Machine Learning Under a Modern Optimization Lens

Neural Networks

Fully Connected Neural Networks

Abaixo está o código de exemplo para criação de uma rede totalmente conectada.

```
using Flux: Chain, Dense

model = Chain(
    Dense(input_size, hidden_size, non_linearity),
    Dense(hidden_size, output_size)
)

# ... train model
```

Lembre que há tutoriais disponibilizados na documentação Flux detalhando o processo de definição de uma rede e seu treinamento (overview e quickstart).

a. Escolha uma função não linear e simule um conjunto de dados, dividindo-o em dois conjuntos: treino e validação. Defina uma rede neural com uma camada escondida, como definida no exemplo acima. Utilizando os dados simulados, treine a rede neural acompanhando sua perda de treino e validação por época. Analise o gráfico.

b. Mudando o número de neurônios na camada escondida, `hidden_size` no exemplo acima, faça um gráfico da melhor perda de validação encontrada para cada quantidade de neurônios.

c. Utilizando o melhor modelo encontrado acima - aquele com a menor perda de validação entre todos os diferentes modelos treinados -, modifique a função perda para considerar regularização dos parâmetros e.g. flux regularization, treinando-o por mais algumas épocas. Faça novamente um gráfico da sua perda por épocas, julgando o resultado final.

```
using Flux: mse

# to access the networks' parameters, use Flux.params(model)
loss(model, X, y) = mse(model(X), y) + # regularization
```

Convolutional Neural Networks

Considere a rede treinada disponível em `cnn_mnist.bson` (como salvar e recuperar modelos).

```
model = Chain(

    # `Conv` expects (width, height, channels, batch_size)
```

```
    # `channels` might be hidden, as there is only one channel.
    # to fix this, it is possible to create a dimension with 1
    # entry using `unsqueeze`, for example:
    # x -> Flux.unsqueeze(x, 3), # (28, 28) -> (28, 28, 1)

    Conv((3, 3), 1 => 16, stride=(1, 1), pad=(1, 1), relu),
    MaxPool((2, 2)),

    Conv((3, 3), 16 => 32, stride=(1, 1), pad=(1, 1), relu),
    MaxPool((2, 2)),

    Conv((3, 3), 32 => 32, stride=(1, 1), pad=(1, 1), relu),
    MaxPool((2, 2)),

    flatten,
    Dense(288, 10),

    # Finally, softmax to get nice probabilities
    softmax,
)
```

Considere também os dados de teste que podem ser importados da seguinte forma. Lembre que o modelo foi treinado com imagens no domínio `[-1, 1]`, enquanto os dados originais estão entre `[0, 1]`, tornando necessário normalizá-los.

```
using MLDatasets: MNIST

dataset = MNIST(split=:test)
X, y = dataset
normalized = @. 2.0f0 * X - 1.0f0

# # `Conv` expects (width, height, channels, batch_size)
# X = reshape(normalized, size(X, 1), size(X, 2), 1, :)
```

Finalmente, considerando uma amostra aleatória do conjunto de teste, `X_test`.

a. A partir de `X_test`, julgue algumas das predições do modelo pré-treinado.

b. A partir de `X_test`, calcule a acurácia do modelo nessa amostra. É importante ressaltar que o modelo faz previsão da probabilidade de pertencimento a uma classe, e não a qual classe a imagem pertence, se fazendo necessário tratar as probabilidade para encontrar a classe de pertencimento (e.g. `argmax`).

Considere a função abaixo, uma implementação de é uma implementação de *Fast Gradient Sign Attack 5 from Tensorflow Example and PyTorch Example*. Ela recebe: um modelo `cnn`, exemplo de entrada `x` (`size(x) = (w, h, c, b)`, como `X_test`), um rótulo a ser considerado `attack_label` (`attack_label = 1`) e um valor para o ruído `epsilon` (`epsilon = 5e-2`).

```
"""
Given a flux model `cnn` and an example `x`, calculates the gradient for each
input pixel. A new image, `perturbed_image` is created with a noise controlled by
`epsilon`. Values pass through `clamp` to ensure it stays on the expected domain
of the trained `cnn`
"""
function fgsm_attack(cnn, x, attack_label::Int = 1, epsilon = 5e-2)
    grad = Flux.jacobian(x -> cnn(x), x)[1] # (labels)
    data_grad = reshape(grad, :, 28, 28)
    perturbed_image = clamp.(
        x + epsilon .* sign.(data_grad[attack_label, :, :]),
        -1., 1.
    )
    return perturbed_image
end
```

c. Utilizando diferentes valores de `epsilon`, julgue a capacidade de generalização da rede.

d. Considerando o ataque acima e alguns exemplos de imagens perturbadas, cite formas que poderiam ser incorporadas no treinamento da rede para reduzir o efeito desse ataque.

Recurrent Neural Networks

a. Escolha ou simule uma série temporal. Divida os conjuntos de dados para as estapas de treino e teste, passando os dados por um pré-processamento, caso necessário. Lembre de criar também um pós-processamento de forma que seja possível transformar a série prevista para o mesmo domínio da original.

Considere o código abaixo, que indica o funcionamento de uma rede recorrente: o cálculo feito em cada camada, a função de ativação mais comum e a manutenção do estado escondido. Também vale considerar a página do pacote sobre modelos recorrentes e essa conversa no julia discourse sobre redes recorrentes.

```
using Flux: tanh
using Flux: RNNCell, Recur

input_size, output_size = 2, 3
# given
# `size(x) = (input_size, batch_size)`
# `size(y) = (output_size, batch_size)`

model = RNNCell(input_size, output_size)
Wxh, Whh, b, _ = Flux.params(model)

isapprox(model(h, x)[1], tanh.(Wxh * x .+ Whh * h .+ b))# true
```

```
rmodel = Flux.Recur(model, h) # keep internal hidden state
# while rmodel keeps its state, model does not
isapprox(model(h, x)[1], rmodel(x)) # true
isapprox(model(h, x)[1], rmodel(x)) # false

recurrent = RNN(input_size, output_size)
Flux.reset!(recurrent) # reset hidden state
```

b. Defina e treine uma rede recorrente com os dados escolhidos.

c. Julgue o treinamento com um gráfico perda por épocas.

d. Faça previsões para o conjunto de teste.