DEEP LEARNINGS:

Lecture 2:

Bayes Rule:

likelihood ↓ Prior: what we know of y before seeing x

$$\underbrace{P(y|x)}_{\text{Posterior}} = \frac{P(x|y)\,P(y)}{\underbrace{P(x)}_{\text{evidence}}}$$

Propensity for observing a certain value of x given a certain value of y.

What we know about "y" after seeing "x"

$$\mathbb{E}[f(x)] = \sum_x f[x]\, Pr(x)$$

$$\mathbb{E}[f(x)] = \int f[x]\, Pr(x)\, dx.$$

Generalizatn Error: Error of the hypothesis / model measured w.r.t. probability dist. from which training samples are drawn.

Basically how well we will do on future data, but we don't know the future data "x_i" and also their labels y_i. We only know the "range" of all possible (x, y).

$$E_{gen} = \int$$

Underfitting / Overfitting: Learning of an algorithm depends on

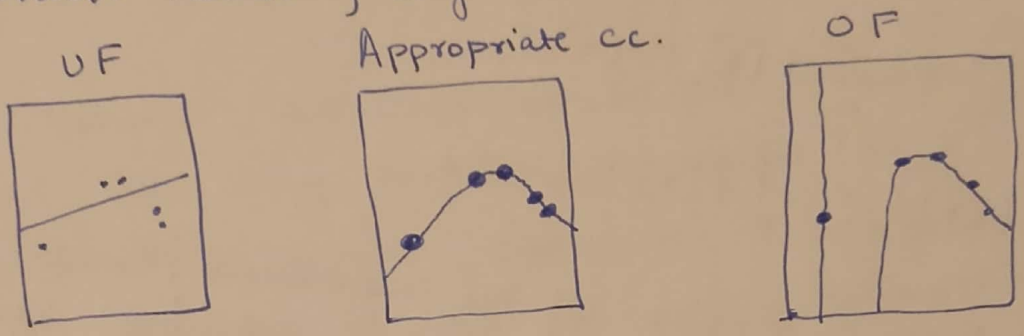
① training error small
② making gap b/w training & test error small

generalizatn gap.

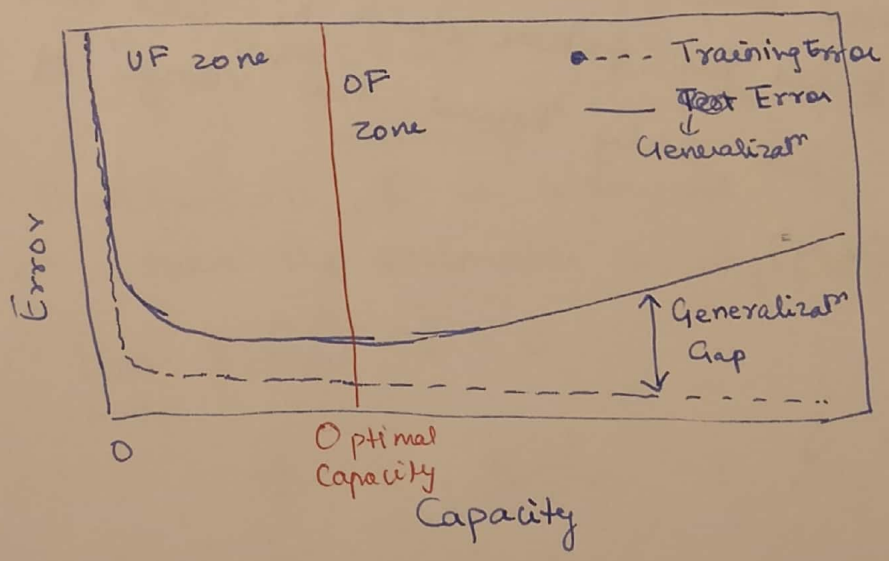
In the case of underfitting: (1) not fulfilled.
Overfitting: (2) F (1) but not (2).

Model Capacity: OF and UF controlled by model capacity. If UF then ↑ the model capacity, while in OF ↓ the model capacity. (because in UF model less expressive while in OF, model memorizes the data).

→ Capacity controlled by hypothesis space. (i.e. set of solutn from which learning algorithm chooses from).



Capacity vs. Error



Hyperparameters: Not set during learning, set in advance or determined through a nested learning procedure. It ① overall, controls the behaviour of algorithm.

Point Estimatn: Provides a single best prediction for a parametr or a function. denoted by $\hat{\theta}$ and $\hat{f}$. The point estimation function doesn't need to guarantee or return something close to true value or it might even not have same range.

For our sake, we assume θ be fixed and $\hat{\theta}$ as r.v.

Function Approx.: $\exists\ f(x)$ which can approx. relationship from x to y i.e. $y = f(x) + \varepsilon$. → error / noise.

→ In functn estimatn, we approximate true f with $\hat{f}$ based on some model. Basically, the class of model (i.e. y) is already known, so functn approx. simplifies to estimating the functn parametr θ. (Can also be seen as point estimatn).

Bias: Bias of estimator

$$\text{Bias}(\hat{\theta}_m) = \underbrace{\mathbb{E}(\hat{\theta}_m)}_{\text{estimator}} - \underbrace{\theta}_{\text{true value}}$$

$\mathbb{E}$ over data

When $\text{Bias}(\hat{\theta}_m) = 0$, $\hat{\theta}$ is unbiased. If bias is zero, only if $m \to \infty$, then the estimator is asymtotically unbiased.

$$\lim_{m\to\infty} \mathbb{E}(\hat{\theta}_m) - \theta = 0$$

$$\Rightarrow \lim_{m\to\infty} \mathbb{E}(\hat{\theta}_m) = \theta.$$

estimator propelty is

Variance: Another variance.

$$\text{Var}(\hat{\theta}_m) = \mathbb{E}\left[(\hat{\theta} - \mathbb{E}(\hat{\theta}))^2\right]$$

$$= \mathbb{E}(\hat{\theta}^2) - (\mathbb{E}(\hat{\theta}))^2$$

$$\text{Standard error} = SE(\hat{\theta}_m) = \sqrt{\text{Var}(\hat{\theta}_m)}$$

Mean Squared Error. (MSE): Good measure of generalizatn error (4)

$$MSE(\hat{\theta}_m) = \mathbb{E}[(\hat{\theta} - \theta)^2]$$

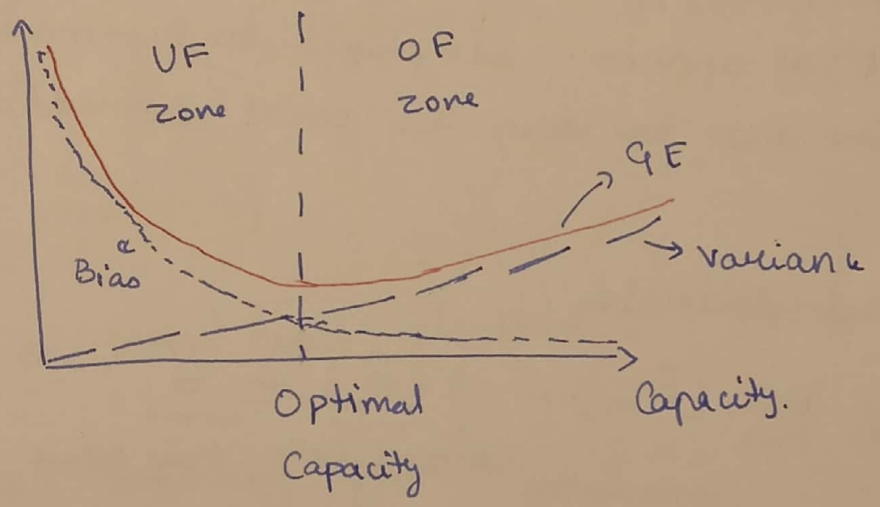
$$= \mathbb{E}(\hat{\theta}^2) - 2\,\mathbb{E}(\theta)\,\mathbb{E}(\hat{\theta}) + \mathbb{E}(\theta^2) \quad (\because \theta \text{ \& } \hat{\theta} \text{ are independent})$$

$$= \mathbb{E}(\hat{\theta}^2) - 2\theta\,\mathbb{E}(\hat{\theta}) + \theta^2 + \mathbb{E}^2(\hat{\theta}) - \mathbb{E}^2(\hat{\theta})$$

$(\because \theta \text{ is fixed})$.

$$= \underbrace{\left(\mathbb{E}(\hat{\theta}) - \theta\right)^2}_{\text{Bias}} + \underbrace{\mathbb{E}(\hat{\theta}^2) - \mathbb{E}^2(\hat{\theta})}_{\text{Var}}$$

$$= \left(\text{Bias}(\hat{\theta}_m)\right)^2 + \text{Var}(\hat{\theta}_m).$$



Maxm Likelihood Estimatn (MLE): We defined estimator as a functn of data i.e. $\hat{\theta}_m = g(x^{(1)}, \ldots x^{(m)})$ where $x^{(1)} \ldots x^{(m)}$ are i.i.d.

But where does $g(\cdot)$ come from?

→ Commonly, it comes from MLE, defined as

$$\hat{\theta}_{ML} = \underset{\theta}{\text{argmax}}\; P_{model}(X, \theta) = \underset{\theta}{\text{argmax}}\; P_{model}(x$$

$$= \underset{\theta}{\text{argmax}} \prod_{i=1}^{m} P_{model}(x^{(i)}, \theta)$$

$$= \underset{\theta}{\text{argmax}} \sum_{i=1}^{m} \log P_{model}(x^{(i)}, \theta)$$

Lecture 3 NN Basics I

⇒ Activation functions should be non-linear and differentiable

→ Regression: For regression, $\sigma(\cdot)$ is simply an identity funct^n i.e. $y_p = a_p$.

→ Classificat^n: logistic sigmoid for binary problems, or softmax for multiple classes.

$$\sigma_{binary}(a) = \frac{e^{(a)}}{1+e^{a}} \qquad \sigma_{multi-class}(a) = \frac{e^{(a_i)}}{\sum_k e^{a_k}}$$

→ $\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$ $\qquad \frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$

→ ReLu = $\max\{0, a\}$.

Advantages of tanh and Sigmoid : Bounded, Smooth, different abl

Disadvantages: Vanishing gradients, Gives only dense representat^n as no way can be zero.

Advantages of ReLU: enforces sparsity, easy to implement.

Disadvantages: unbounded, exploding gradients.

