Lecture 1

Notation (I will try to be consistent!)

w_i: weight from input i
w_{ij}: weight from i to j
i: input index
j: hidden unit index
k: output unit index (not always!)
a_j: weighted sum of inputs to unit j
g():activation function

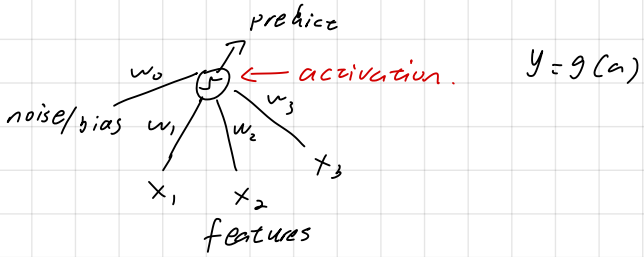
d: number of inputs
c: number of outputs
C_k: k^{th} output category
N: number of patterns
n: pattern label
t: target label
x: input data
y: output activation
z: hidden unit activation

- Unsupervised. (A model of the data)
- Supervised. (A mapping from input ↔ targets)
- Reinforcement. (learn from mistakes).
- Imitation.

Regression:

There is a underlying line corrupted by noise.

ex)



$y = g(a)$

Linear regression:

$$\begin{bmatrix} 1 & x_1^1 & \cdots & x_d^1 \\ 1 & & & \\ 1 & & & x_d^N \end{bmatrix} W \simeq \text{targets}$$

↓

by minimizes the Sum Squared Error (SSE).

$$SSE = \frac{1}{2} \sum_{n=1}^{N} \left(\underbrace{\sum_{j=0}^{d} w_j x_j^n}_{\text{predicted}} - t^n \right)^2$$

$\frac{1}{\text{number of examples/patterns}}$.

Solving for analytical solution: ... $\vec{w} = (X^T X)^{-1} X^T \vec{t}$.

Gradient descent: ...

$y(x) = g(a)$ ← weighted sum of inputs $a^n = w^T x^n$.

↑ activation

Partial derivative of error wrt weights

$$\frac{\partial MSE}{\partial w_i} = \dots = \frac{1}{N} \sum_{n=1}^{N} (y^n - t^n) x_i$$

$(y^n - t^n) = -\delta^n$

so, $w_i = w_i + \alpha \frac{1}{N} \sum_{n=1}^{N} \delta^n x_i^n$.

update rule.

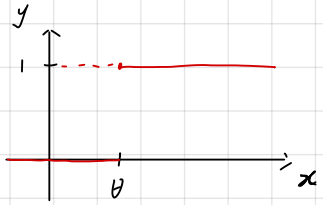
$= w_i + \alpha \frac{1}{N} \sum_{n=1}^{N} (t^n - y^n) x_i^n$.

Batch Learning

Online Learning ⟶ SGD, change weight immediately, for each data point. (train as data comes in stream).

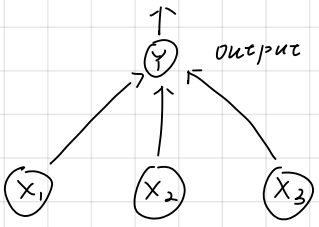
Perceptrons

Trying to fit a function of



(a classifier).

Binary threshold unit:



$$y = \begin{cases} 1 & \text{if } \sum_i w_i x_i \geq \theta \\ 0 & \text{otherwise.} \end{cases}$$

↑ sum of weight.

(regression).

To make a neurally-inspired machine that could categorize inputs and learn to do this from example.

Perceptron Learning (perceptron convergence procedure).

ex)

Input	Target
0 0	0
0 1	1
1 0	1
1 1	1

learn a mapping

Output activation:

$$y = \begin{cases} 1 & \text{if } \sum_{i=0}^{d} w_i x_i \geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad x_0 \triangleq 1$$

Error Correction Learning (only corrected it when it is wrong).

- If output = 0 and should be 0, lower weights
- If output = 1 and should be 1, raise weight

⇓

Form a learning rule:

$$w_i = w_i + \alpha (t - y) x_i$$

or $$w_i = w_i + \alpha \delta x_i$$

α is the learning rate

$(\delta = t - y)$.

difference

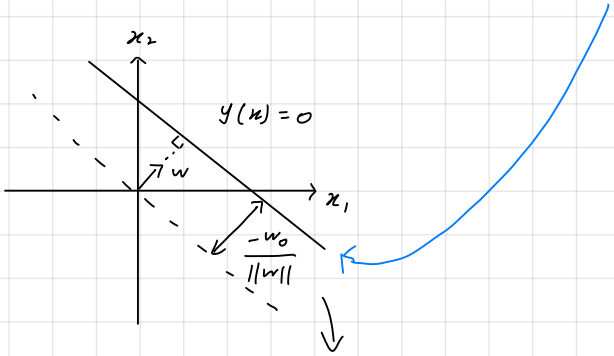
(Delta Rule)

Guarantee: anything a perceptron can compute, can learn to compute and will converge.

We call $y(x) = 0$ the decision boundary where output changes from 0 to 1.

⇓

is a $(d-1)$ dimensional hyperplane in d-dimensional input space.



$$y(x) = 0$$
$$\Rightarrow w_1 x_1 + w_2 x_2 + w_0 = 0 \qquad (2\text{-}D \text{ case}).$$
$$\Rightarrow x_2 = -(w_1/w_2) x_1 - w_0/w_2$$

line form

For two points on decision boundary, x^A, x^B.

$$y(x^A) = y(x^B) = 0$$
$$\Rightarrow y(x^A) - y(x^B) = 0$$
$$\Rightarrow w^T x^A + w_0 - w^T x^B - w_0 = 0$$
$$\Rightarrow w^T (x^A - x^B) = 0$$

$\Rightarrow$ $\vec{w}$ and vector $(x^A - x^B)$ is orthogonal.

The distance l is then:

$$l = \frac{w^T x}{\|w\|} = \frac{-w_0}{\|w\|}, \text{ where } w = (w_1 \dots w_d).$$

Perceptron as a linear discriminant:

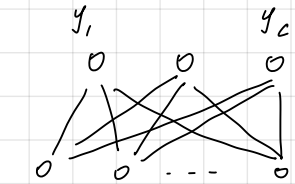
- a two-class discriminat is one such that

$$y(x) = a = w^T x$$ ← no activation.

x in C_1 if $a \geq 0$

else x in C_2.

- For multiple one pat



x is assigned C_k if $k = \arg\max_j y_j(x)$

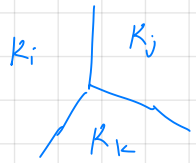
$\Downarrow$

in particular,

each region is convex set

$\Rightarrow x^A, x^B$ in R_i

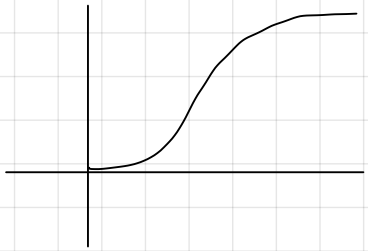
implies $ax^A + (1-a)x^B$ in R_i



Decision boundary is thus

$y_i(x) = y_j(x)$ for C_i and C_j, ...

Logistic Regression



sigmoid function

estimate of prob. of C_1 or C_2.

(still a linear classifier).

$$y(x) = g(w^T x + w_0)$$

$$g(x) = \frac{1}{1 + e^{-x}}, \text{ logistic function as activation.}$$

Derivation:

if we assume our data is gaussian distribution (actually not in many cases).

the data distribution $p(x|C_1) = \frac{1}{\sqrt{2\pi\,\sigma^2}} e^{-\frac{1}{2}\left(\frac{x-\mu_1}{\sigma}\right)^2}$

$$p(x|C_2) = \frac{1}{\sqrt{2\pi\,\sigma^2}} e^{-\frac{1}{2}\left(\frac{x-\mu_2}{\sigma}\right)^2}$$

By Bayes rule $p(C_1|x) = \frac{p(x|C_1)\,p(C_1)}{p(x)}$

$$p(C_2|x) = \frac{p(x|C_2)\,p(C_2)}{p(x)}$$

then, $p(C_1|x) = \frac{p(x|C_1)\,p(C_1)}{p(x|C_1)\,p(C_1) + p(x|C_2)\,p(C_2)}$

Denote $p(C_1|x) = \frac{A}{A+B} = \frac{1}{1+B/A} = \frac{1}{1+e^{\ln(B/A)}} = \frac{1}{1+e^{-\ln(A/B)}}$

$$= \frac{1}{1+e^{-\ln\left(\frac{p(x|C_1)\,p(C_1)}{p(x|C_2)\,p(C_2)}\right)}} = \frac{1}{1+e^{-a}} \quad \text{(sigmoid of } a\text{)}$$

↑ prob class 1 follows a sigmoid as a function of the log ratio of the prob. of class C_1 to the prob. of class C_2. (the posterior)

called the logit.

So $p(C_1|x) = \frac{1}{1+e^{-a}} = g(a)$, where $a = \ln\left(\frac{p(x|C_1)\,p(C_1)}{p(x|C_2)\,p(C_2)}\right)$.

can be written as:

$a = wx + w_0$ where

$w = \frac{\mu_1 - \mu_2}{\sigma}$ and $w_0 = -\frac{1}{2}\frac{\mu_1^2}{\sigma} + \frac{1}{2}\frac{\mu^2}{\sigma} + \ln\frac{p(C_1)}{p(C_2)}$

Most data do not follow gaussian distribution, no analytical solution and we have to learn the weights

⇓ gradient descent

If we use MSE (bad objective function for demonstration):

$$w_i = w_i - \alpha \frac{\partial MSE}{\partial w_i}$$

$$\downarrow = \frac{\partial \frac{1}{2N}\sum_{i=1}^{N}(t^n - y^n)^2}{\partial w_i} = \dots = \frac{1}{N}\sum_{i=1}^{N}(t-y)\left(-\frac{\partial y}{\partial w_i}\right)$$

$$= \frac{1}{N}\sum_{i=1}^{N}(t-y)\left(-\frac{\partial g(a)}{a}\frac{\partial a}{\partial w_i}\right)$$

chain rule for activation.

$$= \frac{1}{N} \sum_{i=1}^{N} (t-y)\left(-g'(a)\frac{\partial a}{\partial w_i}\right)$$

$$= \frac{1}{N} \sum_{i=1}^{N} (t-y)\,(-g'(a))\,x_i$$

$$w_i = w_i + \frac{\alpha}{N} \sum_{i=1}^{N} (t^n - y^n)\, g'(a)\, x_i^n$$

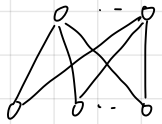
since for $g(a) = \frac{1}{1+e^{-a}}$

$$= w_i + \alpha \frac{1}{N} \sum_{i=1}^{N} (t^n - y^n)\, g(a^n)(1-g(a^n))\, x_i^n$$

$g'(a) = g(a)(1-g(a))$

Use cross-entropy as objective function.

