Knet.jl Documentation

Release 0.7.2

Deniz Yuret

February 10, 2017

Contents

Contents:

CHAPTER 1

Setting up Knet

Knet.jl is a deep learning package implemented in Julia, so you should be able to run it on any machine that can run Julia. It has been extensively tested on Linux machines with NVIDIA GPUs and CUDA libraries, but most of it works on vanilla Linux and OSX machines as well (currently cpu-only support for some operations is incomplete). If you would like to try it on your own computer, please follow the instructions on *Installation*. If you would like to try working with a GPU and do not have access to one, take a look at *Using Amazon AWS*. If you find a bug, please open a GitHub issue. If you would like to contribute to Knet, see *Tips for developers*. If you need help, or would like to request a feature, please consider joining the knet-users mailing list.

1.1 Installation

First download and install the latest version of Julia from http://julialang.org/downloads. As of this writing the latest version is 0.4.6 and I have tested Knet using 64-bit Generic Linux binaries and the Mac OS X package (dmg). Once Julia is installed, type `julia` at the command prompt to start the Julia interpreter. and type `Pkg.add("Knet")` to install Knet.

$ julia

_
_ _ _(_)_ | A fresh approach to technical computing

(_) | (_) (_) | **Documentation: http://docs.julialang.org** _ _ _| |_ __ _ | Type "?help" for help.

| | | | | |/ _` | |
| |_| | | | (_| | | Version 0.4.5 (2016-03-18 00:58 UTC)

_/ |__'_|_|_|__'_| | Official http://julialang.org/ release

|__/ | x86_64-apple-darwin13.4.0

julia> Pkg.add("Knet")

By default Knet only installs the minimum requirements. Some examples use extra packages like ArgParse, GZip and JLD. GPU support requires the packages CUDArt, CUBLAS, CUDNN and CUSPARSE (0.3). These extra packages can be installed using additional `Pkg.add()` commands. If you have a GPU machine, you may need to type `Pkg.build("Knet")` to compile the Knet GPU kernels. If you do not have a GPU machine, you don't need `Pkg.build` but you may get some warnings indicating the lack of GPU support. Usually, these can be safely ignored. To make sure everything has installed correctly, type `Pkg.test("Knet")` which should take a couple of

minutes kicking the tires. If all is OK, continue with the next section, if not you can get help at the knet-users mailing list.

1.2 Tips for developers

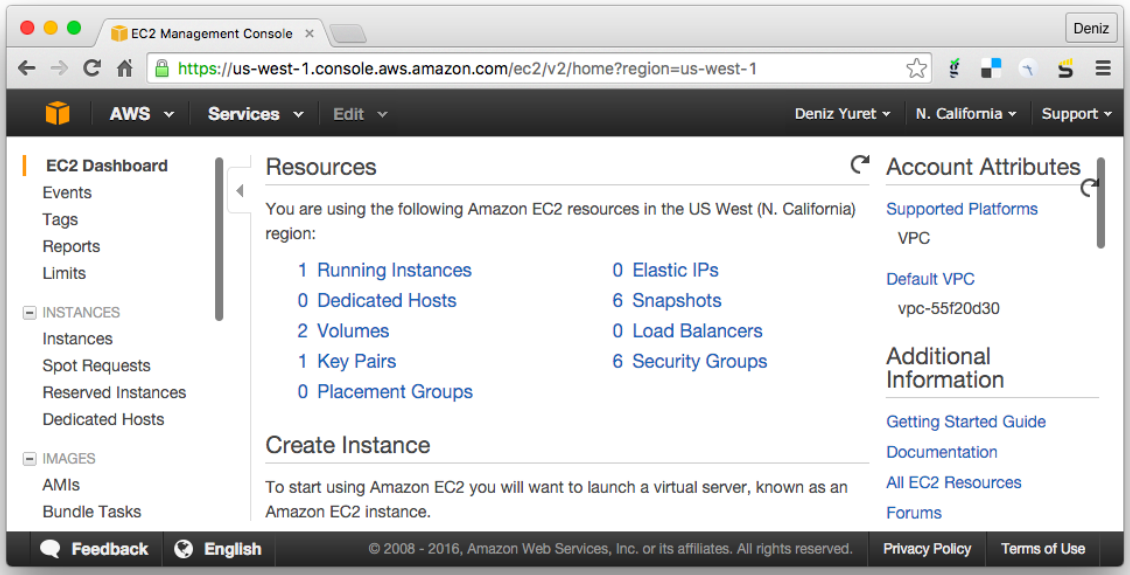
Knet is an open-source project and we are always open to new contributions: bug fixes, new machine learning models and operators, inspiring examples, benchmarking results are all welcome. If you'd like to contribute to the code base, here are some tips:

- Please get an account at github.com.
- Fork the Knet repository.
- Point Julia to your fork using `Pkg.clone("git@github.com:your-username/Knet.jl.git")` and `Pkg.build("Knet")`. You may want to remove any old versions with `Pkg.rm("Knet")` first.
- Make sure your fork is up-to-date.
- Retrieve the latest version of the master branch using `Pkg.checkout("Knet")`.
- Implement your contribution.
- Test your code using `Pkg.test("Knet")`.
- Please submit your contribution using a pull request.

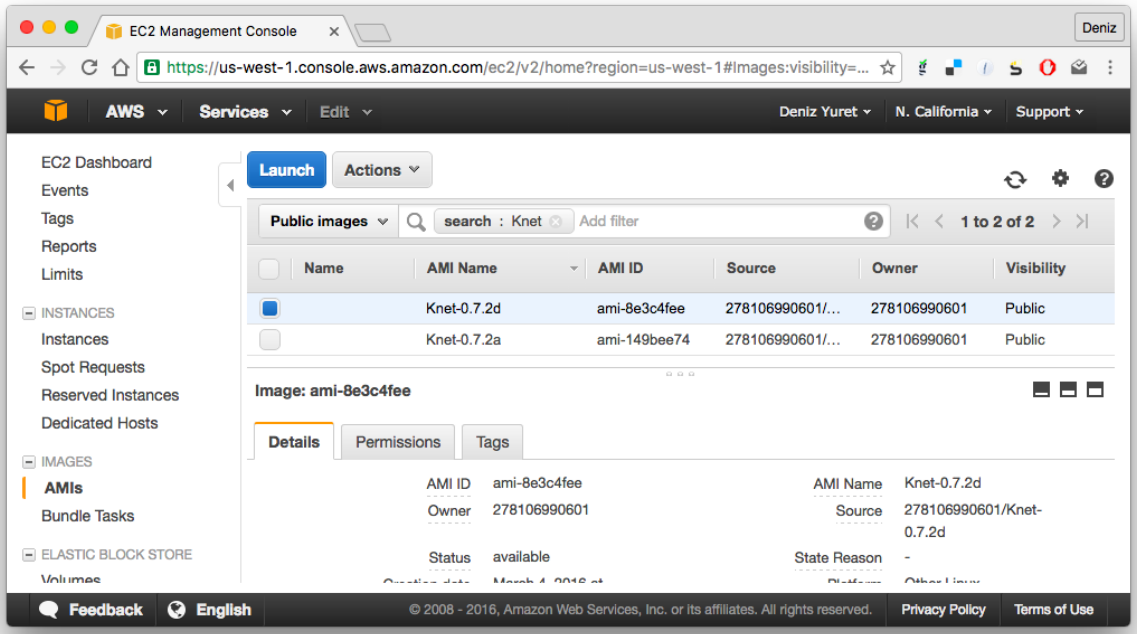
1.3 Using Amazon AWS

If you don't have access to a GPU machine, but would like to experiment with one, Amazon Web Services is a possible solution. I have prepared a machine image (AMI) with everything you need to run Knet. Here are step by step instructions for launching a GPU instance with a Knet image:

1. First, you need to sign up and create an account following the instructions on Setting Up with Amazon EC2. Once you have an account, open the Amazon EC2 console at https://console.aws.amazon.com/ec2 and login. You should see the following screen:



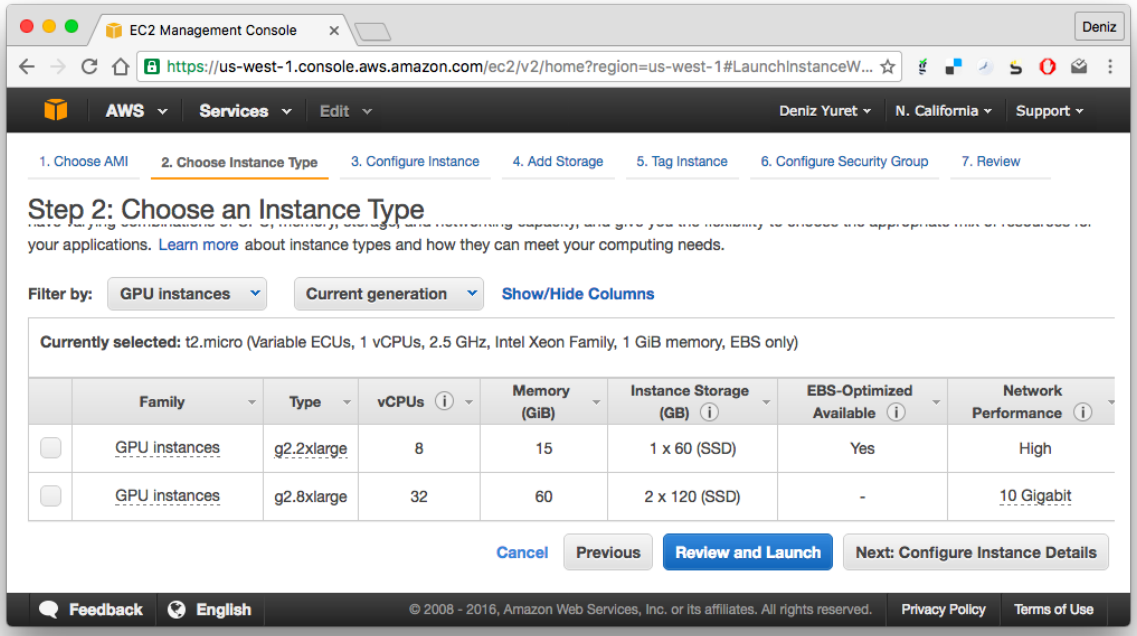
2. Make sure you select the "N. California" region in the upper right corner, then click on AMIs on the lower left menu. At the search box, choose "Public images" and search for "Knet". Click on the latest Knet image (Knet-0.7.2d as of this writing). You should see the following screen with information about the Knet AMI. Click on the "Launch" button on the upper left.



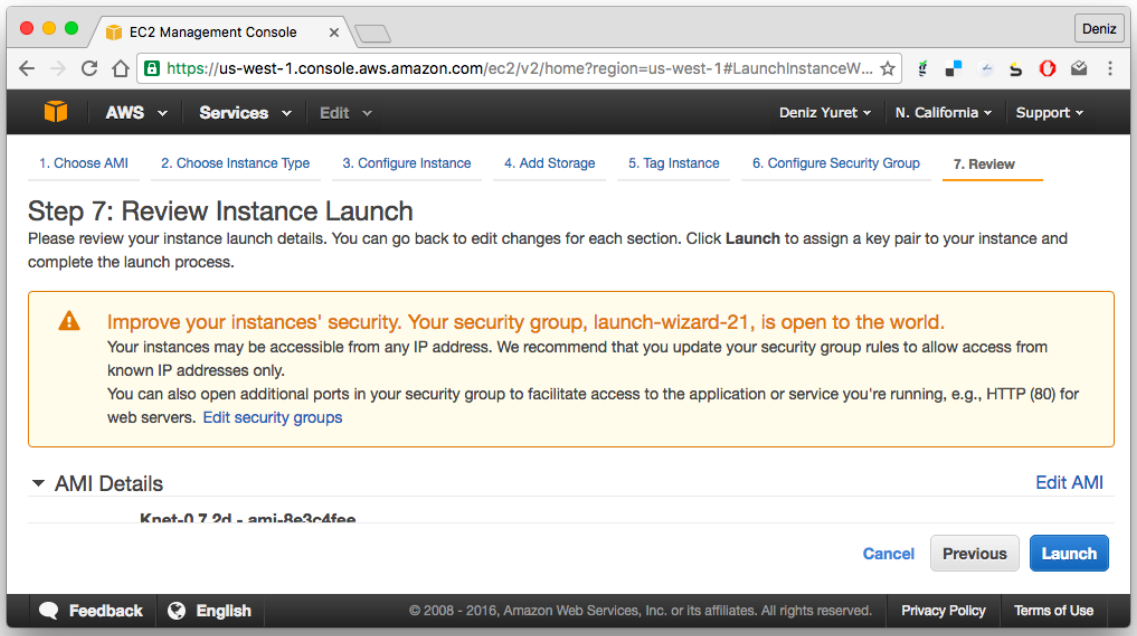
Note: Instead of "Launch", you may want to experiment with "Spot Request" under "Actions" to get a lower price. You may also qualify for an educational grant if you are a student or researcher.

3. You should see the "Step 2: Choose an Instance Type" page. Next to "Filter by:" change "All instance types" to

"GPU instances". This should reduce the number of instance types displayed to a few. Pick the "g2.2xlarge" instance ("g2.8xlarge" has multiple GPUs and is more expensive) and click on "Review and Launch".

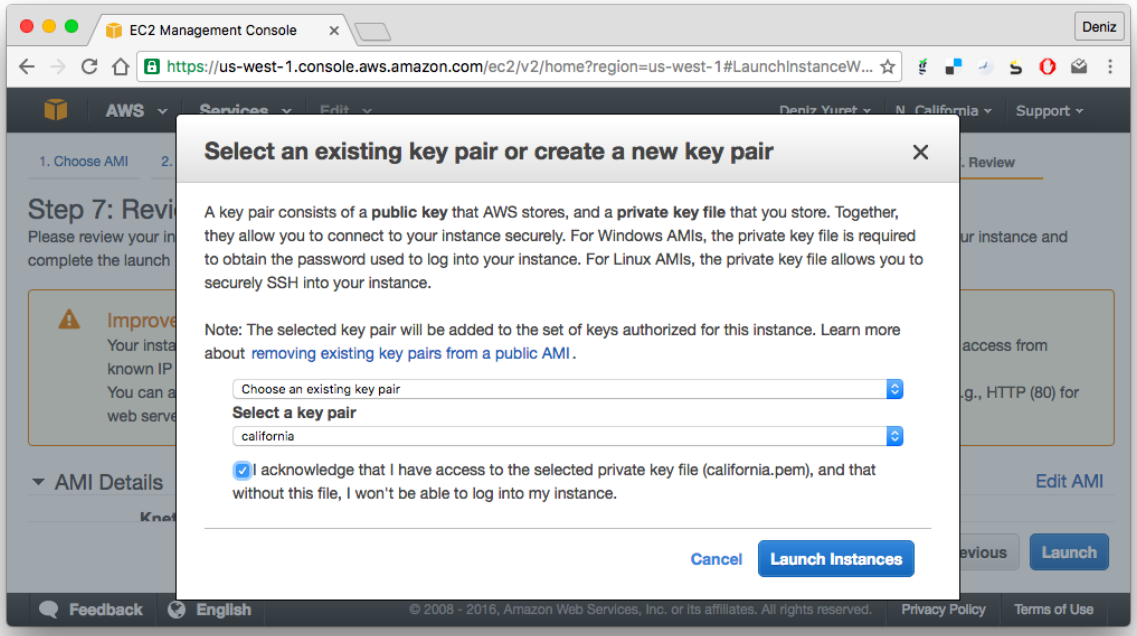


4. This should take you to the "Step 7: Review Instance Launch" page. You can just click "Launch" here:

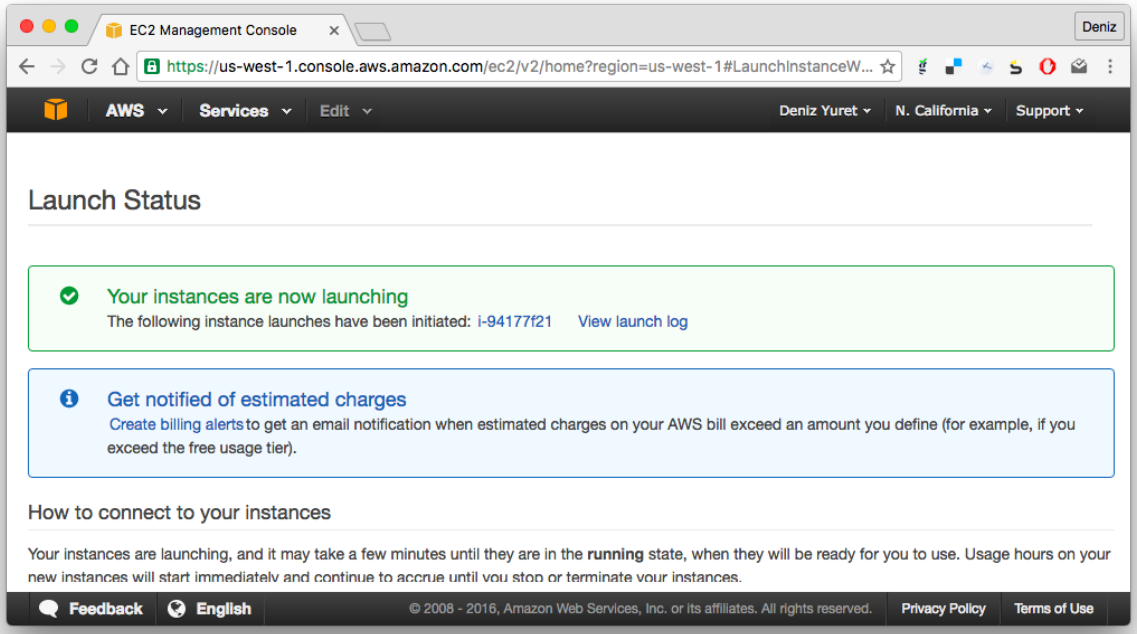


5. You should see the "key pair" pop up menu. In order to login to your instance, you need an ssh key pair. If you have created a pair during the initial setup you can use it with "Choose an existing key pair". Otherwise pick "Create a new key pair" from the pull down menu, enter a name for it, and click "Download Key Pair". Make sure you keep

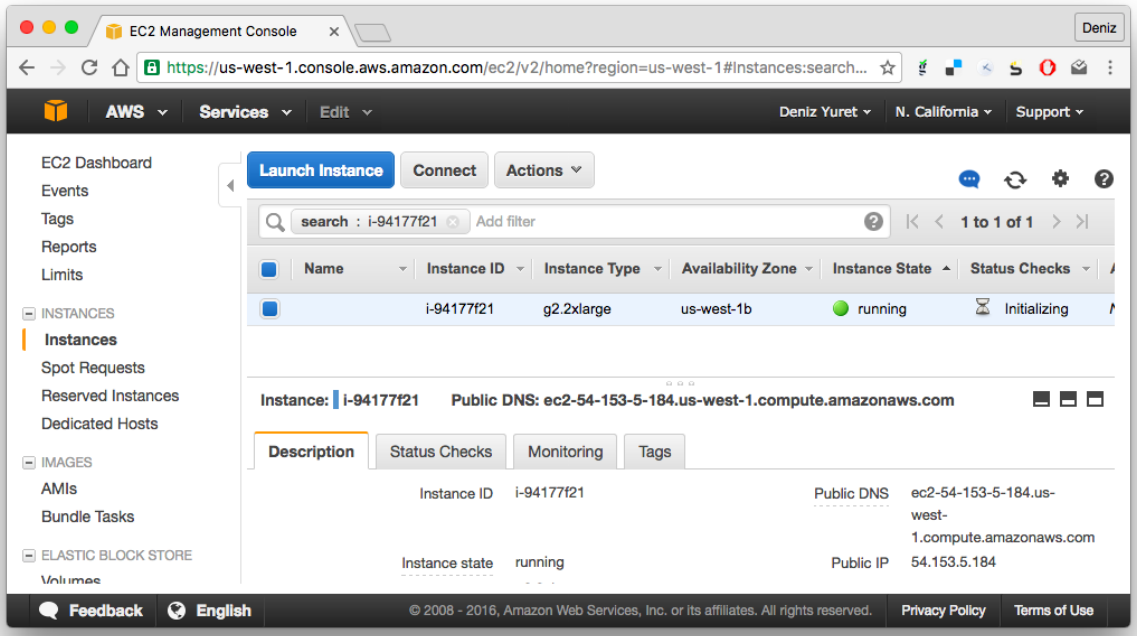
the downloaded file, we will use it to login. After making sure you have the key file (it has a .pem extension), click "Launch Instances" on the lower right.



6. We have completed the request. You should see the "Launch Status" page. Click on your instance id under "Your instances are launching":



7. You should be taken to the "Instances" screen and see the address of your instance where it says something like "Public DNS: ec2-54-153-5-184.us-west-1.compute.amazonaws.com".



8. Open up a terminal (or Putty if you are on Windows) and type:

```
ssh -i knetkey.pem ec2-user@ec2-54-153-5-184.us-west-1.compute.amazonaws.com
```

Replacing `knetkey.pem` with the path to your key file and `ec2-54-153-5-184` with the address of your machine. If all goes well you should get a shell prompt on your machine instance.

9. There you can type `julia`, and at the julia prompt `Pkg.update()` and `Pkg.build("Knet")` to get the latest versions of the packages, as the versions in the AMI may be out of date:

```
[ec2-user@ip-172-31-6-90 ~]$ julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.4.2 (2015-12-06 21:47 UTC)
 _/ |\__'_|_|_|\__'_|  |  Official http://julialang.org/ release
|__/                   |  x86_64-unknown-linux-gnu

WARNING: Terminal not fully functional
julia> Pkg.update()
julia> Pkg.build("Knet")
```

Finally you can run `Pkg.test("Knet")` to make sure all is good. This should take about a minute. If all tests pass, you are ready to work with Knet:

```
julia> Pkg.test("Knet")
INFO: Testing Knet
INFO: Simple linear regression example
...
INFO: Knet tests passed
```

```
julia>
```

CHAPTER 2

A Tutorial Introduction

We will begin by a quick tutorial on Knet, going over the essential tools for defining, training, and evaluating real machine learning models in 10 short lessons. The examples cover linear regression, softmax classification, multilayer perceptrons, convolutional and recurrent neural networks. We will use these models to predict housing prices in Boston, recognize handwritten digits, and teach the computer to write like Shakespeare!

