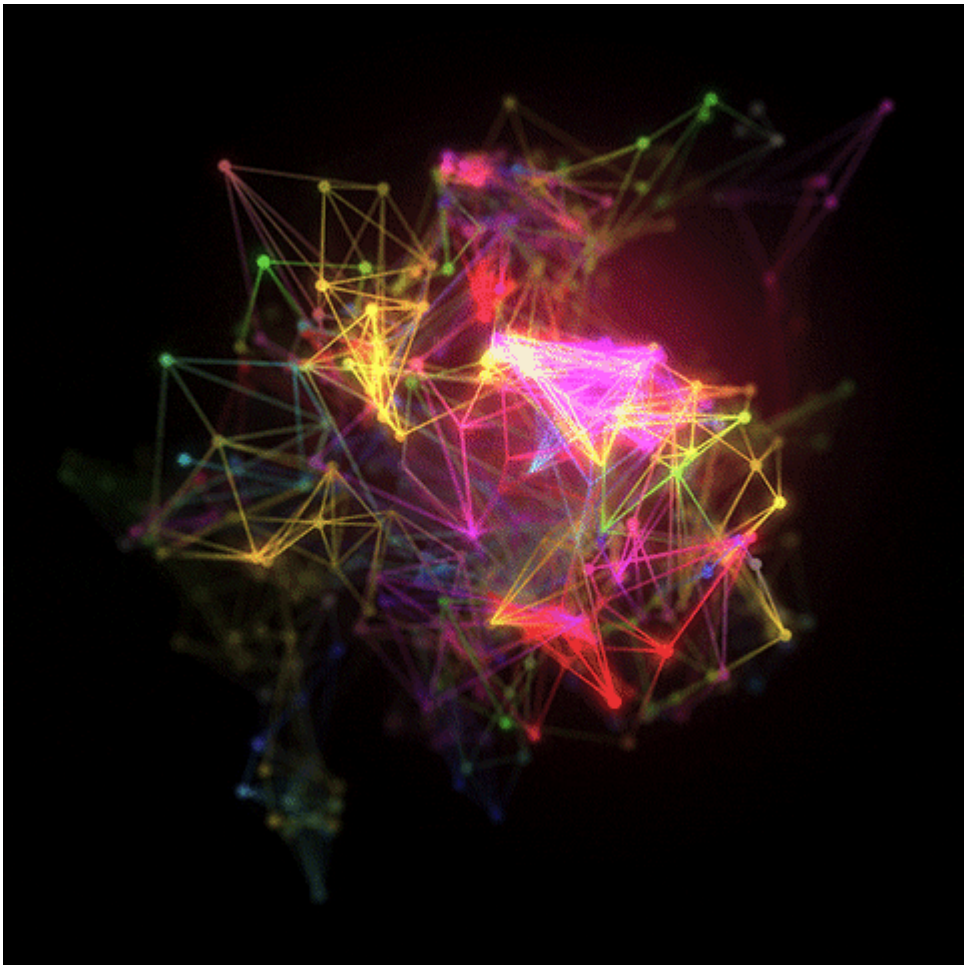
medium.com

Construindo uma Rede Neural do zero | Pytorch - Turing Talks - Medium

Enzo Cardeal Neves

11–15 minutes

“Neurons that fire together, wire together”



Bem-vindos à mais uma edição do ***Turing Talks***! Nessa semana

abordaremos **como desenvolver um modelo de *Deep Learning* usando *Pytorch***.