Generalization of logistic regression to multiple output.



output

To turn output to prob.

Softmax

$$a_k = \sum_{j=0}^{d} w_{jk}\, x_j = \vec{w}_k^{\,T} \vec{x}.$$

where $x_0 = 1$, $w_{0k} = \text{bias}$.

$$\vec{a} = W^T \vec{x}$$

$$y_k = g(a_k) = \frac{e^{a_k}}{\sum_{j=1}^{c} e^{a_j}} > 0.$$

output sum to 1

$$\sum_{k=1}^{c} y_k = \sum_{k=1}^{c} \frac{e^{a_k}}{\sum_{j=1}^{c} e^{a_j}} = 1$$

Softmax as regression: (can train with GD)

$$a_k = \sum_{j=0}^{d} w_{jk}\, x_j = w_k^T x \quad (x_0 = 1,\ w_0 = \text{bias}).$$

$$p(C_k \mid x) = y_k = g(a_k) = \frac{e^{a_k}}{\sum_{j=1}^{c} e^{a_j}}$$

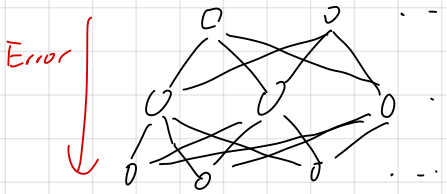
More Activation: $y = g(a) = \max(a, 0)$: ReLU

$y = g(a) = \tanh(a)$: tanh

...

· Create new feature from old feature for learning (old fashion).

Foward Propagation & Backward Propogation :



Chain rule of gradient

$z_i = g(a_i) \xrightarrow{w_{ij}} a_j$

$z_i = x_i \xrightarrow{w_{ij}} a_j$

Back prop. Learning : $w_{ij} = w_{ij} - \frac{\partial J}{\partial w_{ij}}$

(z_i mean output of hidden unit or input).

J is objective.

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}} \rightarrow = \frac{\partial \sum_{l=0}^{h} w_{lj} z_l}{\partial w_{ij}} = \sum_{l=0}^{d} \frac{\partial w_{lj} z_l}{\partial w_{ij}} = z_l$$

every term is 0 except $i = l$

$$= \frac{\partial J}{\partial a_j} z_i$$

so $z_i = g(a_i) \xrightarrow{w_{ij}} a_j$

· Define $\delta_j = -\frac{\partial J}{\partial a_j} \left(= -\frac{\partial X_{ent}}{\partial a} = -\frac{\partial X_{ent}}{\partial y} \frac{\partial y}{\partial a} = (t - y) \right)$ logistic regression

$$w_{ij} \leftarrow w_{ij} - \frac{\partial J}{\partial w_{ij}} = w_{ij} - \frac{\partial J}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}} x_i = w_{ij} + (t_j - y_j) x_i$$

so, we have $\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}} = -\delta_j z_i$

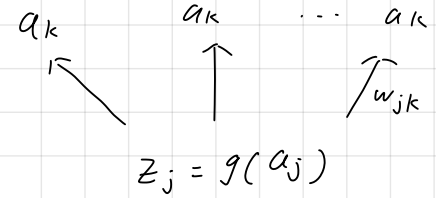
By delta rule, $w_{ij} \leftarrow w_{ij} - \frac{\partial J}{\partial w_{ij}} = w_{ij} + \delta_j z_i$

(weight changes in proportion to the input and the delta at that connection).

For output unit, delta rule is $w_{ij} \leftarrow w_{ij} + (t_j - y_j) z_i$

For hidden unit, $\frac{\partial J}{\partial a_j} = \sum_k \frac{\partial J}{\partial a_k} \frac{\partial a_k}{\partial a_j}$

idea:



So, derivation:

$$\frac{\partial I}{\partial a_j} = \sum_k \frac{\partial I}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

$$= -\sum_k \delta_k \frac{\partial a_k}{\partial a_j}$$

$$= -\sum_k \delta_k \frac{\partial a_k}{\partial z_j} \frac{\partial z_j}{\partial a_j}$$

j independent of k

$$= -\frac{\partial z_j}{\partial a_j} \sum_k \delta_k \frac{\partial a_k}{\partial z_j}$$ def of a_k

$$= -\frac{\partial z_j}{\partial a_j} \sum_k \delta_k \sum_i \frac{\partial w_{ik} z_i}{\partial z_j}$$ every term is 0 except $i=j$

$$= -\frac{\partial z_j}{\partial a_j} \sum_k \delta_k w_{jk}$$

since $\frac{\partial z_j}{\partial a_j} = \frac{\partial g(a_j)}{\partial a_j} = g'(a_j)$ derivative of activation.

$$= -g'(a_j) \sum_k \delta_k w_{jk}$$

Thus, we derive that $\frac{\partial I}{\partial a_j} = -g'(a_j) \sum_k \delta_k w_{jk}$, if j is hidden unit.

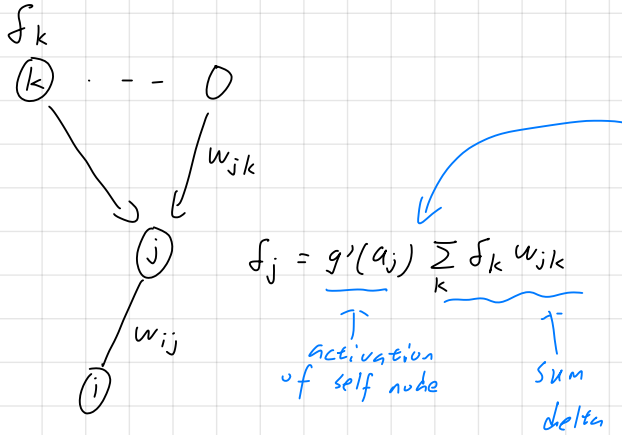
Since delta is $-\frac{\partial I}{\partial a_j}$, $\delta_j = g'(a_j) \sum_k \delta_k w_{jk}$.

Backprop Learning: $w_{ij} = w_{ij} - \alpha \frac{\partial I}{\partial w_{ij}} = w_{ij} + \alpha \delta_j z_j$

in which
$$\begin{cases} \delta_j = (t_j - y_j) & \text{if } j \text{ is an output unit.} \\ \delta_j = g'(a_j) \sum_k \delta_k w_{jk} & \text{if } j \text{ is a hidden unit} \end{cases}$$

(recursive definition of delta).

For hidden unit, the idea is



Back prop is linear operation.

Vanishing / Exploding gradient

As result of chain rule of activation.

(online) Back prop :

- present pattern to network, feed forward until output.
- compare δ (usually $t-y$)
- back prop δ through network. Every unit has a delta.
- update w_{ij}

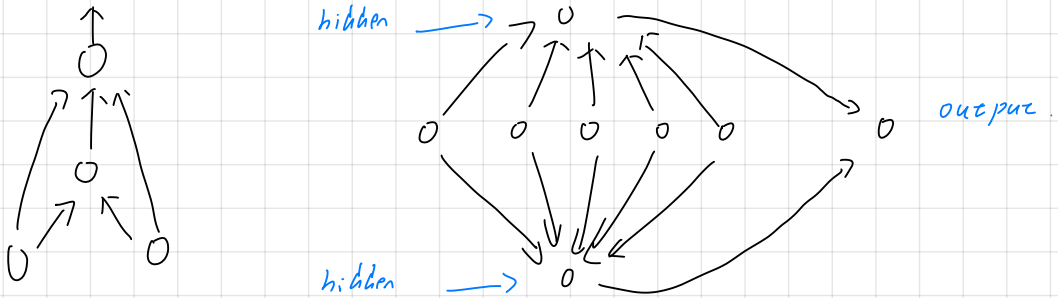
$$w_{ij} \leftarrow w_{ij} + \alpha \delta_j z_j$$

Depend on initial condition, back prop gives many solution.
(highly non-convex network).

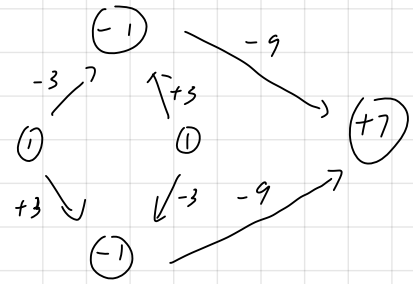
Learn internal representation through hidden units.

ex) XOR

symmetry around center



ex)



ex) NetTalk Architecture

- input uses a one-hot
- averaging vector in tree structure (hidden unit).

ex) Hinton's Family tree

Objective Function & MLE.

