Deep Learning Models

Ziping Zhao

School of Information Science and Technology
ShanghaiTech University, Shanghai, China

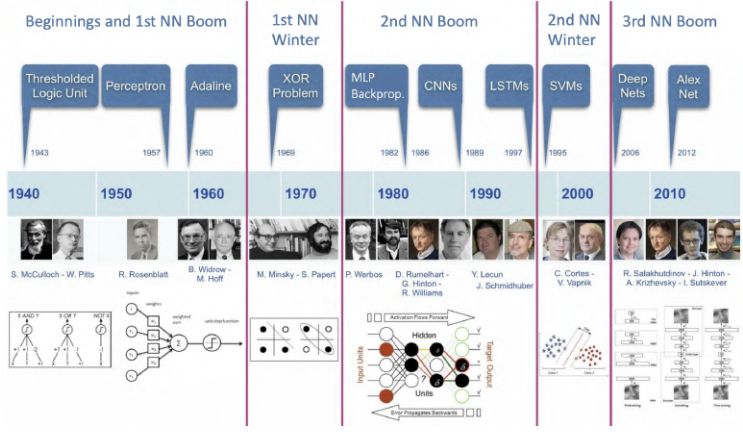
CS182: Introduction to Machine Learning (Fall 2022)
http://cs182.sist.shanghaitech.edu.cn

Ch. 12 of I2ML

Outline

Outline

A Brief History of Neural Networks

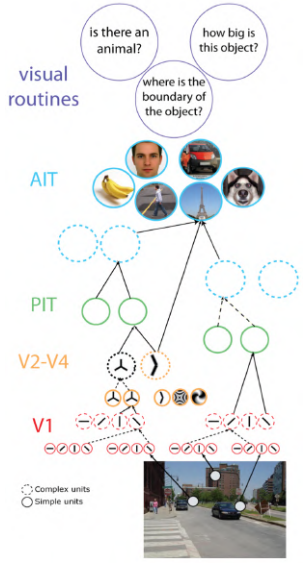


- The Adaptive Linear Neuron (Adaline) is a binary classifier and a single-layer NN (i.e., a simple perceptron) proposed in the field of signal processing for geophysics, adaptive antennas, adaptive filtering, adaptive prediction, and pattern recognition.

Introduction

- An effective way to overcome the curse of dimensionality is to reduce the dimensionality of the input data (see more later in Dimensionality Reduction topic).
- Converting low-level, high-dimensional input data into higher-level, lower-dimensional feature representation, i.e., the feature extraction problem, often plays a very crucial role in real applications.
- The models we have studied so far have rather shallow architecture which cannot represent very rich features.
- The human brain is known to have a much deeper layered architecture in which each level learns features and representations at increasing levels of abstraction, e.g., the physiological components involved in visual perception.

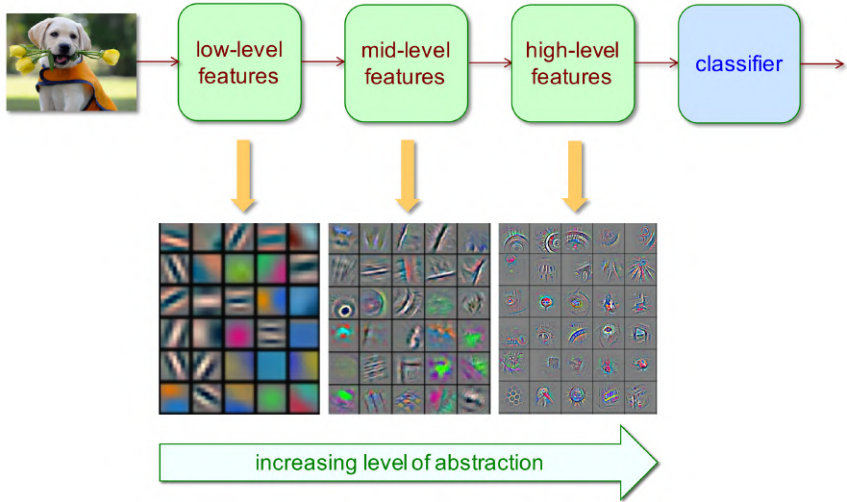
Layered Model of Visual Cortex



Learning Rich Feature Representations

- Traditionally, rich features specific to applications have always been handcrafted. This approach has shortcomings:
  - It is a very tedious and time-consuming human-engineered process.
  - Usually, highly application-dependent features handcrafted for one application cannot be transferred easily to other applications.
- Deep learning (a.k.a. representation learning) seeks to learn rich features automatically through a feature learning process which may be unsupervised or supervised.
- The word 'deep' in deep learning refers to the layered model architectures which are usually deeper than conventional learning models.

Low-Level to High-Level Features



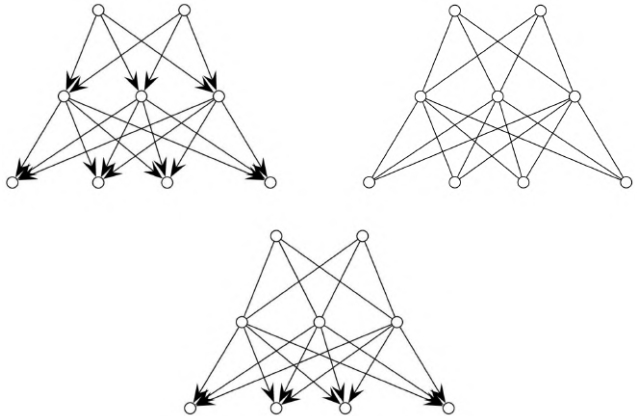
Some Deep Learning Models

- Deep generative models, e.g.
  - Deep directed networks
  - Deep Boltzmann machines
  - Deep belief networks
- Deep neural networks, e.g.
  - Deep multilayer perceptrons
  - Deep autoencoders
  - Stacked denoising autoencoders
  - Convolutional neural networks

Outline

Deep Generative Models for Unsupervised Learning

- Directed, undirected, and mixed models:

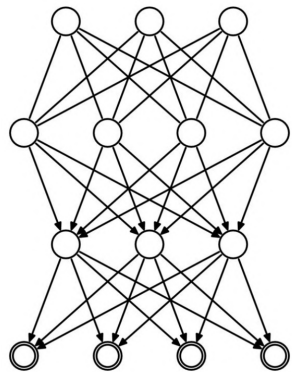


Deep Belief Networks I

- Problems with back-propagation for MLP training:
  - It requires labeled training data. (Almost all data is unlabeled.)
  - The learning time does not scale well. (It is very slow in networks with multiple hidden layers.)
  - It can get stuck in poor local optima.
- In the 1990's, many researchers abandoned neural networks with multiple adaptive hidden layers because SVMs worked better.