The goal is to get you to the point where you can create your own models and apply machine learning to your own problems as quickly as possible. So some of the details and exceptions will be skipped for now. No prior knowledge of machine learning or Julia is necessary, but general programming experience will be assumed. It would be best if you follow along with the examples on your computer. Before we get started please complete the *installation instructions* if you have not done so already.

2.1 1. Functions and models

See also:

@knet, function, compile, forw, get, :colon

In this section, we will create our first Knet model, and learn how to make predictions. To start using Knet, type `using Knet` at the Julia prompt:

```
julia> using Knet
...
```

In Knet, a machine learning model is defined using a special function syntax with the `@knet` macro. It may be helpful at this point to review the Julia function syntax as the Knet syntax is based on it. The following example defines a @knet function for a simple linear regression model with 13 inputs and a single output. You can type this definition at the Julia prompt, or you can copy and paste it into a file which can be loaded into Julia using `include("filename")`:

```
@knet function linreg(x)
    w = par(dims=(1,13), init=Gaussian(0,0.1))
    b = par(dims=(1,1),  init=Constant(0))
    return w * x .+ b
end
```

In this definition:

- `@knet` indicates that `linreg` is a Knet function, and not a regular Julia function or variable.
- `x` is the only input argument. We will use a `(13,1)` column vector for this example.
- `w` and `b` are model parameters as indicated by the `par` constructor.

- `dims` and `init` are keyword arguments to `par`.
- `dims` gives the dimensions of the parameter. Julia stores arrays in column-major order, i.e. `(1,13)` specifies 1 row and 13 columns.
- `init` describes how the parameter should be initialized. It can be a user supplied Julia array or one of the supported *array fillers* as in this example.
- The final `return` statement specifies the output of the Knet function.
- The `*` denotes matrix product and `.+` denotes elementwise broadcasting addition.
- Broadcasting operations like `.+` can act on arrays with different sizes, such as adding a vector to each column of a matrix. They expand singleton dimensions in array arguments to match the corresponding dimension in the other array without using extra memory, and apply the operation elementwise.
- Unlike regular Julia functions, only a *restricted set of operators* such as `*` and `.+`, and statement types such as assignments and returns can be used in a @knet function definition.

In order to turn `linreg` into a machine learning model that can be trained with examples and used for predictions, we need to compile it:

```
julia> f1 = compile(:linreg)        # The colon before linreg is required
...
```

To test our model let's give it some input initialized with random numbers:

```
julia> x1 = randn(13,1)
13x1 Array{Float64,2}:
 -0.556027
 -0.444383
  ...
```

To obtain the prediction of model `f1` on input `x1` we use the `forw` function, which basically calculates `w * x1 .+ b`:

```
julia> forw(f1,x1)
1x1 Array{Float64,2}:
 -0.710651
```

We can query the model and see its parameters using `get`:

```
julia> get(f1,:w)           # The colon before w is required
1x13 Array{Float64,2}:
 0.149138  0.0367563  ... -0.433747  0.0569829

julia> get(f1,:b)
1x1 Array{Float64,2}:
 0.0
```

We can also look at the input with `get(f1,:x)`, reexamine the output using the special `:return` symbol with `get(f1,:return)`. In fact using `get`, we can confirm that our model gives us the same answer as an equivalent Julia expression:

```
julia> get(f1,:w) * get(f1,:x) .+ get(f1,:b)
1x1 Array{Float64,2}:
 -0.710651 DBG
```

You can see the internals of the compiled model looking at `f1`. It consists of 5 low level operations:

```
julia> f1
1 Knet.Input() name=>x,dims=>(13,1),norm=>3.84375,...
2 Knet.Par() name=>w,dims=>(1,13),norm=>0.529962,...
```

```
3 Knet.Par() name=>b,dims=>(1,1),norm=>0 ,...
4 Knet.Dot(2,1) name=>##tmp#7298,args=>(w,x),dims=>(1,1),norm=>0.710651,...
5 Knet.Add(4,3) name=>return,args=>(##tmp#7298,b),dims=>(1,1),norm=>0.710651,...
```

You may have noticed the colons before Knet variable names like `:linreg`, `:w`, `:x`, `:b`, etc. Any variable introduced in a @knet macro is not a regular Julia variable so its name needs to be escaped using the colon character in ordinary Julia code. In contrast, `f1` and `x1` are ordinary Julia variables.

In this section, we have seen how to create a Knet model by compiling a @knet function, how to perform a prediction given an input using `forw`, and how to take a look at model parameters using `get`. Next we will see how to train models.

2.2 2. Training a model

See also:

back, update!, setp, lr, quadloss

OK, so we can define functions using Knet but why should we bother? The thing that makes a Knet function different from an ordinary function is that Knet functions are **differentiable programs**. This means that for a given input not only can they compute an output, but they can also compute which way their parameters should be modified to approach some desired output. If we have some input-output data that comes from an unknown function, we can train a Knet model to look like this unknown function by manipulating its parameters.

We will use the Housing dataset from the UCI Machine Learning Repository to train our `linreg` model. The dataset has housing related information for 506 neighborhoods in Boston from 1978. Each neighborhood has 14 attributes, the goal is to use the first 13, such as average number of rooms per house, or distance to employment centers, to predict the 14'th attribute: median dollar value of the houses. Here are the first 3 entries:

```
0.00632  18.00   2.310  0  0.5380  6.5750  65.20  4.0900   1  296.0  15.30 396.90   4.98  24.00
0.02731   0.00   7.070  0  0.4690  6.4210  78.90  4.9671   2  242.0  17.80 396.90   9.14  21.60
0.02729   0.00   7.070  0  0.4690  7.1850  61.10  4.9671   2  242.0  17.80 392.83   4.03  34.70
...
```

Let's download the dataset and use `readdlm` to turn it into a Julia array.

```
julia> url = "https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data";
julia> file = Pkg.dir("Knet/data/housing.data");
julia> download(url, file)
  ...
julia> data = readdlm(file)'  # Don't forget the final apostrophe to transpose data
14x506 Array{Float64,2}:
   0.00632    0.02731    0.02729 ...   0.06076    0.10959    0.04741
  18.0        0.0        0.0     ...   0.0        0.0        0.0
   ...
```

The resulting `data` matrix should have 506 columns representing neighborhoods, and 14 rows representing the attributes. The last attribute is the median house price to be predicted, so let's separate it:

```
julia> x = data[1:13,:]
13x506 Array{Float64,2}:...
julia> y = data[14,:]
1x506 Array{Float64,2}:...
```

Here we are using Julia's array indexing notation to split the `data` array into input `x` and output `y`. Inside the square brackets `1:13` means grab the rows 1 through 13, and the `:` character by itself means grab all the columns.

You may have noticed that the input attributes have very different ranges. It is usually a good idea to normalize them by subtracting the mean and dividing by the standard deviation:

```
julia> x = (x .- mean(x,2)) ./ std(x,2);
```

The `mean()` and `std()` functions compute the mean and standard deviation of `x`. Their optional second argument gives the dimensions to sum over, so `mean(x)` gives us the mean of the whole array, `mean(x,1)` gives the mean of each column, and `mean(x,2)` gives us the mean of each row.

It is also a good idea to split our dataset into training and test subsets so we can estimate how well our model will do on unseen data.

```
julia> n = size(x,2);
julia> r = randperm(n);
julia> xtrn=x[:,r[1:400]];
julia> ytrn=y[:,r[1:400]];
julia> xtst=x[:,r[401:end]];
julia> ytst=y[:,r[401:end]];
```

`n` is set to the number of instances (columns) and `r` is set to `randperm(n)` which gives a random permutation of integers $1 \ldots n$. The first 400 indices in `r` will be used for training, and the last 106 for testing.

Let's see how well our randomly initialized model does before training:

```
julia> ypred = forw(f1, xtst)
1x106 Array{Float64,2}:...
julia> quadloss(ypred, ytst)
307.9336...
```

The quadratic *loss function* `quadloss()` computes $(1/2n)\sum(\hat{y}-y)^2$, i.e. half of the mean squared difference between a predicted answer $\hat{y}$ and the desired answer y. Given that y values range from 5 to 50, an RMSD of $\sqrt{2 \times 307.9} = 24.8$ is a pretty bad score.

We would like to minimize this loss which should get the predicted answers closer to the desired answers. To do this we first compute the loss gradient for the parameters of `f1` – this is the direction in parameter space that maximally increases the loss. Then we move the parameters in the opposite direction. Here is a simple function that performs these steps:

```
function train(f, x, y)
    for i=1:size(x,2)
        forw(f, x[:,i])
        back(f, y[:,i], quadloss)
        update!(f)
    end
end
```

- The `for` loop grabs training instances one by one.
- `forw` computes the prediction for the i'th instance. This is required for the next step.
- `back` computes the loss gradient for each parameter in `f` for the i'th instance.
- `update!` moves each parameter opposite the gradient direction to reduce the loss.

Before training, it is important to set a good learning rate. The learning rate controls how large the update steps are going to be: too small and you'd wait for a long time, too large and `train` may never converge. The `setp()` function is used to set *training options* like the learning rate. Let's set the learning rate to 0.001 and train the model for 100 epochs (i.e. 100 passes over the dataset):

```
julia> setp(f1, lr=0.001)
julia> for i=1:100; train(f1, xtrn, ytrn); end
```

This should take a few seconds, and this time our RMSD should be much better:

```
julia> ypred = forw(f1, xtst)
1x106 Array{Float64,2}:...
julia> quadloss(ypred,ytst)
11.5989...
julia> sqrt(2*ans)
4.8164...
```

We can see what the model has learnt looking at the new weights:

```
julia> get(f1,:w)
1x13 Array{Float64,2}:
 -0.560346  0.924687  0.0446596  ...  -1.89473  1.13219  -3.51418 DBG
```

The two weights with the most negative contributions are 13 and 8. We can find out from UCI that these are:

```
13. LSTAT: % lower status of the population
 8. DIS: weighted distances to five Boston employment centres
```

And the two with the most positive contributions are 9 and 6:

```
9. RAD: index of accessibility to radial highways
6. RM: average number of rooms per dwelling
```

In this section we saw how to download data, turn it into a Julia array, normalize and split it into input, output, train, and test subsets. We wrote a simple training script using `forw`, `back`, and `update!`, set the learning rate `lr` using `setp`, and evaluated the model using the `quadloss` loss function. Now, there are a lot more efficient and elegant ways to perform and analyze a linear regression as you can find out from any decent statistics text. However the basic method outlined in this section has the advantage of being easy to generalize to models that are a lot larger and complicated.

2.3 3. Making models generic

See also:

keyword arguments, size inference

Hardcoding the dimensions of parameters in `linreg` makes it awfully specific to the Housing dataset. Knet allows keyword arguments in @knet function definitions to get around this problem:

```
@knet function linreg2(x; inputs=13, outputs=1)
    w = par(dims=(outputs,inputs), init=Gaussian(0,0.1))
    b = par(dims=(outputs,1), init=Constant(0))
    return w * x .+ b
end
```

Now we can use this model for another dataset that has, for example, 784 inputs and 10 outputs by passing these keyword arguments to `compile`:

```
julia> f2 = compile(:linreg2, inputs=784, outputs=10);
```

Knet functions borrow the syntax for keyword arguments from Julia, and we will be using them in many contexts, so a brief aside is in order: Keyword arguments are identified by name instead of position, and they can be passed in any order (or not passed at all) following regular (positional) arguments. In fact we have already seen examples: `dims` and `init` are keyword arguments for `par` (which has no regular arguments). Functions with keyword arguments are defined using a semicolon in the signature, e.g. `function pool(x; window=2, padding=0)`. The semicolon

is optional when the function is called, e.g. both `pool(x, window=5)` or `pool(x; window=5)` work. Unspecified keyword arguments take their default values specified in the function definition. Extra keyword arguments can be collected using three dots in the function definition: `function pool(x; window=2, padding=0, o...)`, and passed in function calls: `pool(x; o...)`.

In addition to keyword arguments to make models more generic, Knet implements **size inference**: Any dimension that relies on the input size can be left as 0, which tells Knet to infer that dimension when the first input is received. Leaving input dependent dimensions as 0, and using a keyword argument to determine output size we arrive at a fully generic version of linreg:

```
@knet function linreg3(x; out=1)
    w = par(dims=(out,0), init=Gaussian(0,0.1))
    b = par(dims=(out,1), init=Constant(0))
    return w * x .+ b
end
```

In this section, we have seen how to make @knet functions more generic using keyword arguments and size inference. This will especially come in handy when we are using them as new operators as described next.

2.4 4. Defining new operators

See also:

@knet function as operator, soft

The key to controlling complexity in computer languages is **abstraction**. Abstraction is the ability to name compound structures built from primitive parts, so they too can be used as primitives. In Knet we do this by using @knet functions not just as models, but as new operators inside other @knet functions.

To illustrate this, we will implement a softmax classification model. Softmax classification is basically linear regression with multiple outputs followed by normalization. Here is how we can define it in Knet:

```
@knet function softmax(x; out=10)
    z = linreg3(x; out=out)
    return soft(z)
end
```

The `softmax` model basically computes `soft(w * x .+ b)` with trainable parameters `w` and `b` by calling `linreg3` we defined in the previous section. The `out` keyword parameter determines the number of outputs and is passed from `softmax` to `linreg3` unchanged. The number of inputs is left unspecified and is inferred when the first input is received. The `soft` operator normalizes its argument by exponentiating its elements and dividing each by their sum.

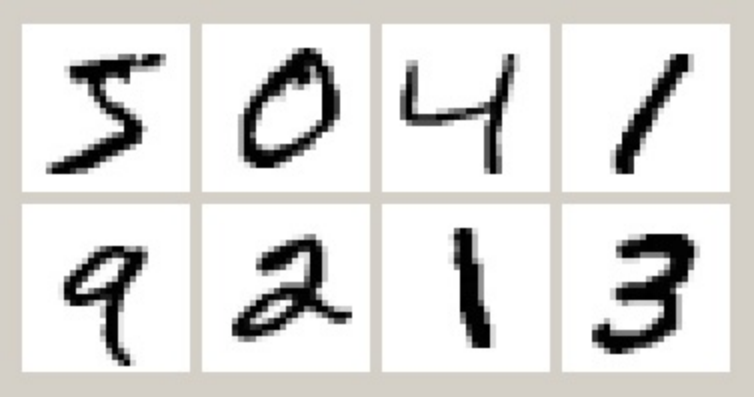
In this section we saw an example of using a @knet function as a new operator. Using the power of abstraction, not only can we avoid repetition and shorten the amount of code for larger models, we make the definitions a lot more readable and configurable, and gain a bunch of reusable operators to boot. To see some example reusable operators take a look at the *Knet compound operators* table and see their definitions in kfun.jl.

2.5 5. Training with minibatches

See also:

minibatch, softloss, zeroone

We will use the softmax model to classify hand-written digits from the MNIST dataset. Here are the first 8 images from MNIST, the goal is to look at the pixels and classify each image as one of the digits 0-9:



The following loads the MNIST data:

```
julia> include(Pkg.dir("Knet/examples/mnist.jl"))
INFO: Loading MNIST...
```

Once loaded, the data is available as multi-dimensional Julia arrays:

```
julia> MNIST.xtrn
28x28x1x60000 Array{Float32,4}:...
julia> MNIST.ytrn
10x60000 Array{Float32,2}:...
julia> MNIST.xtst
28x28x1x10000 Array{Float32,4}:...
julia> MNIST.ytst
10x10000 Array{Float32,2}:...
```

We have 60000 training and 10000 testing examples. Each input x is a 28x28x1 array representing one image, where the first two numbers represent the width and height in pixels, the third number is the number of channels (which is 1 for grayscale images, 3 for RGB images). The softmax model will treat each image as a `28*28*1=784` dimensional vector. The pixel values have been normalized to $[0, 1]$. Each output y is a ten-dimensional one-hot vector (a vector that has a single non-zero component) indicating the correct class (0-9) for a given image.

This is a much larger dataset than Housing. For computational efficiency, it is not advisable to use these examples one at a time during training like we did before. We will split the data into groups of 100 examples called **minibatches**, and pass data to `forw` and `back` one minibatch at a time instead of one instance at a time. On my laptop, one epoch of training softmax on MNIST takes about 0.34 seconds with a minibatch size of 100, 1.67 seconds with a minibatch size of 10, and 10.5 seconds if we do not use minibatches.

Knet provides a small `minibatch` function to split the data:

```
function minibatch(x, y, batchsize)
    data = Any[]
    for i=1:batchsize:ccount(x)
        j=min(i+batchsize-1,ccount(x))
        push!(data, (cget(x,i:j), cget(y,i:j)))
    end
    return data
end
```

`minibatch` takes `batchsize` columns of `x` and `y` at a time, pairs them up and pushes them into a `data` array. It works for arrays of any dimensionality, treating the last dimension as "columns". Note that this type of minibatching is fine for small datasets, but it requires holding two copies of the data in memory. For problems with a large amount of data you may want to use subarrays or iterables.

Here is `minibatch` in action:

```
julia> batchsize=100;
julia> trn = minibatch(MNIST.xtrn, MNIST.ytrn, batchsize)
600-element Array{Any,1}:...
julia> tst = minibatch(MNIST.xtst, MNIST.ytst, batchsize)
100-element Array{Any,1}:...
```

Each element of `trn` and `tst` is an x, y pair that contains 100 examples:

```
julia> trn[1]
(28x28x1x100 Array{Float32,4}: ...,
 10x100 Array{Float32,2}: ...)
```

Here are some simple train and test functions that use this type of minibatched data. Note that they take the loss function as a third argument and iterate through the x,y pairs (minibatches) in data:

```
function train(f, data, loss)
    for (x,y) in data
        forw(f, x)
        back(f, y, loss)
        update!(f)
    end
end

function test(f, data, loss)
    sumloss = numloss = 0
    for (x,ygold) in data
        ypred = forw(f, x)
        sumloss += loss(ypred, ygold)
        numloss += 1
    end
    return sumloss / numloss
end
```

Before training, we compile the model and set the learning rate to 0.2, which works well for this example. We use two new *loss functions*: `softloss` computes the cross entropy loss, $E(p \log \hat{p})$, commonly used for training classification models and `zeroone` computes the zero-one loss which is the proportion of predictions that were wrong. I got 7.66% test error after 40 epochs of training. Your results may be slightly different on different machines, or different runs on the same machine because of random initialization.

```
julia> model = compile(:softmax);
julia> setp(model; lr=0.2);
julia> for epoch=1:40; train(model, trn, softloss); end
julia> test(model, tst, zeroone)
0.0766...
```

In this section we saw how splitting the training data into minibatches can speed up training. We trained our first classification model on MNIST and used two new loss functions: `softloss` and `zeroone`.

2.6 6. MLP

2.7 7. Convnet

Deprecated

See also:

@knet as op, kwargs for @knet functions, function options (f=:relu). splat. lenet example, fast enough on cpu?

To illustrate this, we will use the LeNet convolutional neural network model designed to recognize handwritten digits. Here is the LeNet model defined using only the *primitive operators of Knet*:

```
@knet function lenet1(x)     # dims=(28,28,1,N)
    w1 = par(init=Xavier(),   dims=(5,5,1,20))
    c1 = conv(w1,x)          # dims=(24,24,20,N)
    b1 = par(init=Constant(0),dims=(1,1,20,1))
    a1 = add(b1,c1)
    r1 = relu(a1)
    p1 = pool(r1; window=2) # dims=(12,12,20,N)

    w2 = par(init=Xavier(),   dims=(5,5,20,50))
    c2 = conv(w2,p1)         # dims=(8,8,50,N)
    b2 = par(init=Constant(0),dims=(1,1,50,1))
    a2 = add(b2,c2)
    r2 = relu(a2)
    p2 = pool(r2; window=2) # dims=(4,4,50,N)

    w3 = par(init=Xavier(),   dims=(500,800))
    d3 = dot(w3,p2)          # dims=(500,N)
    b3 = par(init=Constant(0),dims=(500,1))
    a3 = add(b3,d3)
    r3 = relu(a3)

    w4 = par(init=Xavier(),   dims=(10,500))
    d4 = dot(w4,r3)          # dims=(10,N)
    b4 = par(init=Constant(0),dims=(10,1))
    a4 = add(b4,d4)
    return soft(a4)          # dims=(10,N)
end
```

Don't worry about the details of the model if you don't know much about neural nets. At 22 lines long, this model looks a lot more complicated than our linear regression model. Compared to state of the art image processing models however, it is still tiny. You would not want to code a state-of-the-art model like GoogLeNet using these primitives.

If you are familiar with neural nets, and peruse the *Knet primitives table*, you can see that the model has two convolution-pooling layers (commonly used in image processing), a fully connected relu layer and a final softmax output layer (I separated them by blank lines to help). Wouldn't it be nice to say just *that*:

```
@knet function lenet2(x)
    a = conv_pool_layer(x)
    b = conv_pool_layer(a)
    c = relu_layer(b)
    return softmax_layer(c)
end
```

`lenet2` is a lot more readable than `lenet1`. But before we can use this definition, we have to solve two problems:

- `conv_pool_layer` etc. are not primitive operators, we need a way to add them to Knet.
- Each layer has some attributes, like `init` and `dims`, that we need to be able to configure.

Knet solves the first problem by allowing @knet functions to be used as operators as well as models. For example, we can define `conv_pool_layer` as an operator with:

```
@knet function conv_pool_layer(x)
    w = par(init=Xavier(), dims=(5,5,1,20))
    c = conv(w,x)
    b = par(init=Constant(0), dims=(1,1,20,1))
    a = add(b,c)
    r = relu(a)
```

```
    return pool(r; window=2)
end
```

With this definition, the the first `a = conv_pool_layer(x)` operation in `lenet2` will work exactly as we want, but not the second (it has different convolution dimensions).

