Exercício-Programa 4: Introdução à redes neurais

Entrega: 30/06/2019

Motivação

O objetivo deste EP é implementar uma rede neural simples capaz de identificar dígitos escritos à mão. Para alcançar este objetivo, o programa será dividido em três partes para se obter um melhor entendimento de cada operação realizada no processo de implementar uma rede neural.

Instruções gerais

Para esse EP você deve ter recebido um único arquivo compactado denominado `EP4MAC1425.zip` contendo os seguintes arquivos:

```
EP4MAC1425.zip
├── ep4.py
├── nn.py
├── backend.py
└── autograder.py
```

Além dos arquivos a cima também é necessário que seja feito o download dos arquivos no link: https://drive.google.com/file/d/1p-a-aCquVFfUag6DryrIoetF2shY8A_W/view?usp=sharing, eles contém o dataset para poder realizar os testes. Faça o download deles e os compacte em um `.npz` chamado `mnist.npz`, se não existir ainda, crie um diretório chamado `data/` e coloque o `mnist.npz` dentro dele, ao final você deve ter o seguinte:

```
EP4MAC1425/
├── ep4.py
├── nn.py
├── backend.py
├── autograder.py
└── data/
    └── mnist.npz
```

Também é necessário a instalação de 2 bibliotecas externas, **numpy** e **pyplot**, ambas são muito úteis em aplicações python e tem potencial de serem usadas em matérias além de Inteligência Artificial. A primeira delas, numpy, é responsável por realizar operações em N dimensões e realizar diversas funções matemáticas. Pyplot, por sua vez, é uma biblioteca gráfica que permite construir e exibir gráficos em python. É possível testar se todas as dependências estão instaladas com o comando:

```
python autograder.py --check-dependencies
```

Você deverá implementar sua solução no esqueleto de código fornecido no arquivo `ep4.py`. Não há a necessidade de se implementarem novas funções e você não deve modificar o protótipo das funções existentes. Cada função no esqueleto que você deve modificar está inicialmente implementada com o comando:

```
raise NotImplementedError
```

Parte 1: Criando um perceptron

Funções a serem utilizadas

Dentre os arquivos fornecidos, `nn.py` contém as classes que você terá que utilizar em suas funções. Elas e suas operações que podem ser utilizadas nesta parte serão brevemente explicadas a seguir:

- **nn.Parameter**: Esta classe cria um parâmetro treinável para a rede neural, isto é, uma matriz 2D cujos valores podem ser modificados com treinamento. Ela recebe para sua inicialização as dimensões (X, Y) da matriz, no caso do perceptron, como existe apenas um vetor de peso ela terá tamanho $(1, Y)$
- **nn.DotProduct**: Realiza o produto escalar entre dois *nn.Parameter*.
- **nn.as_scalar**: Diversas operações podem devolver como saída uma matriz 1×1, caso você queira recuperar este valor como um escalar, utilize esta função.

Além destas funções, a classe `dataset` implementada em `backend.py` contém o método **iterate_once** que deve ser usado para iterar sobre um dataset. Ele recebe como argumento o tamanho do batch b e devolve este número de instâncias x e y sendo x a entrada para a rede e y o que é esperado da saída deste valor.

Não há a necessidade de implementar ou utilizar qualquer outra função além das explicadas acima.

Implementação

Você deve implementar um perceptron completando a classe **PerceptronModel** em `ep4.py`.

Como visto em aula, um perceptron recebe como entrada um vetor x de N dimensões, realiza um produto escalar entre este vetor e seus pesos w, e devolve um valor que pode ser $+1$ ou -1, com isso é possível tentar realizar uma decisão binária com o objetivo de separar linearmente um conjunto de dados, possíveis aplicações para perceptrons incluem: identificar se uma imagem de um dígito representa o número 1 ou o número 0, ou identificar se um e-mail é um spam a partir de características de seus textos. A implementação que vocês farão simplesmente irá dividir um conjunto de dados genéricos que pode ter qualquer número de *features*.

O método **get_weights** deve recuperar este vetor de pesos, **run** deve devolver o produto escalar e **get_prediction** irá analisar este valor e retornar $+1$ ou -1.

