Julia
Neural nets
Parallelism
Parsing

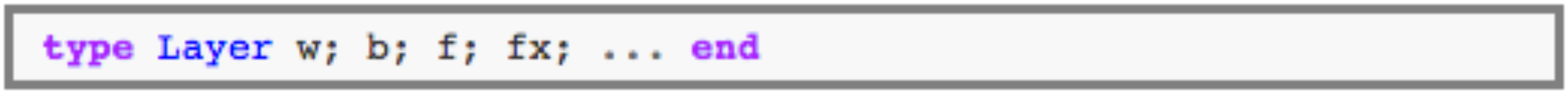
Deniz Yuret

June 16, 2015

Related links

- https://goo.gl/NvHHnL: these slides
- http://julia.readthedocs.org: Julia docs
- Beginning deep learning with 500 lines of Julia (v0.0)
- Beginning deep learning with 500 lines of Julia (v0.1)
- Parsing the Penn Treebank in 60 seconds
- https://github.com/denizyuret/KUnet.jl
- https://github.com/denizyuret/KUparser.jl

Layers and Nets in Julia



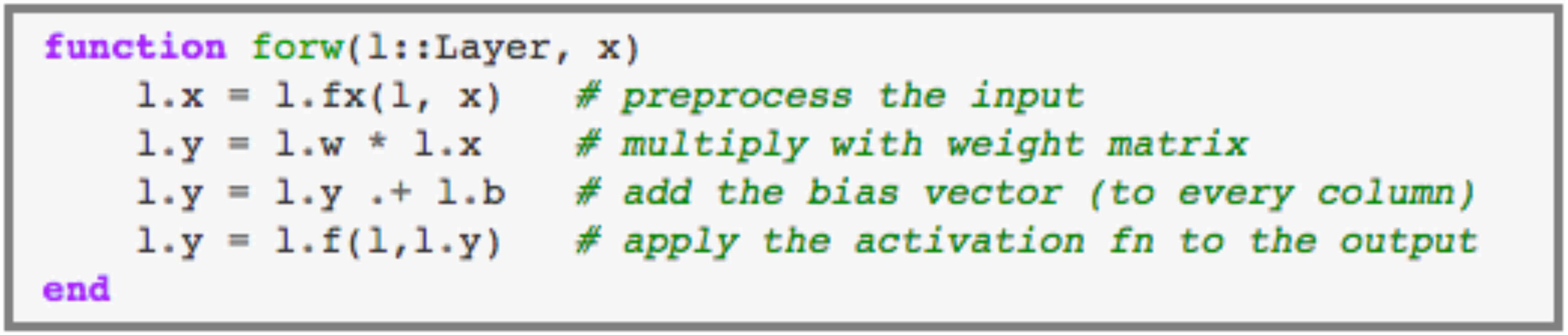
```julia
type Layer w; b; f; fx; ... end
```

Layers and Nets in Julia

```julia
type Layer w; b; f; fx; ... end
```

```julia
typealias Net Array{Layer,1}
```

Going forward



```
function forw(l::Layer, x)
    l.x = l.fx(l, x)    # preprocess the input
    l.y = l.w * l.x     # multiply with weight matrix
    l.y = l.y .+ l.b    # add the bias vector (to every column)
    l.y = l.f(l,l.y)    # apply the activation fn to the output
end
```

(*) Note that in the latest version of KUnet each step in forward (dropout, mmul, bias, activation) has been split into a separate layer.

