Linear algebra review

① Fundamentals

scalars single numbers

vectors 1d array of numbers, $\tilde{x}$. i^{th} element as x_i. position in $\mathbb{R}^n$ space.

$$\tilde{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}$$

matrices 2d array of numbers, A. i^{th} row, j^{th} column as A_{ij}. $A = \begin{bmatrix} A_{11}, A_{12} \dots \\ A_{21}, A_{22} \dots \\ \vdots \quad \vdots \end{bmatrix}$

transpose $(A^T)_{ij} = A_{ji}$

tensors a general n-dimensional array of numbers. Element identified by $A_{ij\dots n}$

dot product $\tilde{x} \cdot \tilde{y} = \tilde{x}^T \tilde{y} = \sum_n x_i \times y_i$ as matrix mult

$$\frac{\tilde{x} \cdot \tilde{y}}{\|\tilde{x}\|_2 \|\tilde{y}\|_2} = \cos\theta$$

matrix multiplication $A \times B = C$ ($m_1 \times n$, $n \times m_2$, $m_1 \times m_2$)

$C_{ij} = \sum_k A_{ik} B_{kj}$ = dot product of row i of A and column j of B

matrix hamadard product elementwise multiplication.

↳ sometimes we need to measure the size of a vector

norm any function f that : $f(\tilde{x}) = 0 \Rightarrow \tilde{x} = \tilde{0}$

$f(\tilde{x} + \tilde{y}) \leq f(\tilde{x}) + f(\tilde{y})$, triangle inequality

$\forall \alpha \in \mathbb{R}, f(\alpha \tilde{x}) = |\alpha| f(\tilde{x})$

L^p norm $\|x\|_p = \left[\sum_i |x_i|^p \right]^{1/p}$

L^∞ norm $\|x\|_\infty = \max_i |x_i|$

↳ sometimes, we may also wish to measure the size of a matrix.

Frobenius norm $\|A\|_F = \sqrt{\sum_i \sum_j A_{ij}^2}$

② gradients

vector in, scalar out $y = f(\tilde{x})$ $f: \mathbb{R}^n \to \mathbb{R}$

$$\frac{\partial y}{\partial \tilde{x}} = \left(\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2} \cdots\right) = \left(\frac{\partial y}{\partial x_i}\right)_{i \in \{1 \ldots n\}}$$

vector in, vector out : Jacobian $\tilde{y} = f(\tilde{x})$ $f: \mathbb{R}^n \to \mathbb{R}^m$

$$J = \frac{\partial \tilde{y}}{\partial \tilde{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & & \\ \frac{\partial y_m}{\partial x_1} & & \end{pmatrix}$$

→ x

→ wrt to each y, for each x

$$J_{ij} = \frac{\partial y_i}{\partial x_j}$$

↓ J

↓ wrt each x, for each y

matrix chain rule as Jacobian products

$$\frac{\partial \tilde{z}}{\partial \tilde{x}} = \frac{\partial \tilde{z}}{\partial \tilde{y}} \cdot \frac{\partial \tilde{y}}{\partial \tilde{x}} = J_{zy} \times J_{yx}$$

useful identities

$$\tilde{z} = W\tilde{x} \Rightarrow \frac{\partial \tilde{z}}{\partial \tilde{x}} = W$$

$$\tilde{z} = \tilde{x}W \Rightarrow \frac{\partial \tilde{z}}{\partial \tilde{x}} = W^T$$

$$\tilde{z} = \tilde{x} \Rightarrow \frac{\partial \tilde{z}}{\partial \tilde{x}} = I$$

$$\tilde{z} = W\tilde{x}, \ \tilde{f} = \frac{\partial j}{\partial \tilde{z}} \Rightarrow \frac{\partial j}{\partial W} = f^T x^T$$

$$\tilde{z} = \tilde{x}W, \ \tilde{f} = \frac{\partial j}{\partial \tilde{z}} \Rightarrow \frac{\partial j}{\partial W} = x^T f$$

$$\tilde{y} = \text{softmax}(\theta), \ j = \text{cross entropy}(\tilde{y}, \hat{y}) \Rightarrow \frac{\partial j}{\partial \theta} = \tilde{y} - \hat{y}$$

or $(\hat{y} - y)$ if y is a column vector (i.e. $c \times 1$)

usually we do $c \times n$ so yes

image classification with neural networks

① the image classification problem

images as matrices and vectors — image as a 3D tensor : stacked matrices of pixels, one matrix for each input channel. stack of images as 4D tensor

task definition — assigning an input image one label from a fixed set of categories

challenges

- **Viewpoint variation**. A single instance of an object can be oriented in many ways with respect to the camera.
- **Scale variation**. Visual classes often exhibit variation in their size (size in the real world, not only in terms of their extent in the image).
- **Deformation**. Many objects of interest are not rigid bodies and can be deformed in extreme ways.
- **Occlusion**. The objects of interest can be occluded. Sometimes only a small portion of an object (as little as few pixels) could be visible.
- **Illumination conditions**. The effects of illumination are drastic on the pixel level.
- **Background clutter**. The objects of interest may *blend* into their environment, making them hard to identify.
- **Intra-class variation**. The classes of interest can often be relatively broad, such as *chair*. There are many different types of these objects, each with their own appearance.

A good image classification model must be invariant to the cross product of all these variations, while simultaneously retaining sensitivity to the inter-class variations.