Maximum Likelihood.

want w that maximize $P(w|D) = \frac{P(D|w)P(w)}{P(D)}$ (likelihood · prior; $P(D)$: normalizing constant)

$$\Downarrow$$

$$\max P(w|D) = \max P(D|w)$$

(maximize likelihood)

How we model the data

$$L = \prod_{n=1}^{N} P(x^n)$$

Find parameter by $\underset{\mu, \sigma}{\text{argmax}}\ L \Rightarrow \ln L = \ln \prod_{n=1}^{N} P(x^n)$ (standard distribution example)

$$\Downarrow$$

$$\underset{i}{\text{argmin}} - \ln \prod_{i=1}^{n} P(x^n)$$

$= \dots$

$= \dots$

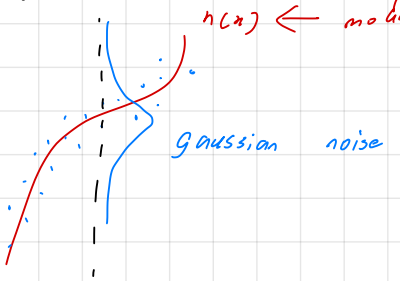
= leads to setting μ to empirical mean.

Modeling input-output data:

$$L = \prod_{n=1}^{N} P(x^n, t^n) = \prod_{n=1}^{N} P(t^n|x^n) P(x^n)$$

$$-\ln L = -\sum_{n=1}^{N} (\ln P(t^n|x^n) + \ln P(x^n)).$$

- Assume gaussian noise



$$P(t^n|x^n) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(t^n - h(x^n))^2}{2\sigma^2}}$$

$$\Rightarrow$$

$$\Downarrow$$

$$L = \frac{1}{\sqrt{2\pi\sigma^2}^N} \prod_{p=1}^{N} e^{-\frac{(t^n - y(x^n; w))^2}{2\sigma^2}} \quad \text{(for all data)}$$

The error (negative log likelihood).

$$E = -\ln \frac{1}{\sqrt{2\pi\sigma^2}^N} \prod_{p=1}^{N} e^{-\frac{(t^n - y(x^n; w))^2}{2\sigma^2}}$$

$$= \frac{1}{2\sigma^2} \sum_{n=1}^{N} (t^n - y(x^n, w))^2 + \ln\left(\sqrt{2\pi\sigma^2}^N\right)$$

(constant)

$\Downarrow$

sum squared loss!

Cross-Entropy:

$y(x^n) = P(C_1|x^n)$

Modeling two categories is like modeling coin flip

$$\Downarrow$$

Bernoulli distribution

$$\Downarrow$$

$$p(t^n|x^n) = (y^n)^{t^n}(1-y^n)^{(1-t^n)}$$

if $t^n = 1$, $p(t^n|x^n) = y^n(1-y^n)^{(1-1)} = y^n$.

So, the likelihood: $L = \prod_{n=1}^{N} p(t^n|x^n) = \prod_{n=1}^{N} (y^n)^{t^n}(1-y^n)^{(1-t^n)}$

$$-\ln L = -\ln \prod_{n=1}^{N} (y^n)^{t^n}(1-y^n)^{(1-t^n)}$$

$$= -\sum_{n=1}^{N} \ln (y^n)^{t^n}(1-y^n)^{(1-t^n)}$$

$$X_{ent} = -\sum_{n=1}^{N} t^n \ln(y^n) + (1-t^n)\ln(1-y^n)$$

(cross-entropy error)

Entropy: a measure of information in a message about a random variable.

$-p(x)\log p(x)$

Entropy of distribution: $-\sum_{n=1}^{N} p(x^n)\ln p(x^n)$.

↓

Cross entropy: $-\sum_{n=1}^{N} p(t^n)\ln p(y^n)$.

Multinomial Regression:

$p(C_k|x^n) = y_k(x^n)$, $t_k = 1$ if from category k, 0 otherwise

$$\Rightarrow p(t^n|x^n) = \prod_{k=1}^{n} (y_k^n)^{t_k^n}$$

$$\Rightarrow \prod_{n=1}^{N} p(t^n|x^n) = \prod_{n=1}^{N}\prod_{k=1}^{C} (y_k^n)^{t_k^n} = \cdots = -\sum_{n=1}^{N}\sum_{k=1}^{C} t_k^n \ln y_k^n$$

Siamese Neural Network:

↓

"trained to move output closer or farther apart."

if x_1, x_2 from same category, minimize dist y^1 and y^2.

" different " maximize "

The loss function: $L(w, Y, \vec{x}_1, \vec{x}_2) = (1-Y)\frac{1}{2}(D_w)^2 + (Y)\frac{1}{2}\{\max(0, m - D_w)\}^2$.

↳ $D_w(\vec{x}_1, \vec{x}_2) = \|G_w(\vec{x}_1) - G_w(\vec{x}_2)\|_2$

↑ output vector of network on stimulus x.

- D is distance between outputs of two networks
- m is a margin.
- Y is 1 for a "different" pair 0 for "same" pair.

Generalization

Deal With Overfitting ?

- More data → Data Augmentation:
  ex) take different crops of the data or resize it.
- Regularization.
  ↓ reduce model complexity ($J = E + \lambda C$)
  - L_2 → minimize $\|w\|_2^2$, weight smaller in proportion to its size.
  - L_1 → minimize $|w|$, weight smaller at a constant rate.
  - Minimize $C = \|w\|^2/(\|w\|^2+1)$, penalize big weight less while penalizing small weights more.
- Dropout

=> probabilistically turn off some fraction of hidden units.

- Early stopping

...

- Add noise to inputs / weight / hidden unit activations.
  ↓ make model more robust to perturbations.

Neural Nets Trick:

SGD:

1. get one example
2. compute gradient
3. change weight
4. If seen all example, reshuffle

Batch Learning

1. get one example
2. compute gradient
3. add to running average
4. If seen all example, change weight

mini-batch learning.
→ by running a small batch of data each time changing the weight.
↓ efficient due to matmul.

Shuffling data before partition to mini-batch
↓ should have diverse data representation.

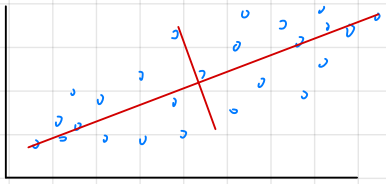
PCA on input:

Idea: ex) if all positive input, $w_{ij} = w_{ij} + \alpha \delta_j x_i$
(δ_j: same sign for all weights.)

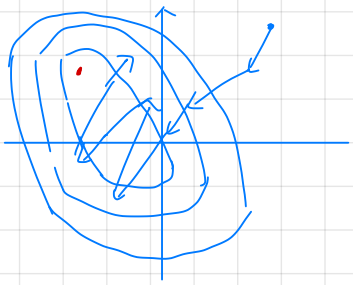
if x_i's are all positive, then the weight changes are all + or −.

↓ optimization

ex) highly correlated variables



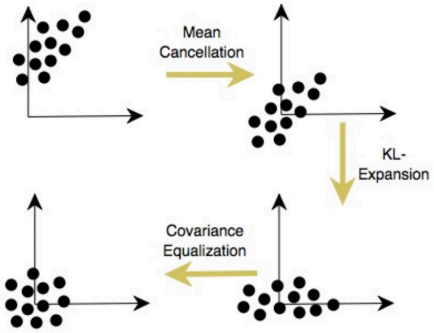
⇒ get two different information.



ex) different scale of input

PCA ⟶
- shifts mean of input variable to 0.
- decorrelates the input.
- throw away dimension with smaller eigenvalues (dimensionality reduction).
- divide by standard deviation, make them all of equal size.

PCA vs. Z-scoring:



- Z-scoring shifts the mean of the input variables to be 0 – not all positive or negative!
- Z-scoring does ***not*** decorrelate the inputs – because it is applied to each variable independently.
- And...you ***cannot*** throw away the dimensions with the smallest eigenvalues –because you don't compute the eigenvectors.
- Z-scoring divides by the standard deviation, making all of the variables 0 mean and unit standard deviation.
- So the variables are all of the same size

Different activation function:

problem: standard sigmoid makes all input to next layer positive.

recommended sigmoid: $f(x) = 1.7159 * \tanh(0.667x)$.

- $f(+/-1) = +/-1$ (if input has unit variance, output has unit variance).

Weight Initialization:

Weight cannot initialize to 0

if initialize to 0 => δ's of the hidden will all be the same.

⇓

weighs from each input be the same.

Weight initialization coordinated with:

- Input normalization
- choice of sigmoid.

Suppose x_i mean 0, unit variance. w_{ij} mean 0, on average, STD of $a_j = x w$ will be:

$$Var(XW) = Var(x)\,Var(W) + Var(x)\,(E(w))^2 + Var(w)\,(E(x))^2.$$

$$= Var(w).$$

So, STD of $a_j = \sqrt{Var(w)} = \sqrt{\sum_i (w_{ij})^2}$

To let $\sigma(a_j) = 1$, we set w's to be $\left(\frac{1}{m}\right)^{1/2}$.

where m is the fan-in to node j.

assume input have been normalized, and sigmoid is $f(x) = 1.7159 \tanh(2/3\, x)$.

Intuitively, if hidden unit has big fan-in, small changes on many of its incoming weights can cause learning to overshoot.

=> smaller incoming weight when fan-in is big, and vice-versa.

Batch Normalization

idea: normalizes all inputs and inputs to each hidden layer throughout the network.

(on a per unit basis, over each mini-batch).

1. Z-score each input variable over mini-batch.

$$\hat{a}_i = \frac{a_i - u_i}{\sqrt{\sigma_i^2 + \epsilon}}$$ where u_i, σ_i^2 are mean and variance of a_i

↑

net input to a unit are any layer of the network.

2. Allow network learn to undo if necessary:

$$\tilde{a}_i = \gamma_i \hat{a}_i + \beta_i$$

learnable parameter

Input: Values of a over a mini-batch: $\mathcal{B} = \{a_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{\tilde{a}_i = BN_{\gamma,\beta}(a_i)\}$

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

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

$$\hat{a}_i \leftarrow \frac{a_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$\tilde{a}_i \leftarrow \gamma \hat{a}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(a_i) \qquad \text{// scale and shift}$$

Momentum

weight change at time t.

$$\Delta w(t) = \gamma \Delta w(t-1) - \alpha \frac{\partial E}{\partial w}(t)$$

previous information.