Por fim, o método **train** deve treinar o perceptron utilizando o dataset recebido **até ele conseguir classificar perfeitamente todos os pontos**. Para isso ele deve iterar sobre os dados atualizando o seus pesos de acordo com a seguinte função:

$$self.w.data \leftarrow self.w.data + y^* * prediction$$

Você pode visualizar e avaliar a sua implementação executando o comando:

```
python autograder.py -q q1
```

Parte 2: Aproximando a função seno

Funções a serem utilizadas

Além das funções utilizadas na parte anterior, você também poderá utilizar as seguintes funções:

- **nn.ReLU**: Aplica em cada elemento x_i de um vetor x a função *Rectified Linear Unit*: $relu(x_i) = \max(x_i, 0)$
- **nn.AddBias**: Soma um vetor de pesos w a um vetor de vieses b.
- **nn.Linear**: Aplica uma multiplicação de matrizes entre dois `nn.Parameter`.
- **nn.SquareLoss**: Realiza a função de perda quadrática entre dois vetores, o de valores obtidos pela rede e o de valores esperados.
- **nn.gradients**: Recebe como argumento o resultado da função **get_loss** (que você irá implementar) e um vetor com parâmetros que devem ser modificados, então ela devolve um vetor de gradientes onde cada elemento indica em que direção o parâmetro correspondente deve ser alterados.
- **nn.Parameter.update**: Recebe como entrada um parâmetro o gradiente e o *learning rate* e aplica a operação: $parameter \leftarrow parameter - grad * lr$ no parâmetro.

Implementação

Você deve implementar uma rede neural simples que aproxime a função seno, ou seja, recebe escalares como entrada e após passa-los na rede, devolve um resultado que deve se assemelhar à aplicar a função seno a esse escalar. É importante notar que diferente de um perceptron, um neuron da rede neural **precisa somar um viés** ao vetor após aplicar os pesos w ao vetor de entrada.

Para isso o método **___init___** deve inicializar os pesos e *bias* da rede. Vocês poderão decidir como será a arquitetura do seu modelo.

Ele deve receber como entrada um dataset composto de um vetor x composto de *batchs* de escalares que devem ser aplicados a função seno, e um vetor y com os valores esperados do resultado dessa função.

O método **run** deve receber como entrada o x e passa-lo na rede, não se esqueça que a operação para uma rede de duas camadas seria:

$$f(x) = relu(x * W_1 + b_1) * W_2 + b_2$$

Sendo W_1 e W_2 os pesos da rede e b_1 e b_2 seus respectivos *bias*. Para uma rede de três camadas a operação seria:

$$f(x) = relu(relu(x * W_1 + b_1) * W_2 + b_2) * W_3 + b_3$$

Note que qualquer que seja o número de camadas, a função de ativação é sempre aplicada em todas elas exceto a última.

O método **get_loss** deve devolver a função de perda quadrática entre $f(x)$, x depois de passar pela rede e y os valores esperados.

Por fim, a função **train** deve treinar a rede operando sobre o dataset **até que a média do valor da função de perda seja menor do que** 2%.

Realize os testes executando:

```
python autograder.py -q q2
```

Parte 3: Classificação de dígitos

Funções a serem utilizadas

Você usará praticamente as funções das partes anteriores com a adição de:

- **dataset.get_validation_accuracy**: Recupera a acurácia do seu modelo usando o dataset de validação.
- **nn.SoftmaxLoss**: Realiza a função de perda *Soft Max* entre dois vetores, o de valores obtidos pela rede e o de valores esperados.

Além destas funções, o autograder fara o download **automaticamente** do dataset que deverá ser utilizado.

Implementação

Nesta parte, vocês irão resolver um problema clássico de implementar um classificador de dígitos escritos à mão. A database usada neste exercício é chamada de MNIST e consiste de diversas imagens de 28×28 pixels remodelada como um vetor de 784 valores com sua respectiva classificação.

Você deve completar a classe **DigitClassificationModel** exatamente da mesma forma do exercício anterior. A maior diferença será na arquitetura do modelo que deve receber como entrada um vetor de tamanho 784 e devolver um vetor de tamanho 10 com a avaliação de cada dígito (0-9). Para a função perda você deve utilizar a função *nn.SoftmaxLoss*.

Diferente do exercício anterior, o seu modelo será avaliado em um dataset diferente do de treinamento. Este outro conjunto usado para avaliação é chamado de dataset de validação e a acurácia do seu modelo nele pode ser recuperada por meio da função *dataset.get_validation_accuracy*. É necessário que seu modelo tenha ao menos 96% de acurácia.

Da mesma forma que nas partes anteriores, você pode testar a sua função executando:

```
python autograder.py -q q3
```

Instruções para entrega

Você deve submeter via Paca apenas o arquivo *ep4.py* contendo a sua solução até às 23:55 do dia **30/06/2019**. Para evitar que seu EP seja zerado, certifique-se que o arquivo foi submetido sem problemas, baixando e executando o arquivo do site. O arquivo deve conter um programa escrito para a versão 3.X do Python

Avaliação

Embora seja muito importante que seu código passe em todos os testes, isso não é garantia de que seu código receberá nota máxima. É possível que os testes não verifiquem alguns casos peculiares onde seu programa pode vir a falhar.