→ Why is it important to enforce that the activat^n funct^{n}s should be non-linear?

→ Because, if they are linear, than essentially the multi-layer feed forward network can be scaled down effectively to one layer network!!

→ Suppose we have network with N hidden layers and "k" activations.

$$y_k(x,w) = \sigma\left(w_k^{(N)} h^{(N-1)} \cdots h^{(2)}\left(w_k^{(2)} \cdot h^{(1)}\left(w_j^{(1)} \cdot x\right)\right)\right)$$

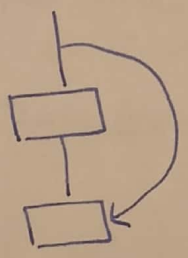
$$= \sigma\left(w_n^{(N)} h_\bullet^{(N-1)} \cdots h^{(2)}\left(w_k^{(2)} \cdot w_j^{(1)} \cdot h^{(1)} \cdot x\right)\right)$$

$$= \sigma\left(\underbrace{w_n^{(N)} \cdots w_k^{(2)}}_{w'} \cdot \underbrace{h^{(N-1)} \circ \cdots \circ h^{(2)} \circ h^{(1)}(x)}_{g}\right)$$

$$= \sigma(w' g(x)) \quad \text{Equivalent collapsed network.}$$

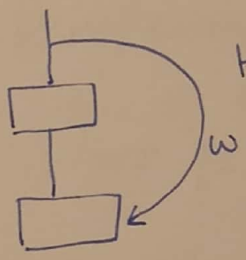
→ Depth of NN: # of activations or weight matrices.!!

Skip Connections:



$H(x) = F(x) + x$

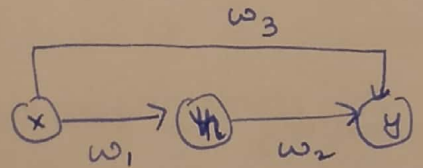
"Residual network" where "weights" are set to identity, only connections are made.



$H(x) = F(x) + wx$

Skip connect": where we connect all the neurons of "i^{th}" layer by skipping some layers to some "j^{th}" layer with weights associated to the connection.

Skip Connect Ex. in two layered NN.



$h = \sigma(w_1 x)$

$y = \sigma(w_2 h + w_3 x)$

$E = \frac{1}{2}(y - y_t)^2$ $\quad y_t$ = true label.

$$\frac{\partial E}{\partial y} = (y - y_t)$$

$$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial y} \times \frac{\partial y}{\partial w_2} = (y - y_t)\,\sigma(w_2 h + w_3 x)\,(1 - \sigma(w_2 h + w_3 x))\,h.$$

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial y} \times \frac{\partial y}{\partial w_3} = (y - y_t)\,\sigma(w_2 h + w_3 x)\,(1 - \sigma(w_2 h + w_3 x))\,x.$$

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial y} \times \frac{\partial y}{\partial h} \times \frac{\partial h}{\partial w_1}$$

$$= (y - y_t) \times \sigma(w_2 h + w_3 x)\left[1 - \sigma(w_2 h + w_3 x)\right] w_2 \times \sigma(w_1 x)(1 - \sigma(w_1 x))\,x$$

Weight Space Symmetry:

Odd Activation Funct^n's: Odd funct^n's such as $\tanh(a)$, Since $\tanh(-a) = -\tanh(a)$. If consider a fully connected two layered NN, then setting any consecutive wt's w_{ji} and w_{kj} negative cancels. M hidden units leads to M symmetries, so any "w" given wt. vector will be one of the 2^M leading to same O/P.

Non-unique Ordering: Also can swap hidden unit's wts and i/p's, which leaves network unchanged. $\Rightarrow$ M! orderings.

$\Rightarrow$ Total $M!\,2^M$ symmetries $\rightarrow$ This is not limited only for tanh funct^n.

Universal Approximators!!: Basically given enough model capacity can approximate any funct^n (i.e. Turing Complete).

Why would Learning fail?

① Insufficient # HU's.
② Optimizatn alg. cannot find optimal parameter values
③ training algo. chooses wrong functn ~~from~~ by overfitting

Training Feed forward NN.

→ Gradient Descent Optimizatn.

$$w^{(t+1)} = w^{(t)} - \underbrace{\eta}_{\text{learning rate}} \nabla E(w^{(t)})$$

→ Stochastic Gradient Descent. (SGD).

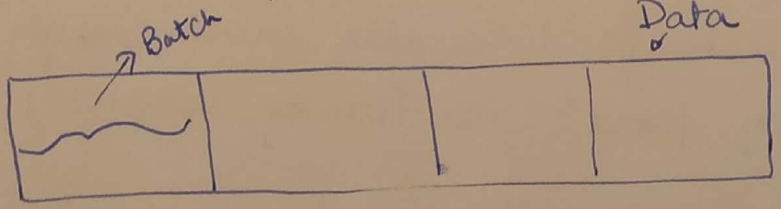
For iid's based on ML, we can decompose the error function into sum of terms, one for each training sample.

⇒ $E(w) = \sum_n E_n(w)$.

→ Online Learning: Updates are made to "w" based on the error gradient one data pt. at a time i.e

$$w^{(t+1)} = w^t - \eta \nabla E_n(w^{(t)})$$

→ Minibatches are an intermediate when we update based on small batches of data pts. (SGD).



epochs: # times you iterate over the whole training data.

Minibatch / Batch size: Divide your data accordingly to do back propagatn after seeing that subset of data.

iterations: # times you do BP to complete one epoch.

Initializing wt's : If w is near zero, so is w.x. Then if we apply activatn as sigmoid, then operating sigmoid becomes roughly linear. ⇒ Network collapses into an approximate linear model.

Still

→ Always advisable to start wt's near zero randomly in the initial layers so that model starts linearly and then wt's ↑ and in a way become non-linear. (Ciriculum learning)

→ Large wt's lead to bad results. (Next Chp May be).

→ Generally advisable to scale the i/p's s.t. they have zero-mean and std. deviatn 1. (Standardizatn of Data)

⇒ Ensures all dim. are handled equally in the regularizatn.

How to standardize data? $\frac{x^{(i)} - \mu}{\sigma}$, where μ = mean & σ = STD.

Directional derivative: of $f(x)$ is in the direction of unit vector "u" is the instantaneous rate of change of f in dir^n of "u".

$$\nabla_u f(x) = \lim_{\alpha \to 0} \frac{f(x+\alpha u) - f(x)}{\alpha} = u^T \nabla f(x).$$

To minimize "f", want to find dir^n "u" in which it $\downarrow$'s

$$\min_{u, u^T u = 1} u^T \nabla_x f(x) = \|u\|_2 \, \|\nabla_x f(x)\|_2 \cos\theta$$

where θ is $\angle$ b/w "x" and "u".

→ This comes down to $\min \cos\theta$, since $\nabla_x f(x)$ does not depend upon u, i.e. in the opposite dir^n of $\nabla_x f(x)$.

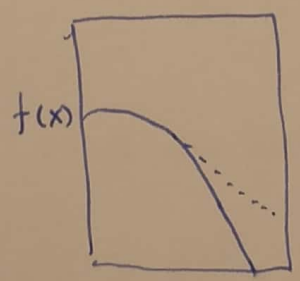
i.e. $x' = x - \epsilon \nabla_x f(x)$, where ϵ = learning rate

↑ how big a step to take in the opposite dir^n of ~~derivative~~ gradient.

First Order Derivatives → Jacobian

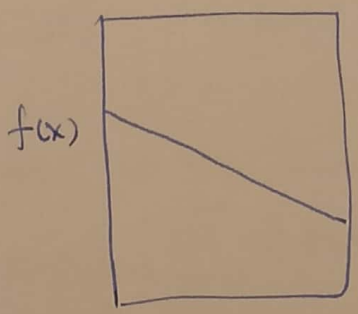
Second Order " → Help us understand the properties or the nature of the curvature and estimates how much improvement does the gradient step has.

-ve curvature



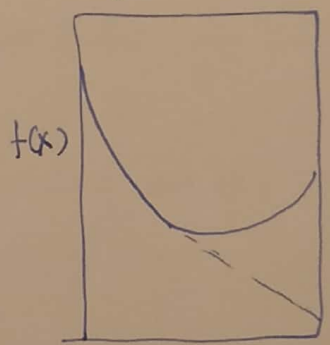
Cost funct^n $\downarrow$ faster than gradients predicts.

No curvature



Gradient predicts $\downarrow$ correctly.

+ve curvature



funct^n $\downarrow$'s slower than expected and eventually $\uparrow$'s, so taking large steps would $\uparrow$ the funct^n again.

THE HESSIAN MATRIX.

$$H(f)(x)_{ij} = \frac{\partial^2}{\partial x_i \partial x_j} f(x).$$

Directional second derivatn in dirn of unit vector d. is given by $d^T H d$.

→ GD in NN.

→ In most of cases NN, can be thought of as a parametric model that defines cond. distributn $p(y|x;\theta)$. It assigns the cost functn based on ML.

→ $\theta_{ML} = \underset{\theta}{\text{argmin}} - \sum_{i=1}^{m} \log p(y^{(i)} | x^{(i)}, \theta)$

→ For some given mapping $\hat{y} = f(x,\theta)$, we donot want a functn "f" that estimates a single $\hat{y}$, but rather a distributn (i.e $p(y|x;\theta)$)

→ Suppose we use Gaussian to model this conditional i.e. $\mathcal{N}(y; \hat{y} = f(x,\theta), \sigma)$, the conditional log-likelihood is

$$\sum_{i=1}^{m} \log p(y^{(i)} | x^{(i)}, \theta) = -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^{m} \frac{\| f(x,\theta) - y^{(i)} \|^2}{2\sigma^2}$$

→ we are only interested in the terms depending on θ,

$$\Rightarrow \theta_{ML} = \underset{\theta}{\text{argmin}} \sum_{i=1}^{m} \| f(x,\theta) - y^{(i)} \|^2$$

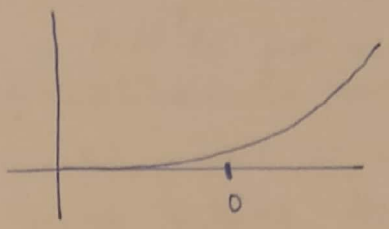
i.e. in some way equivalent to MSE.

