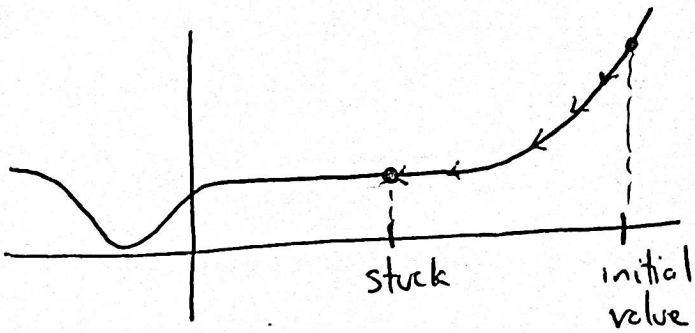
Vanishing Gradient Problem

$(W, B) \mapsto J_{W,B}(X, Y)$

In practice, the function for a neural network $\mathcal{N}$ is non convex, so there may be flat regions one may "get stuck" in during (stochastic) gradient descent



Factors that affect "flatness" of cost function

1. Choice of loss function
   - SE: lot of flat regions
   - CE: less
   - BCE: least
2. Choice of activations
   - the ones we discussed so far are good choices

Note: Using several sigmoids in hidden layers can cause flatness.

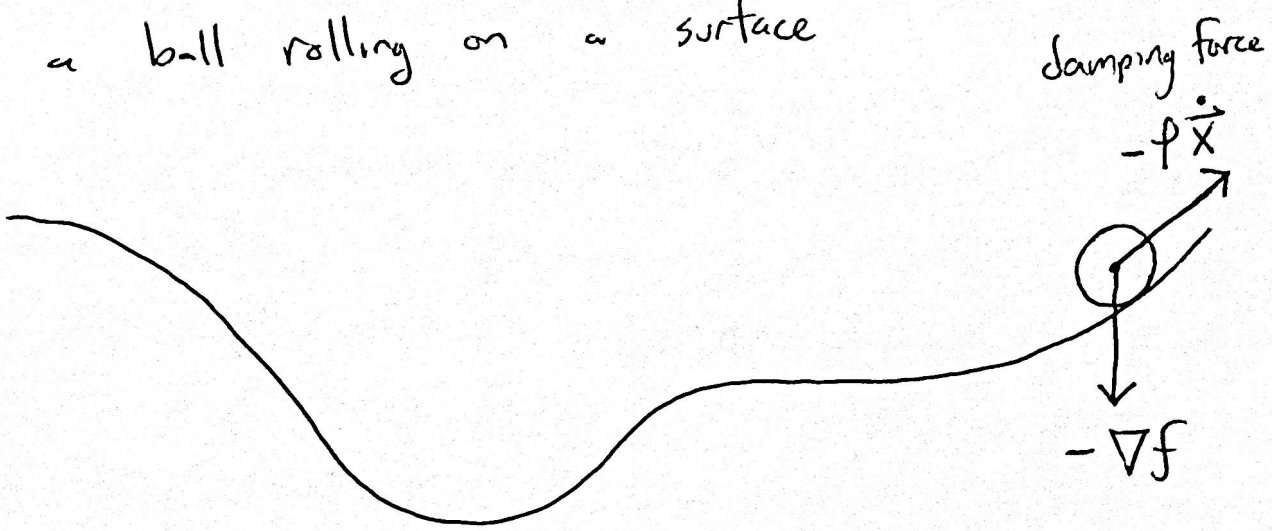
Suggestion: Use ReLU in hidden layers

3. Momentum
4. Weight Initialization

Momentum Method

Motivation (from physics)

Consider a ball rolling on a surface



f = potential function $\quad f(\vec{x}) = mgx_3$

$$\sum \vec{F} = m\vec{a} \implies \ddot{\vec{x}} = -\rho\dot{\vec{x}} - \nabla f(\vec{x})$$

To solve: Express as a system of 1st order ODEs

$$\dot{\vec{V}} = -\rho\vec{V} - \nabla f(\vec{x})$$
$$\dot{\vec{x}} = \vec{V}$$

Write this as a finite difference system

(Pick Δt small enough so $\rho\Delta t < 1$)

$$\vec{V}_{n+1} - \vec{V}_n = -\rho\vec{V}_n\Delta t - \nabla f(\vec{x}_n)\Delta t$$
$$\vec{x}_{n+1} - \vec{x}_n = \vec{V}_{n+1}\Delta t$$