---

② linear classification as parametrized scoring

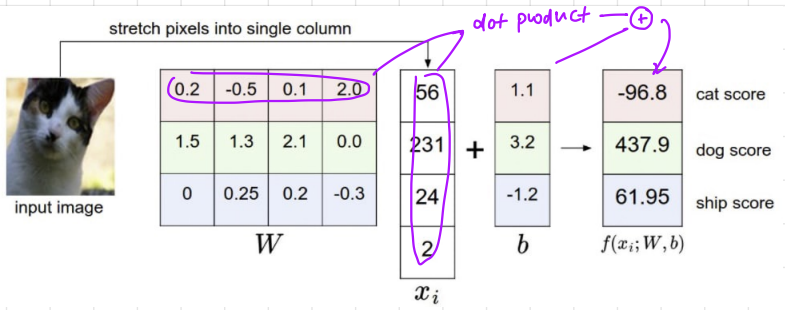
↳ An intuitive way to map images to a label is to use a **score function** that maps all the pixel values of an image to a **confidence score** for each class

↳ we can formalize this function by considering an input image tensor as a single (reshaped) vector of dimension D. eg. $3 \times 32 \times 32 \Rightarrow 3072$

$$f: \mathbb{R}^D \rightarrow \mathbb{R}^C$$

↳ we can start with the simplest function : a linear mapping

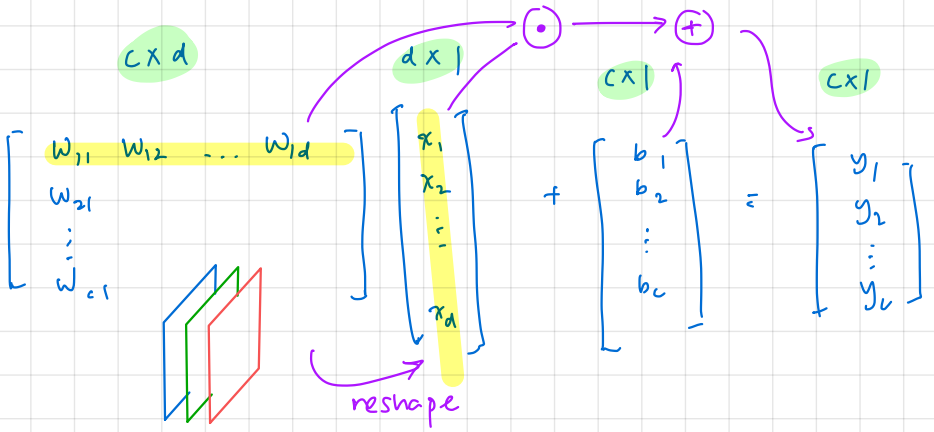
$$f(\tilde{x}, W, b) = W\tilde{x}^T + \tilde{b} \qquad \hat{\text{label}} = \arg\max \left[f(\tilde{x}, W, b) \right]$$



interpreting linear classifiers : template matching

↳ observe that the matrix product $W\vec{x}$ takes the i^{th} row of W and takes that dot product with x to get a value

↳ a natural interpretation that arises is thus using each row of W as a template, and their inner product as a score of "template matching"



interpretation as multiple linear regression lines

↳ observe that dot product is simply a sum of products

$$y_i = \sum_{j=1}^{d} W_{ij} x_j \Rightarrow$$ linear regression, where each y_i has its own line but "access" to all x

---

③ saving with the softmax and sigmoid

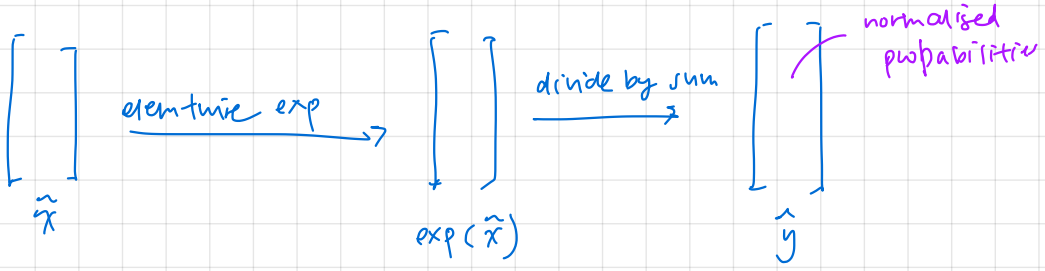
↳ an intuitive concept to interpret the scores would be through the notion of normalized class probabilities ⇒ we can post-process the scores by normalising them such that they sum to 1.

↳ the softmax and sigmoid functions can do that in a way that doesn't leave the result overly sensitive to extreme values, and are naturally interpretable.

As $x_i \to -\infty$, $\hat{y}_i \to 0$. As $x_i \to \infty$, $\hat{y}_i \to 1$

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

$$\text{softmax}(\vec{x}) \Rightarrow y_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$



④ notion of quality: loss functions

↳ we have it now that $\hat{y} = \text{softmax}(W\vec{x} + b)$, and $\hat{y}_i$ is the probability of a given class given the input image.

↳ we can evaluate the quality of the model by computing the average confidence with which it predicts the correct class over the dataset.

⇒ we use the geometric mean

With a training set containing N labeled pictures :

$$Q_{\text{mean}} = \left(\prod_{i=1}^{N} \text{entry cl}(i) \text{ of vector } \mathbf{p}^{(i)} \right)^{1/N}$$

- cl(i): The class index of the i^{th} picture (integer)
- entry cl(i) of vector $\mathbf{p}^{(i)}$: How confident the network is that the i^{th} picture belongs to the class cl(i)
- $\mathbf{p}^{(i)}$: Network output for the i^{th} picture $\mathbf{x}^{(i)}$:

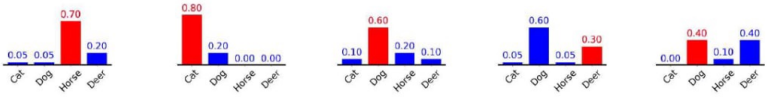
$$\mathbf{p}^{(i)} = \text{Softmax}(W\mathbf{x}^{(i)} + \mathbf{b})$$

