Deep Learning com deeplearn.js

Roberto Stelling[1]

[1]Programa de Pós-Graduação em Informática (PPGI) – Universidade Federal do Rio de Janeiro (UFRJ)
Caixa Postal 68.530 – 21.941-590 – Rio de Janeiro – RJ – Brasil

roberto@stelling.cc

Abstract. *This article describes the implementation of a neural network using the framework deeplearn.js and the construction of a data visualization to visualize the neural network iterative training process.*

Resumo. *Este artigo descreve a implementação de uma rede neural utilizando a biblioteca deeplearn.js e a construção de uma visualização de dados para acompanhar o processo iterativo de treinamento da rede neural.*

1. deeplearn.js

Como definido em deeplearnjs (2017), o deeplearn.js é uma biblioteca TypeScript com aceleração por hardware para desenvolvimento de soluções de machine intelligence que permite treinar redes neurais em um navegador ou executar modelos pré-treinados em modo de inferência.

A biblioteca possui dois tipos de APIs:

• APIs com modelo de execução imediata, semelhante à biblioteca de computação científica NumPy da linguagem de programação Python.

• APIs com modelo de execução "diferido" que espelham as APIs do TensorFlow, também uma biblioteca de computação científica da linguagem de programação Python.

O deeplearn.js foi originalmente desenvolvido pela equipe do Google Brain PAIR para construir ferramentas de aprendizado interativas para o navegador, mas pode ser usada em vários outros contextos como educação, para modelar entendimento e até mesmo projetos de arte.

Como outros projetos da Google, o deeplearn.js faz uso de TypeScript, um superconjunto tipado de JavaScript com características de programação orientada a objetos que pode ser transpilado diretamente para JavaScript. Apesar disto a biblioteca também está disponível em JavaScript.

2. Machine Learning

De acordo com Murphy (2010) machine learning é um conjunto de métodos que pode automaticamente detectar patrões em dados e, então, usar os padrões descobertos para prever dados futuros ou executar outros tipos de tomada de decisão em ambientes de incerteza (como planejar sobre como coletar mais dados!).

Machine learning é usualmente dividido em dois tipos principais. Na abordagem preditiva ou supervisionada, o objetivo é aprender um mapeamento de entradas x para saídas y, dados pares etiquetados de entrada-saída $D=\{(x_i,y_i)\}^N_{i=1}$. Aqui, D é chamado de conjunto de treinamento e N é o número de exemplos de treinamento. O segundo

tipo principal de machine learning é a abordagem descritiva ou aprendizado não-supervisionado. Aqui temos apenas as entradas, $D=\{(x_i)\}^{N}_{i=1}$, e o objetivo é encontrar "padrões interessantes" nos dados.

Há ainda um terceiro tipo de machine learning, conhecido como reinforcement learning, que é útil para aprender como agir ou se comportar na presença de sinais ocasionais de premiação ou punição (por exemplo, considere como uma criança aprende a caminhar).

Machine learning é igualmente importante na academia e no mundo dos negócios: em 2016 Eric Schmidt, membro do board da Google, disse: "Machine learning será responsável por todo IPO de sucesso em 5 anos", TechWorld (2016).

3. Redes neurais artificiais

A maioria dos modelos de redes neurais artificiais são baseados nos neurônios weighted-sum-and-threshold, dos pioneiros McCullogh e Pitts.

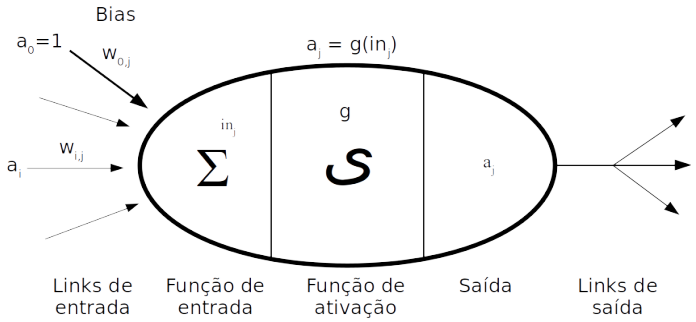


Figura 1: Modelo de neurônio McCullock & Pitts (1943)

Embora este seja o modelo predominante de redes neurais artificiais é importante citar os sistemas neurais sem peso, que são baseados em redes de nós de memória de acesso randômico (RAM), Aleksander et al. (2009).

O exemplo de rede neural implementado nesse artigo é baseado no modelo de neurônio de McCulloch and Pitts, com aprendizado supervisionado, implementado com backpropagation e gradient descent, como mostrado na figura 2, adaptada de Chollet, F. (2017).

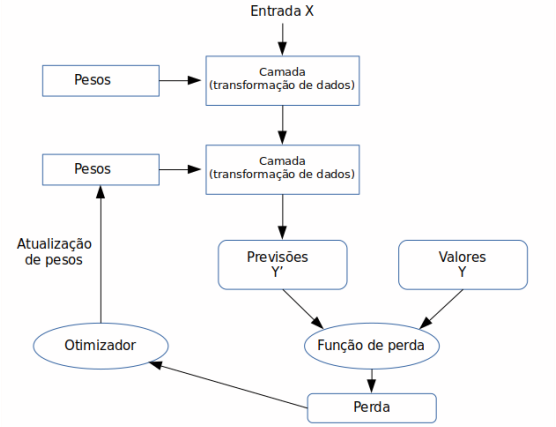


Figura 2: Modelo de backpropagation com gradient descent. Adaptado de Chollet, F. (2017)