Let $\varepsilon = \Delta t \quad \mu = 1 - \beta\varepsilon \quad 0 \le \mu < 1$

$$\vec{V}_{n+1} = \mu \vec{V}_n - \varepsilon \nabla f(\vec{x}_n)$$

$$\vec{x}_{n+1} = \vec{x}_n + \varepsilon \vec{V}_{n+1}$$

Let $\vec{V}\varepsilon = \tilde{\vec{V}}$ (new scale for velocity)

$$\frac{\tilde{\vec{V}}_{n+1}}{\varepsilon} = \mu \frac{\tilde{\vec{V}}_n}{\varepsilon} - \varepsilon \nabla f(\vec{x}_n)$$

$$\vec{x}_{n+1} = \vec{x}_n + \tilde{\vec{V}}_{n+1}$$

Define $\alpha = \varepsilon^2$

$$\boxed{\begin{aligned} \tilde{\vec{V}}_{n+1} &= \mu \tilde{\vec{V}}_n - \alpha \nabla f(\vec{x}_n) \\ \vec{x}_{n+1} &= \vec{x}_n + \tilde{\vec{V}}_{n+1} \end{aligned}}$$

If $\mu = 0$ this is just gradient descent

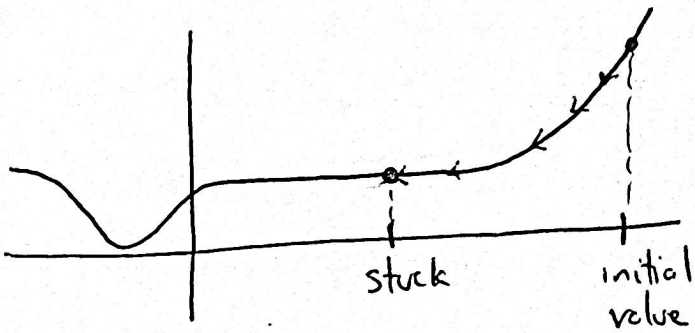
Weight Initialization

Neural net performance depends heavily on $(W^{(\ell)})$ weight initialization <u>especially</u> if D is large. Depends less on bias initialization $(b^{(\ell)})$ but this depends on architecture.

Vanishing Gradient Problem

$(W, B) \mapsto J_{W,B}(X, Y)$

In practice, the function for a neural network $\mathcal{N}$ is non convex, so there may be flat regions one may "get stuck" in during (stochastic) gradient descent



Factors that affect "flatness" of cost function

1. Choice of loss function
   - SE: lot of flat regions
   - CE: less
   - BCE: least
2. Choice of activations
   - the ones we discussed so far are good choices

Note: Using several sigmoids in hidden layers can cause flatness.