Vector with **4 entries**

correct label of i^{th} sample

in best case ⇒ 1
in worst case ⇒ $\frac{1}{C}$, like random guess

Network prediction



Quality of the network

$$Q_{\text{mean}} = (0.70 \times 0.80 \times 0.60 \times 0.30 \times 0.40)^{1/5} = 0.53$$

Geometric mean

In average, the network gives a 53% confidence score to the correct class

↳ but the computation of product functions (and their gradients) can be very expensive ⇒ we take the -log to convert to sums!

$$-\log\left[\left(\prod_{i=1}^{N} \text{entry cl}(i) \text{ of } \tilde{p}_i\right)^{1/N}\right] = -\frac{1}{N}\log \prod_{i=1}^{N} \text{entry cl}(i) \text{ of } \tilde{p}_i$$

$$= -\frac{1}{N}\sum_{i=1}^{N} \log(\text{entry cl}(i) \text{ of } \tilde{p}_i)$$

$$= -\frac{1}{N}\sum_{i=1}^{N} \text{entry cl}(i) \text{ of } \log(\tilde{p}_i)$$

log softmax

$$= -\frac{1}{N}\sum_{i=1}^{N} \tilde{y} \cdot \log(\tilde{p}_i)$$

one hot encoded, all but entry cl(i) is zero dot product as it select.

$$= \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_j \log(p_j)$$

interpretation of loss ↓ avg. cross entropy for the data set

2) information theory view

softmax classifier the probabilistic interpretation of our softmax classifier means that we can input $\hat{y}_j$ as the probability of x being the j^{th} class, conditioned on x and w. observe that the output vector $\hat{y}$ can be interpreted as a probability distribution and y as the true conditional distribution (conditioned on x).

entropy is expected number of bits required to transmit information about a distribution.

$$H = -\sum_x p(x) \log_2 p(x)$$

- $p(x)$: frequency of occurence
- $\log_2 p(x)$: information about event (its probability)
- $\log_2 p(x)$: bits to encode the event information
- $\sum_x$: for each value in distribution

cross entropy the cross entropy between a "true" distribution $p(x)$ and estimated distribution $q(x)$ is defined as

$$H(p, q) = -\sum_x p(x) \log_2 q(x)$$

and is intuitively the amount of information (in bits) on average required to transmit information of a distribution q based on frequencies in distribution p.

$\Rightarrow$ in a classification problem, we encode y in one hot encodings. which is to say, the entropy of a correct prediction is 0, since

$$\sum_x p(x) \log p(x) \quad \text{, where } x = 1 \text{ if correct, else } 0, \; = (1) \log(1) = 0.$$

i.e. for a correct prediction, there is no information to be gained from this sample. but for a wrong prediction, $p(x) \neq q(x)$ in general, and the loss value is thus the expected number of bits to represent information about $\hat{p}$ (distribution predicted by model) given true distribution y.

In other words, the error signal is "error correction" information.

⑤ non-linear classification : neural networks

1) need for non-linearity

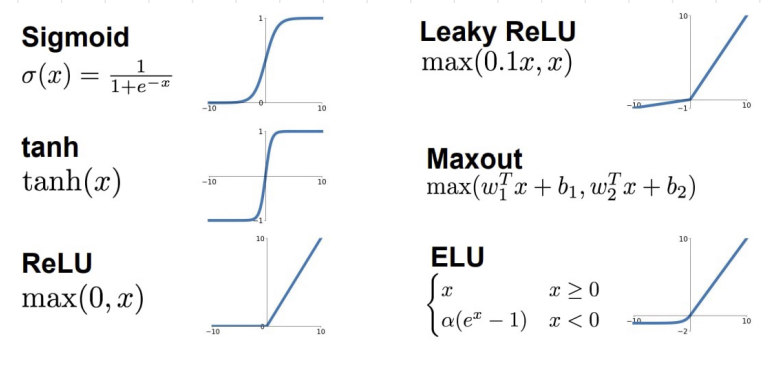
↳ a linear model is fundamentally unable to express non-linear relationships

↳ mathematical intuition : linear regression is just linear regression : without non-linear functions, y_i will never be a non-linear function of $\tilde{x}$, even if you compose functions.

$$B(Ax) \Rightarrow (BA)x = cx \Rightarrow \text{linear!}$$

how do we inject non-linearity?

2) activation functions



applied elementwise

3) MLPs as composed functions

↳ a multi-layer perceptron or neural network is exactly a composition of composed linear and non-linear functions that can approximate any "true" function.

mathematically

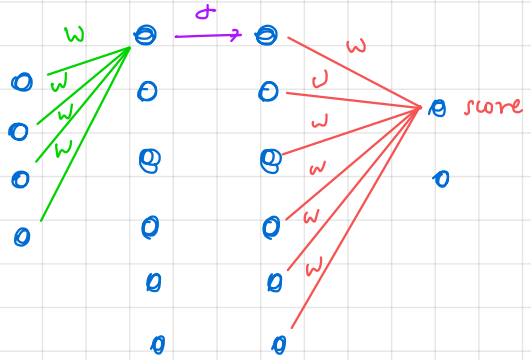
$$\sigma\left(W_2\left[\sigma(W_1\tilde{x} + b_1)\right] + b_2\right) \dots$$

inner

again

visually as graphs

- nodes as values
- edges as weights or transforms



⇒ why do MLPs work so well?

composed non-linear functions give more room for movement and non-linearly combine input x_i, for each hidden value h_i, all and the next hidden value can do the same for its input, and so on, until $\hat{y}_i$

⑥ optimisation as goal search : gradient descent

↳ we now have an arbitrarily expressive model — an MLP, and a loss function that measures the performance of the model wrt. a ground truth conditional distribution

↳ how do we decide the weights $\{W_i\}_{i \in \text{layers}}$?

