Slide 1

TP555 - Inteligência Artificial e Machine Learning:
Redes Neurais Artificiais com TensorFlow

Inatel

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

Slide 2

Treinando uma Rede MLP com a API de alto nível do TensorFlow

- A maneira mais simples de treinar uma rede MLP com o TensorFlow é usar a API de alto nível ***TF.Learn***, que é bastante semelhante às APIs da biblioteca do Scikit-Learn.
- A classe ***DNNClassifier*** torna fácil o treinamento de uma rede neural profunda com qualquer número de camadas ocultas e uma camada de saída ***softmax*** usada para gerar 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)
```

Slide 3

Treinando uma Rede MLP com a API de alto nível do TensorFlow

- Se executarmos esse código no conjunto de dados MNIST (base de dados com imagens de dígitos escritos à mão), nós podemos obtemos um modelo que atinje 98,1% de precisão no conjunto de testes.

```
from sklearn.metrics import accuracy_score
y_pred = list(dnn_clf.predict(X_test))
accuracy_score(y_test, y_pred)

0.98180000000000001
```

- A biblioteca ***TF.Learn*** também fornece algumas funções úteis para avaliar os modelos:

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

- Sob os panos, a classe **DNNClassifier** cria todas as camadas de ***nós***, com base na ***função de ativação ReLU*** (podemos mudar isso configurando o hiperparâmetro ***activation_fn***). A camada de saída depende da ***função de ativação softmax***, e a função de custo utilizada é a da ***entropia cruzada***.
- **Exemplo**: MLPWithTensorFlowHighLevelAPI.ipynb

A entropia cruzada mede o desempenho de um modelo de classificação cuja saída é um valor de probabilidade entre 0 e 1. A entropia cruzada aumenta à medida que a probabilidade prevista diverge do rótulo, ou seja, da saída desejada. A entropia cruzada pode ser calculada como:

$$-(y \log(p) + (1-y) \log(1-p))$$

Treinando uma Rede DNN usando a API de baixo nível do TensorFlow

- Caso desejarmos ter mais controle sobre a arquitetura da rede neural, nós podemos, então, preferir usar a API de baixo nível do TensorFlow.
- À seguir, nós vamos criar o mesmo modelo dos slides anteriores usando a API de baixo nível e iremos implementar o algoritmo do gradiente descendente em mini-batch para treinar a rede neural para classificar imagens de dígitos escritos à mão.
- O primeiro passo é a ***fase de construção***, ou seja 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.

Slide 5

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
```

- Em seguida, usamos ***nós de 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")
```

A dimensão de X é apenas parcialmente definida. Sabemos que X será um tensor 2D (ou seja, uma matriz), com exemplos ao longo da primeira dimensão e atributos ao longo da segunda dimensão, e sabemos que o número de atributos será 28 x 28 (um atributos por pixel, pois lembre-se que neste exemplo iremos classficar imagens de dígitos escritos à mão e caeda imagens tem 28x28 pixels) , mas ainda não sabemos quantos exemplos cada mini-batch de treinamento conterá. Portanto, a forma de X é (None, n_inputs). Da mesma forma, sabemos que y será um tensor 1D com uma entrada por exemplo de treinamento, mas novamente não sabemos o tamanho do mini-batch de treinamento neste momento, e portanto a dimensão de y é (None).

Slide 6

Fase de Construção

- Agora vamos criar a rede neural propriamente dita.
- O ***nó de placeholder*** X atuará como a ***camada de entrada***.
- Durante a ***fase de execução***, ele será substituído por um mini-batch por vez.
- Observe que todos os exemplos em um mini-batch serão processados simultaneamente pela rede neural.
- Agora 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.
- A camada de saída também é muito semelhante às outras, mas ela usa uma ***função de ativação softmax*** em vez de ***uma função de ativação ReLU***.
- Para facilitar, vamos criar uma função chamada ***neuron_layer()*** que usaremos para criar uma camada por vez.
- Serão necessários alguns parâmetros para especificar as entradas: o número de ***nós***, a função de ativação e o nome da camada. O trecho da função é mostrado ao lado.

