Exploring Deep Learning through Flux.jl: Insights into Core Mechanisms and Datasets

Mannes Yolhan

December 07 2023

1 Introduction

2 Core Mechanisms

3 Flux Built-in Layers

4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

1 Introduction
- What is Deep Learning?
- What is Flux.jl ?

2 Core Mechanisms

3 Flux Built-in Layers

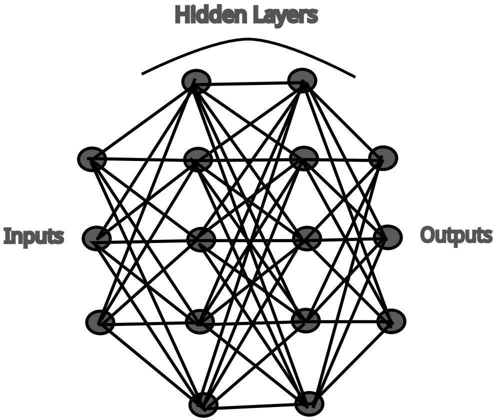
4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

What is Deep Learning?

- A subset of machine learning where artificial neural networks, algorithms inspired by the human brain, learn from large amounts of data.



What is Deep Learning?

- A subset of machine learning where artificial neural networks, algorithms inspired by the human brain, learn from large amounts of data.
- Ability to learn and improve from experience without being explicitly programmed.

What is Deep Learning?

- A subset of machine learning where artificial neural networks, algorithms inspired by the human brain, learn from large amounts of data.
- Ability to learn and improve from experience without being explicitly programmed.
- Efficient in processing large datasets and recognizing patterns.

1 Introduction
- What is Deep Learning?
- What is Flux.jl ?

2 Core Mechanisms

3 Flux Built-in Layers

4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

What is Flux.jl ?

Flux.jl is a comprehensive package within the Julia programming ecosystem, designed specifically for deep learning applications.

- Open source Julia package dedicated to deep learning.

Michael Innes *et al.*
Fashionable Modelling with Flux
CoRR, 2018

FluxAI
Flux, The Elegant Machine Learning Stack
`https://fluxml.ai/`

What is Flux.jl ?

Flux.jl is a comprehensive package within the Julia programming ecosystem, designed specifically for deep learning applications.

- Open source Julia package dedicated to deep learning.
- Full support for GPU utilization and Automatic Differentiation.

Michael Innes *et al.*
Fashionable Modelling with Flux
CoRR, 2018

FluxAI
Flux, The Elegant Machine Learning Stack
`https://fluxml.ai/`

What is Flux.jl ?

Flux.jl is a comprehensive package within the Julia programming ecosystem, designed specifically for deep learning applications.

- Open source Julia package dedicated to deep learning.
- Full support for GPU utilization and Automatic Differentiation.
- Wide array of tools for efficient data processing.

What is Flux.jl ?

Flux.jl is a comprehensive package within the Julia programming ecosystem, designed specifically for deep learning applications.

- Open source Julia package dedicated to deep learning.
- Full support for GPU utilization and Automatic Differentiation.
- Wide array of tools for efficient data processing.
- Vast selection of predefined layers for various neural network architectures.

1 Introduction

2 Core Mechanisms
- Decomposition of complex nested structures
- Automatic reverse differentiation
- Descent-based minimization methods

3 Flux Built-in Layers

4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

Core Mechanisms

The core mechanisms of Flux are the following:

- Decomposition of complex nested structures (Functors.jl)

Core Mechanisms

The core mechanisms of Flux are the following:

- Decomposition of complex nested structures (Functors.jl)
- Automatic reverse differentiation (Zygote.jl)

FluxML
Zygote.jl
https://github.com/FluxML/Zygote.jl

Core Mechanisms

The core mechanisms of Flux are the following:

- Decomposition of complex nested structures (Functors.jl)
- Automatic reverse differentiation (Zygote.jl)
- Descent-based minimization methods (Optimisers.jl)

1 Introduction

2 Core Mechanisms
- Decomposition of complex nested structures
- Automatic reverse differentiation
- Descent-based minimization methods

3 Flux Built-in Layers

4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

Nested structures and deep learning

- Nested structures are commonly employed in deep learning, primarily due to their efficiency in data processing.

```julia
using Flux,Flux.Functors

struct Linear{T1<:Real,T2<:Function}
    W ::Matrix{T1}
    b ::Vector{T1}
    f ::T2
end ✓

@functor Linear


struct Chain
    layers ::Vector{Linear}
end

@functor Chain
```