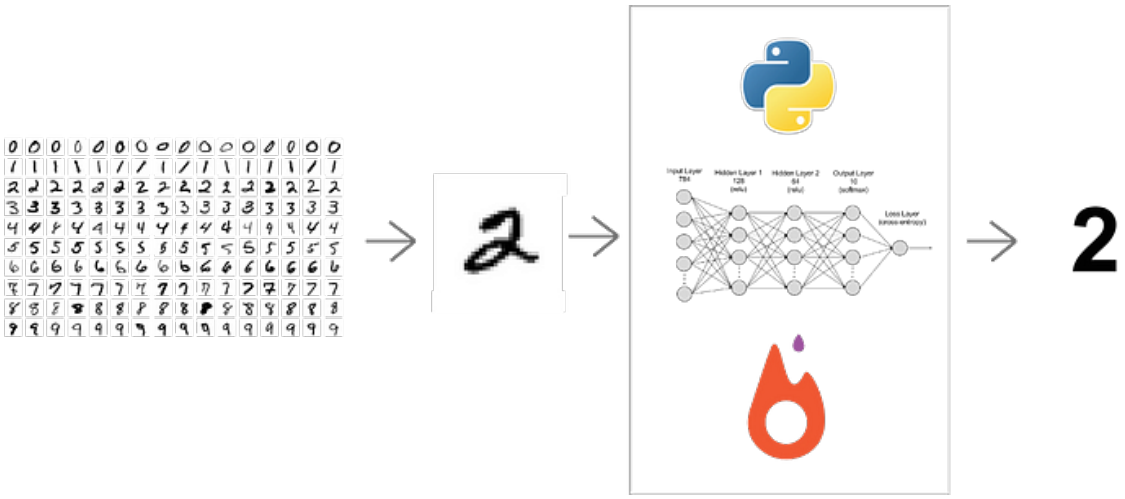
Para entender bem o post é esperado que o leitor tenha uma boa noção dos seguintes tópicos:

- Aspectos de programação orientada a objetos em Python
- familiaridade com Numpy
- Compreensão básica de como redes neurais funcionam

Em relação a redes neurais, as seguintes fontes são suficiente para conseguir acompanhar o post:

Ou para quem entende inglês, pode assistir a playlist a seguir que é bem didática:

Com essas habilidades, vamos fazer um passo a passo do "*Hello World*" das redes neurais (RNs): a classificação de números escritos a mão.



O que é Pytorch?

Uma biblioteca open source para criar redes neurais que permite tanto o uso da CPU quanto da GPU para o treinamento do modelo. Por ser mais flexível que o TensorFlow e muito bem documentada, é ótima para se fazer pesquisas (na verdade, é o *framework* mais usado no meio acadêmico para construir RNs) e para montar a sua primeira rede neural :)

Algumas das suas principais características são:

- Grafos computacionais dinâmicos(será explicado melhor posteriormente no post)
- Ecossistema robusto, se integrando muito bem com outros *frameworks* voltados para a construção de modelos *deep learning*
- Treinamento distribuído, sendo possível alocar várias GPUs para treinar o modelo
- Estrutura “pythônica”, ou seja, a sintaxe é bem parecida com a do python

Fazendo e aprendendo

Para cada etapa da construção, iremos mostrar a aplicação em código e explicar o que está sendo feito, dessa forma ficará mais fácil de acompanhar o que está acontecendo.

Antes de mais nada, você deve instalar a biblioteca, recomendamos que use o Anaconda para isso. Como ambiente de trabalho, utilizaremos o Jupyter Notebook.

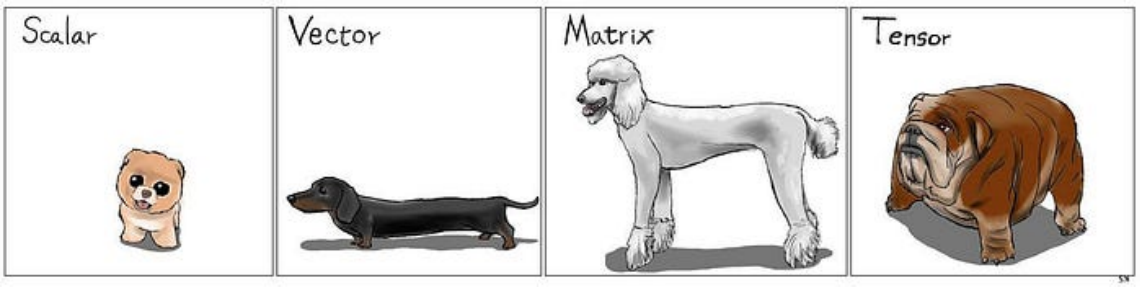
Etapa 1 - Carregando as bibliotecas e o dataset

Primeiramente importamos todas as bibliotecas necessárias.

