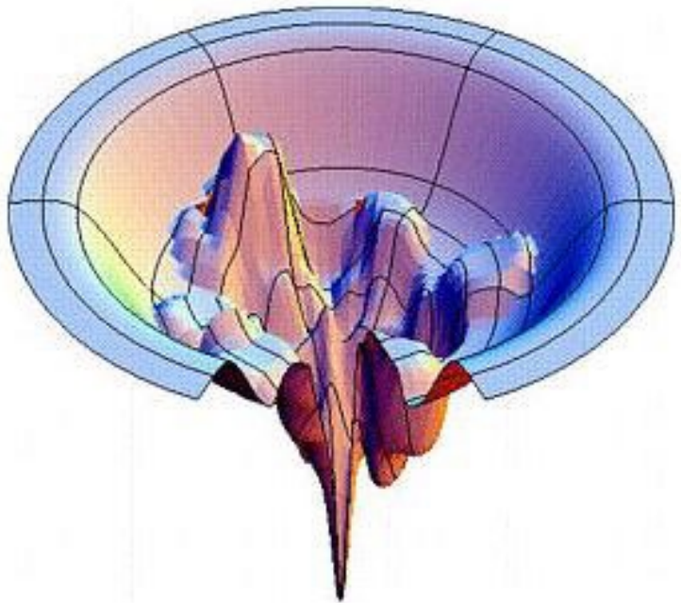
An Introduction to Deep Learning

Pratik Chaudhari

Nov 17, 2016



This material is at

https://github.com/pratikac/cs268-dl

Based on

CS231n at Stanford, http://cs231n.github.io

Nando de Freitas' class at Oxford, https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning

Introductory book: http://www.deeplearningbook.org

Some problems you can solve with it…

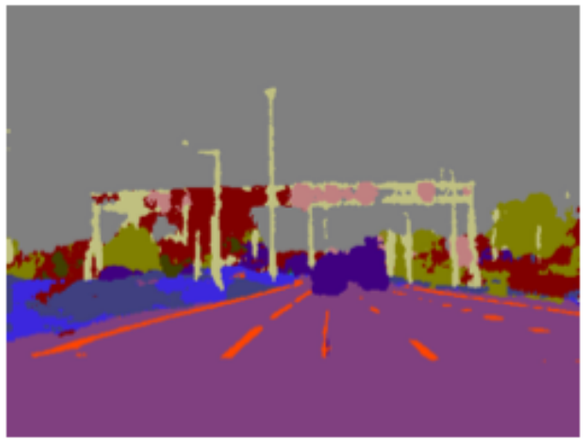
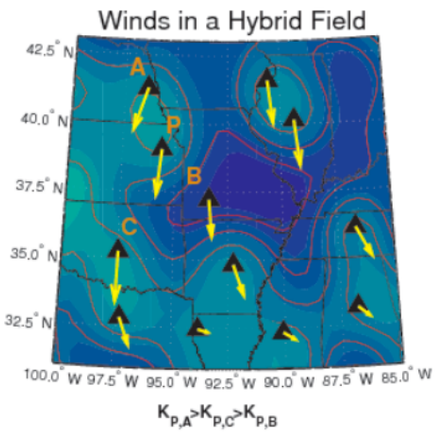