This brings us to the second problem, layer configuration. It would be nice not to hard-code numbers like `(5,5,1,20)` in the definition of a new operation like `conv_pool_layer`. Making these numbers configurable would make such operations more reusable across models. Even within the same model, you may want to use the same layer type in more than one configuration. For example in `lenet2` there is no way to distinguish the two `conv_pool_layer` operations, but looking at `lenet1` we clearly want them to do different things.

Knet solves the layer configuration problem using keyword arguments. Knet functions borrow the keyword argument syntax from Julia, and we will be using them in many contexts, so a brief aside is in order: Keyword arguments are identified by name instead of position, and they can be passed in any order (or not passed at all) following regular (positional) arguments. In fact we have already seen examples: `dims` and `init` are keyword arguments for `par` (which has no regular arguments) and `window` is a keyword argument for `pool`. Functions with keyword arguments are defined using a semicolon in the signature, e.g. `function pool(x; window=2, padding=0)`. The semicolon is optional when the function is called, e.g. both `pool(x, window=5)` or `pool(x; window=5)` work. Unspecified keyword arguments take their default values specified in the function definition. Extra keyword arguments can be collected using three dots in the function definition: `function pool(x; window=2, padding=0, o...)`, and passed in function calls: `pool(x; o...)`.

Here is a configurable version of `conv_pool_layer` using keyword arguments:

```
@knet function conv_pool_layer(x; cwindow=0, cinput=0, coutput=0, pwindow=0)
    w = par(init=Xavier(), dims=(cwindow,cwindow,cinput,coutput))
    c = conv(w,x)
    b = par(init=Constant(0), dims=(1,1,coutput,1))
    a = add(b,c)
    r = relu(a)
    return pool(r; window=pwindow)
end
```

Similarly, we can define `relu_layer` and `softmax_layer` with keyword arguments and make them more reusable. If you did this, however, you'd notice that we are repeating a lot of code. That is almost always a bad idea. Why don't we define a `generic_layer` that contains the shared code for all our layers:

```
@knet function generic_layer(x; f1=:dot, f2=:relu, wdims=(), bdims=(), winit=Xavier(), binit=Constant
    w = par(init=winit, dims=wdims)
    y = f1(w,x)
    b = par(init=binit, dims=bdims)
    z = add(b,y)
    return f2(z)
end
```

Note that in this example we are not only making initialization parameters like `winit` and `binit` configurable, we are also making internal operators like `relu` and `dot` configurable (their names need to be escaped with colons when passed as keyword arguments). This generic layer will allow us to define many layer types easily:

```
@knet function conv_pool_layer(x; cwindow=0, cinput=0, coutput=0, pwindow=0)
    y = generic_layer(x; f1=:conv, f2=:relu, wdims=(cwindow,cwindow,cinput,coutput), bdims=(1,1,coutp
    return pool(y; window=pwindow)
end

@knet function relu_layer(x; input=0, output=0)
    return generic_layer(x; f1=:dot, f2=:relu, wdims=(output,input), bdims=(output,1))
end
```

```
@knet function softmax_layer(x; input=0, output=0)
    return generic_layer(x; f1=:dot, f2=:soft, wdims=(output,input), bdims=(output,1))
end
```

Finally we can define a working version of LeNet using 4 lines of code:

```
@knet function lenet3(x)
    a = conv_pool_layer(x; cwindow=5, cinput=1,  coutput=20, pwindow=2)
    b = conv_pool_layer(a; cwindow=5, cinput=20, coutput=50, pwindow=2)
    c = relu_layer(b; input=800, output=500)
    return softmax_layer(c; input=500, output=10)
end
```

There are still a lot of hard-coded dimensions in `lenet3`. Some of these, like the filter size (5), and the hidden layer size (500) can be considered part of the model design. We should make them configurable so the user can experiment with different sized models. But some, like the number of input channels (1), and the input to the `relu_layer` (800) are determined by input size. If we tried to apply `lenet3` to a dataset with different sized images, it would break. Knet solves this problem using **size inference**: Any dimension that relies on the input size can be left as 0, which tells Knet to infer that dimension when the first input is received. Leaving input dependent dimensions as 0, and using keyword arguments to determine model size we arrive at a fully configurable version of LeNet:

```
@knet function lenet4(x; cwin1=5, cout1=20, pwin1=2, cwin2=5, cout2=50, pwin2=2, hidden=500, nclass=1
    a = conv_pool_layer(x; cwindow=cwin1, coutput=cout1, pwindow=pwin1)
    b = conv_pool_layer(a; cwindow=cwin2, coutput=cout2, pwindow=pwin2)
    c = relu_layer(b; output=hidden)
    return softmax_layer(c; output=nclass)
end
```

To compile an instance of `lenet4` with particular dimensions, we pass keyword arguments to `compile`:

```
julia> f = compile(:lenet4; cout1=30, cout2=60, hidden=600)
...
```

In this section we saw how to use @knet functions as new operators, and configure them using keyword arguments. Using the power of abstraction, not only did we cut the amount of code for the LeNet model in half, we made its definition a lot more readable and configurable, and gained a bunch of reusable operators to boot. I am sure you can think of more clever ways to define LeNet and other complex models using your own set of operators. To see some example reusable operators take a look at the *Knet compound operators* table and see their definitions in kfun.jl.

2.8 8. Conditional Evaluation

See also:

if-else, runtime conditions (kwargs for forw), dropout

There are cases where you want to execute parts of a model *conditionally*, e.g. only during training, or only during some parts of the input in sequence models. Knet supports the use of **runtime conditions** for this purpose. We will illustrate the use of conditions by implementing a training technique called dropout to improve the generalization power of the LeNet model.

If you keep training the LeNet model on MNIST for about 30 epochs you will observe that the training error drops to zero but the test error hovers around 0.8%:

```
for epoch=1:100
    train(net, trn, softloss)
    println((epoch, test(net, trn, zeroone), test(net, tst, zeroone)))
end
```

```
(1,0.020466666666666505,0.024799999999999996)
(2,0.013649999999999905,0.01820000000000001)
...
(29,0.0,0.008100000000000003)
(30,0.0,0.008000000000000004)
```

This is called *overfitting*. The model has memorized the training set, but does not generalize equally well to the test set.

Dropout prevents overfitting by injecting random noise into the model. Specifically, for each `forw` call during training, dropout layers placed between two operations replace a random portion of their input with zeros, and scale the rest to keep the total output the same. During testing random noise would degrade performance, so we would like to turn dropout off. Here is one way to implement this in Knet:

```
@knet function drop(x; pdrop=0, o...)
    if dropout
        return x .* rnd(init=Bernoulli(1-pdrop, 1/(1-pdrop)))
    else
        return x
    end
end
```

The keyword argument `pdrop` specifies the probability of dropping an input element. The `if ... else ... end` block causes conditional evaluation the way one would expect. The variable `dropout` next to `if` is a global condition variable: it is not declared as an argument to the function. Instead, once a model with a `drop` operation is compiled, the call to `forw` accepts `dropout` as an optional keyword argument and passes it down as a global condition:

```
forw(model, input; dropout=true)
```

This means every time we call `forw`, we can change whether dropout occurs or not. During test time, we would like to stop dropout, so we can run the model with `dropout=false`:

```
forw(model, input; dropout=false)
```

By default, all unspecified condition variables are false, so we could also omit the condition during test time:

```
forw(model, input)  # dropout=false is assumed
```

Here is one way to add dropout to the LeNet model:

```
@knet function lenet5(x; pdrop=0.5, cwin1=5, cout1=20, pwin1=2, cwin2=5, cout2=50, pwin2=2, hidden=5(
    a = conv_pool_layer(x; cwindow=cwin1, coutput=cout1, pwindow=pwin1)
    b = conv_pool_layer(a; cwindow=cwin2, coutput=cout2, pwindow=pwin2)
    bdrop = drop(b; pdrop=pdrop)
    c = relu_layer(bdrop; output=hidden)
    return softmax_layer(c; output=nclass)
end
```

Whenever the condition variable `dropout` is true, this will replace half of the entries in the `b` array with zeros. We need to modify our `train` function to pass the condition to `forw`:

```
function train(f, data, loss)
    for (x,y) in data
        forw(f, x; dropout=true)
        back(f, y, loss)
        update!(f)
    end
end
```

Here is our training script. Note that we reduce the learning rate whenever the test error gets worse, another precaution against overfitting:

```
lrate = 0.1
decay = 0.9
lasterr = 1.0
net = compile(:lenet5)
setp(net; lr=lrate)

for epoch=1:100
    train(net, trn, softloss)
    trnerr = test(net, trn, zeroone)
    tsterr = test(net, tst, zeroone)
    println((epoch, lrate, trnerr, tsterr))
    if tsterr > lasterr
        lrate = decay*lrate
        setp(net; lr=lrate)
    end
    lasterr = tsterr
end
```

In 100 epochs, this should converge to about 0.5% error, i.e. reduce the total number of errors on the 10K test set from around 80 to around 50. Congratulations! This is fairly close to the state of the art compared to other benchmark results on the MNIST website:

```
(1,0.1,0.020749999999999824,0.01960000000000001)
(2,0.1,0.013699999999999895,0.01600000000000001)
...
(99,0.0014780882941434613,0.0003333333333333334,0.005200000000000002)
(100,0.0014780882941434613,0.0003666666666666668,0.005000000000000002)
```

In this section, we saw how to use the `if ... else ... end` construct to perform conditional evaluation in a model, where the conditions are passed using keyword arguments to `forw`. We used this to implement `dropout`, an effective technique to prevent overfitting.

2.9 9. Recurrent neural networks

See also:

read-before-write, simple rnn, lstm

In this section we will see how to implement **recurrent neural networks** (RNNs) in Knet. A RNN is a class of neural network where connections between units form a directed cycle, which allows them to keep a persistent state (memory) over time. This gives them the ability to process sequences of arbitrary length one element at a time, while keeping track of what happened at previous elements. Contrast this with feed forward nets like LeNet, which have a fixed sized input, output and perform a fixed number of operations. See (Karpathy, 2015) for a nice introduction to RNNs.

To support RNNs, all local variables in Knet functions are static variables, i.e. their values are preserved between calls unless otherwise specified. It turns out this is the only language feature you need to define RNNs. Here is a simple example:

```
@knet function rnn1(x; hsize=100, xsize=50)
    a = par(init=Xavier(), dims=(hsize, xsize))
    b = par(init=Xavier(), dims=(hsize, hsize))
    c = par(init=Constant(0), dims=(hsize, 1))
    d = a * x .+ b * h .+ c
```

```
    h = relu(d)
end
```

Notice anything strange? The first three lines define three model parameters. Then the fourth line sets `d` to a linear combination of the input `x` and the hidden state `h`. But `h` hasn't been defined yet. Exactly! Having read-before-write variables is the only thing that distinguishes an RNN from feed-forward models like LeNet.

The way Knet handles read-before-write variables is by initializing them to 0 arrays before any input is processed, then preserving the values between the calls. Thus during the first call in the above example, `h` would start as 0, `d` would be set to `a * x .+ c`, which in turn would cause `h` to get set to `relu(a * x .+ c)`. During the second call, this value of `h` would be remembered and used, thus making the value of `h` at time t dependent on its value at time t-1.

It turns out simple RNNs like `rnn1` are not very good at remembering things for a very long time. There are some techniques to improve their retention based on better initialization or smarter updates, but currently the most popular solution is using more complicated units like LSTMs and GRUs. These units control the information flow into and out of the unit using gates similar to digital circuits and can model long term dependencies. See (Colah, 2015) for a good overview of LSTMs.

Defining an LSTM in Knet is almost as concise as writing its mathematical definition:

```
@knet function lstm(x; fbias=1, o...)
    input  = wbf2(x,h; o..., f=:sigm)
    forget = wbf2(x,h; o..., f=:sigm, binit=Constant(fbias))
    output = wbf2(x,h; o..., f=:sigm)
    newmem = wbf2(x,h; o..., f=:tanh)
    cell = input .* newmem + cell .* forget
    h  = tanh(cell) .* output
    return h
end
```

The `wbf2` operator applies an affine function (linear function + bias) to its two inputs followed by an activation function (specified by the `f` keyword argument). Try to define this operator yourself as an exercise, (see kfun.jl for the Knet definition).

The LSTM has an input gate, forget gate and an output gate that control information flow. Each gate depends on the current input `x`, and the last output `h`. The memory value `cell` is computed by blending a new value `newmem` with its old value under the control of `input` and `forget` gates. The `output` gate decides how much of the `cell` is shared with the outside world.

If an `input` gate element is close to 0, the corresponding element in the new input `x` will have little effect on the memory cell. If a `forget` gate element is close to 1, the contents of the corresponding memory cell can be preserved for a long time. Thus the LSTM has the ability to pay attention to the current input, or reminisce in the past, and it can learn when to do which based on the problem.

In this section we introduced simple recurrent neural networks and LSTMs. We saw that having static variables is the only language feature necessary to implement RNNs. Next we will look at how to train them.

2.10 10. Training with sequences

(Karpathy, 2015) has lots of fun examples showing how character based language models based on LSTMs are surprisingly adept at generating text in many genres, from Wikipedia articles to C programs. To demonstrate training with sequences, we'll implement one of these examples and build a model that can write like Shakespeare! After training on "The Complete Works of William Shakespeare" for less than an hour, here is a sample of brilliant writing you can expect from your model:

```
LUCETTA. Welcome, getzing a knot. There is as I thought you aim
  Cack to Corioli.
MACBETH. So it were timen'd nobility and prayers after God'.
FIRST SOLDIER. O, that, a tailor, cold.
DIANA. Good Master Anne Warwick!
SECOND WARD. Hold, almost proverb as one worth ne'er;
  And do I above thee confer to look his dead;
  I'll know that you are ood'd with memines;
  The name of Cupid wiltwite tears will hold
  As so I fled; and purgut not brightens,
  Their forves and speed as with these terms of Ely
  Whose picture is not dignitories of which,
  Their than disgrace to him she is.
GOBARIND. O Sure, ThisH more.,
  wherein hath he been not their deed of quantity,
  No ere we spoke itation on the tent.
  I will be a thought of base-thief;
  Then tears you ever steal to have you kindness.
  And so, doth not make best in lady,
  Your love was execreed'd fray where Thoman's nature;
  I have bad Tlauphie he should sray and gentle,
```

First let's download "The Complete Works of William Shakespeare" from Project Gutenberg:

```
julia> using Requests
julia> url="http://gutenberg.pglaf.org/1/0/100/100.txt";
julia> text=get(url).data
5589917-element Array{UInt8,1}:...
```

The `text` array now has all 5,589,917 characters of "The Complete Works" in a Julia array. If `get` does not work, you can download `100.txt` by other means and use `text=readall("100.txt")` on the local file. We will use one-hot vectors to represent characters, so let's map each character to an integer index $1 \ldots n$:

```
julia> char2int = Dict();
julia> for c in text; get!(char2int, c, 1+length(char2int)); end
julia> nchar = length(char2int)
92
```

`Dict` is Julia's standard associative collection for mapping arbitrary keys to values. `get!(dict,key,default)` returns the value for the given key, storing `key=>default` in `dict` if no mapping for the key is present. Going over the `text` array we discover 92 unique characters and map them to integers $1 \ldots 92$.

We will train our RNN to read characters from `text` in sequence, and predict the next character after each. The training will go much faster if we can use the minibatching trick we saw earlier and process multiple inputs at a time. For that, we split the text array into `batchsize` equal length subsequences. Then the first batch has the first character from each subsequence, second batch contains the second characters etc. Each minibatch is represented by a `nchar x batchsize` matrix with one-hot columns. Here is a function that implements this type of sequence minibatching:

```
function seqbatch(seq, dict, batchsize)
    data = Any[]
    T = div(length(seq), batchsize)
    for t=1:T
        d=zeros(Float32, length(dict), batchsize)
        for b=1:batchsize
            c = dict[seq[t + (b-1) * T]]
            d[c,b] = 1
        end
        push!(data, d)
    end
```

```
    return data
end
```

Let's use it to split `text` into minibatches of size 128:

```
julia> batchsize = 128;
julia> data = seqbatch(text, char2int, batchsize)
43671-element Array{Any,1}:...
julia> data[1]
92x128 Array{Float32,2}:...
```

The data array returned has `T=length(text)/batchsize` minibatches. The columns of minibatch `data[t]` refer to characters `t`, `t+T`, `t+2T`, ... from `text`. During training, when `data[t]` is the input, `data[t+1]` will be the desired output. Now that we have the data ready to go, let's talk about RNN training.

RNN training is a bit more involved than training feed-forward models. We still have the prediction, gradient calculation and update steps, but not all three steps should be performed after every input. Here is a basic algorithm: Go forward `nforw` steps, remembering the desired outputs and model state, then perform `nforw` back steps accumulating gradients, finally update the parameters and reset the network for the next iteration:

```
function train(f, data, loss; nforw=100, gclip=0)
    reset!(f)
    ystack = Any[]
    T = length(data) - 1
    for t = 1:T
        x = data[t]
        y = data[t+1]
        sforw(f, x; dropout=true)
        push!(ystack, y)
        if (t % nforw == 0 || t == T)
            while !isempty(ystack)
                ygold = pop!(ystack)
                sback(f, ygold, loss)
            end
            update!(f; gclip=gclip)
            reset!(f; keepstate=true)
        end
    end
end
```

Note that we use `sforw` and `sback` instead of `forw` and `back` during sequence training: these save and restore internal state to allow multiple forward steps followed by multiple backward steps. `reset!` is necessary to zero out or recover internal state before a sequence of forward steps. `ystack` is used to store gold answers. The `gclip` is for gradient clipping, a common RNN training strategy to keep the parameters from diverging.

With data and training script ready, all we need is a model. We will define a character based RNN language model using an LSTM:

```
@knet function charlm(x; embedding=0, hidden=0, pdrop=0, nchar=0)
    a = wdot(x; out=embedding)
    b = lstm(a; out=hidden)
    c = drop(b; pdrop=pdrop)
    return wbf(c; out=nchar, f=:soft)
end
```

`wdot` multiplies the one-hot representation `x` of the input character with an embedding matrix and turns it into a dense vector of size `embedding`. We apply an LSTM of size `hidden` to this dense vector, and dropout the result with probability `pdrop`. Finally `wbf` applies softmax to a linear function of the LSTM output to get a probability vector of size `nchar` for the next character.

(Karpathy, 2015) uses not one but several LSTM layers to simulate Shakespeare. In Knet, we can define a multi-layer LSTM model using the high-level operator `repeat`:

```
@knet function lstmdrop(a; pdrop=0, hidden=0)
    b = lstm(a; out=hidden)
    return drop(b; pdrop=pdrop)
end

@knet function charlm2(x; nlayer=0, embedding=0, hidden=0, pdrop=0, nchar=0)
    a = wdot(x; out=embedding)
    c = repeat(a; frepeat=:lstmdrop, nrepeat=nlayer, hidden=hidden, pdrop=pdrop)
    return wbf(c; out=nchar, f=:soft)
end
```

In `charlm2`, the `repeat` instruction will perform the `frepeat` operation `nrepeat` times starting with input `a`. Using `charlm2` with `nlayer=1` would be equivalent to the original `charlm`.

In the interest of time we will start with a small single layer model. With the following parameters, 10 epochs of training takes about 35-40 minutes on a K20 GPU:

```
julia> net = compile(:charlm; embedding=256, hidden=512, pdrop=0.2, nchar=nchar);
julia> setp(net; lr=1.0)
julia> for i=1:10; train(net, data, softloss; gclip=5.0); end
```

After spending this much time training a model, you probably want to save it. Knet uses the JLD module to save and load models and data. Calling `clean(model)` during a save is recommended to strip the model of temporary arrays which may save a lot of space. Don't forget to save the `char2int` dictionary, otherwise it will be difficult to interpret the output of the model:

```
julia> using JLD
julia> JLD.save("charlm.jld", "model", clean(net), "dict", char2int);
julia> net2 = JLD.load("charlm.jld", "model")        # should create a copy of net
...
```

TODO: put load/save and other fns in the function table.

Finally, to generate the Shakespearean output we promised, we need to implement a generator. The following generator samples a character from the probability vector output by the model, prints it and feeds it back to the model to get the next character. Note that we use regular `forw` in `generate`, `sforw` is only necessary when training RNNs.

```
function generate(f, int2char, nchar)
    reset!(f)
    x=zeros(Float32, length(int2char), 1)
    y=zeros(Float32, length(int2char), 1)
    xi = 1
    for i=1:nchar
        copy!(y, forw(f,x))
        x[xi] = 0
        xi = sample(y)
        x[xi] = 1
        print(int2char[xi])
    end
    println()
end

function sample(pdist)
    r = rand(Float32)
    p = 0
    for c=1:length(pdist)
        p += pdist[c]
```

```
        r <= p && return c
    end
end
```

```
julia> int2char = Array(Char, length(char2int));
julia> for (c,i) in char2int; int2char[i] = Char(c); end
julia> generate(net, int2char, 1024)  # should generate 1024 chars of Shakespeare
```

TODO: In this section...

2.11 Some useful tables

Table 1: Primitive Knet operators

Operator	Description
`par()`	a parameter array, updated during training; kwargs: `dims, init`
`rnd()`	a random array, updated every call; kwargs: `dims, init`
`arr()`	a constant array, never updated; kwargs: `dims, init`
`dot(A,B)`	matrix product of `A` and `B`; alternative notation: `A * B`
`add(A,B)`	elementwise broadcasting addition of arrays `A` and `B`, alternative notation: `A .+ B`
`mul(A,B)`	elementwise broadcasting multiplication of arrays `A` and `B`; alternative notation: `A .* B`
`conv(W,X)`	convolution with filter `W` and input `X`; kwargs: `padding=0, stride=1, upscale=1, mode=CUDNN_CONVOLUTION`
`pool(X)`	pooling; kwargs: `window=2, padding=0, stride=window, mode=CUDNN_POOLING_MAX`
`axpb(X)`	computes `a*x^p+b`; kwargs: `a=1, p=1, b=0`
`copy(X)`	copies `X` to output.
`relu(X)`	rectified linear activation function: `(x > 0 ? x : 0)`
`sigm(X)`	sigmoid activation function: `1/(1+exp(-x))`
`soft(X)`	softmax activation function: `(exp xi) / (Σ exp xj)`
`tanh(X)`	hyperbolic tangent activation function.