Por ser um dataset tão famoso, o MNIST já vem incluso na biblioteca, então podemos baixá-lo diretamente a partir do `import datasets`.

Tensor

Um tensor é como se fosse uma matriz, a diferença é que suas dimensões não estão limitadas a 2. Ele pode tanto ter dimensão 0 (**escalar**) quanto dimensão n.



Uma boa analogia rs

Precisamos converter os dados para tensor, pois é assim que a biblioteca trabalha com as informações.

- a partir da `transforms.ToTensor()` definimos a variável `transform` como a conversão dos dados do dataset para tensor.
- carregamos o set de treino na variável `trainset`. o primeiro argumento é o local onde será salvado o dataset, `train=true` determina que você quer carregar a parte reservada para treino (85,7% dos dados) e no último argumento, passamos a transformação definida anteriormente.
- por ser um dataset muito grande(70.000 imagens), devemos criar um buffer para otimizar a tarefa e acessá-lo em subgrupos menores. Para isso, definimos a variável `trainloader` que é um objeto iterável que nos fornece lotes(batchs) de itens até que o set seja todo percorrido. Passamos como argumento o `trainset`, `batch_size=64` para que os subgrupos sejam de 64 itens e `shuffle=true` que determina que iremos tomar os itens de forma aleatória, ou seja, um

mesmo item pode estar presente em mais de um subgrupo.

- repetimos o mesmo processo para o set de validação

Etapa 2 - Conferindo a estrutura dos dados

Vamos abrir um dos itens do dataset para termos uma ideia de seu formato. Cada item possui uma imagem e uma etiqueta, este ultimo é o digito que a imagem representa.

Analisando a dimensão do tensor imagem e do tensor etiqueta(um número de 0 a 9):

O tensor etiqueta não possui dimensão nenhuma por ser um escalar.

Já o tensor imagem possui 3 dimensões. Por quê?

A primeira dimensão é o número de canais da nossa imagem. Como é uma imagem preto e branco, possui apenas um, que diz respeito a intensidade do preto em cada pixel. Imagens coloridas costumam ter 3 canais (vermelho, azul e verde).

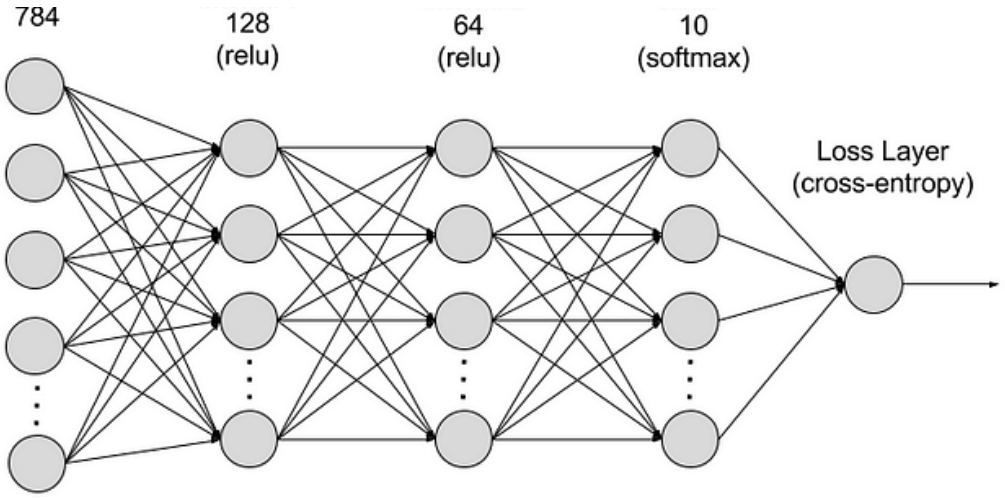
As duas ultimas dimensões representam a quantidade de pixels, nesse caso 28 x 28 = 784 pixels.

