteca **SciKit-Learn**.
- Entretanto, a API original do **TensorFlow** oferece muito mais flexibilidade (ao custo de uma maior complexidade) para criar todos os tipos de modelos que nós possamos imaginar.

Treinando redes MLP com a API de alto nível

- A maneira mais simples de se criar e treinar uma rede MLP com o TensorFlow é usando a **API de alto nível *TFLearn***, que é bastante semelhante às APIs da biblioteca **SciKit-Learn**.
- A classe ***DNNClassifier*** (*Deep Neural Network* - DNN) torna muito fácil a criação e o treinamento de uma rede neural (profunda ou não) com qualquer número de camadas ocultas e uma camada de saída ***softmax*** usada para calcular as probabilidades das classes estimadas.
- Por exemplo, o código abaixo treina uma rede DNN para classificação com duas camadas ocultas (uma com 300 neurônios e a outra com 100 neurônios) e uma camada de saída ***softmax*** com 10 neurônios:

```
import tensorflow as tf

feature_columns = tf.contrib.learn.infer_real_valued_columns_from_input(X_train)

dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300, 100], n_classes=10, feature_columns=feature_columns)
dnn_clf.fit(x=X_train, y=y_train, batch_size=50, steps=40000)
```

Instancia um objeto da classe `DNNClassifier`.

Número de saídas (ou classes)

O parâmetro `feature_columns` define o tamanho da camada de entrada.

2 camadas escondidas com 300 e 100 nós, respectivamente

Treinando redes MLP com a API de alto nível

- Se usarmos esse código com o conjunto de dados MNIST (base de dados com imagens de dígitos escritos à mão), nós obtemos um modelo que atinge 98.1% de precisão no conjunto de testes.
- A biblioteca **TFLearn** também fornece algumas funções úteis para avaliar os modelos.
- Debaixo dos panos, a classe **DNNClassifier** cria todas as camadas da rede com base na função de ativação **ReLU** (podemos mudar isso configurando o hiperparâmetro **activation_fn**).
- Por padrão, o otimizador usado é o **AdaGrad**, mas isto pode ser alterado com o hiperparâmetro **optimizer**.
- A camada de saída utiliza a função de ativação **softmax**, e a função de custo utilizada é a da **entropia cruzada**.

```
from sklearn.metrics import accuracy_score
```

```
y_pred = list(dnn_clf.predict(X_test))  
accuracy_score(y_test, y_pred)
```

```
0.9818000000000001
```

```
dnn_clf.evaluate(X_test, y_test)  
{'accuracy': 0.98180002, 'global_step': 40000,  
 'loss': 0.073678359}
```

Treinando redes MLP com a API de baixo nível

- Caso desejarmos ter mais controle sobre a arquitetura do modelo, então nós devemos usar a **API de baixo nível** do TensorFlow.
- A seguir, cria-se o mesmo modelo usado anteriormente usando a API de baixo nível.
- Para isso, nós iremos implementar o algoritmo do **gradiente descendente em mini-batch** para treinar uma rede neural que classifique imagens de dígitos escritos à mão (MNIST).
- O primeiro passo é a **fase de construção**, ou seja a construção do **grafo de computação**.
- O segundo passo é a **fase de execução**, na qual se executa o **grafo** para treinar o modelo da rede neural.

Exemplo: [MLPWithTensorFlowLowLevelAPI.ipynb](#)

Fase de Construção

- Inicialmente, importamos a biblioteca tensorflow. Em seguida, especificamos o número de entradas e saídas e definimos o número de **nós** ocultos em cada camada:

```
import tensorflow as tf
```

```
n_inputs = 28*28 # MNIST
```

```
n_hidden1 = 300
```

```
n_hidden2 = 100
```

```
n_outputs = 10
```

Imagens com 28 por 28 pixels,
onde cada pixel será um
atributo de entrada.

- Em seguida, usamos **nós** do tipo **placeholder** para representar os dados de treinamento (i.e., atributos e saídas desejadas, os rótulos):

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

```
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

None: número de
exemplos é variável.

Fase de Construção

- Agora criamos a rede neural propriamente dita.
- O **nó** do tipo ***placeholder X*** atuará como a ***camada de entrada***.
- Durante a ***fase de execução***, ele será substituído por um mini-batch a cada iteração.
- Observem que todos os exemplos em um mini-batch serão processados simultaneamente pela rede neural.
- Em seguida, nós criamos as duas camadas ocultas e a camada de saída.
- As duas camadas ocultas são quase idênticas: elas diferem apenas pelas entradas às quais estão conectadas e pelo número de ***nós*** que contêm.


