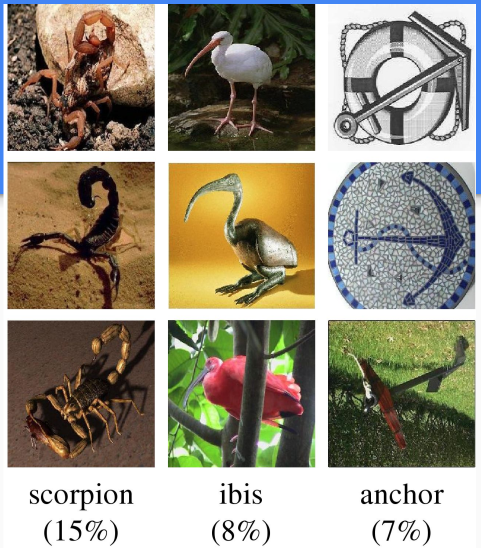
Convolutional neural networks for visual recognition in q

(with additional q-integrated CUDA/GPU functions for performance gains)

Ryan Sparks November 2017

Overview

- What this is all about/why I did this
- Background on visual recognition
- Rehash on neural nets
- What are convnets and important layers/general architecture of convnets
- Key techniques to improve accuracy
- How I got this working/issues encountered



https://raweb.inria.fr/rapportsactivite/RA2007/lear/uid64.html

+

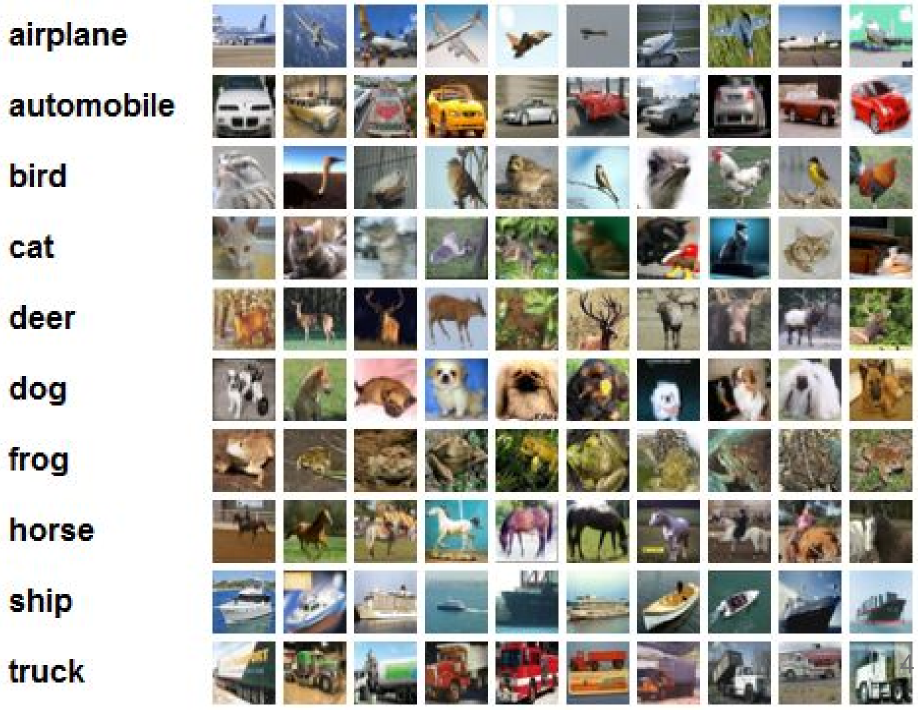
q)

Overview

- Improvements I found helpful
- Setting up Google Cloud to run this (with a GPU), and hook up to a cell phone
- Using CUDA to complement q, some egs (matrix multiply, shortest path)
- Overall assessment of doing this in q vs existing languages/frameworks
- Future plans/TODOs

What’s this about

- Image classification in q
- Following stanford’s cs231n class, implemented using the CIFAR 10 dataset: https://www.cs.toronto.edu/~kriz/cifar.html (more detail later slides)
- 10 different classes, 49,000 training images, 1000 validation images



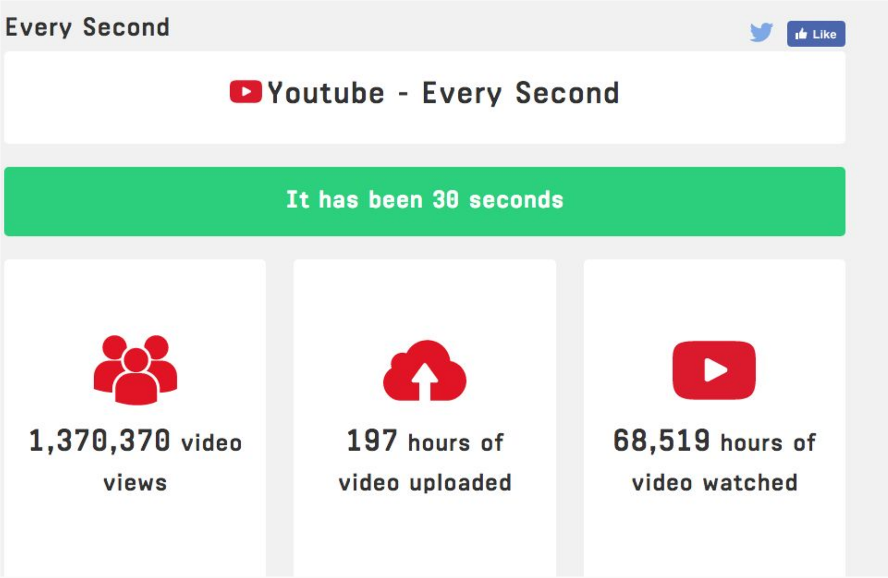
http://karpathy.github.io/2011/04/27/manually-classifying-cifar10/

