Freie Universität Berlin

Data Science
Lecture 12

11.07.2014

Dr. Kashif Rasul
kashif @krasul #167

Shoaib Burq
sabman @sabman #167

Welcome to Data Science Lecture 12.

Last Time

- SVM
- Unsupervised Methods:
  - PCA
  - K-Means
  - Hierarchical clustering

Today

- Neural Networks

Today we will give you an intuition about Neural Networks and why and how they work for non-linear supervised machine learning problems. These are older ideas that went out of fashion but they are back in trend nowadays and are among the most sophisticated methods around.

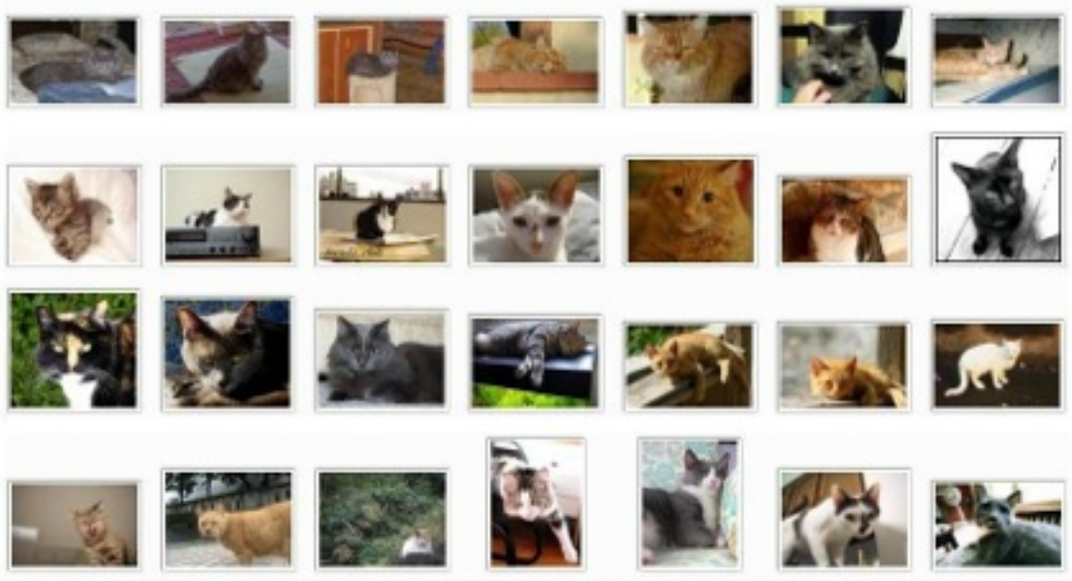
Why Neural Networks?

- For non-linear classification via logistic regression we could specify many non-linear features and their powers in a model
- When p=100, a quadratic model will have about 5000 features… and this grows $O(p^2)$
- We could include a subset of features but it will not fit the non-linear hypothesis too well

And logistic regression works well when we have few features… but for a lot of interesting problems we have a lot more features.

And including a cubic model for p=100 we have 170K features and at this point this is not efficient computationally.

Object Recognition in Images



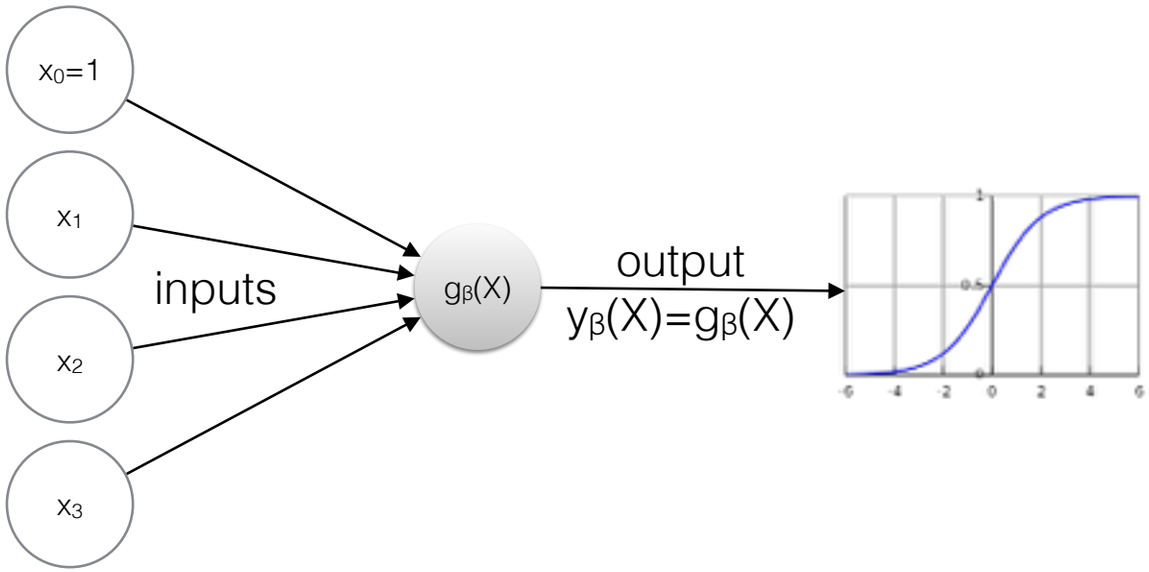
Consider the problem of object recognition in images. A simple problem for a brain, but computers can only see pixels values. For a 32x32 pixel image we end up with 1024 pixels and if it is RGB that is p=3072 features. And each input is a vector of length 3072.

So now if were to fit say a quadratic model we would end up with around 9 million features! And for a 100x100 grayscale image we would have around 50 million quadratic features. So this is not a good way to learn such hypothesis.