4. Deep Learning

Há várias definições de Deep Learning, mas de acordo com Deng, L. e Yu, D. (2014), há em comum, na maioria destas definições, dois aspectos principais:

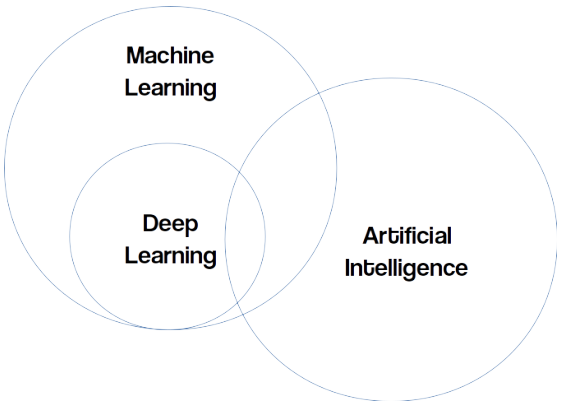
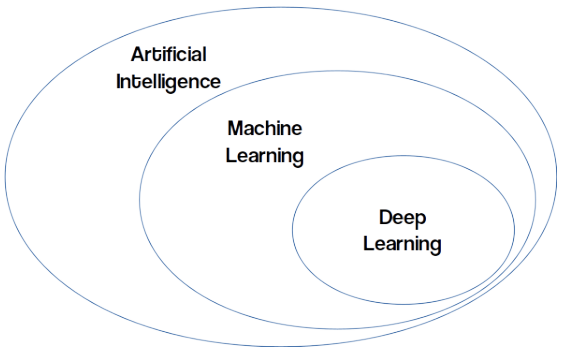
1. Modelos que consistem de múltiplas camadas ou estágios de processamento não linear;
2. Métodos para aprendizado supervisionado ou não de representação de características em camadas sucessivas e cada vez mais abstratas.

Deep Learning está na intercessão entre áreas de pesquisa de redes neurais, inteligência artificial, modelos gráficos, otimização, reconhecimento de padrões e processamento de sinais.

É interessante mencionar que o mercado ainda tem opiniões inconsistentes sobre o relacionamento entre Inteligência Artificial, Machine Learning e Deep Learning, como mostra a Tabela 1, com imagens adaptadas dos livros grokking Deep Learning, de Trask, W. (2017) e Deep Learning with Python, de Chollet, F. (2017).

A nossa visão é consistente com a visão de Chollet, F. (2017), onde Deep Learning é uma área de aplicação de Machine Learning, que por sua vez é uma área da Inteligência Artificial.

Tabela 1: Visão da relação entre AI, ML e DL em duas publicações recentes

grokking Deep Learning (2017)	Deep Learning with Python (2017)
	

As definições de Deep Learning na literatura comercial se referem, em geral, às arquiteturas de redes neurais com peso, com vários níveis completamente conectados de neurônios, sem especificar exatamente quantos níveis são necessários e sem importar se o aprendizado é supervisionado ou não.

5. Implementação

Baseando-se em um exemplo existente na biblioteca de demos do deeplearn.js adaptamos uma rede neural para simular uma função de conversão de cores RGB para sua cor complementar e desenhamos e implementamos uma interface de visualização de dados para esta rede neural, que permite acompanhar o processo de aproximações sucessivas da rede durante o seu treinamento.

Para gerar as cores complementares fazemos uso do algoritmo descrito por Edd (2016)

em um questionamento no site StackOverflow. Esse algoritmo foi utilizado para gerar as cores complementares e servir de entrada dos exemplos Y classificados e comparados com os valores Y' previstos. Ou seja, o modelo foi treinado com os valores calculados a partir deste algoritmo e aproxima as previsões aos valores calculados em iterações sucessivas.

As cores são representadas como RGB, portanto temos um espaço amostral de 256^3 cores ~= 16,7 milhões de cores. Foram geradas, randomicamente, 10.000 cores para treinamento em batches de 300 cores. A cada 5 iterações atualizamos a visualização dos dados aproximados pela rede neural.

A rede neural possui 64 nós na primeira camada, 32 nós na segunda camada, 16 nós na terceira camada e 3 nós na camada de previsão. Todas as camadas são completamente conectadas e todas as unidades eram originalmente do tipo ReLU. Unidades ReLU são definidas como f(x)=0(x<0)+1(x>=0)(x). Ou seja, f(x) = 0 se x<0 e f(x) = x se x>= 0, ou ainda f(x) = max(0, x).

A implementação faz uso da característica do deeplearn.js de possuir uma camada de abstração de operações matemáticas na GPU. Nesse modelo é possível especificar quais são as operações que serão executadas e a biblioteca se encarrega de fazer as operações na GPU e descarregar para a CPU quando requisitado. A biblioteca também se encarrega de fazer um "fallback" de operações para a CPU caso uma GPU não esteja disponível, sem que seja necessário fazer nenhuma previsão para essa possibilidade no código original.

5.1. ReLU Morto

Apesar de o modelo convergir, na maioria das execuções, observamos que, em alguns casos, um dos valores RGB inferido se mantinha como 0 do início do treinamento até a parada do modelo pela quantidade de iterações. Como mostrado na figura 3, após 4245 iterações.