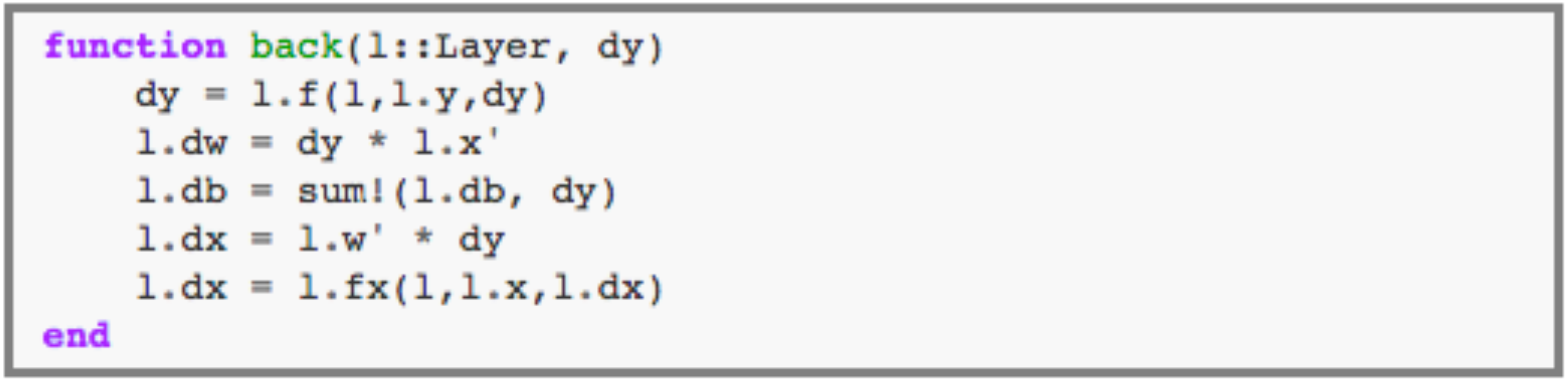
Going forward

```
function forw(l::Layer, x)
    l.x = l.fx(l, x)    # preprocess the input
    l.y = l.w * l.x     # multiply with weight matrix
    l.y = l.y .+ l.b    # add the bias vector (to every column)
    l.y = l.f(l,l.y)    # apply the activation fn to the output
end
```

```
forw(n::Net, x)=(for l=n x=forw(l,x) end; x)
```

(*) Note that in the latest version of KUnet each step in forward (dropout, mmul, bias, activation) has been split into a separate layer.

Going backward



```
function back(l::Layer, dy)
    dy = l.f(l,l.y,dy)
    l.dw = dy * l.x'
    l.db = sum!(l.db, dy)
    l.dx = l.w' * dy
    l.dx = l.fx(l,l.x,l.dx)
end
```

Going backward

```
function back(l::Layer, dy)
    dy = l.f(l,l.y,dy)
    l.dw = dy * l.x'
    l.db = sum!(l.db, dy)
    l.dx = l.w' * dy
    l.dx = l.fx(l,l.x,l.dx)
end
```

```
back(n::Net, dy) = (for i=length(n):-1:1 dy=back(n[i],dy) end)
```

Training with backpropagation

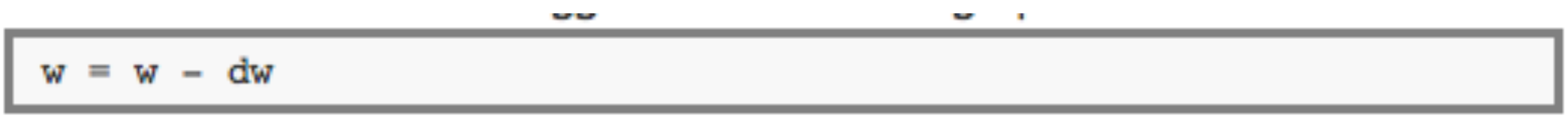
```
backprop(net, x, y)=(forw(net, x); back(net, y))
```

Training with backpropagation

```
backprop(net, x, y)=(forw(net, x); back(net, y))
```

```
train(net, x, y)=(backprop(net, x, y); update(net))
```