1) formulation as goal search

↳ goal : we want to find an optimal set of weights that will produce a globally minimal loss on data that it sees.

↳ constraints : There are no closed form solutions.

2) loss as a function of model parameters

↳ we recognise mathematically that in the forward pass, the output distribution is a function of the model weights — and we extend that notion to the loss function

3) naive approaches : random search and local search

naive : random search

↳ try randomly until good set is found

better : random local search

↳ we can express loss function as function of W.

$$\text{loss} = \text{Loss}(\hat{y}, y)$$
$$= \text{Loss}(f(W, \tilde{x}), y)$$

↳ random search by perturbation :

$W \leftarrow W + \delta$, update iff loss decreases

(δ : random perturbation)

can we intelligently decide how to perturb W ?

⑦ gradient descent

gradient descent

↳ recall that for $y = f(x_1, x_2 \dots) \Rightarrow \frac{\partial y}{\partial x} = \left(\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2} \dots\right)$ and that $\frac{\partial y}{\partial x}$ is the vector direction of steepest ascent (gradient) since it is exactly how much y will change by

↳ we can capitalize on this by:

$$w_{ij} = w_{ij} - \alpha \frac{\partial L}{\partial w_{ij}}$$

descent; direction of steepest descent; learning rate = control step size

```
W ← random W
b ← random b
α ← some value
while λ > ε:
    Ŷ = σ(WX + b)
    λ = Y^T · log Ŷ
    ∂λ/∂W = compute_gradients(λ, W)
    ∂λ/∂b = compute_gradients(λ, b)
    W = W − α ∂L/∂W
    b = W − α ∂L/∂b
```

matrix formulation of gradient update

$$\lambda = Y^T \cdot \log\left(\sigma\left(\begin{bmatrix} & \\ & \end{bmatrix}\begin{bmatrix} & \\ & \end{bmatrix} + \begin{bmatrix} \\ \end{bmatrix}\right)\right) / n$$

W, X, b; averaging is done

$$W = W - \alpha \cdot \frac{\partial L}{\partial W} \qquad b = b - \alpha \frac{\partial L}{\partial b}$$

minibatch gradient descent

↳ in large scale applications, it is expensive to compute the full loss over the entire training set only to perform a single parameter update

↳ supposing our sample (or batch) of the training set is representative on average, then we can get a consistent estimate of the true gradient since we use the average loss

↳ the noise from random sampling also helps with traversing noisy optimisation landscapes full of local minima

⇒ an update is performed every batch.

$$\lambda_{batch} = Y_{batch} \cdot \log(\hat{Y}_{batch}) / \text{batch size}$$

$$W = W - \alpha \frac{\partial \lambda_{batch}}{\partial W}$$

stochastic gradient descent

↳ the extreme case of minibatch gradient descent where batch size = 1

↳ often used interchangeably w/ minibatch gradient descent. Nature of stochasticity arises from random sampling of input to compute loss and update.

---

⑦ back propagation

↳ gradient descent gives us a good way to update gradients. So how do we compute gradients?

↳ we could compute them manually for each parameter, but there could be a lot of repeated computation since the function is composed and the chain rule applies sequentially when

⇒ idea: dynamic programming — cache gradient results and "back propagate" them (reuse)

chain rule

↳ consider a two layer network

$$\tilde{y} = W_1 \hat{x} \quad (d \times d)(d \times n)$$

$$\hat{y} = f(y), \ f \text{ is activation}$$

$$\tilde{z} = W_2 \hat{y} \quad (c \times d)(d \times n)$$

$$p = \text{softmax}(\tilde{z}) \quad c \times n$$

$$L = y^T \cdot \log(p) / n \quad (n \times c)(c \times n)$$

$$\frac{\partial L}{\partial W_2} = \underbrace{\frac{\partial L}{\partial p} \cdot \frac{\partial p}{\partial z}}_{\text{input}} \cdot \underbrace{\frac{\partial z}{W_2}}_{\text{local}}$$