```
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
```

Slide 7

Fase de Construção

- Vamos analisar o código anterior linha por linha:
  - Primeiro, nós criamos um ***escopo de nome*** usando o nome da camada. Esse escopo conterá todos os ***nós de computação*** dessa camada de neurônios.
  - Em seguida, obtemos o número de entradas através da segunda dimensão da matriz de entrada, sendo a primeira dimensão o número de exemplos.
  - As próximas três linhas criam uma variável ***W*** que manterá 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***. A matriz ***W*** é inicializada aleatoriamente, usando-se uma distribuição Gaussiana normal truncada com desvio padrão igual a $2/\sqrt{}$n_inputs.
  - A próxima linha cria uma variável ***b*** para armazenar os valores de ***bias***, inicializada como 0, com um parâmetro de **bias** por ***neurônio***.
  - Em seguida, criamos um ***sub-grafo*** para calcular ***z = XW + b***. Essa implementação vetorizada calcula eficientemente as somas ponderadas das entradas, mais o termo de ***bias*** para cada ***neurônio*** da camada, para todas os exemplos do mini-bacth 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)), ou então apenas o valor de ***z*** (função de ativação linear).

- A criação de escopo de nomes para cada camada é opcional, mas a vizualização no grafo ficará muito melhor no TensorBoard se seus nós estiverem bem organizados.
- A matriz de pesos W tem dimensão igual a n_inputs x n_neurons.
- O uso de um desvio padrão igual a $2/\sqrt{}$n_inputs ajuda o algoritmo a convergir muito mais rapidamente. É importante inicializar pesos das conexões aleatoriamente para todas as camadas ocultas, para evitar simetrias que o algoritmo do gradiente descendente não conseguiria se desvencilhar.
- O uso de uma distribuição normal truncada em vez de uma distribuição normal regular garante que não haverá pesos com valores muito grandes , o que pode atrasar o processo de treinamento.
- No caso da variável b sendo incializada com zeros não há problema de simetria.

OBS.: Para a camada de saída, a função de ativação ***softmax*** é geralmente uma boa opção para tarefas de ***classificação*** (quando as classes são mutuamente exclusivas). Para tarefas de regressão, você pode simplesmente usar nenhuma função de ativação.

Slide 8

Fase de Construção

- Agora vamos usar a função ***neuron_layer*** para criar a rede neural profunda.
- A primeira camada oculta recebe X como entrada. A segunda 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 softmax***. Por razões de otimização, trataremos do cálculo do softmax posteriormente.

Observe que mais uma vez usamos um ***escopo de nome*** para deixar o código mais claro.

Como sabemos, o TensorFlow apresenta muitas funções úteis para criar camadas de redes neurais; portanto, muitas vezes não há necessidade de se definir sua própria função ***neuron_layer()*** como acabamos de fazer. Por exemplo, a função ***fully_connected()*** do TensorFlow cria uma camada totalmente conectada, onde todas as entradas são conectadas a todos os neurônios da camada. Ela cuida da criação das variáveis de pesos e bias, com a estratégia de inicialização adequada, e usa a função de ativação ReLU por padrão (podemos mudar isso usando o argumento ***activation_fn***). O código abaixo utiliza a função ***fully_connected()*** ao invés da função ***neuron_layer()***:

```
from tensorflow.contrib.layers import fully_connected
with tf.name_scope("dnn"):
  hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
  hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
  logits = fully_connected(hidden2, n_outputs, scope="outputs", activation_fn=None)
```

OBS.: O modulo contrib foi removido no TensorFlow e portanto, ao invés de utilizar a função fully_connected() deve se utilizer a classe Dense() do modulo tf.keras.layers. Portanto, agora é preferível usar a classe tf.keras.layers.Dense, porque qualquer coisa no módulo contrib pode mudar ou ser excluída sem aviso prévio. A classe Dense() é quase idêntica à função fully_connected(), exceto por algumas pequenas diferenças:

- vários parâmetros foram renomeados: scope tornou-se name, activation_fn tornou-se activation (e da mesma forma que o sufixo _fn foi removido de outros parâmetros como normalizer_fn), weights_initializer tornou-se kernel_initializer etc.
- a ativação padrão agora é None em vez de tf.nn.relu.

- Ao invés de passar os dados diretamente para a classe, agora deve-se instancia-la e sem seguida passar a entrada para o objeto.

Slide 9

Fase de Construção

- Agora que já temos o modelo de rede neural pronto, precisamos definir a ***função de custo*** que usaremos para treiná-lo.
- Iremos usar a medida de ***entropia cruzada*** como ***função de custo***. O TensorFlow fornece várias funções para calcular a ***entropia cruzada***.
- Para isso, usaremos a função ***sparse_soft_max_cross_entropy_with_logits()***. Ela é equivalente a aplicar a ***função de ativação softmax*** e depois computar a ***entropia cruzada***, mas é mais eficiente e cuida de casos como quando os valores de ***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 softmax***) e espera rótulos na forma de números inteiros que variam de 0 ao número total de classes menos 1.
- Isso nos dará um tensor 1D contendo a ***entropia cruzada*** para cada exemplo. Em seguida, podemos usar a função ***reduce_mean()*** do TensorFlow para calcular a ***entropia cruzada média*** sobre todos os exemplos.

Referência básica sobre entropia cruzada: https://en.wikipedia.org/wiki/Cross_entropy

A entropia cruzada penaliza modelos que estimam uma baixa probabilidade para a classe-alvo.

A função ***sparse_softmax_cross_entropy_with_logits()*** é equivalente a aplicar a função de ativação softmax e depois computar a entropia cruzada, mas é mais eficiente e cuida adequadamente de casos como logits iguais a 0. É por isso que não aplicamos a ativação softmax na função anterior. Há também outra função chamada ***softmax_cross_entropy_with_logits()***, que recebe rótulos na forma de vetores codificados como one-hot-encoding em vez de ints de 0 ao número de classes menos 1.

Slide 10

Fase de Construção

- Temos agora o modelo de rede neural e a ***função de custo*** e agora, como fizemos anteriormente, precisamos definir um ***otimizador*** que irá ajusta os ***pesos*** do modelo para minimizar a ***função de custo***.