Neural Networks

- Algorithms that try to mimic the brain in particular neural connections
- After a lull in the 90's have become more popular
- Recently: provide state of the art techniques for many applications

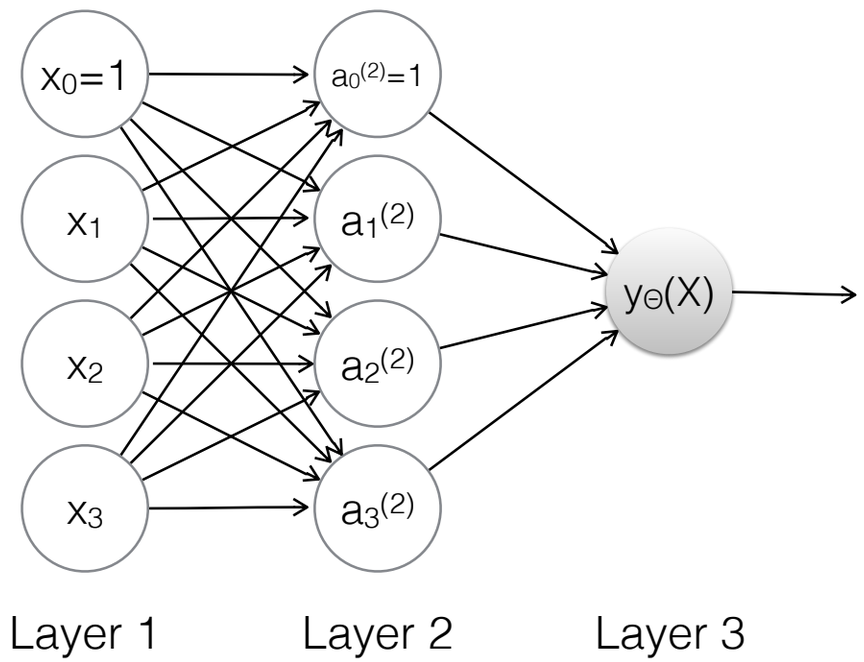
The recent resurgence in neural networks is manly due to faster computers (since they are computationally expensive) and bigger labeled data sets for training these networks.



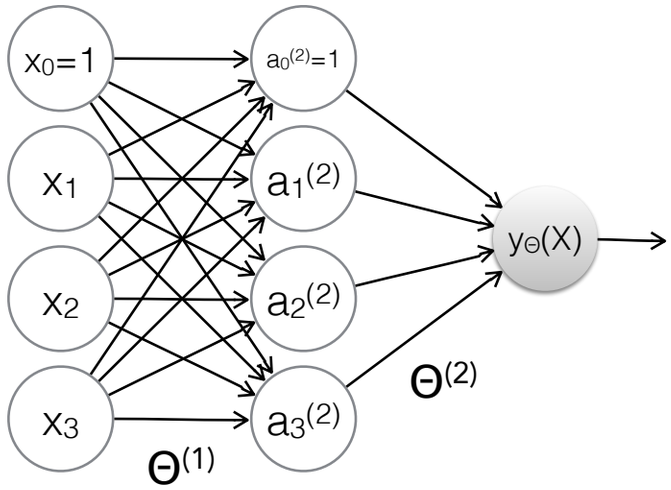
The simplest possible neural "network" we can consider is a single neuron denoted by this diagram and called a Logistic Unit where $y_\beta(X) = 1/(1+\exp(-\beta^T X))$ for some coefficient vector β. The x_0 node is called the bias unit and always outputs 1.

This is an extremely simple model of what a neuron does, it gets a number of inputs and it outputs some value computed by the function which in this case is the sigmoid or logistic function with parameter β.

In the neural network lingo the β is also referred to as weights.



Then a neural network is a bunch of these neurons connected together as in the diagram above. So now the Layer 1 is called the input layer. Layer 3 is called the output layer and outputs our hypothesis. Layer 2 in between, actually any layer in between these two layers is called the hidden layer.

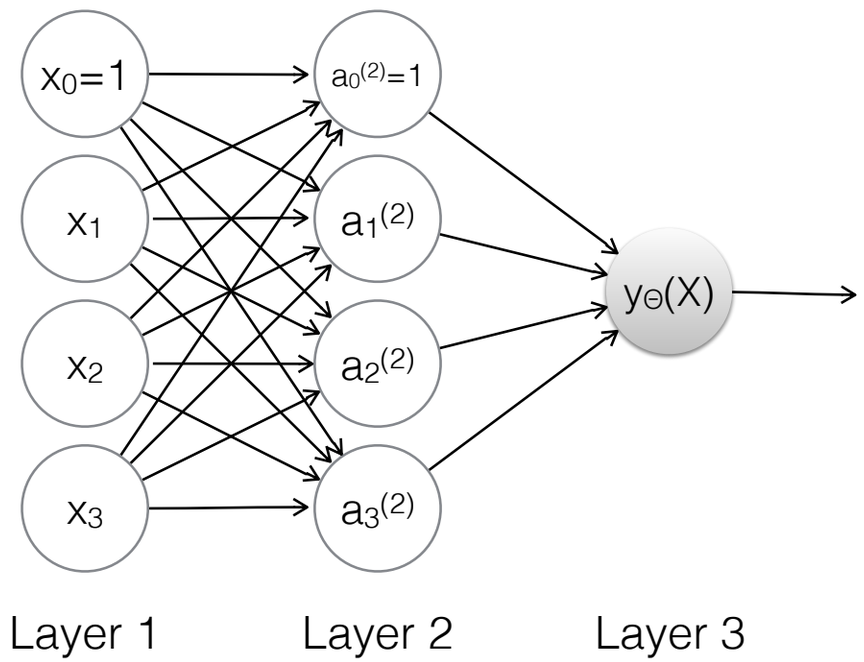


$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$

$$y_\Theta(X) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

So what is happening computationally? Well for each $\Theta^{(j)}$ a given matrix of weights controlling function mappings from layer j to layer j+1 we can compute the activation $a_i^{(j)}$ of unit i in layer j by the Logistic function g().

So this defines a neural network for mapping from X to our model given by some weights Θ. This is called forward propagation. We propagate the activations forward.