Deep Belief Networks II

- A deep belief network (DBN) is a partially directed and partially undirected probabilistic generative model.
- Directed edges in all layers except the topmost one which corresponds to an undirected bipartite graph.
- Two consecutive layers connected by undirected edges correspond to a model called restricted Boltzmann machine (RBM).
- DBN is a model for the pretraining of an MLP.

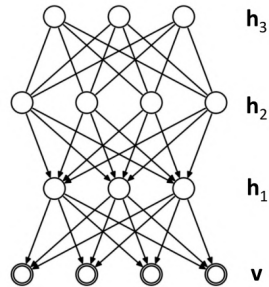


Joint Distribution

- Consider a DBN with 1 visible layer and 3 hidden layers.
- Joint distribution:

$$
\begin{aligned}
& p(\mathbf{v}, \mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3 \mid \boldsymbol{\theta}) \\
&= \prod_i \mathrm{Ber}(v_i \mid \sigma(\mathbf{h}_1^T \mathbf{w}_{1i})) \prod_j \mathrm{Ber}(h_{1j} \mid \sigma(\mathbf{h}_2^T \mathbf{w}_{2j})) \\
&\quad \times \frac{1}{Z(\boldsymbol{\theta}) \exp\left(\sum_{kl} h_{2k} h_{3l} W_{3kl}\right)}
\end{aligned}
$$

where $\boldsymbol{\theta}$, $\mathrm{Ber}(\cdot \mid \cdot)$, and $\sigma(\cdot)$ denote the network weights, Bernoulli distribution, and sigmoid function, respectively, and $Z(\boldsymbol{\theta})$ serves as a normalization factor.



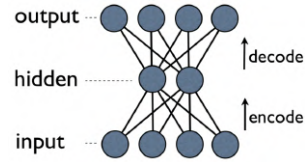
Model Inference

- The hidden states can be inferred in an efficient bottom-up fashion.
- Training of the RBMs is done in a greedy layer-wise manner starting from the bottommost layer.
- It can be shown that this inference procedure increases a lower bound of the observed data likelihood.
- After the greedy layer-wise training procedure, it is common to fine-tune the weights using a technique called backfitting which involves both bottom-up and top-down operations. However, this procedure is very slow.

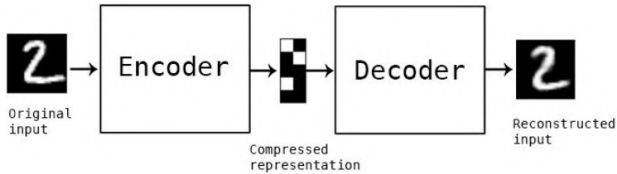
Outline

Autoencoders I

- An autoencoder (AE) is a feedforward neural network for learning a compressed representation or latent representation (encoding) of the input data by learning to predict the input itself in the output.



- The hidden layer in the middle is constrained to be a narrow bottleneck (with fewer units than the input and output layers) to ensure that the hidden units capture the most relevant aspects of the data.
- The input-to-hidden part corresponds to an encoder while the hidden-to-output part corresponds to a decoder.



Autoencoders II

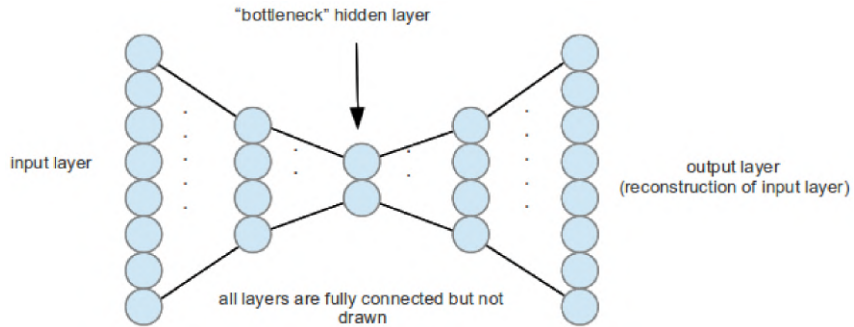
- Minimize the reconstruction loss

$$\underset{\mathbf{W}_1,\mathbf{W}_2,\mathbf{w}_{01},\mathbf{w}_{02}}{\text{minimize}} \quad \frac{1}{2}\sum_{\ell=1}^{N}\left\|\mathbf{x}^{(\ell)} - \underbrace{(\mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x}^{(\ell)}+\mathbf{w}_{10})+\mathbf{w}_{20})}_{=\hat{\mathbf{x}}^{(\ell)}}\right\|_2^2$$

- For linear hidden units, it can be shown that the weights to the k hidden units span the same subspace as the first k principal components of the data, i.e., linear autoencoders are equivalent to principal component analysis (PCA) (to be covered in Dimensionality Reduction topic).
- Nonlinear representations of the data can be obtained by using nonlinear hidden units, e.g., units with the sigmoid function.

Deep Autoencoders I

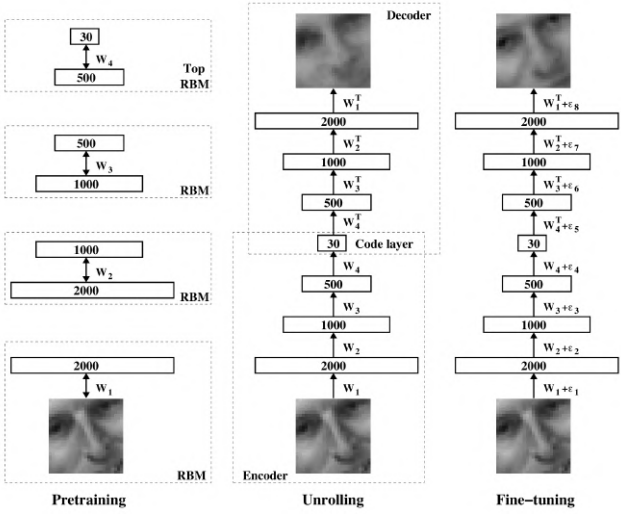
- A deep autoencoder (or stacked autoencoder) is a multilayer extension of the simple autoencoder by using multiple hidden layers.
- The middle layer still acts as a bottleneck.



Deep Autoencoders II

- Training deep autoencoders using the standard backpropgation learning algorithm does not work well because:
  - the gradient signal becomes too small as it passes back through multiple layers
  - the learning algorithm often gets stuck in poor local minima.
- One solution is to train a series of RBMs through a pretraining procedure and then use them to initialize an autoencoder. Afterwards, the whole network is fine-tuned using backpropgation in the usual way.