Why

- Visual recognition seems to be one of the most exciting, useful, and rapidly expanding fields in technology/AI at the moment
- Wanted to understand myself how it worked, best way for me to truly learn is write code to do something
- I like coding in q, for the same reason as everyone here (also the language I'm most proficient in)
- One of the most important/modern methods of visual recognition can be done in q - it’s not just dark magic
- Leads onto other exciting areas as I’ll mention later

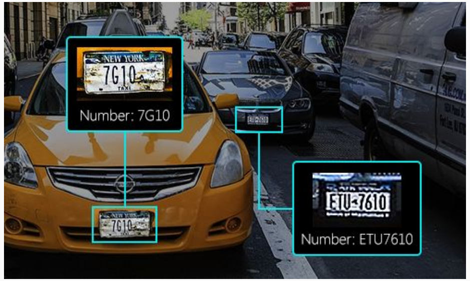
Visual recognition - importance

- Number of cameras in the world overtaking number of people, > 2 billion smartphones
- Up to 80% of all internet traffic is video
- Majority of data is visual data, critical, but difficult to for algos understand
- http://www.everysecond.io/youtube 6 hours of youtube every second
- Impossible for humans to look at and interpret all the visual data
- Convolutional neural networks recently emerged to help solve this problem

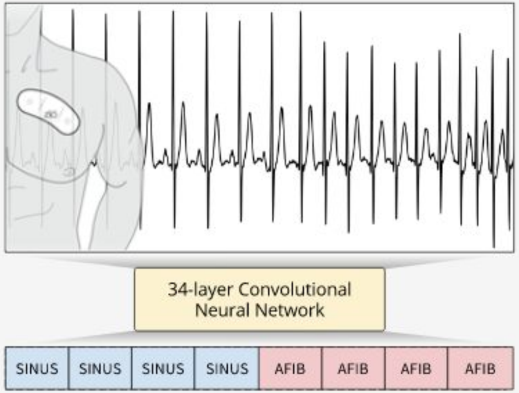


Convolutional neural networks: endless applications

https://c.tribune.com.pk/2016/09/1191315-selfdrivecar-1475231284.jpg

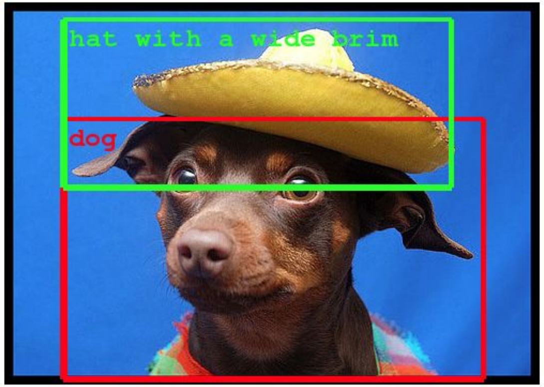


https://c.tribune.com.pk/2016/09/1191315-selfdrivecar-1475231284.jpg

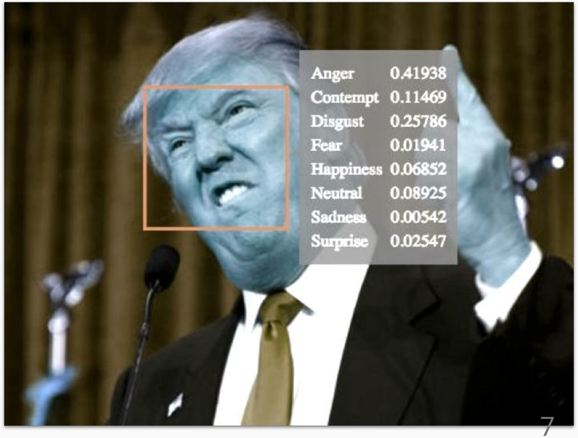


https://stanfordmlgroup.github.io/projects/ecg/

http://blog.twmg.com.au/wp-content/uploads/2015/09/colorized-historic-photo-15-30.jpg



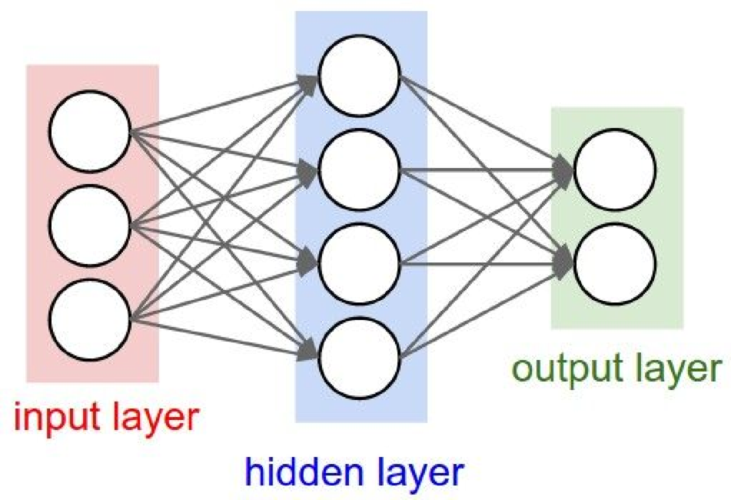
https://research.googleblog.com/2014/09/building-deeper-understanding-of-images.html