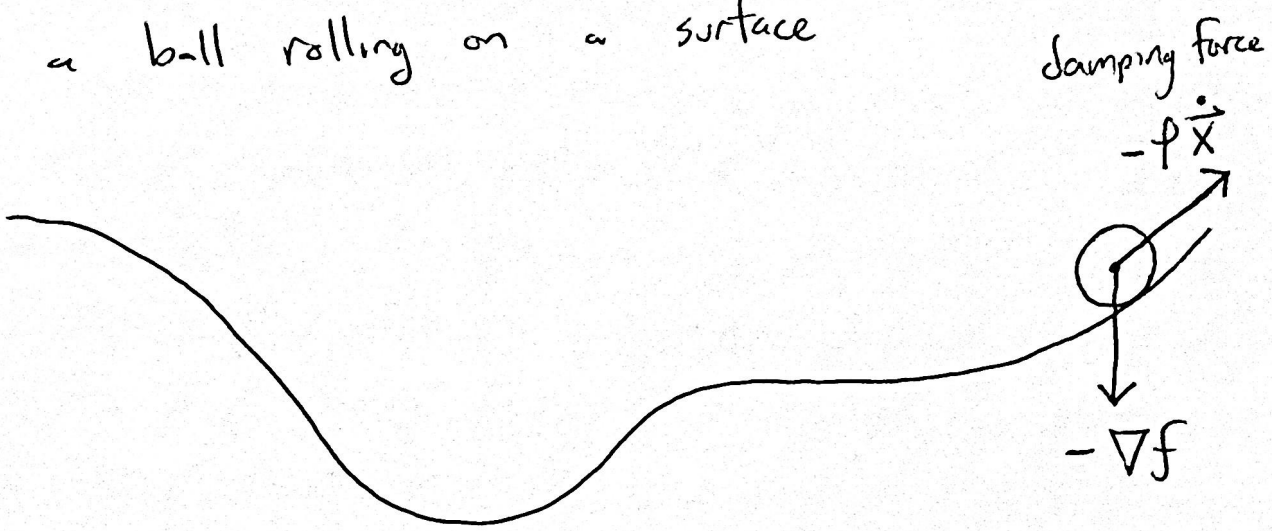
Suggestion: Use ReLU in hidden layers

3. Momentum
4. Weight Initialization

Momentum Method

Motivation (from physics)

Consider a ball rolling on a surface



f = potential function $\quad f(\vec{x}) = mgx_3$

$$\sum \vec{F} = m\vec{a} \implies \ddot{\vec{x}} = -\rho\dot{\vec{x}} - \nabla f(\vec{x})$$

To solve: Express as a system of 1st order ODEs

$$\dot{\vec{V}} = -\rho\vec{V} - \nabla f(\vec{x})$$
$$\dot{\vec{x}} = \vec{V}$$

Write this as a finite difference system

(Pick Δt small enough so $\rho\Delta t < 1$)

$$\vec{V}_{n+1} - \vec{V}_n = -\rho\vec{V}_n\Delta t - \nabla f(\vec{x}_n)\Delta t$$
$$\vec{x}_{n+1} - \vec{x}_n = \vec{V}_{n+1}\Delta t$$

Let $\varepsilon = \Delta t \quad \mu = 1 - \beta\varepsilon \quad 0 \le \mu < 1$

$$\vec{V}_{n+1} = \mu \vec{V}_n - \varepsilon \nabla f(\vec{x}_n)$$

$$\vec{x}_{n+1} = \vec{x}_n + \varepsilon \vec{V}_{n+1}$$

Let $\vec{V}\varepsilon = \tilde{\vec{V}}$ (new scale for velocity)

$$\frac{\tilde{\vec{V}}_{n+1}}{\varepsilon} = \mu \frac{\tilde{\vec{V}}_n}{\varepsilon} - \varepsilon \nabla f(\vec{x}_n)$$

$$\vec{x}_{n+1} = \vec{x}_n + \tilde{\vec{V}}_{n+1}$$

Define $\alpha = \varepsilon^2$

$$\boxed{\begin{aligned} \tilde{\vec{V}}_{n+1} &= \mu \tilde{\vec{V}}_n - \alpha \nabla f(\vec{x}_n) \\ \vec{x}_{n+1} &= \vec{x}_n + \tilde{\vec{V}}_{n+1} \end{aligned}}$$

If $\mu = 0$ this is just gradient descent

Weight Initialization

Neural net performance depends heavily on $(W^{(\ell)})$ weight initialization <u>especially</u> if D is large. Depends less on bias initialization $(b^{(\ell)})$ but this depends on architecture.

Claim: ① If the initialized values of $W^{(l)}$ are too small, the variance of $s^{(l)}$ decreases at each layer

② If initialized weights are too large, variance $s^{(l)}$ amplifies at each layer

A Solution (More of a suggestion)

Xavier Initialization: Based on the assumption that $\text{Var}(s^{(l)})$ should remain constant at each layer.

For $P(n_0, \ldots, n_D)$ entries of $W^{(l)}$ should be sampled from $\mathcal{N}(x; 0, \frac{2}{n_{l-1}+n_l})$.

Many other "suggestions" exist

- Read §8.4 of the deep learning book
- Read §6.3 of Deep Learning Architectures

Bias can usually be initialized to $\vec{0}$.