(sum of previous weight change and the new gradient).

direction of steepest descent does not point at the minimum.

How it behaves:

$$\Delta w(\rho) = \frac{1}{1-\gamma}\left(-\alpha \frac{\partial E}{\partial w}\right)$$ (by solving recurrence relation).

Nesterov Method:

- first make a big jump in direction of previous accumulated gradient.
- then measure the gradient where you end up and make a correction.

Apaptive learning rate:

$$\Delta w_{ij} = -\varepsilon\, g_{ij}\, \frac{\partial E}{\partial w_{ij}}$$

- ε: global learning rate
- g_{ij}: local adjustements.

$g_{ij}(t) = g_{ij}(t-1) + 0.05$ ↑ $\left(\frac{\partial E}{\partial w_{ij}}(t)\,\frac{\partial E}{\partial w_{ij}}(t-1)\right) > 0$

- increase if gradient$_{(ij)}$ does not change sign.
- decrease otherwise.

↓

$g_{ij}(t) = g_{ij}(t-1) \cdot 0.95$.

⇒ big gain decay rapidly when oscillation starts.

Apaptive learning rate only deals with axis-aligned effects.

↓

Apaptive learning rate affect individual weights, which change the learning rate along that axis.

Resilient Backprop (Rprop):

Only for full-batch learning:

- increase step size for a weight multiplicatively (times 1.2) if the signs of its last two gradients agree
- otherwise, decrease the step size multiplicatively (times 0.5).

Does not work for mini-batch learning:

↓

Rprop is to use the gradient but also divide by the size of the gradient ($\text{sign}(x) = \frac{x}{|x|}$), problem is that we divide by a different number for each mini-batch.

↓

Solution — RMSprop:

Keep a moving average of squared gradient for each weight.

$$\text{MeanSquare}(w,t) = 0.9\,\text{MeanSquare}(w,t-1) + 0.1\left(\partial E/\partial w(t)\right)^2$$

Then, divide the gradient by $\sqrt{\text{MeanSquare}(w,t)}$.

Convolutional Network (CNN).

Problem: • Large images size (1140 x 648)

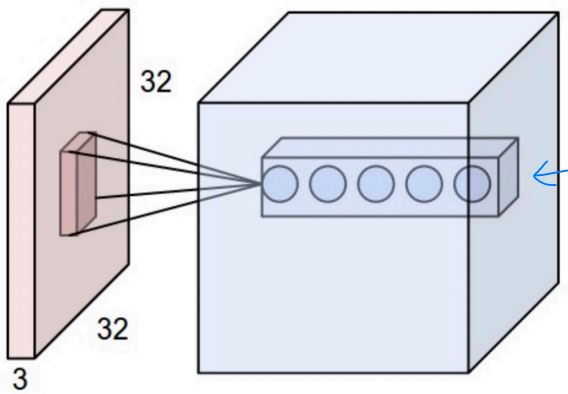
Properties of Real World Image:
- Nearby pixel correlate with nearby pixel: locality
- Statistics of pixel are relatively uniform: stationary statistics
- object doesn't depend on its location in image.: translation invariance
- object made of pares. Compositionality

Why CNN useful to solve above problem?

⇓

- "Locality": small, local receptive field and learned features (kernel).
- "Stationary statistics": replicated across the image
  (if a feature is useful in one place, it's useful in others).
- "Translation invariance": spatial pooling.
- Objects are made of pares: receptive fields see larger deeper in the net.

Receptive field: size of visual world that innervates the neuron
Visual feature in visual world that most stimulate neuron.



Convolutional Block.

5 neurons
each one compute a different features.

↓

Computation is exactly the same
(inner product of weights and pink patch).

Activation Maps

pooling ⟶  ⇒ .

2×2 set of response to 1 number (Translation Invariance).
(1/4 size)

Convolution leads to shift equivariance.

(shifting number in different location does not affect the feature map).

Original image Feature map for 1 feature

If we move the 3 over… the features just shift over

This is shift *equivariance*

Image Net Data

Alex Net 8 layers

↓

VGG 19 layers

↓

Google Net 22 layers

↓

Res Net 152 layers

Overview of Convnets.

- Feed-forward
  - Convolve input.
  - Non-linearity (rectified linear)
  - Pooling (local max).
- Supervised
- Convolution filter trained by back prop.

Images ⟶ Convolution ⟶ Non-linear ⟶ Pooling ⟶ Feature Maps
(filter)

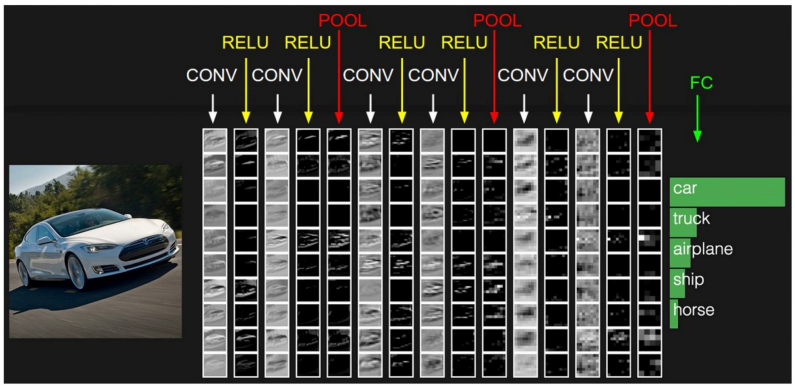
Notes From Stanford CS231n. CNN

ex)

INPUT → CONV (12 filters) → RELU → POOL → FC (softmax).

[32×32×3] [32×32×12] [32×32×12] [16×16×12] [1×1×10]

	CONV	RELU	POOL	FC
contain parameters	X			X
hyperparameter	X		X	X



Convolutional Layer

- each filter will produce a 2-D activation map.
- Stacking activation map along the depth dimension produce the output volume.
- connect each neuron to only a local region of the input volume (receptive field).
- the extend of connectivity along the depth axis is always equal to the depth of input volume.

ex) input volume [32×32×3]

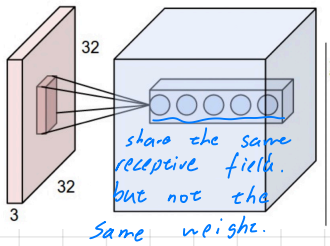
receptive field (filter size) [5×5×3].

then a total of 5×5×3 = 75 weights.

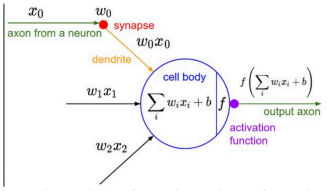
ex) input volume [16 x 16 x 20].

receptive field [3 x 3 x 20].

then a total of 3 x 3 x 20 = 180 weights.



share the same receptive field, but not the same weight.



- W : input volume size
- F : receptive field size
- S : stride
- P : amount of zero padding

Then output volume size is

$$\frac{(W - F + 2P)}{S} + 1$$

choice of stride is important to ensure that the above division yields an integer.

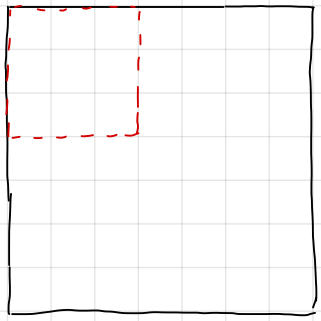
Thus, to have input and output the same size, we set zero padding

$$P = (F - 1)/2$$

Three hyperparameters to control output volume

- Depth : # of filters (K)
- Stride : # of jumps when filter moves
- Zero-padding : whether to pad input volume with 0 around the border.

3x3 filter



7x7

$$\Rightarrow \frac{7 - 3 + 2 \cdot 0}{1} + 1$$

$$= 4 + 1 = 5$$

$\Rightarrow$ 5 x 5 output.

stride = 1, pad = 0

Parameter sharing.

Since we are using filter striding over the images, the # of weights will be

F x (# of filter) x (depth of output).

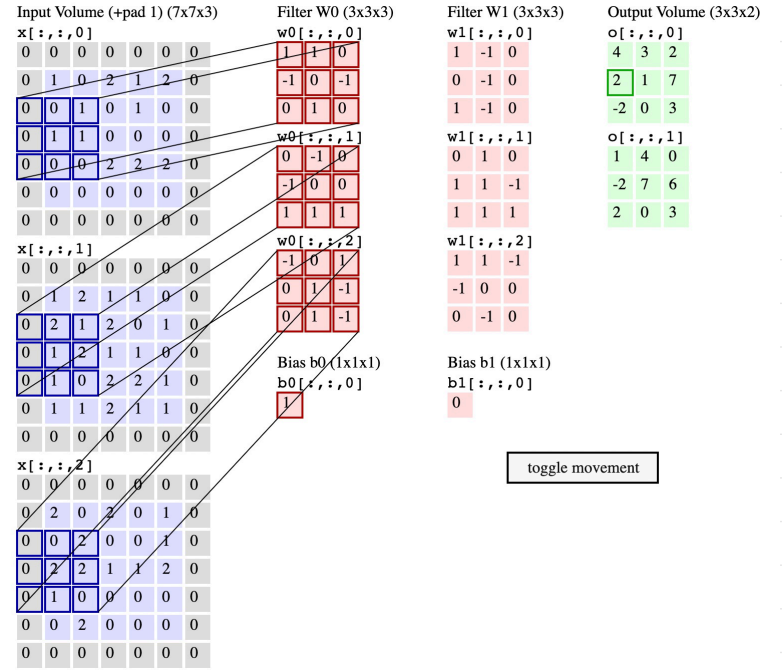
this is guarantee by translationally invariance, if detecting a feature is important at some location, it should be useful at some other location as well.

If learning different feature on different location of images is necessary, then it's common to relax parameter sharing schema

⇓

Locally-Connected Layer.