```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "outputs")
```

Fase de Construção

- A camada de saída também é muito semelhante às outras, mas ela usa uma função de ativação do tipo ***softmax*** em vez de uma função de ativação do tipo ***ReLU***.
- Para facilitar, nós implementamos uma função chamada ***neuron_layer()*** que cria cada uma das camadas.
- São necessários alguns parâmetros para especificar as entradas da função: o placeholder ***X***, o número de ***nós*** da camada, o nome da camada e a função de ativação.
- O trecho de código da função ***neuron_layer()*** é mostrado ao lado.

None: não usa nenhuma função de ativação.



```
def neuron_layer(X, n_neurons, name, activation=None):  
    with tf.name_scope(name):  
        n_inputs = int(X.get_shape()[1])  
        stddev = 2 / np.sqrt(n_inputs)  
        init = tf.truncated_normal([n_inputs, n_neurons], stddev=stddev)  
        W = tf.Variable(init, name="weights")  
        b = tf.Variable(tf.zeros([n_neurons]), name="biases")  
        z = tf.matmul(X, W) + b  
        if activation=="relu":  
            return tf.nn.relu(z)  
        else:  
            return z
```


Fase de Construção

- Vamos analisar o código da função linha por linha:

- Primeiro, nós criamos um **escopo de nomes** usando o nome da camada. Esse escopo conterá todos os **nós** dessa camada.
- Em seguida, obtemos o número de entradas através da segunda dimensão da matriz de entrada **X**, sendo a primeira dimensão o número de exemplos, que é variável.
- As próximas três linhas criam uma variável **W** que conterá a matriz de pesos. Ela será um tensor 2D (i.e., uma matriz) contendo todos os pesos de conexão entre cada entrada e cada **nó** da camada.
- A matriz **W** é inicializada aleatoriamente, usando-se uma **distribuição Gaussiana normal truncada** com desvio padrão igual a $2/\sqrt{n_inputs}$ (inicialização de He).

```
def neuron_layer(X, n_neurons, name, activation=None):  
    with tf.name_scope(name):  
        n_inputs = int(X.get_shape()[1])  
        stddev = 2 / np.sqrt(n_inputs)  
        init = tf.truncated_normal([n_inputs, n_neurons], stddev=stddev)  
        W = tf.Variable(init, name="weights")  
        b = tf.Variable(tf.zeros([n_neurons]), name="biases")  
        z = tf.matmul(X, W) + b  
        if activation=="relu":  
            return tf.nn.relu(z)  
        else:  
            return z
```

- Ela descarta e retira novamente quaisquer amostras que estejam a mais de dois desvios padrão da média.
- Usar uma distribuição normal truncada em vez de uma distribuição normal regular garante que não haverá pesos com magnitudes elevadas, o que pode retardar o treinamento.

Fase de Construção

- A próxima linha cria uma variável ***b*** para armazenar os valores de ***bias*** de cada um dos nós da camada. Note que ***b*** é um vetor.
- Cada elemento do vetor ***b*** é inicializado com o valor 0.
- Em seguida, criamos uma operação para calcular a ***combinação linear***: $z = XW + b$.
- Essa implementação vetorizada calcula eficientemente a combinação linear das entradas mais o termo de ***bias*** para cada ***nó*** da camada, para todas os exemplos do mini-batch em apenas uma iteração.
- Finalmente, se o parâmetro que especifica a ***função de ativação*** estiver definido como ***relu***, o código retorna o valor de ***relu(z)*** (ou seja, $\max(0, z)$), caso contrário, apenas o valor de ***z*** é retornado (***função de ativação linear***).

```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal([n_inputs, n_neurons], stddev=stddev)
        W = tf.Variable(init, name="weights")
        b = tf.Variable(tf.zeros([n_neurons]), name="biases")
        z = tf.matmul(X, W) + b
        if activation=="relu":
            return tf.nn.relu(z)
        else:
            return z
```

Fase de Construção

- Agora vamos usar a função ***neuron_layer()*** para criar a rede neural.
- A primeira camada oculta recebe o ***placeholder X*** como entrada.
- A segunda camada oculta recebe a saída da primeira camada oculta como entrada.
- E, finalmente, a camada de saída recebe a saída da segunda camada oculta como entrada.