Sendo assim, para o Pytorch, cada imagem é uma “matriz” (na verdade é um tensor, nunca se esqueça) 28 x 28 com valores para cada elemento variando de 0 a 1 de forma continua.

Etapa 3 - Construindo os elementos da Rede Neural

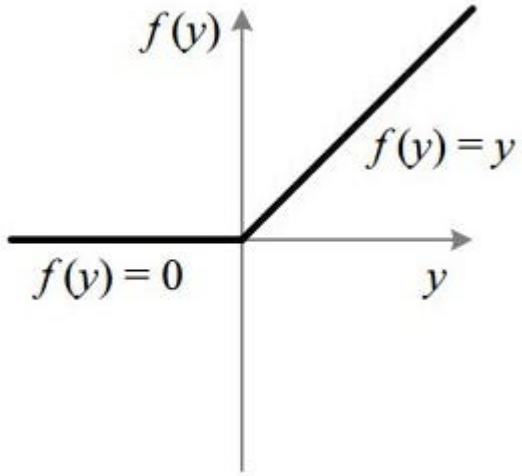
Uma boa prática é construir uma classe para conter a estrutura da nossa RN. Essa modularizarão facilita adaptações futuras no nosso modelo. Por sorte, o Pytorch possui uma superclasse que auxilia nessa tarefa, a `nn.Module`.

A rede que queremos montar é a seguinte:

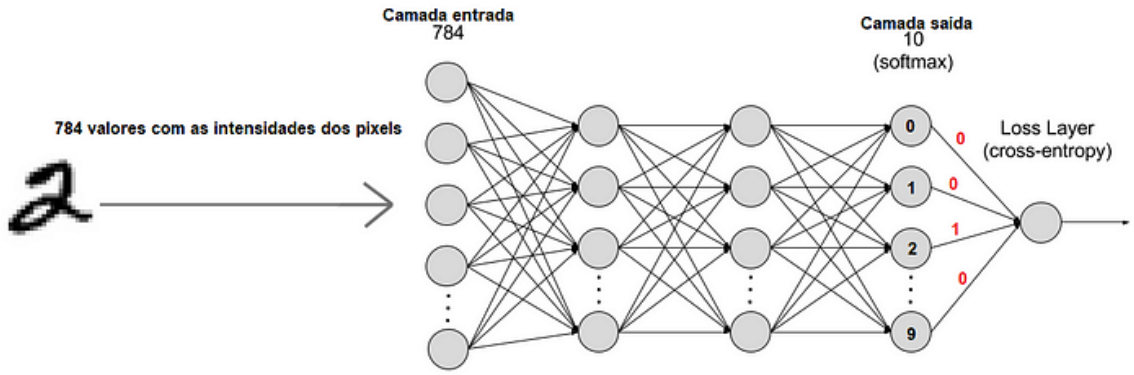


Na camada entrada temos 28 x 28 = 784 neurônios, um para cada pixel da imagem. Esses neurônios irão receber o valor entre 0 e 1 de acordo com a intensidade do preto no pixel correspondente. Nas camadas internas 1 e 2, temos 128 e 64 neurônios, respectivamente. Essa escolha foi arbitrária e, ao final do post, veremos que foi boa o suficiente para conseguirmos um resultado satisfatório.

Utilizaremos a **ReLU** como função de ativação das camadas: entrada → interna 1 e interna 1 → interna 2. Essa é uma função do seguinte formato: f(y) = y, se y>0, caso contrário, f(y) = 0.



