

Seção 2

O curso se inicia apresentando o conceito de deep learning, conteúdo abordado no módulo 13-15 do bootcamp, porém logo é apresentado o pytorch, uma biblioteca em python destinada a machine learning, sendo uma ferramenta fácil de se utilizar, portanto acelerando o processo de criação e treinamento de modelos, além de possuir uma grande comunidade que a utiliza.

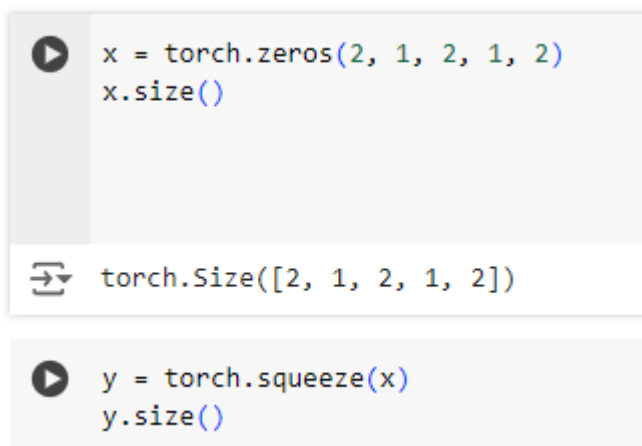
Tensores

Um tensor é uma estrutura de dados feita para criar redes neurais, sendo uma matriz de 1, 2 ou 3 dimensões criadas pelo pytorch através da função tensor, utilizados para backpropagation a fim de estimar o gradiente no treinamento, e para representar as predições de saída do modelo.

Também é possível realizar operações em um tensor, sendo possível realizar as 4 operações básicas em todos os números de um tensor com apenas um comando.

Para modificar a visualização de um tensor, utiliza-se `torch.view(x,y)`, expondo o tensor em uma matriz com x colunas e y fileiras.

Para remover dimensões de um tensor onde o tamanho seja 1, é utilizado o `torch.squeeze(t, x)` onde se removerá as dimensões especificadas em x de um tensor t, contando que estas dimensões especificadas possuam tamanho 1, `unsqueeze(t,x)` realiza o oposto, adicionando uma dimensão a um tensor t, onde x é obrigatório desta vez, pois informará o índice onde esta dimensão estará localizada.



```
x = torch.zeros(2, 1, 2, 1, 2)
x.size()

torch.Size([2, 1, 2, 1, 2])

y = torch.squeeze(x)
y.size()
```

The screenshot shows a Jupyter Notebook interface with three code cells. The first cell defines a tensor `x` with dimensions (2, 1, 2, 1, 2) using `torch.zeros` and prints its size. The second cell prints the output of `torch.size()`, which is `torch.Size([2, 1, 2, 1, 2])`. The third cell defines a tensor `y` by squeezing `x` using `torch.squeeze` and prints its size.

Outras funções do pytorch incluem multiplicação de matrizes através de `torch.matmul`, concatenação e permutação

Vantagens de se utilizar tensores

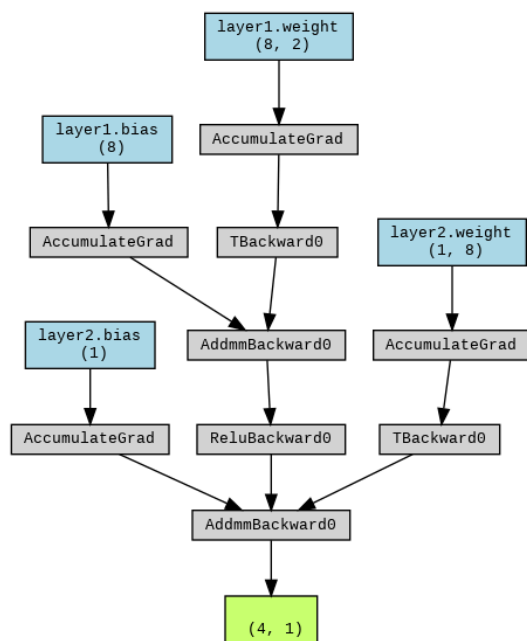
Os tensores são mais adequados para se trabalhar com redes neurais por conta de utilizarem menos processamento de GPUs, acelerando o treinamento de um modelo além dos benefícios para backpropagation supracitados.