Decomposition of complex nested structures

- Goal : Facilitate access to the parameters that need optimization.

```julia
using Flux,Flux.Functors ✓
struct Linear{T1<:Real,T2<:Function}
    W ::Matrix{T1}
    b ::Vector{T1}
    f ::T2
end ✓
@functor Linear ✓
struct Chain
    layers ::Vector{Linear}
end ✓
@functor Chain ✓

Linear(n,m,f) = Linear(randn(m,n),randn(m),f); ✓
model = Chain([
    Linear(1,64,tanh),
    Linear(64,64,tanh),
    Linear(64,64,tanh),
    Linear(64,1,identity)
]); ✓
Flux.params(model) .|> length |> sum 8513
```

Decomposition of complex nested structures

- Goal : Facilitate access to the parameters that need optimization.
- Limitation : All parameters must be stored on the heap.

```julia
using Flux,Flux.Functors ✓
struct Linear{T1<:Real,T2<:Function}
    W ::Matrix{T1}
    b ::Vector{T1}
    f ::T2
end ✓
@functor Linear ✓
struct Chain
    layers ::Vector{Linear}
end ✓
@functor Chain ✓

Linear(n,m,f) = Linear(randn(m,n),randn(m),f); ✓
model = Chain([
    Linear(1,64,tanh),
    Linear(64,64,tanh),
    Linear(64,64,tanh),
    Linear(64,1,identity)
]); ✓
Flux.params(model) .|> length |> sum 8513
```

1 Introduction

2 Core Mechanisms
- Decomposition of complex nested structures
- Automatic reverse differentiation
- Descent-based minimization methods

3 Flux Built-in Layers

4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

Automatic differentiation

- Computational technique used to calculate the gradient of a function relative to its inputs.

Automatic differentiation

- Computational technique used to calculate the gradient of a function relative to its inputs.
- ForwardDiff : Forward mode (Number of inputs $<$ Number of outputs).

Automatic differentiation

- Computational technique used to calculate the gradient of a function relative to its inputs.
- ForwardDiff : Forward mode (Number of inputs < Number of outputs).
- Zygote : Forward mode (Number of inputs > Number of outputs).

```
using Flux.Zygote ✓

f(x) = 2. * x.^2 |> sum; ✓

x = rand(3); ✓
gradient(f,x)[1] == 4 .* x true
```

Mixing Zygote and Functors

- Zygote is capable of differentiating nested structures, provided they are appropriately tagged by Functors.

```
(l::Linear)(x) = l.f.(muladd(l.W,x,l.b)) ✓

model = Linear(100,1,identity); ✓

compute_model(model,x) = model(x)[1]; ✓

x = rand(100); ✓

g = gradient(compute_model,model,x); ✓

g[1].W 1×100 Matrix{Float64}:

g[1].b 1-element Vector{Float64}:
```

1 Introduction

2 Core Mechanisms
- Decomposition of complex nested structures
- Automatic reverse differentiation
- Descent-based minimization methods

3 Flux Built-in Layers

4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

Descent-Based Minimization Methods

- Basic Descent Method:
  - $f : \mathbb{R}^n \to \mathbb{R}$ at least C^1,

```
using Flux.Zygote ✓
using Flux.Optimisers ✓
struct DescentAlg <: Optimisers.AbstractRule
    α :: Float64
end ✓
function Optimisers.apply!(o::DescentAlg, state, x, x̄)
    newx̄ =  o.α .* x̄
    nextstate = state + 1
    return nextstate, newx̄
end ✓
Optimisers.init(o::DescentAlg, x::T) where T = 1 ✓
opt_rule = DescentAlg(0.1); ✓
x = rand(3); ✓
opt = Optimisers.setup(opt_rule, x); ✓
f(x) = 2. * x.^2 |> sum; ✓
f(x) 0.3587933958011746
for _ in 1:10
    Optimisers.update!(opt,x,gradient(f,x)[1]);
end ✓
f(x) 1.3118055022973196e-5
```