Image segmentation

Sound of silence

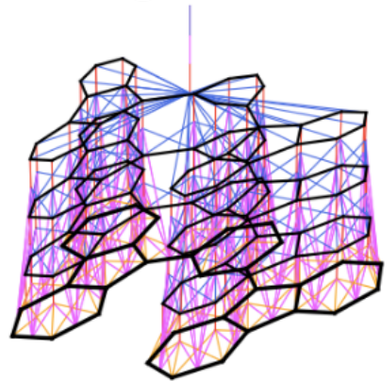
Autonomous driving



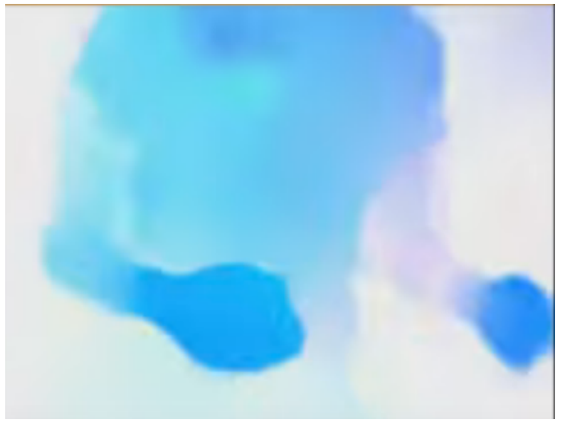
Weather

Image coloration

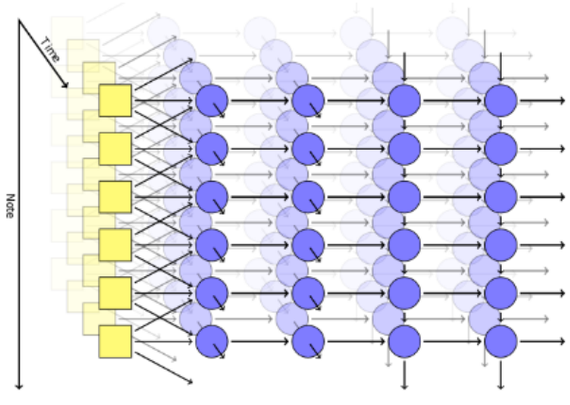
2D-3D videos



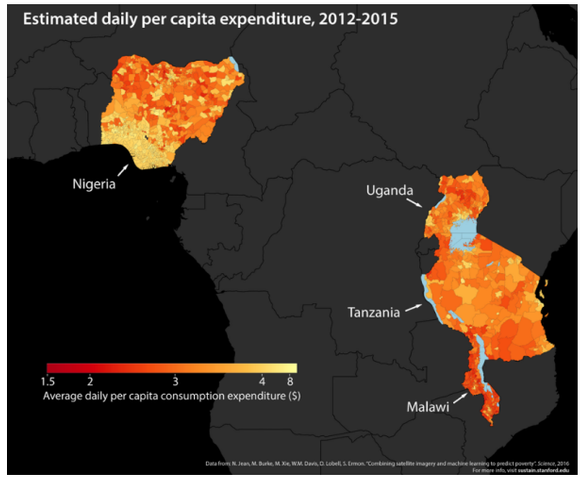
Neural fingerprint



FlowNet



Generate music



Per-capita income

Let's start with linear regression

$$X \in \mathbb{R}^{n \times N}$$
$$w \in \mathbb{R}^{n}$$
$$Y \in \mathbb{R}^{N}$$

- Linear model

$$\widehat{Y} = w^\top X$$

prediction ← $\widehat{Y}$; weights ← w; input ← X

- Quadratic loss

$$w^* = \arg\min \frac{1}{2\,N} \|Y - \widehat{Y}(w)\|_2^2$$

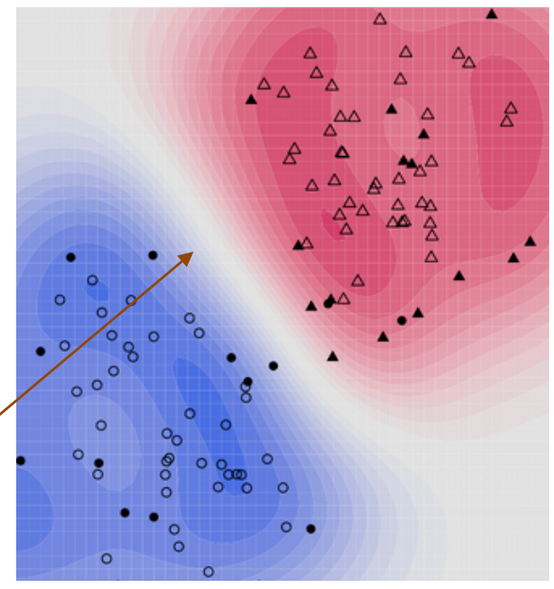
$\widehat{Y}$ ← quadratic loss

- Closed-form optimum

$$w^* = \left(X^\top X\right)^{-1} X^\top Y$$

- Similarly, for binary-classification one has SVMs

$$\widehat{Y} = \text{sigmoid}\left(w^\top X\right)$$



linear decision boundary

- Use non-linear kernels for complex features

Deep models

- Linear deep model

$$W_k \in \mathbb{R}^{n_{k+1} \times n_k}$$
$$X \in \mathbb{R}^{n_p}$$
$$Y \in \mathbb{R}$$

