Neural Networks

A lot of hype!

- Very efficient in practice

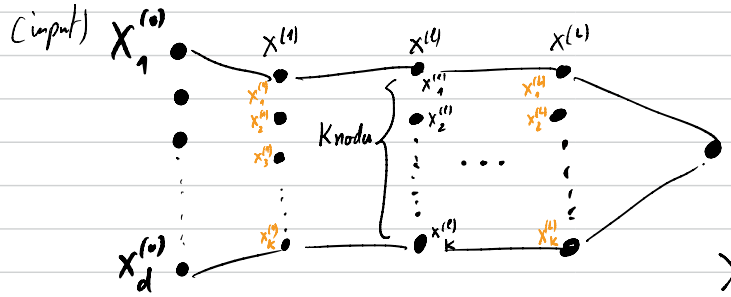
Neural Network ?

- So far we have considered supervised learning (X, Y) → given a new X predict Y
- Linear methods work well when used with good features
  - Before: domain experts derive efficient features.
  - Now: Neural Net. learning from the data a good representation of them

The Basic Structure

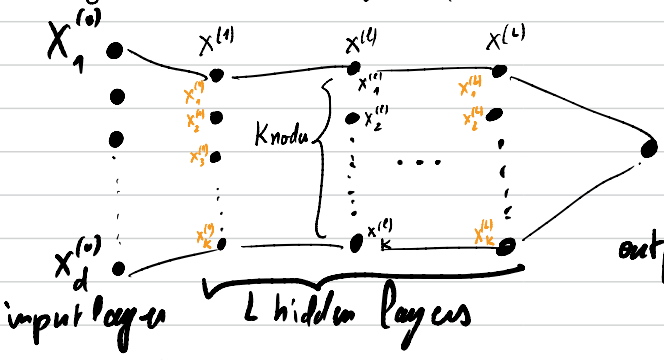
$$S_{train} = \{(x_i, y_i)\}_{i=1}^{N} \quad x_i \in \mathbb{R}^d$$

- A N-N with L hidden layers of size K and one output layer is:



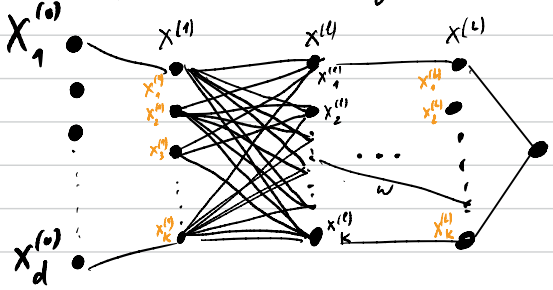
$X_i^{(l)}$: value for layer # l node # i

- each layer has the same # of nodes K
- There is no feedback loop (⇒ Feed Forward Network)

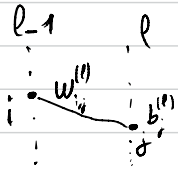


(L, K are very large these days)

- The first hidden L layers represent some transformation of the data.



- each node is connected to all the previous nodes (the network is fully connected)
- each edge has an associated weight:
  - weight $W_{i,j}^{(l)}$ from (l-1 to l) for node i of layer $l-1$ to node j of layer l
- every node has an associated bias:
  - $b_j^{(l)}$, bias of node j in layer l



⇒ These W and b are the parameters of the NN

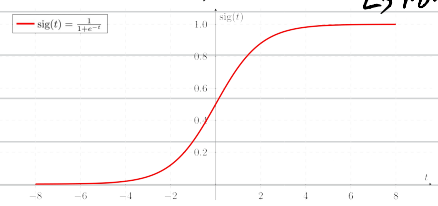
How to compute of the fct at the ℓ^{th} layer:

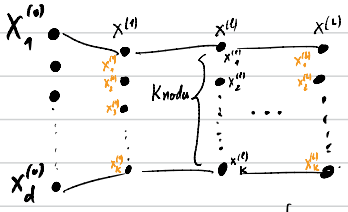
- add at each layer some Non-linearity

$$X_j^{(\ell)} = \phi\left(\sum_{i=1}^{K} X_i^{(\ell-1)} w_{ij}^{(\ell)} + b_j^{(\ell)}\right) \quad \text{with } \phi: \text{activation function}$$

↳ Non-Linear fct

i.e.: $\phi(x) = \frac{1}{1+e^{-x}}$





$\Leftrightarrow$ $f: \mathbb{R}^d \to \mathbb{R}^K$, $f(x) = \begin{pmatrix} \phi_1(x) \\ \vdots \\ \phi_K(x) \end{pmatrix}$

Rq: if K is big enough then one hidden layer is enough to learn.