Conv a complete visualization

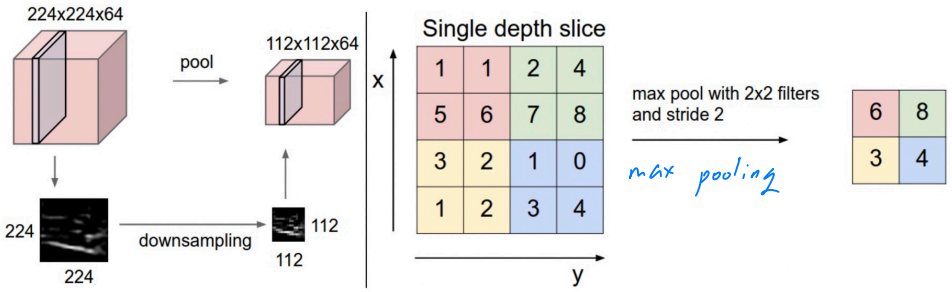


Pooling layer

- progressively reduce the spatial size of the representation to reduce the parameters.
  - W_1, H_1, D_1 representing input volume size.
  - For pooling layer, it requires two hyperparameters:
    - spatial extent F
    - stride S
  - Then output volume: $W_2 = (W_1 - F)/S + 1$

$$H_2 = (H_1 - F)/S + 1$$

$D_2 = D_1$



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. Left: In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. Right: The most common downsampling operation is max, giving rise to max pooling, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

ConvNet Architectures

Most common form:

$$INPUT \rightarrow [[CONV \rightarrow RELU] \times N \rightarrow POOL?] \times M \rightarrow [FC \rightarrow RELU] \times K \rightarrow FC$$

(POOL? = optional pooling)

Filtering

- (Weight learned during training)
- each unit looks at local window.
- same feature computed everywhere.

Non-linearity:

- ReLU (applied per-pixel).
- output = max(0, input).

Spatial Pooling

- max is best
- sum or average used less
- applied to each filter layer separately.
- Stride: how far to slide over each image

role: invariance to small transformation.
larger receptive field.

Component of each layer:

pixel / feature => filter with learned dictionary => Non-linearity

=> Spatial local max pooling

=> Output features.

- Batch normalization.

How to choose architecture?

- Hyperparameter
  - ex) # layers, # feature maps, params of feature maps.

- Cross-validation
- Grid-search / random-search.

Best practice:

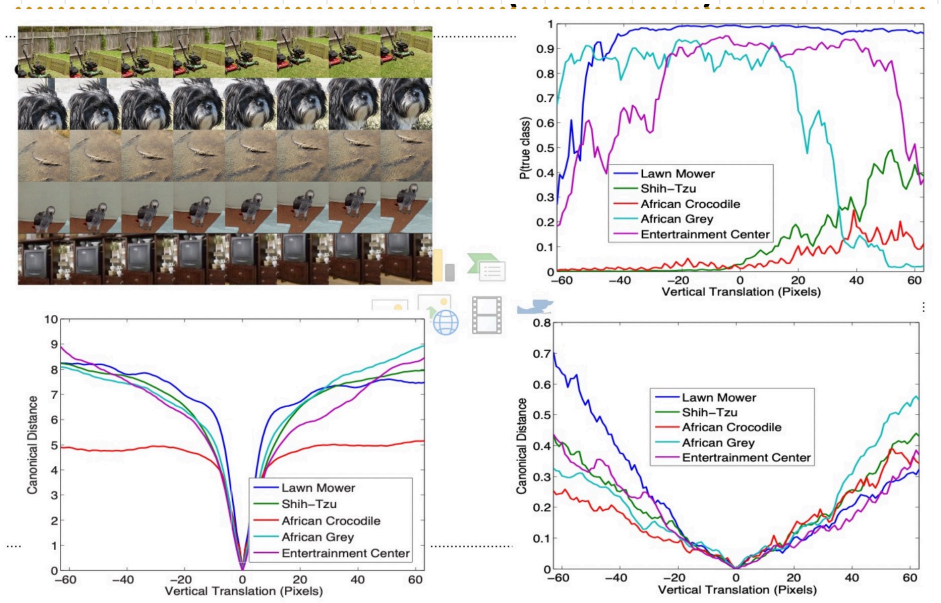
- width: 3
- stride: 1
- zero-padding: 1

Trick: Stack convolutions, then use max pooling.

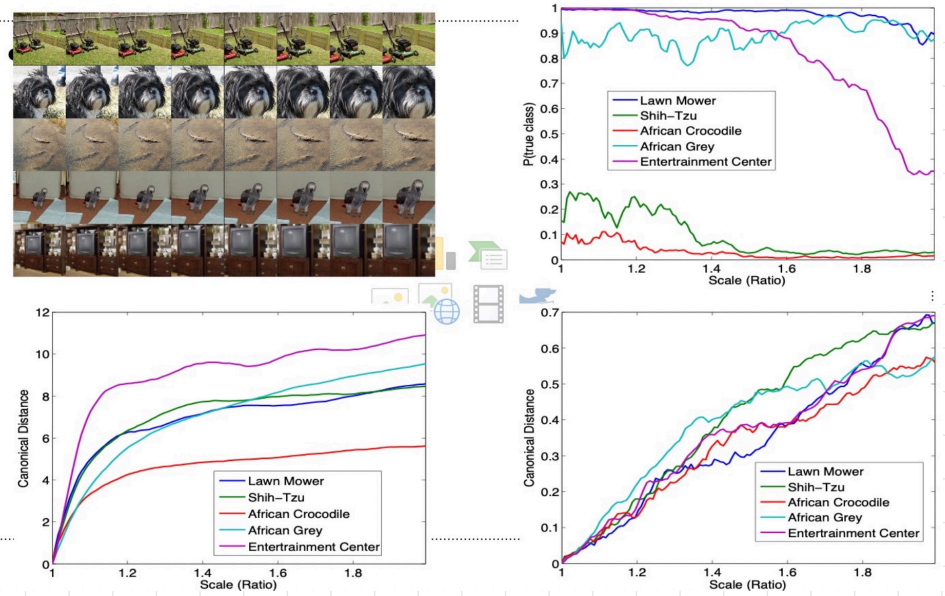
conv → conv → max pooling

Invariance Abilities

Translation (Vertical):



Scale Invariance:



Deep Model:

idea: design one modular structure, then just add more of them

3×3 conv layer better than single 7×7 b/c non-linearity.

- Network-in-network:
  add two per-pixel fully connected layers before pooling.

Conv → 1×1 Conv → 1×1 Conv

NiN Block

- global average pooling.
  replace final fully connected layer: one feature map per category.
- What is 1×1 Conv?

[56 × 56 × 64] ——(1×1 Conv with 32 filters)——> [56 × 56 × 32].

(each filter size (1×1×64), performs a 64-dim dot product).

⇒ essentially a dot product of filter weights and 64 channels

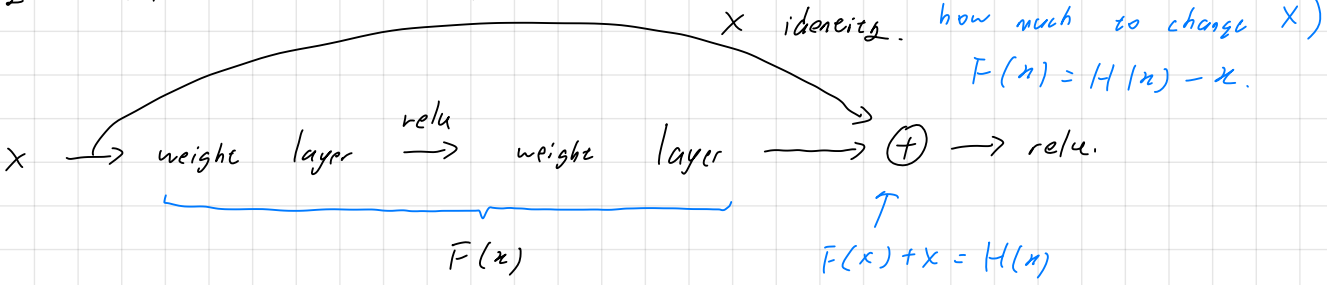
advantage: complete control of computation!!!

Residual Network

idea: introduce "pass through" into each layer.

⇓

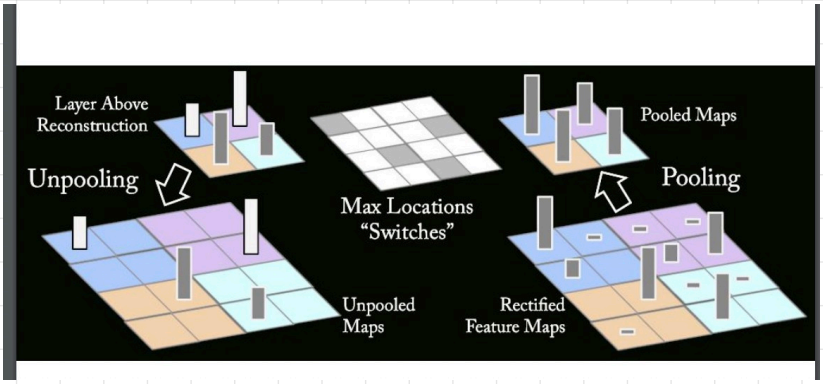
only residual needs to be learned.



(the network learn how much to change X)

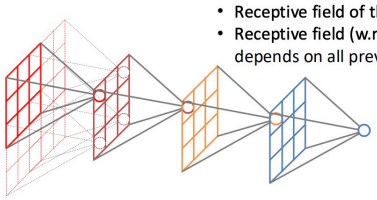
F(x) = H(x) − x.

Visualization for higher layer network:



Max unpooling

- max activation from feature map associated with each filter.
- deconvnet to project back to pixel space.
- pooling switch.