$$\widehat{Y} = w^\top \; W_1 \; W_2 \; \ldots W_p \; X$$

"last layer" like SVM — weights of each layer

  - Known by many different names, viz., dictionary learning, topic models…

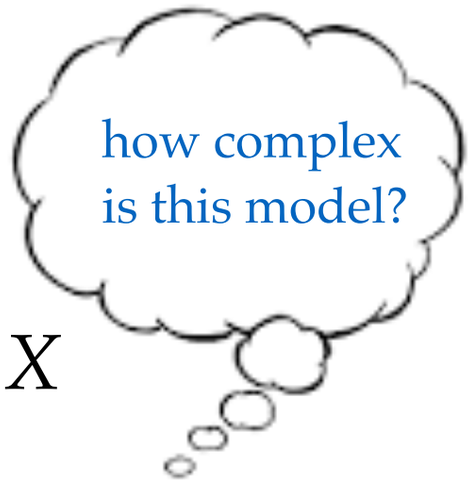
- Does this add any representational power? How complex is this model?
  - No, can simply rewrite it as…

$$A \in \mathbb{R}^{1 \times n_p} \qquad \widehat{Y} = A \; X$$

- Non-linearities make it much more interesting

$$\widehat{Y} = \sigma \left(w^\top \; \sigma \left(W_1 \; \sigma \; \left(W_2 \; \ldots W_p \; \right) \right) \right) X$$

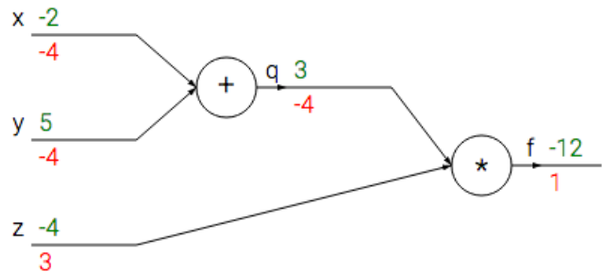
"last layer" *is* SVM — your favorite non-linear function



Back-propagation

Consider …

$$f(x, y, z) = z\ (x + y)$$



a circuit to compute f

Loss of a neural network

$$\text{loss} = \frac{1}{2}\,(y - w_1\ w_2\ x)^2$$

```
# set some inputs
x = -2; y = 5; z = -4

# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12

# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
dfdz = q # df/dz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
# now backprop through q = x + y
dfdx = 1.0 * dfdq # dq/dx = 1. And the multiplication here is the chain rule!
dfdy = 1.0 * dfdq # dq/dy = 1
```

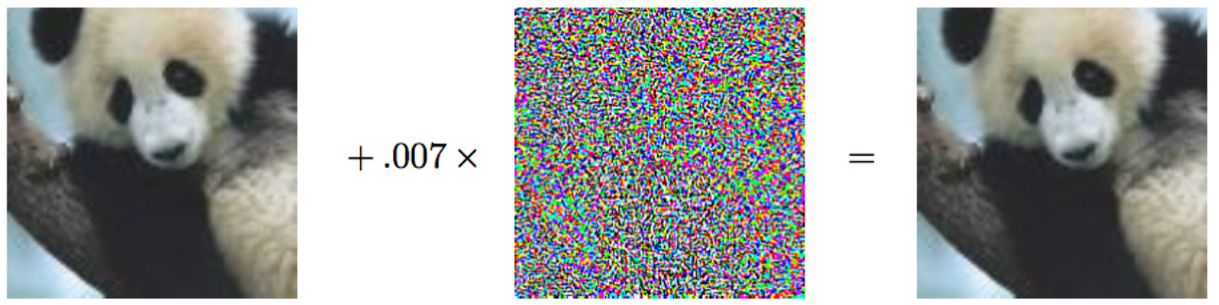
Want to compute how loss depends on weights

$$\frac{\partial\ \text{loss}}{\partial\ w_1} \quad \frac{\partial\ \text{loss}}{\partial\ w_2} \quad \frac{\partial\ \text{loss}}{\partial\ x}$$

gradient wrt weights

by product of back-prop

this was used in the "fooling deep network experiments"