Pretraining, Unrolling, and Fine-tuning

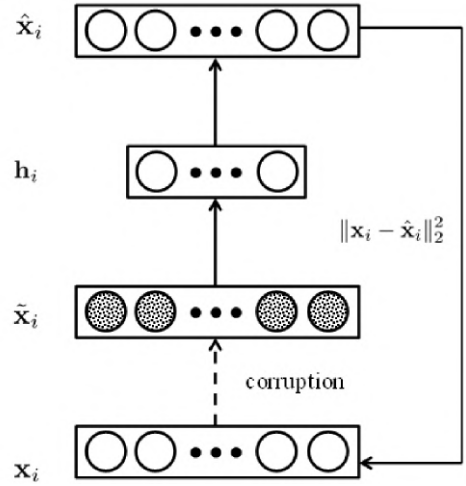


Outline

Denoising Autoencoders I

- To prevent the model from simply learning the identity mapping, conventional autoencoders use a bottleneck hidden layer to produce an under-complete representation.
- Another possibility, based on sparse coding, is to produce an over-complete yet sparse representation by imposing sparsity constraints on the hidden units.
- Yet another possibility is to add noise to the input, leading to a denoising autoencoder which learns a good representation by cleaning partially corrupted input.

Denoising Autoencoders II



Ideas Underlying Denoising Autoencoders

- A higher-level representation should be rather stable and robust under corruptions of the input.
- Performing the denoising task well requires extracting features that capture useful structure in the input distribution.
- Denoising is not the primary goal. It is advocated and investigated as a training criterion for learning to extract useful features that will constitute better higher-level representation.

Optimization Problem

- A denoising autoencoder solves the following (regularized) optimization problem:

$$\underset{\mathbf{W}_1,\mathbf{W}_2,\mathbf{w}_{01},\mathbf{w}_{02}}{\text{minimize}} \quad \frac{1}{2}\sum_{\ell} \|\mathbf{x}^{(\ell)} - \hat{\mathbf{x}}^{(\ell)}\|_2^2 + \lambda\Big(\|\mathbf{W}_1\|_F^2 + \|\mathbf{W}_2\|_F^2\Big)$$

where

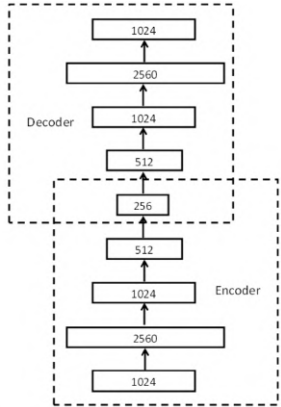
$$\mathbf{h}^{(\ell)} = \sigma(\mathbf{W}_1\tilde{\mathbf{x}}^{(\ell)} + \mathbf{w}_{01})$$
$$\hat{\mathbf{x}}^{(\ell)} = \sigma(\mathbf{W}_2\mathbf{h}^{(\ell)} + \mathbf{w}_{02})$$

Here λ is a regularization parameter and $\|\cdot\|_F^2$ is a matrix norm, called the Frobenius norm, which is defined as

$$\|\mathbf{A}\|_F^2 = \sum_{i,j} a_{ij}^2 \qquad \text{where } \mathbf{A} = [a_{ij}] \text{ is a matrix}$$

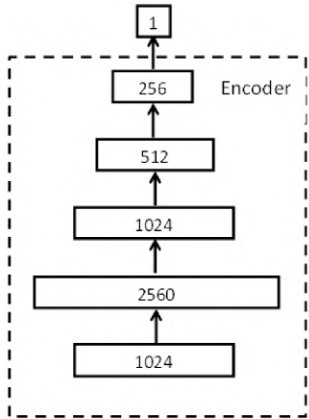
Stacked Denoising Autoencoders

- Stacked denoising autoencoders can be formed by stacking the denoising autoencoders in a layer-wise manner like deep autoencoders with the same pretraining, unrolling, and fine-tuning steps.



Encoder for Feature Extraction

- The encoder part of the stacked denoising autoencoder will then be used as a feature extractor for the subsequent task, such as classification, by performing further model training.



Outline

Convolutional Neural Networks

- The development of convolutional neural networks (CNN), or convolutional networks (ConvNet), has been inspired by neurophysiological experiments that David H. Hubel and Torsten N. Wiesel (winners of the Nobel Prize in Physiology or Medicine 1981) conducted in the 50s and 60s to understand how the mammalian vision system works.
- Some neurons in the visual cortex will be associated with specific local receptive fields.
- CNNs are a special type of feedforward neural networks for processing data with a known grid-like topology, e.g.
  - Spatial data: 2 spatial dimensions
  - Spatiotemporal data: 1 spatial dimension, 1 temporal dimension
- At least one layer (called convolutional layer) in a CNN uses convolution in place of matrix multiplication.

Convolution Operation

- Convolution is a linear mathematical operation on two functions producing a third function, widely used in signal processing.
- Convolution of functions f and g defined on continuous domains:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau$$

- Discrete convolution of functions x and w defined on discrete domains:

$$(x * w)[n] = \sum_{m=-\infty}^{\infty} x[m]w[n-m] = \sum_{m=-\infty}^{\infty} x[n-m]w[m]$$

Convolution in CNNs

- Finite 2-D domains:

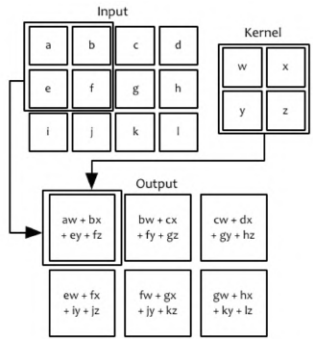
$$s[i,j] = (x * w)[i,j] = \sum_{m=-M}^{M} \sum_{n=-N}^{N} x[m,n]w[i-m, j-n]$$
$$= \sum_{m=-M}^{M} \sum_{n=-N}^{N} x[i-m, j-n]w[m,n]$$

- Functions involved in convolution operation:
  - Input (2-D array of data): x
  - Convolution kernel (2-D array of learnable parameters): w
  - Feature map (2-D array of processed data): s

Convolution with no Kernel Flipping

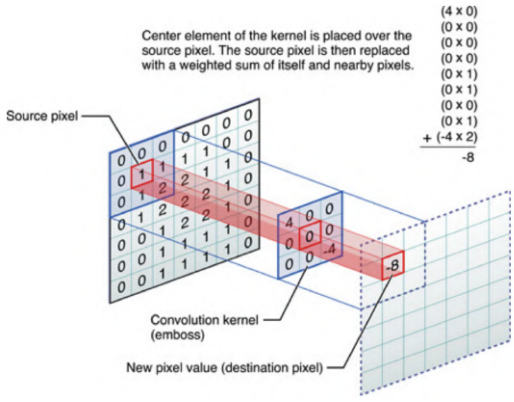
- In typical implementations, cross-correlation (same as convolution but without flipping the kernel) is usually used instead:

$$s[i,j] = (x * w)[i,j] = \sum_{m=-M}^{M} \sum_{n=-N}^{N} x[i+m, j+n] w[m,n]$$



Spatial Convolution

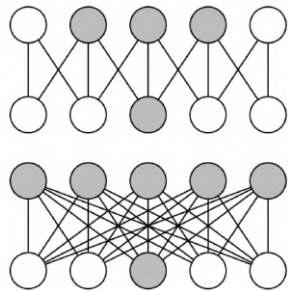
- Spatial convolution as a linear spatial filter:



- Typically there are multiple feature maps, one for each convolution operator.