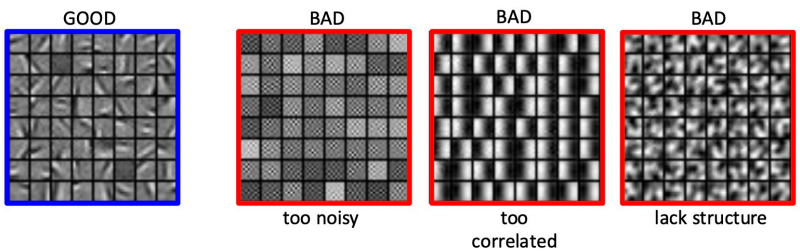


- Receptive field of the first layer is the filter size
- Receptive field (w.r.t. input image) of a deeper layer depends on all previous layers' filter size and strides

- Correspondence between a feature map pixel and an image pixel is not unique
- Map a feature map pixel to the center of the receptive field on the image in the SPP-net paper

Training Tips

- Annealing learning rate.
  (-> start large, slowly reduce.
- Visualize feature map.



Good training: learned filters exhibit structure and are uncorrelated.

- Training diverges -> decrease learning rate
- Network underperforming -> make net larger.

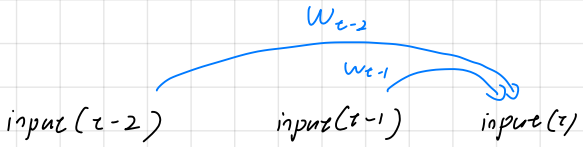
Recurrent Network:

Represent time in network?

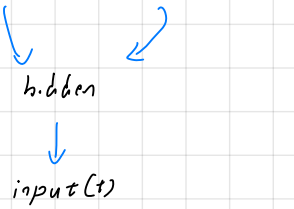
- map time into space
- map time into state of the network.

Time → Space.

- Auto regressive:

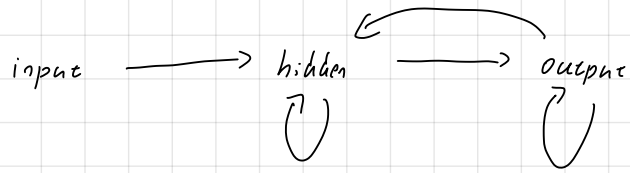


- Feed-forward net:

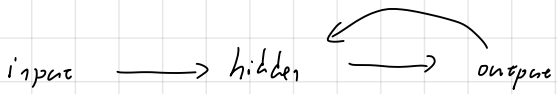


Time → State

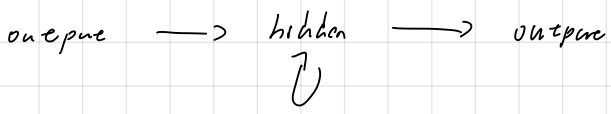
- use activation memory:
  new things are processed based upon what has come before.
  (network state).



=> ① Jordan net

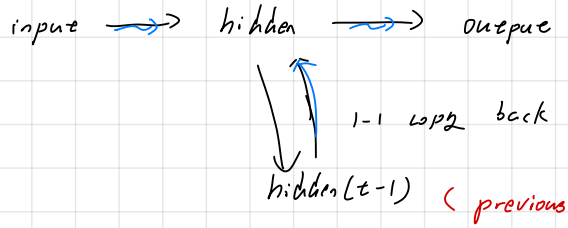


② Elman net (simple recurrent net).

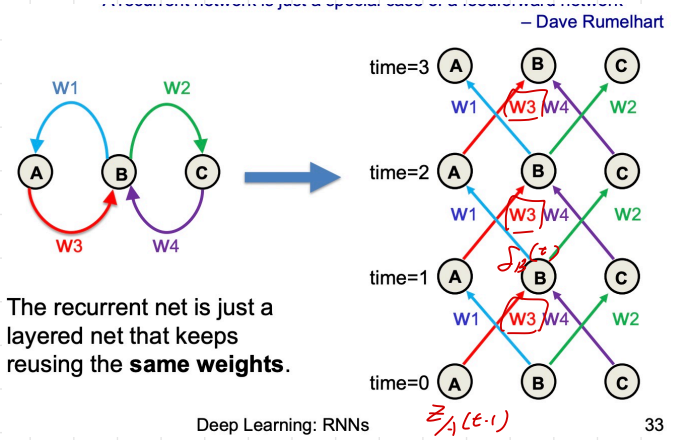


equivalently,

(one-step backprop through time).



Unrolling networks in time.



- calculate total gradient over time as an average and then modify the weights.

↓

- propagate activation forward in time ($z_A(t)$, $z_B(t)$ ...)
- propagate delta backward.
- update weights,

ex) $w_3 \leftarrow = \alpha \frac{1}{3} \sum_{t=1}^{3} \delta_B(t) z_A(t-1)$

Hard to train b/c backward pass is linear.

problem of exploding / vanishing gradient: → • switch logistic to ReLU.

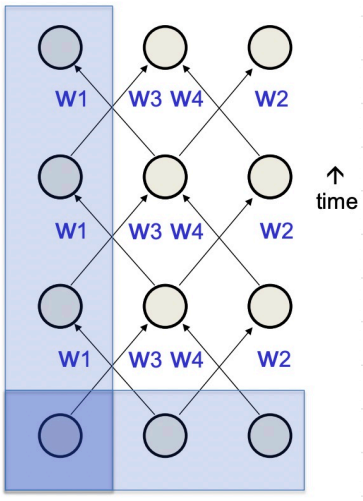
- weights small, gradient shrinks exponentially.
- weights big, gradient grow exponentially.

In RNN, we can:

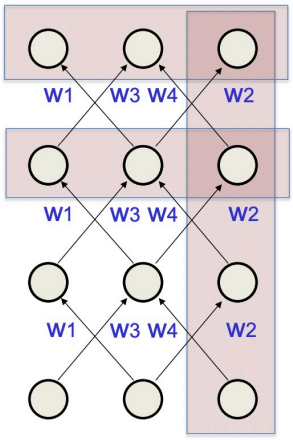
- exploding gradient.
  limit length of gradient vector (gradient clipping)
- vanishing gradient:
  skip connection

We can specify inputs in several ways:

- Specify the initial states of all the units.
- Specify the initial states of a subset of the units (the rest are hiddens)
- Specify the states of the same subset of the units *at every time step.* I.e., we have *inputs* at every time step.



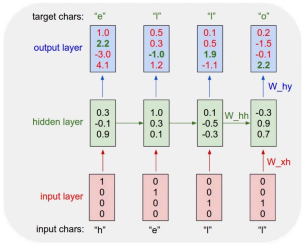
- We can specify targets in several ways:
  - Specify desired final activities of all the units
  - Specify desired activities of all units for the last few steps
    - Good for learning attractors
    - It is easy to add in extra error derivatives as we backpropagate.
  - Specify the desired activity of a subset of the units.
    - The other units are input or hidden units.



RNN for language generation:

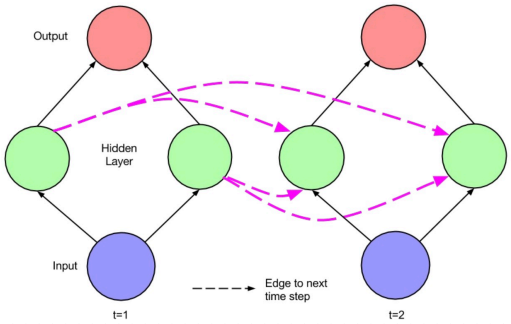
A Recurrent Network for Language Generation

- Commonly a *character-level* language model
- Start with a huge corpus of text and then ask it to model the probability distribution of the next character given the prior sequence of characters
- Simple example shown where our dictionary is only {'h', 'e', 'l', 'o'}
- Note we have shared weights: All wts with the same color are the same



TIME ->

Recurrent Net (Unfolded)

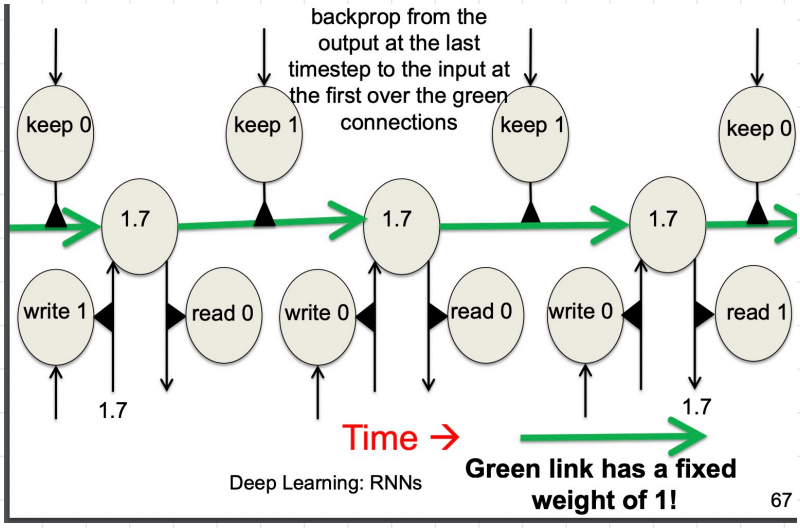


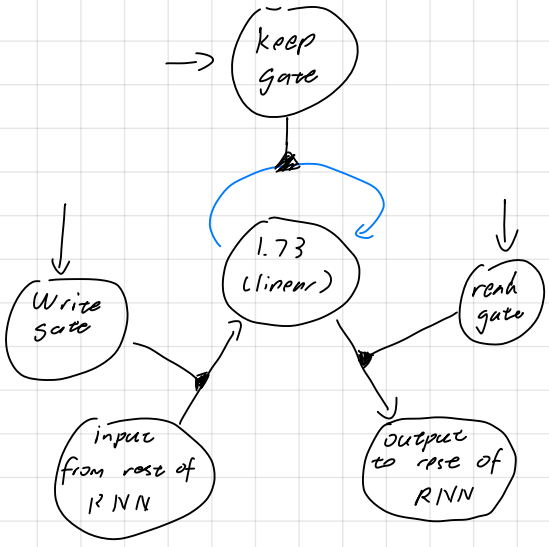
$$h^{(t)} = \sigma(W_{hx} x^{(t)} + W_{hh} h^{(t-1)} + b_h)$$

$$\hat{y}^{(t)} = \text{softmax}(W_{yh} h^{(t)} + b_y)$$

LSTM when

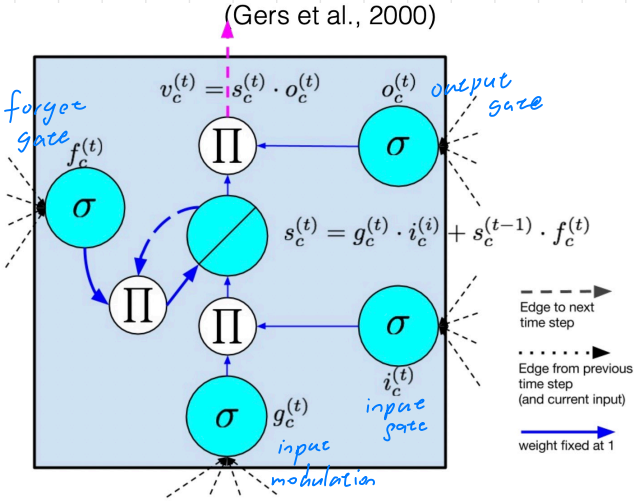
- info goes into cell when "write" gate on.
- info stays in cell when "keep" gate on.
- info read from cell when "read" gate on.