$\boldsymbol{x}$
"panda"
57.7% confidence

$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence

$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Back-propagation works across "any" differentiable function

Convolutional layer

- **Instead of having a large, dense weight matrix, use a special, structured one**

- **Consider RGB images, arranged as**

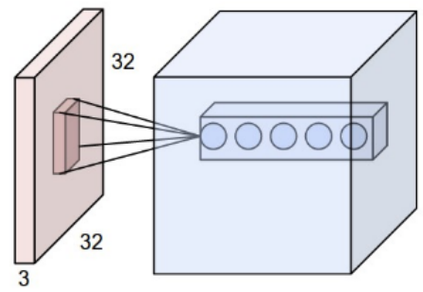
batch $\times$ channels $\times$ width $\times$ height

4D input tensor

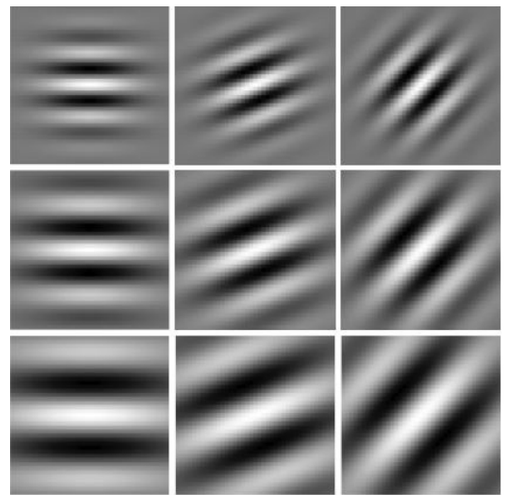
$$\begin{bmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & \ddots & & \vdots \\ a_2 & a_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{-1} & a_{-2} \\ \vdots & & \ddots & a_1 & a_0 & a_{-1} \\ a_{n-1} & \cdots & \cdots & a_2 & a_1 & a_0 \end{bmatrix}$$

Toeplitz / doubly-circulant matrices

- **A convolutional layer transforms this 4D tensor into another 4D tensor**

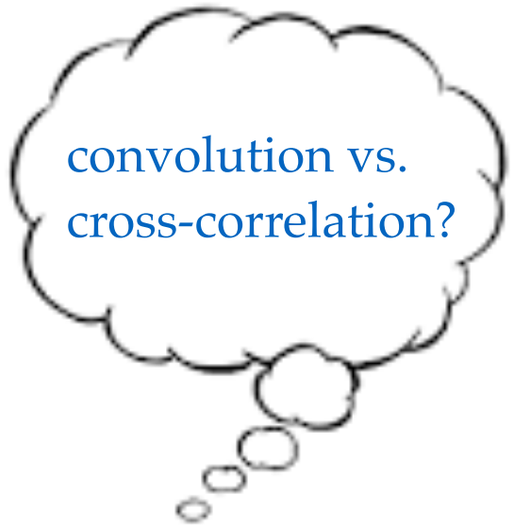


- **Why do we use convolutions?**
  - Resizing an image into a vector destroys spatial information
  - Convolutional kernels have a small "receptive" field, i.e., sparse interactions, lots of shared parameters
  - Equivariant representations
  - But mostly, proof is via results…