Updating weights



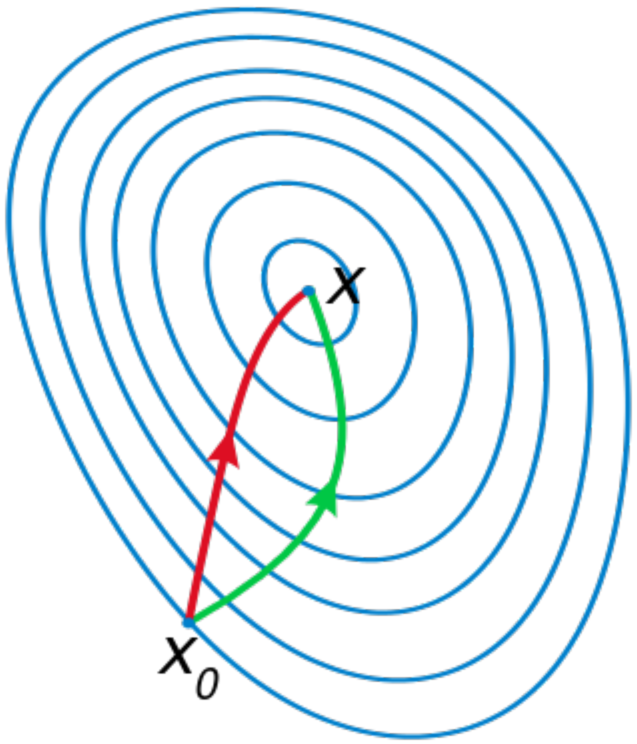
```
w = w - dw
```

Updating weights

```
w = w - dw
```

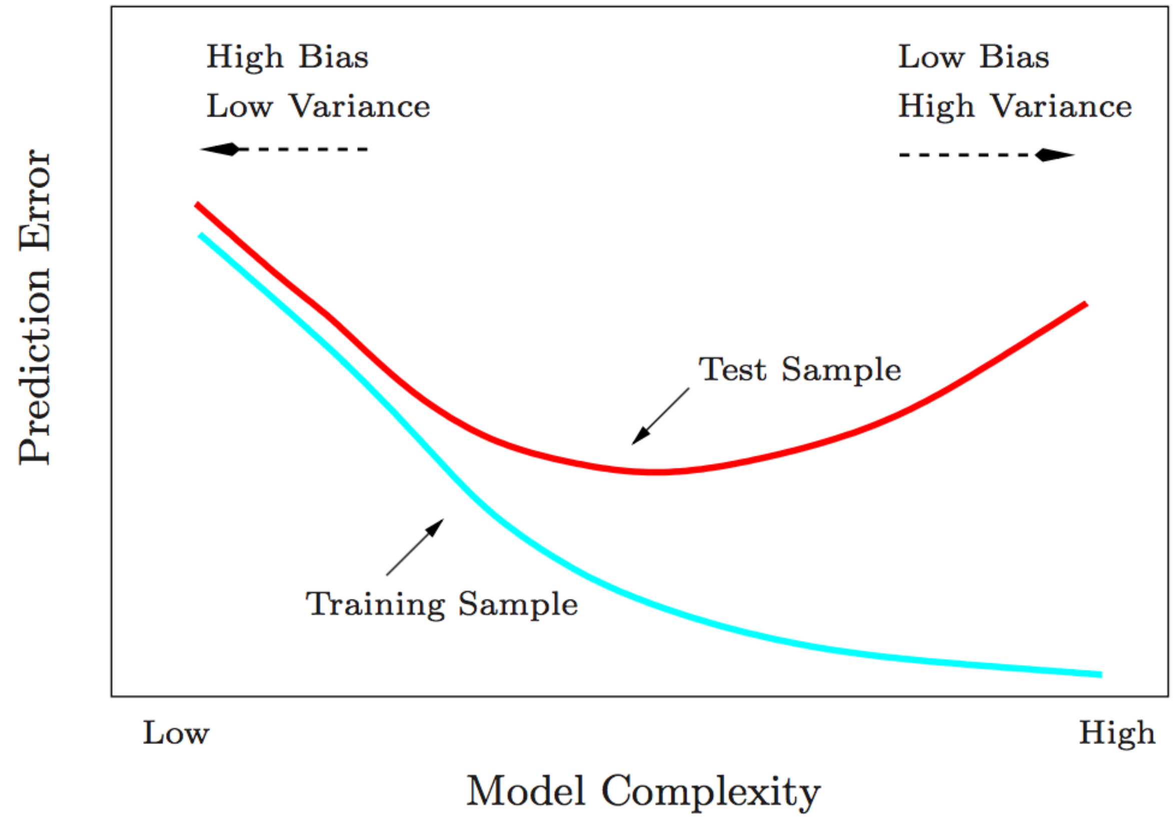
```
w = w - learningRate * dw
```

Updating weights



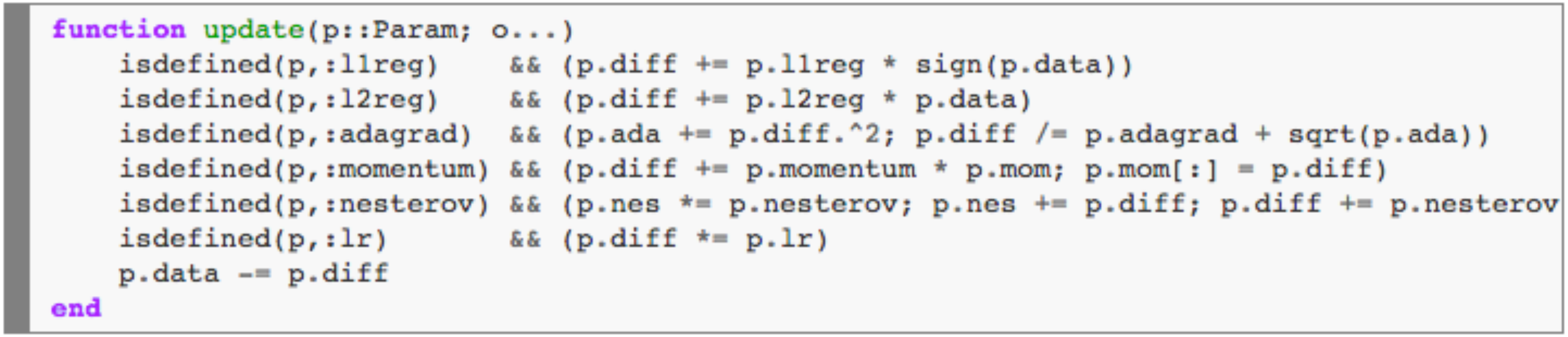
Which direction?

Updating weights



Which objective?

Updating weights



```julia
function update(p::Param; o...)
    isdefined(p,:l1reg)    && (p.diff += p.l1reg * sign(p.data))
    isdefined(p,:l2reg)    && (p.diff += p.l2reg * p.data)
    isdefined(p,:adagrad)  && (p.ada += p.diff.^2; p.diff /= p.adagrad + sqrt(p.ada))
    isdefined(p,:momentum) && (p.diff += p.momentum * p.mom; p.mom[:] = p.diff)
    isdefined(p,:nesterov) && (p.nes *= p.nesterov; p.nes += p.diff; p.diff += p.nesterov
    isdefined(p,:lr)       && (p.diff *= p.lr)
    p.data -= p.diff
end
```

KUnet usage example

Let us load mnist

```
julia> include(Pkg.dir("KUnet/test/mnist.jl"))
julia> using KUnet
julia> using MNIST: xtrn, ytrn, xtst, ytst
```

```
julia> xtrn, ytrn, xtst, ytst
(
784x60000 Array{Float32,2}: ...
10x60000 Array{Float32,2}: ...
784x10000 Array{Float32,2}: ...
10x10000 Array{Float32,2}: ...
)
```

Create a small net

```
net = [ Mmul(64,784), Bias(64), Relu(),
        Mmul(10,64),  Bias(10), XentLoss() ]
```

… and do some training

```
for i=1:100
    train(net, xtrn, ytrn; batch=128)
    println((i, accuracy(ytst, predict(net, xtst)),
                accuracy(ytrn, predict(net, xtrn))))
end
```