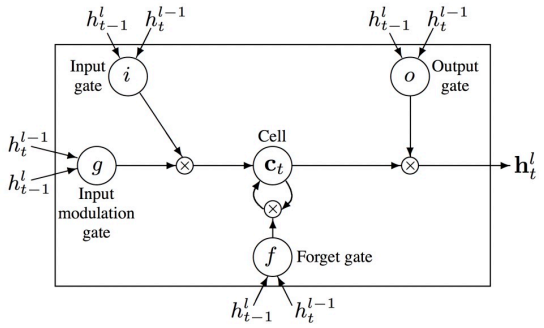




(LSTM).

(Gers et al., 2000)





$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} T_{2n,4n} \begin{pmatrix} \mathbf{D}(h_t^{l-1}) \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$

LSTM Forward Pass:

$$g^{(t)} = \phi(W_{gx} x^{(t)} + W_{ih} h^{(t-1)} + b_g)$$ input modulation

$$i^{(t)} = \sigma(W_{ix} x^{(t)} + W_{ih} h^{(t-1)} + b_i)$$ input gate

$$f^{(t)} = \sigma(W_{fx} x^{(t)} + W_{fh} h^{(t-1)} + b_f)$$ forget gate

$$o^{(t)} = \sigma(W_{ox} x^{(t)} + W_{oh} h^{(t-1)} + b_o)$$ output gate.

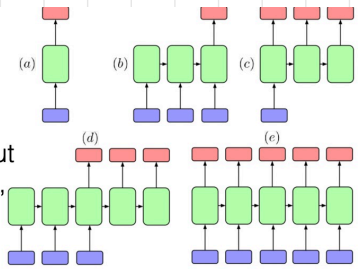
$$s^{(t)} = g^{(t)} \odot i^{(t)} + s^{(t-1)} \odot f^{(t)}$$

$$h^{(t)} = s^{(t)} \odot o^{(t)}$$

input modulation; control forget

KNN architecture

a) One to one (not recurrent!)
b) Many to one
c) One to many
d) Many in, then many out
e) Simultaneous many in, many out



KNN as generative model

- generative text model ...
- Seq to Seq
- Image to text
- ...

Memory is a matrix of linear neurons.

$$M = \begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}$$

(row as a "work" of memory).

index row as $i = 1, 2, 3$.

$$r_t \leftarrow \sum_i w_t(i) M_t(i)$$

Transformer Network

Map time into space using shared weights

↓

communicate via attention

⇓

when you focus on particular aspect of your env.

- Overt attention: move your eyes toward something
- Covert attention: direct your attention to something without moving your eyes

How to direct attention in Neural Net?

↓

Softmax + multiplicative connections. (Content-based Addressing).

$$w_t^c(i) \leftarrow \frac{\exp(\beta_t K[k_t, M_t(i)])}{\sum_j \exp(\beta_t K[k_t, M_t(j)])}$$

ex) $M = \begin{matrix} 3 & 1 & 4 & 1 \\ 1 & 2 & 3 & 4 \\ 2 & 7 & 2 & 8 \end{matrix}$

key $k = (3, 1, 0, 0)$.

attention weighs: $w^c = (0.998, 0.001, 0.001)$.

read: $r_t \leftarrow \sum_i w_t(i) M_t(i)$

so, $r_t = (2.997, 1.007, 3.997, 1.010)$

$\approx (3, 1, 4, 1)$

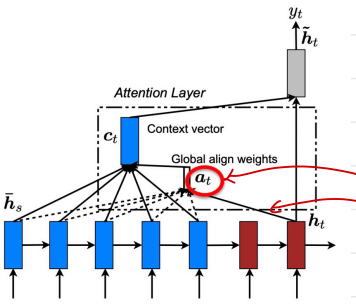
- key vector was computed by the network.
  (dynamic, network know what it was looking for).
- Attention weight filtered the memory row via multiplication.
- Match b/t key and memory doesn't have to be perfect.

Why transformer? ——> decoder needs different information at different time steps.

- train much faster
- process each position in parallel.

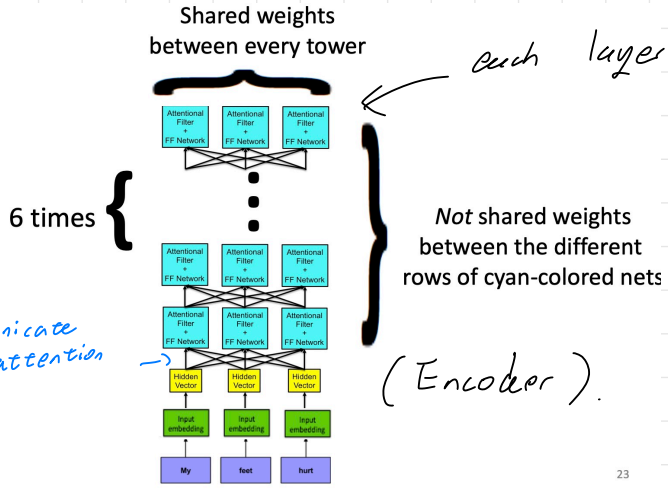
Attention-based methods ——> problem with recurrence.

No parallelization since hidden states all depend on previous hidden states.



attention vector computed dynamically used to gate the previous hidden state.

Transformer Network:



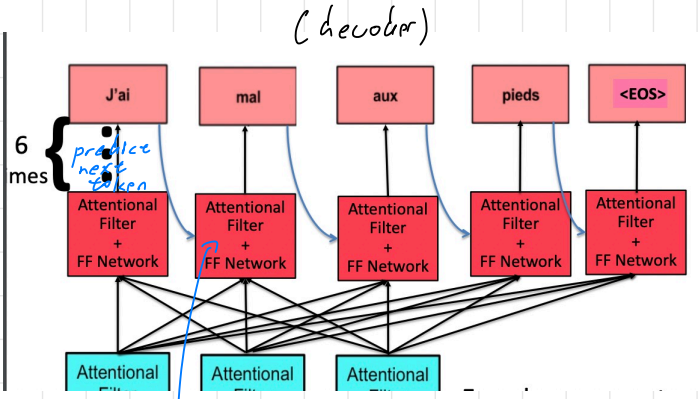
each layer computes a different function.

communicate via attention

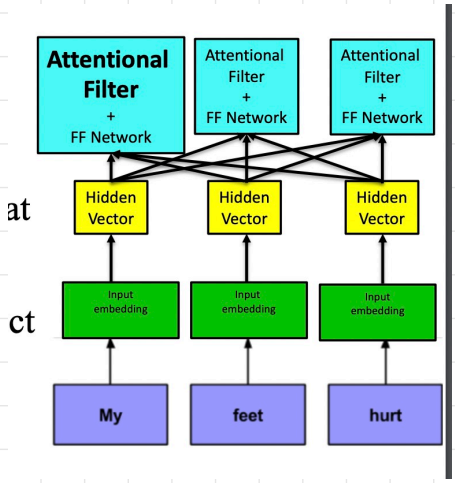
(Encoder).

23

a copy of network for every input position.



retrieves info from all encoder steps.



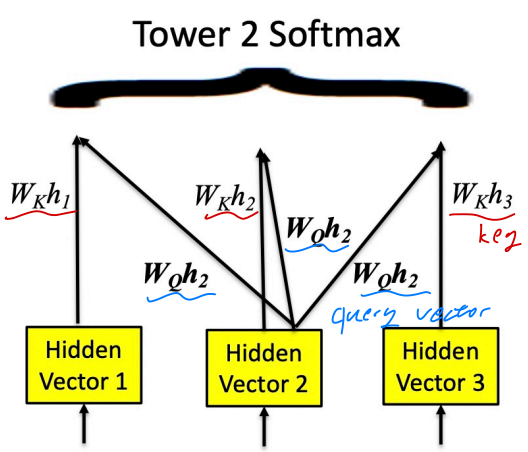
inner produce attention.

- each tower generates a vector that represents what it is looking for. (Query)
- each tower generates a vector that represents what information it has. (key).

Query + key

⇒ allow one tower to select information from other towers.

(dynamically re-wire itself and communicate between word meaning to construct sentence meaning).



(query vector) $\cdot$ $(W_k h_i)$, ← the ith hidden vector

$\Rightarrow$ softmax $\left((W_k h_1)^T (W_Q h_2),\ (W_k h_2)^T (W_Q h_2),\ \dots\ (W_k h_N)^T (W_Q h_2) \right) / \sqrt{d}$.

key vector

softmax "interest" vector over N hidden vector. ↑ tower 2 query dot product all tower's key.

idea: the most similar key and query vector will be a peak in softmax, selecting where to obtain value.