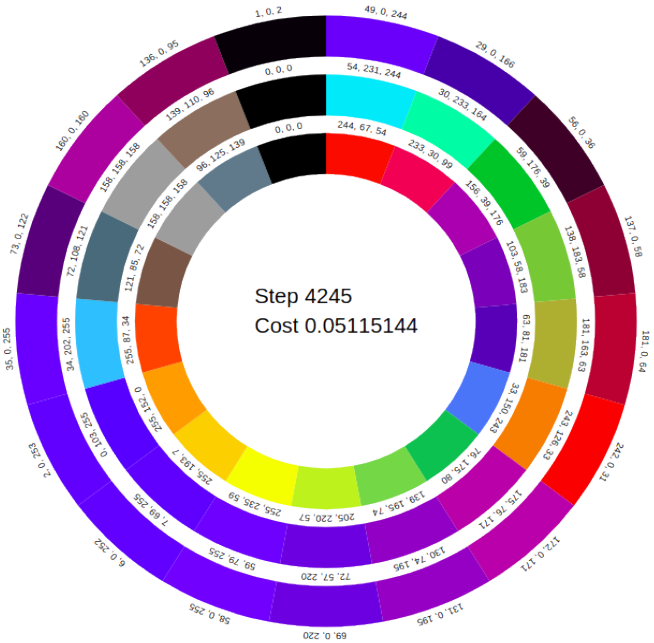


Figura 3: Exemplo de execução com parada por limite de iterações

Importante frisar que o modelo da nossa implementação interrompe o treinamento quando a função de utilidade alcança um valor pré-determinado ou quando o número de iterações ultrapassa outro valor pré-determinado

Na figura 3, o anel mais interno representa as cores originais, e seus valores RGB, o anel mediano representa as cores complementares calculadas pelo algoritmo, e seus valores RGB, e o anel mais externo representa as cores inferidas pelo modelo e seus valores RGB. A cada iteração esse anel externo é atualizado com as novas predições do modelo. Nesse exemplo em particular o canal verde (G) se manteve como 0 durante todo o processo de treinamento/inferência.

A nossa suposição para o problema de convergência foi que o modelo provavelmente sofria do problema de ReLU morto, descrito em CS231n (2017), e também bias shift, citado por Clevert et al. (2016). Esses problemas podem ser resolvidos com a troca das unidades ReLU por Leaky ReLU, como indicado em CS231n (2017): Leaky ReLUs são uma das alternativas de resolver o problema de "ReLU morto". Ao invés da função ser 0 quando x<0, uma Leaky ReLU possui uma pequena inclinação negativa (de 0,01 aproximadamente). Desta forma a função é computada como: f(x)=1(x<0)(αx)+1(x>=0)(x), onde α é uma constante de pequeno valor. Após a troca de todas unidades de ReLU para Leaky ReLU com α = 0.001, não verificamos mais nenhum caso de parada por limite no número de iterações. A figura 4, retirada de Clevert et al. (2016), compara as funções ELU (α=1,0), LReLU (Leaky ReLU, α=0,1), ReLU e SreLU.

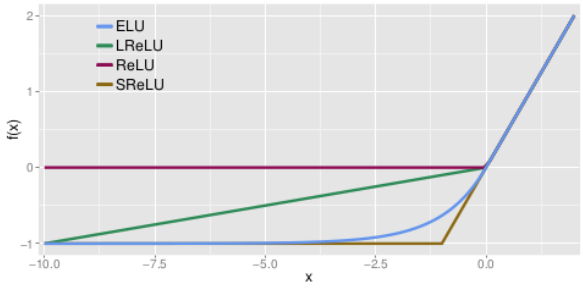


Figura 4: Gráficos de funções de ativação, de Clevert et al. (2016)

É importante observar que a troca das unidades de ReLU por Leaky ReLU foi possível pelas duas funções estarem disponíveis no deeplearn.js. O objetivo original era trocar as unidades ReLU para ELU, que Clevert et al. (2016) indicam ter melhor performance que as outras alternativas, porém a função ELU ainda está em fase de implementação no deeplearn.js.

6. Conclusões

Após a implementação da versão final, com código disponível no Github em Stelling (2017), concluímos que:

- O modelo da biblioteca deeplearn.js é eficiente e adequado para a criação de redes neurais de pequeno e médio porte no navegador.
- Confirmamos a afirmação que redes neurais altamente conectadas são capazes de aproximar uma função complexa.

- Verificamos com testes em notebooks com e sem GPU, embora os resultados não tenham sido comentados nesse artigo, que diferenças de hardware e eventual fallback de GPU para CPU impactam drasticamente a duração da execução do algoritmo implementado.
- O modelo é sensível ao tipo de unidade utilizada na rede neural, afetando, e eventualmente impedindo, a convergência do modelo, pelo menos dentro do critério do limite de 4245 iterações. A substituição das unidades de ReLU para Leaky ReLU foi suficiente para resolver o problema encontrado.
- O modelo de visualização desenvolvido, e em particular a atualização da visualização a cada 5 iterações de treinamento, parece ser efetivo no entendimento de conceitos de redes neurais e, incidentalmente, por este ser um modelo de predição de cores, no entendimento de conceitos de percepção visual.
- Em praticamente todas execuções as cores aproximadas diferiam dos resultados esperados em, pelo menos, dois dos três componentes RGB. Porém o resultado visual das cores alcançadas era sempre satisfatório e praticamente indiferenciáveis para o olho humano, como mostrado no exemplo da Figura 5.