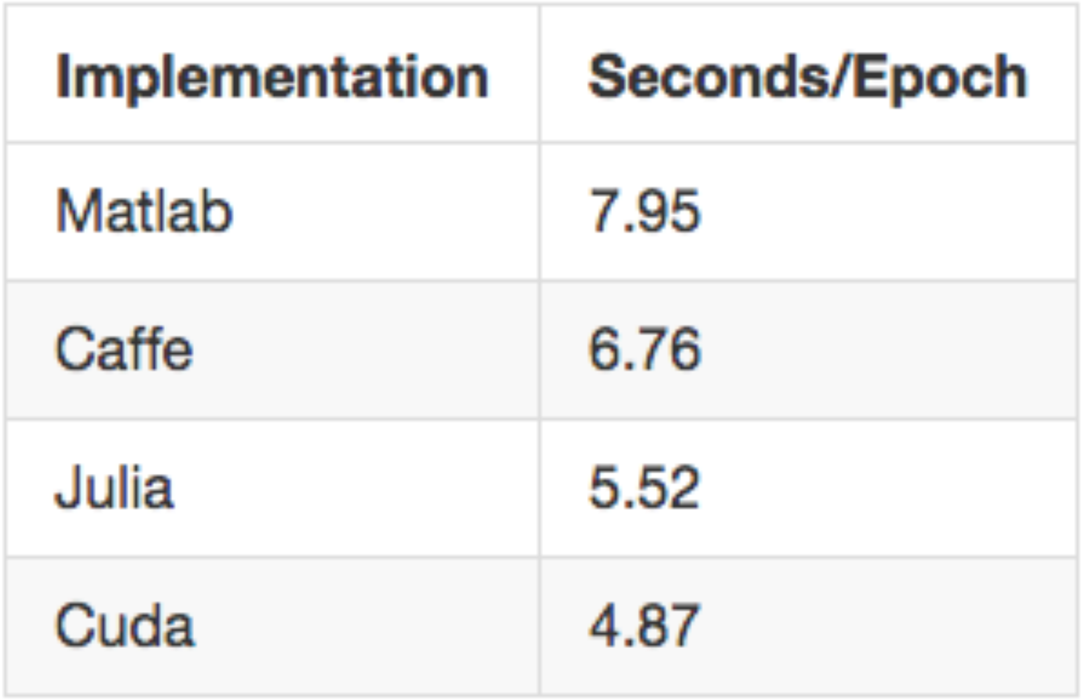
```
(1,0.9152,0.9171833333333334)
(2,0.9431,0.9440333333333333)
(3,0.959,0.9611666666666666)
...
(59,0.9772,0.9999833333333333)
(60,0.9773,1.0)
...
(100,0.9776,1.0)
elapsed time: 39.738191211 seconds (1526525108 bytes allocated, 3.05% gc time)
```

Here is LeNet (a convolutional net)

```
net = [Conv(5,5,1,20), Bias(20), Relu(), Pool(2),
       Conv(5,5,20,50), Bias(50), Relu(), Pool(2),
       Mmul(500,800), Bias(500), Relu(),
       Mmul(10,500), Bias(10), XentLoss()]
```

```
# same for loop
...
(100,0.9908,1.0)
elapsed time: 360.722851006 seconds (5875158944 bytes allocated, 1.95% gc time)
```

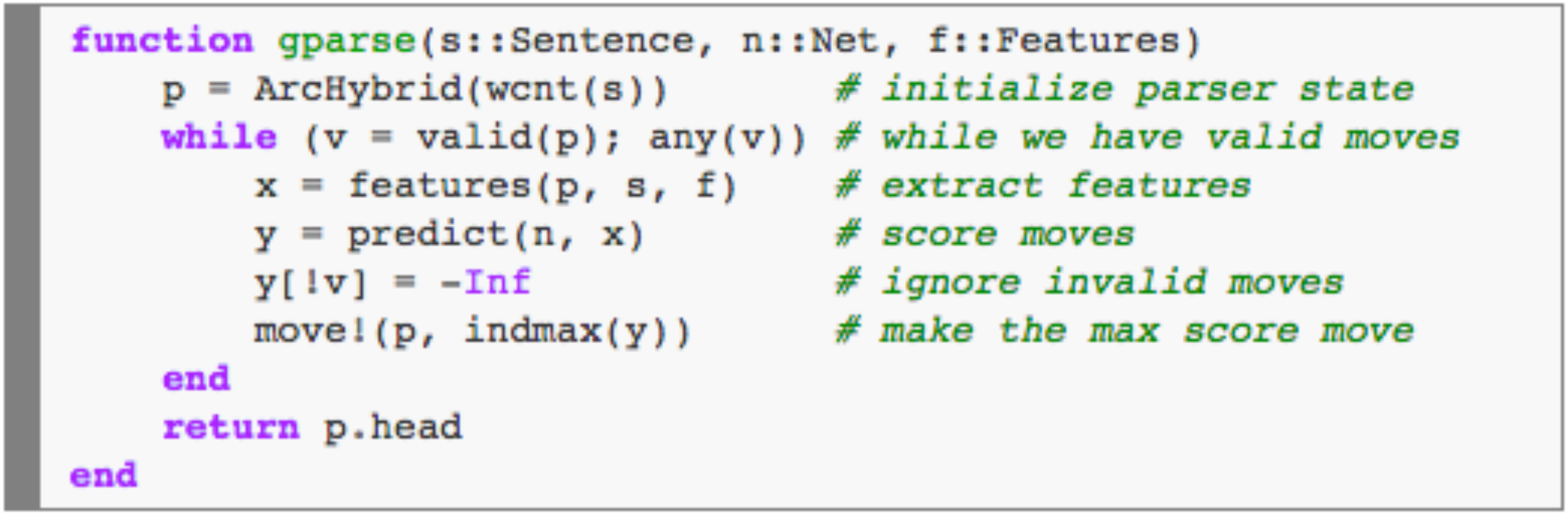
Performance comparison (GPU)