Por ser um problema de classificação, no tensor devolvido pela camada saída(10 valores entre 0 e 1, cada um correspondendo a chance, segundo o modelo, de a imagem fornecida ser o número do índice correspondente, como exemplificado na imagem abaixo), aplicaremos a função **LogSoftmax**, que devolve um tensor com as mesmas dimensões da saída.



Representação de um modelo bem treinado

A partir deste tensor e do target(nesse caso é a etiquete pois ela define qual o digito que a imagem sendo analisada representa), aplicamos a função **negative log-likelihood loss(NLLLoss)** e calculamos a **perda**. A lógica é a mesma do erro quadrático médio(MSE), quanto mais próximo de zero, melhor.

Definimos então a classe `Modelo`:

A função `forward(self, X)` nos retorna o tensor que será utilizado para calcular a perda. **Isso é o que o modelo retorna**.

Repare aqui que todas as funções que utilizamos são do `import torch.nn.functional as F`, funções especiais para trabalhar com tensores, e o tipo das camadas foram definidos a partir do `import nn`. Utilizamos camadas lineares por simplicidade. Para classificação de imagens, o ideal seria também utilizarmos camadas de convolução 2D.

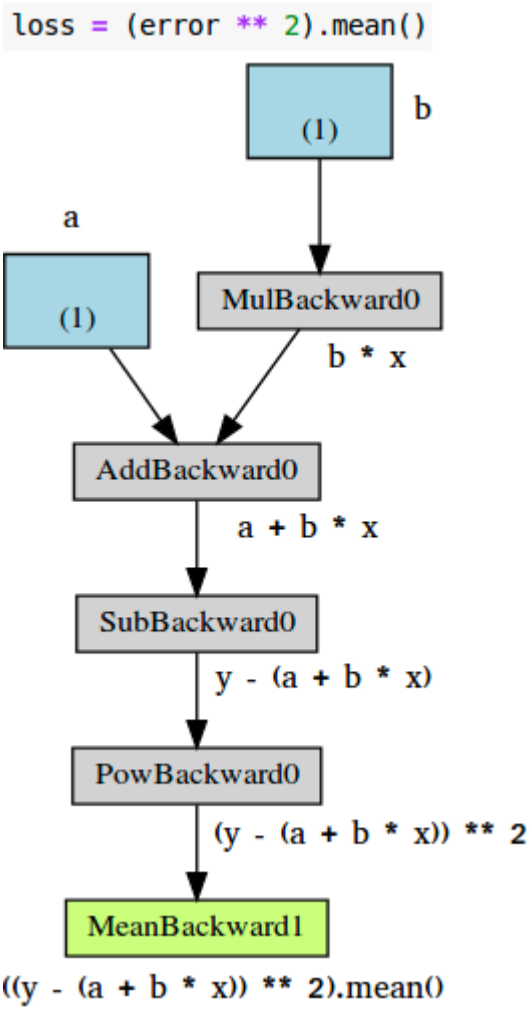
Vamos agora definir a estrutura de treino do modelo. As principais etapas são:

- Calcular a perda a partir da comparação entre as predições e as etiquetas(target) do subgrupo sendo analisado
- Com a perda, calcular o gradiente em relação aos pesos e as bias
- a partir do gradiente e de uma politica de otimização, atualizar os pesos e as bias

Graças ao Pytorch, para defini-las será um passeio no parque.

Autograd

A partir dos grafos computacionais dinâmicos, por padrão o Pytorch armazena uma espécie de histórico das operações realizadas. No exemplo a seguir isso fica mais claro.



https://towardsdatascience.com/understanding-pytorch-with-an-example-a-step-by-step-tutorial-81fc5f8c4e8e#3806

É possível calcular as derivadas parciais da **loss = ((y-(a+b*x))**2).mean()** em relação a **a** e **b**(caixas azuis) já que foi armazenado a "história" dessas variáveis.

Assim, calculamos o gradiente facilmente, pela regra da cadeia, da perda em relação aos pesos e bias. CALMA! a biblioteca fará todo esse processo para você.