Descent-Based Minimization Methods

- Basic Descent Method:
  - $f : \mathbb{R}^n \to \mathbb{R}$ at least C^1,
  - $x \in \mathbb{R}^n$,

```
using Flux.Zygote ✓
using Flux.Optimisers ✓
struct DescentAlg <: Optimisers.AbstractRule
    α :: Float64
end ✓
function Optimisers.apply!(o::DescentAlg, state, x, x̄)
    newx̄ =  o.α .* x̄
    nextstate = state + 1
    return nextstate, newx̄
end ✓
Optimisers.init(o::DescentAlg, x::T) where T = 1 ✓
opt_rule = DescentAlg(0.1); ✓
x = rand(3); ✓
opt = Optimisers.setup(opt_rule, x); ✓
f(x) = 2. * x.^2 |> sum; ✓
f(x) 0.3587933958011746
for _ in 1:10
    Optimisers.update!(opt,x,gradient(f,x)[1]);
end ✓
f(x) 1.3118055022973196e-5
```

Descent-Based Minimization Methods

- Basic Descent Method:
  - $f : \mathbb{R}^n \to \mathbb{R}$ at least C^1,
  - $x \in \mathbb{R}^n$,
  - x^k such that,

$$x^{k+1} = x^k - \alpha \nabla f(x^k),$$

where α is the step size.

```julia
using Flux.Zygote
using Flux.Optimisers
struct DescentAlg <: Optimisers.AbstractRule
    α :: Float64
end
function Optimisers.apply!(o::DescentAlg, state, x, x̄)
    newx̄ =  o.α .* x̄
    nextstate = state + 1
    return nextstate, newx̄
end
Optimisers.init(o::DescentAlg, x::T) where T = 1
opt_rule = DescentAlg(0.1);
x = rand(3);
opt = Optimisers.setup(opt_rule, x);
f(x) = 2. * x.^2 |> sum;
f(x) 0.3587933958011746
for _ in 1:10
    Optimisers.update!(opt,x,gradient(f,x)[1]);
end
f(x) 1.3118055022973196e-5
```

Descent-Based Minimization Methods

- Basic Descent Method:
- Adam Method :

$$m^{k+1} = \beta_1 m^k + (1-\beta_1)\nabla f(x^k)$$

$$v^{k+1} = \beta_2 v^k + (1-\beta_2)\nabla f(x^k)^2$$

```julia
@kwdef struct AdamAlg <: Optimisers.AbstractRule
    α = 0.001
    beta = (0.9, 0.999)
    epsilon = 1e-8
end AdamAlg

Optimisers.init(o::AdamAlg, x) = (zero(x), zero(x), o.beta) ✓

function Optimisers.apply!(o::AdamAlg, state, x, dx) where T
    α, β, ϵ = o.α, o.beta, o.epsilon
    mt, vt, βt = state

    mt =@. β[1] * mt + (1 - β[1]) * dx
    vt =@. β[2] * vt + (1 - β[2]) * abs2(dx)
    dxp =@. mt / (1 - βt[1]) / (sqrt(vt / (1 - βt[2])) + ϵ) * α

    return (mt, vt, βt .* β), dxp
end ✓

opt_rule = AdamAlg(α= 0.1); ✓
```

$$x^{k+1} = x^k - \frac{\alpha m^{k+1}}{(1-(\beta_1)^k)\left(\sqrt{\frac{v^{k+1}}{1-(\beta_2)^k}}+\varepsilon\right)}$$

where α is the step size, β_1, β_2 and ε are parameters.

Mixing Optimisers, Zygote and Functors

- Optimisers is capable of optimizing nested structures, provided they are appropriately tagged by Functors.

```
(l::Linear)(x) = l.f.(muladd(l.W,x,l.b)) ✓

model = Linear(100,1,identity); ✓

compute_model(model,x) = model(x)[1]; ✓

x = rand(100); ✓

opt_rule = AdamAlg(α= 0.1); ✓
opt = Optimisers.setup(opt_rule, model); ✓

model(x)[] -4.991810896328357

for _ in 1:10
    g = gradient(compute_model,model,x)[1];
    Optimisers.update!(opt,model,g);
end ✓

model(x)[1] -58.87701713966101
```

1 Introduction

2 Core Mechanisms

3 Flux Built-in Layers
- Basic Flux Examples with Dense Layers
- The Convolution Layer
- The MultiHeadAttention(MHA) Layer

4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

Commonly Used Layers in Neural Networks

- The Dense Layer or fully connected layer : Connects every neuron in one layer to every neuron in the next layer.

Commonly Used Layers in Neural Networks

- The Dense Layer or fully connected layer : Connects every neuron in one layer to every neuron in the next layer.
- The Convolution Layer for image processing : Applie a convolution operation to the input, passing the result to the next layer.