→ In the cost funct^n when $p_{model}(y|x)$ is a gaussian $N(y, f(x,\theta), I)$, then we get $J(\theta) = \mathbb{E}_{x,y \sim \hat{p}_{data}} \| f(x,\theta) - y \|^2$

Activation funct^n's.

→ Softplus funct^n: $\zeta(x) = \log(1+\exp(x)) = \log(1+e^x)$.

softened version of $x^+ = \max(0, x)$.



Binary Classificat^n: If use sigmoid funct^n at the output layer, than use log loss funct^n (ML) and not ~~MSE~~ least squares error.

(final) i/p into O/p layer

$$\log \hat{P}(y) = yz \Rightarrow \hat{P}(y) = e^{yz}$$

$$P(y) = \frac{e^{yz}}{\sum_{y=\{0,1\}} e^{yz}} \Rightarrow \sigma((2y-1)z)$$

↓

Parameterized by Bernoulli

Multi label Classificat^n with Softmax: Generalizes sigmoid (σ).

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_k e^{x_k}}$$

use feed i/p "z" into softmax as the log of unnormalized prob.

$$z_i = \log \hat{P}(y=i|x)$$

In training, want to maximize log-likelihood ie

$$\max \log P(y=i|z) = \max \log \text{softmax}(z_i).$$

where $\log\text{softmax}(z_i) = z_i - \log \sum_j e^{z_j}$

⇒ i/p z_i will always have direct contribut^n to the cost function, and in principle this term cannot saturate (since it is linear w/t ing of previous layers) learning can always progress !!

→ The more stable version of softmax is $\text{softmax}(z) = \text{softmax}(z - \max_i z_i)$ (13)
This is ∵ softmax is invariant to addition of some scalar i.e. $\text{softmax}(z) = \text{softmax}(z+c)$. This allows us to ~~eva~~ evaluate softmax with only small numerical errors when z contains extremely large #'s.

→ Sigmoid – Use the negative log-likelihood as functn of z (last input) to output rather than $\hat{y} = P(y=i|x)$. This is so that sigmoid underflows to zero, and can avoid taking log of $\hat{y}_i$ which results to $-\infty$.

→ ReLU = $\max\{0, x\}$.

Generalizing ReLU.

$$h_i(a_i, \alpha_i) = \max(0, a_i) + \alpha_i \min(0, a_i).$$

↓ non-zero slope for unit "i" when $a_i < 0$.

Three variations include:

① Absolute Value Rectificatn: fixes $\alpha_i = -1$ and get $z = |a|$. Used for object recognizatn from images, where it makes sense to learn features that are invariant under polarity reversal to input illuminatn's.

② Leaky ReLU: fixes α_i to very small value eg. 0.01

③ Parametric ReLU: treats α_i as learnable parameter.

→ During initializatn, generally advisable to initialize wt's with value 0.1, so that they can pass through the activatn layer.

→ Max Out Units: Generalize ReLU's by dividing "a" into "k" grps of values. Each max out unit then o/p's max of one of these grps.

$$z = h_i(a_i) = \max_{j \in [1,k]} a_{ij}$$

→ Since max-out unit is parameterized by "k" weight vectors instead of one, learning such units needs more "regularization" than std. ReLU. Can work well in large data ↓ training ↓ # pieces per unit is low.

→ Since max-out units have multiple filters, it is difficult for them to forget how to perform tasks based on previous tasks, ⇒ catastrophic forgetting.

Lecture 5 Optimizat^n

Cost function: Usually avg. over training set.

$$J(\theta) = \mathbb{E}_{(x,y)\sim \hat{p}_{data}} \left[L(f(x,\theta), y) \right].$$

where L is per sample loss based predicted o/p $f(x,\theta)$ and target y.

$$\mathbb{E}_{(x,y)\sim \hat{p}_{data}} \left[L(f(x,\theta), y) \right] = \frac{1}{m} \sum_{i=1}^{m} L\left(f(x^{(i)};\theta), y^{(i)}\right)$$

→ Mini batch algorithms converge faster using approx^n gradient rather than exact slower gradients.

Ill-Conditioning of Hessian: When this is the case, SGD gets stuck, so that even a small step ↑'s the cost funct^n.

$$x = x^0 - \varepsilon g.$$

(2^{nd} Order Taylor series approx^n)

$$f(x) \approx f(x^0) + (x-x^0)^T g + \frac{1}{2}(x-x^0)^T H (x-x^0)$$

$$= \underbrace{f(x^0)}_{\text{orig. cost}} - \underbrace{\varepsilon g^T g}_{\text{expected improvement}} + \underbrace{\frac{1}{2}\varepsilon^2 g^T H g}_{\text{correct}^n \text{ to account for curvature of funct}^n}.$$

→ The last term when too large can cause gradient to move uphill. i.e. 3^{rd} term $> \varepsilon g^T g$. (2^{nd} term)

→ When ill-conditioning, then learning becomes very slow despite of strong gradient, and one needs to ↓ the learning rate inorder to nullify for the strong curvature.

Exploding Gradients: NN have steep regions in cost funct^n. Such a structure comes from multiplying many large wt's.

→ When parameters get close to the cliff, a small update can throw the parameters very far away, thus undoing all the optimizat^n done before.

Gradient Clipping: Exploding gradients can be overcomed by GC. heuristic. In an update gradient does not specify step size, only the dir^n, while in SGD step size α gradient, but in clipping one can "cap" the step size to prevent from parameter moving away far.

Stochastic Gradient Descent

Require: learning rate ε_k, initial parameter θ.

while stopping criterion not met do

Sample mini-batch of m examples from training data

Compute gradient $\hat{g} \leftarrow + \frac{1}{m} \nabla_\theta \sum_i L(\quad)$

Update $\theta \leftarrow \theta - \varepsilon \hat{g}$.

→ In practice, it is common to decay the learning rate with some schedule. eg. linear ↓ until t iterat^n

$$\varepsilon_k = (1-\alpha)\varepsilon_0 + \alpha \varepsilon_t \qquad \alpha = \frac{k}{t}.$$

→ Only use validat^n for hyperparameter tuning, no wt's get update when you train on validat^n (i.e. no BP). If you update wt's, then you most probably invite overfitting!!

→ Monitoring validation error over the progress of learning and when we face error plateaus, ↓ the learning rate & see what happens.

Momentum : Using it, one can accelerate learning when there is high curvature, small consistent gradients, or noisy gradients.

→ Aggregates exp. decay, gradients from past iterat'n's to continue stepping in that dir^n.

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(\,) \right)$$

• current gradient elements

$$\theta \leftarrow \theta + v.$$

$$\alpha \in [0, 1).$$

→ Momentum tries to solve the ① poor conditioning of Hessian ② variance in the ~~SG~~ stochastic gradient.

→ SGD with Momentum

I/P: ϵ, α momentum parameter, θ, initial velocity v.

while stopping criterion not met do

sample "m" examples

Compute GD: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x_i, \theta), y)$

Compute velocity update: $v \leftarrow \alpha v - \epsilon g$

Apply update: $\theta \leftarrow \theta + v$

end while

→ Nesterov Momentum: Same as Momentum, but gradients are evaluated after applying current velocity i.e.

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(x_i, \theta + \alpha v), y) \right)$$

↳ difference b/w Momentum and NM.

$$\theta \leftarrow \theta + v$$

→ Can speed up convergence for convex gradients.

Approximating 2nd Order Methods.

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T H (\theta - \theta_0)$$

Solving for critical pt.

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0). \quad \text{(Newton Update)}$$

→ The update is more accurate as Hessian captures/determines the local curvature of the loss

Initialization's

→ Should do random initializatn ∵ we don't than an deterministic algorithm can be used to update the wt's of two hidden units (having same wt's and activatn functn) and update will take place in same manner.

→ Large wt. Initializatn's : (A) Helps in breaking symmetry and avoids losing signal during fwd propagatn.

(D) Exploding gradients, both in FP & BP

(D) Makes the activatn function saturated.

→ Small wt. Initializatn's : (A) Good from regularizatn perspective eg. wt. decay,

(A) they induce prior for final parameters being close to initial parameters

(D) If we initialize with large values, then our prior specifies which units should interact with each other and how they should interact.

→ wt. Initializatn as a hyperparameter if you have good amount of computational power, then train on one minibatch and see activations and gradients and appropriately scale the wt's.

→ Bias Initializat^n.: Usually zero initializat^n works but can also do

① to match o/p's to marginal statistics

② avoid saturat^n eg. ReLU units bias should be initialized as 0.1 instead of 0, avoiding saturat^n at initializat^n.

Adaptive Learning: Learning rate should ↓ periodically.

① Adagrad - scales a fixed learning rate inversely proportional to root of sum of all historical values of gradients.

② RMSProp - changes gradient accumulat^n of Adagrad into an exp. wt'd moving avg.

③ Adam - adds momentum to RMSProp.

① Adagrad : $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}, \theta), y^{(i)})$

gradient:

Accumulate squared, $r \leftarrow r + g \odot g$ → elemest wise

Update : $\Delta\theta \leftarrow \frac{-\varepsilon}{\delta + \sqrt{r}} \odot g$

Apply update : $\theta \leftarrow \theta + \Delta\theta$.

② RMS Prop : $g \leftarrow$ same

Accumulate sqd gradient : $r \leftarrow \rho r + (1-\rho)\, g \odot g$

Update : $\Delta\theta \leftarrow \frac{-\varepsilon}{\sqrt{\delta + r}} \odot g$

Apply update : $\theta \leftarrow \theta + \Delta\theta$.

③ ~~Adagrad~~ RMS Prop with Momentum

$\tilde{\theta} \leftarrow \theta + \alpha v$

$g \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(x^{(i)}, \tilde{\theta}), y^{(i)})$

$r \leftarrow \rho r + (1-\rho)\, g \odot g$

Velocity update $v \leftarrow \alpha v - \frac{\varepsilon}{\sqrt{r}} \odot g$

Update $\theta \leftarrow \theta + v$.

Batch Normalizatn : Method of adaptive-reparameterizatn

→ Can be applied to any input or hidden layer and reduces the problem of coordinating updates across many layers.

→ H is a minibatch of activations, each column represents the activatn of a given unit layer while a row represents activatn of an example.

$$\underbrace{H'}_{\text{normalized}} = \frac{H-\mu}{\sigma}$$, where μ & σ are column wise means & Std. deviats

$$\mu = \frac{1}{m}\sum_{i=1}^{m} H_i \qquad \sigma = \sqrt{(\delta) + \frac{1}{m}\sum_{i=1}^{m}(H-\mu)_i^2}$$

δ : usually 10^{-8} to avoid undefined gradients.

