Knet.jl Documentation

Release 0.7.2

Deniz Yuret

February 10, 2017

Contents

Note: You are not using the most up to date version of the documentation. denizyuret.github.io/Knet.jl has the latest version.

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. To install Knet just use `Pkg.add("Knet")`:

```
$ julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.4.6 (2016-06-19 17:16 UTC)
 _/ |\__'_|_|_|\__'_|  |  Official http://julialang.org/ release
|__/                   |  x86_64-apple-darwin13.4.0

julia> Pkg.add("Knet")
```

Some Knet examples use additional packages such as ArgParse, GZip and JLD. These are not required by Knet, you can install them manually when needed using Pkg.add("PkgName").

Run `Pkg.build("Knet")` to recompile Knet after optional packages are installed and to compile the Knet GPU kernels at first installation if you have a GPU machine. To make sure everything has installed correctly, type `Pkg.test("Knet")` which should take a minute 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

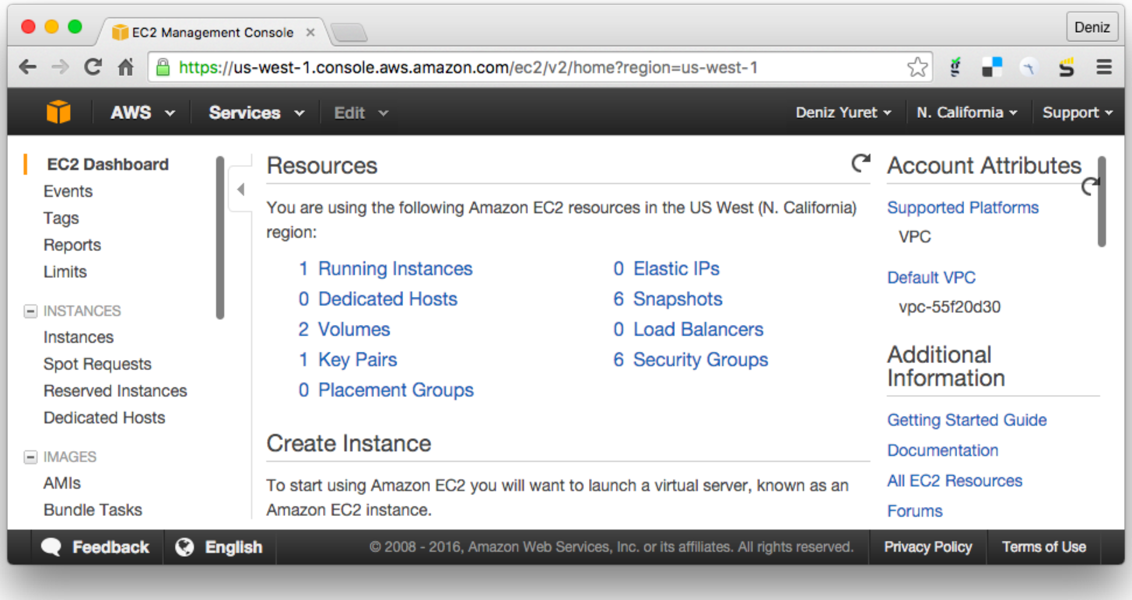
Knet is an open-source project and we are always open to new contributions: bug fixes, new machine learning models and operators, inspiring examples, benchmarking results are all welcome. If you'd like to contribute to the code base, please sign up at the [knet-dev](https://groups.google.com/forum/#!forum/knet-dev) mailing list and follow these 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:



Resources

You are using the following Amazon EC2 resources in the US West (N. California) region:

1 Running Instances	0 Elastic IPs
0 Dedicated Hosts	6 Snapshots
2 Volumes	0 Load Balancers
1 Key Pairs	6 Security Groups
0 Placement Groups	

Create Instance

To start using Amazon EC2 you will want to launch a virtual server, known as an Amazon EC2 instance.

Account Attributes

Supported Platforms
VPC

Default VPC
vpc-55f20d30

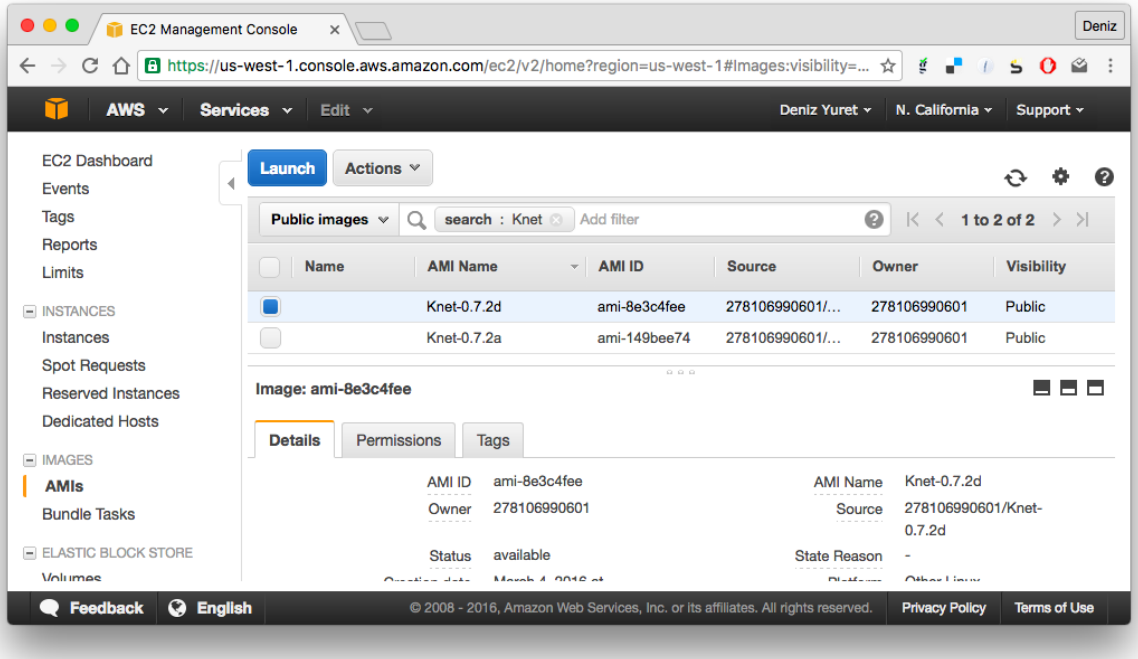
Additional Information

Getting Started Guide
Documentation
All EC2 Resources
Forums

© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.

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-0.8.0

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.



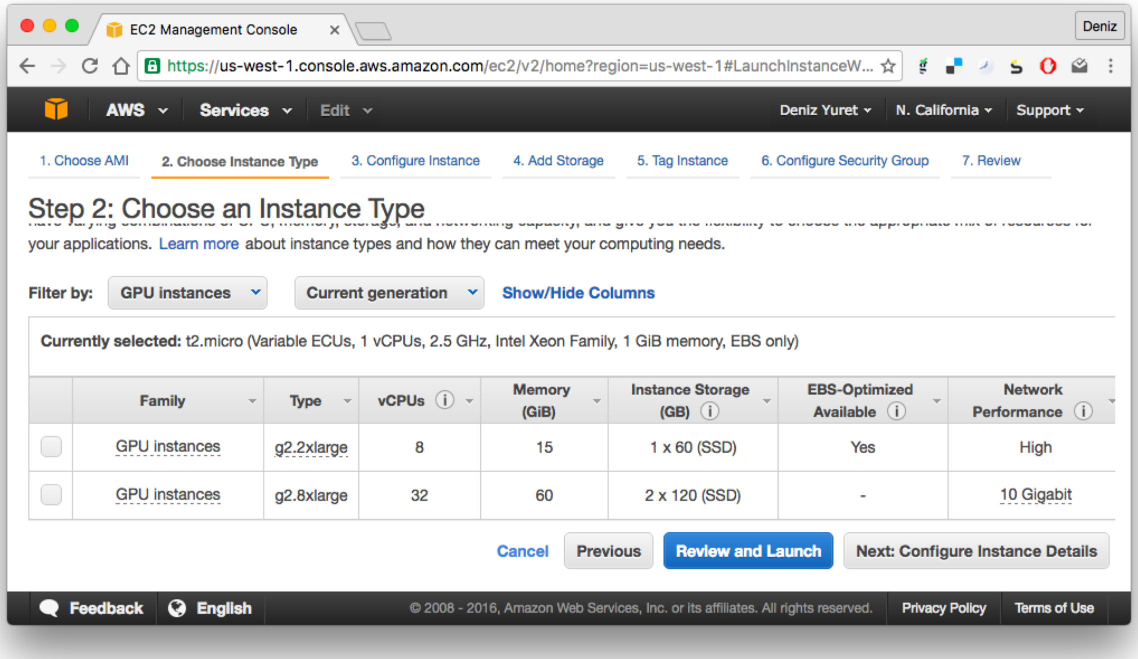
□	Name	AMI Name	AMI ID	Source	Owner	Visibility
☑		Knet-0.7.2d	ami-8e3c4fee	278106990601/...	278106990601	Public
□		Knet-0.7.2a	ami-149bee74	278106990601/...	278106990601	Public

Image: ami-8e3c4fee

AMI ID: ami-8e3c4fee
AMI Name: Knet-0.7.2d
Owner: 278106990601
Source: 278106990601/Knet-0.7.2d
Status: available
State Reason: -

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".



Step 2: Choose an Instance Type

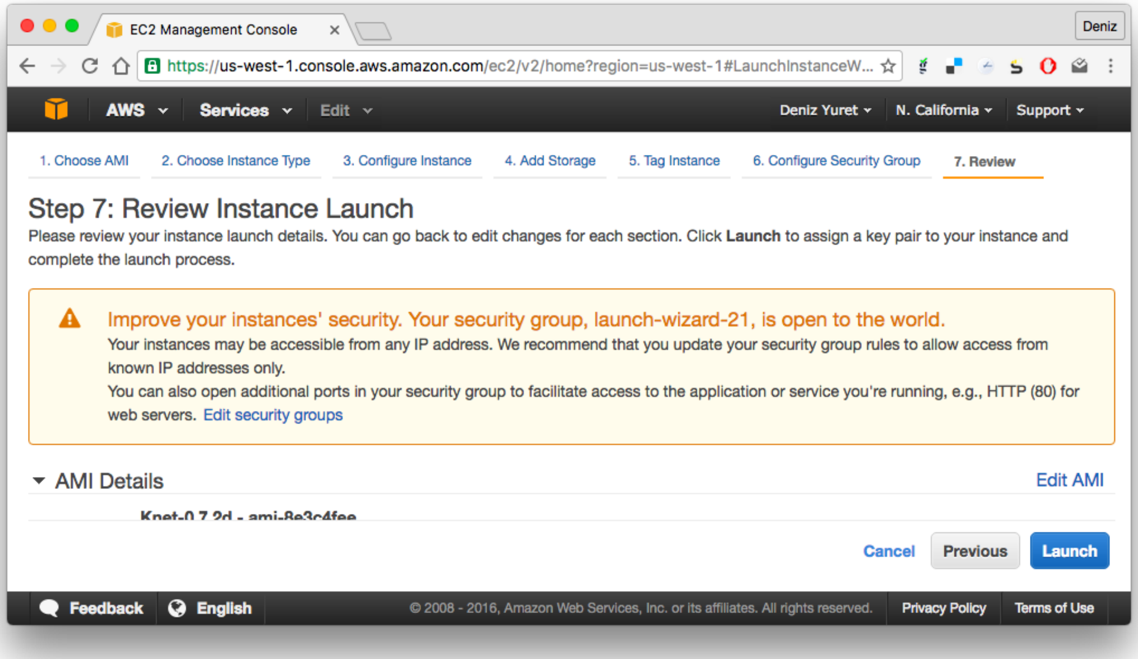
have varying combinations of CPU, memory, storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. Learn more about instance types and how they can meet your computing needs.

Filter by: GPU instances | Current generation

Currently selected: t2.micro (Variable ECUs, 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GiB memory, EBS only)

	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
□	GPU instances	g2.2xlarge	8	15	1 x 60 (SSD)	Yes	High
□	GPU instances	g2.8xlarge	32	60	2 x 120 (SSD)	-	10 Gigabit

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



Step 7: Review Instance Launch

Please review your instance launch details. You can go back to edit changes for each section. Click **Launch** to assign a key pair to your instance and complete the launch process.

Improve your instances' security. Your security group, launch-wizard-21, is open to the world.