Convolutional layer: strides, padding and such…

http://cs231n.github.io/assets/conv-demo/index.html

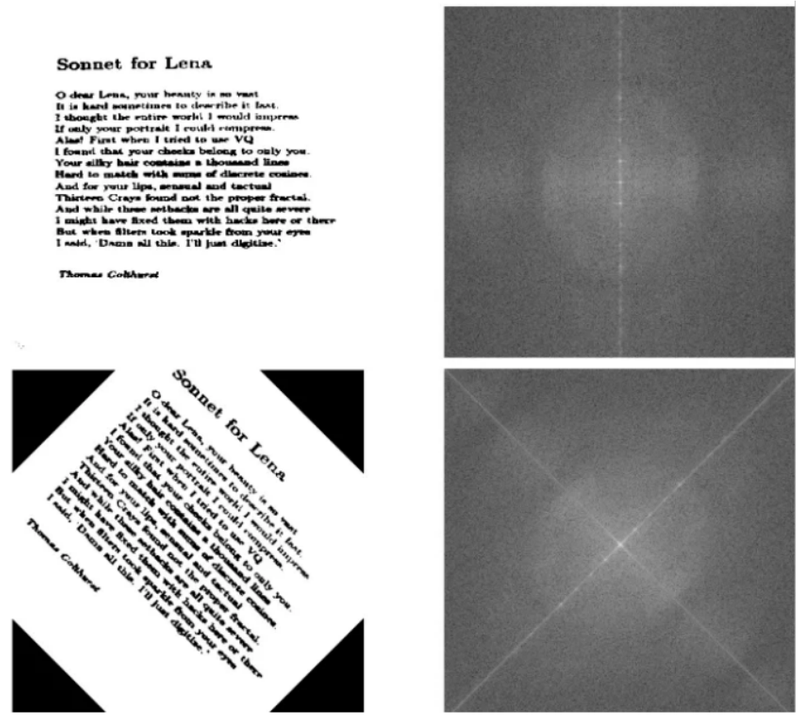


Types of convolutions

- dilated convolutions ← increase receptive field with a wide kernel
- 1x1 convolutions ← like a fully-connected layer, but fewer parameters

How are convolutions implemented?

- simple double for-loop for small kernels
- im2col: resize the receptive field as a vector and write the convolutional layer as “big” matrix multiplication
- Construct the Toeplitz matrix
- Fast Fourier Transform for larger kernels ~7x7+ ← convolution of two 1D signals is multiplication of their FFT, same holds for matrix kernels



Images by Fisher & Koryllos (1998). Source

Non-linearities

Sigmoids

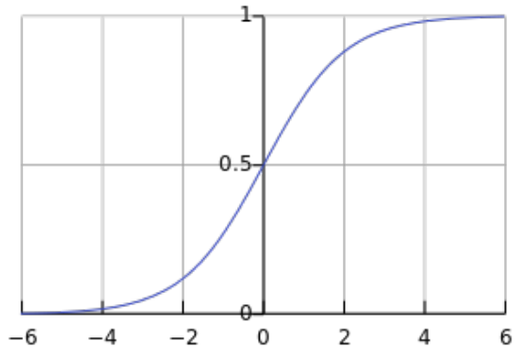
- Motivated from binary classification, want

$$p := \mathrm{P}(Y = 1 \mid X) \approx w^\top X + w_0$$

unbounded

- Use log-odds and set

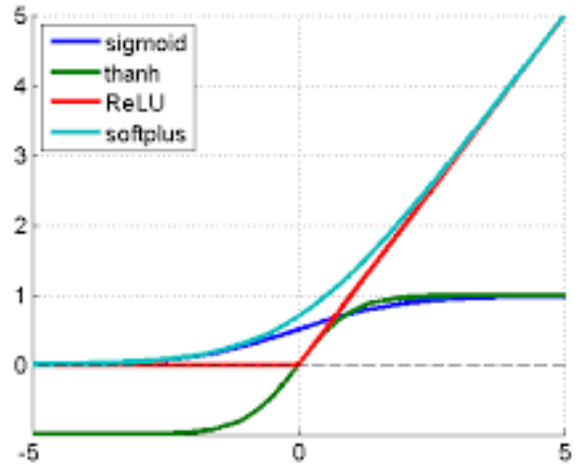
$$\log \frac{p}{1-p} = w^\top X + w_0 \quad \longrightarrow \quad p = \sigma(w^\top X + w_0)$$



$$\sigma(x) = \left(1 + e^{-x}\right)^{-1}$$

Rectified Linear Units (ReLUs)