Table 2: Compound Knet operators

These operators combine several primitive operators and typically hide the parameters in their definitions to make code more readable.

Operator	Description
`wdot(x)`	apply a linear transformation `w * x`; kwargs: `out=0, winit=Xavier()`
`bias(x)`	add a bias `x .+ b`; kwargs: `binit=Constant(0)`
`wb(x)`	apply an affine function `w * x .+ b`; kwargs: `out=0, winit=Xavier(), binit=Constant(0)`
`wf(x)`	linear transformation + activation function `f(w * x)`; kwargs: `f=:relu, out=0, winit=Xavier()`
`wbf(x)`	affine function + activation function `f(w * x .+ b)`; kwargs: `f=:relu, out=0, winit=Xavier(), binit=Constant(0)`
`wbf2(x,y)`	affine function + activation function for two variables `f(a*x .+ b*y .+ c)`; kwargs:`f=:sigm, out=0, winit=Xavier(), binit=Constant(0)`
`wconv(x)`	apply a convolution `conv(w,x)`; kwargs: `out=0, window=0, padding=0, stride=1, upscale=1, mode=CUDNN_CONVOLUTION, cinit=Xavier()`
`cbfp(x)`	convolution, bias, activation function, and pooling; kwargs: `f=:relu, out=0, cwindow=0, pwindow=0, cinit=Xavier(), binit=Constant(0)`
`drop(x)`	replace `pdrop` of the input with 0 and scale the rest with `1/(1-pdrop)`; kwargs: `pdrop=0`
`lstm(x)`	LSTM; kwargs:`fbias=1, out=0, winit=Xavier(), binit=Constant(0)`
`irnn(x)`	IRNN; kwargs:`scale=1, out=0, winit=Xavier(), binit=Constant(0)`
`gru(x)`	GRU; kwargs:`out=0, winit=Xavier(), binit=Constant(0)`
`repeat(x)`	apply operator `frepeat` to input `x` `nrepeat` times; kwargs: ``frepeat=nothing, nrepeat=0`

Table 3: Random distributions

This table lists random distributions and other array fillers that can be used to initalize parameters (used with the `init` keyword argument for `par`).

Distribution	Description
`Bernoulli(p,scale)`	output `scale` with probability `p` and 0 otherwise
`Constant(val)`	fill with a constant value `val`
`Gaussian(mean, std)`	normally distributed random values with mean `mean` and standard deviation `std`
`Identity(scale)`	identity matrix multiplied by `scale`
`Uniform(min, max)`	uniformly distributed random values between `min` and `max`
`Xavier()`	Xavier initialization: deprecated, please use Glorot. Uniform in $[-\sqrt{3/n}, \sqrt{3/n}]$ where n=length(a)/size(a)[end]

Table 4: Loss functions

Function	Description
`softloss(ypred,ygold)`	Cross entropy loss: $E[p \log \hat{p}]$
`quadloss(ypred,ygold)`	Quadratic loss: $E[(y - \hat{y})^2]$
`zeroone(ypred,ygold)`	Zero-one loss: $E[\arg\max y \neq \arg\max \hat{y}]$

Table 5: Training options

We can manipulate how exactly `update!` behaves by setting some training options like the learning rate `lr`. I'll explain the mathematical motivation elsewhere, but algorithmically these training options manipulate the `dw` array (sometimes using an auxiliary array `dw2`) before the subtraction to improve the loss faster. Here is a list of training options supported by Knet and how they manipulate `dw`:

Option	Description
`lr`	Learning rate: `dw *= lr`
`l1reg`	L1 regularization: `dw += l1reg * sign(w)`
`l2reg`	L2 regularization: `dw += l2reg * w`
`adagrad`	Adagrad (boolean): `dw2 += dw .* dw; dw = dw ./ (1e-8 + sqrt(dw2))`
`rmsprop`	Rmsprop (boolean): `dw2 = dw2 * 0.9 + 0.1 * dw .* dw; dw = dw ./ (1e-8 + sqrt(dw2))`
`adam`	Adam (boolean); see http://arxiv.org/abs/1412.6980
`momentum`	Momentum: `dw += momentum * dw2; dw2 = dw`
`nesterov`	Nesterov: `dw2 = nesterov * dw2 + dw; dw += nesterov * dw2`

Table 6: Summary of modeling related functions

Function	Description
`@kfun function ... end`	defines a @knet function that can be used as a model or a new operator
`if cond ... else ... end`	conditional evaluation in a @knet function with condition variable `cond` supplied by `forw`
`compile(:kfun; o...)`	creates a model given @knet function `kfun`; kwargs used for model configuration
`forw(f,x; o...)`	returns the prediction of model `f` on input `x`; kwargs used for setting conditions
`back(f,ygold,loss)`	computes the loss gradients for `f` parameters based on desired output `ygold` and loss function `loss`
`update!(f)`	updates the parameters of `f` using the gradients computed by `back` to reduce loss
`get(f,:w)`	return parameter `w` of model `f`
`setp(f; opt=val...)`	sets training options for model `f`
`minibatch(x,y,batchsize)`	split data into minibatches

CHAPTER 3

Backpropagation

Note: Concepts: supervised learning, training data, regression, squared error, linear regression, stochastic gradient descent

Arthur Samuel, the author of the first self-learning checkers program, defined machine learning as a "field of study that gives computers the ability to learn without being explicitly programmed". This leaves the definition of learning a bit circular. Tom M. Mitchell provided a more formal definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E," where the task, the experience, and the performance measure are to be specified based on the problem.

We will start with **supervised learning**, where the task is to predict the output of an unknown system given its input, and the experience consists of a set of example input-output pairs, also known as the **training data**. When the outputs are numeric such problems are called **regression**. In **linear regression** we use a linear function as our model:

$$\hat{y} = Wx + b$$

Here x is the model input, $\hat{y}$ is the model output, W is a matrix of weights, and b is a vector of biases. By adjusting the parameters of this model, i.e. the weights and the biases, we can make it compute any linear function of x.

"All models are wrong, but some models are useful." George Box famously said. We do not necessarily know that the system whose output we are trying to predict is governed by a linear relationship. All we know is a finite number of input-output examples:

$$\mathcal{D} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$$

It is just that we have to start model building somewhere and the set of all linear functions is a good place to start for now.

A commonly used performance measure in regression problems is the **squared error**, i.e. the average squared difference between the actual output values and the ones predicted by the model. So our goal is to find model parameters that minimize the squared error:

$$\arg\min_{W,b} \frac{1}{N} \sum_{n=1}^{N} \|\hat{y}_n - y_n\|^2$$

Where $\hat{y}_n = Wx_n + b$ denotes the output predicted by the model for the n th example.

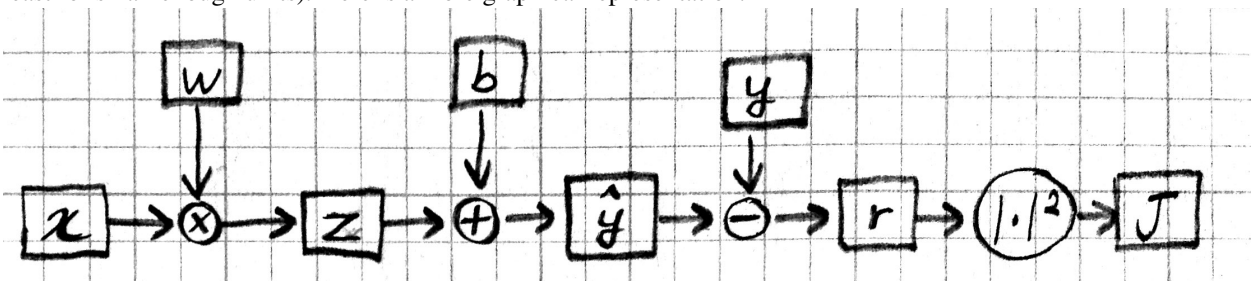
There are several methods to find the solution to the problem of minimizing squared error. Here we will present the **stochastic gradient descent** (SGD) method because it generalizes well to more complex models. In SGD, we take the training examples one at a time (or in small groups called minibatches), compute the gradient of the error with respect to the parameters, and move the parameters a small step in the direction that will decrease the error. First some notes on the math.

3.1 Partial derivatives

When we have a function with several inputs and one output, we can look at how the function value changes in response to a small change in one of its inputs holding the rest fixed. This is called a partial derivative. Let us consider the squared error for the n th input as an example:

$$J = \|Wx_n + b - y_n\|^2$$

So the partial derivative $\partial J/\partial w_{ij}$ would tell us how many units J would move if we moved w_{ij} in W one unit (at least for small enough units). Here is a more graphical representation:



In this figure, it is easier to see that the machinery that generates J has many "inputs". In particular we can talk about how J is effected by changing parameters W and b, as well as changing the input x, the model output $\hat{y}$, the desired output y, or intermediate values like z or r. So partial derivatives like $\partial J/\partial x_i$ or $\partial J/\partial \hat{y}_j$ are fair game and tell us how J would react in response to small changes in those quantities.

3.2 Chain rule

The chain rule allows us to calculate partial derivatives in terms of other partial derivatives, simplifying the overall computation. We will go over it in some detail as it forms the basis of the backpropagation algorithm. For now let us assume that each of the variables in the above example are scalars. We will start by looking at the effect of r on J and move backward from there. Basic calculus tells us that:

$$J = r^2$$
$$\partial J/\partial r = 2r$$

Thus, if $r = 5$ and we decrease r by a small ϵ, the squared error J will go down by 10ϵ. Now let's move back a step and look at $\hat{y}$:

$$r = \hat{y} - y$$
$$\partial r/\partial \hat{y} = 1$$

So how much effect will a small ϵ decrease in $\hat{y}$ have on J when $r = 5$? Well, when $\hat{y}$ goes down by ϵ, so will r, which means J will go down by 10ϵ again. The chain rule expresses this idea:

$$\frac{\partial J}{\partial \hat{y}} = \frac{\partial J}{\partial r}\frac{\partial r}{\partial \hat{y}} = 2r$$

Going back further, we have:

$$\hat{y} = z + b$$
$$\partial \hat{y}/\partial b = 1$$
$$\partial \hat{y}/\partial z = 1$$

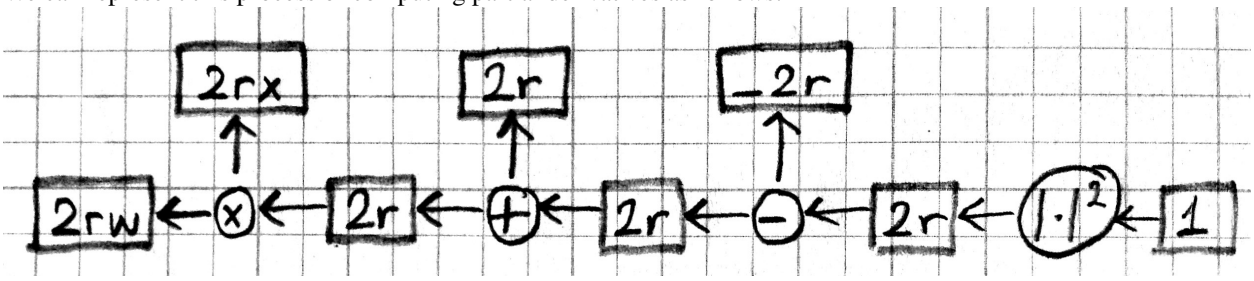
Which means b and z have the same effect on J as $\hat{y}$ and r, i.e. decreasing them by ϵ will decrease J by $2r\epsilon$ as well. Finally:

$$z = wx$$
$$\partial z/\partial x = w$$
$$\partial z/\partial w = x$$

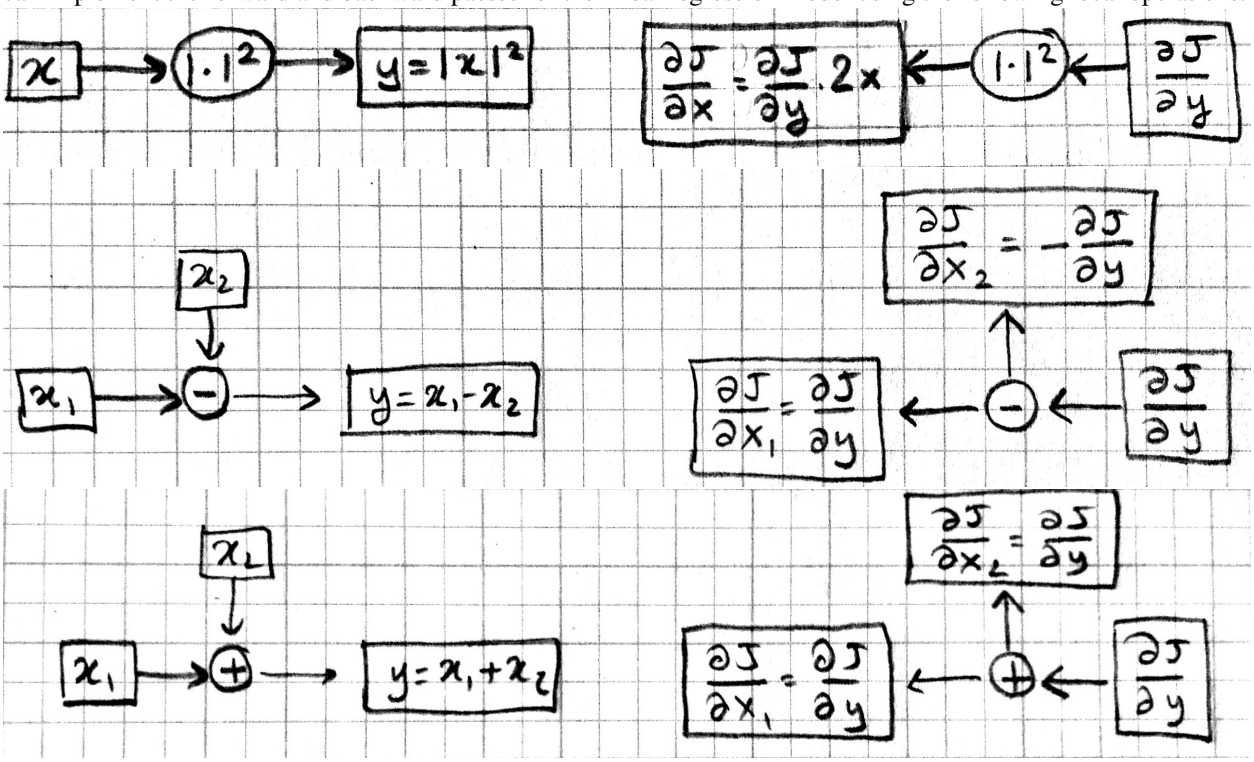
This allows us to compute the effect of w on J in several steps: moving w by ϵ will move z by $x\epsilon$, $\hat{y}$ and r will move exactly the same amount because their partials with z are 1, and finally since r moves by $x\epsilon$, J will move by $2rx\epsilon$.

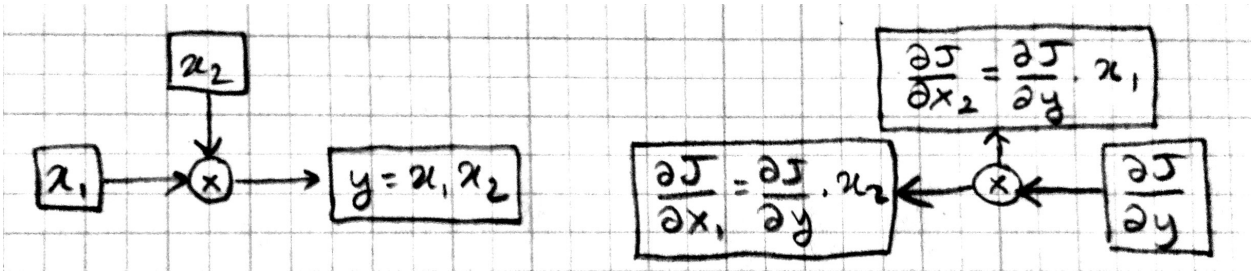
$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial r}\frac{\partial r}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w} = 2rx$$

We can represent this process of computing partial derivatives as follows:



Note that we have the same number of boxes and operations, but all the arrows are reversed. Let us call this the backward pass, and the original computation in the previous picture the forward pass. Each box in this backward-pass picture represents the partial derivative for the corresponding box in the previous forward-pass picture. Most importantly, each computation is local: each operation takes the partial derivative of its output, and multiplies it with a factor that only depends on the original input/output values to compute the partial derivative of its input(s). In fact we can implement the forward and backward passes for the linear regression model using the following local operations:





3.3 Multiple dimensions

Let's look at the case where the input and output are not scalars but vectors. In particular assume that $x \in \mathbb{R}^D$ and $y \in \mathbb{R}^C$. This makes $W \in \mathbb{R}^{C \times D}$ a matrix and $z, b, \hat{y}, r$ vectors in $\mathbb{R}^C$. During the forward pass, $z = Wx$ operation is now a matrix-vector product, the additions and subtractions are elementwise operations. The squared error $J = \|r\|^2 = \sum r_i^2$ is still a scalar. For the backward pass we ask how much each element of these vectors or matrices effect J. Starting with r:

$$J = \sum r_i^2$$
$$\partial J / \partial r_i = 2r_i$$

We see that when r is a vector, the partial derivative of each component is equal to twice that component. If we put these partial derivatives together in a vector, we obtain a **gradient** vector:

$$\nabla_r J \equiv \langle \frac{\partial J}{\partial r_1}, \cdots, \frac{\partial J}{\partial r_C} \rangle = \langle 2r_1, \ldots, 2r_C \rangle = 2\vec{r}$$

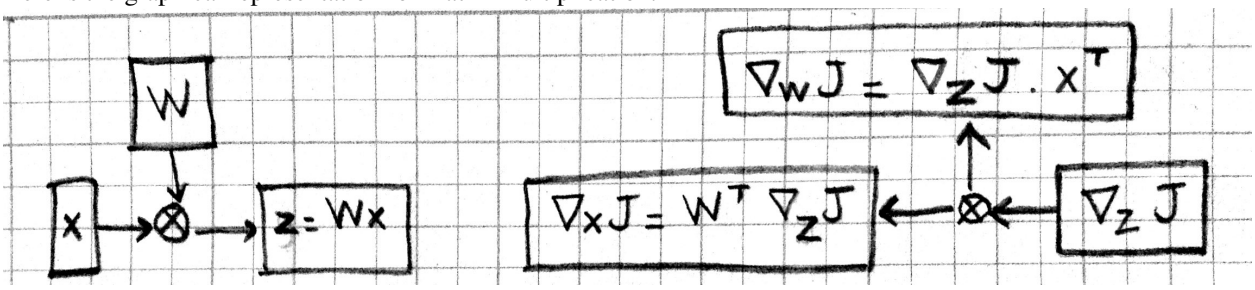
The addition, subtraction, and square norm operations work the same way as before except they act on each element. Moving back through the elementwise operations we see that:

$$\nabla_r J = \nabla_{\hat{y} J = \nabla_b J = \nabla_z J = 2\vec{r}}$$

For the operation $z = Wx$, a little algebra will show you that:

$$\nabla_W J = \nabla_z J \cdot x^T$$
$$\nabla_x J = W^T \cdot \nabla_z J$$

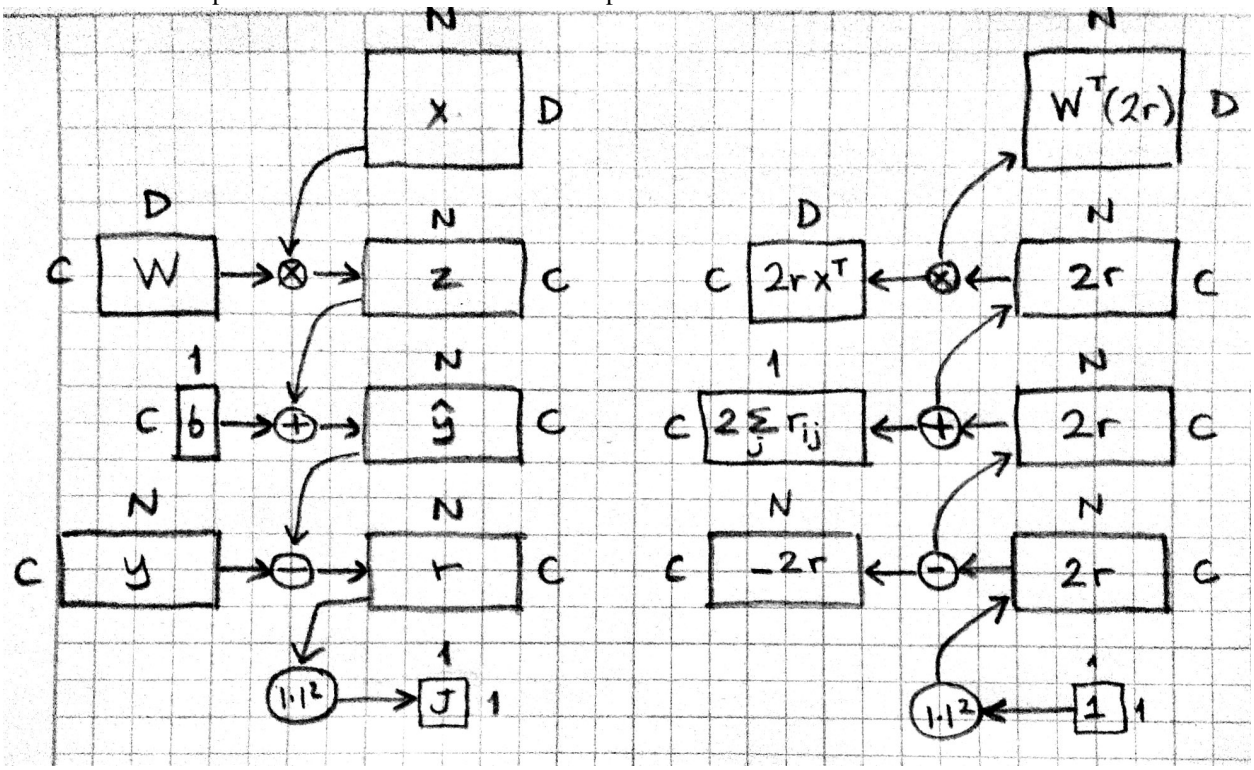
Note that the gradient of a variable has the same shape as the variable itself. In particular $\nabla_W J$ is a $C \times D$ matrix. Here is the graphical representation for matrix multiplication:



3.4 Multiple instances

We will typically process data multiple instances at a time for efficiency. Thus, the input x will be a $D \times N$ matrix, and the output y will be a $C \times N$ matrix, the N columns representing N different instances. Please verify to yourself

that the forward and backward operations as described above handle this case without much change: the elementwise operations act on the elements of the matrices just like vectors, and the matrix multiplication and its gradient remains the same. Here is a picture of the forward and backward passes:



The only complication is at the addition of the bias vector. In the batch setting, we are adding $b \in \mathbb{R}^{C\times 1}$ to $z \in \mathbb{R}^{C\times N}$. This will be a broadcasting operation, i.e. the vector b will be added to each column of the matrix z to get $\hat{y}$. In the backward pass, we'll need to add the columns of $\nabla_{\hat{y}} J$ to get the gradient $\nabla_b J$.