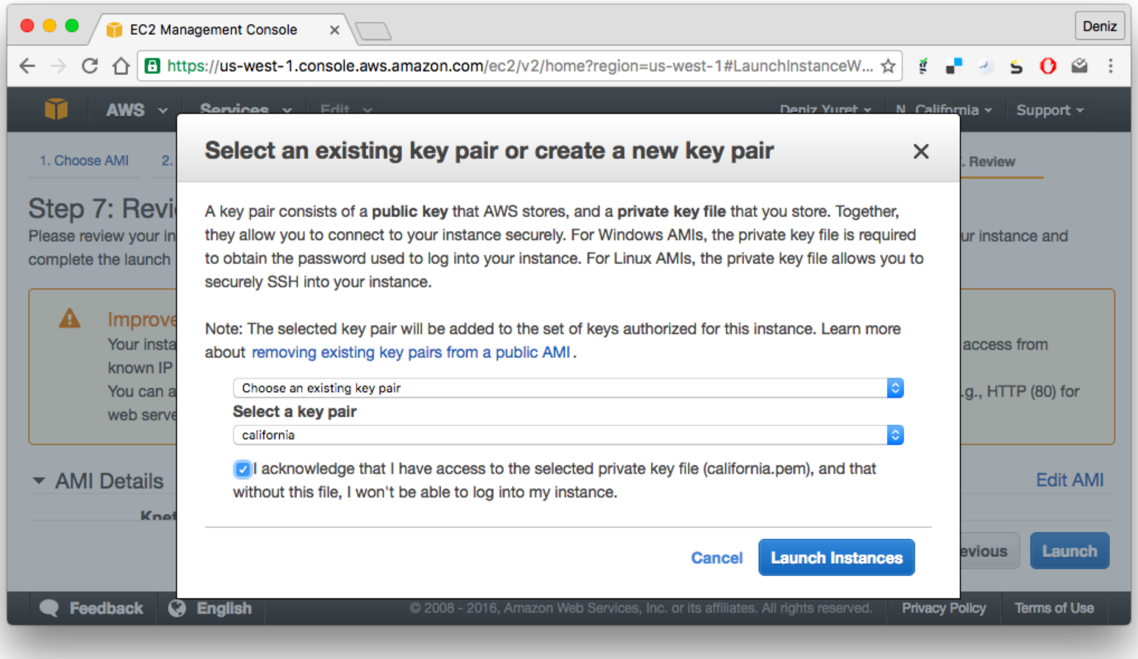
Your instances may be accessible from any IP address. We recommend that you update your security group rules to allow access from known IP addresses only.

You can also open additional ports in your security group to facilitate access to the application or service you're running, e.g., HTTP (80) for web servers. Edit security groups

AMI Details

Knet-0.7.2d - ami-8e3c4fee

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.



EC2 Management Console
Deniz
https://us-west-1.console.aws.amazon.com/ec2/v2/home?region=us-west-1#LaunchInstanceW...
AWS
Services
Edit
Deniz Yuret
N. California
Support
1. Choose AMI
Review
Step 7: Revi
Edit AMI
Previous
Launch
Feedback
English
© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Select an existing key pair or create a new key pair ✕

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about removing existing key pairs from a public AMI.

Choose an existing key pair

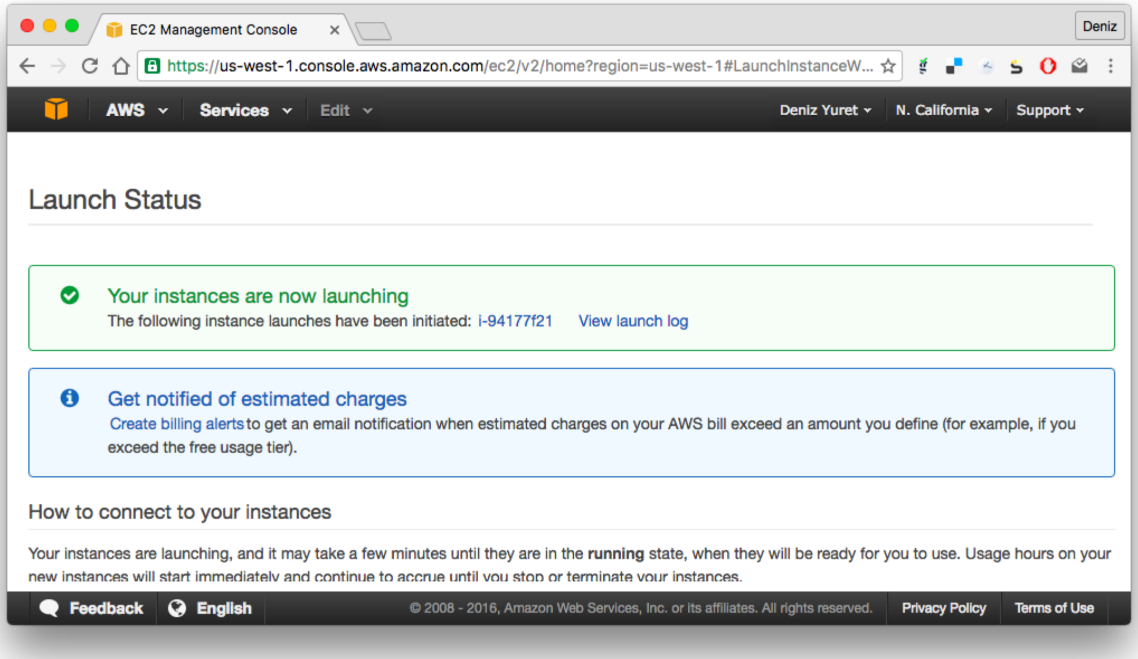
Select a key pair

california

☑ I acknowledge that I have access to the selected private key file (california.pem), and that without this file, I won't be able to log into my instance.

Cancel | Launch Instances

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



EC2 Management Console
Deniz
https://us-west-1.console.aws.amazon.com/ec2/v2/home?region=us-west-1#LaunchInstanceW...
AWS
Services
Edit
Deniz Yuret
N. California
Support
Feedback
English
© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Launch Status

✓ Your instances are now launching

The following instance launches have been initiated: i-94177f21 View launch log

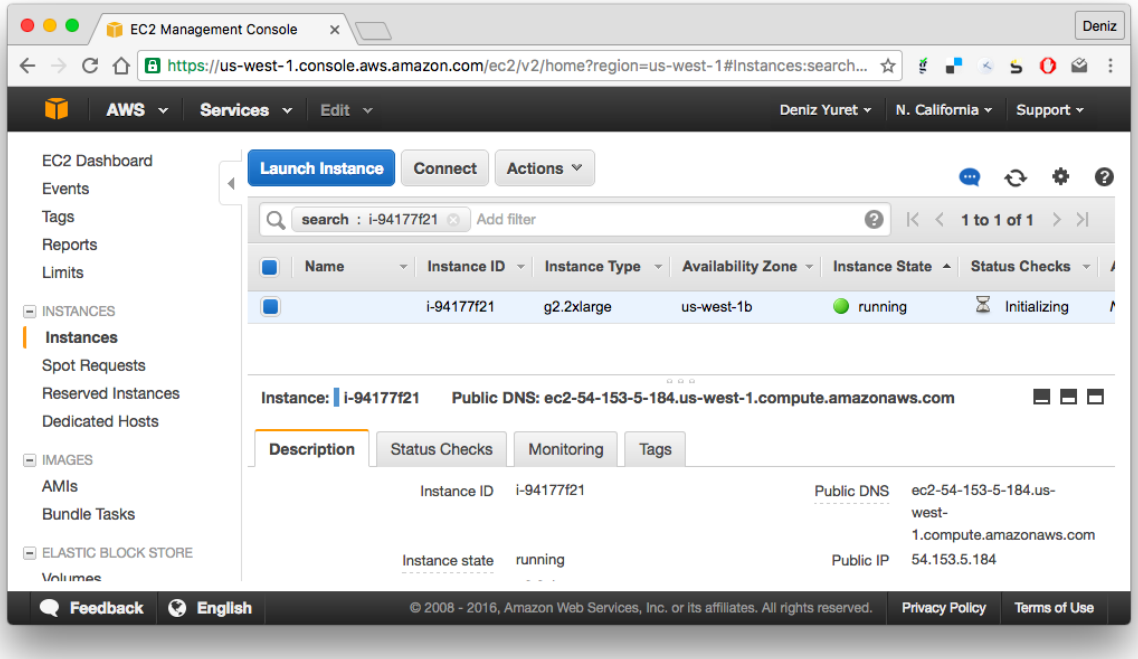
Get notified of estimated charges

Create billing alerts to get an email notification when estimated charges on your AWS bill exceed an amount you define (for example, if you exceed the free usage tier).

How to connect to your instances

Your instances are launching, and it may take a few minutes until they are in the **running** state, when they will be ready for you to use. Usage hours on your new instances will start immediately and continue to accrue until you stop or terminate your instances.

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".



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks
	i-94177f21	g2.2xlarge	us-west-1b	● running	Initializing

Instance: i-94177f21 Public DNS: ec2-54-153-5-184.us-west-1.compute.amazonaws.com

Instance ID	i-94177f21	Public DNS	ec2-54-153-5-184.us-west-1.compute.amazonaws.com
Instance state	running	Public IP	54.153.5.184

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

Introduction to Knet

Knet (pronounced "kay-net") is the Koç University deep learning framework implemented in Julia by Deniz Yuret and collaborators. Unlike gradient generating compilers like Theano and TensorFlow which force users into a restricted mini-language, Knet allows the definition and training of machine learning models using the full power and expressivity of Julia. Models are defined by describing only the forward calculation in plain Julia allowing helper functions, loops, conditionals, recursion, closures, tuples and dictionaries, array indexing and concatenation and almost everything else Julia offers. High performance is achieved by combining automatic differentiation of most of Julia with efficient GPU kernels and memory management. The computations can be performed on the GPU by simply using KnetArray instead of Array for parameters and data. Check out the full documentation (in progress) and the examples directory for more information.

2.1 Contents

2.2 Installation

You can install Knet using `Pkg.add("Knet")`. Some of the examples use additional packages such as ArgParse, GZip, and JLD. These are not required by Knet and are installed automatically when needed. See the detailed installation instructions as well as the section on using Amazon AWS to experiment with GPU machines on the cloud with pre-installed Knet images.

2.3 Examples

In Knet, a machine learning model is defined using plain Julia code. A typical model consists of a *prediction* and a *loss* function. The prediction function takes model parameters and some input, returns the prediction of the model for that input. The loss function measures how bad the prediction is with respect to some desired output. We train a model by adjusting its parameters to reduce the loss. In this section we will see the prediction, loss, and training functions for five models: linear regression, softmax classification, fully-connected, convolutional and recurrent neural networks.

2.3.1 Linear regression

Here is the prediction function and the corresponding quadratic loss function for a simple linear regression model:

```
predict(w,x) = w[1]*x .+ w[2]

loss(w,x,y) = sumabs2(y - predict(w,x)) / size(y,2)
```

The variable w is a list of parameters (it could be a Tuple, Array, or Dict), x is the input and y is the desired output. To train this model, we want to adjust its parameters to reduce the loss on given training examples. The direction in the parameter space in which the loss reduction is maximum is given by the negative gradient of the loss. Knet uses the higher-order function grad from AutoGrad.jl to compute the gradient direction:

```
using Knet

lossgradient = grad(loss)
```

Note that grad is a higher-order function that takes and returns other functions. The lossgradient function takes the same arguments as loss, e.g. dw = lossgradient(w,x,y). Instead of returning a loss value, lossgradient returns dw, the gradient of the loss with respect to its first argument w. The type and size of dw is identical to w, each entry in dw gives the derivative of the loss with respect to the corresponding entry in w. See @doc grad for more information.

Given some training data = [(x1,y1),(x2,y2),...], here is how we can train this model:

```
function train(w, data; lr=.1)
    for (x,y) in data
        dw = lossgradient(w, x, y)
        for i in 1:length(w)
            w[i] -= lr * dw[i]
        end
    end
    return w
end
```

We simply iterate over the input-output pairs in data, calculate the lossgradient for each example, and move the parameters in the negative gradient direction with a step size determined by the learning rate lr. Let's train this model on the Housing dataset from the UCI Machine Learning Repository.

```
julia> url = "https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
julia> rawdata = readdlm(download(url))
julia> x = rawdata[:,1:13]'
julia> x = (x .- mean(x,2)) ./ std(x,2)
julia> y = rawdata[:,14:14]'
julia> w = Any[ 0.1*randn(1,13), 0 ]
julia> for i=1:10; train(w, [(x,y)]); println(loss(w,x,y)); end
366.0463078055053
...
29.63709385230451
```

The dataset has housing related information for 506 neighborhoods in Boston from 1978. Each neighborhood is represented using 13 attributes such as crime rate or distance to employment centers. The goal is to predict the median value of the houses given in $1000's. After downloading, splitting and normalizing the data, we initialize the parameters randomly and take 10 steps in the negative gradient direction. We can see the loss dropping from 366.0 to 29.6. See housing.jl for more information on this example.