Implementation	Seconds/Epoch
Matlab	7.95
Caffe	6.76
Julia	5.52
Cuda	4.87

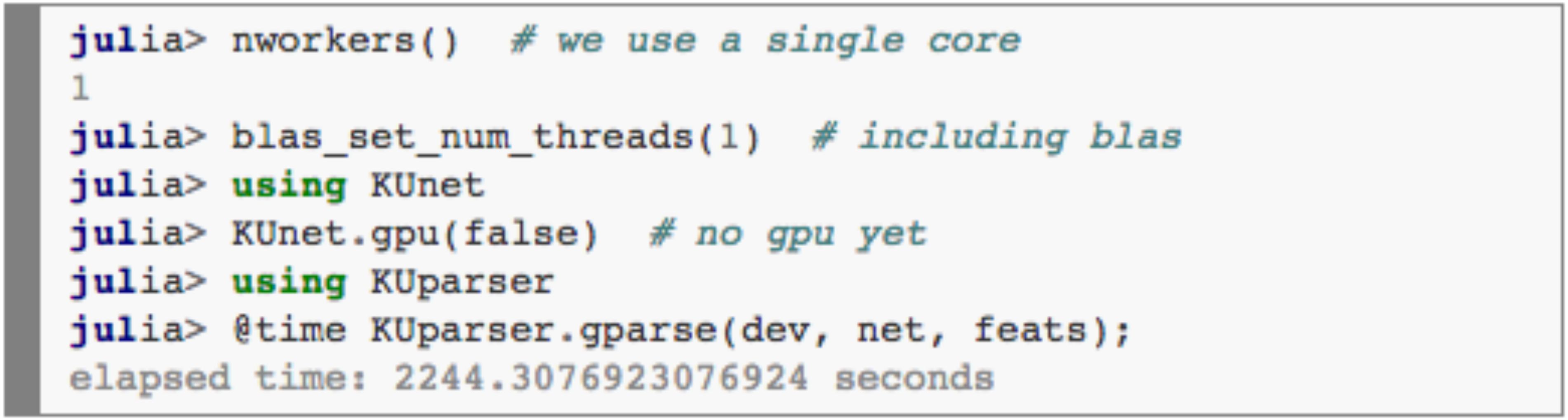
Parsing
Parallelism in Julia

Greedy transition based dependency parsing in Julia



```
function gparse(s::Sentence, n::Net, f::Features)
    p = ArcHybrid(wcnt(s))          # initialize parser state
    while (v = valid(p); any(v)) # while we have valid moves
        x = features(p, s, f)     # extract features
        y = predict(n, x)         # score moves
        y[!v] = -Inf              # ignore invalid moves
        move!(p, indmax(y))       # make the max score move
    end
    return p.head
end
```

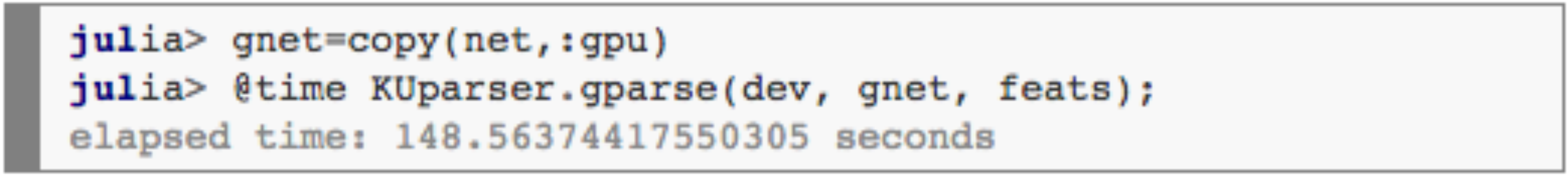
Timing on a single CPU core



```
julia> nworkers()  # we use a single core
1
julia> blas_set_num_threads(1)  # including blas
julia> using KUnet
julia> KUnet.gpu(false)  # no gpu yet
julia> using KUparser
julia> @time KUparser.gparse(dev, net, feats);
elapsed time: 2244.3076923076924 seconds
```