```
learning_rate = 0.01
with tf.name_scope("train"):
  optimizer = tf.train.GradientDescentOptimizer(learning_rate)
  training_op = optimizer.minimize(loss)
```

- 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 previsão da rede neural 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. Em seguida, convertemos esses valores booleanos em floats e calculamos a média. Isso dará a precisão geral da rede.

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

- 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.

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

Fase de execução

- Primeiro, vamos carregar a base de dados MNIST usando uma função disponibilizada pelo TensorFlow. Em seguida, eles são re-dimensionados e escalonados (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
```

Slide 12

Fase de execução

- Em seguida, podemos treinar o modelo:

```
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")
```

- Esse código abre uma ***sessão*** do TensorFlow e executa o ***nó init*** que por sua vez inicializa todas as variáveis.
- Em seguida, o ciclo de treinamento principal é executado: em cada época, o código itera através de vários mini-batches que correspondem ao tamanho do conjunto de treinamento.
- Cada mini-batch é lido 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 o modelo com último mini-batch utilizado e com o conjunto de validação e imprime o resultado.
- Finalmente, os parâmetros do modelo são salvos em disco.

```
def shuffle_batch(X, y, batch_size):
    rnd_idx = np.random.permutation(len(X))
    n_batches = len(X) // batch_size
    for batch_idx in np.array_split(rnd_idx, n_batches):
        X_batch, y_batch = X[batch_idx], y[batch_idx]
        yield X_batch, y_batch
```

Slide 13

Usando o modelo treinado

- Agora que a rede neural está treinada, nós podemos usá-la para fazer previsões.
- Para fazermos isso, reutilizamos o código da ***fase de construção***, mas alteremos a ***fase de execução*** da seguinte maneira:

```
with tf.Session() as sess:
  saver.restore(sess, "./my_model_final.ckpt")
  X_new_scaled = [...] # some new images (scaled from 0 to 1)
  Z = logits.eval(feed_dict={X: X_new_scaled})
  y_pred = np.argmax(Z, axis=1)
```

- Inicialmente, carregamos os parâmetros do modelo armazenados em disco.
- Em seguida, carregamos algumas novas imagens 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()***).
- **Exemplo**: MLPWithTensorFlowLowLevelAPI.ipynb

OBS.:

- Lembre-se de aplicar a mesma escala aos atributos dos dados de treinamento (nesse caso, dimensione-os de 0 a 1).
- Se você quiser conhecer todas as probabilidades estimadas da classe, é necessário aplicar a função ***softmax()*** aos logits.

Ajuste fino dos hiperparâmetros de uma rede neural

- A flexibilidade 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 peso e muito mais.
- Sendo assim, como saberíamos qual combinação de hiperparâmetros é a melhor para uma dada tarefa?

Slide 15

Ajuste fino dos hiperparâmetros de uma rede neural

- Nós poderímos usar ***GridSearch*** com ***validação cruzada*** para encontrar os hiperparâmetros ótimos, mas como existem muitos hiperparâmetros para 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 ***pesquisa aleatória***. Uma outra opção seria usar uma ferramenta como o ***Oscar***, que implementa algoritmos mais complexos para nos ajudar a encontrar rapidamente um bom conjunto de hiperparâmetros.
- Ajuda a ter uma ideia de quais valores são razoáveis para alguns hiperparâmetros, para que nós possamos restringir o espaço de pesquisa.