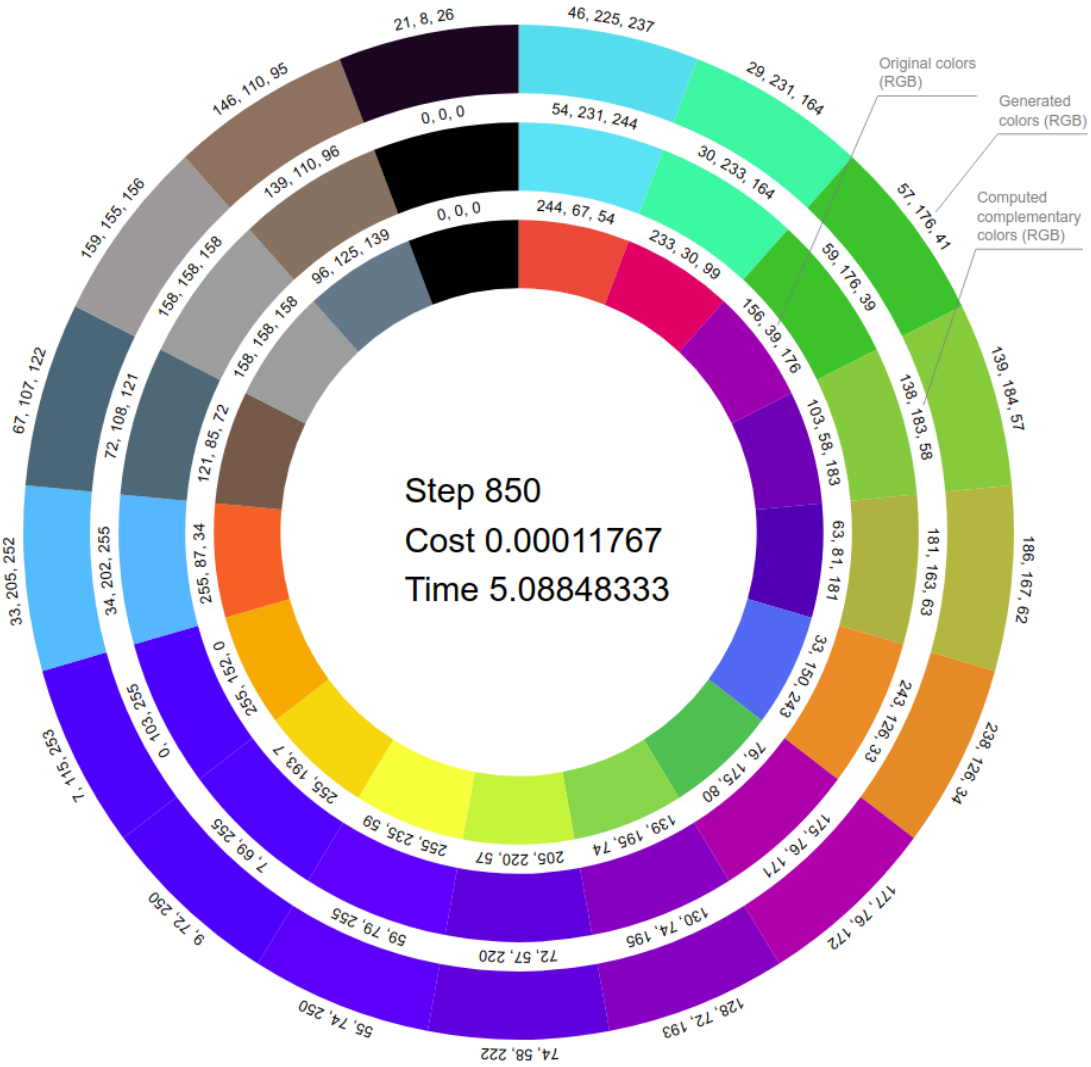


Figura 5: Final da execução com cores aproximadas

Referências

Aleksander, I., De Gregorio, M., França, F.M.G., Lima, P.M.V. e Morton, H. (2009) "A brief introduction of Weightless Neural Systems", ESANN'2009 proceedings.

Chollet, F. (2017) Deep Learning with Python, Manning Publications Co. 1st edition.

Clevert, D., Unterthiner, T. e Hochreiter, S. (2016) "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)", Publicado como paper de conferência no ICLR 2016

CS231n (2017), "Convolutional Neural Networks for Visual Recognition", Spring 2017, Stanford.edu, http://cs231n.github.io/neural-networks-1/ Acesso: dezembro 2017

deeplearnjs (2017) "deeplearn.js" http://deeplearnjs.org/ Acesso: dezembro 2017

Deng, L. Yu, Dong. (2014) Deep Learning Methods and Applications, em Foundations and Trends in Signal Processing, 7:3-4, p. 198-387

Edd (2016), "Complementary color calculation", https://stackoverflow.com/a/37657940 Acesso: dezembro 2017

McCulloch, W. S. e Pitts W. (1943) "A Logical Calculus of the Ideas Immanent in Nervous Activity", Em: Bulletin of Mathematical Biophysics, volume 5, p. 115-133, 1943

Murphy, K. P. (2012) Machine Learning A Probabilistic Approach, The MIT Press

Russell, S. e Norvig, P. (2010) Artificial Intelligence A Modern Approach. Prentice Hall, 3rd edition.

Stelling, R. (2017) "Complementary Color Prediction" https://github.com/RobStelling/Complementary-Color-Prediction, Acesso: dezembro 2017.

TechWorld (2017) "Google's Eric Schmidt: Machine learning will be basis of 'every huge IPO' in five years", https://www.techworld.com/data/eric-schmidt-machine-learning-will-be-basis-of-every-huge-ipo-in-five-years-3637206/ Acesso: dezembro 2017

Trask, A. W. (2017) grokking Deep Learning. Manning Publications Co. Pre-edition.