A Way to do SGD on neural net $\mathcal{N}$ (for $P(n_0, \dots, n_D)$)

Pick $\alpha > 0$, $\lambda > 0$, $\mu > 0$, max_iters, batch_size
(step size; reg. parameter; momentum)

For $0 \le l \le D-1$ initialize $W^{(l)}$ by sampling $\mathcal{N}(x; 0, \frac{2}{n_{l-1}+n_l})$

$b^{(0)} = \vec{0}$

initialize $V_w^{(l)} = np.zeros(W^{(l)}.shape)$

$V_b^{(l)} = np.zeros(b^{(l)}.shape)$

epochs = 0

while epochs $\le$ max_iters:

randomly pick $B \subseteq \{1, \dots, m\}$ with $|B|$ = batch_size

for $i \in B$:

compute $[(x_i^{(0)}, s_i^{(0)}), (x_i^{(1)}, s_i^{(1)}), \dots, (x_i^{(D-1)}, s_i^{(D-1)}), (x_i^{(D)})]$

compute $\delta_i^{(D-1)}, \dots, \delta_i^{(0)}$

⟵ compute $\frac{\partial J}{\partial W^{(l)}}, \frac{\partial J}{\partial b^{(l)}}$ (batch approximations

for $0 \le l \le D-1$:

$$V_w^{(l)} = \mu V_w^{(l)} - \alpha \frac{\partial J}{\partial W^{(l)}}$$

$$V_b^{(l)} = \mu V_b^{(l)} - \alpha \frac{\partial J}{\partial b^{(l)}}$$

$$W^{(l)} = W^{(l)} + V_w^{(l)}$$

$$b^{(l)} = b^{(l)} + V_b^{(l)}$$

append cost to list of costs

return W, B, costs

Convolutional Neural Networks

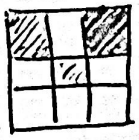
Idea: NN architecture should be chosen to complement the type of structure of input data if possible

E.g. ① $\vec{x}$ = statistics on houses

(no real structure so use fully connected architecture)

② $\vec{x}$ = flattened vector of pixel info on an image

E.g. image is 3×3



$\longrightarrow (2,0,0,0,1,0,3,0,0)$

than x_4 is "nearby" x_1 and x_7

Q: How can we incorporate this extra structure?

A: "Convolution" and "Pooling"

Def: A discrete 1-dim'l signal is a sequence

$$y = (y_i)_{i \in \mathbb{Z}}$$ of real numbers

and is L^1-finite if $\sum_k |y_k| < \infty$

compact support if $\exists\, N$ s.t.

$$|k| > N \implies y_k = 0$$

A 1-dim'l kernel (or filter) is a compact support signal of "weights"

$$w = (w_i)_{i \in \mathbb{Z}}$$

Note: Usually want w to be a probability distribution so $\sum_i w_i = 1$

The convolution of y and w is

$$z = y * w$$

(often called the "cross correlation" in signal processing)

where $z_j = \sum_{k \in \mathbb{Z}} y_{j+k} w_k$

(well defined because w has compact support)

E.g. (Moving average signal)

$$w_i = \begin{cases} \frac{1}{2} & i = 0, 1 \\ 0 & \text{else} \end{cases}$$

$$(y * w)_i = \sum_{k \in \mathbb{Z}} y_{i+k} w_k = \frac{y_i + y_{i+1}}{2}$$

Similarly $w_i = \begin{cases} \frac{1}{M} & i = 0, \ldots, M-1 \\ 0 & \text{else} \end{cases}$

$$(y * w)_i = \frac{y_i + y_{i+1} + \cdots + y_{i+M-1}}{M}$$

E.g. $y = (\underset{0}{1}, 2, 3, 4, 5, \underset{5}{6})$ $\quad w = (\underset{0}{\tfrac{1}{2}}, \underset{1}{\tfrac{1}{2}})$

$$y * w = (\underset{-1}{\tfrac{1}{2}}, \underset{0}{\tfrac{3}{2}}, \underset{1}{\tfrac{5}{2}}, \tfrac{7}{2}, \tfrac{9}{2}, \tfrac{11}{2}, \underset{5}{3})$$