Note that `grad` was the only function used that is not in the Julia standard library. This is typical of models defined in Knet.

2.3.2 Softmax classification

In this example we build a simple classification model for the MNIST handwritten digit recognition dataset. MNIST has 60000 training and 10000 test examples. Each input x consists of 784 pixels representing a 28x28 image. The corresponding output indicates the identity of the digit 0..9. (image source)

Classification models handle discrete outputs, as opposed to regression models which handle numeric outputs. We typically use the cross entropy loss function in classification models:

```
function loss(w,x,ygold)
    ypred = predict(w,x)
    ynorm = ypred .- log(sum(exp(ypred),1))
    -sum(ygold .* ynorm) / size(ygold,2)
end
```

Other than the change of loss function, the softmax model is identical to the linear regression model. We use the same `predict`, same `train` and set `lossgradient=grad(loss)` as before. To see how well our model classifies let's define an `accuracy` function which returns the percentage of instances classified correctly:

```
function accuracy(w, data)
    ncorrect = ninstance = 0
    for (x, ygold) in data
        ypred = predict(w,x)
        ncorrect += sum(ygold .* (ypred .== maximum(ypred,1)))
        ninstance += size(ygold,2)
    end
    return ncorrect/ninstance
end
```

Now let's train a model on the MNIST data:

```
julia> include(Pkg.dir("Knet/examples/mnist.jl"))
julia> using MNIST: xtrn, ytrn, xtst, ytst, minibatch
julia> dtrn = minibatch(xtrn, ytrn, 100)
julia> dtst = minibatch(xtst, ytst, 100)
julia> w = Any[ -0.1+0.2*rand(Float32,10,784), zeros(Float32,10,1) ]
julia> println((:epoch, 0, :trn, accuracy(w,dtrn), :tst, accuracy(w,dtst)))
julia> for epoch=1:10
           train(w, dtrn; lr=0.5)
           println((:epoch, epoch, :trn, accuracy(w,dtrn), :tst, accuracy(w,dtst)))
       end

(:epoch,0,:trn,0.11761667f0,:tst,0.121f0)
(:epoch,1,:trn,0.9005f0,:tst,0.9048f0)
...
(:epoch,10,:trn,0.9196f0,:tst,0.9153f0)
```

Including `mnist.jl` loads the MNIST data, downloading it from the internet if necessary, and provides a training set (xtrn,ytrn), test set (xtst,ytst) and a `minibatch` utility which we use to rearrange the data into chunks of 100 instances. After randomly initializing the parameters we train for 10 epochs, printing out training and test set accuracy at every epoch. The final accuracy of about 92% is close to the limit of what we can achieve with this type of model. To improve further we must look beyond linear models.

2.3.3 Multi-layer perceptron

A multi-layer perceptron, i.e. a fully connected feed-forward neural network, is basically a bunch of linear regression models stuck together with non-linearities in between. (image source)

We can define a MLP by slightly modifying the predict function:

```
function predict(w,x)
    for i=1:2:length(w)-2
        x = max(0, w[i]*x .+ w[i+1])
    end
    return w[end-1]*x .+ w[end]
end
```

Here `w[2k-1]` is the weight matrix and `w[2k]` is the bias vector for the k'th layer. max(0,a) implements the popular rectifier non-linearity. Note that if w only has two entries, this is equivalent to the linear and softmax models. By adding more entries to w, we can define multi-layer perceptrons of arbitrary depth. Let's define one with a single hidden layer of 64 units:

```
w = Any[ -0.1+0.2*rand(Float32,64,784), zeros(Float32,64,1),
         -0.1+0.2*rand(Float32,10,64),  zeros(Float32,10,1) ]
```

The rest of the code is the same as the softmax model. We use the same cross-entropy loss function and the same training script. The code for this example is available in mnist.jl. The multi-layer perceptron does significantly better than the softmax model:

```
(:epoch,0,:trn,0.10166667f0,:tst,0.0977f0)
(:epoch,1,:trn,0.9389167f0,:tst,0.9407f0)
...
(:epoch,10,:trn,0.9866f0,:tst,0.9735f0)
```

2.3.4 Convolutional neural network

To improve the performance further, we can use a convolutional neural networks (CNN). See the course notes by Andrej Karpathy for a good introduction to CNNs. We will implement the LeNet model which consists of two convolutional layers followed by two fully connected layers. (image source)

Knet provides the `conv4(w,x)` and `pool(x)` functions for the implementation of convolutional nets (see `@doc conv4` and `@doc pool` for details):

```
function predict(w,x0)
    x1 = pool(max(0, conv4(w[1],x0) .+ w[2]))
    x2 = pool(max(0, conv4(w[3],x1) .+ w[4]))
    x3 = max(0, w[5]*mat(x2) .+ w[6])
    return w[7]*x3 .+ w[8]
end
```

The weights for the convolutional net can be initialized as follows:

```
w = Any[ -0.1+0.2*rand(Float32,5,5,1,20),  zeros(Float32,1,1,20,1),
         -0.1+0.2*rand(Float32,5,5,20,50), zeros(Float32,1,1,50,1),
         -0.1+0.2*rand(Float32,500,800),   zeros(Float32,500,1),
         -0.1+0.2*rand(Float32,10,500),    zeros(Float32,10,1) ]
```

Currently convolution and pooling are only supported on the GPU for 4-D and 5-D arrays. So we reshape our data and transfer it to the GPU along with the parameters by converting them into KnetArrays (see `@doc KnetArray` for more information):

```
dtrn = map(d->(KnetArray(reshape(d[1],(28,28,1,100))), KnetArray(d[2])), dtrn)
dtst = map(d->(KnetArray(reshape(d[1],(28,28,1,100))), KnetArray(d[2])), dtst)
w = map(KnetArray, w)
```

The training proceeds as before giving us even better results. The code for the LeNet example can be found in lenet.jl.

```
(:epoch,0,:trn,0.12215f0,:tst,0.1263f0)
(:epoch,1,:trn,0.96963334f0,:tst,0.971f0)
...
(:epoch,10,:trn,0.99553335f0,:tst,0.9879f0)
```

2.3.5 Recurrent neural network

In this section we will see how to implement a recurrent neural network (RNN) in Knet. An RNN is a class of neural network where connections between units form a directed cycle, which allows them to keep a persistent state 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. (image source)

As an example, we will build a character-level language model inspired by "The Unreasonable Effectiveness of Recurrent Neural Networks" from the Andrej Karpathy blog. The model can be trained with different genres of text, and can be used to generate original text in the same style.

It turns out simple RNNs are not very good at remembering things for a very long time. Currently the most popular solution is to use a more complicated unit like the Long Short Term Memory (LSTM). An LSTM controls the information flow into and out of the unit using gates similar to digital circuits and can model long term dependencies. See Understanding LSTM Networks by Christopher Olah for a good overview of LSTMs. (image source)

The code below shows one way to define an LSTM in Knet. The first two arguments are the parameters, the weight matrix and the bias vector. The next two arguments hold the internal state of the LSTM: the hidden and cell arrays. The last argument is the input. Note that for performance reasons we lump all the parameters of the LSTM into one matrix-vector pair instead of using separate parameters for each gate. This way we can perform a single matrix multiplication, and recover the gates using array indexing. We represent input, hidden and cell as row vectors rather than column vectors for more efficient concatenation and indexing. `sigm` and `tanh` are the sigmoid and the hyperbolic tangent activation functions. The LSTM returns the updated state variables `hidden` and `cell`.

```
function lstm(weight,bias,hidden,cell,input)
    gates   = hcat(input,hidden) * weight .+ bias
    hsize   = size(hidden,2)
    forget  = sigm(gates[:,1:hsize])
    ingate  = sigm(gates[:,1+hsize:2hsize])
    outgate = sigm(gates[:,1+2hsize:3hsize])
    change  = tanh(gates[:,1+3hsize:end])
    cell    = cell .* forget + ingate .* change
    hidden  = outgate .* tanh(cell)
    return (hidden,cell)
end
```

The LSTM has an input gate, forget gate and an output gate that control information flow. Each gate depends on the current `input` value, and the last hidden state `hidden`. The memory value `cell` is computed by blending a new value `change` with the old `cell` 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` 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.

To build a language model, we need to predict the next character in a piece of text given the current character and recent history as encoded in the internal state. The `predict` function below implements a multi-layer LSTM model. `s[2k-1:2k]` hold the hidden and cell arrays and `w[2k-1:2k]` hold the weight and bias parameters for the k'th LSTM layer. The last three elements of `w` are the embedding matrix and the weight/bias for the final prediction. `predict` takes the current character encoded in `x` as a one-hot row vector, multiplies it with the embedding matrix, passes it through a number of LSTM layers, and converts the output of the final layer to the same number of dimensions as the input using a linear transformation. The state variable `s` is modified in-place.

```julia
function predict(w, s, x)
    x = x * w[end-2]
    for i = 1:2:length(s)
        (s[i],s[i+1]) = lstm(w[i],w[i+1],s[i],s[i+1],x)
        x = s[i]
    end
    return x * w[end-1] .+ w[end]
end
```

To train the language model we will use Backpropagation Through Time (BPTT) which basically means running the network on a given sequence and updating the parameters based on the total loss. Here is a function that calculates the total cross-entropy loss for a given (sub)sequence:

```julia
function loss(param,state,sequence,range=1:length(sequence)-1)
    total = 0.0; count = 0
    atype = typeof(getval(param[1]))
    input = convert(atype,sequence[first(range)])
    for t in range
        ypred = predict(param,state,input)
        ynorm = logp(ypred,2) # ypred .- log(sum(exp(ypred),2))
        ygold = convert(atype,sequence[t+1])
        total += sum(ygold .* ynorm)
        count += size(ygold,1)
        input = ygold
    end
    return -total / count
end
```

Here `param` and `state` hold the parameters and the state of the model, `sequence` and `range` give us the input sequence and a possible range over it to process. We convert the entries in the sequence to inputs that have the same type as the parameters one at a time (to conserve GPU memory). We use each token in the given range as an input to predict the next token. The average cross-entropy loss per token is returned.

To generate text we sample each character randomly using the probabilities predicted by the model based on the previous character:

```julia
function generate(param, state, vocab, nchar)
    index_to_char = Array(Char, length(vocab))
    for (k,v) in vocab; index_to_char[v] = k; end
    input = oftype(param[1], zeros(1,length(vocab)))
    index = 1
    for t in 1:nchar
        ypred = predict(param,state,input)
        input[index] = 0
        index = sample(exp(logp(ypred)))
        print(index_to_char[index])
        input[index] = 1
    end
    println()
end
```

Here `param` and `state` hold the parameters and state variables as usual. `vocab` is a Char->Int dictionary of the characters that can be produced by the model, and `nchar` gives the number of characters to generate. We initialize the input as a zero vector and use `predict` to predict subsequent characters. `sample` picks a random index based on the normalized probabilities output by the model.