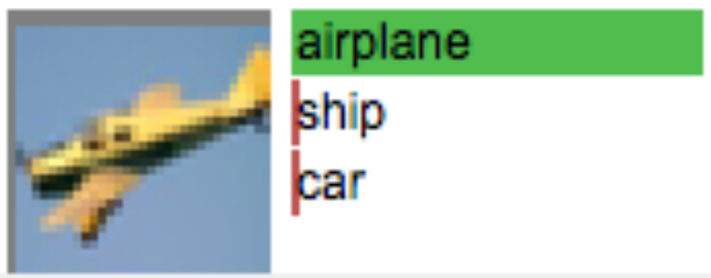
- Gradient of sigmoid is zero away from the origin
- Make it linear on the positive side
- Leaky-ReLUs, Exponential units etc.
- Kind of work equally well…



$$\sigma(x) = \mathbf{1}_{\{x \geq 0\}}$$

Soft-max and Cross-entropy loss

- Interpret the output as a probability vector
- Loss: force the output to give high weight to the correct class



$$\text{soft-max}(k) = \frac{e^{-k}}{\sum_{k'} e^{-k'}}$$

Regularization

Max-pooling, mean-pooling

- Input is say, 224x224 whereas output has 1000 classes
- Convolutions do not reduce size (except padding)

$$y_{ij} = \max_{i',j' \in \Omega_{ij}} x_{ij}$$

Ω_{ij} ← receptive field of a pixel

- Operationally, max operation reinforces strong features, removes weak ones

Batch-normalization

- Make the weights invariant to scale of input
- Subtract batch-wise mean of input,
  divide by standard deviation

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

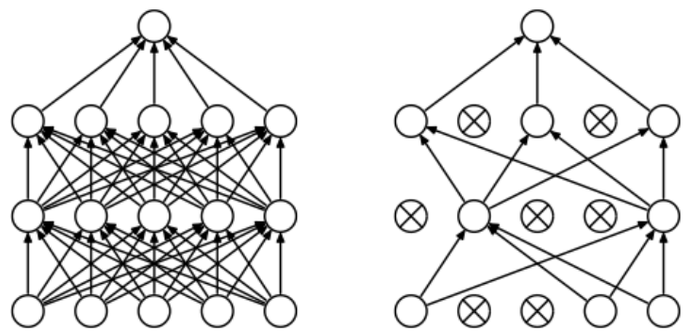
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Dropout

- Deep networks are over-parametrized
- An ensemble of sparse networks
- Efficiently share weights among many networks



How do I design a network for my problem?

Lego pieces: Mix-n-match "blocks" from popular networks

Classification

- VGG, AlexNet, Inception, ResNet

Object detection

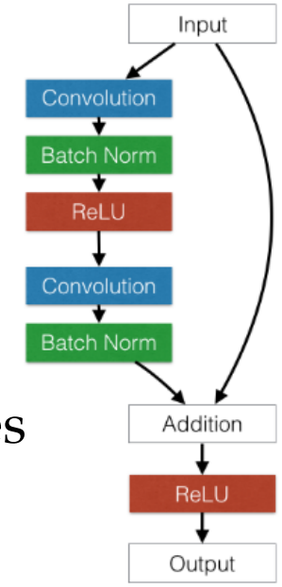
- No soft-max, large fully-connected layer regresses upon ground-truth locations of objects

Optical flow, mixed-modalities

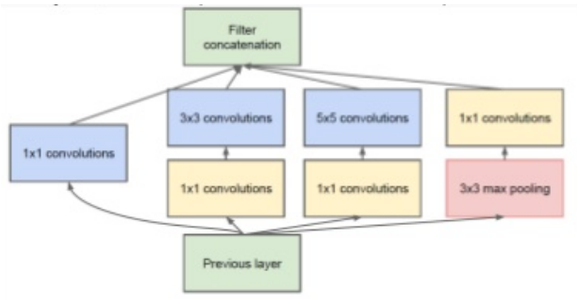
- Two concurrent channels
- Parallel branches join at the top

De-convolutional Layer

- A way to "increase" input size
- Remembers the pixels killed during max-pooling and "un-pools" them.