7. Anexo – Introdução ao deeplearn.js

Em todos os exemplos deste anexo utilizaremos trechos de código TypeScript. Para obter o código JavaScript correspondente pode ser necessário remover anotações de tipo e converter definições TypeScript para JavaScript.

Por exemplo, a código console.log(await ndarray.data()), seria escrito em JavaScript como ndarray.data().then(data => console.log(data)), utilizando a notação de funções =>.

7.1. NDArray

A unidade central de dados do deeplearn.js é o NDArray. Um NDArray é formado por um conjunto de valores de ponto flutuante organizado em um vetor com um número arbitrário de dimensões. Os objetos NDArray possuem um atributo shape, que define a a forma do seu conjunto de dados. A biblioteca fornece subclasses para NDArrays simples: Scalar, Array1D, Array2D, Array3D e Array4D.

Exemplo de definição de uma matriz 2x3:

```
const shape = [2,3]; // 2 linhas, 3 colunas
const a = Array2D.new(shape, [1.0, 2.0, 3.0, 10.0, 20.0, 30.0]);
```

NDArrays podem armazenar dados na GPU como WebGLTexture (a interface WebGLTexture é parte da API WebGL e representa um objeto opaco de texto que provê armazenamento e estado para operações de textura) onde cada pixel armazena um valor de ponto flutuante, ou diretamente na CPU como um TypedArray JavaScript. Se for feita uma chamada a NDArray.getValuesAsync(), em um NDArray da GPU, a biblioteca baixará a textura para a CPU e removerá a textura original. Idealmente os dados devem ser mantidos o máximo de tempo possível na GPU para maximizar a performance da aplicação desenvolvida.

7.2. NDArrayMath

A classe base NDArrayMath define um conjunto de funções matemáticas que operam nos NDArrays.

NDArrayMathCPU

As operações matemáticas, executadas na CPU, bloqueiam e são executadas imediatamente no TypedArray em JavaScript padrão. A princípio usar NDArrayMathCPU não oferece ganho de performance sobre operações matemáticas normais em JavaScript.

NDArrayMathGPU

Quando se utiliza a implementação NDArrayMathGPU as operações matemáticas enfileiram programas shader (rotinas de sombreamento em GPUs) para serem executados diretamente na GPU. De forma distinta da NDArrayMathCPU, estas operações não bloqueiam mas o usuário, se desejar, pode sincronizar a CPU com a GPU chamando getValuesAsync() no NDArray, como descrito no exemplo abaixo.

Exemplo de cálculo da diferença média quadrática entre duas matrizes:

```
const math = new NDArrayMathGPU(); // Define o espaço para as
```

```
operações

const a = Array2D.new([2, 2], [1.0, 2.0, 3.0, 4.0]);
const b = Array2D.new([2, 2], [0.0, 2.0, 4.0, 6.0]);

// Chamadas math não bloqueantes
const diff = math.sub(a, b);
const squaredDiff = math.elementWiseMul(diff, diff);
const sum = math.sum(squaredDiff);
const size = Scalar.new(a.size);
const average = math.divide(sum, size);

console.log('diferença média quadrática: ' + await average.val());
```

Importante: A princípio não se deve chamar funções como get() ou getValuesAsync() entre as operações matemáticas da GPU a não ser para depuração. Essas chamadas forçam a descarga da textura da GPU para a CPU e as chamadas subsequentes de NDArrayMathGPU terão que ser recarregadas para uma nova textura, com forte impacto na performance da solução.

7.3. Treinamento

Grafos de fluxo de dados em deeplearn.js utilizam um modelo de execução "atrasada", da mesma forma que a biblioteca TensorFlow do Python. É necessário seguir as seguintes etapas para a execução de um fluxo de dados:

1. Construir um grafo que represente as operações que serão executadas;
2. Treinar e/ou inferir sobre o grafo. NDArrays são passados para o grafo com o uso de FeedEntrys.

Nota: NDArrayMath e NDArrays são suficientes para o modo de inferência. Só é necessário um grafo em caso de treinamento.

7.4. Grafos e Tensores

O objeto Graph é a classe núcleo para a construção de grafos de fluxo de dados. Objetos Graph não possuem dados NDArray, mas possuem estruturas que representam a conectividade entre as operações que serão executadas.

Quando um método de grafo é chamado para acrescentar uma operação, o método retorna um objeto Tensor que só possui informação de conectividade e forma. Veja mais sobre tensores na seção 7.6.

Exemplo de grafo que multiplica uma entrada por uma variável:

```
const g = new Graph();

const inputShape = [3];
// placeholder representa o dado de entrada. É o local que
// alimentaremos com um NDArray de entrada quando executarmos o grafo
const inputTensor = g.placeholder('input', inputShape);
const labelShape = [1];
const labelTensor = g.placeholder('label', labelShape);

// variable armazena valores que podem ser atualizados em treinamento.
// Já um placeholder não é atualizado durante o treinamento.
// Nesse exemplo inicializamos a variável multiplier com valores
// randômicos
const multiplier = g.variable('multiplier', Array2D.randNormal([1,
3]));
```

```
// Métodos de alto nível de graph recebem Tensores e
// retornam Tensores
const outputTensor = g.matmul(multiplier, inputTensor);
const costTensor = g.meanSquaredCost(outputTensor, labelTensor);

// Tensores, como NDArrays, possuem o atributo shape
console.log(outputTensor.shape);
```

7.5. Session e FeedEntry

A execução de grafos é comandada por objetos Session. O objeto FeedEntry (similar ao feed_dict do TensorFlow) provê dados para a execução, alimentando um valor para o Tensor de um NDArray dado.