https://benheubl.github.io/data%20analysis/fr/

Rehash on neural nets

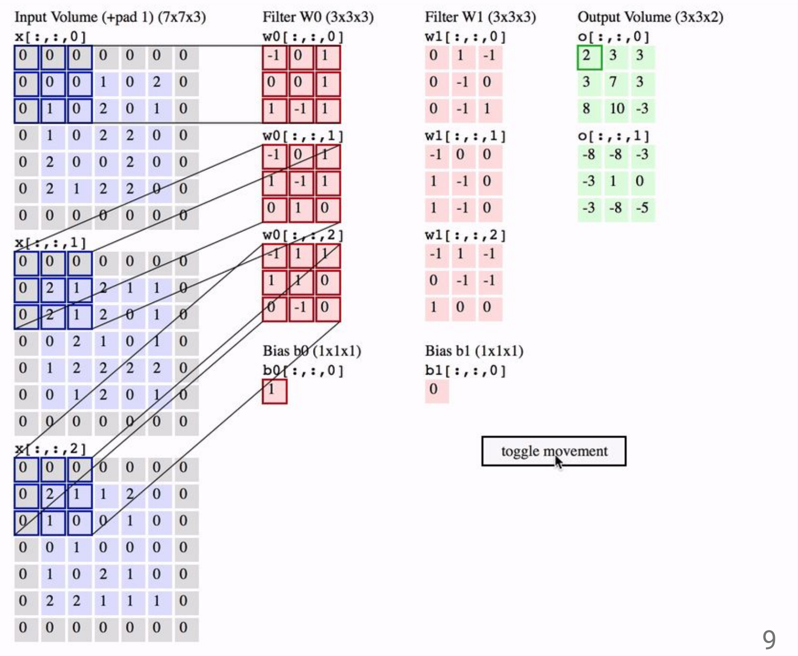
- Simplified models of brains
- Lots of neurons arranged in layers, with connections to all neurons in next layers
- Each connection has associated weight
- Training information fed through, certain neurons fire (activation function)
- Compare output to expected, then adjust weights using simple calculus, and repeat
- Primarily a matrix multiply (at least in compute time)



http://cs231n.github.io/assets/nn1/neural_net.jpeg

Convolutional neural networks (convnets)

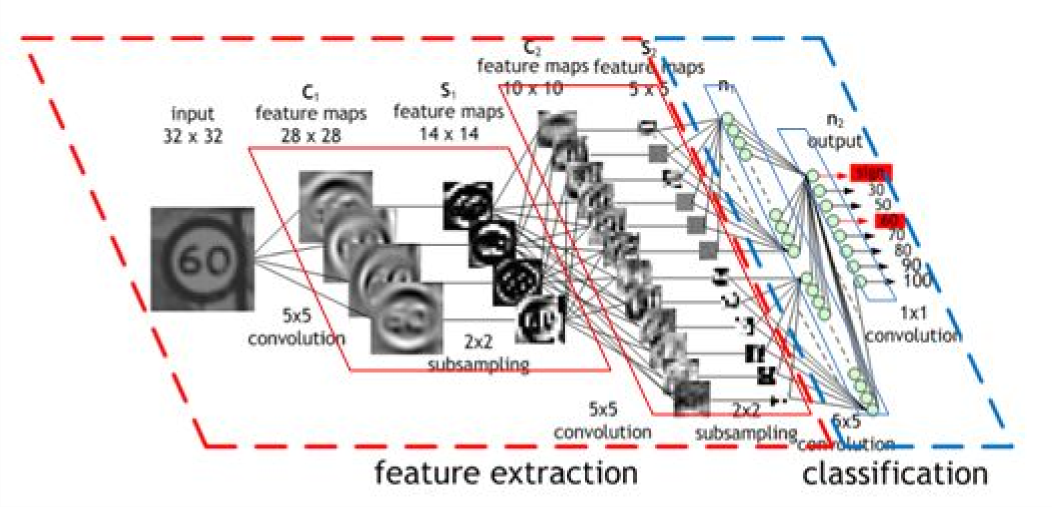
- Category of neural network that is very good at image recognition
- Been around since 80’s, but didn’t scale well until 2012 (when GPUs and huge image sets were available e.g ImageNet)
- 4 main layers:
  - Convolution
  - Non linearity (reLu)
  - Pooling
  - Classification (fully connected)



http://cs231n.github.io/assets/nn1/neural_net.jpeg

Convnets

- Convolution step extracts features from input image
- Slides a small filter across an image, pixels are treated near each other which helps make sense of the image and reduce number of parameters
- Useful for 3D data, as it's embedded in architecture



https://devblogs.nvidia.com/wp-content/uploads/2015/11/fig1.png

Convnet layers

- Convolution layer: main function is equivalent to python's np.lib.stride_tricks.as_strided