At this point we can train the network on any given piece of text (or other discrete sequence). For efficiency it is best to minibatch the training data and run BPTT on small subsequences. See charlm.jl for details. Here is a sample run on 'The Complete Works of William Shakespeare':

```
$ cd .julia/Knet/examples
$ wget http://www.gutenberg.org/files/100/100.txt
$ julia charlm.jl --data 100.txt --epochs 10 --winit 0.3 --save shakespeare.jld
... takes about 10 minutes on a GPU machine
$ julia charlm.jl --load shakespeare.jld --generate 1000

    Pand soping them, my lord, if such a foolish?
  MARTER. My lord, and nothing in England's ground to new comp'd.
    To bless your view of wot their dullst. If Doth no ape;
    Which with the heart. Rome father stuff
    These shall sweet Mary against a sudden him
    Upon up th' night is a wits not that honour,
    Shouts have sure?
  MACBETH. Hark? And, Halcance doth never memory I be thou what
    My enties mights in Tim thou?
  PIESTO. Which it time's purpose mine hortful and
    is my Lord.
  BOTTOM. My lord, good mine eyest, then: I will not set up.
  LUCILIUS. Who shall
```

2.4 Benchmarks

Each of the examples above was used as a benchmark to compare Knet with other frameworks. The table below shows the number of seconds it takes to train a given model for a particular dataset, number of epochs and minibatch size for Knet, Theano, Torch, Caffe and TensorFlow. Knet has comparable performance to other commonly used frameworks.

model	dataset	epochs	batch	Knet	Theano	Torch	Caffe	TFlow
LinReg	Housing	10K	506	2.84	1.88	2.66	2.35	5.92
Softmax	MNIST	10	100	2.35	1.40	2.88	2.45	5.57
MLP	MNIST	10	100	3.68	2.31	4.03	3.69	6.94
LeNet	MNIST	1	100	3.59	3.03	1.69	3.54	8.77
CharLM	Hiawatha	1	128	2.25	2.42	2.23	1.43	2.86

The benchmarking was done on g2.2xlarge GPU instances on Amazon AWS. The code is available at github and as machine image `deep_AMI_v6` at AWS N.California. See the section on using Amazon AWS for more information. The datasets are available online using the following links: Housing, MNIST, Hiawatha. The MLP uses a single hidden layer of 64 units. CharLM uses a single layer LSTM language model with embedding and hidden layer sizes set to 256 and trained using BPTT with a sequence length of 100. Each dataset was minibatched and transferred to GPU prior to benchmarking when possible.

2.5 Function reference

We implement machine learning models in Knet using regular Julia code and the `grad` function. Knet defines a few more utility functions listed below. See `@doc <function>` for full details.

`grad`	returns the gradient function.
`KnetArray`	constructs a GPU array.
`gradcheck`	compares gradients with numeric approximations.
`Knet.dir`	returns a path relative to Knet root.
`gpu`	determines which GPU Knet uses.
`relu`	returns `max(0,x)`
`sigm`	returns `(1./(1+exp(-x)))`
`invx`	returns `(1./x)`
`logp`	returns `x .- log(sum(exp(x),[dims]))`
`logsumexp`	returns `log(sum(exp(x),[dims]))`
`conv4`	executes convolutions or cross-correlations.
`pool`	replaces several adjacent values with their mean or maximum.
`mat`	reshapes its input into a two-dimensional matrix.
`update!`	updates the weight depending on the gradient and the parameters of the optimization method

2.6 Optimization methods

In the examples above, we used simple SGD as the optimization method and performed parameter updates manually using `w[i] -= lr * dw[i]`. The `update!` function provides more optimization methods and can be used in place of this manual update. In addition to a weight array `w[i]` and its gradient `dw[i]`, `update!` requires a third argument encapsulating the type, options, and state of the optimization method. The constructors of the supported optimization methods are listed below. See `@doc Sgd` etc. for full details. Note that in general we need to keep one of these state variables per weight array, see optimizers.jl for example usage.

`Sgd`	encapsulates learning rate
`Momentum`	encapsulates learning rate, gamma and velocity
`Adam`	encapsualtes learning rate, beta1, beta2, epsilon, time, first and second moments
`Adagrad`	encapsualtes learning rate, epsilon and accumulated gradients (G)
`Adadelta`	encapsulates learning rate, rho, epsilon, accumulated gradients (G) and updates (delta)
`Rmsprop`	encapsulates learning rate, rho, epsilon and accumulated gradients (G)

2.7 Under the hood

Knet relies on the AutoGrad package and the KnetArray data type for its functionality and performance. AutoGrad computes the gradient of Julia functions and KnetArray implements high performance GPU arrays with custom memory management. This section briefly describes them.

2.7.1 AutoGrad

As we have seen, many common machine learning models can be expressed as differentiable programs that input parameters and data and output a scalar loss value. The loss value measures how close the model predictions are to desired values with the given parameters. Training a model can then be seen as an optimization problem: find the parameters that minimize the loss. Typically, a gradient based optimization algorithm is used for computational efficiency: the direction in the parameter space in which the loss reduction is maximum is given by the negative gradient of the loss with respect to the parameters. Thus gradient computations take a central stage in software frameworks for machine learning. In this section I will briefly outline existing gradient computation techniques and motivate the particular approach taken by Knet.

Computation of gradients in computer models is performed by four main methods (Baydin et al. 2015):

- manual differentiation (programming the derivatives)
- numerical differentiation (using finite difference approximations)
- symbolic differentiation (using expression manipulation)
- automatic differentiation (detailed below)

Manually taking derivatives and coding the result is labor intensive, error-prone, and all but impossible with complex deep learning models. Numerical differentiation is simple: $f'(x) = (f(x+\epsilon) - f(x-\epsilon))/(2\epsilon)$ but impractical: the finite difference equation needs to be evaluated for each individual parameter, of which there are typically many. Pure symbolic differentiation using expression manipulation, as implemented in software such as Maxima, Maple, and Mathematica is impractical for different reasons: (i) it may not be feasible to express a machine learning model as a closed form mathematical expression, and (ii) the symbolic derivative can be exponentially larger than the model itself leading to inefficient run-time calculation. This leaves us with automatic differentiation.

Automatic differentiation is the idea of using symbolic derivatives only at the level of elementary operations, and computing the gradient of a compound function by applying the chain rule to intermediate numerical results. For example, pure symbolic differentiation of $\sin^2(x)$ could give us $2\sin(x)\cos(x)$ directly. Automatic differentiation would use the intermediate numerical values $x_1 = \sin(x)$, $x_2 = x_1^2$ and the elementary derivatives $dx_2/dx_1 = 2x_1$, $dx_1/dx = \cos(x)$ to compute the same answer without ever building a full gradient expression.

To implement automatic differentiation the target function needs to be decomposed into its elementary operations, a process similar to compilation. Most machine learning frameworks (such as Theano, Torch, Caffe, Tensorflow and older versions of Knet prior to v0.8) compile models expressed in a restricted mini-language into a computational graph of elementary operations that have pre-defined derivatives. There are two drawbacks with this approach: (i) the restricted mini-languages tend to have limited support for high-level language features such as conditionals, loops, helper functions, array indexing, etc. (e.g. the infamous `scan` operation in Theano) (ii) the sequence of elementary operations that unfold at run-time needs to be known in advance, and they are difficult to handle when the sequence is data dependent.

There is an alternative: high-level languages, like Julia and Python, already know how to decompose functions into their elementary operations. If we let the users define their models directly in a high-level language, then record the elementary operations during loss calculation at run-time, the computational graph can be constructed from the recorded operations. The cost of recording is not prohibitive: The table below gives cumulative times for elementary operations of an MLP with quadratic loss. Recording only adds 15% to the raw cost of the forward computation. Backpropagation roughly doubles the total time as expected.

op	secs
`a1=w1*x`	0.67
`a2=w2.+a1`	0.71
`a3=max(0,a2)`	0.75
`a4=w3*a3`	0.81
`a5=w4.+a4`	0.85
`a6=a5-y`	0.89
`a7=sumabs2(a6)`	1.18
+recording	1.33
+backprop	2.79

This is the approach taken by the popular autograd Python package and its Julia port AutoGrad.jl used by Knet. In these implementations `g=grad(f)` generates a gradient function `g`, which takes the same inputs as the function `f` but returns the gradient. The gradient function `g` triggers recording by boxing the parameters in a special data type and calls `f`. The elementary operations in `f` are overloaded to record their actions and output boxed answers when their inputs are boxed. The sequence of recorded operations is then used to compute gradients. In the Julia AutoGrad package, derivatives can be defined independently for each method of a function (determined by argument types) making full use of Julia's multiple dispatch. New elementary operations and derivatives can be defined concisely using Julia's macro and meta-programming facilities. See AutoGrad.jl for details.

2.7.2 KnetArray

GPUs have become indispensable for training large deep learning models. Even the small examples implemented here run up to 17x faster on the GPU compared to the 8 core CPU architecture we use for benchmarking. However GPU implementations have a few potential pitfalls: (i) GPU memory allocation is slow, (ii) GPU-RAM memory transfer is slow, (iii) reduction operations (like `sum`) can be very slow unless implemented properly (See Optimizing Parallel Reduction in CUDA).

Knet implements KnetArray as a Julia data type that wraps GPU array pointers. KnetArray is based on the more standard CudaArray with a few important differences: (i) KnetArrays have a custom memory manager, similar to ArrayFire, which reuse pointers garbage collected by Julia to reduce the number of GPU memory allocations, (ii) array ranges (e.g. `a[:,3:5]`) are handled as views with shared pointers instead of copies when possible, and (iii) a number of custom CUDA kernels written for KnetArrays implement element-wise, broadcasting, and scalar and vector reduction operations efficiently. As a result Knet allows users to implement their models using high-level code, yet be competitive in performance with other frameworks as demonstrated in the benchmarks section.

2.8 Contributing

Knet is an open-source project and we are always open to new contributions: bug reports and fixes, feature requests and contributions, new machine learning models and operators, inspiring examples, benchmarking results are all welcome. If you need help or would like to request a feature, please consider joining the knet-users mailing list. If you find a bug, please open a GitHub issue. If you would like to contribute to Knet development, check out the knet-dev mailing list and tips for developers. If you use Knet in your own work, here is a paper you can cite:

```
@inproceedings{knet2016mlsys,
  author={Yuret, Deniz},
  title={Knet: beginning deep learning with 100 lines of Julia},
  year={2016},
  booktitle={Machine Learning Systems Workshop at NIPS 2016}
}
```

Current contributors:

- Deniz Yuret
- Ozan Arkan Can
- Onur Kuru
- Emre Ünal
- Erenay Dayanık
- Ömer Kırnap
- İlker Kesen
- Emre Yolcu
- Meriç Melike Softa
- Ekrem Emre Yurdakul

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.

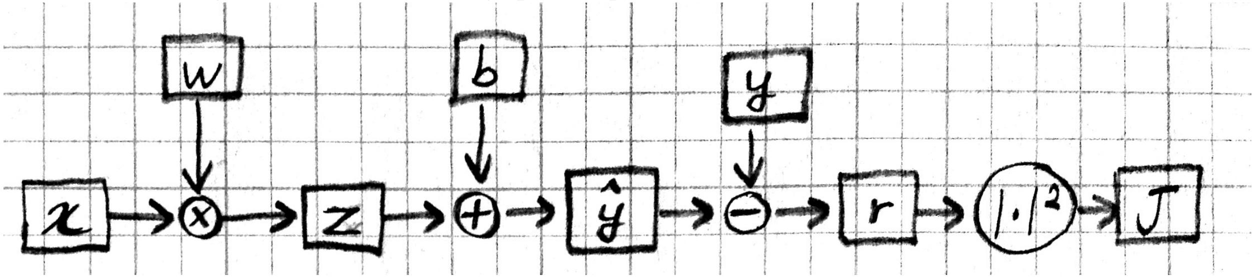
There are several methods to find the solution to the problem of minimizing squared error. Here we will present the **stochastic gradient descent** (SGD) method because it generalizes well to more complex models. In SGD, we take the training examples one at a time (or in small groups called minibatches), 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:

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

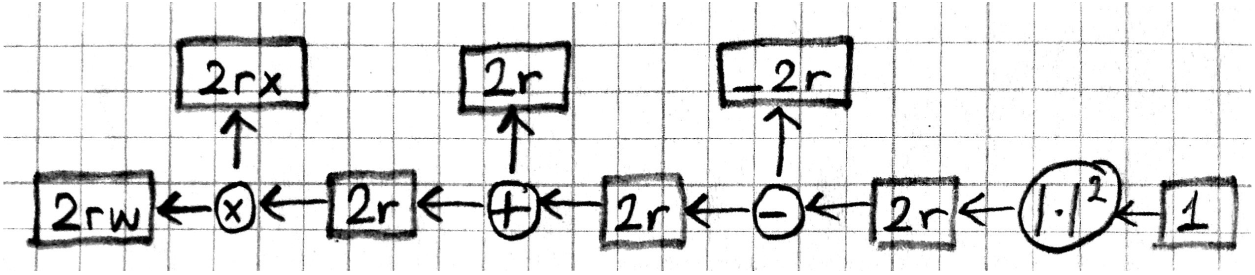
Which means b and z have the same effect on J as $\hat{y}$ and r, i.e. decreasing them by ϵ will decrease J by $2r\epsilon$ as well. Finally:

$$\begin{aligned} z &= wx \\ \partial z/\partial x &= w \\ \partial z/\partial w &= x \end{aligned}$$

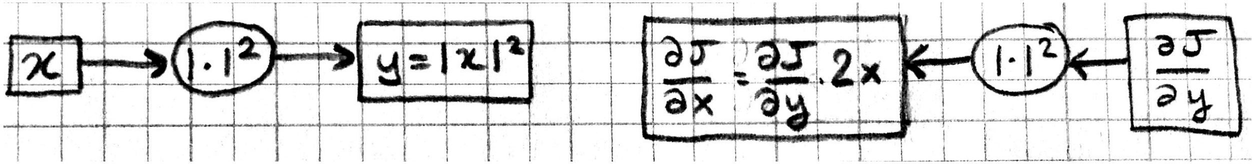
This allows us to compute the effect of w on J in several steps: moving w by ϵ will move z by $x\epsilon$, $\hat{y}$ and r will move exactly the same amount because their partials with z are 1, and finally since r moves by $x\epsilon$, J will move by $2rx\epsilon$.