3.5 Stochastic Gradient Descent

The gradients calculated by backprop, $\nabla_w J$ and $\nabla_b J$, tell us how much small changes in corresponding entries in w and b will effect the error (for the last instance, or minibatch). Small steps in the gradient direction will increase the error, steps in the opposite direction will decrease the error.

In fact, we can show that the gradient is the direction of steepest ascent. Consider a unit vector v pointing in some arbitrary direction. The rate of change in this direction is given by the projection of v onto the gradient, i.e. their dot product $\nabla J \cdot v$. What direction maximizes this dot product? Recall that:

$$\nabla J \cdot v = |\nabla J|\ |v| \cos(\theta)$$

where θ is the angle between v and the gradient vector. $\cos(\theta)$ is maximized when the two vectors point in the same direction. So if you are going to move a fixed (small) size step, the gradient direction gives you the biggest bang for the buck.

This suggests the following update rule:

$$w \leftarrow w - \nabla_w J$$

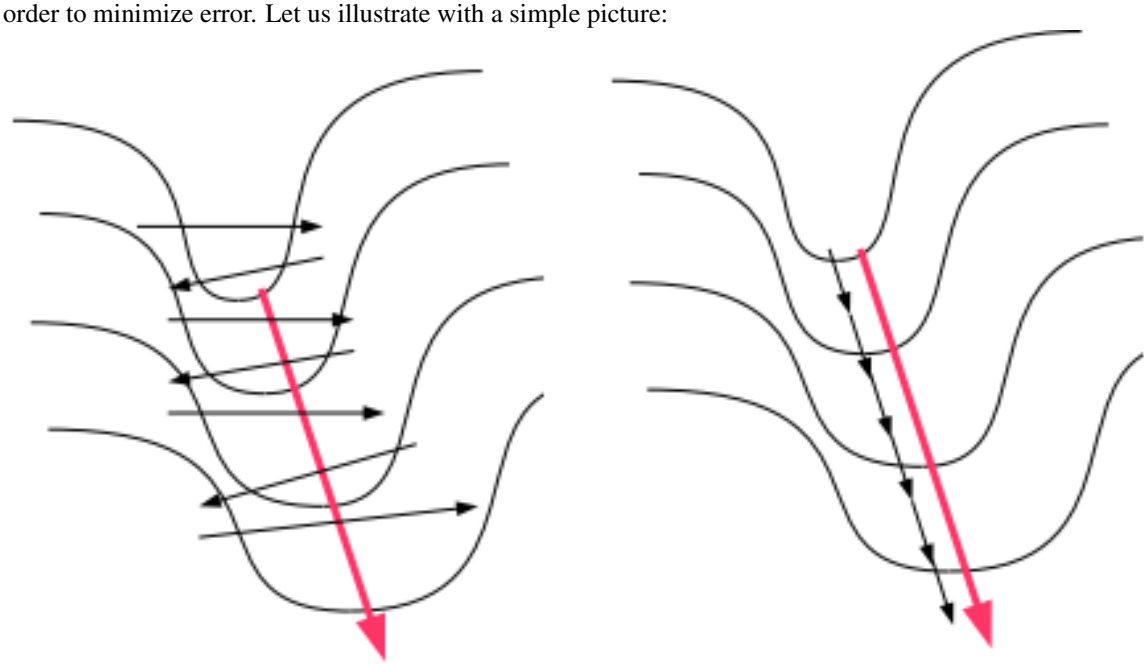
This is the basic idea behind Stochastic Gradient Descent (SGD): Go over the training set instance by instance (or minibatch by minibatch). Run the backpropagation algorithm to calculate the error gradients. Update the weights and biases in the opposite direction of these gradients. Rinse and repeat...

Over the years, people have noted many subtle problems with this approach and suggested improvements:

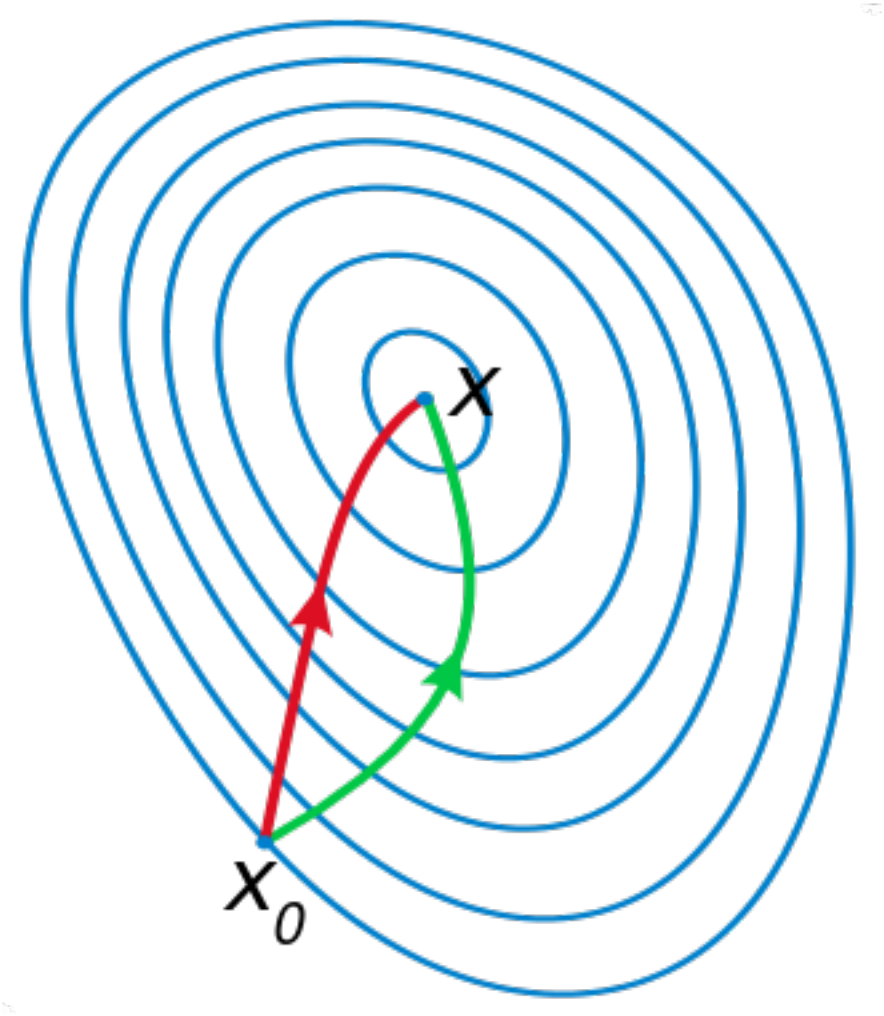
Step size: If the step sizes are too small, the SGD algorithm will take too long to converge. If they are too big it will overshoot the optimum and start to oscillate. So we scale the gradients with an adjustable parameter called the learning rate η:

$$w \leftarrow w - \eta \nabla_w J$$

Step direction: More importantly, it turns out the gradient (or its opposite) is often NOT the direction you want to go in order to minimize error. Let us illustrate with a simple picture:



The figure on the left shows what would happen if you stood on one side of the long narrow valley and took the direction of steepest descent: this would point to the other side of the valley and you would end up moving back and forth between the two sides, instead of taking the gentle incline down as in the figure on the right. The direction across the valley has a high gradient but also a high curvature (second derivative) which means the descent will be sharp but short lived. On the other hand the direction following the bottom of the valley has a smaller gradient and low curvature, the descent will be slow but it will continue for a longer distance. Newton's method adjusts the direction taking into account the second derivative:



In this figure, the two axes are w1 and w2, two parameters of our network, and the contour plot represents the error with a minimum at x. If we start at x0, the Newton direction (in red) points almost towards the minimum, whereas the gradient (in green), perpendicular to the contours, points to the right.

Unfortunately Newton's direction is expensive to compute. However, it is also probably unnecessary for several reasons: (1) Newton gives us the ideal direction for second degree objective functions, which our objective function almost certainly is not, (2) The error function whose gradient backprop calculated is the error for the last minibatch/instance only, which at best is a very noisy approximation of the real error function, thus we shouldn't spend too much effort trying to get the direction exactly right.

So people have come up with various approximate methods to improve the step direction. Instead of multiplying each component of the gradient with the same learning rate, these methods scale them separately using their running average (momentum, Nesterov), or RMS (Adagrad, Rmsprop). Some even cap the gradients at an arbitrary upper limit (gradient clipping) to prevent unstabilities.

You may wonder whether these methods still give us directions that consistently increase/decrease the objective function. If we do not insist on the *maximum* increase, any direction whose components have the same signs as the gradient vector is guaranteed to increase the function (for short enough steps). The reason is again given by the dot product $\nabla J \cdot v$. As long as these two vectors carry the same signs in the same components, the dot product, i.e. the rate of change along v, is guaranteed to be positive.

Minimize what? The final problem with gradient descent, other than not telling us the ideal step size or direction, is that it is not even minimizing the right objective! We want small error on never before seen test data, not just on the

training data. The truth is, a sufficiently large model with a good optimization algorithm can get arbitrarily low error on any finite training data (e.g. by just memorizing the answers). And it can typically do so in many different ways (typically many different local minima for training error in weight space exist). Some of those ways will generalize well to unseen data, some won't. And unseen data is (by definition) not seen, so how will we ever know which weight settings will do well on it?

There are at least three ways people deal with this problem: (1) Bayes tells us that we should use all possible models and weigh their answers by how well they do on training data (see Radford Neal's fbm), (2) New methods like dropout that add distortions and noise to inputs, activations, or weights during training seem to help generalization, (3) Pressuring the optimization to stay in one corner of the weight space (e.g. L1, L2, maxnorm regularization) helps generalization.

3.6 References

- http://ufldl.stanford.edu/tutorial/supervised/LinearRegression

CHAPTER 4

Softmax Classification

Note: **Concepts:** classification, likelihood, softmax, one-hot vectors, zero-one loss, conditional likelihood, MLE, NLL, cross-entropy loss

We will introduce classification problems and some simple models for classification.

4.1 Classification

Classification problems are supervised machine learning problems where the task is to predict a discrete class for a given input (unlike regression where the output was numeric). A typical example is handwritten digit recognition where the input is an image of a handwritten digit, and the output is one of the discrete categories $\{0, \ldots, 9\}$. As in all supervised learning problems the training data consists of a set of example input-output pairs.

4.2 Likelihood

A natural objective in classification could be to minimize the number of misclassified examples in the training data. This number is known as the **zero-one loss**. However the zero-one loss has some undesirable properties for training: in particular it is discontinuous. A small change in one of the parameters either has no effect on the loss, or can turn one or more of the predictions from false to true or true to false, causing a discontinuous jump in the objective. This means the gradient of the zero-one loss with respect to the parameters is either undefined or not helpful.

A more commonly used objective for classification is conditional likelihood: the probability of the observed data given our model *and the inputs*. Instead of predicting a single class for each instance, we let our model predict a probability distribution over all classes. Then we adjust the weights of the model to increase the probabilities for the correct classes and decrease it for others. This is also known as the **maximum likelihood estimation** (MLE).

Let $\mathcal{X} = \{x_1, \ldots, x_N\}$ be the inputs in the training data, $\mathcal{Y} = \{y_1, \ldots, y_N\}$ be the correct classes and θ be the parameters of our model. Conditional likelihood is:

$$L(\theta) = P(\mathcal{Y}|\mathcal{X}, \theta) = \prod_{n=1}^{N} P(y_n|x_n, \theta)$$

The second equation assumes that the data instances were generated independently. We usually work with log likelihood for mathematical convenience: log is a monotonically increasing function, so maximizing likelihood is the same

as maximizing log likelihood:

$$\ell(\theta) = \log P(\mathcal{Y}|\mathcal{X}, \theta) = \sum_{n=1}^{N} \log P(y_n|x_n, \theta)$$

We will typically use the negative of ℓ (machine learning people like to minimize), which is known as **negative log likelihood** (NLL), or **cross-entropy loss** (`softloss` in Knet).

4.3 Softmax

The linear regression model we have seen earlier produces unbounded y values. To go from arbitrary values $y \in \mathbb{R}^C$ to normalized probability estimates $p \in \mathbb{R}^C$ for a single instance, we use exponentiation and normalization:

$$p_i = \frac{\exp y_i}{\sum_{c=1}^{C} \exp y_c}$$

where $i, c \in \{1, \ldots, C\}$ range over classes, and p_i, y_i, y_c refer to class probabilities and values for a single instance. This is called the **softmax function** (`soft` operation in Knet). A model that converts the unnormalized values at the end of a linear regression to normalized probabilities for classification is called the **softmax classifier**.

We need to figure out the backward pass for the softmax function. In other words if someone gives us the gradient of some objective J with respect to the class probabilities p for a single training instance, what is the gradient with respect to the input of the softmax y? First we'll find the partial derivative of one component of p with respect to one component of y:

$$\frac{\partial p_i}{\partial y_j} = \frac{[i=j] \exp y_i \sum_c \exp y_c - \exp y_i \exp y_j}{(\sum_c \exp y_c)^2} = [i=j]\, p_i - p_i p_j$$

The square brackets are the Iverson bracket notation, i.e. $[A]$ is 1 if A is true, and 0 if A is false.

Note that a single entry in y effects J through multiple paths (y_j contributes to the denominator of every p_i), and these effects need to be added for $\partial J / \partial y_j$:

$$\frac{\partial J}{\partial y_j} = \sum_{i=1}^{C} \frac{\partial J}{\partial p_i} \frac{\partial p_i}{\partial y_j}$$

4.4 One-hot vectors

When using a probabilistic classifier, it is convenient to represent the desired output as a **one-hot vector**, i.e. a vector in which all entries are '0' except a single '1'. If the correct class is $c \in \{1, \ldots, C\}$, we represent this with a one-hot vector $p \in \mathbb{R}^C$ where $p_c = 1$ and $p_{i \neq c} = 0$. Note that p can be viewed as a probability vector where all the probability mass is concentrated at c. This representation also allows us to have probabilistic targets where there is not a single answer but target probabilities associated with each answer. Given a one-hot (or probabilistic) p, and the model prediction $\hat{p}$, we can write the log-likelihood for a single instance as:

$$\ell = \sum_{c=1}^{C} p_c \log \hat{p}_c$$

4.5 Gradient of log likelihood

To compute the gradient for log likelihood, we need to make the normalization of $\hat{p}$ explicit:

$$\ell = \sum_c p_c \log \frac{\hat{p}_c}{\sum_k \hat{p}_k}$$

$$= \sum_c p_c \log \hat{p}_c - \sum_c p_c \log \sum_k \hat{p}_k$$

$$= (\sum_c p_c \log \hat{p}_c) - (\log \sum_k \hat{p}_k)$$

$$\frac{\partial \ell}{\partial \hat{p}_i} = \frac{p_i}{\hat{p}_i} - \frac{1}{\sum_k \hat{p}_k} = \frac{p_i}{\hat{p}_i} - 1$$

The gradient with respect to unnormalized y takes a particularly simple form:

$$\frac{\partial \ell}{\partial y_j} = \sum_i \frac{\partial \ell}{\partial \hat{p}_i} \frac{\partial \hat{p}_i}{\partial y_j}$$

$$= \sum_i (\frac{p_i}{\hat{p}_i} - 1)(\,[i=j]\,\hat{p}_i - \hat{p}_i \hat{p}_j)$$

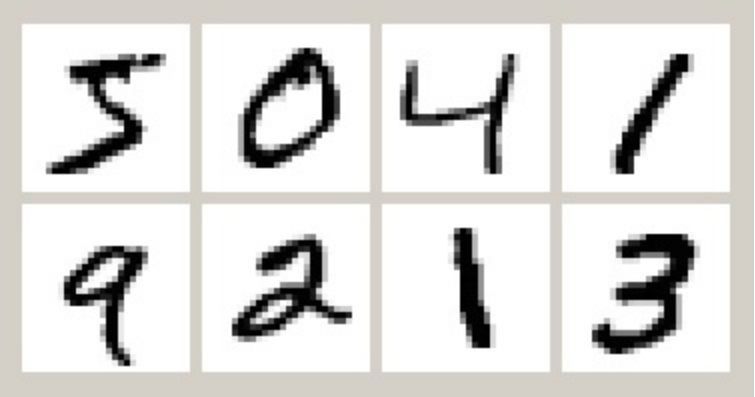
$$= p_j - \hat{p}_j$$

$$\nabla \ell = p - \hat{p}$$

The gradient with respect to $\hat{p}$ causes numerical overflow when some components of $\hat{p}$ get very small. In practice we usually skip that and directly compute the gradient with respect to y which is numerically stable.

4.6 MNIST example

Let's try our softmax classifier on the MNIST handwritten digit classification dataset. Here are the first 8 images from MNIST, the goal is to look at the pixels and classify each image as one of the digits 0-9:



See *5. Training with minibatches* for more information about the MNIST task, loading and minibatching data, and simple train and test scripts.

Here is a softmax classifier in Knet:

```
@knet function mnist_softmax(x)
    w = par(init=Gaussian(0,0.001), dims=(10,28*28))
    b = par(init=Constant(0), dims=(10,1))
    y = w * x + b
    return soft(y)
end
```

We will compile our model and set an appropriate learning rate:

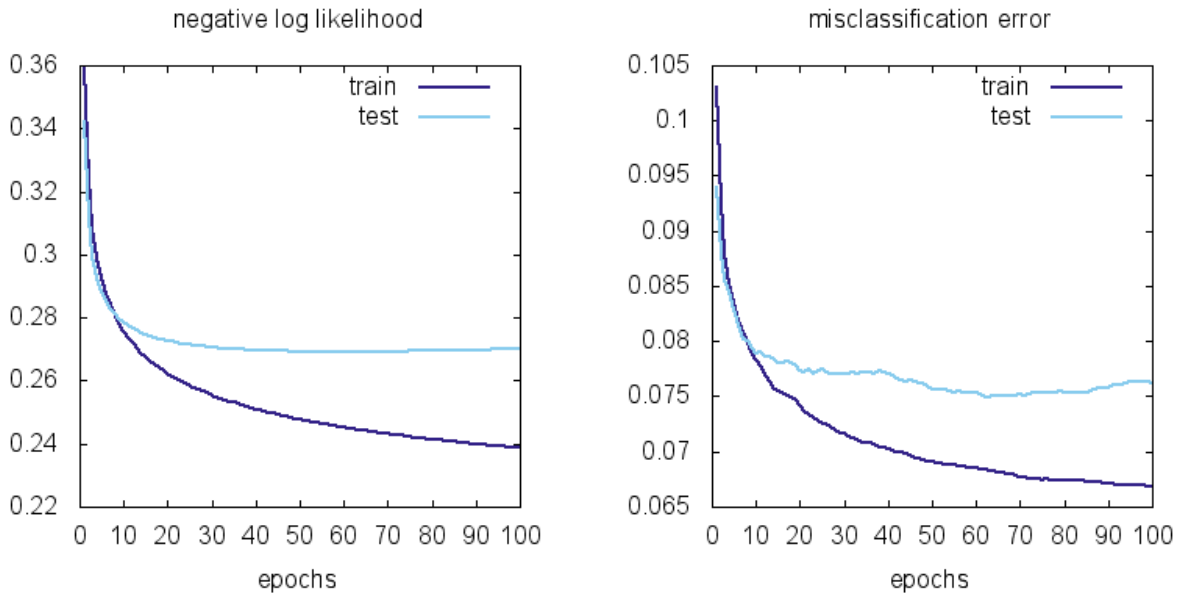
```
julia> model = compile(:mnist_softmax);
julia> setp(model; lr=0.15);
```

Let us train our model for 100 epochs and print out the negative log likelihood (`softloss`) and classification error (`zeroone`) on the training and testing sets after every epoch:

```
for epoch=1:nepochs
    train(model, dtrn, softloss)
    @printf("epoch:%d softloss:%g/%g zeroone:%g/%g\n", epoch,
            test(model, dtrn, softloss),
            test(model, dtst, softloss),
            test(model, dtrn, zeroone),
            test(model, dtst, zeroone))
end
```

```
epoch:1 softloss:0.359311/0.342333 zeroone:0.103033/0.094
epoch:2 softloss:0.32429/0.311829 zeroone:0.0924667/0.088
...
epoch:99 softloss:0.238815/0.270058 zeroone:0.0668667/0.0763
epoch:100 softloss:0.238695/0.270091 zeroone:0.0668333/0.0762
```

Here is a plot of the losses vs training epochs:



We can observe a few things. First the training losses are better than the test losses. This means there is some **overfitting**. Second, it does not look like the training loss is going down to zero. This means the softmax model is not flexible enough to fit the training data exactly.