→ Can also be considered as whitening of data, since each dimension (unit of a layer) is de-correlated and all have a unit variance

Learning alg.

→ Normalizatn makes BP efficient : Doesn't waste time to adjust means and Std deviatn. Also it prevents scaling errors from propagating through the network since o/p's are normalized after normalizatn layer.

→ For a layer of form $\phi(WX+b)$, where ϕ is the non-linear activatn funct^n, it is recommended to apply batch normalizatn on XW and remove "b", since bias is redundant with β of BN.

Usually $\gamma H' + \beta$, where γ & β are learned parameters.

Lecture 7 CNN I

(Regularizatn lec Skipped)

→ Building invariance directly into the network is the basis for convolutional neural networks., widely applied to images

→ Invariance: System should remain Invariant to translations, scaling and small rotations and also invariant to some elastic deformations. (small nonlinear transformatn.)

→ 3 Pillars of convolution in CNN!

→ The convolutn operation allows us to do this for i/p's of varying sizes and

① local receptive fields (sparse connectivity in range vs fully connected)

② parameter sharing (allows for detectn of same features over image)

③ equivariant representatn and subsampling (reduces informatn hierarchically. (i.e. downsampling !! ?).

① local receptive fields imply sparse connectivity and interactn. Though, the connectivity is sparse, the neurons still interact with each other after some depth and eventually some neurons which did not interact, will interact when we apply/add fc-layer at the end before o/p.

② Parameter sharing.

③ Translatⁿ equivariance - Change in i/p, changes o/p in the same manner. i.e. $t(g(x)) = g(t(x))$, t is equivariant to g.

→ Through parameter sharing, convolutⁿ becomes spatially invariant.

→ Suppose now that remapping is shift of pixel locatⁿ's i.e. $I' = g(I) = I(x-1, y)$. Translatⁿ equivariance means that applying convolution to the remapped I' is equivalent to first applying convolutⁿ to I and then applying the remapping $g(\cdot)$.

→ Why is equivariance useful & and when is it not?

→ Also prove the shift equivariance property of convolution!!

Convolution variants: On each image / layer we generally apply multiple kernels in parallel. We can think of a kernel as a feature extractor.

Multiple Image Channel: Color image is 3-d tensor and thus WxHxC channel. (R, G, B).

Stride: For saving computational expense skip some pixels. Can be thought of as downsampling in the o/p of fully ~~connected~~ convolutional funct^n.

Zero Padding: Depends upon the size of kernel.

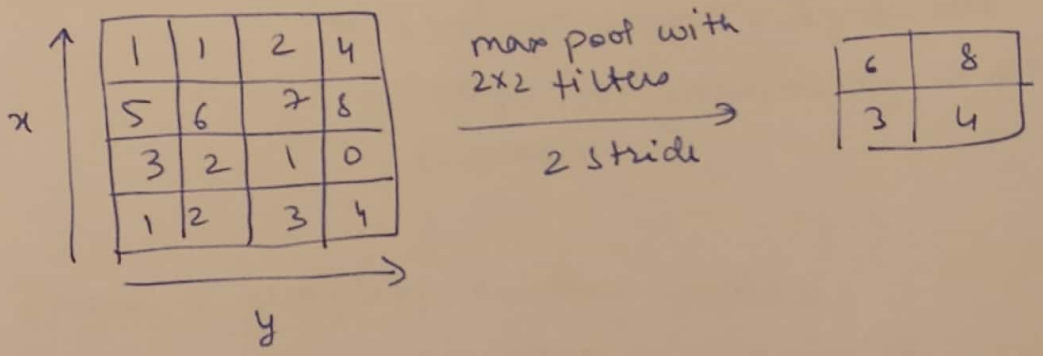
CNN Layer: 3 Stages

① Convolut^n operat^n ② non-linear activation eg. ReLU

③ pooling funct^n (max, avg.) → some summary statistic.

↓

can also define strides..



→ Pooling over spatial regions results in invariance to translat^n. Can also lead to other types of ~~eqp~~ invariance when parameterizat^n is changed.

→ Pooling also allows handling input images of varying sizes. By changing the stride, the size of feature map ~~can be~~ will remain the same for images of varying sizes.

→ Do an example of counting # parameters in the CNN architecture !!

Lecture 8. CNN II

→ Alex Net , VGGNet , GoogLeNet , MSRA

DenseNet.

CNN Visualizatn.

① Visualize filters

② Visualizing last layers (k-NN)

③ Multiple layer Activations - Feature map visualizatn.

④ Saliency Maps: How to tell which pixels matter for classificatn?

$\frac{\partial E}{\partial x}$ → Compute gradient score (unnormalized) class w.r.t. image pixels, take absolute value & max over RGB channels

⑤ Gradient Ascent

$$I^* = \arg\max_I f(I) + R(I)$$

Neuron value

Natural image regularizer

Guided BP: Find an part of image that a neuron responds to

- Gradient ascent: Generate synthetic image that maximally activates a neuron.

⑥ Deep Dream: amplify existing features.

Rather than synthesizing an image to maximize a specific neuron, instead try to amplify the neuron activations at some layer in a network

Choose an image & a layer in CNN: repeat

① Fwd: compute activatn's at choosen layer

② Set gradient of chosen layer equal to its activatn

③ Backward: Compute gradient on image

④ Update image

Equivalent to $I^* = \arg\max_I \sum_i f_i(I)^2$

CNN Applications

① Transfer Learning (Just like we did our project on Synthesizing images).

- Take pretrained model.
- Remove the last fc-layer
- Use pretrained model as a fixed feature extractor for the new dataset
- Train a new classifier for the new dataset i.e linear SVM.

Also fine tune all the layers of pretrained model. Also keep some earlier layers fixed and only fine-tune some higher-level layers as their impact in the loss function is more of impact !!

② R-CNN

③ Faster R-CNN.

④ YOLO - You only Learn Once

} Detecting objects in the video.

Lec- 6 Regularizatn.

Regularizatn → Loosely defined as modificatn to a learning algorithm which reduces its generalizatn error but not training error.

How & what to do?

- add constraints to the model parameters (θ)
- extra ~~con~~ terms in the objective function (i.e. loss functn)

→ Most strategies focus on regularizing the estimator which trades off increased bias for reduced variance.

→ 3 Bias Variance Situations

① Model excludes true data generating data and underfits ∴ excluding bias

② model matches true data generating process

③ model includes true data generating process also with other possible generating processes & overfits, thus allowing variance to dominate estimation error (instead of bias).

↓ Most Probable, But try to shift to ②

Parameter Norm Penalties.: Add norm penalty $\Omega(\theta)$ to objective functn. i.e.

$$\tilde{J}(\theta, X, y) = J(\theta, X, y) + \alpha \Omega(\theta).$$

↑ hyper parameter (ideally diff. α for every layer but expensive)

Larger α means more generalizatn, smaller less.

→ Ω only penalizes wt's of affine transform & not bias (b).

→ Regularizing bias(b) may induce UF → each wt. specifies how two variables interact, so fitting a wt. well requires observing both in variety of conditions, ∴ bias only requires single variable hence needs less data

L^2 - Regularizer (Ridge regression). : $\Omega(\theta) = \frac{1}{2}\|w\|^2$.

→ Also called wt. decay as it tries to decrease the wt. and in turn makes gradients smoother.

→ How does wt. decay work?

$$\tilde{J}(w, X, y) = J(w, X, y) + \frac{\alpha}{2} w^T w.$$

$$\nabla_w \tilde{J}(w, X, y) = \nabla_w J(w, X, y) + \alpha . w$$

$$w \leftarrow w - \varepsilon (\nabla_w J(w, X, y) + \alpha w)$$

$$\equiv \underbrace{w(1-\alpha)} - \varepsilon \nabla_w J(w, X, y)$$

↳ This implies that at every update, the wt-decay regularizer performs a multiplicative shrink.

→ Simple wt. decay is inconsistent with certain scaling properties of network.

→ 2 layer perceptron, $\{x_i\}$ (i/p) $\{y_k\}$ (o/p).

$$z_{j0} = h\left(\sum_i w_{ij} x_i + w_{j0}\right) \quad \& \quad y_k = \sum_j w_{kj} z_j + w_{k0}.$$

→ Suppose we add linear term $x_i \to x_i' = a x_i + b$.

$$w_{ji} \to w'_{ji} = \frac{1}{a} w_{ji} \quad \text{and} \quad w_{j0} \Rightarrow w'_{j0} - \frac{b}{a} \sum_i w_{ji}$$

Similarly for o/p's $y_k \to y'_k = c y_k + d$, we can adjust wt's of 2nd layer

$$w_{kj} \to w'_{kj} = c w_{kj} \quad \& \quad w_{k0} \to w'_{k0} = c w_{k0} + d.$$

→ Network Consistency: Model is equivariant (i.e. if networks are consistent, then the wt's should differ only by linear transformations)

→ Simple wt. decay gives same importance to bias and wt's and prefers "smaller weights".

→ Consistent way to avoid inconsistency of L_2-regularizatn

$$\frac{\lambda_1}{2}\sum_{w\in W_1} w^2 + \frac{\lambda_2}{2}\sum_{w\in W_2} w^2$$

, where W_1 & W_2 are wts of layers 1 & 2 (excluding bias)

→ By doing this, wts are invariant to scaling.

$$\lambda_1 \to \lambda_1' = a^{1/2}\lambda_1 \qquad \lambda_2 \to \lambda_2' = a^{-1/2}\lambda_2.$$

$$p(w|\alpha_1,\alpha_2) \propto e^{\left(\frac{\alpha_1}{2}\sum_{w\in W_1} w^2 + \frac{\alpha_2}{2}\sum_{w\in W_2} w^2\right)}.$$ where α_i's are HP.

New regularizer corresponds to a prior.

→ Prior of this term cannot be normalized since bias is unconstrained. To break the symmetry, we do shift-invariance property.

L^1- regularizatn: $\Omega(\theta) = ||w||_1 = \sum_i |w_i|$

$$\tilde{J}(w,X,y) = \alpha||w_1|| + J(w,X,y)$$

$$\nabla_w \tilde{J}(w,X,y) = \alpha \cdot \text{sign}(w) + \nabla_w J(w,X,y).$$

In general L^1- regularizatn gives sparse wts, where some wts have values close to zero.