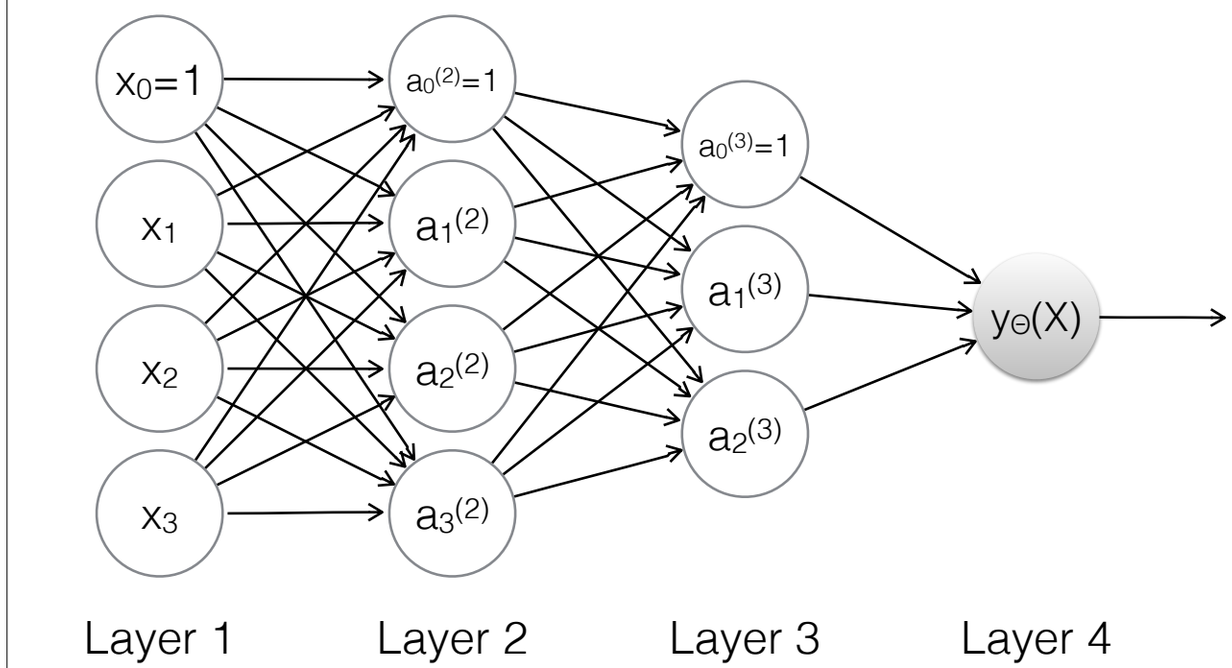


So now an intuition about what neural networks might be doing and how they might help us in learning non-linear hypothesis. If we hid Layer 1, what is left is just logistic regression by using the features from Layer 2. So Layer 3 just computes the normal logistic function from the with the weight matrix playing the role of the parameters and using features a_1, a_2 and a_3.

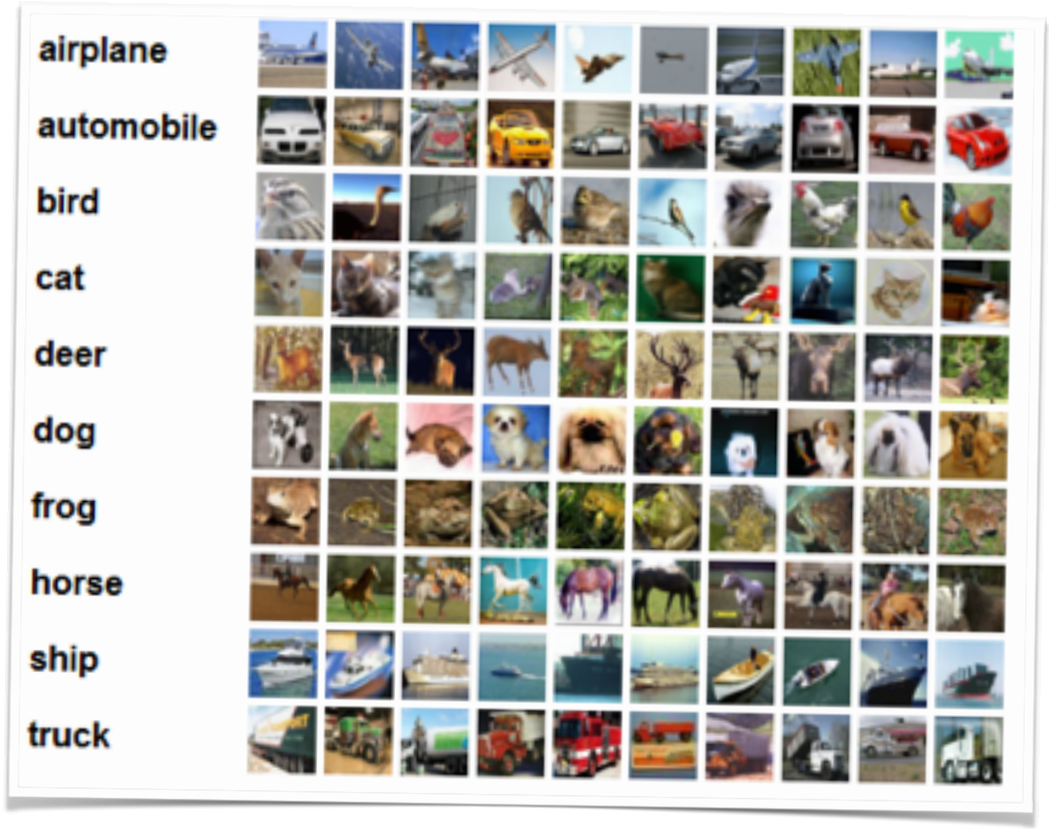
The great thing now is that these a_1, a_2 and a_3 are learned as functions of the inputs. So some other parameters $\Theta^{(1)}$ is choosing the parameters to then feed into the Layer 3 which is doing logistic regression. So now depending on what $\Theta^{(1)}$ is set to (via learning) we can learn interesting features of our data.

Therefore we hope to get a better estimate than just polynomial regression because we have the flexibility to learn whatever features we want in order to get the Layer 2 which we feed into Layer 3.