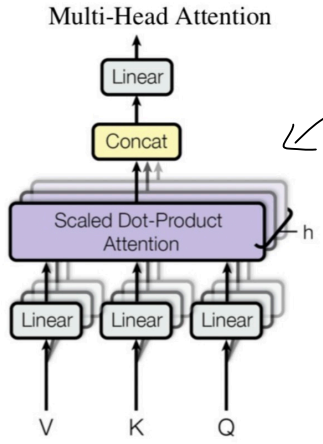
In matrix notation:

$W_{key} H = K$ (key)

$W_{query} H = Q$ (query)

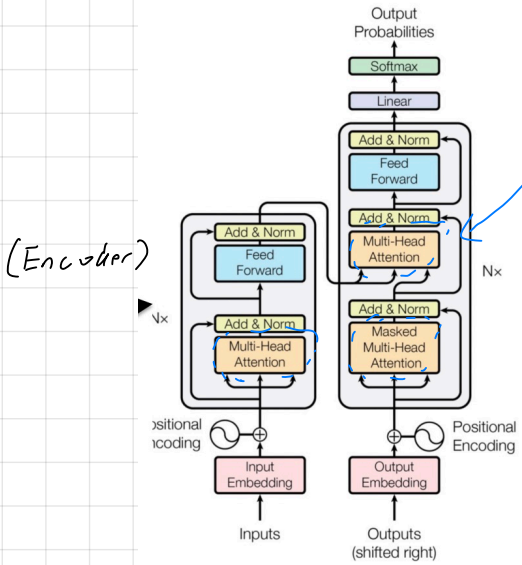
$W_{value} H = V$ (value)

result: softmax $(K Q) \times V$



multi-head attention allowing the network to pay attention to multiple locations in the input at once.

Full Transformer Architecture



(Encoder)

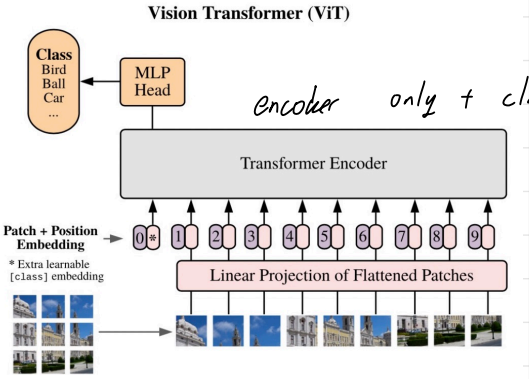
(Decoder).

Generative Pre-Training 2.0 (GPT 2.0)

↳ trained to predict next word in a massive dataset.

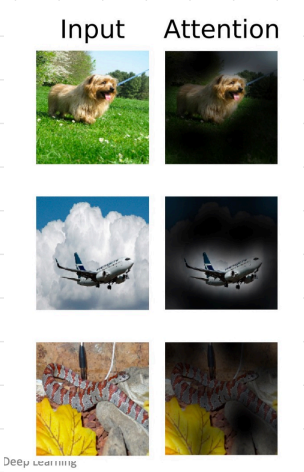
Image Transformer

- Takes in patches of image (no convolution)



encoder only + classifier.

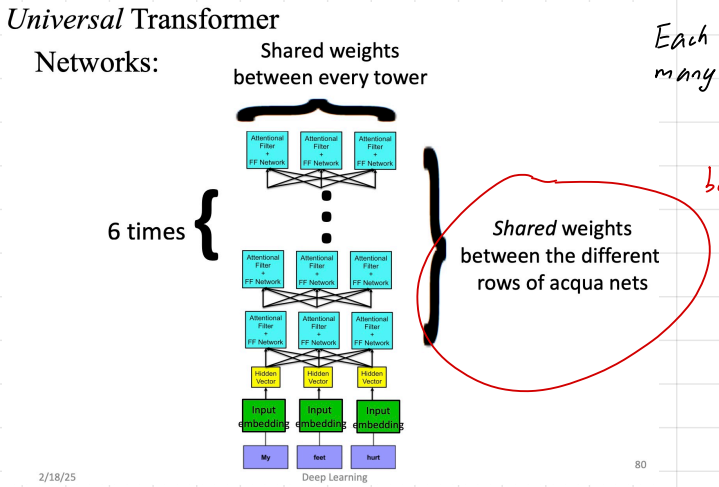
what it is learning?



Universal Transformer Network:

↳ shared weights between different layers

(unrolled recurrent network - same computation is performed over and over).

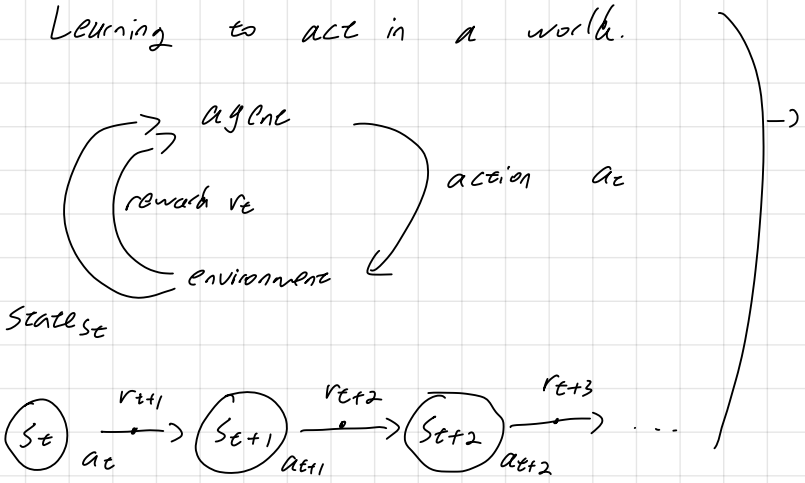


Each tower can decide adaptively how many iterations to run.

between layers.

RL, Deep Nets and Atari

Learning to act in a world.



Agent learns a policy π

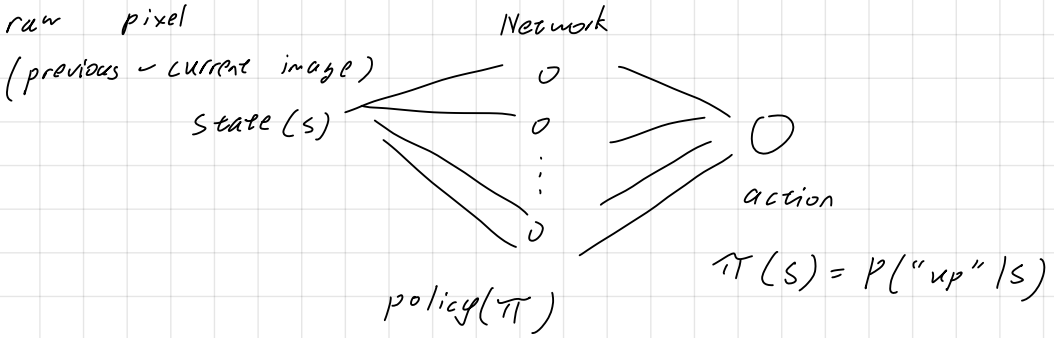
$\pi(s) = P(a_1, a_2, \cdots a_n | s)$

function from states to action probabilities.

↓

"raise prob. of action a_i when it was a good move, lower it if it was a bad move".

ex) PONG Policy network



At the end, we use sign of the reward on the gradient and apply it to every gradient we produced over the whole game.

=> win -> encourage actions that led to win
lose -> discourage actions that led to loss.

Policy gradient:

① At each step of play, sample from the softmax distribution at output:

$\pi(s) = (0.1, 0.02, 0.6, \cdots, 0.01)$

↑
network, mapping from states to probabilities of action

(2) Treat the sample as "teacher"

$t = (0, 0, 1, 0, \cdots, 0)$

for state/action pair.

(3) Compute weight change, add then to a running average of weight change. (some form of "gradient")

(4) Multiply weight change by the sign of reward.

(5) Change the weights after one game.

TD-Gammon and Alpha Go

goal of learning:

maximizing long-term expected reward.

Let r_t be reward at time t.

↳ maximize $R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$

$0 \leq \gamma \leq 1$ is the discount rate.

↓

ensures that expected reward converges.

Policy: $\pi_t(s, a)$ = prob. that $a_t = a$ when $s_t = s$.

State should satisfy Markov Property:

Model-based and Model-free.

Value Function:

$V^{\pi}(s)$ = expected long-term return of being in this state, following policy π.

TD - Gammon.

- first application of learning value function using NN function approximator.
- representation of board as input
- learned value of board position as output.
- weighted update using temporal difference rule on every move.

↓

current estimate updated to be closer to next board's value.

TD rule

$$w_{t+1} - w_t = \alpha \left(Y_{t+1} - Y_t \right) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

← control temporal credit assignment of how much of an error detected at a given time step feeds back to previous estimate

Y_t: expected value of board position. (will be minimized when $Y_{t+1} - Y_t = 0$).

Alpha – Go

- Train two policy networks by supervised training on expere positions.
  - one is shallow and fast → used for rollouts
  - one is deep and accurate → train another network by policy gradient method.

$$\Delta w = \frac{\partial t \ln(y)}{\partial w} \text{sign}(r)$$

After supervised training, network is improved by playing itself.

A 3rd network: value network trained to predict winner from every state.

→ from every state seen during play, train to predict win rate.

Masked Autoencoder

masking then reconstruct input in pre-training.

After pre-training, use a simple decoder.

achieve good result with minimum decoder

BERT

Step ①: break image with non-overlapping patches.

②: mask 75% of patches.

③: positional encoding of the patches.

④: learns to reconstruct the whole image.

- self-supervised learning without any examples.
- self-supervised learning with few data argumentation.

AGI (advanced ML/AI)

- agent learn as much as they can about the world without interaction (by observation).
- differentiable architectures for planning and reasoning
- learn multi-level abstraction through long-term prediction and long-term planning.