Commonly Used Layers in Neural Networks

- The Dense Layer or fully connected layer : Connects every neuron in one layer to every neuron in the next layer.
- The Convolution Layer for image processing : Applie a convolution operation to the input, passing the result to the next layer.
- The Normalizing Layer or the batch normalization : Improve model stability and reduce overfitting by normalizing layer inputs.

Commonly Used Layers in Neural Networks

- The Dense Layer or fully connected layer : Connects every neuron in one layer to every neuron in the next layer.
- The Convolution Layer for image processing : Applie a convolution operation to the input, passing the result to the next layer.
- The Normalizing Layer or the batch normalization : Improve model stability and reduce overfitting by normalizing layer inputs.
- The Multi-Head Attention Layer for processing sequential data (ex : text) : Multiple dense layers devided in key, query and value.

1 Introduction

2 Core Mechanisms

3 Flux Built-in Layers
- Basic Flux Examples with Dense Layers
- The Convolution Layer
- The MultiHeadAttention(MHA) Layer

4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

The Linear Regression Model

- The linear regression model is a Dense Layer with only one neuron. In this setup, the identity function is used as the activation function.

```
using Flux

model = Flux.Dense(1,1,identity);

f(x) = x^3-x^2+x+2;

N=1000;

x = rand(-1:0.01:1,1,N);

y = f.(x);

loss(model,x,y) = Flux.mse(model(x),y);

loss(model,x,y)   2.9885269487981727
```

The Linear Regression Model

- The linear regression model is a Dense Layer with only one neuron. In this setup, the identity function is used as the activation function.
- The primary objective in this model is to minimize the mean squared error (MSE), which is the loss function.

```
using Flux

model = Flux.Dense(1,1,identity);

f(x) = x^3-x^2+x+2;

N=1000;

x = rand(-1:0.01:1,1,N);

y = f.(x);

loss(model,x,y) = Flux.mse(model(x),y);

loss(model,x,y)  2.9885269487981727
```

The Linear Regression Model

- The built-in layers, such as Dense, are Functors by default in many deep learning frameworks.

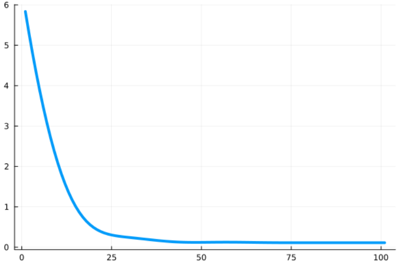
```
N=1000; ✓
epoch = 100; ✓

x = rand(-1:0.01:1,1,N); ✓
y = f.(x); ✓

loss(model,x,y) = Flux.mse(model(x),y); ✓
loss(model,x,y) 4.0829329362030355

opt_rule = Flux.Adam(0.1); ✓
opt = Flux.setup(opt_rule, model); ✓

for _ in 1:epoch
    g = gradient(loss,model,x,y)[1];
    Flux.update!(opt,model,g);
end ✓
loss(model,x,y) 0.12288050564779106
```

The Linear Regression Model

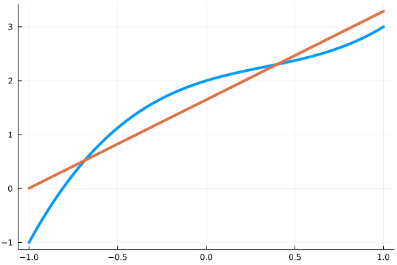
- The built-in layers, such as Dense, are Functors by default in many deep learning frameworks.
- In this context, N represents the number of training examples, and 'epoch' refers to the number of iteration for the optimization process.

```
N=1000; ✓
epoch = 100; ✓

x = rand(-1:0.01:1,1,N); ✓
y = f.(x); ✓

loss(model,x,y) = Flux.mse(model(x),y); ✓
loss(model,x,y) 4.0829329362030355

opt_rule = Flux.Adam(0.1); ✓
opt = Flux.setup(opt_rule, model); ✓

for _ in 1:epoch
    g = gradient(loss,model,x,y)[1];
    Flux.update!(opt,model,g);
end ✓
loss(model,x,y) 0.12288050564779106
```

The Linear Regression Model



(a) Plot of the loss function



(b) Plot of the function(blue) and the model(red)

1 Introduction

2 Core Mechanisms

3 Flux Built-in Layers
- Basic Flux Examples with Dense Layers
- The Convolution Layer
- The MultiHeadAttention(MHA) Layer