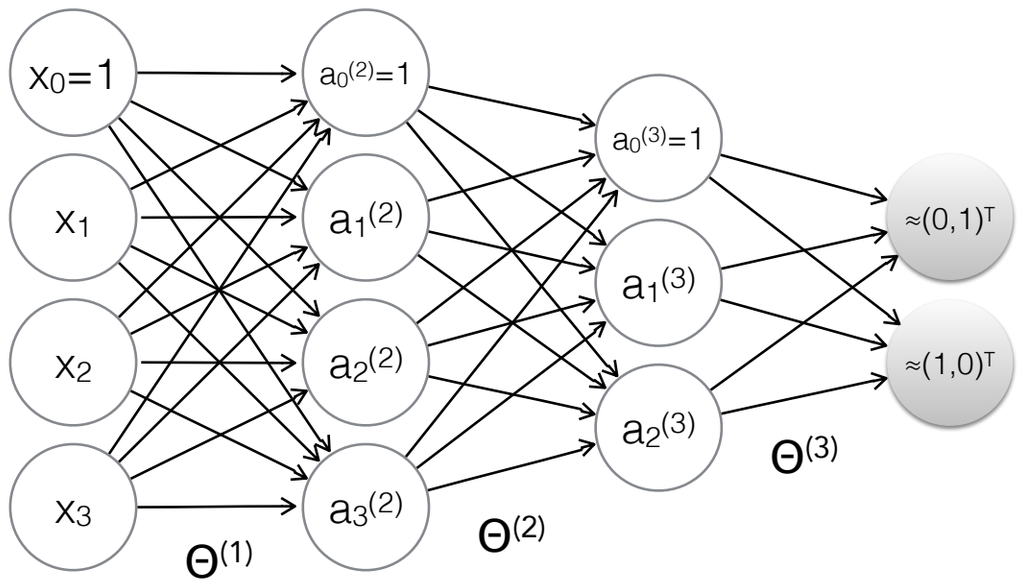
So this is the intuition of why neural networks might be able to learn complex non-linear models.



An of course we can define other network architectures e.g. now Layer 2 is computing some complex features and Layer 3 is taking these complex features and computing even more complex features so that by the time we get to Layer 4 we end up with a very sophisticated non-linear model.



So we can now extend this idea to multi-class logistic classification too where for example in the problem of object recognition we can have more than one category which we are trying to distinguish between. Handwriting detection is also a multi-class classification problem.



We can generalise our architecture so that our output layer now has multiple nodes and we wish for each one to correspond to the appropriate output vector. In this case we have two classes. And the predictors are now vectors of probabilities.

So now given the weights parameters it is relatively easy to make predictions on new data via a feed forward mechanism described before. So now the issue is how do we choose these weight parameters?

One thing we could do is to perturb the weight matrices randomly and use them if it leads to smaller mean square error. This is not that efficient actually. Another approach might be to turn on and off some activation nodes which is a bit better but still not computationally optimal.

A better approach is called Back-Propogation which back propagates the errors starting from the Output layers back up the network and is analogous to running something like gradient descent on a cost function of the activation functions. Typically we use a ridge-regression type cost function with the cost parameter λ and try to minimise it for some training data. This process is called training and I will not talk more about back-propogation except to say that we need to initialise our weight matrix to some random value initially rather than having them all as zero.

Training a Neural Network

- Pick an architecture
- Randomly initialise the weights
- Implement the forward propagation to get the $y_\Theta(X)$ for any X input
- Implement code to compute a cost function like residual sum of squares plus L1 penalty
- Implement back propagation to minimise this cost function for the best Θ

So our recipe for training a neural network turn out to be:

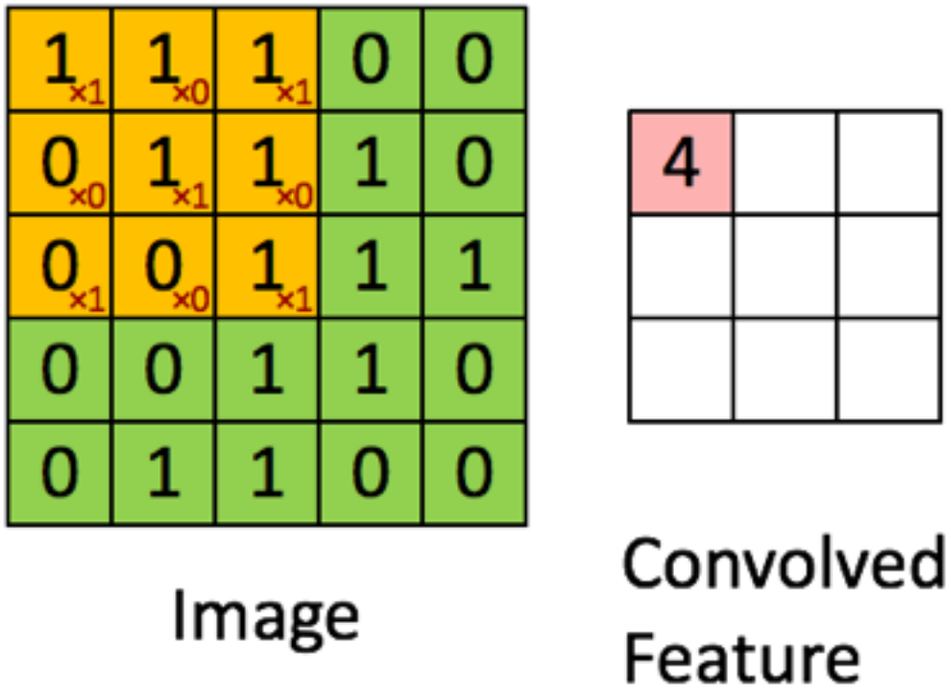
We will prob. not find a global optimum but gradient descent will do a good job in minimising this cost function.

Feature Extraction

- For images a fully connected network for even a 32x32 image is computationally intensive
- One solution: each hidden unit connects to a small subset of input units
- Images also have the property of being “stationary” meaning patches of the image repeat

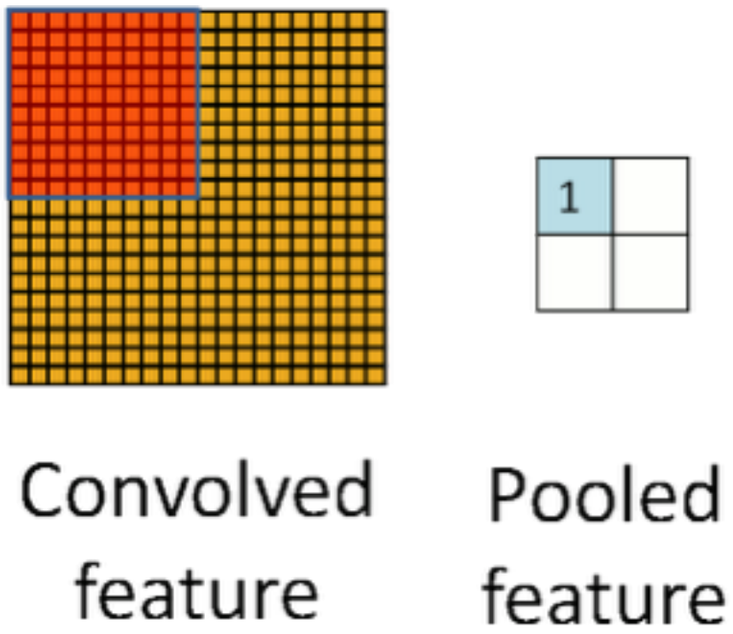
This suggests that the features that we learn at one patch of the image can also be applied to other parts of the image, and we can use the same features at all locations.

