UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E COMPUTAÇÃO CIENTÍFICA
DEPARTAMENTO DE MATEMÁTICA APLICADA

Victor Jeronymo

Implementando métodos de otimização para treinamento de redes neurais com Pytorch*

Monografia apresentada ao Instituto de Matemática, Estatística e Computação Científica da Universidade Estadual de Campinas como parte dos requisitos para obtenção de créditos na disciplina Projeto Supervisionado, sob a orientação do(a) Prof. Dr. Paulo J. S. Silva .

Campinas
2019

*

1 Introdução

As redes neurais tem cada vez mais despertado a curiosidade e o interesse não só do ponto de vista acadêmico, mas também de pessoas que buscam aprender por algum dos vários cursos online disponíveis. Não sem motivo. Suas aplicações variam desde o reconhecimento de dígitos escritos a mão à predição de jogadas no basebol, da matemática à medicina.

O aumento do estudo e, consequentemente, uso de tais redes proporcionou o desenvolvimento de algumas ferramentas que visam facilitar o uso dos algoritmos. Dentre essas ferramentas destaca-se o *Pytorch*: uma biblioteca *open source* do *Python* capaz de proporcionar flexibilidade e velocidade ao estudo do deep learning. Uma das formas que isso é feito é através pré-programação de algumas funções comumente usadas como aquelas que calculam erro da resposta da rede e os otimizadores que garantem o aprendizado e atualização dos parâmetros.

Em face disso nasceu a questão que motiva esse trabalho: e se o otimizador que eu quero usar não estiver pré-implementado ? Como adicionar um otimizador próprio para que ele rode aos moldes dos nativos da biblioteca ?

Ressalto que esse texto não se propõe a explicar todo o funcionamento do *Pytorch*. Dessa forma algumas passagens que não se referem aos otimizadores serão pressupostas como conhecidas.

2 *Torch.Optim.Optizer*

O pacote do *Pytorch torch.optim* contém a implementação de alguns dos otimizadores mais usados, delegando ao usuário apenas a definição de alguns parâmetros.

Mas, além disso, há ainda uma classe chamada *Optimizer* que permite justamente a implementação de otimizadores, sendo a base de todos eles. Dessa forma, a resposta das peguntas acima começa com uma extensão dessa classe.

3 Como fazer ?

O processo se inicia com a declaração da sub-classe que irá herdar os métodos e propriedades do Optimizer. [1]

Dentro da classe precisamos definir as seguintes funções: __init__ e step.

3.1 __init__

É o método que desempenha o papel de construtor. Toda vez que um objeto é criado, ele entra em ação e inicializa os atributos da classe.

No caso, precisamos definir dois atributos: os parâmetros do modelo e o *defaults*. O primeiro especifica os tensores - uma generalização dos vetores porém de dimensão maior, mas para o caso que tratarei serão vetores usuais

[1] Lembrando que o python é uma linguagem orientada a objeto, isto é, ela é realizada por meio de objetos que interagem entre si. Cada objeto, por sua vez, é uma instância de uma classe.

- ou dicionário que desejamos otimizar; o segundo é um dicionário que registra parâmetros auxiliares como a *taxa de aprendizagem* (ou *lr*)e outros hiperparâmetros usados no algoritmo de otimização.

Outro ponto importante dessa etapa é chamar o "__init__" da classe que estamos estendendo. A forma de fazer isso é usando a função *super().__init__(parametros, defaults)*.

3.2 step

Essa é a etapa mais importante pois é aqui onde o algoritmo de otimização é, de fato, implementado. Aqui é o local em as contas que estão no algoritmo são feitas.

A primeira parte do dessa função é um laço cujo o propósito é percorrer todos os parâmetros da rede em que é possível o calculo do gradiente ("for group in self.param_groups:"). Dentro desse laço, para cada um dos grupos de parâmetros - que são tensores - vamos realizar as operações pertinentes ao otimizar e alterar o valor dos critérios.