In practice both K and L are large.

The NN is representing a fct from $\mathbb{R}^d$ to $\mathbb{R}^K$

→ This fct is characterized by:
- The par. (weights and bias) you will learn
- The activation fct ϕ you will pick

- # parameters of NN: $O(K^2 L)$

How powerful is NN

- if we assume that our data comes from some model:

$$y = f(x) + \varepsilon$$

- Is it possible to approx. f?
- can we learn the fct f?

Approximation result of Andrew Barron:

Let $f: \mathbb{R}^d \to \mathbb{R}$ and $\tilde{f}(w) = \int_{\mathbb{R}^d} f(x) e^{-i w^T x} dx$ and assume that $\int |w| \, |\tilde{f}(w)| dw \leq C$ (if is smooth enough)

then for all $n \geq 1$, $\exists f_n$ s.t. $f_n(x) = \sum_{j=1}^{n} c_j \phi(x^T w_j + b_j) + c_0$

s.t.

$$\int_{|x|<r} \left(f(x) - f_n(x)\right)^2 dx \leq \frac{(2Cr)^2}{n}$$

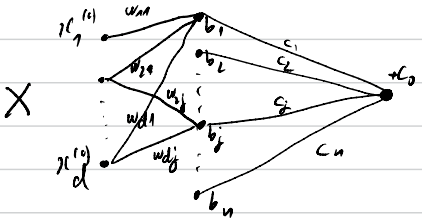
⇒ (a fct which is representable by a NN of 1 hidden layer and n nodes with ϕ act. fct)

Interpretation:
- The more nodes we allow, the smaller the error
- If C ↘, the rate ↘
- If r ↗, the rate ↗

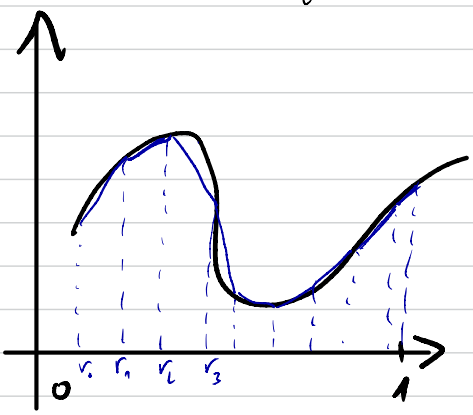
- result is in L_2-norm
- f - any "sigmoid like" act. fct

"all suff. smooth fct can be approx. by a NN by 1 hidden layer with approx. error going down as $1/n$"

$f_n(x) = \sum_{j=1}^{n} c_j \phi(x^T w_j + b_j) + c_0$



- L_∞ bound ? - f continuous in $[0,1]$



- Polynomials : Stone Weierstrass Thm
$\forall \varepsilon, \exists P, \sup_{x \in [0,1]} |f(x) - P(x)| \leq \varepsilon$
- Piecewise linear fct

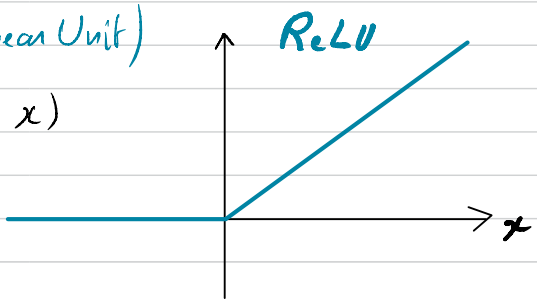
$$\Rightarrow f_n(x) = \sum_{i=1}^{N} (a_i x + b_i) \mathbb{1}_{\{r_{i-1} \leq x \leq r_i\}}$$

with $a_i r_i + b_i = a_{i+1} r_i + b_{i+1}$

→ How can we approx such fct with NN ?