→ ~~L^2 is Gaussian prior~~ L^2 regularizatn is equivalent to MAP with a Gaussian prior of wts. $\log p(\theta|x) + \log p(\theta)$ (cost. + regularizatn term)

→ L^1 is equivalent to using an isotropic Laplace distributn over "w" for prior.

Tangent Propagation: Can also regularize model o/p's (make invariant to transformations), by tangent propagation.

→ If we a continuous transformation on input vector x_n such as (rotation, translatn), then the transformed pattern follows a manifold M with D-dim. input space

→ The transformatn is parameterized by ξ. The manifold swept by x_n is a 1-D space with parameter ξ.

→ $s(x_n, \xi)$ = resultant vector with $s(x, 0) = x$. The tangent to M is directional derivative $\tau = \frac{\partial s}{\partial \xi}$

$$\tau_n = \left.\frac{\partial s(x_n, \xi)}{\partial \xi}\right|_{\xi=0}$$

→ The o/p vector will also change due to transformatn o/ i/p.

$$\left.\frac{\partial y_k}{\partial \xi}\right|_{\xi=0} = \sum_{i=1}^{D} \frac{\partial y_k}{\partial x_i} \times \left.\frac{\partial x_i}{\partial \xi}\right|_{\xi=0} = \sum_{i=0}^{D} J_{ki} \tau_i$$

J_{ki} : Jacobian

$\tilde{E} = E + \lambda \Omega$ (this encourages local invariance on neighborhood of pts)

$$\Omega = \frac{1}{2} \sum_n \sum_k \left(\left.\frac{\partial y_k}{\partial \xi}\right|_{\xi=0}\right)^2 = \frac{1}{2} \sum_n \sum_k \left(J_{nki} \tau_{ni}\right)^2$$

where λ determines the trade-off b/w fitting & learning the invariance.

Reducing Model Capacity

→ Early Stopping: To prevent overfitting. Good form of hyperparameter selection. Disadvantages is that it is computationally expensive and you also take validatn data also as a hyperparameter. Also retraining the model all parameters get change and in the end as we do random initializatn, so no one can tell what will happen when if you train your model again.

→ One can also think it as restricting optimizatn procedure to a relatively small volm of parameter space

→ Parameter Sharing : CNN.

Adjusting Training Data.

→ Data Augmentation : Synthesize more data and train NN on this data.

→ Injecting Noise :

① Noisy I/P : Train your NN with noise-injected i/p's : i.e. add random noise to i/p's.

② Noisy wt's : Noise added to the wt's (RNN's). Consider model wt's as R.V.'s associated with some probability dist.

Noise as a Regularizer.

$$J = \mathbb{E}_{p(x,y)}\left[(\hat{y}(x) - y)^2\right]$$

Now add noise $\in_W \sim \mathcal{N}(\varepsilon, 0, \eta I)$.

$$\tilde{J}_W = \mathbb{E}_{p(x,y,\varepsilon W)}\left[(\hat{y}_{\varepsilon W}(x) - y)^2\right]$$

$$= \mathbb{E}_{p(x,y,\varepsilon W)}\left[\hat{y}^2_{\in W}(x) - 2y\hat{y}_{\varepsilon W}(x) + y^2\right].$$

for η = small, $\min J = \min \tilde{J}$. , with added regularizatn term $\eta \cdot \mathbb{E}_{p(x,y,\in W)}\left[\|\nabla_W \hat{y}(x)\|^2\right]$.

③ Noisy Output. → Hard classificatn problem becomes soft classificatn problem with correct being $1-\varepsilon$ and wrong with ε.

↓ One hot coding

1	0	0	0	0

$(1, 0)$

$1-\varepsilon$	$\varepsilon/4$	$\varepsilon/4$	$\varepsilon/4$	$\varepsilon/4$

$(1-\varepsilon, \varepsilon)$

→ Bagging - can reduce generalizatn error by combining ensemble of models. i.e. multiple models trained separately and then votes on o/p for a given test sample

→ The ensemble performs atleast as well as any members on an average and if members make errors independently, then the ensemble performs better! Error correlatn of members cannot be helped by ensemble.

→ Train different NN on different sampled dataset's sampled from training data.

with exp. large # models.

DropOut : Approximates the process of bagging (i.e. learn ↑ k-different models from k-different datasets).

→ For each sample, create a mask and apply it to a layer.

Dropout 0.6 → 6/10 → 0 ≅

Mask 0.6 → 6/10 → 1. } Masking value 1.
Hiding the wt.

DROPOUT	task of	BAGGING.
→ Models share parameter		→ Models are independent
→ Parameter sharing makes the representatn of an exp. # models easy (less memory)		→ Parameters are more due to models being independent. (Hence more memory)
→ Most not Each models are trained explicitly as parameter sharing helps.		→ Each model is trained to its convergence

→ Both follow bagging algorithm !!

→ Inferencing in DropOut.

① $P_{bagging} = \frac{1}{k} \sum_{i=1}^{k} p^i(y|x)$.

$P_{dropout} = \sum_{u} p(u)\, p(y|x,u)$.

$p(y|x,u)$ → dist. given mask vector u and data

$p(u)$ → prob dist used to sample u for training

This term is intractable, but 20-30 masks suffice.

② Use geometric mean as dropout. (Don't know, don't care)

③ Wt. Scaling Inference Rule at layer i

- Basically, dropout = 0.5, is equivalent to doubling the wt's of that layer and doing traditional ~~trad~~ training!! No need of masking !!

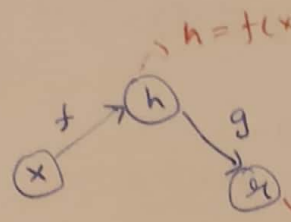
→ Effectiveness of Dropout.

- Computationally cheap ($O(n)$) computat^n per ex. per update.
- Model Agnostic.
- Dropout reduces the effective c.c. of model, hence one should ↑ the capacity of model. to compensate.
- Alternative view as a regularizer: Regularizes each HU to be not merely a good feature but a feature that is good in many contexts.

→ Dropout as injecting noise at HU: adding noise at feature extraction helps to enforce use of multiple features for prediction rather than relying on handful of features.

Skipped (RNN-I, II). Lecture-11 Autoencoders.

→ Special types of NN which are trained to "copy" the input to output.



→ Based on def^n, it is simply a series of identity funct^{n}s, which is a trivial setup, but we restrict h eg. its dimensionality, so that cannot copy the i/p.

→ We are interested in learning the hidden layer representation of i/p.

→ Used largely for dimensionality reduction and feature learning, but with their relation with latent variables, also used for generative modelling.

Discriminative Model : x-data, Y-label.

↓ Models posterior directly & determines class label accordingly

MAP Inference task to find Y^* maximizing $P(Y|X')$.

$$Y^* = \underset{Y}{\operatorname{argmax}}\, P(Y|X).$$

Generative Models: Models the class conditional prob. $P(X|Y)$ and uses Prior $P(Y)$ to find posterior $P(Y|X)$.

↓ to generate new data via $P(X) = \sum_Y P(X|Y)P(Y)$

(Class label determined by)

$$Y^* = \underset{Y}{\operatorname{argmax}} \frac{P(X|Y).P(Y)}{P(X)}$$

↳ what we want to learn and it's a bit more difficult to come with

→ Almost any generative model with a latent variable and the ability to inference on latent variable given i/p can be intered as an auto encoder

→ Encoder & decoder as stochastic mappings with $P_{encodn}(h|x)$ and $P_{decodn}(x|h)$.

Undercomplete AE - constrain the hidden layer to be of lower dimension than that of x, try to capture salient features of x $L(x, f(g(x)))$ → loss functn

→ PROVE this!!

→ If decodn "g" were linear and L was the MSE, then an undercomplete AE would span the same subspace as PCA. !!

Complete & OverComplete AE -

Complete = dim of i/p = dim of hidden layer

OverComplete = dim of i/p < dim of HL. (hidden code)

Even with little capacity can learn to copy data

Regularized AE : Modify the loss functn which encourages the model to learn other properties than copying. like

① Sparsity of representatn
② Smallness of the derivative of representatn
③ Robustness of noise or to missing i/p's.

Sparse AE → Adding the sparsity regularizatn term $\Omega(h)$ on code layer "h" to loss.

$$L(x, g(f(x))) + \Omega(h).$$

→ used for learning features of other tasks eg. Classificatn.

→ In other regularizatn's such as wt. decay, the regularizer depends upon the prior on model parameters. i.e. maximizing log-likelihood $\log p(x|\theta) + p(\theta)$.

→ While there is no interpretatn to regularizatn in AE, as the term depends upon the data.

→ $\Omega(h)$ is not a penalty for copying task, rather it is an approxᵐ to ML training of a generative model with latent variables.

$$P_{model}(x,h) = P_{model}(x|h)\, P_{model}(h)$$

→ prior over latent variables
↳ represents model's beliefs before seeing "x".

$$\Rightarrow \log P_{model}(x) = \log \sum_h P_{model}(x,h).$$

→ The term $\log P_{model}(h)$ can be sparsity inducing.

For ex., with Laplacian prior

$$P_{model}(h) = \frac{\lambda}{2} e^{-\lambda |h_i|}$$

If $\Omega(h) = \lambda \sum_i |h_i|$ we get $-\log P_{model}(h) = \sum_i \left(\lambda |h_i| - \log \frac{\lambda}{2}\right)$

$$= \Omega(h) + \text{const.}$$

→ The result of applying $P_{model}(h)$ on approxᵐ ML learning, is a result of model's distributn over its latent variables

→ This explains why features learned by the AE are useful, since they describe latent impacts on i/p.

Representatn of Depth → Training with one HL is shallow and we cannot add arbitrary constraints while preserving sufficient representatn. Adding depth helps ∵ it decreases the computatnal cost of representg some complex function.

↳ Also b/s the amount of training data needed to learn some representatn of a functn

Stochastic Encoders & Decoders.

→ Minimizing $-\log P_{model}(x|h)$ at decoder.

→ Minimizing negative log likelihood is equivalent to using MSE.

→ $P_{encoder}(h|x) = P_{model}(h|x)$

$P_{decoder}(x|h) = P_{model}(x|h)$.