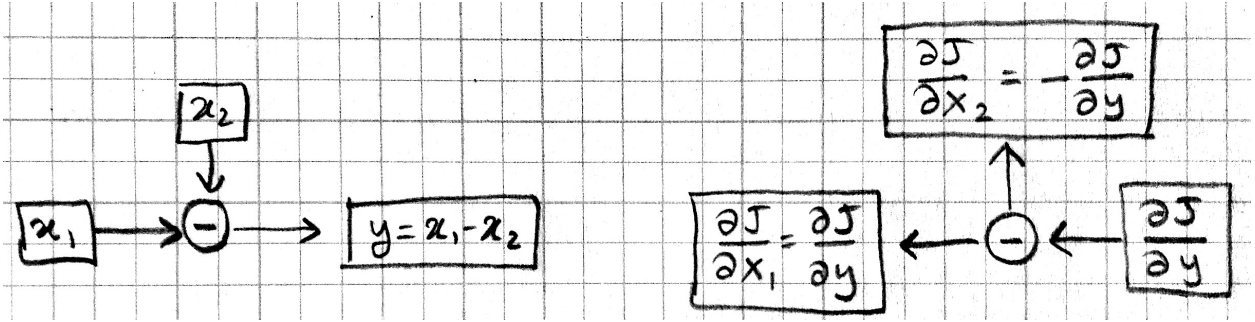
$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial r}\frac{\partial r}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w} = 2rx$$

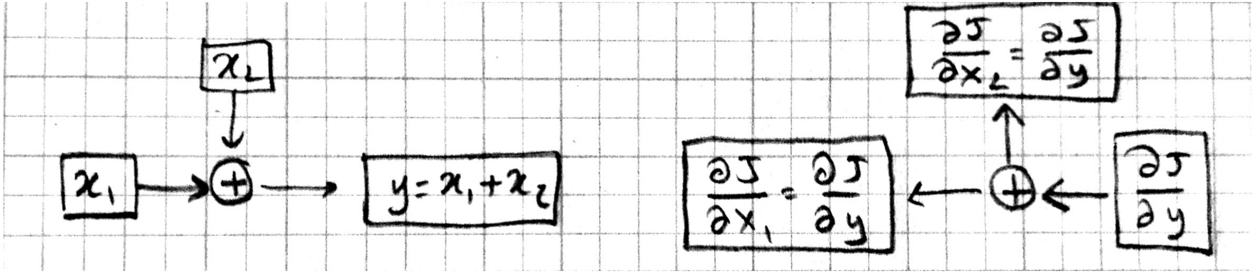
We can represent this process of computing partial derivatives as follows:

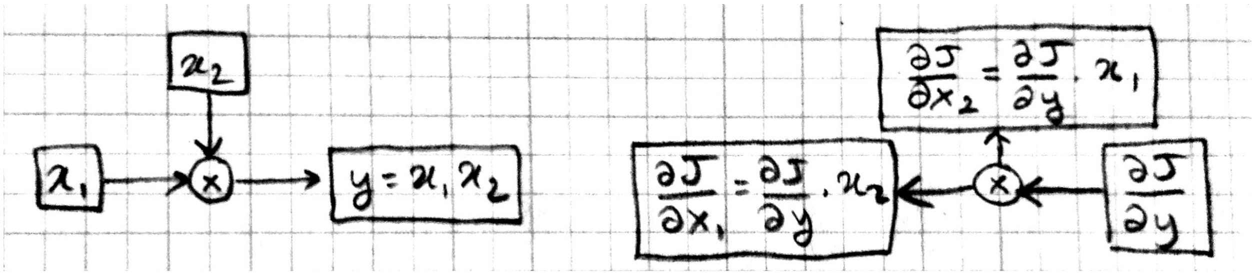


Note that we have the same number of boxes and operations, but all the arrows are reversed. Let us call this the backward pass, and the original computation in the previous picture the forward pass. Each box in this backward-pass picture represents the partial derivative for the corresponding box in 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:








x_2
x_1
×
$y = x_1 x_2$
$\frac{\partial J}{\partial x_2} = \frac{\partial J}{\partial y} \cdot x_1$
$\frac{\partial J}{\partial x_1} = \frac{\partial J}{\partial y} \cdot x_2$
×
$\frac{\partial J}{\partial y}$

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:

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

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:

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

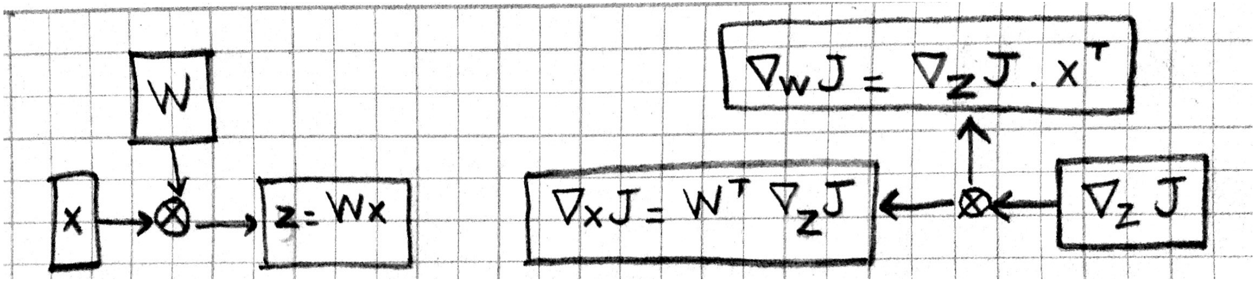
The addition, subtraction, and square norm operations work the same way as before except they act on each element. Moving back through the elementwise operations we see that:

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

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

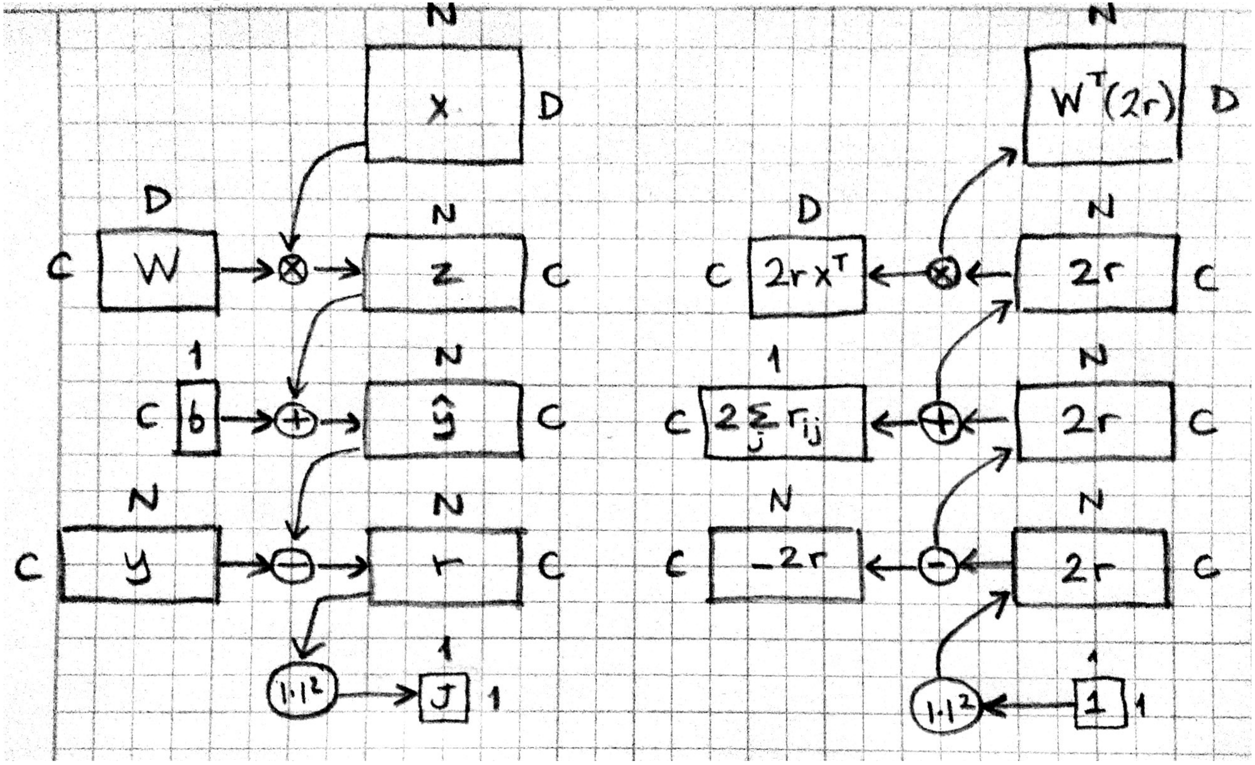
$$\nabla_W J = \nabla_z J \cdot x^T$$
$$\nabla_x J = W^T \cdot \nabla_z J$$

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:


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

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:

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

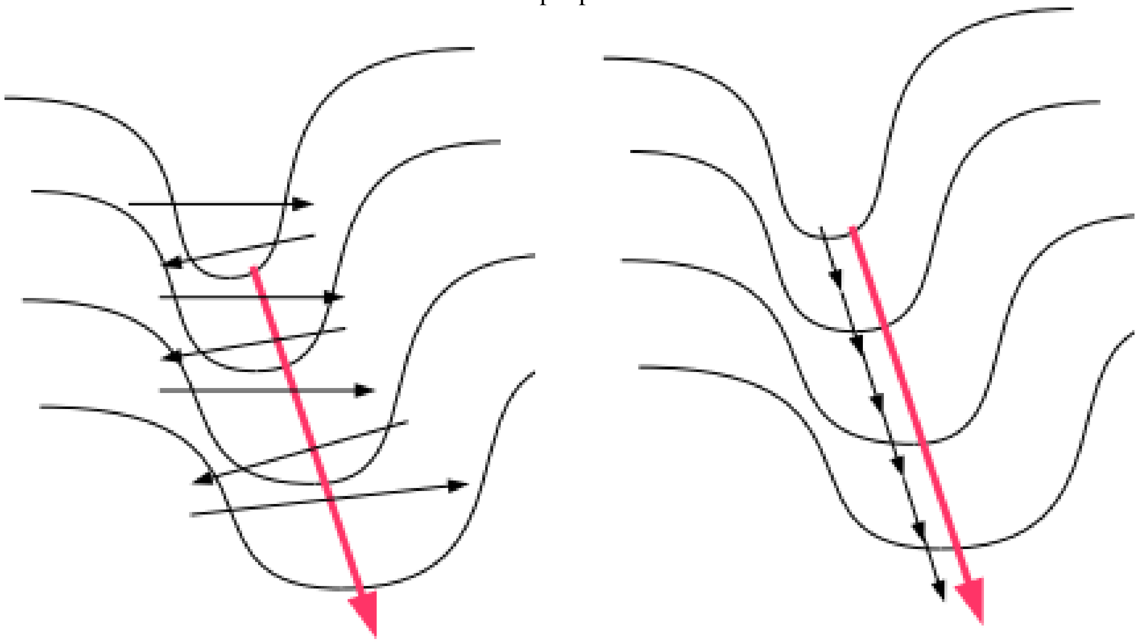
This is the basic idea behind Stochastic Gradient Descent (SGD): Go over the training set instance by instance (or minibatch by minibatch). Run the backpropagation algorithm to calculate the error gradients. Update the weights and biases in the opposite direction of these gradients. Rinse and repeat...