4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

Exemple of convolution operation on matricies

- Let M be a 3x3 matrix, and F a 2x2 filter.
- We apply F to M to compute the resulting matrix N.
- The convolution involves element-wise multiplications and summations.
- Here, we demonstrate the computation of the first element of N.

$$M = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \quad F = \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} \quad N = \begin{bmatrix} N_{11} & N_{12} \\ N_{21} & N_{22} \end{bmatrix}$$

$$N_{11} = M_{11} \cdot F_{11} + M_{12} \cdot F_{12} + M_{21} \cdot F_{21} + M_{22} \cdot F_{22}$$

Define the convolution layer at hand

- Suppose we have a *nxm* matrix *M* and a *lxk* filter *F* with a padding of (p_1, p_2) and a stride of (s_1, s_2).

```julia
struct MyConv
    F ::Array
    D ::Dense
    pad ::Tuple
    stride ::Tuple
end

@functor MyConv

function MyConv(T ::Tuple,n,m,f ::T1;pad=(0,0),
stride=(1,1)) where T1<:Function
    D = Flux.Dense(n=>m,f)
    MyConv(randn(T...,n),D,pad,stride)
end;

model = MyConv((3,3),1,3,tanh,pad=(1,1));
```

Define the convolution layer at hand

- Suppose we have a *nxm* matrix M and a *lxk* filter F with a padding of (p_1, p_2) and a stride of (s_1, s_2).
- Then the operation is,

$$N_{ij} = M_{(i-1)s_1+m,(j-1)s_2+n} F_{m,n}$$

```
function padarray(A::Array, pad::Tuple)
    original_dims = size(A)
    new_dims = original_dims[1:2] .+ 2 .* pad
    B = zeros(eltype(A), new_dims...,original_dims[3:end]...)
    indices = [p+1:p+d for (p, d) in zip(pad, original_dims[1:2])]
    B[indices...,:,:] = A
    return B
end padarray (generic function with 1 method)
function (model ::MyConv)(x)
    P1,P2 = model.pad
    S1,S2 = model.stride
    xpad = padarray(x,(P1,P2))
    F = model.F
    @tullio C[i, j,k,l] :=  xpad[(i-1)*$S1 + m, (j-1)*$S2 + n,k,l]
    * F[m, n,k]
    return permutedims(model.D(permutedims(C,[3,1,2,4])),[2,3,1,4])
end ✓
```

Define the convolution layer at hand

- Suppose we have a *nxm* matrix M and a *lxk* filter F with a padding of (p_1, p_2) and a stride of (s_1, s_2).
- N will be of dimention $\frac{n+2p_1-l}{s_1}+1$ by $\frac{m+2p_2-k}{s_2}+1$.

```
function padarray(A::Array, pad::Tuple)
    original_dims = size(A)
    new_dims = original_dims[1:2] .+ 2 .* pad
    B = zeros(eltype(A), new_dims...,original_dims[3:end]...)
    indices = [p+1:p+d for (p, d) in zip(pad, original_dims[1:2])]
    B[indices...,:,:] = A
    return B
end padarray (generic function with 1 method)
function (model ::MyConv)(x)
    P1,P2 = model.pad
    S1,S2 = model.stride
    xpad = padarray(x,(P1,P2))
    F = model.F
    @tullio C[i, j,k,l] :=  xpad[(i-1)*$S1 + m, (j-1)*$S2 + n,k,l]
    * F[m, n,k]
    return permutedims(model.D(permutedims(C,[3,1,2,4])),[2,3,1,4])
end ✓
```

The flux convolution layer

- Far More Optimized (2 times less allocations)

```
using Flux ✓

model = Conv((3,3),1=>3,tanh,pad=(1,1)); ✓

xtest = rand(32,32,1,100); ✓

model(xtest) 32×32×3×100 Array{Float32, 4}:
```

1 Introduction

2 Core Mechanisms

3 Flux Built-in Layers
- Basic Flux Examples with Dense Layers
- The Convolution Layer
- The MultiHeadAttention(MHA) Layer

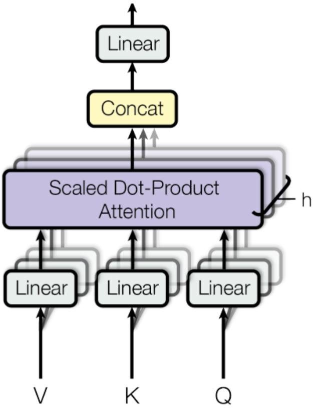
4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