ReLU (activation fct) (Rectified Linear Unit)

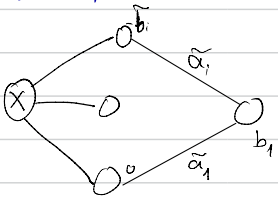
$$ReLU(x) = (x)_+ = \max(0, x)$$



now;

$$f_n(x) = \sum_{i=1}^{N} (a_i x + b_i) \mathbb{1}_{\{r_{i-1} \leq x \leq r_i\}} = \tilde{a}_1 x + \tilde{b}_1 + \sum_{i=2}^{N} \tilde{a}_i (x - \tilde{b}_i)_+$$

↳ (NN ReLU fct.)

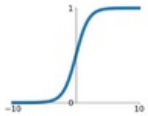


1-Hidden Layer NN

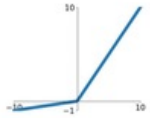
with $\tilde{a}_i = a_i - a_{i-1}$ $i > 1$
$\tilde{b}_i = r_{i-1}$ and $a_i = \sum_{j=1}^{i} \tilde{a}_j$

Activation Functions

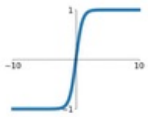
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



Leaky ReLU
$\max(0.1x, x)$

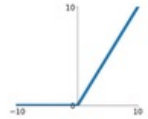


tanh
$\tanh(x)$

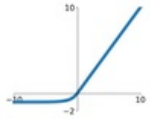


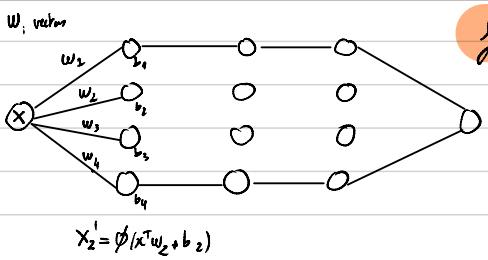
Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU
$\max(0, x)$



ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$





$f_{w,b}(x) : \mathbb{R}^d \to \mathbb{R}^K$

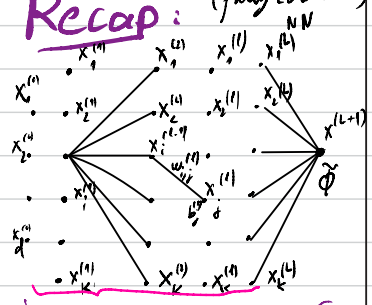
$\forall f, \exists (w,b)$ s.t $\|f - f_{w,b}\|_2 \leq \varepsilon$

$\|f - f_{w,b}\|_\infty \leq \varepsilon$

Neural Networks training (SGD) & Backpropagation

Recap:

(fully connected NN)



$$X_j^{(l)} = \phi\left(\sum_{i=1}^{K} X_i^{(l-1)} W_{ij}^{(l)} + b_j^{(l)}\right)$$

↳ Non-lin. act. fct

$f : \mathbb{R}^d \to \mathbb{R}^K$

Repro. power:
- f is smooth
- Bounded domain
- Cond. on the act. fct
- Arbi. approx. in L^2, but also worst case approx

How to Train a NN:

- Training of NN: $S_{Train} = \{(x_i, y_i)\}_{i=1}^N$ for regression

$$\mathcal{L}_N(f) = \frac{1}{N}\sum_{i=1}^{N} (y_i - f(x_i))^2$$ where f is the fct given by the NN

Task: $\min_{W_{ij}^{(l)},\, b_j^{(l)}} \mathcal{L}_n$ ($O(K^2 L)$ par.)

Rq: We can consider some Reg. los. $\tilde{\mathcal{L}}_n = \mathcal{L}_n + \text{Reg}$

Rq: $\mathcal{L}_n$ is non convex.

- SGD alg.: Pick (x_i, y_i)
  - computing gradient of $\mathcal{L}_i$: $\mathcal{L}_i = \frac{1}{2}(y_i - f(x_i))^2$
  - Take small step towards neg. grad. dir.
  - (iterate)
- Problem: too many par. $O(K^2 L)$ => Sol°: Back propagation

NN: composition of functions:

- $x^{(1)} = f_1(x_i) = f_1(x^{(0)}) = \phi\left[(W^{(1)})^T x^{(0)} + b^{(1)}\right]$

Weight matrices: $W^{(1)} \in \mathbb{R}^{d \times K}$ $\quad W_{ij}^{(1)} = w_{ij}^{(1)}$ (weights)