Seção 3

Nesta seção o apresentador aborda a arquitetura de uma rede neural e o seu treinamento, conteúdo apresentado no módulo 12, um elemento novo é o torchviz, módulo utilizado para visualizar gráficos de execução do pytorch, a construção da rede neural em pytorch se difere da construção no keras, sendo a do keras bem mais intuitiva.

```
class redeNeural(nn.Module):
    def __init__(self):
        # When we call the super.__init__() method we ensure we are inheriting
        super().__init__()
        self.layer1 = nn.Linear(2,8)
        self.activation = nn.ReLU()
        self.layer2 = nn.Linear(8,1)

    # When we pass something through the model object, it calls the forward method
    def forward(self,x):
        x = self.layer1(x)
        x = self.activation(x)
        x = self.layer2(x)
        return x
```



visualização da rede neural

Após a demonstração base do pytorch são abordados novamente os assuntos já vistos anteriormente como ativação, funções de perda, otimizadores, datasets e redes sequenciais, porém aplicando-as no pytorch, portanto esta seção foi útil para aprender outra ferramenta além do keras.

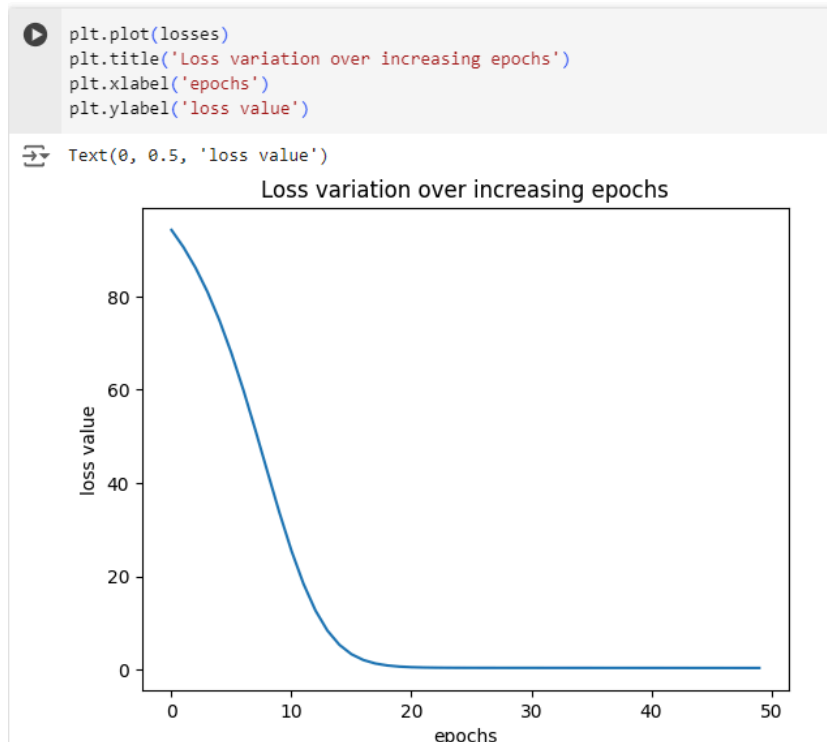


gráfico de perda de um modelo simples utilizando MSE como loss function e SGD como optimizer

Seção 4

Na seção 4 são retratadas as redes neurais destinadas a processamento de imagens, introduzindo a biblioteca opencv, destinada a auxiliar no treinamento de modelos para visão computacional

É apresentado um exemplo de processamento do fashion mnist semelhante ao realizado nos cards anteriores em keras, porém desta vez foi realizado em mnist,