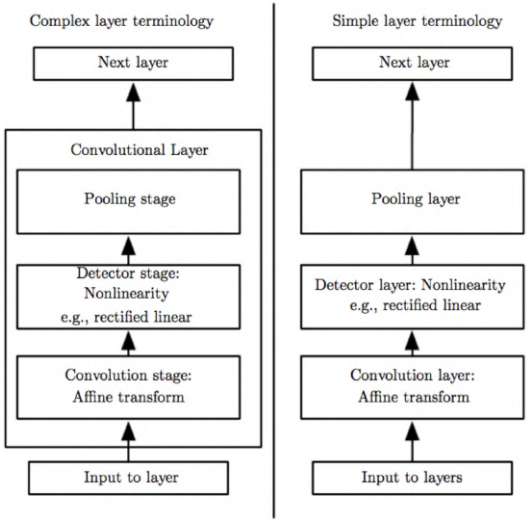
Some Properties of Convolution

- Sparse connectivity:
  - The kernel is (much) smaller than the input.
  - Sparse weights in CNNs vs. fully-connected weights in conventional neural networks:



- Parameter sharing:
  - The parameters of a kernel are the same when applied to different locations of the input.

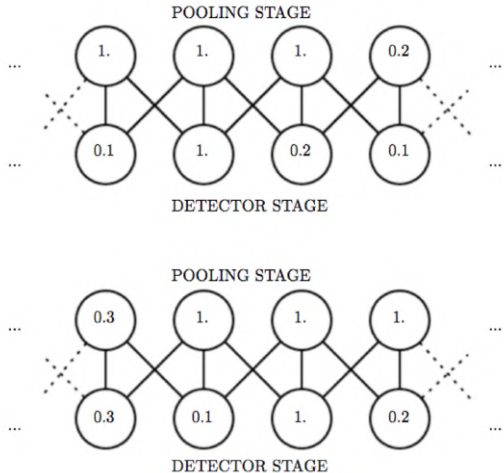
Three Stages of a Convolutional Layer



Pooling

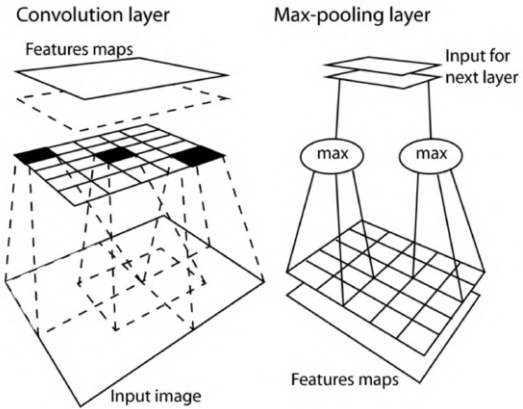
- A pooling function replaces the output at a certain location with a summary statistic of the nearby outputs.
- It helps to make the higher-level representation invariant to small changes of the input and hence more robust.
- Common pooling operations:
  - Max pooling: reports the maximum output within a rectangular neighborhood.
  - Average pooling: reports the average output of a rectangular neighborhood (possibly weighted by the distance from the central pixel).
- By spacing pooling regions $k > 1$ (rather than 1) pixels apart, the next higher layer has roughly k times fewer inputs to process, leading to downsampling.

Local Translational Invariance

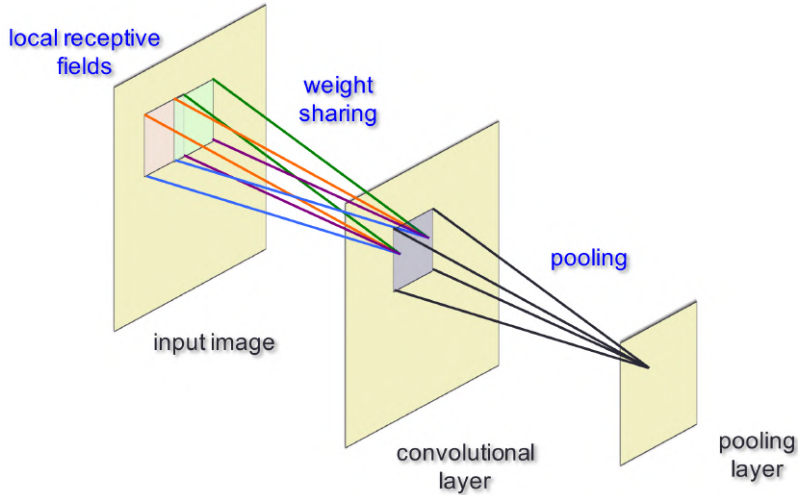


Convolution and Pooling I

- Convolution: feature detector
- Max pooling: local translational invariance



Convolution and Pooling II



Deep CNNs

- There are multiple feature maps per convolutional layer.
- There are multiple convolutional layers for extracting features at different levels.
- Higher-level layers take the feature maps in lower-level layers as input.

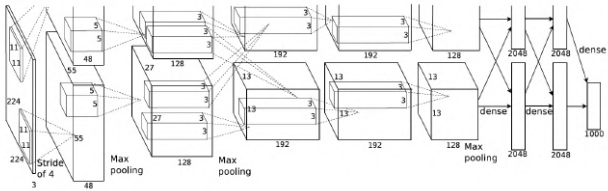
- The hierarchical feature representations are robust nonlinear spatial filters that can tolerate local distortions.
- Representative deep CNNs: LeNet ('98), AlexNet ('12), VGG ('14), GoogLeNet ('14), ResNet ('15), DenseNet ('17)... (You can take a course on "Deep Learning for Computer Vision".)

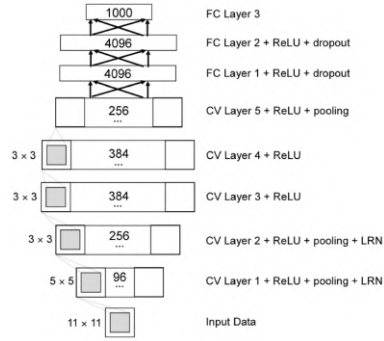
CNN Learning

- CNNs are trained in a supervised manner using labeled data.
- The backpropagation learning algorithm based on mini-batch gradient descent (MBGD) is usually used for training.
- Huge amount of labeled training data is often needed for training deep CNNs with many learnable parameters.

AlexNet: A Winning CNN in ImageNet Challenge 2012

- 5 convolutional (CV) layers and 3 fully-connected (FC) layers





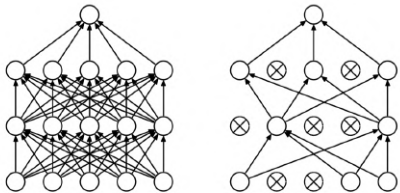
  - Rectified linear unit (ReLU) is an activation function defined as the positive part of its argument $\sigma(x) = x_+ = \max(0, x)$.
  - Local response normalization (LRN) is a contrast enhancement process for feature maps in CNNs.