→ Binary classificatn ⇒ Bernoilli distn. from Lec-2

Multiclass classificatn ⇒ Softmax

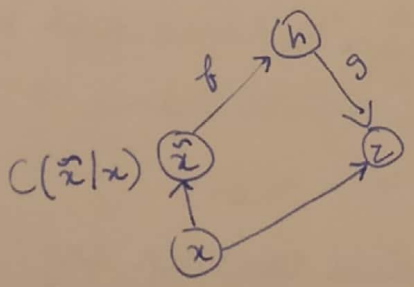
Regularizatn by Denoising AE.

→ Can also regularize by changing the reconstruct error.

→ To prevent eventual identity funct learning the denoising AE minimizes

$L(x, g(\hat{f}(\tilde{x})))$, $\tilde{x}$ = noisy version of x.

→ The AE will undo the noise instead of copying i/p to o/p.



① Sample training ex: "x" from training dat

② Sample $\tilde{x}$ from $C(\tilde{x}|x)$

③ learn f & g, using $(x, \tilde{x})$ where $P(x|\tilde{x}) = P_{decoder}(x|h=f(\tilde{x}))$, where $P_{decoder}$ is determined by $g(h)$.

→ Now, the injected noise in data can be thought learning the manifold of the underlying probability distributn $P(x)$.

→ Core assumptn of AE is that data is concentrated around low-dim. manifold. Goal is to learn the structure of manifold.

→ Tangent plane in manifold - For some pt x in d-dim M, the tangent plane is given by "d" basis vectors spanning the local dirn's of variations allowed on manifold.

→ If data is concentrated in low dim manifold, then only variations tangent to the manifold around "x" need to correspond the changes in h=f(x).

→ Hence, encodr is mapping sensitive changes along the manifold dirn's and insensitive to othogonal dirn's.

How to do Manifold Learning?

Embedding: a low-dimensional vector with less dim than input space, of which the manifold is a low-dim. subset.

→ Many manifold learning algs., learn the embedding of each training example.

→ The fundamental problem with non-parametric approaches to manifold learning is that manifold themselves are not smooth, hence need a lot of training data to cover variations

→ Because such method estimates the shape of manifold by interpolating b/w neighbouring ex., there is no chance to generalize for unseen variations.

→ Alternatively, learning a mapping which can map any pt. from i/p to embedding in hope that it would generalize better for unseen variations than by doing local interpolatn.

Contractive Autoencoders: Regularize the code $h=f(x)$ with $\Omega(h)$ to encourage derivatives of "f" to be as small as possible.

$$\Omega(h) = \lambda \left\| \frac{\partial f}{\partial x} \right\|_F^2 \quad \text{— Frobenius norm}$$

→ $\Omega(h)$ encourages feature encoder to resist small change to the i/p, which in turn maps a neighbourhood of i/p pts to a smaller o/p neighborhood – Contractive AE.

$$\underbrace{L(x, g(f(x)))}_{\substack{\text{reconstruct}^n \\ \text{error} - \\ \text{Only learn} \\ \text{identity funct}^n}} + \underbrace{\lambda \left\| \frac{\partial f}{\partial x} \right\|_F^2}_{\substack{\text{contractive} \\ \text{penalty} - \\ \text{features learned} \\ \text{would be constant w.r.t. } x}}$$

Applicat^{n}s: AE

① Dimensionality Reduction

② Informat^n Retrieval
- Semantic Hashing
- Embed data and query in low dimension.

③ As an initializat^n, AE can also be used for "pre-training" network.
- (a) Pretrain a sequence of shallow AE, greedily, stacking one layer at a time with unsupervised ~~dt~~ data
- (b) train last layer supervised, keep others fixed
- (c) BP & fine-tune the entire network supervised.

Lecture 13 GANS
Deep Generative Models

Why Study Generative Modeling

→ Training and sampling from such models tests out ability to represent and handle high-dim. dist.

→ helps dealing with multi-modal data learning.

→ Simulate possible outcomes to be incorporated to reinforcement

→ Loosely define generative modeling as capturing $P(x)$ for some complex X, but with the more correlated data, it becomes difficult to learn.

→ Hence latent variables (unobserved var.) can capture the variations of one or more observed variable

→ Ex. last lecture, each dim. of the manifold can be considered as latent variables.

Latent Variable Generative Model : learn mapping from some latent variable to a complicated dist. on x.

$$p(x) = \int p(x,z)\,dz \quad \text{where} \quad p(x,z) = \underbrace{p(x|z)}_{\text{difficult}}\,\underbrace{p(z)}_{\text{simple}}$$

Variational AE : Take initial vector as a vector on real #, which can be seen has latent variable and the code (h) is some kind of a representation of the image which can synthesize them.

Image → Encoder → Latent variable → Decode → Image

→ But we don't know how to create latent vector. For this we add a constraint s.t. latent variable's match a unit Gaussian dist.

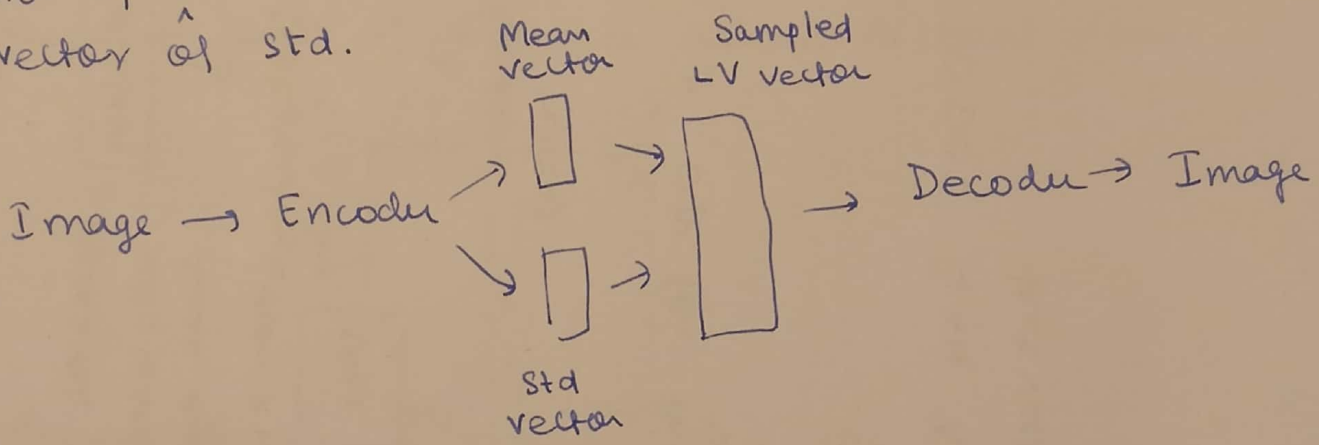
→ This differentiates variational AE from std. AE.

→ For our loss term, we sum two separate losses: ① generative loss ② latent loss

① MSE of reconstruction error

② KL-divergence how closely to LV match a unit Gaussian.

→ In order to optimize KL-divergence, we need to apply simple reparameterization trick: instead of the encoder generating a vector of real values, it will generate a vector of means & a vector of std.



GAN: Generative Adversial Networks

→ Based on game-theory scenario, where two networks compete

Generator - Generates samples $x = g(z, \theta^{(g)})$ with random noise z.

Discriminator - differentiates b/w real and fake, $d(x, \theta^{(d)})$

→ Generator Network: $G(z, \theta^{(g)})$.

- Must be differentiable
- No invertibility requirement
- Trainable for any size "z"
- Can make "x" conditionally Gaussian given "z".

Training a GAN: GAN has a binary entropy objective:

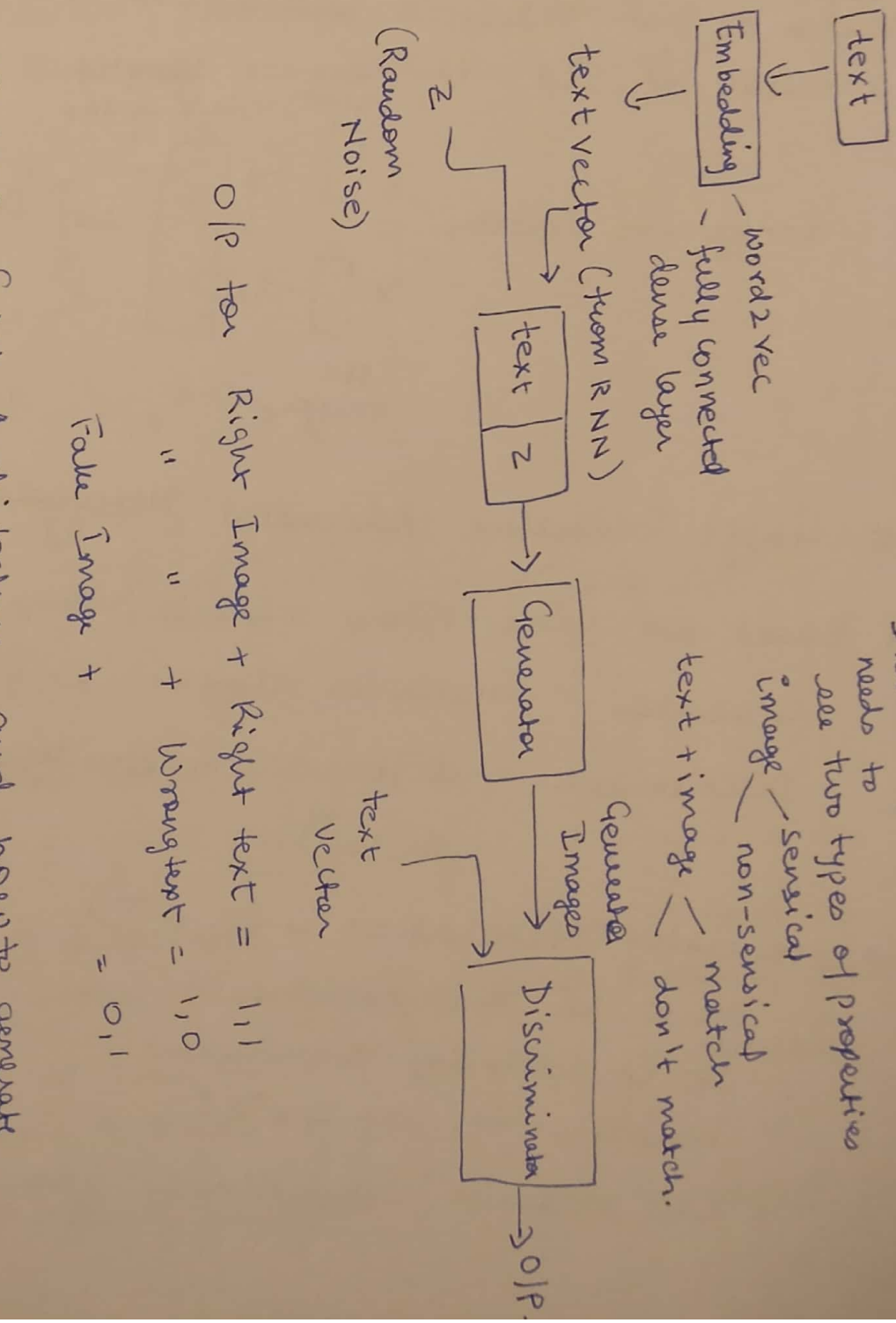
$$J_{GAN} = \log d(x, \theta^{(d)}) + \log\left[1 - d(g(x, \theta^{(g)}), \theta^{(d)})\right].$$

Training alternates b/w

$$\theta^{(g)*} = \underset{\theta^{(g)}}{\operatorname{argmin}}\ J_{GAN} \quad \& \quad \theta^{(d)*} = \underset{\theta^{(d)}}{\operatorname{argmax}}\ J_{GAN}.$$

→ Convergence when generator samples are no longer distinguishable from real data. i.e. d outputs = 1/2.