So if we for example learn features over small, say 4x4, patches sampled randomly from the larger image, we can then apply this learned 4x4 feature detector anywhere in the image. Specifically, we can take the learned 4x4 features and “convolve” them with the larger image, thus obtaining a different feature activation value at each location in the image.



What is convolution? It is the mathematical operation of taking a kernel function which is the size of the patch, and applying it to the image to get a number or representation. Doing this for patches we end up with a layer of convolved features.

After obtaining features we would next like to use them for classification. In theory, one could use all the extracted features with a classifier, but again this is too computationally expensive. Suppose for a 32x32 image we have learned 200 features input size 4x4. Each convolution results in a vector of size (32-4+1)*(32-4+1)=841 and these 200 features give us a vector of size 1.6 million features per example which might also lead to overfitting.



To fix this, recall that decided to obtain convolved features because images have the “stationarity” property, which implies that features that are useful in one region are also likely to be useful for other regions.

Thus, to describe a large image, one natural approach is to aggregate statistics of these features at various locations. For example, one could compute the mean (or max) value of a particular feature over a region of the image. These summary statistics are much lower in dimension (compared to using all of the extracted features) and can also improve results (less over-fitting).

These aggregate operation is called “pooling” or “mean pooling” or “max pooling” depending on the operation.

One can choose the pooling regions to be contiguous areas in the image and only pools features generated from the same (replicated) hidden units. These units will be translation invariant. This means that the same (pooled) feature will be active even when the image undergoes (small) translations. Translation-invariant features are often desirable; in many tasks like object recognition or audio recognition.

Convolution Neural Networks

- One or more convolution layers
- followed by one or more fully connected layers
- Use pooling which results in translation invariant features
- Resulting in easy to train networks with many fewer parameters

A CNN is a neural network comprised of one or more convolutional layers (often with a subsampling step) and then followed by one or more fully connected layers. The architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features.

Another benefit of CNNs is that they are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units.

CNNs in Python

- cuda-convnet (CUDA based)
- PyBrain
- Caffe
- DIY with Theano/Pylearn2 backend for GPUs

In python there are the number of CNN libraries the popular being. Today, we will look at cuda-convnet by Alex Krizhevsky (Google).

cuda-convnet

- Fast C++/CUDA implementation of convolutional neural networks
- Model arbitrary layer connectivity and network depth
- Needs Fermi-generation GPU (GTX 4xx, 5xx)
- https://github.com/dnouri/cuda-convnet

The main advantage of cuda-convnet is that it provides efficient implementation of convolution in CUDA and the modular design makes it easy to add new layer types, neuron activation functions, or objectives if you should need them.

Cuda-convnet saves python pickled objects so computation is check pointed for restarting later.

The github repo is a mirror of the original code source https://code.google.com/p/cuda-convnet/

```
➜ git clone https://github.com/dnouri/cuda-convnet.git
...

➜ cd cuda-convnet
➜ cmake .
...
-- Generating done
-- Build files have been written to: /Users/kashif/cuda-convnet

➜ make -j5
...
Linking CXX shared library convnet.so
[100%] Built target convnet
```


One advantage of github repo is that we use CMake to build the Cuda bindings for python. If your setup has a recent CUDA GPU and development environments setup, then all you need to do is install CMake and make sure you have numpy, python-magic, matplotlib and an appropriate blas/lapack library like Atlas, MKL or Accelerate framework on OSX.

Checking out the code we simply need to run the cmake command to compile everything…

```
➜ ls -lsh  ~/Downloads/cifar-10-py-colmajor
total 363768
   32 -rw-r--r--@ 1 kashif  staff 12K Jun 29  2011 batches.meta
60624 -rw-r--r--@ 1 kashif  staff 30M Jun 29  2011 data_batch_1
60616 -rw-r--r--@ 1 kashif  staff 30M Jun 29  2011 data_batch_2
60624 -rw-r--r--@ 1 kashif  staff 30M Jun 29  2011 data_batch_3
60624 -rw-r--r--@ 1 kashif  staff 30M Jun 29  2011 data_batch_4
60624 -rw-r--r--@ 1 kashif  staff 30M Jun 29  2011 data_batch_5
60624 -rw-r--r--@ 1 kashif  staff 30M Jun 29  2011 data_batch_6
```


To begin with let us discuss the type of data we can train on. Cuda-convnet expects data to be stored as python pickled objects. Next the data must be broken down into batches.

The CIFAR-10 dataset from http://www.cs.toronto.edu/~kriz/cifar.html consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. The dataset is divided into five training batches and one test batch, each with 10000 images in the python pickled format.

```
➜ cat example-layers/layers-80sec.cfg
[data]
type=data
dataIdx=0

[labels]
type=data
dataIdx=1

...
```


To define the architecture of your neural net, you must write a layer definition file.

The first thing you might add to your layer definition file is a data layer. The type=data line indicates that this is a data layer. Our python data provider outputs a list of two elements: the CIFAR-10 images and the CIFAR-10 labels. The line dataIdx=0 indicates that the layer named data is mapped to the CIFAR-10 images, and similarly the line dataIdx=1 indicates that the layer named labels is mapped to the CIFAR-10 labels.

```
[conv1]
type=conv
inputs=data
channels=3
filters=32
padding=2
stride=1
filterSize=5
initW=0.0001
partialSum=4
sharedBiases=1

...
```


Next a convolution layers apply a small set of filters all over their input "images". They're specified like this. The bracketed conv1 is the name we are calling this layer of type conv. inputs=data says that this layer will take as input the layer named data of 3 channels and will apply 32 filters to the image. We add a padding of 2 pixel borders of zero and a stride of 1 to indicate that the distance between successive filter applications should be 1 pixel. The convolution filter is of size 5x5 with initial weights in this layer from a normal distribution with mean zero and standard deviation 0.00001.