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

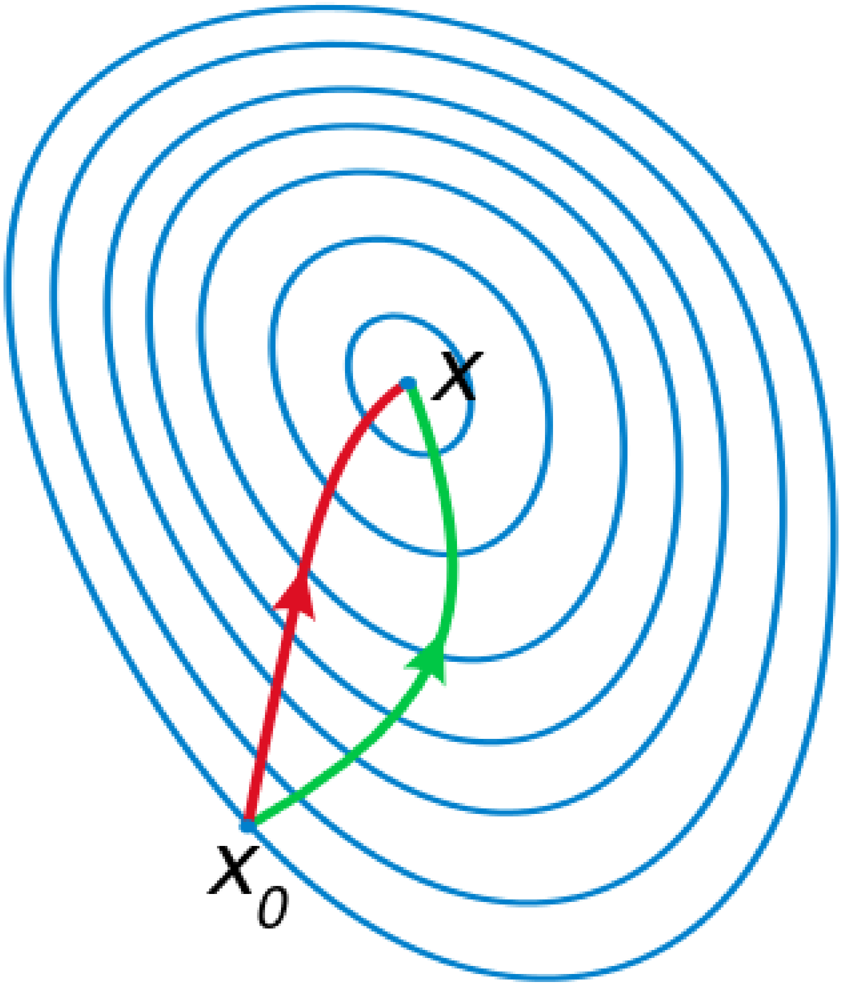
Step size: If the step sizes are too small, the SGD algorithm will take too long to converge. If they are too big it will overshoot the optimum and start to oscillate. So we scale the gradients with an adjustable parameter called the learning rate η:

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

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**.

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**. 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:

$$
\begin{aligned}
\ell &= \sum_c p_c \log \frac{\hat{p}_c}{\sum_k \hat{p}_k} \\
&= \sum_c p_c \log \hat{p}_c - \sum_c p_c \log \sum_k \hat{p}_k \\
&= (\sum_c p_c \log \hat{p}_c) - (\log \sum_k \hat{p}_k) \\
\frac{\partial \ell}{\partial \hat{p}_i} &= \frac{p_i}{\hat{p}_i} - \frac{1}{\sum_k \hat{p}_k} = \frac{p_i}{\hat{p}_i} - 1
\end{aligned}
$$

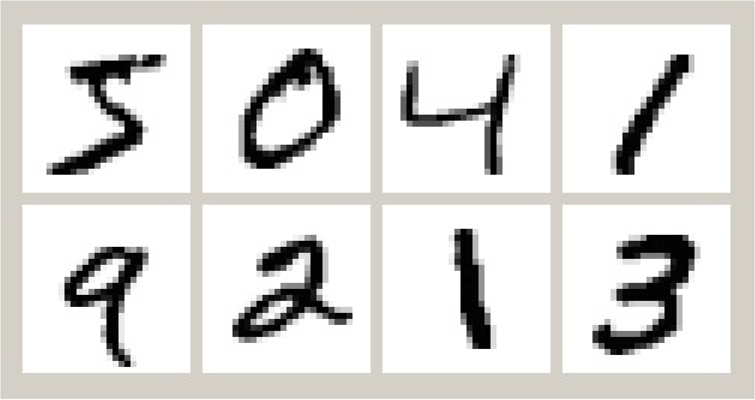
The gradient with respect to unnormalized y takes a particularly simple form:

$$
\begin{aligned}
\frac{\partial \ell}{\partial y_j} &= \sum_i \frac{\partial \ell}{\partial \hat{p}_i} \frac{\partial \hat{p}_i}{\partial y_j} \\
&= \sum_i (\frac{p_i}{\hat{p}_i} - 1)(\,[i = j]\,\hat{p}_i - \hat{p}_i \hat{p}_j) \\
&= p_j - \hat{p}_j \\
\nabla \ell &= p - \hat{p}
\end{aligned}
$$

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 training-with-minibatches for more information about the MNIST task, loading and minibatching data, and simple train and test scripts.

Here is the softmax classifier in Julia:

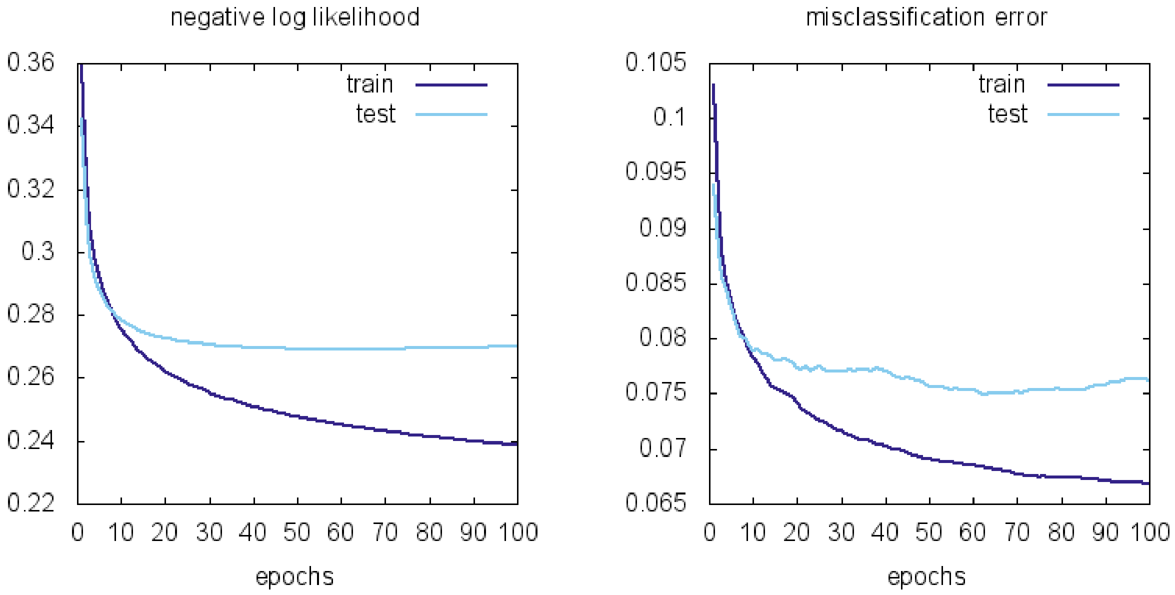
```
function softmax(w,x,ygold)
    ypred = w[1] * x .+ w[2]
    return softloss(ygold, ypred)
end

function softloss(ygold, ypred)
    ynorm = ypred .- log(sum(exp(ypred),1))
    -sum(ygold .* ynorm) / size(ygold,2)
end

softmax_gradient = grad(softmax)
```

Let us train our model for 100 epochs and print out the classification error on the training and test sets after every epoch (see the full example in Pkg.dir("Knet/examples/mnist.jl")):

```
w = Any[0.1*randn(10,784), zeros(10,1)]
for epoch=1:nepochs
    for (x,y) in dtrn  # dtrn is a list of minibatches
        g = softmax_gradient(w, x, y)
        for i in 1:length(w)
            w[i] -= lr * g[i]
        end
    end
    # Print accuracy
end
```

```
(:epoch,0,:trn,0.1135,:tst,0.1097)
(:epoch,1,:trn,0.9008666666666667,:tst,0.9048)
...
(:epoch,99,:trn,0.9274833333333333,:tst,0.9177)
(:epoch,100,:trn,0.92755,:tst,0.9176)
```

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.

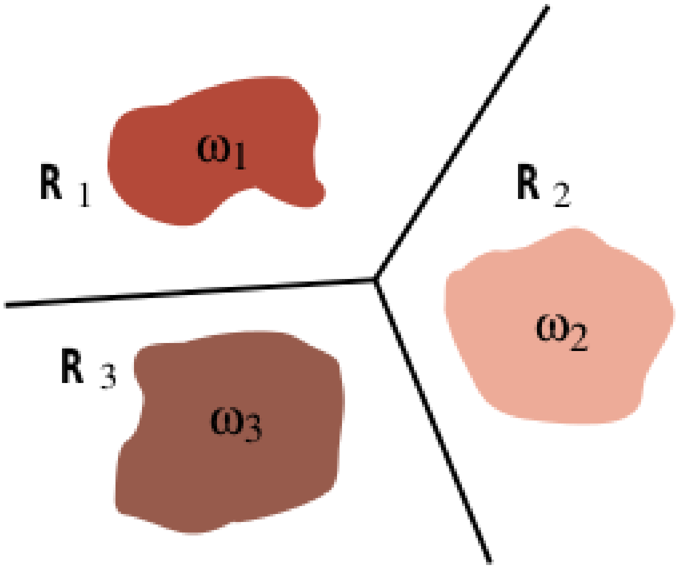
It turns out the softmax classifier is quite limited in its representational power: it can only represent linear classification boundaries. To show this, remember the form of the softmax classifier which gives the probability of the i'th class as:

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

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:

```
function multilinear(w, x, ygold)
    y1 = w[1] * x  .+ w[2]
    y2 = w[3] * y1 .+ w[4]
    return softloss(ygold, y2)
end
```

Note that instead of using `y1` as our prediction, we used it as input to another linear 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:

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

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

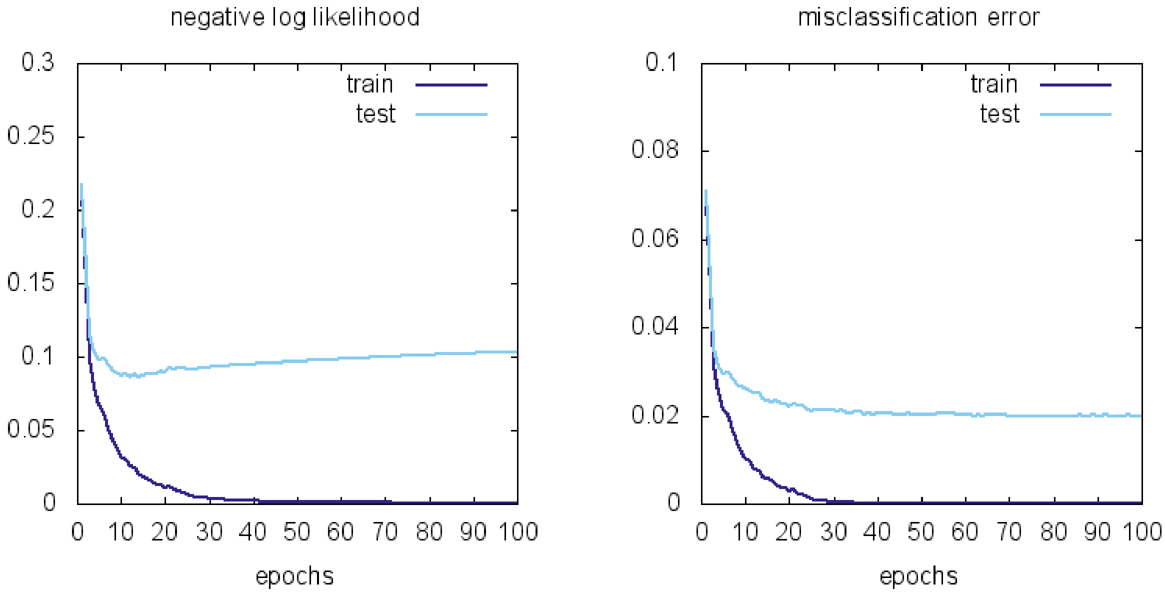
Here is a slightly modified version of the two layer model:

```
function mlp(w, x, ygold)
    y1 = relu(w[1] * x .+ w[2])
    y2 = w[3] * y1 .+ w[4]
    return softloss(ygold, y2)
end
```