4.7 Representational power

So far we have seen how to create a machine learning model as a differentiable program (linear regression, softmax classification) whose parameters can be adjusted to hopefully imitate whatever process generated our training data. A natural question to ask is whether a particular model can behave like any system we want (given the right parameters) or whether there is a limit to what it can represent.

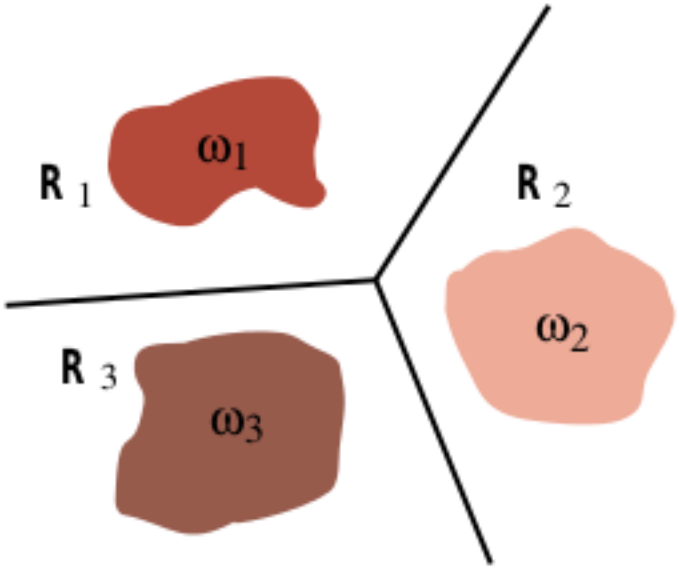
It turns out the softmax classifier is quite limited in its representational power: it can only represent linear classification boundaries. To show this, remember the form of the softmax classifier which gives the probability of the i'th class as:

$$p_i = \frac{\exp y_i}{\sum_{c=1}^{C} \exp y_c}$$

where y_i is a linear function of the input x. Note that p_i is a monotonically increasing function of y_i, so for two classes i and j, $p_i > p_j$ if $y_i > y_j$. The boundary between two classes i and j is the set of inputs for which the probability of the two classes are equal:

$$\begin{aligned} p_i &= p_j \\ y_i &= y_j \\ w_i x + b_i &= w_j x + b_j \\ (w_i - w_j)x + (b_i - b_j) &= 0 \end{aligned}$$

where w_i, b_i refer to the i'th row of w and b. This is a linear equation, i.e. the border between two classes will always be linear in the input space with the softmax classifier:



In the MNIST example, the relation between the pixels and the digit classes is unlikely to be this simple. That is why we are stuck at 6-7% training error. To get better results we need more powerful models.

4.8 References

- http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression

CHAPTER 5

Multilayer Perceptrons

In this section we create multilayer perceptrons by stacking multiple linear layers with non-linear activation functions in between.

5.1 Stacking linear classifiers is useless

We could try stacking multiple linear classifiers together. Here is a two layer model:

```
@knet function mnist_softmax_2(x)
    w1 = par(init=Gaussian(0,0.001), dims=(100,28*28))
    b1 = par(init=Constant(0), dims=(100,1))
    y1 = w1 * x + b1
    w2 = par(init=Gaussian(0,0.001), dims=(10,100))
    b2 = par(init=Constant(0), dims=(10,1))
    return soft(w2 * y1 + b2)
end
```

Note that instead of outputting the softmax of `y1`, we used it as input to another softmax classifier. Intermediate arrays like `y1` are known as **hidden layers** because their contents are not directly visible outside the model.

If you experiment with this model (I suggest using a smaller learning rate, e.g. 0.01), you will see that it performs similarly to the original softmax model. The reason is simple to see if we write the function computed in mathematical notation and do some algebra:

$$\begin{aligned} \hat{p} &= \\ \text{soft}(W_2(W_1 x + b_1) + b_2) &= \\ \text{soft}((W_2 W_1)\, x + W_2 b_1 + b_2) &= \\ \text{soft}(W x + b) & \end{aligned}$$

where $W = W_2 W_1$ and $b = W_2 b_1 + b_2$. In other words, we still have a linear classifier! No matter how many linear functions you put on top of each other, what you get at the end is still a linear function. So this model has exactly the same representation power as the softmax model. Unless, we add a simple instruction...

5.2 Introducing nonlinearities

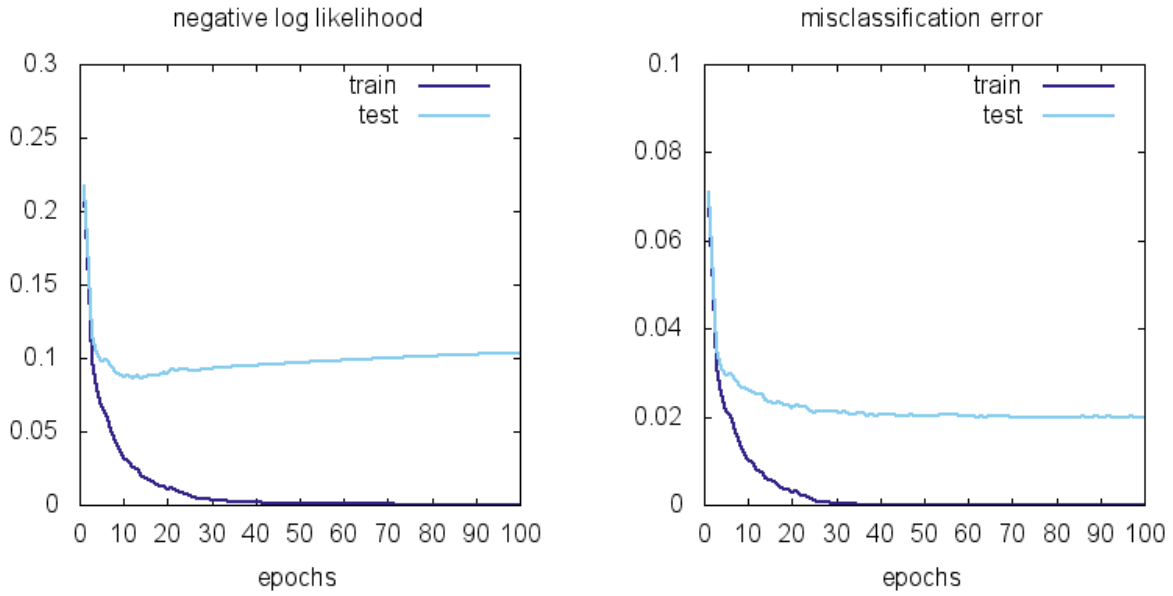
Here is a slightly modified version of the two layer model:

```
@knet function mnist_mlp(x)
    w1 = par(init=Gaussian(0,0.001), dims=(100,28*28))
    b1 = par(init=Constant(0), dims=(100,1))
    y1 = relu(w1 * x + b1)
    w2 = par(init=Gaussian(0,0.001), dims=(10,100))
    b2 = par(init=Constant(0), dims=(10,1))
    return soft(w2 * y1 + b2)
end
```

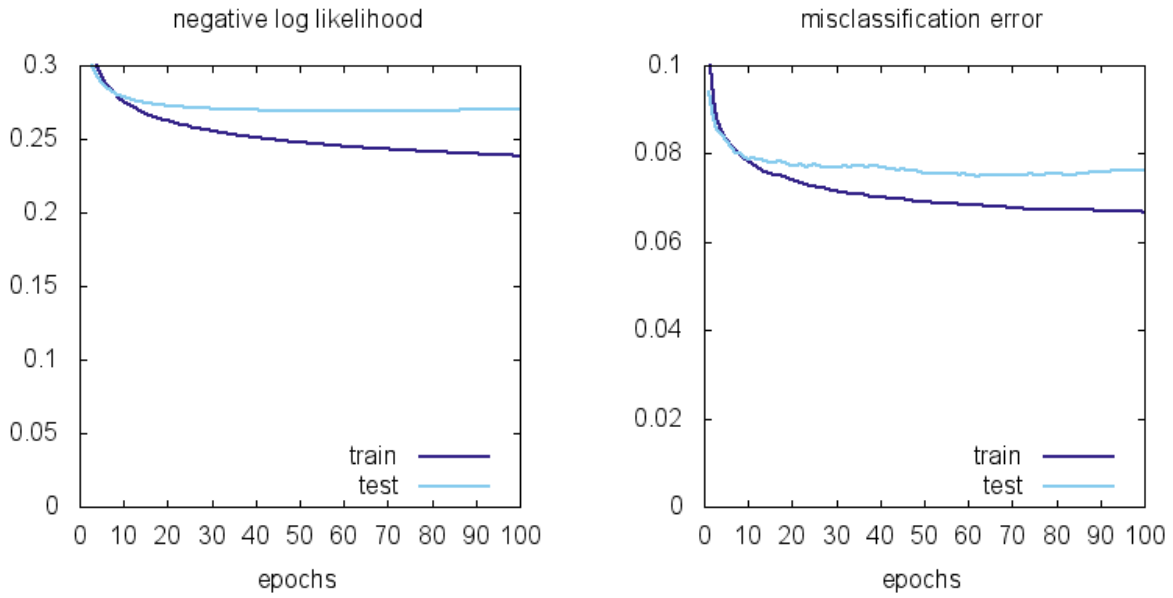
MLP in `mnist_mlp` stands for **multilayer perceptron** which is one name for this type of model. The only difference with the previous example is the `relu` function we introduced in line 4. This is known as the rectified linear unit (or rectifier), and is a simple function defined by `relu(x)=max(x,0)` applied elementwise to the input array. So mathematically what we are computing is:

$$\hat{p} = \text{soft}(W_2\,\text{relu}(W_1 x + b_1) + b_2)$$

This cannot be reduced to a linear function, which may not seem like a big difference but what a difference it makes to the model! Here are the learning curves for `mnist_mlp`:



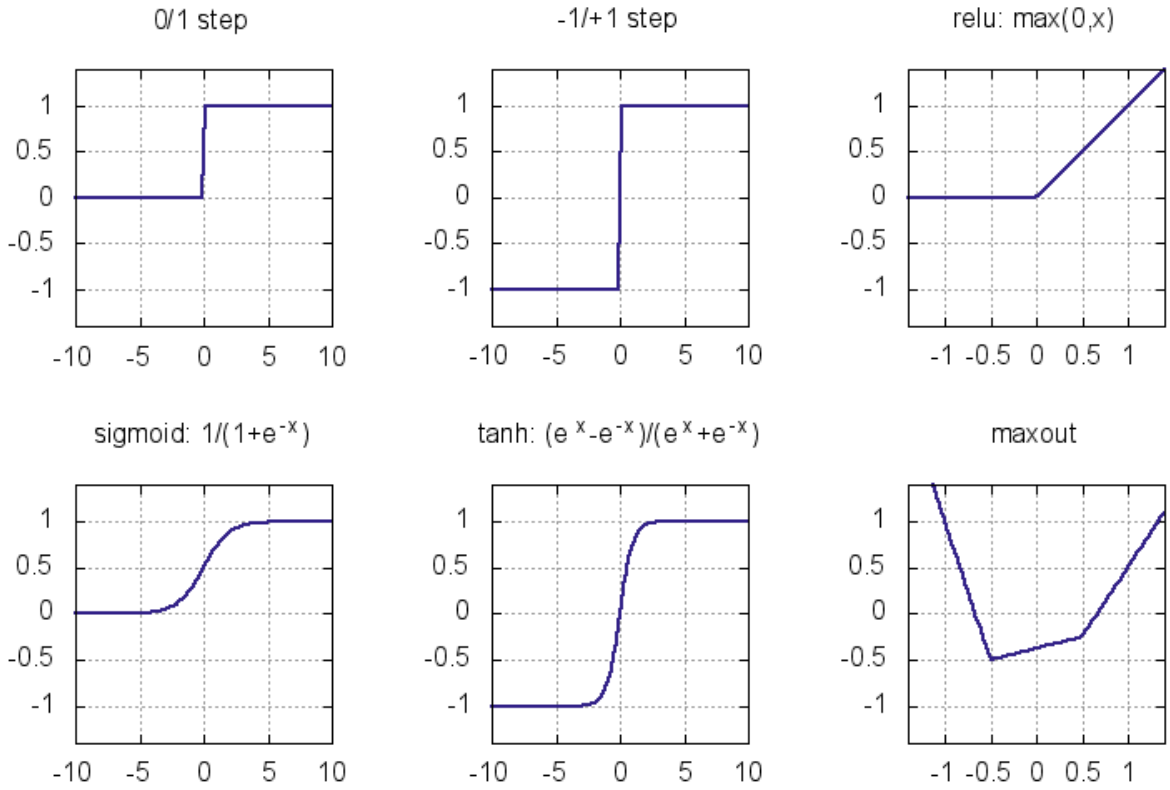
Here are the learning curves for the linear model `mnist_softmax` plotted at the same scale for comparison:



We can observe a few things: using MLP instead of a linear model brings the training error from 6.7% to 0 and the test error from 7.5% to 2.0%. There is still overfitting: the test error is not as good as the training error, but the model has no problem classifying the training data (all 60,000 examples) perfectly!

5.3 Types of nonlinearities (activation functions)

The functions we throw between linear layers to break the linearity are called **nonlinearities** or **activation functions**. Here are some activation functions that have been used as nonlinearities:



The step functions were the earliest activation functions used in the perceptrons of 1950s. Unfortunately they do not give a useful derivative that can be used for training a multilayer model. Sigmoid and tanh (`sigm` and `tanh` in Knet) became popular in 1980s as smooth approximations to the step functions and allowed the application of the backpropagation algorithm. Modern activation functions like relu and maxout are piecewise linear. They are computationally inexpensive (no exponentials), and perform well in practice. We are going to use relu in most of our models. Here is the backward passes for sigmoid, tanh, and relu:

function	forward	backward
sigmoid	$y = \frac{1}{1+e^{-x}}$	$\nabla_x J = y\,(1-y)\nabla_y J$
tanh	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$\nabla_x J = (1+y)(1-y)\nabla_y J$
relu	$y = \max(0, x)$	$\nabla_x J = [y \geq 0]\nabla_y J$

See (Karpathy, 2016, Ch 1) for more on activation functions and MLP architecture.

5.4 Representational power

You might be wondering whether `relu` had any special properties or would any of the other nonlinearities be sufficient. Another question is whether there are functions multilayer perceptrons cannot represent and if so whether adding more layers or different types of functions would increase their representational power. The short answer is that a two layer model can approximate any function if the hidden layer is large enough, and can do so with any of the nonlinearities introduced in the last section. Multilayer perceptrons are universal function approximators!

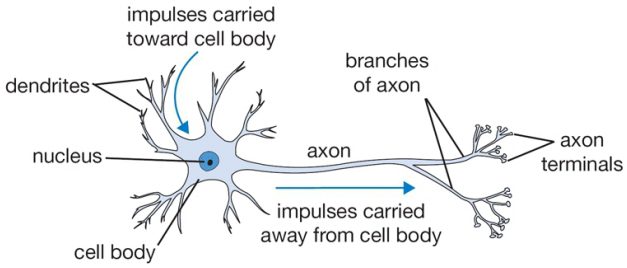
We said that a two-layer MLP is a universal function approximator *given enough hidden units*. This brings up the questions of efficiency: how many hidden units / parameters does one need to approximate a given function and whether the number of units depends on the number of hidden layers. The efficiency is important both computationally and statistically: models with fewer parameters can be evaluated faster, and can learn from fewer examples (ref?). It turns out there are functions whose representations are *exponentially more expensive* in a shallow network compared

to a deeper network (see (Nielsen, 2016, Ch 5) for a discussion). Recent winners of image recognition contests use networks with dozens of convolutional layers. The advantage of deeper MLPs is empirically less clear, but you should experiment with the number of units and layers using a development set when starting a new problem.

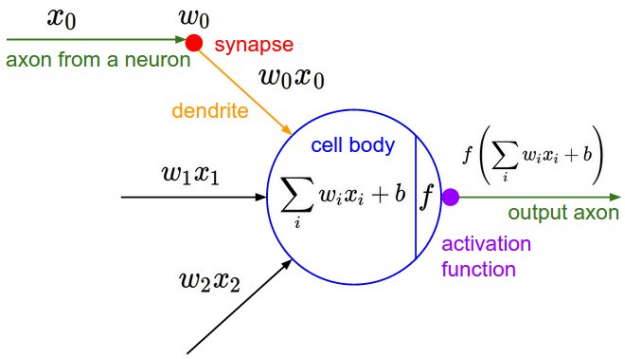
Please see (Nielsen, 2016, Ch 4) for an intuitive explanation of the universality result and (Bengio et al. 2016, Ch 6.4) for a more in depth discussion and references.

5.5 Matrix vs Neuron Pictures

So far we have introduced multilayer perceptrons (aka artificial neural networks) using matrix operations. You may be wondering why people call them neural networks and be confused by terms like layers and units. In this section we will give the correspondence between the matrix view and the neuron view. Here is a schematic of a biological neuron (figures from (Karpathy, 2016, Ch 1)):

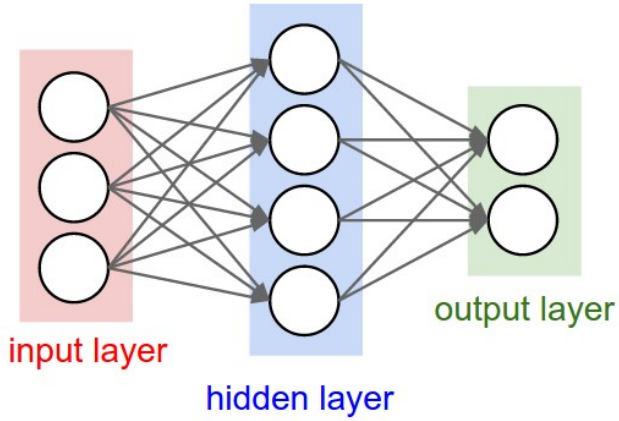


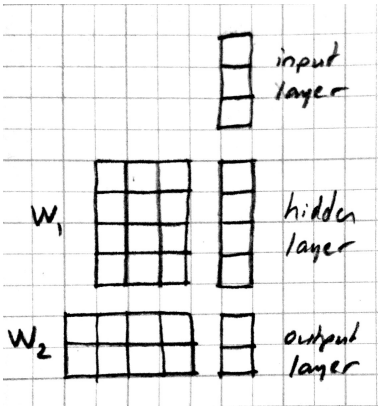
A biological neuron is a complex organism supporting thousands of chemical reactions simultaneously under the regulation of thousands of genes, communicating with other neurons through electrical and chemical pathways involving dozens of different types of neurotransmitter molecules. We assume (do not know for sure) that the main mechanism of communication between neurons is electrical spike trains that travel from the axon of the source neuron, through connections called synapses, into dendrites of target neurons. We simplify this picture further representing the strength of the spikes and the connections with simple numbers to arrive at this cartoon model:



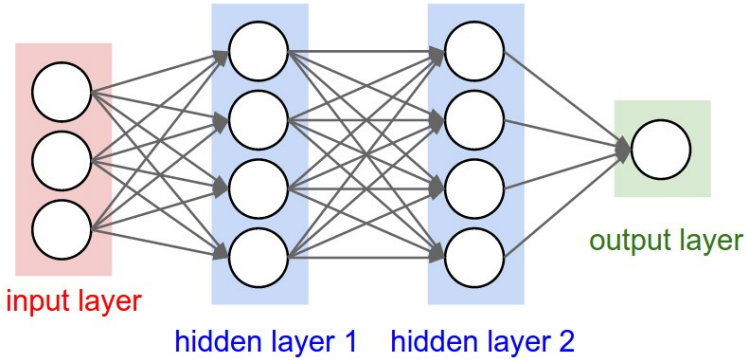
This model is called an artificial neuron, a perceptron, or simply a unit in neural network literature. We know it as the softmax classifier.

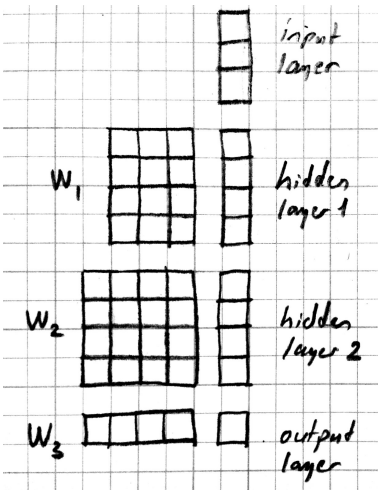
When a number of these units are connected in layers, we get a multilayer perceptron. When counting layers, we ignore the input layer. So the softmax classifier can be considered a one layer neural network. Here is a neural network picture and the corresponding matrix picture for a two layer model:





Here is a neural network picture and the corresponding matrix picture for a three layer model:





We can use the following elementwise notation for the neural network picture (e.g. similar to the one used in UFLDL):

$$x_i^{(l)} = f(b_i^{(l)} + \sum_j w_{ij}^{(l)} x_j^{(l-1)})$$

Here $x_i^{(l)}$ refers to the activation of the i th unit in l th layer. We are counting the input as the 0'th layer. f is the activation function, $b_i^{(l)}$ is the bias term. $w_{ij}^{(l)}$ is the weight connecting unit j from layer $l-1$ to unit i from layer l. The corresponding matrix notation is:

$$x^{(l)} = f(W^{(l)} x^{(l-1)} + b^{(l)})$$

5.6 Programming Example

In this section we introduce several Knet features that make it easier to define complex models. As our working example, we will go through several attempts to define a 3-layer MLP. Here is our first attempt:

```
@knet function mlp3a(x0)
    w1 = par(init=Gaussian(0,0.001), dims=(100,28*28))
    b1 = par(init=Constant(0), dims=(100,1))
    x1 = relu(w1 * x0 + b1)
    w2 = par(init=Gaussian(0,0.001), dims=(100,100))
    b2 = par(init=Constant(0), dims=(100,1))
    x2 = relu(w2 * x1 + b2)
    w3 = par(init=Gaussian(0,0.001), dims=(10,100))
    b3 = par(init=Constant(0), dims=(10,1))
    return soft(w3 * x2 + b3)
end
```

We can identify several bad software engineering practices in this definition:

- It contains a lot of repetition.
- It has a number of hardcoded parameters.