- $x^{(2)} = f_2(x^{(1)}) = \phi\left[(W^{(2)})^T x^{(1)} + b^{(2)}\right]$

  $W^{(2)} \in \mathbb{R}^{K \times K}$

- $x^{(l)} = f_l(x^{(l-1)}) = \phi\left[(W^{(l)})^T x^{(l-1)} + b^{(l)}\right]$

- $y = f_{L+1}(x^{(L)}) = \tilde{\phi}\left[\underset{\in \mathbb{R}^K}{w^{(L+1)}} x^{(L)} + b^{(L+1)}\right]$

$$f = f_{L+1} \circ f_L \circ f_{L-1} \circ \cdots \circ f_l \circ \cdots \circ f_2 \circ f_1$$

- complexity high → Idea: Back propagation

Backpropagation algo:

- Forward pass
- Backward pass

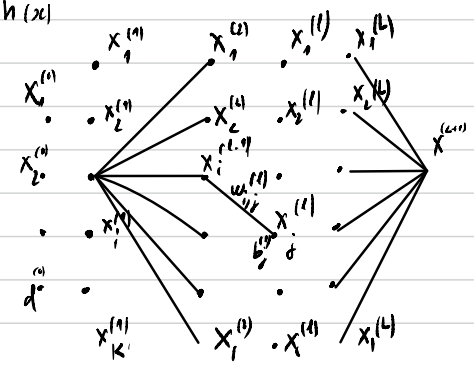
complexity: $O(K^2 L)$

$$\nabla \mathcal{L}_n = \nabla (y_n - f(x_n))^2$$

Goal: $\frac{\partial \mathcal{L}_n}{\partial w_{ij}^{(l)}}$, $\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}}$

Chain rule: $f(x) = g(h(x)) \rightarrow f'(x) = g'(h(x))\, h'(x)$

- We need to compute all values of $x^{(l)}$ (forward pass)



$$W^{(l)} = \left(w_{i,j}^{(l)} \right)$$

$$(W^{(l)})^T = \left(w_{i,j}^{(l)T} \right)$$

• Forward Pass:

$$z^{(l)} = (W^{(l)})^T x^{(l-1)} + b^{(l)}$$

$$x^{(l)} = \phi(z^{(l)})$$

initialization: $x^{(0)} = x_i \in \mathbb{R}^d$

Complexity $= O(K^2 L)$

$$z_j^{(l)} = \sum_{i=1}^{K} (w_{ij}^{(l)})^T x_i^{(l-1)} + b_j^{(l)}$$

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1K}^{(l)} \\ w_{21} & w_{12} & \cdots & w_{2K}^{(l)} \\ & & & w_{KK}^{(l)} \end{pmatrix}$$

$$z_2^{(l)} = \sum_{i=1}^{K} w_{i,2}^{(l)} x_i^{(l-1)} + b_2^{(l)}$$

$$\Rightarrow z^{(l)} = (W^{(l)})^T x^{(l-1)} + b^{(l)}$$

• Backward Pass:

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$$

$$\delta_j^{(l)} = \sum_{k=1}^{K} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \cdot \delta_k^{(l+1)}$$

We have to compute: $\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$

$$z_k^{(l+1)} = \sum_{i=1}^{K} w_{ik}^{(l+1)} x_i^{(l)} + b_k^{(l+1)} = \sum_{i=1}^{K} w_{ik}^{(l+1)} \phi(z_i^{(l)}) + b_k^{(l+1)}$$

$$\Rightarrow \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \phi'(z_j^{(l)})\, w_{jk}^{(l+1)}$$

$$\Rightarrow \delta_j^{(l)} = \sum_{k=1}^{K} \phi'(z_j^{(l)})\, w_{jk}^{(l+1)} \cdot \delta_k^{(l+1)} \Rightarrow \boxed{\delta^{(l)} = W^{(l+1)} \delta^{(l+1)} \circ \phi'(z^{(l)})}$$

$\odot$: pointwise product (scalar like)

- $\delta^{(L+1)} = \frac{\partial}{\partial z^{(L+1)}} (y_n - \tilde{\phi}(z^{(L+1)}))^2$

$$\boxed{\delta^{(L+1)} = 2\, \tilde{\phi}'(z^{(L+1)}) (\tilde{\phi}(z^{(L+1)}) - y_n)}$$