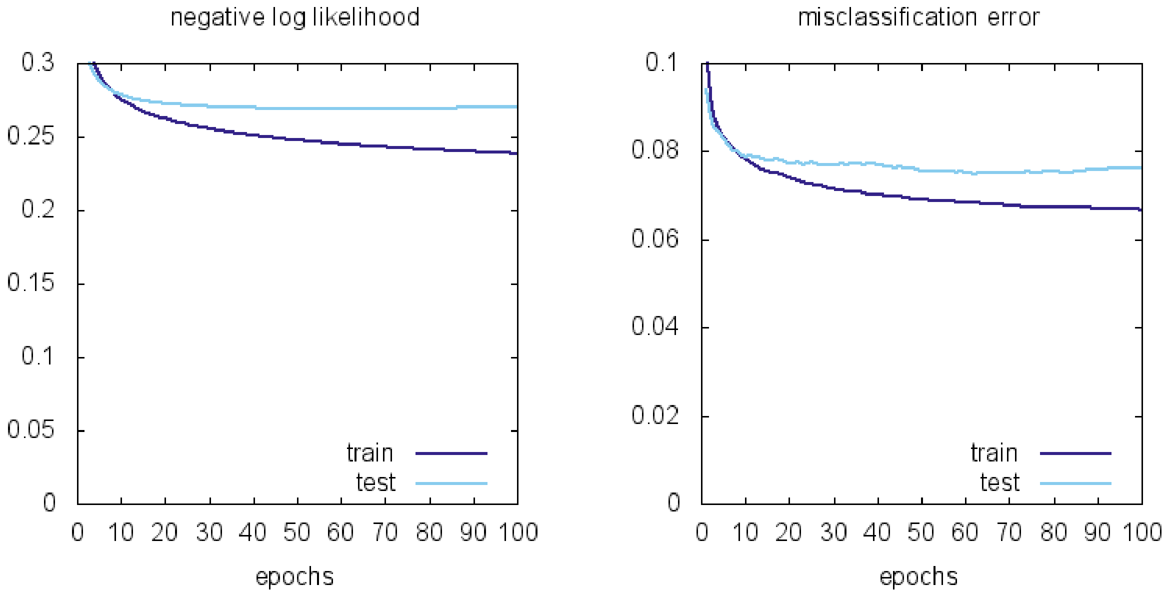
MLP in `mlp` stands for **multilayer perceptron** which is one name for this type of model. The only difference with the previous example is the `relu()` function we introduced in the first line. This is known as the rectified linear unit (or rectifier), and is a simple function defined by `relu(x)=max(x,0)`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

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

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 `mlp` using a hidden layer of size 64:



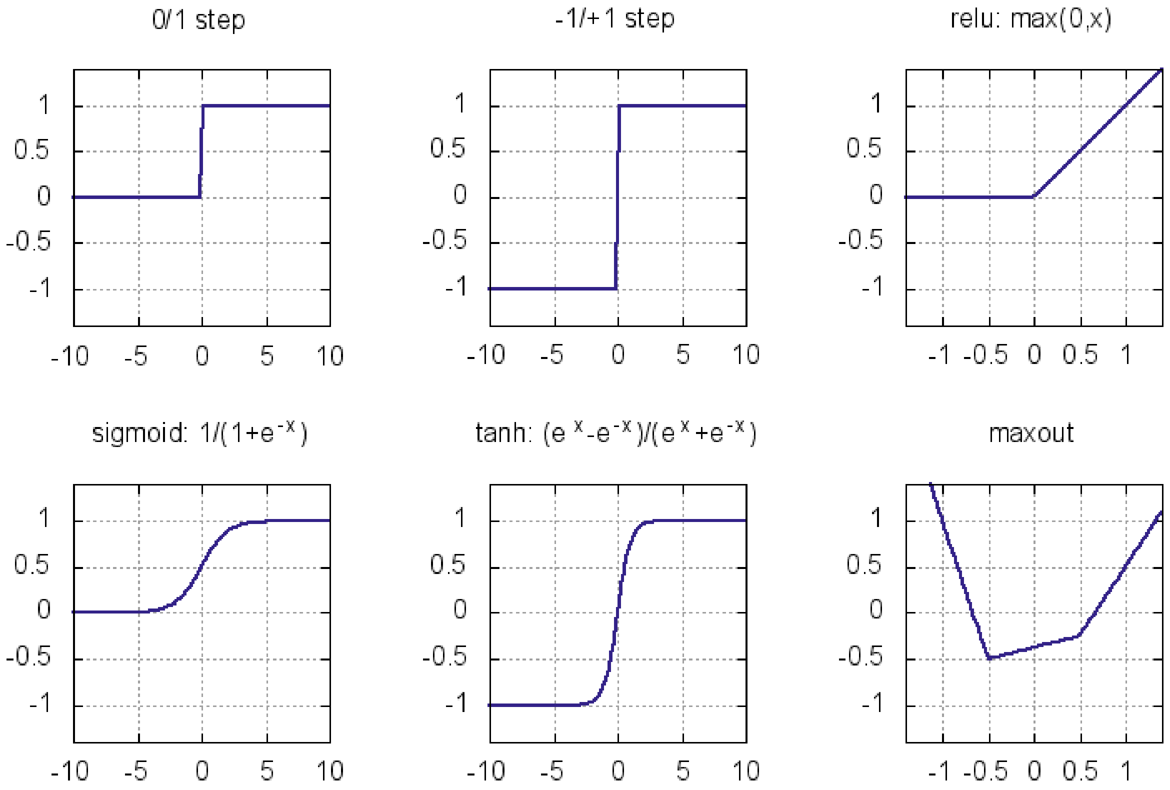
Here are the learning curves for the linear model `softmax` plotted at the same scale for comparison:



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

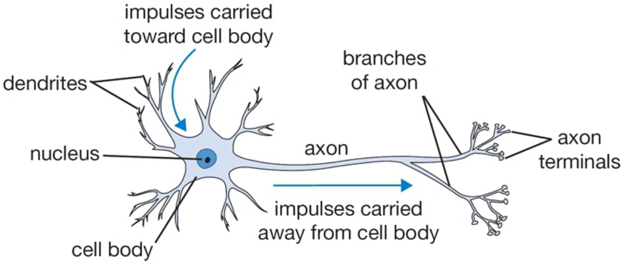
You might be wondering whether `relu` had any special properties or would any of the other nonlinearities be sufficient. Another question is whether there are functions multilayer perceptrons cannot represent and if so whether adding more layers or different types of functions would increase their representational 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.

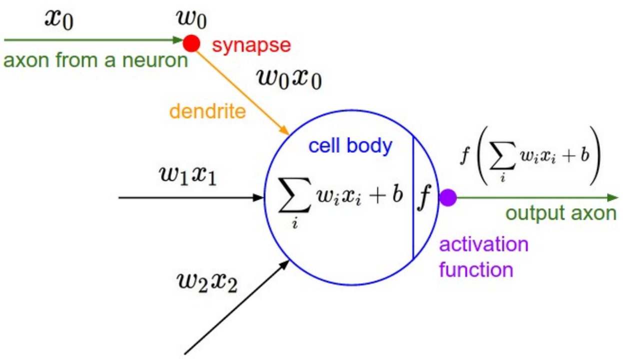
Please see (Nielsen, 2016, Ch 4) for an intuitive explanation of the universality result and (Bengio et al. 2016, Ch 6.4) for a more in depth discussion and references.

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)):

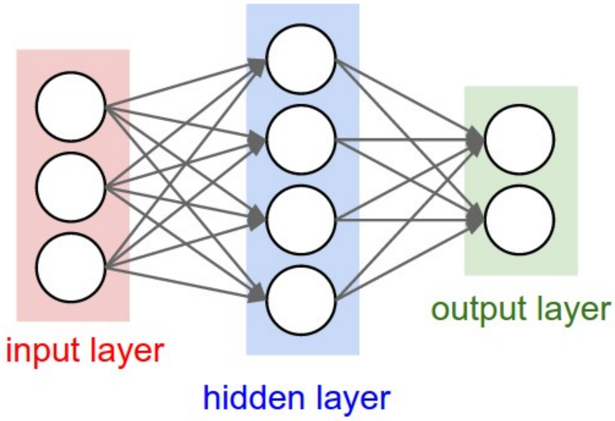

impulses carried toward cell body
dendrites
branches of axon
nucleus
axon
axon terminals
impulses carried away from cell body
cell body

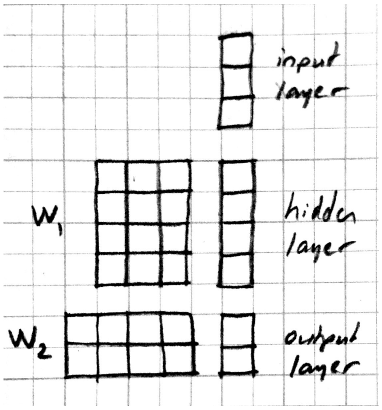
A biological neuron is a complex organism supporting thousands of chemical reactions simultaneously under the regulation of thousands of genes, communicating with other neurons through electrical and chemical pathways involving dozens of different types of neurotransmitter molecules. We assume (do not 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:



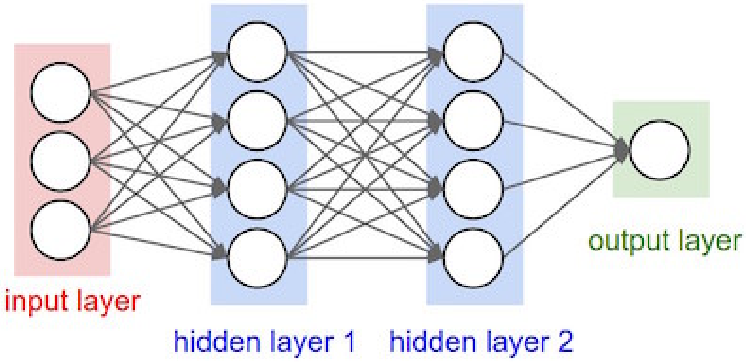
This model is called an artificial neuron, a perceptron, or simply a unit in neural network literature. We know it as the softmax classifier.

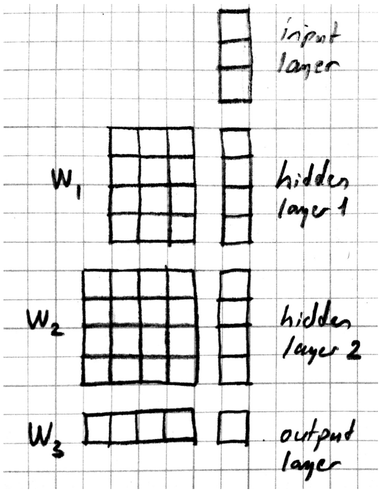
When a number of these units are connected in layers, we get a multilayer perceptron. When counting layers, we ignore the input layer. So the softmax classifier can be considered a one layer neural network. Here is a neural network picture and the corresponding matrix picture for a two layer model:





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:

```
function mlp3a(w, x0)
    x1 = relu(w[1] * x0 .+ w[2])
    x2 = relu(w[3] * x1 .+ w[4])
    return w[5] * x2 .+ w[6]
end
```

We can identify bad software engineering practices in this definition in that it contains a lot of repetition.

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.