56 ms/word (>99% spent on predict)

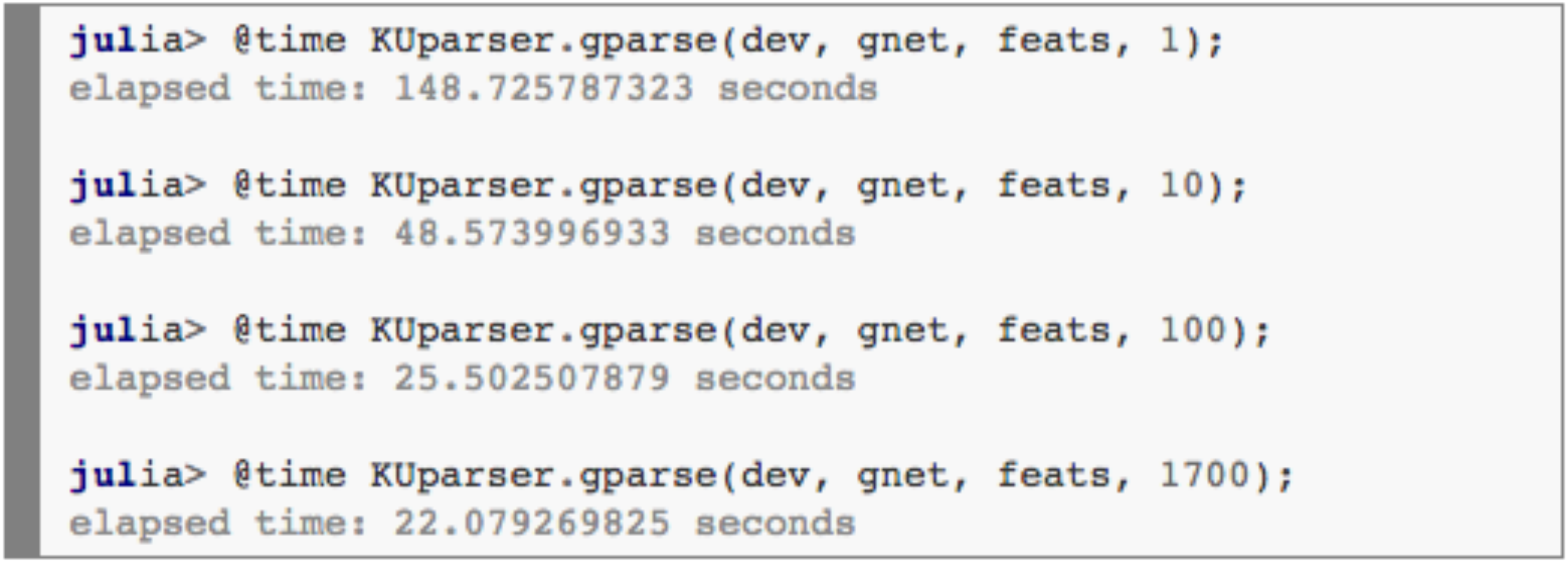
Using the GPU



```
julia> gnet=copy(net,:gpu)
julia> @time KUparser.gparse(dev, gnet, feats);
elapsed time: 148.56374417550305 seconds
```