(now we go backward and compute all $\delta^{(l)}$)

Now

- $z^{(l)} = (W^{(l)})^T x^{(l-1)} + b^{(l)}$; $\Rightarrow z_j^{(l)} = \sum_{i=1}^{K} w_{ij}^{(l)} x_i^{(l-1)} + b_j^{(l)}$

$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \underbrace{\frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}}_{=1} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \quad ; \quad \frac{\partial \mathcal{L}_n}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = \underbrace{x_i^{(l-1)}}_{\text{const}} \cdot \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} = \delta_j^{(l)}$$

Summary:

- **Forward pass:** $x^{(0)} = x_i$ (one sample point)
  computing $z^{(l)} = (W^{(l)})^T x^{(l-1)} + b^{(l)}$
  $x^{(l)} = \phi(z^{(l)})$
  $\Rightarrow$ computing all values

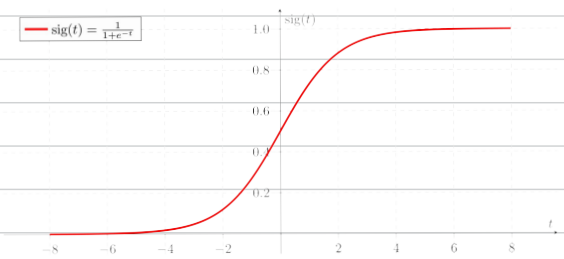
- **Backward pass:** Compute:
  - $\delta^{(l)} = W^{(l+1)} \delta^{(l+1)} \circ \phi'(z^{(l)})$
    $\subset \delta^{(L+1)}$

$$\frac{\partial \mathcal{L}_n}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} x_i^{(l-1)}$$

$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

Activation functions:

Sigmoid



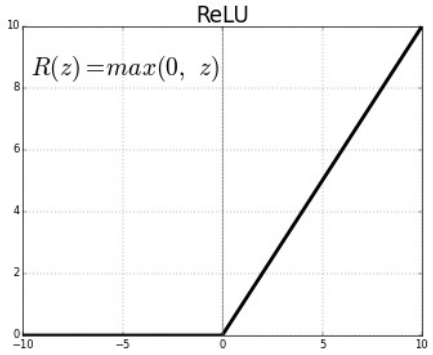
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

⊕ smooth everywhere

⊖ $|\sigma'(x)| <<< 1$ for $|x| >>> 1$ → vanishing gradient ⚠

ReLU

$$ReLU(x) = (x)_+ = \max(0, x)$$



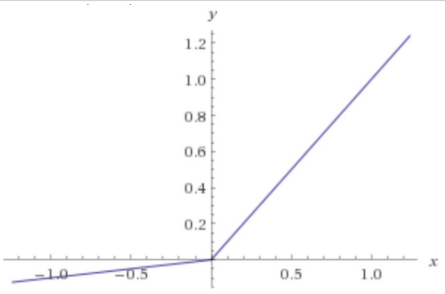
⊕ for $x \geq 0$, No vanishing gradient

⊖ Not diff. in 0
- the deriv. is 0 for $x < 0$

Sol ⇒ → Leaky ReLU

Leaky ReLU

$$LReLU = \max(\alpha x, x)$$
for α small

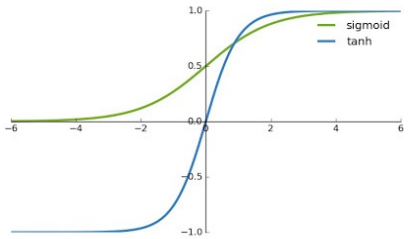


Max Out

$$f(x) = \max\left(x^T w_1 + b_1, \ldots, x^T w_k + b_k\right)$$

Tanh

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$



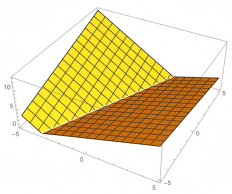


Figure 5: Maxout function with two terms, $\max\{x_1 - 0.5x_2 + 1, -2x_1 + x_2 - 2\}$.