Pesquisa aleatória:
https://www.tensorflow.org/tensorboard/hyperparameter_tuning_with_hparams
Oscar: http://oscar.calldesk.ai/

Slide 16

Ajuste fino dos hiperparâmetros de uma rede neural: Número de camadas ocultas

- Para muitos problemas, nós podemos começar com apenas uma única camada oculta e mesmo assim obteremos 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 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.
- Além disso, redes profundas tiram proveito da estrutura hierárquica presente em dados do mundo real: camadas ocultas próximas à entrada modelam estruturas de baixo nível (por exemplo, segmentos de linha de várias formas e orientações), camadas ocultas intermediárias combinam essas estruturas de baixo nível para modelar estruturas de nível intermediário (por exemplo, quadrados, círculos, etc.) e as camadas ocultas mais próximas à saída juntamente com a camada de saída combinam essas estruturas intermediárias para modelar estruturas de alto nível (por exemplo, faces).

camadas ocultas inferiores modelam estruturas de baixo nível (por exemplo, segmentos de linha de várias formas e orientações), camadas ocultas intermediárias combinam essas estruturas de baixo nível para modelar estruturas de nível intermediário (por exemplo, quadrados, círculos) e as camadas ocultas mais altas A camada de saída combina essas estruturas intermediárias para modelar estruturas de alto nível (por exemplo, faces)

Slide 17

Ajuste fino dos hiperparâmetros de uma rede neural: Número de camadas ocultas

- 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.
- Desta forma, uma rede não precisa aprender do zero todas as estruturas de baixo nível que ocorrem nos dados do mundo real, ela pode reutilizar os pesos de camadas mais baixas (as quais já aprenderam as estruturas de baixo nível) e precisará apenas aprender (treinar o restante das camadas) apenas as estruturas de alto nível.
- 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. O treinamento será muito mais rápido e exigirá muito menos dados.

Ajuste fino dos hiperparâmetros de uma rede neural: 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 disso é que muitas estruturas de baixo nível podem se unir à um número muito menor de estruturas de alto nível.
- 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 uma rede neural: Funções de ativação

- Na maioria dos casos, podemos usar a ***função de ativação 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 que 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 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 à uma classe).
- Para tarefas de regressão, podemos simplesmente usar nenhuma função de ativação, que é também chamada de ***função de ativação identidade***.

Slide 20

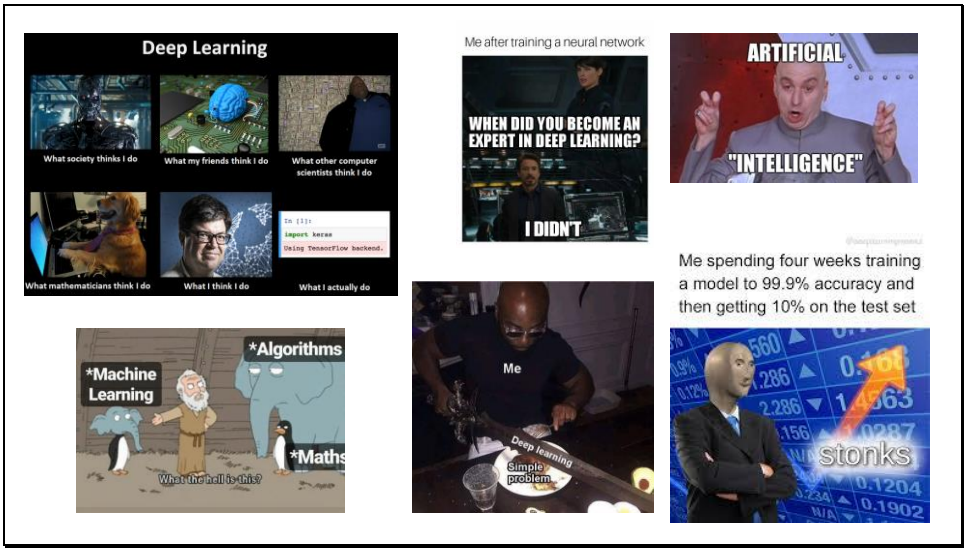
Avisos

- O material já se encontra disponível no site.
- Lista #12 já está disponível no site e pode ser entregue até 30/06.
- Todas as listas, exceto a #12, devem ser entregues até dia 23/06.
- Prova será disponibilizada no site até Domingo, dia 21/06 e deve ser entregue até a meia noite do dia 25/06 (Quinta-Feira).
- Os exercícios de todas as listas estão também disponíveis no site para consulta.
- Todo material entregue fora do prazo será penalizado.

Slide 21

Obrigado!

Slide 22


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
What I actually do
Me after training a neural network
WHEN DID YOU BECOME AN EXPERT IN DEEP LEARNING?
I DIDN'T
ARTIFICIAL
"INTELLIGENCE"
*Machine Learning
*Algorithms
*Maths
What the hell is this?
Me
Deep learning
Simple problem
Me spending four weeks training a model to 99.9% accuracy and then getting 10% on the test set
stonks