Defining new operators

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

```
function mlp3b(w, x0)
    x1 = relu_layer1(w, x0)
    x2 = relu_layer2(w, x1)
    return pred_layer3(w, x2)
end

function relu_layer1(w, x)
    return relu(w[1] * x .+ w[2])
end
```

```
function relu_layer2(w, x)
    return relu(w[3] * x .+ w[4])
end

function pred_layer3(x)
    return w[5] * x .+ w[6]
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 functions like `relu_layer1` and `relu_layer2` are too similar except for the weights they use and can be reduced to a single function.

Increasing the number of layers

We can define a more general mlp model of arbitrary length. With weights of length 2n, the following model will have n layers, n-1 layers having the relu non-linearity:

```
function mlp_nlayer(w,x)
    for i=1:2:length(w)-2
        x = relu(w[i] * x .+ w[i+1]))
    end
    return w[end-1] * x .+ w[end]
end
```

In this example stacking the layers in a loop saved us only two lines, but the difference can be more significant in deeper models.

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 Julia you should be able to calculate by hand:

```
julia> w = KnetArray(reshape([1.0,2.0,3.0], (3,1,1,1)))
3×1×1×1 Knet.KnetArray{Float64,4}: [1,2,3]
julia> x = KnetArray(reshape([1.0:7.0...], (7,1,1,1)))
7×1×1×1 Knet.KnetArray{Float64,4}: [1,2,3,4,5,6,7]
julia> y = conv4(w, x)
5×1×1×1 Knet.KnetArray{Float64,4}: [10,16,22,28,34]
```

`conv4` 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 `conv4` 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.

```
julia> y = conv4(w, x; padding=(1,0))
7×1×1×1 Knet.KnetArray{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 `conv4` operation (the default stride is 1). In the following example we set stride to W such that the consecutive filter applications are non-overlapping:

```
julia> y = conv4(w, x; padding=(1,0), stride=3)
3×1×1×1 Knet.KnetArray{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 by choosing one of the following as the value of the `mode` keyword:

- `0` for convolution
- `1` for 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:

```
julia> w = KnetArray(reshape([1.0:4.0...], (2,2,1,1)))
2×2×1×1 Knet.KnetArray{Float64,4}:
[:, :, 1, 1] =
 1.0  3.0
 2.0  4.0
julia> x = KnetArray(reshape([1.0:9.0...], (3,3,1,1)))
3×3×1×1 Knet.KnetArray{Float64,4}:
[:, :, 1, 1] =
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
julia> y = conv4(w, x)
2×2×1×1 Knet.KnetArray{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:

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

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:

$$\begin{aligned}[256, 256, 3] * [5, 5, 3, 25] &\Rightarrow [256, 256, 25] \\ [256, 256, 25] * [3, 3, 25, 50] &\Rightarrow [256, 256, 50] \\ [256, 256, 50] * [3, 3, 50, 75] &\Rightarrow [256, 256, 75]\end{aligned}$$

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.

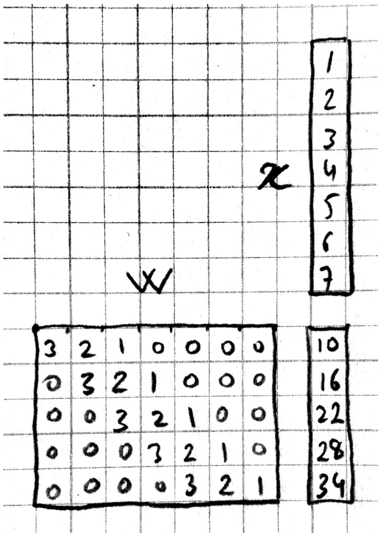
By the way, the arrays in this particular example already exceed 5GB of storage, so you would want to use a smaller minibatch size if you had a K20 GPU with 4GB of RAM.

Note: All the dimensions given above are for column-major languages like Julia. 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.

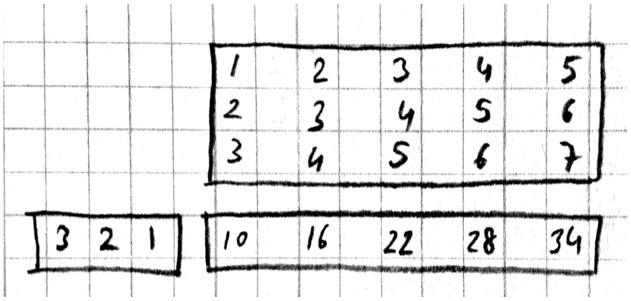
Here is a matrix implementation for our first convolution example $w = [1 \ldots 3]$, $x = [1 \ldots 7]$, $w * x = [10, 16, 22, 28, 34]$:



$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{bmatrix}$$

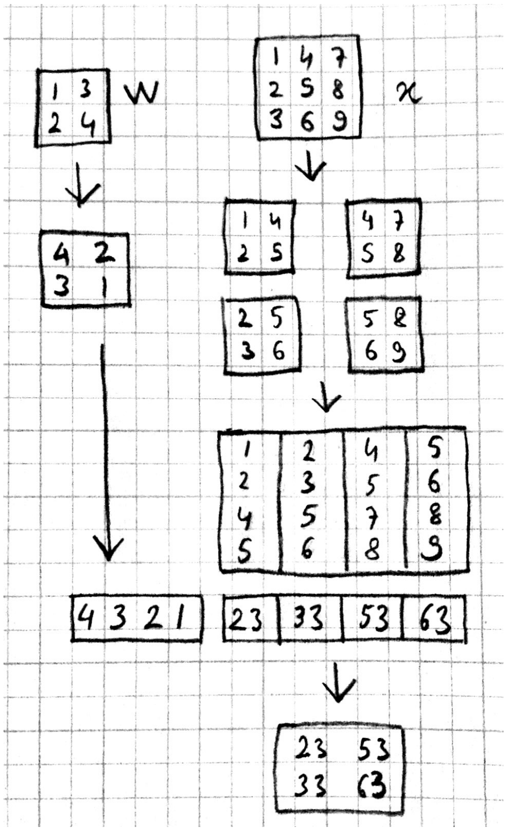
$$\begin{bmatrix} 3 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 3 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 3 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 3 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 10 \\ 16 \\ 22 \\ 28 \\ 34 \end{bmatrix}$$

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
julia> x = KnetArray(reshape([1.0:3.0...], (3,1)))
3×1 Knet.KnetArray{Float64,2}:
 1.0
 2.0
 3.0
julia> w = KnetArray(reshape([1.0:6.0...], (2,3)))
2×3 Knet.KnetArray{Float64,2}:
 1.0  3.0  5.0
 2.0  4.0  6.0
julia> y = w * x
2×1 Knet.KnetArray{Float64,2}:
 22.0
 28.0
julia> x2 = reshape(x, (3,1,1,1))
3×1×1×1 Knet.KnetArray{Float64,4}:
[:, :, 1, 1] =
 1.0
 2.0
 3.0
julia> w2 = KnetArray(reshape(Array(w)', (3,1,1,2)))
3×1×1×2 Knet.KnetArray{Float64,4}:
[:, :, 1, 1] =
 1.0
 3.0
 5.0
[:, :, 1, 2] =
 2.0
 4.0
 6.0
julia> y2 = conv4(w2, x2; mode=1)
1×1×2×1 Knet.KnetArray{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 $y = w \star x$:

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

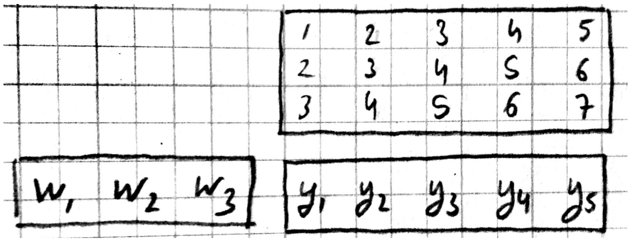
As you can see, because of weight sharing the same w entry is used in computing multiple y entries. This means a single w entry effects the objective function through multiple paths and these effects need to be added. Denoting

$\partial J/\partial y_i$ as y'_i for brevity we have:

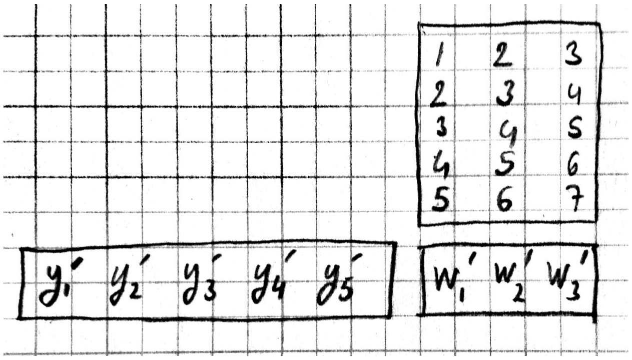
$$w'_1 =$$
$$x_1 y'_1 + x_2 y'_2 + \dots$$
$$w'_2 =$$
$$x_2 y'_1 + x_3 y'_2 + \dots$$
$$w'_3 =$$
$$x_3 y'_1 + x_4 y'_2 + \dots$$
$$\dots$$

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:

$$x'_1 =$$
$$w_1 y'_1$$
$$x'_2 =$$
$$w_2 y'_1 + w_1 y'_2$$
$$x'_3 =$$
$$w_3 y'_1 + w_2 y'_2 + w_1 y'_3$$
$$\dots$$

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:

```
julia> x = KnetArray(reshape([1.0:6.0...], (6,1,1,1)))
6×1×1×1 Knet.KnetArray{Float64,4}: [1,2,3,4,5,6]
julia> pool(x)
3×1×1×1 Knet.KnetArray{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:

```
julia> x = KnetArray(reshape([1.0:6.0...], (6,1,1,1)))
6×1×1×1 Knet.KnetArray{Float64,4}: [1,2,3,4,5,6]
julia> pool(x; window=3)
2×1×1×1 Knet.KnetArray{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 $Y = \lfloor X/W \rfloor$ 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:

```
julia> x = KnetArray(reshape([1.0:6.0...], (6,1,1,1)))
6×1×1×1 Knet.KnetArray{Float64,4}: [1,2,3,4,5,6]

julia> pool(x; padding=(1,0))
4×1×1×1 Knet.KnetArray{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 $Y = \lfloor (X + 2P)/W \rfloor$ 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`. Here is a 1-D example with a stride of 4 instead of the default 2:

```
julia> x = KnetArray(reshape([1.0:10.0...], (10,1,1,1)))
10×1×1×1 Knet.KnetArray{Float64,4}: [1,2,3,4,5,6,7,8,9,10]

julia> pool(x; stride=4)
4×1×1×1 Knet.KnetArray{Float64,4}: [2, 6, 10]
```

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:

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

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 `0` (max pooling) 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. `mode` should be `1` for averaging including padding, and `2` for averaging excluding padding. 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=0 => [1,3,5,6]
mode=1 => [0.5, 2.5, 4.5, 3.0]
mode=2 => [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:

```
julia> x = KnetArray(reshape([1.0:16.0...], (4,4,1,1)))
4×4×1×1 Knet.KnetArray{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> pool(x)
2×2×1×1 Knet.KnetArray{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:

```
julia> x = KnetArray(rand(4,4,2,1))
4×4×2×1 Knet.KnetArray{Float64,4}:
[:, :, 1, 1] =
 0.880221  0.738729  0.317231   0.990521
 0.626842  0.562692  0.339969   0.92469
 0.416676  0.403625  0.352799   0.46624
 0.566254  0.634703  0.0632812  0.0857779

[:, :, 2, 1] =
 0.300799  0.407623   0.26275   0.767884
 0.217025  0.0055375  0.623168  0.957374
 0.154975  0.246693   0.769524  0.628197
 0.259161  0.648074   0.333324  0.46305

julia> pool(x)
2×2×2×1 Knet.KnetArray{Float64,4}:
[:, :, 1, 1] =
 0.880221  0.990521
 0.634703  0.46624

[:, :, 2, 1] =
 0.407623  0.957374
 0.648074  0.769524
```

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 = KnetArray(rand(4,4,1,2))
4×4×1×2 Knet.KnetArray{Float64,4}:
[:, :, 1, 1] =
 0.155228  0.848345  0.629651  0.262436
 0.729994  0.320431  0.466628  0.0293943
 0.374592  0.662795  0.819015  0.974298
 0.421283  0.83866   0.385306  0.36081

[:, :, 1, 2] =
 0.0562608  0.598084  0.0231604  0.232413
 0.71073    0.411324  0.28688    0.287947
 0.997445   0.618981  0.471971   0.684064
 0.902232   0.570232  0.190876   0.339076

julia> pool(x)
2×2×1×2 Knet.KnetArray{Float64,4}:
[:, :, 1, 1] =
 0.848345  0.629651
 0.83866   0.974298

[:, :, 1, 2] =
 0.71073   0.287947
 0.997445  0.684064
```

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. 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).
- http://josephpcohen.com/w/visualizing-cnn-architectures-side-by-side-with-mxnet visualizing popular CNN architectures side by side with mxnet.
- http://cs231n.github.io/understanding-cnn visualizing what convnets learn.
- https://arxiv.org/abs/1603.07285 A guide to convolution arithmetic for deep learning

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