The partialSum parameter affects the performance of the weight gradient computation and it is worth trying a few different values here.

```
[pool1]
type=pool
pool=max
inputs=conv1
start=0
sizeX=3
stride=2
outputsX=0
channels=32
neuron=relu

...
```


This type of layer performs local, per-channel pooling on its input. We use max pooling rather than avg. Again the input comes from the conv1 layer. And we use a rectified linear neuron here. The channels now are 32 since the pooling performed by this layer is per-channel and is equivalent to the number of input channels.

```
[fc64]
type=fc
outputs=64
inputs=pool3
initW=0.1
neuron=relu

[probs]
type=softmax
inputs=fc10
...
```


Here is an example of a fully connected layer getting its input from a pool3 layer. Here we are actually defining two layers -- fc64 is a fully-connected layer with 64 output values and probs is a softmax that takes fc64 as input and produces 64 values which can be interpreted as probabilities. This type of layer is useful for classification tasks.

```
[logprob]
type=cost.logreg
inputs=labels,probs
```


Finally the architecture configuration must define an objective function to optimise. Here we are defining a (multinomial) logistic regression objective. It's specified like this. The cost.logreg objective takes two inputs: true labels and predicted probabilities. We defined the labels layer early on, and the probs layer in the previous slide.

```
➜ cat example-layers/layer-params-80sec.cfg
[conv1]
epsW=0.001
epsB=0.002
momW=0.9
momB=0.9
wc=0.004

[conv2]
epsW=0.001
epsB=0.002
momW=0.9
momB=0.9
wc=0.004

...

[logprob]
coeff=1
```


We also need a parameter file for each of the layers. In this file you can specify learning parameters that you may want to change during the course of training (learning rates, etc.). epsW is the weight learning rate, epsB is the bias training weight, momX are the momentums and wc is the L2 weight decay applied to the weights not the biases.

Cost layers, such as the logprob take one parameter a scalar coefficient of the objective function. This provides an easy way to tweak the "global" learning rate of the network.

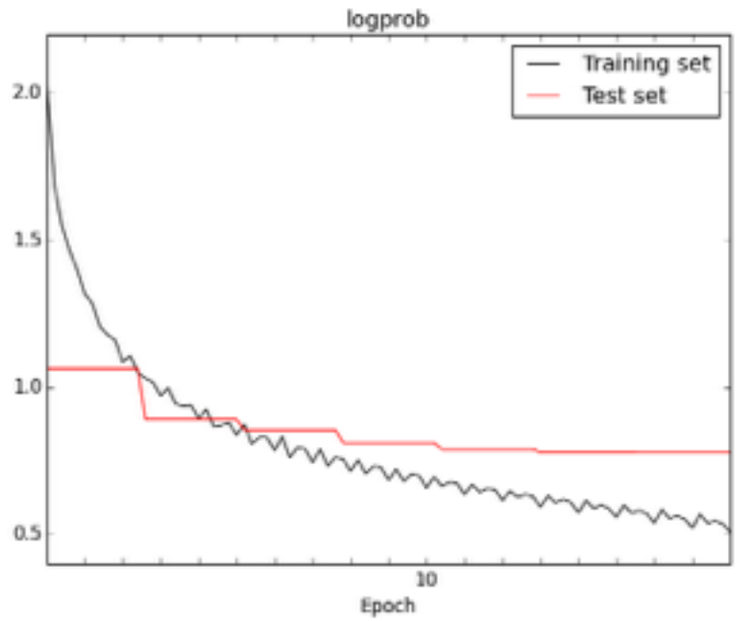
```
➜ python convnet.py
  --data-path=/Users/kashif/Downloads/cifar-10-py-colmajor
  --save-path=/tmp --test-range=6 --train-range=1-5
  --layer-def=example-layers/layers-80sec.cfg
  --layer-params=example-layers/layer-params-80sec.cfg
  --data-provider=cifar --test-freq=13
...
=========================
Running on CUDA device(s) -2
Current time: Thu Jul 10 21:31:24 2014
Saving checkpoints to /tmp/ConvNet__2014-07-10_21.31.24
=========================
1.1... logprob:  2.043771, 0.752000 (8.314 sec)
1.2... logprob:  1.689572, 0.610400 (7.623 sec)
1.3... logprob:  1.550371, 0.553200 (7.642 sec)
1.4... logprob:  1.464099, 0.516600 (7.706 sec)
1.5... logprob:  1.401730, 0.494800 (7.735 sec)
2.1... logprob:  1.317366, 0.466000 (8.932 sec)
2.2... logprob:  1.286320, 0.451600 (10.168 sec)
...
3.3... logprob:  1.049243, 0.369500
======================Test output======================
logprob:  1.065345, 0.367400
```


Once we have a layer definition file, and found some data to train on, we can actually do some training. This will cause the net to train on data located in the data-path, and save checkpoints to save-path. It will use batches 1-5 for training and batch 6 for testing. For every 13 training batches it processes, it will compute the test error once. The testing frequency is also the checkpoint saving frequency.

The output tells you the epoch and batch numbers as well as the values of all of the objectives that you're optimising. The cost.logreg objective happens to return two values -- the average negative log probability of the data under the model (that's the first number) as well as the classification error rate (the second number). You can see that the net here is at 36.7% classification error on the test set. The training will continue till 50000 epochs by default.

We can also provide a different data-provider that trains on slight image translations which improves classification on most image classifiers.

