Machine Learning Under a Modern Optimization Lens

Neural Networks

Fully Connected Neural Networks

a. Escolha uma função não linear, gerando dados e dividindo em dois conjuntos: treino e validação. Defina uma rede neural com uma camada escondida. Visualize a perda por época.

```
using Flux: Chain, Dense

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

# ... train model
```

b. Variando os valores da camada escondida, visualize a perda por neurônios da camada escondida.

c. Utilizando o melhor modelo encontrado acima, modifique a função perda para considerar regularização dos parâmetros 1, treinando-o por mais algumas épocas. Julgue a perda final do modelo em um gráfico perda por épocas.

```
using Flux: mse

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

Os tutoriais disponibilizados na documentação detalhando as funções estão diponíveis em: Overview e Quickstart.

Convolutional Neural Networks

Considere a rede treinada disponível em `cnn-mnist.bson` (Salvando e Recuperando Modelos).

```
model = Chain(

    # `Conv` expects (width, height, channels, batch_size)
    # x -> Flux.unsqueeze(x, 3),

    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,
)
```

a. Julgue algumas de suas predições.

```
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, :)
```

b. Considerando uma amostra aleatória do conjunto de teste, calcule a acurácia do modelo. Repare que o modelo faz previsão da probabilidade de pertencimento a uma classe, e não a qual classe a imagem pertence.

c. Considere a função abaixo, que recebe: exemplo de entrada `x` (`size(x) = (w, h, c, b)`), um rótulo a ser considerado `label` e um valor para o ruído `epsilon`. Teste a função para diferentes exemplos e valores de epsilon. Julgue a capacidade de generalização da rede.

```
function fgsm_attack(x, attack_label::Int = 1, epsilon = 5e-2)
    # Fast Gradient Sign Attack [5] from [Tensorflow Example][6]
    # and [PyTorch Example][7]
    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
```

d. Considerando o ataque acima, cite formas que poderiam ser incorporadas no treinamento da rede para reduzir o efeito desse ataque.

Recurrent Neural Networks

a. Escolha uma série temporal. Após dividir os conjuntos para as etapas de treino e teste, processe os dados. Prepare-os para treinar uma rede recorrente da sua escolha.

b. Treine a rede recorrente. O pacote Flux disponibiliza uma página sobre modelos recorrentes. Além disso há essa uma conversa no julia discourse sobre redes recorrentes.

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

input_size, output_size = 2, 3
x = rand(Float32, input_size)
y = rand(Float32, output_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
```

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

d. Faça previsões para o conjunto de teste.