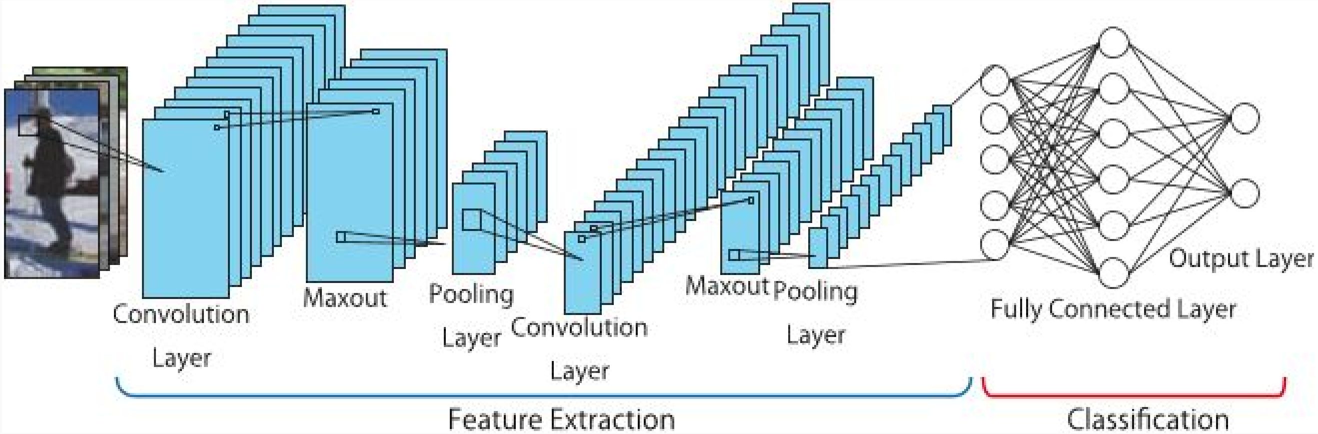
```
q) asStrided:{[m;newshape;strides] newshape#razeo[m]@{raze x+/:raze y}/[reverse[strides]*til each reverse newshape]}
q)\ts 0N! shape asStrided[50 3 34 34 #1f ; 3 3 3 50 32 32; 1156 34 1 3468 34 1 ]
3 3 3 50 32 32
29 40128704
```

- Non linearity: reLu layer, easiest one (advantage over sigmoid is doesn't suffer same vanishing gradient problem when many layers): `0|`
- Pooling: form of subsampling, reduces dimensionality

```
{[m;axes] {[x;ind].[x;ind#(::);max]}/[m;axes]}
```

Convnet layers

- Fully connected: every neuron connected to every neuron in the next layer.
- So flow of information between each input dimension (pixel location) and each output cass. Essentially matrix multiply: `mmu/.qml.mm`



https://ai2-s2-public.s3.amazonaws.com/figures/2016-11-08/2cc00efd62dc6b2ee10918b80bed50cb09ad5cac/1-Figure1-1.png

Convnet layers

- Softmax: takes vector of arbitrary values (from fc layer) and squashes it so that they sum to 1:

```
//  x: Input data, of shape (N, C) where x[i, j] is the score for the jth
class for the ith input.
// y: list of labels, length N, where y[i] is the label for x[i] a
{[x;y]
    probs:{x%sum each x}exp x- max each x;
    loss:sum neg[log probs@'y]%count x;
    dx:@'[probs;y;-;1]%N;
    (loss;dx)
}
```

Design choices that improve accuracy

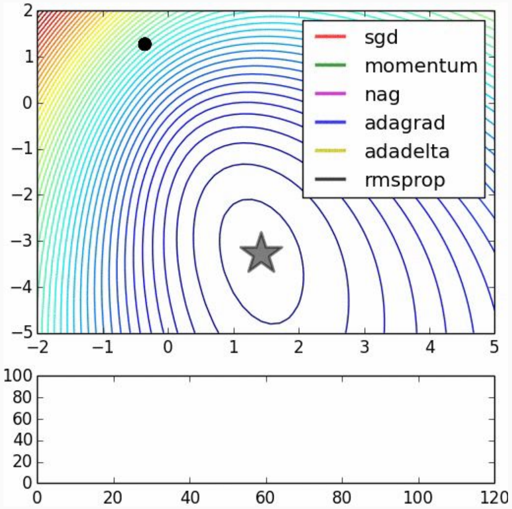
- Batch normalization: forcing gaussian distribution at the start, after conv layers, and fully connected layers (helps prevent init. problems too)
- Regularization: penalizing the squared magnitude of all parameters directly in the objective `loss:dataLoss+0.5*d[`reg]*{x$x}razeo d@d`wParams`
- Dropout: while training, dropout is implemented by randomly setting neurons to 0, with probability p: `x*shapex#((prd[shapex:shape x]?1f)<p)%p`
- Data augmentation: Flip the training images over x-axis; Sample random crops in original image; jitter colors (better to preprocess): `@[xBatch;{neg[x div 2]?x}d`batchSize;reverse each]`

https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

Design choices that improve accuracy

Different parameter update functions can help dramatically

- sgd: vanilla update, change the parameters along the negative gradient direction - `w-dw*config`learnRate`
- sgd with momentum: similar, parameter vector will build up velocity in any direction that has consistent gradient -
```
v:(v*config`momentum)-dw*config`learnRate;
config:config,enlist[`velocity]!enlist v;
(w+v;config)
```
- rmsProp: adaptive learning rate, uses moving avgs and squared gradients -
```
cache:(cache*updateDecayRate)+(1-updateDecayRate)*dx*dx;
nextX:x-learnRate*dx%epsilon+sqrt cache;
config[`cache]:cache;
(nextX;config)
```
- adam: similar/improved versions of rmsProp and adagrad, also uses momentum



https://amva4newphysics.files.wordpress.com/2017/04/all.gif?w=736

Varying hyperparameters

- Initial learning rate
- Learning rate decay
- Regularization strength
- Randomization seems to be better than methodical
- Peach can be useful here