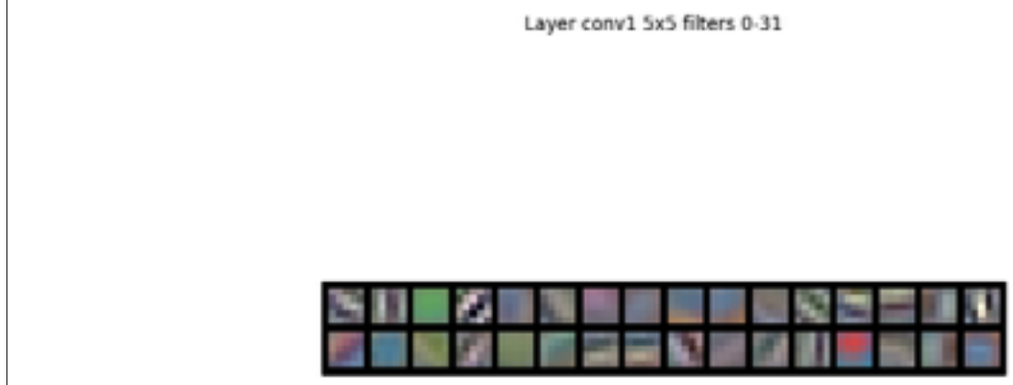
```
➜ python shownet.py
  -f /tmp/ConvNet__2014-07-10_21.31.24
  --show-cost=logprob
```



We can see inside the net too via the shownet.py script. It plots training/test error over time, display the filters that the neural net learned, and showes the predictions (and errors) made by the net.

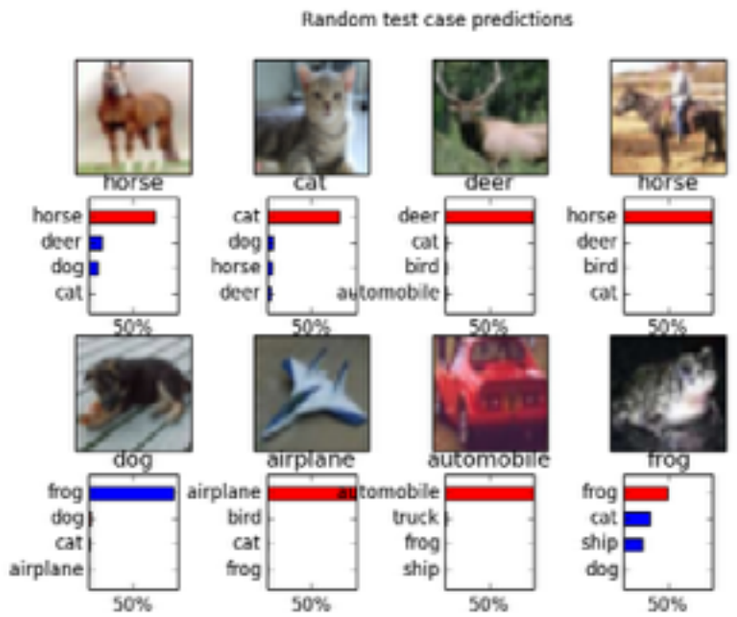
To plot the evolution of the value of some cost function over time.

```
➜ python shownet.py
  -f /tmp/ConvNet__2014-07-10_21.31.24
  --show-filters=conv1
```



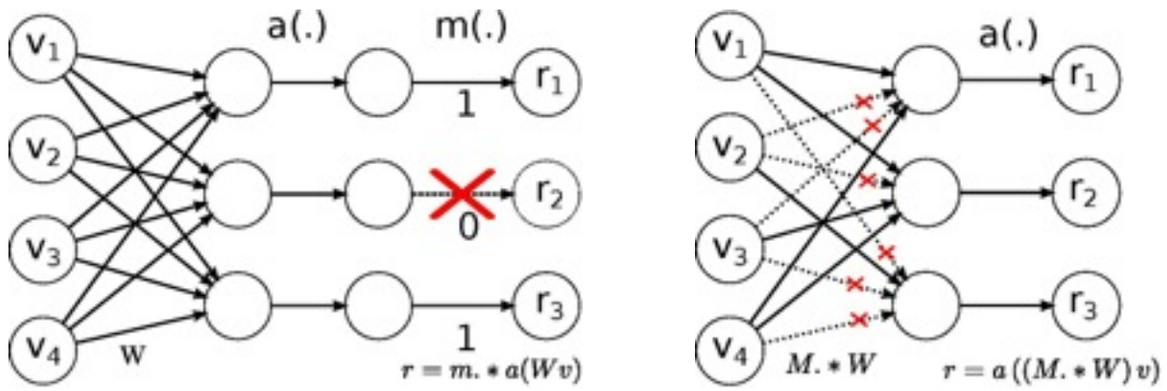
To view the filters that the net learned, call the script like this. Note that you're not likely to get such pretty filters in higher layers. So this visualisation is mostly useful only for looking at data-connected layers. Also it has interpreted the 3 channels in the conv1 layer as RGB.

```
➜ python shownet.py
  -f /tmp/ConvNet__2014-07-10_21.31.24
  --show-preds=probs
```



To see the predictions that the net makes on test data, run the script like this. This shows 8 random images from the test set, their true labels, and the 4 labels considered most probable by the model. The true label's probability is shown in red, if it is among the top 4. You can see that in this case the model only got one of the images wrong.