Dropout

- Deep models have many learnable parameters and hence are prone to overfitting.
- To prevent overfitting, one way is to use a very large dataset but this is not desirable for supervised learning methods (e.g., CNN) which need labeled training data.
- A powerful alternative is the method called dropout which may be seen as a computationally very cheap realization (or approximation) of an ensemble method called bagging (to be covered later).

Dropout in CNNs

- Dropout is very powerful because it effectively trains and evaluates a bagged ensemble of exponentially many deep models which are sub-networks formed by removing (randomly with a dropout rate of say 0.5) units or weights (achieved simply via multiplying their values by zero) from the original CNN.
- Dropout is applied to the fully-connected layers of a CNN.



(a) Standard Neural Net (b) After applying dropout.

- Dropout is actually a general regularization technique which can also be applied to other models that use a distributed representation and can be trained with stochastic gradient descent, e.g., feedforward neural networks, recurrent neural networks, and probabilistic models such as RBM.

Outline

Introduction to Recurrent Neural Networks

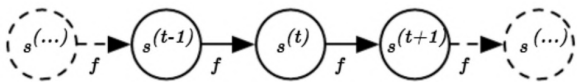
- Recurrent neural networks (RNNs) are extensions of feedforward neural networks for handling sequential data (such as sound, time series (sensor) data, or written natural language) typically involving variable-length input or output sequences.
- There is weight sharing in an RNN such that the weights are shared across different instances of the units corresponding to different time steps.
- Weight sharing is important for processing variable-length sequences so that the absolute time step at which an event occurs does not matter but the context (relative to some other events) in which an event occurs is more important and relevant.

Computational Graphs I

- The computation involved in a dynamical system defined recursively can be unfolded to form a computational graph with a repetitive structure such as a chain of events.
- A classical dynamical system:

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}; \boldsymbol{\theta})$$

  where $\mathbf{s}^{(t)}$ denotes the state of the system at time t and $\boldsymbol{\theta}$ denotes the system parameters which are time-invariant (effectively achieving weight sharing).
- Corresponding computational graph:

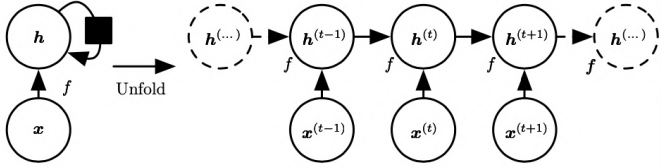


Computational Graphs II

- A more general dynamical system with an external input $\mathbf{x}^{(t)}$:

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

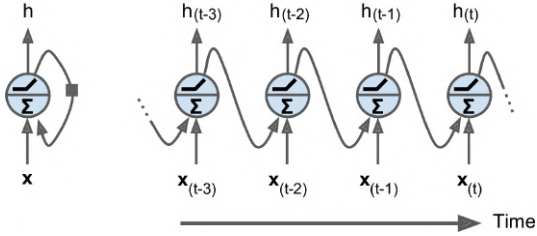
Unfolding a recurrent structure into a computational graph:



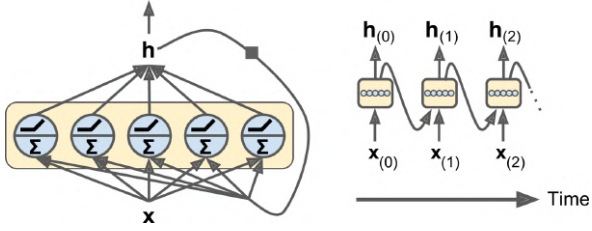
- Computations occurring at different time steps of the recurrent structure (where the black square indicates a delay of 1 time step) are represented as different nodes in the computational graph.
- the hidden state $\mathbf{h}^{(t)}$ may be thought of as a kind of (lossy) summary of the past input sequences up to time t, i.e., $\mathbf{x}^{(\tau)}$ for $\tau \leq t-1$.

Recurrent Neurons

- A recurrent neuron with a scalar output

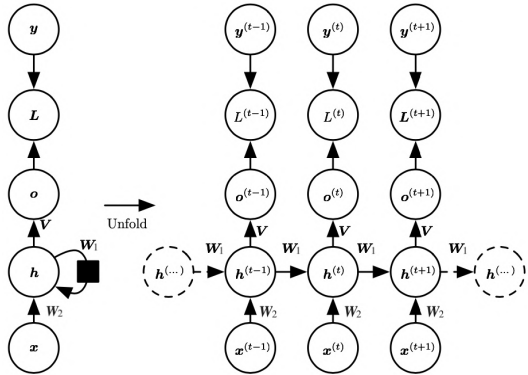


- A layer of recurrent neurons with a vector output



Computational Graphs for RNNs

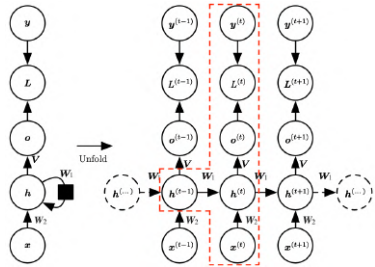
- Computational graphs for showing the information flow in RNNs:
  - Forward in time: for computing outputs and losses
  - Backward in time: for computing gradients
- Unfolding an RNN (with hidden-to-hidden recurrence) into a computational graph:



Forward Propagation

- Forward propagation equations for classification problems:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}_1\mathbf{h}^{(t-1)} + \mathbf{W}_2\mathbf{x}^{(t)}$$
$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)})$$
$$- - - - - - - - - - - - - -$$
$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$$
$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)})$$



where the weight matrices $\mathbf{W}_1$, $\mathbf{W}_2$, and $\mathbf{V}$ are the hidden-to-hidden, input-to-hidden, and hidden-to-output connections and $\mathbf{b}$ and $\mathbf{c}$ denote the bias vectors.