The MultiHeadAttention(MHA) Layer

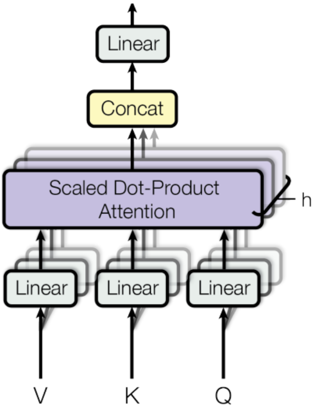
- Three principal components: Value, Key, Query.



Vaswani *et al.*
Attention is all you need
Advances in neural information processing systems,2017

The MultiHeadAttention(MHA) Layer

- Three principal components: Value, Key, Query.
- Split into h parts and pass them through dense layers.

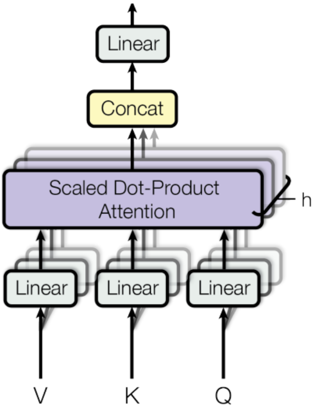


Vaswani *et al.*
Attention is all you need
Advances in neural information processing systems,2017

The MultiHeadAttention(MHA) Layer

- Three principal components: Value, Key, Query.
- Split into h parts and pass them through dense layers.
- Attention for each head :

$$\text{Attention} = \text{softmax}\left(\frac{QK^T}{\sqrt{dim_k}}\right)V,$$



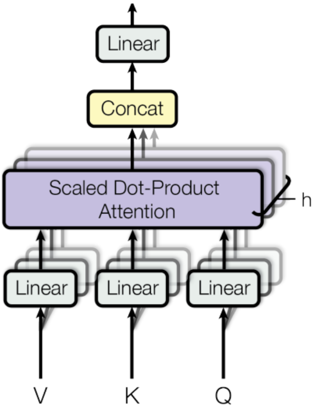
Vaswani *et al.*
Attention is all you need
Advances in neural information processing systems,2017

The MultiHeadAttention(MHA) Layer

- Three principal components: Value, Key, Query.
- Split into h parts and pass them through dense layers.
- Attention for each head :

$$\text{Attention} = \text{softmax}\left(\frac{QK^T}{\sqrt{dim_k}}\right)V,$$

- Concatenate and pass through a final dense layer.



Vaswani *et al.*
Attention is all you need
Advances in neural information processing systems,2017

Define the MHA layer at hand

```julia
struct MHA
    Q ::Dense
    K ::Dense
    V ::Dense
    D ::Dense
    nheads ::Int
end ✓
function MHA(q_dim,k_dim,v_dim,qkv_dim,
out_dim,nheads)
    @assert q_dim % nheads == 0
    @assert v_dim % nheads == 0
    @assert k_dim  == q_dim
    @assert qkv_dim % nheads == 0
    q_dim_h = q_dim ÷ nheads
    k_dim_h = k_dim ÷ nheads
    v_dim_h = v_dim ÷ nheads
    Q = Dense(q_dim_h,qkv_dim÷nheads)
    K = Dense(k_dim_h,qkv_dim÷nheads)
    V = Dense(v_dim_h,qkv_dim÷nheads)
    D = Dense(qkv_dim,out_dim÷nheads)
    MHA(Q,K,V,D,nheads)
end MHA
Flux.@functor MHA ✓
```

```julia
function (mha ::MHA)(q,k,v)
    Q,K,V,D = mha.Q,mha.K,mha.V,mha.D
    qh = reshape(q,size(q,1)÷mha.nheads,
    mha.nheads,size(q,2),size(q,3))
    kh = reshape(k,size(k,1)÷mha.nheads,
    mha.nheads,size(k,2),size(k,3))
    vh = reshape(v,size(v,1)÷mha.nheads,
    mha.nheads,size(v,2),size(v,3))

    qval = Q(qh)
    kval = K(kh)
    vval = V(vh)
    @tullio att[n,m,hi,b] := qval[i,hi,n,
    b] * kval[i,hi,m,b] / sqrt(size(kval,
    1))
    att = Flux.softmax(att)
    @tullio res[i,hi,n,b] := att[n,m,hi,
    b] * vval[i,hi,m,b]
    res = reshape(res,size(res,1)*size
    (res,2),size(v)[2:end]...)
    return D(res),att
end ✓
```