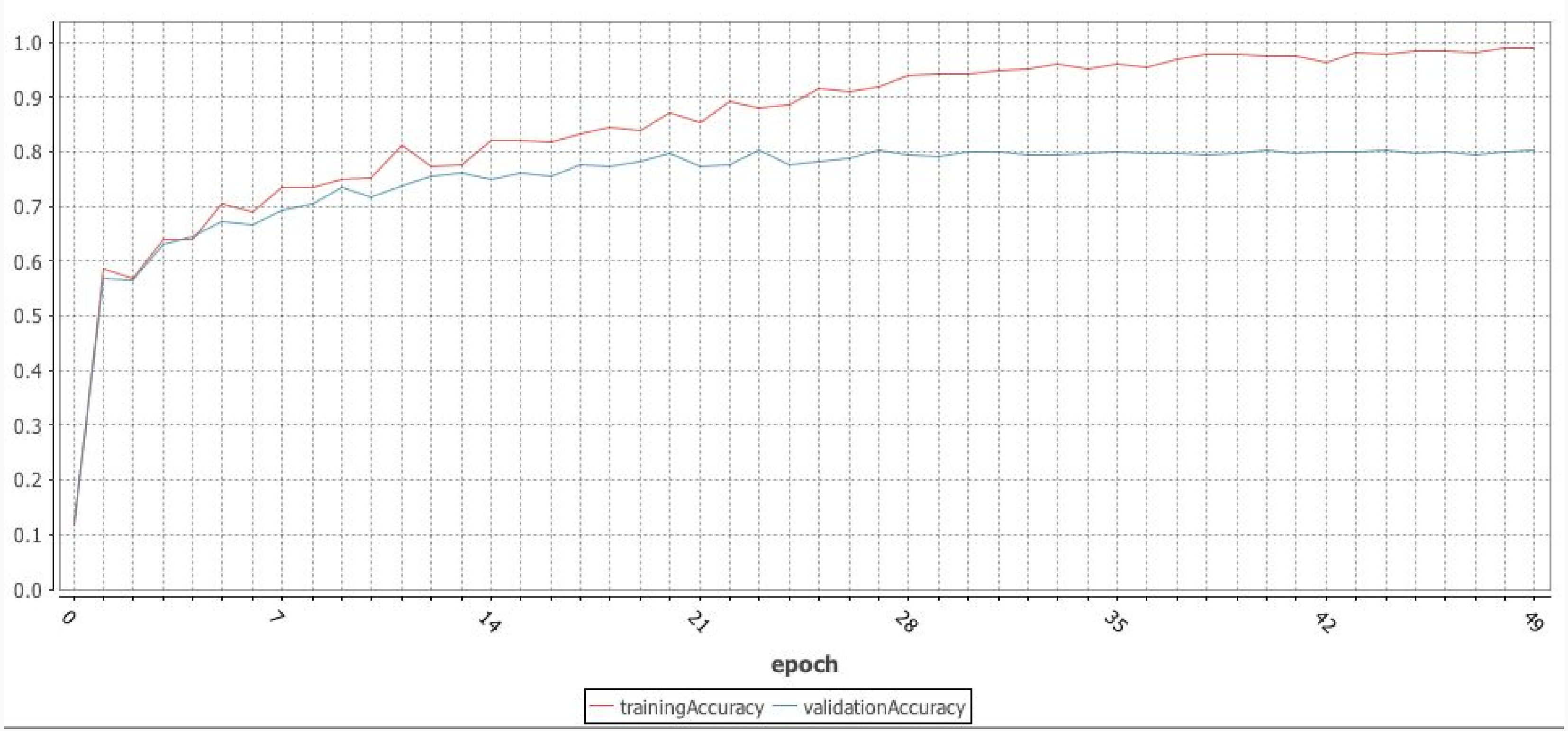
```
randomLearnRates:lrRange[0]+numRandoms?lrRange[1]-lrRange 0;
randomRegs:regRange[0]+numRandoms?regRange[1]-regRange 0;
(.solver.train @[d;`lr`reg;:;]@)peach randomLearnRates,'randomRegs
```

How I went about this

- Followed cs231n stanford course, all lectures on youtube, slides, github repo and notes freely available online
- Lots of googling, heaps of material online (blogs, githubs etc)
- Followed stanford's assignments, which are all in python/numpy
- Approx 50% of the code is provided the they leave you the rest to fill in
- I didn't really know python, used pyq/qpython to go back and forth between and workout what was really happening

Results

Best validation accuracy I found was slightly over 80% with a 5 layer convnet (compared to around 52% with a deep neural net). Definitely some overfitting here unfortunately:



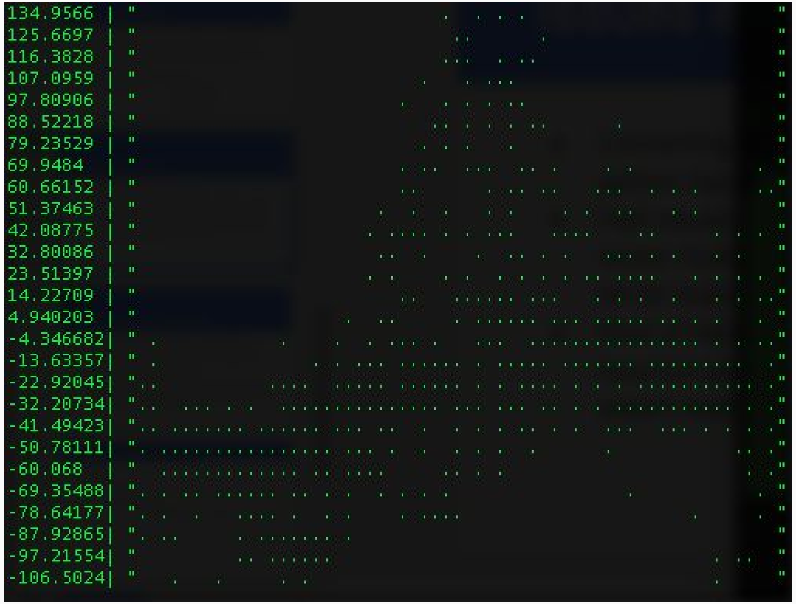
Issues encountered

- Had to learn python for this, as the course was taught in that, found pyq and qpython to be really useful
- Plenty of utils unfortunately seem to exist in numpy that aren't really well documented, had to write these from scratch in q via guess work and comparing with python.
- Wish q had multi variable assignment (for local variables inside functions, like python does), example of current work around:

```
/ convolution layer
conv_convCache:convForwardFast[x;w;b;convParam]; / returns list (conv;convCache)
/ spatial batchnorm layer
norm_normCache:spatialBatchNormForward[ conv_convCache 0;gamma;beta;bnParam]
..
cache:`convCache`normCache`reluCache!( conv_convCache 1;normCache;reluCache);
```

Issues encountered

- Converting an actual photo into an array of floats is more difficult than I had thought,
- Used python library
- 32bit w-aborting on processing the input data – so had to do some messy coding for that,
- kx fortunately gave me a temporary 64 bit license for this project
- Hard to visualize red-green-blue color photos in kdb, I tried Nick Psaris's plot function just using one color channel (as it's designed for black and white), gave up



Pictured: cat (I think?)

Things I found helpful

- Deals primarily with "4 dimensional" data, i.e. lists of 3d data, eg. `2 3 4 5#til 120`.
- This was the obvious first way to do everything (can somewhat see it, in the same way you can in python, making it easy to compare)
- A big bottleneck is matrix multiply, using Andrey Zholos's qml library helped out quite a bit (e.g. 3-5 times faster than q's mmu).
- The slowest function involved a 6 level nested for loop – tried it in q, but quickly decided it needed c (ended up going from the slowest function in the whole project, to one of the fastest)

Deep loop - backward pass for convolution layer

```
// modify arg2's elements, using indexing into arg1
for(n=0;n<N;++n){
   for(c=0;c<C;++c){
      for(hh=0;hh<HH;++hh){
          for(ww=0;ww<WW;++ww){
              for(h=0;h<out_h;++h){
                  for(w=0;w<out_w;++w){
                      kF(kK(kK(kK(arg2)[n])[c])[(stride*h+hh)])[stride*w+ww] +=
                         kF(kK(kK(kK(kK(kK(arg1)[c])[hh])[ww])[n])[h])[w];
                  }
              }
          }
      }
   }
}
```

Things I found helpful

Conceptually I find it easier to work in actual matrices (and higher order list of lists), think numpy just has views on a list of data. The time to reshape adds up though!

```
q)reshapeM:{[m1;m2Shape] m2Shape#razeo m1}
```

Eg turning a 6 dimensional matrix into a 2D one for matrix multiply.

```
q)\ts reshapeM[m;27 51200]
14 30933568
>>> m=np.random.randn(3,3,3,50,32,32)
>>> now=time.time(); res=np.reshape(m,(27,51200));time.time()-now
4.100799560546875e-05 // 0.00004 seconds
```

Things I found helpful

- Began experimenting with gpus and cuBLAS (after Nick mentioned he'd like to try)
- Found that inputs and outputs had to be flat lists, so forced to adapt
- Was only a few weeks ago, I haven't had time to refactor the convnet stuff.
- Haven't had time to refactor convnet stuff, but have rewritten the simpler neuralnet stuff to use lists:

```
q)(::)m:{(x;prd[x]#1f)}3 3 3 50 32 32
3 3 3 50 32 32
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ..
q)reshapeMNew:@[;0;:;]
q)reshapeMNew[m;27 51200]
27 51200
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ..
q)\t:10000 reshapeMNew[m;27 51200]
9
```

Things I found helpful

- Keeping everything as a list, had to make a few additional c binaries
- E.g doing “flip” on a flat version of a matrix, without using inbuilt flip

```
q)m:{(x;prd[x]?100.)}1000 2000         // list version of matrix
(shape;list)
q)m2:(#). m                            // matrix version of m
q)\ts res:flip m2                      // normal matrix flip
54 16400672
q)\ts resSlow:raze flip (#). m         // need to avoid this
72 33176800
q)\ts resFlat:flipFlat[m 1;1000;2000]  // using c shared object flipFlat
30 16777760
q)all(resSlow~resFlat;resFlat~raze res)
1b
```

Example c func needed for list versions

```
K flipFlat(K flatm, K nrows, K ncolumns){
    I i,j,rows,columns,r,c,index1,index2;
    F resvalue;
    rows=nrows->n;
    columns=ncolumns->n;
    K emptyres = ktn(KF,(rows*columns));
    for(r=0;r<rows;++r){
        for(c=0;c<columns;++c){
            index1=r*columns+c;
            index2=c*rows+r;
            resvalue=kF(flatm)[index1];
            kF(emptyres)[index2]=resvalue;
        }
     };
     R(emptyres);
}
```

Things I found helpful

- Function profiling with prof.q
- Using some of the new inbuilt (v 3.5+) debug functions

```
/ utility, determine which parent function is calling a function
.prof.getParentFunc:`$.[;1 1 0]{1_.Q.btx@-100!`}@
```