ICML (2016): Generative Adversial for text to Image Synthesis.



O/P for Right Image + Right text = 1,1

" " + Wrong text = 1,0

Fake Image + = 0,1

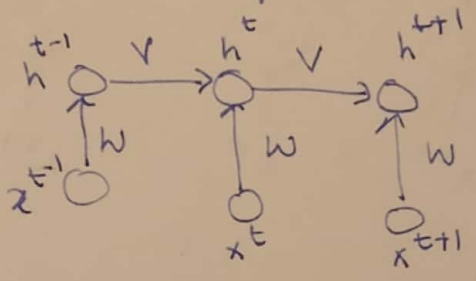
TO DO: GAN Architecture and how to generate text vector via RNN.

Lecture 9 RNN I

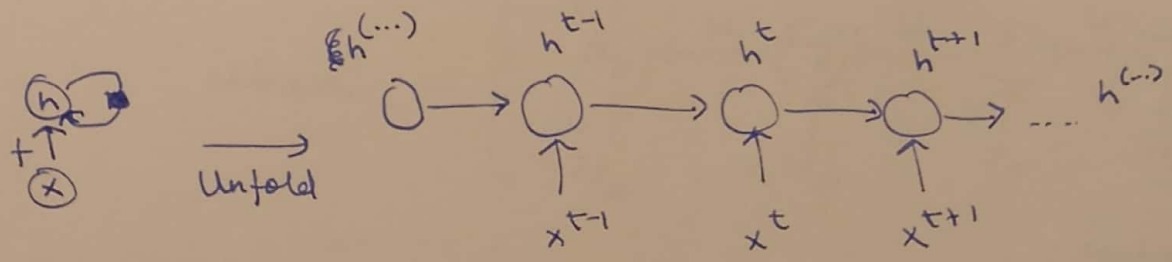
→ The relaxatn of the allowing of cycles in the network gives rise to Recurrent Neural Nets (RNN).

→ The MLP can only form a map from i/p to o/p layers. (vectors). But RNN can map from entire history to each o/p. This gives RNN a memory of previous i/p's.

Parameter Sharing is essential for RNN, ~~for~~ because if the parameter were time specific, models could not generate the sequence of lengths not seen in training, nor share any statistical strength across diff. length of sequences across different times.



→ Recursion : $s^{(t)} = f(s^{(t-1)}, \theta)$.



→ Recursiveness in hidden units

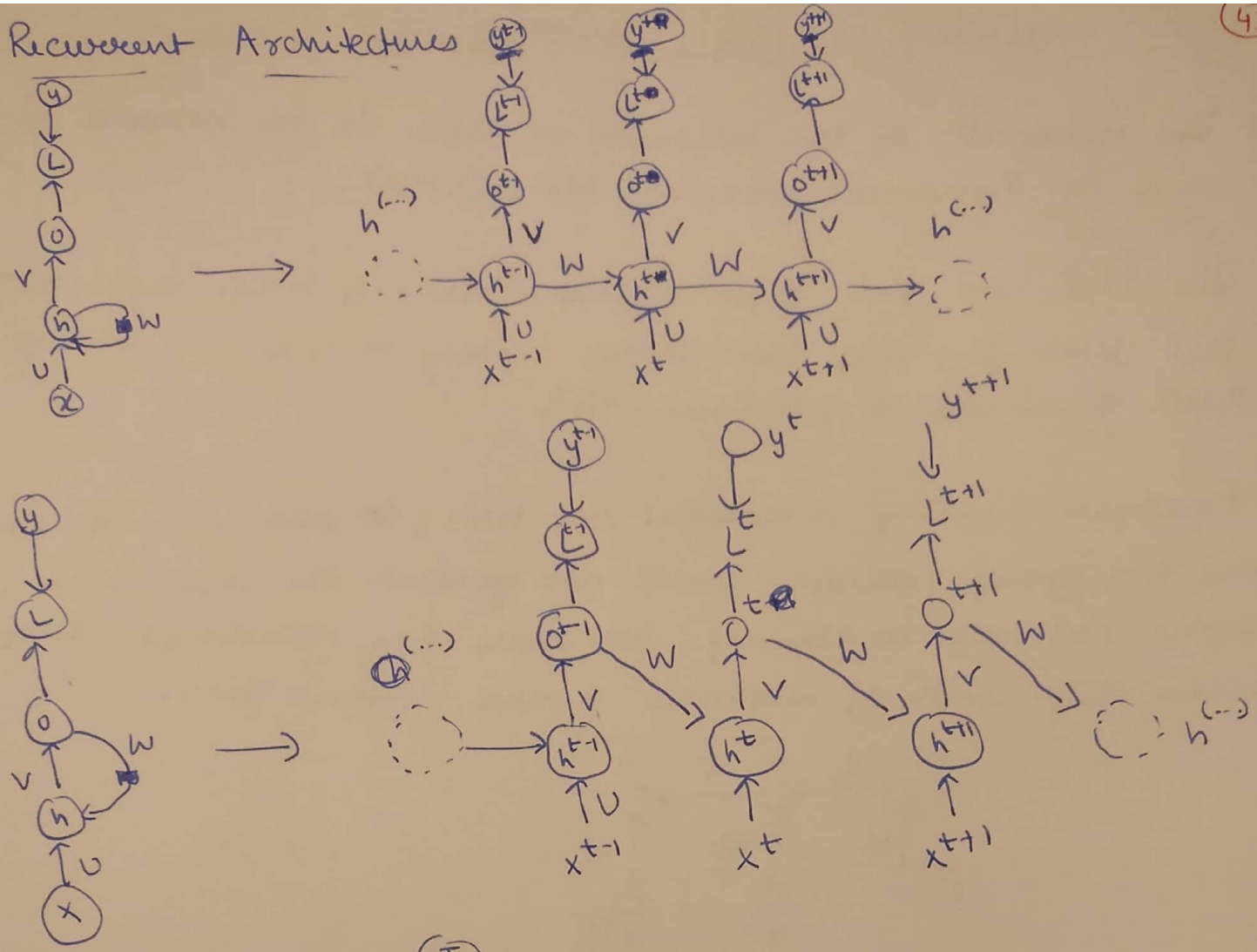
$$h(t) = f(h^{(t-1)}, x^{(t)}, \theta) = g(\ldots\ldots).$$

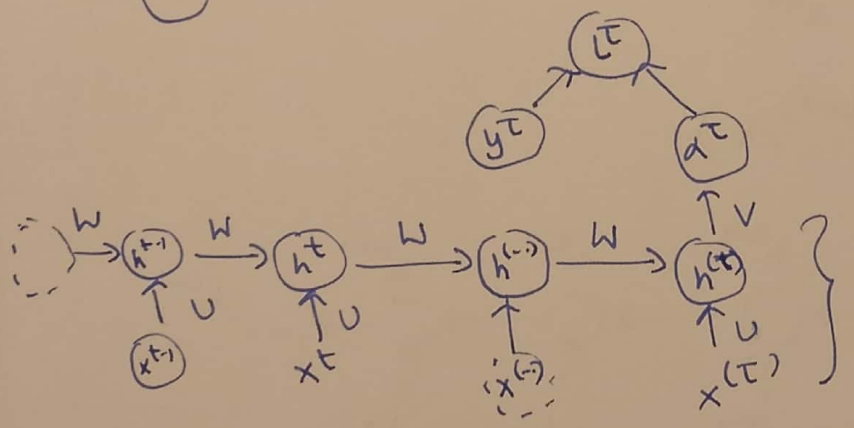
Ⓐ t is sized, ∵ it is specified in terms of transitn from one state to another, rather than variable-length history of states o/g

Ⓐ f can be used at any time 't', with same parameters, with "g" is time specific.

⇒ Needs less # training examples to estimate the model.

→ Recurrent Architectures





→ Encoder

Update:

$$a^{(t)} = b + W h^{(t-1)} + U x^{(t)} \qquad h^{(t)} = \tanh(a^{(t)})$$

$$o^{(t)} = c + V h^{(t)} \qquad \hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

→ $L^{(t)}$ as negative log likelihood of $y^{(t)}$ given $x^{(1)} \dots x^{(t)}$, then

$$L(\{x^{(1)} \dots x^{(\tau)}\}, \{y^{(1)} \dots, y^{(\tau)}\}) = \sum_{t=1}^{\tau} L^{(t)}.$$

$$= -\sum_{t} \log p_{\text{model}}\left(y^{(t)} \mid \{x^{(1)} \dots x^{(t)}\}\right)$$

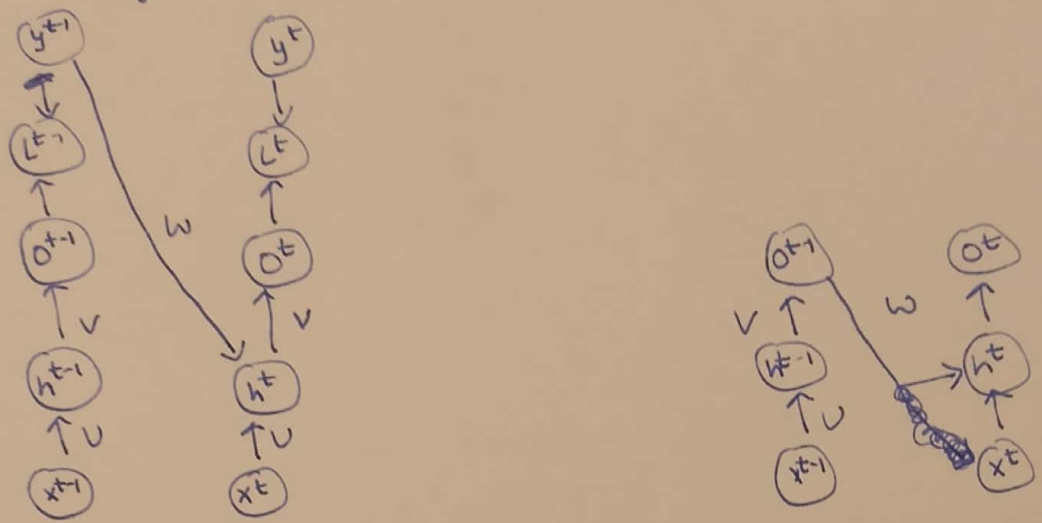
given by $y^{(t)}$ from $\hat{y}^{(t)}$

Back Propogatn through Time (BPTT) Derivation.