```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "outputs")
```

- Observe que ***logits*** é a saída da rede neural antes de passar pela ***função de ativação*** do tipo ***softmax***.
- Por razões de otimização, trataremos do cálculo do ***softmax*** posteriormente.

Funções úteis do TensorFlow

- O TensorFlow possui muitas funções úteis para criar camadas padrão, então geralmente não há necessidade de definirmos nossa própria função ***neuron_layer()*** como acabamos de fazer.
- Por exemplo, a função ***fully_connected()*** cria uma camada totalmente conectada, onde todas as entradas são conectadas a todos os ***nós*** da camada.
- Ela se encarrega de criar as variáveis de ***peso*** e ***bias***, com a estratégia de inicialização adequada, e usa a função de ativação ***ReLU*** por padrão (podemos mudar isso usando o hiperparâmetro ***activation_fn***).
- Ela também oferece suporte a hiperparâmetros de regularização e normalização.

Fase de Construção

- Agora que já temos o modelo da rede neural pronto, precisamos definir a **função de custo** que usaremos para treiná-lo.
- Nós iremos usar a medida de **entropia cruzada** como **função de custo**. Essa é a mesma função de custo que usamos com os regressores logístico e softmax.
- Usaremos a função **`sparse_softmax_cross_entropy_with_logits()`**, que é equivalente a aplicar a **função de ativação** do tipo **softmax** e depois computar a **entropia cruzada**, mas é mais eficiente e cuida de casos como quando os valores dos **logits** são iguais a 0.
- Essa função calcula a **entropia cruzada** com base nos **logits** (ou seja, a saída da rede neural antes de passar pela **função de ativação** do tipo **softmax**) e espera **rótulos** na forma de números inteiros que variam de 0 ao número total de classes menos 1.
- Isso gerará um tensor 1D contendo a **entropia cruzada** para cada exemplo. Em seguida, usamos a função **`reduce_mean()`** do TensorFlow para calcular a **entropia cruzada média** com todos os exemplos.

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
```

Fase de Construção

- Temos agora o modelo da rede neural e a **função de custo**, em seguida, como fizemos anteriormente, precisamos definir um **otimizador** que irá ajustar os **pesos/bias** do modelo para minimizar a **função de custo**.
- O último passo na **fase de construção** é especificar como avaliar o modelo. Nós usaremos a **precisão** como medida de desempenho.
- Primeiro, para cada exemplo de entrada, determinamos se a predição feita pelo modelo está correta, verificando se o **logit** de valor mais alto corresponde ou não à classe correta.
- Para isso, usamos a função **in_top_k()**, que retorna um tensor 1D com valores booleanos indicando se o **logits** de maior valor corresponde à classe em **y**. Em seguida, convertemos esses valores booleanos em floats com a função **tf.cast()** e calculamos a média. Isso dará a precisão da rede neural.
- Finalmente, como de praxe, criamos um **nó** para inicializar todas as variáveis e um **nó** do tipo **Saver** para salvar os parâmetros de modelo treinado em disco.

```
with tf.name_scope("train"):  
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)  
    training_op = optimizer.minimize(loss)
```

```
with tf.name_scope("eval"):  
    correct = tf.nn.in_top_k(logits, y, 1)  
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

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

Fase de execução

- Agora executamos o grafo de computação criado.
- Inicialmente, carregamos a base de dados MNIST usando uma função disponibilizada pelo TensorFlow.
- A base de dados MNIST é um grande banco de dados de dígitos escritos à mão comumente usada para treinar modelos de classificação de imagens.
- Em seguida, as bases de treinamento e validação são re-dimensionadas e escalonadas (valores variando entre 0 e 1).

```
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()  
X_train = X_train.astype(np.float32).reshape(-1, 28*28) / 255.0  
X_test = X_test.astype(np.float32).reshape(-1, 28*28) / 255.0  
y_train = y_train.astype(np.int32)  
y_test = y_test.astype(np.int32)
```

- Na sequência, definimos o número de épocas que queremos executar, bem como o tamanho dos mini-batches:

```
n_epochs = 400  
batch_size = 50
```

Fase de execução

- Em seguida, podemos treinar o modelo.
- O código ao lado cria uma ***sessão*** e executa o ***nó init*** que por sua vez inicializa todas as variáveis.
- Em seguida, o laço de treinamento principal é executado: a cada época, o código itera através de vários mini-batches criados a partir do conjunto de treinamento.
- Cada mini-batch é criado pela função ***shuffle_batch()*** e, em seguida, o código simplesmente executa a ***operação de treinamento***, utilizando os exemplos e rótulos do mini-batch corrente.
- Em seguida, ao final de cada época, o código avalia a acurácia do modelo com o último mini-batch utilizado e com o conjunto de validação e imprime os resultados.
- Finalmente, os parâmetros do modelo são salvos em disco.