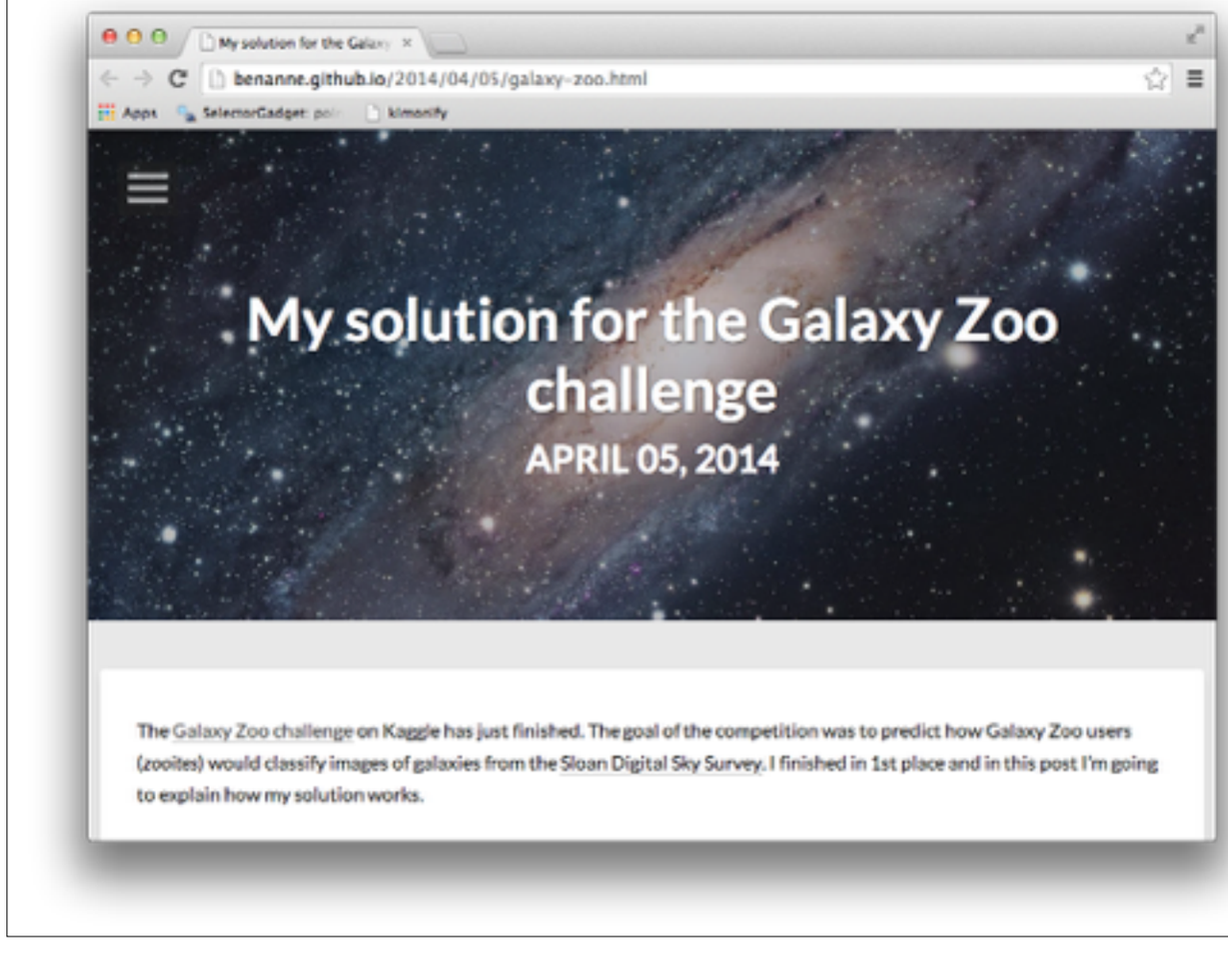
Regularisation



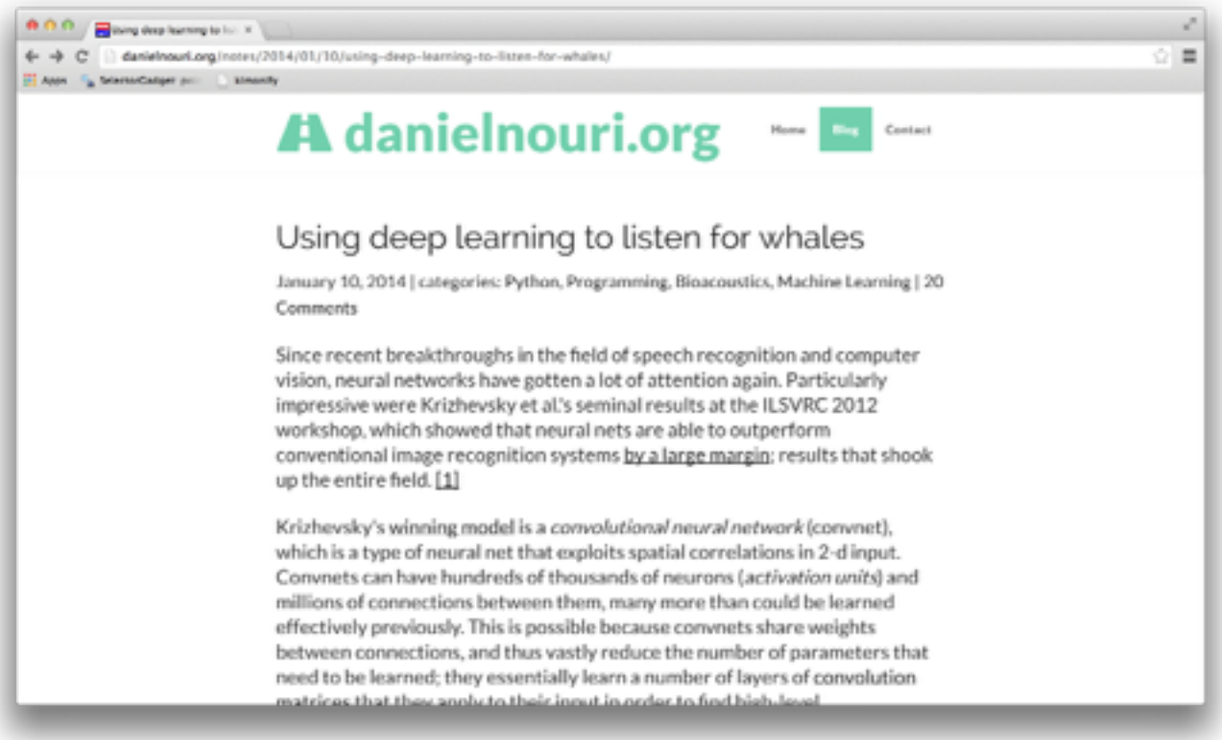
Another way to regularise neural networks is to introduce Dropout introduced by Hinton. When training with Dropout, a randomly selected subset of activations are set to zero within each layer (left image). The reason for this is outside the scope here, but it prevents complex co-adaptations in which a feature detector is only
helpful in the context of several other specific feature detectors. By doing dropout , each
neuron learns to detect a feature that is generally helpful for producing the
correct answer given the combinatorially large variety of internal contexts in
which it must operate.

More recently, DropConnect (right image) instead sets a randomly selected subset of weights within the network to zero. Each unit thus receives input from a random subset of units in the previous layer.

cuda-convnet implements DropOut but not DropConnect.



And a number of Kaggle competitions have been won by using convolutional neural networks. This one to classify galaxies needed 4 convolutional layers and 3 fully connected layers.



Also for speech recognition convolutional neural network work quite well.

Issues

- No easy solution
- Lots of tweaking
- No standard framework for doing large scale neural network learning like Hadoop for Map-Reduce or Spark for Streaming computation

But cuda-convnet2 should be released soon which might address these issues.

Lab

- Setting up AWS