- In this model, the input sequence and the output sequence are of the same length.
- Total loss:

$$L(\mathbf{W}_1, \mathbf{W}_2, \mathbf{V}, \mathbf{b}, \mathbf{c} \mid \mathbf{x}, \mathbf{y}) = \sum_t L^{(t)}(\mathbf{W}_1, \mathbf{W}_2, \mathbf{V}, \mathbf{b}, \mathbf{c} \mid \mathbf{x}^{(t)}, \mathbf{y}^{(t)})$$

Back-Propagation Through Time

- The backpropagation learning algorithm for feedforward neural networks can be generalized to work on the computational graphs for RNNs, leading to the back-propagation through time (BPTT) algorithm.
- Different gradient-based techniques may be used by BPTT for RNN training.
- The following gradients (used by gradient-based techniques) can be computed recursively (details ignored) like the original backpropagation algorithm:

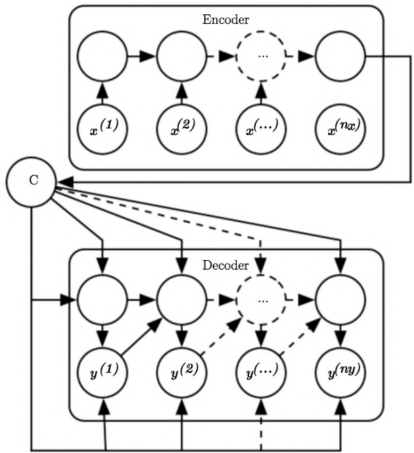
$$\frac{\partial L}{\partial \mathbf{U}}, \quad \frac{\partial L}{\partial \mathbf{V}}, \quad \frac{\partial L}{\partial \mathbf{W}}, \quad \frac{\partial L}{\partial \mathbf{b}}, \quad \frac{\partial L}{\partial \mathbf{c}}$$

Outline

Encoder-Decoder RNN I

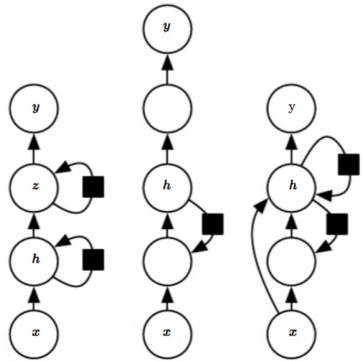
- An encoder-decoder RNN (or sequence-to-sequence learning architecture) learns to map an input sequence to an output sequence not necessarily of the same length (unlike the RNN above).
- Encoder RNN:
  - Processes the input sequence $\mathbf{X} = (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n_x)})$
  - Generates a (usually fixed-length) internal representation which is a simple function of the hidden state of the last hidden layer.
- Decoder RNN:
  - Takes the internal representation as input
  - Generates the output sequence $\mathbf{Y} = (\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n_y)})$.
- In general the lengths n_x and n_y can vary from training pair to training pair.
- The two RNNs are trained jointly over all the training pairs.

Encoder-Decoder RNN II



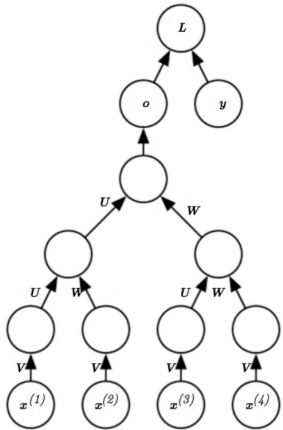
Deep RNNs

- Many ways to increase the network depth, e.g.,
  - Multiple recurrent hidden layers
  - Deeper computation in recurrent layers
  - Mixture of different recurrent connection types (path-lengthening effect may be mitigated by introducing skip connections)



Recursive Neural Networks I

- Recursive neural networks generalize ordinary RNNs in that the unfolded computational graphs are no longer chains but trees.

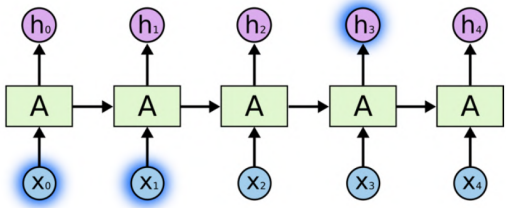


Recursive Neural Networks II

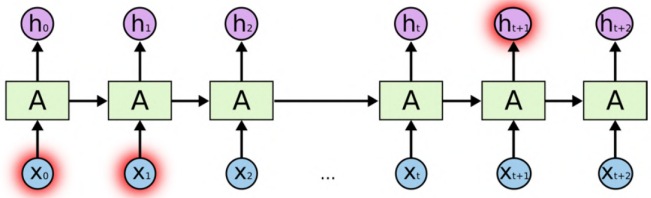
- Recursive neural networks can take data structures that are more general than feature vectors as input.
- They have been used successfully for natural language processing and computer vision applications. (You can take a course on "Deep Learning for Natural Language Processing".)
- To process a sequence of length N, an RNN has a length of N but a recursive neural network reduces it to $O(\log N)$. This helps the latter to deal with long-term dependencies more effectively.

Dependencies between Events in RNNs

- Short-term dependencies:



- Long-term dependencies:



Long-Term Dependencies

- Learning long-term dependencies in RNNs is challenging because:
  - the gradients propagated over many stages tend to vanish (most of the time) or explode (sometimes)
  - exponentially smaller weights are given to long-term interactions involving the multiplication of many Jacobians.
- The use of long short-term memory (LSTM) cells is one way proposed to help the learning of long-term dependencies in RNNs.

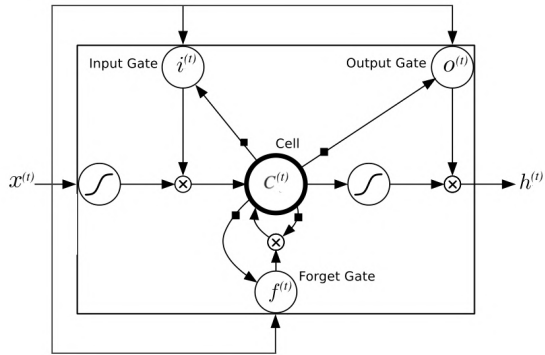
Long Short-Term Memory

- LSTM is based on the idea of leaky units which allow an RNN to accumulate information over a longer duration (e.g., we use a leaky unit to accumulate evidence for each subsequence inside a sequence).
- Once the information accumulated is used (e.g., suffcient evidence has been accumulated for a subsequence), we need a mechanism to forget the old state by setting it to zero and starting to count again from scratch.
- Instead of manually deciding when to clear the state, LSTM provides a way to learn to decide when to do it by conditioning the forgetting on the context.
- Gradients can flow for long durations through a linear self-loop whose weight is controlled by another hidden unit to change the time scale of the integration dynamically.

LSTM Cell

- An LSTM recurrent neural network consists of LSTM cells as hidden units which have an internal recurrence in addition to the recurrence in the recurrent network.
- LSTM RNN is a kind of gated RNN.

- Each LSTM cell has the same inputs and outputs as an ordinary RNN cell, but has more parameters and a system of gating units to control the information flow.
- 3 gates (or gate controllers) in an LSTM cell: external input gate, forget gate, output gate



a peephole LSTM cell

Forget and Input Gates

- For time step t, there is a cell state (or internal state) unit $C^{(t)}$ ("long-tem memory") that has a linear self-loop whose weight is controlled by a forget gate unit $f^{(t)}$ by setting the weight to a value in $(0, 1)$ via a sigmoid unit σ:

$$f^{(t)} = \sigma(b_f + W_{f1}C^{(t-1)} + W_{f2}x^{(t)}) = \sigma(b_f + W_f[C^{(t-1)}; x^{(t)}])$$

  where $x^{(t)}$ is the current input vector, b_f, W_{f1}, and W_{f2} are the bias, recurrent weight, and input weight of the forget gate, and $W_f = [W_{f1}, W_{f2}]$.