Otimizador

A primeira vista você pode pensar em atualizar os pesos e as bias a partir de um loop, calculando cada elemento da seguinte forma: **a = a - lr*grad** (onde **lr** é a taxa de aprendizado e **grad** é a derivada parcial da **perda** em relação a **a**). Isso não é necessário. A partir do `import optim`, podemos escolher qual otimizador usar para fazer essas atualizações. Nesse caso vamos usar o SGD (*Stochastic Gradient Descent*). Seus parâmetros principais são:

- `params:` iterável de parâmetros para serem otimizados(no nosso caso, pesos e bias)
- `lr:` taxa de aprendizado
- `momentum:` valor entre 0 e 1, serve para evitar que o modelo encontre um falso mínimo global. Para uma boa RN, quanto maior a `lr`, menor deve ser o `momentum`.

Definimos então a função `treino:`

Explicando sucintamente o que está sendo feito:

1. inicializamos o otimizador
2. a partir de `modelo.train()` colocamos o modelo no modo treino
3. definimos quantas epochs queremos para treinar o modelo
4. em cada epoch, iteramos pelo `trainloader`(lembrando que este é um objeto iterável) até que todo o set de treino seja percorrido
5. em cada iteração do `trainloader`calculamos a perda instantânea, que representa a perda calculada apenas em relação aos 64 itens atuais. Para calcular a perda da epoch, deve-se tomar a média das perdas instantâneas.
6. a partir de `perda_instantanea.backward()` calculamos o

gradiente da perda instantânea

7. com as derivadas parciais e chamando `otimizador.step()`, atualizamos os pesos e bias
8. Os passos 5, 6 e 7 são repetidos até que todo o `trainset` seja percorrido
9. Volta ao passo 4 até chegar a ultima epoch

Vamos agora definir a função `validacao`**.** Como esse não é o foco principal do post, deixaremos as explicações apenas nos comentários do código.

Etapa 4 - Rodando a rede neural

Primeiro vamos inicializar o modelo

Colocamos para rodar na GPU(se possível) para demonstrar essa funcionalidade.

Treinando o modelo

Validando o Modelo

5 - Salvando o modelo

Claro que depois de todo esse trabalho, queremos guardar a evolução da RN.

Bônus

Para visualizar a uma predição aleatória do set de validação, você pode executar a célula a seguir no seu notebook:

Considerações finais

Com esse post, construímos uma base boa para conseguirmos

treinar Redes Neurais mais complexas utilizando o Pytorch sem grandes dificuldades. Apresentamos as principais características da biblioteca mas sem entrar muito nos detalhes já que nosso foco foi montar a rede em si.

Existem diversos pacotes para facilitar o trabalho com os diferentes tipos de dados, sendo eles:

- **Pillow** e **OpenCV** para imagens
- **scipy** e **libroso** para áudio
- **Cython**, **NLTK** ou **SpaCy** para texto

Independente do tipo dos dados, depois de carregados e convertidos para tensores, o processo para montar a RN é bem similar (usando as camadas, otimizadores e função perda mais adequados) com o que fizemos aqui.

Outra ferramenta interessante e bem integrada com Pytorch é o **TensorBoard**, que permite visualizar, a partir de gráficos, como o modelo está evoluindo. Caso tenha interesse em conhecê-la, suas funcionalidades foram demonstradas em um Turing Talks passado.

Por fim, esperamos ter tirado um pouco desse misticismo que existe em volta da "buzzword" *Machine Learning* e mostrar que, com as ferramentas que temos a nosso dispor, qualquer um pode montar uma rede neural do zero.

Numa abordagem quase que de tentativa e erro para determinar a estrutura do nosso modelo, conseguimos um preditor com uma taxa de acerto de 97% (o estado da arte para esse problema é 99,79%). Claro que apenas reproduzimos técnicas existente há décadas e, se você quiser se aventurar de verdade por esse mundo, montando modelos com alto grau de complexidade, há uma longa jornada de aprendizado a ser percorrida.