The MultiHeadAttention layer

```julia
x = rand(64,100,1000);

mha = MultiHeadAttention((64,64,64)
=>1024=>64,nheads = 4);

res,att= mha(x);

res | 64×100×1000 Array{Float32, 3}:
att | 100×100×4×1000 Array{Float32, 4}:
```

1 Introduction

2 Core Mechanisms

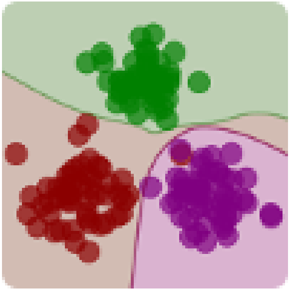
3 Flux Built-in Layers

4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

- A lot of "usual" datasets can be directly dowload within julia and MLDatasets.



- A lot of "usual" datasets can be directly dowload within julia and MLDatasets.
- We always want more, and we can get more from python (PythonCall.jl) or R (Rdatasets.jl).

JuliaML
MLDatasets.jl
https://github.com/JuliaML/MLDatasets.jl

JuliaPy
PythonCall.jl
https://github.com/JuliaPy/PythonCall.jl

JuliaStats
RDatasets.jl
https://github.com/JuliaStats/RDatasets.jl

Comparaison of including datasets

```
using MLDatasets: Iris ✓
iris = Iris(); ✓
datass = iris.features; ✓
target = iris.targets |> Array; ✓
target = reshape(target,length(target)); ✓
target = Dict("$(target[i])"=> target .== target[i] for i in
eachindex(target)) |> DataFrame 150×3 DataFrame
```

```
using PythonCall ✓

skl = PythonCall.pyimport("sklearn.datasets"); ✓
datass, target = skl.load_iris(return_X_y=true,
as_frame=true); ✓
using DataFrames ✓
k = datass.keys(); ✓
vv = pyconvert(Array,datass.values); ✓
datass = Dict([  pyconvert(String,k[i-1]) => vv[:,i]  for i
in axes(vv,2)]...) |> DataFrame 150×4 DataFrame

k= pyconvert(Array,target.keys()); ✓
target = pyconvert(Array,target.values); ✓
target = Dict([  "$(target[i])"=> target .== target[i]  for
i in eachindex(target)]...) |> DataFrame 150×3 DataFrame
```

```
using RDatasets ✓

iris = RDatasets.dataset("datasets",
"iris") 150×5 DataFrame

datass = iris[:,1:4] |> Array; ✓

target = iris[:,5] |> Array; ✓

target = Dict("$(target[i])"=> target .
== target[i] for i in eachindex
(target)) |> DataFrame 150×3 DataFrame
```

1 Introduction

2 Core Mechanisms

3 Flux Built-in Layers

4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

Showcase of CNN and Transformers

Code is available on `https://github.com/yolhan83`.

1 Introduction

2 Core Mechanisms

3 Flux Built-in Layers

4 Datasets For Deep Learning

5 Showcase of CNN and Transformers

6 Conclusion

Conclusion

- Flux is a deep learning library for Julia.

Conclusion

- Flux is a deep learning library for Julia.
- Functors, Zygote and Optimisers are the building blocks of Flux.

Conclusion

- Flux is a deep learning library for Julia.
- Functors, Zygote and Optimisers are the building blocks of Flux.
- Easy to make new layer, loss, optimiser, ect.

Conclusion

- Flux is a deep learning library for Julia.
- Functors, Zygote and Optimisers are the building blocks of Flux.
- Easy to make new layer, loss, optimiser, ect.
- Datasets are fully available in julia.

Conclusion

- Flux is a deep learning library for Julia.
- Functors, Zygote and Optimisers are the building blocks of Flux.
- Easy to make new layer, loss, optimiser, ect.
- Datasets are fully available in julia.
- Complex models can be written in a comprehensive way.

Conclusion

- Flux is a deep learning library for Julia.
- Functors, Zygote and Optimisers are the building blocks of Flux.
- Easy to make new layer, loss, optimiser, ect.
- Datasets are fully available in julia.
- Complex models can be written in a comprehensive way.
- Thank you all for your attention.