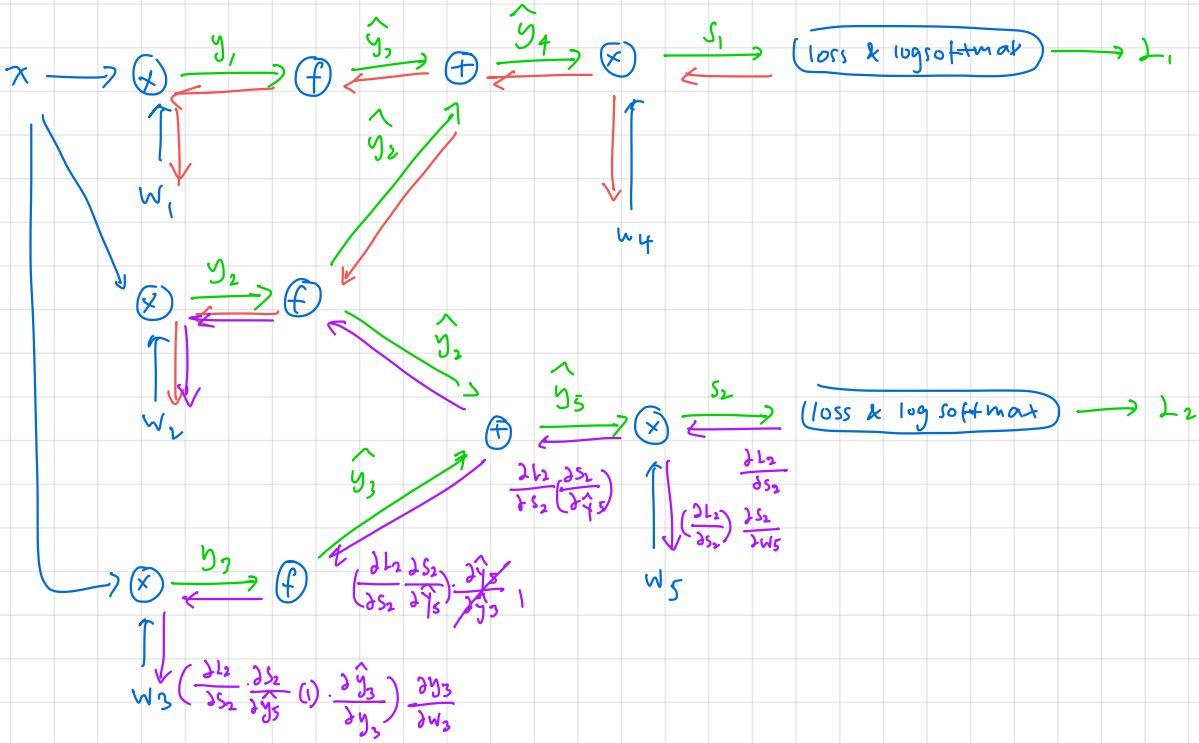
repeated

$$\frac{\partial L}{\partial W_1} = \underbrace{\frac{\partial L}{\partial p} \cdot \frac{\partial p}{\partial z} \cdot \frac{\partial z}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial y}}_{\text{input}} \cdot \underbrace{\frac{\partial y}{\partial W_1}}_{\text{local}}$$

visualisation as a computational graph

at the matrix level, visualise forward and backward passes as a graph. nodes as operations, and "dropout nodes" as weights

$$Y_1 = W_1 X \qquad Y_2 = W_2 X \qquad Y_3 = W_3 X$$
$$\hat{Y}_1 = f(Y_1) \qquad \hat{Y}_2 = f(Y_2) \qquad \hat{Y}_3 = f(Y_3)$$
$$\hat{Y}_4 = \hat{Y}_1 + \hat{Y}_2 \qquad S_1 = W_4 \hat{Y}_4 \qquad \hat{Y}_5 = \hat{Y}_2 + \hat{Y}_3$$
$$L_1 = Y_1^T \cdot \log\text{softmax}(S_1)/n \qquad S_2 = W_5 \hat{Y}_5$$

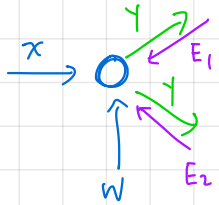


backward gradient = inflow gradient × local gradient (chain rule)

patterns in backward flow

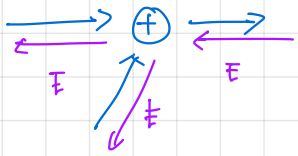
1. gradients add up at forks.

↳ This follows the multivariable chain rule, which states that if a variable "branches" out, then gradients that flow back will add.

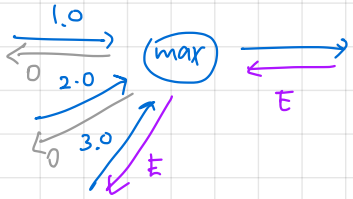


$$\frac{\partial L}{\partial W} = \frac{\partial y}{\partial W} \cdot (E_1 + E_2)$$

2. The "add gate" will simply distribute its gradient back without changing it to all its inputs. Because local gradient for a sum is 1. $y = x + z \Rightarrow \frac{dy}{dx} = 1$

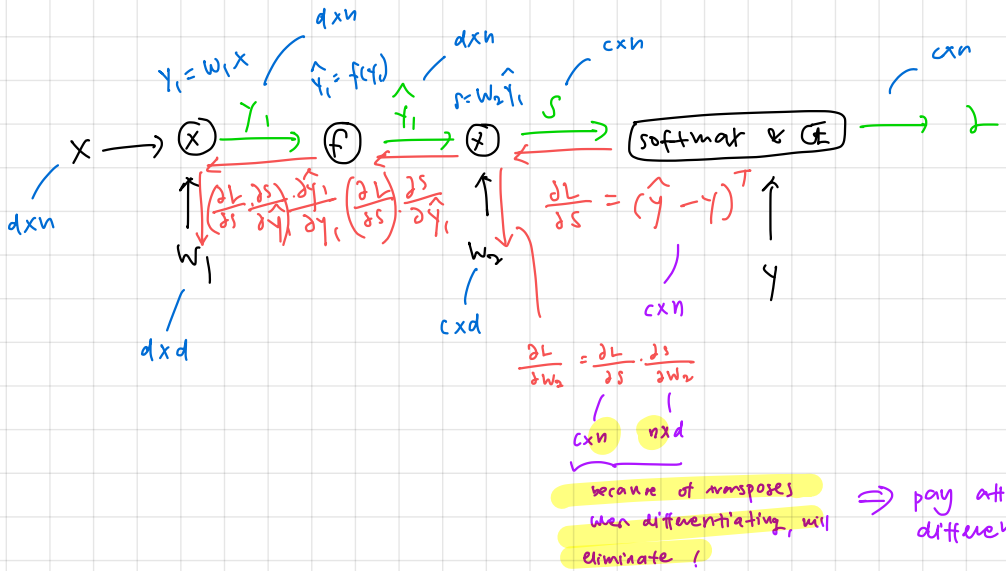


3. The "max" gate will switch its gradient only to its max input. This is because local gradient is 1 for highest and 0 for all others. Another way to think is that it did not contribute to later computation, so no gradients need to be passed back.



4. The "multiply" gate is such that local gradients are the inputs, but switched.
$y = wx \Rightarrow \frac{dy}{dw} = x$ but $\frac{dy}{dx} = w$.

product of Jacobian and consistent dimensions



because of transposes when differentiating, will eliminate!