When using this to process signals in a NN layer, we usually ignore the first and last component. We may also ignore some parts on the inside

i.e. may express this as

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{2} & & & & \\ & \frac{1}{2} & \frac{1}{2} & & & \\ & & \frac{1}{2} & \frac{1}{2} & & \\ & & & \frac{1}{2} & \frac{1}{2} & \\ & & & & \frac{1}{2} & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} \frac{3}{2} \\ \frac{5}{2} \\ \frac{7}{2} \\ \frac{9}{2} \\ \frac{11}{2} \end{pmatrix} \in \mathbb{R}^5$$

maybe we only want to keep $(\frac{3}{2}, \frac{7}{2}, \frac{11}{2})$

Could express this as

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{2} & & & & \\ & & \frac{1}{2} & \frac{1}{2} & & \\ & & & & \frac{1}{2} & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 3/2 \\ 7/2 \\ 11/2 \end{pmatrix} \in \mathbb{R}^3$$

As a neural network layer

A <u>convolutional layer</u> (in a neural network with 1D signals) with kernel $W = (w_1, \dots, w_K)$ and <u>stride r</u> is a layer with $F: \mathbb{R}^{n_l} \to \mathbb{R}^{n_{l+1}}$ given by

$$F(\vec{x}) = W^{(l)}\vec{x} + \vec{b}^{(l)}$$

where

$$W^{(l)} = \begin{pmatrix} w_1 \cdots w_K & & & \\ \longleftrightarrow & w_1 \cdots w_K & & \\ r & & w_1 \cdots w_K & \\ & & & \ddots \\ & & & w_1 \cdots w_K \end{pmatrix}$$

A <u>pooling layer</u> (with 1D inputs) is a layer which applies a map

$$P: \mathbb{R}^N \longrightarrow \mathbb{R}^K \qquad K \leq N$$

defined as follows:

Let Π be a partition of $\{1, \dots, N\}$

i.e. $\Pi = \{B_1, \dots, B_K\}$ s.t.

$B_i \subseteq \{1, \dots, N\} \quad \forall i$

$B_i \cap B_j = \emptyset$ when $i \neq j$

$$\bigcup_{i=1}^{k} B_i = \{1, \ldots, N\}$$

define $P(x_1, \ldots, x_n) = \left(\max_{i \in B_1} x_i, \ldots, \max_{i \in B_k} x_i \right)$

usually $B_1 = \{1, 2, \ldots, b_1\}$ $B_2 = \{b_1+1, b_1+2, \ldots, b_2\}, \ldots$

Note: Alternatively we could use min or avg

E.g. $P: \mathbb{R}^{10} \to \mathbb{R}^4$ $B = \{\{1,2,3\}, \{4,5\}, \{6,7\}, \{8,9,10\}\}$

$$P: \begin{bmatrix} 2 \\ 4 \\ 3 \\ 7 \\ 6 \\ 4 \\ 1 \\ 0 \\ 2 \\ 9 \end{bmatrix} \longmapsto \begin{bmatrix} 4 \\ 7 \\ 4 \\ 9 \end{bmatrix}$$

Two Dimensional Signals

A discrete 2-dim'l signal is $y = (y_{ij})_{i,j \in \mathbb{Z}}$ and is

L^1-finite if $\sum_{i,j \in \mathbb{Z}} |y_{ij}| < \infty$

finite support if $\exists N \in \mathbb{N}$ s.t. $|k| > N$ and $|r| > N$

$$\Longrightarrow y_{k,r} = 0$$

E.g. Grayscale images can be represented as 2-dim'l signals with finite support.

A 2-dim'l kernel is a 2-dim'l signal W of finite support

(Again we usually want this to be a probability distribution so $\sum_{ij} W_{ij} = 1$)

The convolution of y and W is

$$z = y * W \quad \text{where} \quad z_{ij} = \sum_{r,k \in \mathbb{Z}} y_{i+k,j+r} W_{k,r}$$

(often z is called a feature map)