Nota sobre batches: o deeplearn.js ainda não possui mecanismos de tratamento de batches como um objeto ou dimensão externa para suas operações. Isso significa que toda operação de alto nível em grafos, como também toda função matemática, opera em exemplos unitários. Por outro lado batching é importante e atualizações de peso deveriam operar na média de gradientes de um batch. O deeplearn.js consegue simular a operação com batches usando um InputProvider em FeedEntrys de treinamento para fornecer entradas, ao invés de usar NDArrays diretamente. O InputProvider será chamado em cada item em um batch. No exemplo a seguir, para garantir que os conjuntos de entradas estarão mesclados e sincronizados em pares entrada/saída, utilizamos a função InMemoryShuffledInputProviderBuilder.

Vejamos o treinamento com o objeto Graph “g” do exemplo anterior:

```
const learningRate = 0.00001;
const batchSize = 3;
const math = new NDArrayMathGPU();

const session = new Session(g, math);
const optimizer = new SGDOptimizer(learningRate);

const inputs: Array1D[] = [
  Array1D.new([1.0, 2.0, 3.0]),
  Array1D.new([10.0, 20.0, 30.0]),
  Array1D.new([100.0, 200.0, 300.0])
];

const labels: Array1D[] = [
  Array1D.new([4.0]),
  Array1D.new([40.0]),
  Array1D.new([400.0])
];

// Mescla entradas e labels e os mantém sincronizados
const shuffledInputProviderBuilder =
  new InCPUMemoryShuffledInputProviderBuilder([inputs, labels]);
const [inputProvider, labelProvider] =
  shuffledInputProviderBuilder.getInputProviders();

// Mapeia tensores com InputProviders
const feedEntries: FeedEntry[] = [
  {tensor: inputTensor, data: inputProvider},
  {tensor: labelTensor, data: labelProvider}
];
// Batches precisam ser executados de forma explícita
const NUM_BATCHES = 10;
for (let i = 0; i < NUM_BATCHES; i++) {
```

```
  // Trainamento precisa de um tensor de custo para minimizar
  // Treina um batch. Retorna a média do custo como um escalar
  const cost = session.train(
      costTensor, feedEntries, batchSize, optimizer,
CostReduction.MEAN);

  console.log('last average cost (' + i + '): ' + await cost.val());
}

Depois do treinamento, é possível inferir a partir do grafo:
const testInput = track(Array1D.new([0.1, 0.2, 0.3]));

// session.eval pode receber NDArrays como dados de entrada.
const testFeedEntries: FeedEntry[] = [
  {tensor: inputTensor, data: testInput}
];

const testOutput = session.eval(outputTensor, testFeedEntries);

console.log('---inference output---');
console.log('shape: ' + testOutput.shape);
console.log('value: ' + await testOutput.val());
```

7.6. NDArrays, tensores e números

Tensores matemáticos

Matematicamente, um "tensor" é o objeto mais básico de álgebra linear, uma generalização de números, vetores e matrizes. Um vetor pode ser imaginado como uma lista unidimensional de números; uma matriz como uma lista bidimensional de números. Um tensor simplesmente generaliza este conceito para listas n-dimensionais de números. Um tensor é qualquer arranjo de componentes numéricos (até mesmo strings ou outros tipos de dados) organizados em qualquer vetor retangular multidimensional.

Um tensor no deeplearn.js possui várias propriedades:

- Possui um type (tipo), que descreve os tipos de cada um dos seus componentes, por exemplo: integer, float etc. No momento o deeplearn.js suporta apenas tensores do tipo float32.
- Possui um shape (forma), que é uma lista de inteiros que descreve o formato retangular do array de componentes. Quando falamos que uma matriz é "quatro por quatro" (ou seja [4x4]) estamos descrevendo o formato, ou shape, da matriz.
- Possui um rank, que é o comprimento do seu formato/shape; ou seja, é a dimensão do vetor de componentes. Um escalar tem rank 0; um vetor tem rank 1; uma matriz tem rank 2.

Tabela 1

Exemplo de Tensor	***Type***	***Shape***	***Rank***
Escalar: 3,0	float	[]	0
Vetor: (1, 5, -2)	int	[3]	1
Matrix 2x2	int	[2,2]	2

7.7. Number[], NDArray, Tensor: Três tipos de dados para tensores matemáticos

O mesmo objeto que um matemático chamaria de tensor é representado de três formas diferentes no deeplearn.js. A discussão acima (rank, shapes e types) se aplica a todos eles, mas são diferentes entre si e é importante reforçar essa distinção:

- number[] é o tipo javascript subjacente que corresponde a um array de números. Na prática a notação JavaScript correspondente seria number, para um tensor de rank-0, number[] para um tensor de rank-1, number[][] para um tensor de rank-2 e assim por diante. A maioria dos dados no deeplearn.js são mais complexos que esse exemplo, mas é desta forma que são implementados na biblioteca;
- NDArray é a implementação mais poderosa do deeplearn.js. Cálculos envolvendo NDArrays podem ser executados na GPU do cliente, que é a vantagem fundamental da biblioteca. Esse é o formato que os dados de tensores assumem quando os cálculos acontecem. Por exemplo, o NDArray é o formato de dados retornado quando a função Session.eval é chamada (e também o formato das entradas para FeedEntry). É possível converter entre NDArray e number[] usando NDArray.new(number[]) e NDArray.get([indices]);
- Um tensor é um "saco vazio": não contém dados. É apenas um marcador usado quando um Graph é construído, e que registra o formato e o tipo do dado que eventualmente serão representados. Um tensor não possui os valores reais dos seus componentes. Por outro lado, por ter informações de forma e tipo, têm um papel importante para capturar erros no momento de construção do Graph; por exemplo, se você deseja multiplicar uma matriz 2x3 por uma matriz 10x10, o grafo pode retornar um erro quando o nó é criado, antes mesmo que você passe os dados de entrada. Toda análise dimensional das operações que serão efetuadas ficam explicitadas pelas operações entre tensores.