$\Rightarrow$ pay attention to order when differentiating in identities!

convolutional neural networks

① Local & heirarchical features in images

↳ observe that

1. images are compositional — they are formed from local patterns and can be stacked to form heirarchical patterns
2. the traditional MLP would do template matching at the whole-image level, requiring a lot of parameters since images scale with a square
3. patterns in images are translation invariant

→ can we exploit these observations in our architective?

② convolutional operation: local pattern matching

1) convolution operation (strided)

↳ idea: conduct template matching locally using dot products and non-linearities!

per convolution filter

N-dimensional input ⟹ cut out local $N \times K \times K$ portion

⇓

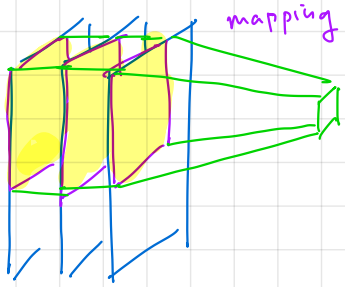
dot product w/ $N \times K \times K$ parameters

i.e. elementwise multiply, sum all

⇓

$1 \times 1 \times 1$ activation at center

stride to next portion



output shape

$n_h \times n_w$ → $K_h \times K_w$ filter, P_h, P_w padding, stride S_h, S_w

$$\left\lfloor \frac{n_i + 2p_i - K_i}{s_i} + 1 \right\rfloor$$

$$\frac{n + 2p - K}{s} + 1 = \begin{cases} n & s = 1,\ p = \frac{K-1}{2} \\ r & p = K - 1 \end{cases}$$

N-D to M-D convolution

$c_1 \times h_1 \times w_1$ → $1 \times h_2 \times w_2$ activation map

$c_1 \times K \times K$ filter

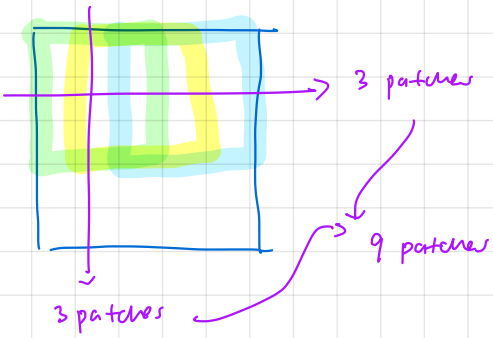
$c_1 \times h_1 \times w_1$ → $c_2 \times h_2 \times w_2$ ⇒ $c_2 \times c_1 \times K \times K$ parameters

$c_2 \times (c_1 \times K \times K$ filter$)$

2) implementation as matrix multiplications

↳ observation 1: convolutions are applied on patches!

eg. 5×5 image & 3×3 filter

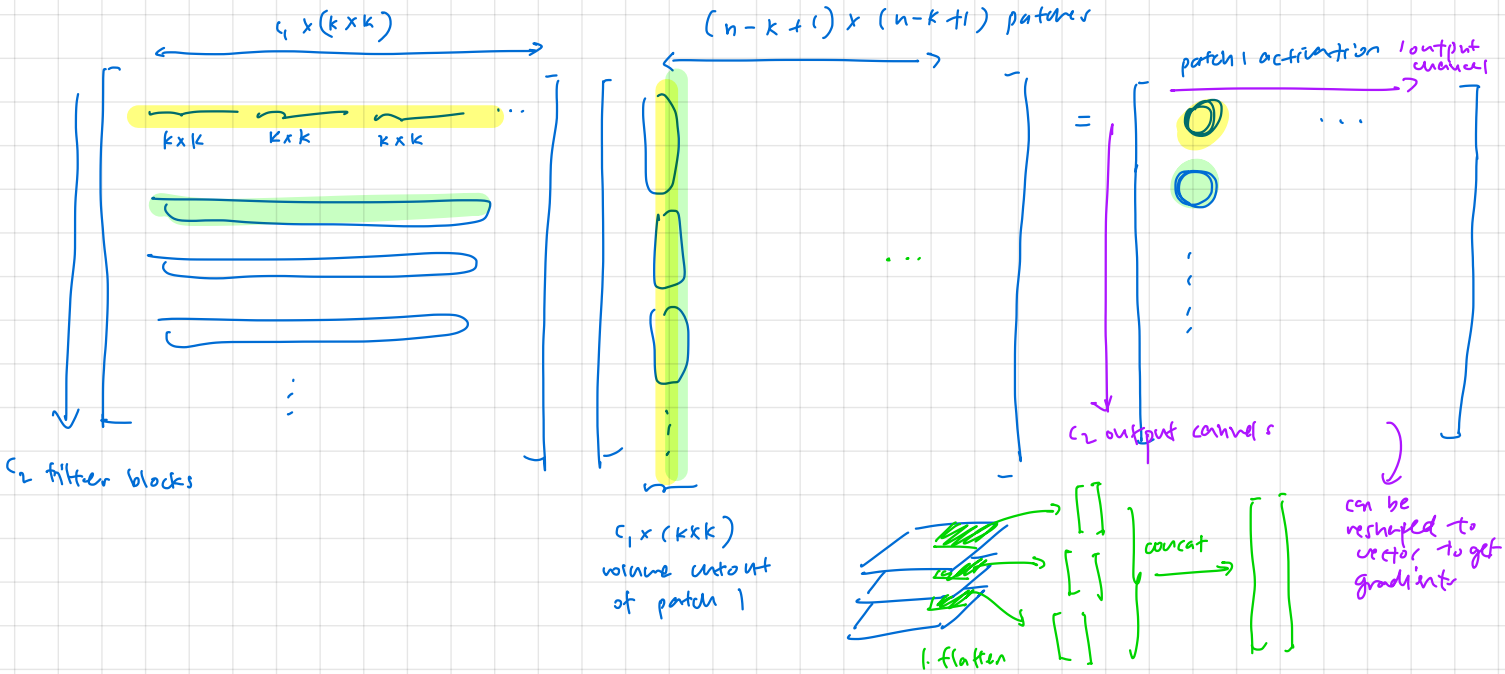


↳ observation 2: N-D convolution parameter "blocks" apply dot product channel-wise