Outro ponto importante de salientar é que nesse loop garantimos só atualizaremos os parâmetros que possuem gradiente.

O gradiente será acessado via o atributo *grad* do grupo p.

Deve-se observar que o gradiente só é calculado após a função objetivo - que nos exemplos foi chamada de *loss* - ter propagado pela rede. Isto é (na linguagem do pytorch) após o loss.backwards(). A importância disso se dá pelo fato do otimizador atuar justamente sobre a função objetivo.

4 Exemplos:

Os exemplos serão de dois otimizadores que já estão presentes no pytorch: o SGD e o ADAM.

4.1 SGD:

SGD é uma sigla para Stochastic Gradient Descent que, em uma tradução livre, significa Método do Gradiente Estocástico.

Como o nome já nos diz trata-se de um algoritmo baseado no método do gradiente porém com aleatorização das componentes da função de custo. A principal diferença entre os dois é que neste o valor do gradiente é calculado para todas os dados do problema antes de se realizar a atualização dos parâmetros, enquanto naquele o gradiente é estimado a partir de dados (que pode ser tanto uma única entrada quanto um lote contendo várias delas - *minibatch*) escolhido aleatoriamente de cada vez.

Um pseudo-código para o algoritmo seria:

- Dado um tensor de parâmetros w, uma função a ser minimizada Q e um tamanho de passo(ou *learning rate*) lr, repita até atingir o mínimo:

- Embaralhe os exemplos do conjunto de dados
- Para i = 1,...,n faça $w := w - lr * \nabla Q_i(w)$

Dessa forma, a fim de implementar o código, o primeiro fator que necessita de atenção é garantir que haja espaço na memória para operar o vetor de parâmetros.

Outro ponto importante é que o algoritmo depende de um conjunto de parâmetros iniciais. No caso do pytorch, ao criar o modelo como uma classe filha da classe Torch.nn[2], esse vetor já é inicializado com pesos aleatórios.

4.1.1 Código implementado:

```
class Otimizador_SGD(optim.Optimizer):

    def __init__(self, parametros_modelo, lr):

        defaults = dict(lr = lr)
        super().__init__(parametros_modelo, defaults)

    def step(self, closure=None):

        loss = None
        if closure is not None:
            loss = closure()

        for group in self.param_groups:

            for p in group['params']:
                if p.grad is None:
                    continue

                p.data = p.data - self.defaults['lr']*p.grad

        return loss
```

4.2 ADAM:

É uma variação do algoritmo acima. Sua principal inovação é o calculo não só das taxas de aprendizagem individuais para cada parâmetro, mas também do primeiro e segundo momento dos gradientes.

O algoritmo, conforme o artigo, é:

[2] o código da rede está disponível nos apêndices caso essa passagem não tenha ficado clara.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figure 1: Algoritmo Adam

O primeiro ponto que gostaria de ressaltar é que neste algoritmo, diferentemente do SGD, há necessidade de guardar a informação de dois dicionários - m e v - de uma iteração para a outra. Esses dois vetores devem ter o mesmo formato dos parâmetros que serão atualizados. Ou seja: para cada vetor de parâmetros que formos atuar, as entradas de m e v devem ter o mesmo *shape* que o dado.

Com isso, a questão de mensurarmos se há ou não memória o suficiente para guardar os hiperparâmetros do algoritmo se torna mas relevante.

```
class Otimizador_ADAM(optim.Optimizer):

    def __init__(self, parametros_modelo, lr, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8):

        defaults = dict(lr=lr, beta1=beta1, beta2 = beta2, eps=epsilon)
        params = parametros_modelo
        self.m = {}
        self.v = {}
        self.t = 0
        super().__init__(params, defaults)

    def step(self, closure=None):

        loss = None
        if closure is not None:
            loss = closure()

        self.t += 1
```

```
for group in self.param_groups:

    for p in group['params']:
        if p.grad is None:
            continue

        g_t = p.grad

        if self.t == 1:
            self.m[p] = torch.zeros_like(g_t)
            self.v[p] = torch.zeros_like(g_t)
            self.beta1t = 1.0
            self.beta2t = 1.0


        self.m[p] = self.defaults['beta1']*self.m[p]
                            + (1-self.defaults['beta1'])*g_t
        self.v[p] = self.defaults['beta2']*self.v[p]
                            + (1-self.defaults['beta2'])*(g_t*g_t)
        self.beta1t *= self.defaults['beta1']
        self.beta2t *= self.defaults['beta2']



        m_hat = self.m[p]/(1-self.beta1t)
        v_hat = self.v[p]/(1-self.beta2t)


        p.data = p.data - self.defaults['lr'] * m_hat/(torch.sqrt(v_hat)
                            + self.defaults['eps'])


return  loss
```