Regularization, Data Augmentation, drop-out & Convolutional Nets

Regularization

$\mathcal{L}(w) + \frac{\lambda}{2}\|w\|_2^2$; For NN: It is interesting to Regular. the weight but not the bias

- $\nabla_w \mathcal{L} + \lambda w$; $\Rightarrow w_{k+1} = w_k - \gamma [\nabla_w \mathcal{L} + \lambda w_k]$

$$w_{k+1} = (1-\gamma\lambda)(w_k) - \gamma \nabla_w \mathcal{L}(w_k)$$

$\Rightarrow$ It is called **weight decay**

- We can also the constrained formulation: $\min_{w; \|w\| \leq r} \mathcal{L}(w)$

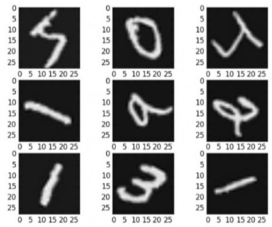
$\Rightarrow$ Can be minimized using Projected GD.

- Both formulations are very close.
  (Sometimes equivalent for cvx fcts)
  - $\rightarrow$ Penalization is simple to implement
  - $\rightarrow$ the Constraint form is easier to interpret.

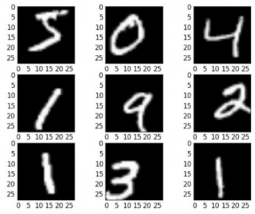
$\Rightarrow$ Regularization can be a good idea, but only on the weights for NN.

Data augmentation

i.e (MNIST data), $S_{Train} = \{(x, y)\}$



rotation



Shifted

Transformation: $\tau : \mathbb{R}^d \to \mathbb{R}^d$
s.t $y(x) = y(\tau(x))$

$$S = S_{Train} \cup \{(\tau(x_i), y_i)_{i=1}^{M}\}$$

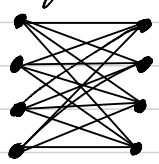
$\rightarrow$ This guarantees that the pred. will be invariant to τ.

$\Rightarrow$ Can be seen as Regularization

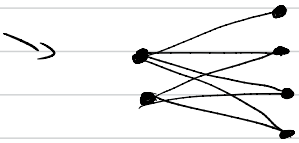
$\hookrightarrow$ Restricting the set of fct we look for

Dropout:

- Experimentally it helps avoid overfitting
- It is also a technique of ensemble averaging
- Original version on the node



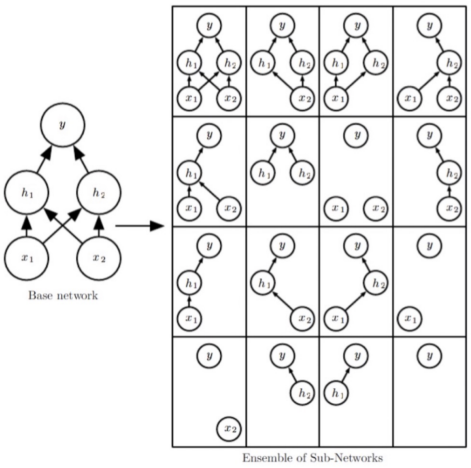
- let $p_j^{(l)}$: prob of keeping the node j at layer l
  → flip indep. biased coin and this will give a residual net.



It is important to rescale the weights you are keeping to get the same behaviour in Expectation

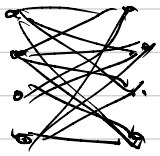
Why?

- It gives a more robust way of learning
- It is very close to train in // many NNs and predict in avg.
  (ensemble averaging)
  (bagging)
  ↳ ↘ variance of pred



Convolutional Network

So far, we have considered fully connected NN:



$O(K^2L)$ parameters

(do we really need all these edges?)

→ Is it possible to get better perf. with sparse set of edges?