E.g. Consider $W_{ij} = \begin{cases} \frac{1}{4} & (i,j) = (0,0), (1,0), (0,1), (1,1) \\ 0 & \text{else} \end{cases}$

let $z = y * W$. Then

$$z_{ij} = \frac{y_{ij} + y_{i+1,j} + y_{i,j+1} + y_{i+1,j+1}}{4}$$

Notation Denote $y * W = (_W(y) = ((y)$ if W is understood

Denote by $T_{a,b}$ the translation by (a,b) operation so

$$(T_{a,b}(y))_{ij} = y_{i-a,j-b}$$

Proposition (Translational Equivariance of convolution)

$$C_W(T_{a,b}(y)) = T_{a,b}(C_W(y)) \quad \text{for } \forall\, W \text{ with compact support}$$

Proof:

$$C_W(T_{a,b}(y))_{ij} = \sum_{k,r \in \mathbb{Z}} T_{a,b}(y)_{i+k,j+r}\, W_{k,r}$$

$$= \sum_{k,r \in \mathbb{Z}} y_{i+k-a,j+r-b}\, W_{k,r}$$

$$= (y * W)_{i-a,j-b}$$

$$= T_{a,b}(C_W(y))$$

∎

Convolutional Layer with 2D inputs

Let $I \in \mathbb{R}^{H \times K}$ (grayscale image represented by matrix) and consider a kernel $W \in \mathbb{R}^{h \times k}$ $(h \leq H,\ k \leq K)$.

Define the convolution of W with I of stride $(1,1)$ by

$$\left(I *_{(1,1)} W\right)_{ij} = \sum_{s,r} I_{i+s,\,j+r} W_{s,r} \qquad \begin{matrix} 1 \leq i \leq H-h+1 \\ 1 \leq j \leq K-k+1 \end{matrix}$$

E.g. $\begin{pmatrix} 1 & 2 & 1 \\ 3 & 4 & 0 \\ 2 & 6 & 1 \end{pmatrix} * \begin{pmatrix} -1 & 2 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 6 & 4 \\ 7 & 2 \end{pmatrix}$.

More generally for stride (a,b) define

$$\left(I *_{(a,b)} W\right)_{ij} = \sum_{s,r} I_{a(i-1)+1+s,\ b(j-1)+1+r} W_{s,r}$$

Question: What are the bounds on i,j?

Answer: If $a=b$ and $h=k$ (square kernel) then output has size $\left(\frac{H-h}{a}+1, \frac{K-h}{a}+1\right)$

E.g. $a=2, h=2$

$$I = \begin{pmatrix} 1 & 2 & 3 & 2 \\ 4 & 0 & -1 & 3 \\ 0 & 1 & 2 & 1 \\ 6 & 2 & -1 & 4 \end{pmatrix} \quad W = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \quad I *_{(2,2)} W = \begin{pmatrix} -1 & 4 \\ 1 & 5 \end{pmatrix} \in \mathbb{R}^{2 \times 2}$$

A <u>feature map</u> with 2D input $I \in \mathbb{R}^{H \times K}$ with stride (a,a) and kernel $W \in \mathbb{R}^{h \times h}$, bias $b \in \mathbb{R}^{\left(\frac{H-h}{a}+1\right)\times\left(\frac{K-h}{a}+1\right)}$ has output

$$G\left(I *_{(a,a)} W + b\right) \in \mathbb{R}^{\left(\frac{H-h}{a}+1\right)\times\left(\frac{K-h}{a}+1\right)}$$

for some activation G (almost always ReLU)

<u>Note:</u> By "unwinding" I it is possible to write $I *_{(a,a)} W$ as a matrix × vector product (see stackexchange 2D convolution as matrix product)

<u>Convolution for Colored Images</u>

A <u>colored image</u> I consists of 3 matrices $I_r, I_g, I_b \in \mathbb{R}^{H \times K}$ $\forall$ i,j the RGB values at pixel (i,j) is $\left(I_r[i,j], I_g[i,j], I_b[i,j]\right)$.

Represent I as a triply indexed quantity (a 3-tensor) so $I_{i,j,c}$ $\begin{matrix} 1 \le i \le H \\ 1 \le j \le K \\ 1 \le c \le 3 \end{matrix}$

In general, a 3-tensor can be thought of as a map $\{1,\dots,n_1\}\times\{1,\dots,n_2\}\times\{1,\dots,n_3\} \longrightarrow \mathbb{R}$ for some n_1, n_2, n_3.