The key to controlling complexity in computer languages is **abstraction**. Abstraction is the ability to name compound structures built from primitive parts, so they too can be used as primitives. In Knet we do this by using @knet functions not as models, but as new operators inside other @knet functions.

Defining new operators

We could make the definition of mlp3 more compact by defining @knet functions for its layers:

```
@knet function mlp3b(x0)
    x1 = relu_layer1(x0)
    x2 = relu_layer2(x1)
    return soft_layer3(x2)
end

@knet function relu_layer1(x)
    w = par(init=Gaussian(0,0.001), dims=(100,28*28))
    b = par(init=Constant(0), dims=(100,1))
    return relu(w * x + b)
end

@knet function relu_layer2(x)
    w = par(init=Gaussian(0,0.001), dims=(100,100))
    b = par(init=Constant(0), dims=(100,1))
    return relu(w * x + b)
end

@knet function soft_layer3(x)
    w = par(init=Gaussian(0,0.001), dims=(10,100))
    b = par(init=Constant(0), dims=(10,1))
    return soft(w * x + b)
end
```

This may make the definition of `mlp3b` a bit more readable. But it does not reduce the overall length of the program. The helper @knet functions like `relu_layer1` contain hardcoded parameters like `dims` and are not reusable.

Using keyword arguments

We can make @knet functions more reusable by using keyword arguments that make them configurable. Here is a more compact definition of mlp3 using a single helper @knet function, `wbf` (mnemonic for $f(w * x + b)$):

```
@knet function mlp3c(x0)
    x1 = wbf(x0; f=:relu, inputs=28*28, outputs=100)
    x2 = wbf(x1; f=:relu, inputs=100, outputs=100)
    return wbf(x2; f=:soft, inputs=100, outputs=10)
end

@knet function wbf(x; f=:relu, inputs=0, outputs=0, winit=Gaussian(0,0.001), binit=Constant(0))
    w = par(init=winit, dims=(outputs,inputs))
    b = par(init=binit, dims=(outputs,1))
    return f(w * x + b)
end
```

Size inference

Knet can infer the size of an array based on the operations and other arrays it interacts with. In particular, when `forw(f,x)` is called Knet uses the size of the input `x` to figure out what size intermediate arrays to allocate when computing `f`. This allows us to define generic models and operators that work on inputs of any size. We still need to specify the number of outputs, but the number of inputs can be left unspecified. By convention 0 represents "unspecified" when declaring dimensions. Here is a more generic version of mlp3 that will work on images of any size:

```
@knet function mlp3d(x0)
    x1 = wbf(x0; f=:relu, out=100)
    x2 = wbf(x1; f=:relu, out=100)
    return wbf(x2; f=:soft, out=10)
end

@knet function wbf(x; f=:relu, out=0, winit=Gaussian(0,0.001), binit=Constant(0))
```

```
    w = par(init=winit, dims=(out,0))
    b = par(init=binit, dims=(out,1))
    return f(w * x + b)
end
```

Higher-order operators

Higher-order operators are ones that take other operators as arguments. We have already seen an example: `wbf` takes an operator `f` as one of its keyword arguments. A useful higher-order operator for multi-layer models is `repeat`, which repeats a given operator specified by `frepeat` configured by other keyword arguments a given number of times specified by `nrepeat`. Here is a definition of mlp3 using repeat:

```
@knet function mlp3e(x; o...)
    h = repeat(x; frepeat=:wbf, nrepeat=2, f=:relu, out=100, o...)
    return wbf(h; f=:soft, out=10)
end

@knet function wbf(x; f=:relu, out=0, winit=Gaussian(0,0.001), binit=Constant(0))
    w = par(init=winit, dims=(out,0))
    b = par(init=binit, dims=(out,1))
    return f(w * x + b)
end
```

In this example `repeat` saved us a single line, but the difference can be more significant in deeper models.

Built-in operators

In addition to primitive operators like `relu`, many compound operators such as `wbf` are already defined in Knet to make it easier to define complex models. Please see the tables of *primitive operators* and *compound operators* for a summary and kfun.jl for exact definitions.

5.7 References

- http://neuralnetworksanddeeplearning.com/chap4.html
- http://www.deeplearningbook.org/contents/mlp.html
- http://cs231n.github.io/neural-networks-1
- http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetwork
- http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch

CHAPTER 6

Convolutional Neural Networks

6.1 Motivation

Let's say we are trying to build a model that will detect cats in photographs. The average resolution of images in ILSVRC is 482×415, with three channels (RGB) this makes the typical input size $482 \times 415 \times 3 = 600,090$. Each hidden unit connected to the input in a multilayer perceptron would have 600K parameters, a single hidden layer of size 1000 would have 600 million parameters. Too many parameters cause two types of problems: (1) today's GPUs have limited amount of memory (4G-12G) and large networks fill them up quickly. (2) models with a large number of parameters are difficult to train without overfitting: we need a lot of data, strong regularization, and/or a good initialization to learn with large models.

One problem with the MLP is that it is fully connected: every hidden unit is connected to every input pixel. The model does not assume any spatial relationships between pixels, in fact we can permute all the pixels in an image and the performance of the MLP would be the same! We could instead have an architecture where each hidden unit is connected to a small patch of the image, say 40×40. Each such locally connected hidden unit would have $40 \times 40 \times 3 = 4800$ parameters instead of 600K. For the price (in memory) of one fully connected hidden unit, we could have 125 of these locally connected mini-hidden-units with receptive fields spread around the image.

The second problem with the MLP is that it does not take advantage of the symmetry in the problem: a cat in the lower right corner of the image is going to be similar to a cat in the lower left corner. This means the local hidden units looking at these two patches can share identical weights. We can take one 40×40 cat filter and apply it to each 40×40 patch in the image taking up only 4800 parameters.

A **convolutional neural network** (aka CNN or ConvNet) combines these two ideas and uses operations that are local and that share weights. CNNs commonly use three types of operations: convolution, pooling, and normalization which we describe next.

6.2 Convolution

Convolution in 1-D

Let w, x be two 1-D vectors with W, X elements respectively. In our examples, we will assume x is the input (consider it a 1-D image) and w is a filter (aka kernel) with $W < X$. The 1-D convolution operation $y = w * x$ results in a vector with $Y = X - W + 1$ elements defined as:

$$y_k \equiv \sum_{i+j=k+W} x_i w_j$$

or equivalently

$$y_k \equiv \sum_{i=k}^{k+W-1} x_i w_{k+W-i}$$

where $i \in [1, X], j \in [1, W], k \in [1, Y]$. We get each entry in y by multiplying pairs of matching entries in x and w and summing the results. Matching entries in x and w are the ones whose indices add up to a constant. This can be visualized as flipping w, sliding it over x, and at each step writing their dot product into a single entry in y. Here is an example in Knet you should be able to calculate by hand:

```
@knet function convtest1(x)
    w = par(init=reshape([1.0,2.0,3.0], (3,1,1,1)))
    y = conv(w, x)
    return y
end
julia> f = compile(:convtest1);
julia> x = reshape([1.0:7.0...], (7,1,1,1))
7x1x1x1 Array{Float64,4}: [1,2,3,4,5,6,7]
julia> y = forw(f,x)
5x1x1x1 Array{Float64,4}: [10,16,22,28,34]
```

`conv` is the convolution operation in Knet (based on the CUDNN implementation). For reasons that will become clear it works with 4-D and 5-D arrays, so we reshape our 1-D input vectors by adding extra singleton dimensions at the end. The convolution of w=[1,2,3] and x=[1,2,3,4,5,6,7] gives y=[10,16,22,28,34]. For example, the third element of y, 22, can be obtained by reversing w to [3,2,1] and taking its dot product starting with the third element of x, [3,4,5].

Padding

In the last example, the input x had 7 dimensions, the output y had 5. In image processing applications we typically want to keep x and y the same size. For this purpose we can provide a `padding` keyword argument to the `conv` operator. If padding=k, x will be assumed padded with k zeros on the left and right before the convolution, e.g. padding=1 means treat x as [0 1 2 3 4 5 6 7 0]. The default padding is 0. For inputs in D-dimensions we can specify padding with a D-tuple, e.g. `padding=(1,2)` for 2D, or a single number, e.g. `padding=1` which is shorthand for `padding=(1,1)`. The result will have $Y = X + 2P - W + 1$ elements where P is the padding size. Therefore to preserve the size of x when W=3 we should use padding=1.

```
@knet function convtest2(x)
    w = par(init=reshape([1.0,2.0,3.0], (3,1,1,1)))
    y = conv(w, x; padding=(1,0))
    return y
end
julia> f = compile(:convtest2);
julia> y = forw(f,x)
7x1x1x1 Array{Float64,4}: [4,10,16,22,28,34,32]
```

For example, to calculate the first entry of y, take the dot product of the inverted w, [3,2,1] with the first three elements of the padded x, [0 1 2]. You can see that in order to preserve the input size, $Y = X$, given a filter size W, the padding should be set to $P = (W - 1)/2$. This will work if W is odd.

Stride

In the preceding examples we shift the inverted w by one position after each dot product. In some cases you may want to skip two or more positions. The amount of skip is set by the `stride` keyword argument of the `conv` operation (the default stride is 1). In the following example we set stride to W such that the consecutive filter applications are non-overlapping:

```
@knet function convtest3(x)
    w = par(init=reshape([1.0,2.0,3.0], (3,1,1,1)))
    y = conv(w, x; padding=(1,0), stride=3)
```

```
    return y
end
julia> f = compile(:convtest3);
julia> y = forw(f,x)
3x1x1x1 Array{Float64,4}: [4,22,32]
```

Note that the output has the first, middle, and last values of the previous example, i.e. every third value is kept and the rest are skipped. In general if stride=S and padding=P, the size of the output will be:

$$Y = 1 + \left\lfloor \frac{X + 2P - W}{S} \right\rfloor$$

Mode

The convolution operation we have used so far flips the convolution kernel before multiplying it with the input. To take our first 1-D convolution example with

$$\begin{aligned} y_1 = \\ x_1 w_W + x_2 w_{W-1} + x_3 w_{W-2} + \dots \\ y_2 = \\ x_2 w_W + x_3 w_{W-1} + x_4 w_{W-2} + \dots \\ \dots \end{aligned}$$

We could also perform a similar operation without kernel flipping:

$$\begin{aligned} y_1 = \\ x_1 w_1 + x_2 w_2 + x_3 w_3 + \dots \\ y_2 = \\ x_2 w_1 + x_3 w_2 + x_4 w_3 + \dots \\ \dots \end{aligned}$$

This variation is called cross-correlation. The two modes are specified in Knet/CUDNN by specifying one of the following as the value of the `mode` keyword:

- `CUDNN_CONVOLUTION`
- `CUDNN_CROSS_CORRELATION`

This option would be important if we were hand designing our filters. However the mode does not matter for CNNs where the filters are learnt from data, the CNN will simply learn an inverted version of the filter if necessary.

More Dimensions

When the input x has multiple dimensions convolution is defined similarly. In particular the filter w has the same number of dimensions but typically smaller size. The convolution operation flips w in each dimension and slides it over x, calculating the sum of elementwise products at every step. The formulas we have given above relating the output size to the input and filter sizes, padding and stride parameters apply independently for each dimension.

Knet supports 2D and 3D convolutions. The inputs and the filters have two extra dimensions at the end which means we use 4D and 5D arrays for 2D and 3D convolutions. Here is a 2D convolution example:

```
@knet function convtest4(x)
    w = par(init=reshape([1.0:4.0...], (2,2,1,1)))
    y = conv(w, x)
    return y
end
julia> f = compile(:convtest4);
julia> x = reshape([1.0:9.0...], (3,3,1,1));
```

```
julia> y = forw(f,x);
julia> x
3x3x1x1 Array{Float64,4}:
[:, :, 1, 1] =
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
julia> get(f,:w)
2x2x1x1 Array{Float64,4}:
[:, :, 1, 1] =
 1.0  3.0
 2.0  4.0
julia> y
2x2x1x1 CudaArray{Float64,4}:
[:, :, 1, 1] =
 23.0  53.0
 33.0  63.0
```

To see how this result comes about, note that when you flip w in both dimensions you get:

```
4 2
3 1
```

Multiplying this elementwise with the upper left corner of x:

```
1 4
2 5
```

and adding the results gives you the first entry 23.

The padding and stride options work similarly in multiple dimensions and can be specified as tuples: padding=(1,2) means a padding width of 1 along the first dimension and 2 along the second dimension for a 2D convolution. You can use padding=1 as a shorthand for padding=(1,1).

Multiple filters

So far we have been ignoring the extra dimensions at the end of our convolution arrays. Now we are ready to put them to use. A D-dimensional input image is typically represented as a D+1 dimensional array with dimensions:

$$[X_1, \ldots, X_D, C]$$

The first D dimensions $X_1 \ldots X_D$ determine the spatial extent of the image. The last dimension C is the number of channels (aka slices, frames, maps, filters). The definition and number of channels is application dependent. We use C=3 for RGB images representing the intensity in three colors: red, green, and blue. For grayscale images we have a single channel, C=1. If you were developing a model for chess, we could have C=12, each channel representing the locations of a different piece type.

In an actual CNN we do not typically hand-code the filters. Instead we tell the network: "here are 1000 randomly initialized filters, you go ahead and turn them into patterns useful for my task." This means we usually work with banks of multiple filters simultaneously and GPUs have optimized operations for such filter banks. The dimensions of a typical filter bank are:

$$[W_1, \ldots, W_D, I, O]$$

The first D dimensions $W_1 \ldots W_D$ determine the spatial extent of the filters. The next dimension I is the number of input channels, i.e. the number of filters from the previous layer, or the number of color channels of the input image. The last dimension O is the number of output channels, i.e. the number of filters in this layer.

If we take an input of size $[X_1, \ldots, X_D, I]$ and apply a filter bank of size $[W_1, \ldots, W_D, I, O]$ using padding

$[P_1, \ldots, P_D]$ and stride $[S_1, \ldots, S_D]$ the resulting array will have dimensions:

$$[W_1, \ldots, W_D, I, O] * [X_1, \ldots, X_D, I] \Rightarrow [Y_1, \ldots, Y_D, O]$$
$$\text{where } Y_i = 1 + \left\lfloor \frac{X_i + 2P_i - W_i}{S_i} \right\rfloor$$

As an example let's start with an input image of 256×256 pixels and 3 RGB channels. We'll first apply 25 filters of size 5×5 and padding=2, then 50 filters of size 3×3 and padding=1, and finally 75 filters of size 3×3 and padding=1. Here are the dimensions we will get:

$$[256, 256, 3] * [5, 5, 3, 25] \Rightarrow [256, 256, 25]$$
$$[256, 256, 25] * [3, 3, 25, 50] \Rightarrow [256, 256, 50]$$
$$[256, 256, 50] * [3, 3, 50, 75] \Rightarrow [256, 256, 75]$$

Note that the number of input channels of the input data and the filter bank always match. In other words, a filter covers only a small part of the spatial extent of the input but all of its channel depth.

Multiple instances

In addition to processing multiple filters in parallel, we will want to implement CNNs with minibatching, i.e. process multiple inputs in parallel. A minibatch of D-dimensional images is represented as a D+2 dimensional array:

$$[X_1, \ldots, X_D, I, N]$$

where I is the number of channels as before, and N is the number of images in a minibatch. The convolution implementation in Knet/CUDNN use D+2 dimensional arrays for both images and filters. We used 1 for the extra dimensions in our first examples, in effect using a single channel and a single image minibatch.

If we apply a filter bank of size $[W_1, \ldots, W_D, I, O]$ to the minibatch given above the output size would be:

$$[W_1, \ldots, W_D, I, O] * [X_1, \ldots, X_D, I, N] \Rightarrow [Y_1, \ldots, Y_D, O, N]$$
$$\text{where } Y_i = 1 + \left\lfloor \frac{X_i + 2P_i - W_i}{S_i} \right\rfloor$$

If we used a minibatch size of 128 in the previous example with 256×256 images, the sizes would be:

$$[256, 256, 3, 128] * [5, 5, 3, 25] \Rightarrow [256, 256, 25, 128]$$
$$[256, 256, 25, 128] * [3, 3, 25, 50] \Rightarrow [256, 256, 50, 128]$$
$$[256, 256, 50, 128] * [3, 3, 50, 75] \Rightarrow [256, 256, 75, 128]$$

basically adding an extra dimension of 128 at the end of each data array.

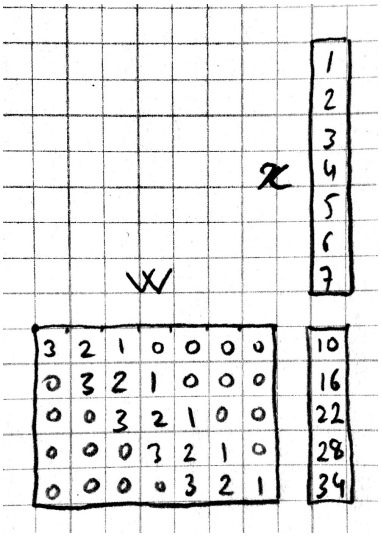
By the way, the arrays in this particular example already exceed 5GB of storage, so you would want to use a smaller minibatch size if you had a K20 GPU with 4GB of RAM.

Note: All the dimensions given above are for column-major languages like Knet. CUDNN uses row-major notation, so all the dimensions would be reversed, e.g. $[N, I, X_D, \ldots, X_1]$.

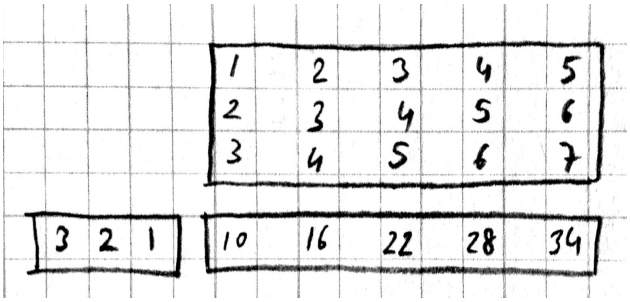
Convolution vs matrix multiplication

Convolution can be turned into a matrix multiplication, where certain entries in the matrix are constrained to be the same. The motivation is to be able to use efficient algorithms for matrix multiplication in order to perform convolution. The drawback is the large amount of memory needed due to repeated entries or sparse representations.

Here is a matrix implementation for our first convolution example $w = [1 \ldots 3]$, $x = [1 \ldots 7]$, $w * x = [10, 16, 22, 28, 34]$:

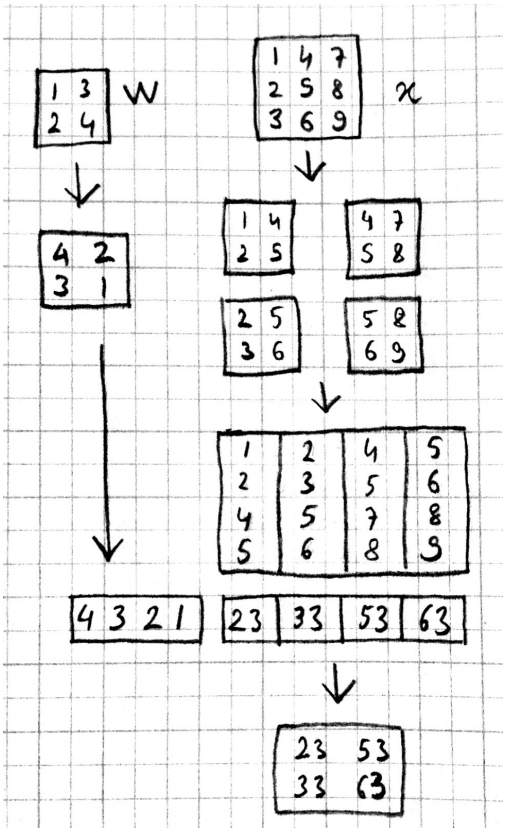


In this example we repeated the entries of the filter on multiple rows of a sparse matrix with shifted positions. Alternatively we can repeat the entries of the input to place each local patch on a separate column of an input matrix:



The first approach turns w into a $Y \times X$ sparse matrix, wheras the second turns x into a $W \times Y$ dense matrix.

For 2-D images, typically the second approach is used: the local patches of the image used by convolution are stretched out to columns of an input matrix, an operation commonly called `im2col`. Each convolutional filter is stretched out to rows of a filter matrix. After the matrix multiplication the resulting array is reshaped into the proper output dimensions. The following figure illustrates these operations on a small example:



It is also possible to go in the other direction, i.e. implement matrix multiplication (i.e. a fully connected layer) in terms of convolution. This conversion is useful when we want to build a network that can be applied to inputs of different sizes: the matrix multiplication would fail, but the convolution will give us outputs of matching sizes. Consider a fully connected layer with a weight matrix W of size $K \times D$ mapping a D-dimensional input vector x to a K-dimensional output vector y. We can consider each of the K rows of the W matrix a convolution filter. The following example shows how we can reshape the arrays and use convolution for matrix multiplication:

```
julia> using Knet, CUDNN
julia> x = reshape([1.0:3.0...], (3,1))
3x1 Array{Float64,2}:
 1.0
 2.0
 3.0
julia> w = reshape([1.0:6.0...], (2,3))
2x3 Array{Float64,2}:
 1.0  3.0  5.0
 2.0  4.0  6.0
julia> y = w * x
2x1 Array{Float64,2}:
 22.0
 28.0
julia> f = compile(:conv, mode=CUDNN_CROSS_CORRELATION);
julia> x2 = reshape(x, (3,1,1,1))
3x1x1x1 Array{Float64,4}:
[:, :, 1, 1] =
 1.0
 2.0
 3.0
julia> w2 = reshape(w', (3,1,1,2))
3x1x1x2 Array{Float64,4}:
```

```
[:, :, 1, 1] =
 1.0
 3.0
 5.0
[:, :, 1, 2] =
 2.0
 4.0
 6.0
julia> y2 = forw(f, w2, x2)
1x1x2x1 CudaArray{Float64,4}:
[:, :, 1, 1] =
 22.0
[:, :, 2, 1] =
 28.0
```

In addition to computational concerns, these examples also show that a fully connected layer can emulate a convolutional layer given the right weights and vice versa, i.e. convolution does not get us any extra representational power. However it does get us representational and statistical efficiency, i.e. the functions we would like to approximate are often expressed with significantly fewer parameters using convolutional layers and thus require fewer examples to train.