Convolution:

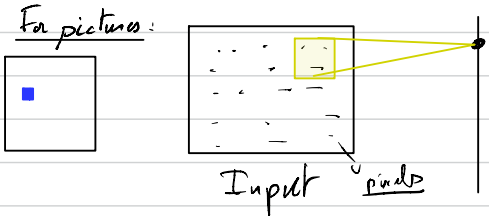
$h = f * g$

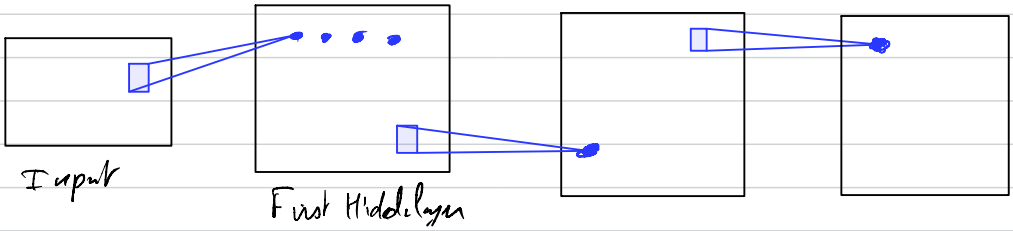
→ by taking the conv. we are doing local averaging

$h[n] = \sum_k [f[k] \cdot g[n-k]]$

↑ filter, kernel

- For pictures:





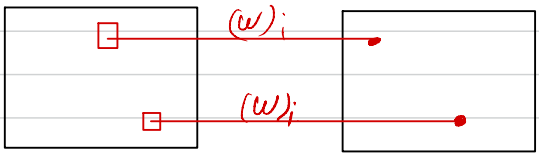
$\square_p$ of size p : local averaging window

$O(K^2L)$ parameters to $O(Kp^2)$ param.

different filters.

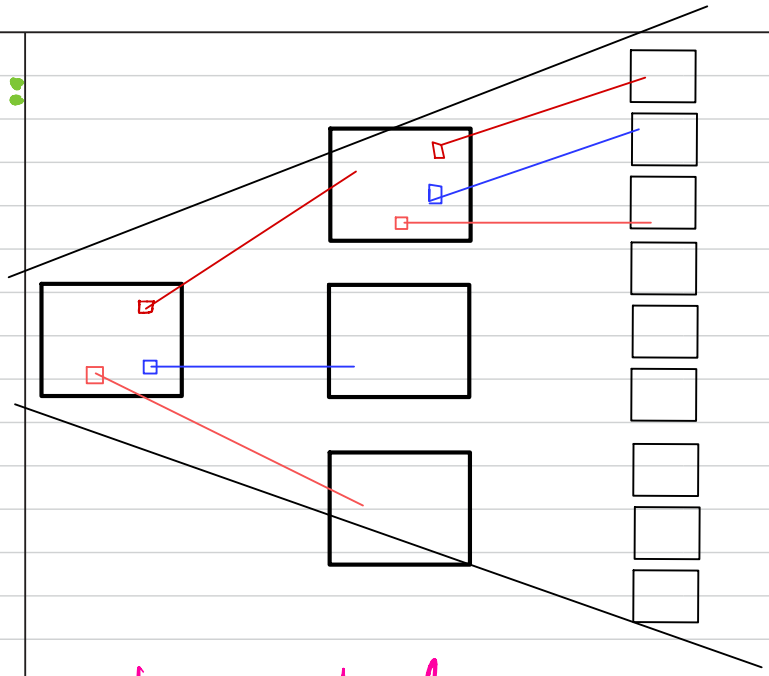
→ often we would use the same filters on all regions.

Weight sharing:



$O(pL)$ parameters

Channels:



Pyramidal form

Back propagation alg.:

- you can do Back prop. whatever the architecture.

- With weight sharing?

$f(x,y,z)$; $\nabla f = \begin{pmatrix} \partial f/\partial x \\ \partial f/\partial y \\ \partial f/\partial z \end{pmatrix} \in \mathbb{R}^3$

"weight sharing":

$g(x,y) = f(x,y,x)$ How to compute ∇g? ; $\nabla g = \begin{pmatrix} \partial g/\partial x \\ \partial g/\partial y \end{pmatrix}$

$$\nabla g = \begin{pmatrix} \partial f/\partial x + \partial f/\partial z \\ \partial f/\partial y \end{pmatrix}$$

$$\frac{\partial g}{\partial x} = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial x} + \frac{\partial f}{\partial x_2}\underbrace{\frac{\partial x_2}{\partial x}}_{=0} + \frac{\partial f}{\partial x_3}\underbrace{\frac{\partial x_3}{\partial x}}_{=1} = \frac{\partial f}{\partial x_1} + \frac{\partial f}{\partial x_3}$$

Conclusion: We Run Back prop with weight sharing, and then sum the derivatives for the weights you sharing.