```
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for X_batch, y_batch in shuffle_batch(X_train, y_train, batch_size):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
            acc_test = accuracy.eval(feed_dict={X: X_valid, y: y_valid})
            print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)
        save_path = saver.save(sess, "./my_model_final.ckpt")
```


Usando o modelo treinado

[Exemplo: MLPWithTensorFlowLowLevelAPI.ipynb](#)

- Agora que o modelo está treinado, nós podemos usá-lo para fazer previsões.
- Para fazermos isso, reutilizamos o código da **fase de construção**, mas alteramos a **fase de execução** da seguinte maneira:

```
with tf.Session() as sess:  
    saver.restore(sess, "./my_model_final.ckpt")  
    X_new_scaled = [...] # novos exemplos  
    Z = logits.eval(feed_dict={X: X_new_scaled})  
    y_pred = np.argmax(Z, axis=1)
```

- Inicialmente, carregamos os parâmetros do modelo armazenado em disco.
- Em seguida, usamos algumas novas imagens escalonadas para serem classificadas.
- Finalmente, o código avalia o **nó logits** e prevê a classe de maior probabilidade.
- Para isso basta escolher a classe que tem o maior valor de **logit** usando a função **argmax()**.
- Se quiséssemos conhecer as probabilidades estimadas para cada classe, seria necessário aplicar a função de ativação **tf.nn.softmax()** aos logits.

Ajuste fino dos hiperparâmetros de um modelo

- A **flexibilidade de configuração** das redes neurais também é uma de suas **principais desvantagens**: existem **muitos hiperparâmetros** para se ajustar.
- Nós podemos não apenas usar qualquer tipo de **topologia** de rede imaginável (i.e., como os neurônios são interconectados), mas também podemos alterar o **número de camadas**, o **número de neurônios** por camada, o tipo de **função de ativação** a ser usada em cada camada, a lógica de **inicialização dos pesos/bias** e muito mais.
- Sendo assim, como fazemos para saber qual combinação de hiperparâmetros é a melhor para uma dada tarefa?

Ajuste fino dos hiperparâmetros de um modelo

- Nós poderíamos usar **GridSearch** com **validação cruzada** para encontrar os hiperparâmetros ótimos, mas como existem muitos hiperparâmetros para se ajustar, e como treinar uma rede neural com um grande conjunto de dados leva muito tempo, nós poderíamos explorar apenas uma pequena parte do espaço de hiperparâmetros em um período de tempo razoável.
- Outra abordagem seria utilizar **busca aleatória (RandomSearch)**.
 - Diferentemente do **GridSearch**, nem todos os valores de parâmetros são testados, mas um número fixo de parâmetro é amostrado a partir das distribuições especificadas.
- Ainda uma outra opção seria usar ferramentas como o **HParams**, **Keras Tuner** e **Oscar** que implementam algoritmos mais complexos para encontrar rapidamente um bom conjunto de hiperparâmetros.

Ajuste fino dos hiperparâmetros de um modelo