Feature maps with 3D input

Given a colored image $I = (I_{ijc})$, let $W = (W_{psc})$ be a 3-tensor kernel

$$1 \le i \le H \qquad 1 \le p \le h$$
$$1 \le j \le K \qquad 1 \le s \le k$$
$$1 \le c \le 3$$

Define $I *_{(1,1)} W$ to be the 3-tensor with

$$(I *_{(1,1)} W)_{i,j,c} = \sum_s \sum_p I_{i+p,j+s,c} W_{p,s,c}$$

(could also define this for arbitrary stride)

∴ a feature map with 3D inputs 各. takes I as input and outputs

$$G(I * W + b)$$

where G is an activation & b has the same shape as $I * W$.

For each layer we would like to use multiple feature maps.

Let (W_t, b_t) $1 \le t \le f$ be a list of kernels and biases (all of same shape)

$f = \#$ features

(Think of each kernel as detecting a certain "feature"
e.g. horizontal lines / vertical lines)

Given (W_t, b_t) $1 \le t \le f$ we can encode this as a 4-tensor $W_{p,s,c,t}$ $1 \le p \le h$, $1 \le s \le h$, $1 \le c \le 3$, $1 \le t \le f$

Given a colored image I, we get features

$$Z_{i,j,c,f} = G\left[\left(\sum_{s,p} I_{i+p,j+s,c} W_{p,s,c,t}\right) + b_{i,j,c,t}\right]$$

$\therefore$ later layers need to take 4-tensors as inputs (and output a 4-tensor)

Given a 4-tensor input $Z^{(l-1)}_{i,j,c,t}$

and $W^{(l)}_{p,s,c,t,t'}$ $b^{(l)}_{i,j,c,t'}$ (weights & biases)

$t \longleftrightarrow$ input features (layer $l-1$)

$t' \longrightarrow$ output features (layer l)

Define $$Z^{(l)}_{i,j,c,t'} = G\left[\left(\sum_{s,p,t} Z^{(l-1)}_{i+p,j+s,c,t} W^{(l)}_{p,s,c,t,t'}\right) + b^{(l)}_{i,j,c,t'}\right]$$

This is a convolutional layer.

Next: Pooling layers

Note: The last formula written down last time should be

$$Z^{(l)}_{i,j,c,(f_{in},f_{out})} = G\left[\left(\sum_{s,p} Z^{(l-1)}_{i+p,j+s,c,f_{in}} W^{(l)}_{p,s,c,(f_{in},f_{out})}\right) + b^{(l)}_{i,j,c,(f_{in},f_{out})}\right]$$