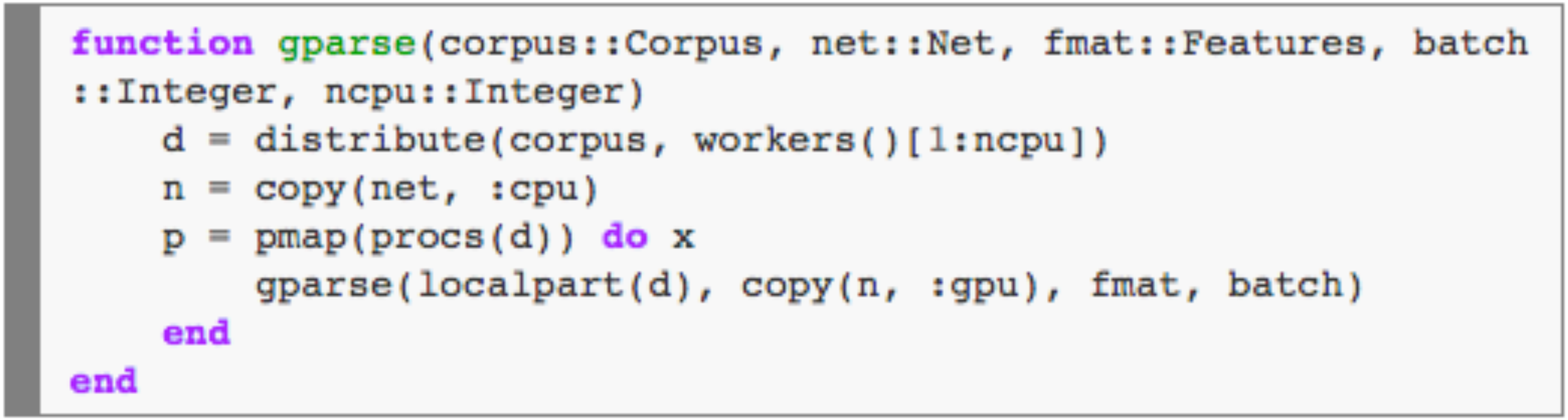
3.7 ms/word (15x speed-up)

Batching the input


```
julia> @time KUparser.gparse(dev, gnet, feats, 1);
elapsed time: 148.725787323 seconds

julia> @time KUparser.gparse(dev, gnet, feats, 10);
elapsed time: 48.573996933 seconds

julia> @time KUparser.gparse(dev, gnet, feats, 100);
elapsed time: 25.502507879 seconds

julia> @time KUparser.gparse(dev, gnet, feats, 1700);
elapsed time: 22.079269825 seconds
```

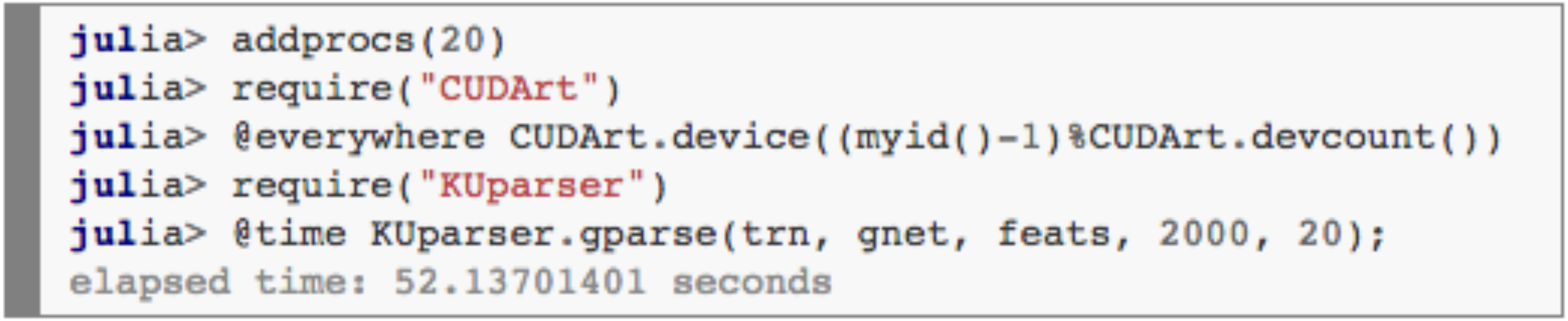
0.55 ms/word (100x speed-up)
prediction 25%, features 75%

Using multiple CPUs



```
function gparse(corpus::Corpus, net::Net, fmat::Features, batch
::Integer, ncpu::Integer)
    d = distribute(corpus, workers()[1:ncpu])
    n = copy(net, :cpu)
    p = pmap(procs(d)) do x
        gparse(localpart(d), copy(n, :gpu), fmat, batch)
    end
end
```

Parsing the Penn Treebank in 52 secs



```
julia> addprocs(20)
julia> require("CUDArt")
julia> @everywhere CUDArt.device((myid()-1)%CUDArt.devcount())
julia> require("KUparser")
julia> @time KUparser.gparse(trn, gnet, feats, 2000, 20);
elapsed time: 52.13701401 seconds
```

0.055 ms/word (1000x speed-up)
on 2x K40 GPU + 20 CPU cores