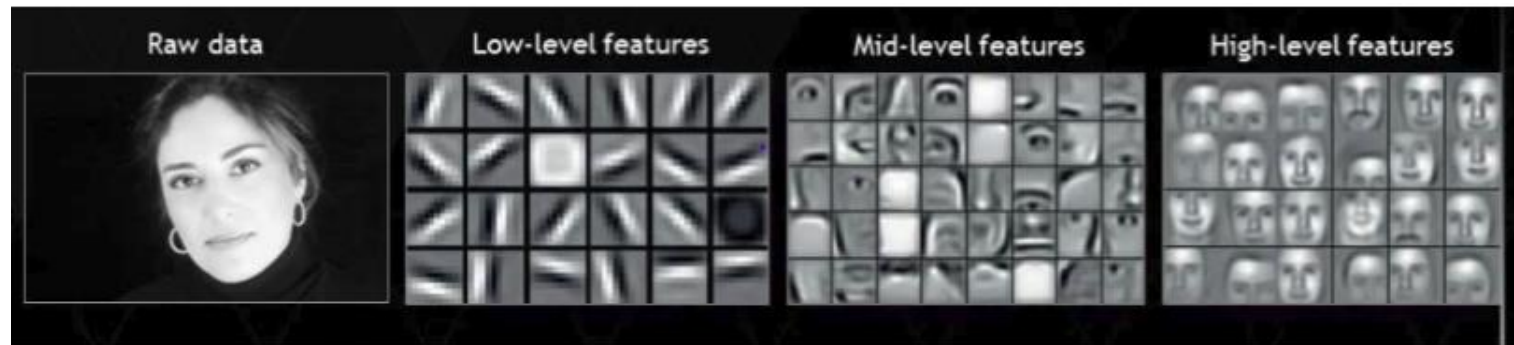
Número de camadas ocultas

- Para muitos problemas, nós podemos começar com apenas uma única camada oculta e mesmo assim obtermos resultados razoáveis.
- Na verdade, foi demonstrado que uma rede MLP com apenas uma única camada oculta pode modelar até as funções mais complexas, desde que a rede possua neurônios suficientes.
- Por um longo tempo, esses fatos convenceram os pesquisadores de que não havia necessidade de se investigar redes neurais mais profundas.
- Entretanto eles ignoravam o fato de que as redes profundas podem modelar funções complexas usando muito menos neurônios do que as redes *rasas*, tornando-as muito mais rápidas para se treinar.

Ajuste fino dos hiperparâmetros de um modelo

Número de camadas ocultas

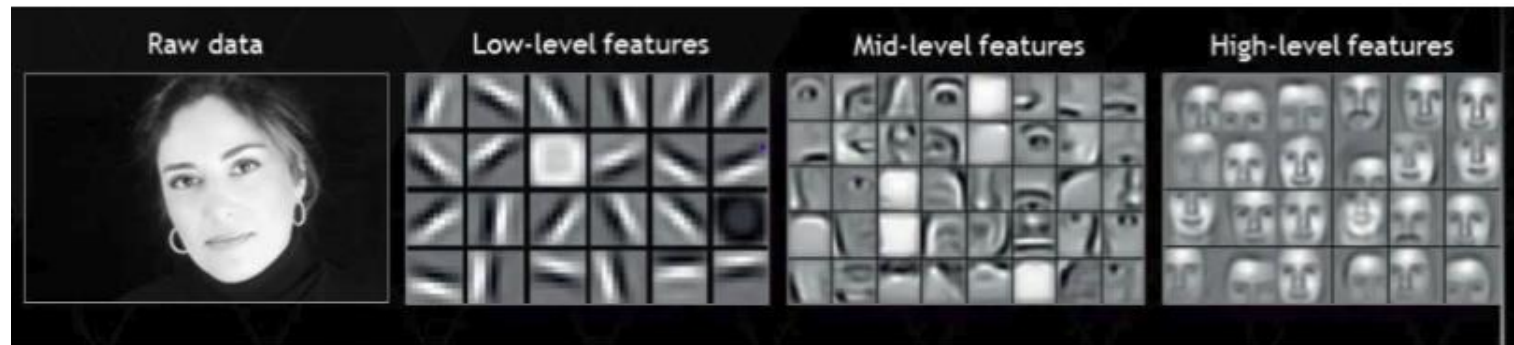
- Além disso, redes profundas tiram proveito da ***estrutura hierárquica*** presente em dados do mundo real.
- Por exemplo, em reconhecimento de faces, camadas ocultas próximas à entrada modelam características de baixo nível: segmentos de linha de várias formas e orientações.
- Camadas ocultas intermediárias combinam essas características de baixo nível para modelar características de nível intermediário (por exemplo, quadrados, círculos, etc.)



Ajuste fino dos hiperparâmetros de um modelo

Número de camadas ocultas

- Por fim, as camadas ocultas mais próximas à saída juntamente com a camada de saída combinam essas características intermediárias para modelar características de alto nível (por exemplo, faces).
- Essa arquitetura hierárquica não apenas ajuda as redes profundas a convergirem mais rapidamente para uma boa solução, como também melhora a capacidade de generalização para ***novos conjuntos de dados***.



Ajuste fino dos hiperparâmetros de um modelo

Número de camadas ocultas

- Por exemplo, se já treinamos um modelo para reconhecer faces e agora desejamos treinar uma nova rede neural para reconhecer estilos de cabelo, podemos iniciar o treinamento reutilizando as camadas inferiores (ou seja, mais próximas à entrada) da primeira rede.
- Em vez de inicializar aleatoriamente os pesos e biases das primeiras camadas da nova rede neural, podemos inicializá-los com os pesos e biases das camadas inferiores da primeira rede.
- Desta forma, uma rede neural não precisa aprender do zero todas as características de baixo nível que ocorrem nos dados (e.g., fotos), ela pode reutilizar os pesos/biases de camadas mais baixas (as quais já aprenderam as estruturas de baixo nível) e precisará apenas aprender (treinar o restante das camadas) apenas as características de alto nível (e.g., estilos de cabelo).

Ajuste fino dos hiperparâmetros de um modelo

Número de camadas ocultas

- Em resumo, para muitos problemas, nós podemos começar com apenas uma ou duas camadas ocultas e com isso obteremos bons resultados.