Eg: find out which functions are calling dot, and profile them in a table, without modifying anything but the dot function:

```
dotTab:([]time:"n"$();func:`$();shape1:();shape2:())
dot:{[x;y]now:.z.n;res:.qml.mm[x;y];
    `dotTab insert (.z.n-now;.prof.getParentFunc[];enlist shape x;enlist shape y);
    res};
```

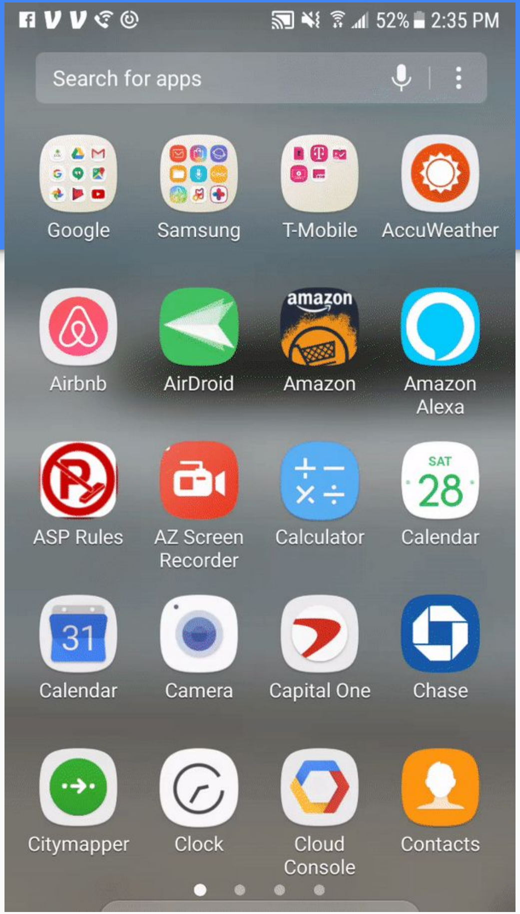
Speedup from converting neural net to use just flat lists instead of matrices

nc	timepc	timepc_old	pct	pct_old	faster
dot	4.20	189.84	67.10	81.92	1
affineBackward	1.57	17.00	7.87	1.29	1
adam	0.72	17.22	7.19	2.62	1
solver.xTrainParser	5.73	79.71	5.85	1.54	1
fullyConnectedNet.loss	4.65	11.18	5.54	0.62	1
solver.genBatch	2.53	35.30	2.54	0.54	1
solver.step	1.80	26.36	1.80	0.40	1
softmaxLoss	0.49	6.13	0.49	0.09	1
reluBackward	0.12	16.11	0.48	0.98	1
solver.checkAccuracy	8.14	8.68	0.15	0.04	1
fullyConnectedForwardPassLoop	0.02	0.02	0.14	0.01	0
affineForward	0.02	1.28	0.13	0.36	1
fullyConnectedBackwardPassLoop|	0.02	0.03	0.12	0.00	1
solver.i.step	0.12	3.41	0.12	0.05	1
reluForward	0.02	1.22	0.12	0.27	1
symi	0.00	0.00	0.11	0.00	1
randArrayFlat	9.90	0.08	0.00		0
affineReluForward	0.01	0.01	0.04	0.00	1
getModelValue	0.01	0.02	0.03	0.00	1
affineReluBackward	0.01	0.40	0.02	0.02	1

..

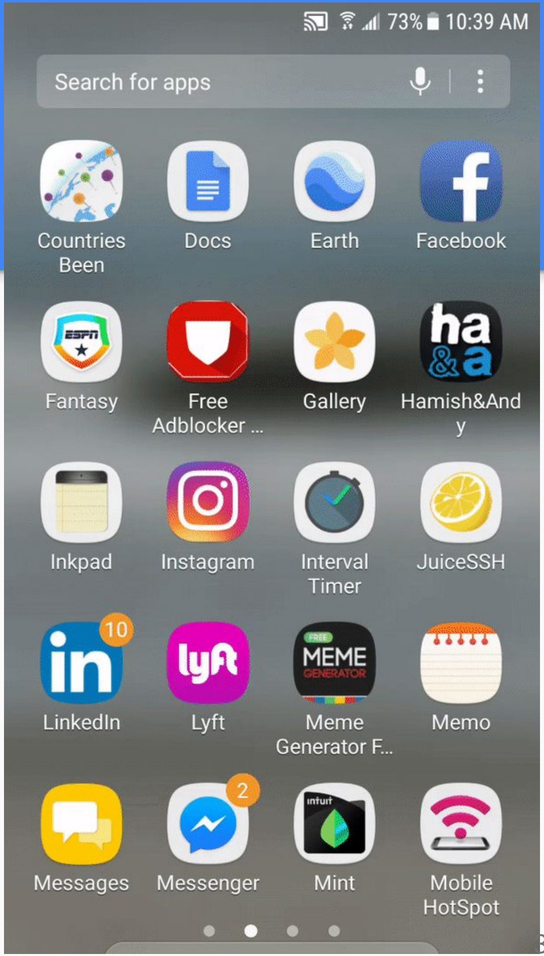
Using a cloud to train

- A lot of convent stuff involves letting it just sit there and "train"
- Not really practical for someone with only one laptop
- Investigated amazon cloud, was really fast but blew heaps of money
- Discovered Google cloud gave you $300 free credit for the first year, way cheaper per hour too (but slower)
- Included my setup commands on github, as setting up a server from scratch may not be obvious
- Can take a few days to train my models