Teacher Forcing: Consider max likelihood criterion for training, where the model receives the true label $y(t)$ as i/p at $t+1$.

$$\log p(y^{(1)}, y^{(2)} \mid x^{(1)}, x^{(2)})$$

$$= \log p(y^{(2)} \mid x^{(1)}, x^{(2)}, y^{(1)}) + \log p(y^{(1)} \mid x^{(1)}, x^{(2)})$$



→ Due to this, training can be done in parallel as we know the labels first hand and they don't depend upon time.

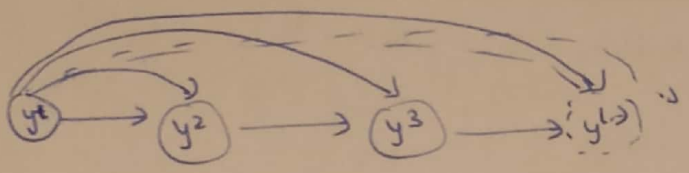
RNN's as Directed Graphs

→ If we consider the loss as max. log likelihood, and consider an RNN as a probabilistic graphical model, the graph would have no edges from any past $y^{(i)}$ to current $y^{(t)}$. Furthermore, y are all conditionally independent, given $x^{(i)}$.

→ But if the network has connections b/w previous o/p & current o/p, then log likelihood

$$P(y^{(t)} \mid x^{(1)} \ldots x^{(t)}, y^{(1)} \ldots y^{(t-1)}).$$

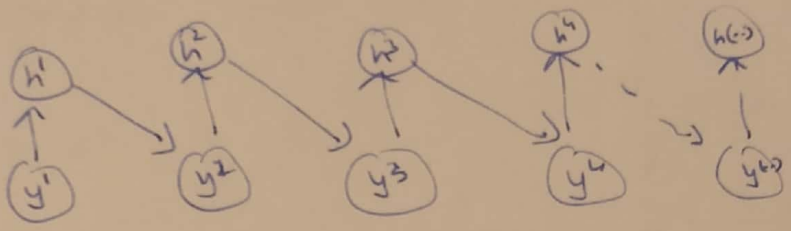
In such a case all $y^{(i)}$ does have connection with $y^{(j)}$, $\forall\ i<j$.



More Parameters

→ What would happen if we introduce a hidden units into such an RNN.

$$\Rightarrow h^{(t+1)} = f(h^{(t-1)}, y^{(t)})$$



Less Parameters.
Lossy Summary.

→ Good parametrizatn doesn't give you free lunch !!
That takes more time to train. (Because of BPTT)
↓
cumbersome nature.

makes an

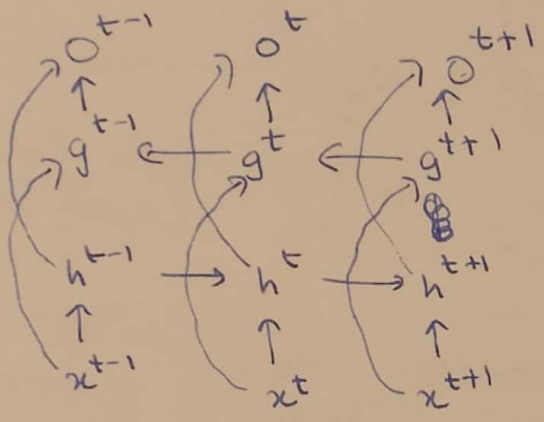
→ Also the models assumptions by parameter sharing that the sequence is stationary i.e. relationships b/w steps doesnot depend on the value of "t" itself, rather only the difference b/w the steps..

→ Sampling can be done from the conditional distribution at each time step.

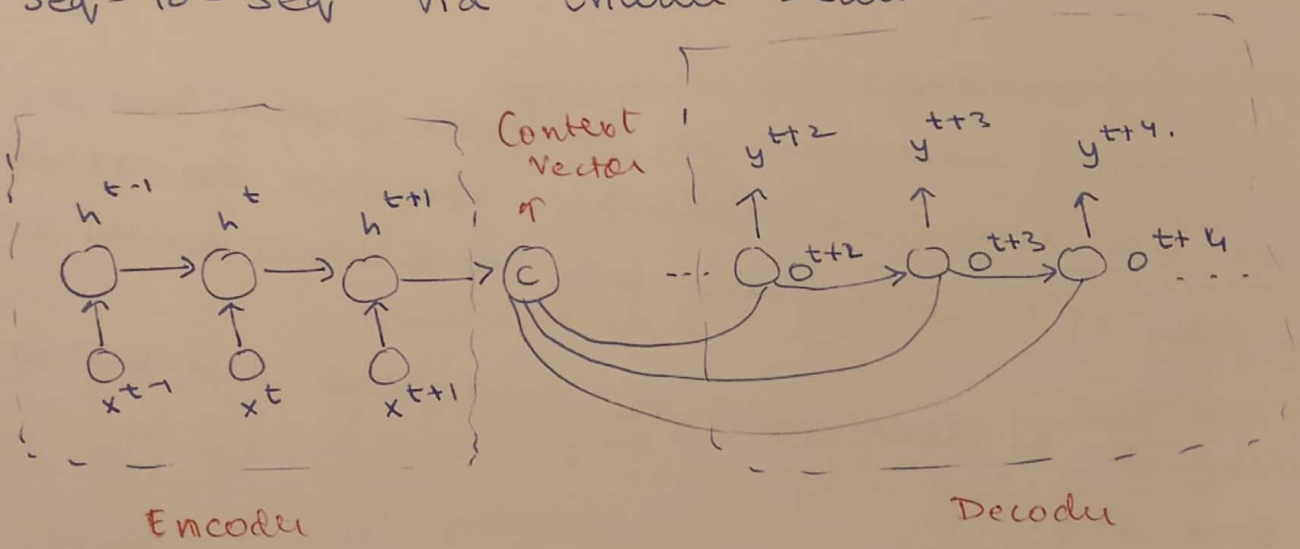
But when do you When to stop.

→ Always beneficial to know the length of the sequence.
→ can add, extra "stop" dimension in output (i.e. y)
→ introduce extra Bernoulli o/p at hidden layer / context layer which decides to continue / halt generating process
→ add an extra o/p in model to predict τ, which samples the value for τ and then sample for τ steps.

Bi-directional RNN.



Seq-to-Seq via Encoder-Decoder



Long term Dependencies (LSTM, GRU, Peephole)

Refer to colah blog for better understanding !! :)

→ Learning long-term dependencies is non-trivial:

① gradients tend to vanish or explode over many time steps

② Even if we have stable network, long term dependencies / interactn's, tend to get exponentially less wt.'s (as Jacobians are multiplied repeatedly) in comparison to short term memories

Leaky Units

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}, \theta).$$

$$h_L^t = \alpha h_L^{(t-1)} + (1-\alpha) f(h_L^{t-1}, x^{(t)}, \theta). \quad \text{where } \alpha \in [0, 1].$$

Gated Recurrence Unit (GRU). [Refer to Colah blog].

→ GRU computes reset gate $r^{(t)}$ and update gate $z^{(t)}$.

$$r^{(t)} = g_r(h^{t-1}, x^t, \theta_r) \qquad z^{(t)} = g_z(h^{t-1}, x^t, \theta_z).$$

Reset gate determines how much of the old memory should propagate (or kept or memorized)

$$\hat{a}^t = b + r^t W h^{t-1} + U x^t \qquad \hat{h}^{(t)} = \tanh(\hat{a}^t).$$

Now, update gate is applied to the memory content & the previous step.

$$h_G^t = z^t h_G^{t-1} + (1 - z^t) \hat{h}^t$$

↑ decides how much past state should matter

→ If $r^{(t)} \approx 0$, then we ignore the previous information and only consider current memory.

→ Has one advantage of forgetting information than the Leaky units.

LSTM (Long-Short Term Memory) Unit

3 Gates: an input gate $i^{(t)}$, a forgetting gate $f^{(t)}$, o/p gate $o^{(t)}$.

$$i^{(t)} = g_i(h^{(t-1)}, x^{(t)}, \theta_i)$$

$$f^{(t)} = g_f(h^{(t-1)}, x^{(t)}, \theta_f)$$

$$o^{(t)} = g_o(h^{(t-1)}, x^{(t)}, \theta_o).$$

→ Now, we define memory cell $c^{(t)}$ by gating previous memory and new content.

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)} \qquad c^{(t)} = f^{(t)} c^{(t-1)} + i^{(t)} \tanh(a^{(t)})$$

→ The final hidden state is a gated activation of memory cell:

$$h^{t}_{LSTM} = o^{(t)} \tanh(c^{(t)})$$

Peephole Connections

→ Allow gate layers to look at cell state.

$$i^{(t)} = g_i(h^{(t-1)}, c^{(t-1)}, x^{(t)}, \Theta_i)$$

$$f^{(t)} = g_f(h^{(t-1)}, c^{(t-1)}, x^{(t)}, \Theta_f)$$

$$o^{(t)} = g_o(h^{(t-1)}, c^{(t-1)}, x^{(t)}, \Theta_o)$$

→ Intuitively, taking into account what is in the state, when deciding which values to forget/update.