↳ reshape as matrix multiplication

stride 1, padding 0

e.g. $c_1 \times n \times n$ to $1 \times (n-k+1) \times (n-k+1)$



---

③ activation aggregation

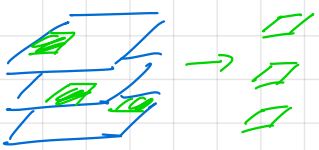
↳ observe that:

1. image data is very large (n^2), can contain a lot of noise
2. image features are naturally translation invariant

↳ we can somehow subsample the image to get a more compressed representation

pooling: sliding filter like convolution, applies max / avg. patch and channel wise

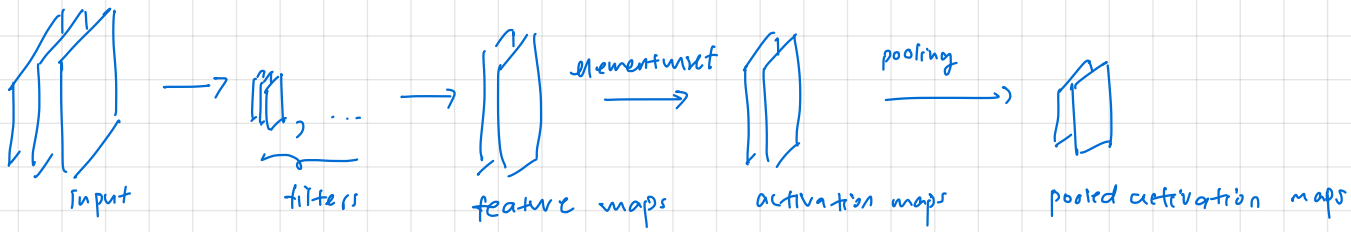
$c_1 \times k \times k$ patch $\rightarrow$ $c_1 \times 1 \times 1$ pooled volume



$c_1 \times n_h \times n_w \xrightarrow{k_i, s_i} \lfloor n_i = \frac{(n_i - k)}{s} + 1 \rfloor$ same as conv.

$c_1 = c_2$, same no. of channels since done channel-wise

④ convolution layer



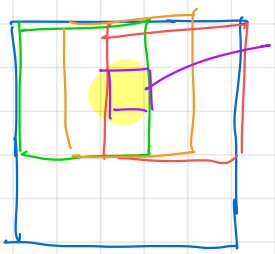
---

⑤ gradients and backpropagation

↳ conventionally, we model an MLP as composed functions.

$$\left.\begin{aligned} y &= Wx \\ \hat{y} &= \sigma(y) \\ l &= l(\hat{y}, y_0) \end{aligned}\right\} \text{ then } \frac{\partial l}{\partial W} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial y} \cdot \frac{\partial y}{\partial x}$$

↳ with convnets, we need to consider the chain rule and observe that a single pixel feeds into many outputs



→ pixel part of three outputs !

⇒ (trick : transform x)

$$\left.\begin{aligned} \hat{X} &= X \text{ image} \end{aligned}\right\} \text{ then } \frac{\partial l}{\partial W} = \frac{\partial l}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial y} \cdot \frac{\partial y}{\partial \hat{X}}$$

patches from image (has repetitions)

↳ then the next question : how to map back to previous layers ?

$$\left.\begin{aligned} \hat{x}_1 &= t(x_1) \\ y_1 &= W_1 \hat{x}_1 \\ \hat{y}_1 &= \sigma(y_1) \\ \hat{x}_2 &= t(\hat{y}_1) \\ y_2 &= W_2 \hat{x}_2 \\ l &= l(y_2, y) \end{aligned}\right\}$$

$$\frac{\partial l}{\partial W_2} = \frac{\partial l}{\partial y_2} \cdot \frac{\partial y_2}{\partial \hat{x}_2}$$

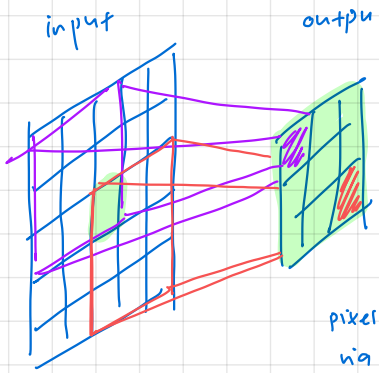
↳ reshape and do as it MLP, easy !

$$\frac{\partial l}{\partial W_1} = \frac{\partial l}{\partial y_2} \cdot \frac{\partial y_2}{\partial \hat{x}_2} \cdot \frac{\partial \hat{x}_2}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial y_1} \cdot \frac{\partial y_1}{\partial \hat{x}_1}$$

what is the gradient of the patch generation function ?

⇒ solution : sum at forks !

anywhere pixel is used for patch, map back.