A princípio não faz sentido converter diretamente entre um Tensor e um NDArray ou number[]. O tensor possui, essencialmente, informações de formato, NDArray e number[] são efetivamente os objetos das operações matemáticas.

O objeto Graph deve ser usado apenas durante o treinamento (ou diferenciação automática). Para usar a biblioteca apenas no modo de inferência, ou para computação numérica em geral, é suficiente utilizar NDArrays com NDArrayMath.

O objeto Graph é essencial durante o treinamento, ele descreve as operações sobre tensores e, quando o treinamento é avaliado, com Session.eval, o resultado será um NDArray.

7.8. Inferência em modo forward e computação numérica

O exemplo a seguir mostra como executar operações matemáticas com a biblioteca. É suficiente construir NDArrays com os seus dados e, a seguir, executar as operações com um objeto NDArrayMath.

Vejamos um exemplo de multiplicação de uma matriz por um vetor na GPU:

```
const math = new NDArrayMathGPU();

const matrixShape = [2, 3];  // 2 rows, 3 columns.
const matrix = Array2D.new(matrixShape, [10, 20, 30, 40, 50, 60]);
const vector = Array1D.new([0, 1, 2]);
const result = math.matrixTimesVector(matrix, vector);
```

```
console.log("shape do resultado:", result.shape);
console.log("resultado", await result.data());
```

A camada NDArray/NDArrayMath pode ser interpretada como similar à biblioteca NumPy do Python.

7.9. Treinamento: execução "atrasada", grafos e sessões

A questão mais importante para entender em relação ao treinamento (diferenciação automática) no deeplearn.js é que a biblioteca utiliza um modelo de execução "atrasada".

O código deverá conter dois estágios distintos:

- Em um primeiro momento o Graph, o objeto que representa os cálculos que serão executados, é definido;
- Em um segundo momento o objeto Graph é executado e os resultados obtidos.

Na maioria dos casos o Graph transformará algumas entradas em algumas saídas e a arquitetura do Graph se manterá fixa, mas conterá parâmetros que serão atualizados automaticamente.

Há dois modos para execução de um Graph: treinamento e inferência.

Inferência é o ato de prover um Graph com uma entrada e gerar uma saída, em função de um treinamento executado previamente.

Treinar um grafo implica em prover o objeto Graph com vários exemplos de pares classificados de entrada/saída e automaticamente atualizar parâmetros do Graph de tal forma que a saída do Graph, quando este estiver avaliando uma entrada, seja o mais próxima possível, dentro de parâmetros definidos, do valor classificado. Durante o processo de treinamento busca-se minimizar o resultado de uma função de utilidade sobre o grafo. A função que gera um escalar (Scalar na biblioteca) representando o quão próximo a saída classificada está da saída computada é chamada de função de custo (também chamada de função de perda). A função de custo deverá ser o mais próximo de zero quanto desejável, quando o modelo está com boa performance. A função de custo deve ser fornecida durante o treinamento.

A biblioteca do deeplearn.js é estruturada de forma bastante similar ao TensorFlow, a biblioteca da Google de machine learning em linguagem Python.

7.10. Grafos como Funções

A diferença pode ser entendida por analogia com código regular em JavaScript. Nos próximos exemplos utilizaremos a função quadrática a seguir:

```
// y = a * x^2 + b * x + c
const x = 4;
const a = Math.random();
const b = Math.random();
const c = Math.random();

const order2 = a * Math.pow(x, 2);
const order1 = b * x;
const y = order2 + order1 + c;
```

No exemplo acima, os cálculos matemáticos são avaliados imediatamente em cada linha na sequência em que são processados.

Compare com o código a seguir, que é análogo à forma que o grafo de inferência do deeplearn.js funciona.

```
function graph(x, a, b, c) {
  const order2 = a * Math.pow(x, 2);
  const order1 = b * x;
  return order2 + order1 + c;
}

const a = Math.random();
const b = Math.random();
const c = Math.random();
const y = graph(4, a, b, c);
```

Este código possui dois blocos: no primeiro ajustamos os parâmetros da função graph e depois a chamamos. O código para ajustar a função graph não executa nenhuma operação matemática até o momento em que a função é chamada na última linha do código acima. Durante esta fase de configuração, erros básicos de compatibilidade de tipo e dimensão podem ser descobertos antes mesmo das computações serem executadas.

Este exemplo é análogo à forma como Graphs funcionam no deeplearn.js. A primeira parte do código configura o grafo, descrevendo:

- Entradas, nesse caso “x”. Entradas são representadas como “placeholders” (e.g. gaph.placeholder());
- Saídas, nesse caso “order1”, “order2”, e a saída final “y”;
- Operações que produzirão os resultados, nesse caso as operações decompostas da função quadrática (x^2, multiplicação, adição);
- Parâmetros atualizáveis, nesse caso “a”, “b” e “c”. Parâmetros atualizáveis são representados como variáveis (e.g. graph.variable()).