Seção 5

A seção 5 aborda redes neurais convolucionais utilizando pytorch, também apresenta data augmentation, processo para geração de mais imagens para executar o treinamento de modelos

```

▶ from torch.optim import SGD, Adam

def get_model():
    model = nn.Sequential(
        nn.Conv2d(in_channels=1, out_channels=64, kernel_size=3),
        nn.MaxPool2d(kernel_size=2),
        nn.ReLU(),
        nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3),
        nn.MaxPool2d(kernel_size=2),
        nn.ReLU(),
        nn.Flatten(),
        nn.Linear(in_features=3200, out_features=256),
        nn.ReLU(),
        nn.Linear(in_features=256, out_features=10)
    ).to(device)

```

estrutura com duas camadas de convolução com MaxPooling realizado após cada uma.

Seção 6

A seção 6 se trata de auto encoders, redes neurais dedicadas a compressão de dados, composta por encoders e decoders. os encoders recebem uma imagem e a transformam em uma representação dimensional chamada de espaço latente, enquanto um decoder recebe esta representação e transforma ela em uma imagem para a mesma poder ser observada por humanos na saída.

Um exemplo é o autoencoder padrão uma rede neural artificial de três camadas, uma camada de entrada, uma oculta e uma saída. Esta rede tem o objetivo de desconstruir e reconstruir a entrada de forma que a entrada continue semelhante.

Os VAE são autoencoders que utilizam a inferência bayesiana, recebe um tensor como entrada, que é comprimido utilizando um Encoder, o auto encoder gera com base na entrada uma média e um desvio padrão, e as utiliza para reconstruir a imagem original mais fielmente.

```

class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        # Encoder - Recebe um canal e retorna 64 canais após duas convoluções
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 256)
        self.fc21 = nn.Linear(256, 20) # mu
        self.fc22 = nn.Linear(256, 20) # logvar
        # mu e logvar são os dois vetores, média e logaritmo da variancia, utilizados para realizar a distribuição probabilística

        # Decoder - realiza as mesmas operações que o encoder em ordem reversa a fim de restaurar a imagem ao seu estado inicial
        self.fc3 = nn.Linear(20, 256)
        self.fc4 = nn.Linear(256, 64 * 7 * 7)
        self.deconv1 = nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=1, output_padding=1)
        self.deconv2 = nn.ConvTranspose2d(32, 1, kernel_size=3, stride=2, padding=1, output_padding=1)

```

Seção 7

Na seção 7 são apresentados 3 projetos para exemplificar o conteúdo apresentado no curso, sendo um de deep fake(criação de imagens falsas utilizando deep learning), uma de neural style transfer para criar imagens utilizando características de duas imagens diferentes, e uma para colorir imagens em preto e branco de forma autônoma.

```
if (ex+1)%100 == 0:
    bs = 5
    a,b,A,B = data
    line('A to B')
    _a = model(a[:bs], 'A')
    _b = model(a[:bs], 'B')
    x = torch.cat([A[:bs],_a,_b])
    subplots(x, nc=bs, figsize=(bs*2, 5))

    line('B to A')
    _a = model(b[:bs], 'A')
    _b = model(b[:bs], 'B')
    x = torch.cat([B[:bs],_a,_b])
    subplots(x, nc=bs, figsize=(bs*2, 5))

log.plot_epochs()
EPOCH: 96.000  lossB: 0.050  lossA: 0.062 (107.02s - 5507.70s remaining)
EPOCH: 97.000  lossB: 0.054  lossA: 0.061 (108.85s - 5501.87s remaining)
EPOCH: 98.000  lossB: 0.054  lossA: 0.060 (109.88s - 5496.40s remaining)
EPOCH: 99.000  lossB: 0.053  lossA: 0.063 (110.91s - 5490.66s remaining)
EPOCH: 100.000  lossB: 0.053  lossA: 0.061 (111.95s - 5485.68s remaining)
```

A TO B