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

pixel contributes to all via sum

$$y = \sum_{\text{channel}} \sum_{\text{patch}} \sum_{\text{index in patch}} w_i x_i$$

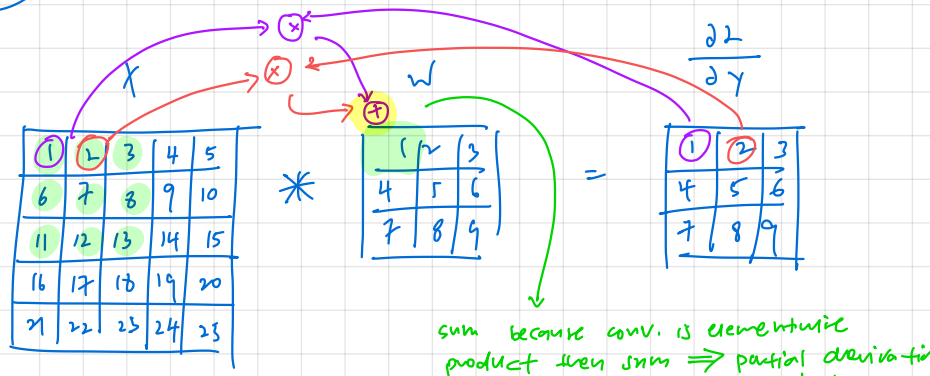
$$\frac{\partial y}{\partial w_i} = \sum_{\text{channel}} \sum_{\text{patch}} \sum_{\text{index in patch}} x_i$$

$$\frac{\partial y}{\partial x_i} = \sum_{\text{channel}} \sum_{\text{patch}} \sum_{\text{index in patch}} w_i$$

1. for each kernel value, track where it multiplied
2. multiply relevant $\frac{\partial L}{\partial y}$ w/ x and sum

$\frac{\partial L}{\partial w}$

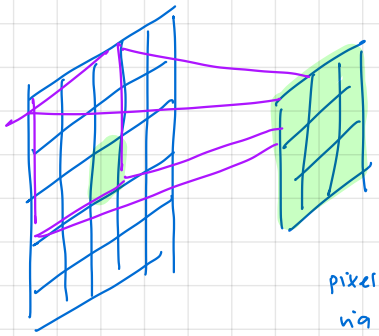
$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w} = \frac{\partial L}{\partial y} \cdot x$$



sum because conv. is elementwise product then sum ⇒ partial derivative also just a sum

$\frac{\partial L}{\partial x}$ for back prop

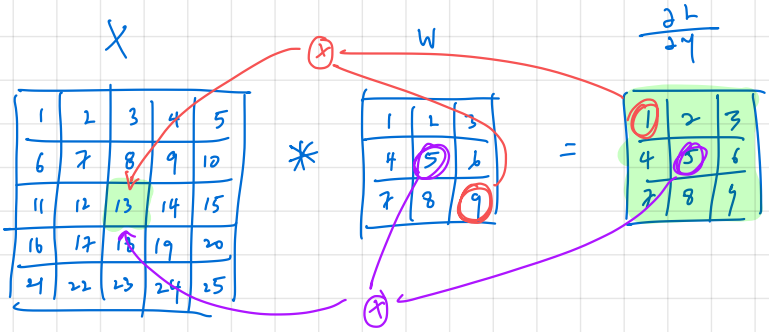
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial x} = \frac{\partial L}{\partial y} \cdot w$$



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

pixel contributes to all via sum



⑥ why do convnets work?

1. regularisation by patch & parameter sharing (less overfitting)
2. translation invariant
3. multistep reasoning by heirarchical features in images, capturing local correlations at each level ⇒ local look at details, global look at abstract level (growth of receptive field)

interpreting CNNs

⑥ receptive field

1) notion of receptive field

receptive field size of the region in the input that produces the feature (pixel)
note that for FCN, we don't care about RF since it has access to the whole input @ every layer

↳ thinking about the receptive field lets us reason about what information a given feature / layer has access to. eg. object detection → small RF, hard to detect large objects.

housing price estimation → small RF, hard to see faraway features like bus stop

2) calculating RF for single path networks

Notable CNN designs

① historically relevant designs

② other designs

- **LeNet**. The first successful applications of Convolutional Networks were developed by Yann LeCun in 1990's. Of these, the best known is the LeNet architecture that was used to read zip codes, digits, etc.
- **AlexNet**. The first work that popularized Convolutional Networks in Computer Vision was the AlexNet, developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton. The AlexNet was submitted to the ImageNet ILSVRC challenge in 2012 and significantly outperformed the second runner-up (top 5 error of 16% compared to runner-up with 26% error). The Network had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).
- **ZF Net**. The ILSVRC 2013 winner was a Convolutional Network from Matthew Zeiler and Rob Fergus. It became known as the ZFNet (short for Zeiler & Fergus Net). It was an improvement on AlexNet by tweaking the architecture hyperparameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller.
- **GoogLeNet**. The ILSVRC 2014 winner was a Convolutional Network from Szegedy et al. from Google. Its main contribution was the development of an *Inception Module* that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M). Additionally, this paper uses Average Pooling instead of Fully Connected layers at the top of the ConvNet, eliminating a large amount of parameters that do not seem to matter much. There are also several followup versions to the GoogLeNet, most recently Inception-v4.
- **VGGNet**. The runner-up in ILSVRC 2014 was the network from Karen Simonyan and Andrew Zisserman that became known as the VGGNet. Its main contribution was in showing that the depth of the network is a critical component for good performance. Their final best network contains 16 CONV/FC layers and, appealingly, features an extremely homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from the beginning to the end. Their pretrained model is available for plug and play use in Caffe. A downside of the VGGNet is that it is more expensive to evaluate and uses a lot more memory and parameters (140M). Most of these parameters are in the first fully connected layer, and it was since found that these FC layers can be removed with no performance downgrade, significantly reducing the number of necessary parameters.
- **ResNet**. Residual Network developed by Kaiming He et al. was the winner of ILSVRC 2015. It features special *skip connections* and a heavy use of batch normalization. The architecture is also missing fully connected layers at the end of the network. The reader is also referred to Kaiming's presentation (video, slides), and some recent experiments that reproduce these networks in Torch. ResNets are currently by far state of the art Convolutional Neural Network models and are the default choice for using ConvNets in practice (as of May 10, 2016). In particular, also see more recent developments that tweak the original architecture from Kaiming He et al. Identity Mappings in Deep Residual Networks (published March 2016).

⑦ variant convolutions

5) dilated convolutions

6) 1×1 convolution

separable convolutions