5 Resultados da implementação

Para testar a eficiência dos otimizadores, foi implementada uma rede neural que utiliza uma parte da "resnet-18" já treinada [3] e depois algumas convoluções[4] para adequar a rede ao problema proposto.

[3] Essa rede pode ser baixada via pytorh com o comando: "from torchvision import models" e depois atrelar a uma variável o comando "models.resnet18(pretrained=True)". Há uma referência sobre a rede nos apêndices.

[4] em linhas gerais são filtros que percorrem a matriz da imagem e extraem informações dela. Nas referências há um texto em inglês que explica com maiores detalhes

O dataset foi tirado de uma das competições do site Kaggle e o objetivo era resolver o desafio de treinar uma rede neural capaz de classificar corretamente imagens de gatos e cachorros.

Exatamente a mesma rede e as mesmas configurações foram utilizadas em todos os experimentos: dataset (1000 imagens para o treino e 100 para validação), rede, tamanho de lote (32 imagens por lote no treino e 16 na validação), tamanho do passo (lr = 1e-3) e número de vezes que faremos o processo (epocas = 10)

5.1 SGD:

O primeiro teste que realizei foi para comparar o SGD que já está implementado no pytorch com a minha implementação. O parâmetro do método original da biblioteca foi apenas a taxa de aprendizagem (lr = 1e-3).

O melhor resultado obtido pelo método nativo foi de 98,0% de acurácia e uma função objetivo avaliada em 0,12:

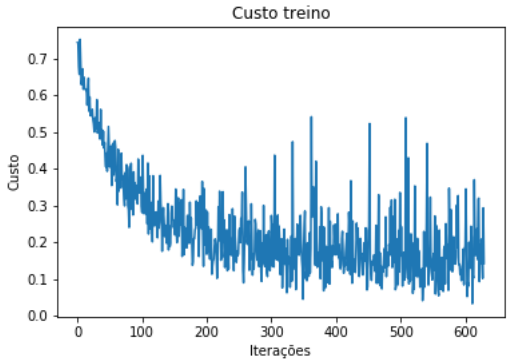

Figure 2: Gráfico do custo por iteração no treino usando o otimizador nativo

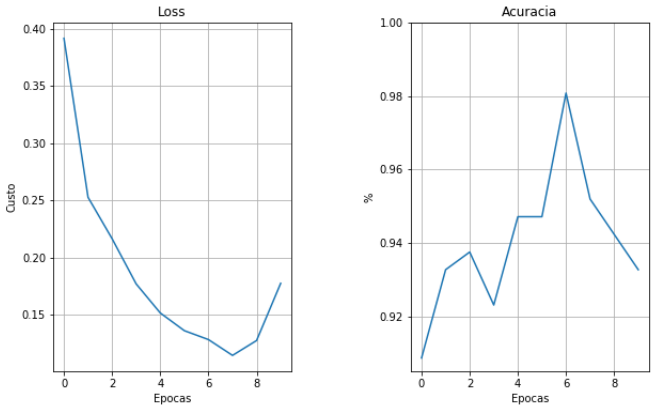

Figure 3: Gráfico do custo e da acurácia na validação a cada época

Minha implementação não possui outro parâmetro além do tamanho do passo (lr = 1e-3). Quanto aos resultados, ela obteve em seu melhor resultado um acurácia de 96,2% minimizando a função objetivo a 0,10:

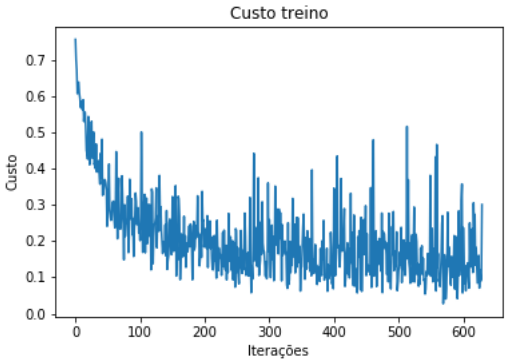

Figure 4: Gráfico do custo por iteração no treino usando o otimizador programado

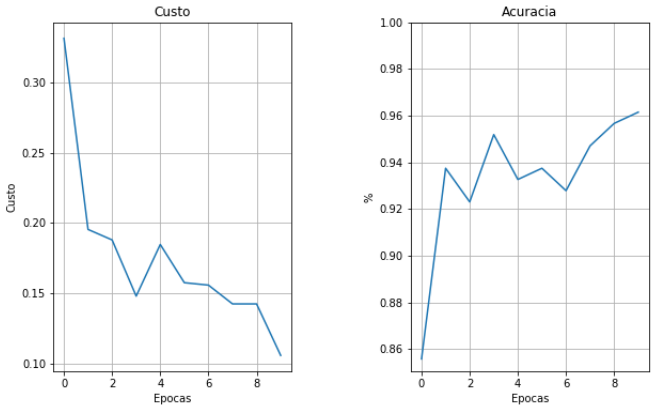

Figure 5: Gráfico do custo e da acurácia na validação a cada época

5.2 ADAM:

O mesmo tipo de teste foi conduzido para o otimizador ADAM: primeiro usei o do pytorch e depois a minha implementação. Nesse caso, ambos os otimizadores foram configurados de acordo com algoritmo (ver seção 5.3) e, portanto, têm os mesmos parâmetros.

O ADAM da biblioteca teve como melhor resultado uma acurácia de 96,2% e 0,13 como valor da função objetivo:

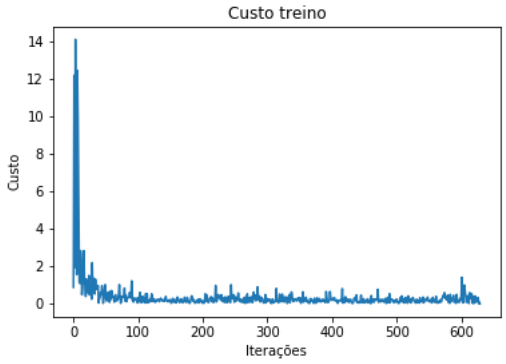

Figure 6: Gráfico do custo por iteração no treino usando o otimizador nativo

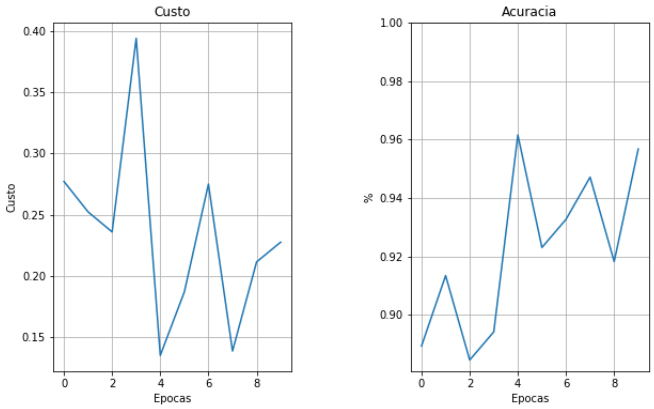

Figure 7: Gráfico do custo e da acurácia na validação a cada época

Já a implementação manual teve uma acurácia de 94,7% com 0,21 como valor da função objetivo:

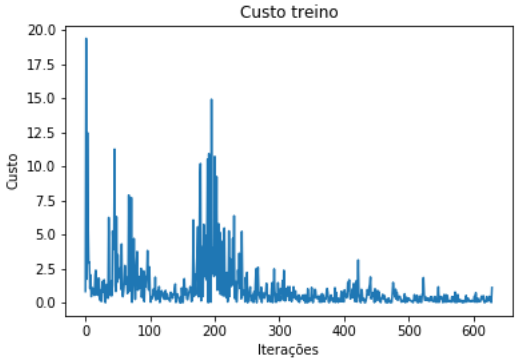

Figure 8: Gráfico do custo por iteração no treino usando o otimizador implementado

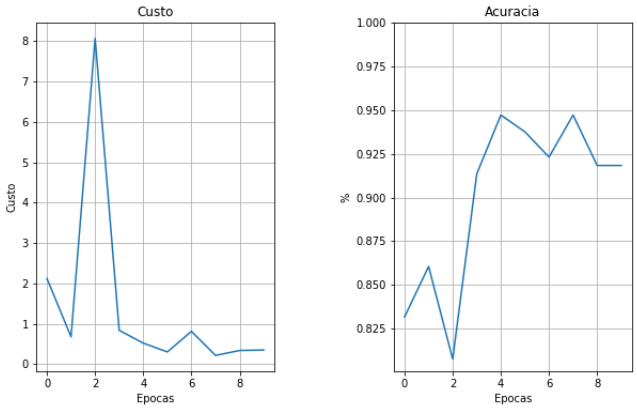

Figure 9: Gráfico do custo e da acurácia na validação a cada época

5.3 Conclusão

Comparando os resultados, noto que os resultados foram semelhantes. Assim, os algoritmos mostrados na seção 4 são eficazes e, dessa forma, é possível implementar novos algoritmos no *Pytorch*

6 Apêndice

6.1 Rede

Para ser testada, a rede está disponível no link: https://github.com/v157490/Projeto-Supervisionado

O modelo usado na rede foi:

```
class Modelo(nn.Module):

    def __init__(self):
        super().__init__()
        self.resnet = nn.Sequential(*list(models.resnet18(pretrained=True).
                                          children())[:-2])
        self.conv1 = nn.Conv2d(in_channels=512,out_channels=256,
                                          kernel_size=(4,4))
        self.conv2 = nn.Conv2d(in_channels=256,out_channels=1,
                                          kernel_size=(5,5))
        self.conv3 = nn.Conv2d(in_channels=1, out_channels=1,
                                          kernel_size=(1,1))

        for params in self.resnet.parameters():
```

```
            params.requires_grad = False

    def forward(self, x):
        x = self.resnet(x)
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)

        return x.squeeze()
```

6.2 SAGA

Houve uma tentativa de implementarmos o otimizador SAGA. Ela foi frustrada devido ao fato desse algoritmo utilizar uma matriz com todos os gradientes de todos os parâmetros. Isso gera um alto custo de memória; para a rede que usei no projeto estima-se haver mais de 61,000,000 entradas na matriz, vezes o tamanho de um float nos dá mais 1500 gb de memória para alocar apenas essa matriz.

Chamo, portanto, a atenção para esse tipo de problema: estimar se o computador consegue alocar memória para todas as etapas do otimizador.

6.3 Referências

- Teoria do SGD: https://arxiv.org/abs/1609.04747
- Teoria do ADAM: https://arxiv.org/abs/1412.6980
- Teoria do SAGA: https://arxiv.org/abs/1407.0202
- Sobre a rede ResNet: https://arxiv.org/abs/1512.03385
- Sobre a Convolução: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/
- PyTorch: https://pytorch.org/
- Optimzer: https://pytorch.org/docs/stable/optim.htmltorch.optim.Optimizer
- Dataset: https://www.kaggle.com/c/dogs-vs-cats/data
- Códigos: https://github.com/v157490/Projeto-Supervisionado