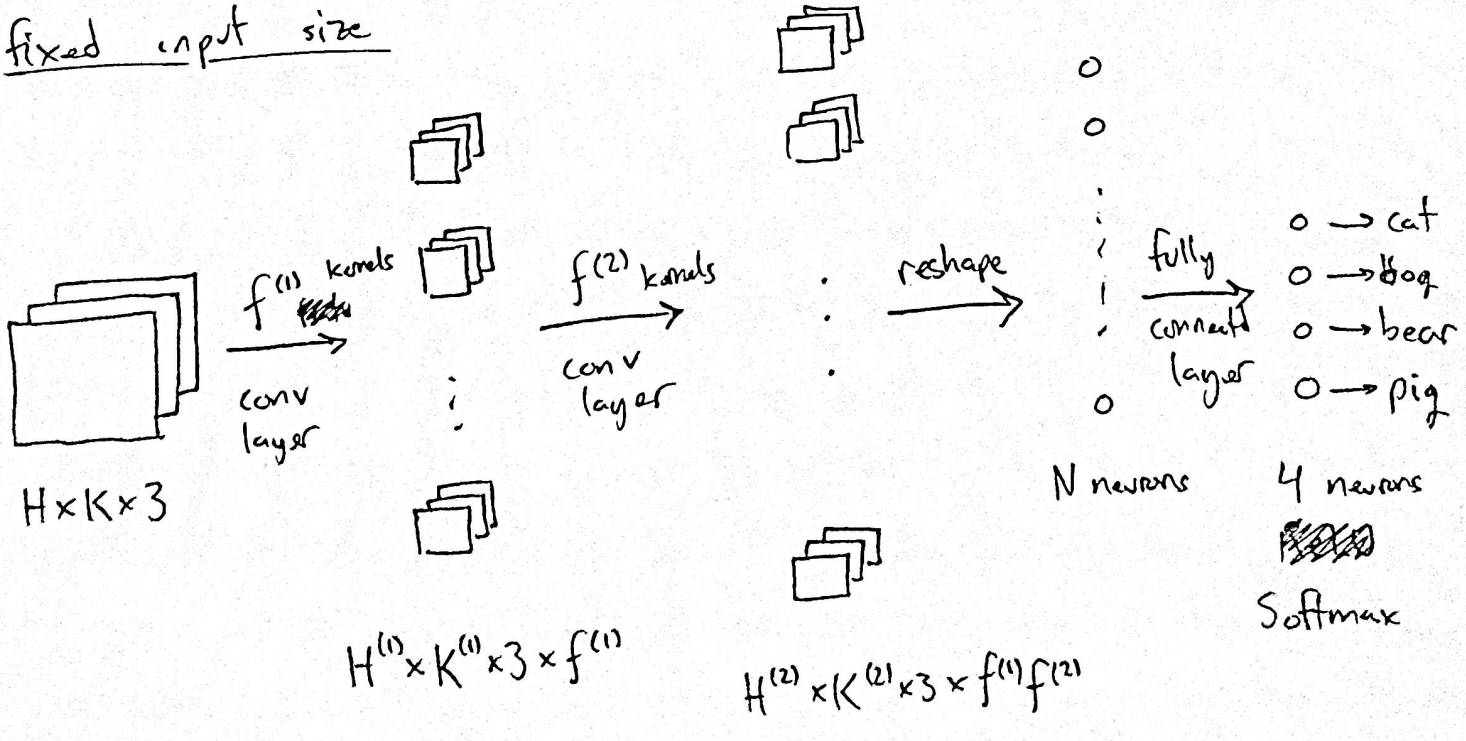
and then "flatten" index (f_{in}, f_{out}) to new index f

so # features in layer l is $f_{in} f_{out}$

∴ # features increases in a convolutional layer.

An example of a CNN to predict cat/dog/bear/pig

fixed input size



Note: N will be very large ... this is not the most efficient approach.

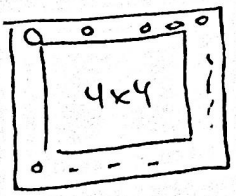
In using convolutional layers, we find

$$H^{(2)} \leq H^{(1)} \leq H$$
$$K^{(2)} \leq K^{(1)} \leq K$$
$$f^{(1)}f^{(2)} \geq f^{(1)} \geq 1 \qquad (\#\text{ features grows})$$

Sometimes we may want to preserve the # pixels in features. Can fix this with "padding" (adding some # of 0's surrounding image)

E.g. $(4\times4) *_{(1,1)} (2\times2) = (3\times3)$

with 1-padding



$*_{(1,1)} (2\times2) = (5\times5)$

If we want to use p-padding to preserve image size with $h\times h$ filter & $H\times K$ image

$$H\times K = (H+2p-h+1)\times(K+2p-h+1)$$

so $p = \frac{h-1}{2}$ (so h must be odd)

$\therefore$ usually people use h = odd (often $h=3$)

Padding to preserve size is called "same" padding.

Pooling Layers

Local pooling Given image $I \in \mathbb{R}^{H \times K}$ / local pooling consists of partitioning I and computing some local statistic (max, min, avg)

E.g.

1	2	3	1
1	5	2	7
10	1	2	4
8	9	1	8

$\xrightarrow[\text{max pool}]{(2,2)}$ $\begin{pmatrix} 5 & 7 \\ 10 & 8 \end{pmatrix}$

Gives a way to reduce image size if needed

For a 4-tensor $Z_{i,j,c,f}$ pool for each fixed c,f to get a 4-tensor $Z'_{i,j,c,f}$ with smaller bounds on i,j & same bounds on c,f.

Global Pooling (Local pooling where we pool over the whole image)

Given 4-tensor $Z_{i,j,c,f}$ take (max, min, avg) over all i,j to get a 2-tensor $Z'_{c,f}$

E.g. Redo the previous example but better