Manage cloud with cell phone

- Can setup your cell phone to ssh into it and control on the go,
- So can run q code on a pretty powerful box using your phone
- I used https://juicessh.com/
- https://www.youtube.com/watch?v=mkfzyQJJoZs
- Useful for tailing logfiles to check on training
- If you really have the urge to test out a q coding idea you can



Using GPUs for speedups

- Wanted to test out trying to implement matrix multiplication using gpus and kdb
- Since I was setting up the cloud figured I'd give it a go
- Set up a cloud with a Tesla K80 gpu ($4k on amazon), you can still do it without paying money on google cloud, it just eats away at your $300 credit much faster
- Have included all the cuda install steps etc. in my github

```
wget https://developer.nvidia.com/compute/cuda/9.0/Prod/local_installers/cuda-repo-ubuntu1604-9-0-local_9.0.176-1_amd64-deb
sudo apt-key add /var/cuda-repo-9-0-local/7fa2af80.pub
sudo dpkg -i cuda-repo-ubuntu1604-9-0-local_9.0.176-1_amd64-deb
sudo apt-get update
sudo apt-get -y install cuda
#check
cat /var/lib/apt/lists/*cuda*Packages | grep "Package:"
# add to bash_profile
export PATH=/usr/local/cuda-9.0/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-9.0/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
#install samples
cuda-install-samples-9.0.sh cudaSamples
cd cudaSamples/NVIDIA_CUDA-9.0_Samples
make
```

Using GPUs for speedups

Compiling wasn't obvious to me:

```
nvcc --compiler-options '-fPIC -DKXVER=3 -O2' -o $QHOME/l64/gpu_mmf.so --shared -lcurand
-lcublas gpu_mmf.cu
```

The actual code ended up being straightforward:

```
// inputs (A: list, input matrix; rA: # of rows in A; cA: # cols in A; B: input list matrix;
rB: # rows in B; rC: # rows in B)
extern  "C" K gpu_mmf(K A, K rA, K cA, K B, K rB, K cB);
..
cudaMemcpy(d_A, host_memoryA, sizeA, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, host_memoryB, sizeB, cudaMemcpyHostToDevice);
..
// Multiply A and B on GPU
cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc);
..
// Copy the result on host memory
cudaMemcpy(host_memoryC,d_C,nr_rows_C*nr_cols_C*sizeof(double),cudaMemcpyDeviceToHost);
..
```

Using GPUs for speedups

Performance was awesome!

```
q) a:1000 2000#aflat:2000000?10f
q) b:2000 3000#bflat:6000000?10f
q)\t flatres:.gpu.mm[aflat;1000;2000;bflat;2000;3000]
time to allocate host and device array mems: 1.358000ms
time to copy inputs to GPU: 9.041000ms
time to perform cublas matrix multiply:  0.024000ms
time to copy result from GPU back to host: 3.494000ms
40
```

Compared mmu and .qml.mm:

```
q)\t res2:mmu[a;b]
2051
/ using qml
q)\t res3:.qml.mm[a;b]
485
q)res2~flip 3000 1000#flatres
1b
```

Using GPUs for speedups

Had a go at the shortest paths problem from the listbox

```
 __global__ void fw_kernel(const unsigned int u, const unsigned int n, int * const d){
     I v1 = blockDim.y * blockIdx.y + threadIdx.y;
     I v2 = blockDim.x * blockIdx.x + threadIdx.x;
     if (v1 < n && v2 < n){
         I newPath = d[v1 * n + u] + d[u * n + v2];
         I oldPath = d[v1 * n + v2];
         if (oldPath > newPath)
         {
            d[v1 * n + v2] = newPath; ...
```

Was approx 10x faster than the fastest q function (even with q using 6 slaves, and my lack of CUDA skills)

function	2000x2000 GPU server 0 slaves	2000x2000 on fast box+6 slaves	4000x4000 GPU server 0 slaves	4000x4000 fast box+6 slaves
bridgeq	13365 49250400	3446 32826032	134495 196802624	33971 131187376
bridgeqUDA	388 592	n/a	2890 592	n/a

Compared to other existing frameworks

- At first, even python/numpy was much faster
- After converting to use lists etc much closer
- In theory, the tools are there to be able to compete – using CUDA, customized compiled C, fast data pulling etc (kdb's forte)
- Already many highly optimized frameworks out there, built full time by teams of people at eg. google/facebook, can easily configure them for say multiple GPUs, and even TPUs.

Future work

- Some more work on recurrent neural networks
- Deep dreaming
- Image captioning
- Experiment with "minimum character-level RNN language model", there's a 110 line python script written by Andrej Karpathy (now director of AI at Tesla). Converted it to 48 lines of q, without being ridiculous/losing readability (not golf code). Reads in a text, starts "hallucinating" new text. Some of the better models have computers writing Shakespeare, baby names, writing linux source code in C, pure math theorems in Latex

https://deepdreamgenerator.com/

http://cv-tricks.com/artificial-intelligence/show-attend-tell-image-captioning-explained/

Acknowledgements/further reading

Stanford university's full course online (including lectures): http://cs231n.stanford.edu/

Andrej Karpathy's blog/github: http://karpathy.github.io/

Lee Zhen Yong's blog/github: https://bruceoutdoors.wordpress.com/cs231n-tutorials/

Nick Psaris's github: https://github.com/psaris/funq

James Neil's paper: https://kx.com/blog/an-introduction-to-neural-networks-with-kdb/

Thanks!

All code will be available on my github

https://github.com/ryantorsparks/

rspa9428@gmail.com