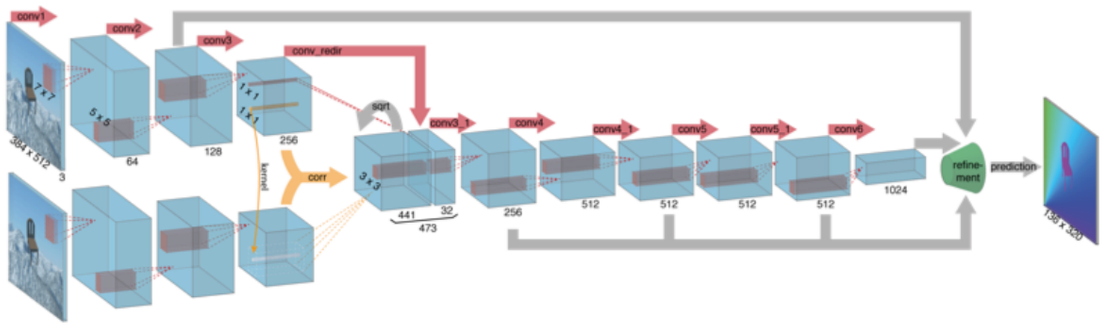


Residual network block

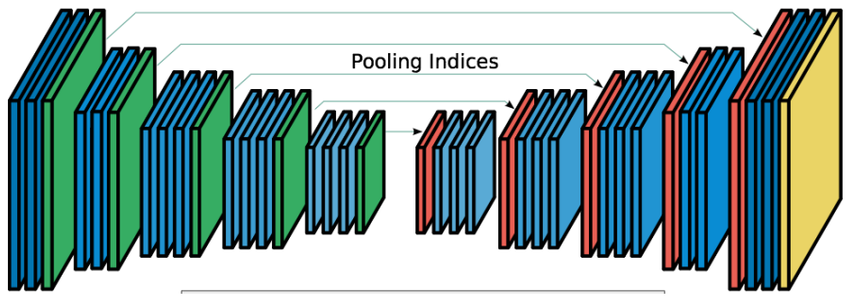


Inception block

could be a depth map →



FlowNet



Segmentation Network

Stochastic gradient descent

Loss function: $\text{minimize } \mathbf{1}_{\{y \neq \widehat{y}\}}$ ← number of mistakes on the entire dataset

Prediction: $\widehat{y} = \text{soft-max } f(w)$ ← a deep network

Back-propagation gives gradient of the loss

$$w_{t+1} \leftarrow w_t - \eta\, g_t$$ ← gradient over "the entire" dataset

vanilla SGD: computes gradient only on a mini-batch

Numerous bells and whistles

- Momentum: bias towards gradient of last batch ← rms-prop

$$g_t \leftarrow (1-\alpha)\, g_{t-1} + \alpha g_t$$

- Adaptive step-size

$$\eta g_t \leftarrow \eta' \frac{\mathrm{E}(g_t)}{\mathrm{stddev}(g_t)}$$ ← AdaGrad, AdaDelta, Adam, Eve …

Some examples

MNIST, the hello-world of deep learning

- 10 digits, 70,000 grayscale images

http://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html

CIFAR-10

- 10 classes: airplane, automobile, bird, cat, deer .., truck

https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

Many classification datasets

ground-truth has bounding boxes for each object

- ImageNet, Tiny-ImageNet, Pascal VOC and MS COCO for segmentation
- Relationships among objects: Visual Gnome
- 3D object datasets with Kinect, LIDAR data
- Driving data: KITTI, comma.ai, Torcs, GTA

How do I start running deep networks?

Pick a framework and learn it inside out..

Tensorflow

- Very popular, lots of functionality
- Clean code for non-standard operations, unfortunately, not-so-great code for standard networks and operations

Torch

- Very fast implementations, clean code
- Good for CNNs, not so great for RNNs
- Lua has a few idiosyncrasies

- Many others…
  matconvnet, theano, libdarknet, mxnet
- **One of the fastest ways to learn deep networks is to write your own library!**

Autograd

- Back-prop any Python function!
- Great tool for small experiments

Walk through some Torch code

How do I start running deep networks?

All set if you have a ~3 GB GRAM GPU

- having a large GPU rig also works…

CPU works okay for small datasets like MNIST

Amazon's Elastic Compute (EC2)

- Quite reasonable at $0.22/hour for K80 if you use spot instances
- Deep learning AMIs already have CUDA, caffe, tensorflow, torch etc. pre-installed

Google Cloud

- $300 credit in the beginning, but no GPU instances (yet)

Questions?