Backpropagation

Convolution is a linear operation consisting of additions and multiplications, so its backward pass is not very complicated except for the indexing. Just like the backward pass for matrix multiplication can be expressed as another matrix multiplication, the backward pass for convolution (at least if we use stride=1) can be expressed as another convolution. We will derive the backward pass for a 1-D example using the cross-correlation mode (no kernel flipping) to keep things simple. We will denote the cross-correlation operation with $\star$ to distinguish it from convolution denoted with $*$. Here are the individual entries of $y = w \star x$:

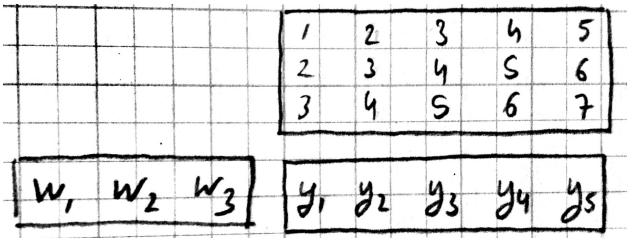
$$\begin{aligned} y_1 &= \\ x_1 w_1 + x_2 w_2 + x_3 w_3 + \dots & \\ y_2 &= \\ x_2 w_1 + x_3 w_2 + x_4 w_3 + \dots & \\ y_3 &= \\ x_3 w_1 + x_4 w_2 + x_5 w_3 + \dots & \\ \dots & \end{aligned}$$

As you can see, because of weight sharing the same w entry is used in computing multiple y entries. This means a single w entry effects the objective function through multiple paths and these effects need to be added. Denoting $\partial J / \partial y_i$ as y'_i for brevity we have:

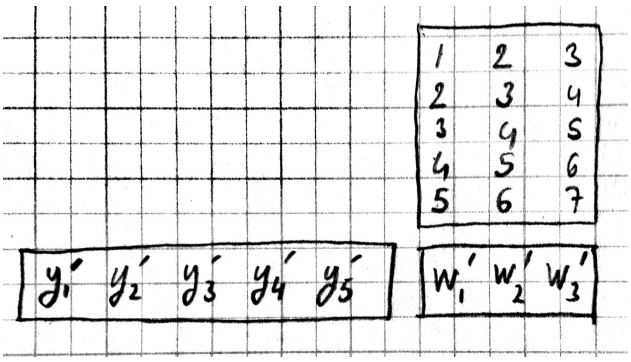
$$\begin{aligned} w'_1 &= \\ x_1 y'_1 + x_2 y'_2 + \dots & \\ w'_2 &= \\ x_2 y'_1 + x_3 y'_2 + \dots & \\ w'_3 &= \\ x_3 y'_1 + x_4 y'_2 + \dots & \\ \dots & \end{aligned}$$

which can be recognized as another cross-correlation operation, this time between x and y'. This allows us to write $w' = y' \star x$.

Alternatively, we can use the equivalent matrix multiplication operation from the last section to derive the backward pass:



If r is the matrix with repeated x entries in this picture, we have $y = wr$. Remember that the backward pass for matrix multiplication $y = wr$ is $w' = y'r^T$:



which can be recognized as the matrix multiplication equivalent of the cross correlation operation $w' = y' \star x$.

Here is the gradient for the input:

$$
\begin{aligned}
x'_1 &= \\
& w_1 y'_1 \\
x'_2 &= \\
& w_2 y'_1 + w_1 y'_2 \\
x'_3 &= \\
& w_3 y'_1 + w_2 y'_2 + w_1 y'_3 \\
& \dots
\end{aligned}
$$

You can recognize this as a regular convolution between w and y' with some zero padding.

The following resources provide more detailed derivations of the backward pass for convolution:

- Goodfellow, I. (2010). Technical report: Multidimensional, downsampled convolution for autoencoders. Technical report, Université de Montréal. 312.
- Bouvrie, J. (2006). Notes on convolutional neural networks.
- UFLDL tutorial and exercise on CNNs.

6.3 Pooling

It is common practice to use pooling (aka subsampling) layers in between convolution operations in CNNs. Pooling looks at small windows of the input, and computes a single summary statistic, e.g. maximum or average, for each window. A pooling layer basically says: tell me whether this feature exists in a certain region of the image, I don't care exactly where. This makes the output of the layer invariant to small translations of the input. Pooling layers use large strides, typically as large as the window size, which reduces the size of their output.

Like convolution, pooling slides a small window of a given size over the input optionally padded with zeros skipping stride pixels every step. In Knet by default there is no padding, the window size is 2, stride is equal to the window size and the pooling operation is max. These default settings reduce each dimension of the input to half the size.

Pooling in 1-D

Here is a 1-D example:

```
@knet function pooltest1(x)
    y = pool(x)
    return y
end
julia> f = compile(:pooltest1)
julia> x = reshape([1.0:6.0...], (6,1,1,1))
6x1x1x1 Array{Float64,4}: [1,2,3,4,5,6]
julia> forw(f,x)
3x1x1x1 CudaArray{Float64,4}: [2,4,6]
```

With window size and stride equal to 2, pooling considers the input windows $[1, 2], [3, 4], [5, 6]$ and picks the maximum in each window.

Window

The default and most commonly used window size is 2, however other window sizes can be specified using the `window` keyword. For D-dimensional inputs the size can be specified using a D-tuple, e.g. `window=(2,3)` for 2-D, or a single number, e.g. `window=3` which is shorthand for `window=(3,3)` in 2-D. Here is an example using a window size of 3 instead of the default 2:

```
@knet function pooltest2(x)
    y = pool(x; window=3)
    return y
end
julia> f = compile(:pooltest1)
julia> x = reshape([1.0:6.0...], (6,1,1,1))
6x1x1x1 Array{Float64,4}: [1,2,3,4,5,6]
julia> forw(f,x)
3x1x1x1 CudaArray{Float64,4}: [3,6]
```

With a window and stride of 3 (the stride is equal to window size by default), pooling considers the input windows $[1, 2, 3], [4, 5, 6]$, and writes the maximum of each window to the output. If the input size is X, and stride is equal to the window size W, the output will have $Y = \lceil X/W \rceil$ elements.

Padding

The amount of zero padding is specified using the `padding` keyword argument just like convolution. Padding is 0 by default. For D-dimensional inputs padding can be specified as a tuple such as `padding=(1,2)`, or a single number `padding=1` which is shorthand for `padding=(1,1)` in 2-D. Here is a 1-D example:

```
@knet function pooltest3(x)
    y = pool(x; padding=(1,0))
    return y
end
julia> f = compile(:pooltest3)
julia> x = reshape([1.0:6.0...], (6,1,1,1))
6x1x1x1 Array{Float64,4}: [1,2,3,4,5,6]
julia> forw(f,x)
3x1x1x1 CudaArray{Float64,4}: [1,3,5,6]
```

In this example, window=stride=2 by default and the padding size is 1, so the input is treated as $[0, 1, 2, 3, 4, 5, 6, 0]$ and split into windows of $[0, 1], [2, 3], [4, 5], [6, 0]$ and the maximum of each window is written to the output.

With padding size P, if the input size is X, and stride is equal to the window size W, the output will have $Y = \lceil (X + 2P)/W \rceil$ elements.

Stride

The pooling stride is equal to the window size by default (as opposed to the convolution case, where it is 1 by default). This is most common in practice but other strides can be specified using tuples e.g. `stride=(1,2)` or numbers e.g. `stride=1`.

In general, when we have an input of size X and pool with window size W, padding P, and stride S, the size of the output will be:

$$Y = 1 + \left\lceil \frac{X + 2P - W}{S} \right\rceil$$

Pooling operations

There are three pooling operations defined by CUDNN used for summarizing each window:

- `CUDNN_POOLING_MAX`
- `CUDNN_POOLING_AVERAGE_COUNT_INCLUDE_PADDING`
- `CUDNN_POOLING_AVERAGE_COUNT_EXCLUDE_PADDING`

These options can be specified as the value of the `mode` keyword argument to the `pool` operation. The default is `CUDNN_POOLING_MAX` which we have been using so far. The last two compute averages, and differ in whether to include or exclude the padding zeros in these averages. For example, with input $x = [1, 2, 3, 4, 5, 6]$, `window=stride=2`, and `padding=1` we have the following outputs with the three options:

```
mode=CUDNN_POOLING_MAX => [1,3,5,6]
mode=CUDNN_POOLING_AVERAGE_COUNT_INCLUDE_PADDING => [0.5, 2.5, 4.5, 3.0]
mode=CUDNN_POOLING_AVERAGE_COUNT_EXCLUDE_PADDING => [1.0, 2.5, 4.5, 6.0]
```

More Dimensions

D-dimensional inputs are pooled with D-dimensional windows, the size of each output dimension given by the 1-D formulas above. Here is a 2-D example with default options, i.e. window=stride=(2,2), padding=(0,0), mode=max:

```
@knet function pooltest1(x)
    y = pool(x)
    return y
end
julia> f = compile(:pooltest1)
julia> x = reshape([1.0:16.0...], (4,4,1,1))
4x4x1x1 Array{Float64,4}:
[:, :, 1, 1] =
 1.0  5.0   9.0  13.0
 2.0  6.0  10.0  14.0
 3.0  7.0  11.0  15.0
 4.0  8.0  12.0  16.0
julia> forw(f,x)
2x2x1x1 CudaArray{Float64,4}:
[:, :, 1, 1] =
 6.0  14.0
 8.0  16.0
```

Multiple channels and instances

As we saw in convolution, each data array has two extra dimensions in addition to the spatial dimensions: $[X_1, \ldots, X_D, I, N]$ where I is the number of channels and N is the number of instances in a minibatch.

When the number of channels is greater than 1, the pooling operation is performed independently on each channel, e.g. for each patch, the maximum/average in each channel is computed independently and copied to the output. Here is an example with two channels:

```
@knet function pooltest1(x)
    y = pool(x)
    return y
end
julia> f = compile(:pooltest1)
julia> x = rand(4,4,2,1)
4x4x2x1 Array{Float64,4}:
[:, :, 1, 1] =
 0.0235776   0.470246  0.829754  0.164617
 0.375611    0.884792  0.561758  0.955467
 0.00740115  0.76617   0.674633  0.480402
 0.979588    0.949825  0.449385  0.956657
[:, :, 2, 1] =
 0.254501  0.0930295  0.640946  0.270479
 0.422195  0.0399775  0.387326  0.234855
 0.102558  0.589408   0.69867   0.498438
 0.823076  0.797679   0.695289  0.888321
julia> forw(f,x)
2x2x2x1 CudaArray{Float64,4}:
[:, :, 1, 1] =
 0.884792  0.955467
 0.979588  0.956657
[:, :, 2, 1] =
 0.422195  0.640946
 0.823076  0.888321
```

When the number of instances is greater than 1, i.e. we are using minibatches, the pooling operation similarly runs in parallel on all the instances:

```
julia> x = rand(4,4,1,2)
4x4x1x2 Array{Float64,4}:
[:, :, 1, 1] =
 0.664524  0.581233   0.949937  0.563411
 0.760211  0.714199   0.985956  0.478583
 0.190559  0.682141   0.43941   0.682127
 0.701371  0.0159724  0.28857   0.166187

[:, :, 1, 2] =
 0.637187  0.279795  0.0336316  0.233479
 0.979812  0.910836  0.410312   0.94062
 0.171724  0.388222  0.597548   0.817148
 0.41193   0.864101  0.178535   0.4956

julia> forw(f,x)
2x2x1x2 CudaArray{Float64,4}:
[:, :, 1, 1] =
 0.760211  0.985956
 0.701371  0.682127

[:, :, 1, 2] =
 0.979812  0.94062
 0.864101  0.817148
```

6.4 Normalization

Draft...

Karpathy says: "Many types of normalization layers have been proposed for use in ConvNet architectures, sometimes with the intentions of implementing inhibition schemes observed in the biological brain. However, these layers have recently fallen out of favor because in practice their contribution has been shown to be minimal, if any." (http://cs231n.github.io/convolutional-networks/#norm) Batch normalization may be an exception, as it is used in modern architectures.

Here are some references for normalization operations:

Implementations:

- Alex Krizhevsky's cuda-convnet library API. (https://code.google.com/archive/p/cuda-convnet/wikis/LayerParams.wiki#Local_response_normalization_layer_(same_map))
- http://caffe.berkeleyvision.org/tutorial/layers.html
- http://lasagne.readthedocs.org/en/latest/modules/layers/normalization.html

Divisive normalisation (DivN):

- S. Lyu and E. Simoncelli. Nonlinear image representation using divisive normalization. In CVPR, pages 1–8, 2008.

Local contrast normalization (LCN):

- N. Pinto, D. D. Cox, and J. J. DiCarlo. Why is real-world visual object recognition hard? PLoS Computational Biology, 4(1), 2008.
- Jarrett, Kevin, et al. "What is the best multi-stage architecture for object recognition?." Computer Vision, 2009 IEEE 12th International Conference on. IEEE, 2009. (http://yann.lecun.com/exdb/publis/pdf/jarrett-iccv-09.pdf)

Local response normalization (LRN):

- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012. (http://machinelearning.wustl.edu/mlpapers/paper_files/NIPS2012_0534.pdf)

Batch Normalization: This is more of an optimization topic.

- Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015). (http://arxiv.org/abs/1502.03167/)

6.5 Architectures

We have seen a number of new operations: convolution, pooling, filters etc. How to best put these together to form a CNN is still an active area of research. In this section we summarize common patterns of usage in recent work based on (Karpathy, 2016).

- The operations in convolutional networks are usually ordered into several layers of convolution-bias-activation-pooling sequences (`cbfp` is the mnemonic used in Knet). Note that the convolution-bias-activation sequence is an efficient way to implement the common neural net function $f(wx + b)$ for a locally connected and weight sharing hidden layer.
- The convolutional layers are typically followed by a number of fully connected layers that end with a softmax layer for prediction (if we are training for a classification problem).

- It is preferrable to have multiple convolution layers with small filter sizes rather than a single layer with a large filter size. Consider three convolutional layers with a filter size of 3×3. The units in the top layer have receptive fields of size 7×7. Compare this with a single layer with a filter size of 7×7. The three layer architecture has two advantages: The units in the single layer network is restricted to linear decision boundaries, whereas the three layer network can be more expressive. Second, if we assume C channels, the parameter tensor for the single layer network has size $[7, 7, C, C]$ whereas the three layer network has three tensors of size $[3, 3, C, C]$ i.e. a smaller number of parameters. The one disadvantage of the three layer network is the extra storage required to store the intermediate results for backpropagation.
- Thus common settings for convolution use 3×3 filters with `stride = padding = 1` (which incidentally preserves the input size). The one exception may be a larger filter size used in the first layer which is applied to the image pixels. This will save memory when the input is at its largest, and linear functions may be sufficient to express the low level features at this stage.
- The pooling operation may not be present in every layer. Keep in mind that pooling destroys information and having several convolutional layers without pooling may allow more complex features to be learnt. When pooling is present it is best to keep the window size small to minimize information loss. The common settings for pooling are `window = stride = 2, padding = 0`, which halves the input size in each dimension.

Beyond these general guidelines, you should look at the architectures used by successful models in the literature. Some examples are LeNet (LeCun et al. 1998), AlexNet (Krizhevsky et al. 2012), ZFNet (Zeiler and Fergus, 2013), GoogLeNet (Szegedy et al. 2014), VGGNet (Simonyan and Zisserman, 2014), and ResNet (He et al. 2015).

6.6 Exercises

- Design a filter that shifts a given image one pixel to right.
- Design an image filter that has 0 output in regions of uniform color, but nonzero output at edges where the color changes.
- If your input consisted of two consecutive frames of video, how would you detect motion using convolution?
- Can you implement matrix-vector multiplication in terms of convolution? How about matrix-matrix multiplication? Do you need reshape operations?
- Can you implement convolution in terms of matrix multiplication?
- Can you implement elementwise broadcasting multiplication in terms of convolution?

6.7 References

- Some of this chapter was based on the excellent lecture notes from: http://cs231n.github.io/convolutional-networks
- Christopher Olah's blog has very good visual explanations (thanks to Melike Softa for the reference): http://colah.github.io/posts/2014-07-Conv-Nets-Modular
- UFLDL (or its old version) is an online tutorial with programming examples and explicit gradient derivations covering convolution, pooling, and CNNs.
- Hinton's video lecture and presentation at Coursera (Lec 5): https://d396qusza40orc.cloudfront.net/neuralnets/lecture_slides/lec5.p
- For a derivation of gradients see: http://people.csail.mit.edu/jvb/papers/cnn_tutorial.pdf or http://www.iro.umontreal.ca/~lisa/pointeurs/convolution.pdf
- The CUDNN manual has more details about the convolution API: https://developer.nvidia.com/cudnn
- http://deeplearning.net/tutorial/lenet.html

- http://www.denizyuret.com/2014/04/on-emergence-of-visual-cortex-receptive.html
- http://neuralnetworksanddeeplearning.com/chap6.html
- http://www.deeplearningbook.org/contents/convnets.html
- http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp
- http://scs.ryerson.ca/~aharley/vis/conv/ has a nice visualization of an MNIST CNN. (Thanks to Fatih Ozhamaratli for the reference).

CHAPTER 7

Recurrent Neural Networks

7.1 References

- https://d396qusza40orc.cloudfront.net/neuralnets/lecture_slides/lec7.pdf (coursera hinton)
- http://karpathy.github.io/2015/05/21/rnn-effectiveness/
- http://colah.github.io/posts/2015-08-Understanding-LSTMs
- http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns
- https://en.wikipedia.org/wiki/Recurrent_neural_network
- https://www.willamette.edu/~gorr/classes/cs449/rnn1.html
- http://www.deeplearningbook.org/contents/rnn.html
- http://cs224d.stanford.edu/ (socher class on deep learning for nlp)

CHAPTER 8

Reinforcement Learning

8.1 References

- http://karpathy.github.io/2016/05/31/rl/
- https://webdocs.cs.ualberta.ca/~sutton/book/the-book.html
- http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html
- http://videolectures.net/rldm2015_silver_reinforcement_learning/?q=david%20silver
- http://cs229.stanford.edu/notes/cs229-notes12.pdf
- http://cs.stanford.edu/people/karpathy/reinforcejs/index.html
- https://www.udacity.com/course/machine-learning-reinforcement-learning–ud820
- http://www.nature.com/nature/journal/v518/n7540/full/nature14236.html
- http://people.csail.mit.edu/regina/my_papers/TG15.pdf
- In http://karpathy.github.io/2015/05/21/rnn-effectiveness: For more about REINFORCE and more generally Reinforcement Learning and policy gradient methods (which REINFORCE is a special case of) David Silver's class, or one of Pieter Abbeel's classes. This is very much ongoing work but these hard attention models have been explored, for example, in Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets, Reinforcement Learning Neural Turing Machines, and Show Attend and Tell.
- In http://www.deeplearningbook.org/contents/ml.html: Please see Sutton and Barto (1998) or Bertsekasand Tsitsiklis (1996) for information about reinforcement learning, and Mnih et al.(2013) for the deep learning approach to reinforcement learning.

CHAPTER 9

Optimization

9.1 References

- http://www.deeplearningbook.org/contents/numerical.html (basic intro in 4.3)
- http://www.deeplearningbook.org/contents/optimization.html (8.1 generalization, 8.2 problems, 8.3 algorithms, 8.4 init, 8.5 adaptive lr, 8.6 approx 2nd order, 8.7 meta)
- http://andrew.gibiansky.com/blog/machine-learning/gauss-newton-matrix/ (great posts on optimization)
- https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf (excellent tutorial on cg, gd, eigens etc)
- http://arxiv.org/abs/1412.6544 (Goodfellow paper)
- https://d396qusza40orc.cloudfront.net/neuralnets/lecture_slides/lec6.pdf (hinton slides)
- https://d396qusza40orc.cloudfront.net/neuralnets/lecture_slides/lec8.pdf (hinton slides)
- http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html
- http://machinelearning.wustl.edu/mlpapers/paper_files/icml2010_Martens10.pdf
- http://arxiv.org/abs/1503.05671
- http://arxiv.org/abs/1412.1193
- http://www.springer.com/us/book/9780387303031 (nocedal and wright)
- http://www.nrbook.com (numerical recipes)
- https://maths-people.anu.edu.au/~brent/pub/pub011.html (without derivatives)
- http://stanford.edu/~boyd/cvxbook/ (only convex optimization)

CHAPTER 10

Generalization

10.1 References

- http://www.deeplearningbook.org/contents/regularization.html
- https://d396qusza40orc.cloudfront.net/neuralnets/lecture_slides/lec9.pdf
- https://d396qusza40orc.cloudfront.net/neuralnets/lecture_slides/lec10.pdf
- http://blog.cambridgecoding.com/2016/03/24/misleading-modelling-overfitting-cross-validation-and-the-bias-variance-trade-off/

CHAPTER 11

Indices and tables

- genindex
- modindex
- search