- Para problemas mais complexos, podemos aumentar gradualmente o número de camadas ocultas, até que a rede comece a sobreajustar demais ao conjunto de treinamento.
- Entretanto, você raramente precisará treinar essas redes do zero: é muito mais comum reutilizar partes de uma rede pré-treinada que executa uma tarefa semelhante.
- Desta forma, o treinamento será muito mais rápido e exigirá muito menos dados.

Ajuste fino dos hiperparâmetros de um modelo

Número de neurônios por camada

- Obviamente, o número de neurônios nas camadas de entrada e saída é determinado pelo tipo de entrada e saída que uma determinada tarefa exige.
- Quanto às camadas ocultas, uma prática comum é dimensioná-las para formar um *funil*, com cada vez menos neurônios em cada camada.
- A lógica por trás dessa abordagem é que muitas características de baixo nível se unem à um número muito menor de características de alto nível.

Ajuste fino dos hiperparâmetros de um modelo

Número de neurônios por camada

- Assim como no número de camadas, nós podemos tentar aumentar gradualmente o número de neurônios até que a rede comece a **sobreajustar**.
- Em geral, obtem-se mais retorno aumentando-se o número de camadas do que o número de neurônios por camada.
- Uma abordagem mais simples é escolher um modelo com mais camadas e neurônios do que se realmente precisa e, em seguida, usar **early stopping** para se evitar que a rede **sobreajuste**.

Ajuste fino dos hiperparâmetros de um modelo

Funções de ativação

- Na maioria dos casos, principalmente com redes profundas, podemos usar a **função de ativação** do tipo **ReLU** nas camadas ocultas.
- A função **ReLU** é um pouco mais rápida de se calcular do que outras funções de ativação.
- Além disso, a probabilidade do **gradiente descendente** ficar preso em platôs é menor, graças ao fato de a função **ReLU** não saturar para valores de entrada grandes (em oposição às **funções de ativação logística** ou **tangente hiperbólica**, que saturam em 1).
- A função de ativação do tipo **softmax** é geralmente uma boa opção para a camada de saída em tarefas de classificação (quando as classes são mutuamente exclusivas, ou seja, quando um exemplo pertence somente a uma classe).
- Para tarefas de regressão, não usamos nenhuma função de ativação, que é também chamada de **função de ativação** do tipo **identidade**.

Avisos

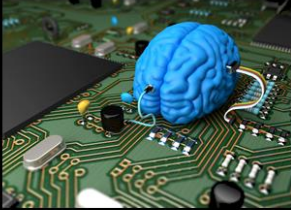
- Material, exemplos e lista de exercícios #12 já estão disponíveis.
- Todas as listas devem ser entregues até dia 02/12.

Obrigado!

Deep Learning



What society thinks I do



What my friends think I do



What other computer scientists think I do



What mathematicians think I do

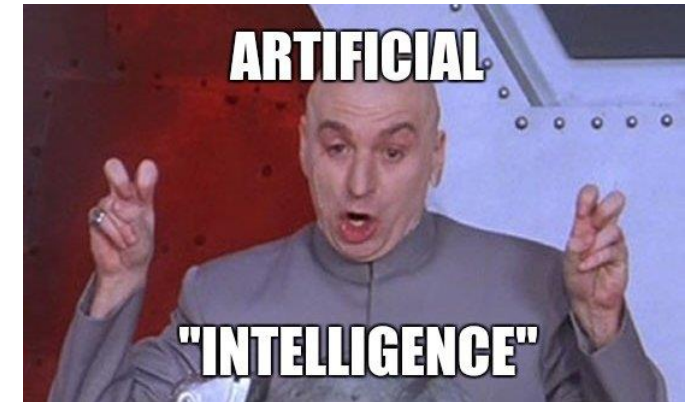


What I think I do

```
In [1]:  
import keras  
Using TensorFlow backend.
```

What I actually do

Me after training a neural network



Me spending four weeks training a model to 99.9% accuracy and then getting 10% on the test set