- The external input gate unit $i^{(t)}$ is computed similarly to the forget gate:

$$i^{(t)} = \sigma(b_i + W_{i1}C^{(t-1)} + W_{i2}x^{(t)}) = \sigma(b_i + W_i[C^{(t-1)}; x^{(t)}])$$

  where b_i, W_{i1}, and W_{i2} are the bias, recurrent weight, and input weight of the input gate, and $W_i = [W_{i1}, W_{i2}]$.

Output Gate

- State update or forward equations:

$$C^{(t)} = f^{(t)} C^{(t-1)} + i^{(t)} \underbrace{\tanh(b + Wx^{(t)})}_{\tilde{s}^{(t)}}$$

  where b and W are the bias and input weight into the LSTM cell.

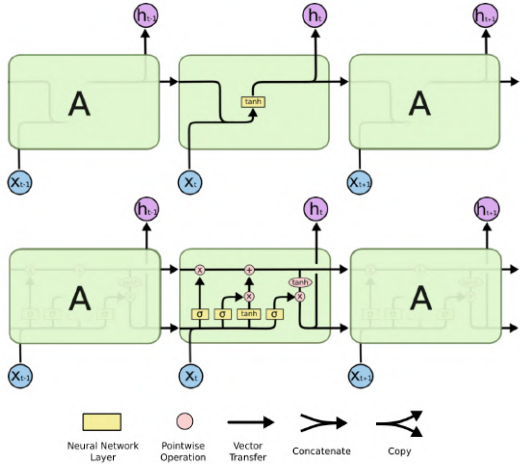
- The output gate unit $o^{(t)}$ can be used to shut off the output hidden state $h^{(t+1)}$ ("short-tem memory") of the LSTM cell:

$$o^{(t)} = \sigma(b_o + W_{o1} C^{(t-1)} + W_{o2} x^{(t)}) = \sigma(b_o + W_o [C^{(t-1)}; x^{(t)}])$$
$$h^{(t)} = o^{(t)} \tanh(C^{(t)})$$

  where b_o, W_{o1}, and W_{o2} are its bias, recurrent weight, and input weight, and $W_o = [W_{o1}, W_{o2}]$.

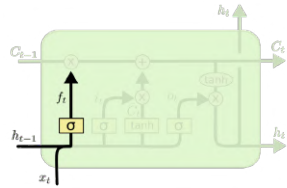
RNNs with ordinary cells vs. RNNs with LSTM cells

- RNN computational graph with another structure of an LSTM cell is as follows.



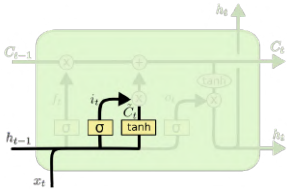
Schematic Illustration: Forget Gate and Input Gate

- Forget gate:



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

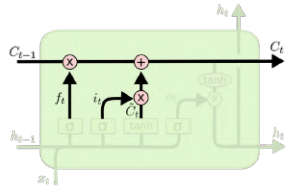
- External Input gate:



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

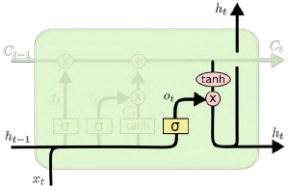
Schematic Illustration: Cell State and Output Gate

- Cell state:

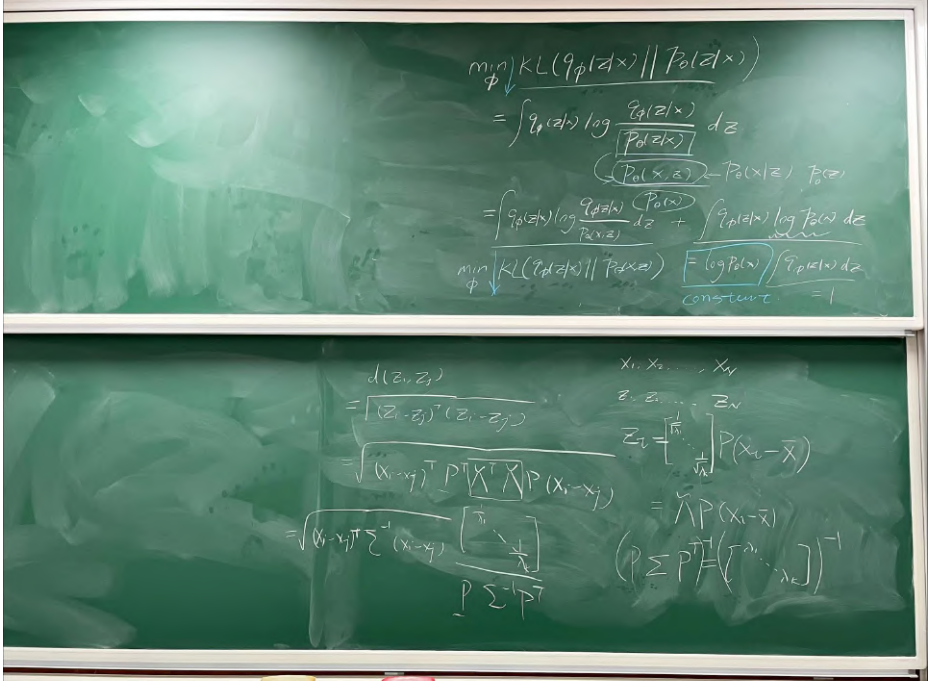


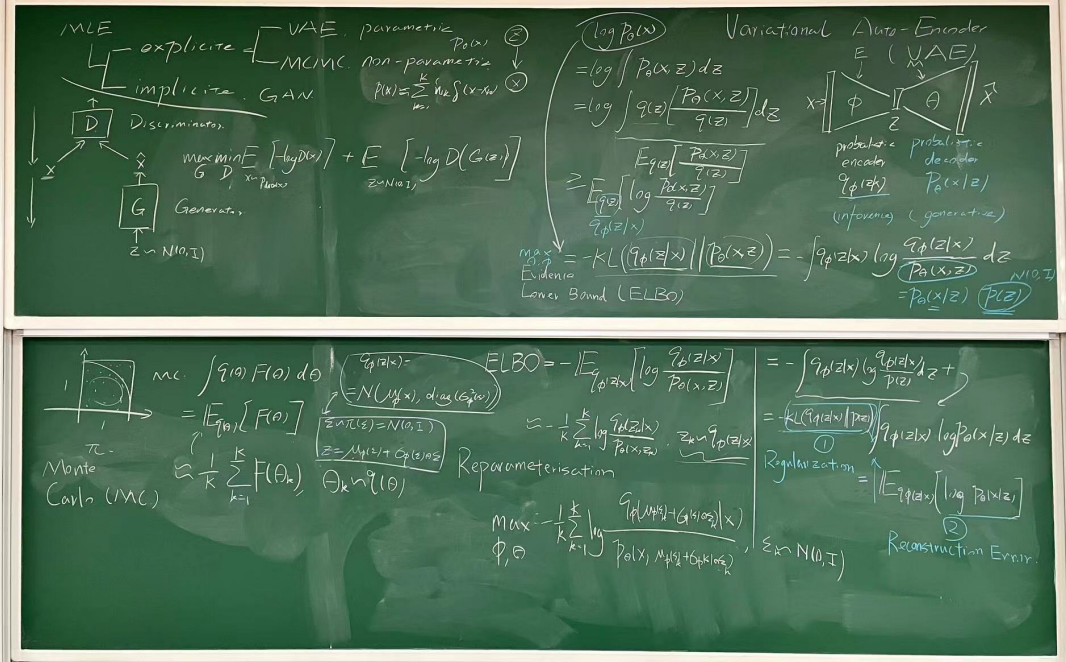
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- Output gate:



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$


$\min_\phi KL(q_\phi(z|x) \| p_\theta(z|x))$
$= \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(z|x)} dz$
$p_\theta(x,z) \leftarrow p_\theta(x|z)\, p_\theta(z)$
$= \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(x,z)} dz + \int q_\phi(z|x) \log p_\theta(x)\, dz$
$\min_\phi KL(q_\phi(z|x) \| p_\theta(z|x))$
$= \log p_\theta(x) \int q_\phi(z|x)\, dz$
constant $= 1$
$d(z_i, z_j)$
$= \sqrt{(z_i - z_j)^T (z_i - z_j)}$
$= \sqrt{(x_i - x_j)^T P^T X^T X P (x_i - x_j)}$
$= \sqrt{(x_i - x_j)^T \Sigma^{-1} (x_i - x_j)}$
$P \Sigma^{-1} P^T$
$x_1, x_2, \ldots, x_N$
$z_1, z_2, \ldots, z_N$
$Z_1 = \begin{bmatrix} \frac{1}{\sqrt{\lambda_1}} & & \\ & \ddots & \\ & & \frac{1}{\sqrt{\lambda_K}} \end{bmatrix} P (x_1 - \bar{x})$
$= \Lambda P (x_1 - \bar{x})$
$\left(P \Sigma P^T = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_K \end{bmatrix} \right)^{-1}$


MLE
explicite
VAE: parametric
MCMC: non-parametric
implicite: GAN
D
Discriminator
G
Generator
$z \sim N(0, I)$
$\max_G \min_D E_{x \sim P_{data}}[\log D(x)] + E_{z \sim N(0,I)}[-\log D(G(z))]$
$P_\theta(x)$
$P(x) = \sum_{k=1}^{k} \pi_k \phi(x - x_k)$
$\log P_\theta(x)$
$= \log \int P_\theta(x, z)\, dz$
$= \log \int q(z) \left[\frac{P_\theta(x,z)}{q(z)}\right] dz$
$\geq E_{q(z)}\left[\log \frac{P_\theta(x,z)}{q(z)}\right]$
$E_{q(z)}\left[\frac{P_\theta(x,z)}{q(z)}\right]$
$q_\phi(z|x)$
Variational Auto-Encoder
E (VAE)
probabilistic encoder
probabilistic decoder
$q_\phi(z|x)$
$P_\theta(x|z)$
(inference)
(generative)
$\max_{\theta,\phi}$ Evidence Lower Bound (ELBO)
$= -KL(q_\phi(z|x) \| P_\theta(x,z)) = -\int q_\phi(z|x) \log \frac{q_\phi(z|x)}{P_\theta(x,z)}\, dz$
$= P_\theta(x|z)\, P(z)$
$N(0,I)$
MC
$\int q(\theta) F(\theta)\, d\theta$
$= E_{q(\theta)}[F(\theta)]$
$\approx \frac{1}{k} \sum_{k=1}^{k} F(\theta_k)$
$\theta_k \sim q(\theta)$
Monte Carlo (MC)
$q_\phi(z|x) = N(\mu_\phi(x), \sigma(\sigma_\phi^2(x)))$
$z \sim N(0,1)$
$z = \mu_{\phi(z)} + \sigma_\phi(z) \cdot \varepsilon$
ELBO $= -E_{q_\phi(z|x)}\left[\log \frac{q_\phi(z|x)}{P_\theta(x,z)}\right]$
$\approx -\frac{1}{k} \sum_{k=1}^{k} \log \frac{q_\phi(z_k|x)}{P_\theta(x, z_k)}$
$z_k \sim q_\phi(z|x)$
Reparameterisation
$\max_{\phi,\theta} -\frac{1}{k} \sum_{k=1}^{k} \log \frac{q_\phi(\mu_{\phi(x)} + \sigma_\phi(x) \cdot \varepsilon_k | x)}{P_\theta(x, \mu_{\phi(x)} + \sigma_\phi(x) \cdot \varepsilon_k)}$, $\varepsilon_k \sim N(0, I)$
$= -\int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p(z)} dz +$
$= \underbrace{KL(q_\phi(z|x) \| p(z))}_{①} \int q_\phi(z|x) \log P_\theta(x|z)\, dz$
Regularization
$= \underbrace{E_{q_\phi(z|x)}[\log P_\theta(x|z)]}_{②}$
Reconstruction Error

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial L_1}\frac{\partial L_1}{\partial \hat{y}_1}\frac{\partial \hat{y}_1}{\partial h_1}\frac{\partial h_1}{\partial W_1} + \frac{\partial L}{\partial L_2}\frac{\partial L_2}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial L_2}\frac{\partial L_2}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial h_2}\frac{\partial h_2}{\partial W_1}$$

Total $\frac{\partial L}{\partial W_1} = ① + ②$

$$W_1 \leftarrow W_1 - \eta \frac{\partial L}{\partial W_1}$$

LSTM

$L = L_1 + L_2$

$\frac{\partial L}{\partial W_1}$

$J = Q \Lambda Q^T$

leaky unit $h_t = \alpha h_{t-1} + (1-\alpha) x_t$, $0 < \alpha < 1$

$$\begin{bmatrix} h_{t,1} \\ \vdots \\ h_{t,d} \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_d \end{bmatrix} \odot \begin{bmatrix} h_{t-1,1} \\ \vdots \\ h_{t-1,d} \end{bmatrix} + \begin{bmatrix} 1-\alpha_1 \\ \vdots \\ 1-\alpha_d \end{bmatrix} \odot \begin{bmatrix} x_{t,1} \\ \vdots \\ x_{t,d} \end{bmatrix}$$

Gated RNN

forget gate

input gate