Mais tarde no código a função do grafo será chamada (Session.eval) para certas entradas, e então os valores para “a”, “b” e “c” serão conhecidos para alguns dados com Session.train.

Uma pequena diferença entre a analogia de funções acima e o Graph do deeplearn.js é que o Graph não especifica a sua saída. Ao contrário, quem chama a função Graph especifica quais tensores devem ser retornados. Isto permite que várias chamadas ao mesmo Graph executem partes diferentes dele. Apenas as partes necessárias para obter os resultados demandados pela chamada serão avaliadas.

A inferência e o treinamento em um Graph são coordenados por um objeto Session. Este objeto contém o estado em tempo de execução, pesos, ativações e gradientes (derivadas), enquanto o objeto Graph apenas mantém a informação de conectividade.

Então a função acima seria implementada em deeplearn.js da seguinte forma:

```
const graph = new Graph();
// Define uma nova entrada no grafo, chamada de x, com
```

```
// shape [] (escalar)
const x: Tensor = graph.placeholder('x', []);
// Define novas variáveis no grafo, 'a', 'b', 'c' com shape []
// e valores iniciais randômicos
const a: Tensor = graph.variable('a', Scalar.new(Math.random()));
const b: Tensor = graph.variable('b', Scalar.new(Math.random()));
const c: Tensor = graph.variable('c', Scalar.new(Math.random()));
// Define novos tensores, que representam a saída das operações
// da função quadrática
const order2: Tensor = graph.multiply(a, graph.square(x));
const order1: Tensor = graph.multiply(b, x);
const y: Tensor = graph.add(graph.add(order2, order1), c);

// Ao treinar, necessitamos prover um label e função de custo
const yLabel: Tensor = graph.placeholder('y label', []);
// Provê uma função de custo médio quadrático para o treinamento
// custo = (y - yLabel)^2
const cost: Tensor = graph.meanSquaredCost(y, yLabel);
// Nesse ponto o grafo está configurado, mas ainda não foi avaliado
// O deeplearn.js necessita de um objeto Session, para avaliar o
// grafo
const math = new NDArrayMathGPU();
const session = new Session(graph, math);

// Para mais informações sobre scope/track, veja a seção
// de performance deste tutorial
await math.scope(async (keep, track) => {
  /*
   * Inferência
   */
  // Agora instruímos o grafo a avaliar (inferir) e gerar um resultado
  // quando provemos o valor 4 para “x”
  // Nota: “a”, “b” e “c” são inicializados randomicamente então o
  // resultado da função também será randômico
  let result: NDArray =
      session.eval(y, [{tensor: x, data: track(Scalar.new(4))}]);
  console.log(result.shape);
  console.log('result', await result.data());
  /*
   * Treinamento
   */
  // Agora aprendemos os coeficientes desta função quadrática quando
  // fornecidos alguns dados. Para tanto é necessário fornecer
  // exemplos de “x” e “y”
  // Os valores dados são para a = 3, b = 2 e c = 1 com ruído
  // randômico acrescentado à saída, portanto não há um fit perfeito
  const xs: Scalar[] = [
    track(Scalar.new(0)),
    track(Scalar.new(1)),
    track(Scalar.new(2)),
    track(Scalar.new(3))
  ];
  const ys: Scalar[] = [
    track(Scalar.new(1.1)),
    track(Scalar.new(5.9)),
    track(Scalar.new(16.8)),
    track(Scalar.new(33.9))
  ];
  // É importante mesclar os dados no treinamento dos dados
  const shuffledInputProviderBuilder =
      new InCPUMemoryShuffledInputProviderBuilder([xs, ys]);
  const [xProvider, yProvider] =
      shuffledInputProviderBuilder.getInputProviders();

  // O treinamento é particionado em batches
  const NUM_BATCHES = 20;
```

```
  const BATCH_SIZE = xs.length;
  // Antes de iniciar o treinamento é ncessário definir um
  // otimizador, que é o objeto responsável pela atualização dos
  // pesos. O parâmetro de aprendizado é o valor que representa
  // o tamanho do passo no gradient descent durante a atualização
  // de pesos. Se for muito grande, pode "sobrepassar" e oscilar,
  // se for muito pequeno, o modelo pode demorar muito para treinar
  const LEARNING_RATE = .01;
  const optimizer = new SGDOptimizer(LEARNING_RATE);
  for (let i = 0; i < NUM_BATCHES; i++) {
    // O treinamento recebe um tensor de custo para minimizar, essa
    // chamada treina um batch e retorna o custo médio do batch como
    // um escalar (Scalar)
    const costValue = session.train(
        cost,
        // Mapeia providers de entrada aos tensores no grafo
        [{tensor: x, data: xProvider}, {tensor: yLabel, data:
yProvider}],
        BATCH_SIZE, optimizer, CostReduction.MEAN);

    console.log('average cost: ' + await costValue.data());
  }
  // Impressão do valor para o modelo treinado, para x = 4 deveria
  // ser 57,0
  result = session.eval(y, [{tensor: x, data: track(Scalar.new(4))}]);
  console.log('result should be ~57.0:');
  console.log(result.shape);
  console.log(await result.data());
});
```

Depois de treinar o modelo é possível fazer inferências com o grafo para obter valores de "y" para um dado "x".

O exemplo acima é muito mais simples que a maioria dos usos do deeplearn.js, mas é suficiente para dar uma ideia esquemática do seu funcionamento. A biblioteca possui funções de álgebra linear aceleradas por hardware que podem ser utilizadas em aplicações como reconhecimento de imagem, geração de texto, entre outras.